

---

# **Torque 3D Documentation**

***Release 4.0***

**GarageGames, LLC**

**Jul 01, 2020**



---

## Working with Torque3D

---

<b>1</b>	<b>What is Torque3D?</b>	<b>3</b>
<b>2</b>	<b>Directory Tour - TODO</b>	<b>7</b>
<b>3</b>	<b>Features</b>	<b>11</b>
<b>4</b>	<b>Assets</b>	<b>17</b>
<b>5</b>	<b>Overview - TODO</b>	<b>19</b>
<b>6</b>	<b>Required Downloads - TODO</b>	<b>21</b>
<b>7</b>	<b>Installing DirectX SDK - TODO</b>	<b>23</b>
<b>8</b>	<b>Install Visual Studio 2015 - TODO</b>	<b>25</b>
<b>9</b>	<b>Setup Visual Studio 2015 - TODO</b>	<b>27</b>
<b>10</b>	<b>Your First Project - TODO</b>	<b>29</b>
<b>11</b>	<b>Basics</b>	<b>31</b>
<b>12</b>	<b>Adding Objects</b>	<b>65</b>
<b>13</b>	<b>Editors</b>	<b>67</b>
<b>14</b>	<b>Tutorials</b>	<b>199</b>
<b>15</b>	<b>Overview of GUI Editor</b>	<b>323</b>
<b>16</b>	<b>Tutorials</b>	<b>327</b>
<b>17</b>	<b>Primer</b>	<b>359</b>
<b>18</b>	<b>Formats</b>	<b>365</b>
<b>19</b>	<b>Exporters</b>	<b>367</b>
<b>20</b>	<b>Tutorials</b>	<b>371</b>

<b>21 Importing Assets</b>	<b>387</b>
<b>22 Overview</b>	<b>389</b>
<b>23 Simple</b>	<b>417</b>
<b>24 Advanced</b>	<b>465</b>
<b>25 Audio</b>	<b>529</b>
<b>26 Lighting</b>	<b>549</b>
<b>27 Rendering</b>	<b>573</b>
<b>28 License</b>	<b>595</b>
<b>Index</b>	<b>597</b>



## Welcome

Torque 3D is a large piece of software. Chances are that most of the applications you have worked on up to this point have only been a fraction of the size of this SDK. This reference manual exists for the sole purpose of giving you, the user, a strong foundation to rely on while learning the engine.

The documentation is divided into multiple sections, each of which contains information related to specific subject. This means of organization allows you to jump to different chapters containing information that is pertinent to what you wish to work on.

- [introduction-index](#)
- [Setup](#)
- [World Editor](#)
- [Gui Editor](#)
- [Artist Guide](#)
- [Scripting](#)
- [Engine](#)



---

## What is Torque3D?

---

### 1.1 What is Torque 3D?

Torque 3D was created by **GarageGames** to make the development of games easier, faster, and more affordable. It is a professional Software Development Kit (“SDK”) that will save you the effort required to build a rendering system, high speed multiplayer networking, real time editors, a scripting system, and much more.

As part of Torque 3D, you receive full access to 100% of our engine source code. This means that you can add to, alter, or optimize any component of the engine down to the lowest level C++ rendering calls. That being said, you don’t need to be an experienced C++ programmer to use Torque 3D. **In fact, you do not need to know C++ at all.** Using TorqueScript and the collection of tools that are included with Torque 3D, you can build complete games (of many different genres) without ever touching a single line of C++ code.

To understand the basics of how the engine is setup and the tools available, a short reference is included below. Further sections in the documentation explain these tools in more depth.

### 1.2 System Requirements

- WindowsPC Windows 7/8.1/10
- MAC 2011+ models (not Intel HD 3000)
- Linux Tested on Ubuntu 16.04, 18.04
- Processor: 1.6 GHz Processor or better
- Memory: Win7+ 2GB RAM
- DirectX: 11, Shader 3.0 supported
- OpenGL: 3.x + ARB Extensions/4.1 Core
- Min GPU: Nvidia 500 series or better, AMD Radeon 5000 series or better, Intel HD Graphics 4000 or better

## 1.3 The Engine

The engine handles all of the elements of a game that run in real time on your computer. The Torque 3D engine is written entirely in C++ and is fully accessible to you as a developer. This means you have access to the inner workings of the code to customize it for your needs. The end result is that Torque 3D allows developers to add functionality, increase optimization, and learn how everything works. Alternatively, you can build a game from scratch to release without delving into the source code. The choice of how to develop **your** game is up to you.

For example, if you wish to add MYSQL database functionality or integrate the Havok SDK to enhance your game, those paths are open to you. Another benefit of source code access is the ability to read through the comments and data structures to gain a better understanding of how the entire system is set up.

Do not be intimidated. This documentation will show you how to create games without touching the source code at all. There is no need to start working with the engine's C++ code until you feel comfortable. In the meantime, you can get going with Torque 3D right away!

## 1.4 TorqueScript

Much of your game play logic, camera controls, and user interface will be written in TorqueScript. It is a powerful and flexible scripting language with syntax similar to C++. The key benefit of TorqueScript is that you do not need to be a code guru or know the nitty-gritty specifics of a particular language like C++. If you are already familiar with basic programming concepts, you will have a head start on building your own game.

Another benefit of using TorqueScript, as opposed to editing the engine's underlying C++ source code, is that you do not have to recompile your executable to see changes in your game. You simply create or modify a script, save, and then run the game from the Toolbox.

There are several TorqueScript articles for new developers that will help you learn the syntax, functionality, and how to use the language with the engine and editors.

## 1.5 Editors

Learning to work with Torque 3D editors is a large part of your initial experience. The key is to remember that the editors work in real-time and are **WYSIWYG** (What You See Is What You Get). When you use the editors to modify your level, you will see the changes immediately in the game.

**World Editor** - The World Editor is a tool that will help you assemble your game levels. With this tool, you will add and position terrain, game objects, models, environmental effects, lighting, and more.

**GUI Editor** - GUI stands for Graphical User Interface. Some examples of GUIs include: splash screens, your main menu, options dialogs and in game Heads Up Displays ("HUDs"). With the GUI Editor, you can design and create your menus, player inventory system, health bars, loading screens, and so on.

## 1.6 The Asset Pipeline

You would not have much of a game without models, textures, and other art assets. For Torque 3D, the preferred file format for 3D art assets is COLLADA.

From the COLLADA website: "COLLADA is a COLLABorative Design Activity for establishing an open standard digital asset schema for interactive 3D applications." In-other-words, it is a 3D model file format supported by most major art applications used to make content for games. You can create a model in 3D Studio Max, Maya, Blender, or any other 3D editor that supports the COLLADA format.

For those of you familiar with previous Torque engines, you can still import DTS (static models) and DSQ (animation data) files for your 3D objects. This includes static shapes, players, buildings, and props. If you already have a library of DTS and DSQs, feel free to use them in Torque 3D. From this point on, we recommend you transition to the COLLADA open standard for new art assets.



---

### Directory Tour - TODO

---

## 2.1 Introduction

After you have installed Torque 3D and experimented with it, we recommend that you familiarize yourself with the directory structure of the Torque 3D SDK.

The stock Torque 3D installation includes dozens of folders and thousands of files. It is easy to be overwhelmed at first, but if you spend a little time to browse through the directory hierarchy while reading along with the Directory and Torque 3D Project Tours below, you'll find that everything is organized in a straightforward and intuitive manner. Following the tours, we have also included a brief description of the file types you will encounter while using Torque 3D.

## 2.2 SDK Tour - TODO

TODO

## 2.3 Torque 3D Project Tour - TODO

Modules and projects created share a common directory structure. Folders are named and organized in such a way that it is easy to locate files based on their type and functionality. The following table describes the purpose of the main folders found in each Torque 3D project.

TODO

## 2.4 File Descriptions

- **.bat** - Windows batch files that contain OS commands which can be run to perform tasks. Mostly used in the Torque environment for deleting multiple files at once (.dso, prefs, .uft, etc).

- **.c / .cpp** - Source code files. These contain the C and C++ language programming instructions which are compiled by developer tools to create the game executable (.exe).
- **.cs** - TorqueScript files contain most of your programmed game logic and processing using the TorqueScript language.
- **..command** - The Mac equivalent of a .bat file.
- **console.log** - This file is generated by your game whenever you run it. A lot of information about the critical events that occur during your game are written and saved here.
- **.dae** - COLLADA files which store model information (geometry, textures, nodes) before Torque 3D converts them to its proprietary DTS format.
- **.dml** - (*Deprecated*) Configuration files used for combining environmental textures, mostly used by precipitation and clouds.
- **.dso** - TorqueScript (.cs) files that have been compiled into an encrypted format. DSO files are more secure than the original uncompiled .cs files that they are created from.
- **.dsq** - (*still supported, replaced with .dae*) Proprietary files which store animation information in the format expected by Torque. A .dsq is used in conjunction with .dts files.
- **.dts** - Proprietary file which stores model information (geometry, textures, nodes) in the format expected by Torque. A .dts is loaded directly into Torque to render 3D models, such as players, items, weapons, and vehicles.
- **.dll** - Dynamically Linked Library files (DLL) contain code that can be called at runtime by application binary files (.exe and .app) instead of being compiled directly into the binary file itself. This is useful for common libraries, such as OpenGL, OpenAL, Havok, etc.
- **.exe / .app** - Binary files created by developer tools during the compiling process. Most commonly used to launch your game, a Torque tool, or any other normal computer application.
- **.glsl** - Contains the configuration information that describes an OpenGL graphics shader.
- **.gui** - Contains the data used to create a Graphical User Interface (GUI). These files are created using TorqueScript but the special .gui extension allows the GUI Editor to open them for visual modification.
- **.h** - Header files contain declarations written in the C and C++ languages for classes, structs, variables, and other programming elements, which are compiled by the engine in conjunction with .c and .cpp files to create binary files.
- **.hlsl** - High Level Shader Language files. Holds the configuration information that describes an MS Windows specific graphics shader.
- **.mis** - Contains descriptions of the terrain, models, lighting, and environment, and other objects which make up the missions in your game in the format expected by Torque.
- **.ogg** - Contains audio data in the Ogg Vorbis format for playing sounds and music in your games.
- **.png / .jpg / .gif / .dds** - Raw image files.
- **.ter** - Contains terrain data in a format which is understood by, and can be loaded directly into, Torque 3D.
- **.torsion** - Contains project configuration information which is understood by, and can be loaded into, Torsion which can be used to edit Torque .cs and .gui files. These are Windows only files.
- **.uft** - These files contain pre-cached font information.
- **.wav** - Contains audio data in the Waveform format for playing sounds and music in your games.



## 2.5 Conclusion

Understanding the Torque 3D folder and file structure will allow you to easily locate the files you need to work on during the development of your games.



Below is a summary of new features we have added to Torque 3D with the 1.2 release. Each summary includes further details on each new feature.

### 3.1 Teleporters

Teleporters are Trigger objects that have a specific behaviour. You define a teleporter with a TriggerData datablock with some special fields defined:

```
datablock TriggerData(TeleporterTrigger : DefaultTrigger)
{
    // Amount of time, in milliseconds, to wait before allowing another
    // object to use this teleportat.
    teleporterCooldown = 0;

    // Amount to scale the object's exit velocity. Larger values will
    // propel the object with greater force.
    exitVelocityScale = 0;

    // If true, the object will be oriented to the front
    // of the exit teleporter. Otherwise the player will retain their original
    // orientation.
    reorientPlayer = true;

    // If true, the teleporter will only trigger if the object
    // enters the front of the teleporter.
    oneSided = false;

    // Effects to play at the entrance of the teleporter.
    entranceEffect = EntranceEffect;
    exiteffect = EntranceEffect;

    // 2D Sound to play for the client being teleported.
```

(continues on next page)

(continued from previous page)

```
    teleportSound = TeleportSound;

};
```

When you define a teleporter you also need to have an exit as defined by the instance’s “exit” field, which is another object in the level – usually another teleporter trigger. Also note that this just defines the trigger itself. If you want your teleporter to have a shape you’ll need to add a shape object to the level, such as a TSSStatic.

## 3.2 Weapon Clip System

Weapons may now use the optional ammunition clip system.

To make use of the clip system each weapon needs to define two ItemData datablocks. The first is the clip itself, which needs to be part of the AmmoClip class. Here is an example:

```
datablock ItemData(LurkerClip)
{
    // Mission editor category
    category = "AmmoClip";

    className = "AmmoClip";

    // Basic Item properties
    shapeFile = "art/shapes/weapons/Lurker/TP_Lurker.DAE";
    mass = 1;
    elasticity = 0.2;
    friction = 0.6;

    // Dynamic properties defined by the scripts
    pickUpName = "Lurker clip";
    count = 1;
    maxInventory = 10;
};
```

The “maxInventory” field indicates the maximum number of clips a player may carry of this type.

Then the ammunition itself needs to be defined as part of the Ammo class:

```
datablock ItemData(LurkerAmmo)
{
    // Mission editor category
    category = "Ammo";

    // Add the Ammo namespace as a parent. The ammo namespace provides
    // common ammo related functions and hooks into the inventory system.
    className = "Ammo";

    // Basic Item properties
    shapeFile = "art/shapes/weapons/Lurker/TP_Lurker.DAE";
    mass = 1;
    elasticity = 0.2;
    friction = 0.6;

    // Dynamic properties defined by the scripts
    pickUpName = "Lurker ammo";
};
```

(continues on next page)

(continued from previous page)

```

    maxInventory = 30;
    clip = LurkerClip;
};

```

The “maxInventory” field indicates the maximum number of bullets a clip may hold. The “clip” field points back to the clip that holds this ammo.

Finally, you add both the clip and the ammo to the weapon’s datablock:

```

datablock ShapeBaseImageData (LurkerWeaponImage)
{
    ...

    // Projectiles and Ammo.
    item = Lurker;
    ammo = LurkerAmmo;
    clip = LurkerClip;

    ...
};

```

When you build out the player’s datablock you have to add both the weapon and the clip as items the player may carry. You don’t need to add the ammo:

```

datablock PlayerData (DefaultPlayerData)
{
    ...

    maxInv[Lurker] = 1;
    maxInv[LurkerClip] = 20;

    ...
};

```

However, when you actually add the weapon and clip to the player during `GameCore::loadOut()` you’ll want to also add the ammunition. This will start the weapon as loaded:

```

function GameCore::loadOut(%game, %player)
{
    ...

    // Set up inventory and weapon cycles
    %player.clearWeaponCycle();

    %player.setInventory(Lurker, 1);
    %player.setInventory(LurkerClip, %player.maxInventory(LurkerClip));
    %player.setInventory(LurkerAmmo, %player.maxInventory(LurkerAmmo));
    %player.addToWeaponCycle(Lurker);

    ...
}

```

By default the weapon system will recover any ammunition that is left in a clip when the player manually reloads. Many games operate in this way. If you wish to have any ammunition that is in a clip when the clip is removed to be discarded see the `WeaponImage::clearAmmoClip()` method and comment out the line indicated.

## 3.3 Per-Player Weapon Cycling

Weapon cycling is no longer set up globally – it is set up for each player instance. This allows for different players to have different weapon load outs, such as a Soldier with his set of guns and an Alien with its own set of guns.

Often you set up a player's weapon cycling in `GameCore::loadOut()` but it can occur at any time on the server. The weapon cycling methods themselves are defined in `script/server/weapon.cs` against the `ShapeBase` class (which `Player` inherits from).

When starting to build the weapon cycling list you call `clearWeaponCycle()` on the player, which clears out any existing list. You then call `addToWeaponCycle()` on the player for each weapon `Item` class that you wish the player to cycle through. The order in which the weapons are added to the list is the same order that the player will cycle through them.

Behind the scenes `ShapeBase::cycleWeapon()` is called on the player whenever the weapons should be cycled. The direction to cycle is passed into this method to move up or down the list.

## 3.4 Multiple Projectiles Per Shot

Weapons may fire multiple projectiles per shot. A shot gun is a good example.

To have multiple projectiles a weapon defines the “projectileNum” field and sets it to a value greater than 1. And if a weapon has a projectile spread defined (with the “projectileSpread” field) then each projectile has its own calculated spread trajectory.

## 3.5 New Damage Reporting

Starting with 1.2 T3D supports custom death messages based on the type of damage that killed the player. To set this up you need to define a `sendMsgClientKilled_XXX` function that provides the death message, where XXX is the damage type. For example, the proximity mine has the following defined for the `MineDamage` damage type as defined in the mine's datablock:

```
// Customized kill message for deaths caused by proximity mines
function sendMsgClientKilled_MineDamage( %msgType, %client, %sourceClient, %damLoc )
{
    if ( %sourceClient $= "" )                // editor placed mine
        messageAll( %msgType, '%1 was blown up!', %client.playerName );
    else if ( %sourceClient == %client )      // own mine
        messageAll( %msgType, '%1 stepped on his own mine!', %client.playerName );
    else                                       // enemy placed mine
        messageAll( %msgType, '%1 was blown up by %2!', %client.playerName,
                    %sourceClient.playerName );
}
```

This system allows for very specific messages to be sent to all clients. The system has custom death messages set up for `Impact`, `Suicide` and a `Default` (a catch all) damage type.

## 3.6 The Asset Pipeline

Characters in T3D can be setup to use different weapon animations and share those animations between different skinned meshes with the same skeleton hierarchy. So a standard T3D character would have `COLLADA` files for the character's skinned mesh and skinned skeleton as well as for the character's animations with just the skeleton for each

weapon pose. The animations can be exported individually or combined in one .dae file that is split up through the shape editor.

For more information, see the Torque 3D Character Primer from the Artist Guide.





With Torque3D 4.0, the engine now utilizes an assets-based system to manage content for your game. Below goes into the high-level breakdown of the elements of it and how they work.

### 4.1 Modules

Modules are the main organizational structure for assets. They act as containers for assets so you can quickly parse them by group. Modules are defined via a module definition file ending with the ‘module’ extension and companion script file.

For example, we have here TestModule.module:

```
<ModuleDefinition
  ModuleId="TestModule"
  VersionId="1"
  Description="A test module"
  ScriptFile="TestModule.cs"
  CreateFunction="create"
  DestroyFunction="destroy"
  Group="Game">
  <DeclaredAssets
    canSave="true"
    canSaveDynamicFields="true"
    Extension="asset.taml"
    Recurse="true" />
</ModuleDefinition>
```

The primary parameters to focus on would be:

- ModuleId: The name of the module when searching and utilizing assets
- ScriptFile: Companion script file that can be utilized for extra loading/unloading management behavior
- CreateFunction: Function called in companion script file upon module being created

- **DestroyFunction:** Function called in companion script file upon module being destroyed
- **Group:** What group this module is part of. Largely utilized for selectively loading sets of modules like ‘Core’, ‘Game’ or ‘Tools’
- **DeclaredAssets:** Subelement that defines the file extension to automatically parse upon load to register assets to this module

## 4.2 Assets

An ‘Asset’ is any specific chunk of content. This can be a singular file, such as an image, or multiple related files together, like a GUI having a gui file and script file, or a level having the level script file, decal cache, postFX settings, forest object cache, etc.

The Assets system is a system by which said content can be registered into a database for easy loading, utilization, and referencing. This is done via the use of Asset Definitions, which is a type of metadata. When modules are loaded, they scan their respective directory and find any asset metadata files within and register them with the Asset Database. This enables them to be easily referenced via the paradigm of <ModuleName>:<AssetName>. If referenced in this way, the engine will automatically find, reference and load the asset, handling the file paths and resource management automatically. Different asset types have different Asset Definitions, but they largely follow a similar structure:

```
<LevelAsset canSave="true" canSaveDynamicFields="true" AssetName="TestLevel" LevelFile="TestLevel.mis"
  LevelName="Test Level" LevelDescription="A simple test level." VersionId="1" />
```

The most important parameter is the AssetName, which is used in combination with it’s owner module to formulate an AssetID. This, referened above as <ModuleName>:<AssetName> when referencing assets and it will handle the referencing and loading behavior automatically

This article will take you through installing the necessary tools to develop games using Torque 3D. We have separated this article into two sections depending on the focus of your team. The first section is oriented towards level designers, artists, and people who are new to Torque 3D and do not need to dive into the source code. The second section is oriented towards programmers who will be working with the source code of the engine.

## 5.1 New Users, Scripters, and Artists

Now that you have decided to work with Torque 3D, this section will take you through setting up your environment.

The setup of Torque 3D is simple.

### **TBD**

Also, make sure that your video and sound drivers are up-to-date with the latest versions. Doing an Internet search on the video or graphics card or visiting the manufacturer web site will usually lead to downloads of the latest drivers. Once these have been installed, you are ready to hit the ground running. Below are some of the common hardware vendor site. Check your system requirements to find your vendor and hardware model.

### **Common Graphic Drivers**

- NVIDIA Drivers
- AMD/ATI Drivers
- Intel Drivers

### **Common Sound Drivers**

- Logitech
- HTIOmega
- ASUS
- RealTek

## 5.2 Programmers

When you download T3D, you receive access to the complete engine source code. This allows you to customize the engine to meet your project needs.

This tutorial will guide you step-by-step through the process of setting up your development environment. Once set up correctly, you will be able to edit and recompile your T3D engine source code.

For this tutorial, we will be using the free Microsoft Visual Studio C++ (“VC++”) Express Edition as our development environment.

This tutorial explains how to:

- Install VC++ Express.
- Download the required files for a successful engine build.
- Set up the environment dependent files correctly.
- Make your first successful engine build.

**TDB**

## CHAPTER 6

---

Required Downloads - TODO

---

### **6.1 Download Torque 3D**

### **6.2 Download The Required SDK files**



---

### Installing DirectX SDK - TODO

---

#### 7.1 Download and Install DirectX SDK





## CHAPTER 8

---

Install Visual Studio 2015 - TODO

---

**TBD**



## CHAPTER 9

---

### Setup Visual Studio 2015 - TODO

---

**TBD**



## CHAPTER 10

---

### Your First Project - TODO

---

#### **10.1 Working with a T3D Project**

#### **10.2 Create a new project**

#### **10.3 Your new Project**

#### **10.4 The Source code**

#### **10.5 The Project**

#### **10.6 Your First Compile**

#### **10.7 The Project Build**

#### **10.8 Run your project**

#### **10.9 Summary**

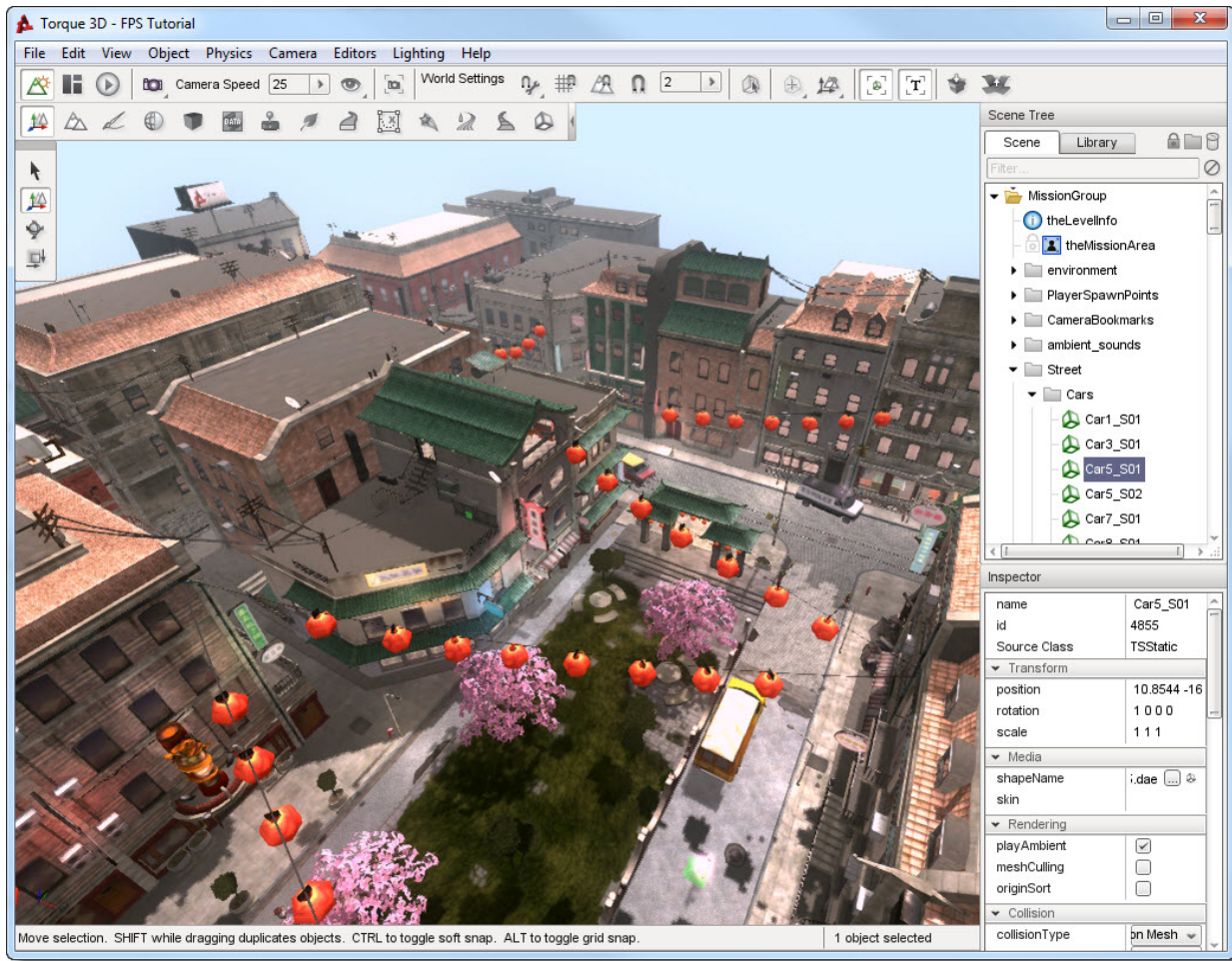


## 11.1 Overview

### 11.1.1 Purpose

The World Editor is used to build and edit game levels. This includes adding and modifying terrains, buildings, foliage, cloud layers, vehicles, environmental effects, lighting effects, and much more. The World Editor is the first (and most important) tool a new user should learn.

A sample game level as seen inside the World Editor:



The World Editor is not a **tool** for creating game objects. Objects must be created using applications appropriate for the object type (i.e., 3DS Max or Blender3D to create a 3D model). However, once an object is loaded it can be modified by the World Editor in a variety of ways. The simplest modification would be a change in scale (size), but more complex modifications are also possible. For example, the Torque **Material Editor** can be used to alter (or completely replace) textures on a 3D object or add shader effects.

A typical World Editor workflow might go as follows (in very simplified terms):

1. Create a 3D model in an application like 3DS Max, Maya, or Blender.
2. Save that model to a sub folder inside your game/art directory.
3. Launch World Editor (which will automatically find that model if step 2 was done correctly).
4. Add the model to your level; position, scale, rotate, and adjust its materials as desired.
5. Test your changes in-game with the push of a single button.
6. Return to the World Editor and continue to tweak your level.

Of course, there is a lot more to the World Editor than positioning 3D models. You will also be working with 2D assets like grey scale height-maps to create terrains, as well as specialized tools for creating rivers, forests, and roads.

Finally, it is worth noting that Torque 3D includes numerous art assets for you to play with... so you can skip steps 1 and 2 above and start building game levels right away!



### 11.1.2 Using the World Editor Documentation

The World Editor documentation follows a logical progression. Those who wish may work through it in a methodical way. Others may choose to skip difficult sections and jump directly to the tutorials at the end or to focus on only the features of interest.

Everyone learns differently, but we've found that a good way for new users to get started quickly is to follow these three steps:

1. Continue reading this document ("Overview") in its entirety. It covers: how to launch the World Editor, how to look and move around in a game level, and it offers a few important tips for new (and experienced) users.
2. With the World Editor open, quickly skim the next document, "**Interface**". You should only spend five-to-ten minutes getting an initial feel for the basic layout of the interface. Do not try to learn any features in detail.
3. Try to add moving clouds to your level by following the **Basic Cloud Layer instructions**. Whether you are successful or not, spend no more than five minutes on this task.

Do not be concerned if you have trouble completing Step 3. Its purpose is to give you a specific task that requires direct interaction with the interface. That small exposure to the interface will go a long way towards making the remainder of the documentation more meaningful and easy to follow.

Once you've completed the three steps above, how you proceed is up to you. For those who prefer to jump around, we recommend you start by carefully reviewing the **Interface** document.

### 11.1.3 How to Launch the World Editor

There are two ways to open the World Editor: from main menu clicking the World Editor button or by using hot keys from within a running Torque game. You will likely use both methods during development.

### 11.1.4 Launching from Within a Running Game

While your game is running, you can open or close the World Editor at any time using hot key combinations:

- **On Windows and Linux**, to open or close the World Editor, press the F11 key.
- **On Mac OS X**, to open or close the World Editor, press CMD+FN+F11.

---

**Note:** When you first launch the World Editor, it is likely you will do so from the Main Menu. However, after you have modified your level, if you decide to test it out by clicking the Play Game button (as described in the "Interface" document ), you will need to use the F11 hotkey to get back to the World Editor. Otherwise, you would be forced to quit your game and relaunch the World Editor.

---

### 11.1.5 Looking and Moving Around

While working in the World Editor, you will need to move and look around to inspect your level.

- **Forward/Back/Left/Right** movement is controlled by the corresponding arrow keys on your keyboard (the WASD keys can also be used). If you have a mouse-wheel, it can be used to move forward or backward.
- **Look Left/Right/Up/Down** by holding the right mouse button down while moving the mouse.
- **Pan Left/Right/Up/Down** by holding down the middle mouse button (Mouse 3) while moving the mouse. On most mice with a scroll wheel, this is achieved by depressing (not scrolling) the mouse wheel.

**Note:** There are a number of Camera options, discussed further in the Interface document, which in some cases may alter the behavior of these controls in minor ways.

---

When play testing your game outside of the World Editor, default control is typical of most First Person Shooters and can be remapped by pressing Ctrl-O (Windows) to bring up an options dialog. A few important controls are listed below:

- **Forward/Back/Left/Right** movement is controlled by the corresponding arrow keys on your keyboard (the WASD key can also be used).
- **Look around** by moving the mouse.
- **Fire/Alt Fire** are triggered by the left and right mouse buttons.
- **Jump** is activated by the Space Bar.
- **First/Third person** view is toggled by pressing TAB.
- **C\*\*hange weapons\*\*** by scrolling the mouse wheel (or press Q key).
- **Exit vehicles** by pressing Control-F.
- **Return to World Editor** by using the F11 hotkey (as discussed above).

### 11.1.6 Tips

The following is a general list of knowledge you should keep in mind while editing a level in your game:

- **Try to design your levels outside of the editor first.** Sometimes it is helpful to have a simple verbal or visual design ready before you actually start editing. Even if it is a simple blueprint on a napkin, a level editor/artist with a reference to work from will cover ground much more quickly.
- **Prioritize your object placement.** It makes sense to polish certain aspects of a level before others. For example, try to finish your Sky, Sun, and Terrain before you move on to adding rivers, foliage, and other objects. Performing major adjustments to a terrain with hundreds of objects already placed could be tedious and counterproductive.
- **Play your level regularly.** After you reach a major milestone, try actually doing the things in your level as a player would. There is a big difference between the experience of a player in a game and that of a designer with a free-floating camera in the World Editor.
- **Do not forget to optimize.** Some specific World Editor objects are more appropriate than others. Use Ground Cover instead of a 3D model with lots of grass or trees attached. As much as possible, use the Sun rather than numerous point lights to handle ambient lighting. There are other such optimizations which will become apparent towards the end of development.
- **SAVE AND SAVE OFTEN.** This cannot be stressed enough. Computers crash, power goes out, cats jump on keyboards, and in rare circumstances you may encounter a yet undiscovered issue which causes data corruption. Any number of accidents can result in hours of work being lost. We recommend you save as often as you can.

### 11.1.7 Conclusion

Now that you know the purpose of the World Editor, and how to access it, we can move on to learning how to use it. Before you start placing objects or creating levels, you should learn the interface. Continue on to the World Editor Interface.

## 11.2 Interface

The default World Editor view consists of five main sections:

**File Menu (Yellow)** Found at the very top of the World Editor window, you will find menus that controls the global functionality of the editor, such as opening/saving levels, toggling camera modes, opening settings dialogs, and so on.

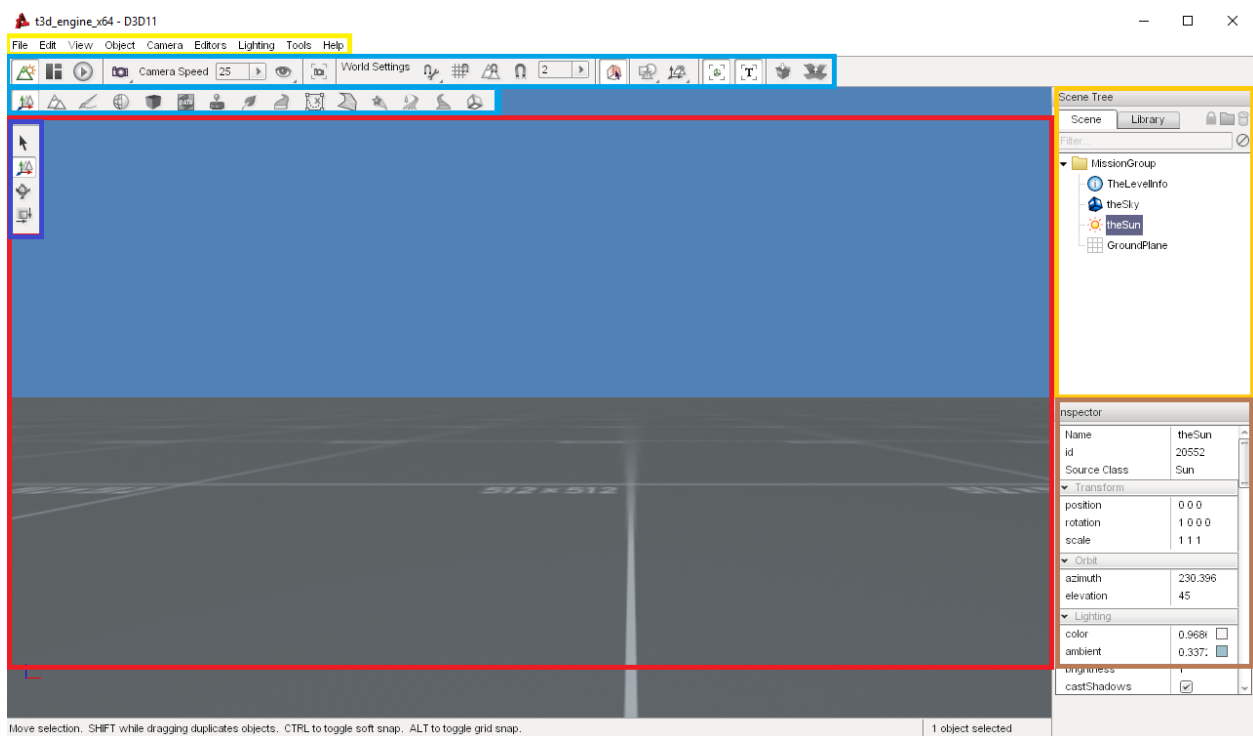
**Tools Bar (Blue)** Located just below the File Menu, this bar contains shortcuts to all of the tools, their settings, and some options found in the File Menu.

**Tool Palette (Dark Blue)** The Tool Palette changes based on what Tool you are currently using. For example, when using the Object Editor you will have icons for moving and rotating an object, whereas the Terrain will have icons for moving and rotating an object, whereas the Terrain Editor display icons for elevation tools.

**Scene View (Red)** Main scene view of your level and its objects. In the upper left you can see the Tool Palette.

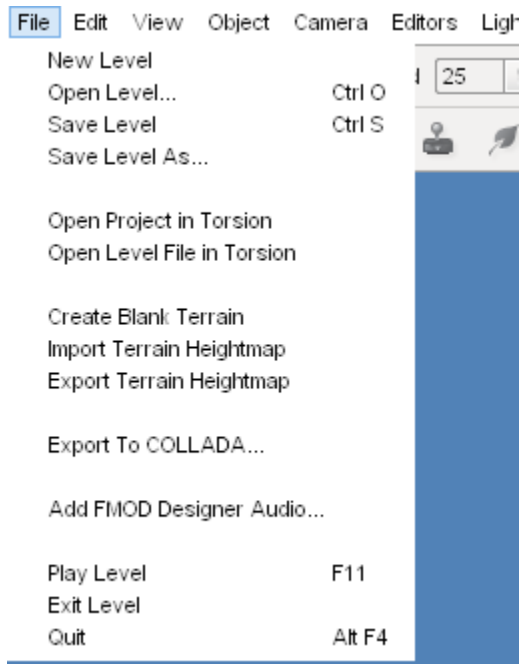
**Scene Tree Panel (Gold)** While using the Object Editor, one of the floating panels available to you is the Scene Tree. It is composed of two tabs: Scene and Library. The Scene tab contains a list of objects currently in your level. The Library tab is what you will use to add new objects to your level after which they will appear in the Scene tab.

**Inspector Panel (Brown)** While using the Object Editor, a selected object's properties will be shown in this panel. Most of your object editing will be performed here.

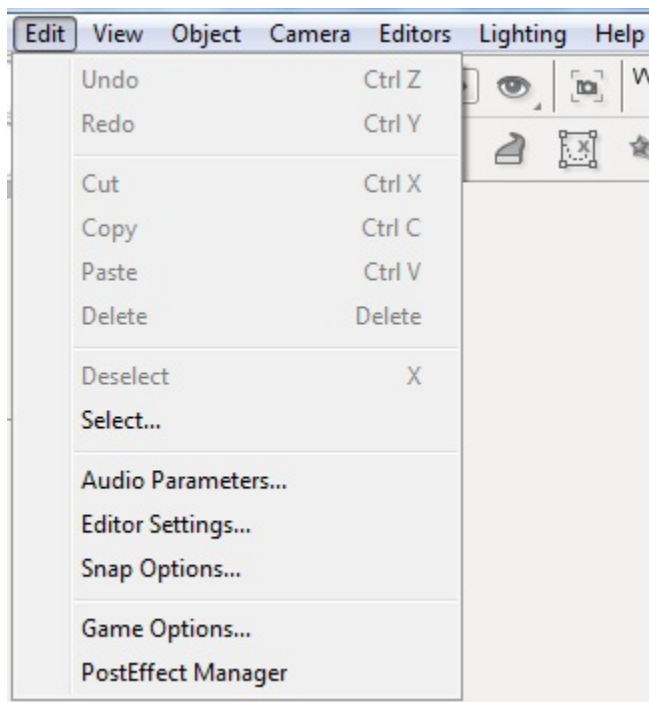


### 11.2.1 File Menu

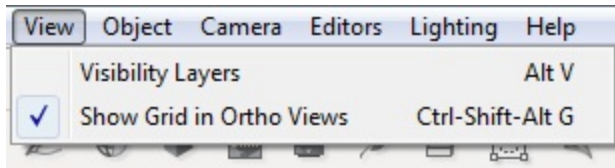
File Menu allows you to: Create, save, open, and close levels; Open, import, and export level data to/from other tools; Run your level to test it and exit the World Editor.



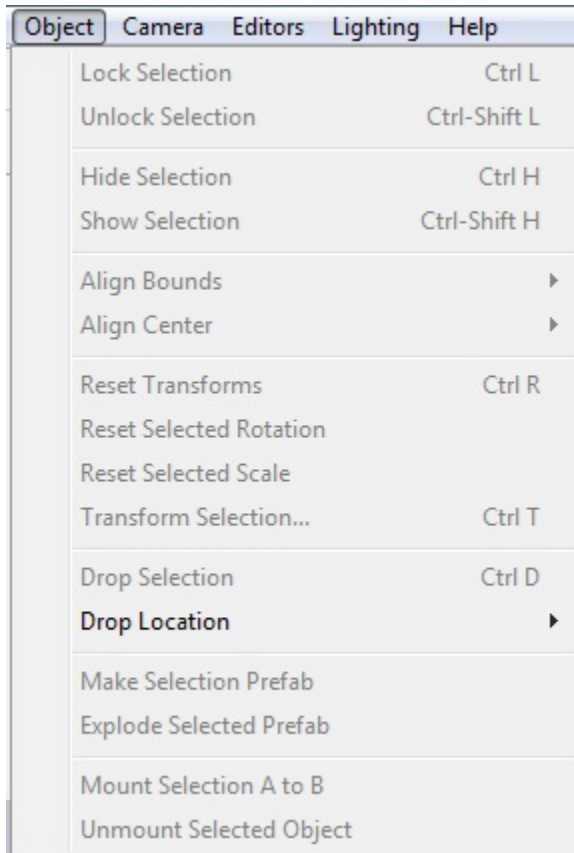
The Edit Menu allows you to: Control editor actions such as undo and redo; Cut, copy, paste, and delete objects you have selected; Select objects using a name pattern or by type filtering; Access dialogs to control various World Editor settings.



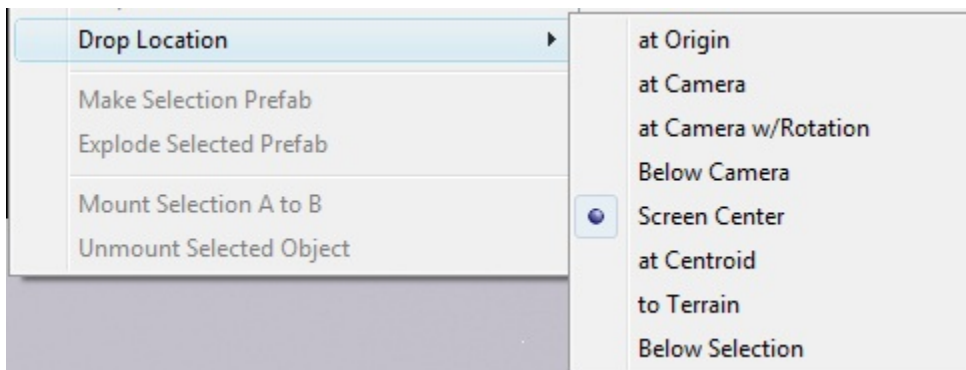
The View Menu: Opens the Visibility Layers dialog which toggles debug rendering modes; Toggle the visibility of other aspects of the editor.



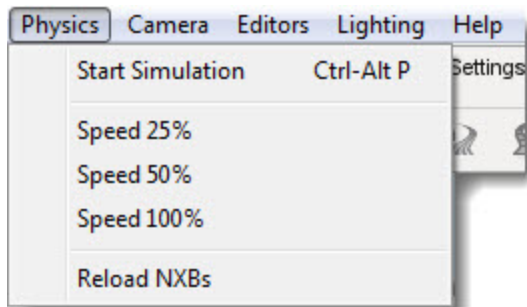
The Object Menu allows you to: Manipulate a selected object's settings by locking/unlocking it, hiding/showing the object, resetting its transforms, and so on.



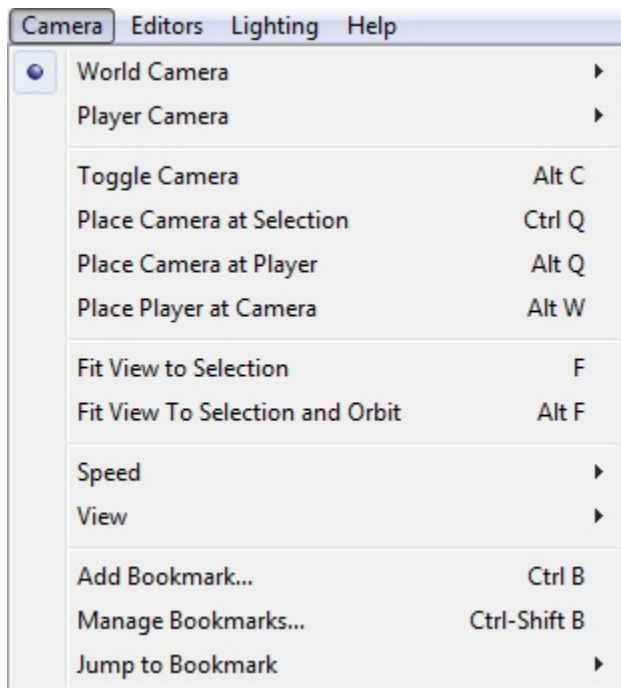
The Drop Location sub-menu selection informs the World Editor where it should place newly created objects.



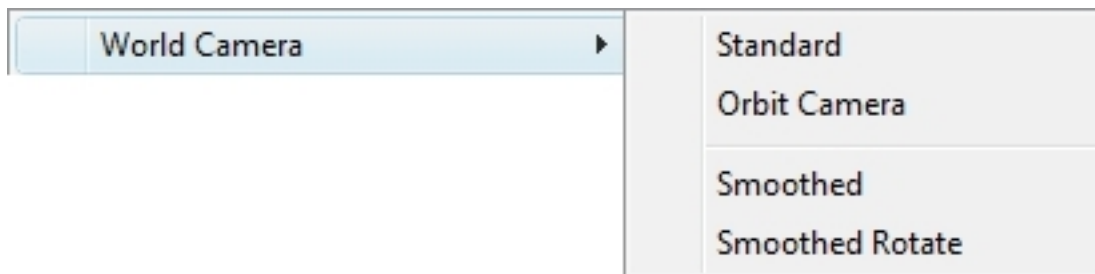
The Physics Menu allows you to Start and Stop PhysX simulation in the Editor, set the simulation speed and reload all PhysX objects and actors. (this menu is enabled if T3D game engine have enabled Nvidia PhysX.)



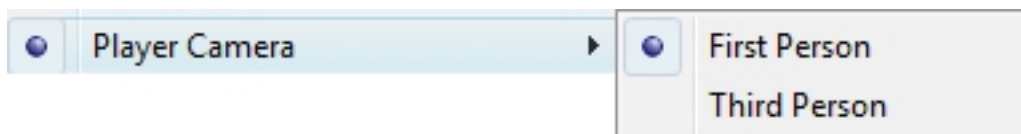
The Camera Menu allows you to choose your camera type, adjust its speed and motion, and drop it at certain locations.



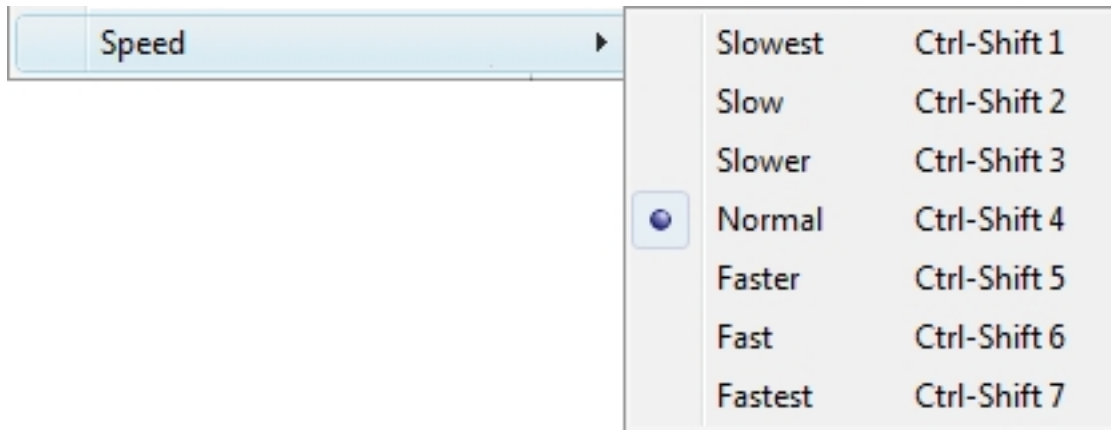
The World Camera sub-menu allows you to change the way the camera moves.



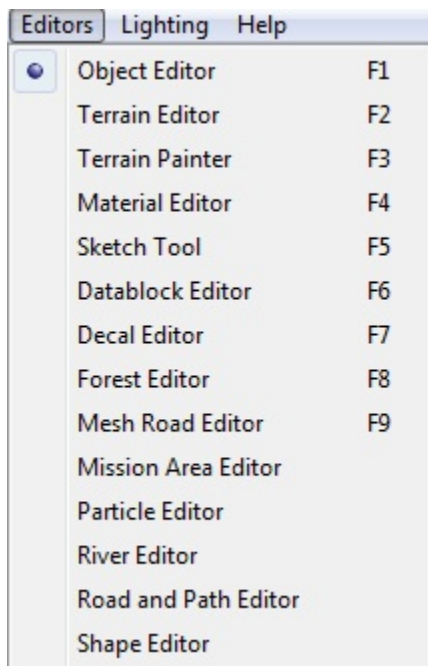
The Player Camera sub-menu allows you to switch between perspectives while moving around as a player.



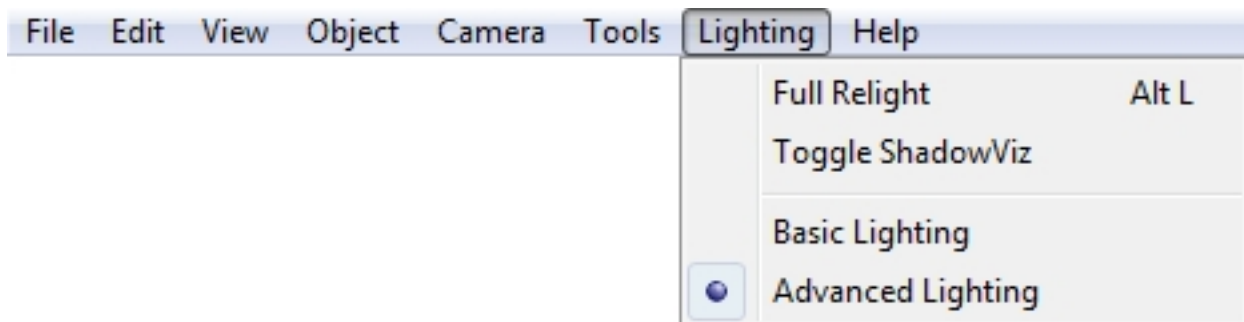
The Camera Speed sub-menu allows you to adjust how fast the camera moves.



The Editors Menu allows you to select which set of editing tools is currently active in the World Editor.

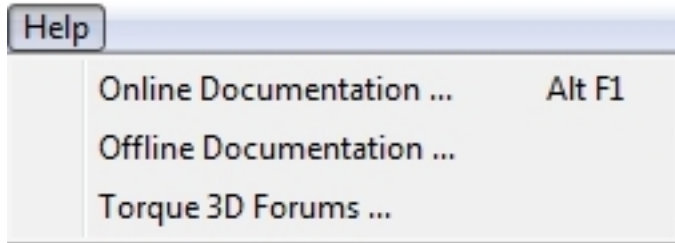


The Lighting Menu allows you to switch between Advanced and Basic lighting modes, as well as perform level relights.



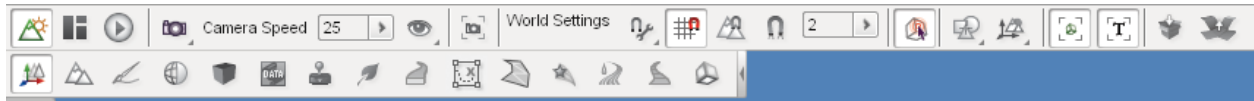
Contains shortcuts to documentation and forums for Torque 3D. **TODO - update image**





## 11.2.2 Tools Bar

The Tools Bar is the best way to switch between tools. It is made of two components: Tool Settings (top bar) and Tools Selector (bottom bar).



Tool Settings is made of up three sub-sections: the editor selector, camera settings, and Object Editor. The editor selector and camera setting will always be displayed. The Object Editor will display available settings for the currently selected tool. The Tools Selector will always display the same shortcuts for selecting tools.

This section focuses on the elements of Tool Settings.

The first three icons switch between the editor's operating modes. Each icon represents a different editing mode and only one mode can be active at any time. There are three modes: World Editor, GUI Editor, and Game Mode. The World Editor is represented by the mountain icon. The GUI Editor is represented by the boxes icon. The Game Mode is represented by the arrow icon.



World Editor mode provides tools for manipulating the “world” of your game including terrain, creatures, and so on.

GUI Editor mode provides tools for manipulating the Graphical User Interface (GUI) of your game such as health meters, cursors, and so on.

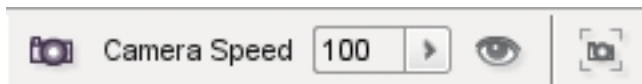
Play Game Mode runs your game and lets you play through it.

---

**Note:** When you use this icon to play your game the World Editor actually closes completely. To return to the World Editor you must press F11 or exit the game and relaunch the World Editor from the Main Menu.

---

Next to the editor selector, you will find the camera and visibility settings.



The camera icon will let you choose your camera type. The drop-down menu next to it will let you switch between camera speeds. The eye icon is the visualization settings which toggle debug rendering modes for various graphical modules, such as normal mapping, wireframe, specular shading, etc. The icon that looks like a camera in a box will move your camera to whatever object you have selected, filling up your view with its boundaries.





The World Settings make up the rest of this bar when using the tools. The first icon lets you determine your snapping options (snapping to terrain, a bounding box of an object, which axis, etc.). The next icon toggles snapping to a grid. The magnet icon determines soft snapping to other objects. The numeric indicator determines the distance of the snap option.

The box icon with an arrow is a selection tool that allows you to select an object according to its bounding box. This makes selecting small, detailed objects much easier. The next icon that looks like a bullseye will change the selection target from the object center to the bounding box center. The small icon with arrows and mountains will change the object transform and the world transform.

The next two icons show descriptors in your scene. The first icon that looks like a box in a square will display object icons for the various objects in your scene. The second icon will show text descriptors for the objects in your scene.

The last two icons in the bar are prefab icons. The first icon lets you group selected items into a “prefab” (or prefabricated collection) of objects. The second icon will ungroup your prefab items.

### 11.2.3 Tool Selector and Palette

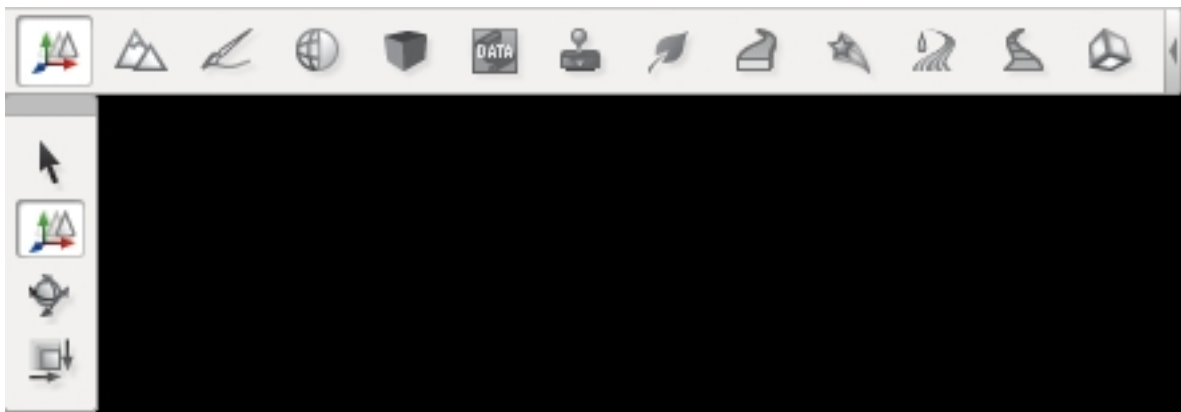


Fig. 1: Object Editor: Used to place objects in the world, group them, and lay out your scenes.

### 11.2.4 Scene Tree

The Scene Tree panel is available while using the Object Editor tool. It is composed of two tabs: Scene and Library. The Scene tab contains a list of objects currently in your level. You can select specific objects to modify them.

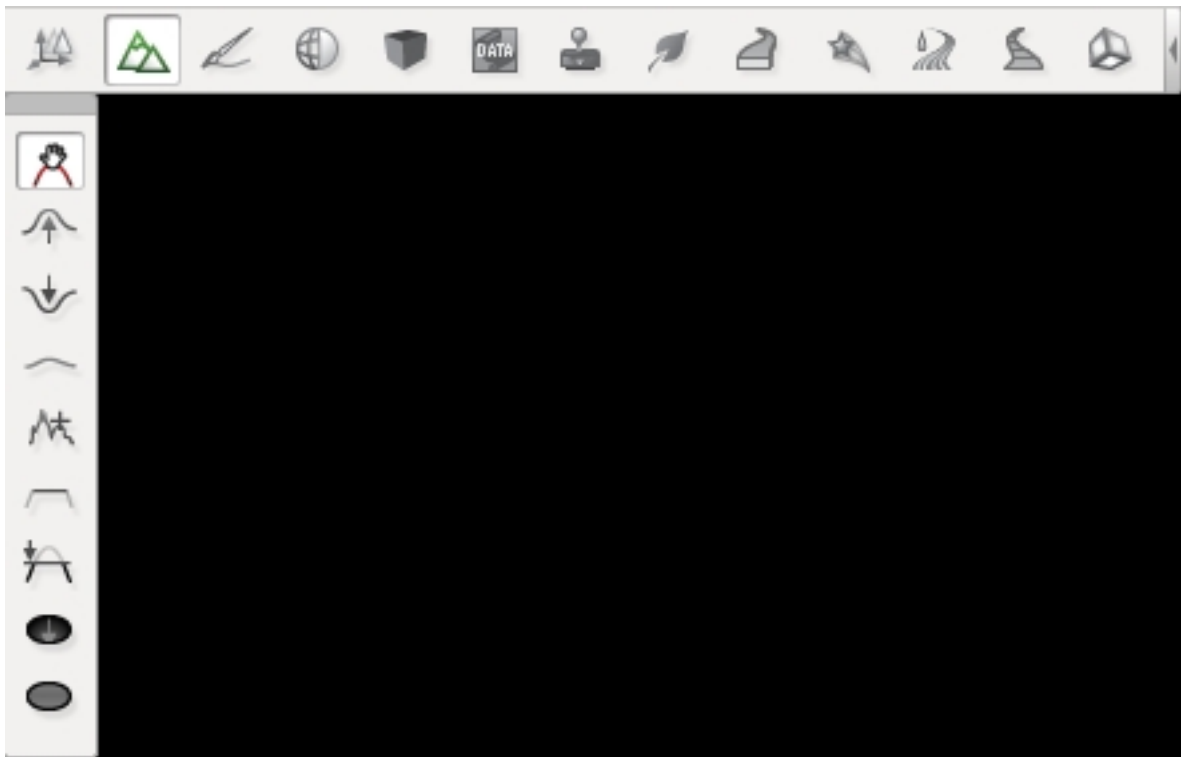


Fig. 2: Terrain Editor: Used to edit, save and load terrain objects in your scenes.

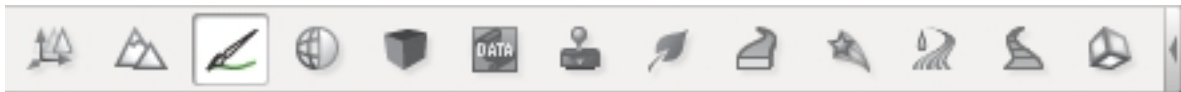


Fig. 3: Terrain Painter: To paint textures onto terrains in scenes.

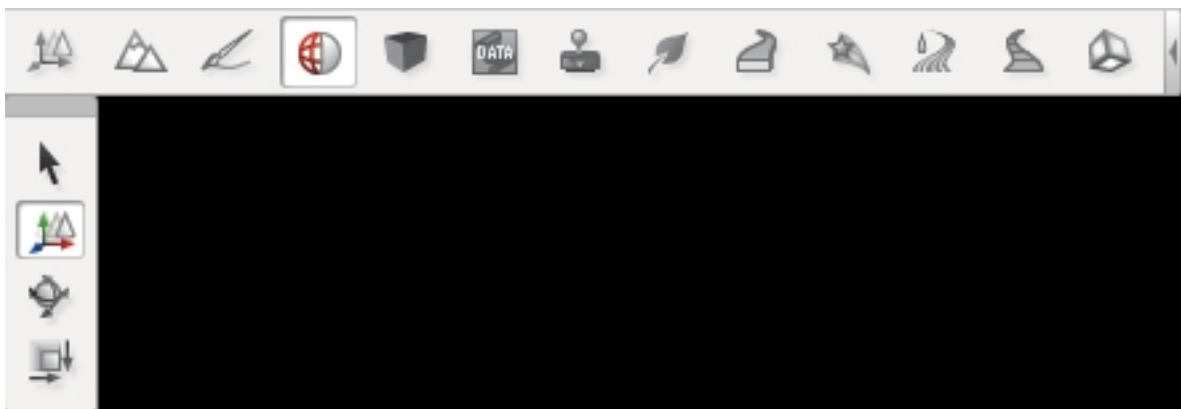


Fig. 4: Material Editor: Change texture and shader properties.

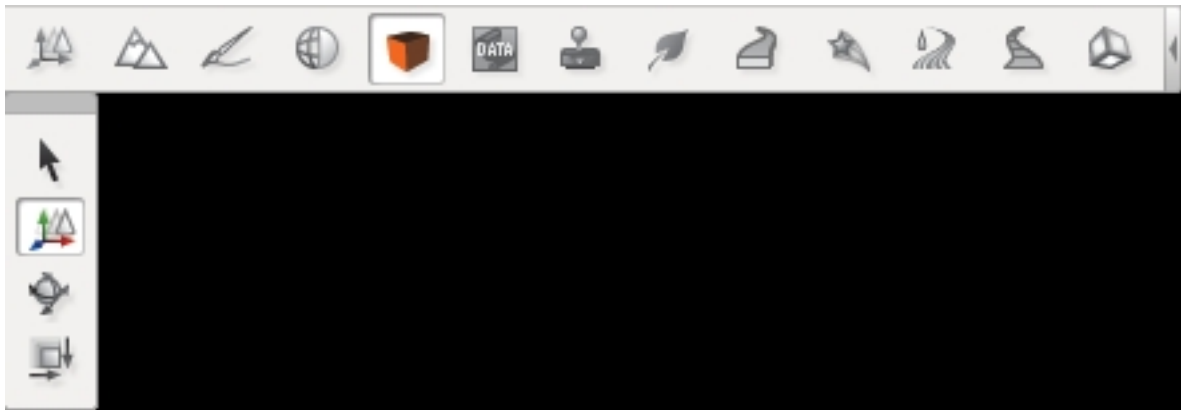


Fig. 5: Sketch Tool: Create prototype geometry for quick level layout testing.

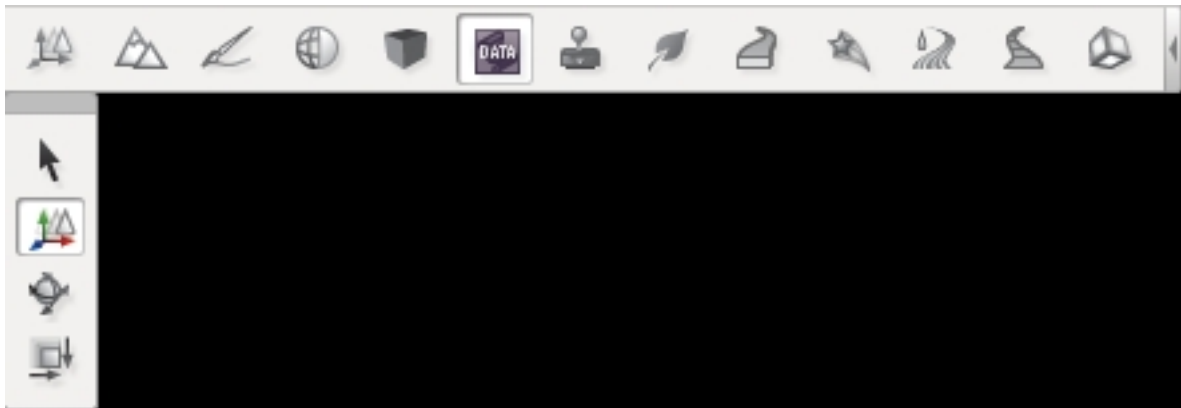


Fig. 6: Datablock Editor: Edit properties of objects in the scene.

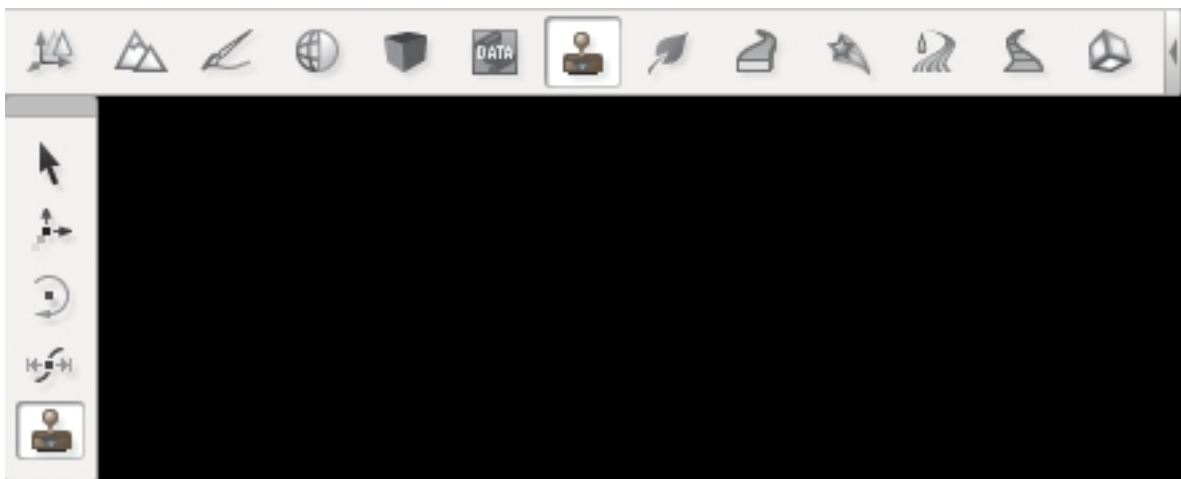


Fig. 7: Decal Editor: Edit decals and decal properties.



Fig. 8: Forest Editor: Edit forest areas in the scene.

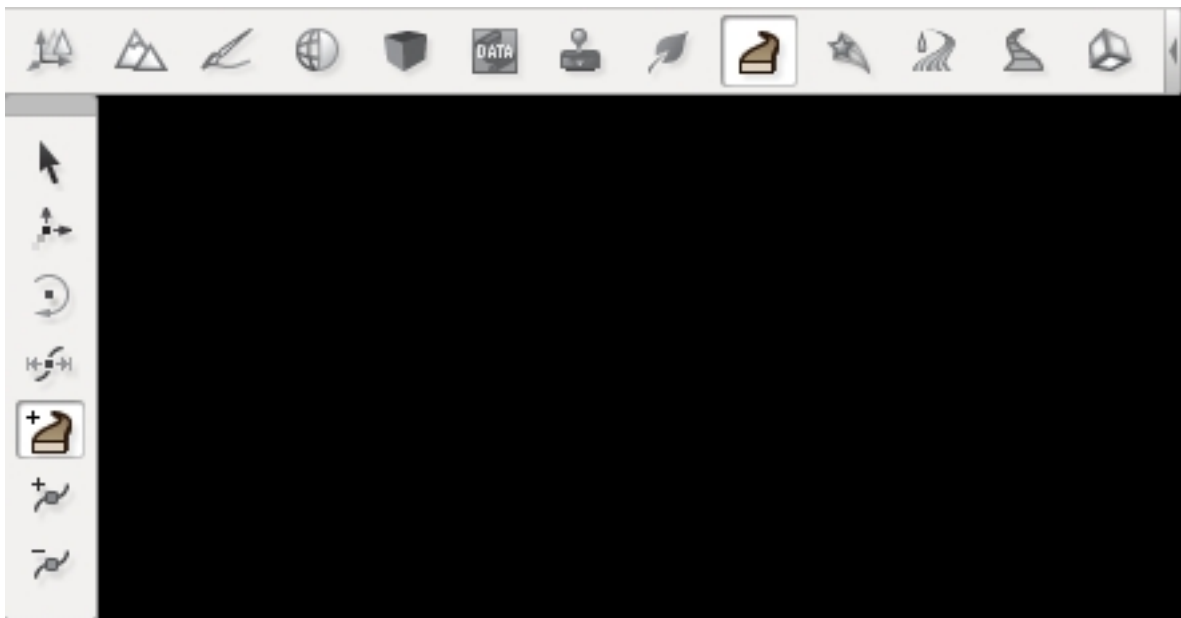


Fig. 9: Mesh Road Tool: Create mesh roadways along the terrain.



Fig. 10: Mission Area Editor: Edit the Mission Area object.

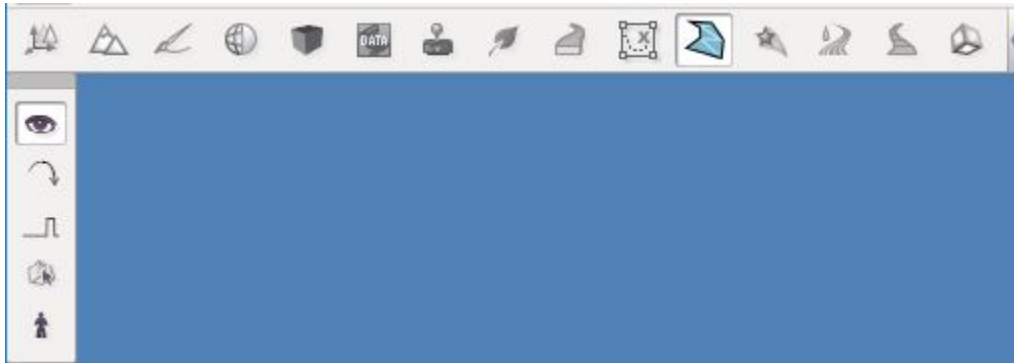


Fig. 11: Navigation Editor: Create and edit navigation mesh used in artificial intelligence (AI) for pathfinding.

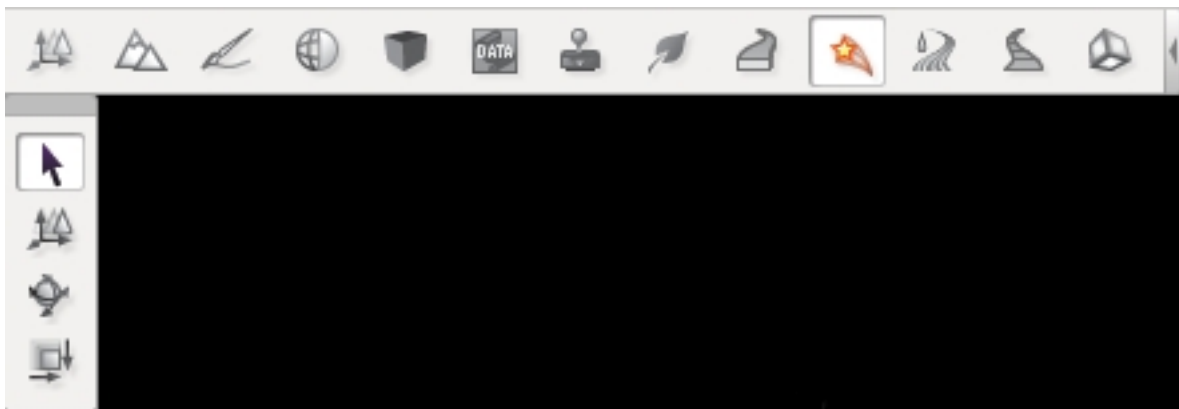


Fig. 12: Particle Editor: Create and edit particle properties for particle effects in the scene.

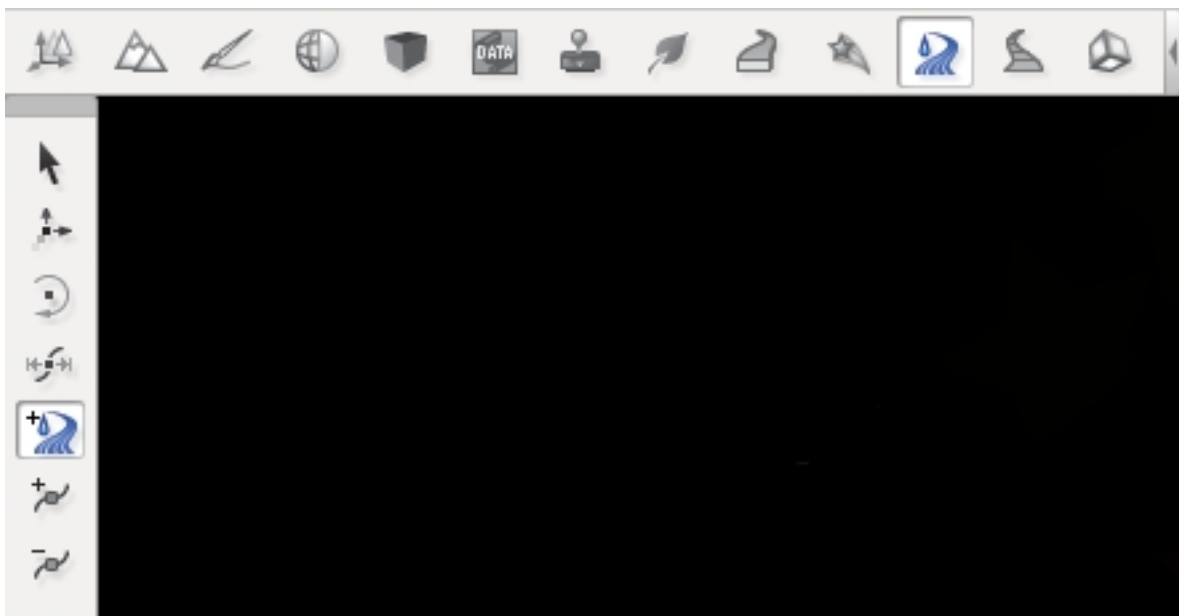


Fig. 13: River Tool: Create rivers in the scene.

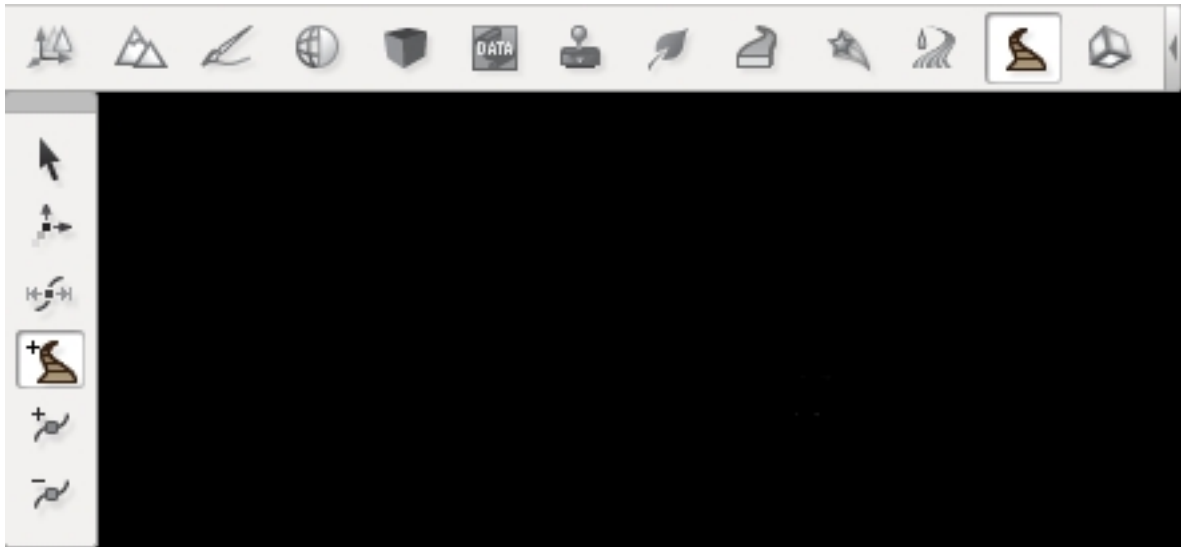


Fig. 14: Decal Road Tool: Create decal roadways in the scene.

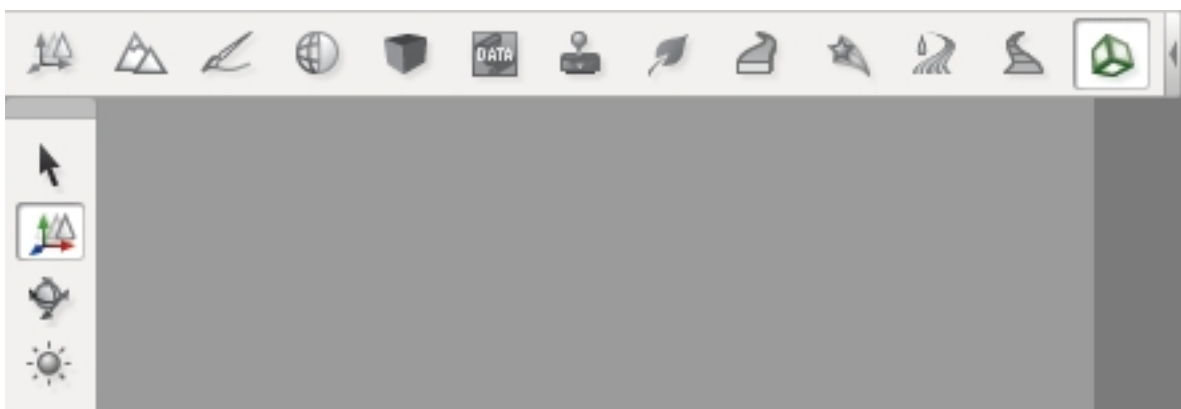
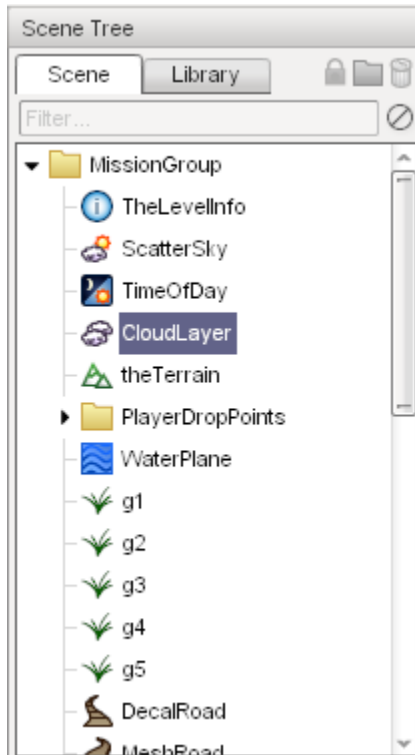


Fig. 15: Shape Editor: Edit, change, and set properties on meshes.

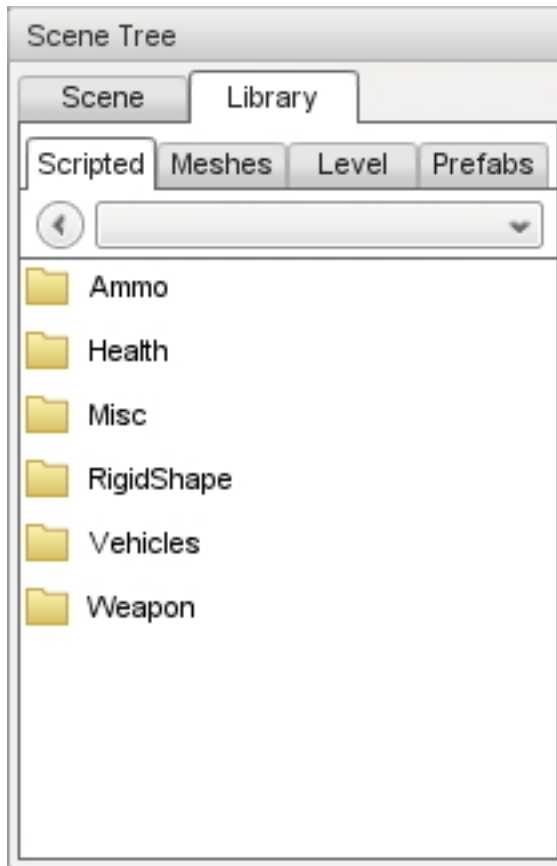


Each object in the tree has an icon, unique ID, an object type, and a name. Whenever you click on an object in the tree, it is selected in the level and vice versa. Most of your objects can stand alone in the tree, but you can also use a SimGroup object to organize related entries.

At first glance, a SimGroup looks like a folder and acts much like one to help organize your tree. It does not physically exist in your level, but you can reference it by name or ID from script or the engine. This is handy for grouping several game objects you might need to iterate through and invoke an action on. Even if you do not use that feature, it is still a good idea to group similar objects under a SimGroup to help organize and better navigate your trees as some levels can grow to a large number of objects.

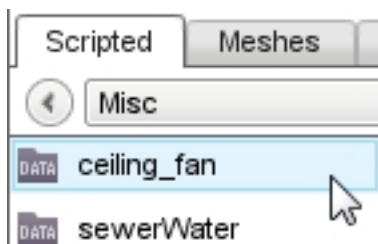
### 11.2.5 Library Tab

The Library tab is what you will use to add objects to your level. Once an object has been added to your level, it will appear in the Scene tab (described above). There are four sub-categories on the Library tab, which are separated as sub-tabs: Scripted, Meshes, Level, and Prefabs. Each category contains objects that serve very specific purposes.



### Scripted Tab

The first tab, Scripted, is automatically populated with game objects that have been created via script. For example, let's say you have a ceiling fan object with an associated script which controls how and when the fan blades rotate. It would appear in the Scripted tab as follows:

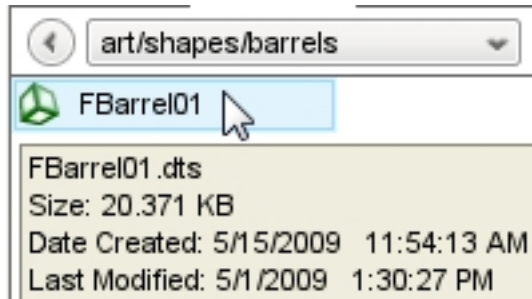


A discussion of scripting and how to associate scripts with an object is beyond the scope of this document. See the **TorqueScript Tutorial** for more information.

### Meshes Tab

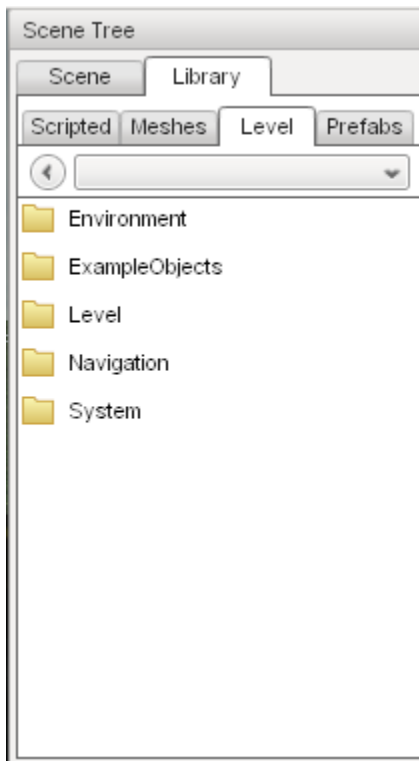
When you simply wish to add a 3D art asset, you will use the Meshes Tab. You can browse the various folders containing assets in your project's "art" directory. From here you can add DTS, COLLADA, and DIF files.



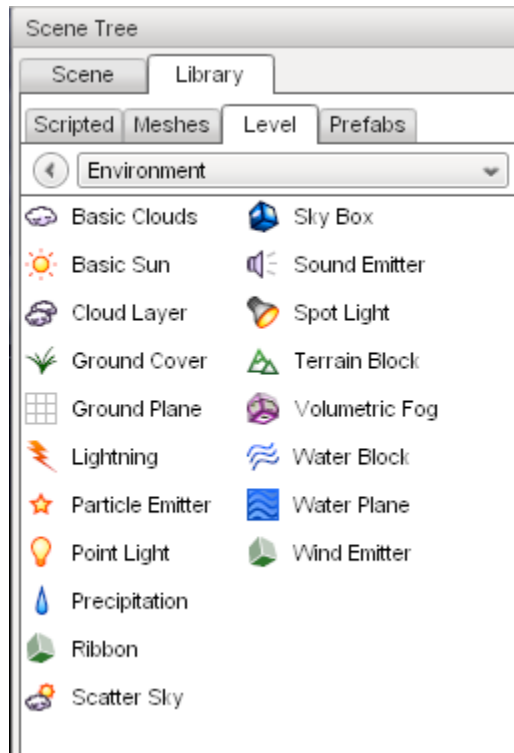


## Level Tab

The Level Tab lists all the Torque 3D objects that can be used to populate your level. Objects are further divided into category folders. To view what is in a folder, double click it. To leave a folder and view the folder list, click the left pointing arrow icon. To move directly to another folder, select it from the drop down list.



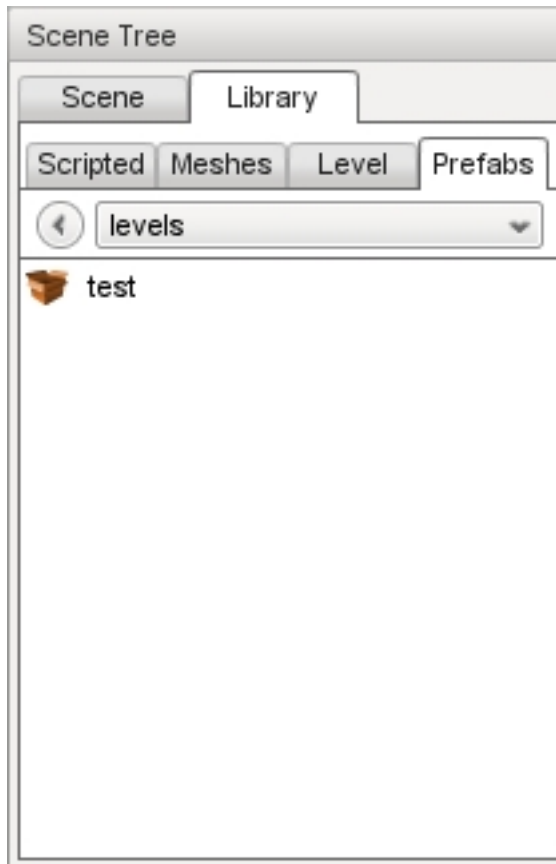
Each sub-category contains objects with similar themes:



- The Environment sub-category contains most of the objects you will add to your level, such as Terrain, Sun, Clouds, Waterblocks, and similar objects.
- The ExampleObjects sub-category contains example rendering classes created in C++.
- The Level sub-category contains objects that manage Time of Day, level boundaries, and similar objects.
- The System sub-category contains engine-level objects such as SimGroups.

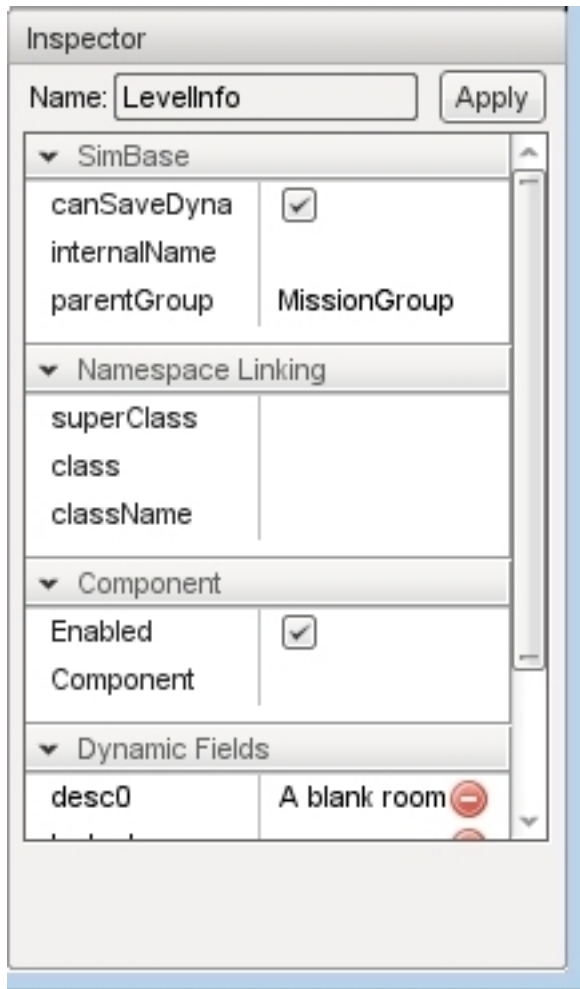
### Prefabs Tab

The prefab system allows you to group multiple objects together and combine them into a single file. This new object can then be repeatedly placed into your level as a whole, making it easier for you to add complex groups of objects with only a few mouse clicks. When you create a prefab from multiple selections, you will save it to a \*.prefab file using the group prefab icon. The World Editor will automatically load these files in the Prefabs tab.



### 11.2.6 Inspector

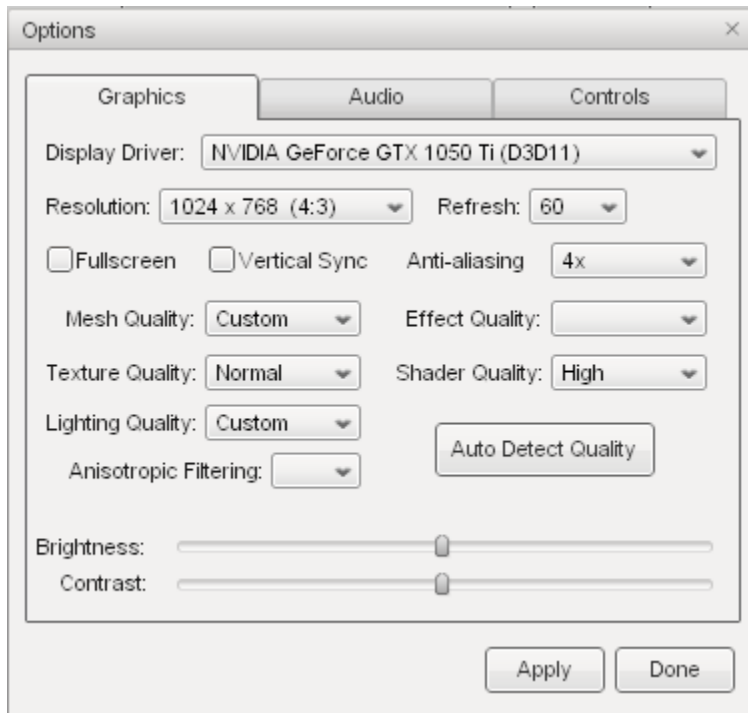
Whenever you add an object to a level, you will most likely start modifying them immediately. You can use the Inspector Panel to change the properties of an object **TODO - Update image**



While there are a few shared property sections, most object types will have a unique set of properties. Editing is as simple as selecting an object in the level, locating a field that you want to change, such as “className” or “media”, then either editing the existing value or entering a value if no default value is given. There are different types of values such as strings, numbers, check boxes, vectors, and even values that require the use of a file browser or color picker.

### 11.2.7 Options

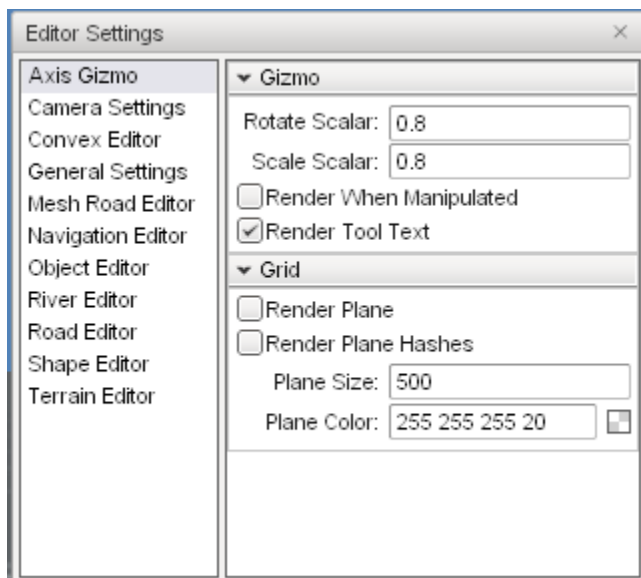
The Options dialog is used to change your current session’s audio and video properties as well as mouse and keyboard control bindings. The Options dialog is accessed from the main menu by selecting Edit > Game Options... in the **Full Template**



You will use the Graphics tab to adjust your game resolution, screen mode, detail levels, and so on. The Audio tab allows you to adjust your current game's volume, both globally and channel specific.

### 11.2.8 World Editor Settings

The World Editor Setting dialog is important to editing.



Through this dialog, you can change various aspects of how your tools render and function. The Axis Gizmo section will control what is rendered on your object, such as the Axis gizmo while moving the object, increase/decrease the scalar. You can also adjust the rendering of the editing plane in relation to the object.

The Camera section here you can change the default values and adjust it to your needs, like invert the Y axis, camera

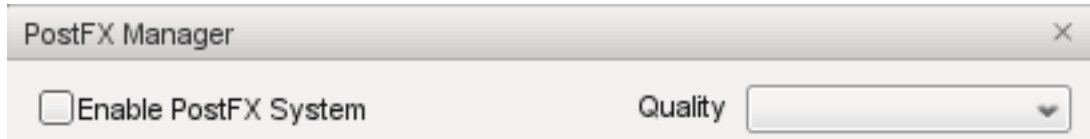
speed, etc

The Object editor section will allow you to modify the render text of the object or icon and change some colors.

There are several options you can tweak the sensitivity, add defaults options, adjust colors, adjust visibility or have more precise or dramatic modifications.

### 11.2.9 PostFX Manager

The PostFX Manager GUI allows level editors to control various post-processing effects. Select the *Enable PostFX* checkbox to toggle PostFX on and off.



Use the effect tabs to access the effect settings; select one of the effect tabs to view details and an in-game example of the effect, and use the checkbox to toggle the current effect on and off.

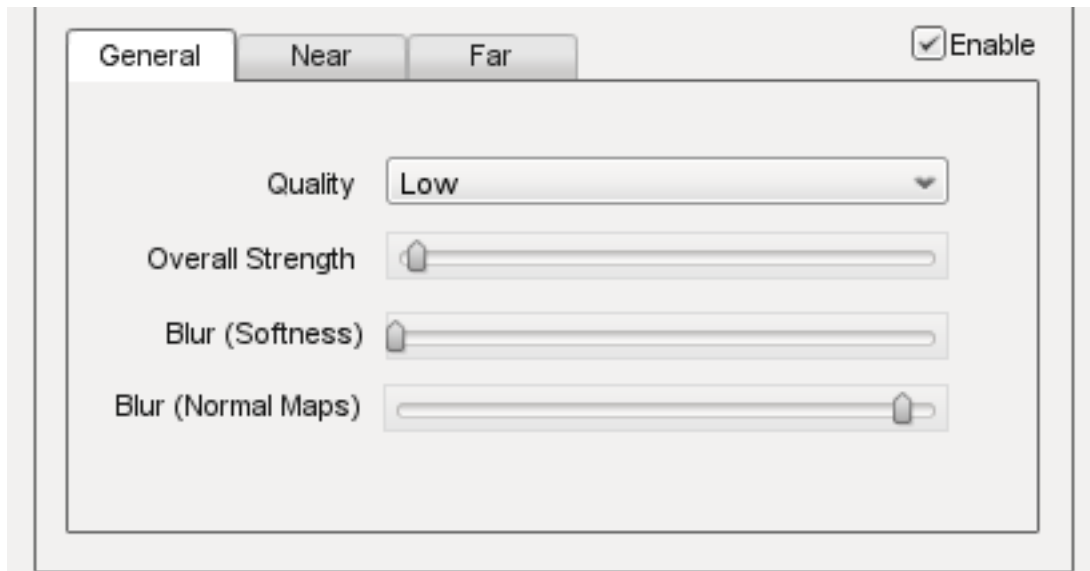


PostFX settings can be saved to file and loaded automatically with the level. To achieve this, simply save the settings with the same name as the level file. For example, for Burg.mis, save the PostFX settings in a file called Burg.postfxpreset.cs in the same folder as the level file.



#### SSAO

Screen space ambient occlusion (SSAO) is an approximation of true Ambient Occlusion. Enabling the effect will darken creases and surfaces that are close together. Outdoor areas with brighter ambient light will show the effect better.

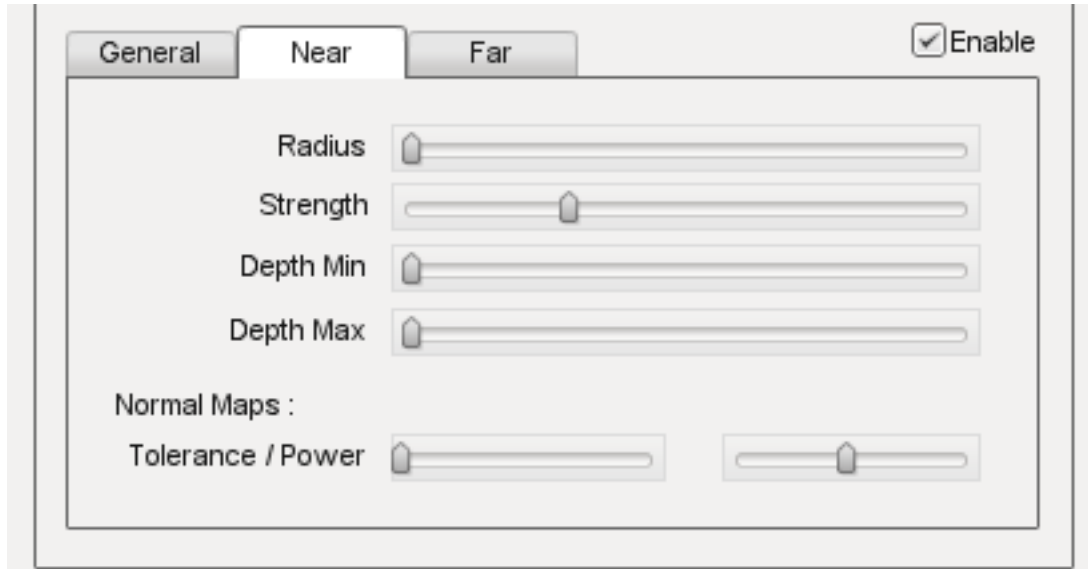


**Quality** Controls the number of ambient occlusion samples taken; higher quality is more expensive to compute.

**Overall Strength** Controls the overall intensity/darkness of the effect (applied on top of near/far strength).

**Blur (Softness)** Blur depth tolerance.

**Blur (Normal Maps)** Blur normal tolerance.



SSAO parameters for pixels near to the camera (small depth values).

**Radius** Occlusion radius.

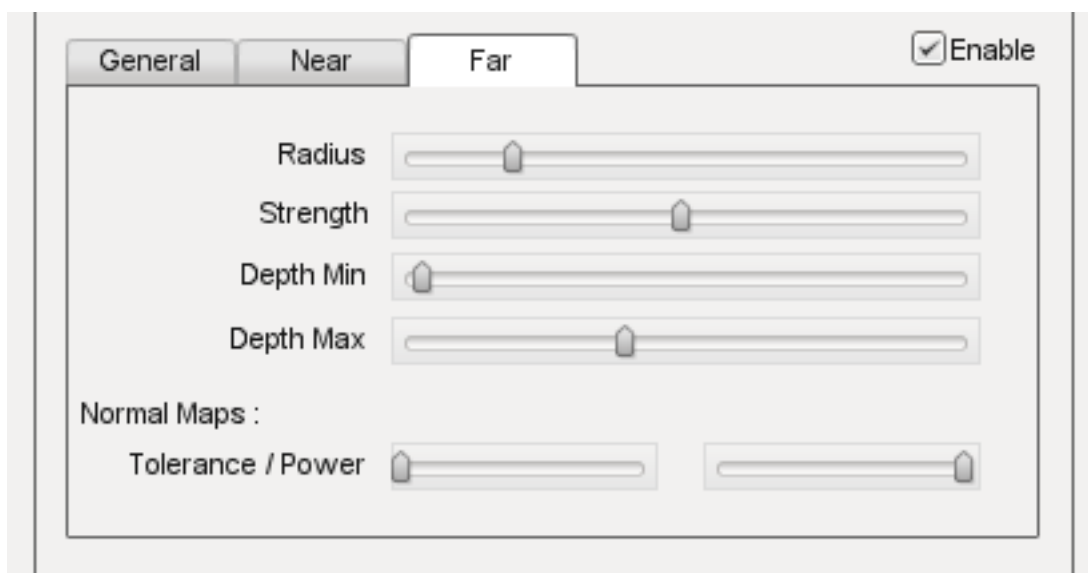
**Strength** Occlusion intensity/darkness.

**Depth min** Minimum screen depth at which to apply effect.

**Depth max** Maximum screen depth at which to apply effect.

**Tolerance** *Unused*

**Power** *Unused*



SSAO parameters for pixels far away from the camera (large depth values).

**Radius** Occlusion radius.

**Strength** Occlusion intensity/darkness.

**Depth min** Minimum screen depth at which to apply effect.

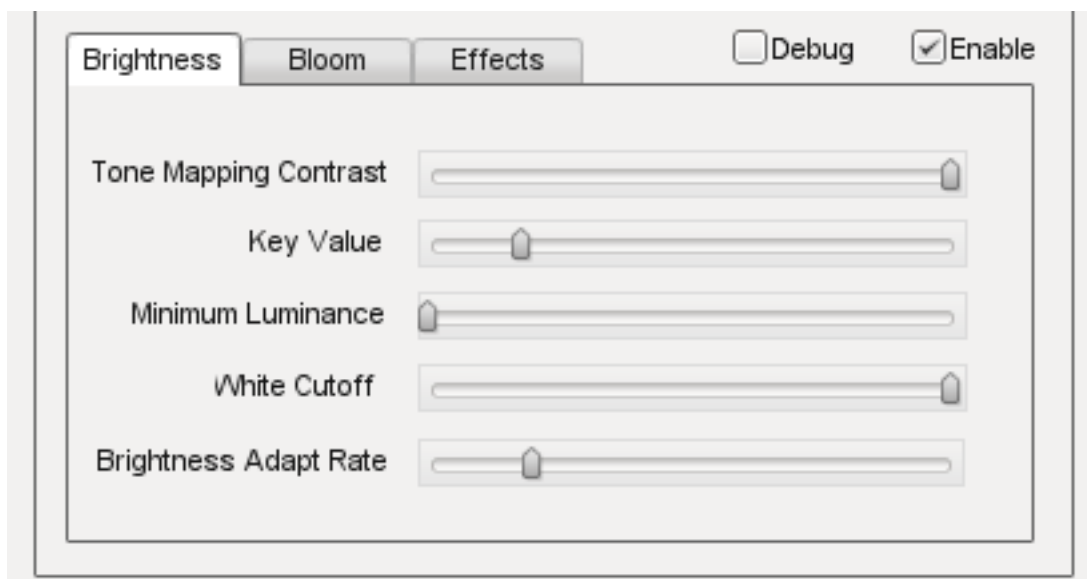
**Depth max** Maximum screen depth at which to apply effect.

**Tolerance** *Unused*

**Power** *Unused*

## HDR

Control several High Dynamic Range (HDR) effects including Bloom and Tone mapping.



**Tone Mapping Contrast** Amount of interpolation between the scene and the tone mapped scene.

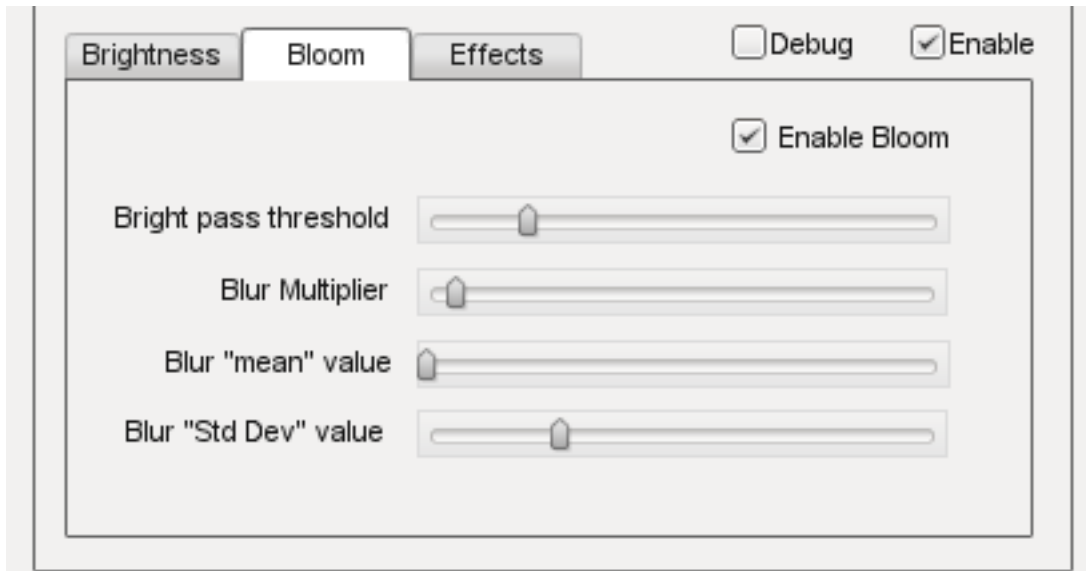
**Key Value** The tone mapping middle grey or exposure value used to adjust the overall “balance” of the image.

**Minimum Luminance** The minimum luminance value to allow when tone mapping the scene. Is particularly useful if your scene very dark or has a black ambient color in places.

**White Cutoff** The lowest luminance value which is mapped to white. This is usually set to the highest visible luminance in your scene. By setting this to smaller values you get a contrast enhancement.

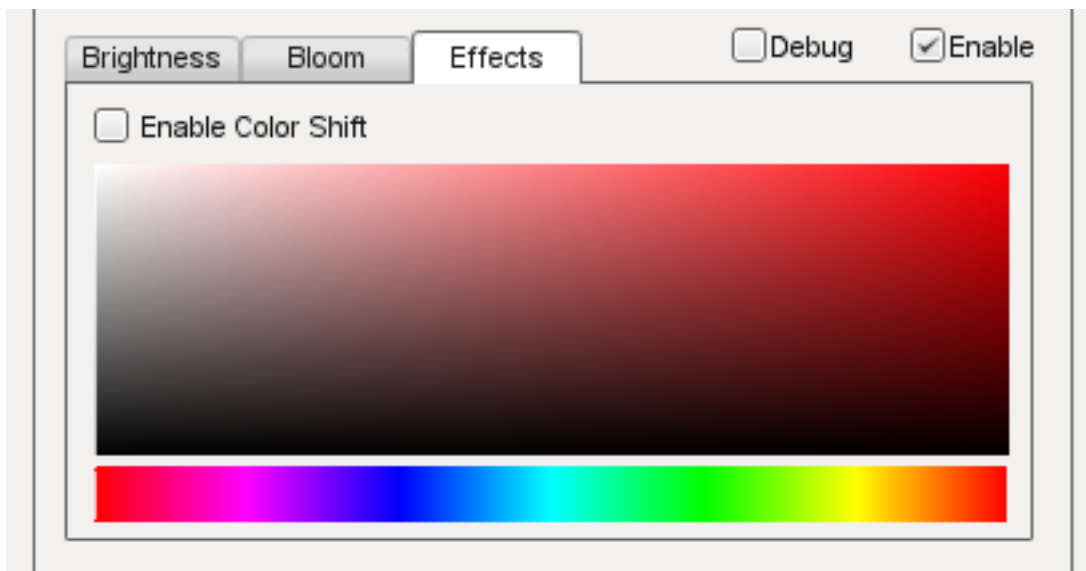
**Brightness Adapt Rate** The rate of adaptation from the previous and new average scene luminance.





**Bright Pass Threshold** The threshold luminance value for pixels which are considered “bright” and need to be bloomed.

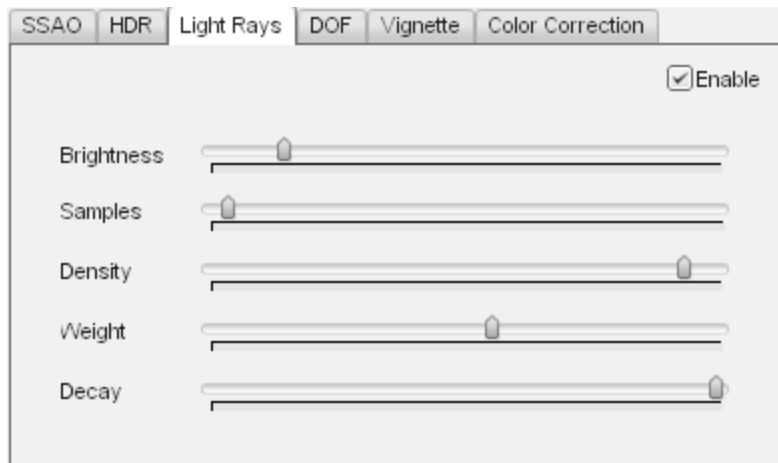
**Blur multiplier/mean/Std Dev** These control the gaussian blur of the bright pass for the bloom effect.



**Enable color shift** Enables a scene tinting/blue shift based on the selected color, for a cinematic desaturated night effect.

## Light Rays

This effect creates radial light scattering (also known as god rays). It works best when the scene contains a very bright light, in the example outpost level you should be able to see some scattering occurring around the trees.



**Brightness** Controls how bright the rays and the object casting them are in the scene.

**Samples** The number of samples for the shader.

**Density** Controls the density of the rays.

**Weight** Add or remove weight of the rays for a better effect.

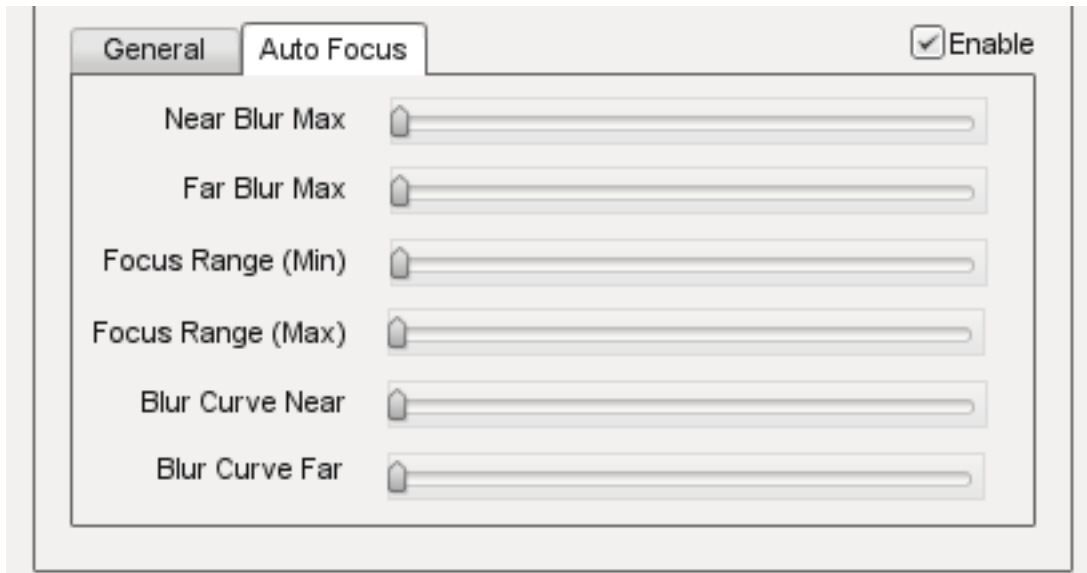
**Decay** Controls the decay of the rays.

## DOF

Depth of Field (DOF) simulates a camera lens, and blurs pixels based on depth from the focal point. DOF is commonly used when zooming in with a weapon.



**Enable Auto Focus** Determines how the focal depth is calculated. When auto-focus is disabled, focal depth is set manually by calling `DOFPostEffect::setFocalDist`. When auto-focus is enabled, focal depth is calculated automatically by performing a raycast at the screen-center.



**Near/Far Blur Max** Sets maximum blur for pixels closer/further than the focal distance.

**Focus Range (Min/Max)** The min and max range parameters control how much area around the focal distance is completely in focus.

**Blur Curve Near/Far** Controls the gradient of the near/far blurring curve. A small number causes bluriness to increase gradually at distances closer/further than the focal distance. A large number causes bluriness to increase quickly.

## Vignette

This effect add a vignette around the vision of the player, like if your where using a helmet or goggles.



**Radius** Adjust the maximum exposure of vignetting.

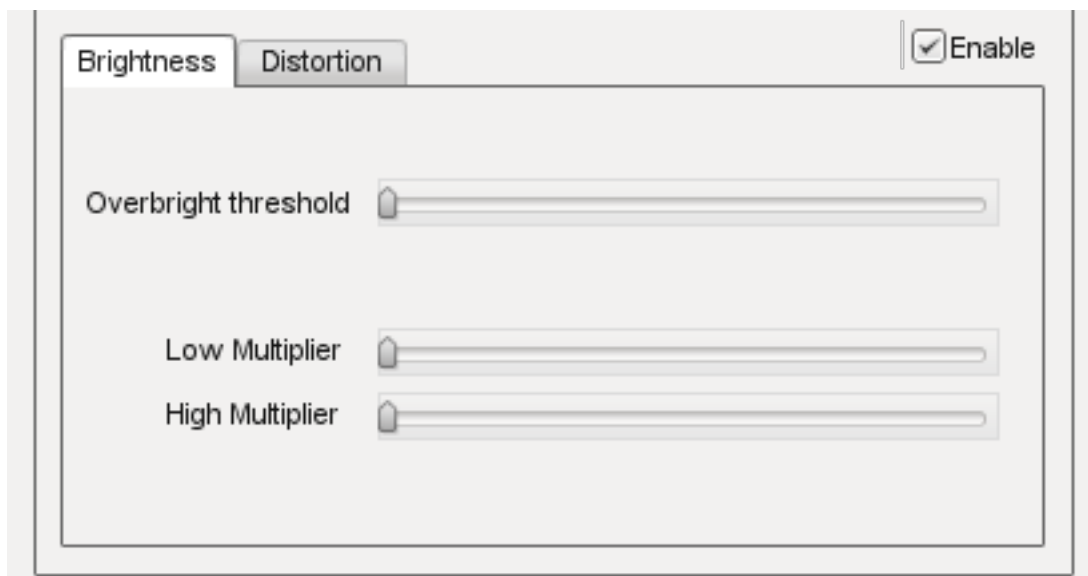
## Sharpness

This effect is currently unavailable.

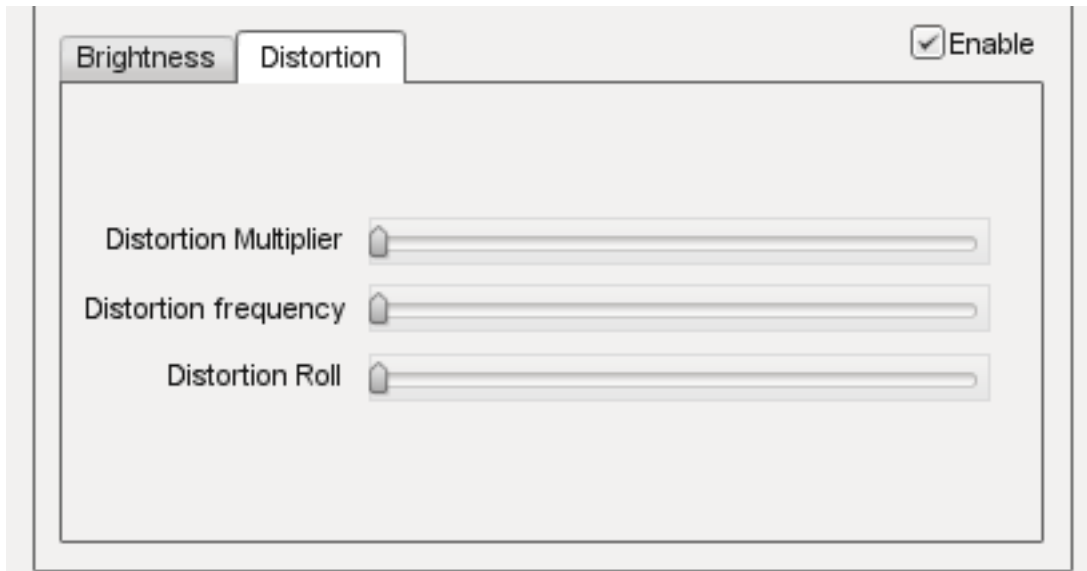


## Nightvision

This effect is currently unavailable.



This effect is currently unavailable.



### 11.2.10 Manipulators

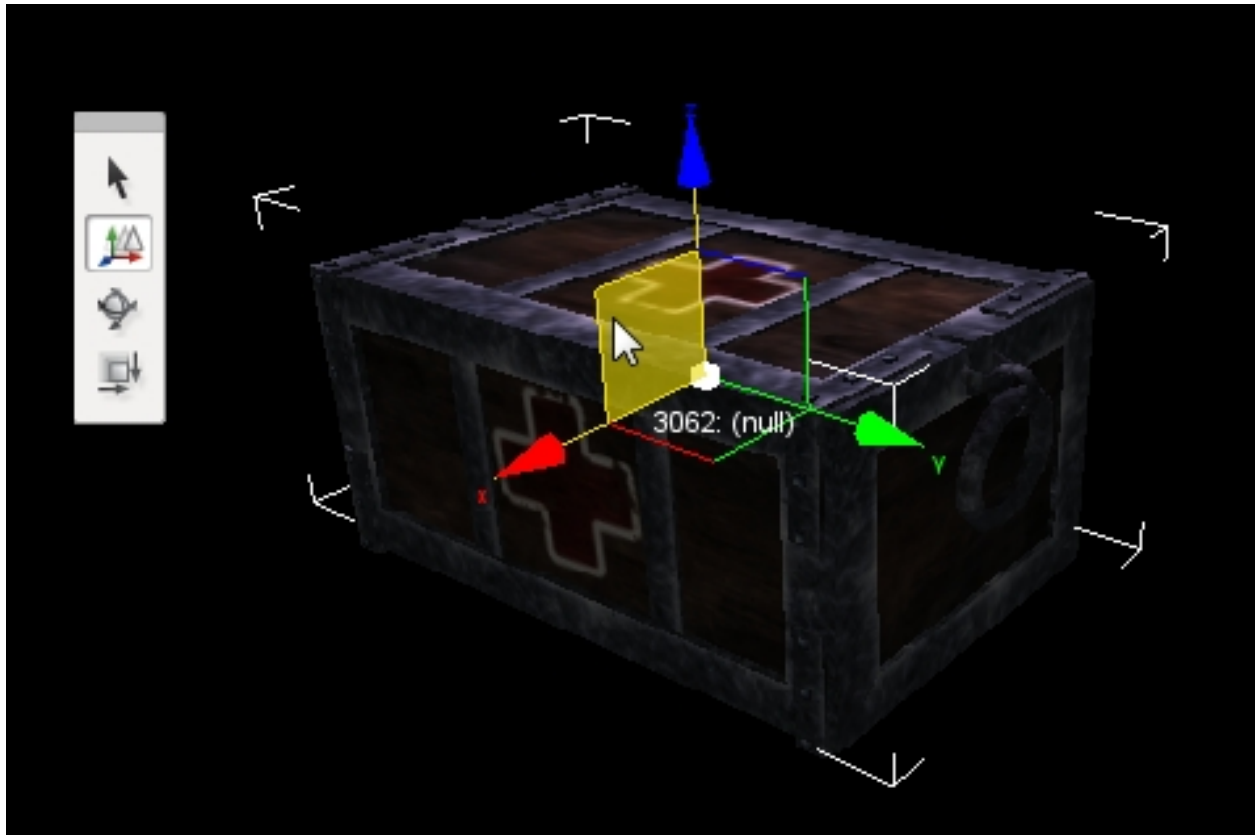
The last World Editor visual we will describe is the gizmo. A gizmo is a three dimensional rendering of an object's transforms. While using the Object Editor tool, you can use a gizmo to adjust an object's location, rotation, and scale without having to manually input number values in the Inspector Panel.

Each gizmo has a unique appearance to notify you of what you are adjusting based upon the tool that you are using.

#### Move Tool Gizmo

When you wish to move an object from one place to another, you will use the Move Tool. This is represented by a gizmo with arrows pointing toward different axes.

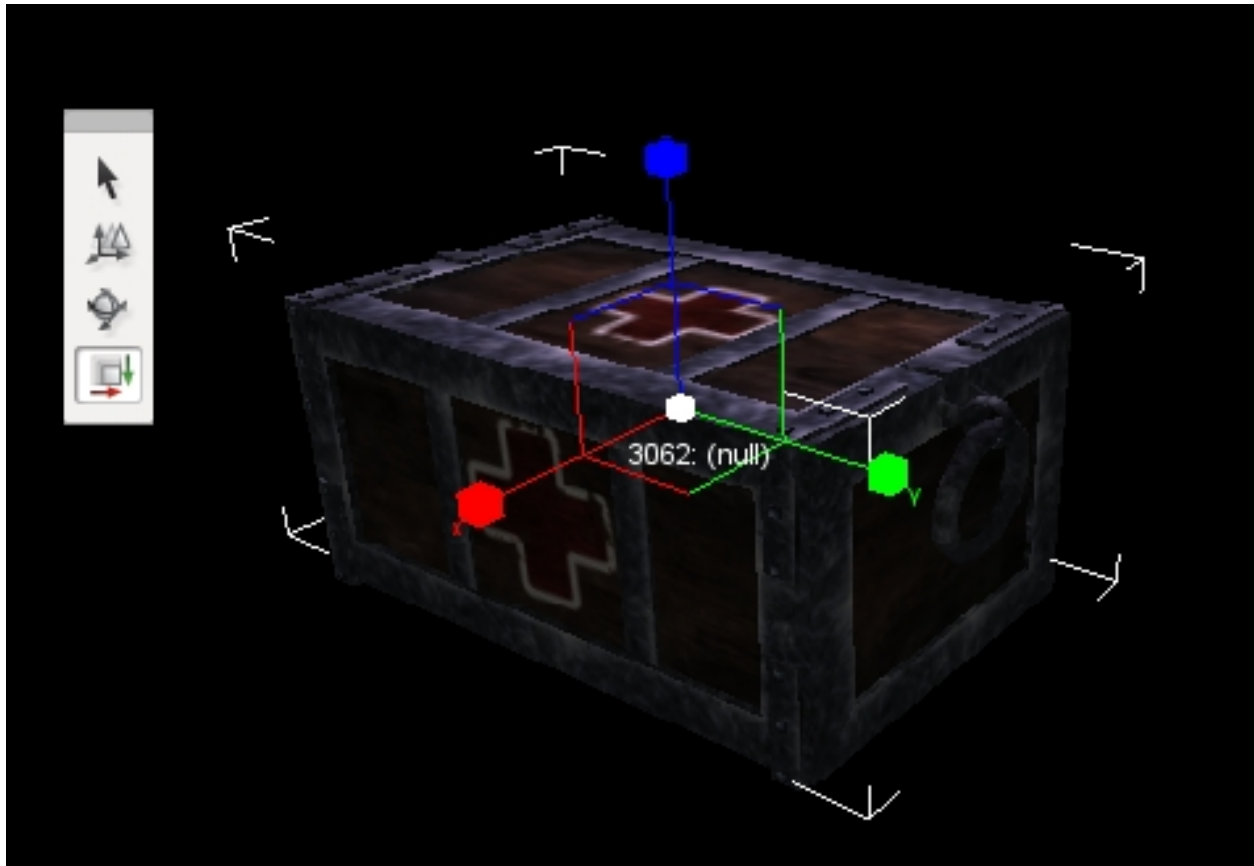
You can grab an arrow to move the object along an axis, or grab a space between two arrows to move it in both directions.



If you look carefully, you should see letters at the end of each arrow. These correspond to Torque 3D's world coordinate system. The engine utilizes the right-handed (or positive) Cartesian coordinate system, where Z is up (top), X is side (right), and Y is front (forward). This applies to the rest of the gizmos.

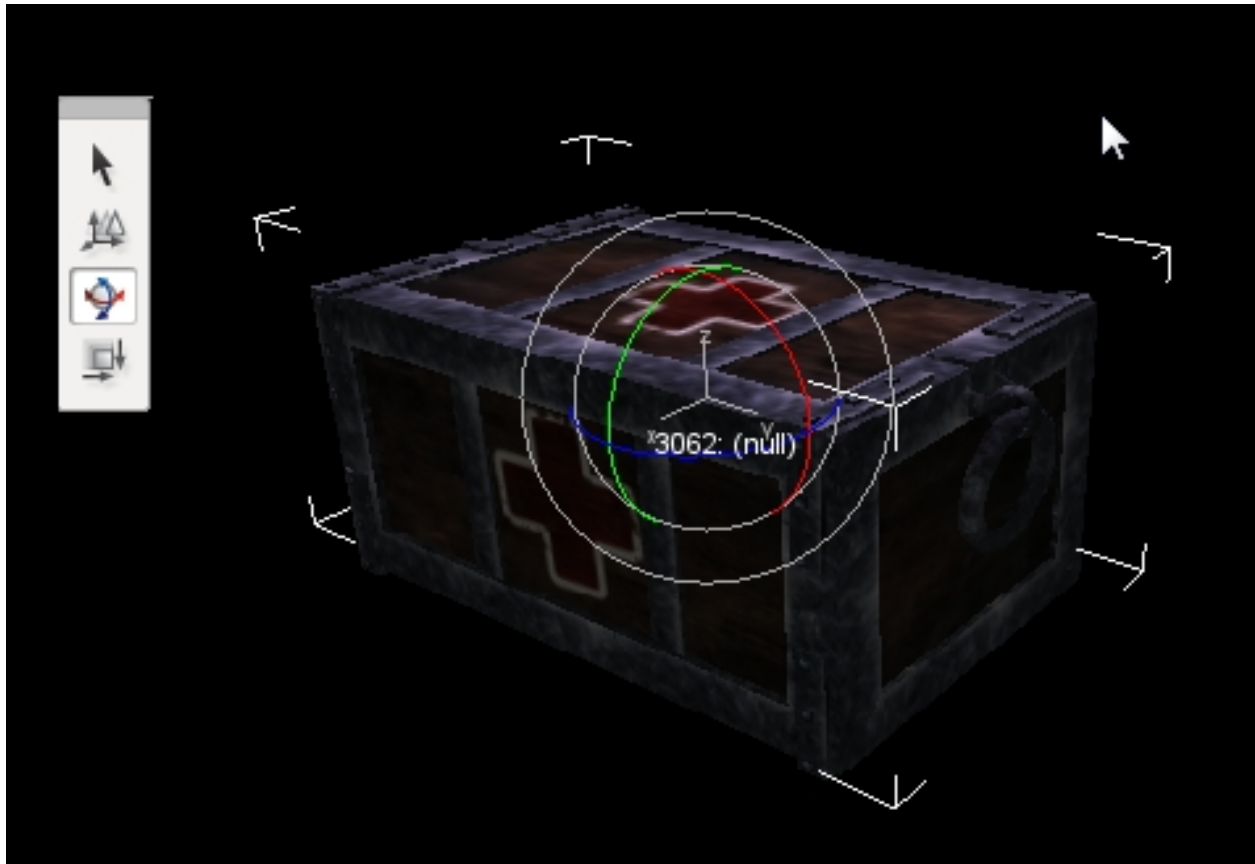
### Scaling Tool Gizmo

The Scaling Tool is represented by a gizmo that looks similar to the Translate gizmo. Instead of arrows, there are blocks at the end of the gizmo lines. Dragging one of the boxes in a direction will shrink or grow your object, depending on which direction you move.



### Rotation Tool Gizmo

While using the Rotation Tool, the orientation gizmo will be rendered. This gizmo looks and acts much differently than the previous two. Instead of straight lines, multiple circles will surround your object.



Dragging the red circle in a direction will rotate the object along the X-Axis. Green rotates around the Y-Axis. Blue rotates around the Z-axis. The off color circles allow you to rotate an object along multiple axes.



## CHAPTER 12

---

### Adding Objects

---



### 13.1 Terrain Editor

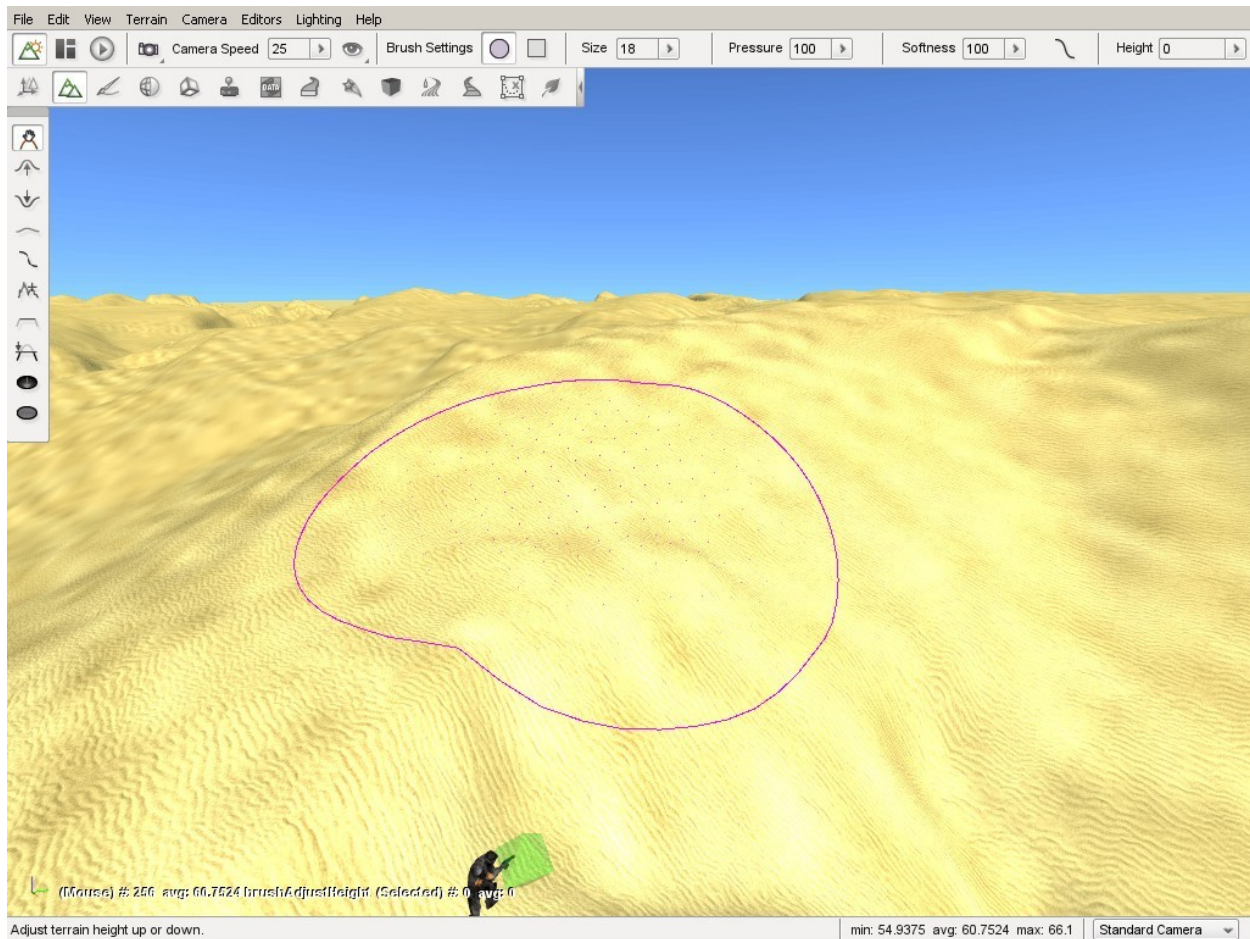
The Terrain Editor is used to modify a TerrainBlock's surface in real time within the World Editor.

With the Terrain Editor, you can elevate, excavate, smooth, flatten, and randomize sections of your terrain as if you were painting on the ground with a simple set of brushes. This is a great way to add the final details and polish to your level before populating it with objects.

The Terrain Editor is a powerful tool and allows for more than just adding hills or holes. You can cut channels for rivers, generate valleys for a mountainous region, turn a rocky mountain chain into a series of smooth hills, and other similar advanced operations.

#### 13.1.1 Interface

To switch to the Terrain Editor press the F2 key or from the main menu select Editors > Terrain Editor.



There are three main areas of the interface. On the far left is the tool palette, which is used to select what kind of modification you wish to make.

**Grab Terrain** Allows you to manually raise or lower terrain under brush.

**Raise Height** Adds dirt to terrain beneath brush, thus elevating it.

**Lower Height** Excavates terrain below brush, thus lowering it and creating holes.

**Smooth** Smooths jagged terrain beneath brush, creating more rounded elevation.

**Smooth Slope** Smooth slopes in terrain.

**Paint Noise** Creates random divots, elevation, and depressions under brush. Used for adding detail.

**Flatten** Flattens terrain, elevated or excavated, to the level of brush's starting point.

**Set Height** Set terrain to fixed height.

**Clear Terrain** Make holes in terrain.

**Restore Terrain** Cover holes in terrain made with the Clear Terrain tool.

At the top, the Toolbar has updated to show various brush options. The brush options will change the intensity and pattern of your terrain modification.

**Shape** Toggles between a round or square brush.

**Size** Changes the size of the grid that makes up the brush, increasing or decreasing the amount of terrain being modified.

**Pressure** Determines the amount of modification being applied to the terrain.

**Softness** Determines how much of the brush is affected by the pressure and intensity.

**Softness Curve** Customize how softness is applied.

**Height** The brush starting height.

Finally, while moving the mouse over the terrain, you will see a circle drawn around the mouse cursor. This is the Terrain Editor brush and is controlled by your mouse. You can use this brush to “paint” your terrain adjustments by left clicking and dragging the cursor.

### 13.1.2 Brush Settings

Now we will go into deeper explanation of how the brush options can affect your terrain editing, and how the editor lets you know what you are doing. On the toolbar at the top of the screen, find the Brush Settings section.

#### Shape

You should see a circular icon and square icon. Toggling between the two will change the shape of your brush.

#### Size

In the Size section you will find a box with a number in it. When you click on the arrow, a slider will appear. This slider goes from 0 to 100, and it changes the size of your brush allowing it to modify larger sections of the terrain.

#### Pressure

The Pressure setting is a decimal number ranging from 0.0 to 100.0, and determines how much change is applied to the terrain under the brush.

The easiest way to understand brush pressure is to think of using a real paintbrush on a piece of paper. The harder you press as you stroke, the more paint the brush will leave on the paper. Pressing lightly but going over and over the same spot again and again will also leave more paint behind.

Since you can not press harder or softer on the terrain with a mouse button the Pressure value lets you simulate how much affect the brush has when you are “painting” terrain changes. The higher the Pressure setting, the more dramatic the change will be under the brush over time.

For example: select the Raise Height tool; set the Pressure to 1; place the mouse over the terrain; then quickly press the mouse button and release it. You will see the terrain slightly rise under the brush. Now change the Pressure to 100 and click elsewhere on the terrain in the same manner. You will see the terrain under the brush rise much more quickly and higher than during the first operation.

Like a real paintbrush, our terrain brush left more change behind because its pressure was greater. The same change can be accomplished by clicking the same spot on the terrain multiple times with a lower setting. The lower the setting, the more control you will have to make accurate changes to the terrain.

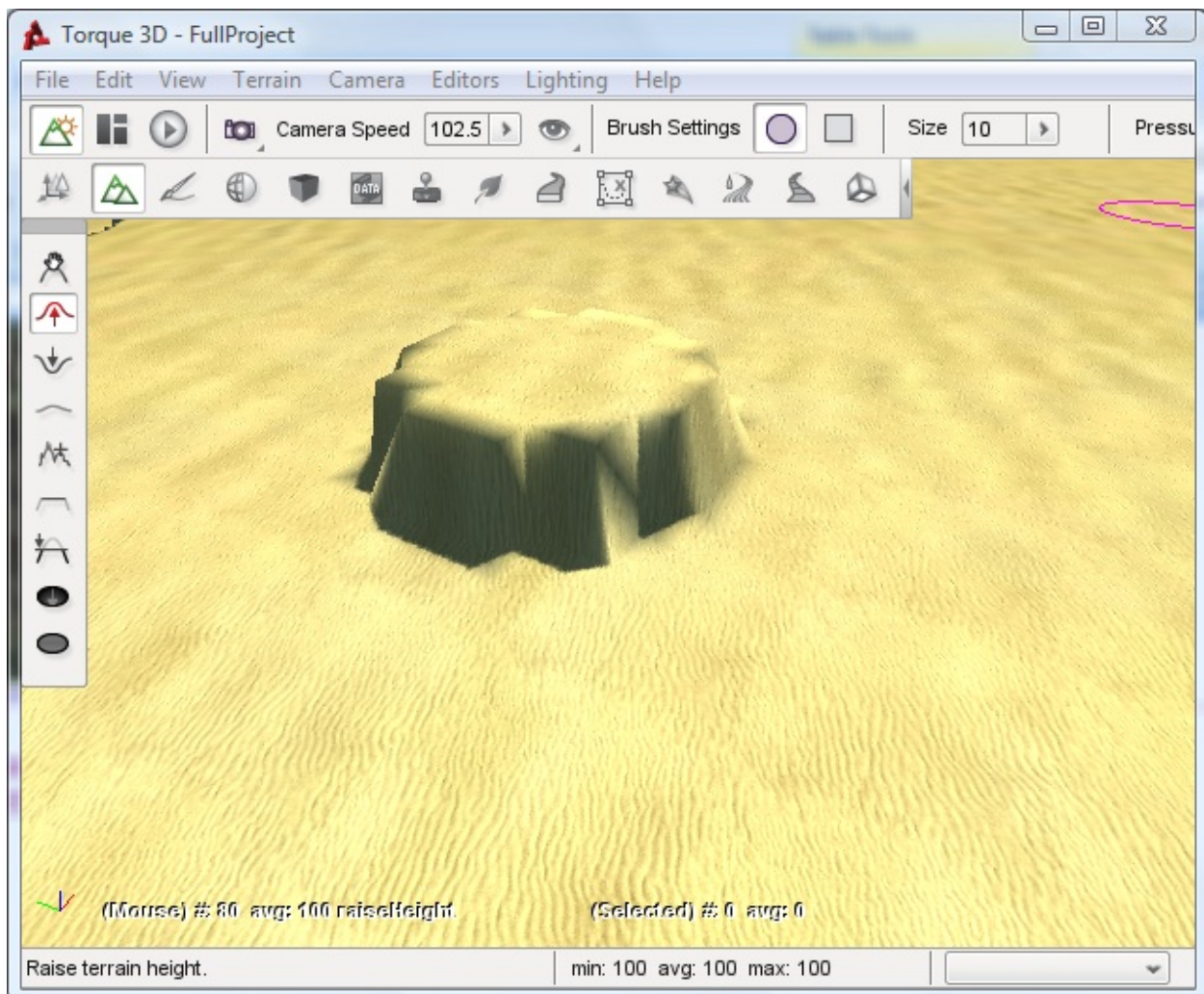
You will also notice that the ground rose higher in the center than around the edges of the brush. Again, this mimics the real world where the pressure around the edges of a paintbrush will be less because there are less bristles, which makes the edges of the brush softer than the center. Therefore the edges will leave less paint behind than the center.

## Softness

The Softness setting directly affects the intensity of the entire brush's surface. Like the others, it is represented by a slider. The number is a decimal value ranging between 0.0 and 100.0, with 100.0 being the softest and 0.0 the hardest.

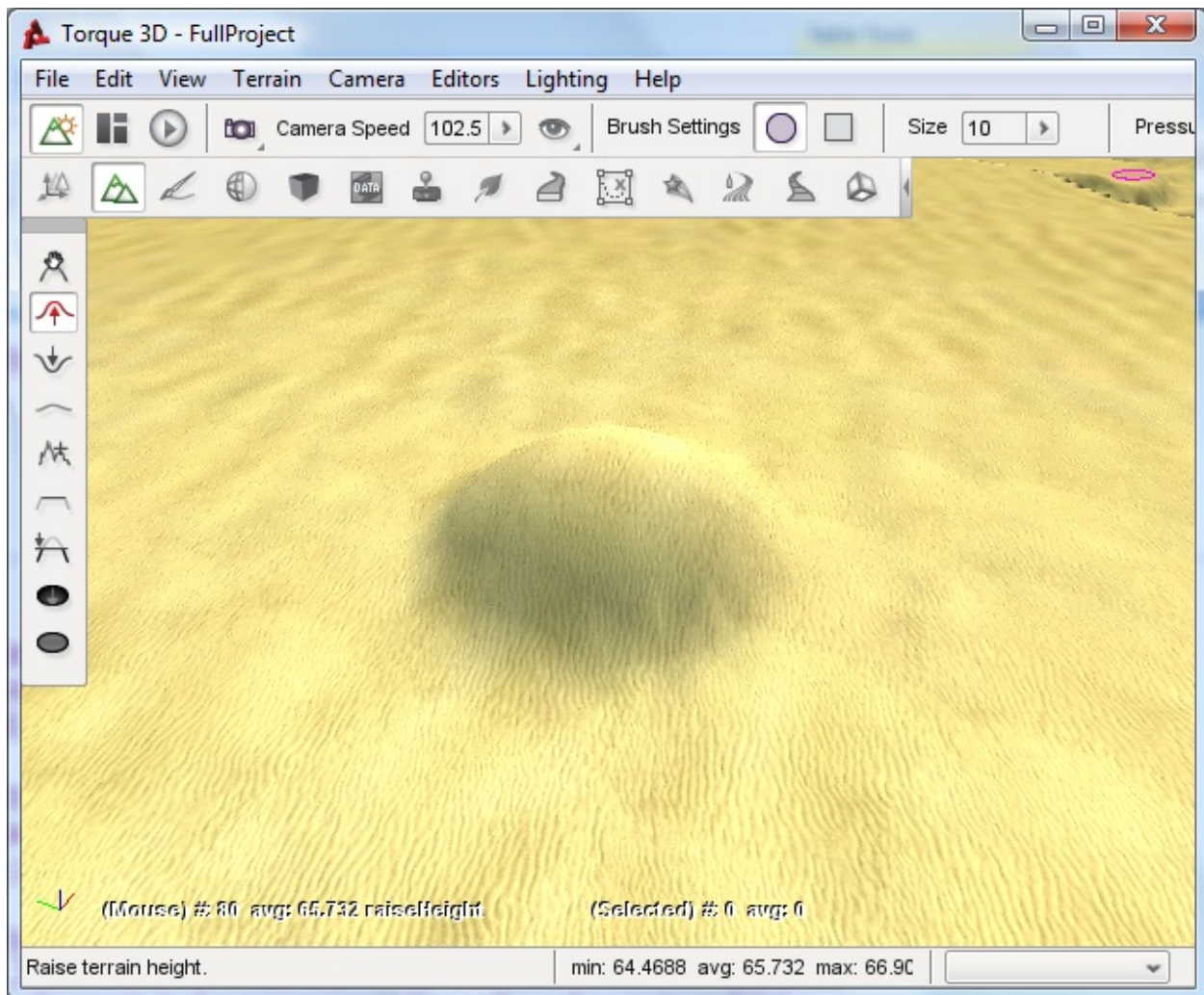
The harder the brush is, the more dramatically the terrain under the brush will change. Think of a paint brush that has been dipped in paint and allowed to somewhat dry. The entire brush will be harder and thus will leave more paint behind than the same brush which has not dried and hardened. Since the entire brush has hardened the pattern that it leaves will be the same, that is more paint in the center and less at the edges, but the overall amount of paint will increase across the whole surface. If you allow the brush to completely dry then the entire brush will be the same hardness and thus will leave the same amount of paint across its entire surface.

Because you can not change the softness of a mouse cursor, the Terrain Editor provides the Softness settings to emulate these characteristics. Setting the softness to 0.0, meaning the brush has no softness at all, will result in the entire brush being hard. The edges will be just as hard as the center and so the entire brush will leave the same amount of change behind. The result will be in a sharp rise between the terrain and the brush edges, producing a cliff.



Conversely, if you set the brush to 100.0, meaning maximum softness the brush will exhibit its natural behaviour returning to very soft edges and a harder center. Setting the Softness to 100.00 (maximum softness) will cause the change at the edges to be much less dramatic than the change in the center and will result in a gentle rise from the edges to the center, producing a rolling hill.

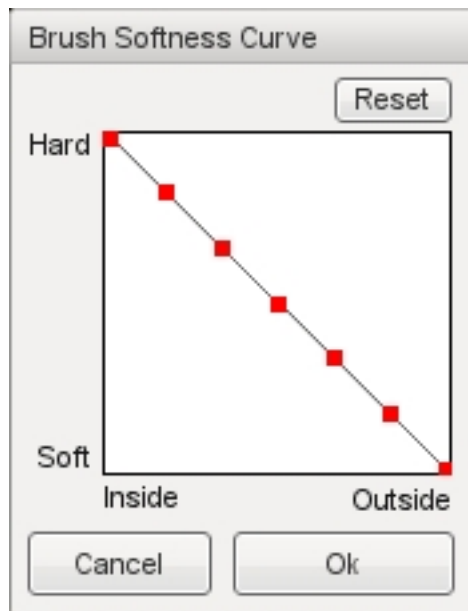




## Softness Curve

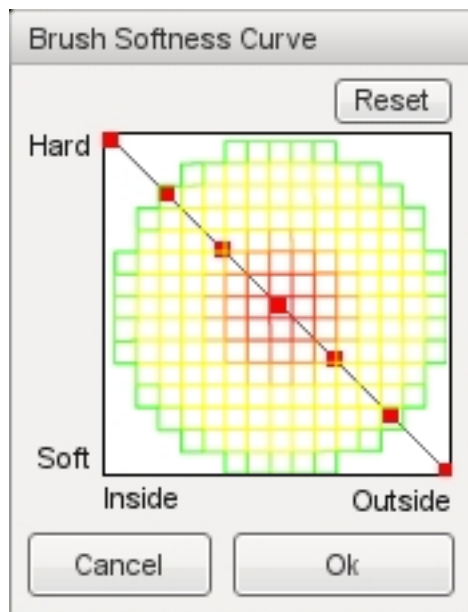
The previous section discussed how changing the Softness affects the brush over its entire surface mimicking the natural effects of a brush which is harder in the center and softer at the edges due to the distribution of its bristles. The Brush Softness Curve allows you to customize this behaviour further by changing the way softness and hardness is distributed within the brush.

Click the curved line next to the brush Softness slider. The Brush Softness Curve dialog box will appear.



The graph contains multiple nodes which can be moved by clicking and dragging them up or down. Modifying the nodes will determine which parts of your brush are hard or soft. As the graph shows, going from left to right will determine where in the grid you are changing the hardness.

Left nodes are closer to the center of the brush, and each node moving to the right will move toward the outer edges of the brush. The higher a node is situated, the harder it is. The following image is a visual of how this system works:



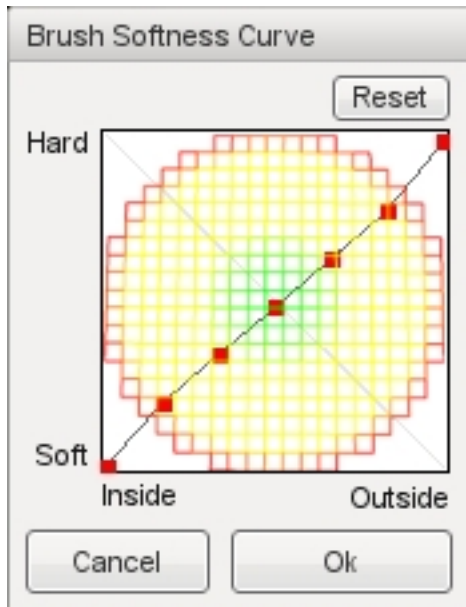
The circular pattern represents the shape of the brush looking straight on at its tip. Hardness of the brush is represented by red, softness by green, and yellow indicates variations in between. The node in the upper left represents the very center of the brush, since it is at the far left on the Inside-Outside axis. Because it is also at the very top of the Hard-Soft axis, it means that the brush is at its hardest at that location. So the combination of these two node positions indicates that the brush is at its hardest (indicated by red) in the very center.

On the other end of the graph line, the node in the lower right represents the very edge of the brush, since it is at the far right on the Inside-Outside axis. Because it is also at the very bottom of the Hard-Soft axis it means that the brush is at its softest at that location. So the combination of these two node positions indicates that the brush is at its softest

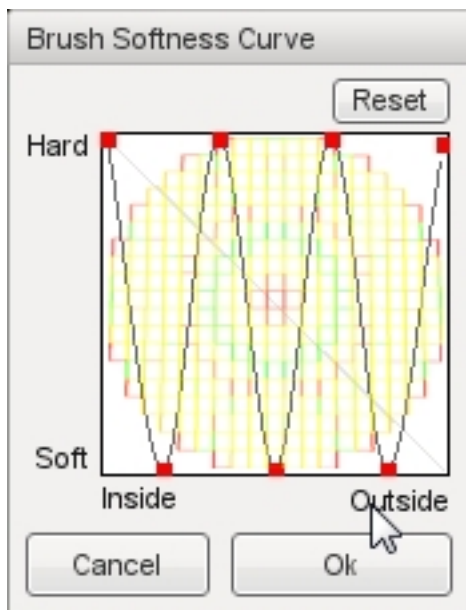


(indicated by green) all around the edge.

If you were to drag each node so that the line is reversed, the brush will be softer toward the center and harder toward the edges.



To get an unusual setting, you can create a “wavy” version of the curve. Alternating the nodes in extremes from top to bottom, will result in rings of softness with the brush.

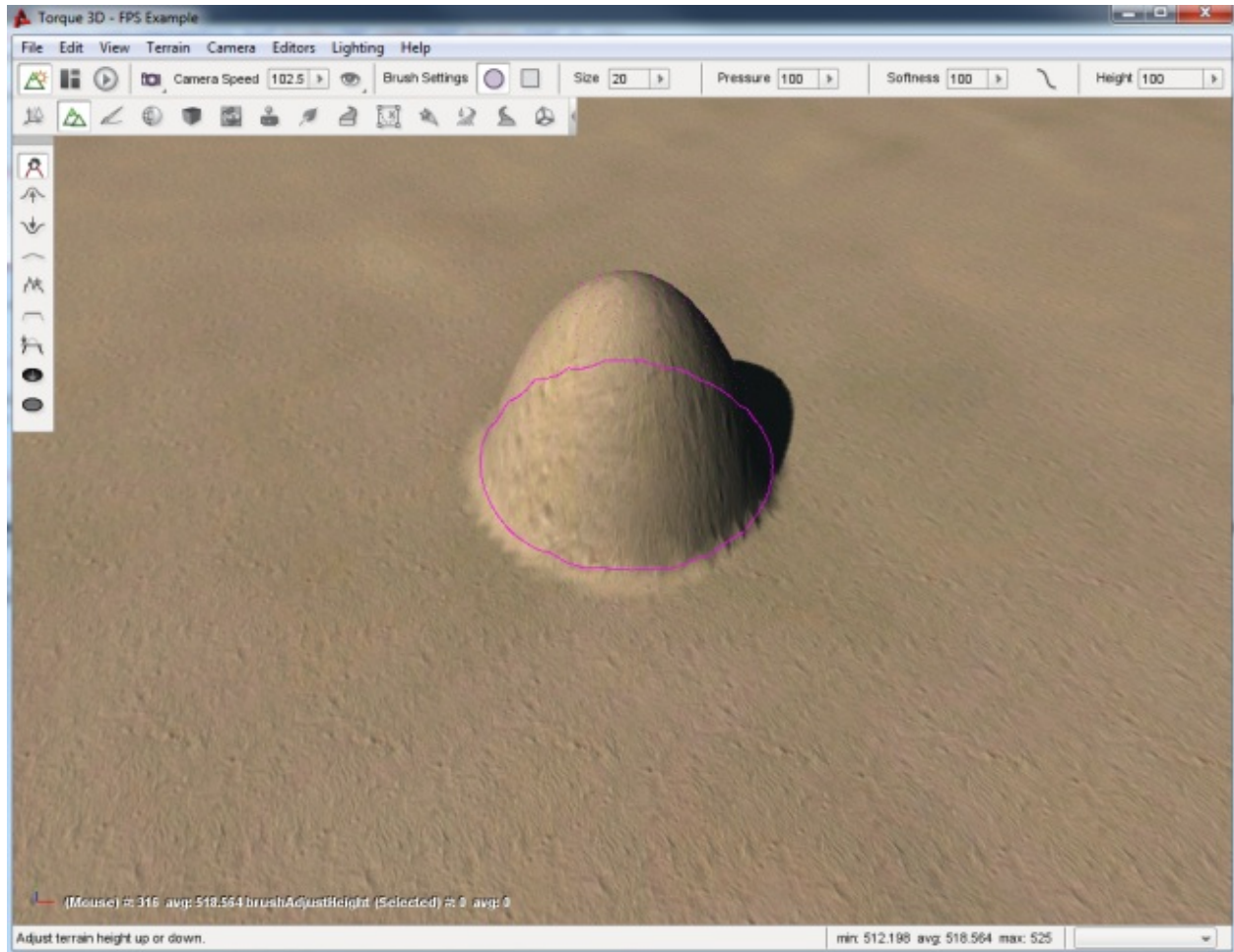


Now that you are familiar with the interface, it is time to edit the terrain.

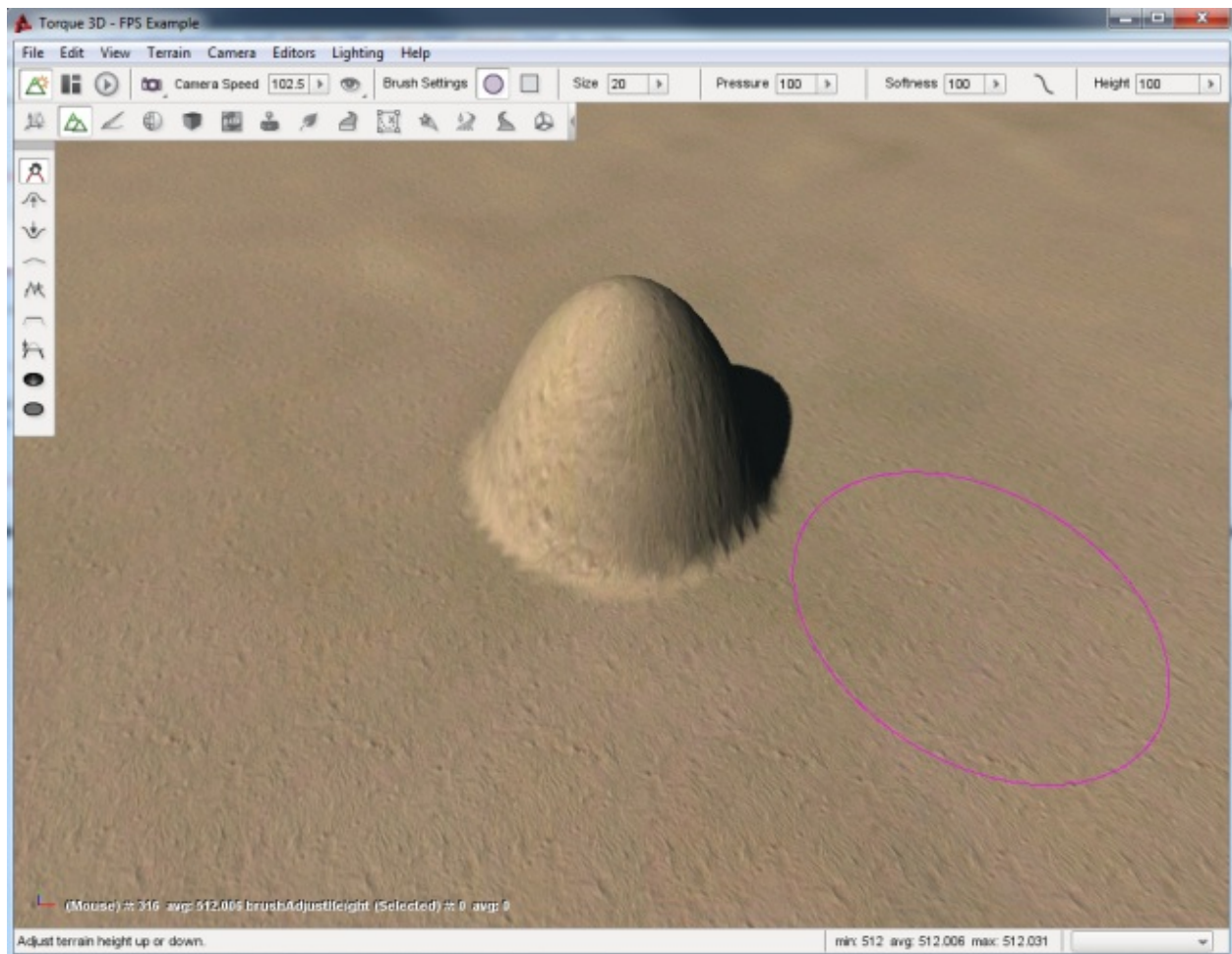
### 13.1.3 Grab Terrain Tool

Let's start by selecting the Grab Terrain tool from the palette. With the Grab Terrain tool, you can move a section of terrain up or down depending on which direction you are dragging your mouse.

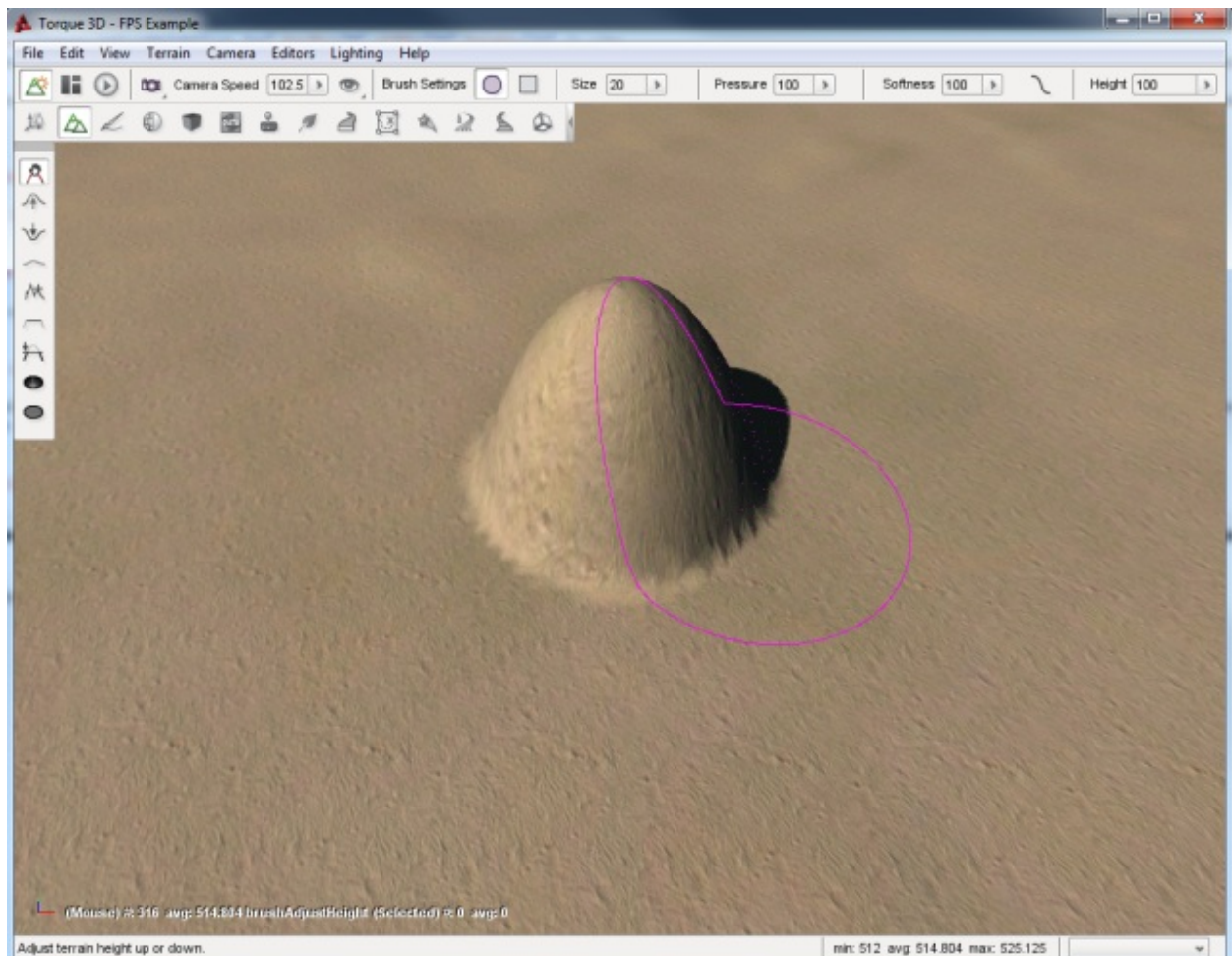
Use the circle brush, size 20, 100 pressure, and 100 softness. Hover your brush over a section of terrain, hold down the left mouse button, then move your mouse up. The terrain should dynamically adjust to your cursor location.



When you are satisfied with the height, let go of the mouse to see your terrain modification.

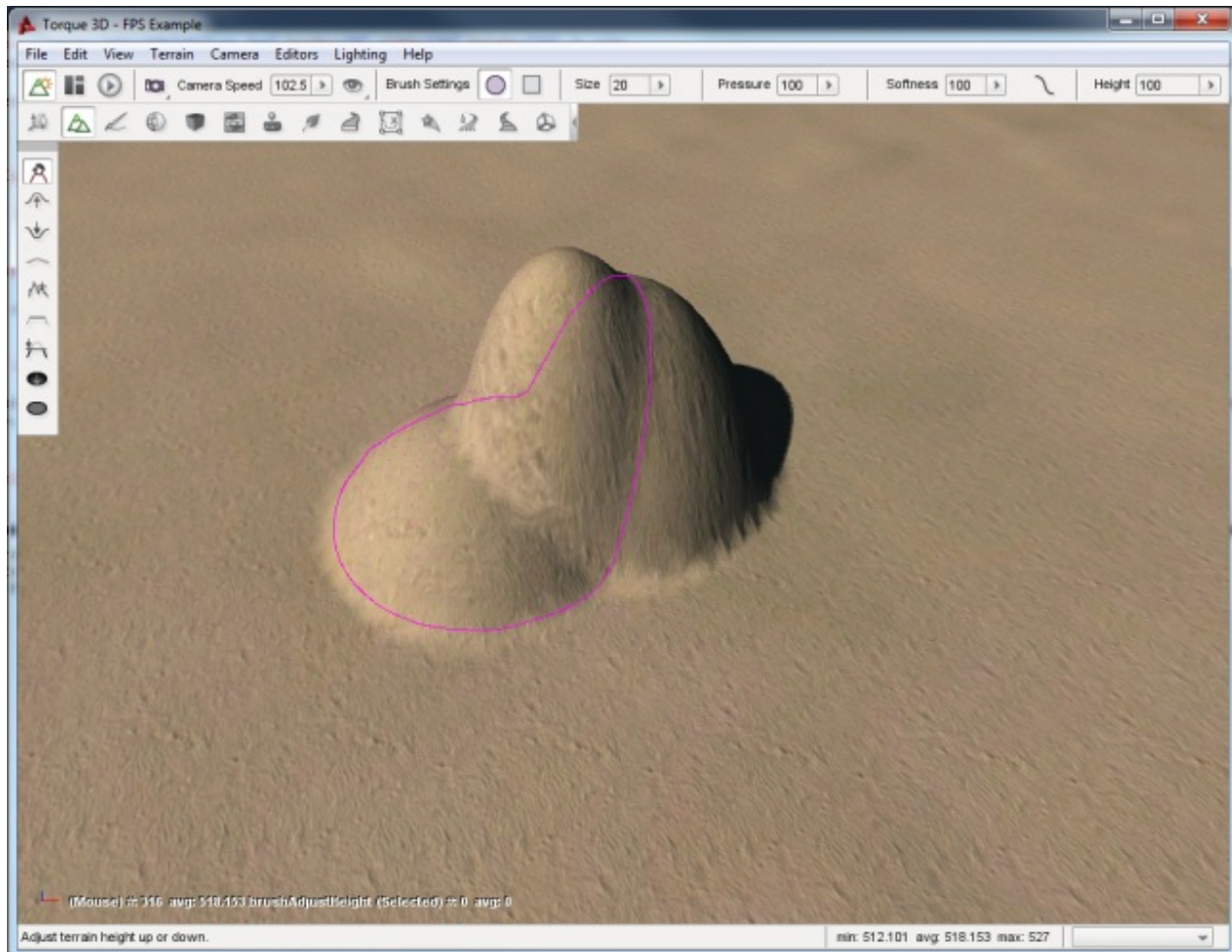


Use the mouse to cover part of your new adjustment with the brush. Notice how the brush clamps to the terrain, maintaining the shape you are using while still selecting a section.

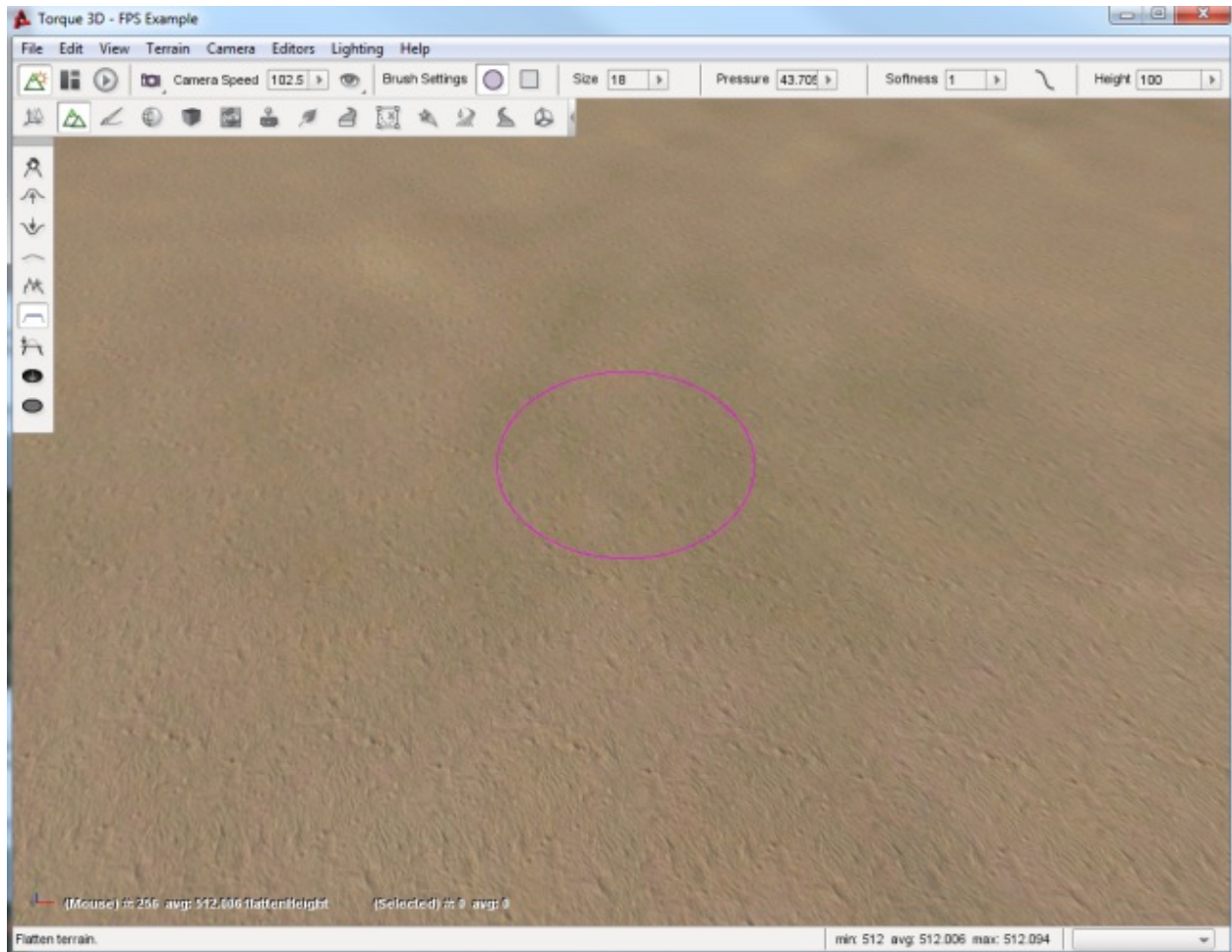


Despite the elevation of your current selection, the section under the hardest part of the brush will still adjust more dramatically. Using the default Softness Curve, if the center of the brush is just to the edge of a hill, you can adjust nearby terrain to match elevation. Terrain under the softer part of the brush will still elevate, but not as much.

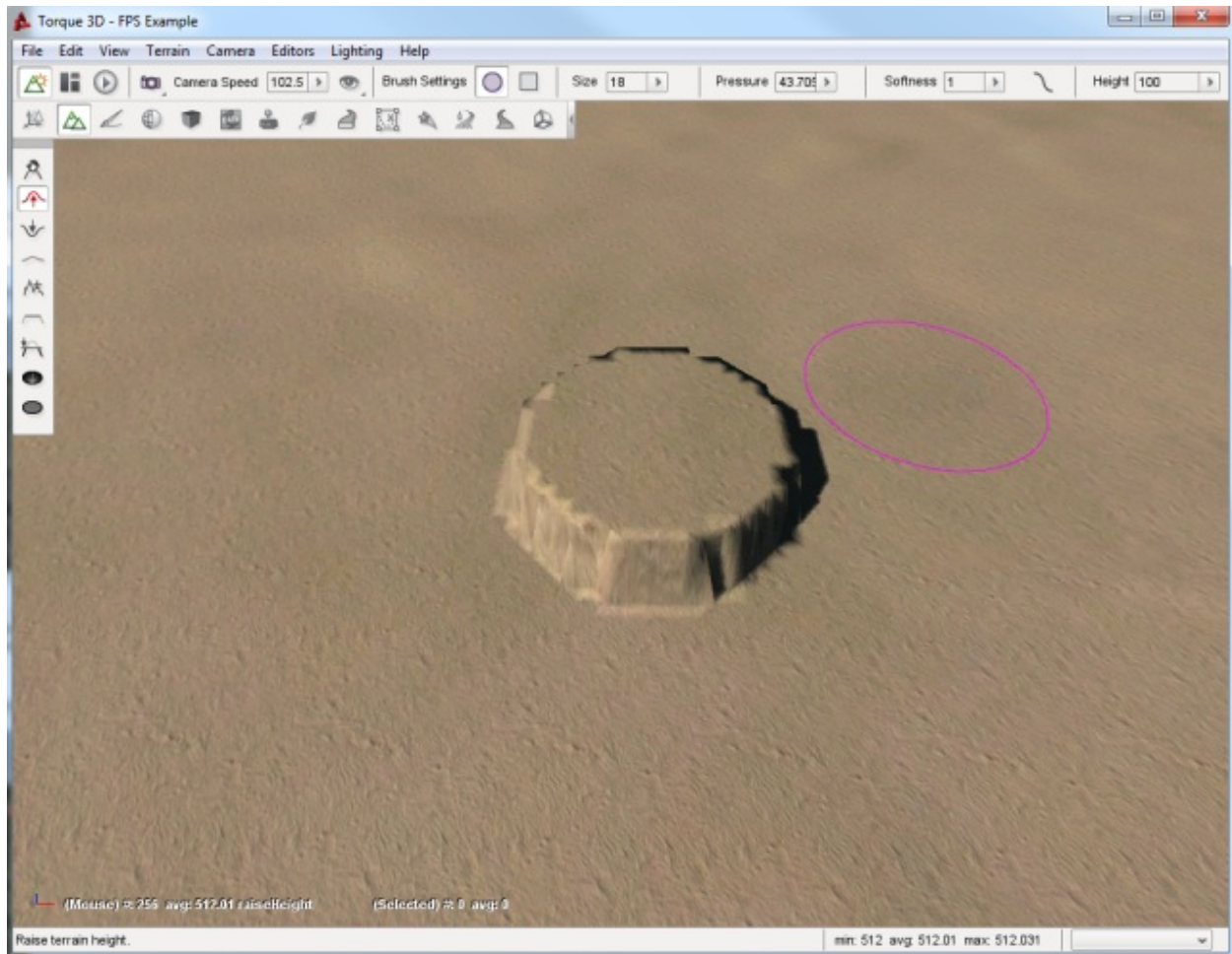




Before moving on to the next tool, we will experiment with the softness value. Set the softness of your current brush to 1 (very hard). Move the brush over a flat section of the terrain.



Click on the terrain and drag your mouse up. Instead of an elevated hill with a smooth slope, your brush should have created a flat plateau with completely vertical sides. With a softness of 1, your brush's shape will be used to extrude the terrain in a sharp manner.

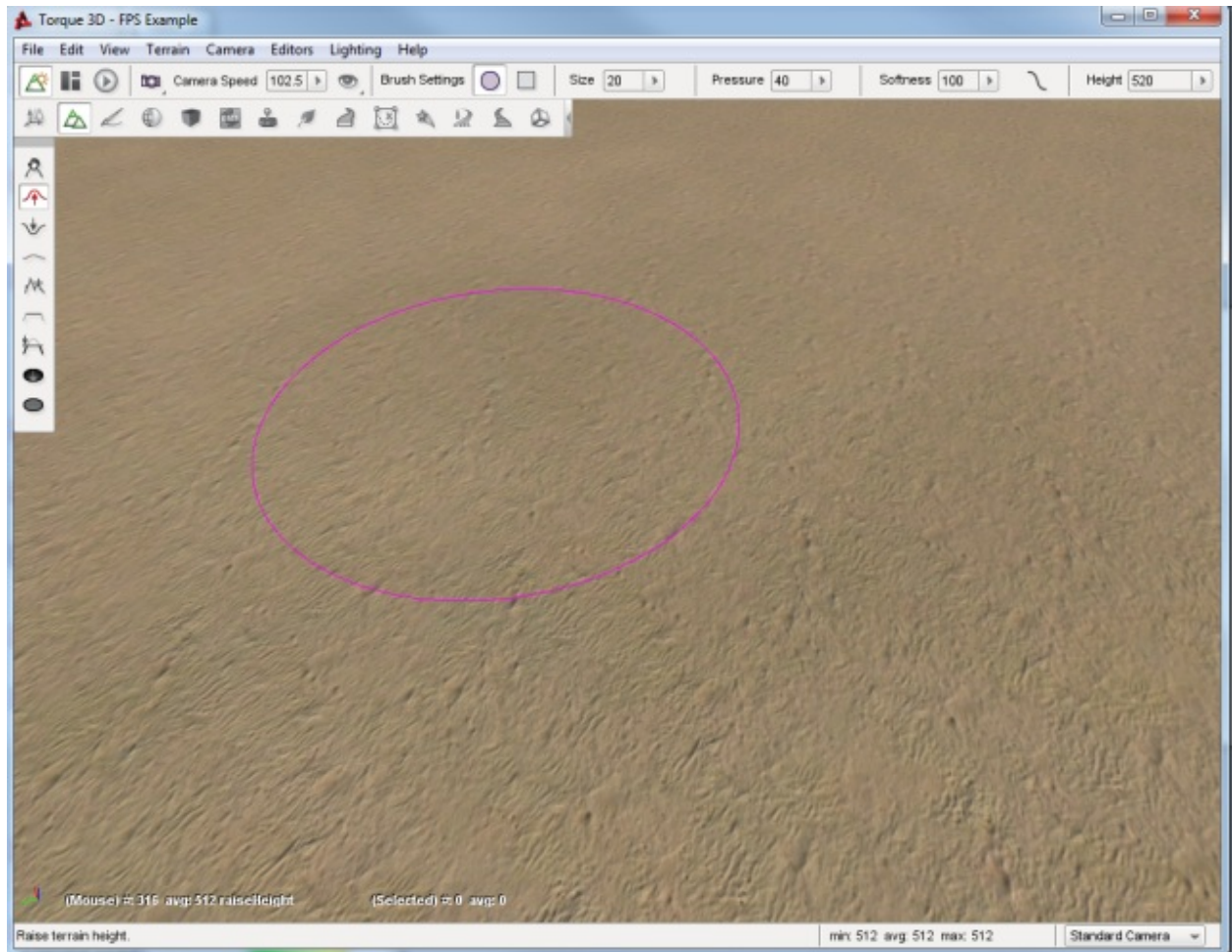


### 13.1.4 Raise Height Tool

The Raise Height tool can only elevate the terrain, but it does so in a very controlled manner. Instead of manually lifting, you can “paint” the terrain in a sweeping motion by dragging the mouse while holding the left button. The longer you keep the brush in one location, the higher that section will be and the higher the Pressure setting, which was reviewed earlier, the faster it will change.

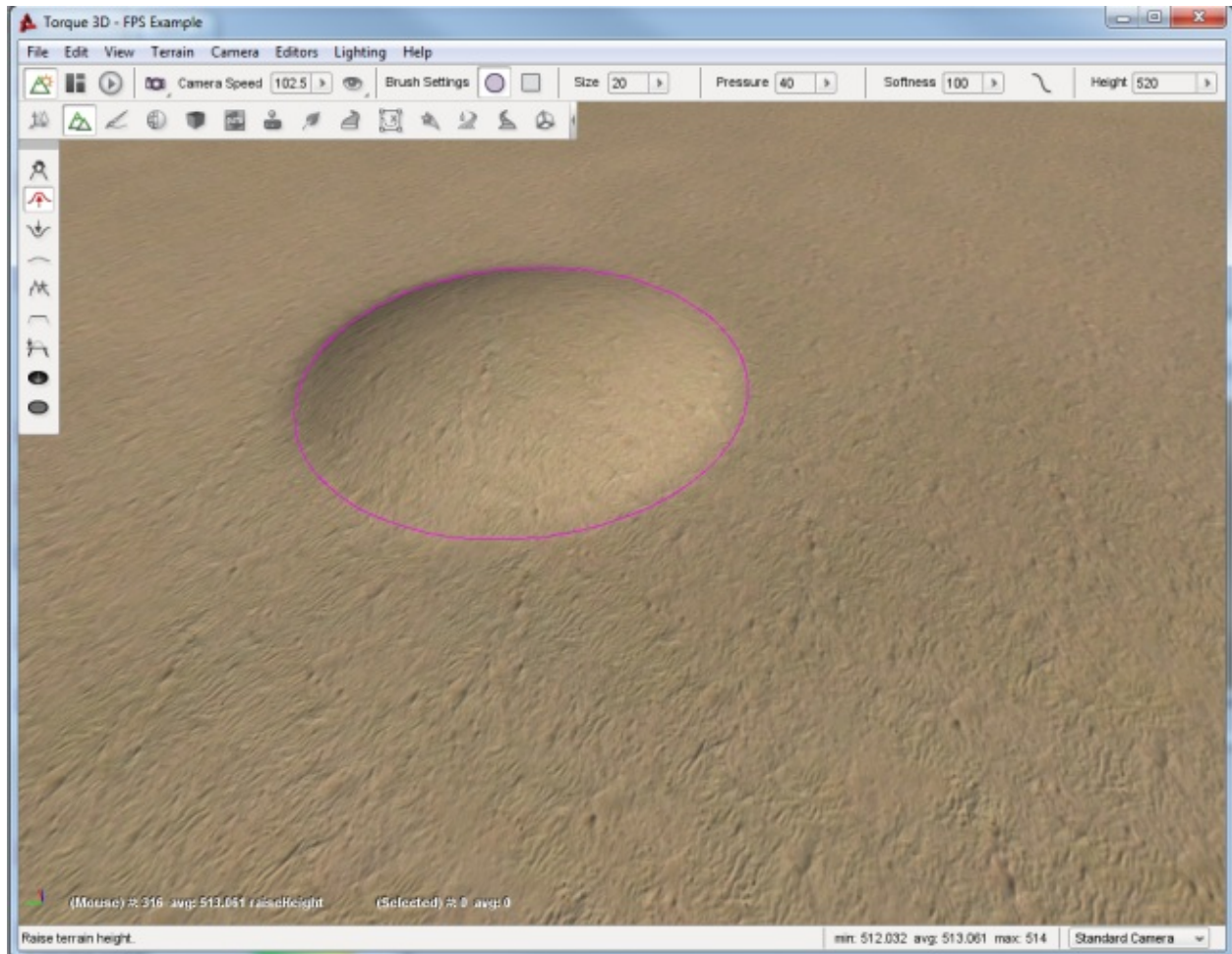
Set your brush size to 20, pressure to 40, and softness to 100. Find a flat section of the terrain and move your mouse cursor to that location to hover the brush.





When you are ready, click and hold the left mouse button and begin dragging your brush in a direction. The terrain should elevate wherever your brush passes over. You can use this to create a hill over a long section of terrain.

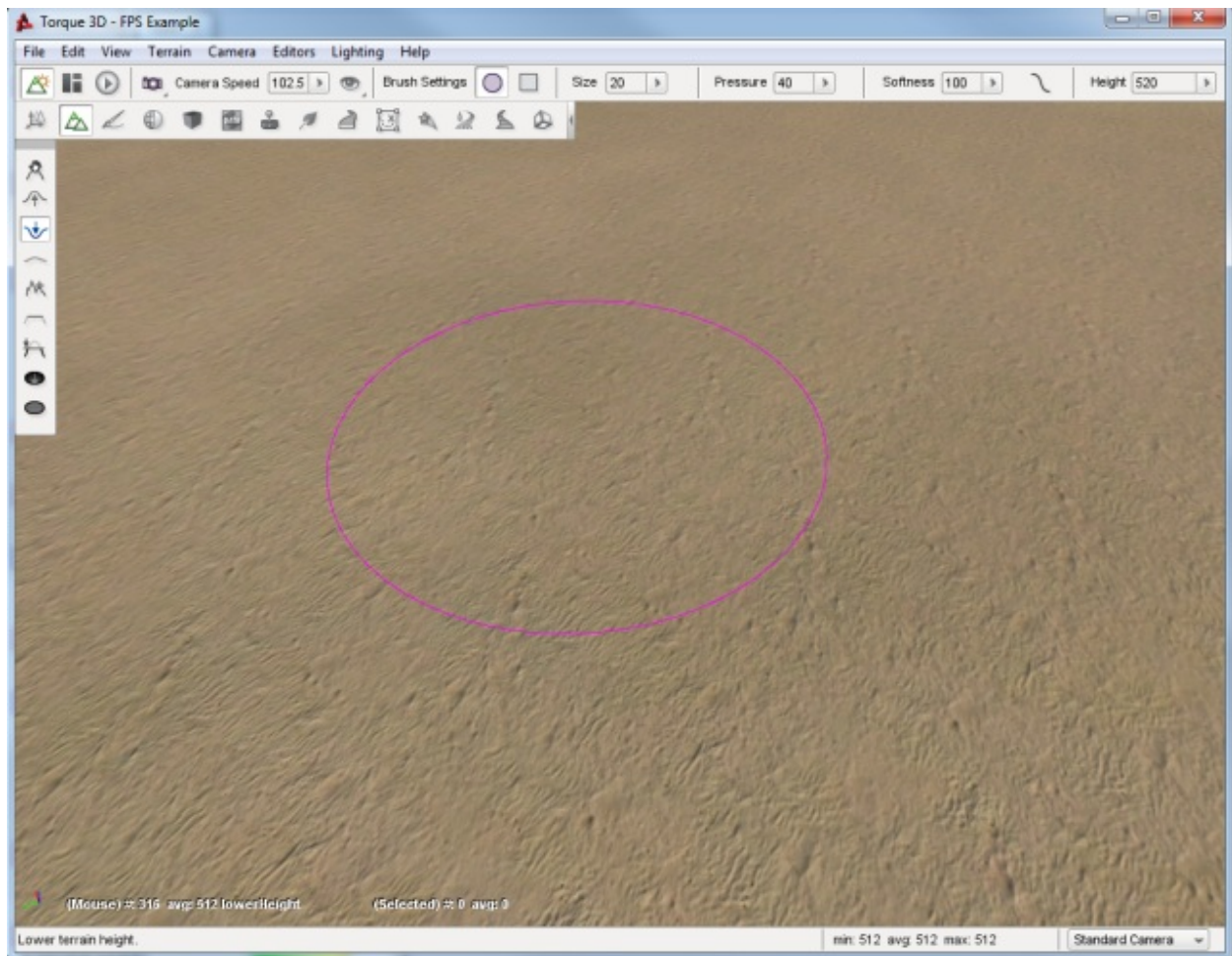




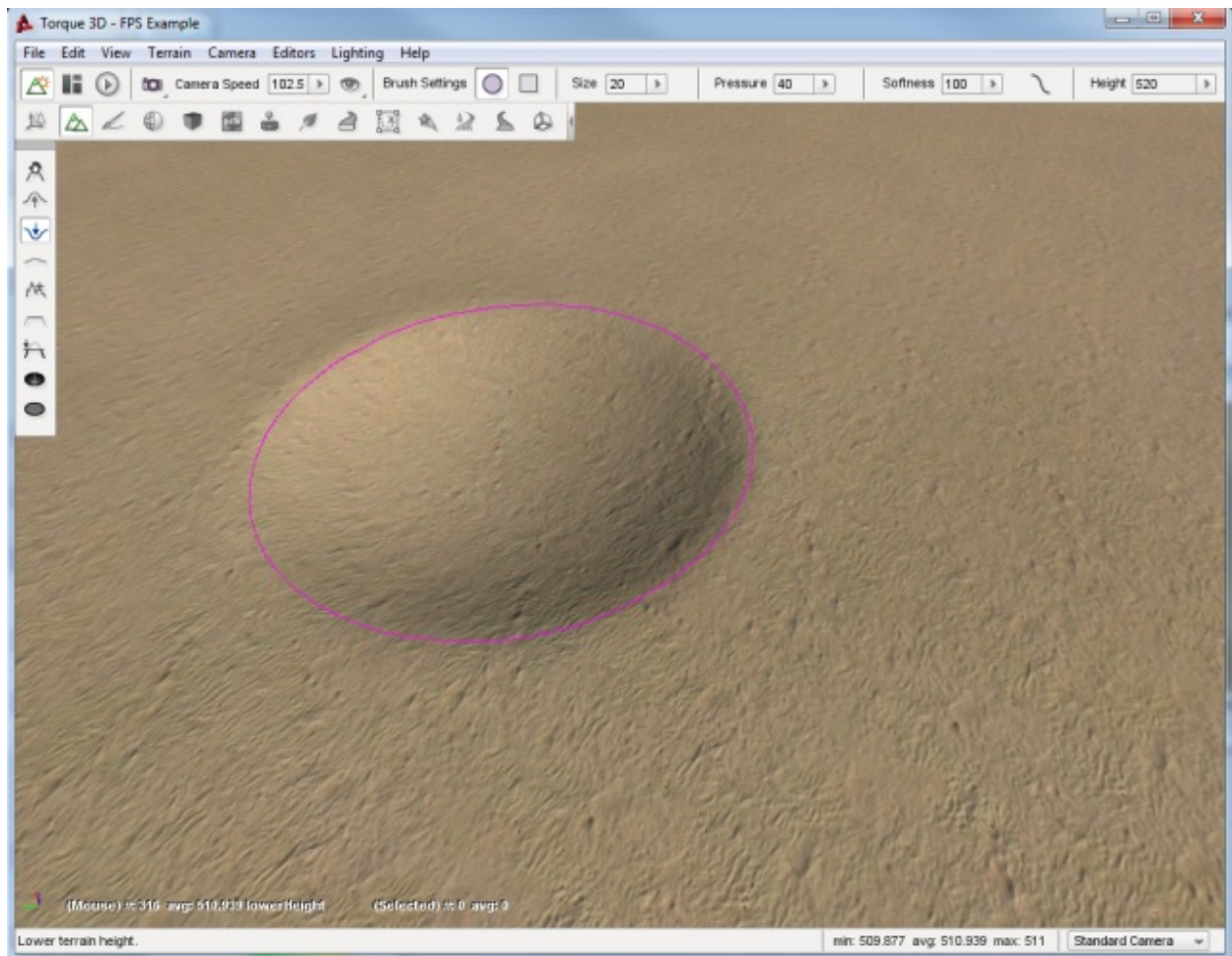
Using a lower brush pressure results in less dramatic terrain elevation as you “paint”. This allows you to be more exact in cases where you need to.

### 13.1.5 Lower Height Tool

The Lower Height tool functions completely opposite of the Raise Height tool. Instead of elevating, you can dig holes in the terrain with this tool. Again, use a circular brush with 20 size, 40 pressure, and 100 softness. With the Lower Height tool selected, locate a flat section of the terrain and hover your brush over it.



Click and hold down the left mouse button. As you do so, the terrain will sink down below the brush. If you sweep your mouse as if you are painting, you will create a path of lowered terrain. The longer you hold the mouse in a single location, the deeper the hole will be.

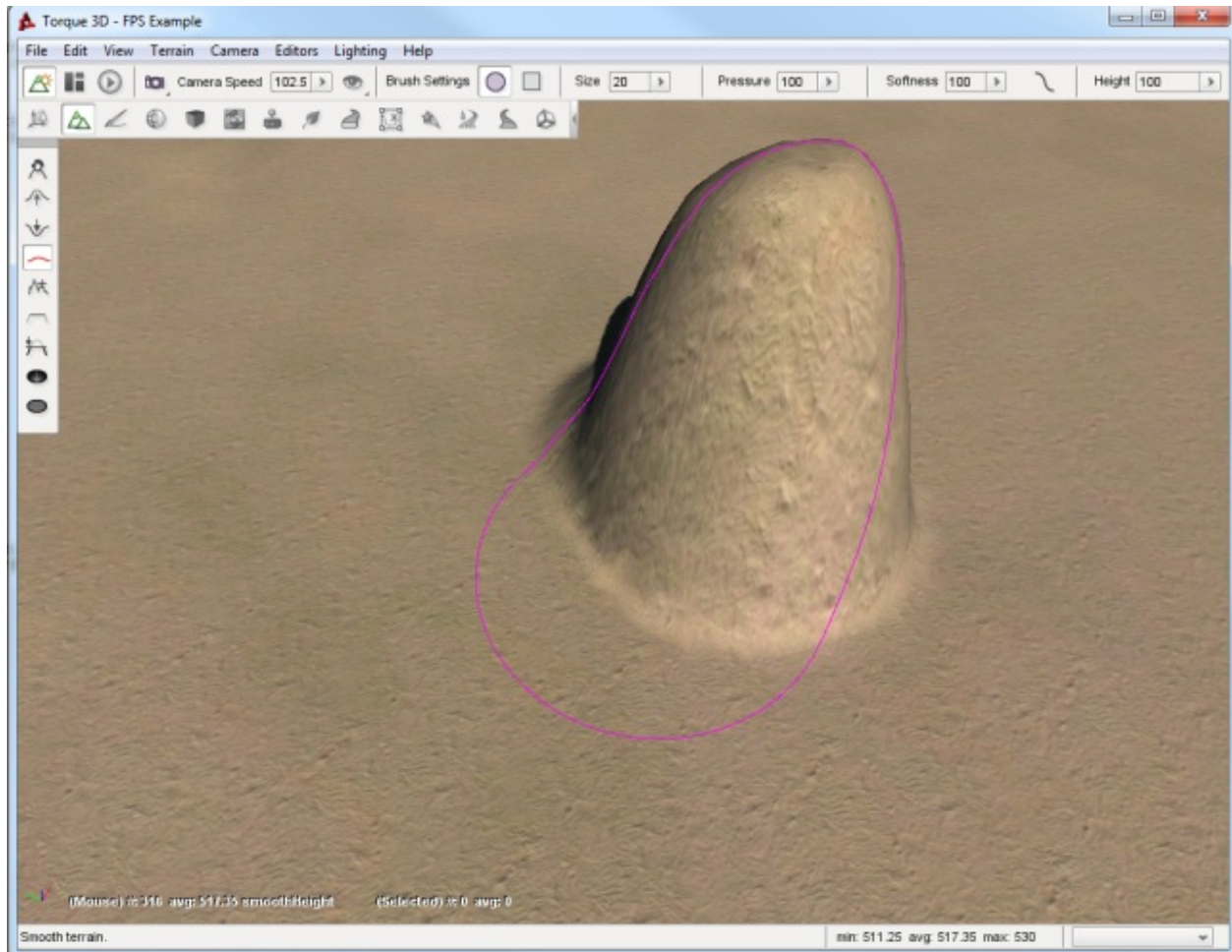


### 13.1.6 Smooth Tool

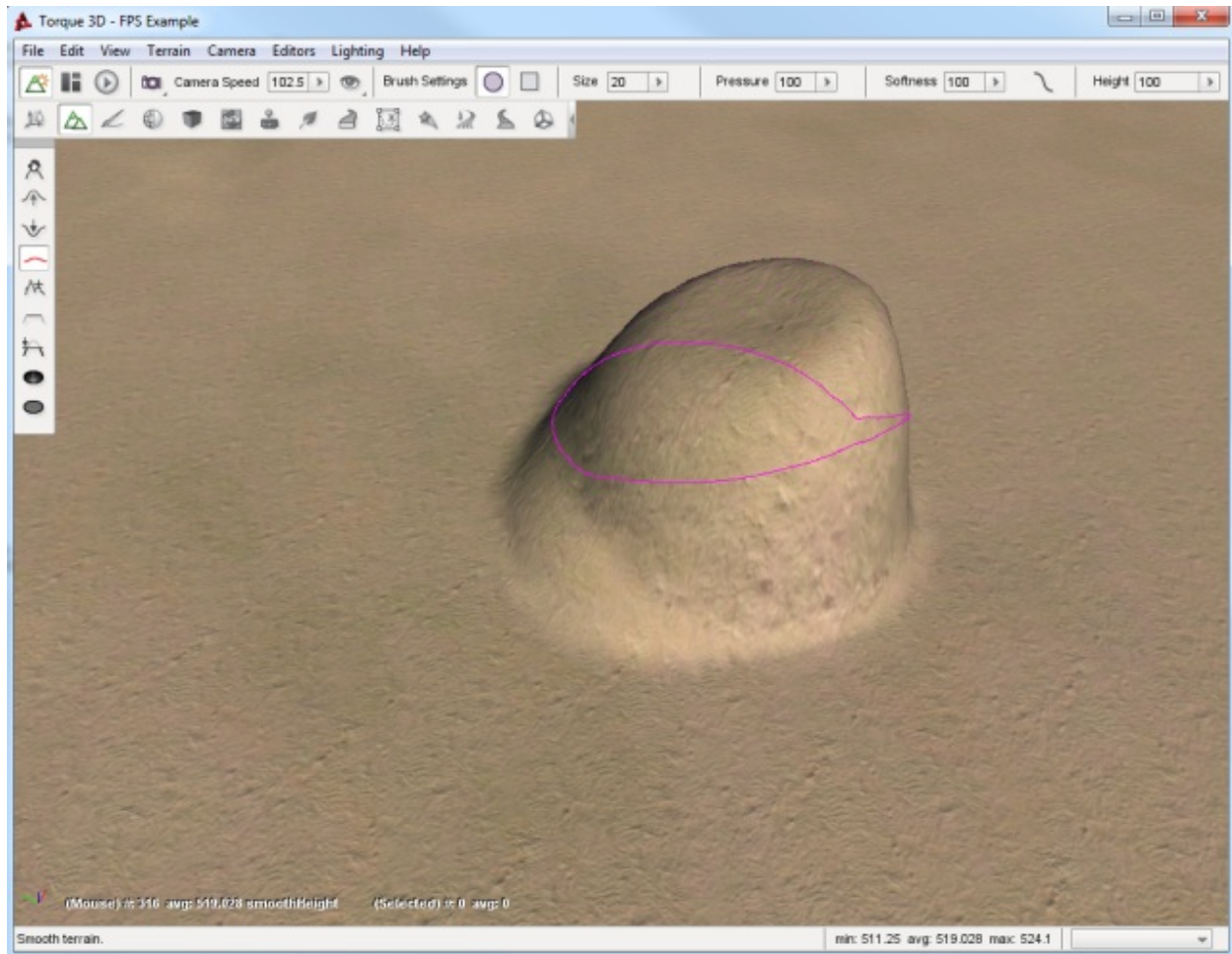
The Smooth tool erodes jagged terrain sections under the brush to create a smoother surface. This tool will only work if you sweep the brush across a surface. Simply holding down the left mouse button will have little to no effect.

Keeping the same settings we have been working, locate a jagged section of terrain. If you have to, create one with the Raise Height tool first. Make sure the elevation difference is significant. Select the Smooth tool then hover the brush over the applicable terrain.





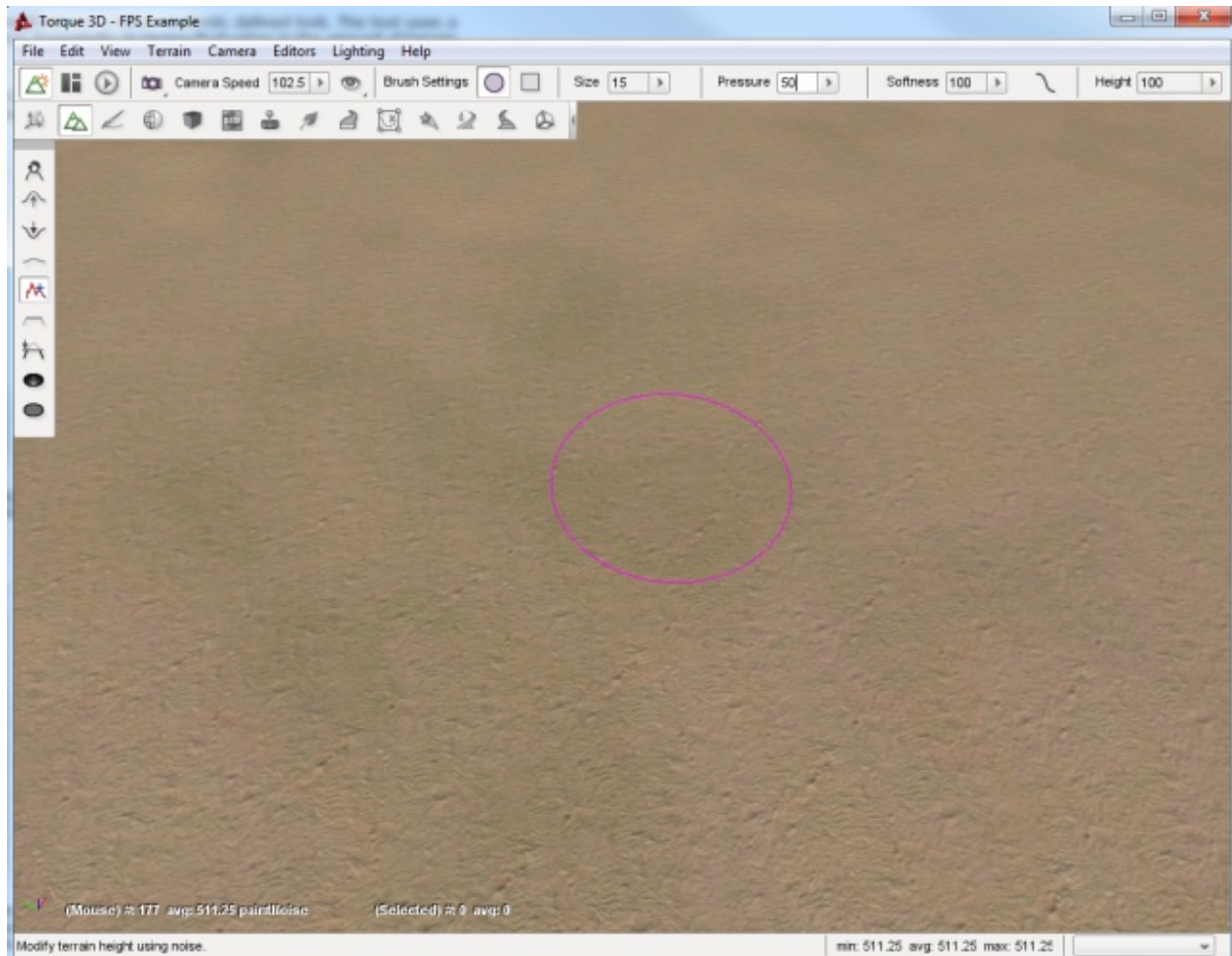
Click and hold the left mouse button, then make small circles around the peak of the terrain section. The tip should lower and have a broader surface. The broader your sweep, the more terrain is affected by the smoothing process.



### 13.1.7 Paint Noise Tool

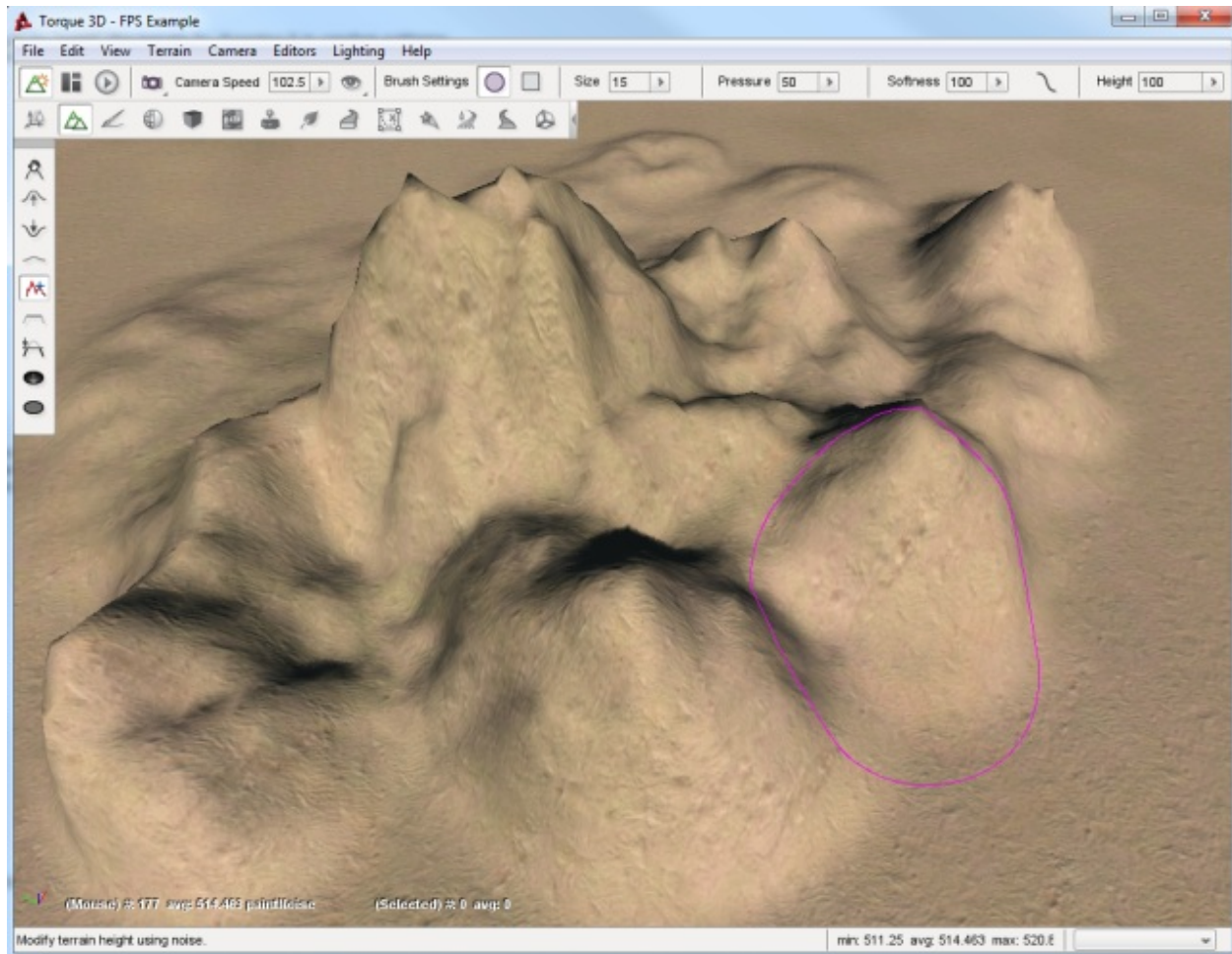
The Paint Noise tool is used to give your terrain modifications a more randomly defined look. The tool uses a noise algorithm for sporadic elevation and excavation. Essentially, it causes fluctuation in the amount of terrain it modifies and how intensely it changes.

Select the Paint Noise tool, then set your brush to size 15, 50 pressure, and 100 softness. Locate a large section of flat terrain and move your camera to a high elevation.

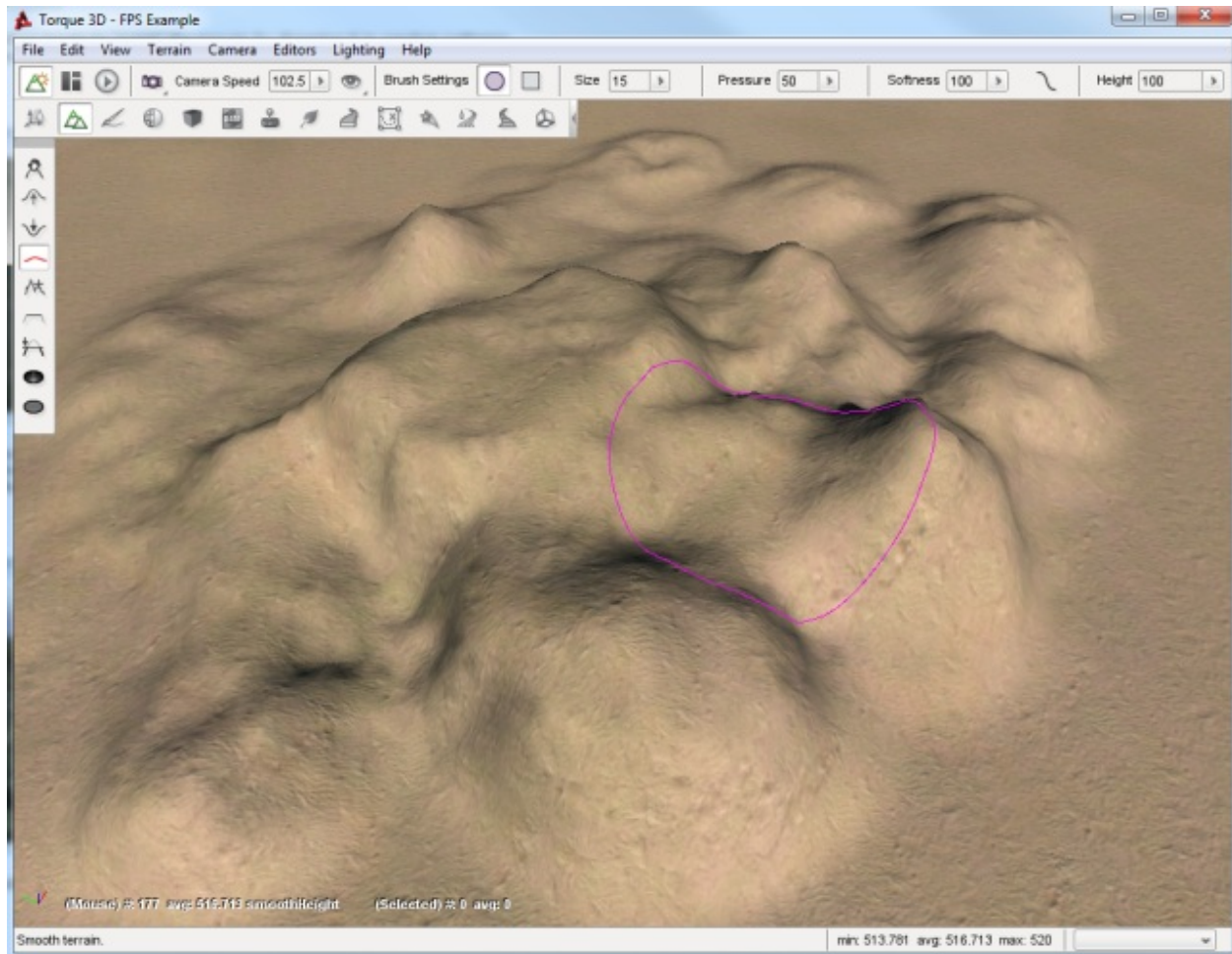


Click and hold the left mouse button down then begin to “paint” the terrain by dragging it in random patterns. Try making several concentric circles, varying spirals, zig-zag motions, etc. You should eventually see some definition forming.





When you are finished with the tool, fly your camera around the section of the terrain to see how the terrain was affected. Keep in mind that most of these changes were random, which can add much needed detail to your terrain but can cause some weird effects. The Smooth tool can be used to go back and blend out any such effects that do not look natural.



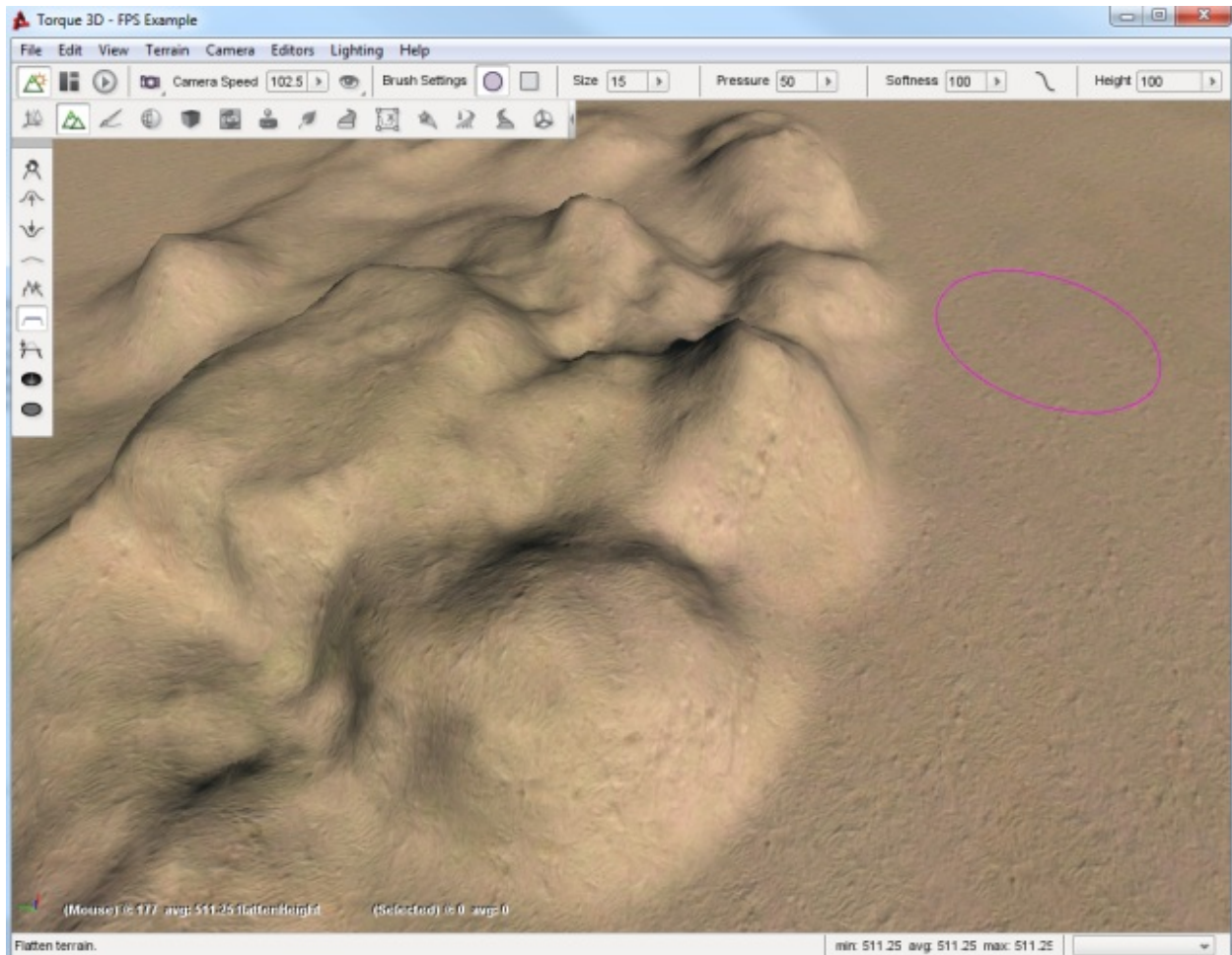
You can use this tool on terrain that has already been modified to remove unrealistic adjustments, such as perfectly smooth or flat slopes.

### 13.1.8 Flatten Tool

The Flatten tool is used to make the terrain surrounding the brush's starting point be equal to that point's elevation. In other words, this either lowers or raises your terrain to the same elevation as that starting point.

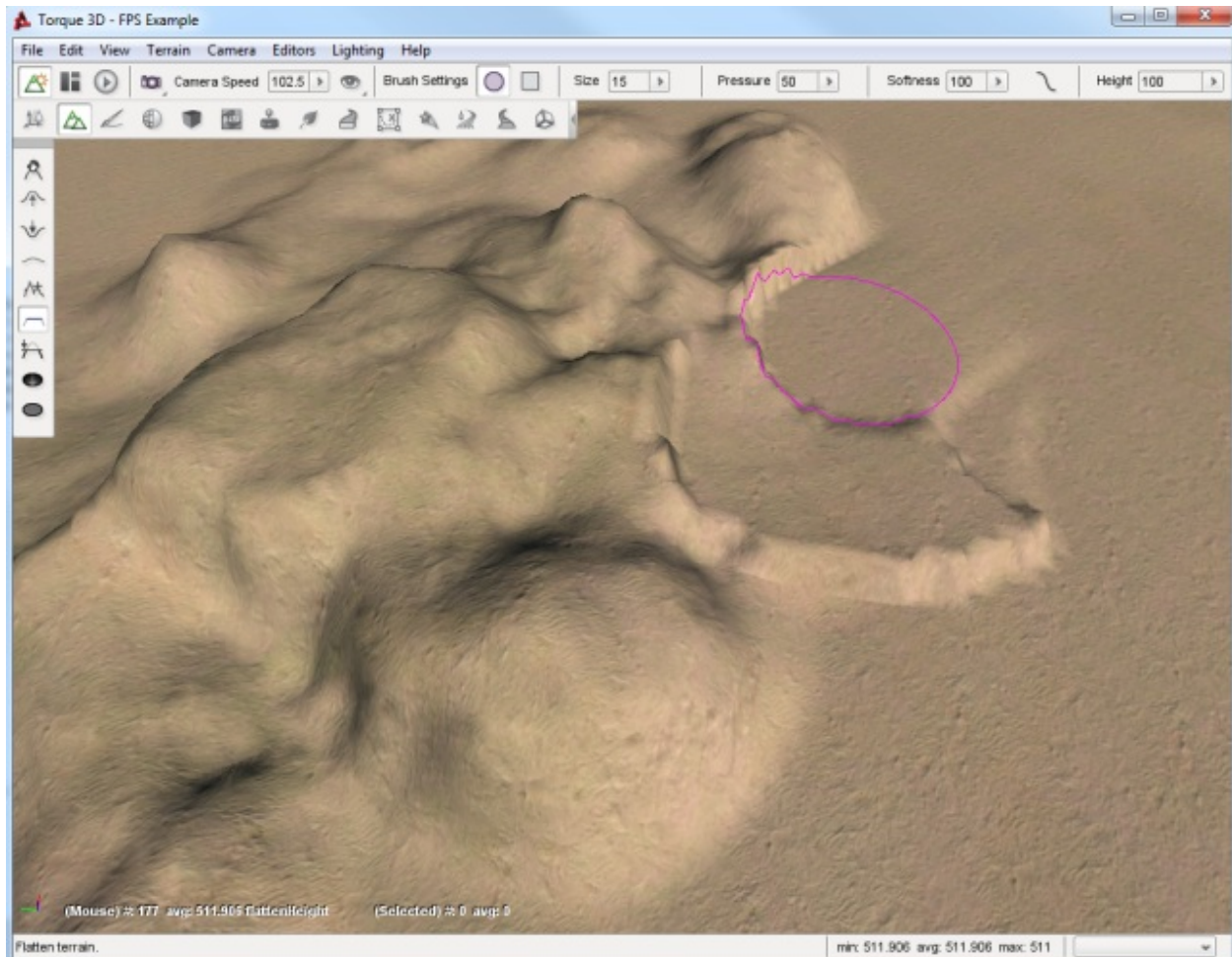
Use a circular brush with 15 size, 50 pressure, and 100 softness. Find a section of terrain that is elevated. Position your brush near it, but on a flatter section of the terrain.





Click and hold your left mouse button, then drag it toward and over the elevated terrain until you have swept over most of it. You should see that the tool has flattened a strip of terrain, based on the brush's location as it swept. The flattening process will become weaker the further you take the brush into the higher terrain such that it will not cut a path that is exactly the elevation of the starting point but rather relative to it and the terrain you are crossing. If you sweep the Flatten tool slowly across a hilly terrain, you will see that it is well suited for specialized tasks such as creating road and rail beds or mountain passes. Creating these types of features can be accomplished using the other tools but this tool in particular makes that job much easier.

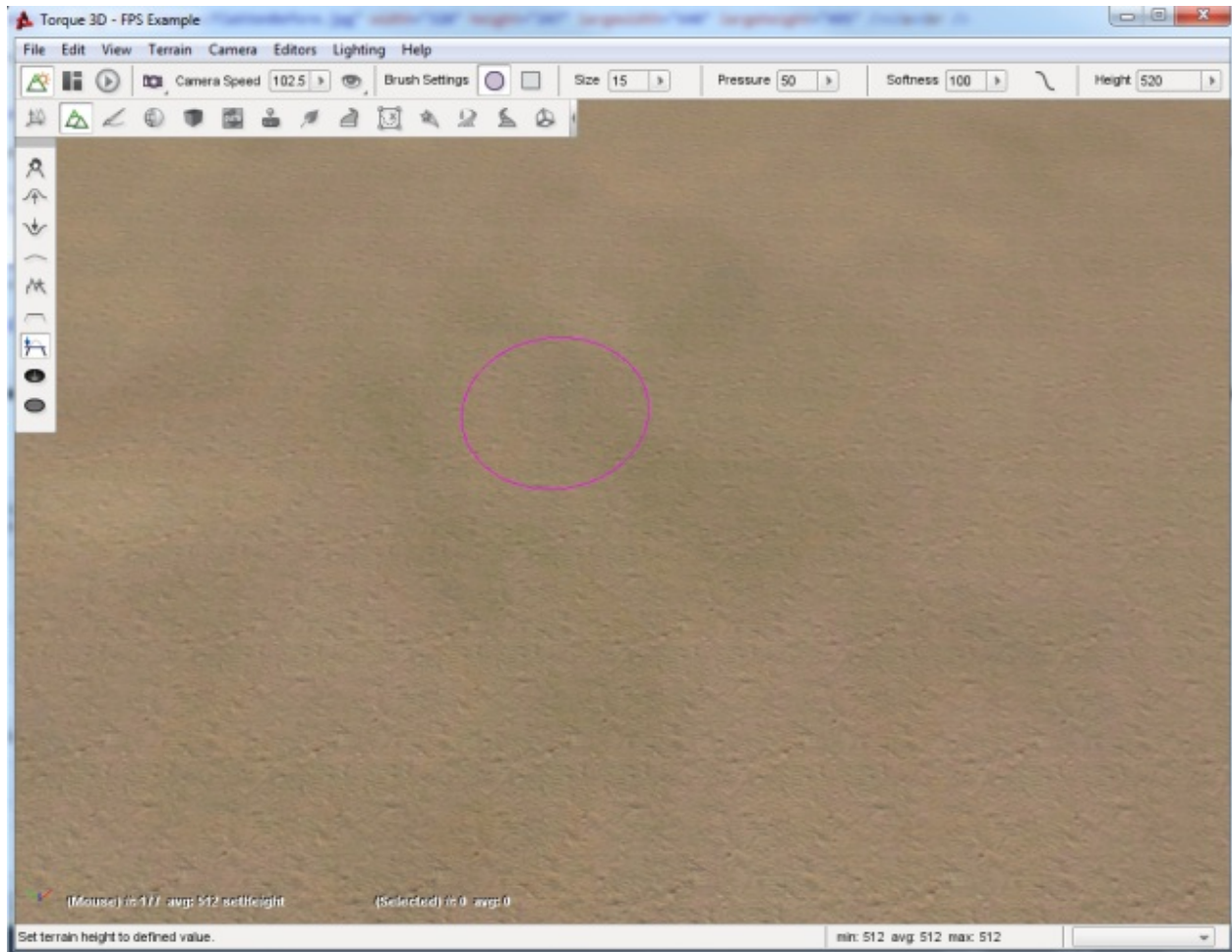
If you make several sweeps in the same direction, from the same starting point, your terrain will eventually smooth out into a flat plateau almost level with your original starting point.



This is generally handy for clearing a smooth path from one elevation to another. However, this is not the optimal approach for flattening huge sections of terrain. The other tools can perform that process much faster and more efficiently.

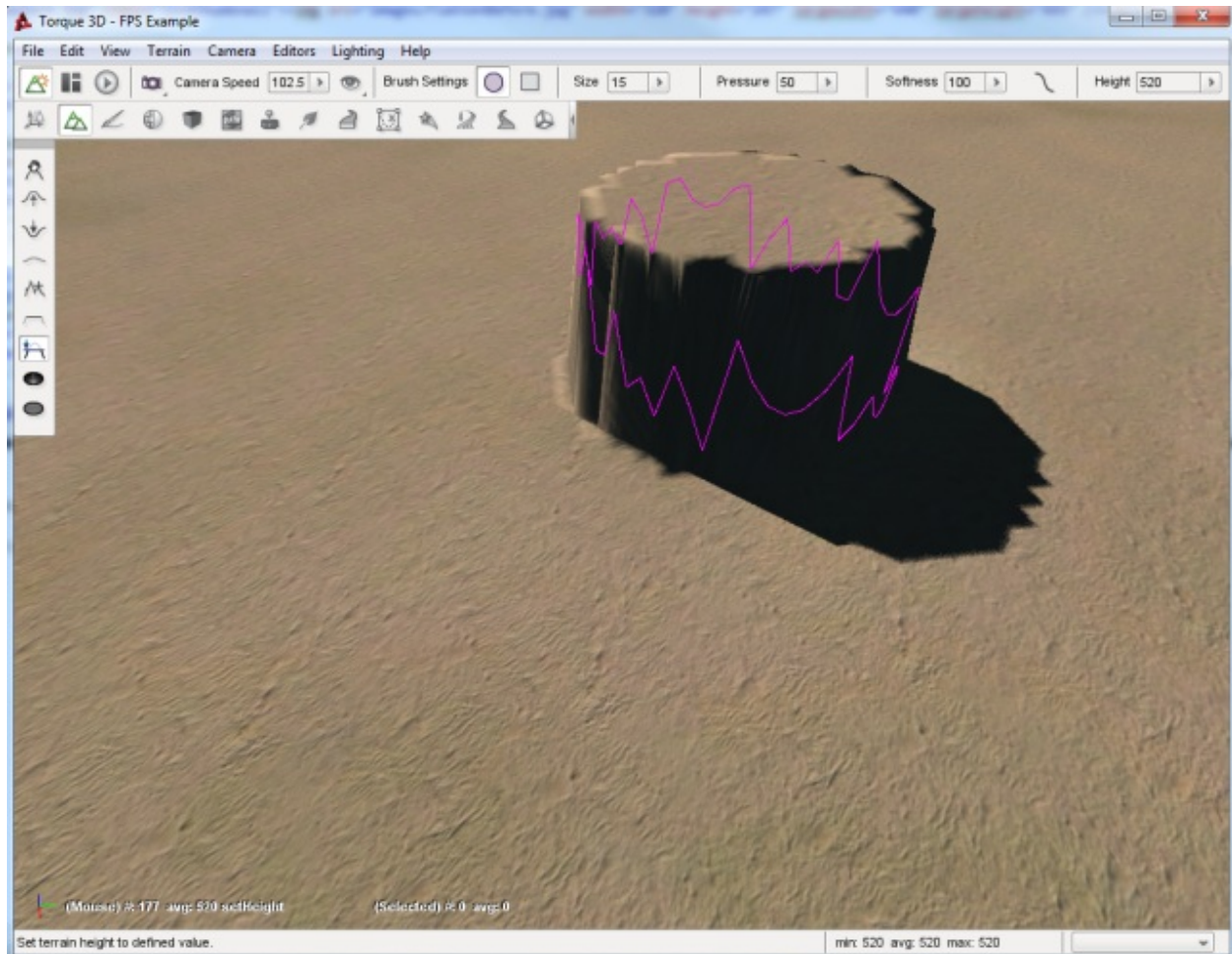
### 13.1.9 Set Height Tool

The Set Height tool will allow you to determine the exact height for the terrain brush. Use a circular brush with a size of 15, pressure of 50 and softness of 100, and a height of 520.



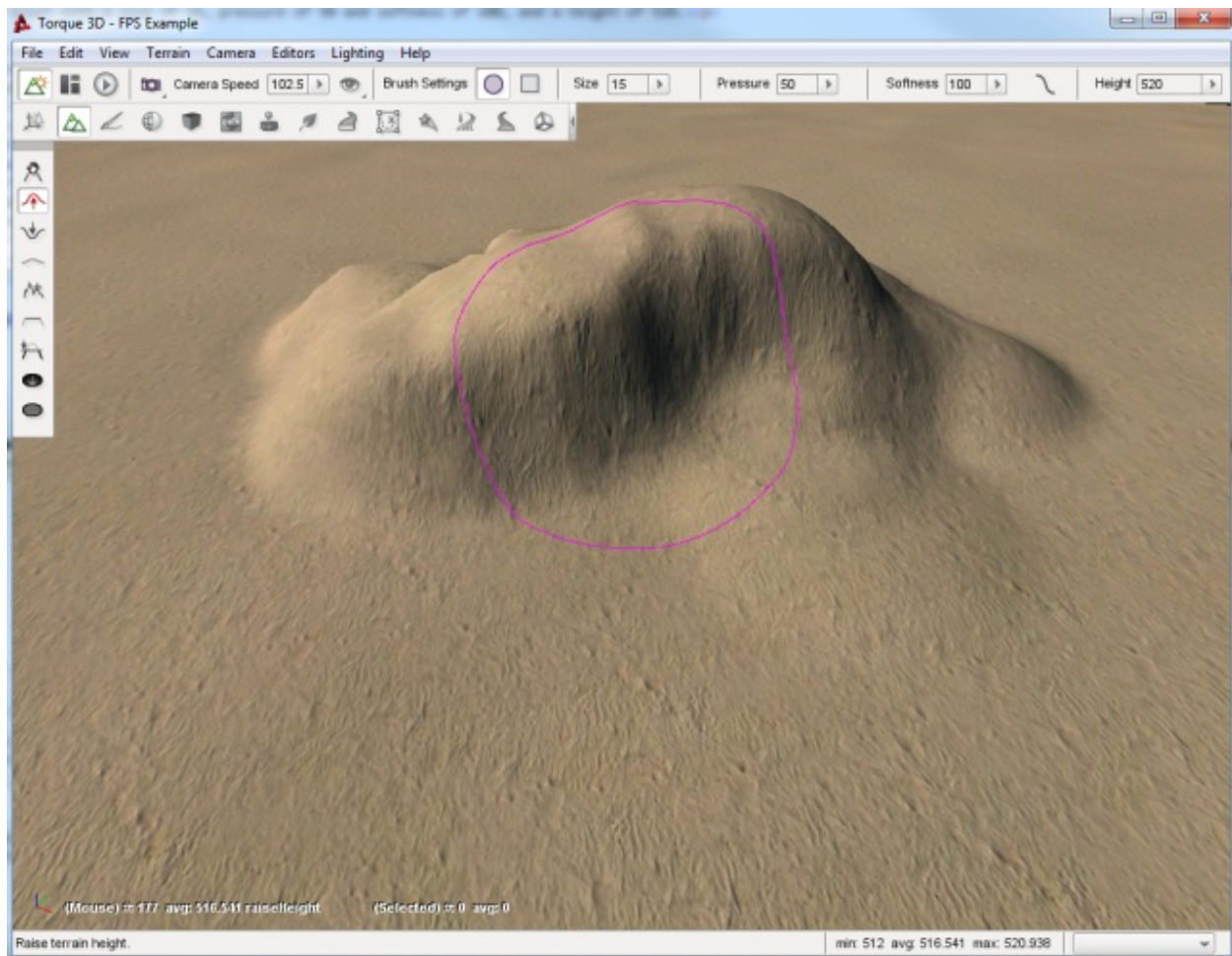
Now, when you press the left mouse button, it will create a plateau at exactly that height.



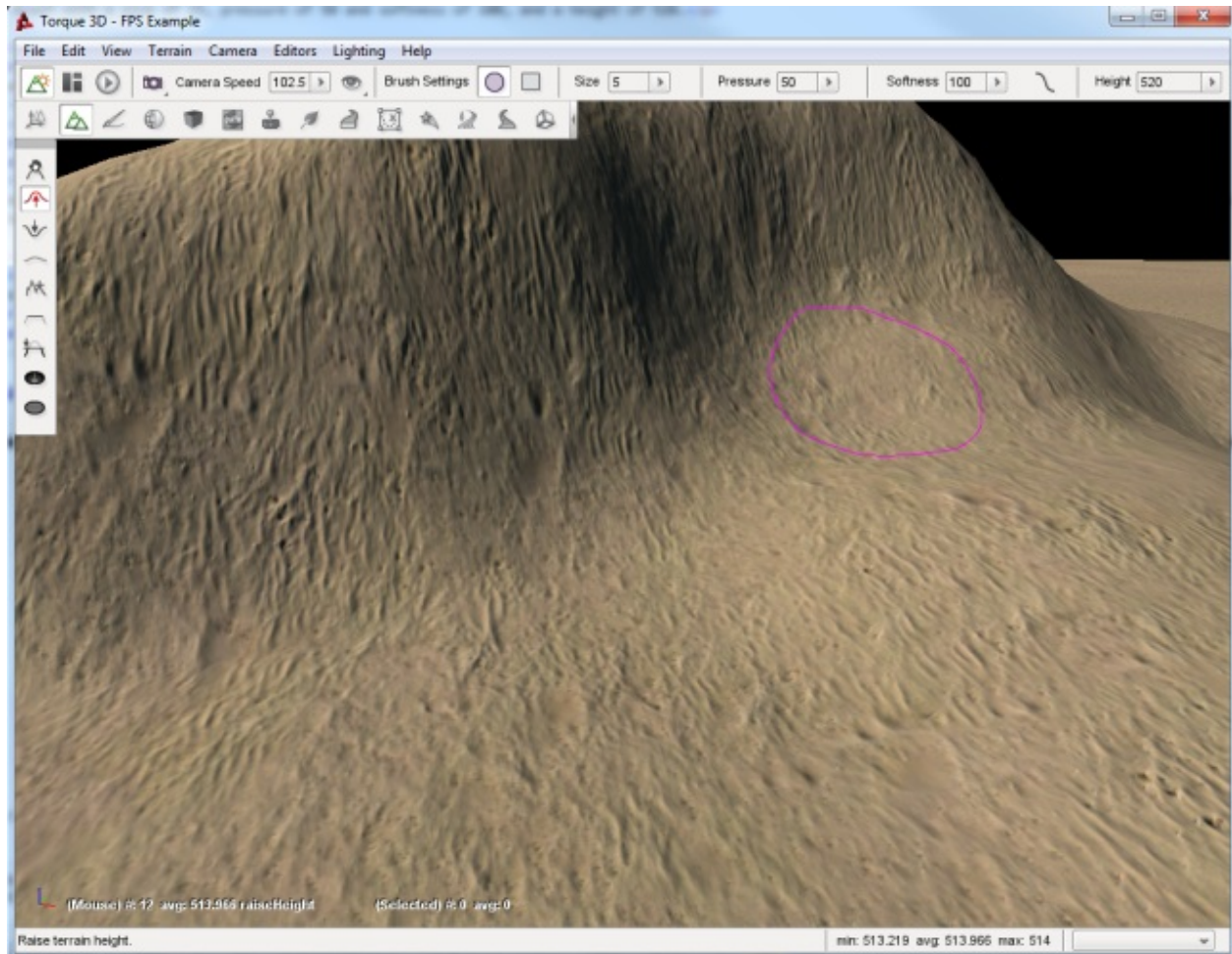


### 13.1.10 Clear Terrain Tool

The Clear Terrain tool will allow you to remove pieces of the terrain. This is an effective way to carve out entrances to caves. That way your artist can create a detailed cave level and your level editor can “carve” out an entrance in the terrain. Using the previous tools, create a small hillside.

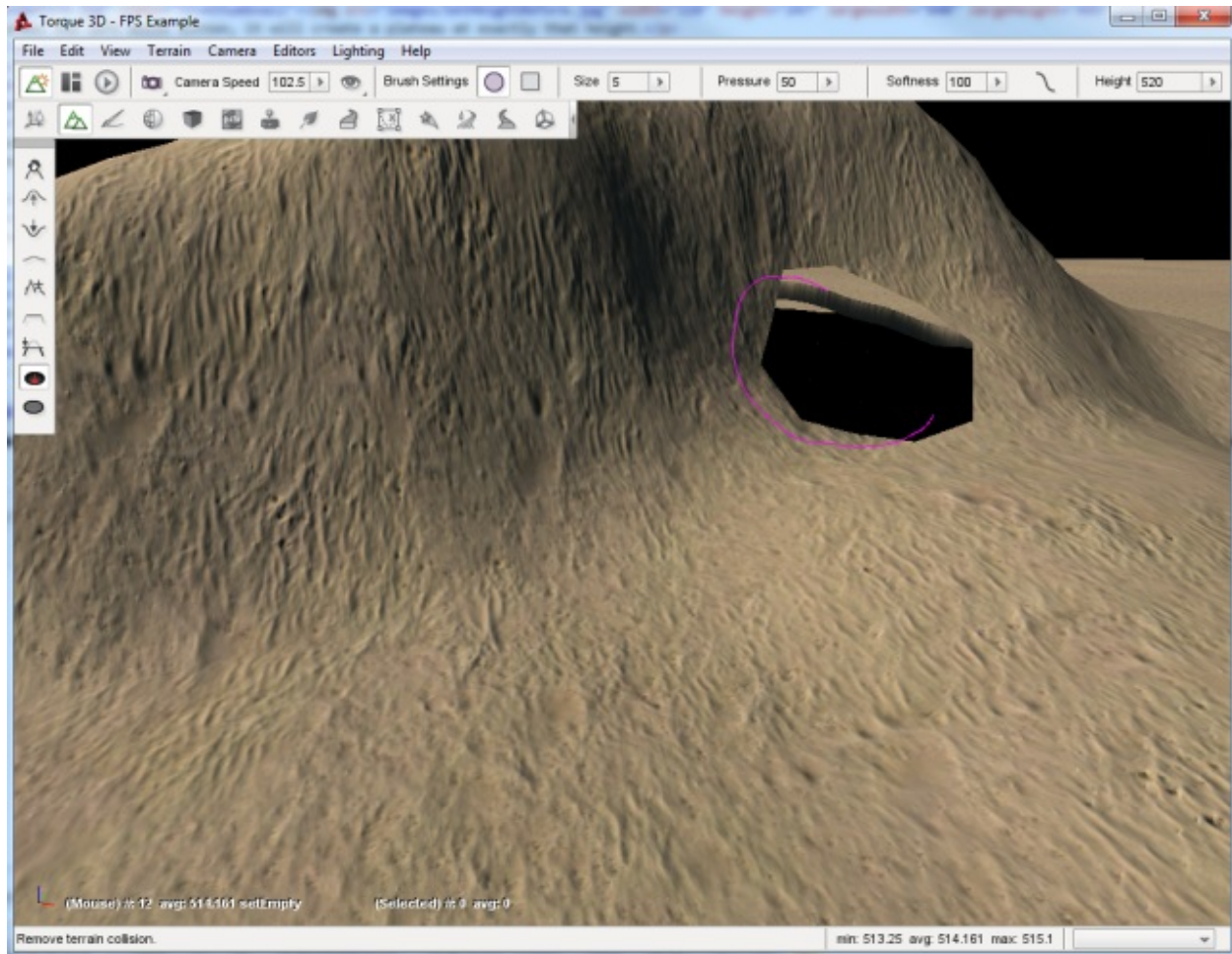


Now, set your brush size to 5 and zoom into an area that looks like a promising cave entrance.



When you select the terrain area, it will remove the mesh data from the terrain, creating an opening.

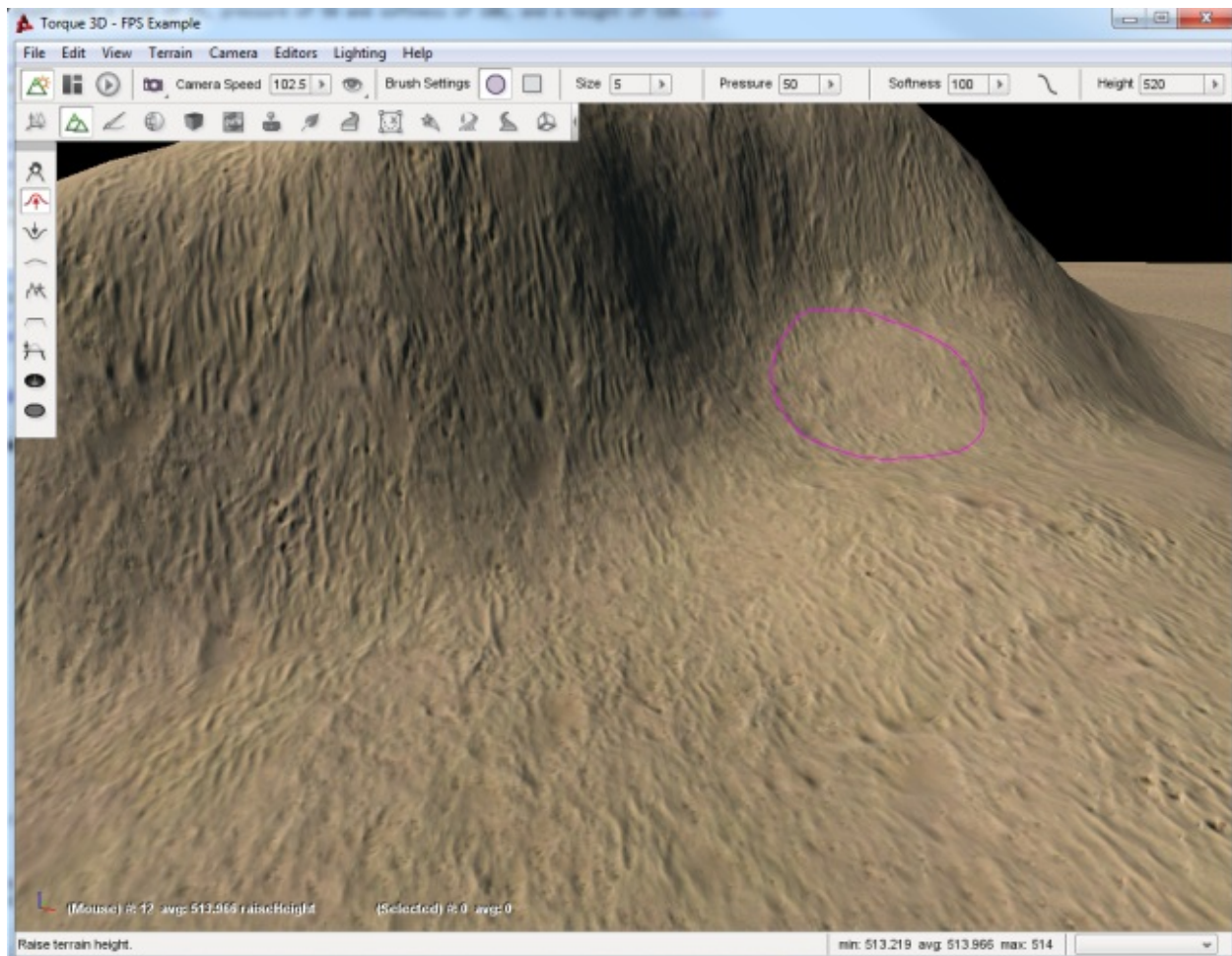




Now you can place your cave model underneath the terrain so that the player can explore the world under your terrain.

### 13.1.11 Restore Terrain Tool

The Restore Terrain tool complements the Clear Terrain tool. It will restore the mesh data for the terrain. That way, you can have better control over the transition between your models and terrains. If you select the Restore Terrain button and then left-click on the previously cleared area, you will see it restore the terrain to its previous state.



## 13.2 Terrain Painter

Just as the name implies, the Terrain Painter is a tool built into Torque 3D's World Editor which allows you to paint your terrain with various materials, such as grass, dirt, rocks, and so on.

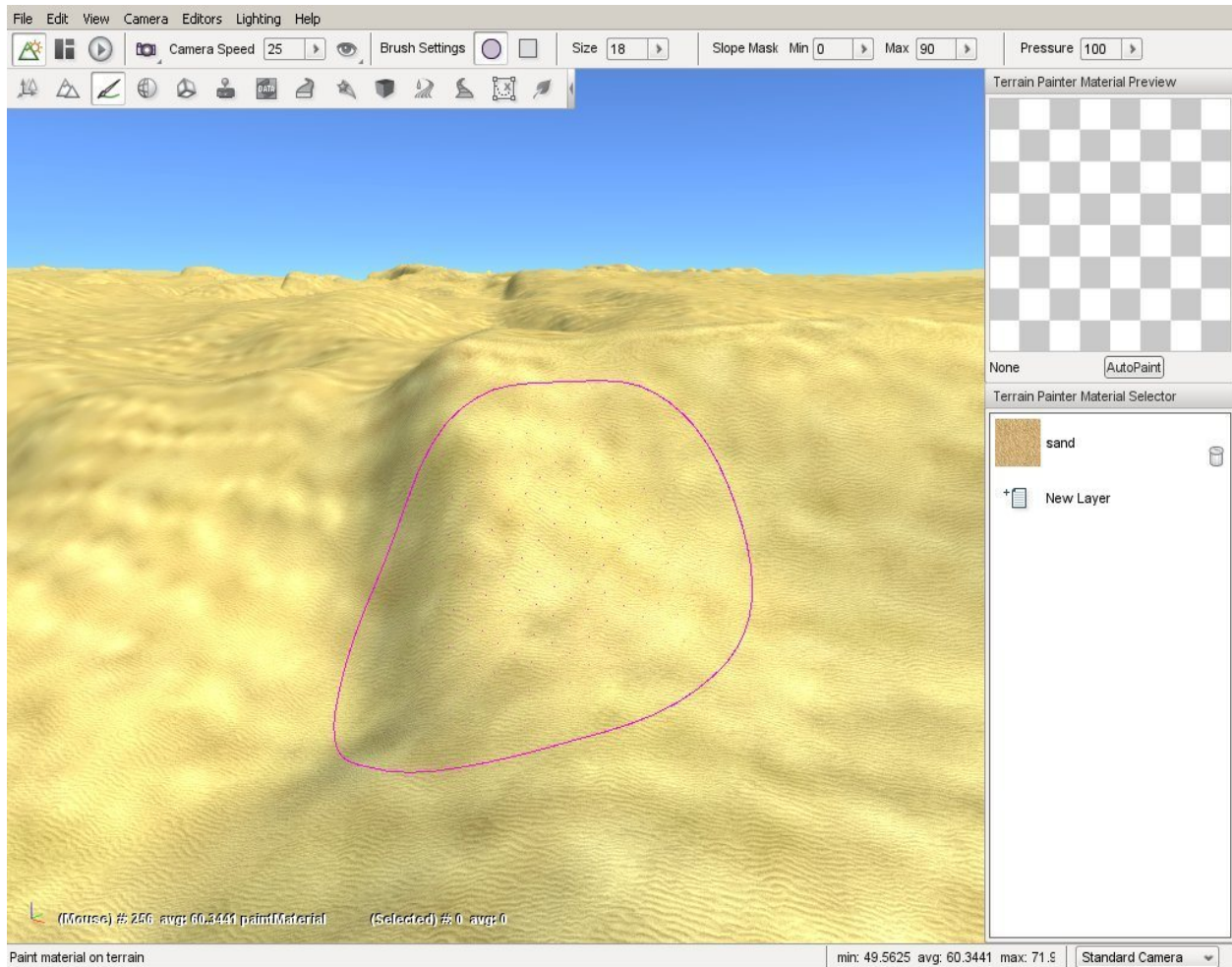
Like the rest of the editors, the Terrain Painter is a WYSIWYG editor. As you change your terrain materials and paint the surface, you can see what the changes will look like in real time as if you were playing your game.

You can use the Terrain Painter to make wide-spread modification to a blank terrain, or use finer and more detailed brushes to touch up imported terrain layers/textures. Let's get started by setting up your environment.

### 13.2.1 Interface

To switch to the Terrain Editor press the F3 key or from the main menu select Editors > Terrain Painter.

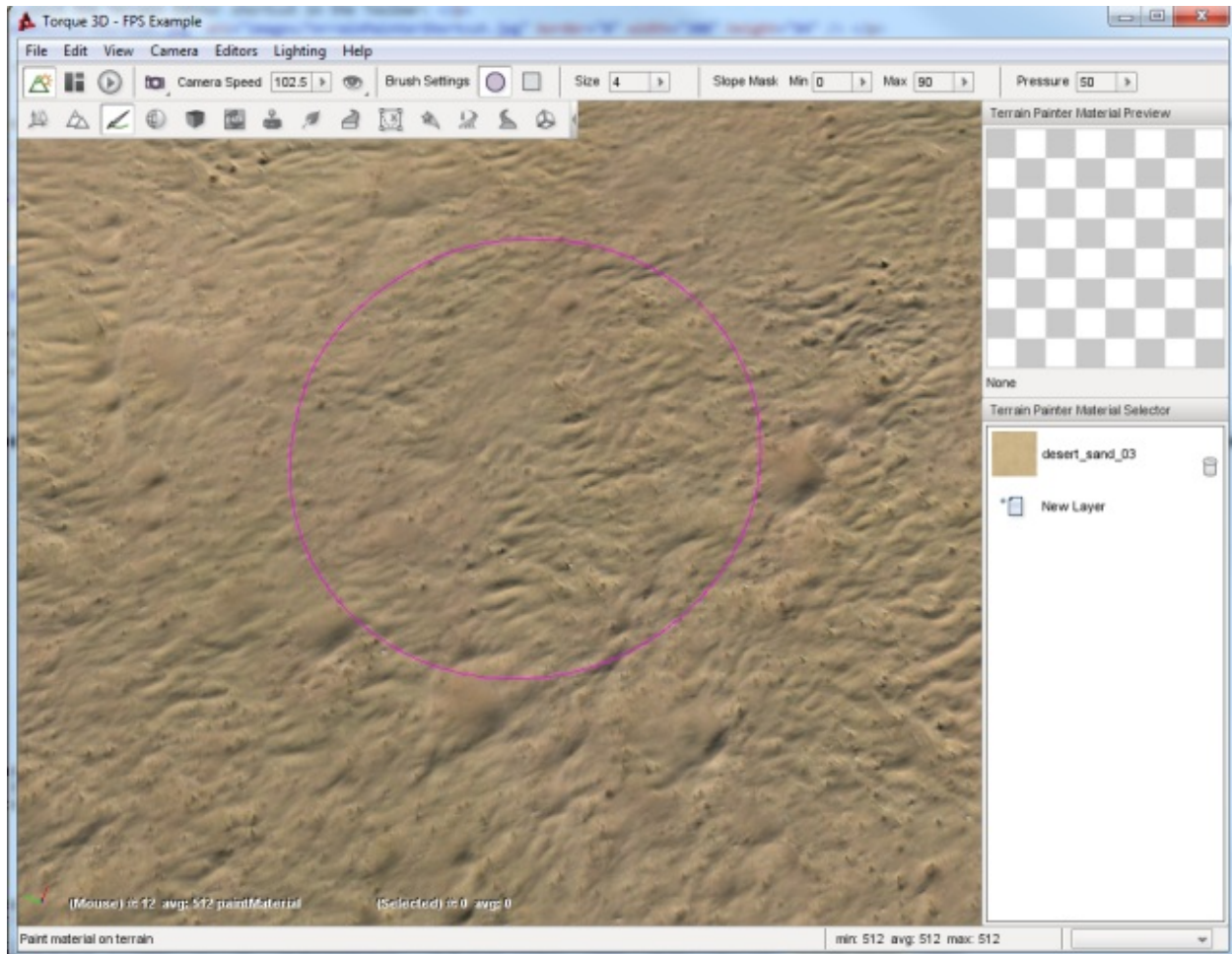




There are four main areas of the interface you will focus on while using this tool.

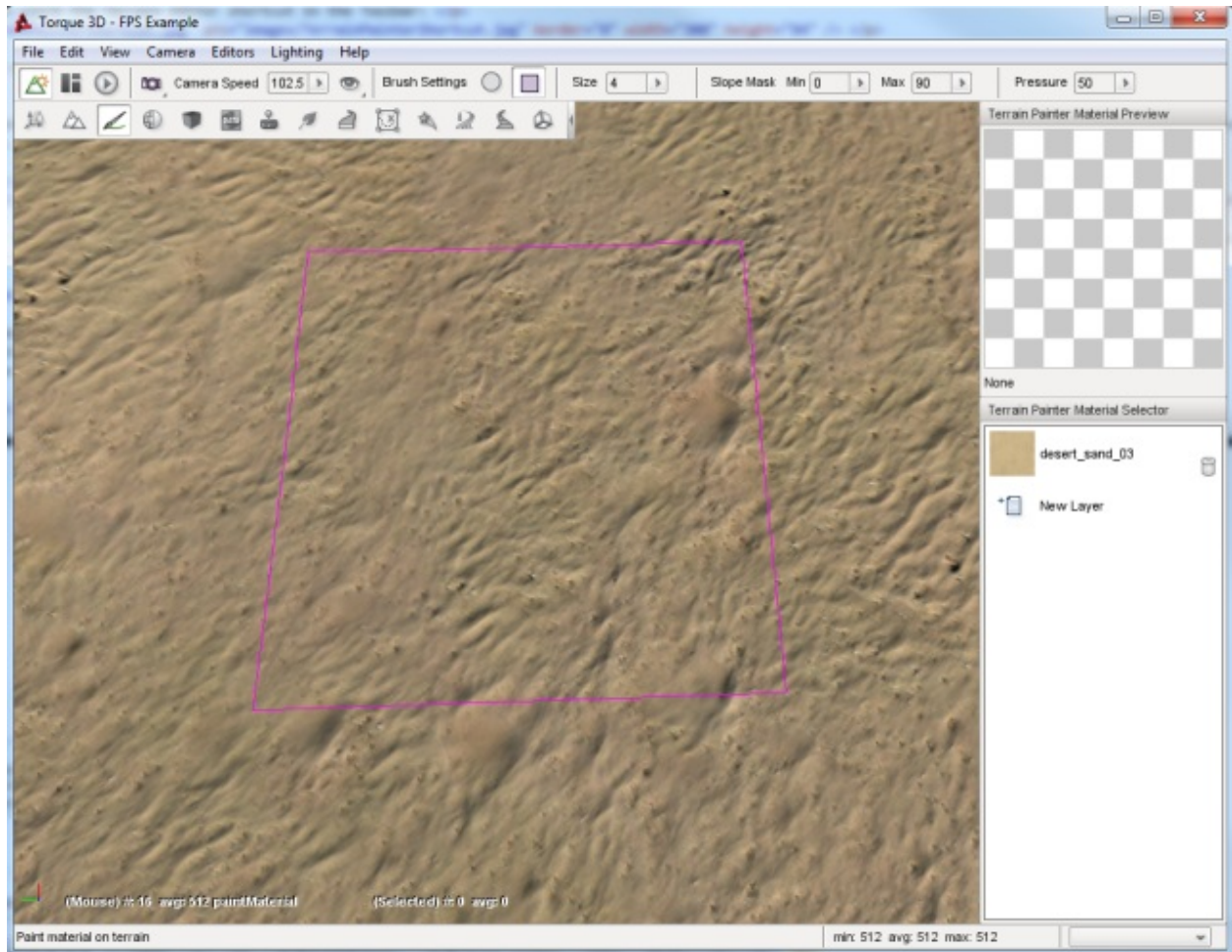
### 13.2.2 The Brush

Using the Terrain Painter is very similar to painting on a piece of paper with a brush except here you are painting on the terrain by dragging the mouse across the screen. Your brush is represented as a circle or a square in your scene's view. This visual outline allows you to know where your brush is located and what portion of the terrain it will affect when you move it.



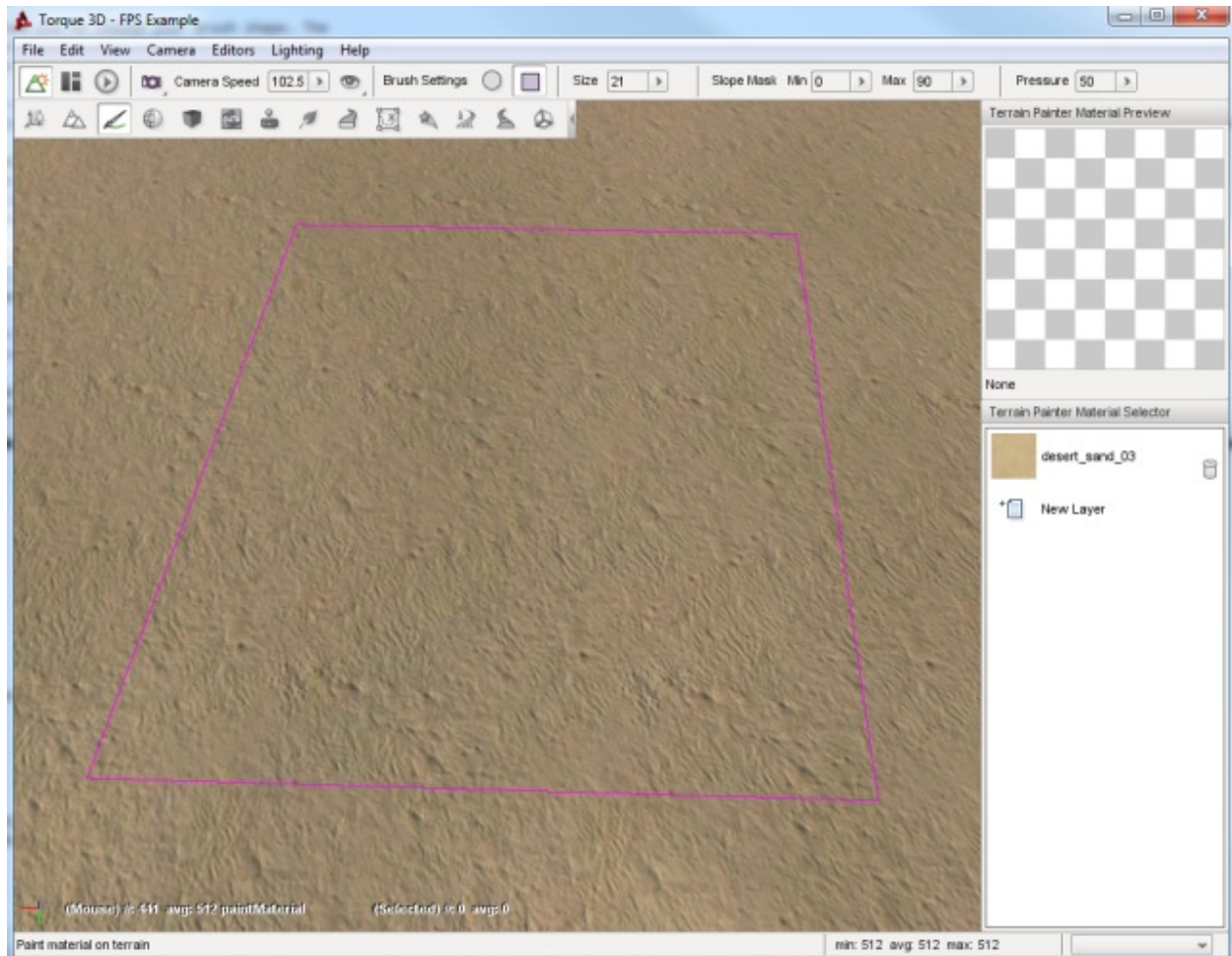
The image shown above is displaying the default brush style when you first open the Terrain Painter. If you wish to change your brush type, you can modify it via the Brush Settings found in the Tool Settings toolbar at the top of the screen. Brush Settings are only active while using the Terrain Painter.

The image shown above is displaying the default brush style when you first open the Terrain Painter. If you wish to change your brush type, you can modify it via the Brush Settings found in the Tool Settings toolbar at the top of the screen. Brush Settings are only active while using the Terrain Painter.

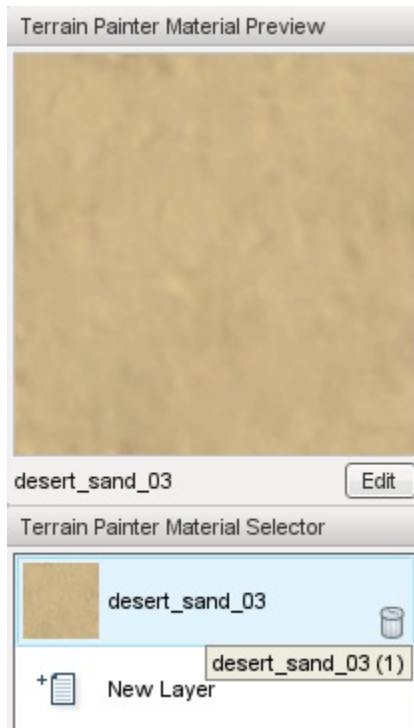


You will find the Brush Size slider next to the shape settings. You can move the slider from left (smaller) to right (larger) to change the size. The stock value is typically small, usually a 9x9 grid. The more you increase the slider value, the greater the grid will grow. The change will add an equal number of rows and columns, as shown below.





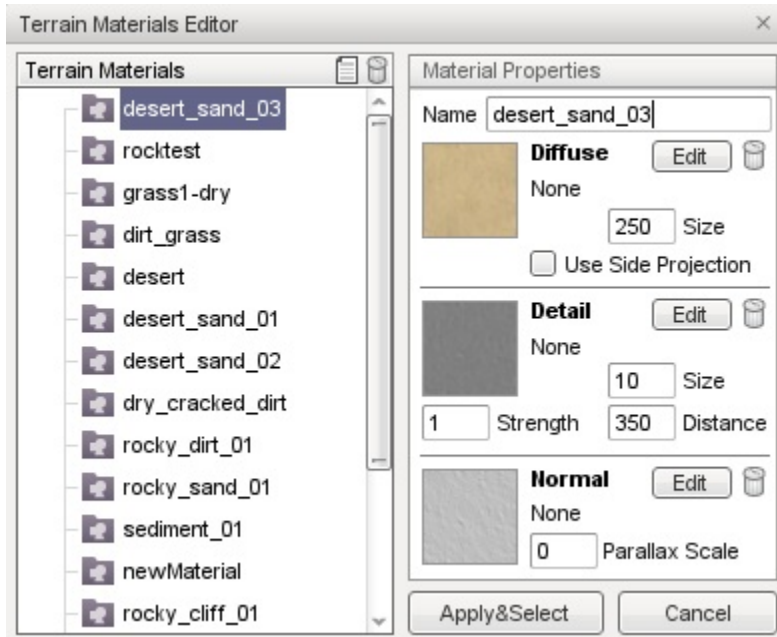
You can find the Terrain Painter palette docked on the right side of the editor. This panel is similar to a traditional painter's palette in the real world. Instead of swatches of color, the Terrain Painter's palette is populated by Terrain-Materials which you use to paint the terrain.



A TerrainMaterial is a collection of three textures combined into a single layer. The three textures are the base (also known as diffuse), detail, and normal map. A preview of which TerrainMaterial (or layer) is shown in the box at the top of the palette labeled Terrain Painter Material Preview.

### 13.2.3 Terrain Materials Editor

When you wish to add a new TerrainMaterial, click on the New Layer entry in the palette. Once you click on the entry, the Terrain Materials Editor window will appear. This tool is completely separate from the basic Material Editor, as TerrainMaterials are structured and used much differently than other Torque 3D materials which are used on shapes in the world placed with the World Editor.



**Terrain Materials** The TerrainMaterials list contains all the currently available textures for creating terrain materials.

**New Button** Clicking the Page icon in the Terrain materials header creates a new TerrainMaterial entry for editing.

**Delete Button** Clicking the Trash can icon in the Terrain materials header deletes the currently selected TerrainMaterial.

**Apply & Select Button** Clicking this button closes the Terrain Materials Editor and returns to what ever operation brought you to the dialog, for the purposes of this article it returns you back to the Terrain Painter Material Selector and adds the selected TerrainMaterial as a new material ready to be used for painting.

**Cancel Button** Close editor without making a choice.

Clicking on an entry in the Terrain Materials list updates the Material Properties pane on the right to display the current properties of that material.

The Material Properties pane contains a Name field, which is used as the label assigned to the material and three sub-sections which describe the textures that define the material.

The Diffuse sub-section shows a preview and the properties of the materials Diffuse texture, which provides the color and base appearance of the material. The Diffuse texture is also commonly referred to as the Base texture for this reason.

The Detail sub-section shows a preview and the properties of the materials Detail Map, which gives the material a more defined, crisp look. If you are familiar with advanced rendering concepts this is accomplished using additive blending and per-layer fade distance techniques.

The Normal sub-section shows a preview and the properties of the materials Detail Map, which gives the material a more defined, crisp look. If you are familiar with advanced rendering concepts this is accomplished using additive blending and per-layer fade distance techniques.

**Name** Assigns the name of the TerrainMaterial which will appear in the Terrain Materials list.

**Edit Button** Clicking this button allows you to select the texture to assign to this aspect of the material.

**Trash Can Button** Clicking this button clears the texture that has been selected for this section.

**Use Side Projection** Terrain diffuse textures are normally applied top-down, which can result in stretching. This toggle causes a material to smoothly merge and conform to steep terrain if needed.

**Diffuse Size** Controls the physical size, in meters, of the base texture.

**Detail Size** How close the camera must be before the detail map begins rendering in meters.

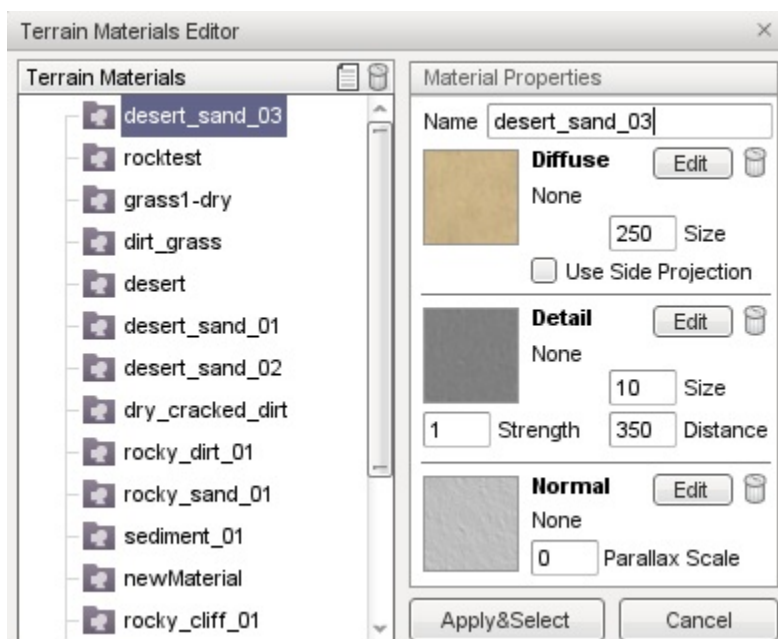
**Detail Distance** Determines how bold the detail appears on the base texture.

**Parallax Scale** Adjusts the intensity of the parallax depth in normal maps.

### 13.2.4 Painting

Before we begin painting, we will add a second TerrainMaterial to our palette (if the project you have open already has more than one feel free to skip this step).

To add a new material click the New Layer button in the Terrain Painter Material Selector. The Terrain Materials Editor will open. Click any TerrainMaterial in the list other than the one that is already in your palette, such as the “rocktest” material shown here.



Once you have the material selected, click the Apply & Select button. Once you have done this, the new layer will have been added to your palette and available for painting.

This is a good time to take a look behind the scenes to understand a little of how Torque 3D organizes materials and how it uses them for other operations to your advantage. What you can not see in the interface is that the system has associated each of the TerrainMaterials in your palette with a numbered layer. Throughout these documents you will see, or may have already seen, that material layers are used to control aspects of object placement such as which layer automatic object placement will occur on.

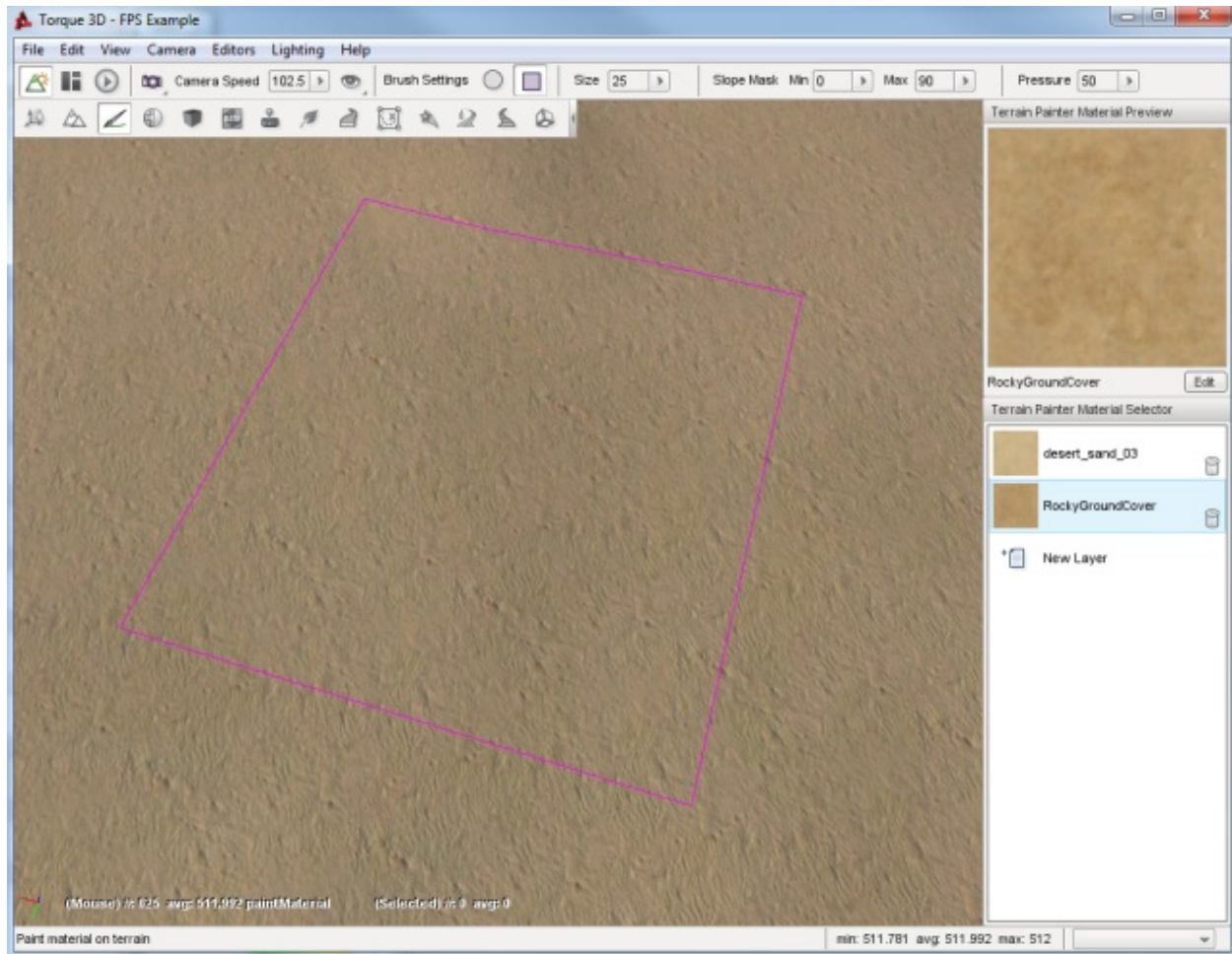
If you started with a project that was created with the Full template and added the rocktest material in the last step then the system now considers grass1 to be layer0 and rocktest to be layer1. This allows you, whenever asked, to select layers using something meaningful to you rather than remembering some random numbering system. When asked to select a layer you can simply pick the grass or rocktest layer from a list and the system will use and apply the proper numbered layer to perform the related operation.

All this becomes very important in reducing the amount of work that is needed to create realistic terrain. The Terrain-Materials that you apply with the Terrain Painter tool not only give the terrain the appearance of natural materials but they can be used to automatically generate and restrict foliage and other shapes when used in conjunction with objects such as GroundCover.

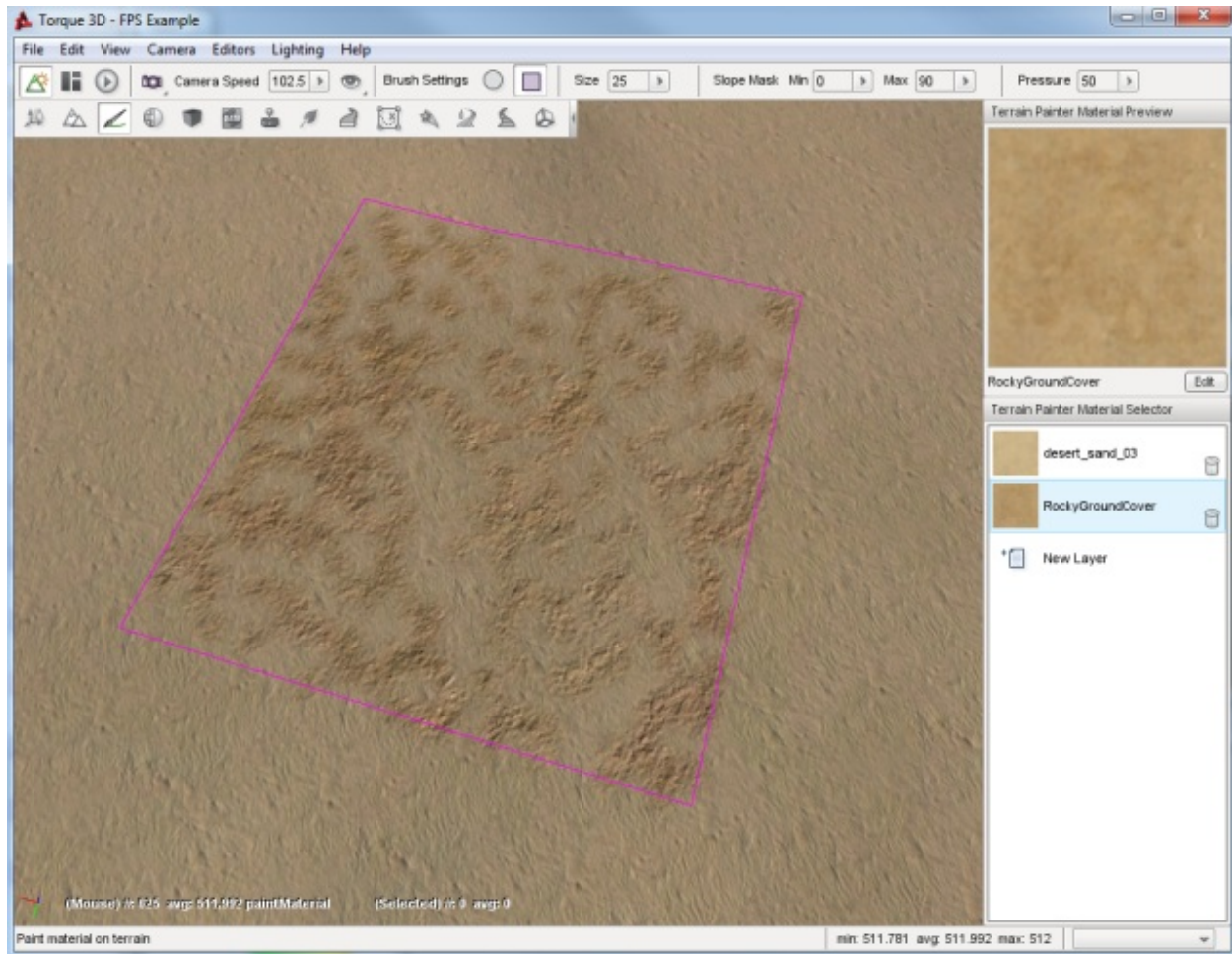


Now, on to learning to paint. Make sure you have the new material selected in the Terrain Painter Material Selector. So we can more easily see the modifications we are about to make, set your brush size to about 25. Now, find a section of the terrain you wish to paint. Here, we started in a corner of the TerrainBlock.

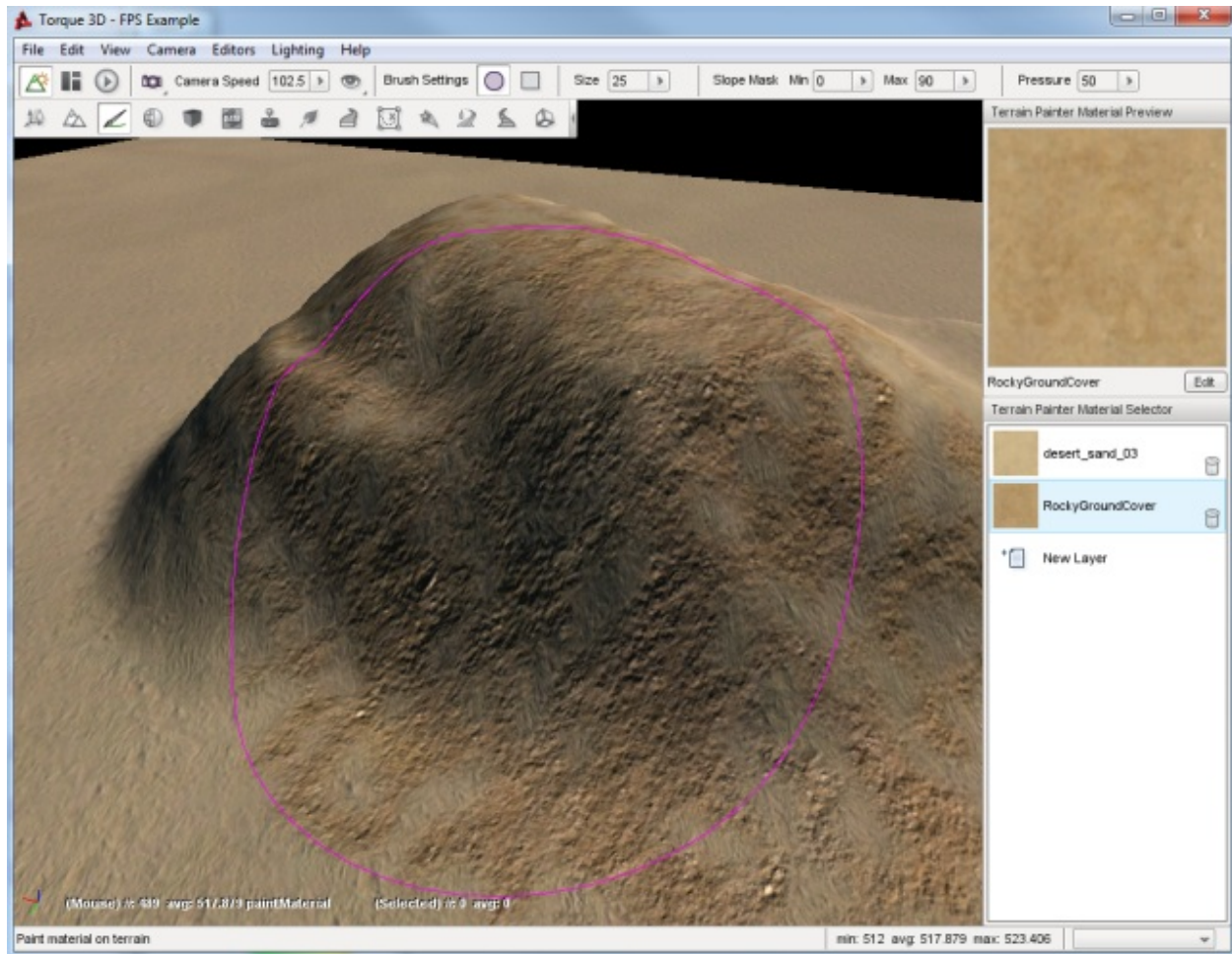




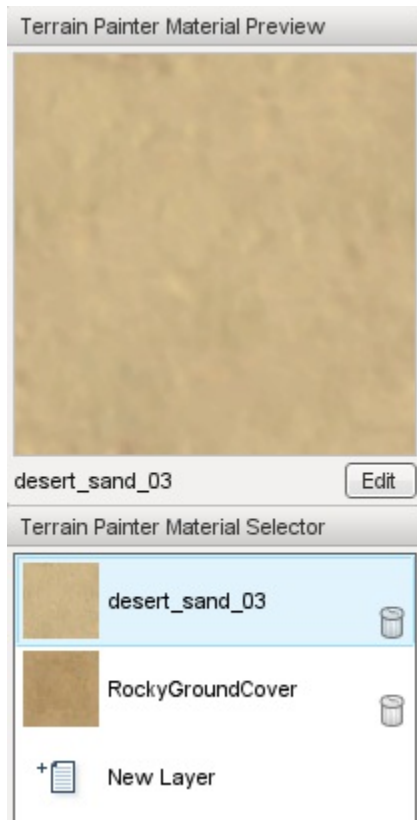
Click and hold down the left mouse button, then begin dragging the brush around the screen in a sweeping motion. The terrain will update in real time to reflect the painting of the new TerrainMaterial. When you let go of the mouse button, the Terrain Painter will stop laying down the material.



You should have noticed that your brush clamped to the terrain as long as the cursor was over the block. This happens regardless of any terrain modification or elevation occurring, as shown in the following example. Notice how the brush distorts to wrap around the elevated terrain.

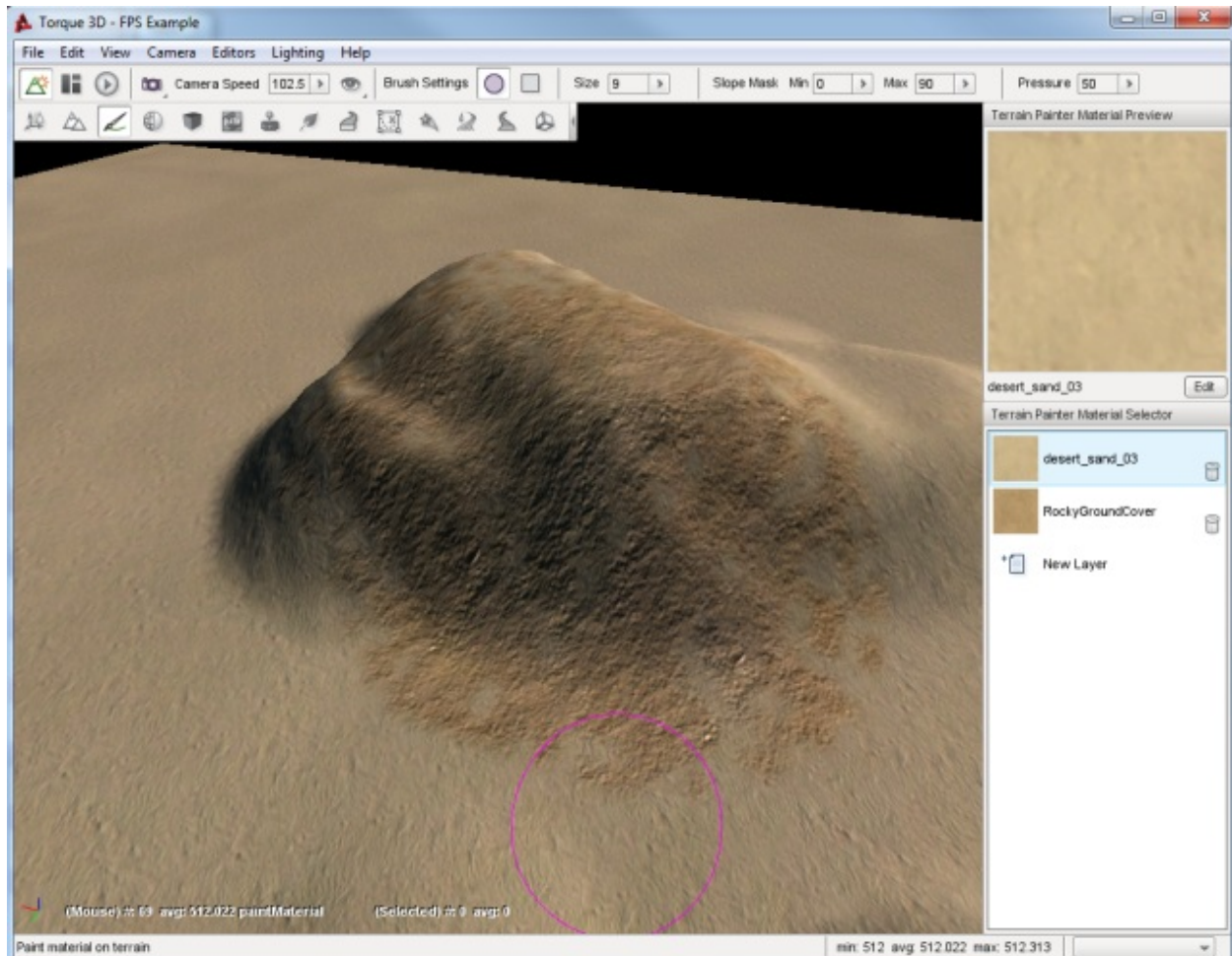


Even though you just paved a large section of rock material, you can still paint over it. Decrease the Brush Size to approximately 9, so we can paint a more exact line of terrain. We are going to paint a path over our rocky area. In the Terrain Painter palette, select the first material (`desert_sand_03` in this image).

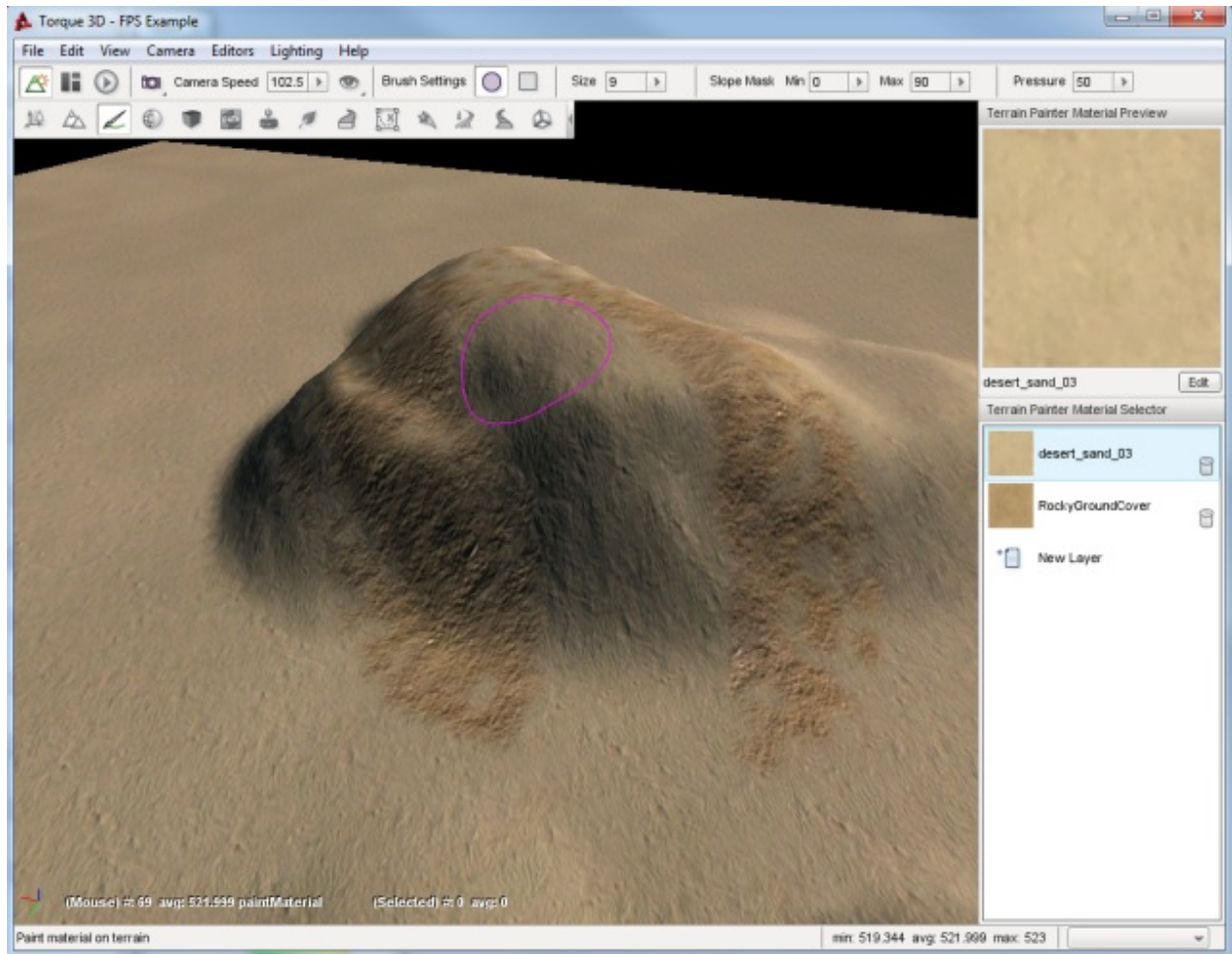


Now, using your mouse cursor move the brush to the edge of our rocky area. You can start it just before the rocky area, or even on top of it.





Click and hold down the left mouse button to begin painting then sweep your mouse in a curving motion across the rocky area. When you are finished, let go of the mouse and examine or your winding path made of grass.



If you were to drop down to the player's camera view, you can see where the two TerrainMaterials meet each other after editing.



Take the time to experiment with different brush sizes and shapes to see what kind of patterns you can come up with. When you are ready, read on to learn how to add a new `TerrainMaterial` with higher quality and detail.

## 13.3 Material Editor

Torque 3D's Material Editor allows an artist to quickly create and edit a game object's materials without ever touching a line of code. This tool can preview and edit materials mapped to an object in real time from the World Editor. Materials are categorized for ease of organization, and the editor is designed to be backwards-compatible with any existing script files. Materials are defined within script files named `materials.cs`

The Material Editor quickly comes into play when you are building your level by placing objects into the scene. As an example situation, let us assume you have a light fixture you or another artist has exported for use in Torque 3D. The creation of this object was a multi-step process.

The object's geometry had to be created in a 3D modeling app, such as 3DS Max, Maya or Blender. Once the geometry was finished, a 2D graphical application was used to create the various textures that make up the high quality appearance: base texture (diffuse map), normal map, detail map, etc.

The rendered view of this object looks fine in the originating application, which leads you to believe it is ready to be imported into your game's level. When you drop the lighting fixture into the world, there is a good chance it will fit right in based on your existing design. The theme and color scheme are likely a match.

However, once your object is in the level you might notice something is off. While the stand alone object viewed in an

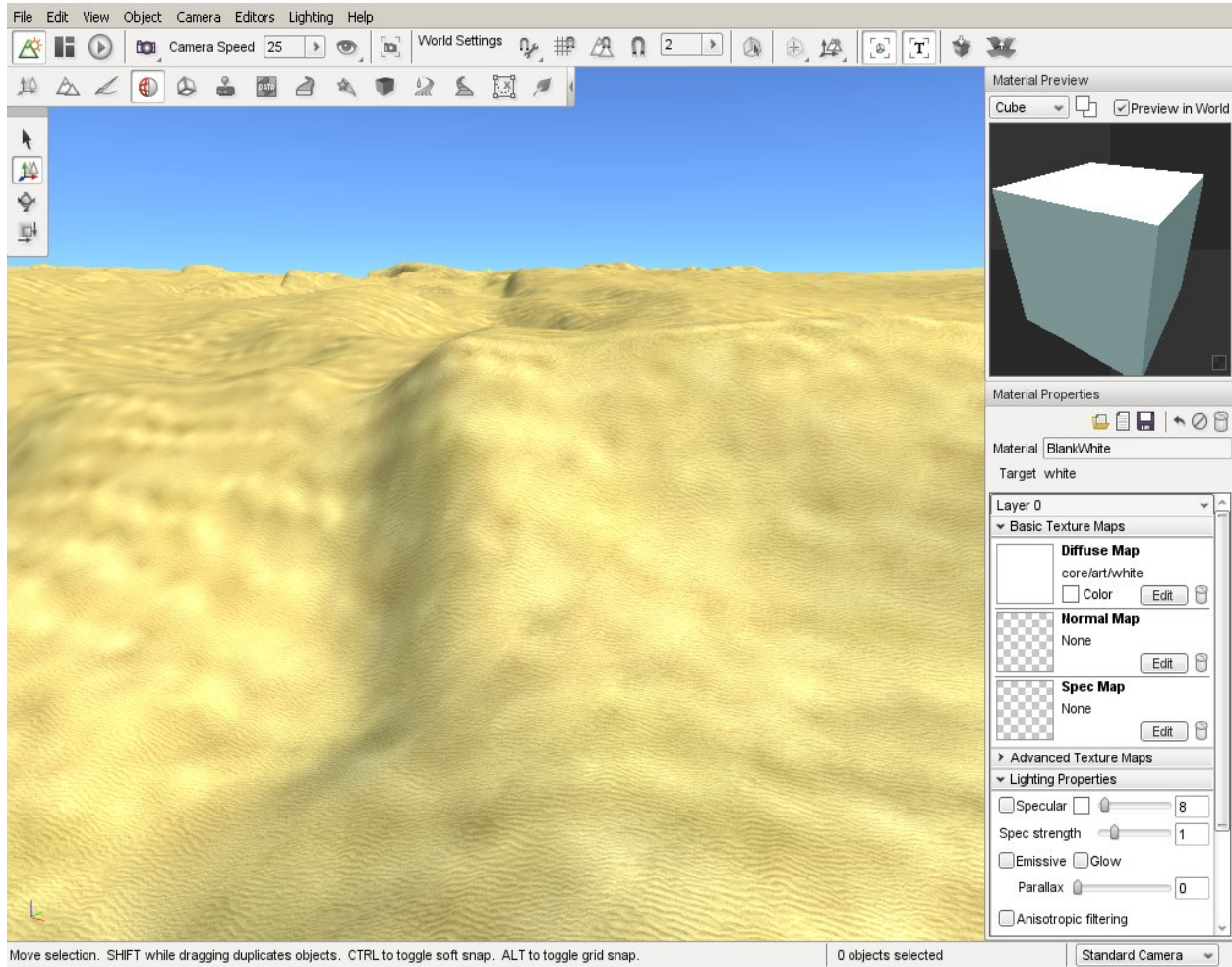


external tool looked great, the object now seems out of sync with your level's lighting, tone, or specific room theme.

This is not the fault of the artist or design. What has happened is the materials for your object have either not been assigned or need to be tweaked to perfection in specific instances. Instead of going back into the art tool or adjusting properties in code, you can use the Material Editor to edit the appearance of your object by adjusting the texture maps and their properties.

### 13.3.1 Interface

To switch to the Material Editor press the F4 key or from the main menu select Editors -> Material Editor.



#### Main Editor

Each major section of the Material Editor is separated via a header, such as Lighting Properties or Animation Properties. The fields found in these sections directly manipulate a materials properties in real time.

#### Material Preview

**Cube** Changes the Preview Mesh.

**Square Symbols** Change Normal Light Color.



**Preview in World** Show changes made in material editor on selected object in scene view.

**Bottom Right** Click square to change color of background preview.

## Material Properties

**Edit Material** Select and Edit an Existing Material (E).

**Floppy Disk** Save material.

**Trash Can** Delete Material.

**Name.dts** Name of 3D asset using this material.

**First Drop Down** Texture associated with material.

**Material** Name of material.

**Square with Ball** Swap current material mapped to this mesh for another.

**2nd Trash Icon** Remove this material from mesh target.

## Basic Texture Maps

**Diffuse Map** Base texture for material.

**Normal Map** Bump map that provides higher detail to mesh without extra polygons.

**Overly Map** Texture draw on top of other maps.

**Detail Map** Texture providing additional detail via lightening and darkening base map using high pass filter.

**Light Map** Texture using baked lighting info.

**Tone Map** Map which scales the RGB values of material. Used to calculate HDR.

## Advanced Texture Maps

**Diffuse Map** Base texture for material.

**Normal Map** Bump map that provides higher detail to mesh without extra polygons.

**Overly Map** Texture draw on top of other maps.

**Detail Map** Texture providing additional detail via lightening and darkening base map using high pass filter.

**Light Map** Texture using baked lighting info.

**Tone Map** Map which scales the RGB values of material. Used to calculate HDR.

## Lighting Properties

**Specular** Enables the use of Pixel Specular (shininess) for this layer. The slider adjust the strengt, and you can set the color of the specularity.

**Glow** Determines if this layer will Glow or not.

**Exposure** Intensifies glow and emission.

**Emissive** Causes an object to not be affected by lights. Good for materials from light source objects.

### Animation Rotate Properties

**Purpose** Causes material to rotate along the surfaces of the mesh it is mapped to.

**U and V Sliders** Determines the direction of U/V coordinate rotation.

**Speed** Rate of coordinate rotation.

### Animation Scroll Properties

**Purpose** Causes material to scroll along the surfaces of the mesh it is mapped to.

**U and V Sliders** Determines the direction of U/V coordinate scrolling.

**Speed** Rate of coordinate scrolling.

### Animation Wave Properties

**Purpose** Causes the material to scroll in a wavy manner along the surfaces of the mesh it is mapped to.

**Wave Type** Switch between sine, triangle, and square wave patterns.

**Amplitude** Changes the positive and negative crest of the wave (intensity).

**Frequency** Adjust wave length, which is the number of waves per time interval.

### Animation Sequence Properties

**Purpose** Animates texture by frames.

**Frames per Sec** How many frames to display per second.

**Frames** Number of total frames in the sequence.

### Advanced Properties

**Purpose** Adjusts advanced parameters that affects transparency calculations.

**Transparency Blending** Sets material to use transparent blending modes.

**Transparent Z Write** Can be used to help force a proper Z Ordering when Z Ordering issues occur. Only valid on materials with Transparency.

**Alpha Threshold** When enabled, causes pixels under a specific alpha threshold to get discarded rather than be computed.

**Cast Shadows** Material determines whether target mesh is allowed to cast shadows.

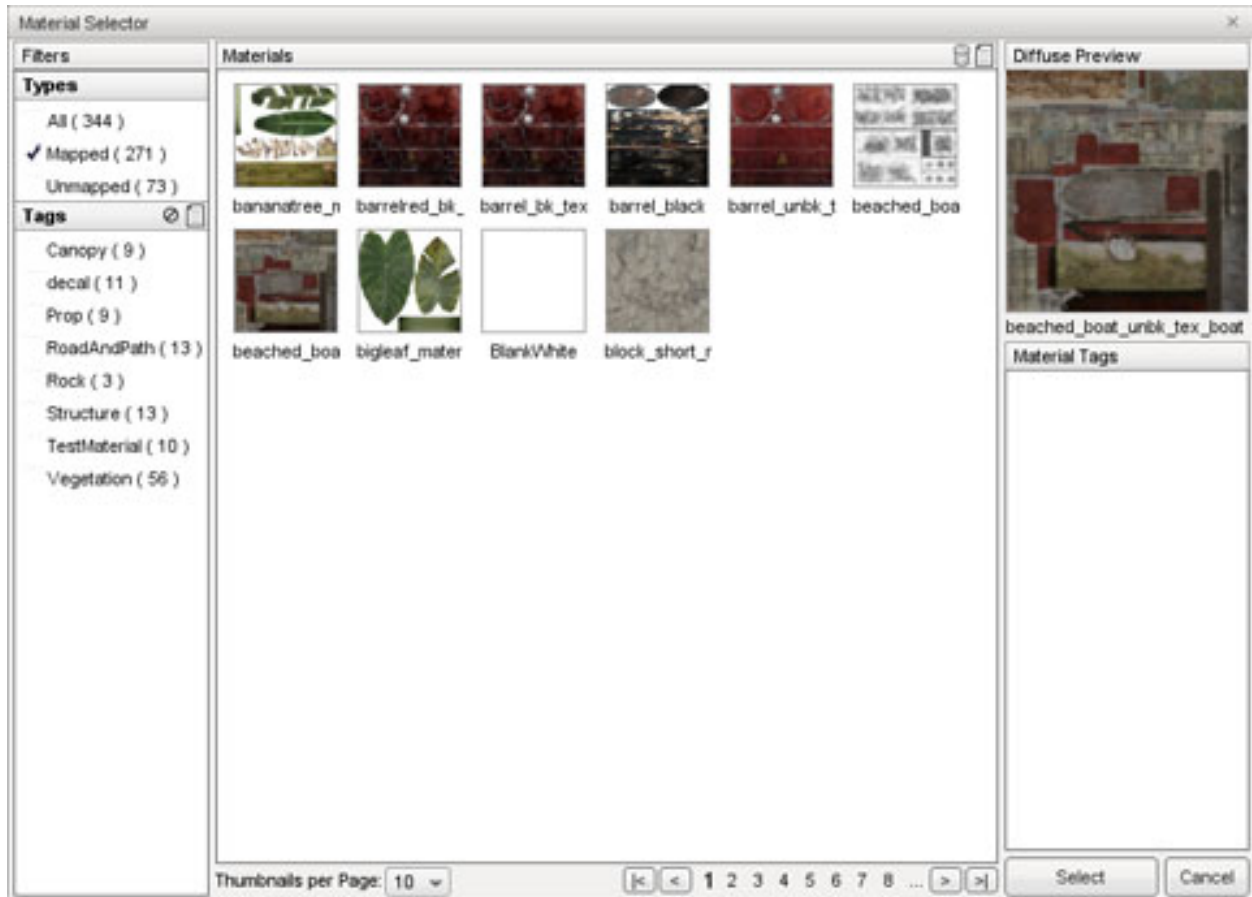
**Double Sided** Determines if this material will be rendered from both sides of a polygon.

**Blending Box** Determines type of blending and reflection applied on the transparent object.

## Material Selector

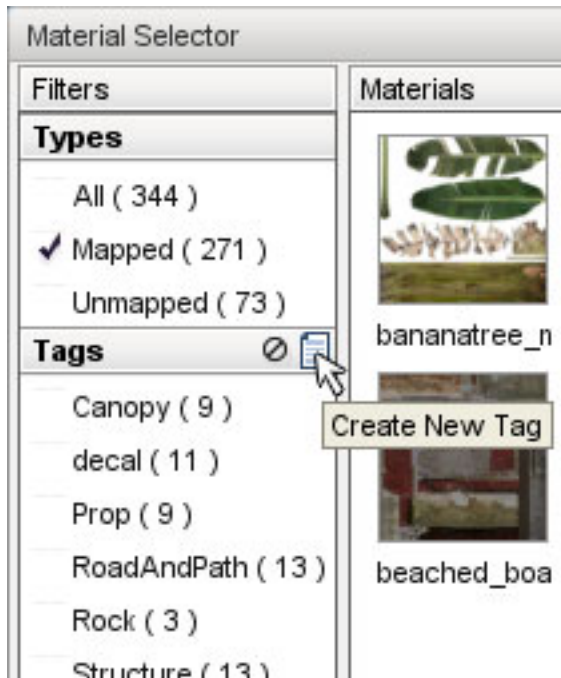
When you wish to swap the material mapped to an object or create a new material, you will use the Material Selector. To change the material on an object, it must first be selected. If you do not know how to select an object, refer to the Object Editor documentation, then switch back to the Material Editor (F4). The Material Properties pane on the right side of the screen displays the properties that describe the material of the selected object.

At the top-right of the pane there is a value named Material. Click on the globe to the right side of it. This will bring up the Material Selector window.



The center section of this dialog displays a list of all materials currently loaded in the game. Clicking on any material selects it which will cause the panes on the right to update and display information about the material. This information is limited to a preview of the material's Diffuse texture, the name of the diffuse texture, and a list of filter tags.

On the left is a list of filters. The filter system is used to organize your materials for ease of use, and contains types and tags. To create a new tag, click the new tag button:



The Create New Tag dialog will pop up. Enter a name for your new the tag then click the Create button. In this example, you will be grouping all the materials related to cliffs. Whenever a material is selected, the Material Tags section on the right will be updated to show all the tags that you have created, each with a checkbox. Clicking the box of a specific tag will associate that tag with the current material. Clicking a checked box will dissociate the tag from the material.

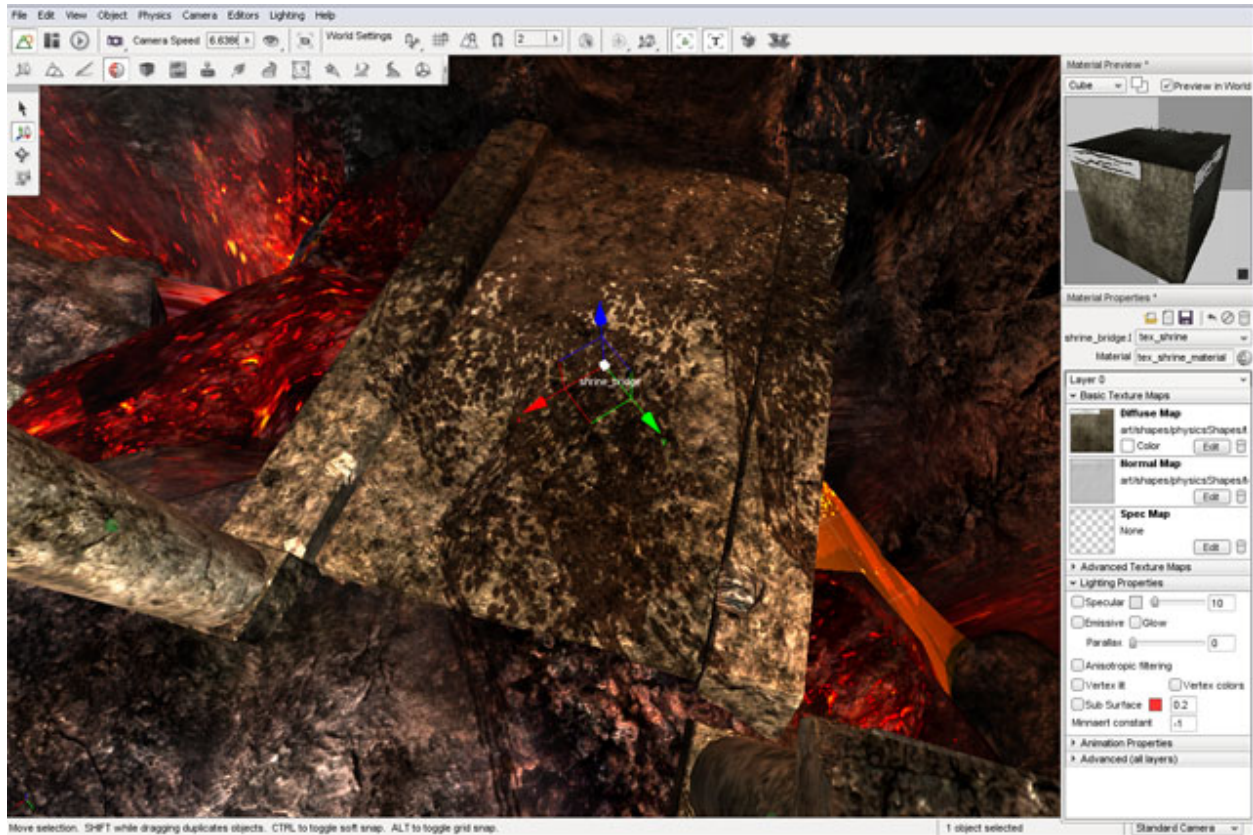
The list of materials can be filtered using the tags assigned to them. To filter the material list use the tags section on the far left. When you click on the check box for tag it tells the system to include materials that have that tag in the list. Any materials that do not have at least one of the checked tags will be filtered out of the list.

### 13.3.2 Editing an Existing Material

Your game's levels can potentially contain thousands of different objects with varying purposes: explosive barrels, ammo crates, static light fixtures, solid walls, etc. Each one will have a material that might need subtle tweaking to fit in, such as a glowing light bulb.

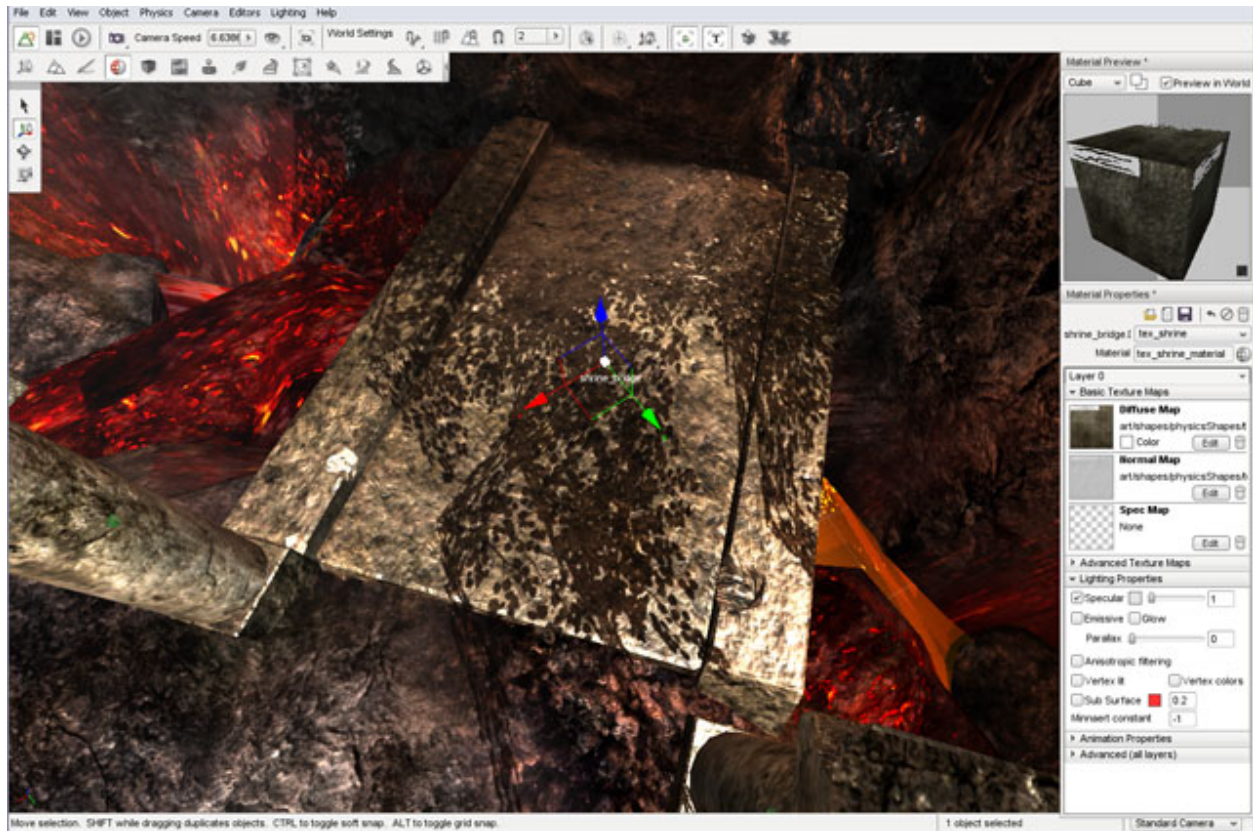
In this example, you will adjust the properties of this bridge materials.

Remember that you can preview the changes in the scene as well as the preview box in the Material Editor. You will start by toggling the Specular property of the material used for the metal pipe. Without Specular enabled, an object will not have a shine and will thus appear flat.



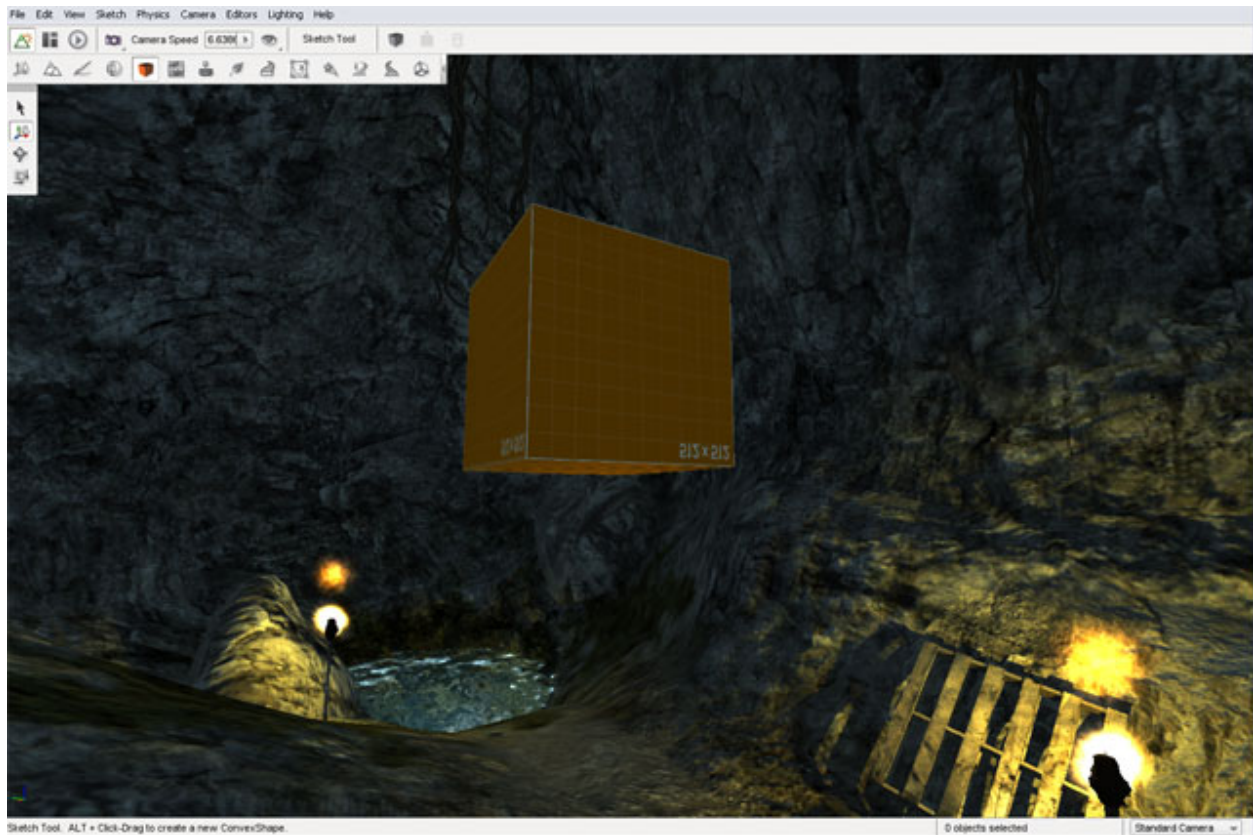
When the Specular property is enabled, the cube in the preview box will have a shiny appearance. In the scene, the metal will also be shinier due to the lighting reflection.





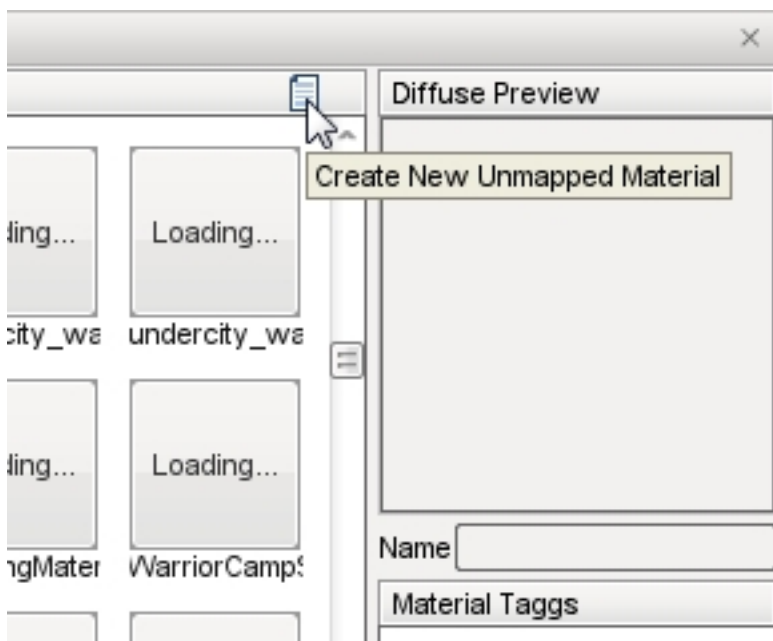
### 13.3.3 Creating a New Material

While developing your game, you will most likely be using your own assets. When you add a model to the scene, it will be assigned the default “No Material” texture which serves as a warning to the designer that no material has been assigned to an object. This material is automatically used for all assets before they have a mapped materials.



If you have already created the textures for your object, creating and assigning a material is a simple process. Start by clicking the globe symbol next to the Material name box.

The Material Selector dialog will appear. Click the Create New Unmapped Material button found at the top right of the Material section's header.



A new material will be added to the list with a name similar to newMaterial\_0. Click on the material to view it in the Diffuse Preview section.

Click the Select button to use that selected material for the object you are editing. After the Material Selector closes, you will be prompted to save any material changes that you may have made before entering the Material Selector. Do so if you wish to retain any changes that you made prior to creating the material.

Your new material will have replaced the material selection in the Material Properties pane back in the Material Editor and should now be displayed in the Material field. Type in the real name you want for your new material to be known by then press the Enter key. In this example, the name of the material is “boxxy.”

Before editing anything else, click the Save Material button, represented by the floppy disk symbol to save the new material.

---

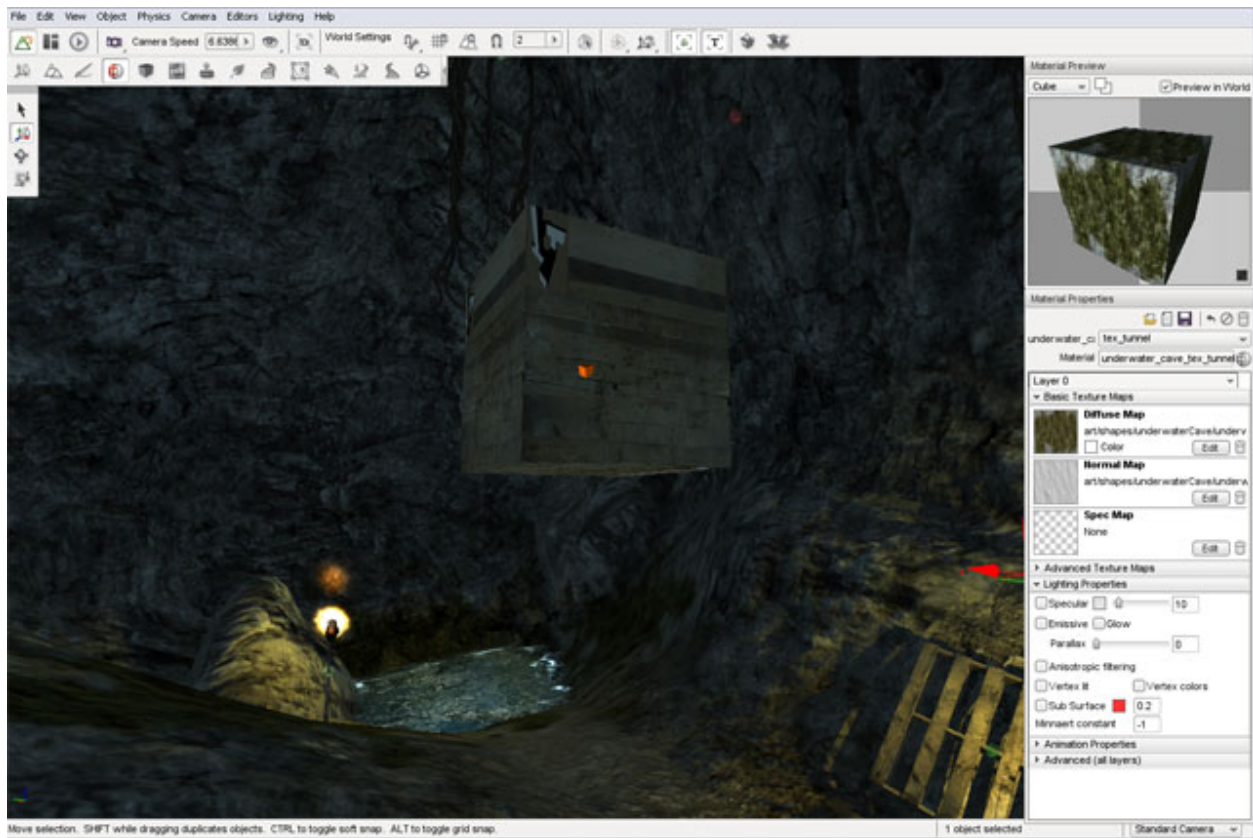
**Note:** You MUST press the Enter key after typing the material name BEFORE clicking the Save Material button or the material will not be properly saved.

---

Now, scroll down to the Texture Maps section of the Material Editor. This is where you will be adding the actual texture files that define this new material. Click on the Diffuse Map preview or the Edit button in that section to open a file browser. Navigate to your diffuse texture, or sometimes referred to as the base texture. Select the file that you want to use as for this new material then click the Open button.

Your preview window and scene should immediately be updated to reflect the addition of your texture.

Repeat the process to add your Normal map. Click on the preview or edit button in the Normal Map section. When the file browser appears, select your normal map texture. Once again, your scene will be updated to reflect the changes that have been made to the material. Click the save button to retain these changes.



If you open the Material Selector again, you will notice your new material has been saved in the list. This material is now available to be assigned to any other meshes within the project without having to go through the whole process of redefining it again.

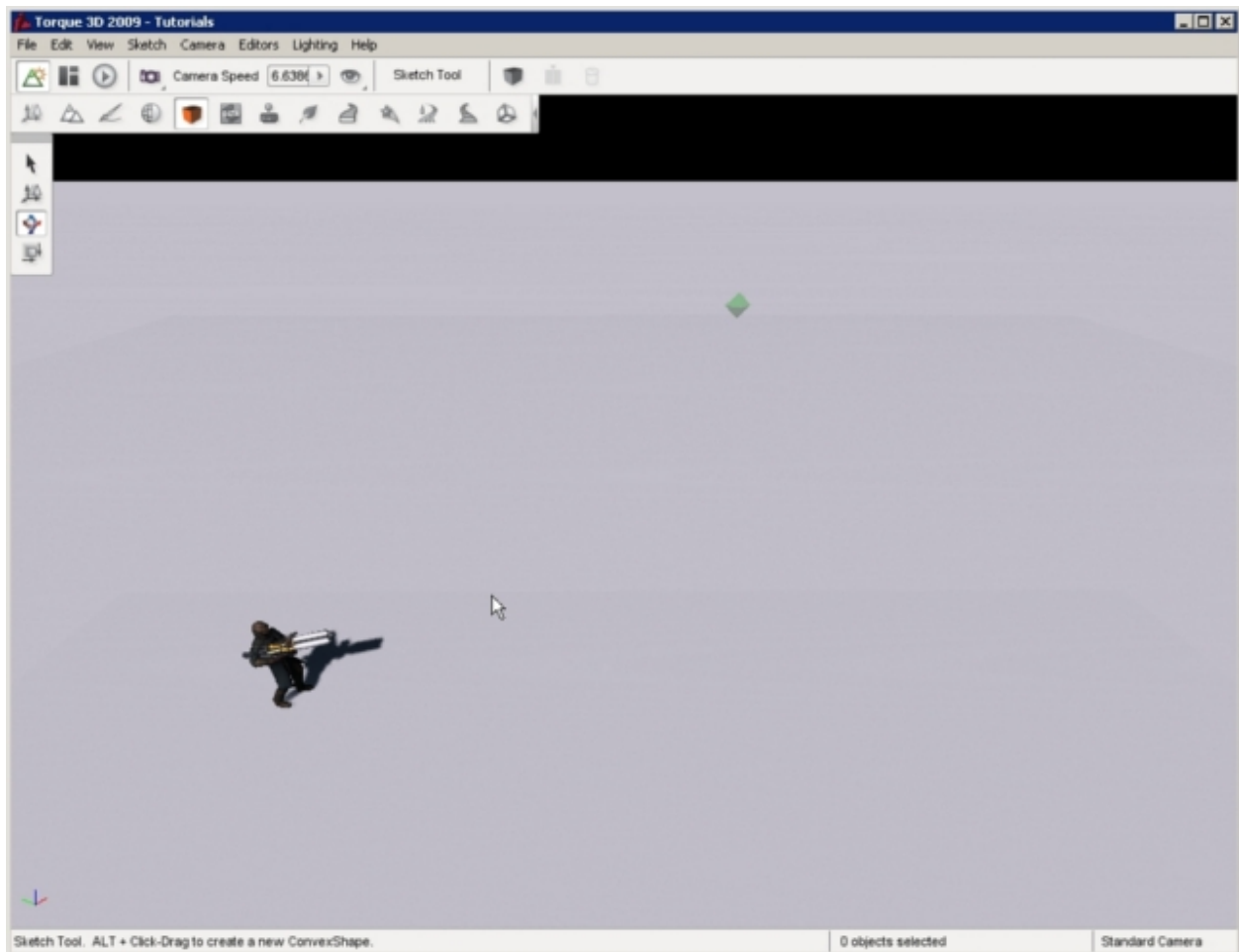


## 13.4 Sketch Tool

The Sketch Tool is a tool that allows you to quickly generate meshes without going to 3rd party modeling applications, such as Maya or 3DS Max. It is not meant to create final or game ready art, just rough shapes that are placeholders for your real art. For example, you can use this tool to sketch the shape of a building you want. The rough design can fit your needs for a simple design and estimated measurements.

### 13.4.1 Interface

To switch to the Material Editor press the F4 key or from the main menu select Editors > Material Editor or click on the orange box icon to get started.



The Tools Palette will populate with basic manipulation icons:

**Select Object** Select a convex object or individual face

**Translate** Move an individual face

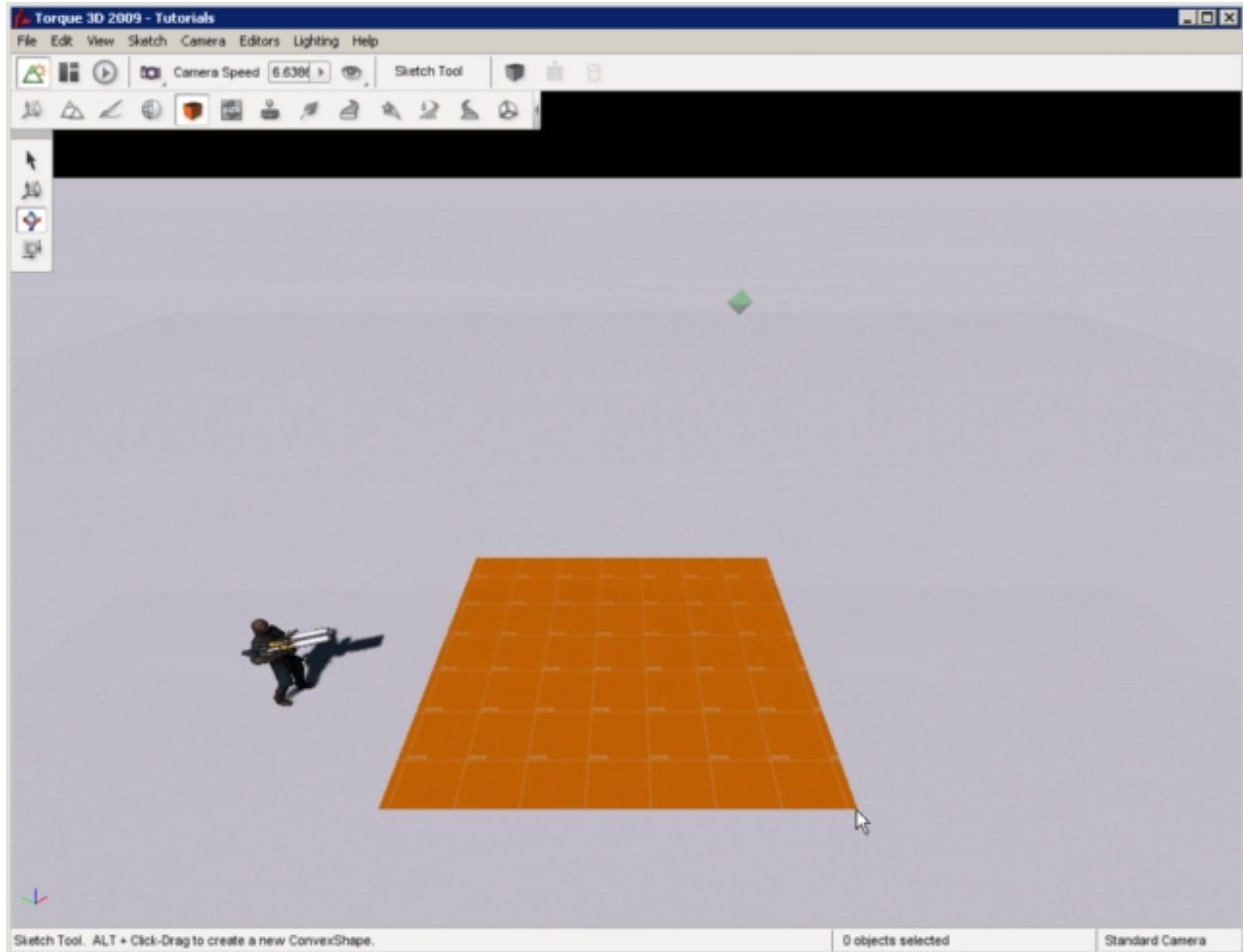
**Rotate Object** Rotate an individual face

**Scale Object** Grow or shrink an individual face

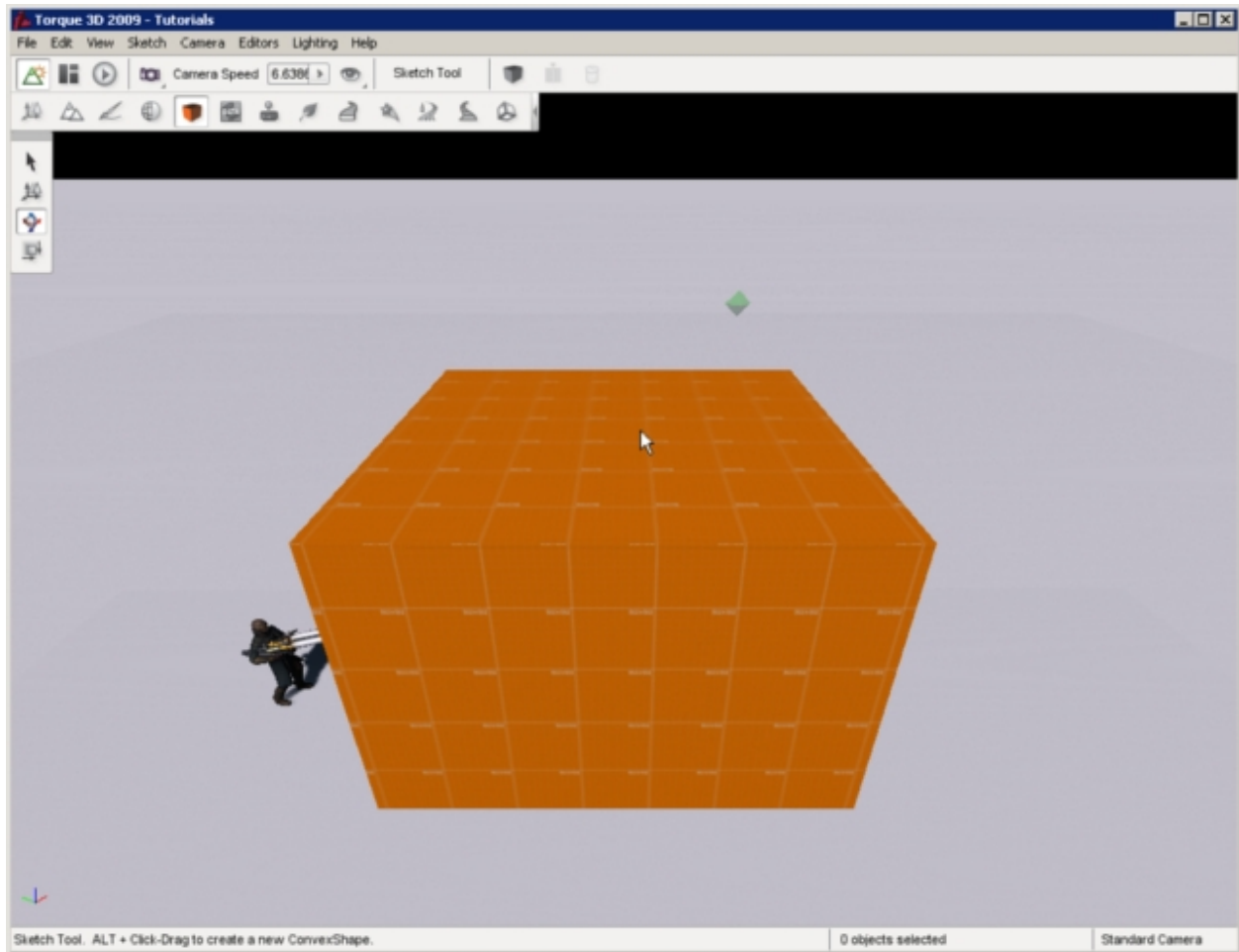
As with the other editors, extremely helpful usage hints will be displayed in the bottom left corner of the editor. Shortcuts and basic descriptions will appear based on which tool you are using.

### 13.4.2 Creating a Convex Shape

The very basic interface allows you to quickly sketch out convex shapes. All of your editing can be performed via mouse actions. To begin creating a convex shape, hold down the Alt key and left mouse button to begin drawing a base. The base will follow where your mouse cursor is being dragged, shrinking or growing as it goes.

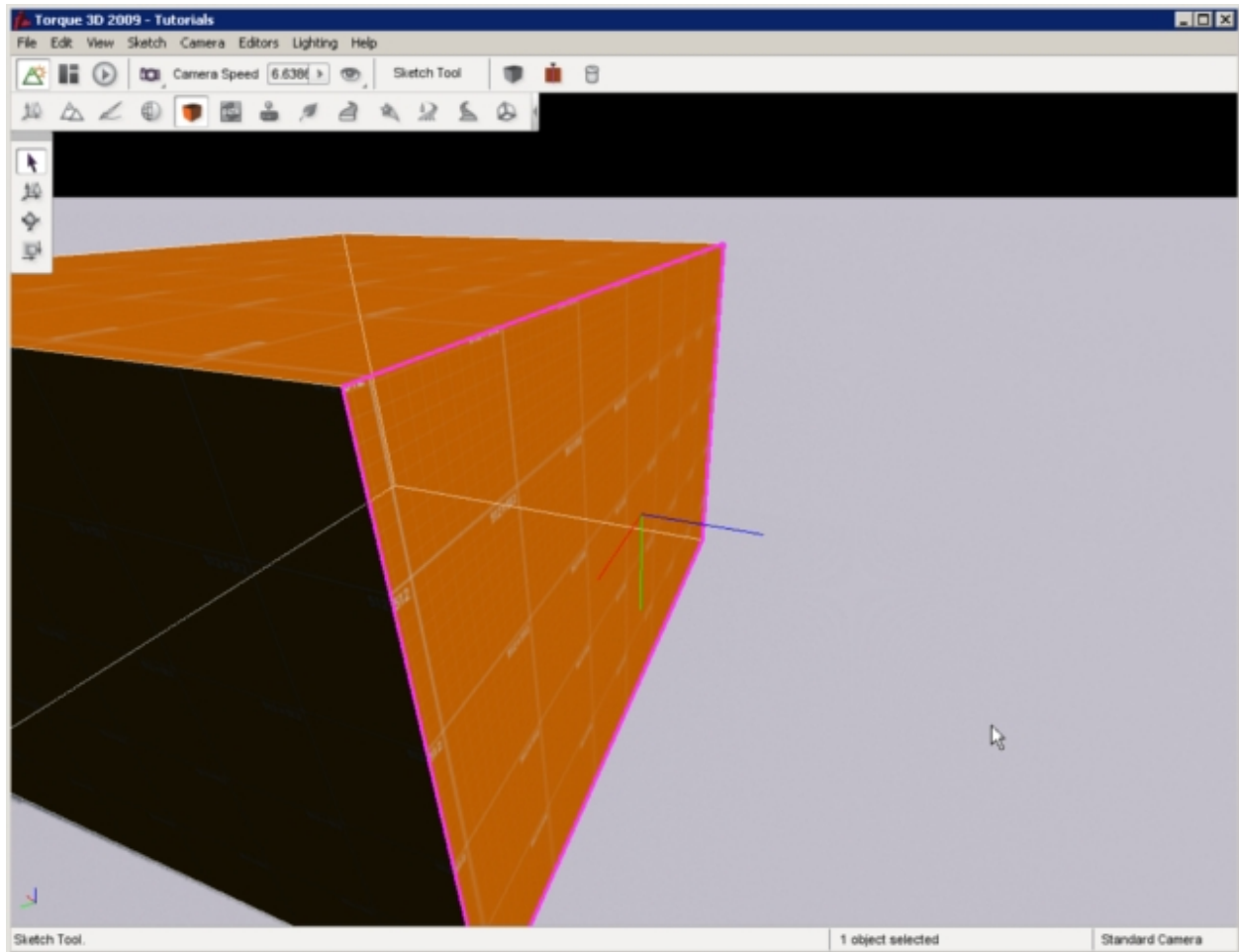


Once you let go of your mouse button, the base will stop growing. From here you can move your mouse cursor up and down to change the height of your new box. You do not have to hold down the mouse button during this time.



Once you are happy with your convex shape's height, left click one last time. The box will become a solid object and automatically be selected. If you make a mistake, hit `Ctrl-Z` to undo and erase the shape then repeat the process.

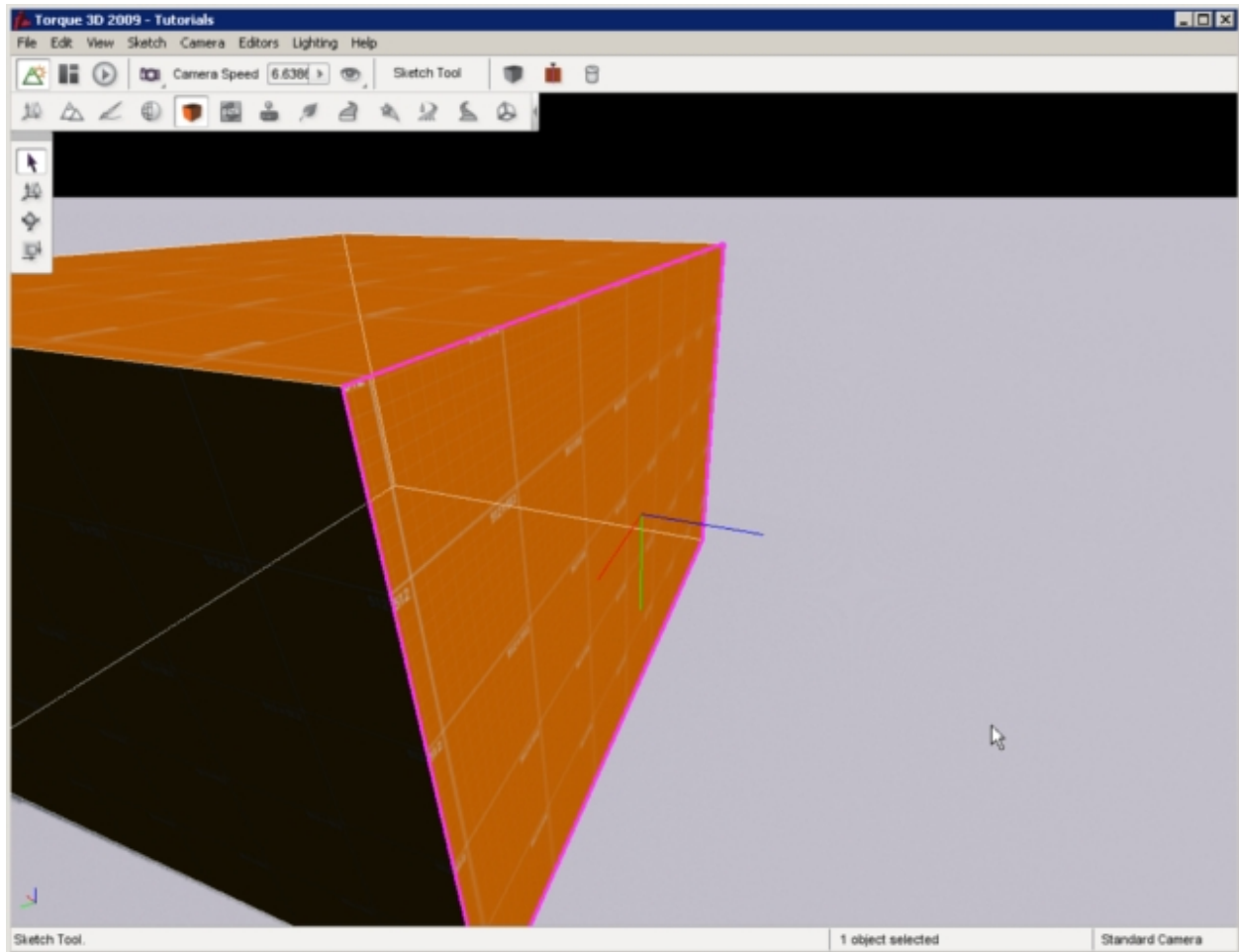
When you are ready to begin shaping the box, left click one of its faces. The currently selected face will be highlighted in bright pink:



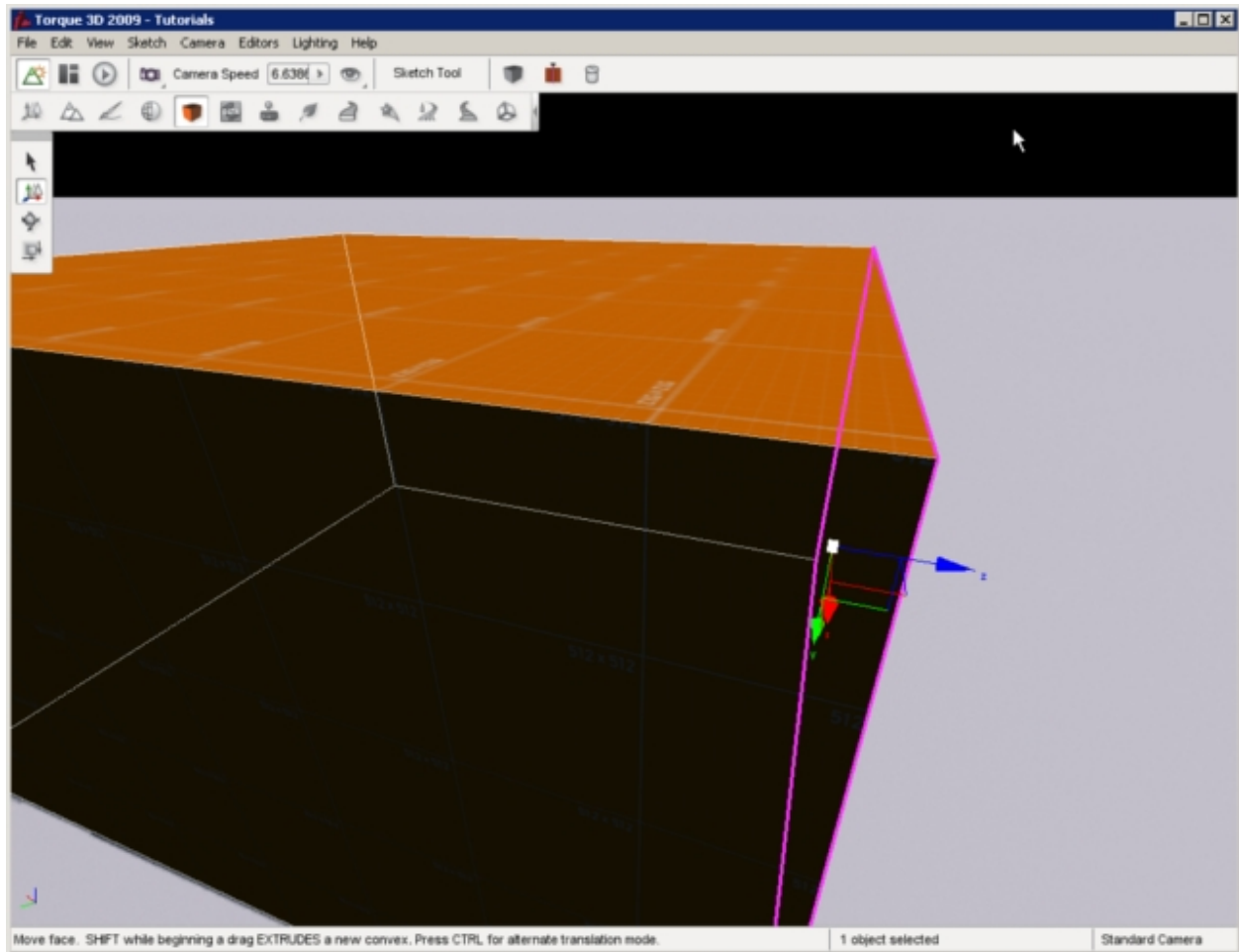
At this point, you can start using the Sketch tools to edit the specific face you have selected.

### 13.4.3 Editing a Convex Shape

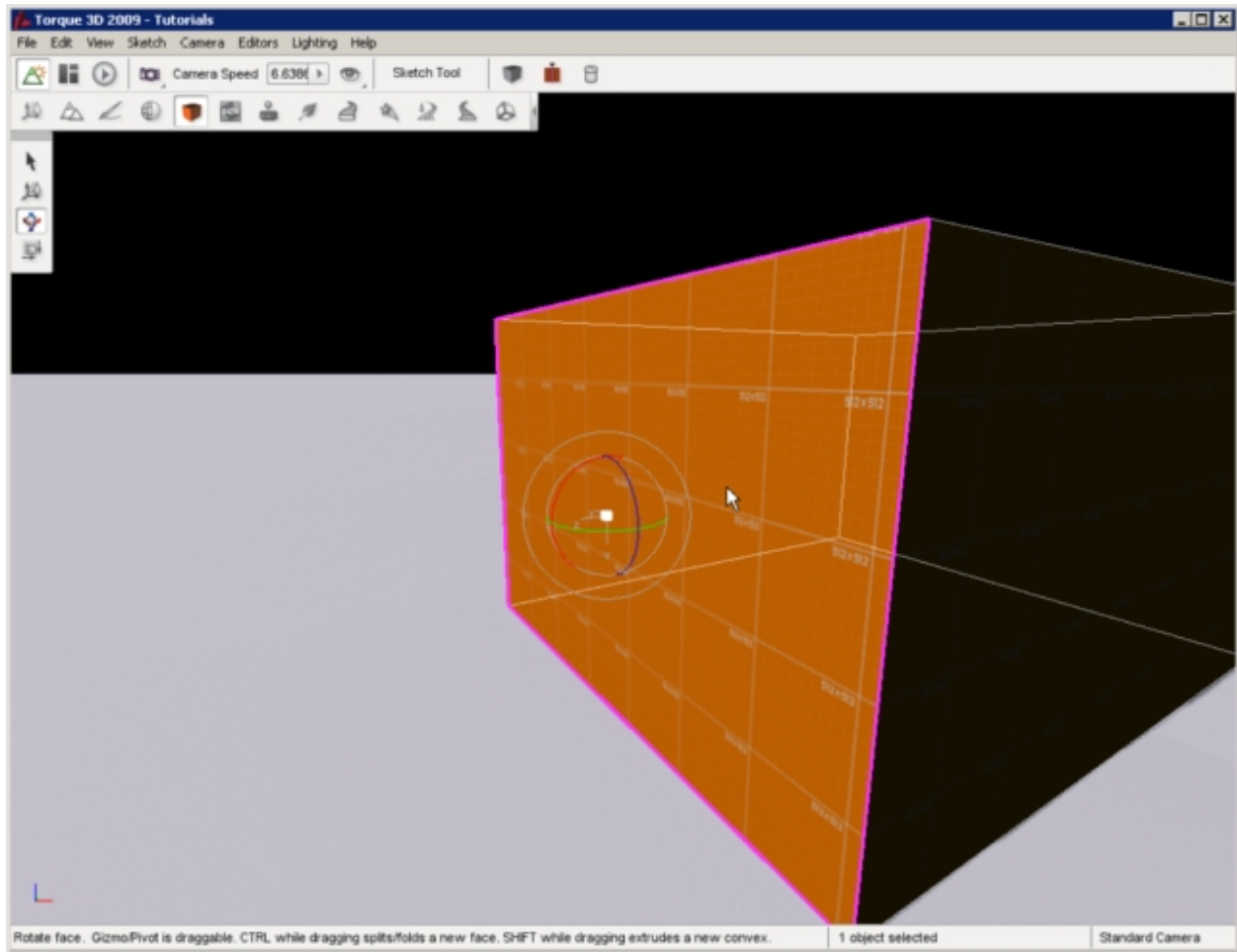
Let's move some surfaces around. Start by selecting a face of the object by (left clicking on it). Three colored lines will now extend from the center of that face - these represent the axes for the three dimensions x, y and z. This is called the axis gizmo. Activate the Move Selection tool by clicking the icon on the Tools Palette on the left of the screen or press the shortcut key 2:



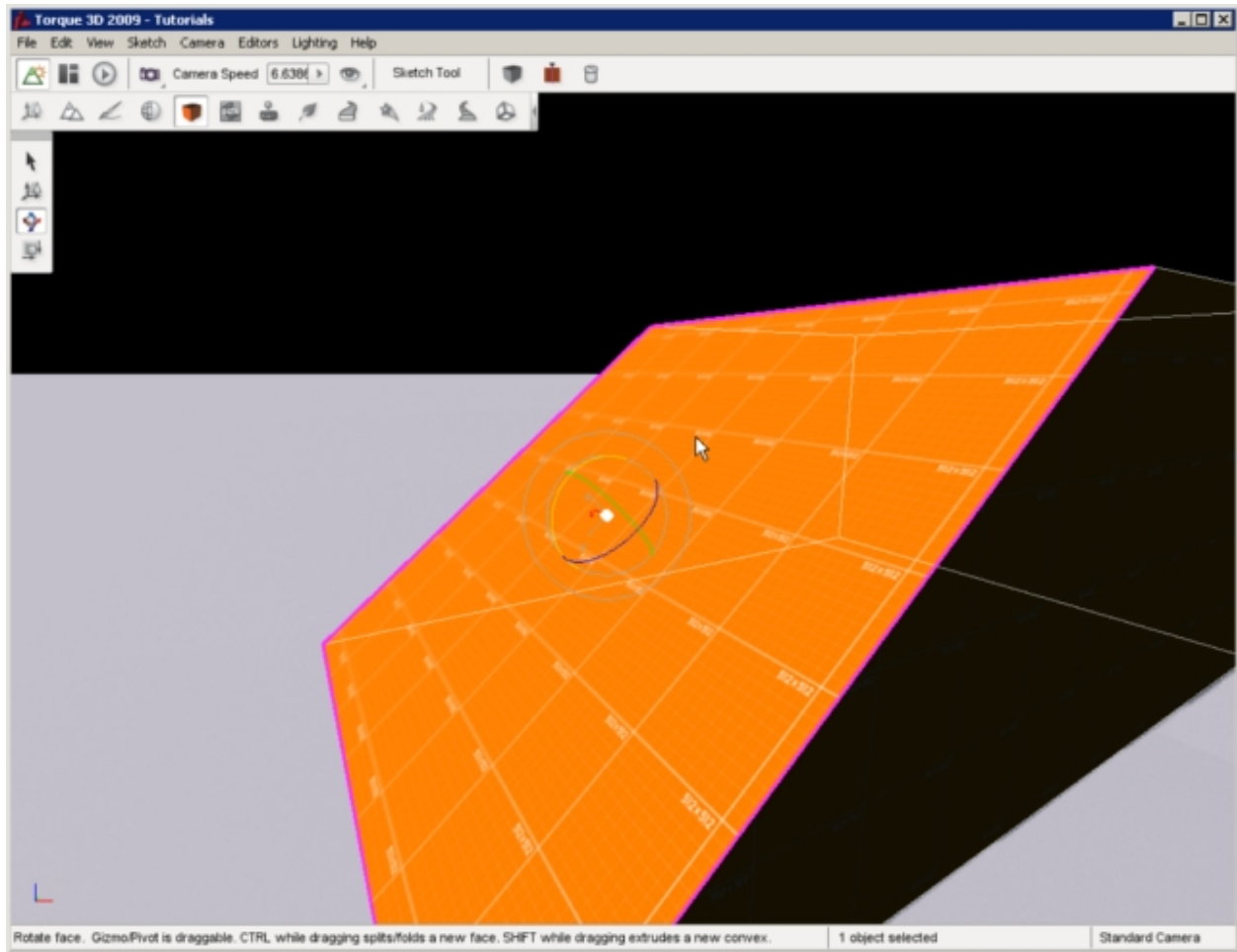
Once the Move Selection tool is activated, arrows will appear on the ends of the axis gizmo. Click on the X-axis and drag it outward. Your face will move in the direction you are dragging your mouse. The entire convex shape will adjust according to where the face as moved. You will be able to move the face in any direction in three-space:



Next, activate the Rotate Selection tool by clicking the icon on the Tools Palette on the left of the screen or press the shortcut key 3. A spherical gizmo will appear representing the orientation manipulators. The axis gizmo straight lines will now be displayed as three curved colored lines:

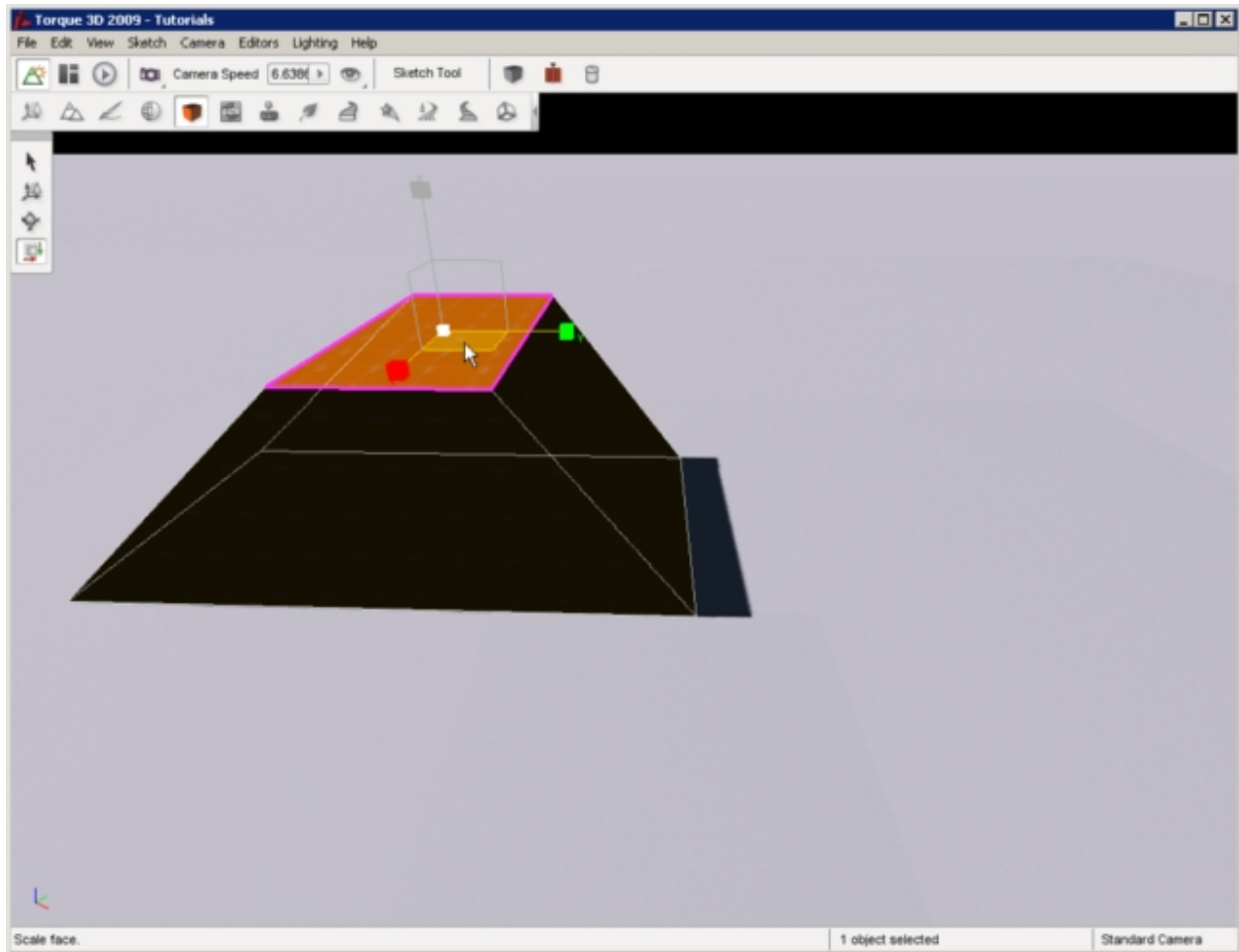


Click and hold one of the colored lines (an axis), and drag it in a direction. The selected face will begin to slope according to the new orientation:

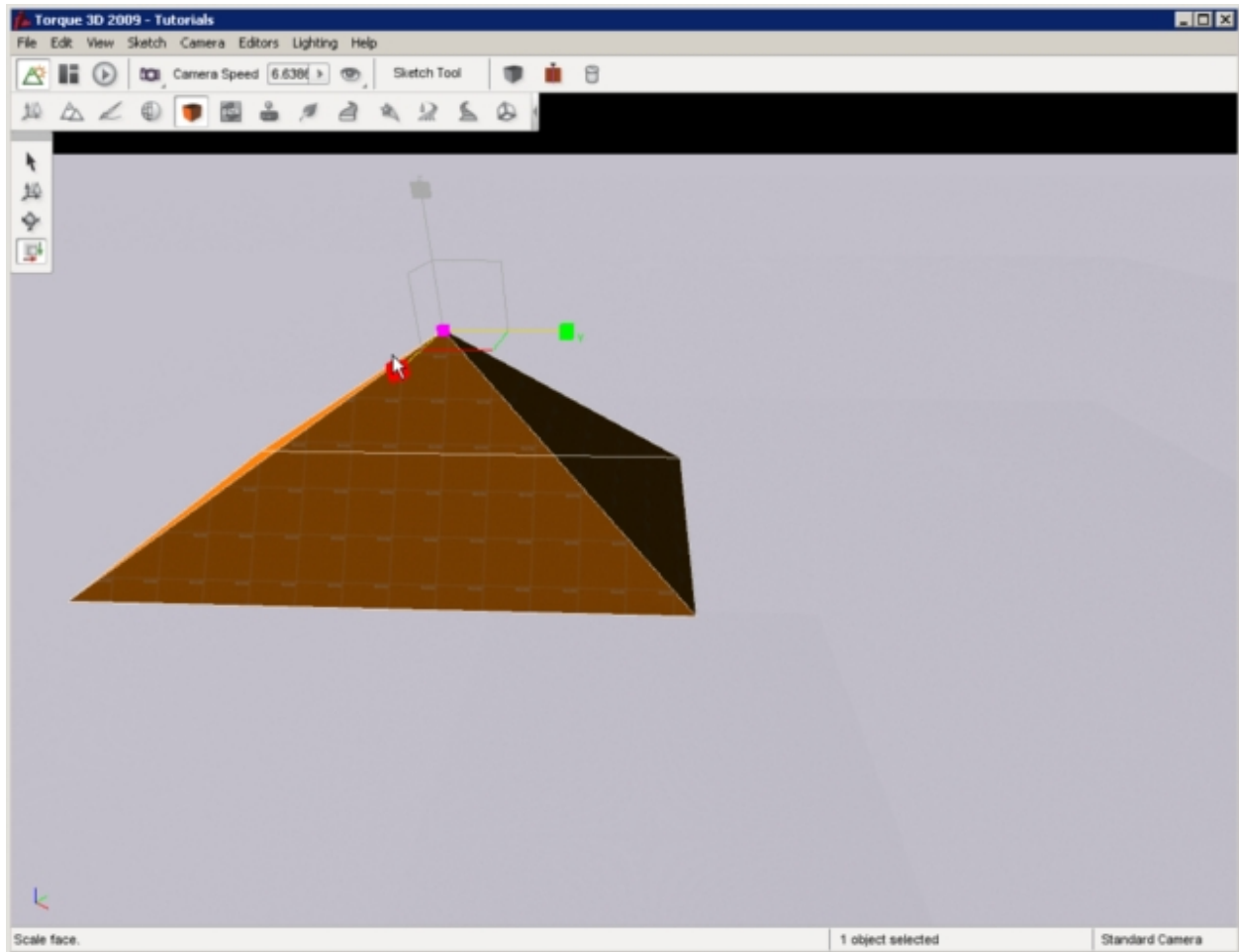


Finally, activate the Scale Selection tool by clicking the icon on the Tools Palette on the left of the screen or pressing the shortcut key 4. Click on the top face of the box. A squared like gizmo will appear which will allow you to choose what parameters to adjust. You can adjust the (width, height, depth, or any combination of the three):

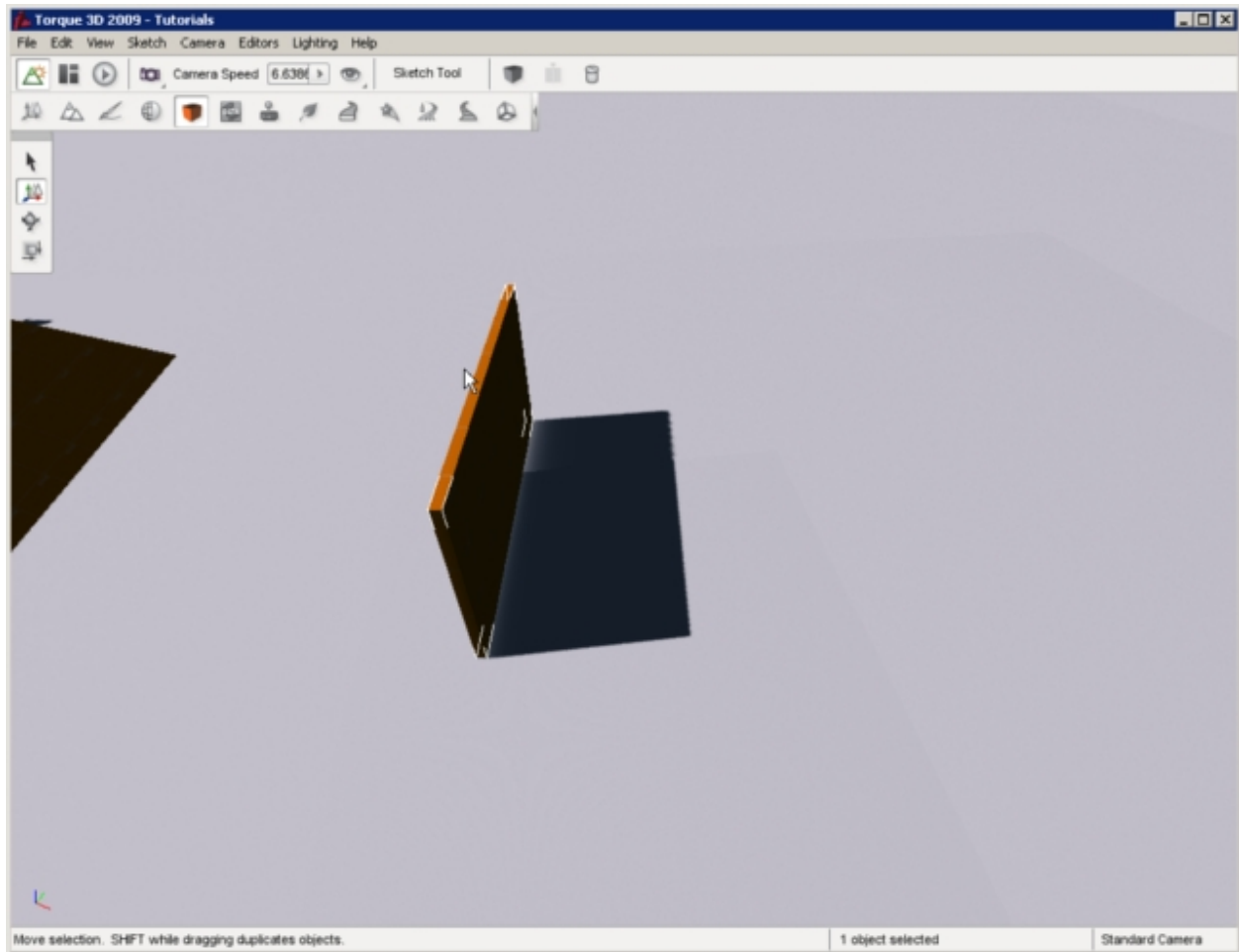




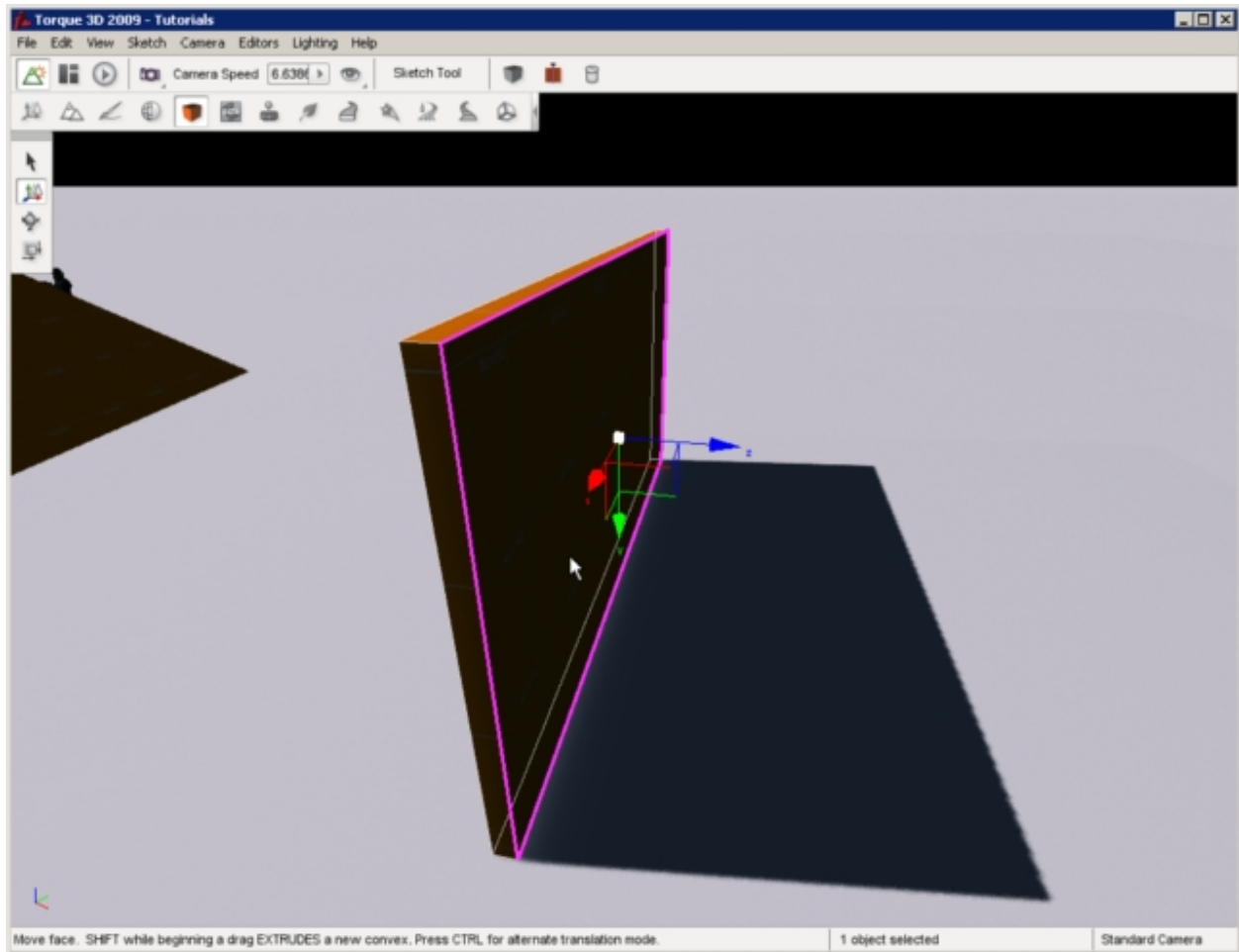
Instead of adjusting one parameter at a time, we are going to adjust width and height. Move your mouse over the different squares to see how they highlight. Click the bottom square of the gizmo, in between the red X and yellow Y axis and hold down the mouse button. Drag your mouse in either direction to shrink or grow the face. The more you shrink, the more like a pyramid it will be come:



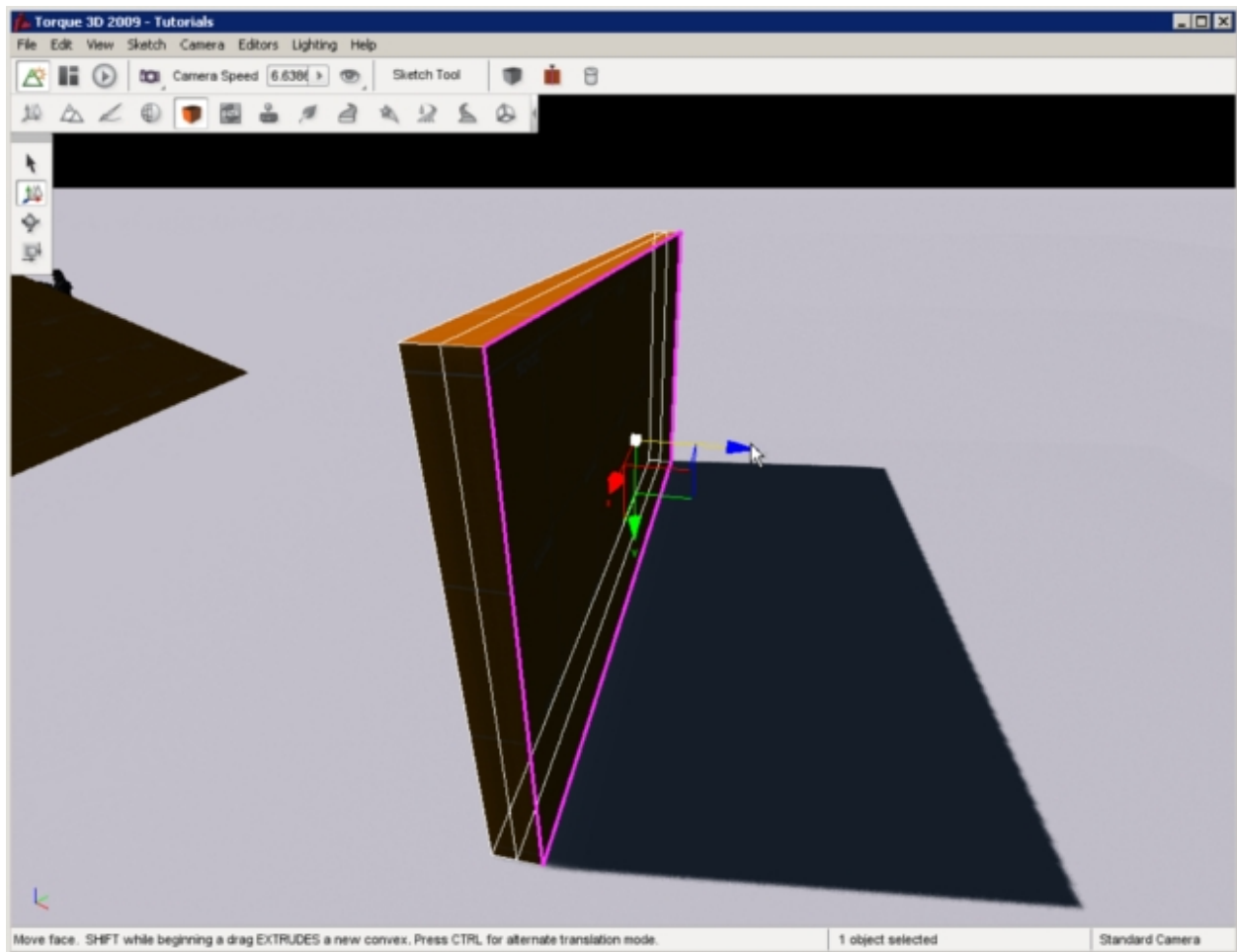
The last action this guide will address is extruding. The Sketch Tool extruding feature creates new geometry from a selected a face. Start by creating a new convex shape - review the section, [Creating a Convex Shape](#), if you don't remember how.



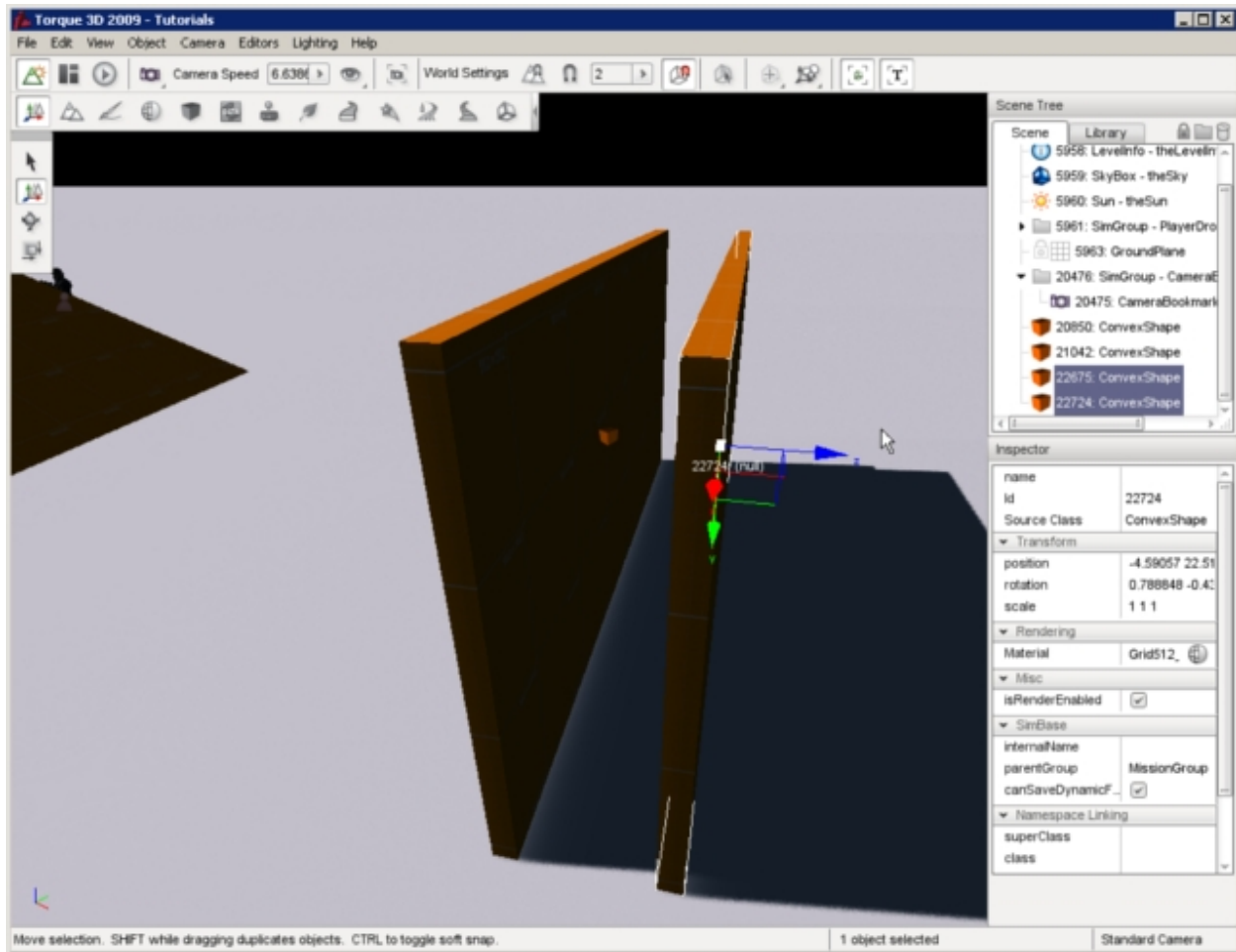
Next click on a single face of the shape. Make sure you have a face selected, and not the entire object. The selected face should be highlighted with a in bright pink. Activate the Move Selection tool. A hint will display at the bottom of the editor: “Move selection. Shift while beginning a drag *extrudes* a new convex.”



Perform this action as described. With the face selected, hold down the `Shift` key, move the mouse over one of the colored arrows, and click and drag outwards from the object. The exact dimensions of the original face will be duplicated, constructing a new convex based on those parameters. This may not be apparent until you click on a face and see that the area of the new face is separate from the original:



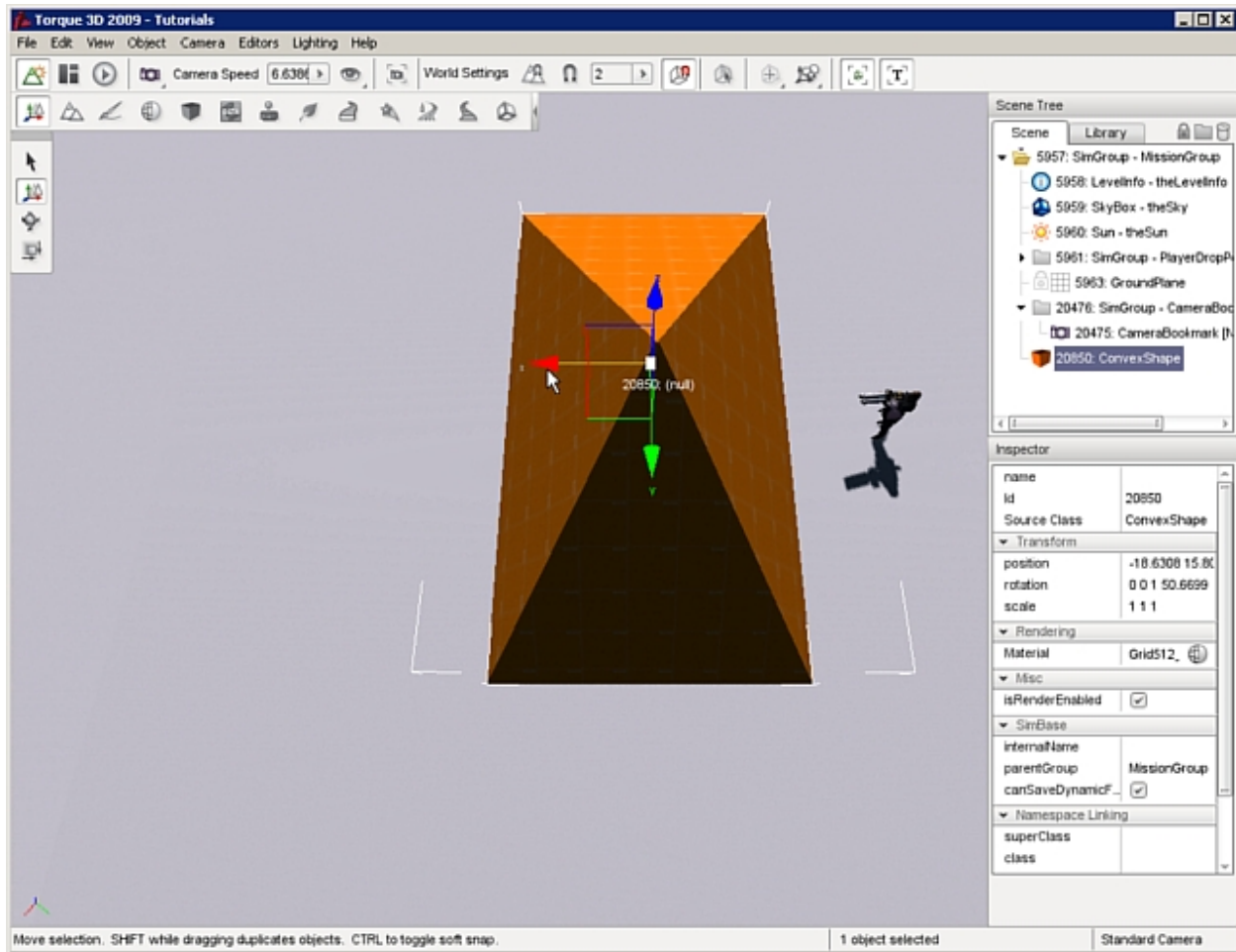
You can now select faces on the new convex object and continue editing it as a new object:



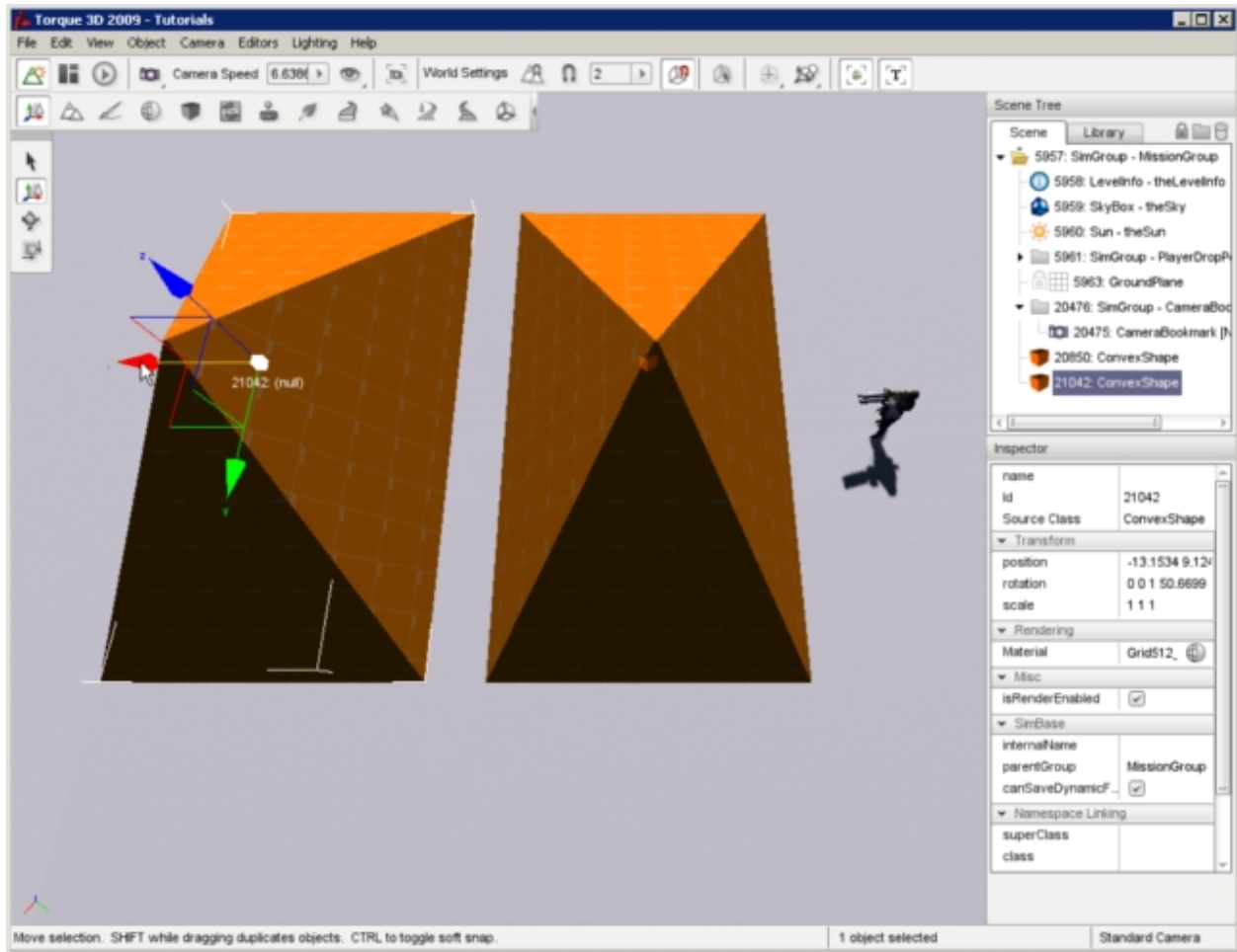
### 13.4.4 Object Manipulation

When you are finished sculpting a convex shape, you can manipulate it as you would with any other game object using the Object Editor. This includes selection, translation/rotation/scaling, and editing specific properties.

Unlike the Sketch Tool, selecting a convex shape using the Object Editor it treats the object as a whole. There is no individual face selection. Switch to the Object Editor by pressing the F1 key then click on one of your objects:



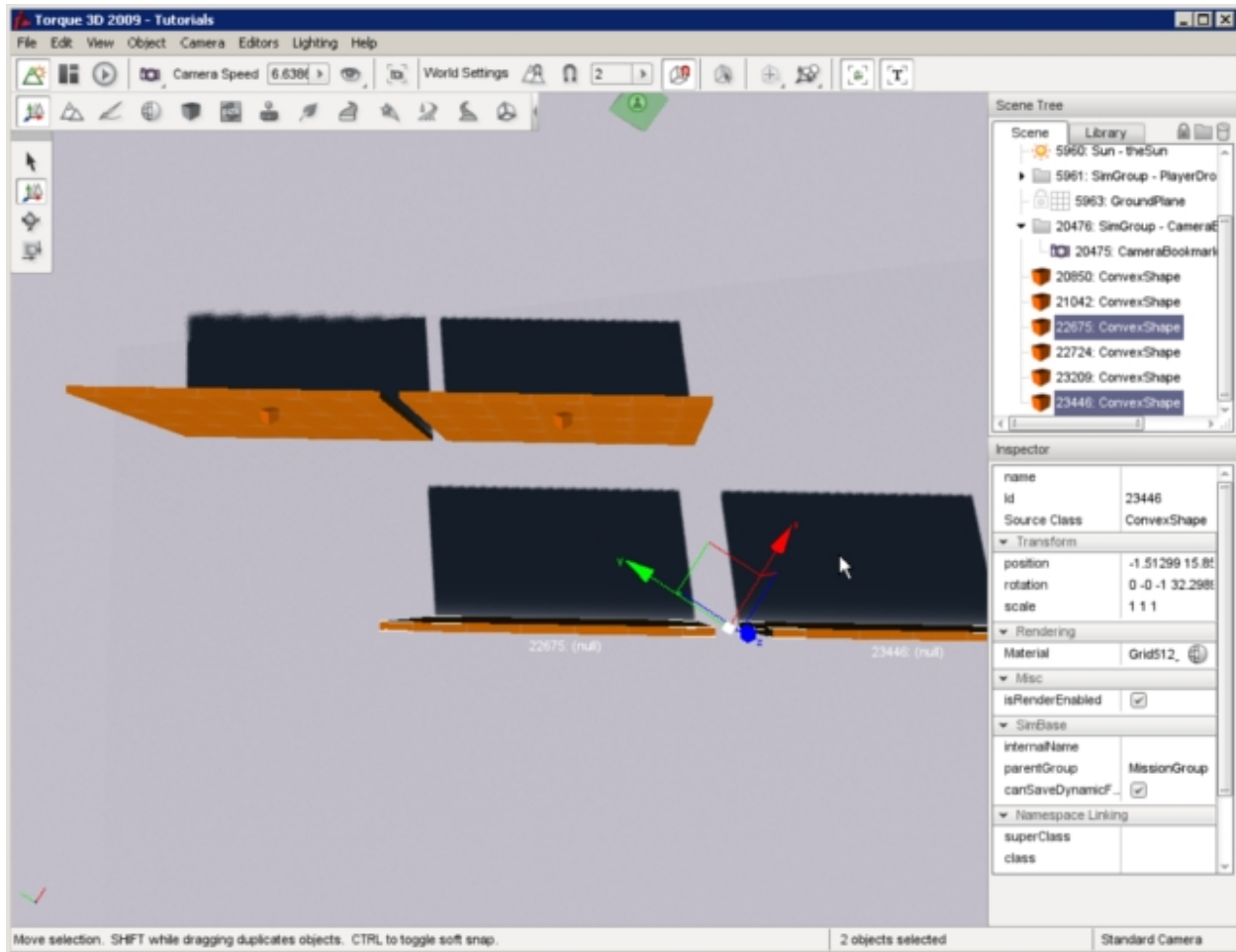
You may then manipulate the object using the normal Move Selection, Rotate Selection, and Scale Selection tools of the Object Editor. You can even use the other more complex Object Editor commands such as copying an object. To copy the object: hold the `Shift` key; activate the Move Selection tool by pressing the `2` hotkey; press and hold the `Shift` key; then drag the mouse to a new position in any direction. When you release the mouse button you will have a new duplicate copy of the original object:



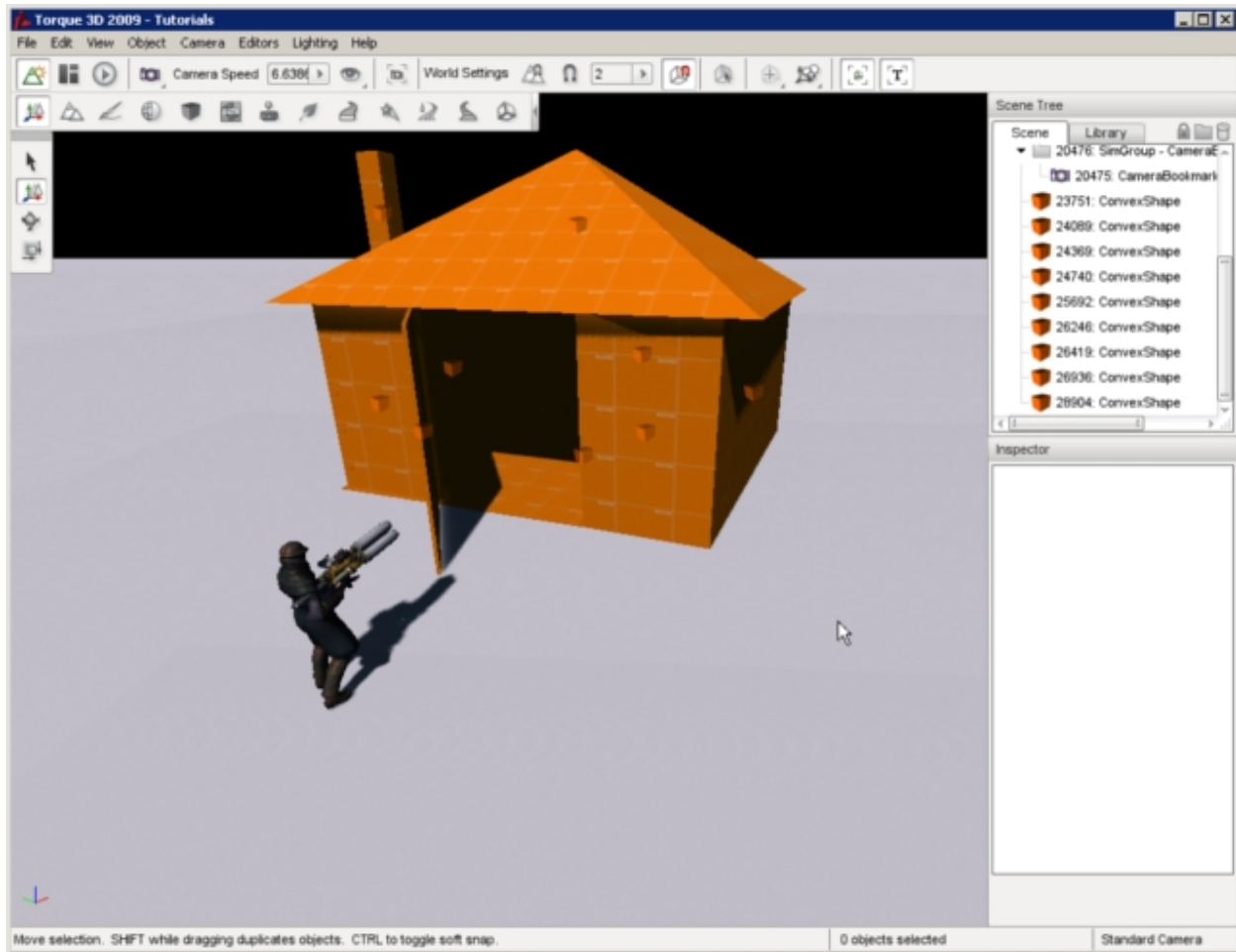
This can also be used for mass production of objects by copying multiple objects at once. Change back to the Select Arrow tool by pressing the 1 hotkey. Click one of your objects to select it. Take note of the position of the gizmo that appears and the little cube at the gizmo's origin. Now select the other object. Again take note of the position of the gizmo and the cube for this object. Now press and hold down the `Shift` key then click your other object again. You will notice there are now two small cubes, one over each object, and one gizmo relatively near the center of the two objects. This indicates that both objects are currently selected. You now have a selection group.

Change to the Move Tool by pressing the 2 hotkey. The cubes will disappear, and large arrows will appear on the ends of the gizmo. If you mouse over either object, you will see a faint transparent cube pop up. This indicates that object is a part of the selection group. Clicking any arrow and dragging the mouse will not move all the objects at once. Likewise, pressing and holding down the `Shift` key, then clicking and drag will duplicate all the objects in the selection group:





The new copy of the objects will now be the current selection and they can be moved as a group or immediately copied again with another **Shift**-drag operation. The above method combined with rearranging the individual objects after copying them is a great way to piece together multiple convex shapes to create more complex arrangements. For example, you might have unique convex objects for a roof, wall, chimney, and so on. You could only create one wall, then duplicate it four times so that they are all the same size then arrange them into a building with the Move Selection tool:



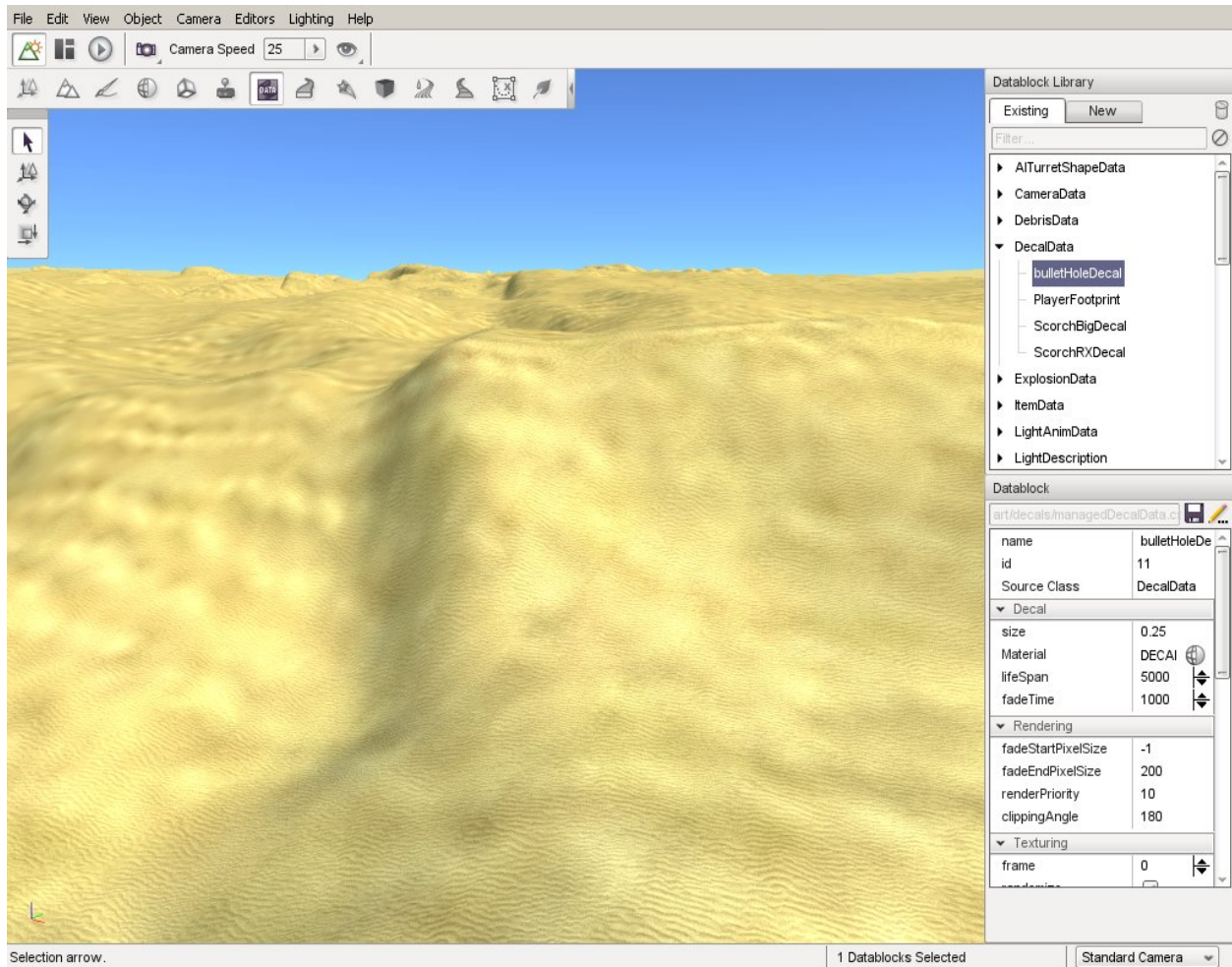
Once you get the hang of the Sketch Tool, you can sculpt unique and complex shapes. Entire levels can be prototyped to use placeholder art, created right inside Torque 3D, while you or your artists work on the final assets using the tools that they are familiar with.

## 13.5 Datablock Editor

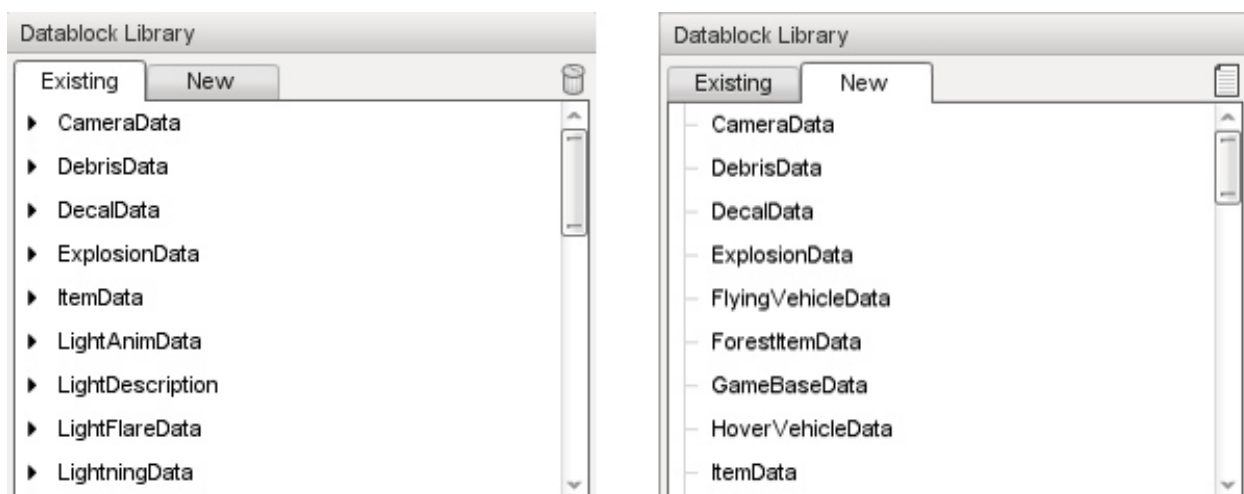
The configuration properties that describe dynamic objects in Torque 3D are stored in information structures called datablocks. The T3D Datablock Editor is used to quickly and easily change any parameter of any datablock from within the world Editor.

### 13.5.1 Interface

To switch to the Datablock Editor press the F6 key or from the main menu select Editors > Datablock Editor. Or alternately click the Datablock icon from the World Editor toolbar.

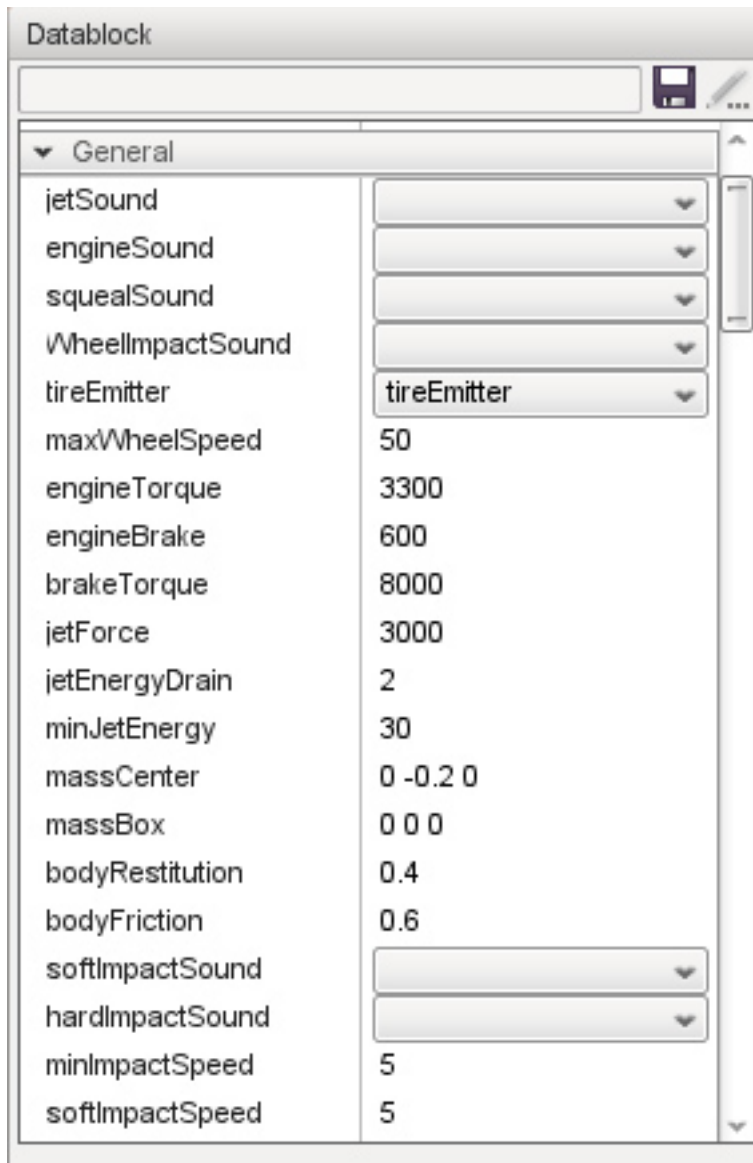


The Datablock editor has two components: the Datablock Library pane and the Datablock properties pane. These panes appear at the right of the screen whenever the Datablock Editor is active. The Datablock Library pane is further divided into two tabs. The first, labelled Existing, contains a categorized list of all the existing datablocks. The second, labelled New, is used to create new instances of those datablocks.



Clicking any existing datablock will cause the Datablock properties pane to update to display the current properties of that datablock.

The image below shows the selection of the DefaultCar datablock, under the WheeledVehicleData category. This datablock contains variables related to vehicle performance.



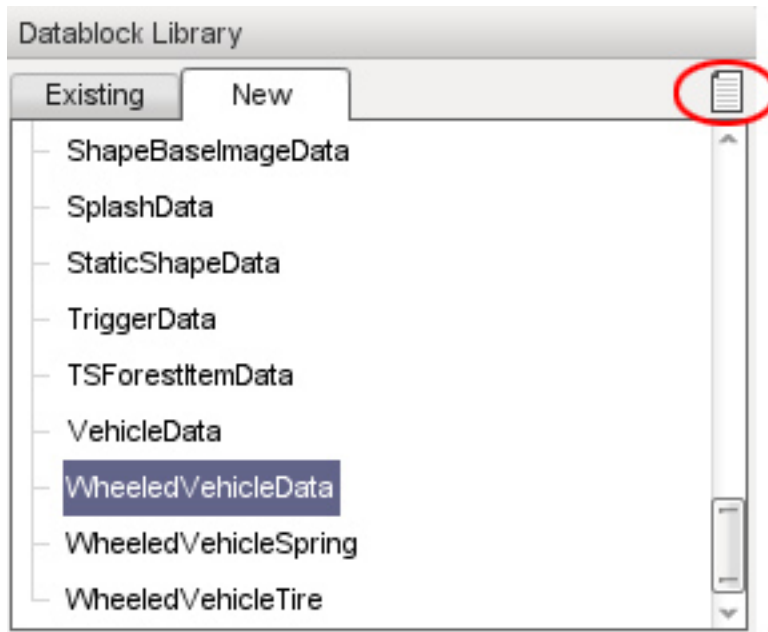
The screenshot shows the 'Datablock' editor window. At the top is a title bar 'Datablock' and a toolbar with a save icon and a pencil icon. Below the toolbar is a tabbed interface with a 'General' tab selected. The main area is a table of variables and their values. The variables are listed on the left, and their values are on the right. Some variables have dropdown menus next to them, indicating they can be selected from a list. The variables and their values are:

Variable	Value
jetSound	[dropdown]
engineSound	[dropdown]
squealSound	[dropdown]
WheelImpactSound	[dropdown]
tireEmitter	tireEmitter [dropdown]
maxWheelSpeed	50
engineTorque	3300
engineBrake	600
brakeTorque	8000
jetForce	3000
jetEnergyDrain	2
minJetEnergy	30
massCenter	0 -0.2 0
massBox	0 0 0
bodyRestitution	0.4
bodyFriction	0.6
softImpactSound	[dropdown]
hardImpactSound	[dropdown]
minImpactSpeed	5
softImpactSpeed	5

### 13.5.2 Creating a new Datablock

Creating a new datablock can be done by creating a copy from an already existing datablock. To do so first select the New tab in the Datablock Library pane.

Then choose the type of datablock you wish to create from the list. Then press the New icon.



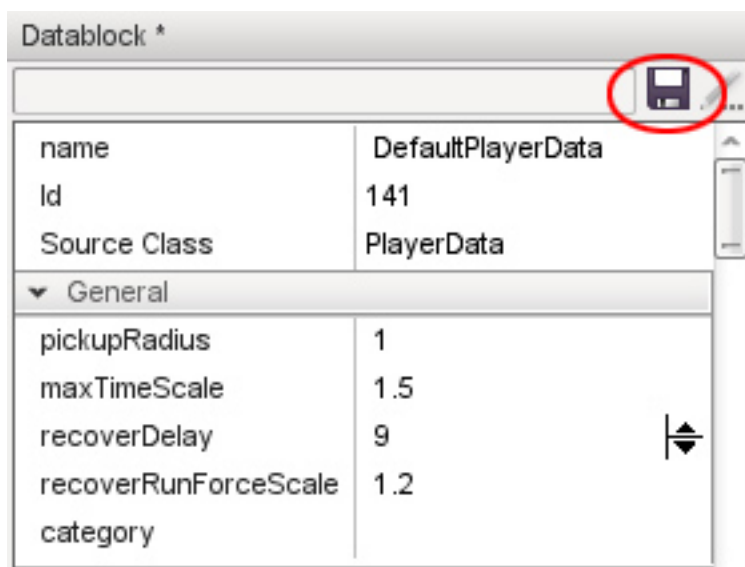
You will be presented with a new window giving you the option to name the new datablock and to copy values from one of the existing instances of the datablock type, if you want to. For example, in this scenario the DefaultCar datablock would be available in the dropdown box because it already exists at the time when creating a new datablock.

After clicking the Create button a new copy of the datablock will be added to the library, under the datablock type you first selected. In this example, you will create a new WheeledVehicleData datablock and name the new version “raceCar”. This new version can now be found in the Library, under the Existing tab, in the WheeledVehicleData section.

### 13.5.3 Saving a Datablock

After editing the new datablock or any other datablock, you will need to save it. You will see a small “\*” in the header of the properties right after the Datablock label if the datablock needs saving.

Click the small floppy disk icon to save your datablock changes.



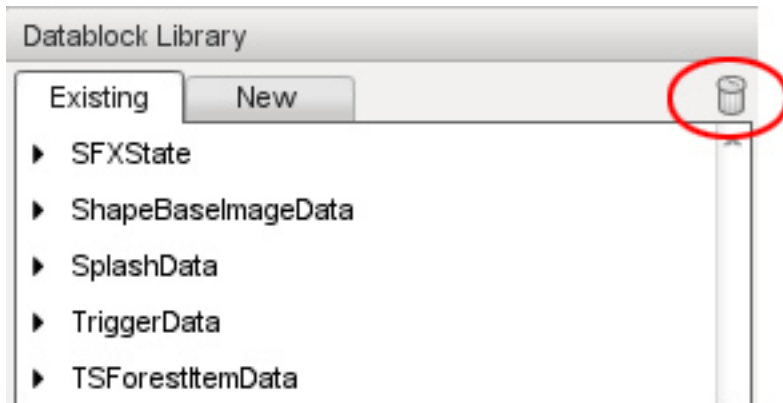


**Note:** Any new datablock which has been saved will be added into the managedDatablocks.cs document which can be found at the location: projectgameartdatablocksfor your scripters to access later.

---

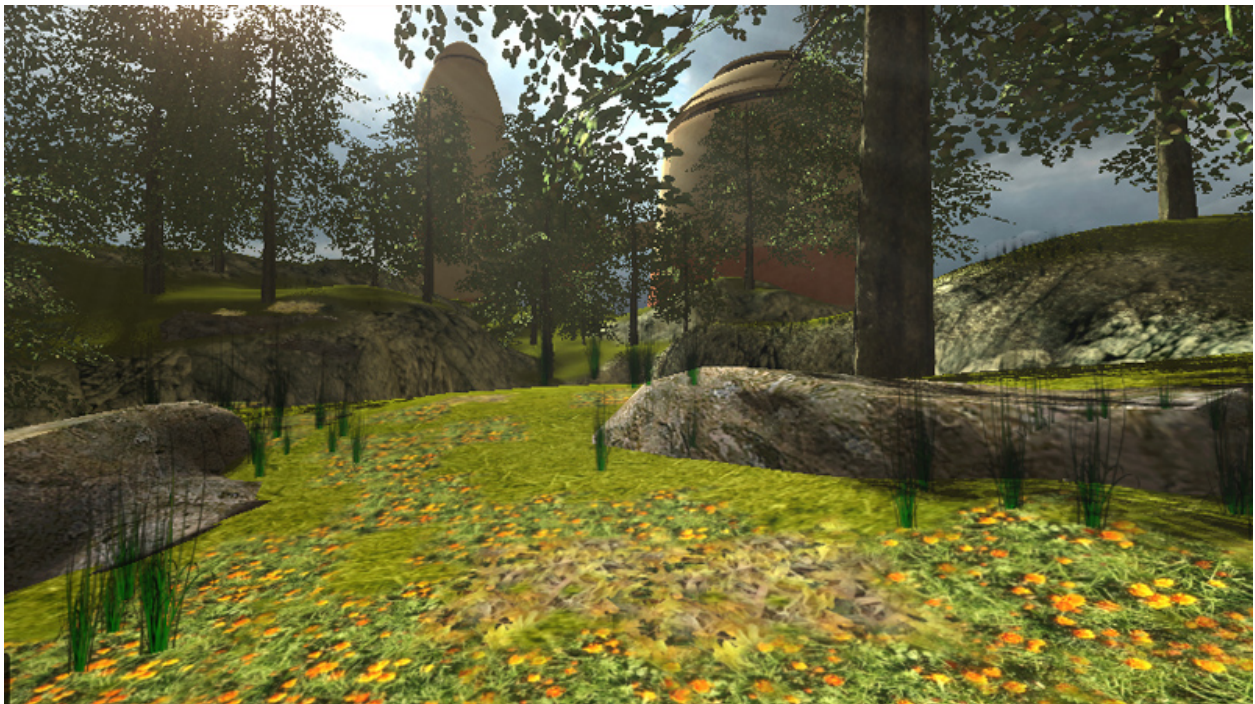
### 13.5.4 Deleting a Datablock

If you no longer need a datablock you can easily delete it by selecting the Delete icon.



After pressing this icon you will get a notification window stating that the datablock has been removed. The World Builder will need to be restarted to completely remove the file.

## 13.6 Decal Editor



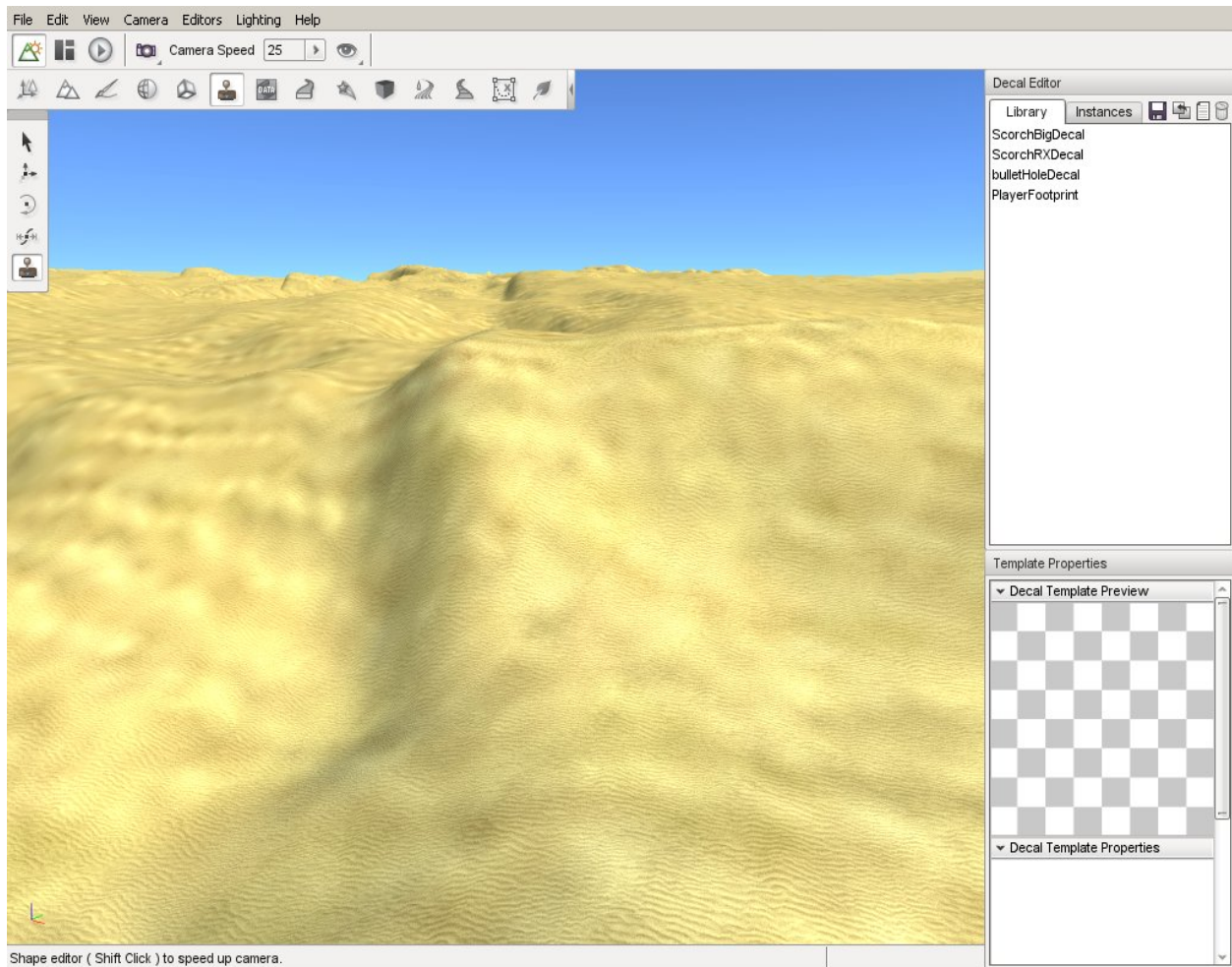
Decals in Torque 3D refer to image textures that are overlaid on objects such as the terrain or players to give the appearance of surface affects such as leaves, flowers, or litter on the ground, or for dynamic changes to the environment,

such as foot prints, or burn marks from explosions without the need to create and place special terrain materials or objects to represent the effects. Torque 3D's Decal Editor provides you with control over decals of any type, including placement and other attributes. Decals can be placed via the editor to any visible surface within the world.

As with the other Torque 3D editors this can be easily achieved by using the built-in WYSIWYG (What-You-See-Is-What-You-Get) editing tools.

### 13.6.1 Interface

To access the Decal Editor press the F7 key or select it from the drop down menu at the top of the World Editor, by choosing Editors > Decal Editor. Or select the Decal icon from the World Editor tool bar.



The Decal editor has two main parts, one where you can add and manipulate the size, rotation and position of the decal, and the other for adjusting the decal properties. On the upper-left hand side of your screen, you'll see a toolbar which provides tools for decal placement and manipulation. On the right of the screen, there is the Decal Editor pane which displays a list of decals, and the Template Properties pane below it, which displays properties of the selected decal along with a texture preview.

In the Decal Editor pane on the top right, under the Library tab, is a list of the decal datablocks defined in the system, which are really decal descriptions. In the Instances tab, there is a list of all decals that have been created within the current project.



### 13.6.2 Adding a New Decal Datablock to the Library

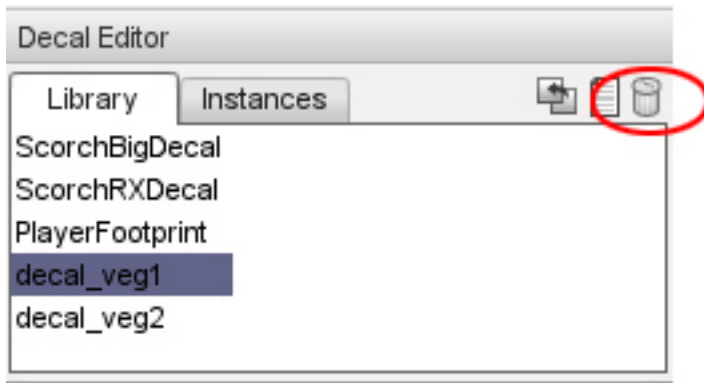
Before you can paint a decal to your world, the datablock needs to exist in the decal library. To add a new decal, simply press the New Decal icon at the top of the Decal Editor pane. This will add a new blank decal to the library, ready for you to select or add a material and set up its properties.

### 13.6.3 Naming a New Decal Datablock

After creating a new decal you will want to name it. This can be achieved by selecting the name property and entering a new name, then pressing the Enter key.

### 13.6.4 Removing Decal Datablock from the Library

To remove an existing decal from the library simply select the decal in the list then click the Delete icon.

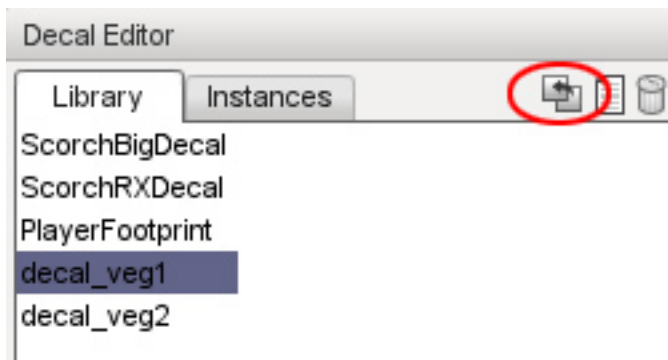


If you select Yes all instances of the selected decal will be removed. The datablock will continue to exist until the World Editor is restarted.

### 13.6.5 Missing Decals

If a decal exists in the level, but is not rendering, it means the datablock for it has either been deleted, renamed, or corrupted. The Retarget button allows you to assign an existing datablock to a missing decal.

To open the retarget dialogue, select the decal that you wish to fix then click the Retarget icon at the top of the Decal Library pane.



The Retarget Decal Instances dialog box will open. From here you can reassign the decal datablock. Select a Decal Datablock from the drop down list that you wish to use for the selected decal.

### 13.6.6 Editing Tools

The Decal Editor placement and manipulation tools appear on the toolbar down the left of your screen whenever the Decal Editor is active. Each tool can be activated by clicking the appropriate icons or by pressing its hotkey. Hotkeys are assigned 1 through 5 from top to bottom in the toolbar and are visible by hovering the mouse over the icon. These tools will enable you to quickly place, scale and rotate your decals.

#### Adding a Decal

Before adding a decal, a datablock for the decal must exist in the Decal Library. Please see the section “Adding a New Decal Datablock to the Library” above.

To add a decal to a level select a decal from the Library tab of the Decal editor pane, click the Add Decal tool, and then click on the terrain where you would like to see your decal instance appear. The same decal can be placed in a level multiple times. Each such copy is referred to as an instance of the decal.

#### Selecting a Decal

The selection tool enables you to directly select a decal instance that has already been placed in the level by simply clicking on it. Its datablock properties will then be shown in the Template Properties pane for viewing and/or editing.

#### Deleting a Decal Instance

There will be times when you need to delete an a decal instance. The decal can be selected by the Selection tool (see the “Selecting a Decal” section) and then pressing the Delete key. Or you can select the required decal from the Instances tab in the Decal Editor pane, then press the Delete key or press the Delete button, represented by the trash can icon.

---

**Note:** This will only delete the selected decal instance in the world, not the decal’s datablock listed on the Library tab.

---

#### Moving a Decal

To move a decal instance simply select the decal using any method described above then click the Move tool icon. The normal Object Editor movement gizmo will appear. Click any axis arrow using the left mouse button, then hold the button down and drag the mouse to move the decal in that direction. Release the mouse button to drop the decal at the new location.

#### Scaling a Decal

If you find that the decal is either too small or too large you can use the Scale tool to resize the decal. This uses the standard world gizmo, but will not scale on the vertical axis due to decals being restricted to two-dimensions. Click any axis cube using the left mouse button, then hold the button down and drag the mouse to scale the decal in that direction. Release the mouse button to leave the decal at the new scale.

### Rotating a Decal

If for any reason you find that you need to rotate a decal, you can use the Rotate Tool do so. To rotate a decal, select the decal by any method described above, and then click the Rotate Tool icon. The standard world rotational gizmo will appear. Click any rotation circle using the left mouse button, then hold the button down and drag the mouse to rotate the decal in that direction. Release the mouse button to leave the decal at the new location.

---

**Note:** Because decals are two-dimensional rotating a decal will never cause the decal to leave the surface upon which it has been placed. Rather, when a decal is rotated by any axis the decal will rotate in two-dimensions locked to that surface. This can cause some strange effects if the surface that contains a decal is curved with a radius less than the size of the decal. As with all the other T3D editors, the more that you experiment and use the tools the more familiar you will be with them. With practice and time you will find many uses for the Torque 3D decal system.

---

### 13.6.7 Properties

A Decal has only a small amount of properties which can be edited using the Template Properties pane:

**Size** The size of the decal rendered onto the surface.

**Material** Specifies the Material selected to display as the decal.

**Lifespan** Time in Ms (milliseconds) that the decal will exist in the world after being placed dynamically.

**FadeTime** Time for the decal to fade out in Ms (milliseconds).

**Frame** Index of texture rectangle to use for this decal, if the texture consists of multiple images.

**Randomize** Randomizes the texture rectangle (frame) used for each instance of the decal. So it essentially uses a random frame.

**TexRows** Defines the number of image rows in a multiple image material.

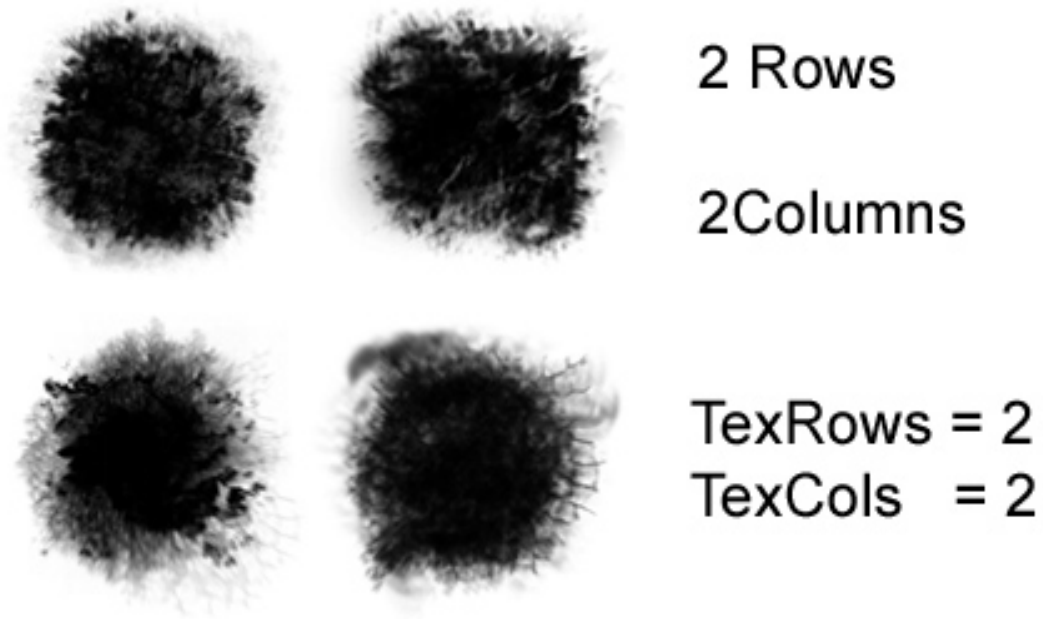
**TexCols** Defines the number of image columns in a multiple image material.

**ScreenStartRadius** Distance check for rendering the alpha channel of the decal. Visibility check based on the scale of the decal

**ScreenEndRadius** Distance check for rendering the alpha channel of the decal. Visibility check based on the scale of the decal

**Render Priority** If more than one decal are on top of each other the decal with the higher priority will rendered first

**Clipping Angle** The angle in degrees used to display geometry that faces away from the decal projection direction.



## 13.7 Forest Editor



The Forest Editor is a tool that allows you to quickly create massive amounts of vegetation for your level including patches of trees, forests, and fields of smaller elements such as shrubs and plants. Entire forests can be laid down using simple techniques similar to painting on a canvas, where instead of paint your brushes, lay down 3D models on the terrain.

### 13.7.1 Interface

To access the Forest Editor press the F8 key, or select it from the drop down menu at the top of the World Editor, by choosing Editors > Forest Editor, or click on the leaf icon to get started.

The Tools Palette on the left of the screen will populate with Forest Editor specific tool buttons represented by icons.

**Select Item** Select an individual object in forest

**Translate Item** Move the currently selected object

**Rotate Item** Rotate the currently selected object

**Scale Item** Grow or shrink the currently selected object

**Paint** Used for painting objects on terrain

**Erase** Used for erasing objects from a terrain

**Erase Selected** Used to delete the currently selected objects

The Forest Editor has two main panels which will appear on the right of the screen whenever the Forest Editor is active.. On the top is the Forest Editor pane which is divided into two tabs: Brushes and Meshes. The Forest Editor works in a manner similar to painting on a canvas with a brush, except instead of paint the Forest Editor lays down shapes onto the terrain of your level. A Brush in the Torque 3D Forest Editor is composed of one or more mesh elements, which can be alternated between when painting.

The Meshes tab contains a list of all Forest Mesh elements which can be assigned to a brush. A forest mesh is really a datablock which is an information structure that defines a model and the properties which control it in the forest.

On the bottom of the right side of the screen is the Properties pane. The Properties pane displays information about the currently selected element in active tab of the Forest Editor pane.

Before we can use these tools and paint a forest, we need to import a forest mesh and set up a brush.

### 13.7.2 Creating a Forest Mesh

To create a forest mesh start by clicking on the Meshes tab in the Forest Editor pane. There are two icons in the top right. The trash bin deletes the currently selected existing mesh, and the leaf icon will add a new mesh. Click on the Add New Mesh icon.

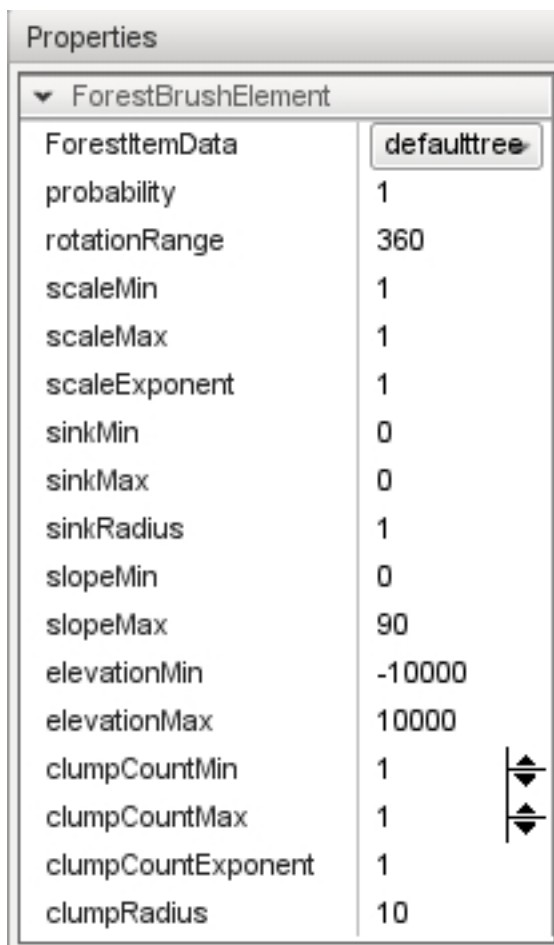


A file browser should appear. Locate the sample tree mesh file, defaulttree.DAE, which can be located in the game/art/shapes/trees/defaulttree folder.

A new mesh will be added to the tab using the same name as the file you selected:



The Properties pane will also be populated with fields and values which describe the new mesh.

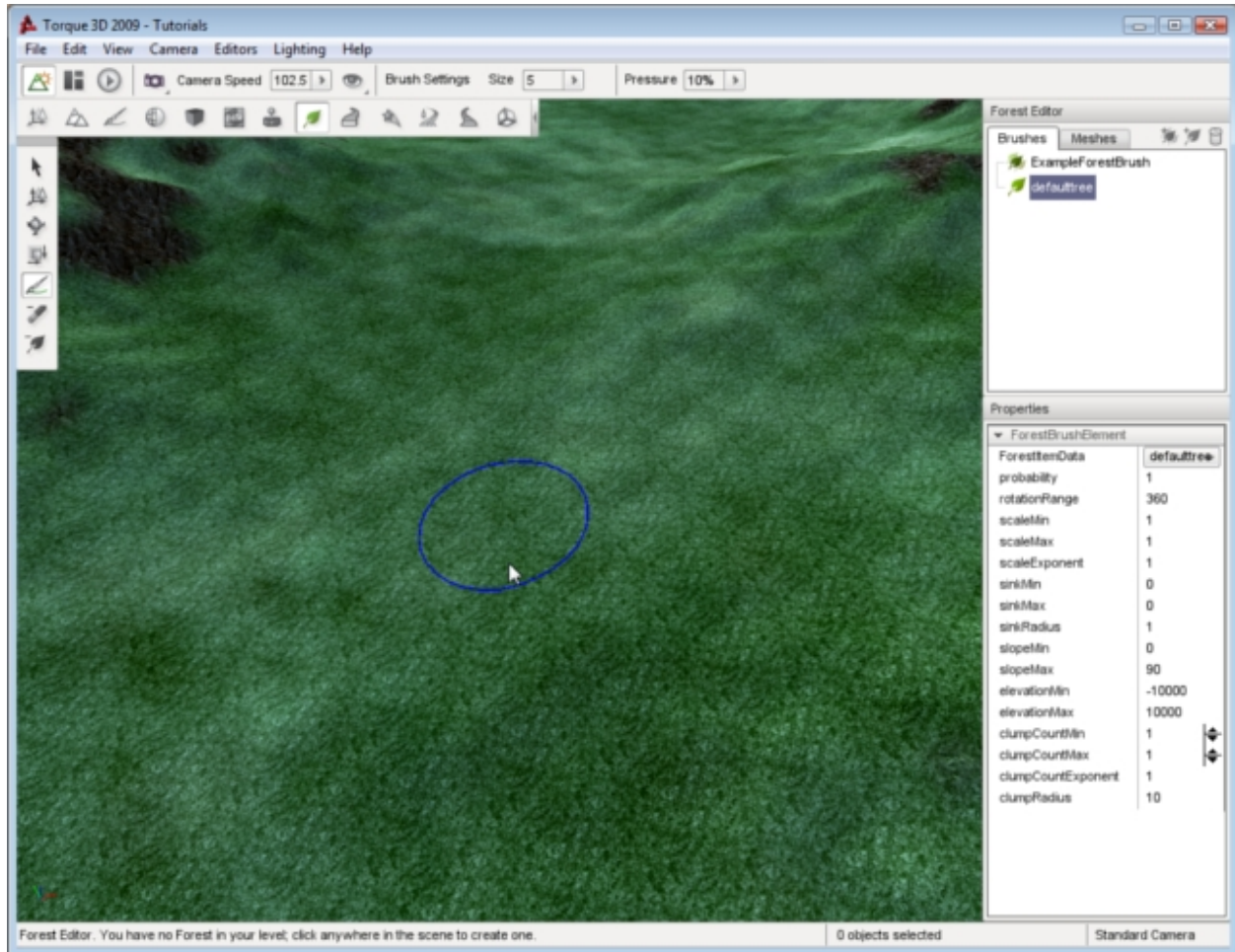




Switch to the Brushed Tab. You will see that the new mesh has also automatically been added to the list of Brushesallow you to select it as the element to paint with. Select the new brush by clicking on its name. The Properties pane will be updated to display the properties of the brush which can be used to randomize the placement and appearance of the selected mesh.

### 13.7.3 Using a Brush

Now that you have an available brush you can begin painting a forest. Select the defaulttree brush from the sample assets. Move the mouse until a blue circle appears on the terrain. This is the outline of your forest brush and shows where you are going paint. To begin painting, left click the mouse and drag it around on the terrain.



If this is the first time you have painted a forest in a level then no Forest object exists in the level yet. However, a forest object must exist, so you will be prompted to confirm that you want to add one.

Answering No will abort the forest painting operation. Answering Yes will automatically create a new Forest object, add it to your level, and return you to the level with the brush still active ready to continue painting. You can examine the forest object the same way as any other object using the Object Editor but it has no useful properties to edit so let's skip it for now and continue on with our forest painting operation.

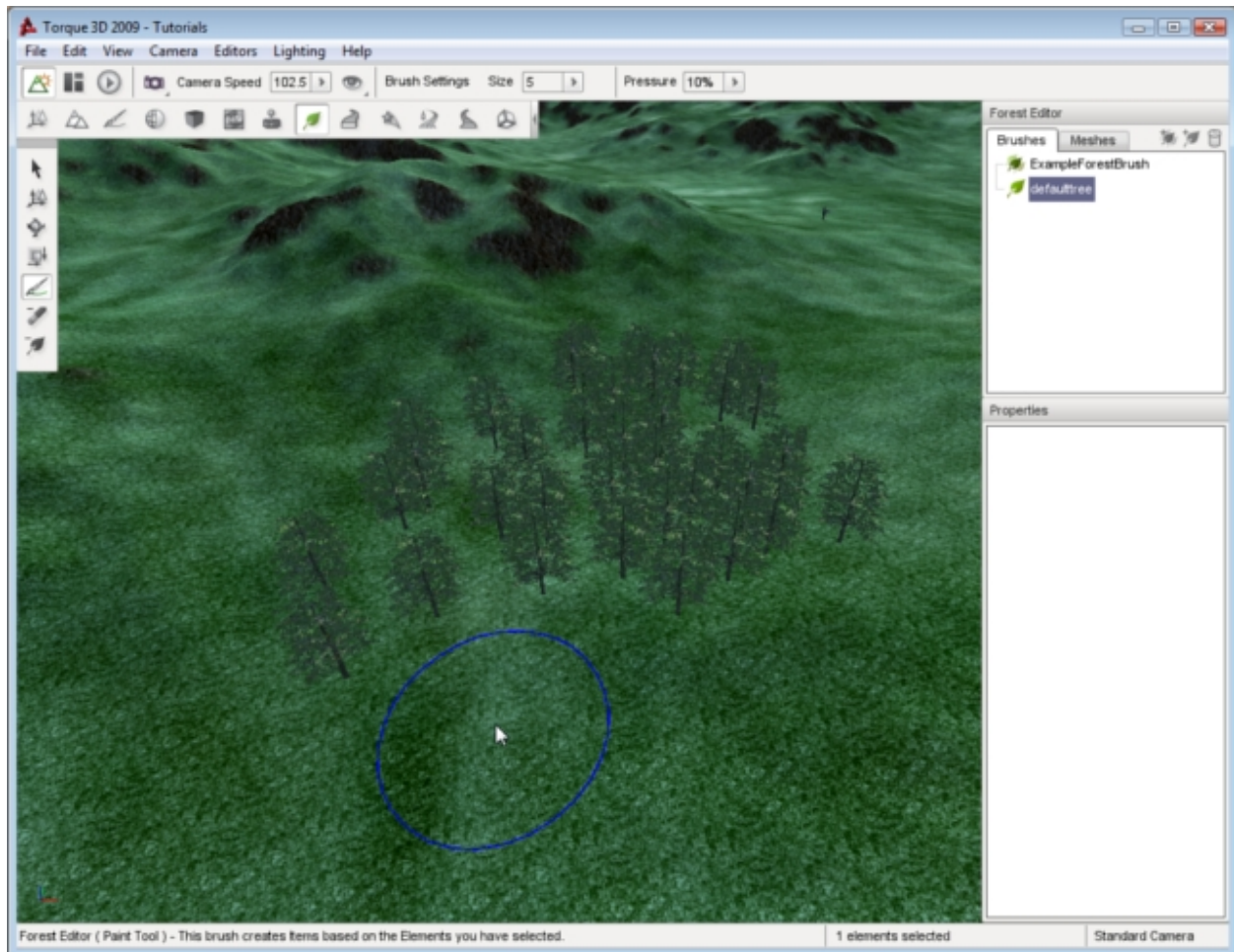
Once again, hold down the left mouse button and drag the mouse over terrain. As you move the brush trees will begin showing up in the wake of your brush. To change the size of your brush, pull the mouse wheel toward you to increase the size or push it away to decrease the size. The blue circle will grow or shrink to indicate your new size.

Note that you do not have the ability to move the camera forward and back in the Forest Editor because of the

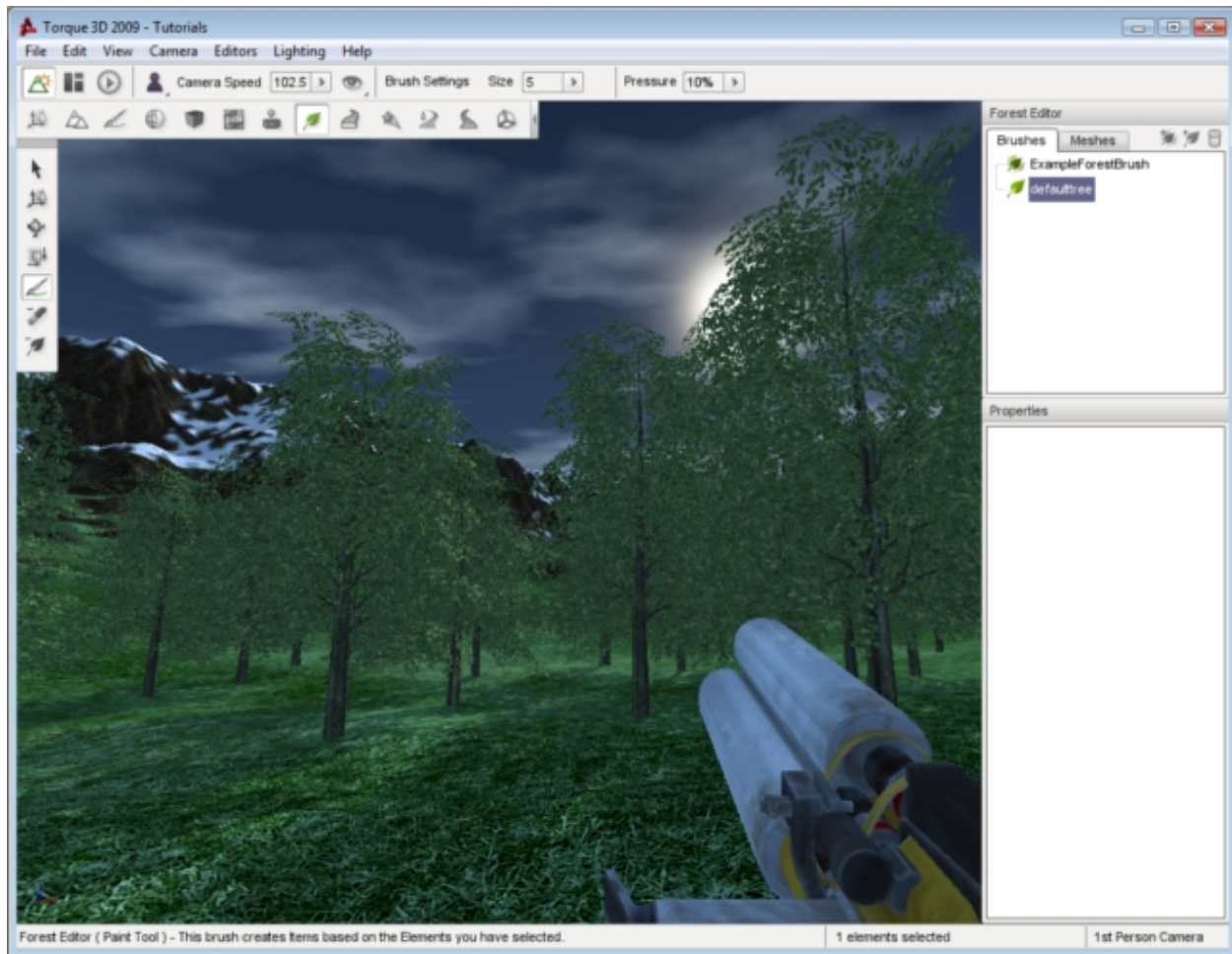


availability of the brush resizing feature. To move the camera forward and back while using the Forest Editor press the Up Arrow to move forward and the Down arrow to move backward.

Keep painting until you have a decent patch of trees:



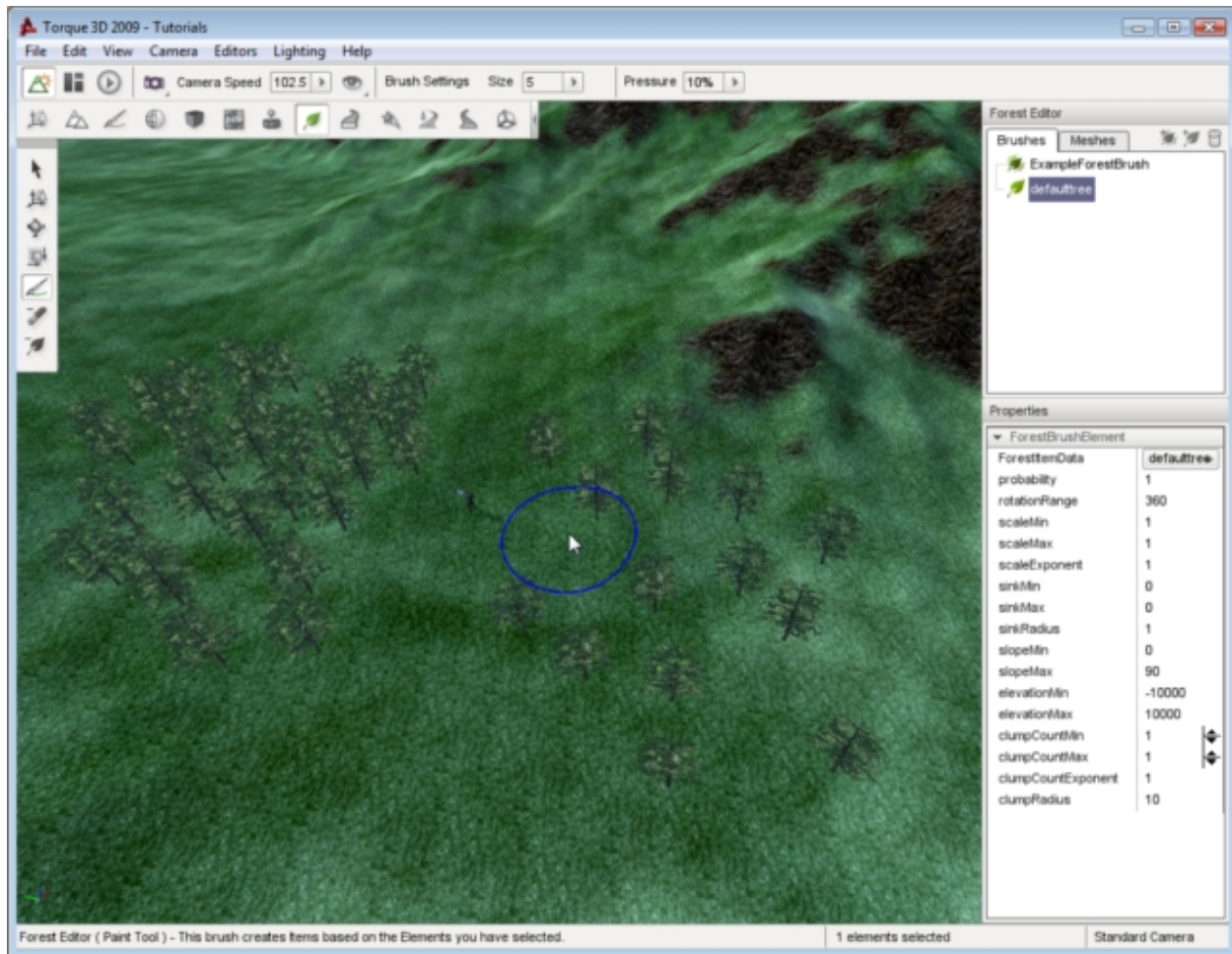
If you move your camera down to ground level, you can see how your forest will look from a player perspective. You'll notice that these are full 3D objects that react to collision, sunlight, and external forces.



### 13.7.4 Adjusting Properties

You can edit the properties of a Mesh to adjust how each tree is placed when painting. To adjust the density of mesh placement switch to the Meshes tab then select your defaulttree entry. The Properties pane will update to display the properties of your mesh. Change the radius property from 1 to 2 then press the Enter key.

This radius tells the tool a rough amount of space this item takes up. The value is a decimal value and has no limits, but remember that if the value is too low your trees may overlap, and if it is too high you may not get any trees to appear because the spacing might be larger than the brush itself. Now when you paint you should get more spacing between the placed meshes.



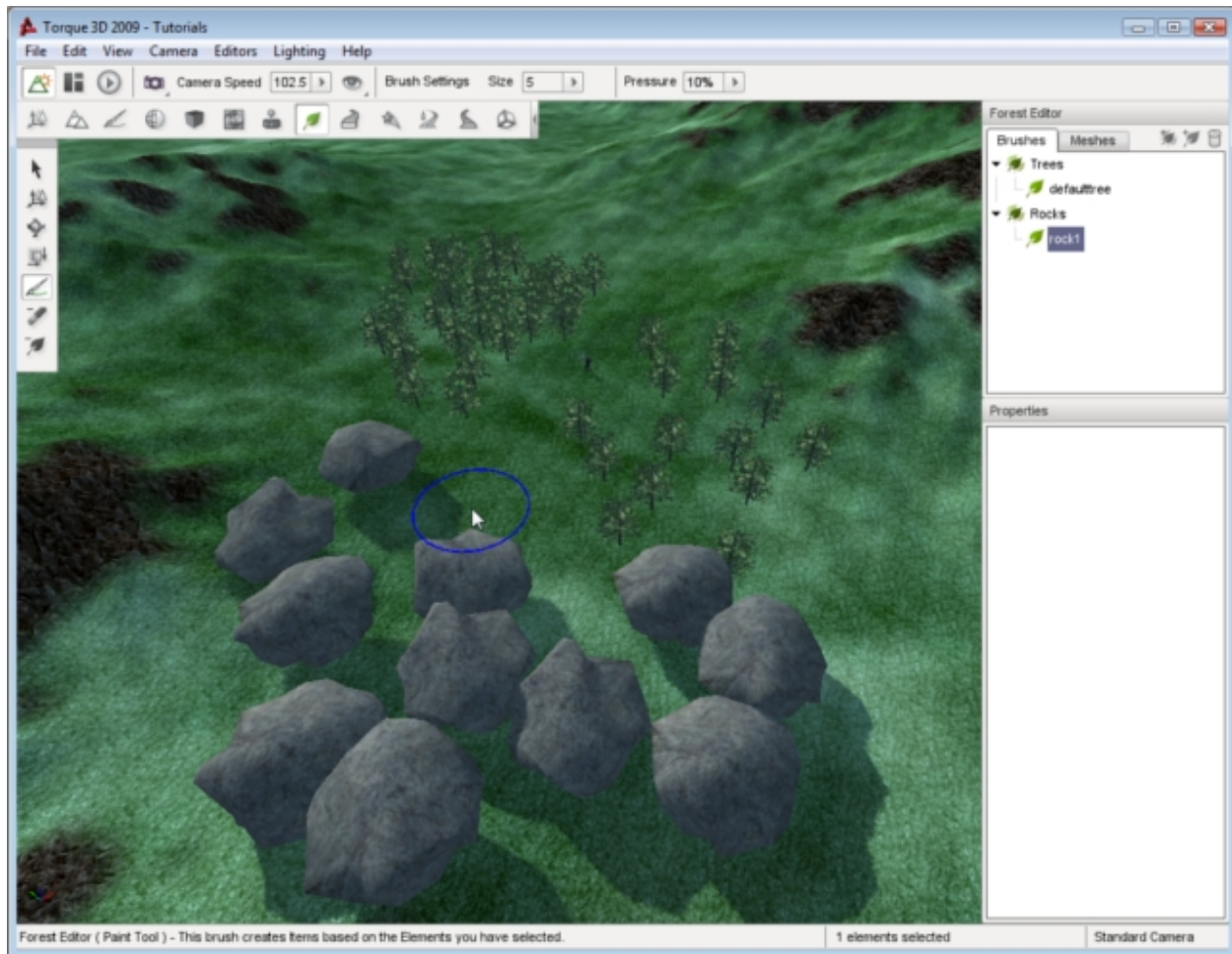
As mentioned previously, you can use the Forest Editor to paint additional environmental objects such as rocks, shrubs, or any other 3D model. Since you can paint different types of objects, you might want to organize your brushes and meshes.

In the Brushes tab, click on the Add New Brush Group icon. This will add a new entry in the brush list, called “Brush”. Click on the text of the new brush group. This will allow you to edit the text of the brush. Name the brush group “Trees” then press the enter key. Now, you can click on the defaulttree element and drag it onto the Trees brush group. Switch to the Meshes tab, and click the Add New Mesh tree icon to add a new one. Select game/art/shapes/rocks/rock1.dts as your model.

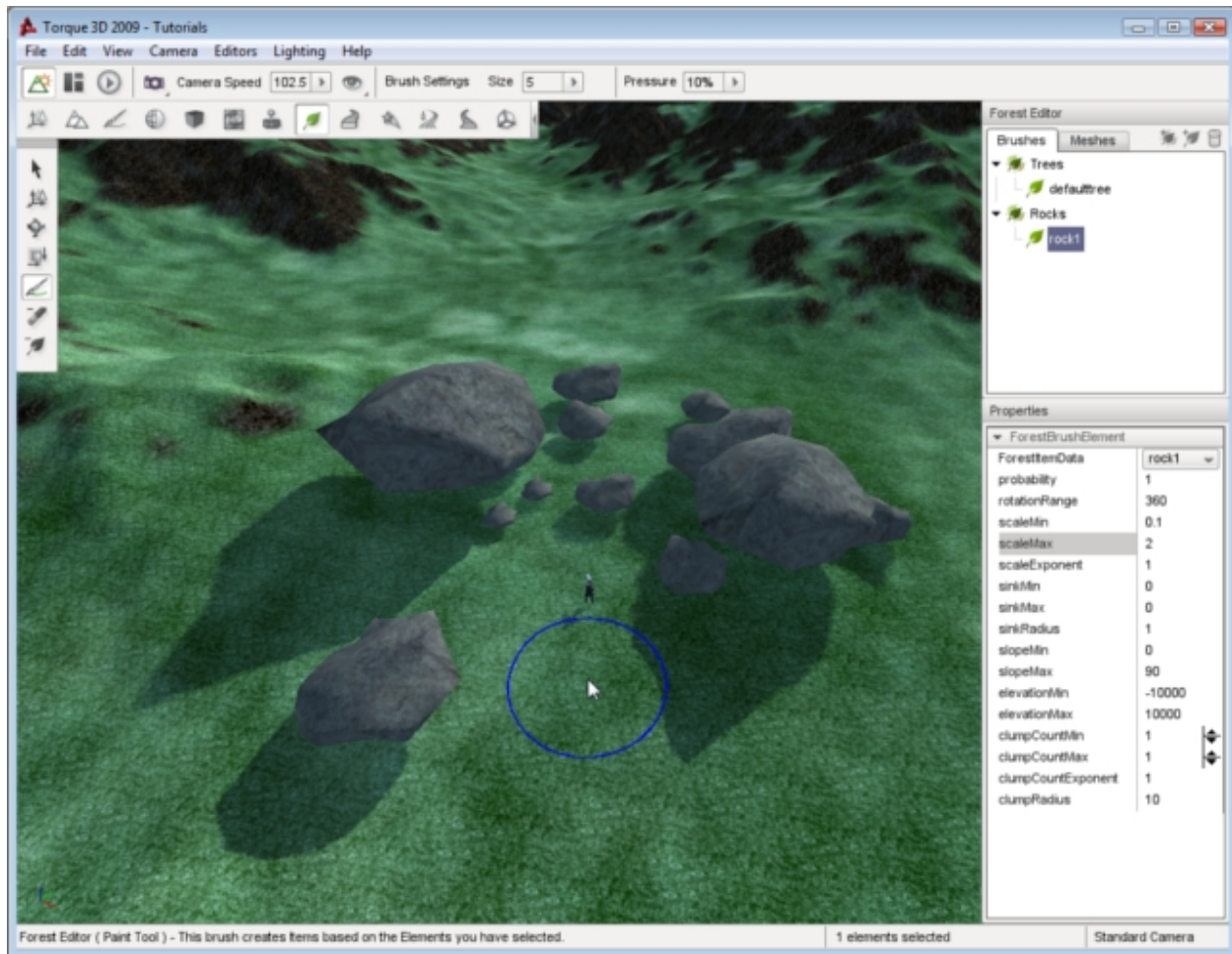
The rock1 mesh will be added to your Meshes list. Unlike trees, the rock1 mesh is fairly large and somewhat spherical. Spreading out the placement of this mesh will help prevent dense blobs of rocks being placed. In the Mesh properties tab, increase the rock1 radius to 3.

Switch back to the Brushes tab. Create a new brush group and name it Rocks. Your rock1 mesh element should already be in the list, so drag it onto the Rocks brush group to keep things organized. Go ahead and paint down some rocks in your level. You should end up with a patch of huge boulders with fairly even spacing:



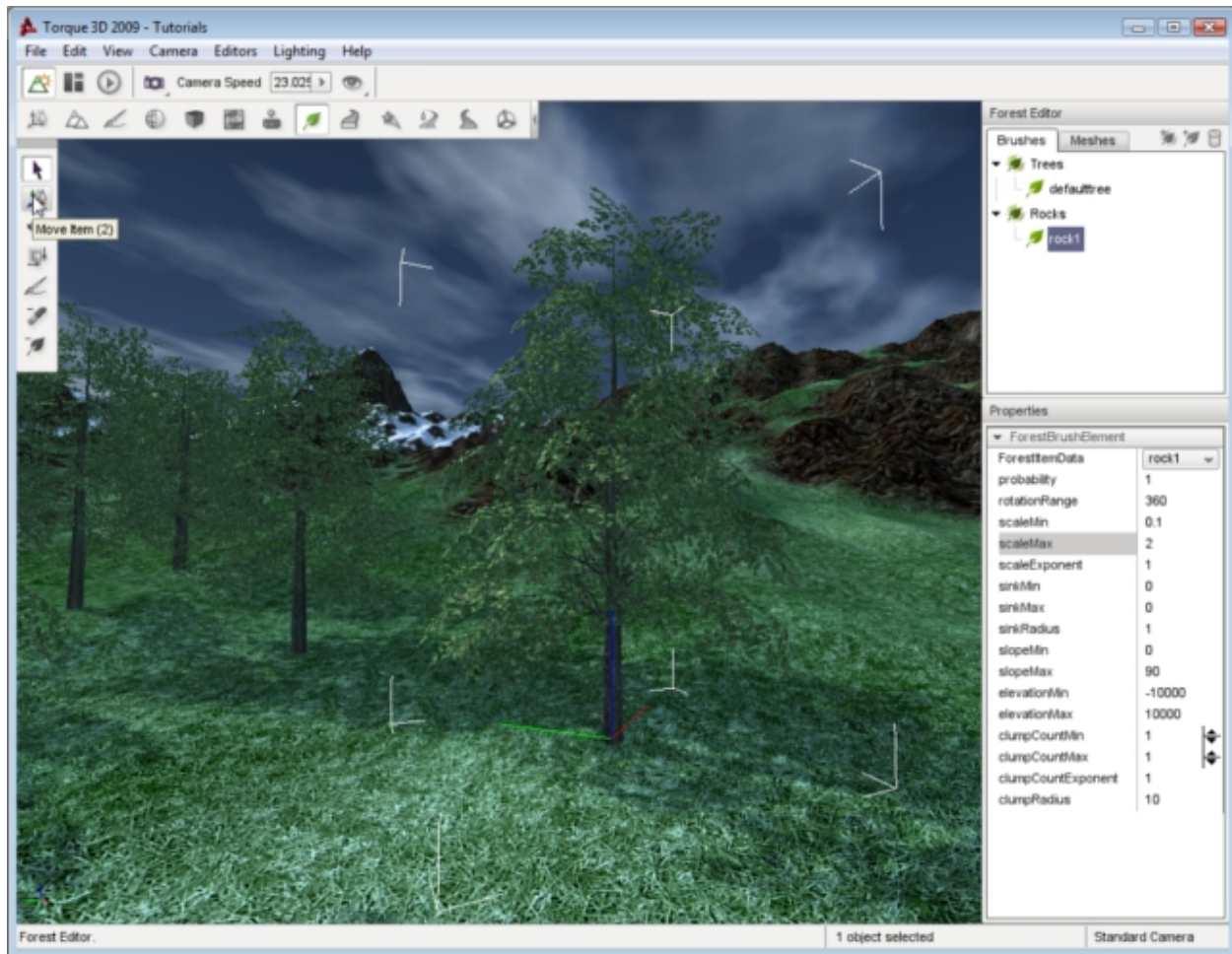


You might have noticed all the boulders are the same size. For added realism, you can adjust the brush properties to randomize its appearance. Select `rock1`, then decrease the `scaleMin` and increase the `scaleMax`. Begin painting a new set of rocks. Now, you will end up with rocks of varying sizes. Some will be as small as your player, while others could be twice the size of the original mesh.



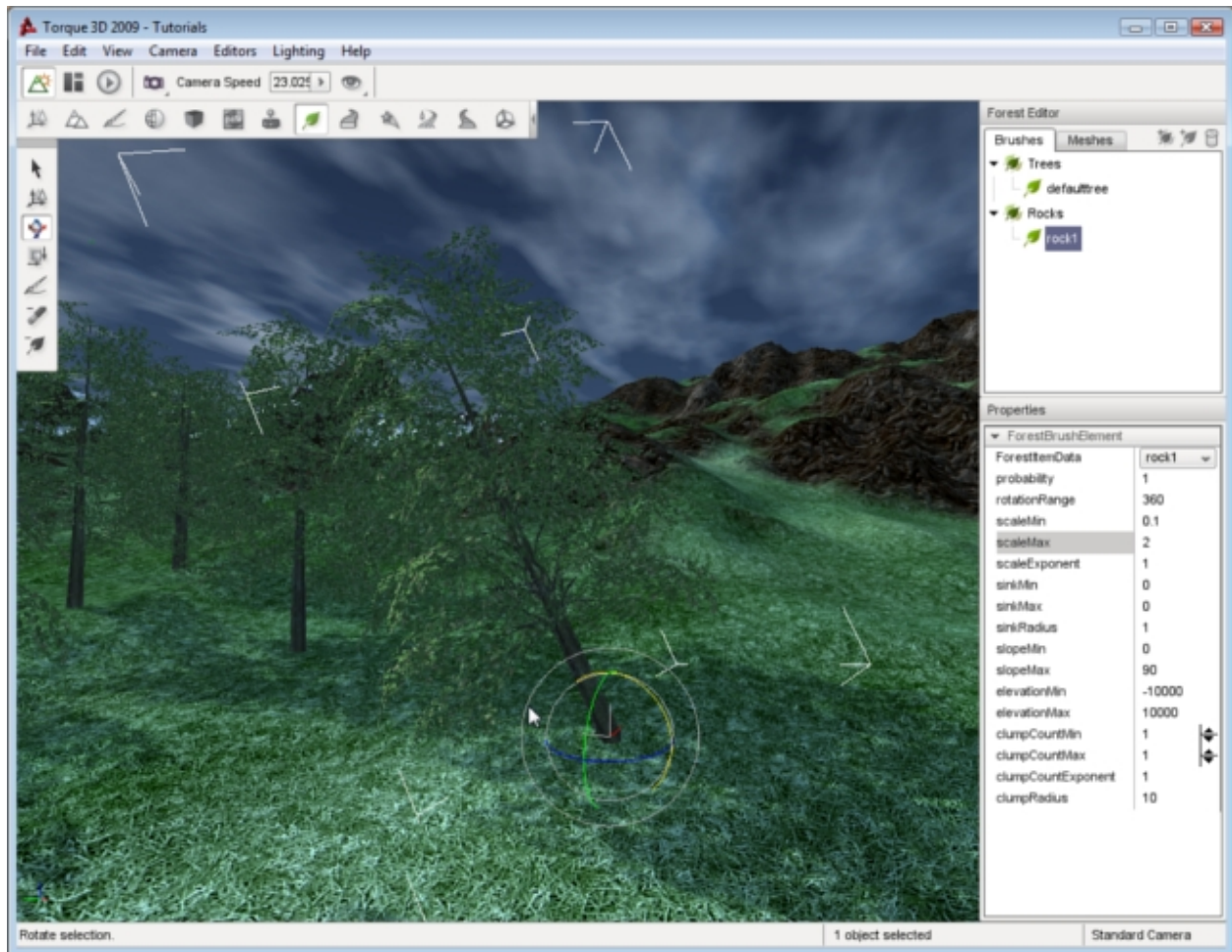
### 13.7.5 Editor Settings

The actions available in the tools palette give you absolute control of your forest placement. The first four tools allow you to adjust individual elements of your forest, such as a single tree. The Select Item tool allows you to select an individual element, which is indicated by a colored axis gizmo appearing on top of the item:



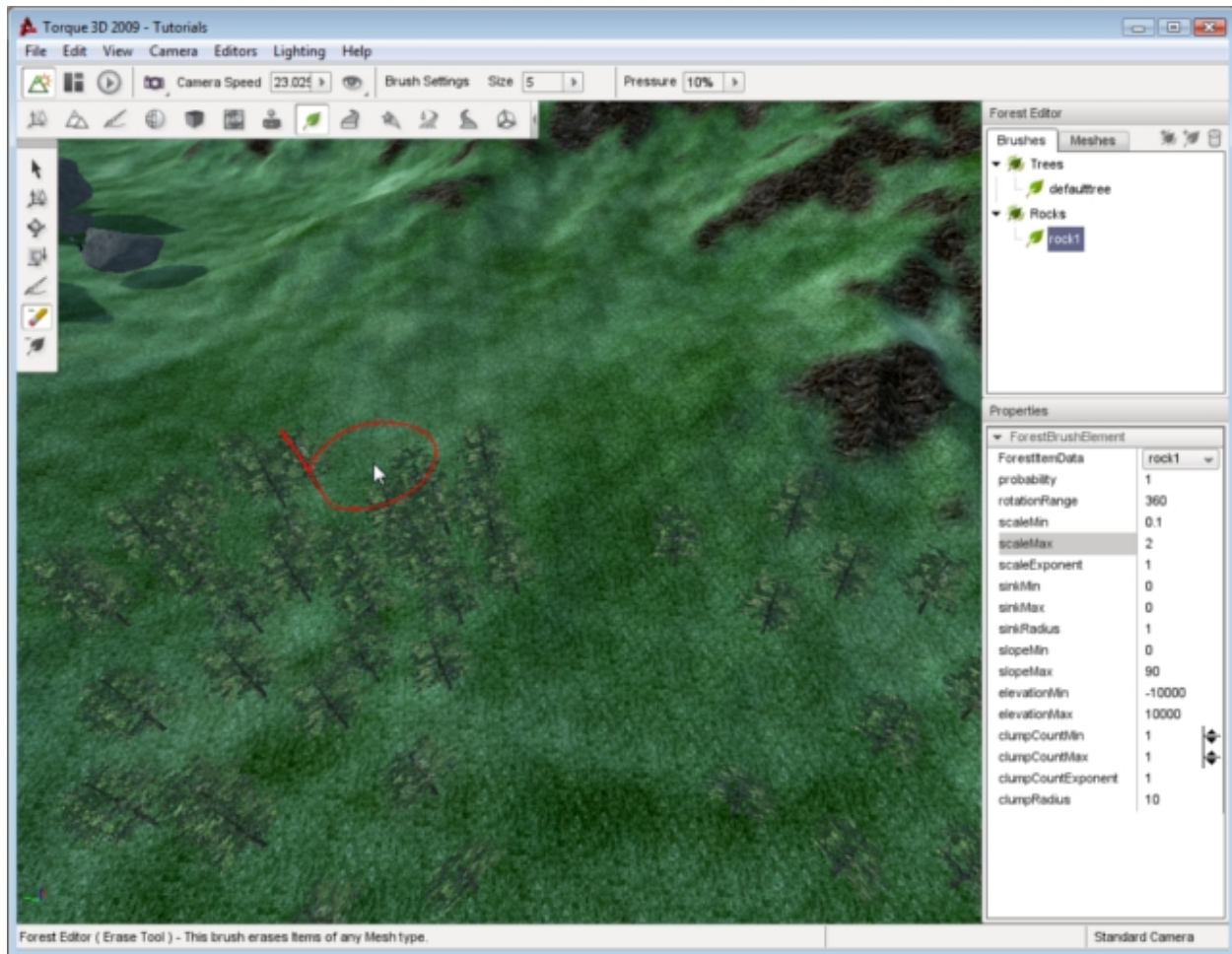
Once you have a tree selected, you can change its location without moving the entire forest. With the tree selected, activate the Move Item tool. The arrows gizmo will appear, allowing you to drag the tree around in the world. The Rotate tool, represented by a spherical gizmo, allows you to adjust the orientation of the tree in 3D space. You can use this to make individual trees lean in a specific direction. The Scale tool can be used to shrink or grow an individual tree. When you need to tidy up a forest, such as removing rogue trees, pressing the delete key when you have a tree selected will remove it from the scene.





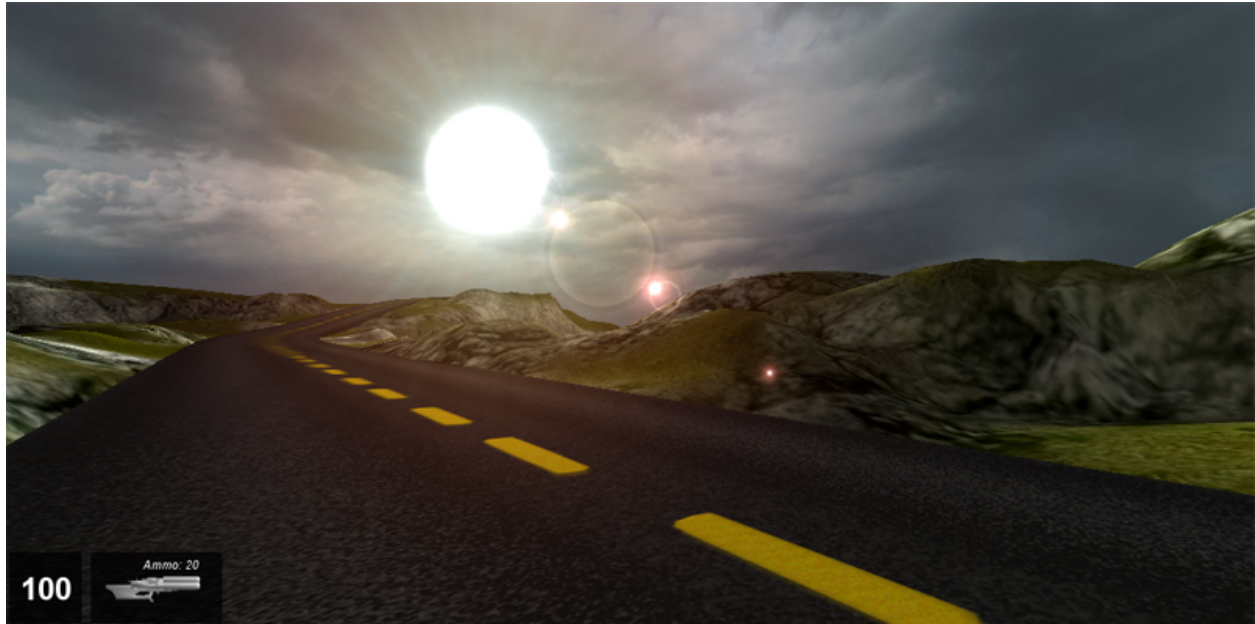
If you need to delete entire sections of a forest, you may not want to delete each tree individually. Instead, you should use the Erase tool. The Erase tool is located directly below the Paint tool. When activated, the circle representing your brush in the world will turn from blue to red when you move your brush over the terrain:





Left click your mouse and drag the brush over a section of trees. Any trees under your brush will be removed from the forest object. This is much faster than deleting individual trees. If you want to remove a larger amount of trees such as clearing an area for a road, you can set the width of the brush to a specific width. Locate the Size dropdown on the Tool Settings bar and click on it. A slider will appear so you can increase the circumference of your brush. Set it to something fairly large, like 20.

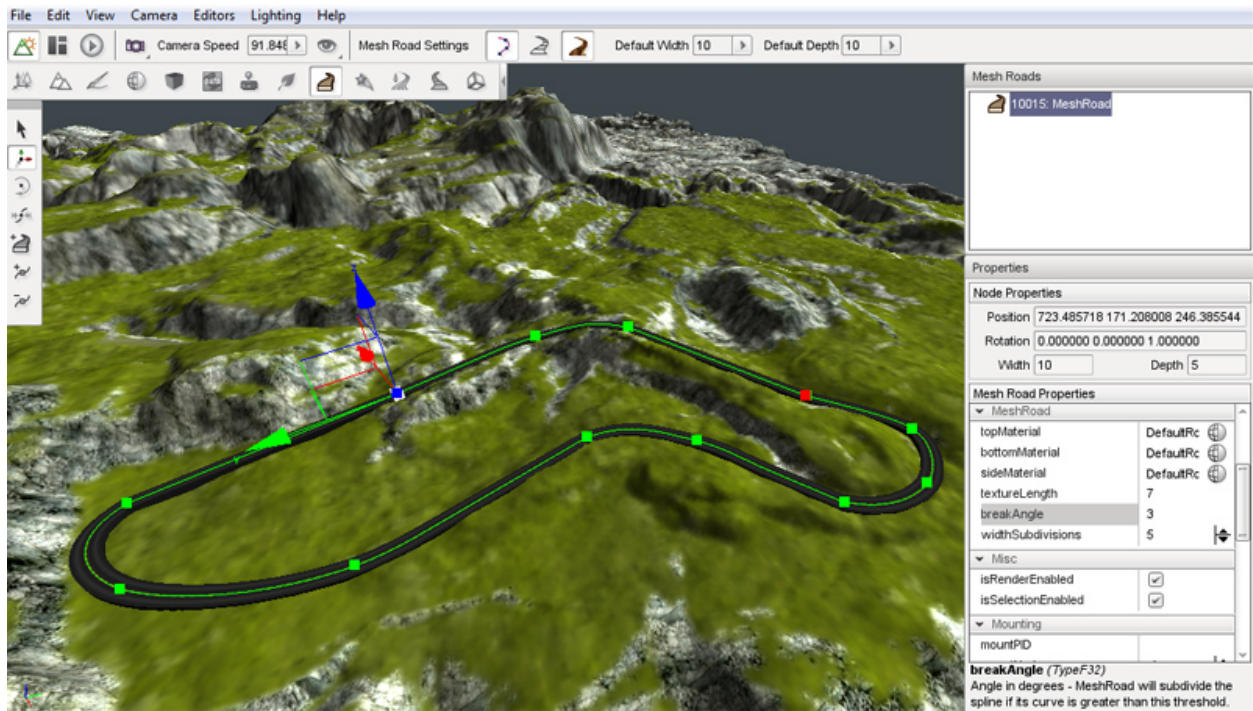
## 13.8 Mesh Road Editor



The Mesh Road editor will help you create a solid mesh road structure upon your terrain. A mesh road is actually a 3D model representation of your road and is not solely dependent upon the terrain height. A mesh road can be raised above the terrain, or suspended between hills unlike a decal road, which painted on the surface of your terrain as a texture, and must follow the terrain exactly. Decal roads are created with the Decal Road Editor.

### 13.8.1 Interface

To access the Mesh Road Editor press the `F9` key or activate it from the main menu of the World Editor by selecting Editors>Mesh Road Editor. Alternatively you can click the Mesh Road icon from the World Editor Tool Selector Bar.



Whenever the Mesh Road Editor is active three sections of the screen are updated to contain the editors tool. On the right side of the screen are the Mesh Roads pane and the Properties pane. At the top is the Mesh Roads pane which contains a list of all the road meshes currently in the level, if any are present. At the bottom is the Properties Pane which displays the properties of the currently selected road mesh.

At the left of the screen the Mesh Road placement tools will appear and are used to create and modify road meshes. At the top of the screen in the World Editor Tool Settings bar, a new set of icons will appear. These icons and their associated values will enable you to quickly set up the width and depth of the control points and modify the editor to show and hide some visual aids which can be used to guide your road placement.

### 13.8.2 Adding a Mesh Road

A road mesh is created by placing a number of control points across the terrain. Each point can be edited for Road height, Road width and Road depth. By adjusting these points we have full control over how our road will look. The default width and depth of control points can be set using the Default Width and Default Height properties on the Tool Settings Bar at the top of the editor window. Any new road meshes will be created using these settings until you change their values again.

To create a new road mesh select the Create Road icon from the tool bar then click on the terrain with the left mouse button where you would like to start your road. Move the mouse away from the clicked location to see the results. Each time you click the terrain you will see three things:

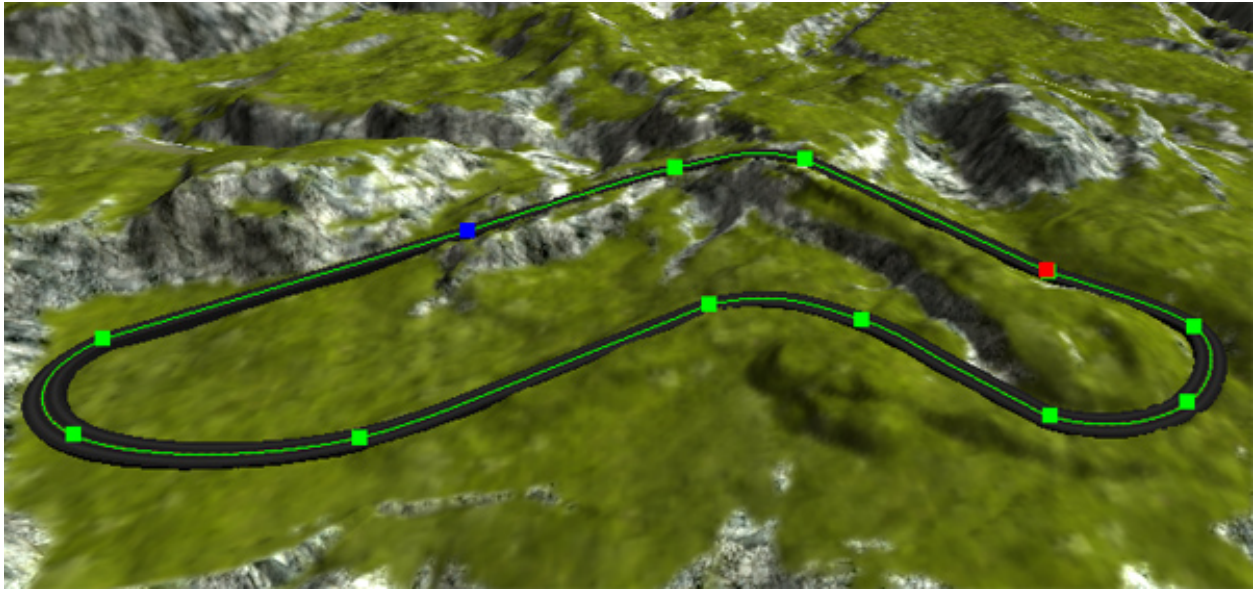
1. a green square which represents the road location that you just placed
2. a blue square which represents the next location that will be placed the road if you press mouse button again
3. the surface of the road that will be placed the next time you click the button

Move the mouse to the next point on the terrain that you wish your road to travel to and then click again. Continue moving and clicking until you are finished with the initial placement of your road.

To complete the road placement process press the `ESC` key. This action will exit the Create Road tool leaving your new road selected and ready for adjustments.



To abort a road creation operation without placing a road at all press the `ESC` key before selecting a second road point. Once a second road point has been placed the only way to remove the road completely is to delete it, as explained later.



The new road will also show up in the Roads Mesh pane above the road Properties pane.

### 13.8.3 Editing a Mesh Road

The Road Mesh Editor provides several tools for modifying roads after they have been created. If at any time you make a mistake with any tool, you can press `CTRL+Z` to undo it.

#### Selection Tool

Once you have created your initial road you may need to edit some or all of the control points. This tool will allow you to directly select any created point for further editing. To activate the Selection Tool click its icon on the Tool Selector bar. Note that the Road Mesh Editor will automatically select this tool when you have finished creating a new road.

An entire road mesh can be selected by clicking anywhere on a road mesh other than one of its control points. This type of selection will result in the road being highlighted with a “spline”, which is a curved line that runs along the center line of the road, and a series of green squares which represent the roads control points. There are no operations that can be performed on a road as a whole within the Mesh Road Editor. Selecting a road allows you to see its centerline and its control points for individual selection and manipulation. To perform operations on the entire road such as moving it to a new location use the Object Editors tools as with any other shape in your level.

Control points can be selected individually to adjust each point as necessary. To select a control point left click on one of the colored squares that represent a roads control points. The selected control point will turn blue.

Selecting a control point also causes the Properties pane on the right of the screen to be updated to display the current property values of the control point. The Node Properties section will display the position, rotation, width and depth of the selected control point. Values can be directly entered into these fields to modify the point or the Move Tool can be used to manipulate the point using the mouse.

## **Moving a Road**

If at any time you are unhappy with the placement of a selected Road Mesh control point you can use the Move Tool to adjust its position. To activate the Move Tool click its icon in the Tool Selector bar. The move gizmo will appear. The move gizmo is used to move the road point to a new location. Left mouse click on any arrowhead then drag the mouse to move the point along that arrows axis. Release the mouse button to relocate the control point to that new location. Left mouse click on the colored square at the origin of the axes then drag the mouse to freely move the point to without regard to any axis.

## **Scaling a Road**

The width and depth of a road can be directly adjusted at a selected control point by using the Scale tool. To activate the Scale Tool click on its icon on the Tool Selector. The scaling gizmo will appear. Left mouse click on the colored cube at the end of any axis then drag the mouse while holding the button down to increase or decrease the size of the road along that axis. Note that if you drag the blue cube to adjust the depth of the road you may not visibly see the adjustment take place because the road depth may be increasing down into the terrain. To adjust the width and depth at the same time left mouse click on the colored cube at the origin of the axes then drag the mouse while holding down the button. Release the mouse button to change the road to that new width and depth.

## **Rotate a Road**

The Rotate Tool can be used to rotate a road at any selected control point. To activate the Rotate Tool click its icon on the Tool Selector. The rotate gizmo will appear. Left click on any colored circle then drag the mouse while holding the button down to rotate the roads surface around that axis at the control point.

## **Inserting Extra Points**

The Insert Point tool can be used to add extra points in a road to create a smoother curve. In order to insert a new point into a road the road must first be selected. See the Selection Tool above for details on how to select a road. To activate the Insert Point tool once a road has been selected click its icon on the Tool Selector bar. To place a new point on the selected road click on the road where you would like the new point to be placed. A new point will be added to the road mesh and will immediately the currently selected point as indicated by the blue square.

## **Removing Points**

The Remove Point tool can be used to delete a control from a road mesh. In order to remove a new point from a road the road must first be selected. See the Selection Tool above for details on how to select a road. To activate the Remove Point tool click its icon on the Tool Selector bar. To remove a control from the selected road point click on the control point. This will remove only the selected point leaving all the others in place. No adjustments will be performed on the other existing control points.

## **13.8.4 Properties**

The Properties pane on the right side of the screen can be used to configure a Mesh Road.

### **Transform**

The transform section contains properties which control the placement, rotation and scale of the Road Mesh as a whole.

**Position** The transform section contains properties which control the placement, rotation and scale of the Road Mesh as a whole.

**Rotation** Indicates the rotation of the entire Road Mesh in the level.

**Scale** Indicates the scale of the entire Road Mesh in the level.

## Mesh Road

The Mesh Road section contains properties which determine and control the textures used to display the Road Mesh. To change any of the textures for the Road mesh click the globe icon to its right. Clicking one of these icons you will open up the Material Selection window.

Click on the material you want to use for the road mesh property then click the Select button. The material will be entered in the property's field and will be used as the material for that portion of the Road Mesh.

**Top Material** Indicates the Material to use for the top surface of the road mesh.

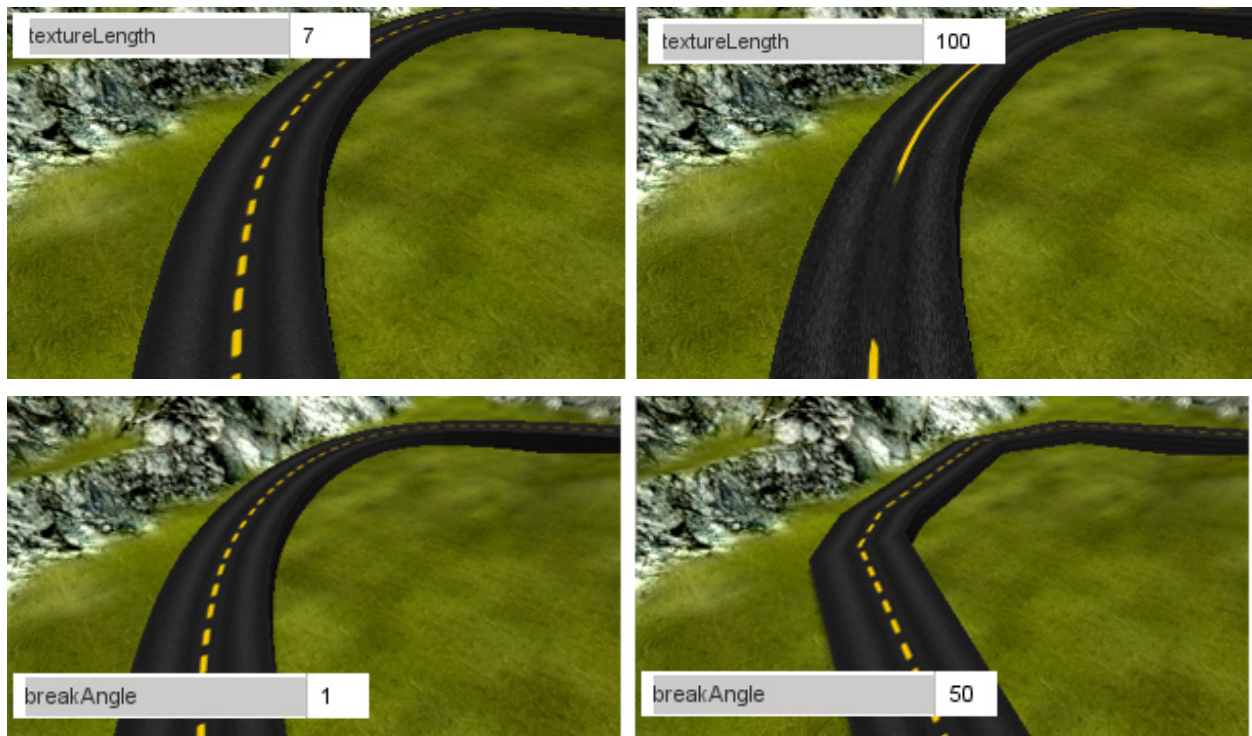
**Bottom Material** Indicates the Material to use for the underside surface of the road mesh.

**Side Material** Indicates the Material to use for the sides of the road mesh.

**Texture Length** Indicates the size in meters of the texture measured along the road center.

**Break Angle** Indicates the angle in degrees that the mesh road's spline will be subdivided into if its curve becomes greater than this threshold.

**Width Subdivisions** Subdivide segments width-wise this many times when generating vertices.





## 13.9 Particle Editor



Torque 3D provides a full featured particle system with many parameters which can be manipulated to fine tune your particle effects. Particle effects are things such as fire balls, smoke, and water splashes that you create and place into your levels. The Torque 3D Particle Editor is the tool of choice for full control over the look and feel of your effects. At its most basic level a particle effect consists of: an emitter, a particle to be emitted from the emitter, and an image rendered to represent that particle.

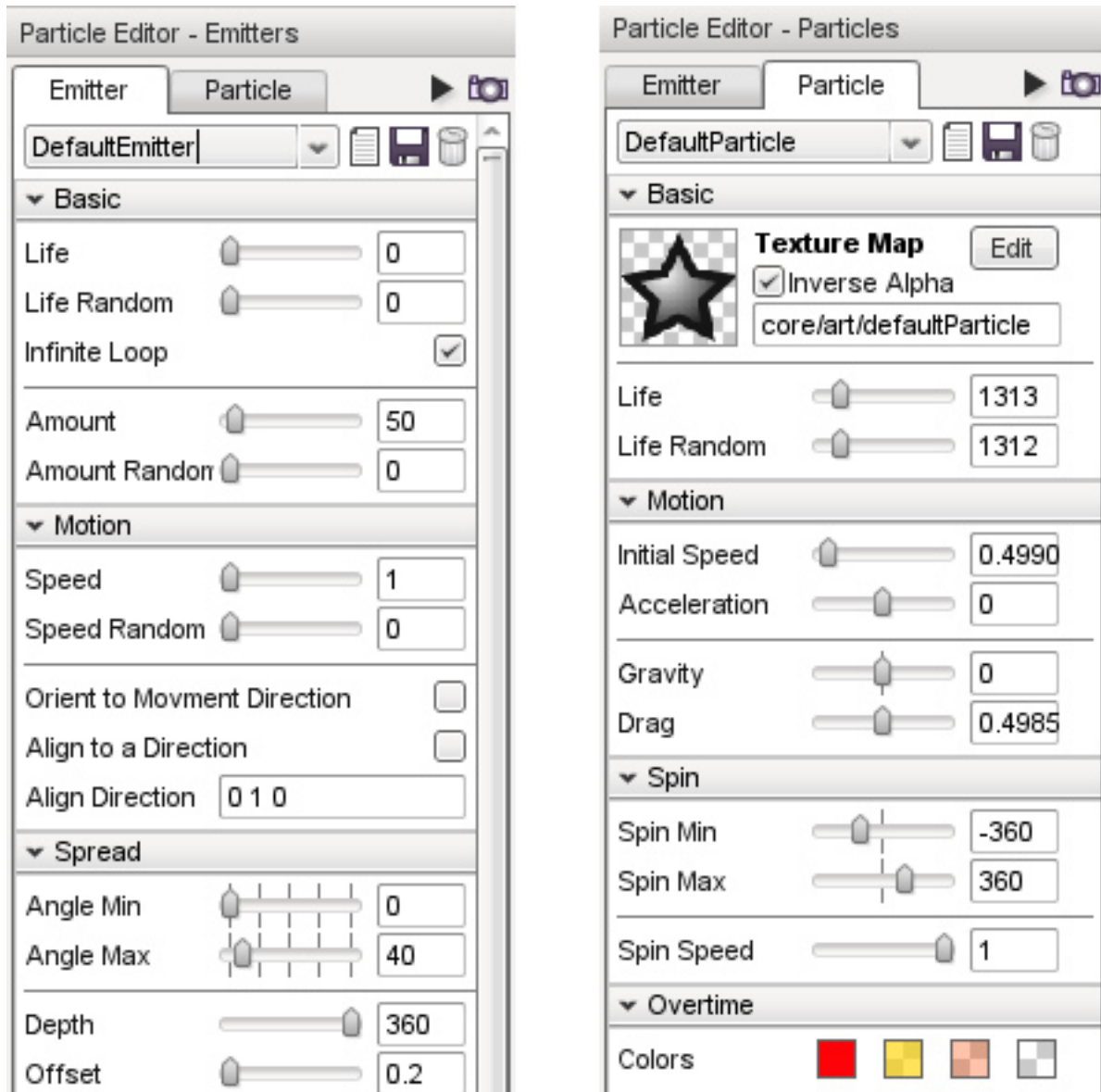
The emitter controls: the creation of the particles; their movement; which directions the particles will travel, also referred to as the spread pattern and: how each particle blends into the world.

The particle controls its own life span, what image will be shown; how big the image is; what it's color over time is; and some basic force settings.

### 13.9.1 Interface

The Particle Editor can be activated from the dmain menu by selecting Editors -> Particle Editor. Or alternately, click the Particle Icon from the Tool Selector bar. Whenever the Particle Editor is active the *Particle Editor – Emitters* pane will be present on the right side of the screen. This pane is further divided into two tabs:

1. The Emitter tab contains properties about the currently selected emitter
2. The Particles tab contains properties about the currently selected particle.



Select either the Emitter tab or particle tab depending upon which object you wish to work with. In addition to the tabs there are also two buttons within the header of the Particle Editor.

### One Shot Effect Types

There are two types of particle effect:

1. continuous effects, which constantly emit particles
2. one-shot effects, which only produce particles for a short time and then stop

Continuous effects run constantly so your changes can be seen in real-time as you adjust the properties of the emitter and its particles. In order to see your changes for one-shot emitters you need to replay the emission. To replay a one-shot emitter click the arrow icon to the right of the tabs.

### The Temporary Emitter

When you open the Particle Editor you may have noticed it creates a temporary particle emitter in your current view. This temporary emitter is very useful for quickly trying out different particle editor settings. If your view is changed and you no longer see temporary emitter, press the little camera icon to the right of the tabs to place it back into view. It will always be placed in the center of your current view. The temporary particle emitter can be moved, rotated, and scaled like any other shape using the Object Editor.

### New Emitter / Particle

To create a new blank emitter or particle that is ready to be configured, press the new icon on the Emitter or Particle tab as appropriate.

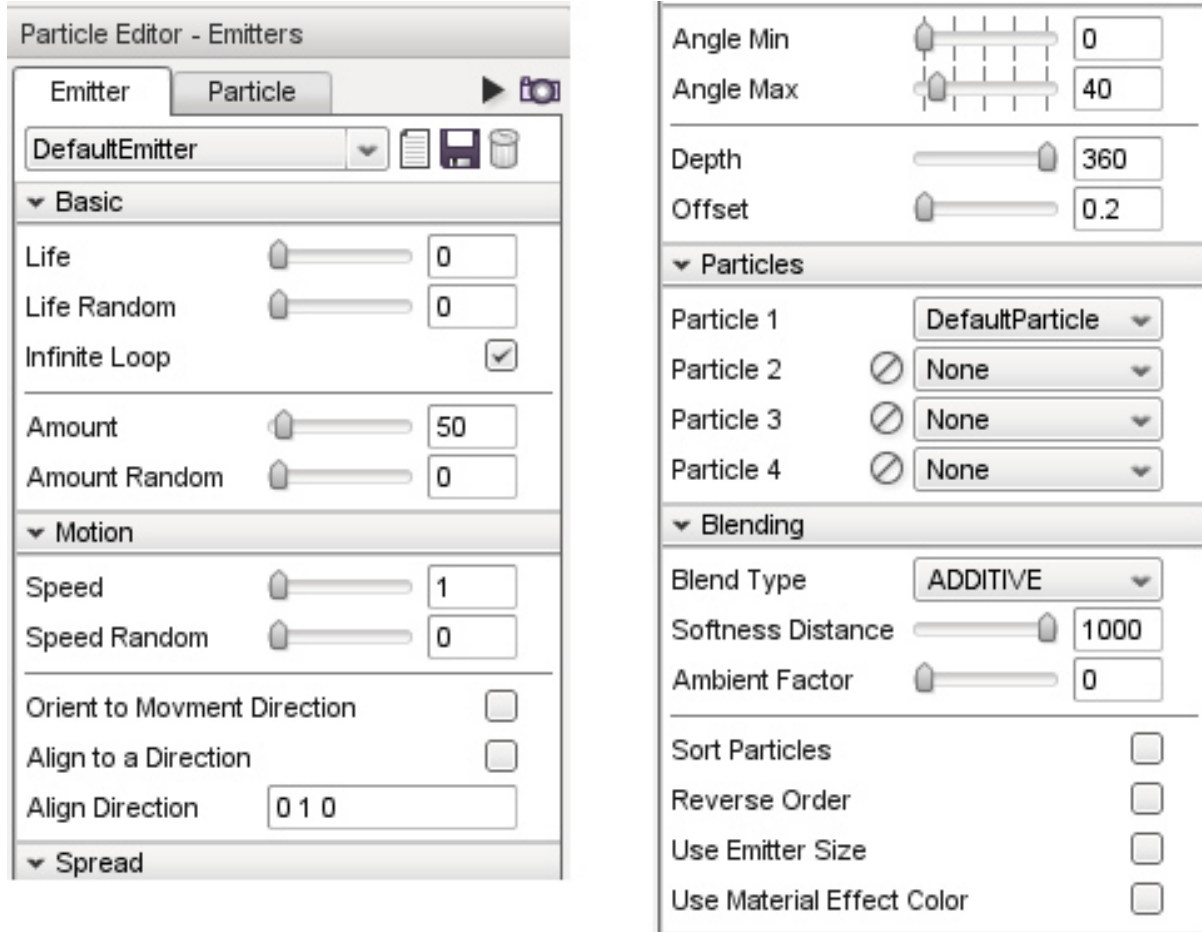
### Save Emitter / Particle

After editing an emitter or particle save the new settings by pressing the save icon on the Emitter or Particle tab as appropriate. Particle emitters are updated in real-time. Any changes to a particle or emitter will be reflected throughout your level when changes are saved. Any instances of the emitter or particle that you are editing will also be changed. As with a lot of Torque 3D Editors the Particle Editor writes the resulting data to script files which the engine runs to create the particle emitter when your game is being played.

- Emitters can be found in a file named: `projectName/game/art/levels/levelName.mis`
- Particles can be found in a file named: `projectName/game/art/shapes/particles/managedParticleData.cs`

## 13.9.2 Emitter Properties

The Emitter tab contains the properties that define an Emitter. Properties are grouped into sections:



## Basic

Basic properties affect the base emitter:

**Life** The time duration in ms that the effect will emit particles.

**Life (Random)** Substitutes a random value for the life property.

**Infinite Loop** When enabled this emitter will continuously produce particles. This setting effectively causes the Life and Random Life properties to have no affect on the emitter.

**Amount** The time in ms between each individual particle released from the emitter.

**Amount Random** Random Variation amount to be applied to the amount setting.

## Motion

These settings will affect the emitter spread pattern, speed, and particle image orientation:

**Speed** The velocity the particle will leave the emitter in the defined spread pattern.

**Speed Random** A random setting for varying the speed.

**Orient to Movement Direction** Enabling this option fixes the particles image to the velocity direction of the particle. Note this will over ride any particle spin settings.

**Align to a Direction** Enabling this option aligns the particles to a predefined vector set up in Align Direction option.

**Align Direction** The vector used for particle alignment if the Align to a Direction option is checked.



## Spread

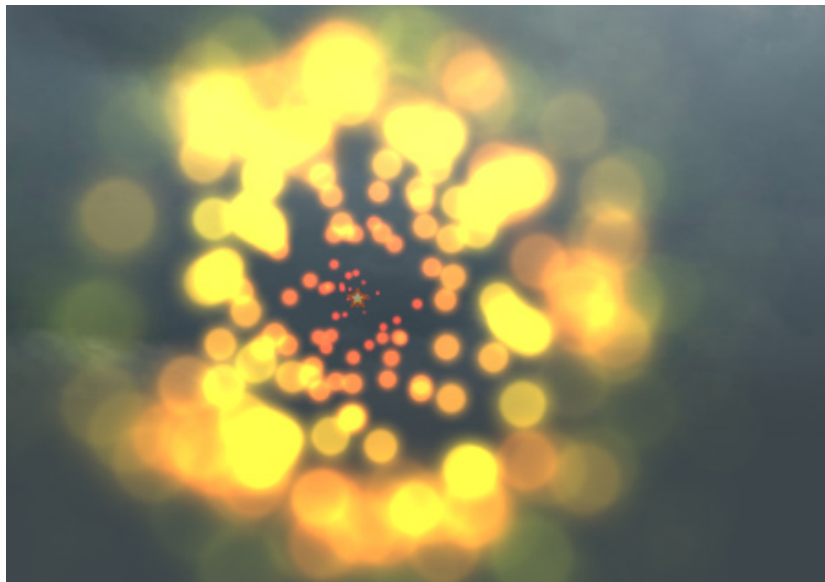
These settings affect how the spread pattern will be dispersed:

**Angle Min** The minimum angle for the emitter spread pattern.

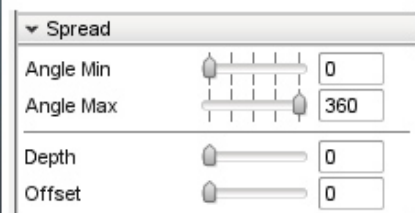
**Angle Max** The maximum angle for the emitter spread pattern.

**Depth** The depth of the released pattern. A setting of 360 will create a spherical spread pattern when Angle Max is set to 360.

**Offset** The distance from the emitter that particles will be released. Effectively the distance that the particle will be visible to the viewer.



Example of Spread  
Angle min /max







Example of Spread  
Angle min /max

▼ Spread	
Angle Min	0
Angle Max	187.5
Depth	0
Offset	0



Example of Spread Offset

▼ Motion	
Speed	0.1
Speed Random	0
Orient to Movement Direction	<input type="checkbox"/>
Align to a Direction	<input type="checkbox"/>
Align Direction	0 1 0
▼ Spread	
Angle Min	0
Angle Max	360
Depth	0
Offset	0.833

## Particles

This affect assigns which particle(s) will be emitted from this emitter:

**Particle 1 - 4** Select the particle from the drop down list to be used with this emitter. If at any time you need to remove a particle press the clear icon. Particle 1 can not be removed.

## Blending

These setting affect how the particle(s) are rendered.

**Blend Type** The types of blending available to be applied to the particles.

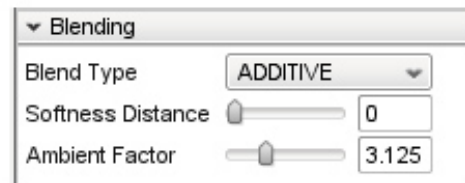
**Softness Distance** The particle edge blending distance. Removes the hard edges where the particle meets an object.



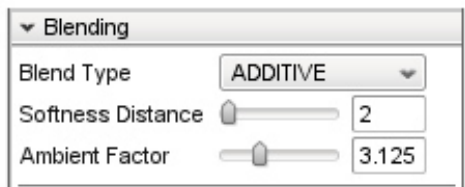
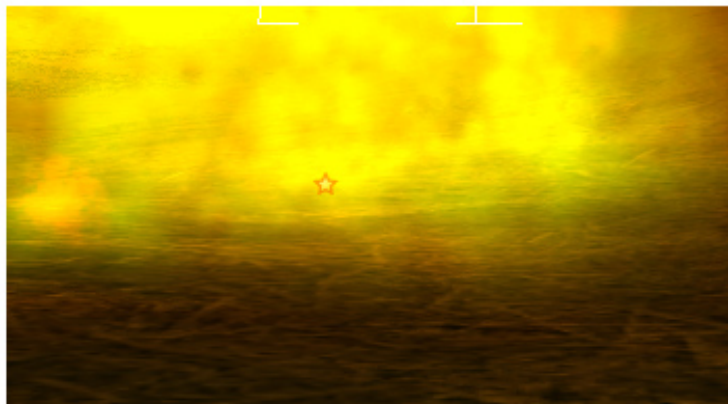
**Ambient Factor** Adjusts the alpha blend (level of the particles which affects how transparent they are).

**Sort Particles** The order in which particles are rendered.

**Reverse Order** When enabled, reverses the render order set in the Sort Particles setting



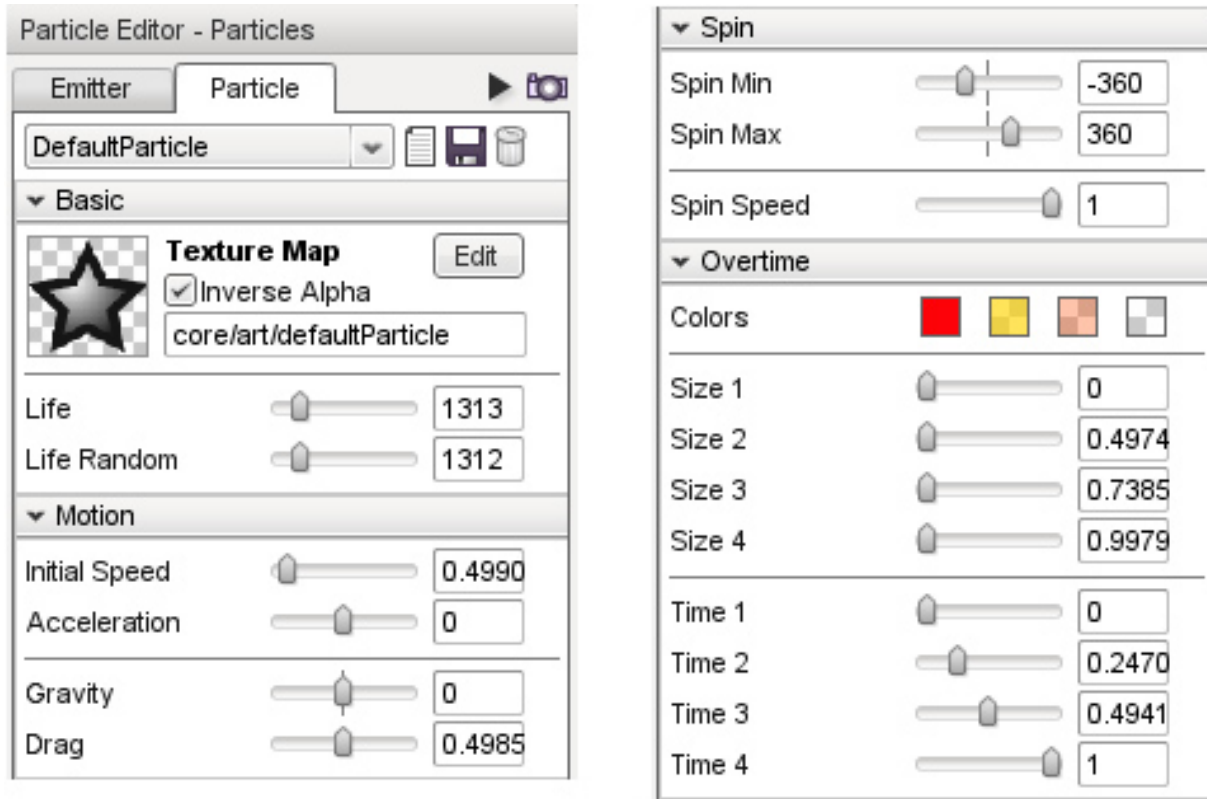
Note the hard edges where the particle meets the Terrain



Softness Distance helps to blend these edges

### 13.9.3 Particle Properties

The Particle tab contains the properties that define a Particle. Properties are grouped into sections:



## Basic

Particle basic settings.

**Texture Map** The image that will be used on the emitted particle. The Edit button will open a file browser to locate and select a particle image.

**Inverse Alpha** Invert the alpha channel on the particle image (if one exists).

**Life** The time in ms (milliseconds) after its creation that the particle will exist for.

**Life Random** Random variation to the particle life span.

## Motion

These settings affect the velocity of the particle.

**Initial Speed** The initial velocity, that the particle will travel at after being emitted. (Not to be confused with emitter spread speed.)

**Acceleration** The rate at which the particle's velocity will increase or decrease. Positive values cause a particle to speed up over time after being emitted. Negative values cause a particle to slow down over time after being emitted.

**Gravity** The gravitational force to be applied to particle. Positive values cause the particle to fall to the ground. Negative values cause the particle to rise from the ground.

**Drag** The amount of force working against the particle velocity. Drag will slow a particle's movement.

## Spin

These settings affect if, and how, a particle rotates in degrees.

**Spin Min** The minimum rotation to be applied to the particle.

**Spin Max** The maximum rotation to be applied to the particle.

**Spin Speed** The speed of particle's rotation.

## Overtime

These settings affect the particle based upon how long it has been in existence for. Each particle can have up to four color and size settings, which can be set to change over time.

**Colors** Four color swatches indicate the color phases which a particle can pass through. To set any color click that swatch. To set a color value you may: enter R (red), G (green), and B (blue) color values; click anywhere within the gradient on the left or; click anywhere in the vertical “rainbow” strip. Red, green and blue color values range from 0 to 255 and indicate the amount of that color present in the overall particle color. The alpha value which represents the transparency of the particle color can be set by entering a decimal number between 0.0 and 1.0 in the Alpha field or by moving the slider with the mouse. The higher the number the less transparent the color will be.

**Size 1-4** Each slider sets the size for the particle during each time stage.

**Time 1-4** Each slider sets the time for that stage.

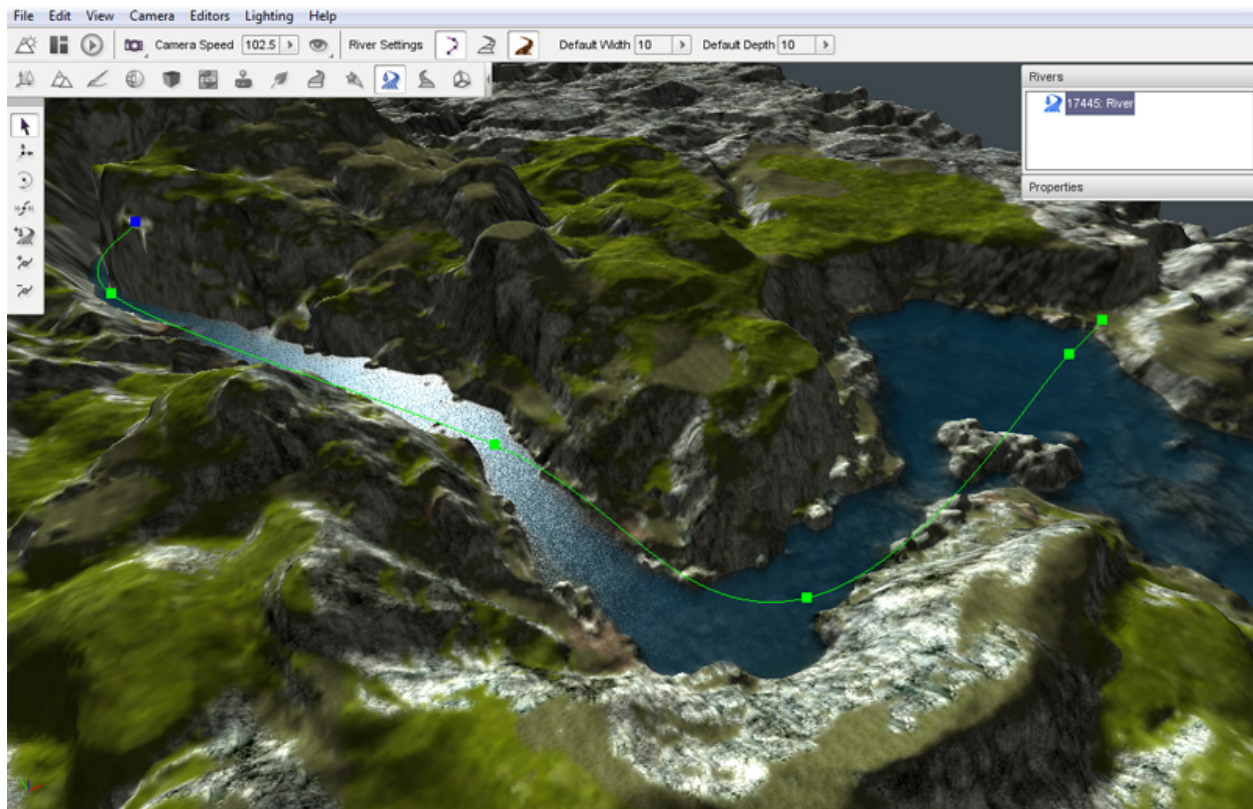
## 13.10 River Editor



The Torque 3D World Editor has a complete system for creating rivers and other small bodies of water. The River Editor is built-in WYSIWYG (What-You-See-Is-What-You-Get) editor with real-time feedback, giving you full control over how you would like to your river to appear.

### 13.10.1 Interface

To access the River Editor you can either activate it from the main menu by selecting Editors > River Editor. Alternatively you can click the River icon from the World Editor Tools Selector Bar.



The editor has two main parts, one where you can add and manipulate the river nodes (control points) for creation, the other for adjusting the river properties to create the river style (depth, width, flow, ripples etc). Whenever the River Editor is active three sections of the screen are updated to contain the editors tool. On the right side of the screen are three panes. At the top is the Rivers pane which contains a list of all the rivers currently in the level, if any are present. In the middle is the River Nodes pane which displays the properties of the currently selected river control point. At the bottom is the Properties Pane which displays the properties of the currently selected river. At the left of the screen the River placement tools will appear and are used to create and modify rivers and their control points. At the top of the screen in the World Editor Tool Settings Bar, a new set of icons will appear when the River Editor is active. These icons and their associated values will enable you to quickly set up the width and depth of the river and modify the editor to show and hide some visual aids which can be used to guide your river placement.

### 13.10.2 Adding a River

The river is created by placing a series of control points across the terrain which defines the path you would like your river to follow. Each control point, also called a “node”, will give you control of how the river will look at any given point. By adjusting each of these points we can have full control of where our river will go, its size, and its orientation.

The default width and depth of control points can be set using the Default Width and Default Height properties on the Tool Settings Bar at the top of the editor window. Any new rivers will be created using these settings until you change their values again.

To create a new river select the Create River icon from the tool bar then click on the terrain with the left mouse button where you would like to start your river. Move the mouse away from the clicked location to see the results. Each time you click the terrain you will see three things:

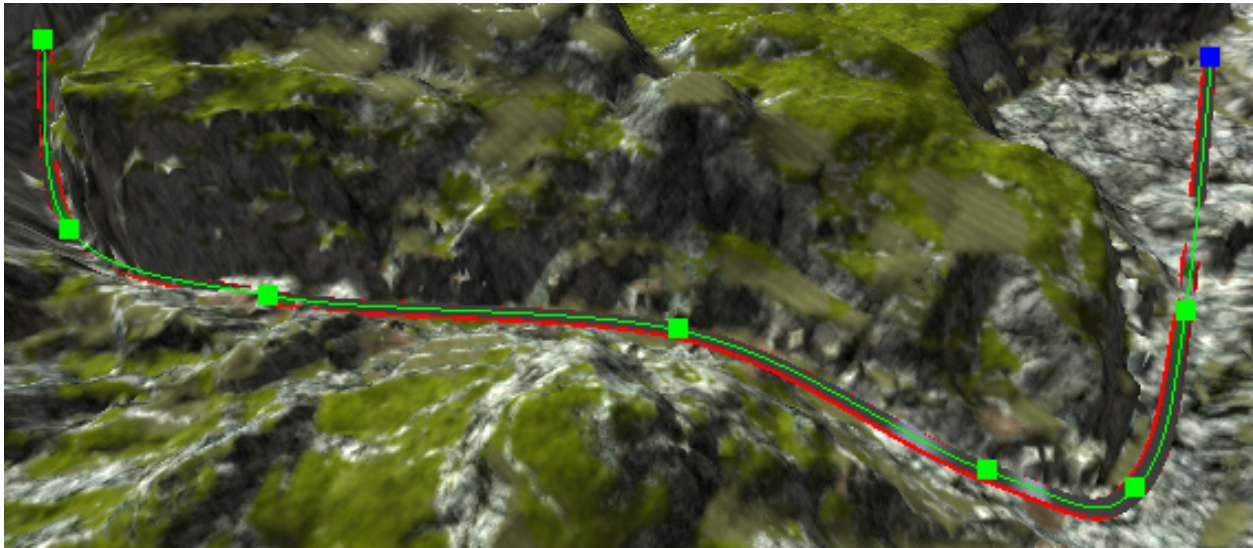


1. a green square which represents the river location that you just placed
2. a blue square which represents the next location that will be placed the river if you press mouse button again
3. the surface of the river that will be placed the next time you click the button

Move the mouse to the next point on the terrain that you wish your river to travel to and then click again. Continue moving and clicking until you are finished with the initial placement of your river.

To complete the river placement process press the `ESC` key. This action will exit the Create River tool leaving your new river selected and ready for adjustments.

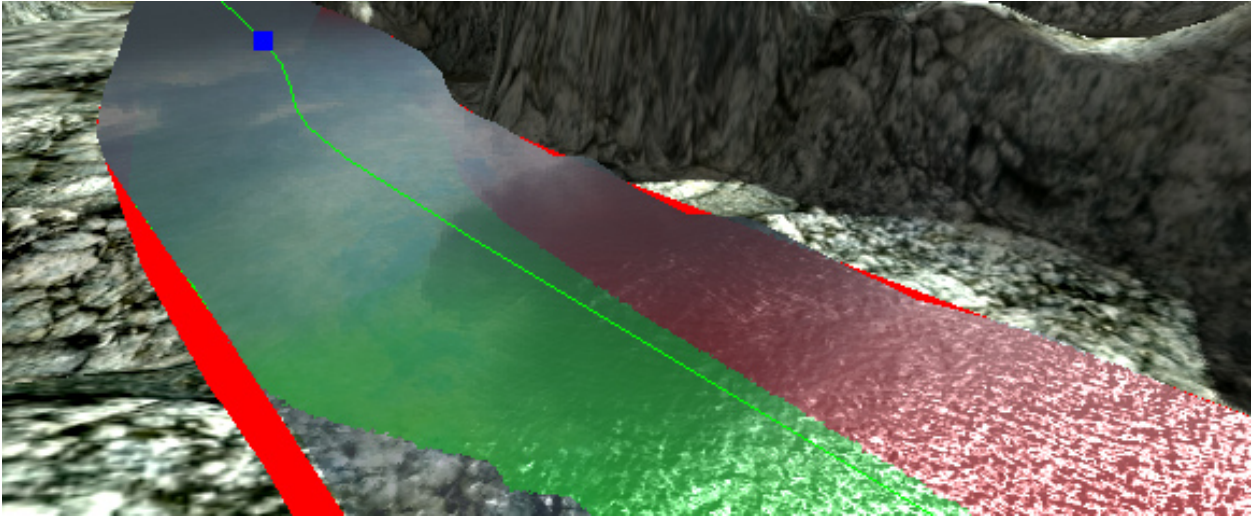
To abort a river creation operation without placing a river at all press the `ESC` key before selecting a second river point. Once a second river point has been placed the only way to remove the river completely is to delete it, as explained later.



The new River will also show up in the Rivers list at the right top of the screen. You will notice that the river itself is color-coded. These visual aids will be of help in adjusting your river.

**RED** The red lines at the edges of the river represent the river bounds. These lines need to be moved so that they are hidden by the terrain river bank to avoid gaps between the edge of the water and the surrounding terrain.

**GREEN** The green surface represents the depth of the river. This surface needs to be adjusted to be just below the terrain at every point in order for underwater views to be correct. Whenever this surface is above the terrain it will cause an “air bubble” between the bottom of the water and the terrain.



The new River may not look correct but with the following set of tools you will be able to adjust the width, depth and path.

### 13.10.3 Editing a River

The River Editor provides several tools for modifying rivers after they have been created. If at any time you make a mistake with any tool, you can press CTRL+Z to undo it.

#### Selection Tool

Once you have created your initial river you may need to edit some or all of the control points. This tool will allow you to directly select any created point for further editing. To activate the Selection Tool click its icon on the Tool Selector bar. Note that the River Editor will automatically select this tool when you have finished creating a new river.

An entire river can be selected by clicking anywhere on a river other than one of its control points. This type of selection will result in the river being highlighted with a “spline”, which is a curved line that runs along the center line of the river, and a series of green squares which represent the rivers control points. There are no operations that can be performed on a river as a whole within the River Editor. Selecting a river allows you to see its centerline and its control points for individual selection and manipulation. To perform operations on the entire river such as moving it to a new location use the Object Editors tools as with any other shape in your level.

Control points can be selected individually to adjust each point as necessary. To select a control point left click on one of the colored squares that represent a rivers control points. A selected control point will turn blue.

Selecting a control point also causes the Properties pane on the right of the screen to be updated to display the current property values of the control point. The Node Properties section will display the position, rotation, width and depth of the selected control point. Values can be directly entered into these fields to modify the point or the Move Tool can be used to manipulate the point using the mouse. A selected control point will turn blue.

#### Moving a River

If at any time you are unhappy with the placement of a selected River control point you can use the Move Point tool to adjust its position. To activate the Move Point tool click its icon in the Tool Selector bar. The move gizmo will appear. The move gizmo is used to move the river point to a new location. Left mouse click on any arrowhead then drag the mouse to move the point along that arrows axis. Release the mouse button to relocate the control point to that new location. Left mouse click on the colored square at the origin of the axes then drag the mouse to freely move the point to without regard to any axis.

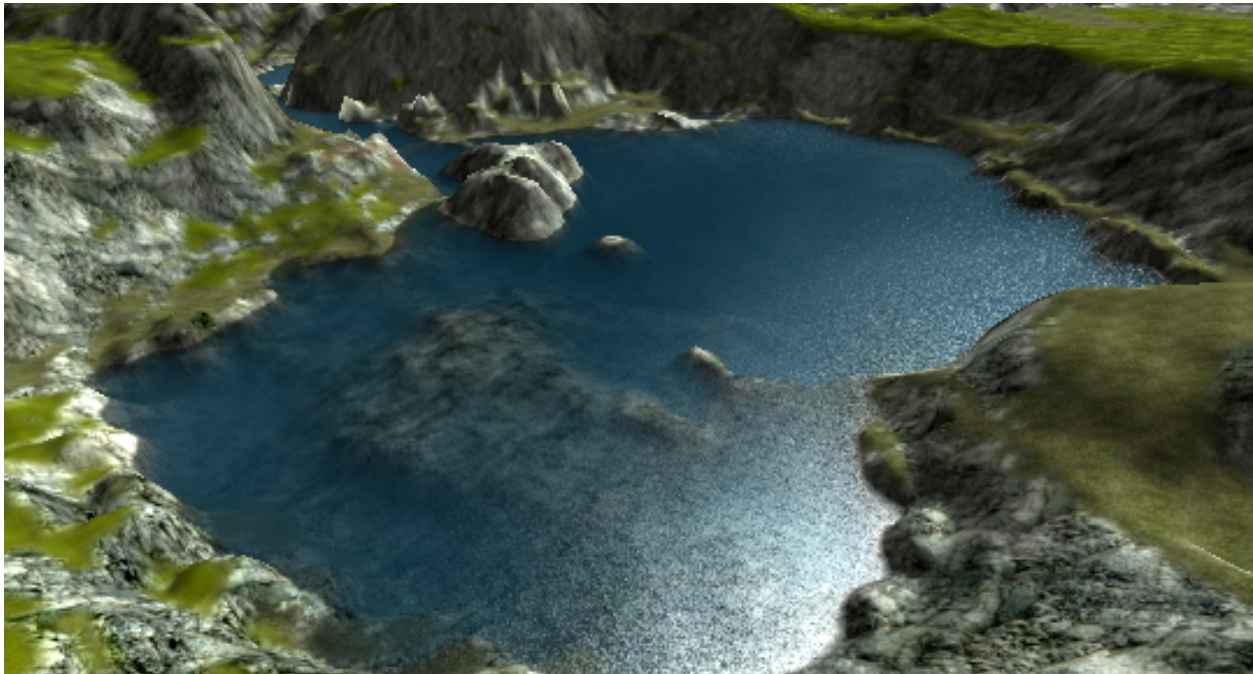


## Scaling a River

The width and depth of a river can be directly adjusted at a selected control point by using the Scale Point tool. To activate the Scale Point tool click on its icon on the Tool Selector. The scaling gizmo will appear.

Left mouse click on the colored cube at the end of any axis then drag the mouse while holding the button down to increase or decrease the size of the river along that axis. To adjust the width and depth at the same time left mouse click on the colored cube at the origin of the axes then drag the mouse while holding down the button. Release the mouse button to change the river to that new width and depth. Changing the width and depth of the river in this manner is the main method to make sure that the red edges and the green surface, mentioned above, are concealed by the terrain.

The Scale point tool will allow you to quickly create very wide river sections, even as wide as a small lake, without having to use a WaterBlock.



## Rotating a river

The Rotate Tool can be used to rotate a river at any selected control point. To activate the Rotate Tool click its icon on the Tool Selector. The rotate gizmo will appear. Rotating a river at each control in along the path of a river can make a river appear to be flowing downhill as opposed to a flat surface as is created by default.

## Adding extra Points

The Insert Point tool can be used to add extra points in a river to create a smoother curve. In order to insert a new point into a river the river must first be selected. See the Selection Tool above for details on how to select a river. To activate the Insert Point tool once a river has been selected click its icon on the Tool Selector bar. To place a new point on the selected river click on the river where you would like the new point to be placed. A new point will be added to the river and will immediately the currently selected point as indicated by the blue square.

## Removing Points

The Remove Point tool can be used to delete a control from a river. In order to remove a new point from a river the river must first be selected. See the Selection Tool above for details on how to select a river. To activate the Remove Point tool click its icon on the Tool Selector bar. To remove a control from the selected road point click on the control point. This will remove only the selected point leaving all the others in place. No adjustments will be performed on the other existing control points.

## 13.10.4 Properties

The Properties pane on the right side of the screen can be used to configure or modify various facets of the river object, such as its flow, colors, underwater effects, etc.

### Transform

This section contains properties which control the placement, rotation and scale of the River as a whole.

**Position** Indicates the position of entire River in the level.

**Rotation** Indicates the rotation of the entire River in the level.

**Scale** Indicates the scale of the entire River in the level.

### River

This section contains properties which control how the River is rendered which in turn will have an effect on the wave settings.

**Segment Length** The river will be divided into segments of this length, in meters.

**Subdivide Length** River segments will be subdivided in a way that each quad (four-sided polygon) is not any wider or longer than this distance in meters.

**Flow Magnitude** The magnitude of the force vector applied to dynamic objects that are within the River. This will affect how things floating or suspended in the water are driven by the flow of the river.

**Low LOD Distance** Segments of the river at this distance in meters or more will be rendered as a single un-subdivided area without any undulation wave effects.

### Water Object

This section contains properties that control the look and action of the water and contains several sub-sections.

**Density** Affects the buoyancy of an object entering the water.

**Viscosity** Affects a submerged object's drag force.

**Liquid Type** Type of datablock used to represent the type of liquid contained in the river (i.e. water, lava, etc.)

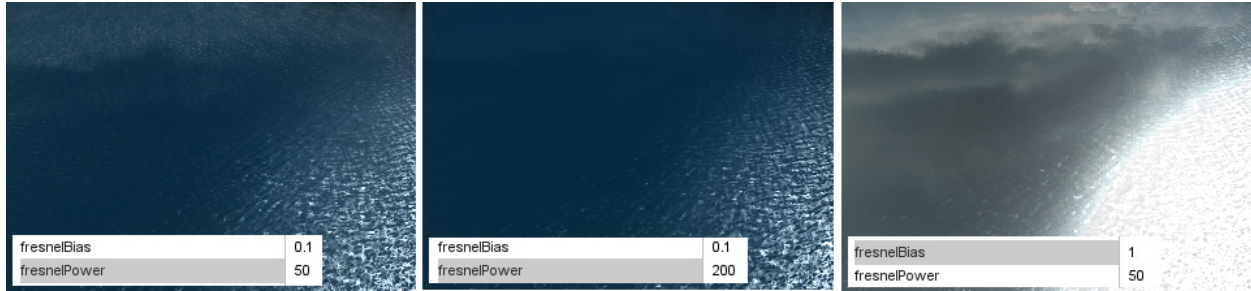
**Base Color** Changes the color of the underwater fog which has the effect of coloring the water surface.

**Fresnel Bias** Extent of Fresnel (reflection level based on viewing angle) affecting reflection fogging

**Fresnel Power** Measures the intensity of the effect on the reflection, based on fogging.

**Specular Power** Power used for specularity (lighting reflection) on the water surface (sun only)

**Specular Color** Specular color used for the water surface ( sun only )



## Waves

This sub-section contains properties that control the undulations on the water. Note: This effect actually moves the vertices of the mesh. This section has further sub-sections for controlling three wave sets, each sub-section is composed of the following properties that define the wave set.

**Wave Dir** A vector describing the direction the waves flow towards the river banks.

**Wave Speed** Speed of the wave undulation.

**Wave Magnitude** Height of the wave.

**Overall Wave Magnitude** This master parameter affects the depth of all the wave subsets, like a global wave height parameter.

**Ripple Texture** The Normal map used for simulating the surface ripples.

## Ripples

This sub-section contains properties that control the animation that simulates the effect of ripples bouncing off the river bank. This animation is performed using normal map to represent the ripples. This section has further sub-sections for controlling three ripple sets, each sub-section is composed of the following properties that define the ripple set.

**Ripple Dir** A vector that modifies the surface ripple direction.

**Ripple Speed** Controls the ripple speed.

**Ripple Tex Scale** Intensifies the affect of the surface ripples by scaling the texture.

**Ripple Magnitude** Intensifies the ripple effect.

**Overall Ripple Magnitude** This parameter affects the depth of all the ripple subsets, like a global ripple intensity variable.

**Foam Tex** The texture used to render the ripple effect.

## Reflect

This section contains properties that control the rendering of surface reflections:

**CubeMap** Cubemap to use instead of the default reflection texture, which is the current sky, if Full Reflect is turned off. Handy if you have not yet set up a sky for your project.

**FullReflect** Enables dynamic reflection rendering, which causes the water surface to reflect the current sky, if available.

**Reflect Priority** Affects the sort order of reflected objects.

**Reflect Max Rate Ms** Affects the sort time of reflected objects.

**Reflect Detail Adjust** Scale up or down the detail level for objects rendered in a reflection.

**Reflect Normal Up** The reflection normal.

**Use Occlusion Query** Turn off reflection rendering when occluded.

**Reflect Text Size** Texture size used for the reflections.

## Underwater Fogging

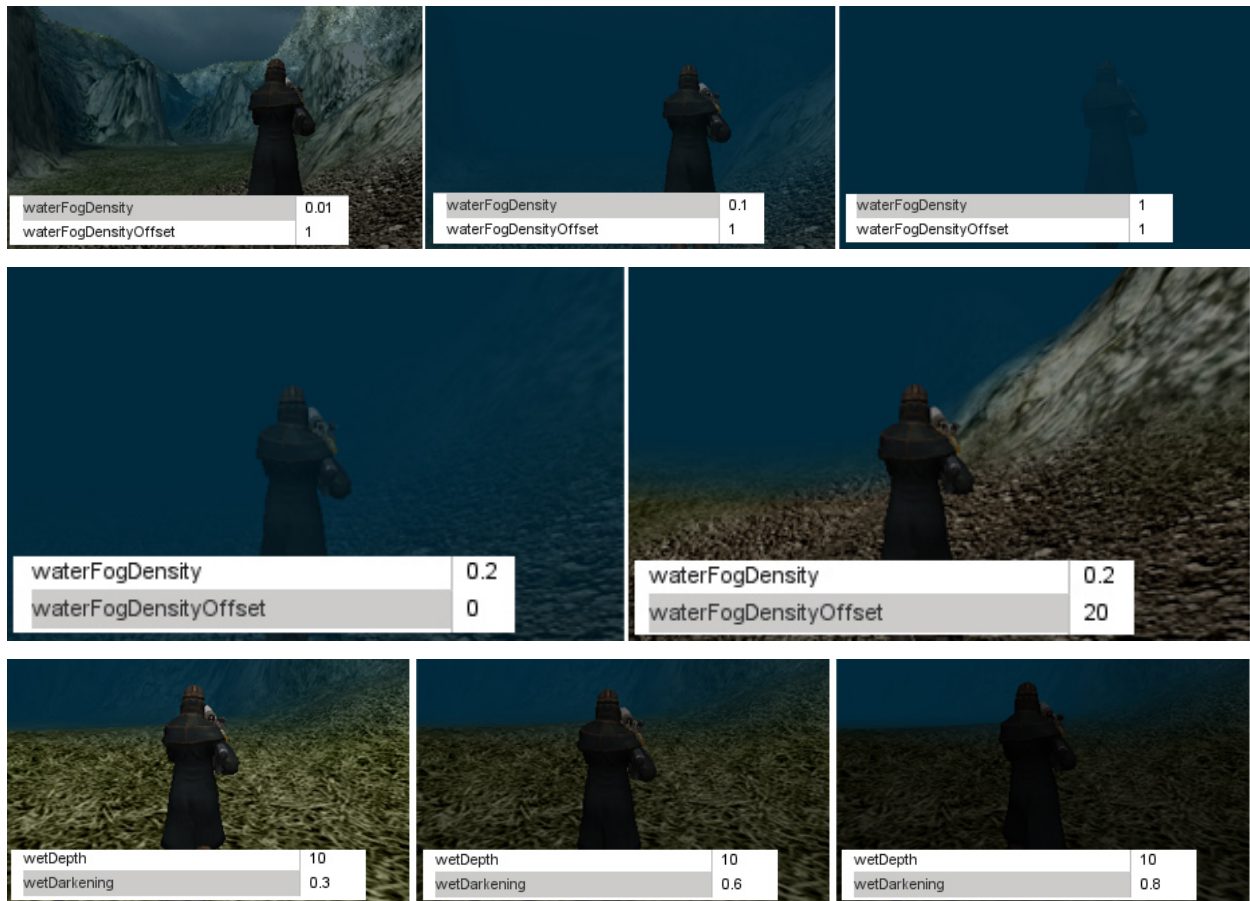
This section contains properties that control how the underwater view appears.

**Water Fog Density** The intensity of the underwater fogging.

**Water Fog Density Offset** The offset distance before the fogging occurs.

**Wet Depth** The depth in world units at which full darkening will occur, giving a wet look to objects underwater.

**Wet Darkening** The refract color intensity scaled to the depth of the player (wetDepth). The deeper under the water you go, the darker it will get.



## Misc

Other uncategorized properties.

**Depth Gradient Tex** Texture for the gradient as the players moves deeper.

**Depth Gradient Max** Maximum depth for the gradient texture.



## Foam

**Foam Opacity** Overall foam opacity.

**Foam Max Depth** The depth that the foam will be visible from underwater.

**Foam Ambient Lerp** An RGB color value that interpolates linearly between the base foam color and ambient color. This prevents bright white colors be viewable during situations such as Night.

**Foam Ripple Influence** Intensity of the foam effect on ripples.

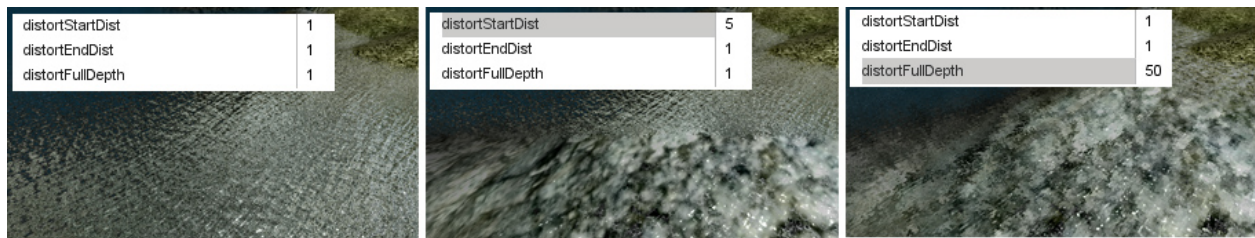
## Distortion

This section contains properties that control how the water distorts the under water terrain when viewed from above.

**Distort Start Dist** Determines the start of the distortion effect from the camera.

**Distort End Dist** Max Distance that the distortion algorithm is performed. Lower values will show more of the distortion effect.

**Distort Full Depth** Sets the scaling down value for the distortion in shallow water. The lower the value the more the distortion will be applied to the shallow area.



## Basic Lighting

This section contains properties that control the basic lighting effects on and in the water:

**Clarity** Opacity or transparency of the water surface.

**Underwater Color** Changes the color shading of objects beneath the water surface

## Sound

This section contains properties that control sound under the water.

**Sound Ambience** Ambient sound environment for when the listener is submerged.

## Editing

This section contains properties that control whether the river can be edited.

**isRenderEnabled** Toggles whether the object is rendered on the client.

**isSelectionEnabled** Toggles whether the object can be selected in the tools.

**hidden** Toggles whether the object is visible.

**locked** Toggles whether the object can be edited.

## Mounting

This section contains properties that control whether the river can be mounted to another world object, for example a sewer pipe or a cave.

**mountPID** PersistentID of object we are mounted to.

**mountNode** Node we are mounted to.

**mountPos** Position where object is mounted.

**mountRot** Rotation where object is mounted.

## Object

This section contains properties that control whether the river object is persistent in the world.

**internalName** Internal name of this object.

**parentGroup** Group to which this object belongs.

**class** Class to which this object belongs.

**superClass** SuperClass to which this object belongs.

## Persistence

This section contains properties that control whether the river object is persistent in the world.

**canSave** Whether the object can be saved to the mission file.

**canSaveDynamicField** Whether dynamic properties are saved at runtime.

**persistentID** Unique ID of this object.



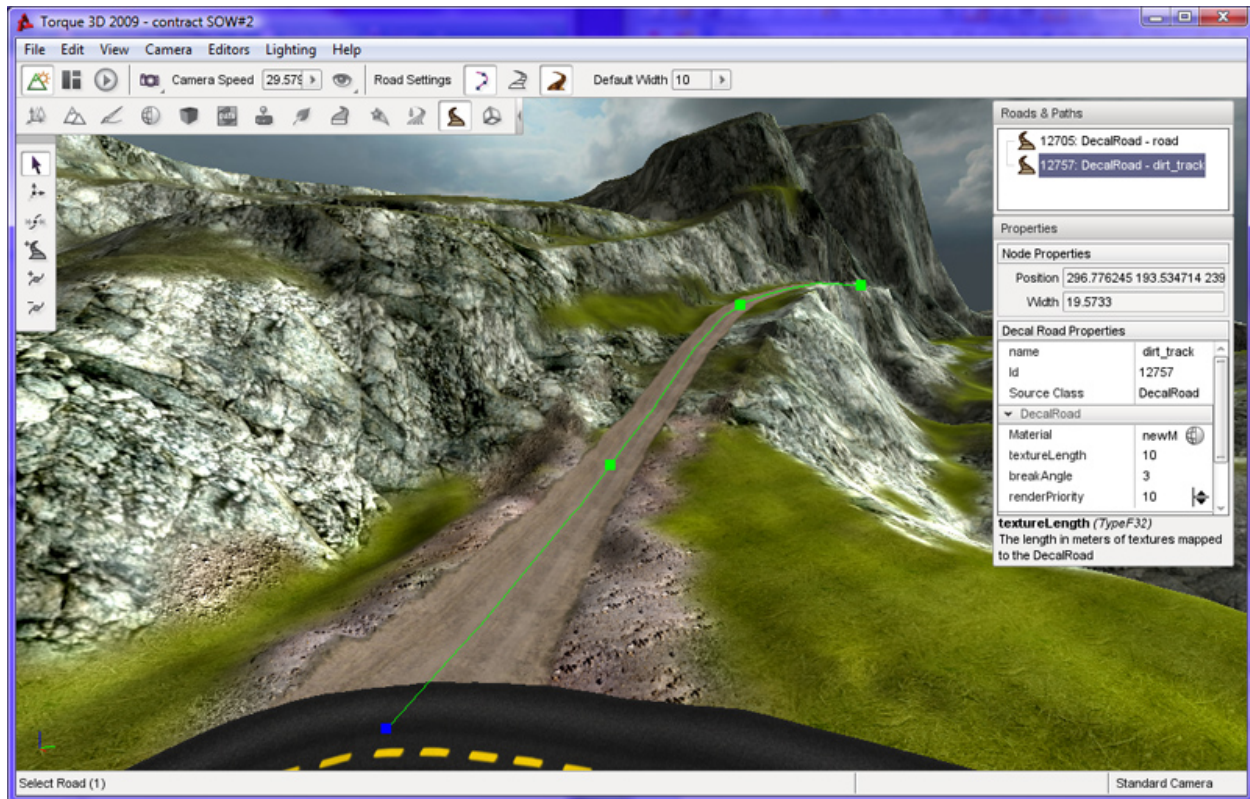
## 13.11 Decal Road Editor



The Road and Path Editor is used to create decal-based roads on your terrain. A decal-based road follows every contour of the terrain, unlike a Mesh Road which is a solid 3D object. By using the Road and Path Editor and a few choice materials, you can easily create dirt tracks, trails, paths, and simple roads. To create more complicate roads that rise above the terrain, span hills, or contain bridges use the Mesh Road Editor. The Road and Path Editor is a built-in WYSIWYG (What-You-See-Is-What-You-Get) editing tool which provides near real-time feedback so that you can see the changes and additions as you make them.

### 13.11.1 Interface

To access the Road and Path Editor activate it from the main menu of the World Editor by selecting Editors > Road and Path Editor. Alternatively, you can click the Road and Path icon from the World Editor Tool Selector Bar.



Whenever the Road and Path Editor is active three sections of the screen are updated to contain the editors tools. On the right side of the screen are the Roads and Paths pane and the Properties pane. At the top is the Roads and Paths pane which contains a list of all the decal-based roads currently in the level, if any are present. At the bottom is the Properties Pane which displays the properties of the currently selected road. At the left of the screen the Road and Path placement tools will appear which are used to create and modify your roads. At the top of the screen in the world editor tool bar, a new set of icons will appear after selecting the Road Editor. These icons and their associated values will enable you to quickly set up the width of the control points and modify the editor to show and hide some visual aids which can be used to guide your road placement.

There is no depth parameter for decal-roads as there is for Mesh based Roads. As mentioned earlier decal-based roads sit right on the terrain surface and follow the terrain exactly. They do not have their own geometry.

### 13.11.2 Adding a Decal Road

A decal-based road is created by placing a number of control points across the terrain. Each point can be edited for width at any time. By adjusting these points we have full control over how our road will look. The default width of control points can be set using the Default Width property on the Tool Settings Bar at the top of the editor window. Any new roads will be created using these settings until you change their values again.

To create a new road select the Create Road icon from the tool bar then click on the terrain with the left mouse button where you would like to start your road. Move the mouse away from the clicked location to see the results. Each time you click the terrain you will see three things:

1. a green square which represents the road location that you just placed
2. a blue square which represents the next location that the road will be if you press mouse button again
3. the road decals that were just placed

Depending upon the power of your computer there may be a delay between when you click the terrain and when the

decals appear. Move the mouse to the next point on the terrain that you wish your road to travel to and then click again. Continue moving and clicking until you are finished with the initial placement of your road.

To complete the road placement process press the `ESC` key. This action will exit the Create Road tool leaving your new road selected and ready for adjustments.

To abort a road creation operation without placing a road at all press the `ESC` key before selecting a second road point. Once a second road point has been placed the only way to remove the road completely is to delete it, as explained later.

### 13.11.3 Editing a Decal Road

The Road and Path Editor provides several tools for modifying roads after they have been created. If at any time you make a mistake with any tool, you can press `CTRL+Z` to undo it. As with road placement, depending upon the power of your computer there may be a delay between when you perform an editing action and when the change appears in the scene.

#### Selection Tool

Once you have created your initial road you may need to edit some or all of the control points. This tool will allow you to directly select any created point for further editing. To activate the Selection Tool click its icon on the Tool Selector bar. Note that the Road and Path Editor will automatically select this tool when you have finished creating a new road.

The selection tool allows two types of selection relating to roads:

- An entire road can be selected by clicking anywhere on a road other than one of its control points. This type of selection will result in the road being highlighted with a “spline”, which is a curved line that runs along the center line of the road, and a series of green squares which represent the road's control points. The only operations that can be performed on a road as a whole is deleting it. To delete an entire road press the `Del` key and confirm the operation using the dialog box that will pop up. Unlike a Mesh Road you can not move a decal road as a whole using either the Road and Path Editor or the Object Editor. Selecting a road allows you to see its centerline and its control points for individual selection and manipulation.
- Selecting a control point also causes the Properties pane on the right of the screen to be updated to display the current property values of the control point. The Node Properties section will display the position and width of the selected control point. Values can be directly entered into these fields to modify the point or the Move Point Tool and Scale Point Tool can be used to manipulate the point using the mouse.

#### Moving a Road

If at any time you are unhappy with the placement of a selected Road Mesh control point you can use the Move Tool to adjust its position. To activate the Move Tool click its icon in the Tool Selector bar.

This editor's move mode works a little bit different than the other editors as there is no gizmo over the selected control point when the tool is active. To move the selected control point: click the left mouse button on the control point; hold down the button; and drag it to a new position. The road decal will always follow the contour of the terrain. There may be a small delay as the editor updates the decal road.

#### Scaling a Road

If you feel that the road is not the correct width, or you just want to make some variations in a dirt track, you can use the Scale Point tool to change the width. This tool works in a similar fashion to the Move Point tool as there is no gizmo over the selected control point when the tool is active. To activate the Scale Point tool click its icon on the Tool Selector.

To change the width of the road select the control point you would like to scale; click the control point using the left mouse button; hold the button down; and drag the point to the left to reduce the width, or drag it to the right to increase the width. As with the Move tool there may be a small delay as the editor updates the decal road. Release the button to leave the road at that width at any time.

### Adding Extra Points

The Insert Point tool can be used to add extra points in a road to create a smoother curve. In order to insert a new point into a road the road must first be selected. See the Selection Tool above for details on how to select a road. To activate the Insert Point tool once a road has been selected click its icon on the Tool Selector bar. To place a new point on the selected road click on the road where you would like the new point to be placed. A new point will be added to the road and will immediately be the currently selected point as indicated by the blue square.

### Removing Points

The Remove Point tool can be used to delete a control from a road. In order to remove a new point from a road the road must first be selected. See the Selection Tool above for details on how to select a road. To activate the Remove Point tool click its icon on the Tool Selector bar. To remove a control from the selected road point click on the control point. This will remove only the selected point leaving all the others in place. No adjustments will be performed on the other existing control points.

## 13.11.4 Properties

The Properties pane on the right side of the screen can be used to configure a decal-based Road.

### Decal Road

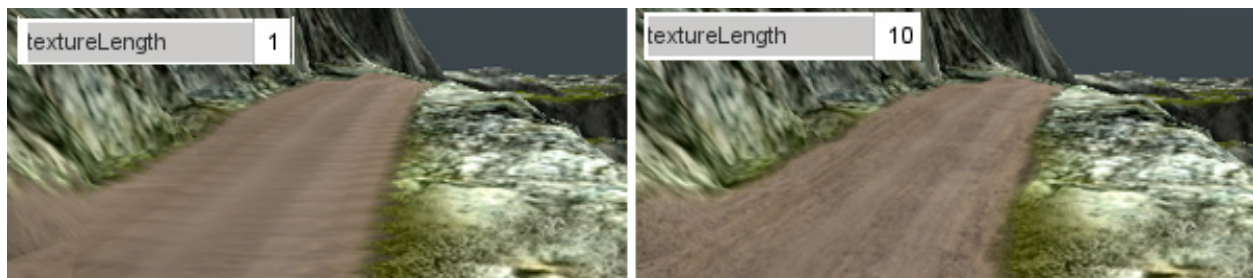
This section contains properties that control the roads appearance.

**Material** The texture assigned to this property will be used as the decal that displays on the terrain to represent the roads surface. Clicking the small round icon to it right will open the Torque 3D Material Selector window. From this window you can select a new material to assign to the Material property. For full details on how to use the Material Selector and how to create new materials see the Material Editor article.

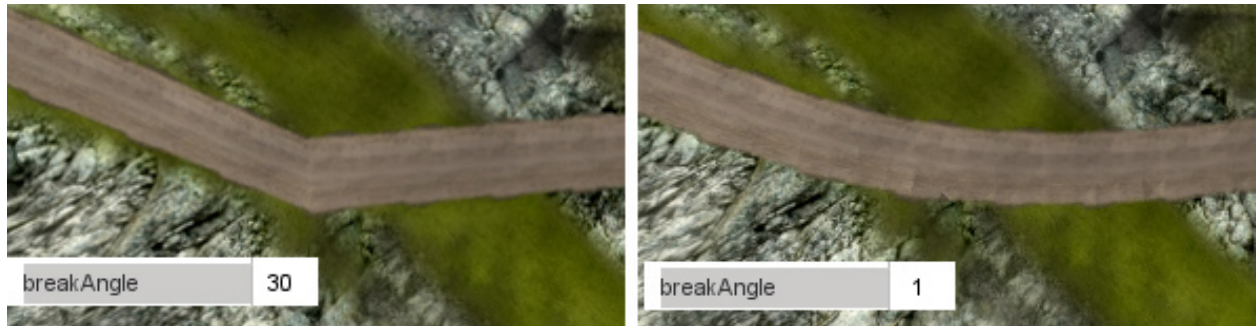
**Texture Length** The length the texture will be rendered at in meters, measured along the centerline of the road.

**Break Angle** Indicates the angle in degrees that the mesh roads spline will be subdivided into if its curve becomes greater than this threshold.

**Render Priority** Decal roads are rendered in descending order.





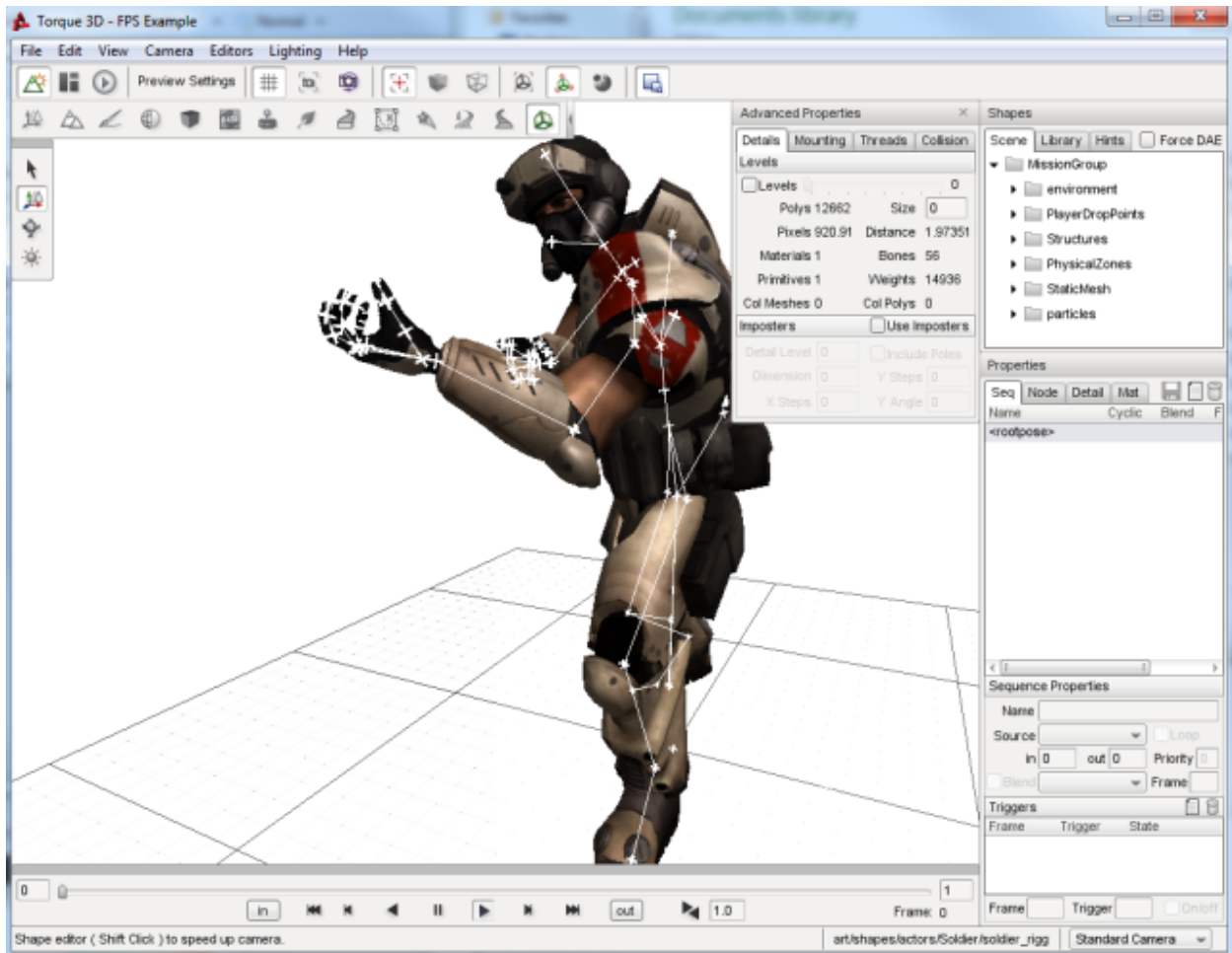


## 13.12 Shape Editor

The Shape Editor is used to view and edit the shapes that can be placed into your levels using the Object Editor. The Shape Editor can view and manipulate files in both the DTS (.dts) and COLLADA (.dae) formats. It can be used to quickly preview shapes before they are added to a level, and provides an easy way to add, edit and delete animation sequences, skeleton nodes, and rendering detail levels.

### 13.12.1 Interface

The Shape Editor can be activated from the main menu by selecting Editors > Shape Editor.



The Shape Editor interface consists of five primary sections whenever it is active.

**Shape Selector** The Shape Selector panel is used to choose a shape file for viewing and editing. It is composed of 3 tabs: Scene, Library and Hints. The Scene tab allows you to select a shape that has been placed in the current level. The Library tab allows you to browse and select any DTS or COLLADA shape from your project's art folder. Finally, the Hints tab displays information about which nodes and sequences are expected by Torque for a given type of shape object.

**Shape View Window** The main window shows a 3D view of the selected shape, and includes animation playback controls to play and single-step the selected sequence.

**Properties Window** The Properties panel is used to view and edit the sequences, nodes, details and materials in the shape.

**Advanced Properties Window** The Advanced Properties window displays Level-of-Detail information, and provides the ability to mount objects to other objects and animation thread control.

**Toolbar and Palette** The Shape Editor adds several buttons to the standard World Editor toolbar to control the 3D shape view, and uses the familiar select/move/rotate palette for node transform manipulation.

## 13.12.2 Shape Selection

To start using the Shape Editor, first you need to select a shape. There are three ways to do this:



1. Select an object in the World Editor, then activate the Shape Editor from the menu bar or the toolbar. If the object uses a DTS or COLLADA file, it will automatically be selected in the Shape Editor.
2. Select an object using the Scene tree in the Shape Editor. This view is the same as that used in the World Editor, and provides a convenient way to select objects that have already been placed in the level. Note that the Shape Editor only allows selection of objects that use DTS or COLLADA files; selection of Interior or ConvexShape objects will be ignored.
3. Select a shape file using the Library tab. This view is the same as that used in the World Editor Meshes tab, and allows you to browse the DTS and COLLADA assets in your project's art folder. This method allows you to view and edit shape files that have not yet been placed in the level.

The Force DAE option checkbox at the top of the Shapes panel forces Torque to load the COLLADA file, even if an up-to-date cached.dts file is present. Note that if the model is already present in the scene (and thus already loaded into the Torque Resource Manager), the Force DAE option will have no effect, as the shape will be opened from memory instead of from disk. This option is also available in the Editor Settings panel when working in the World Editor.

You will be prompted to save if there are unsaved changes in the current shape when a new shape is selected. The selected shape will appear in the View window, and listings of its sequences, nodes and materials will be displayed in the Properties window.

### 13.12.3 Shape Hints

The Hints tab in the Shape Selection window shows you which nodes and sequences are required for a particular type of shape to work with Torque.

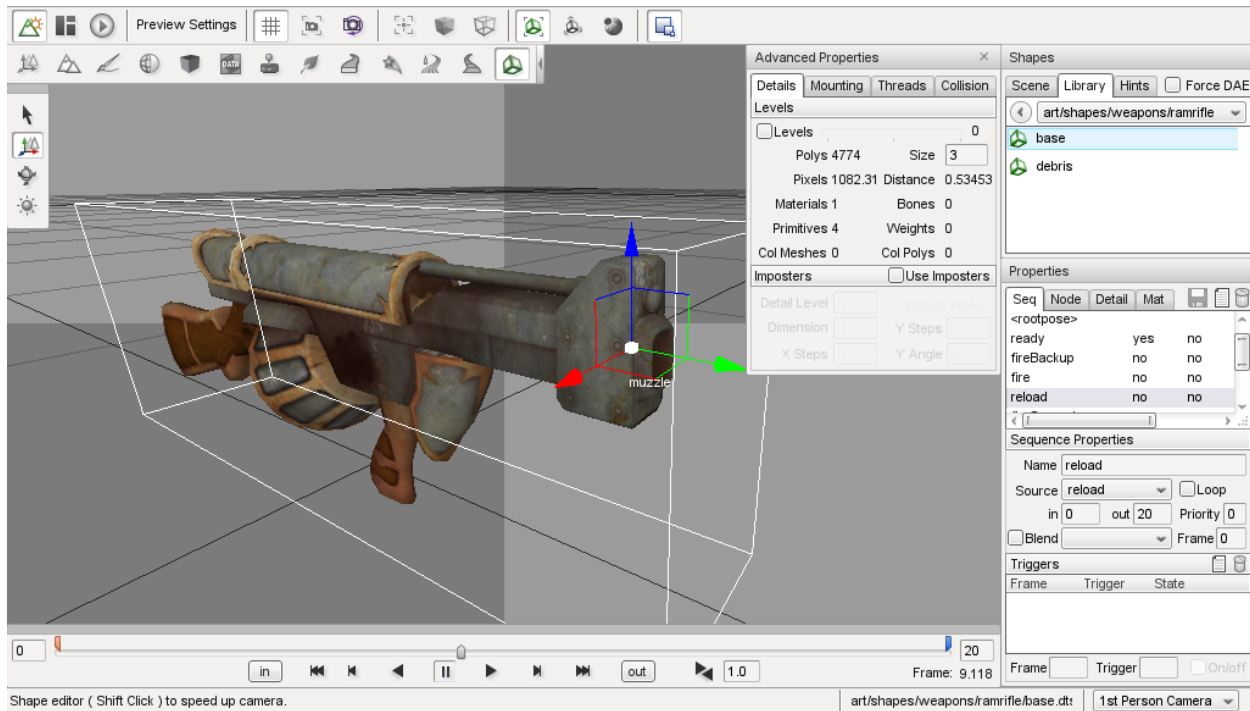
Simply select the desired object type from the dropdown menu and the list of required nodes and sequences will be displayed underneath. Items that are present in the selected shape will be marked with a tick mark. Hovering the mouse cursor over an item will display a short description of the item. Double-click the item to add it to the current shape.

Most items are optional - the shape will still load and run without a particular node or sequence, but the object may not perform correctly in-game. A Player object for example uses a node called cam as the 3rd person camera position. If this node does not exist, the shape origin is used instead, which will probably not be correct for most character shapes.

It is easy to extend the Shape Editor hints for custom object types by adding to the list in: `tools/shapeEditor/scripts/shapeEditorHints.ed.cs`.

### 13.12.4 Shape View

The Shape View window displays the shape as it would be seen in-game, and also provides helpful rendering modes such as transparent, wireframe and visible nodes.



The camera can be rotated by dragging the right mouse button, translated by dragging the middle mouse button (drag left+right buttons if your mouse does not have a middle button), and zoomed using the mouse wheel. Use the Camera->View menu (or the dropdown list in the bottom right corner) to switch between the Standard/Perspective view and the orthographic views (Top, Bottom, Left, Right, Front, Back).

Hovering the mouse over a shape node will display the node name, and left clicking a node in the view will select it in the Node Properties panel (and vice versa). Once a node is selected, its transform can be modified by dragging the 3D gizmo similar to how objects are positioned in the World Editor.

## Animation Controls

At the bottom of the Shape View window are the animation playback controls:



As well as allowing the selected sequence to be scrubbed with the slider, stepped one frame at a time, or played normally, the start and end frames of the sequence can be easily modified to facilitate sequence splitting or to correct off-by-one-frame looping errors. Sequence triggers appear as thin, vertical bars at the appropriate frame (as shown in the image above).

Pressing the in or out button, or modifying the text box directly (remember to hit Return to apply the change), will set the start or end frame of the sequence to the current slider position.

### 13.12.5 Properties Window

The Properties window is where you can view and edit the sequences, nodes, detail levels and materials in the shape. The top right corner has three buttons, which do the following:

- Save the shape
- Add a new sequence, node or detail; and

- Delete the selected sequence, node or detail

### Sequences Tab

The Sequences tab (displayed as “Seq” onscreen) lists the sequences available in the shape, as well as a number of different properties about the selected sequence. In addition, the ‘root’ (non-animated) pose can be selected for display in the Shape View window. The sequence properties available to view and edit are:

**Name** The name of the sequence. To rename a sequence, simply edit the value and press Enter.

**Source** The source animation data for the sequence, for example, the path to an external DSQ file, or the name of another sequence in the shape.

**Priority** The priority of the sequence. This determines which sequence will take precedence when more than one sequence is attempting to control the same node.

**in** The first frame in the source sequence used for this sequence. Change this value to clip the start of the source sequence. This sequence will then start on the specified frame of the source regardless of what other frames may be before it in the source sequence.

**out** The last frame in the source sequence used for this sequence. Change this value to clip the end of the source sequence. This sequence will then end on the specified frame of the source regardless of what other frames may be after it in the source sequence.

**Loop** Flag indicating whether this sequence loops around when it reaches the last frame.

**Blend sequence** Name of the sequence to use as a reference for generating blend transforms.

**Blend flag** Flag indicating whether this sequence is a blend, that is, whether it can be played over top of another sequence.

**Blend frame** Frame in the Blend sequence to use as a reference.

**Triggers** The list of triggers in the sequence. Select a trigger and edit the values to modify the trigger.

### Adding a sequence

An important feature of the Shape Editor is the ability to add new sequences to a shape from external animation files (DSQ or DAE). This allows animations to be shared by shapes that have a common skeleton (such as character models).

To add a new sequence click the New Sequence button. If a sequence is currently selected when the button is clicked, the new sequence will use that selected sequence as its initial source for animation keyframes. You can change the Source using the dropdown menu to select a different sequence, or to Browse for an external DSQ or DAE file. If the <rootpose is selected, pressing the New Sequence button will open the Browse window automatically.

Once the sequence has been created, you can edit its properties - including the start and end frames - using the Sequence Properties panel.

### COLLADA <animation\_clips

Currently, very few 3D modeling packages support the COLLADA <animation\_clip element, which means a model with several animations will appear to have only a single sequence (or ‘clip’) when loaded into Torque. The Shape Editor allows you to split this single animation into multiple sequences by specifying different start and end frames for each sequence. The procedure for splitting animations is as follows:

1. Select the combined animation sequence (usually called ambient).

2. Press the New Sequence button to make a copy of this sequence, then rename the new sequence as desired.
3. Use the animation slider in the 3D view to find the desired keyframe that you want the new split sequence to start at. Press the In button to set the start frame.
4. Use the animation slider in the 3D view to find the desired keyframe that you want the new split sequence to stop at. Press the Out button to set the start frame.

## Blend Animations

A blend animation is special in that it stores node transforms relative to a reference keyframe, instead of absolute transforms like other animations. This allows the sequence to be played on top of another sequence without forcing the animated nodes to a particular position.

The Shape Editor allows you to set and clear the blend flag for a sequence, as well as change the reference keyframe if desired. Each of these operations requires that a valid reference sequence and reference frame number is specified.

For example, most Player characters will have a blended look animation. The animation is a blend so that the character's head can be made to look around while also doing something else (like running or swimming). To make the look animation a blend, first we set the reference sequence (e.g. root) and frame (e.g. 0), then we can set the blend flag.

## Nodes Tab

The nodes tab shows the node hierarchy and various properties of the selected node. The node properties available to view and edit are:

**Name** The name of the node. To rename, simply edit the value and press Enter.

**Parent** The parent of the node in the hierarchy. A new parent can be selected from the dropdown menu if desired.

**Transform** The position and orientation of the node. Node transforms can be edited in either World mode (where the transform is relative to the shape origin), or Object mode (where the transform is relative to the node's parent). Node transforms can also be edited visually in the Shape View window by selecting the node and dragging the axis gizmo, similar to how object transforms are edited in the World Editor. In World mode, the gizmo uses the global X,Y,Z axes, while in Object mode, the gizmo uses the node relative X,Y,Z axes (useful for seeing which way the node points for eye or cam nodes)

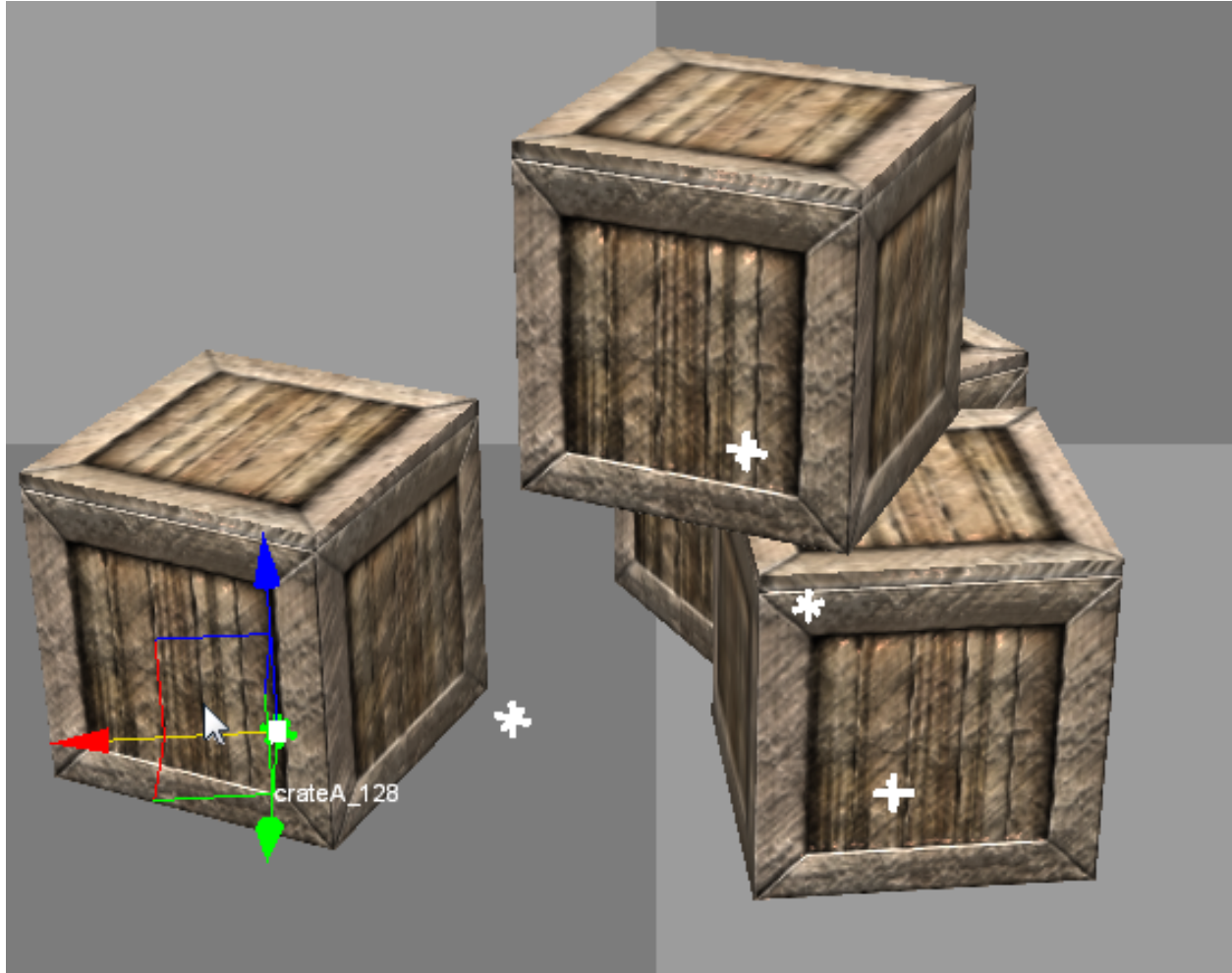
## Editing Nodes

The Shape Editor allows shape nodes to be added, moved, renamed and deleted.

To add a node, simply press the New Node button in the top right corner of the Properties panel. If a node is currently selected, it will automatically be used as the initial parent for the new node. A new parent node can be selected using the dropdown menu. Renaming the node is as simple as typing a new name in the edit box and pressing Enter to apply the change.

There are two ways to edit node transforms: The first way is to manually edit the position and rotation values in the Node Property panel. This method is most useful when trying to set an explicit value. For example, you may require that a node be offset by exactly 2 units in the X direction from its parent node. Node transforms can be specified as either relative-to-parent (Object mode) or relative-to-origin (World mode).

The second way to edit node transforms is in the 3D Shape View. Simply select the desired node in the 3D view or in the node tree then drag the axis gizmo to the correct position and orientation.



It should be noted that the Shape Editor tool is not intended as a replacement for a fully-functional 3D modeling application, and as such, it only allows the non-animated transforms of the shape nodes to be edited. That is, the node transforms when the shape is in the root pose. You cannot use the Shape Editor tool to define new animation keyframes. For this reason, it is recommended to edit node transforms only when the <rootpose is selected in the Sequence Properties list. Node transforms can be edited when any sequence is selected, but the results may not be as expected, since animated parent nodes will affect the node transform as seen in the Shape View.

To delete a node, simply select it in the 3D Shape View or the node tree and press the Delete button in the top right corner of the Properties panel. Note that deleting a node will also delete all of its children.

### Detail Tab

The Detail tab of the Properties pane lists the detail levels and associated geometry (meshes) in the shape, as well as allowing certain properties to be edited.

If the Levels checkbox is checked on the Details tab in the Advanced Properties window, then selecting a mesh or detail level in the tree will switch to that detail level in the 3D view. The bounding box for the selected object can be displayed using the Toggle Bounding Box button on the toolbar. To view all collision geometry (as wireframe) no matter which detail level is selected, use the Toggle Collision Mesh button on the toolbar.

**Name (top-left field in Detail/Object Properties section)** The name of the mesh or detail level. Note that an object may contain multiple level-of-detail meshes. Changing the object name will change the name of all meshes for that object.

**Size (top-right field in Detail/Object Properties section)** The pixel size of the mesh or detail level. Changing the size for a mesh will move it from one detail level to another (creating a new detail level if required). Changing the size of a detail level will change the size for all meshes in that detail.

**Billboarding** Allows a mesh to be set as a billboard.

**Object Node** The name of the node this object is attached to. Changing this value will change the node for all level-of-detail meshes of the object.

**Import Shape into...** Import geometry from another shape file. See Importing Geometry for more details.

**Re-compute bounds** Recalculate the shape bounding box using the current pose and detail level.

The Shape Editor also allows meshes to be hidden inside the 3D view (equivalent to the `ShapeBase::setMeshHidden` script method). Simply right-click a mesh in the detail tree to toggle the hidden state. Note that all detail-level meshes for that object share the same hidden state, so hiding the Head 2 mesh will also hide any other meshes for the Head object.

## Importing Geometry

The Shape Editor allows you to import geometry from another DAE or DTS file into the current shape via the “Import Shape into...” button. Geometry in the external file may be added to the currently selected detail level or to a new, automatically created detail level. The size of the new detail level can be edited after the geometry has been added.

The dropdown to the right of the “Import Shape into...” button has two options:

1. The current detail option is useful when combining separate files that you want to be rendered at the same detail level. For example, if a player character was split into body part models as follows:

```
player_torso.dts
player_head.dts
player_left_arm.dts
player_right_arm.dts
player_left_leg.dts
player_right_leg.dts
```

To combine the models, open `player_torso.dts` in the Shape Editor, switch to the Details tab then Import each of the other files into the current detail. When the shape is rendered, all body parts will be rendered together.

```
+--base01
  +-start01
    +-Torso           Object Torso with details: 2
    +-Head            Object Head with details: 2
    +-LeftArm         Object LeftArm with details: 2
    +-RightArm        Object RightArm with details: 2
    +-LeftLeg         Object LeftLeg with details: 2
    +-RightLeg        Object RightLeg with details: 2
```

2. The new detail option is useful when combining separate files that represent different detail levels of the same shape. For example, a vehicle model may have the following detail level files:

```
truck_lod400.dts
truck_lod200.dts
truck_lod60.dts
truck_col_lod-1.dts
```

To combine the models, open `truck_lod400.dts` in the Shape Editor, switch to the Details tab then Import each of the other files into new detail levels. The single truck object now has 3 visible detail levels (at pixel sizes 400, 200 and 60), and a single, invisible collision detail level (size -1).



```
+--base01
+-start01
+-Truck           Object Truck with details: 400 200 60
+-Collision       Object Collision with details: -1
```

Note that when the new detail option is selected, the Shape Editor examines the filename of the imported model to determine the detail size. If the filename ends in “\_LODX” (where X is a number), the new detail level will be created with size X. The detail level size can be changed after import if needed.

## Materials Tab

The Materials tab (labelled as “Mat” in the window) shows the materials specified in the shape, as well as the Material each one is mapped to.

Selecting a material while the Highlight selected Material option is set will highlight all of the primitives that use the material in the shape view. Pressing Edit the selected Material will open the Material Editor dialog, allowing you to modify the Material properties and view the results in real-time in the Shape Editor view window. Hit the Back to Previous Editor button in the upper-left corner of the Material Properties pane to return to the Shape Editor. Do not forget to save any changes you make before returning to the Shape Editor.

## 13.12.6 Advanced Properties Window

The Advanced Properties Window allows you to further change the settings of the model loaded in the shape editor.

### Details Tab

The detail size and mesh characteristics for each LOD need to be carefully determined in order to reduce the visual artifacts associated with switching and rendering detail levels. The Details Tab of the Advanced Properties window provides a convenient way to view and edit detail levels without having to re-export the model. It also allows non-rendered collision and LOS-collision detail levels to be visualised. The detail level properties available to view and edit are:

**Levels** When set, the current detail level is selected by moving the slider. When unset, the current detail level is selected based on the camera distance, in the same way as LOD is handled in-game.

**Current DL** The index of the currently selected detail level is shown to the right of the slider track.

**Polys** The number of polygons (triangles) in the current detail level.

**Size** The size (in pixels) above which the current detail level will be selected. This value can be edited to change the size of the current detail level (remember to press Return after editing the value to apply the change).

**Pixels** The current size (in pixels) of the shape. This value is an approximation based on the shape bounding box, viewport height, and camera distance.

**Distance** The distance from the shape origin to the camera.

**Materials** The number of different materials used by all meshes at the current detail level.

**Bones** The number of bones used by all skinned meshes at the current detail level. Non-skinned meshes will display 0 for this value.

**Primitives** The total number of primitives (triangle lists, strips or fans) in all meshes at the current detail level. This is the minimum number of draw calls that will be executed for this detail level.

**Weights** The number of vertex weights used by all skinned meshes at the current detail level. Non-skinned meshes will display 0 for this value.

**Col Meshes** The total number of collision meshes in this shape.

**Col Polys** The total number of polygons (triangles) in all collision meshes in this shape.

The Details Tab of the Advanced Properties window allows imposter detail levels to be added and edited. Imposters are a series of snapshots of the object from various camera angles which are rendered instead of the object when this detail level is selected. An imposter detail level is usually the last visible detail level (smallest positive size value).

## Mounting Tab

The Mounting Tab of the Advanced Properties window allows you to attach other models to the main shape to visualise how they would look in-game, or to fine tune the position and rotation of mount nodes. When a model is mounted, it inherits the position and rotation of the node it is mounted to and will animate along with it. Press the Mount New Shape or Delete Mounted Shape buttons to add or remove mounted models respectively. The following properties of the selected mount can be modified:

**Shape** DTS or DAE model file to mount.

**Node** Node (on the main shape) to mount to. Only nodes that follow the mountX and hubX naming conventions will appear here.

### Type

- **Object:** Mount the model as a SceneObject. The model's origin is attached to the selected mount node. This is equivalent to mounting the object using the following script call:

```
%obj.mountObject(%obj2, 0);
```

- **Image:** Mount the model as a ShapeBaseImage. The model's mountPoint node (or origin if not present) is attached to the selected mount node. This is equivalent to mounting the object using the following script call:>

```
%obj.mountImage(%image, 0);
```

- **Wheel:** Mount the model as a WheeledVehicle tire. The mounted shape's origin is attached to the selected mount node, and it is rotated to face the right way (whether on the left or right side of the vehicle). This is equivalent to mounting the object using the following script call:

```
%car.setWheelTire(0, %tire);
```

**Sequence** Select a sequence for the mounted shape to play. Playback can be controlled using the slider and play/pause button to the right of the sequence dropdown box.

## Threads Tab

The Threads Tab of the Advanced Properties window allows you to set up threads to play multiple sequences simultaneously, and to view transitions between sequences. A set of animation sequence playback controls that mirror the main animation controls are provided as a convenience so you don't have to mouse too far to test out a new thread. The mini-timeline slider is also used to indicate sequence transition information.

**Thread** The index of the thread. Press the Add New Thread or Delete Selected Thread buttons to add or remove threads respectively. If the shape contains any sequences, there will always be at least one thread (index 0) defined.

**Sequence** Select the sequence for this thread to play. Changing the selected sequence while the thread is playing (and transitions are enabled) will cause a transition to the new sequence.

**Transition flag** If enabled, changing the selected sequence for the thread will cause a transition from the current pose to the target pose. During the transition period, node transforms are smoothly interpolated towards the target pose. If transitions are disabled, changing the selected sequence for the thread will switch node transforms to the new sequence immediately.

**Transition lasts** Transition duration in seconds. The default for Torque 3D is 0.5.

**Transition to** Selects the start frame in the target sequence; the target sequence begins playing from this point. When slider position is selected the target sequence will play from wherever the mini-timeline slider has been set. Torque 3D defaults to having the new sequence start at position 0.0 so it is likely that you'll want to keep the mini-timeline slider all the way to the left when in this mode. When synched position is selected, the new sequence will start playing at the same position along the timeline as the currently playing sequence. While in this mode, the mini-timeline slider will change from yellow to red during the transition period.

**Target anim** Controls whether the target sequence plays during the transition period. When plays during transition is selected the target sequence will play during the transition; node transforms will be interpolated towards the changing target pose. When pauses during transition is selected the target sequence will not play during the transition, but will start once the transition has ended. Node transforms will be interpolated towards the initial target sequence frame.

## Collision Tab

The Shape Editor can auto-fit geometry to a part or the whole of the shape for use in collision checking. Each time the settings are changed, the geometry in detail size -1 is replaced with the new auto-fit geometry. The node Col-1 (and any child nodes) may also be modified.

**Fit Type** The type of mesh to auto-fit for this collision detail (see table below for details).

**Fit Target** The geometry used to generate the auto-fit mesh. The target is either 'Bounds' (fit to the whole shape) or one of the shape sub-objects.

**Max Depth** For convex hull auto-fit meshes, this specifies the maximum decomposition recursion depth. Increase this value to increase the number of potential hulls generated.

**Merge %** For convex hull auto-fit meshes, this specifies the volume percentage used to merge hulls together. Increase this value to make merging less likely, and thus increase the number of final hulls.

**Concavity %** For convex hull auto-fit meshes, this specifies the volume percentage used to detect concavity. Decrease this value to be more sensitive to concavity (and thus more likely to split a mesh).

**Max Verts** For convex hull auto-fit meshes, this specifies the maximum number of vertices per hull. Increase this value to produce more complex (and CPU expensive) hulls.

**Box %** For convex hull auto-fit meshes, this specifies the maximum volume error below which a hull may be converted to a box. Increase this value to allow more hulls to be converted to boxes.

**Sphere %** For convex hull auto-fit meshes, this specifies the maximum volume error below which a hull may be converted to a sphere. Increase this value to allow more hulls to be converted to spheres.

**Capsule %** For convex hull auto-fit meshes, this specifies the maximum volume error below which a hull may be converted to a capsule. Increase this value to allow more hulls to be converted to capsules.

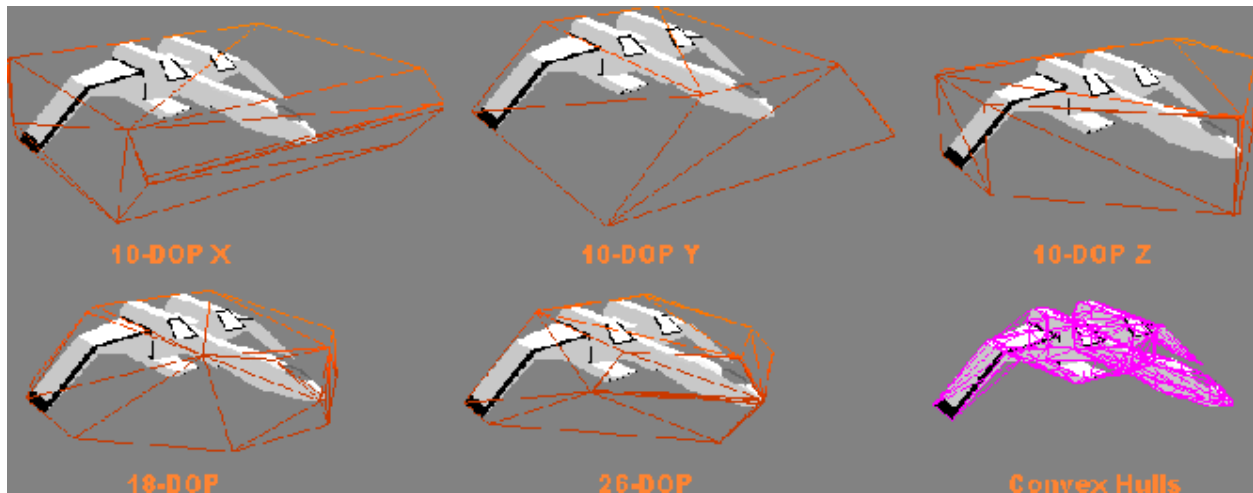
**Update Hulls** Re-compute convex hulls using the current parameters.

**Revert Changes** Revert convex hull parameters to the values used for the most recent hull update.

The following types of geometry can be generated. The Box, Sphere and Capsule types are generally the most CPU efficient, and are converted to true collision primitives when the shape is loaded. The other types are treated as convex triangular meshes - the more triangles, the more expensive it is to test for collision against the mesh.

Type	Description
Box	Minimum extent object aligned box
Sphere	Minimum radius sphere that encloses the target
Capsule	Minimum radius/height capsule that encloses the target
10-DOP	Axis-aligned box with four edges bevelled; you can choose X, Y or Z aligned edges
18-DOP	Axis-aligned box with all edges bevelled
26-DOP	Axis-aligned box with all edges and corners bevelled
Convex Hull	Set of convex hulls

The k-DOP (K Discrete Oriented Polytope) types push ‘k’ axis-aligned planes as close to the mesh as possible, then form a convex hull from the resulting points as shown below.



The Convex Hull fit type performs a convex decomposition of the target geometry to generate a set of convex hulls. The basic algorithm is described here. For each hull that is produced, the hull volume is compared to the volume of a box, sphere and capsule that would enclose the hull. The hull is replaced with the primitive type that is closest in volume to the hull with volume % difference less than Box, Sphere or Capsule % respectively. If none of the primitive volumes are less than their respective error setting, the hull will be retained as a triangular mesh.

### 13.12.7 Shape Editor Settings

The Shape Editor settings dialog can be accessed from the main menu by selecting Edit Editor Settings, and allows the appearance of the editor to be customized. These settings are persistent and will be automatically saved and restored between sessions.

### 13.12.8 Saving Changes

The Shape Editor does not modify the DTS or COLLADA asset file directly. Instead, changes made in the editor are saved to a TSShapeConstructor object in a separate TorqueScript file. This file is automatically read by Torque before the asset is loaded, meaning you can safely re-export the DTS or COLLADA model without overwriting changes made in the Shape Editor tool. The change set will be re-applied to the shape when it is next loaded by Torque.

If needed, you can also re-edit the generated TSShapeConstructor object, either manually with a text editor, or by using the Shape Editor tool again.

To save changes to the current shape, simply press the save button in the top right corner of the Properties window. The script filename is the same as the DTS or COLLADA asset filename, only with a .cs extension. For example, saving changes to ForgeSoldier.dts would save to the file ForgeSoldier.cs in the same folder.

The Shape Editor TSShapeConstructor object may also be accessed directly from the console. For example:

- Dump the shape hierarchy to the console (handy for debugging shape issues):

```
ShapeEditor.shape.dumpShape();
```

- Save the modified shape to DTS (instead of saving the change-set to a TSShapeConstructor script):

```
ShapeEditor.shape.saveShape("myShape.dts");
```

- Set ground transform information (not yet available in Shape Editor UI):

```
ShapeEditor.shape.setSequenceGroundSpeed("run", "0 4 0", "0 0 0");
```

## 14.1 Building Terrains

### 14.1.1 Introduction

In this tutorial, we are going to create a lush valley using sample assets provided by Sickhead Games. For this guide, the terrain will be created by importing a heightmap, opacity maps, and creating new materials.

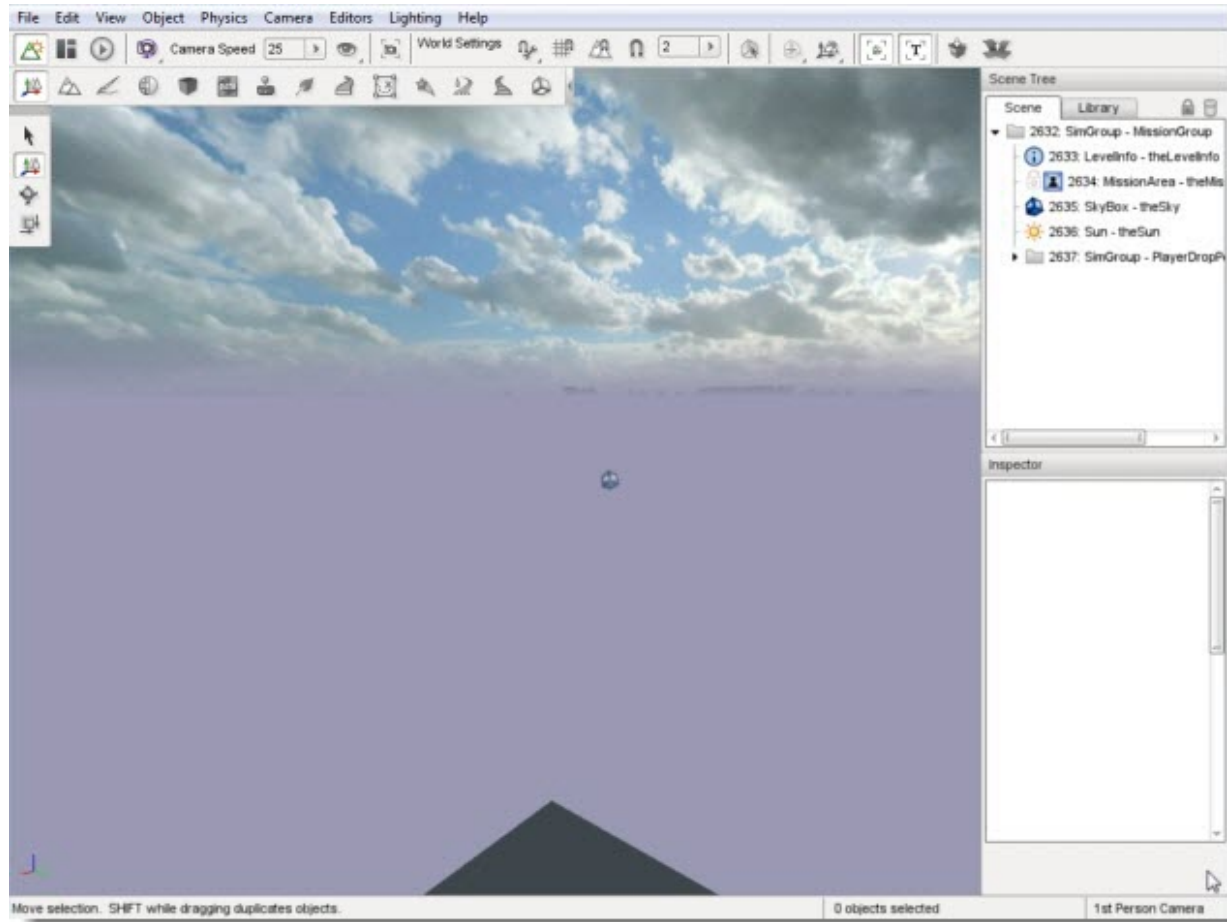
### 14.1.2 Setup

This article was written using a newly generated project with the Full Template, which ships with plenty of free assets for testing and learning. For this specific tutorial, you will want to download a zip file containing additional assets for testing: [CLICK HERE TO DOWNLOAD THE ZIP FILE](#).

None of the modifications you are about to make are required for future tutorials, so feel free to create a new level or use an existing one for testing. As long as you have access to existing materials, you are good to go.

You will want to remove any existing TerrainBlocks since we will be creating one from scratch. Go ahead and delete any blocking objects, so you have a clean view.





### 14.1.3 Heightmap, Opacity Layer, Terrain Textures

To get high-quality and professional looking terrain, you will want to use a 3rd party external tool. Examples include L3DT and GeoControl. These tools allow you to generate extremely detailed heightmaps that can be imported by Torque 3D and generate terrain data.

Several assets are required to successfully import and render a high quality heightmap. Most terrain generating applications provide proper exporters to get the job done. First, we will cover what these assets are. The follow example assets were provided by [Russell Fincher](#) at [Sickhead Games](#). These files are available for download in the setup section of this tutorial.

The primary asset required is the heightmap, which is an image used to store elevation data rendered in 3D by the engine. The heightmap itself needs to be a 16-bit greyscale image, power of two, and square. The lighter an area of a heightmap, the higher the elevation will be in that terrain location.

#### Example Heightmap



Next, you will want to use an opacity map. This map acts as a mask, which is designed to assign opacity layers. Opacity layers need to match the dimensions of the heightmap. 512x512 heightmap can only use a 512x512 opacity map.

If the opacity map is a RGBA image, four opacity layers will be used for the detailing (one for each channel). If you use an 8-bit greyscale image, only a single channel. You can then assign materials to the layers. This allows us to have up to 255 layers with a single ID texture map, saving memory which we can apply to more painting resolution.

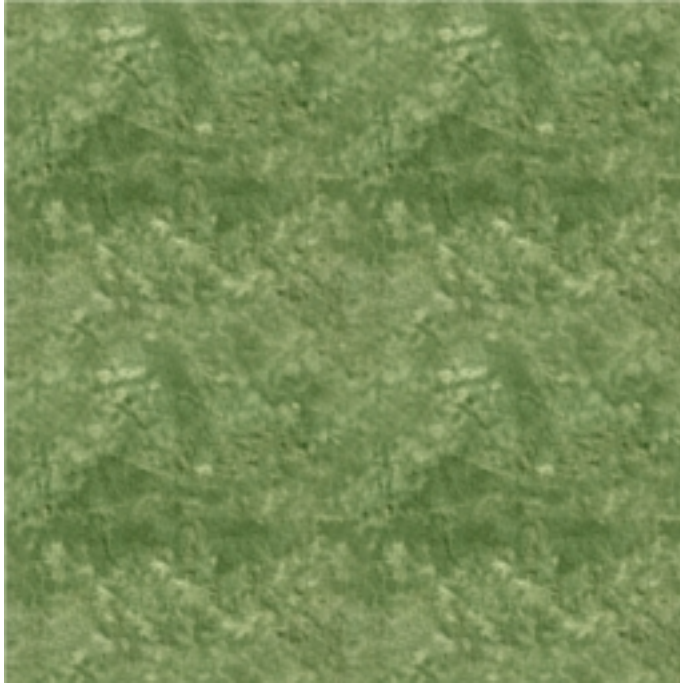
Notice that the following example Opacity Map resembles the original heightmap.

#### **Example Opacity Map**

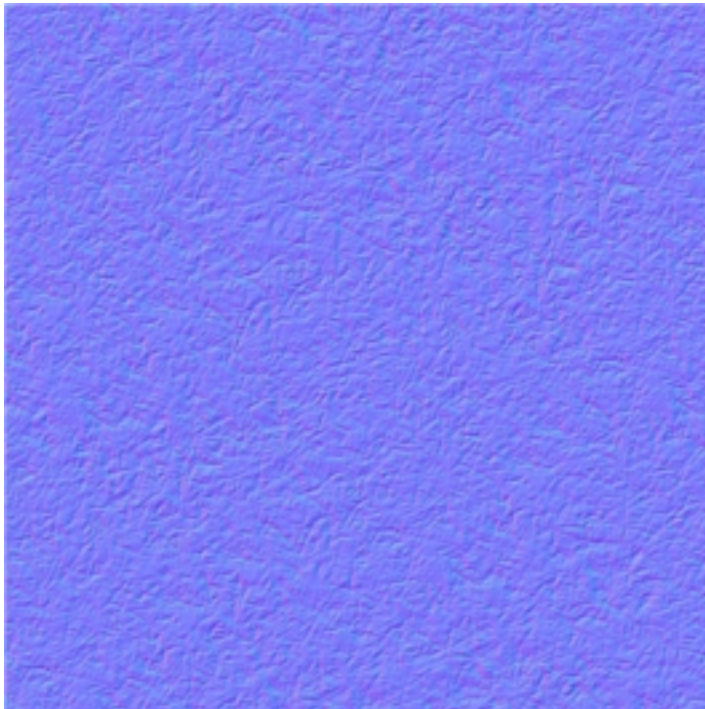


Finally, of course we want to use textures to paint the terrain. Instead of hand painting them, the opacity layer will automatically assign textures based on what channel they are loaded into. You will want to have three textures: a base (diffuse), a normal map, and a detail mask.

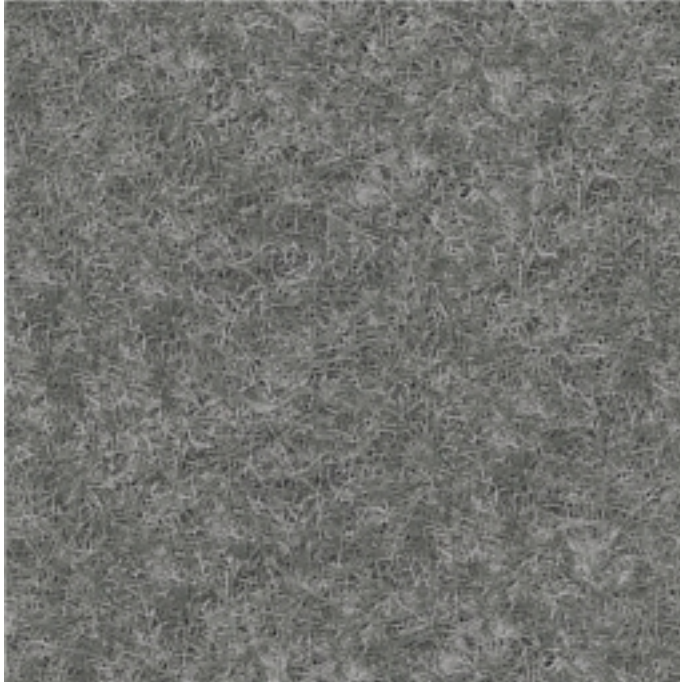
### Diffuse



### Normal



### Detail



The base represents the color and flat detail of the textures. The normal map is used to render the bumpiness or depth of the texture, even though it is flat. Finally, the detail map provides up-close detail, but it absorbs most of their colors from the base map.

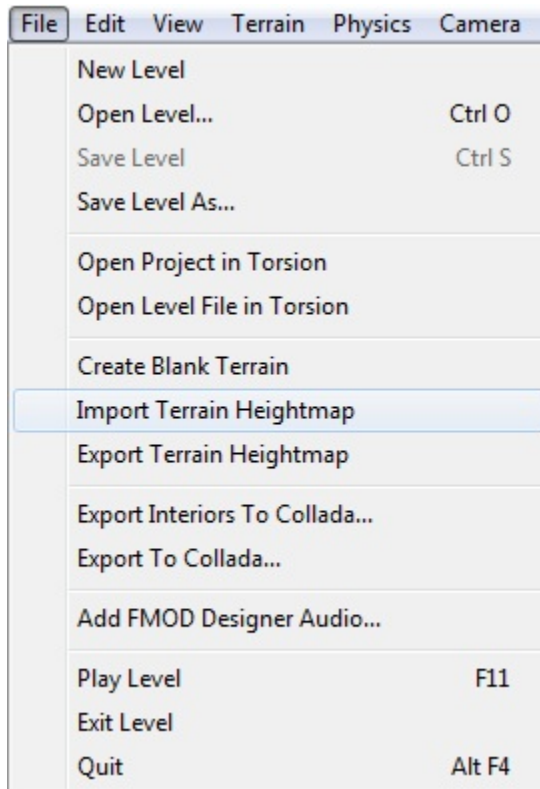
#### 14.1.4 Importing A Heightmap

Now that you know what assets are required, we are going to import our first heightmap. What we are going to do is create a highly detailed valley scene, with snowcapped mountains. Since this section focuses on the World Editor, and not 3rd party tools, you are going to use sample assets. This will save time and allow you to learn the World Editor functionality first.

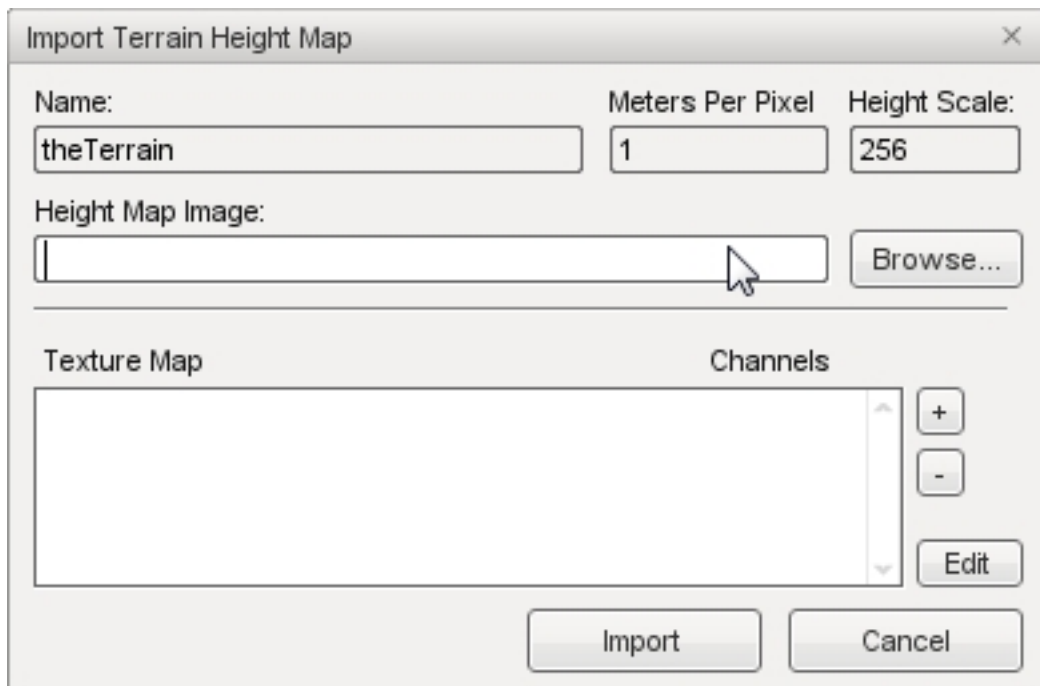
If you do not have the required files, they are available for download in the setup section of this tutorial. Again, these high quality assets were provided by [Russell Fincher](#) of [Sickhead Games](#) - Thanks Sickhead Games!

Create a folder in the `game/art/terrains` directory of your project called “sampleTerrain.” Unzip the contents of the [file you downloaded](#) into this new folder. You should have two heightmaps, identical except for varying resolution. You will also receive three sets of textures and opacity maps.

With your blank room running in the World Editor, click on File->Import Terrain Heightmap



A floating dialog will appear and allow you to setup your new terrain before importing it.



**Name:** If you specify the name of an existing TerrainBlock in the dialog, it will update that TerrainBlock and its associated .ter file. Or else it creates a new one.

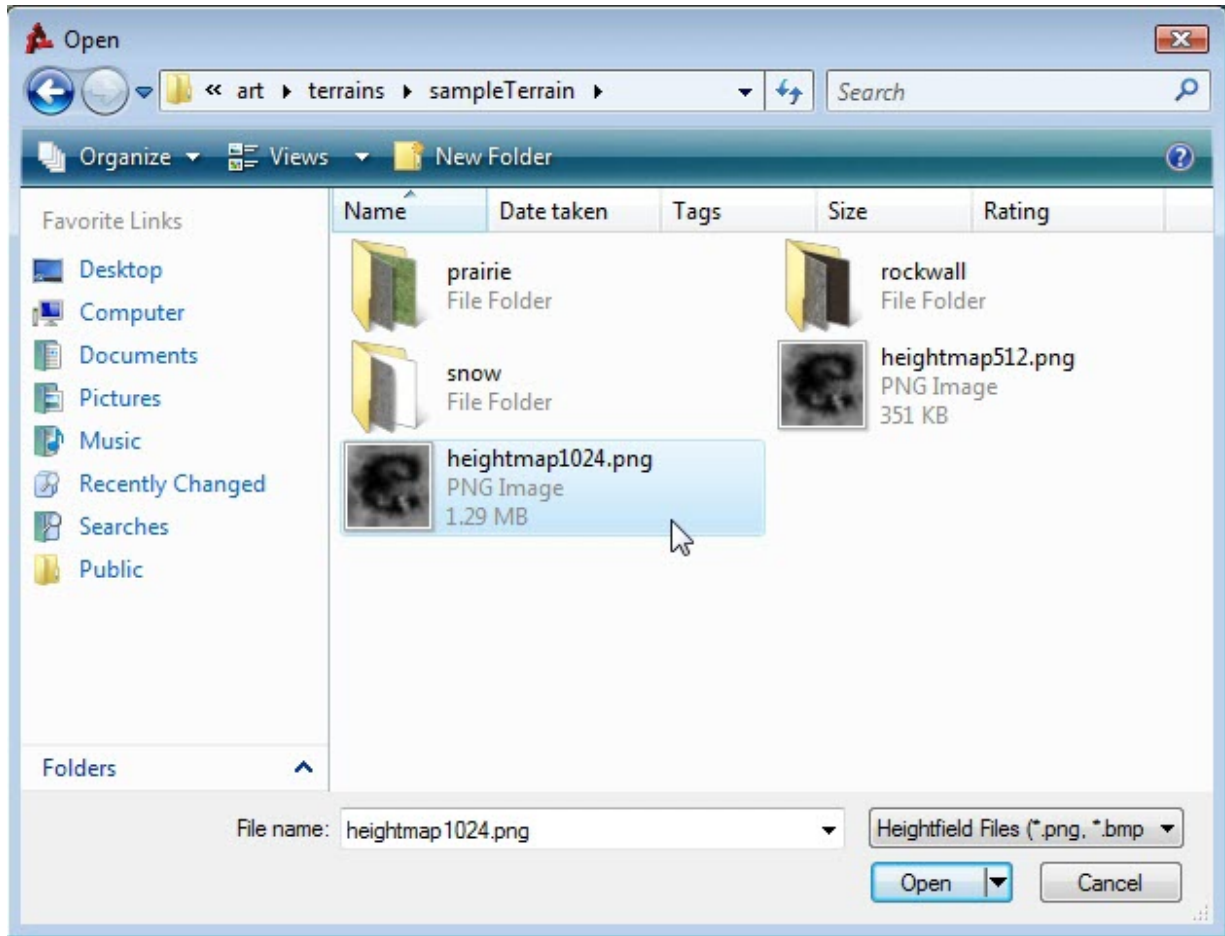
**Meters Per Pixel:** What was the TerrainBlock SquareSize, which is a floating point value that does not require power of 2 values.

**Height Scale:** The height in meters you want white in the heightmap to be.

**Height Map Image:** File path to .png or .bmp heightmap itself. Remember, this needs to be a 16-bit greyscale image, power of two, and square.

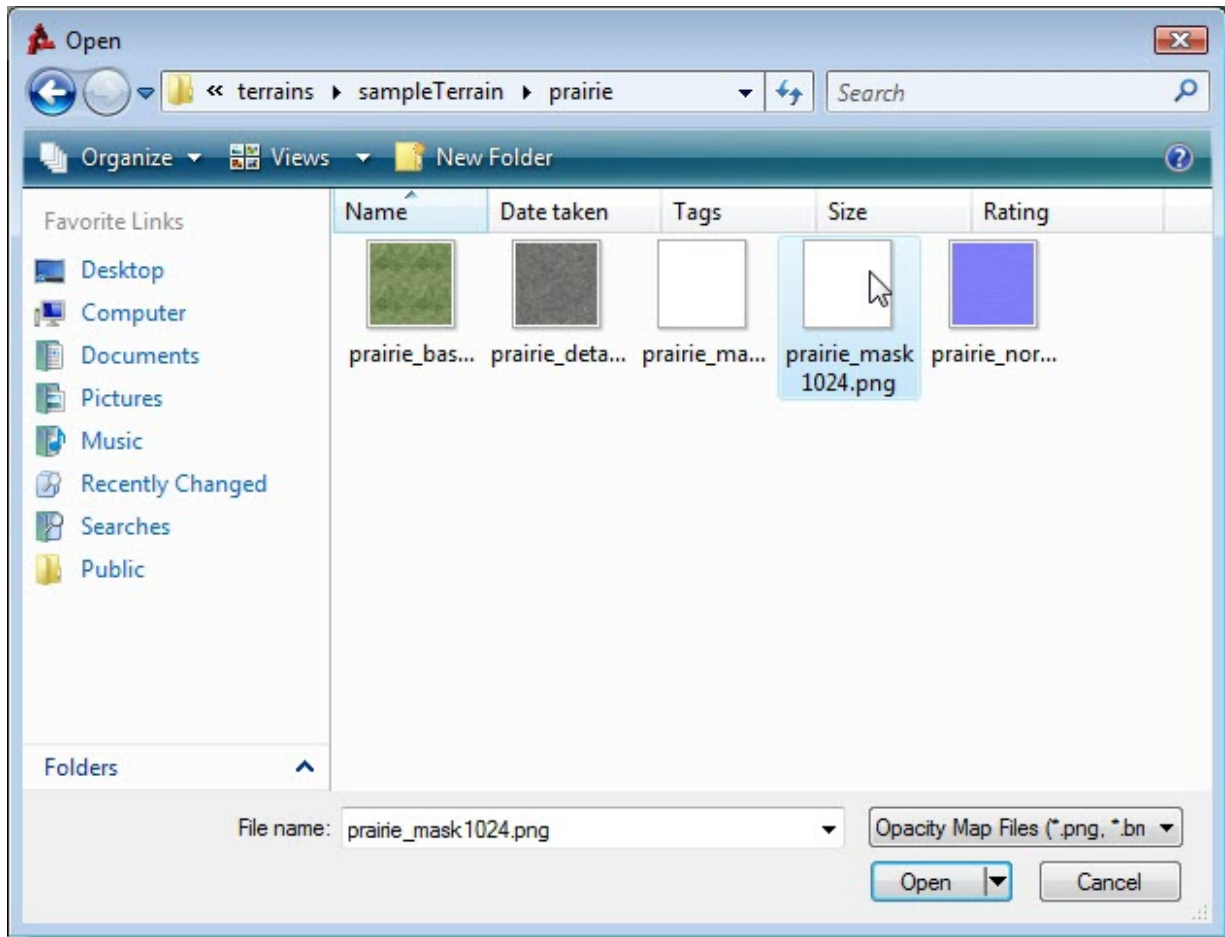
**Texture Map:** This involves opacity layers, which need to match the dimensions of the heightmap. If you add an RGBA image it will add 4 opacity layers to the list, one for each channel. If you add an 8-bit greyscale image, it will be added as a single channel. You can then assign materials to the layers. If you do not add any layers the terrain will be created with just the Warning Material texture.

Keep the name default value, theTerrain. Click the browse button near Height Map Image to open a file browser dialog. Go to where you saved the terrain files, game/art/terrains/sampleTerrain and open the **heightmap1024.png**.



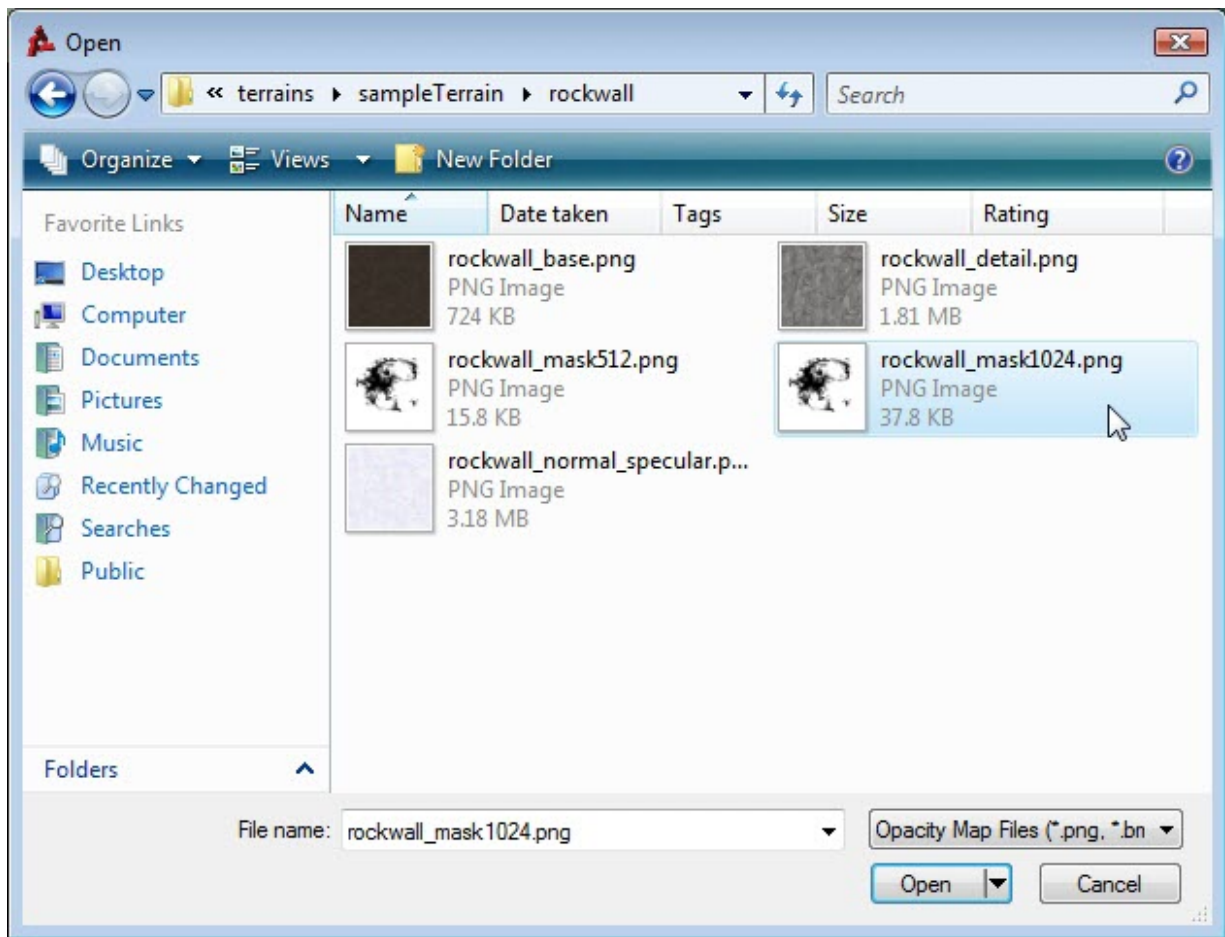
Next, click on the + button next to Texture Map to open another file browser. This is where we are going to add our opacity layers. Start by locating the prairie mask (game/art/terrains/sampleTerrain/prairie/prairie\_maskX.png). You can choose the 512 or 1024, but you have to stick with that resolution for the rest of the files we will be adding.



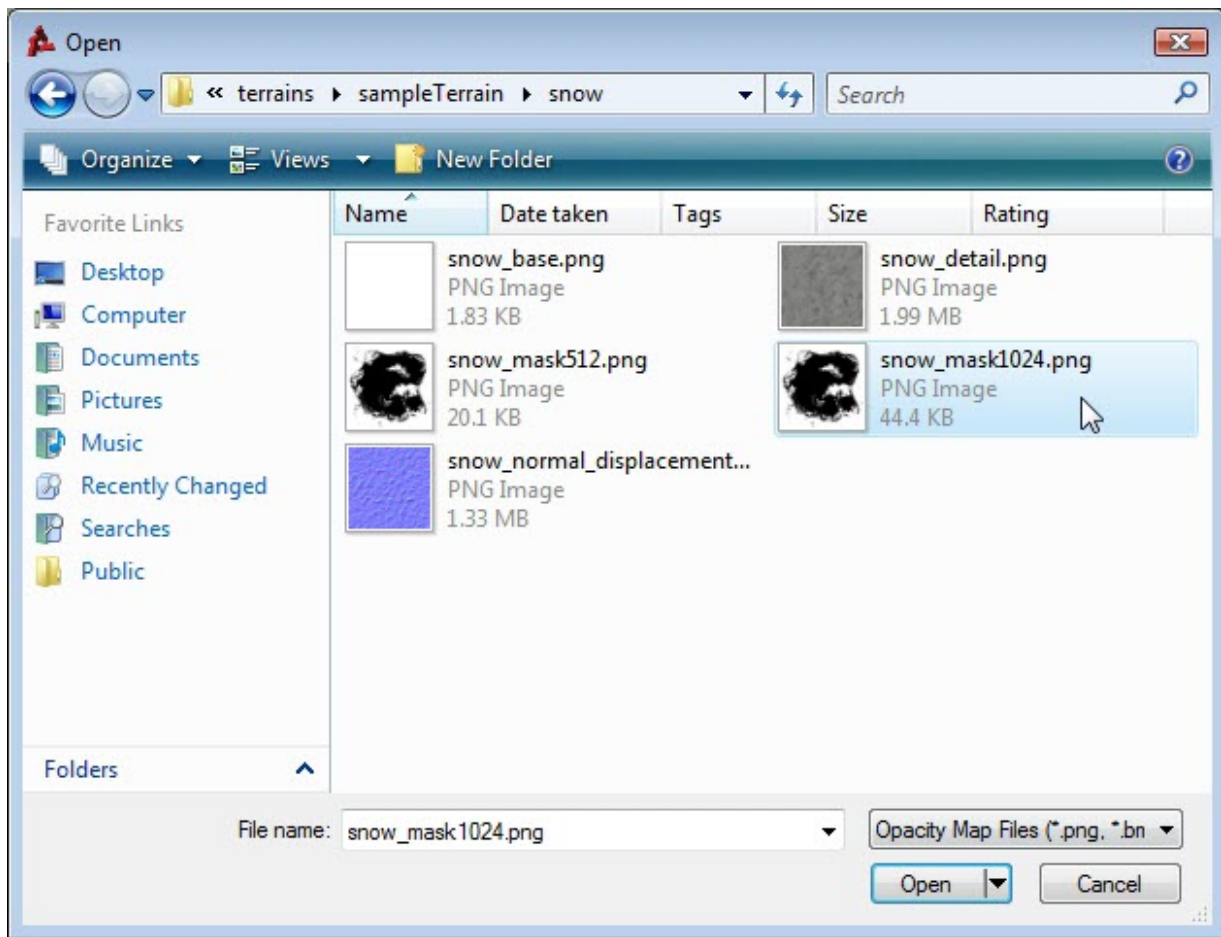


Do not worry if you do not see the detail, as the mask is supposed to be solid white.

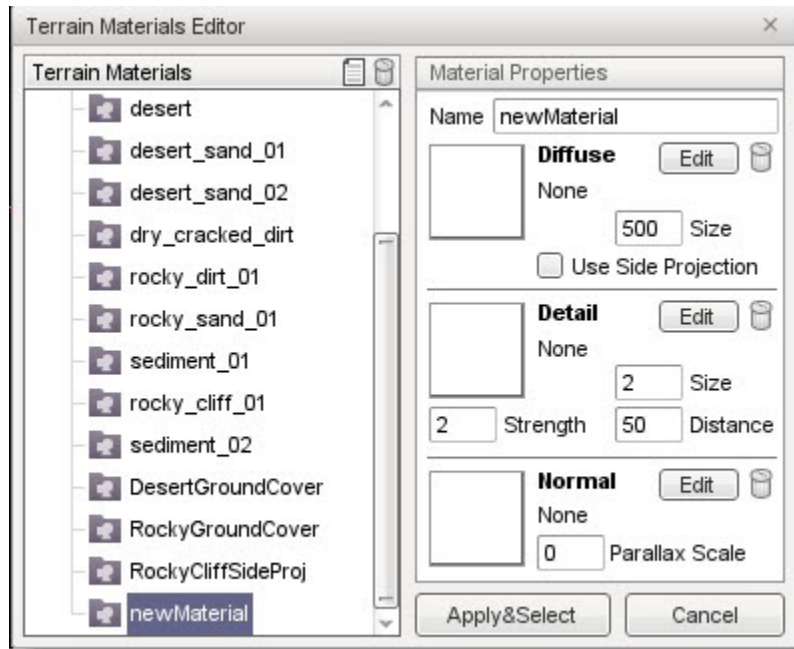
Repeat the process to add the rock wall mask.



Perform this task one last time to add the snow mask.

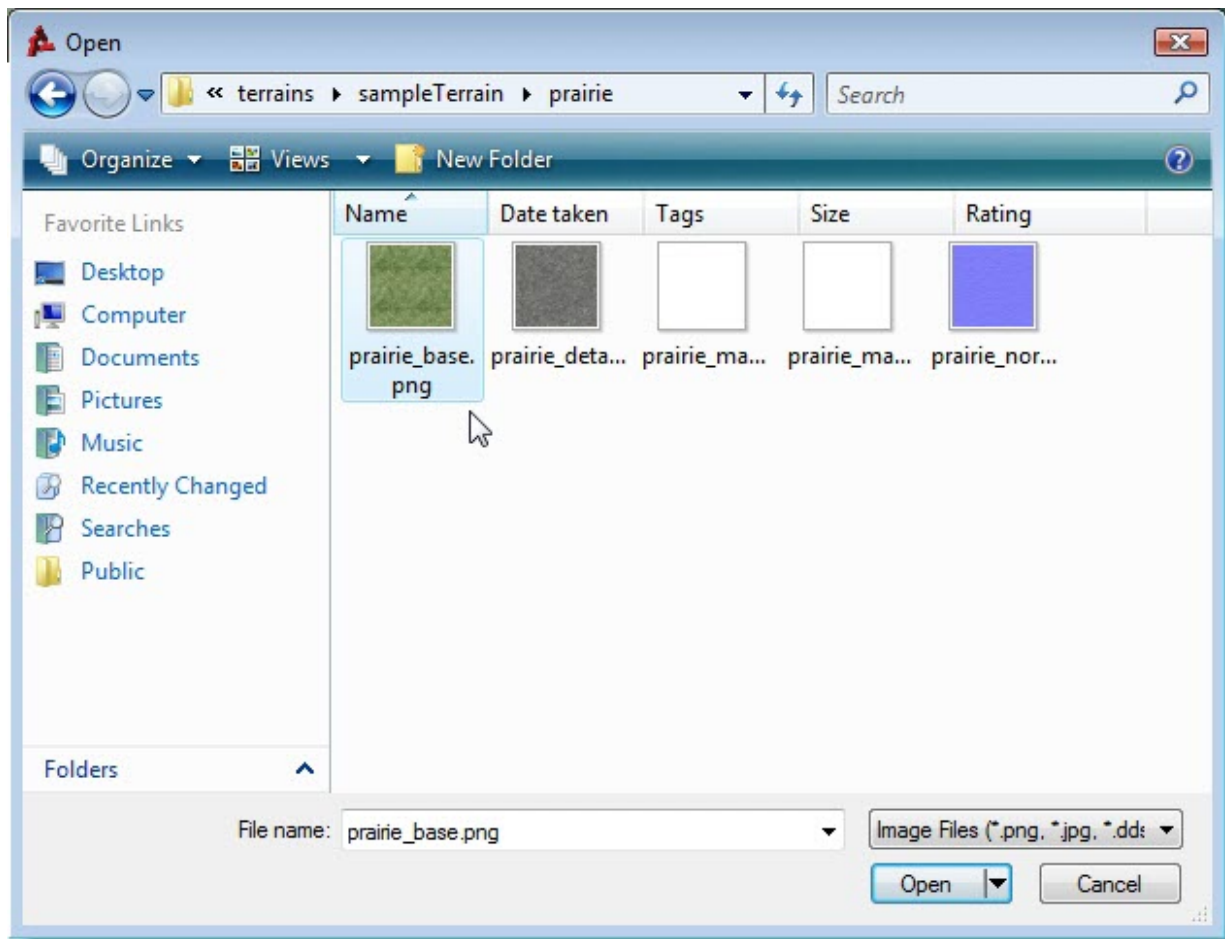


Now that our opacity layers have been added, we are going to assign a material to each one. Click on the prairie layer, then click the Edit button in the bottom right. You will now see the Terrain Materials Editor.

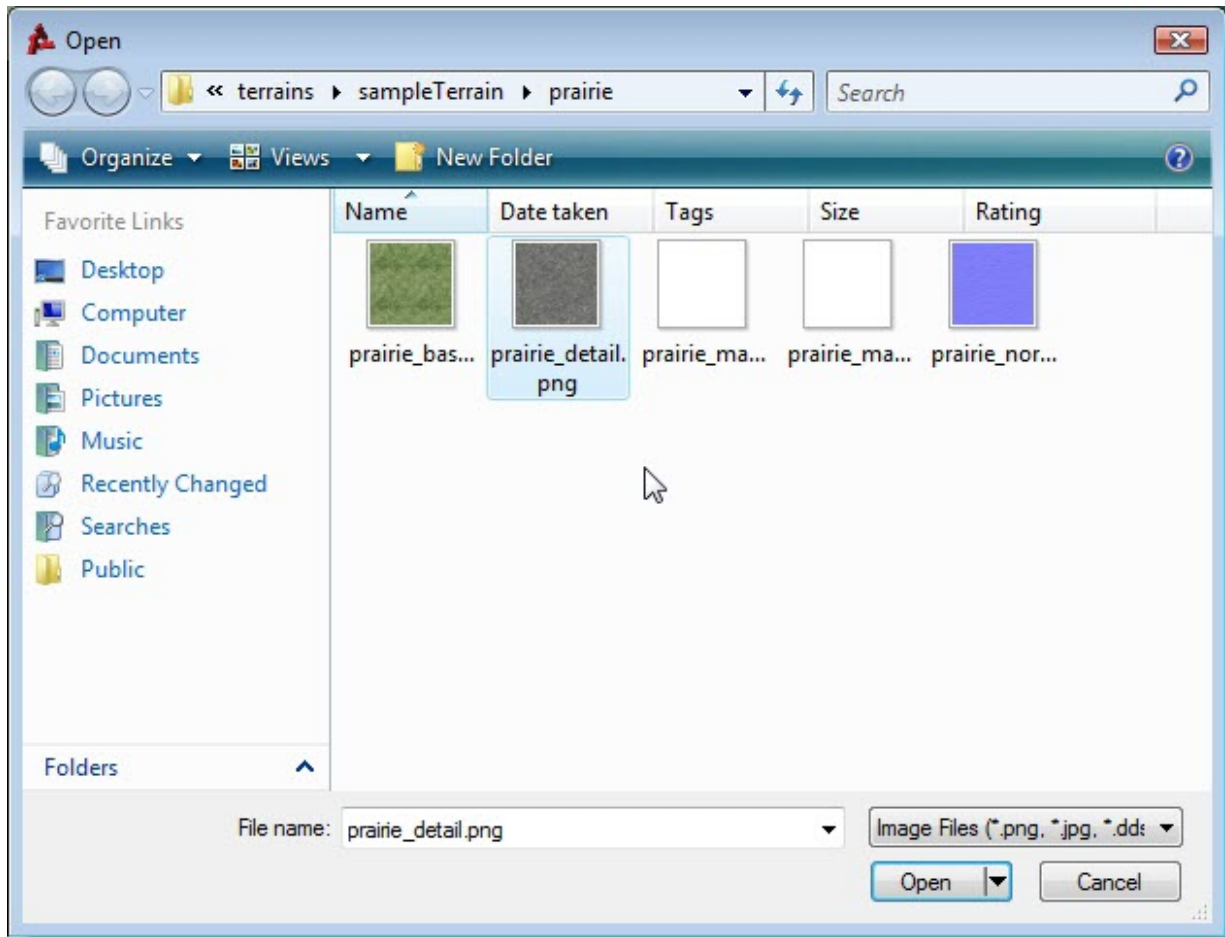


Click the New button, found at the top next to the garbage bin, to add a new material. You should see the entry newMaterial appear at the bottom of the list to the left, under Terrain Materials. On the right side of the gui under Material Properties, in the Name field type in Prairie, then hit Enter. If you don't hit Enter after naming your new material, it will not be saved. In the list on the left, the newMaterial entry should change to Prairie.

Next click the Edit button next to the Diffuse preview box. Again, a file browser will pop up allowing you to open the base texture file for the prairie material - select and open the file **gameartterrainssampleTerrain-prairieprairie\_base.png**. Alternatively, you can click the preview box itself, which is a checkered image until you add a texture.

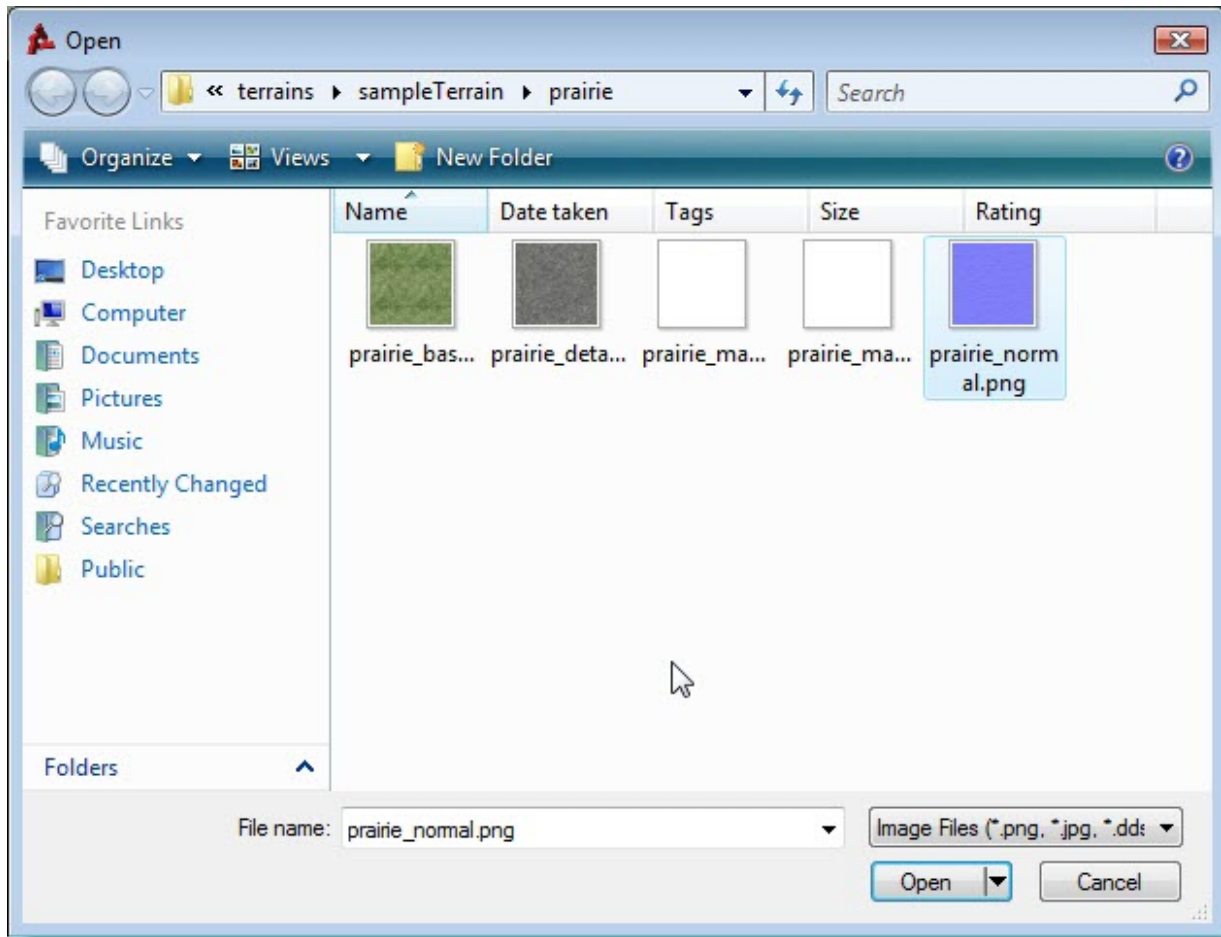


Once you have added the base texture, the preview box will update to show you what you opened. Next we'll do the same thing for the detail map. In the Detail preview box, below the diffuse section, click the Edit button. Using the file browser, open the detail map for our prairie material - `gameartterrainssampleTerrainprairieprairie_detail.png`.



Lastly, do the same thing for the normal map. In the Normal preview box, below the Detail section, click the Edit button. Use the file browser to open the prairie normal map - **gameartterrainssampleTerrain-prairieprairie\_normal.png**

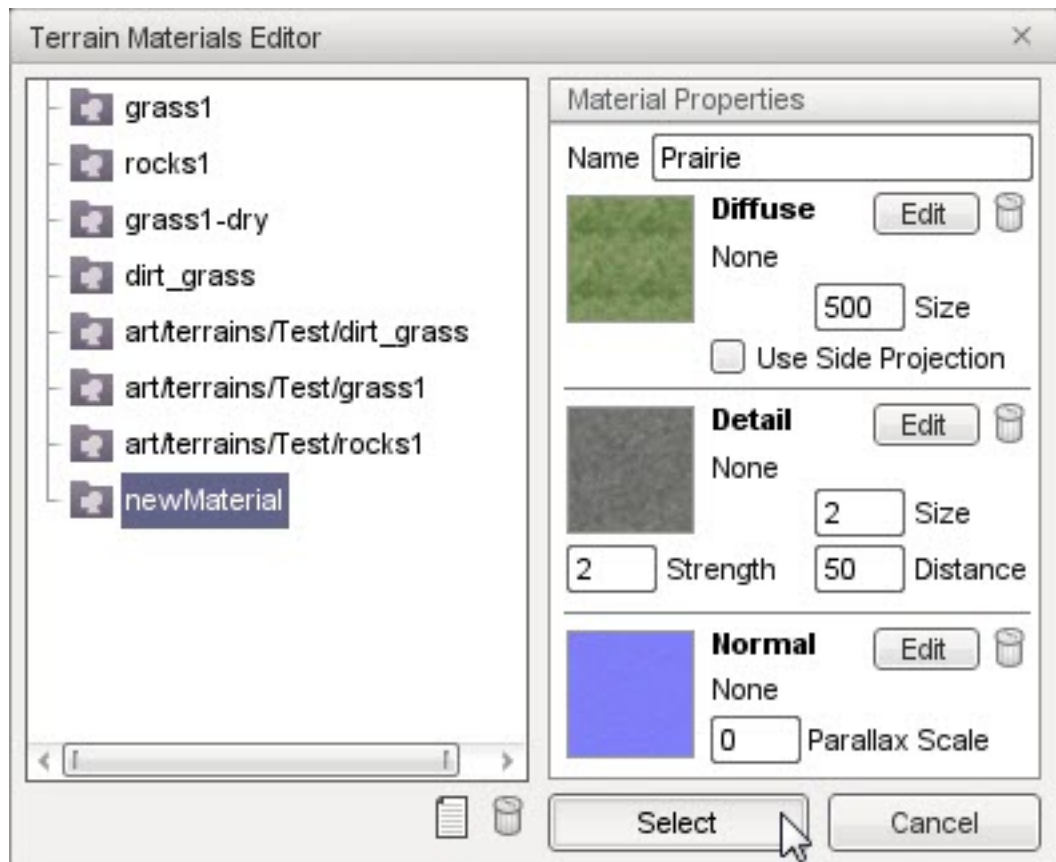




Now we need to set some parameters. In the Diffuse box, the Size parameter controls the physical size in meters of the base texture - set it to 500.

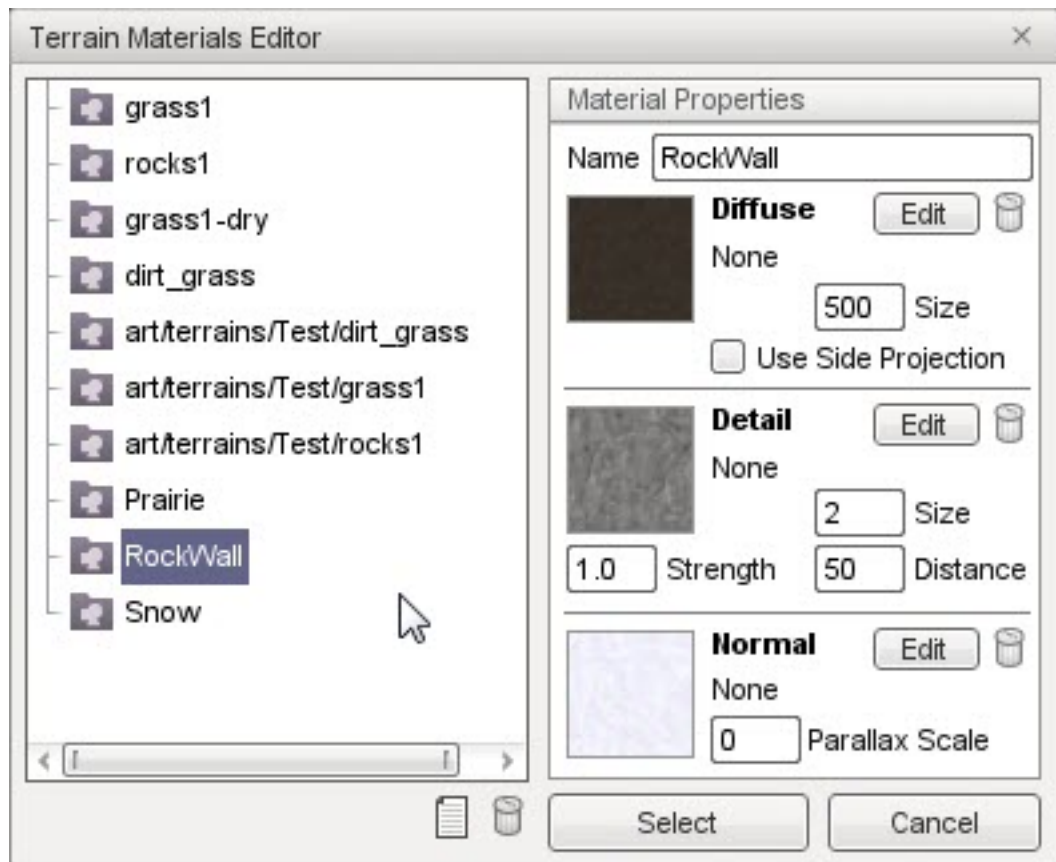
In the Detail box, set Size to 2. This means that the material will be scaled to two meters on the terrain. On a terrain that is 1024 square meters, the Prairie material will repeat a little less than 205 times. The Distance parameter determines how far away from the camera must be before the detail map renders - set it to 50. Set the Strength parameter to 2.

Your final material properties should look like the following:



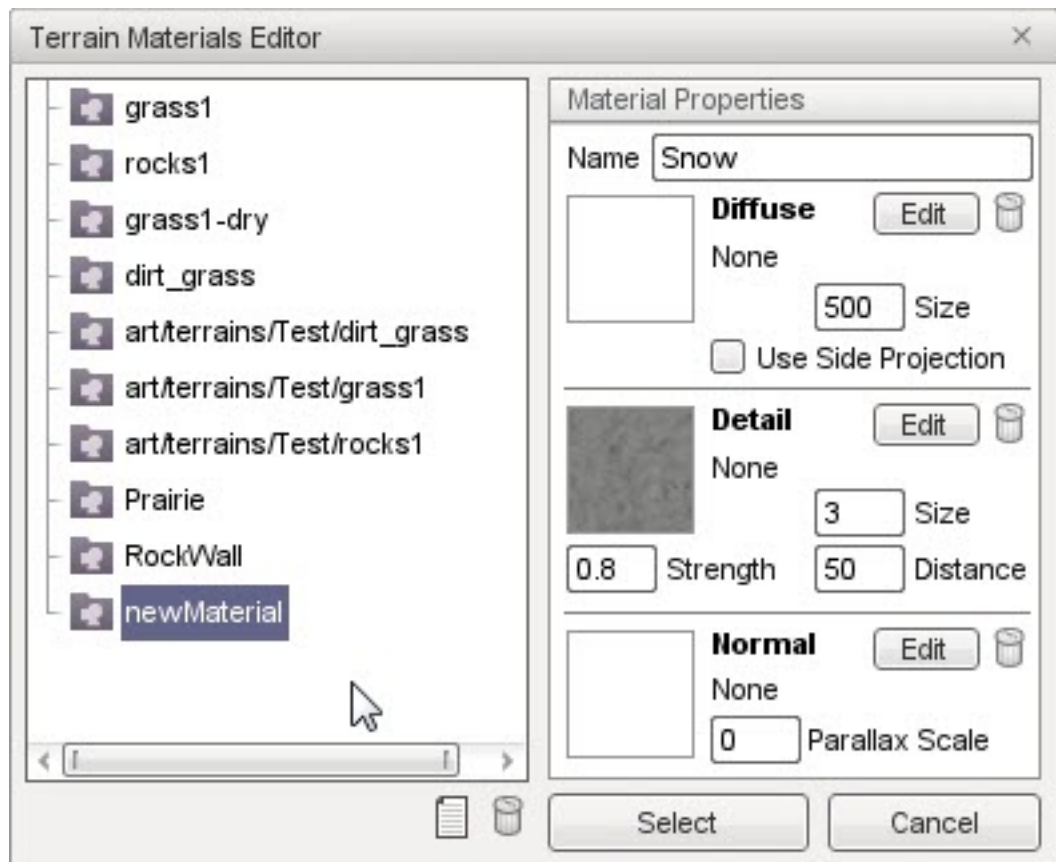
Click the Apply & Select button to assign the new Prairie material to the opacity layer.

The Import Terrain Height Map dialog will appear. Next, we will add the rock wall terrain material. In the Texture Map list, select the rockWallMask opacity layer, then click Edit. Repeat the process of creating a new terrain material, using the rock wall textures. Your final result will look like this:

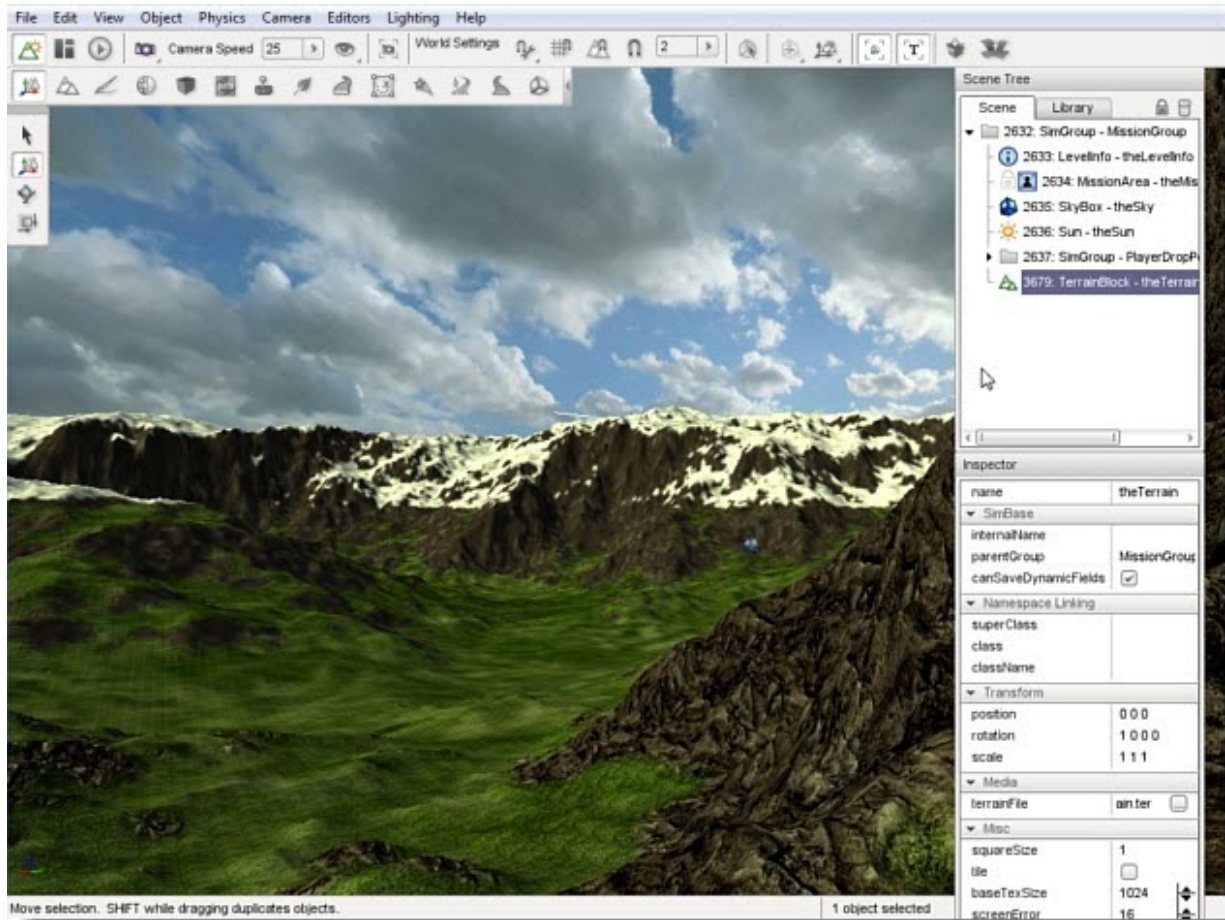


Notice that I have set the detail size to 2, and the detail distance to 50.

We are going to add our final terrain material now. Back in the Import Terrain Height Map dialog, select the snowMask opacity layer then click edit. Repeat the process of creating a new terrain material, using the snow textures. Your final result will look like this:

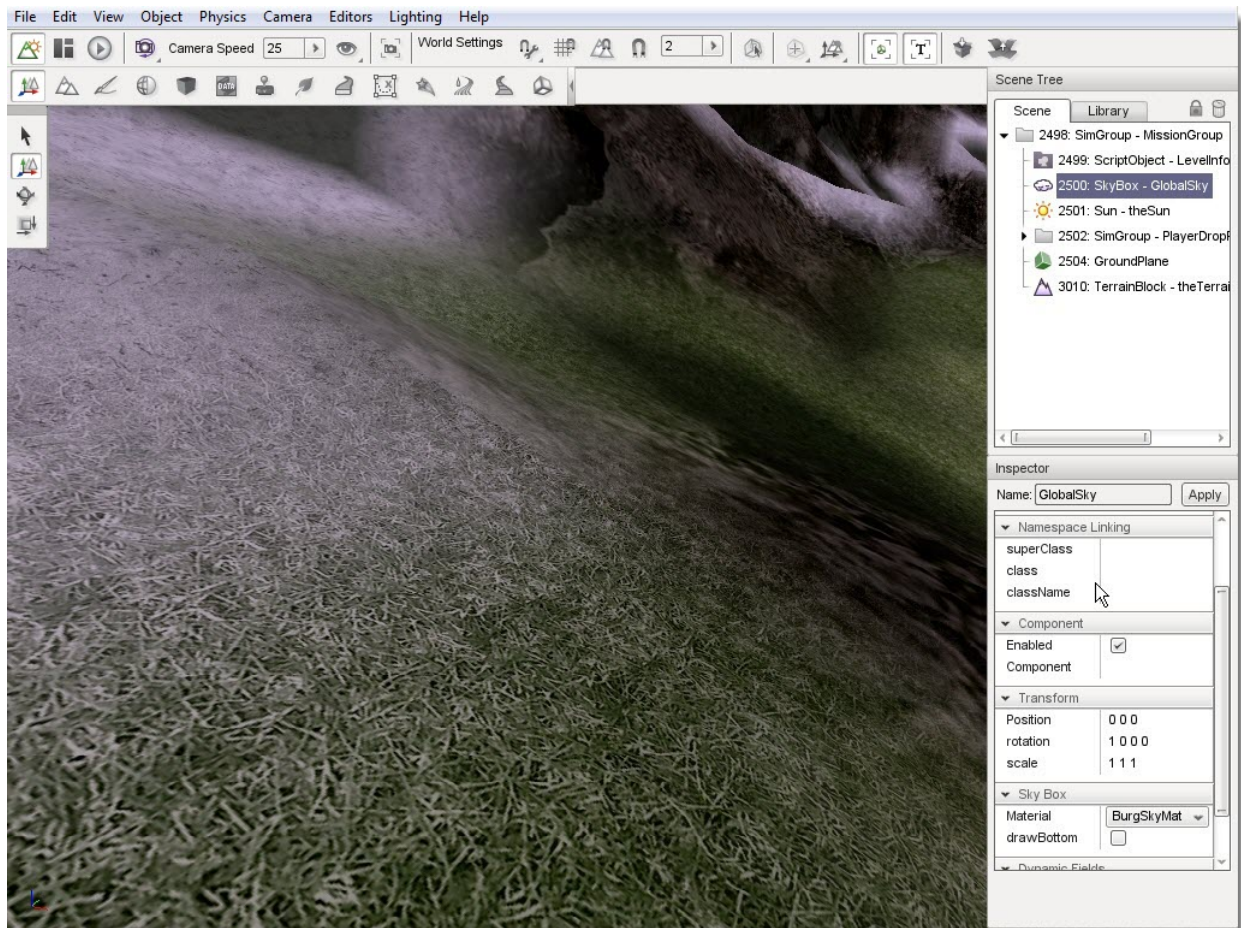


Now, we are all set to generate the terrain. Back in the Import Terrain Height Map dialog, click on the import button. It will take a few moments for Torque 3D to generate the terrain data from our various assets. When the import process is complete, the new TerrainBlock will be added to your scene (you might need to move your camera to see it).



If you zoom in close to where materials overlap, you can notice the high quality detail and smooth blending that occurs.





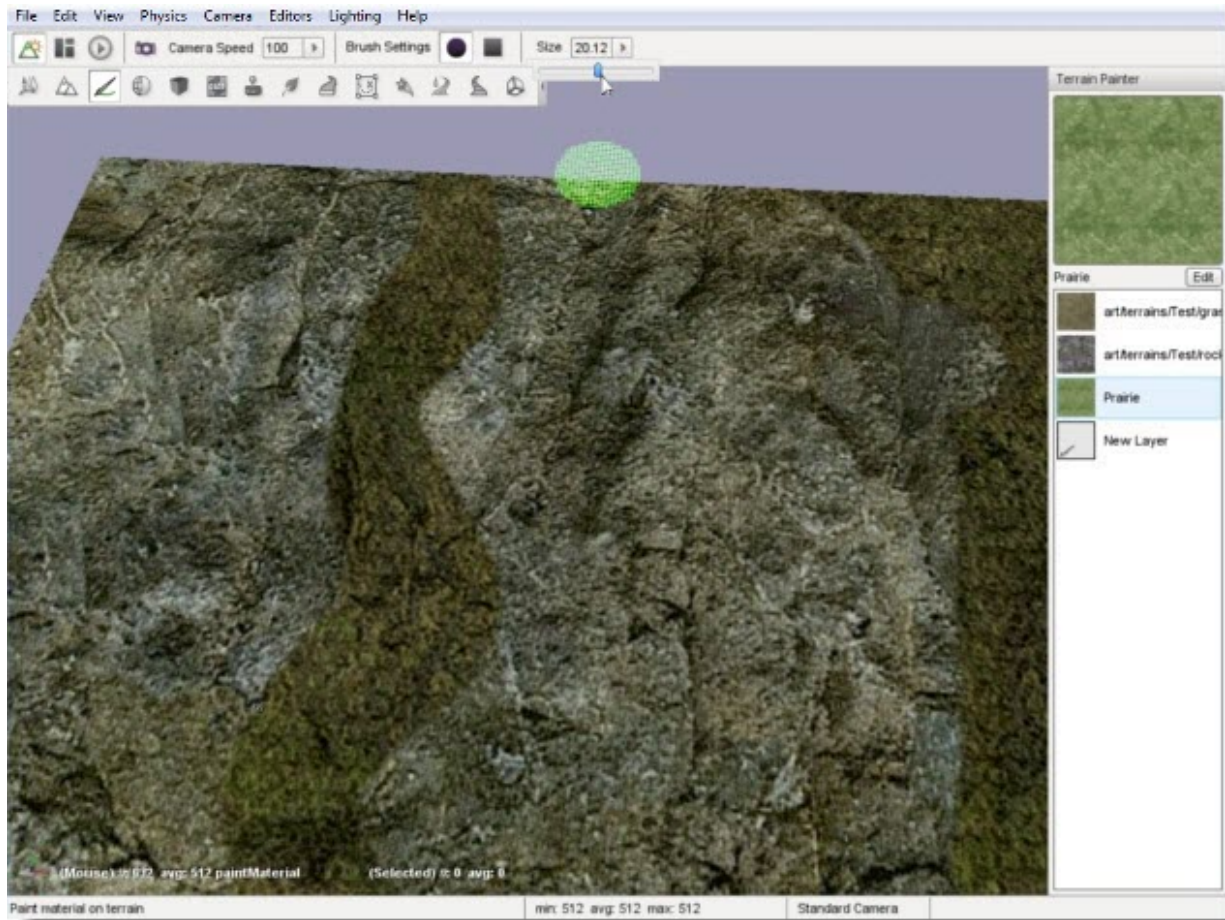
## Painting/Adjusting New Material

Go ahead and select the Prairie material in your Terrain Painter palette.

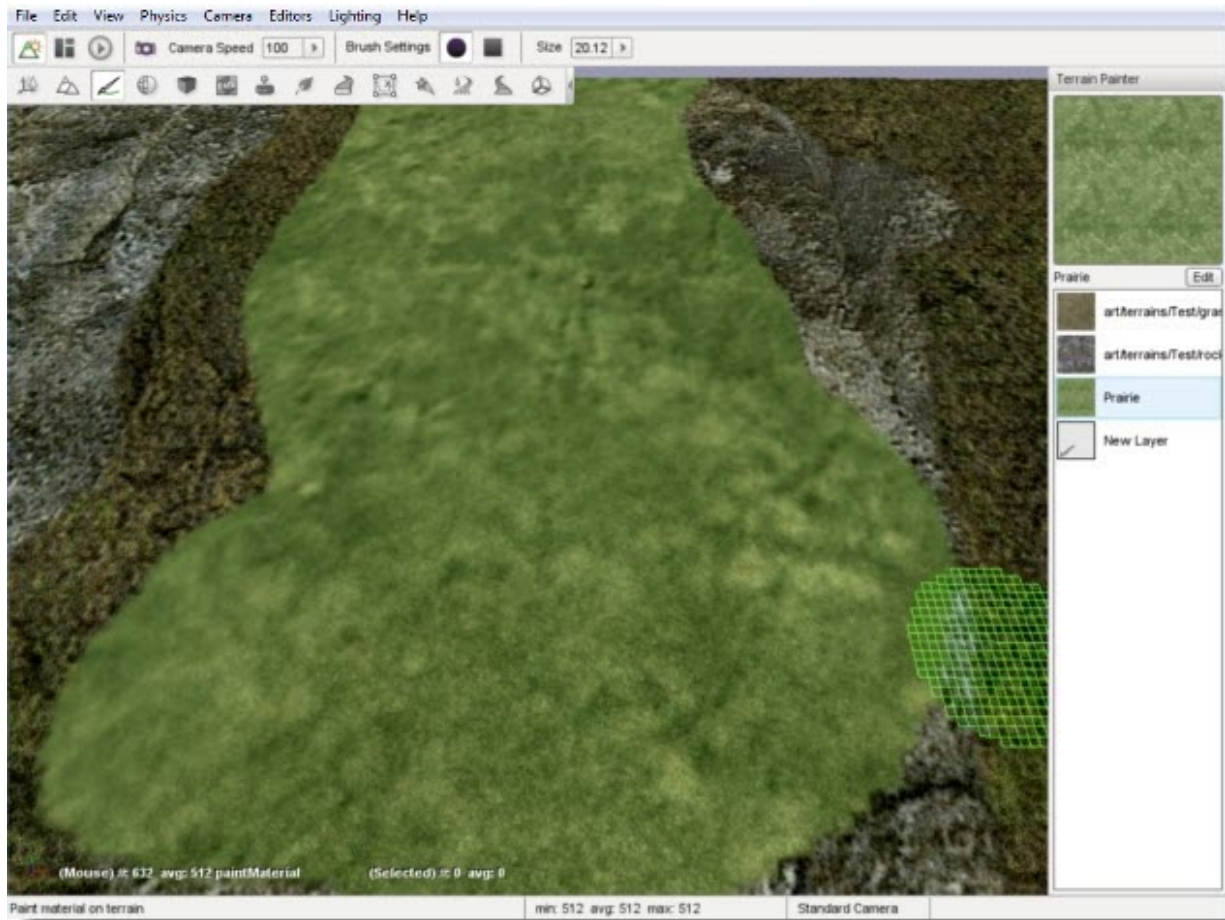




Pick any location on your terrain, using any size or shape brush you wish. It does not matter where you start.

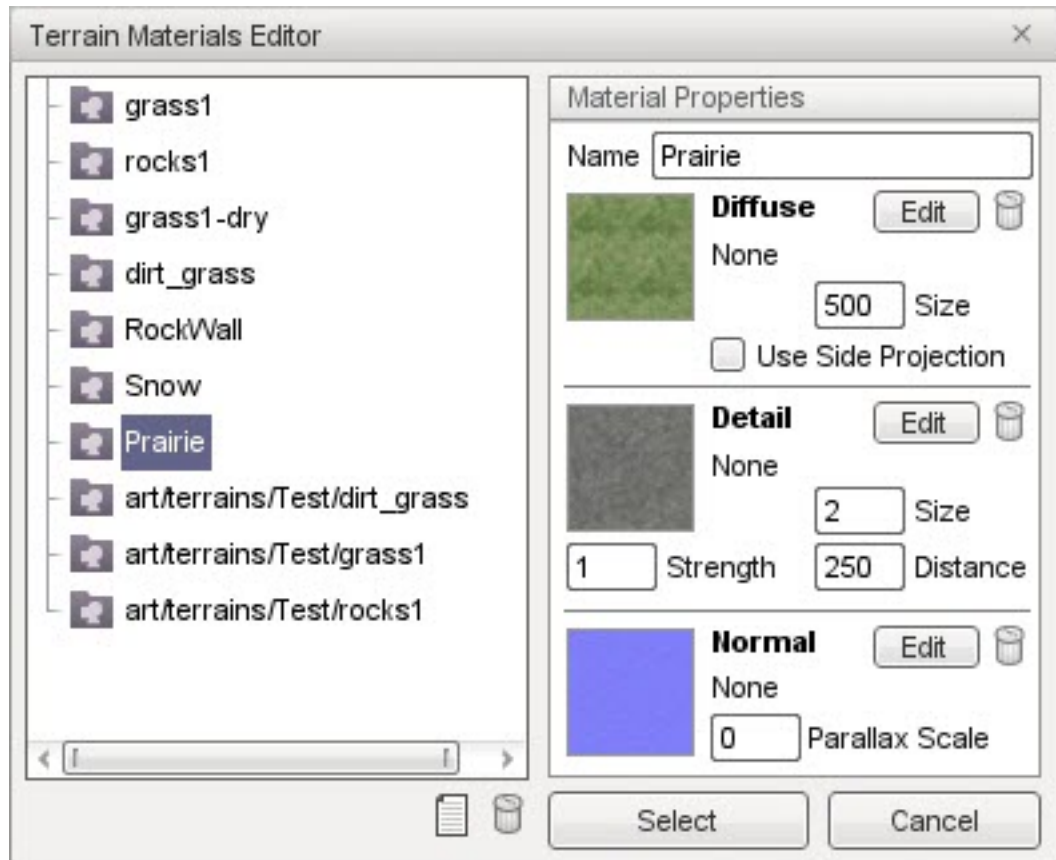


Once you are set, click and hold down the left mouse button to begin painting. Make sure you paint a fairly large area. We will be changing the properties of this material shortly, so we need to be able to see it from a distance.



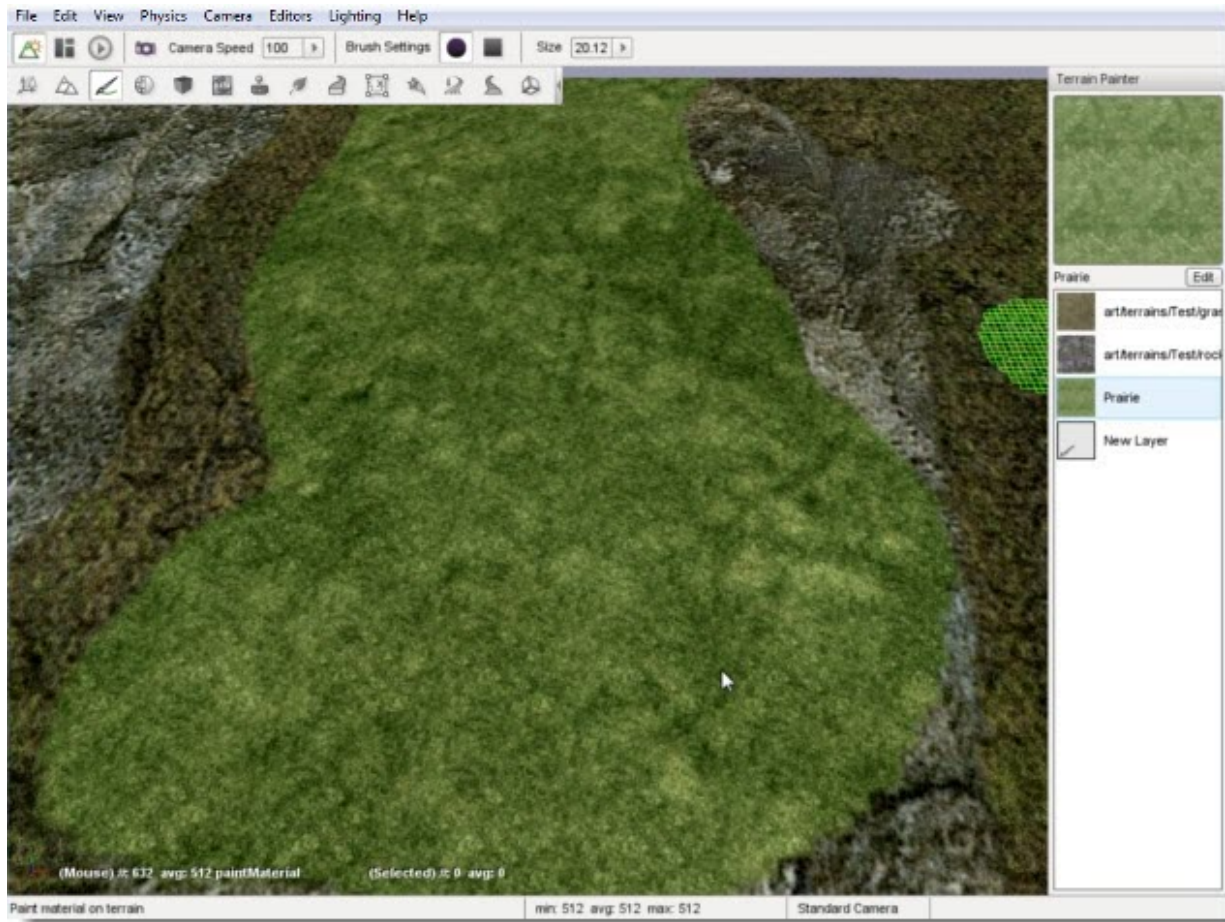
From a distance, you may notice that your Prairie material looks blurry and undefined. Even though the material has a detail texture, it is not visible from this far away. Double click on your Prairie TerrainMaterial in the Terrain Painter palette.

Once the editor pops up, click on the Prairie entry to view its properties. Up the Detail Distance from 50 to 250.

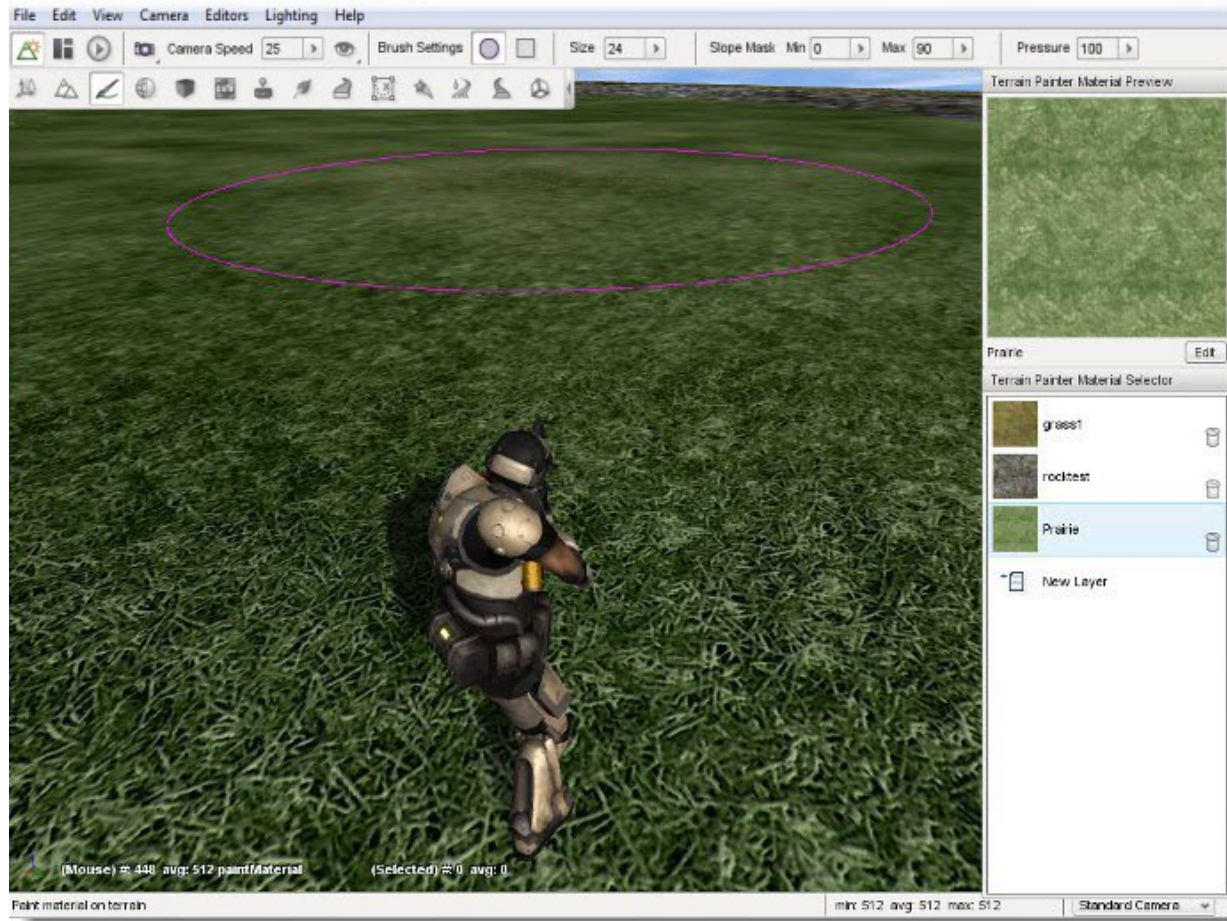


Click select to close the editor. The terrain you have painted with the Prairie material has updated, and you should now see more definition even at a distance.



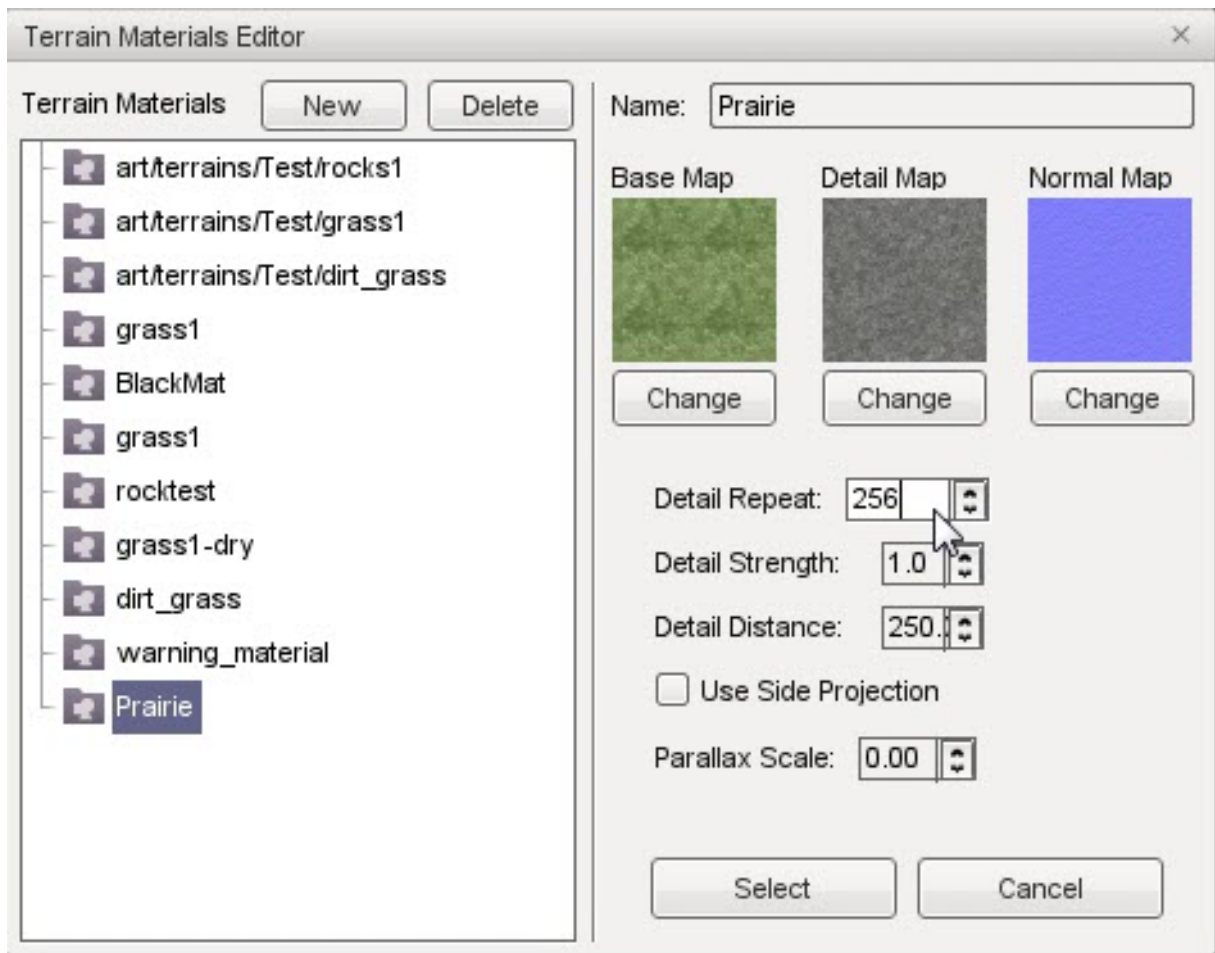


Now that the terrain looks better at a distance, what about close up? A closer view of the the Prairie-painted terrain will show off the detail texture quite well.

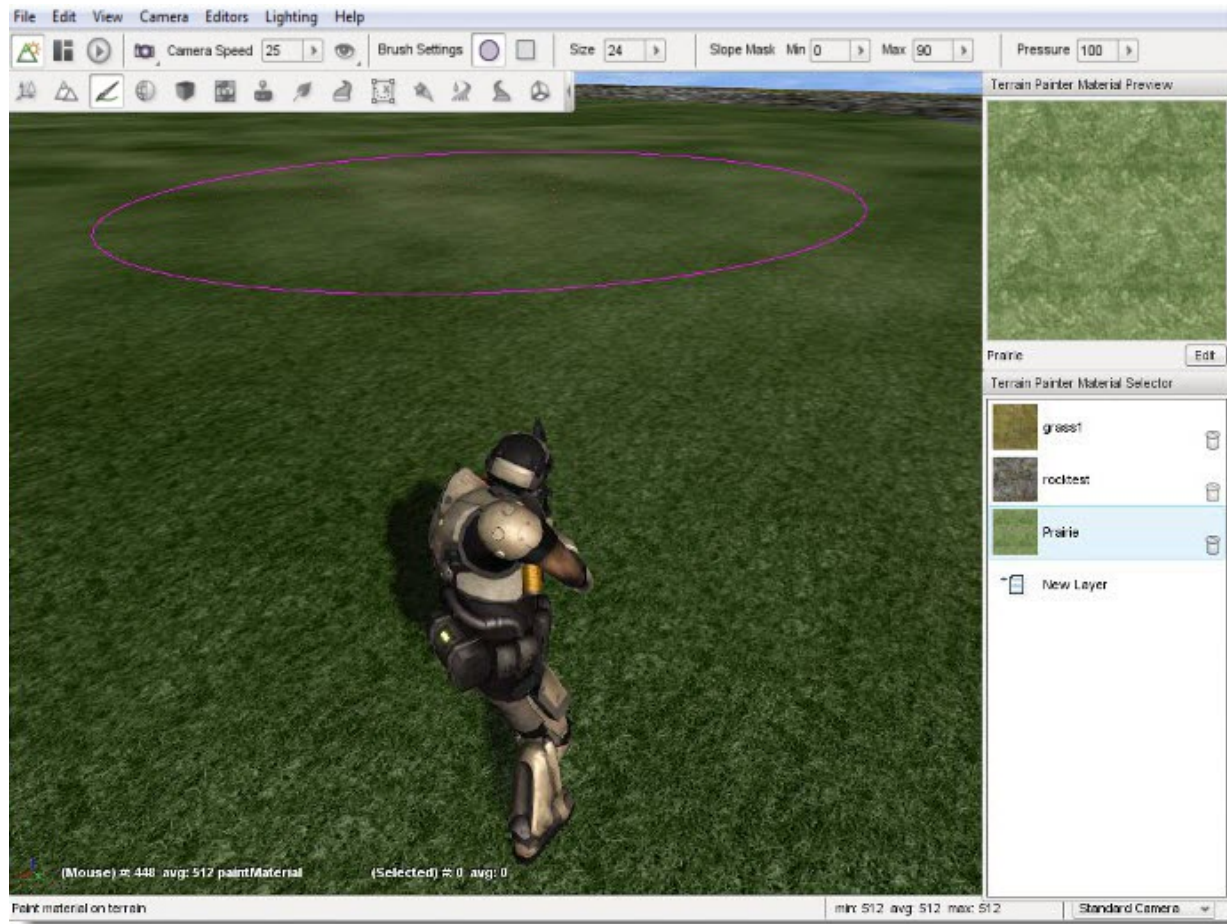


If you think the “grassy” appearance is too large or stretched, we can tweak that from the Terrain Materials Editor. With the Prairie layer still selected, open the editor. Lower the Detail Size value, which will cause the detail texture to repeat more often per meter.





Click select to apply the changes. Again, your painted terrain will update immediately to reflect the changes you just made. Notice how much more detailed the TerrainMaterial.

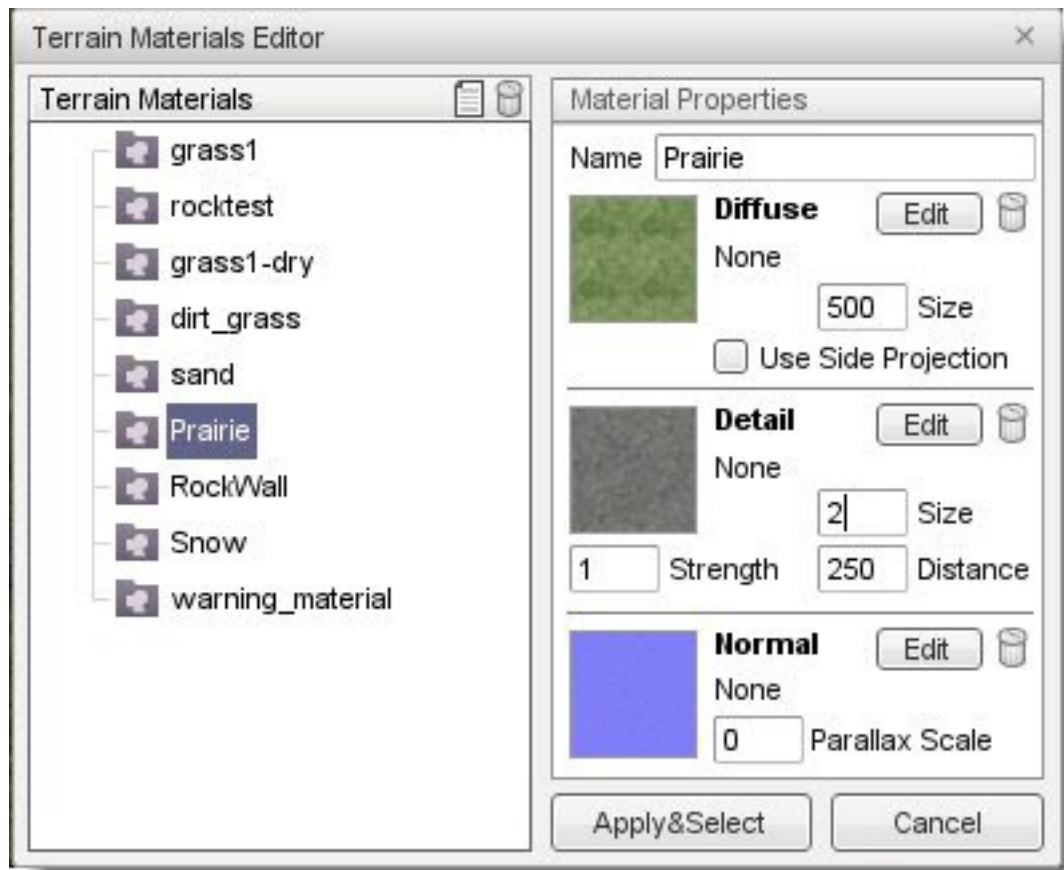


The values we just set are somewhat extreme. You will need to experiment with the values on your own assets to find a balanced setting that looks well up close and from a distance. The last task we are going to accomplish is swapping TerrainMaterials between layers.

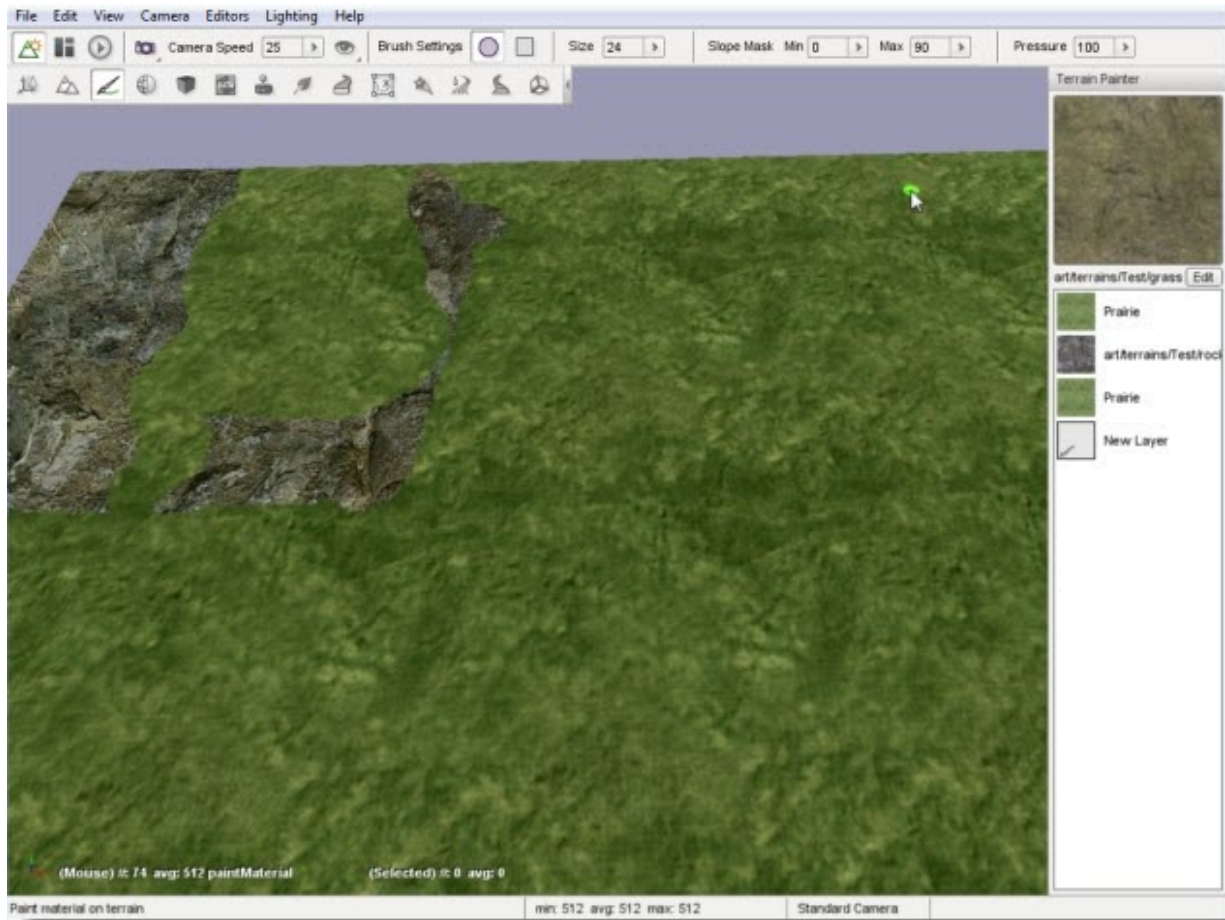
In this tutorial, grass1 is layer0 and Prairie is layer2. Since the first layer is the base material applied to the terrain, it makes up the majority of the level. Start by selecting the first layer (grass1) in the palette.



Instead of manually painting the entire terrain a separate material, we can flip the layers. Double click the grass1 layer to open it up in the Terrain Materials Editor. Once it is open, select the Prairie TerrainMaterial from the list.



Click the select button. The Prairie TerrainMaterial will now be used for layer0, thus covering the majority of the TerrainBlock.

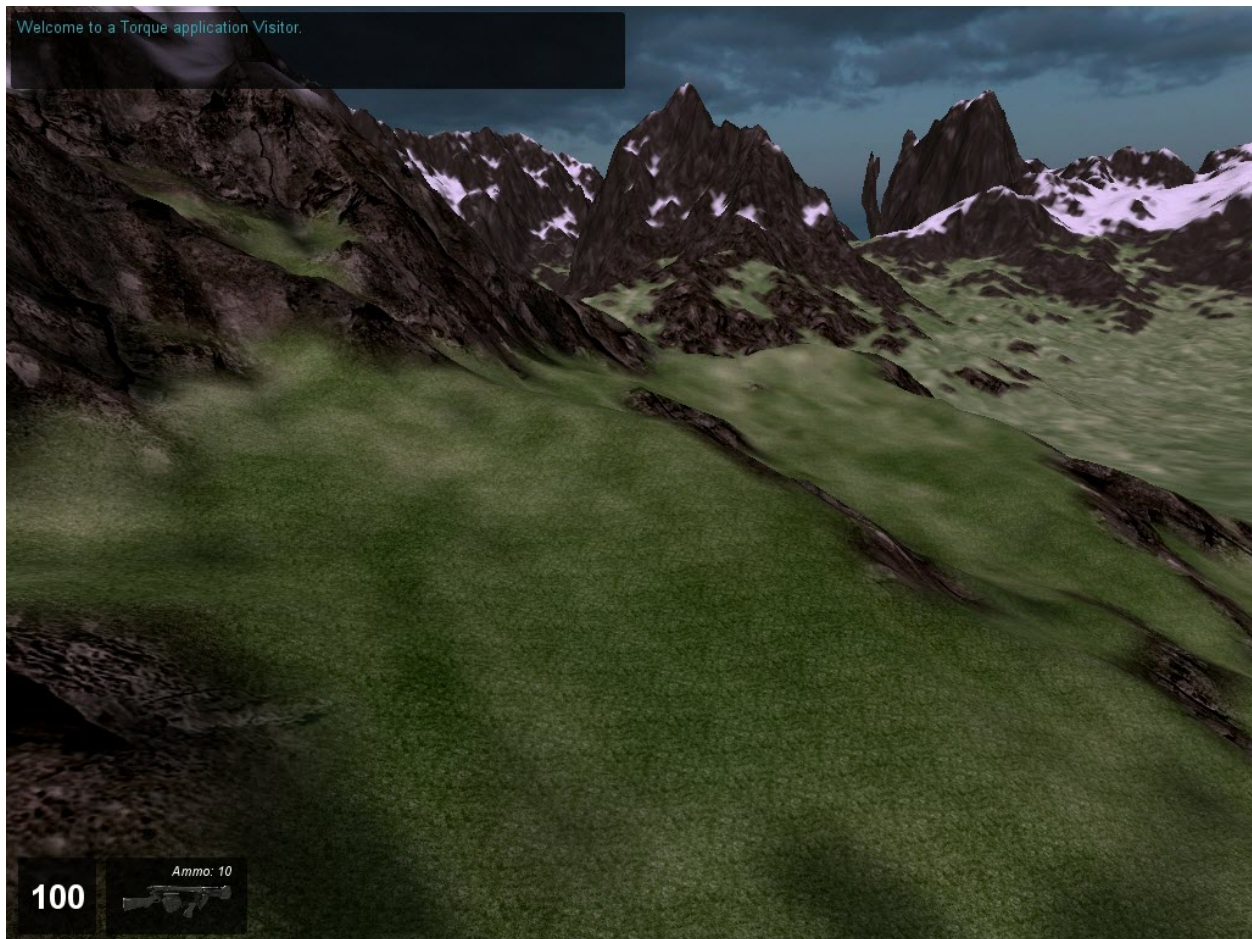


The intricacy of using TerrainMaterials and the layer system becomes much more prominent when working with opacity layers, advanced modification, and adding specific objects such as GroundCover. Also keep in mind that any asset files you modify outside of Torque 3D will automatically update in the editor.

These last two shots are used to show you the scale of this massive terrain, which retains its high level detail and levels of detail (LODs):

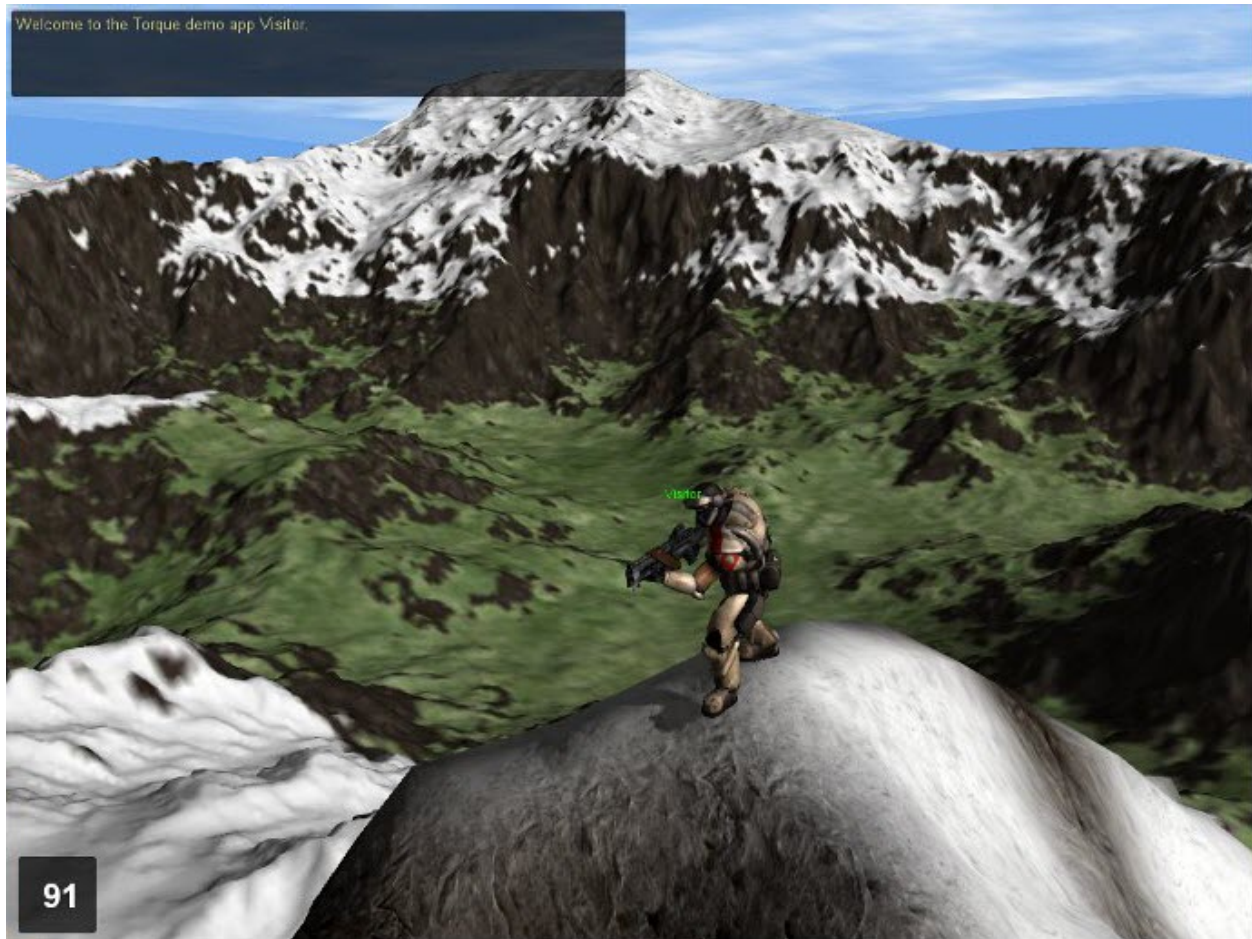
### **From a Distance**





Compare to Player Scale





### 14.1.5 Conclusion

This tutorial showed you how to create a high resolution terrain from scratch by importing a quality heightmap and opacity maps. Even after you have your terrain, you can continue to tweak it using the Terrain Editor and Terrain Painter tools.

## 14.2 Creating a Sky

### 14.2.1 Introduction

In this tutorial, we are going to create a basic sky for your level. Our sky will consist of the following three objects:

- Sun
- Skybox
- Basic Clouds

All three objects will be added from scratch and modified to work with each other to create a nice looking atmosphere. To get started, follow the setup procedure below.

### 14.2.2 Setup

This article assumes that you have read through the **Toolbox** and **World Editor Basics** sections of this documentation so that you are familiar with basic operations such as creating a new level and activating tools such as the Object Editor. If you have not, please review those documents and then return here to continue.

None of the modifications you are about to make are required for future tutorials, so feel free to create a new level or use an existing one now. As long as you have access to stock materials you are ready to go. For this article, we are going to use a new level.

Because every new mission starts with some kind of sky object (Sun, Skybox, etc), you are not technically starting from scratch. You will want to clear your level before beginning.

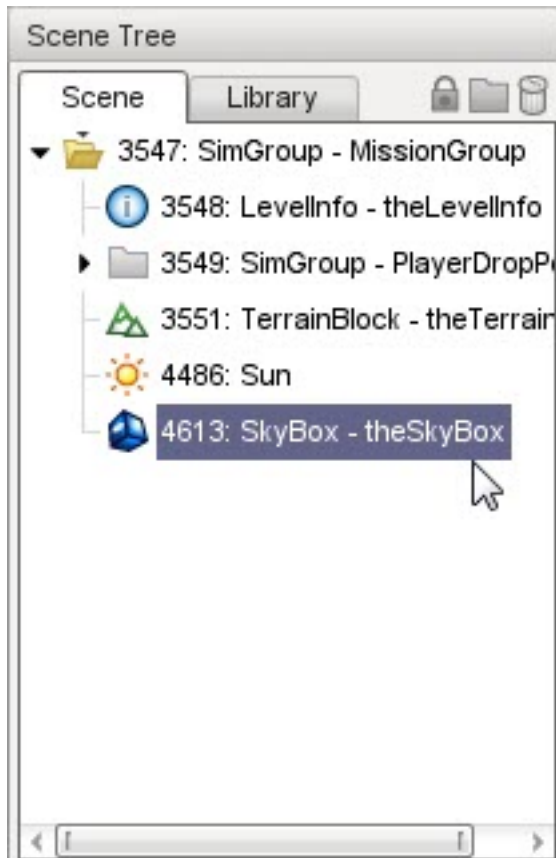
None of the modifications you are about to make are required for future tutorials, so feel free to create a new level or use an existing one. As long as you have access to stock materials, you are good to go. For this article, we are going to be using a new level.

Because every new mission starts with some kind of sky object (Sun, Skybox, etc), you are not technically starting from scratch. You will want to clear your level before beginning.

Finally, the textures for a simple Skybox are provided through this link: [click here to download](#). You will use these later in the tutorial when creating a new Skybox material.

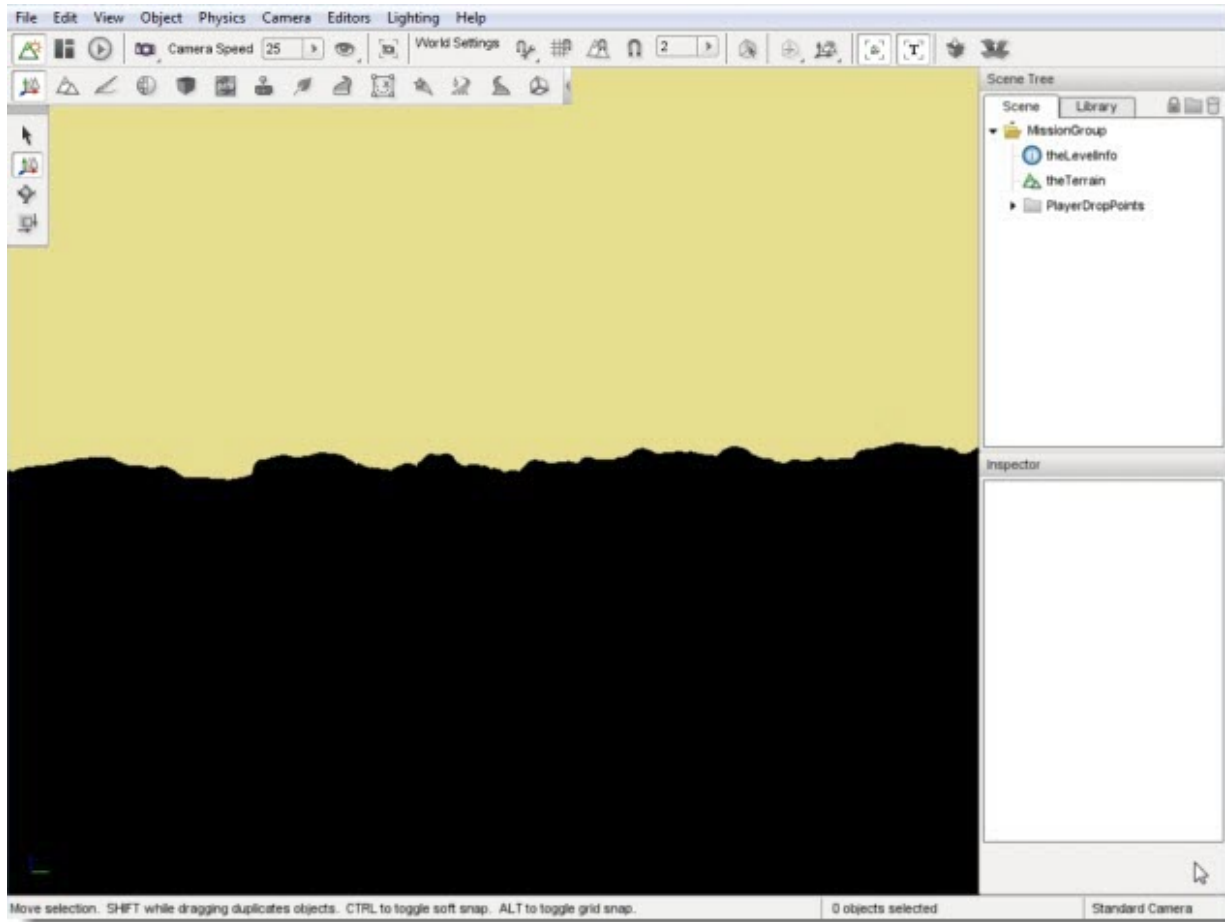
### 14.2.3 Delete Existing Objects

Start by switching to the Object Editor tool. Locate the Scene Tree panel, then select the Scene tab. Inside the MissionGroup locate any objects that contain the words sky, sun, or clouds in them. Select the object, then delete it by pushing the delete key, or by clicking the trash can icon.



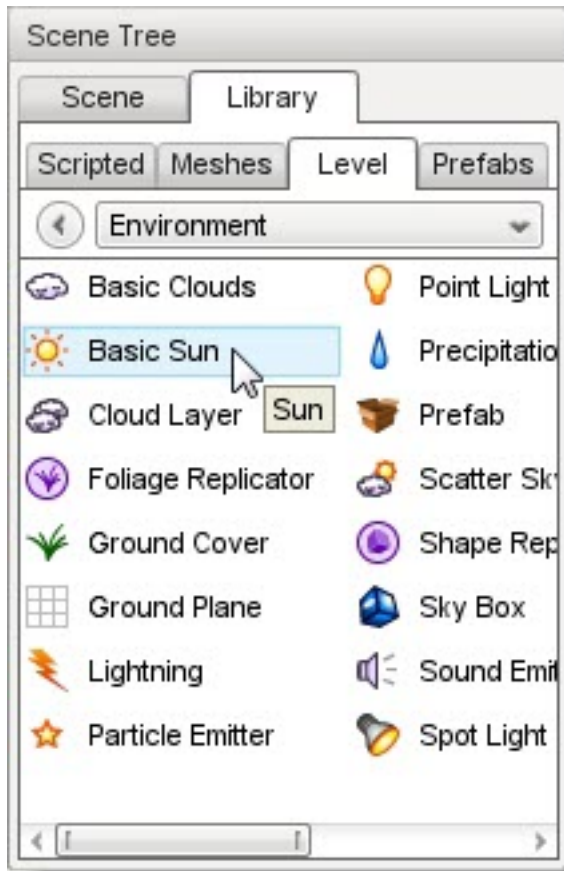
The objects should no longer be present in the Scene Tree. More importantly, the sky has now been removed from your level. Because nothing is rendered beyond a Skybox, Torque 3D will be rendering absolutely nothing where the sky used to exist.

Additionally, after you delete the Sun there might not be any lighting in your level. This will result in everything being completely shadowed, but we will fix that in just a minute. For now, save your blank level before continuing.



### 14.2.4 A New Sun

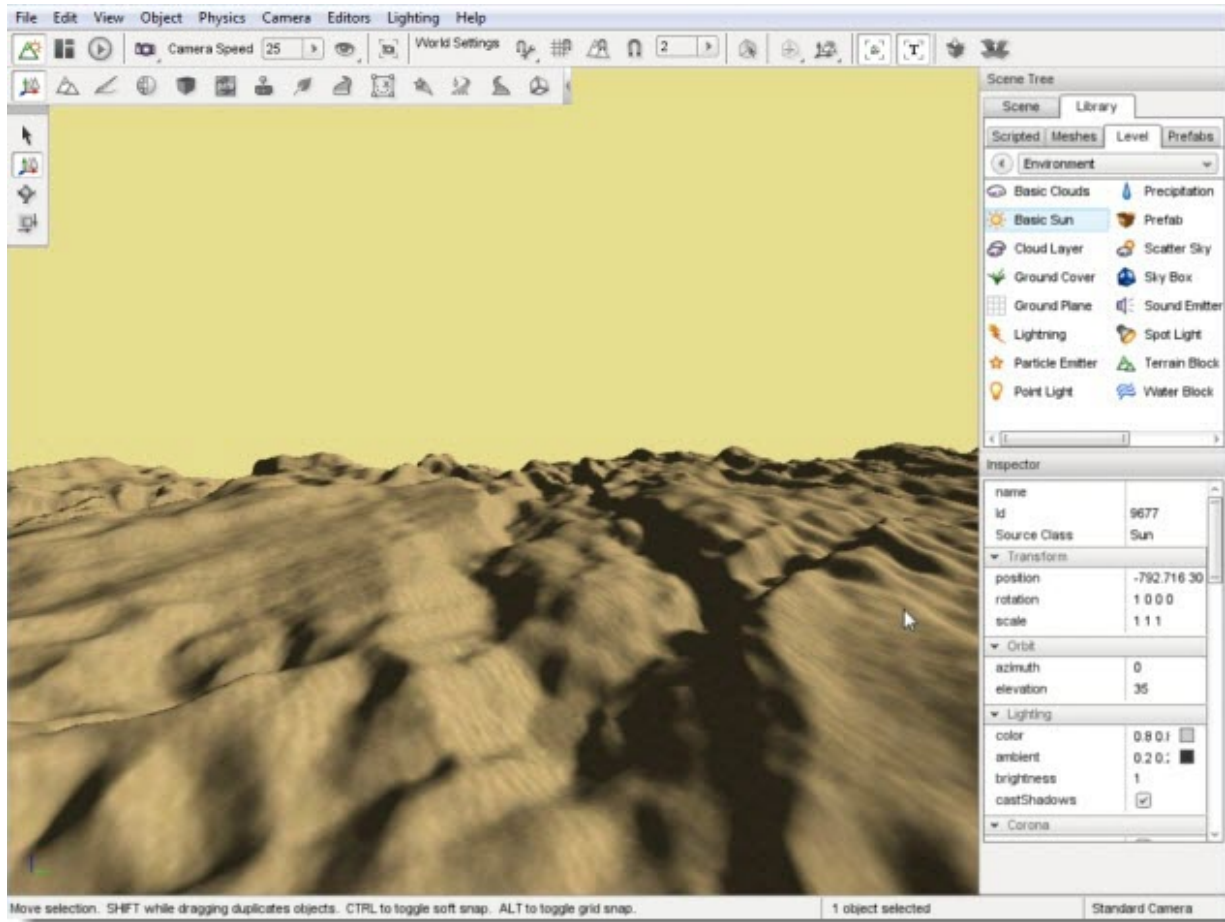
Now we can add a Sun with default attributes. Start by opening the Library tab in the Scene Tree dialog. Once the Library tab is active, click on the Level tab, then double click the Environment subcategory. The list of available environment objects should now be visible.



Double-click on the Basic Sun symbol. Once you do, the Create Object dialog will pop up.



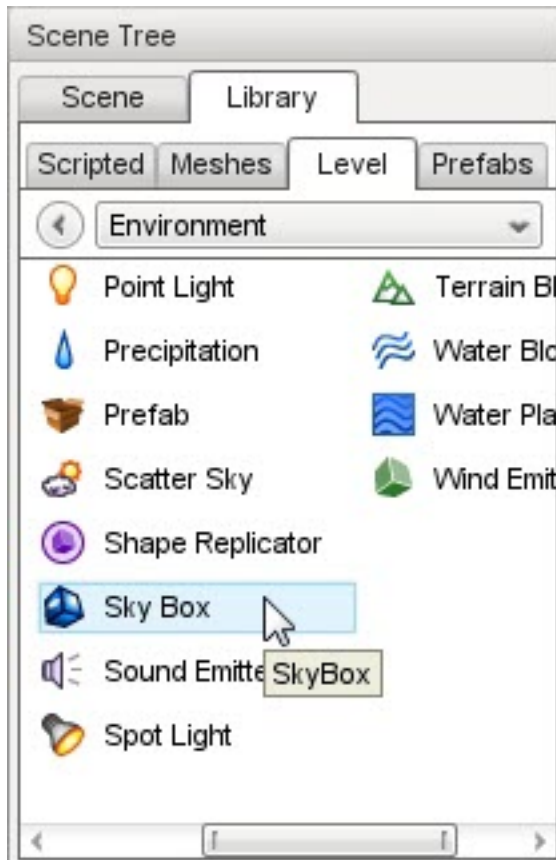
Since we want to use stock values for everything, just click Create New without changing anything.



### 14.2.5 Adding A Skybox

Now we can add a Skybox from scratch. You should still be viewing the environment category which is also where the Skybox object is located. If not, change back to the Library tab in the Scene Tree panel, then select the Level tab then double click the Environment folder. Locate the Skybox entry and double-click it.





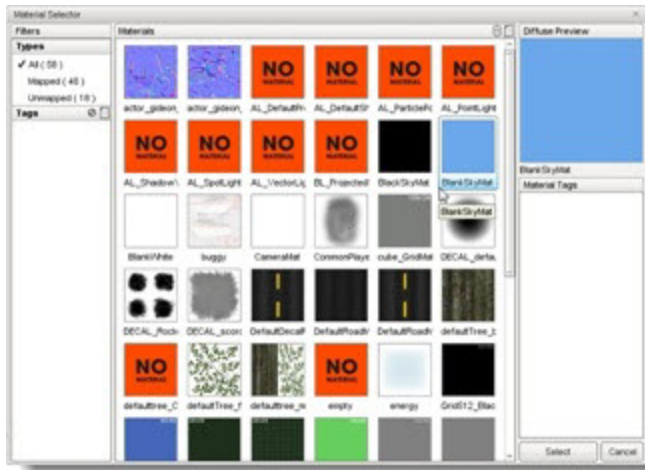
With your sky restored in the level, we can now tweak some settings to make it look nicer.

### 14.2.6 Changing Skybox Material

Without a designated material, your Skybox will still be rendering a single color. To change that use the Scene Tree click the Scene tab and select the Skybox. With the Skybox object selected, scroll through its properties until you find the Material field.

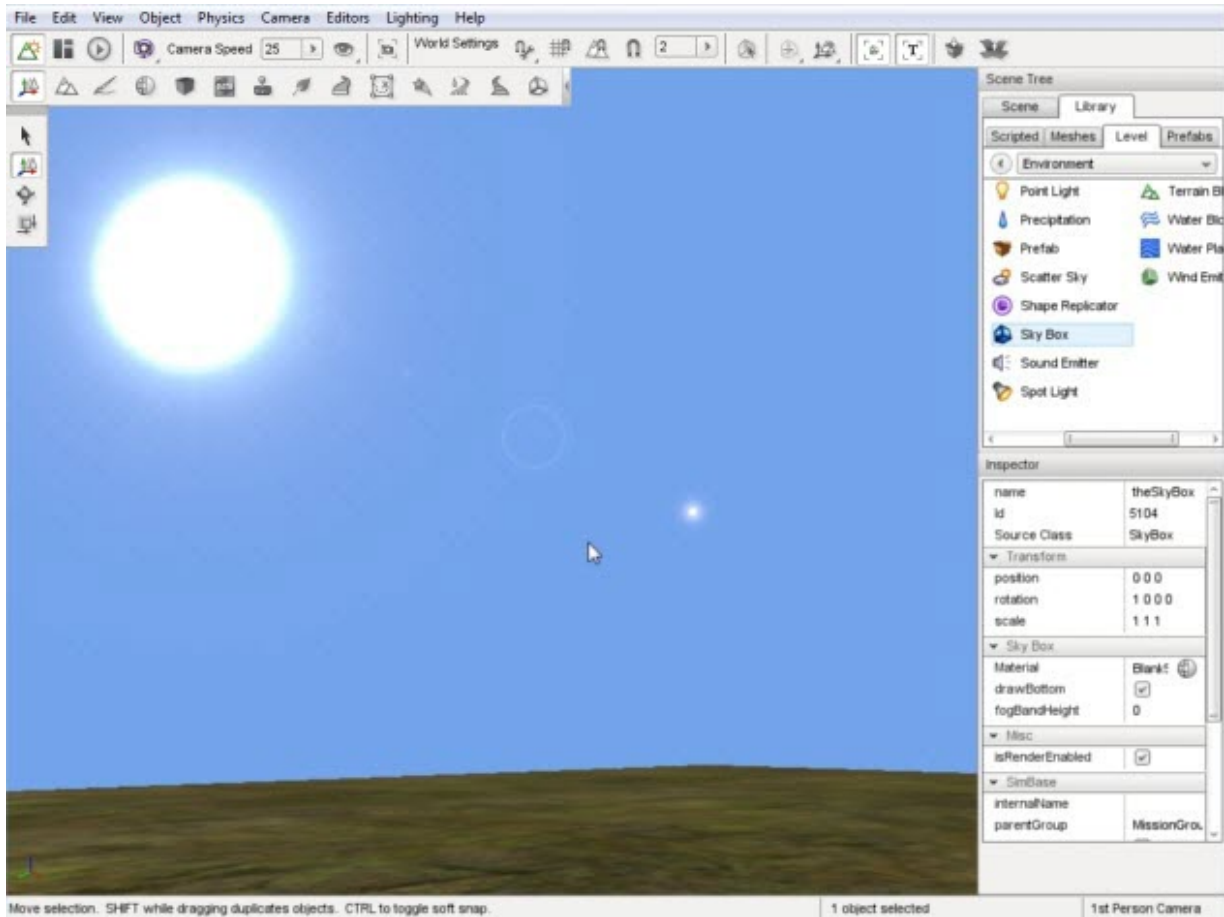


Click on the small globe icon to open the Material Selector:

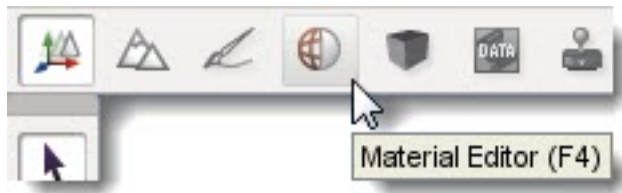


The Material Selector will show all of the materials that could be found by the engine. Look for one containing “Sky” in the description (e.g. BlankSkyMat). Select it by single clicking on the image, then press the Select button.

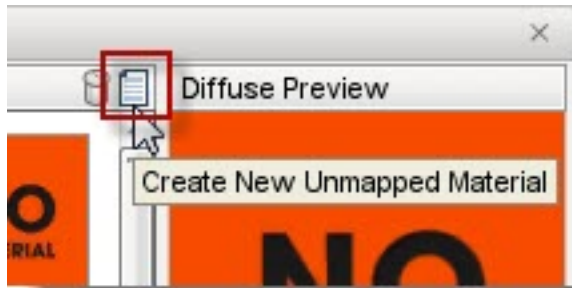
After the Material Selector dialog closes your scene will update and render your new sky choice:



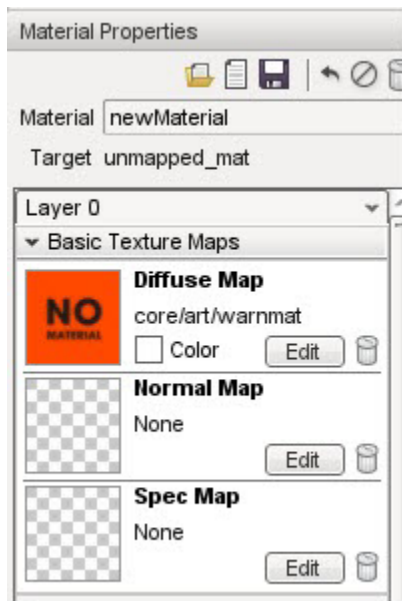
While technically rendering a material, a solid blue sky is not much to look at. If you want a photo-realistic Skybox created from digital images or Skybox creators, you will need to create a cubemap. You can create a cubemap using the Material Editor. Go ahead and click on the Material Editor icon to activate the tool:



When the Material Editor loads, look for the Material Properties section on the right side of the screen, underneath the Material Preview window. At the top of the Material Properties section, you'll see a small icon that looks like a blank piece of paper. That is the Create New Material button. Go ahead and click that icon.



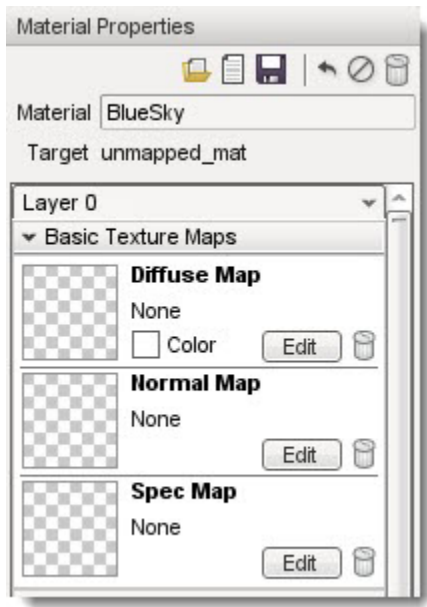
This action will create a base material definition similar to this:



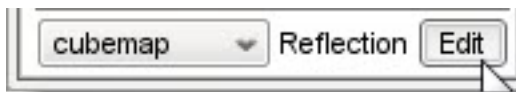
Cubemaps do not make use of diffuse, normal, or spec maps, so those can be deleted. Click on the trash bin icon Image:TrashIcon.jpg in each of the following groups - Diffuse Map, Normal Map, and Spec Map. This will delete the material associated with that group if it had one. The gray and white checkerboard pattern means that no material is assigned to a group. Change the name of your material by entering BlueSky in the text box next to the Mission label, then pressing the Enter key on the keyboard, then click the save button - the little floppy disk icon at the top.

**NOTE:** It is very important that you press Enter after typing your material name. If you just type the text and click directly on the save icon the material **will not be saved!**

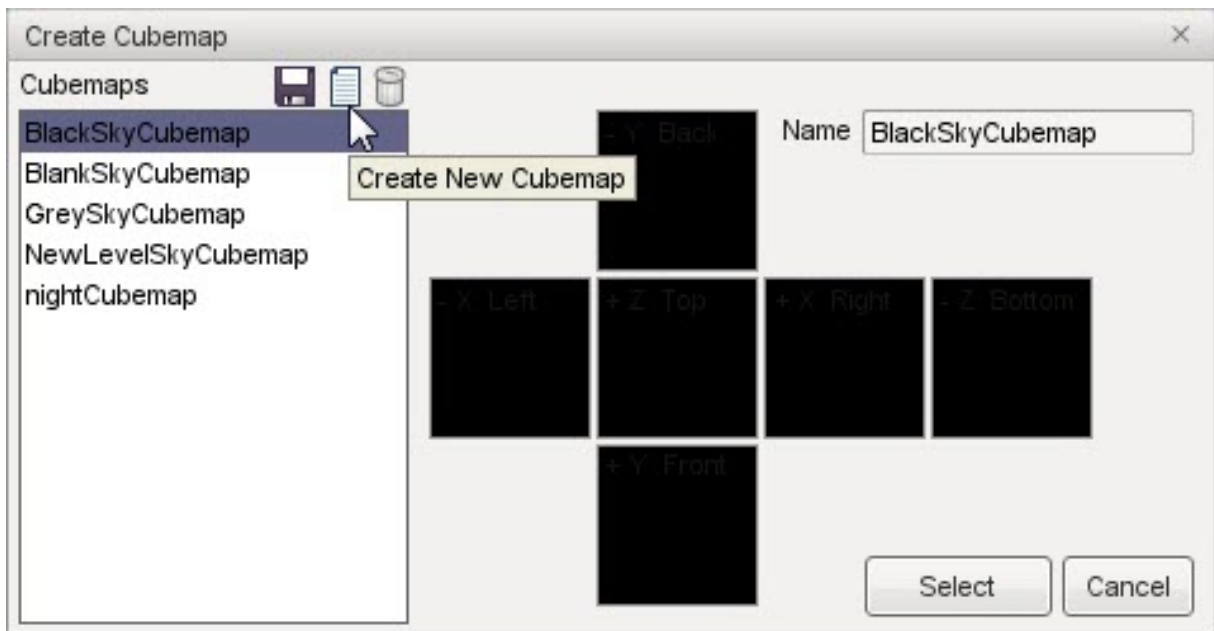
Your current definition should resemble the following:



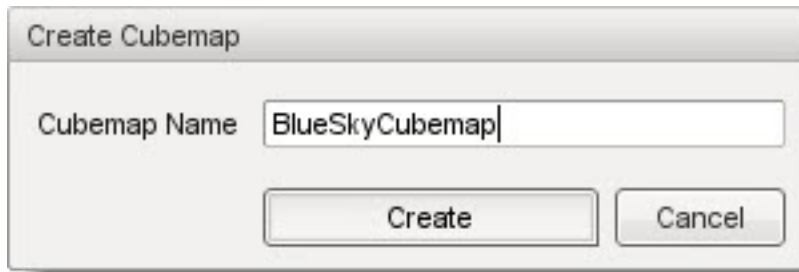
Now that the base material is setup, you can create a cubemap. Scroll down to the Advanced (all layers) section of the material in the Material Properties pane. Click on the heading to expand it if the content is not visible. In the middle is a drop down box that reads “None” and to the right is the word “Reflection.” Click this box then select the cubemap entry. Once you have done this, click on the Edit button:



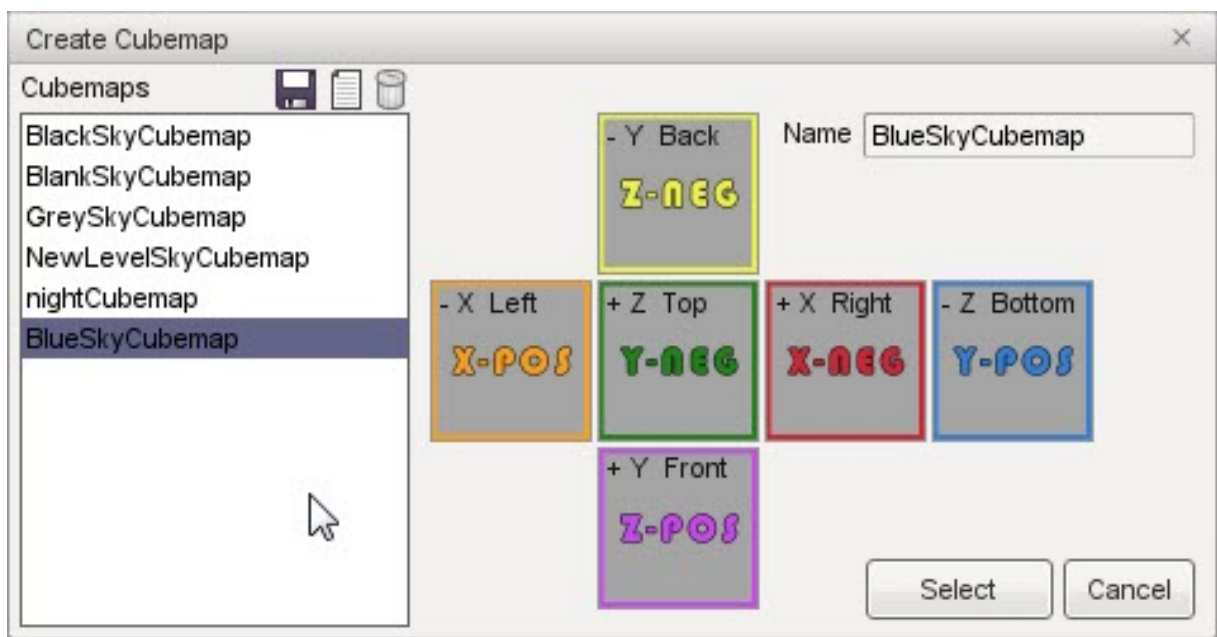
The Create Cubemap dialog will appear. From here you can select an existing sky cubemap or create a new one, which is what we are going to do. At the top of the cubemap list you will find three icons. Click on the page icon to create a new cubemap:



You will be prompted to name your new cubemap. Call it “*BlueSkyCubemap*”. Click the create button once you have finished.

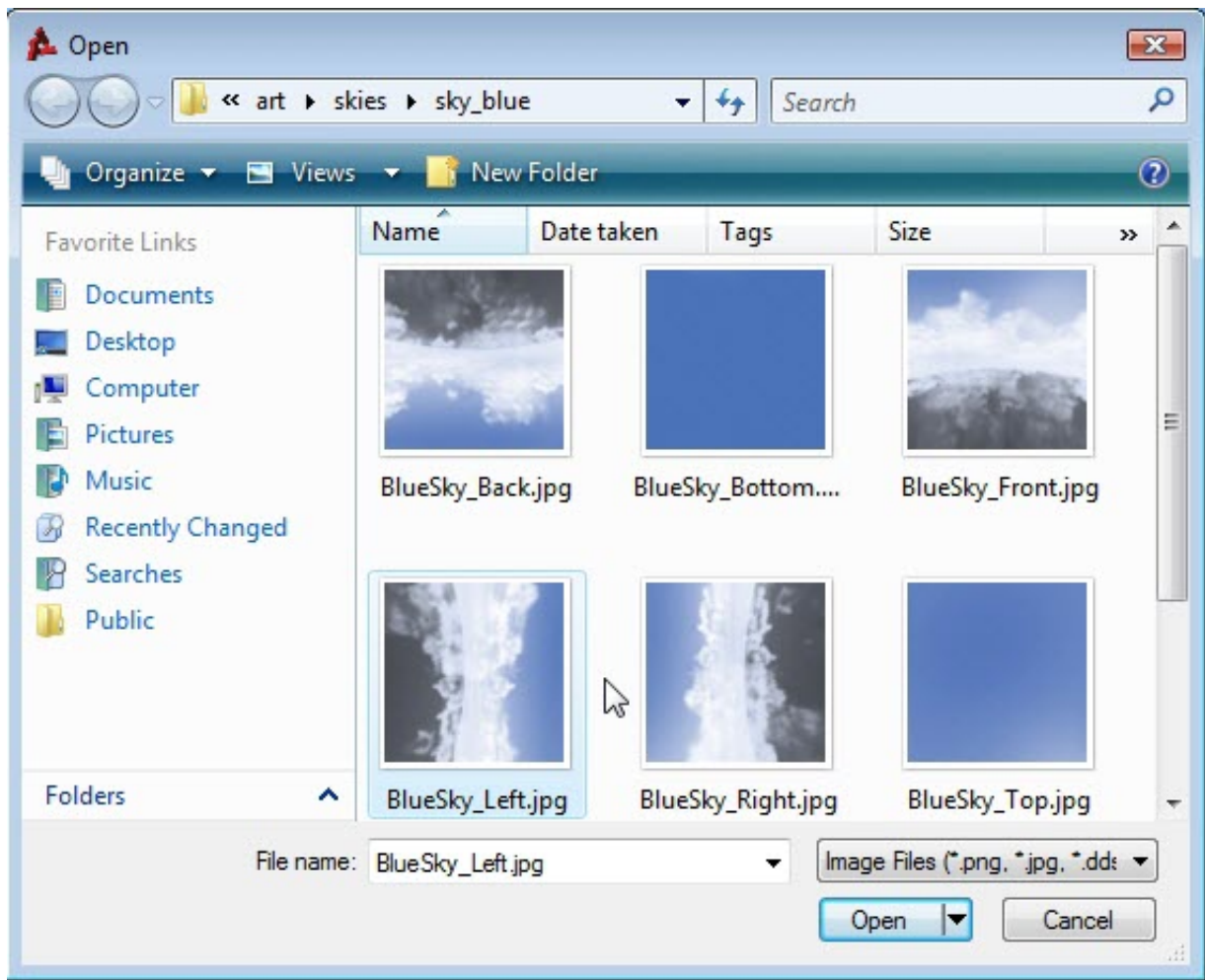


You should now see the cubemap template consisting of six colorful squares bordering each other. It may not make sense at first glance, but each square represents a section of a cube if you were to slice it at the seams and lay it out.



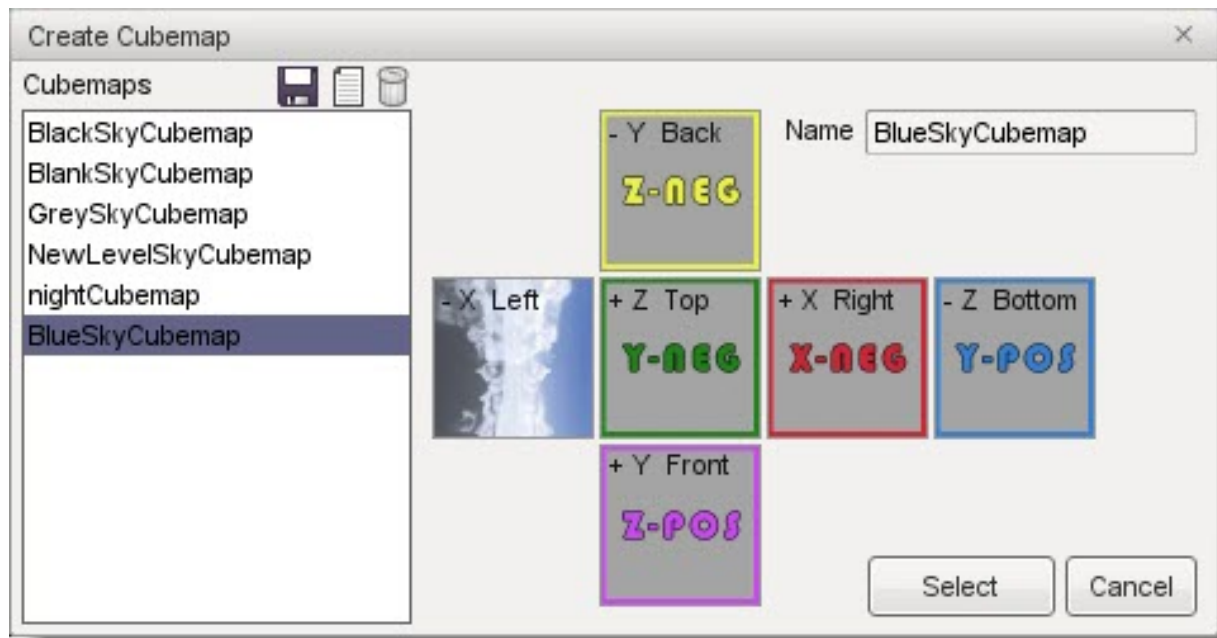
The +/- X, Y, Z labels are coordinates, but we have also given them directional names (Left, Right, Top, Bottom, Front, and Back). If you need more visualization, imagine you have a box placed over your head with the sky painted on the inside. If you are looking straight up, you are viewing the Top. If the box unfolded at the edges, you would see exactly what you are viewing here.

We will now build the cubemap for the sky to demonstrate this. Make sure you have [downloaded the Blue\\_Sky files](#), and unzipped them into your **Torque3D/My Projects/ game/art/skies/game/art/skies** directory, where is the name of the project you opened or created for this tutorial. Click on the X-POS (Left) icon on the cubemap display. A browser window will open:



Navigate to the directory where your sky art is located, click on the blue\_0004.jpg file then click the Open button. The sky image will be placed on the left section of your cubemap:



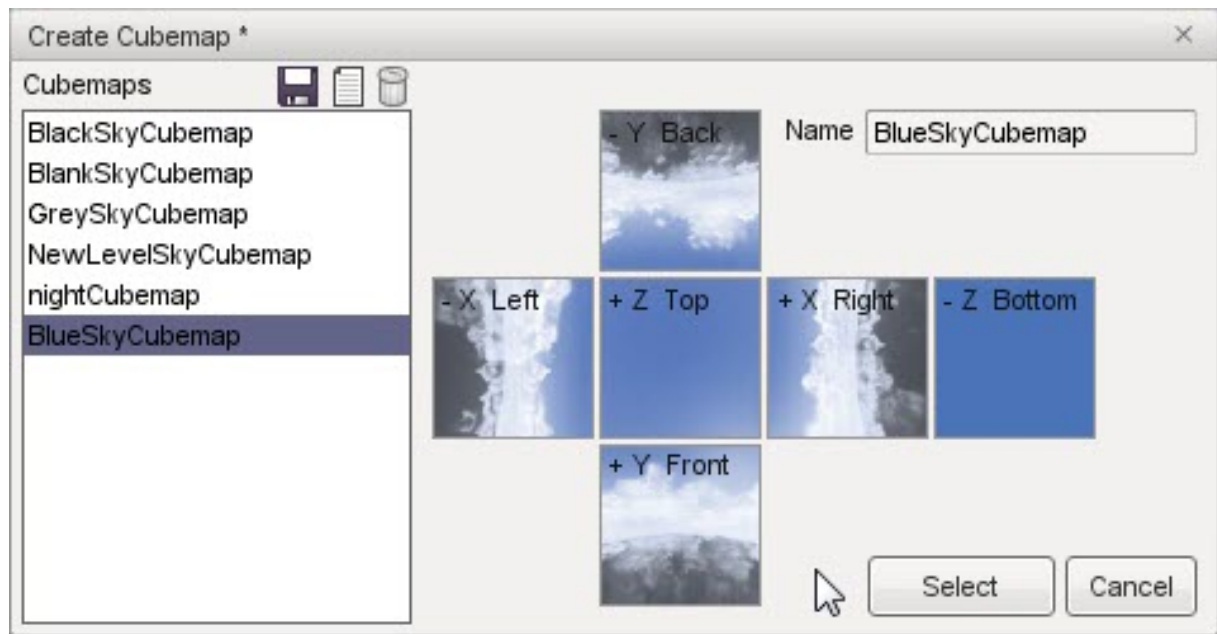


The “sky” portion of the image will be on the right side of the picture if you selected the proper image.

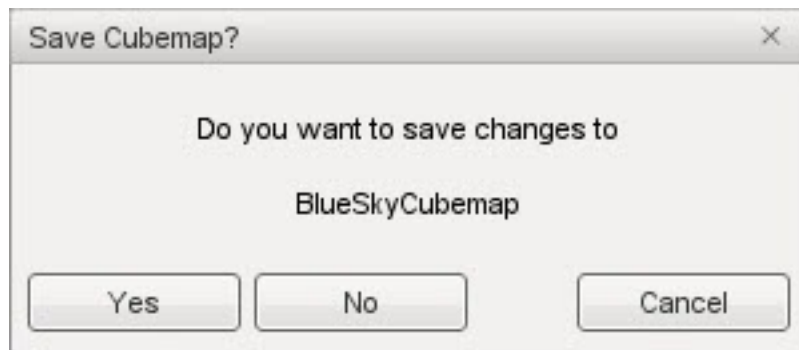
Repeat this process for Front, Right, Back, Top, and Bottom. Here is the placement for the different graphic files:

- Right - blue\_0002.jpg
- Front - blue\_0001.jpg
- Back - blue\_0003.jpg
- Top - blue\_0005.jpg
- Bottom - blue\_0006.jpg

Your final cubemap should look like the following:

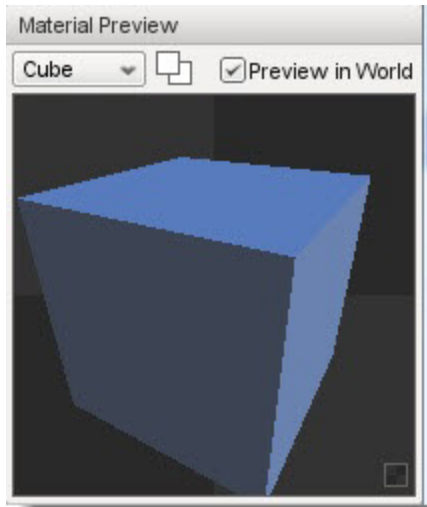


As you can now see, it is as if you are looking straight up at a sky with clouds surrounding your view. Once you are finished, click on the save icon (small floppy disk at the top-left of the dialog). You will be prompted to save your cubemap before continuing:



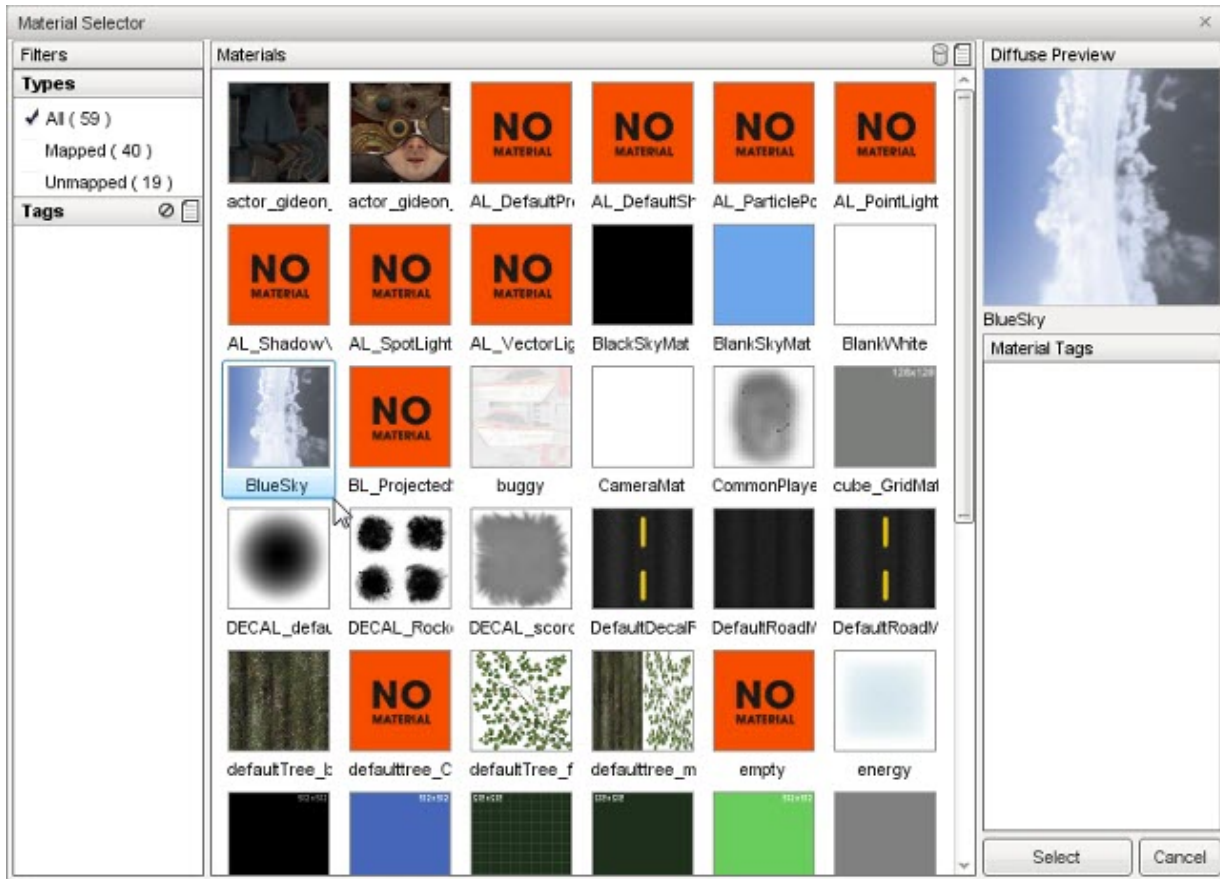
Click Yes to save the cubemap and return to the Create Cubemap dialog. Click the Select button to close the dialog and apply the new cubemap to the Material Properties.

When you are finished with your cubemap, your BlueSky material preview should show a Skybox with a strong reflection. This is completely normal, and just shows that this image can be applied to both Skybox objects and water reflections:

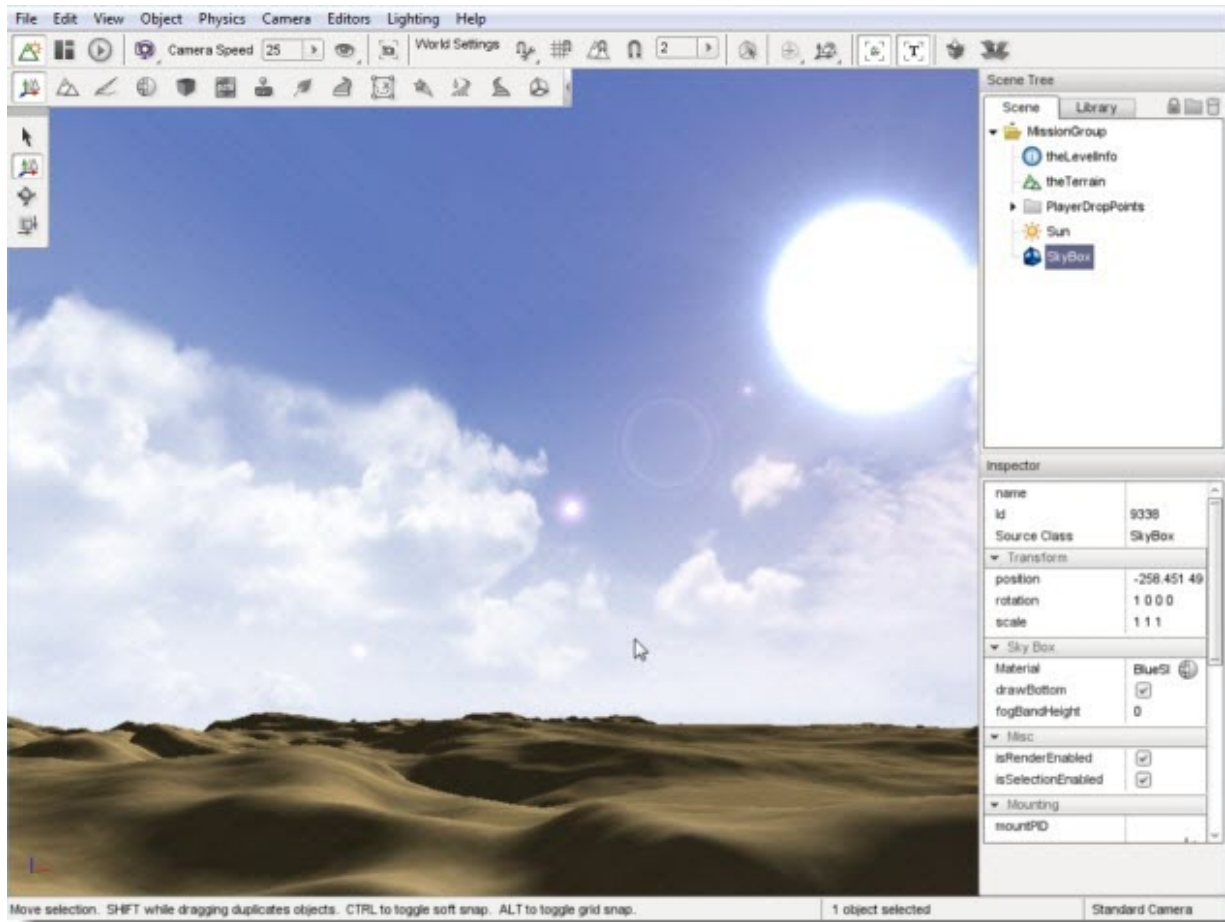


You will also notice that the new sky cubemap has been applied to the scene. This is only a preview and we have not yet told the scene to keep that cubemap as its Skybox. In the upper right hand corner of the Material Preview section there is a checkbox labelled “Preview in World”. If you uncheck this box you will see that the new Skybox has not really been assigned to your Skybox yet. We'll do that next.

Switch back to the object editor (F1 shortcut) and make sure your Skybox object is selected. Scroll down to the Skybox section of the properties, then click on the Material Selector icon .. image:: img/GCMatPropIcon.jpg in the Material field. When the Material Selector appears, locate the BlueSky material and click Select.

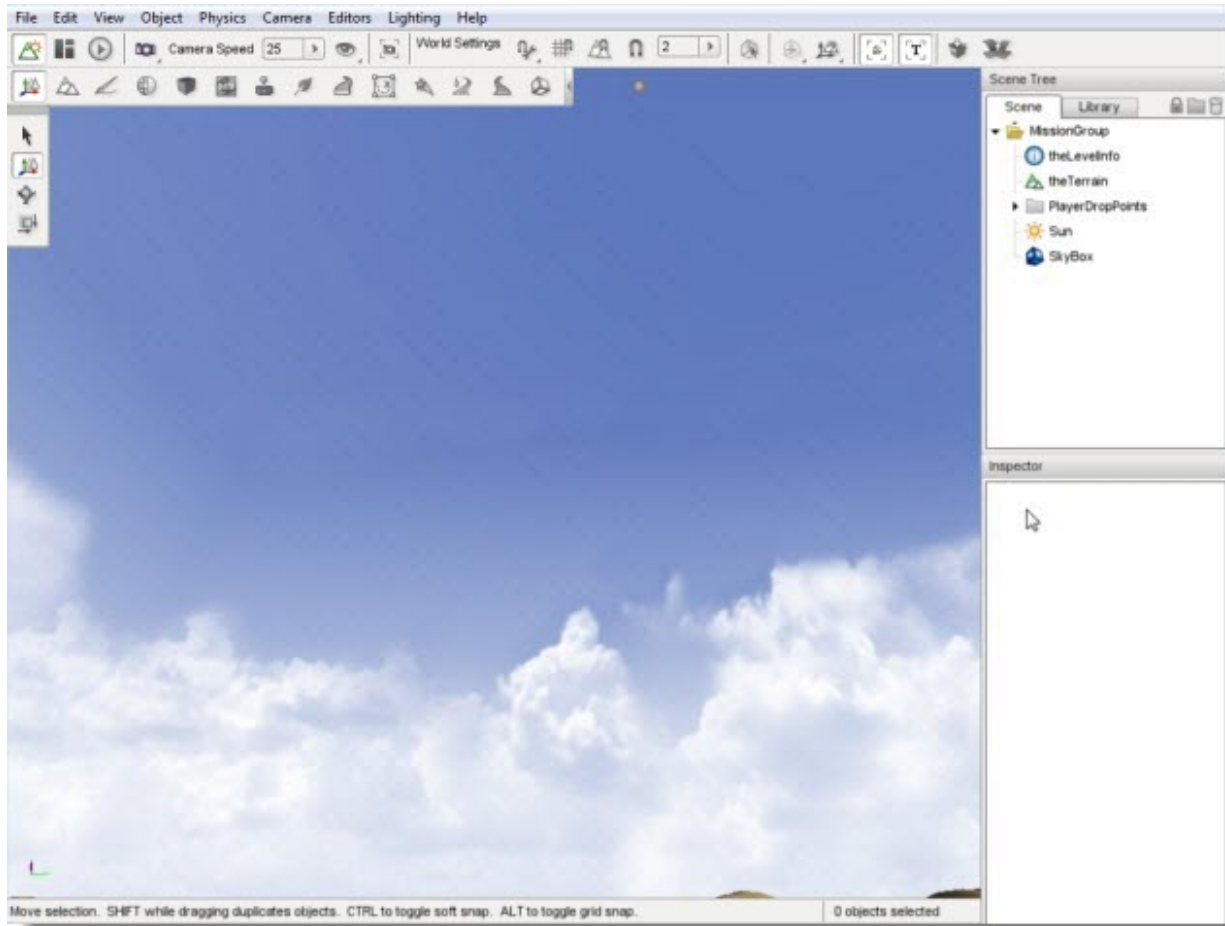


Your Skybox should now be rendering the cubemap you created earlier. Instead of a bland, solid color you now have a more realistic Skybox with some clouds simulated in the distance. Much more interesting.

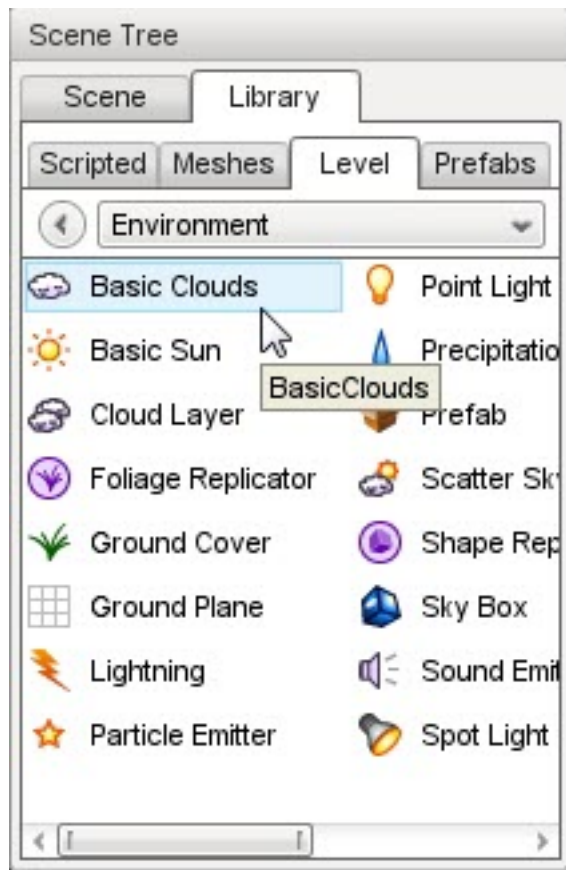


### 14.2.7 Adding Clouds

Finally, we are going to add some real clouds that are not a part of the Skybox. You have two choices for cloud layers: Basic Clouds and Cloud Layer. Since this a simple scene, we are going to go with Basic Cloud. Before proceeding, look at your Skybox and carefully note how the sky looks without a cloud object:



When you are ready to add clouds start by switching to the Library tab in the Scene Tree panel. Click on the Level tab then select the environment folder. Once that is open, locate the Basic Clouds object (not the Cloud Layer object):

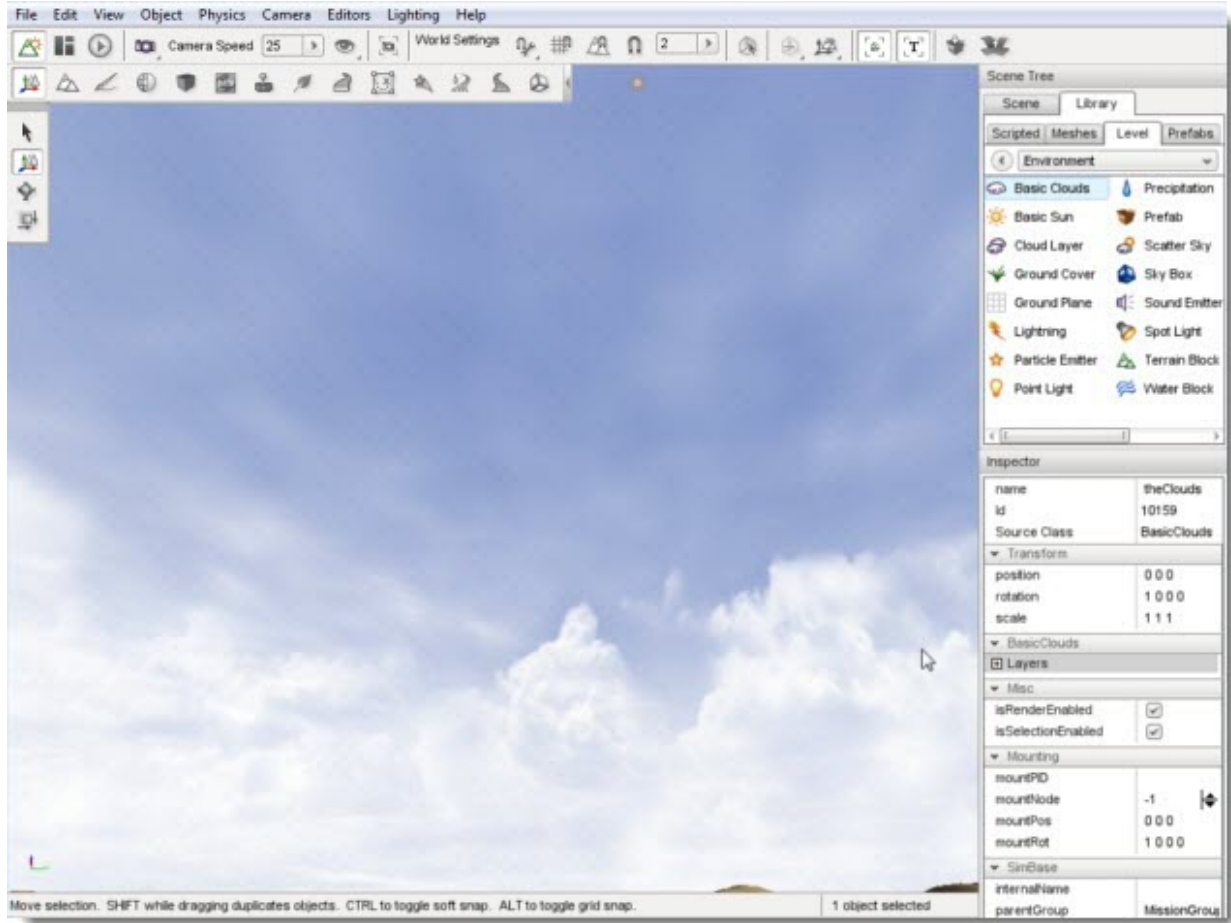


Double click the Basic Cloud object. A dialog will appear allowing you to fill out initial details. The Object Name is what you want your Cloud layer to be called in your MissionGroup. For now, just type in theClouds for the name.



Click Create New and the Basic Clouds object will be added to your level. Three separate cloud layers will be rendering and moving across the sky slowly:





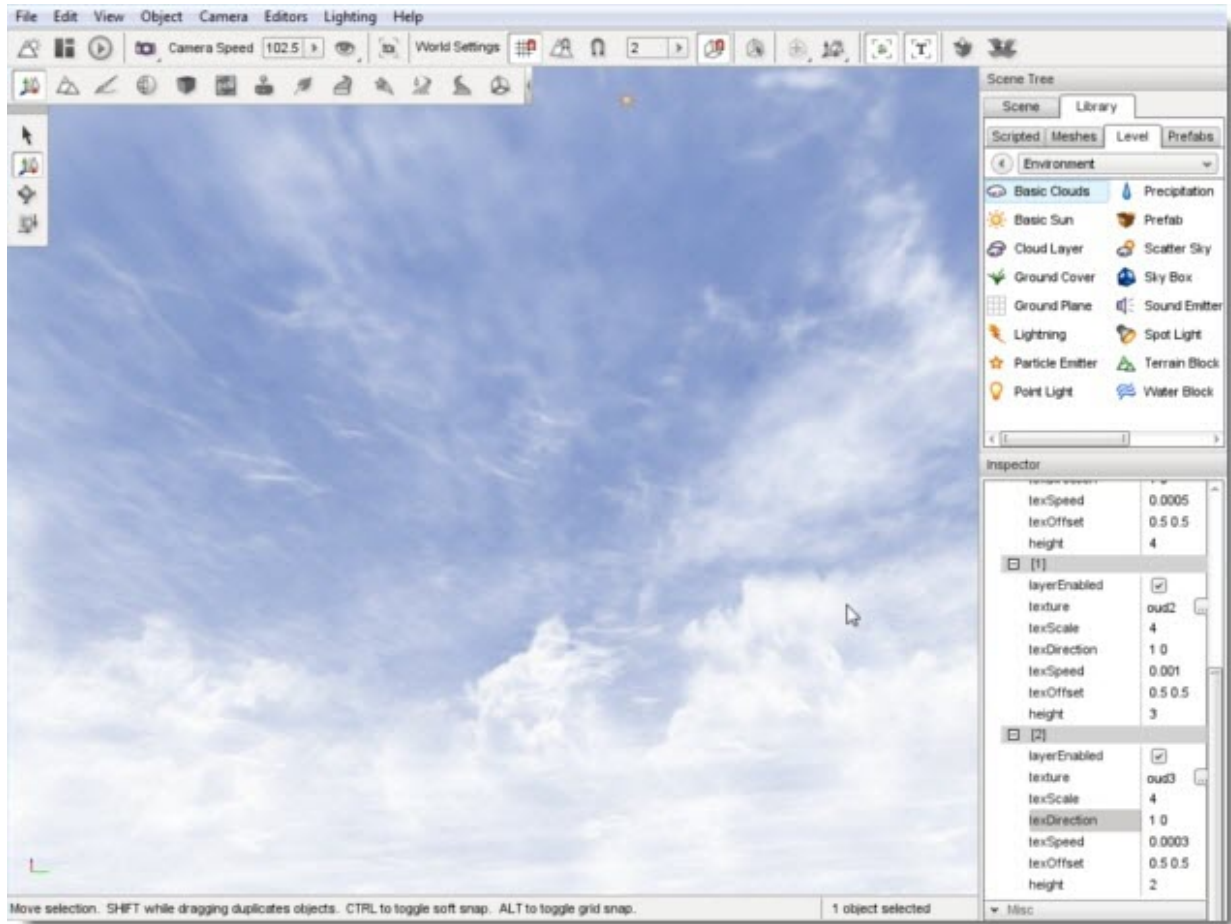
Select the new cloud layer - switch to the Scene tab in the Scene Tree pane, and select **theClouds** from the list. Scroll to the **BasicClouds** section in the Inspector pane. Expand the **Layers** entry by clicking the + icon, then expand the entries [0], [1], and [2]. This displays the information on the three layers in this object.

The stock object will populate the three layers with sample cloud images. These are located by default in /My Projects/<Project>/game/core/art/skies. If for some reason you do not have these assets, you can download them by [CLICKING HERE](#). Each image is a transparent PNG, which means that portions of the image are clear so that you can see the background through them. Transparency is a requirement for the clouds to render properly and with realistic depth since you need to be able to see each layer without it blocking those behind it.

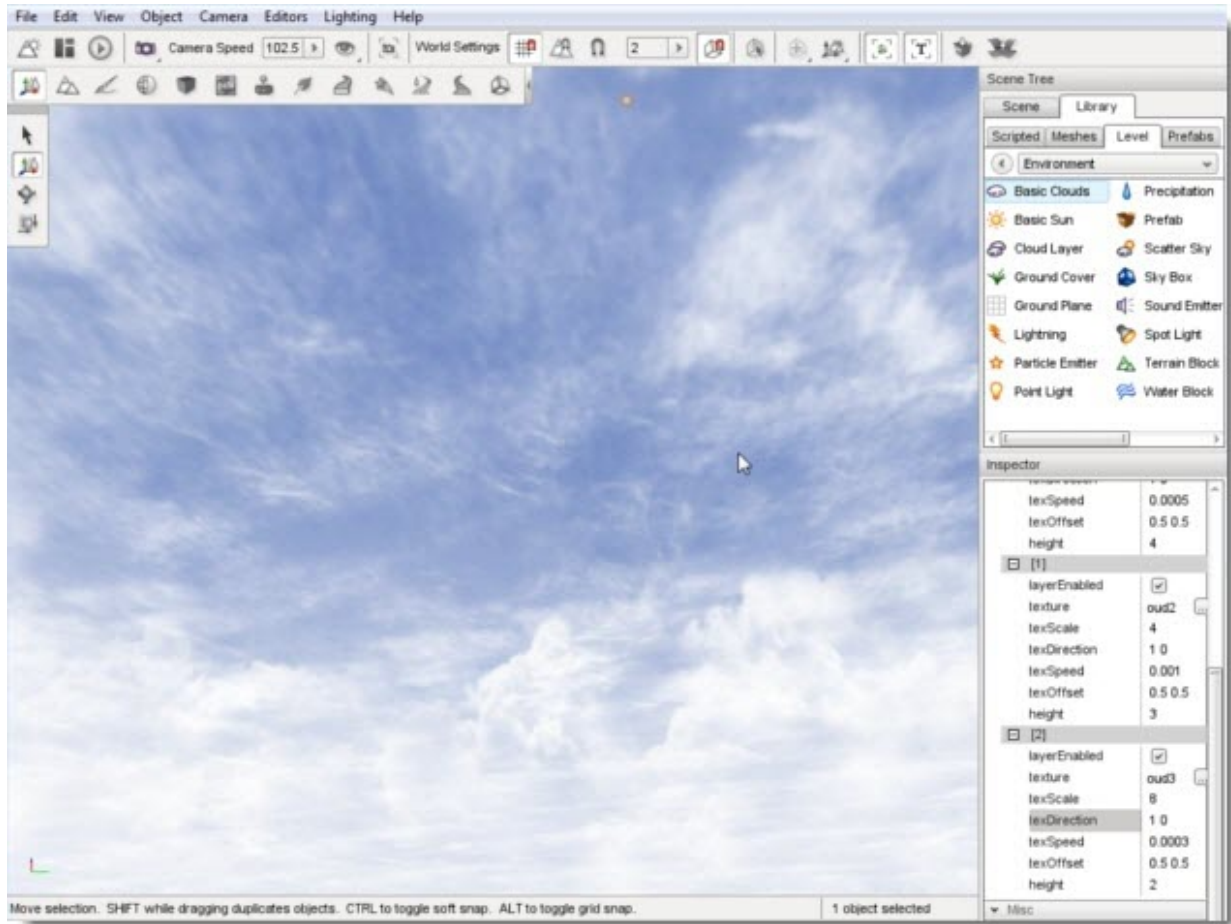
Right now, the cloud layers are stretched and look very hazy, or perhaps not very visible at all depending on your computers monitor brightness and contrast. This stretching causes the clouds to not match the static clouds that are present right within the Skybox cubemap, so we are going to make some changes. A more desired appearance will be wispy, very white clouds.

The **texScale** property determines how often the texture will repeat on this layer. Increasing texScale will cause the texture to be repeated, which is referred to as being “tiled”, over a smaller area, which is be useful for low detail textures such as those used in this tutorial.

Increasing the texture repeat will make the layers appear to be more detailed and defined. For each layer [0], [1], and [2], set the **texScale** property to 4.:

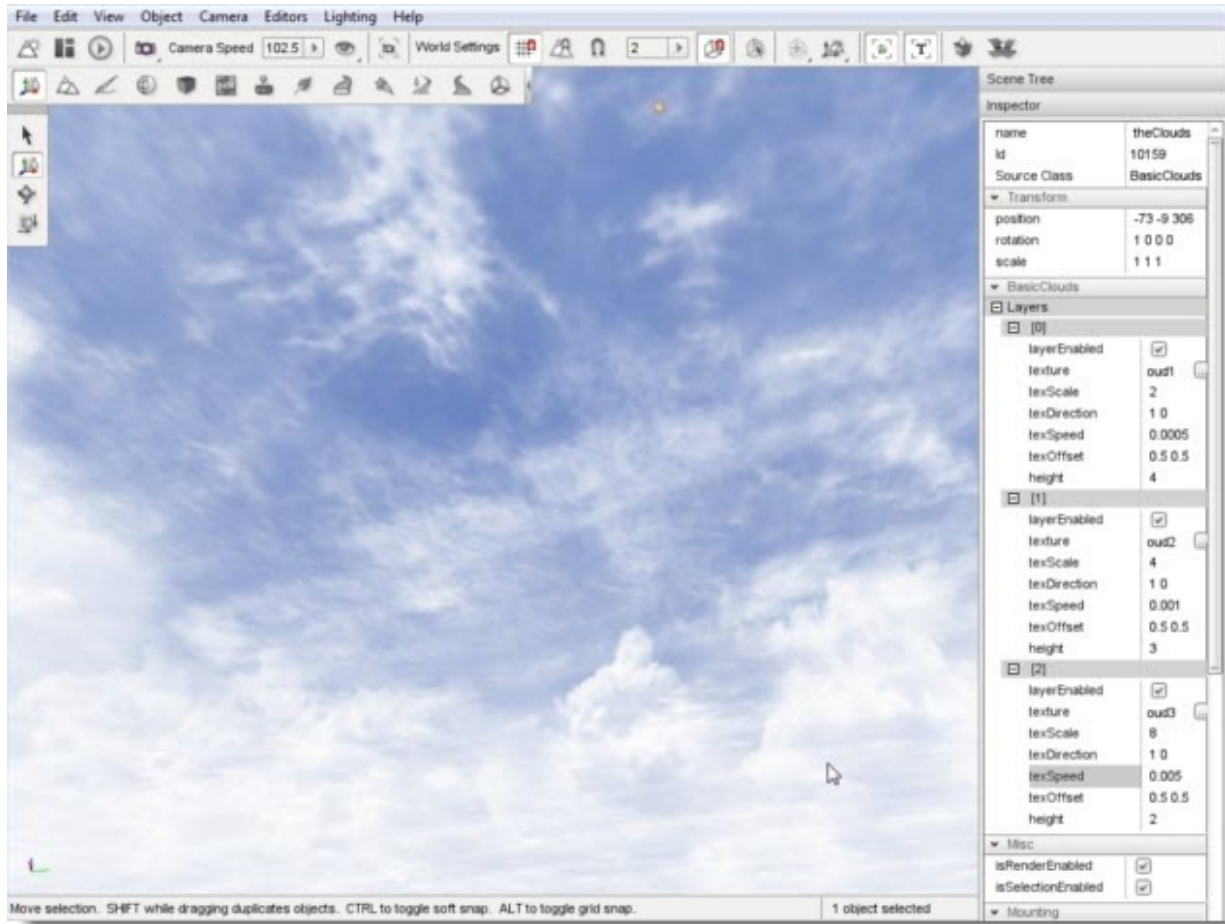


This is a good start, but the closest layer should be the most defined. Scroll down to **Layers[2]** and set the **texScale** to 8. This texture will repeat more often, making it appear to be closer and clearer:

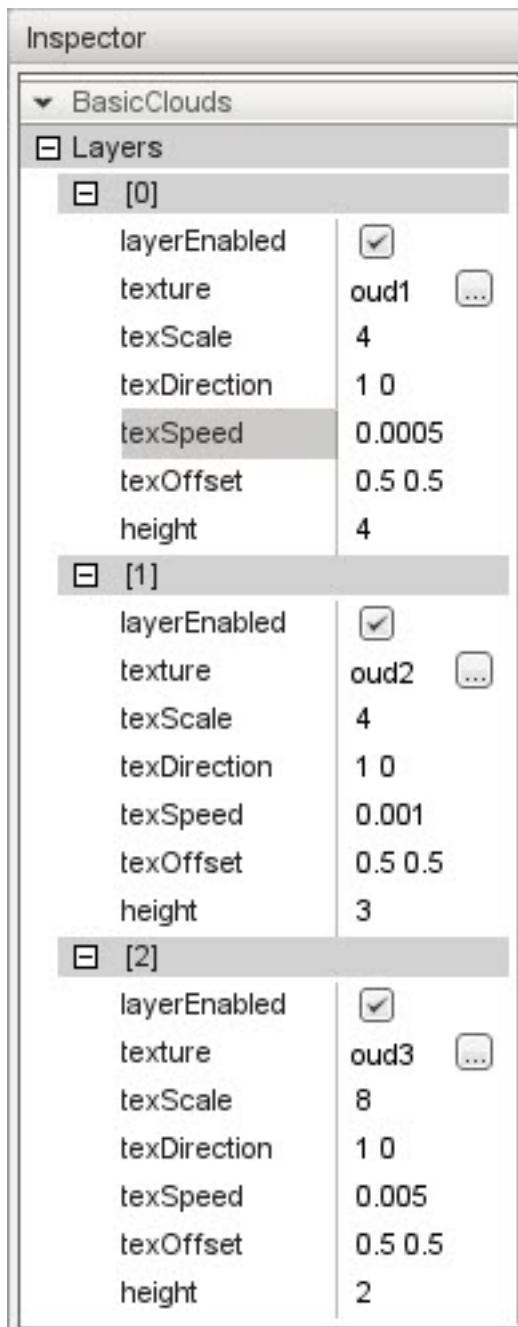


The last adjustment will affect the movement of the clouds. Since Layers[2] is the closest and most defined, wind simulation should be more dramatic. In other words, we want the closest cloud layer to move the fastest. A single property controls how fast the cloud layer moves: **texSpeed**.

If the property is set to 0, the cloud layer will not move. The higher the number, the faster your cloud texture will scroll across the sky. The stock value for Layers[2] texSpeed is 0.0003. Increase this value to 0.005, which will cause the clouds to scroll faster:



For reference, the following are the properties I have set in my scene. You can use these, or continue to make adjustments to your liking:



In the end, you should have a very nice looking blue sky with realistic clouds. The more clouds farther in the distance are presented by the Skybox, while the closer clouds are generated by the Basic Clouds object:





## 14.2.8 Conclusion

In this tutorial, you learned how to create a basic sky using the Sun, a Skybox, and Basic Clouds. These objects are simpler and have less impact upon a computer's performance than the Scatter Sky and Cloud Layer. With the right images and software, artists can make really amazing Sky boxes and cloud textures.

Other tutorials make use of these objects, but in different ways. Feel free to continue experimenting with this scene to see what results you can come up with.

## 14.3 Adding Lakes

### 14.3.1 Introduction

In this tutorial, we are going to create a lake using Torque 3D's WaterBlock object. Since this will be an isolated body of water, you should not need more than one WaterBlock to get the job done. We will be adjusting several WaterBlock properties to obtain a very placid appearance.

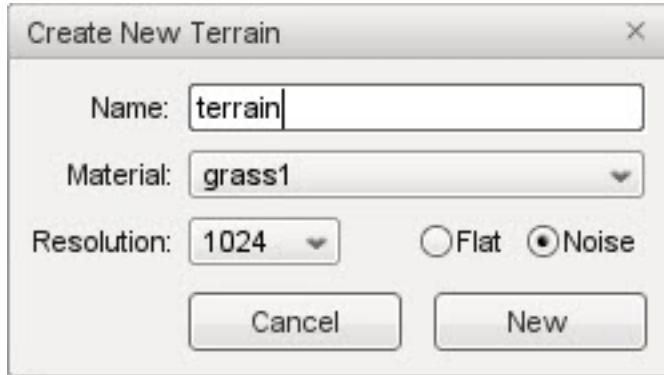


### 14.3.2 Setup

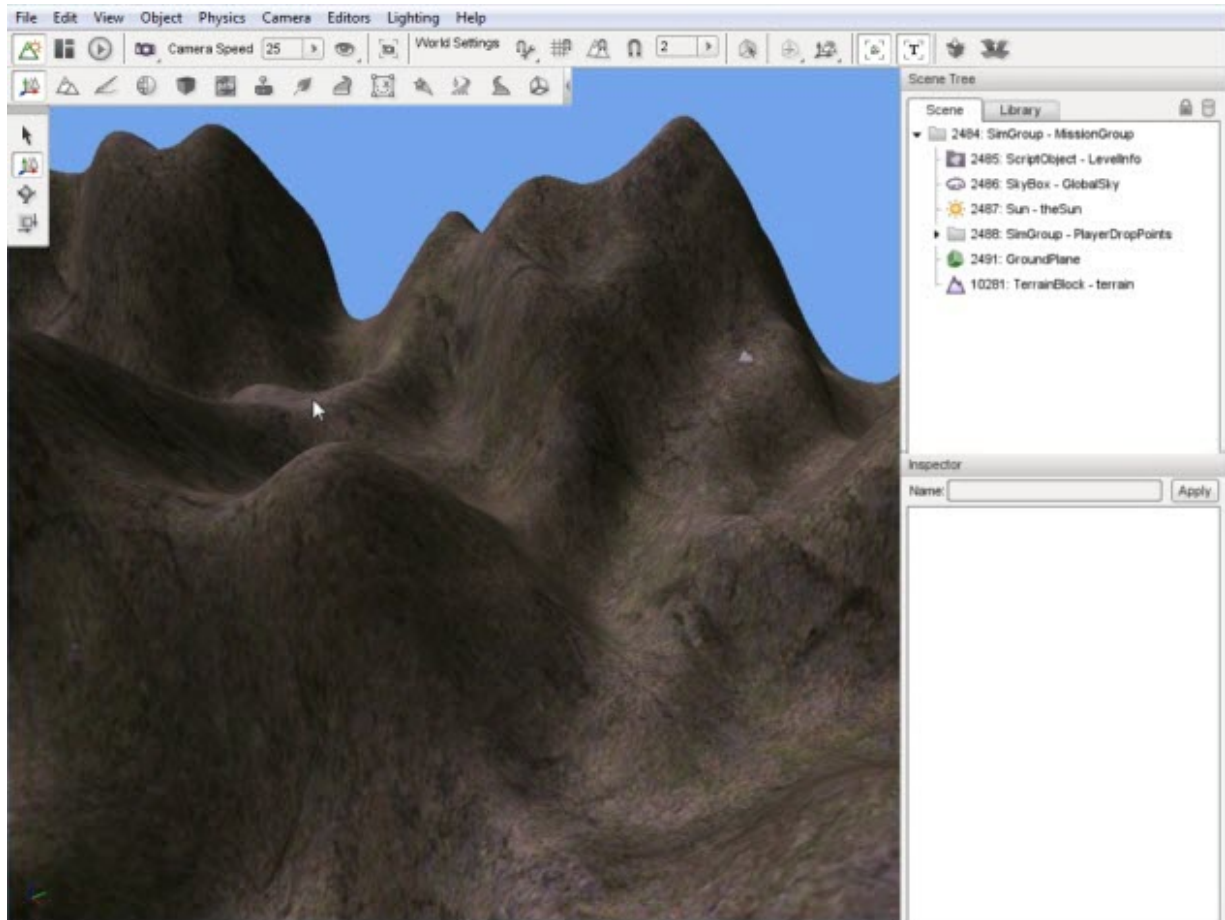
This article will use a new project created from the Full template, which includes sample assets for testing and learning. To save time and focus on the World Editor we will use the sample assets and learn about asset creation later.

None of the modifications you are about to make are required for future tutorials, so feel free to create a new level or use an existing one for testing. As long as you have access to existing materials, you are good to go. For this article, we are going to be using a new level.

In order to simulate a realistic body of water, we are going to start by adding a new TerrainBlock. If you do not know how to add terrain, [click here to review the TerrainBlock Guide](#). If you are familiar with the process, go ahead and create a new terrain from the File Menu. We do not need anything fancy, so use the following information:



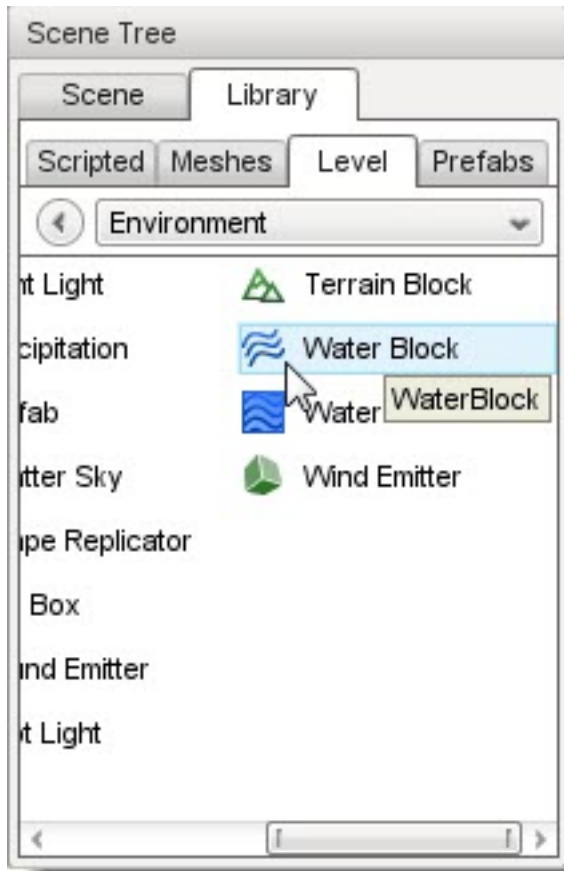
This should result in a mountainous terrain with plenty of valleys in which to add a body of water. If you are not happy with the results, use the Terrain Editor or create a new TerrainBlock all together. In the end your terrain should look similar to this:



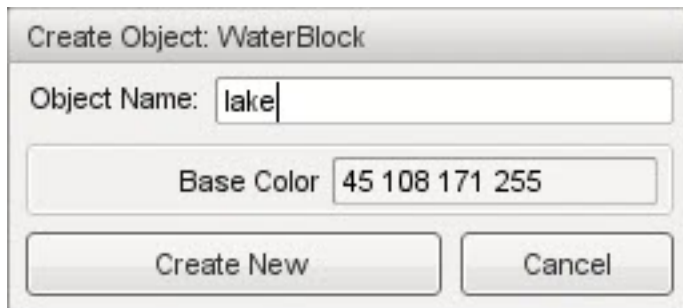
Now that we have basic terrain, we will move on to the actual WaterBlock creation.

### 14.3.3 Adding a WaterBlock

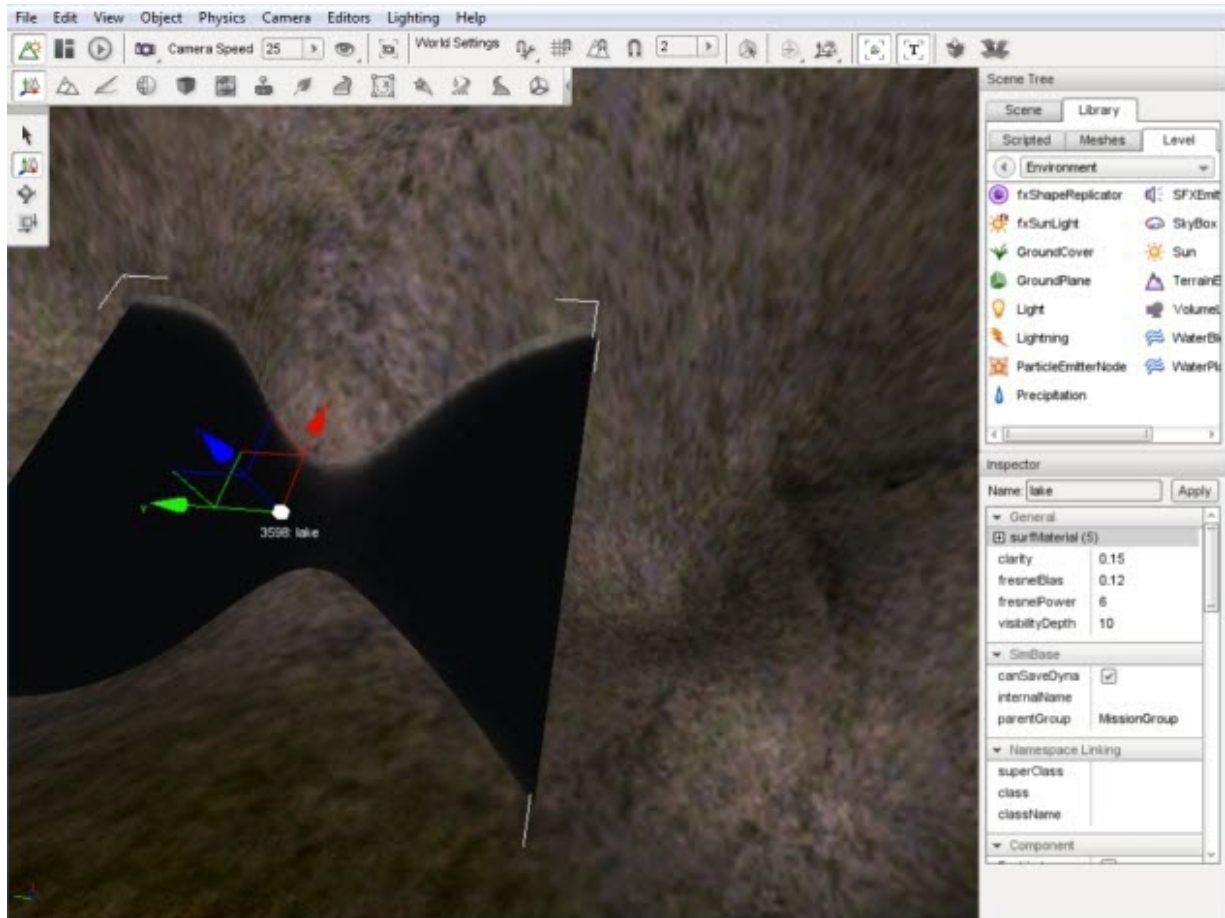
To add a Water Block: switch to the Object Editor tool; click the Library tab; click the Level sub-tab; double-click the Environment folder; then locate the Water Block entry:



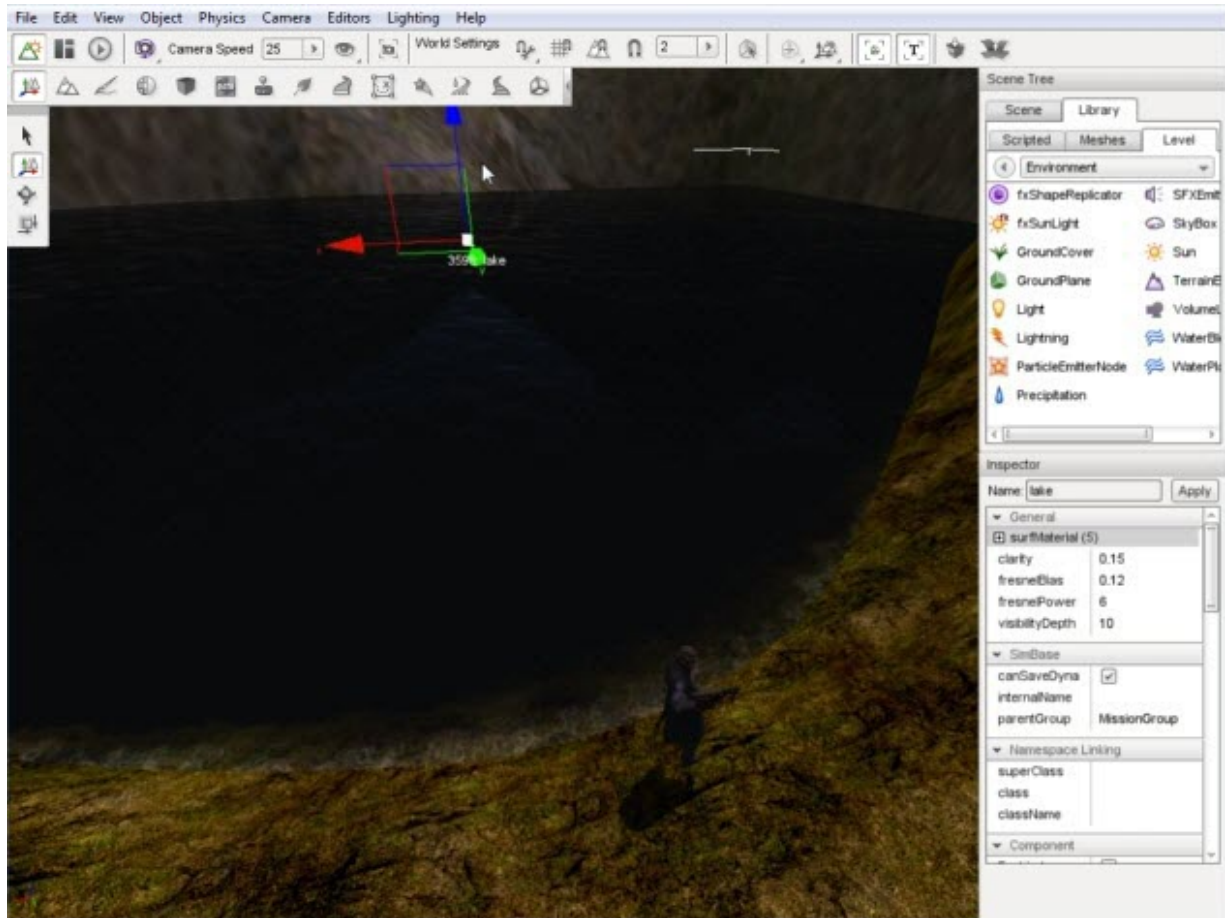
Double-click the Water Block entry. The Create Object dialog box should appear:



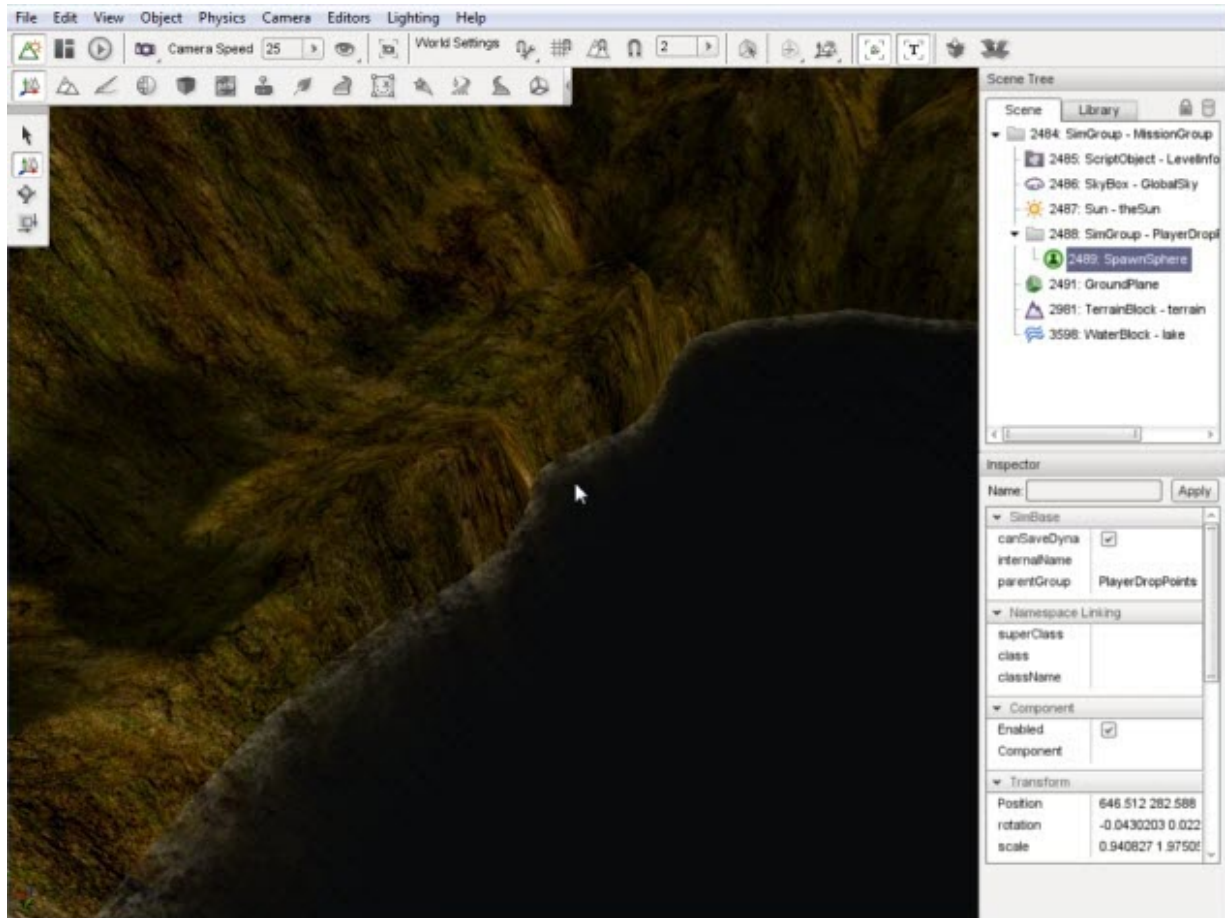
Enter a name for your new Water Block then click the Create New button. A square body of water will be added to the scene. This is your WaterBlock. Like any other object, you can manipulate its transform using the gizmos.



Before we proceed with modifying the WaterBlock properties, we should move the body of water to a more appropriate area. With the object selected, use the Transform Tool to move it to a valley in your terrain. It does not have to be perfect, but it will help simulate a lake in a crater.



The WaterBlock edges will clip appropriately and reflect the interaction with the terrain via incoming wave textures wherever it meets the land.



Now that we have a positioned WaterBlock, we can begin editing its properties.

### 14.3.4 Color and Fog

We are now going to perform a few minor manipulations. For this article, we are going to take a stock WaterBlock and turn it into a lake. Landlocked bodies of water such as lakes tend to be calmer than an ocean or river.

The default WaterBlock is too choppy, fast, and murky. We are going to simulate a calmer, clearer lake similar to the placid Crater Lake in Oregon:



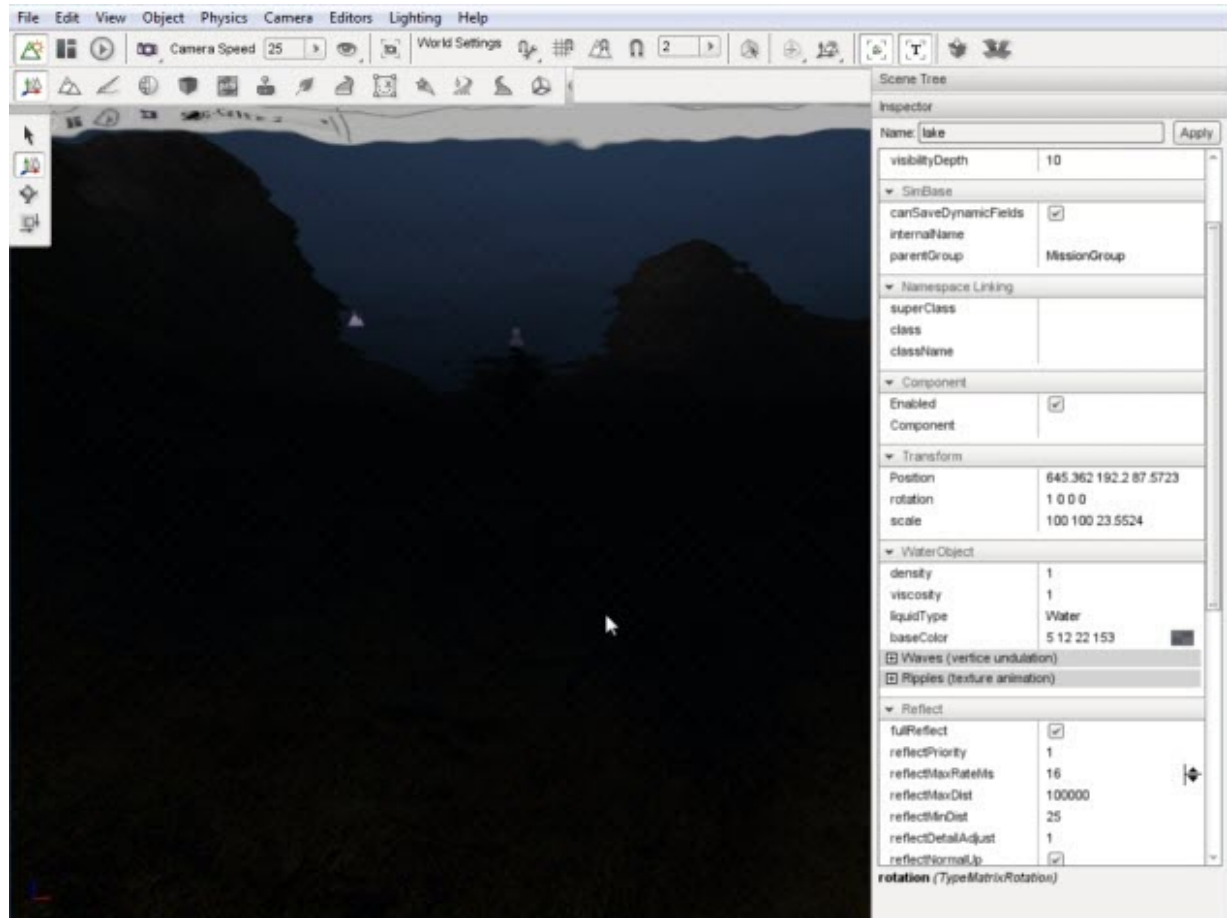


We will not need to modify all of them to simulate a lake. First, we are going to decrease the murkiness of the WaterBlock, which represents how clear the water is. Scroll through the properties until you see the Underwater Fogging section. Change the waterFogDensity field to 0.01.

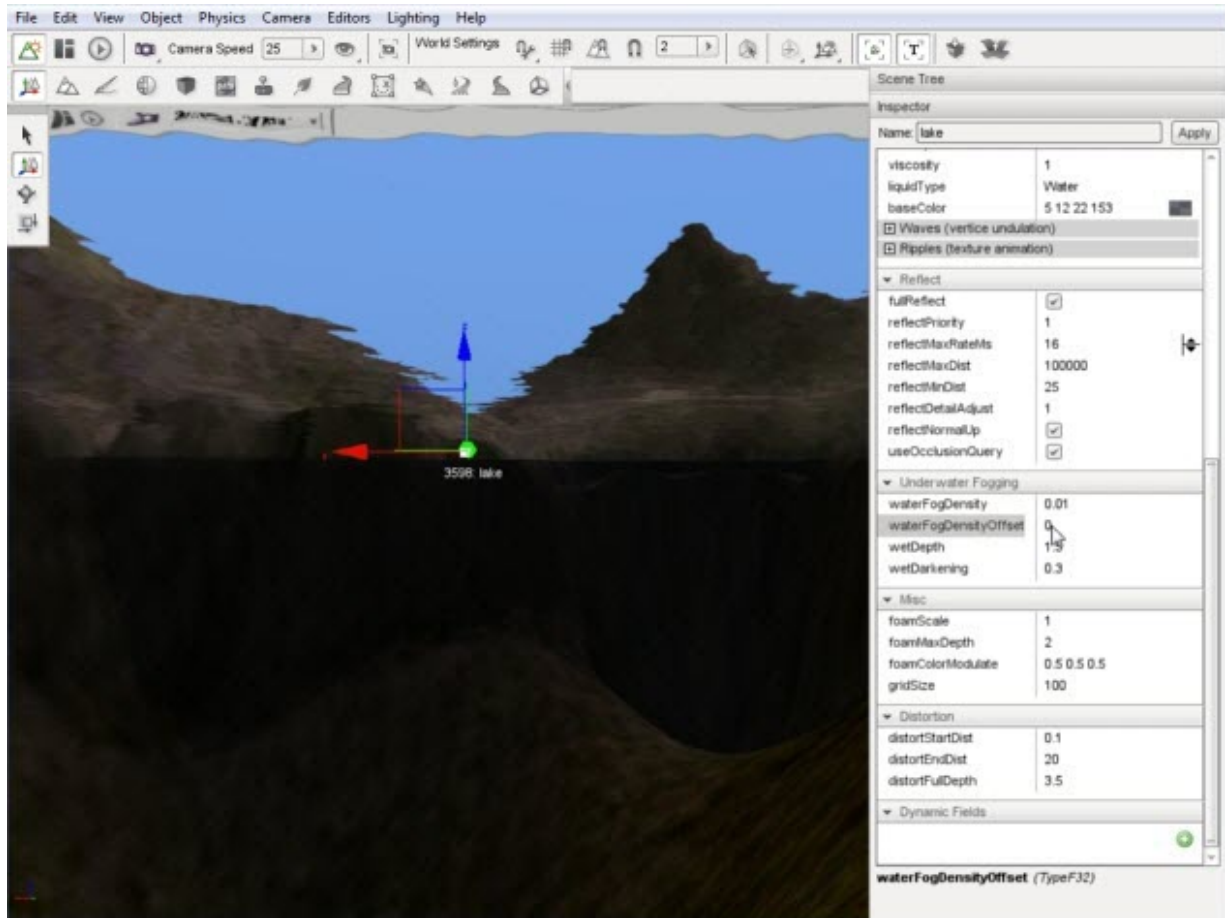
▼ Underwater Fogging	
waterFogDensity	0.01
waterFogDensityOffset	0
wetDepth	1.5
wetDarkening	0.3

This is a drastic reduction. If you move your camera under the water, the difference is immediately noticeable.

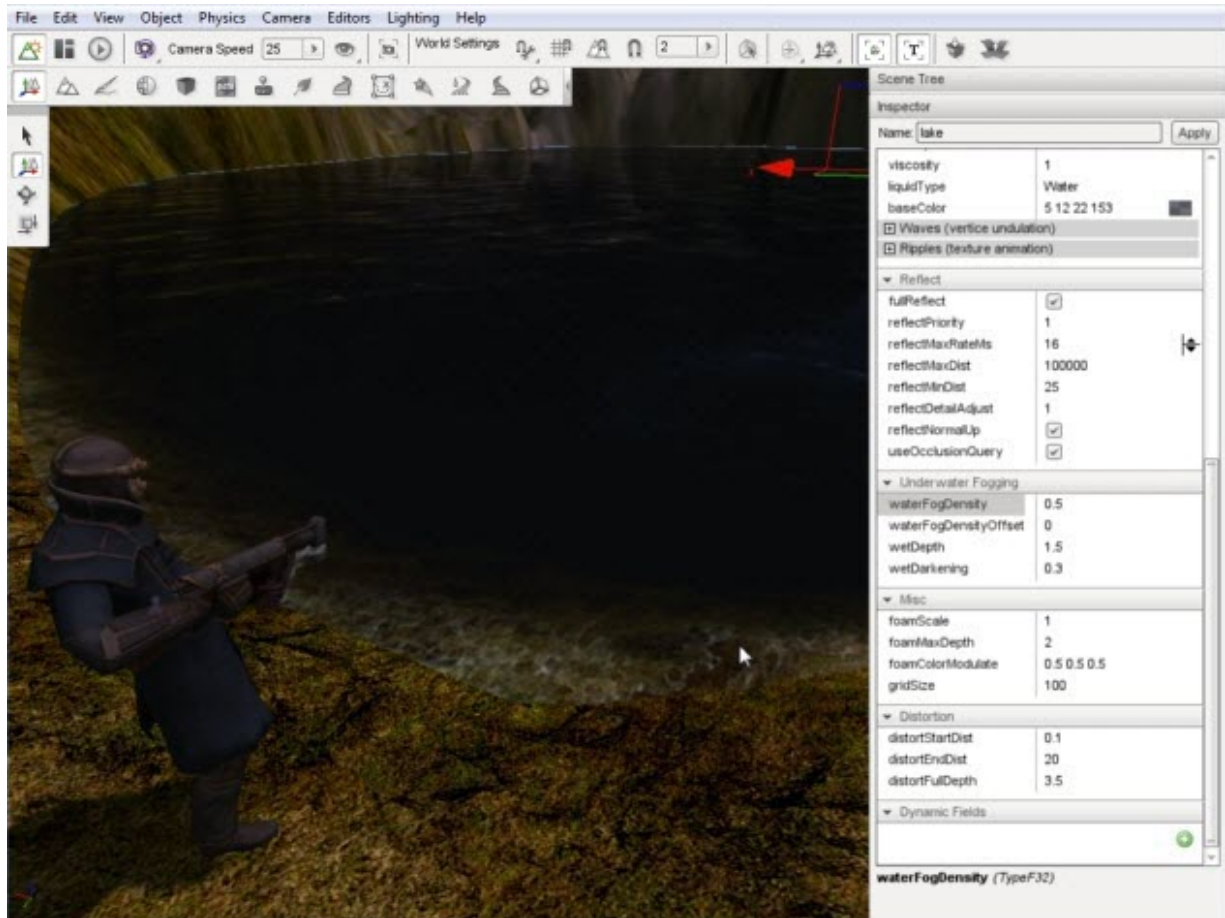
### 0.5 Fog Density Under Water



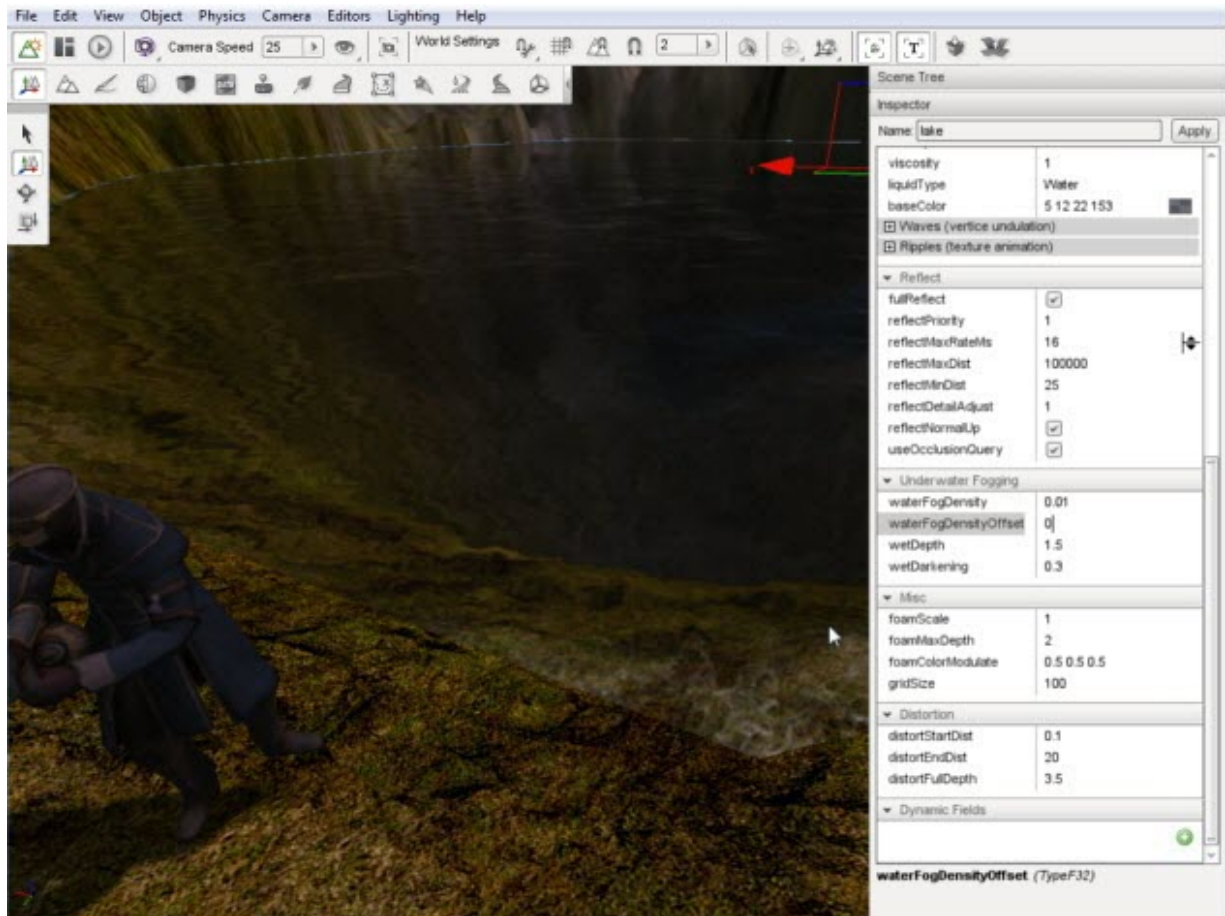
### 0.01 Fog Density Under Water



### 0.5 Fog Density Above Water



#### 0.01 Fog Density Above Water

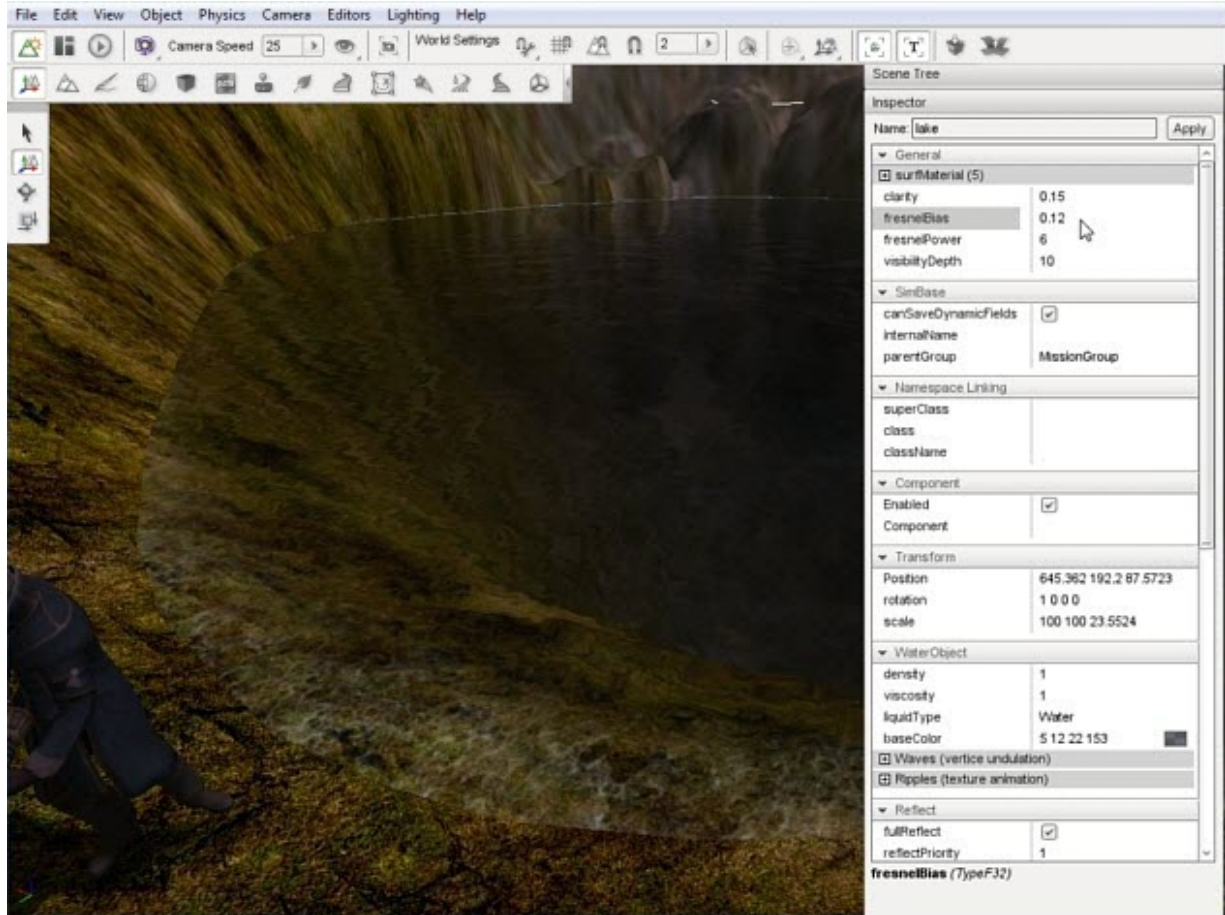


Our lake is a little too clear at this point. Instead of playing around with the fog, we are going to adjust another property to reflect the color changes at lower depth. Under the Water Object section of the properties, find the `fresnelBias` field. Change the value to 0.01.

The change is subtle, but if you focus on the deepest points of the water you should see a difference. What we have done is decreased the water's reflection amount based on the underwater fog intensity, which will stronger toward the middle where the water is deepest.

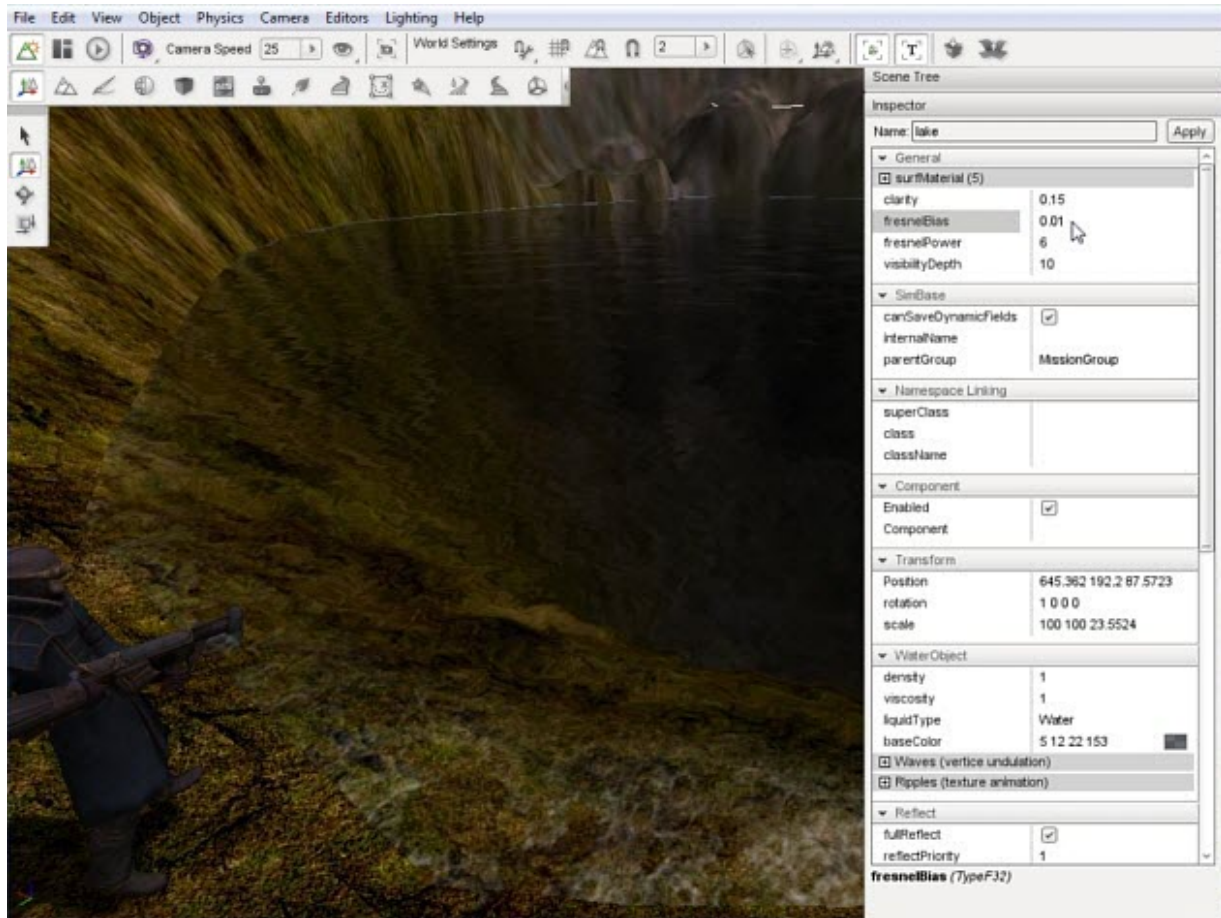
## 0.12 Fresnel Bias



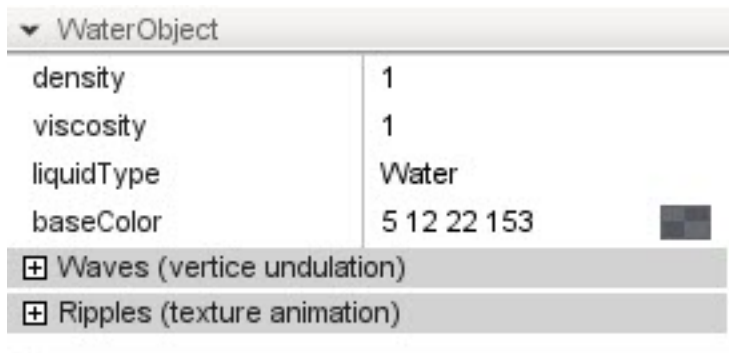


### 0.01 Fresnel Bias

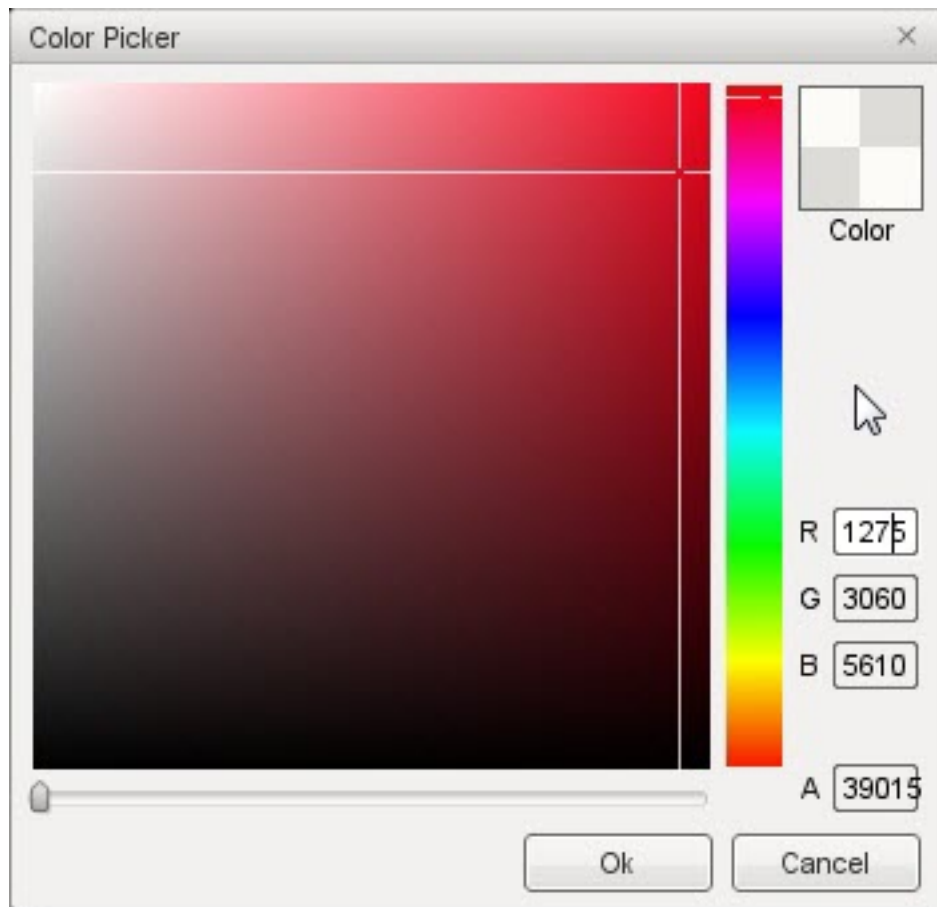




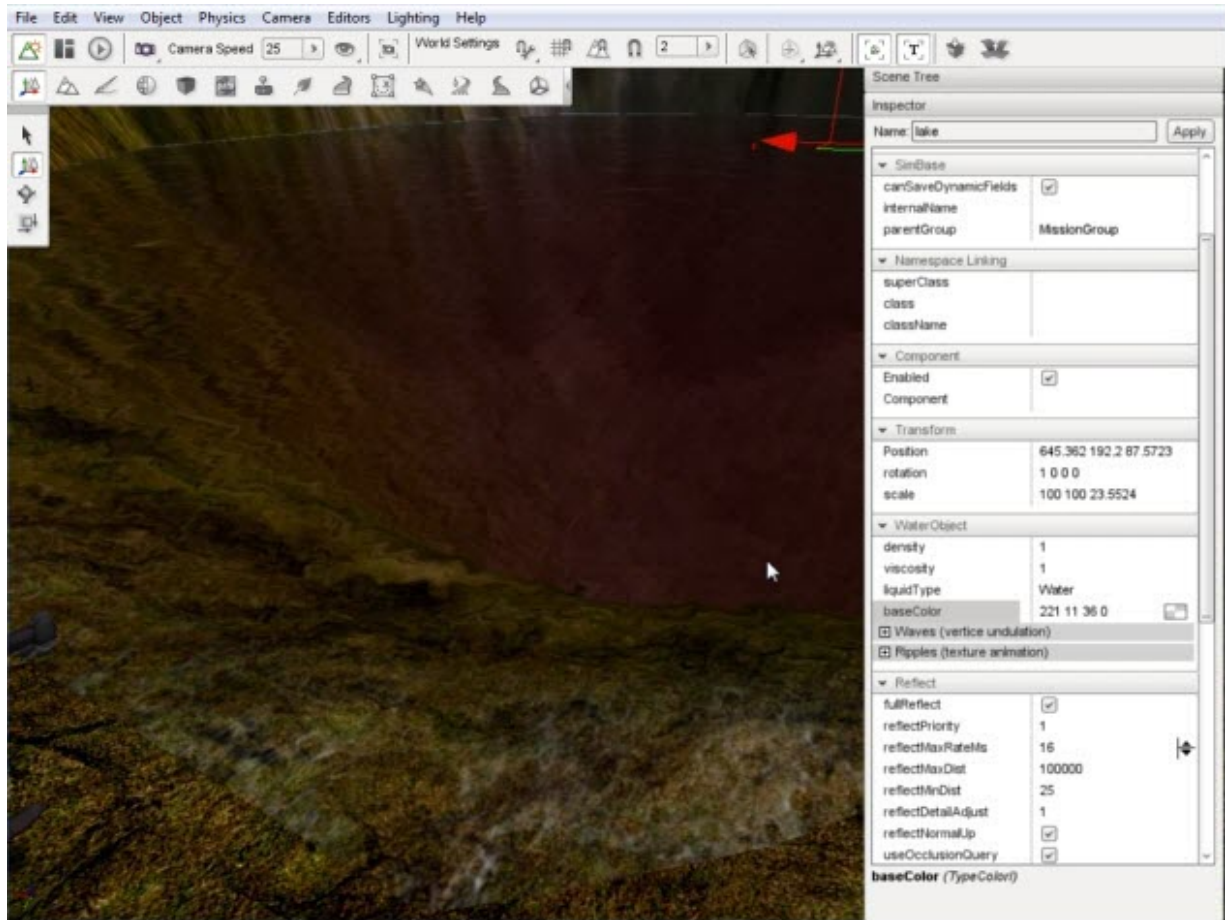
We will make one last adjustment to polish the high level appearance of the lake. The stock color is a little dark, but that can be modified easily. Under the WaterObject property section there is a field called baseColor. You can modify the value manually, but we are going to use a color picker by clicking on the square box at the end of the line.



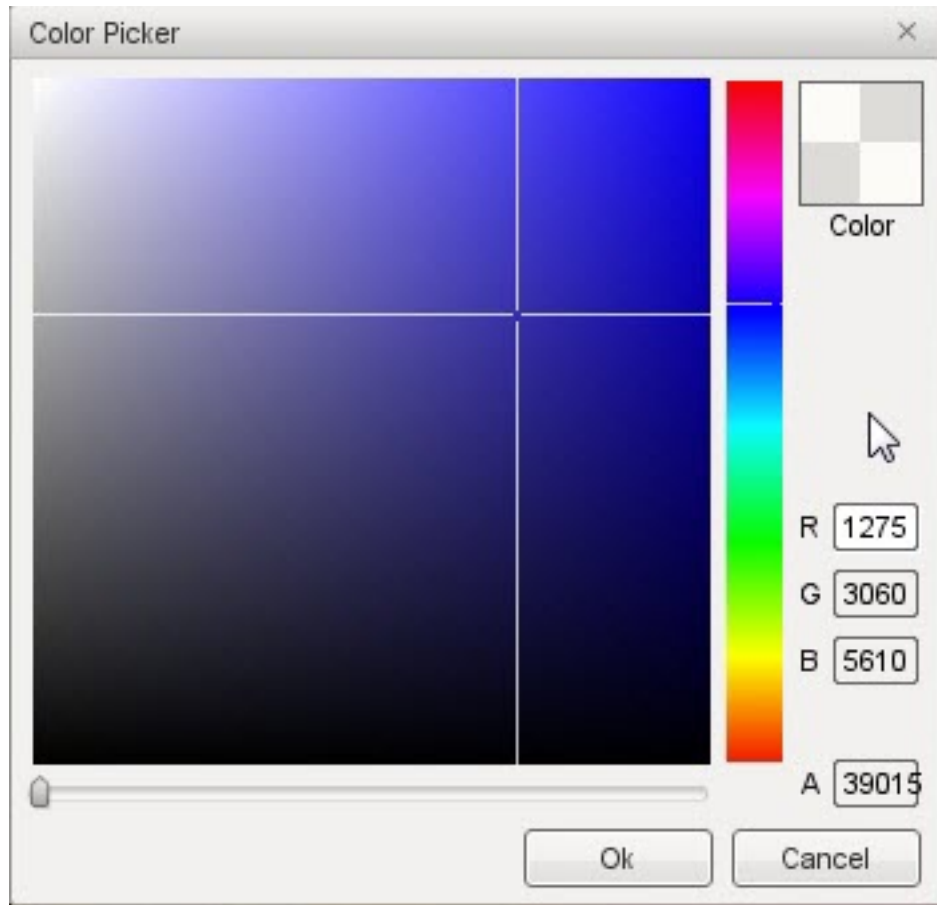
The Color Picker dialog will appear immediately. With this dialog, we can adjust the color's hue, intensity, and alpha. To illustrate a dramatic change, adjust the picker's location to get a strong red value.



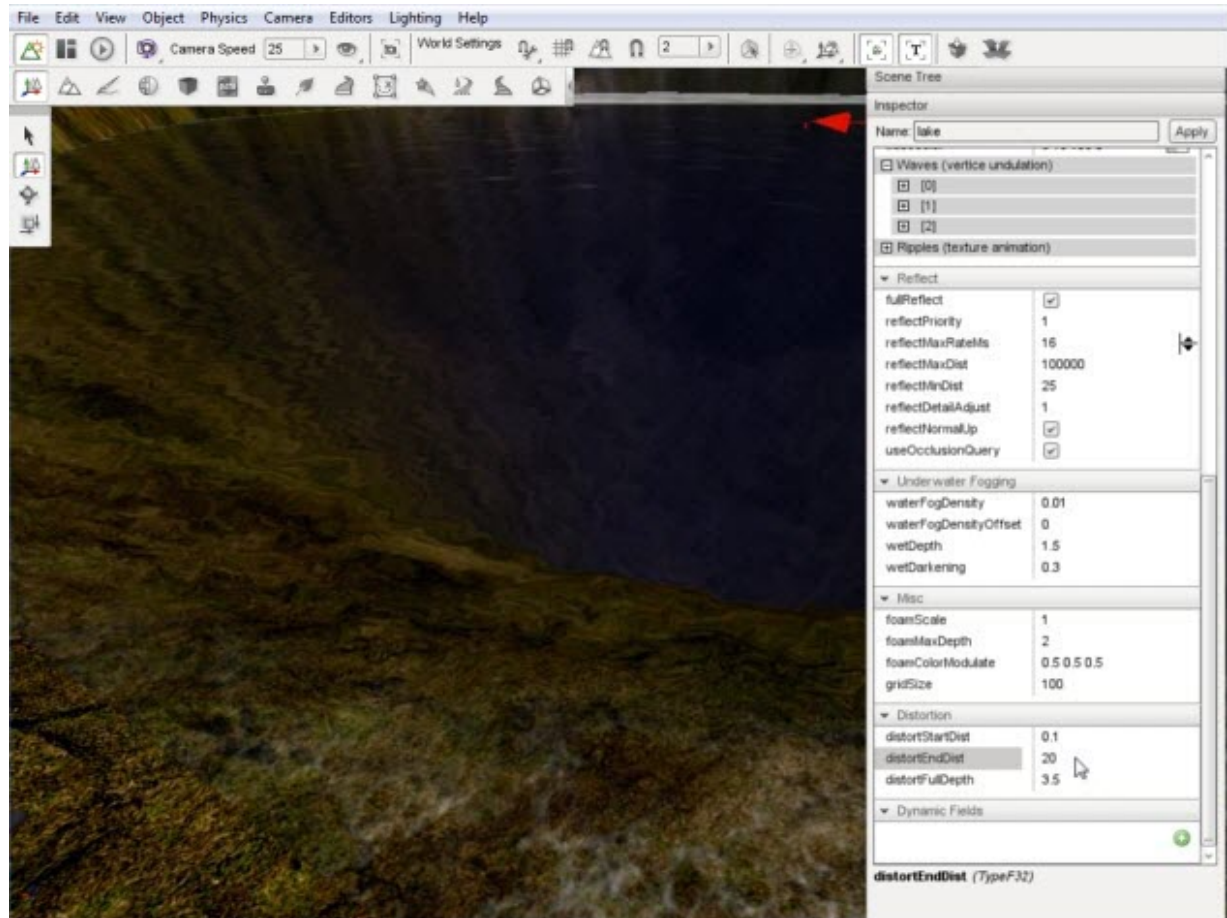
After you click OK, our lake will take on a morbid appearance.



If we were making a horror game this would fit in nicely, but we are making a normal lake. Open the Color Picker again and aim for a softer blue hue.



We have progressed from a murky pond to a clear, blue lake. Even with these small adjustments, we already have a drastic change in appearance from the default WaterBlock.

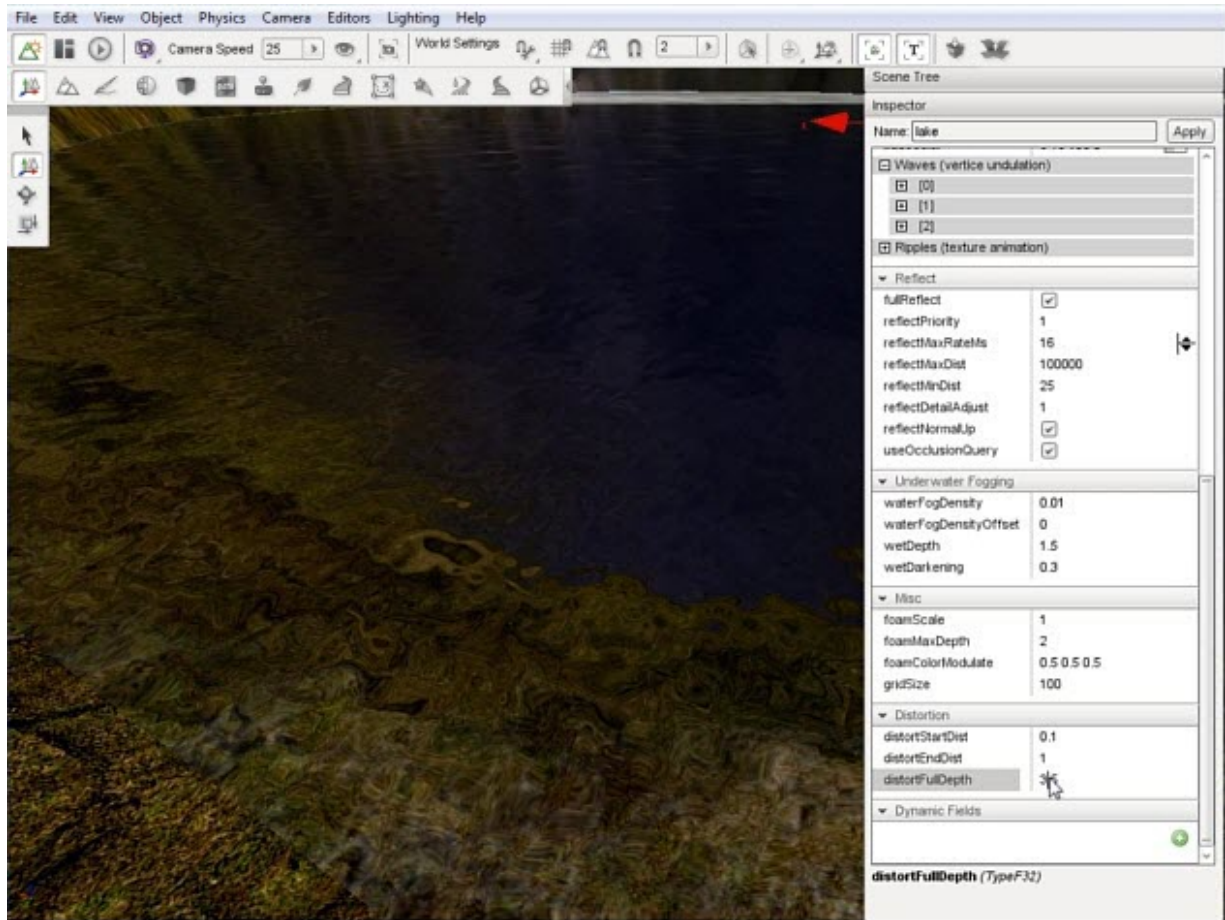


### 14.3.5 Calming the Water

The final changes we are going to make will reduce activity of the WaterBlock to simulate the waves on a lake. The oceans tend to be wavier than a lake because the wave action is created by currents, wind, and the gravity from the moon. On smaller lakes the wave action is caused solely by wind so they tend to be much less wavy than the ocean. Locate the Distortion section of the properties.

We need to intensify the distortion, which we can do by reducing that range. Reduce the `distortEndDist` from 20 to 1. After you make this change, you should immediately notice that objects below the water are more distorted.





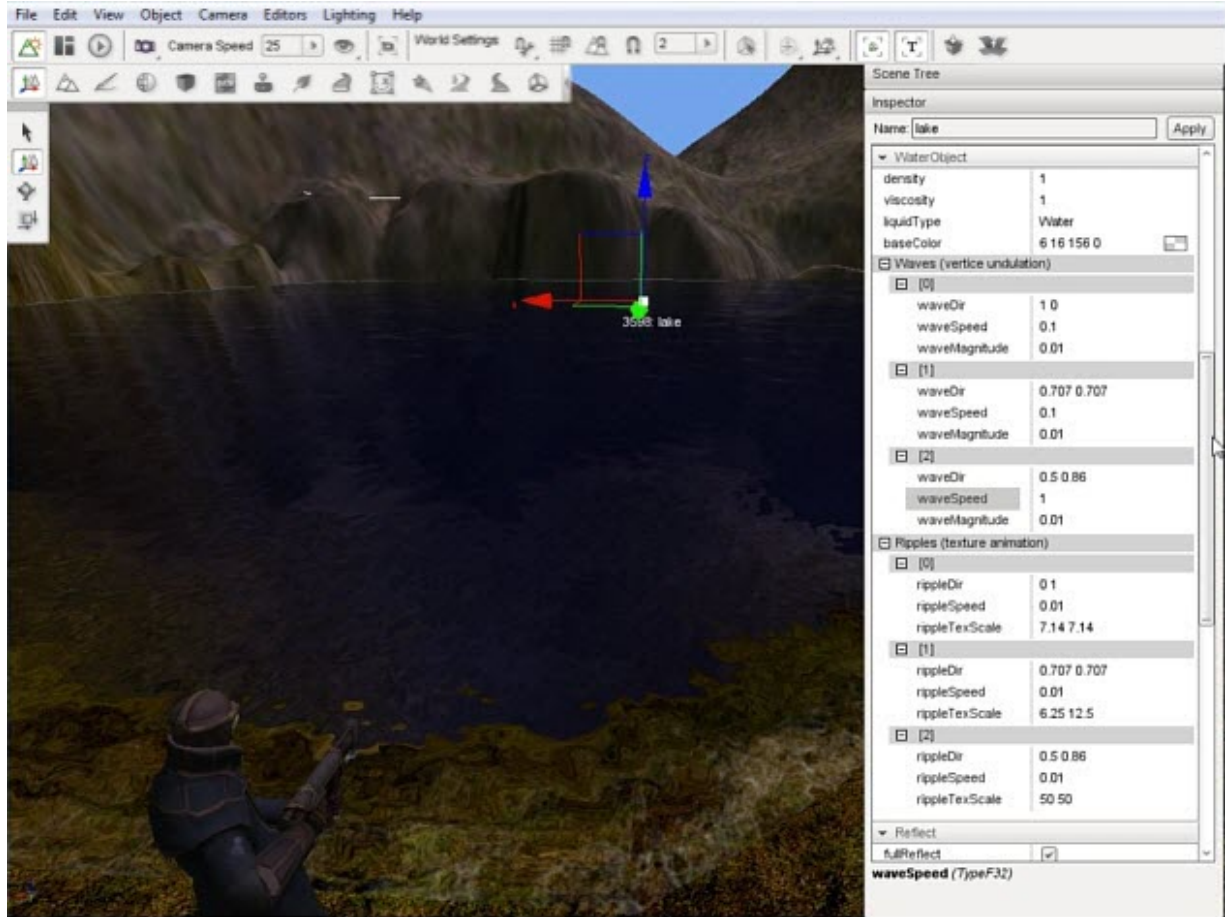
Finally, jump to the WaterObject section of the properties and make the changes listed in the table below to alter the wave action:

Property	New Value
Wave[0]waveDir	1 0
Wave[0]waveSpeed	0.1
Wave[0]waveMagnitude	0.01
Wave[1]waveSpeed	0.01
Wave[1]waveMagnitude	0.01
Ripple[0]rippleSpeed	0.01
Ripple[1]rippleSpeed	0.01
Ripple[2]rippleSpeed	0.01

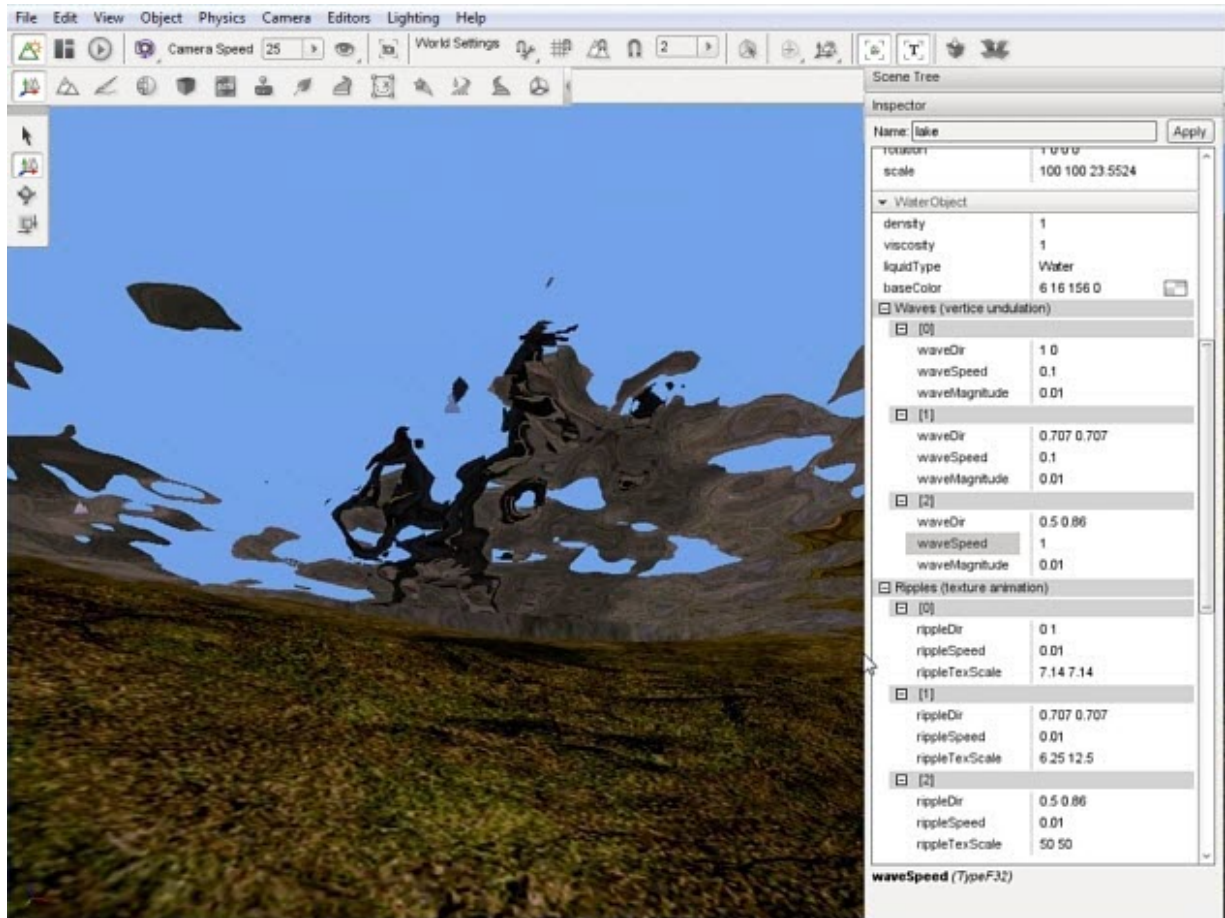
After you have made these changes, your lake should be complete. Your waves will be slower and more dramatic.

### Final Above Water Appearance





Final Below Water Appearance



### 14.3.6 Conclusion

This article has provided a strong starting point for adding lakes to your terrains to create more visually appealing levels.

## 14.4 Adding Foliage

### 14.4.1 Introduction

In this tutorial, we are going to create multiple types of foliage using a single GroundCover object. By the end, you should know how to add a GroundCover object, create a 2D material for it, assign terrain layers, and mix different variations of 3D shapes and 2D images.

### 14.4.2 Setup

This article will use a new project created from the Full template, which includes sample assets for testing and learning. To save time and focus on the World Editor, we will use the sample assets and learn about asset creation later.

If you have not covered the [Building Terrains Tutorial](#), then please do so before working through this tutorial. It assumes that you have gone through that one before starting here.

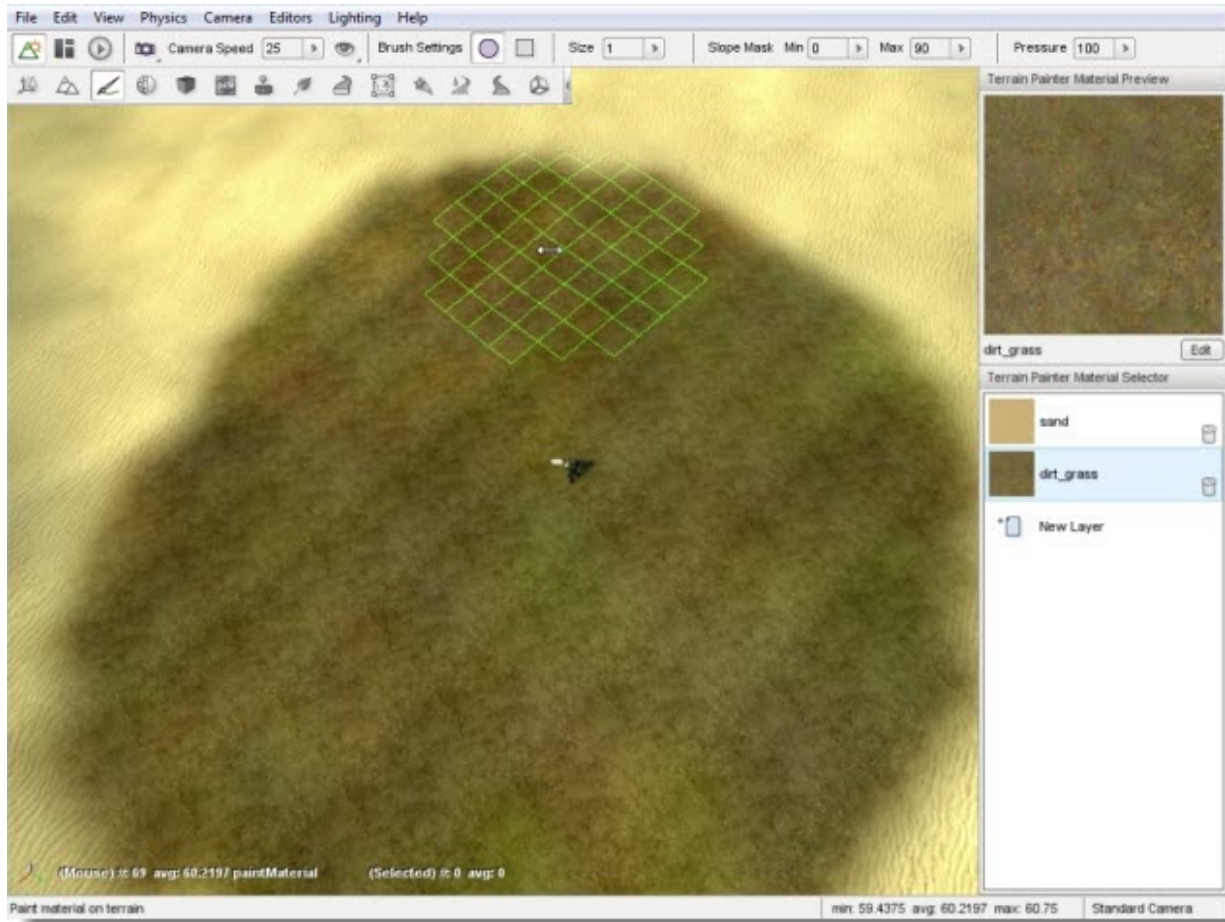
Switch to the Terrain Painter editor by choosing Editors->Terrain Painter from the menu or by hitting F3. On the right side of the screen, find the Terrain Painter Material Selector pane. This lists the current materials loaded and available to paint the terrain.

Since GroundCover is directly tied to Terrain Materials, you will want to make sure you have more than one available. If you only have one material available, click the New Layer button in the Terrain Painter Material Selector pane, and select a new material to load.

In this example, I am using a sand texture and a grass texture:

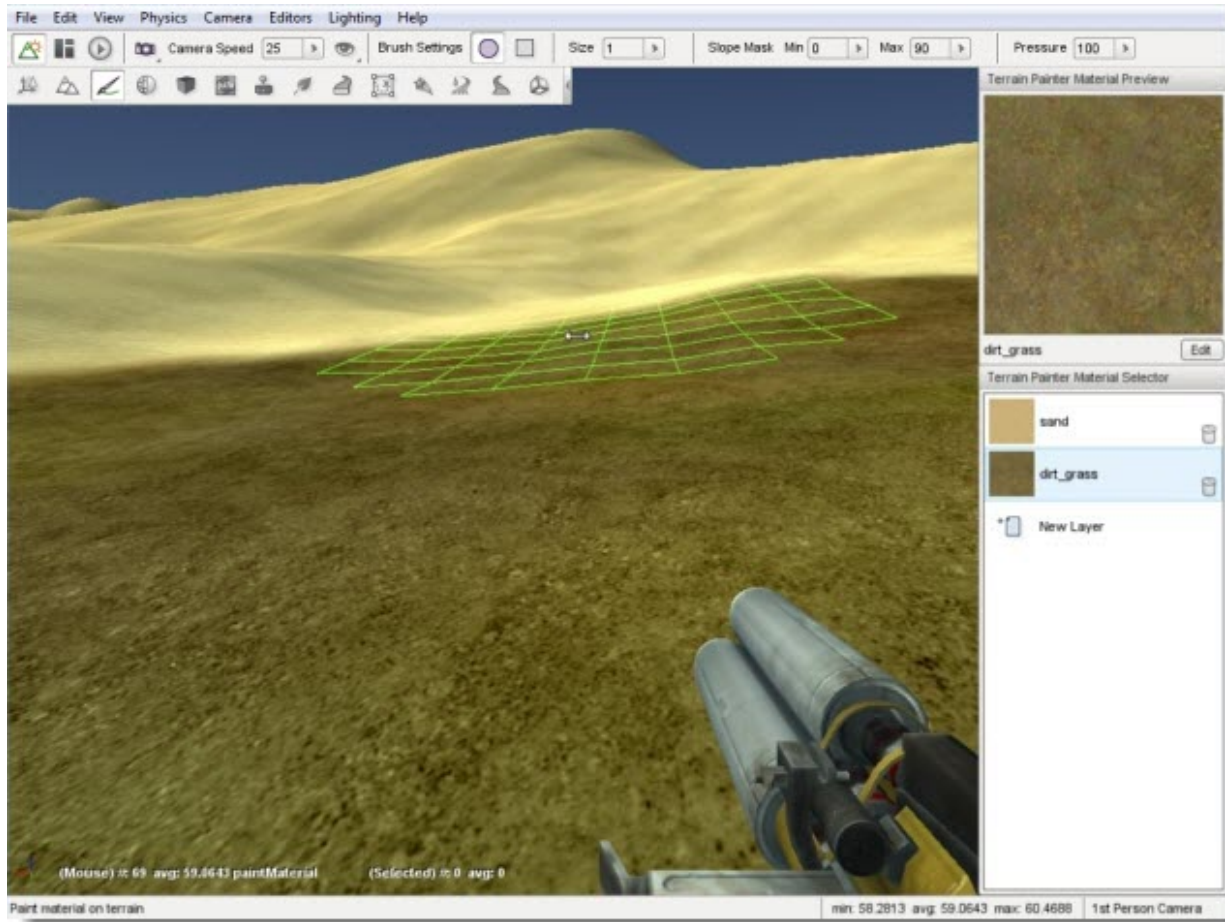


Select one of the materials and paint a small area on the terrain. We will be isolating a GroundCover example to this patch:



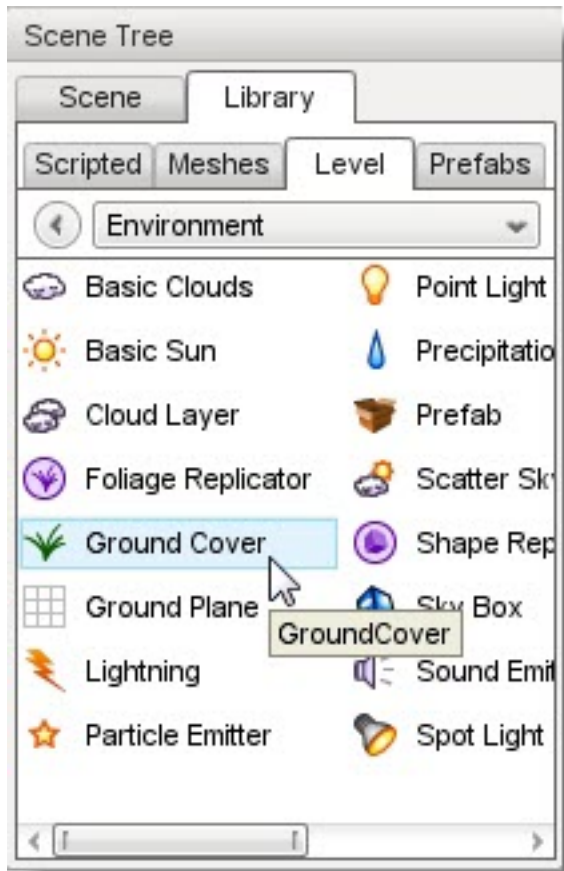
From the player's perspective, you should still be able to distinguish between the terrain textures. This will help demonstrate the GroundCover populating on a specific layer:



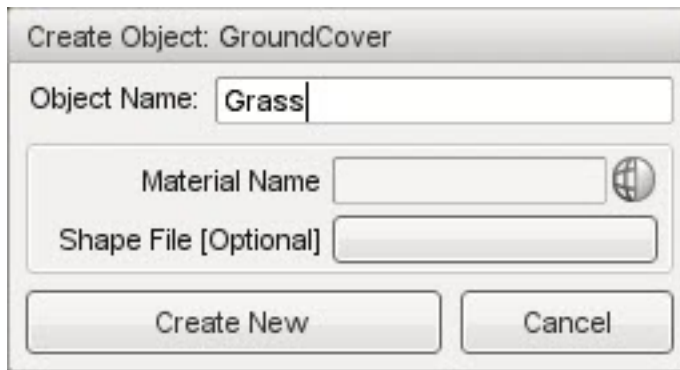


### 14.4.3 Adding GroundCover

To add a GroundCover object: switch to the Object Editor (F1); select the Library tab in the Scene Tree panel; click on the Level tab; double-click the Environment folder; and locate the GroundCover entry:

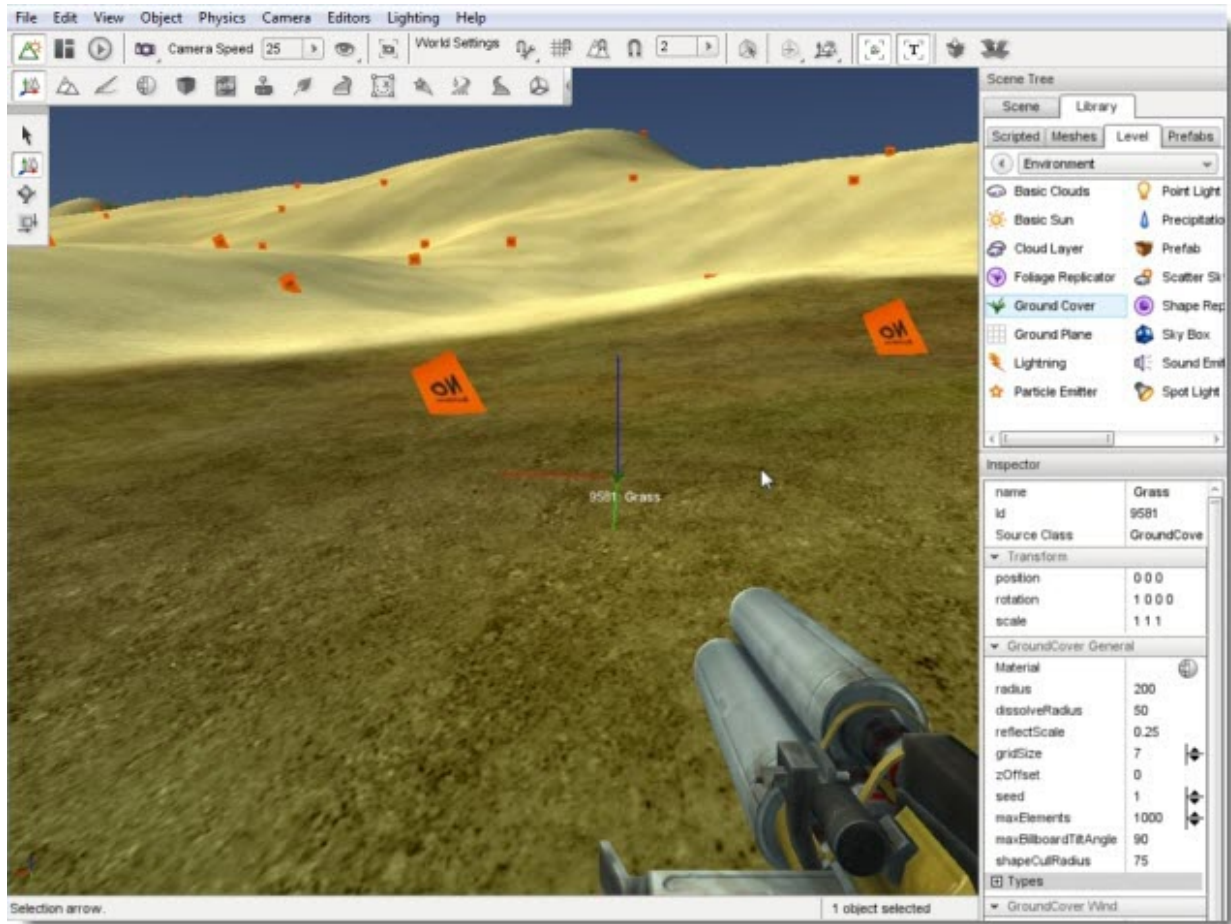


Double-click the GroundCover entry. The Create Object dialog box will appear:



Enter a name for your GroundCover object. We will be creating a GroundCover object from scratch, so leave the rest of the fields blank for now, then click the Create New Object button. A new GroundCover object will be added to your level.





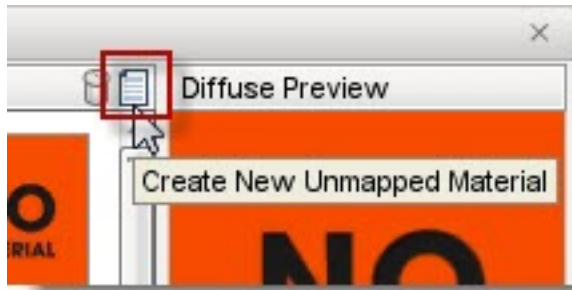
#### 14.4.4 Creating GroundCover Material

Because we did not select a material, the system will render small simple shapes on the terrain with the default orange “No Material” texture. To assign a material, scroll through the GroundCover properties until you get to the **GroundCover General** section. In the Material field, click on the globe icon to open the Material Selector:

##### Material Field



When the Material Selector appears, you have the option to pick an existing material or create a new one. If you are working with the Full template, there will not be a decent grass texture loaded so we are going to create one. Click on the “Create New Unmapped Material” button:

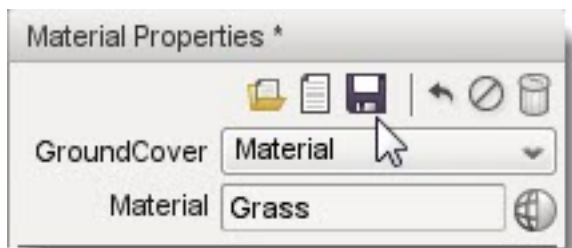


Now click the Select button at the bottom right of the dialog. At this point, the new material has been applied to the GroundCover. The system is still rendering an orange shape, but for a very different reason. Previously, we had no material selected. Now we have a material, but it has not been assigned a texture. Click on the Material Editor icon in the Tool Selector bar to activate the Material Editor tool:

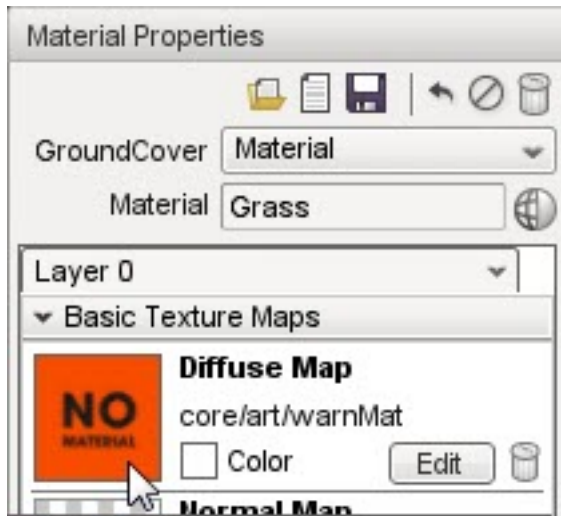
#### Activate Material Editor



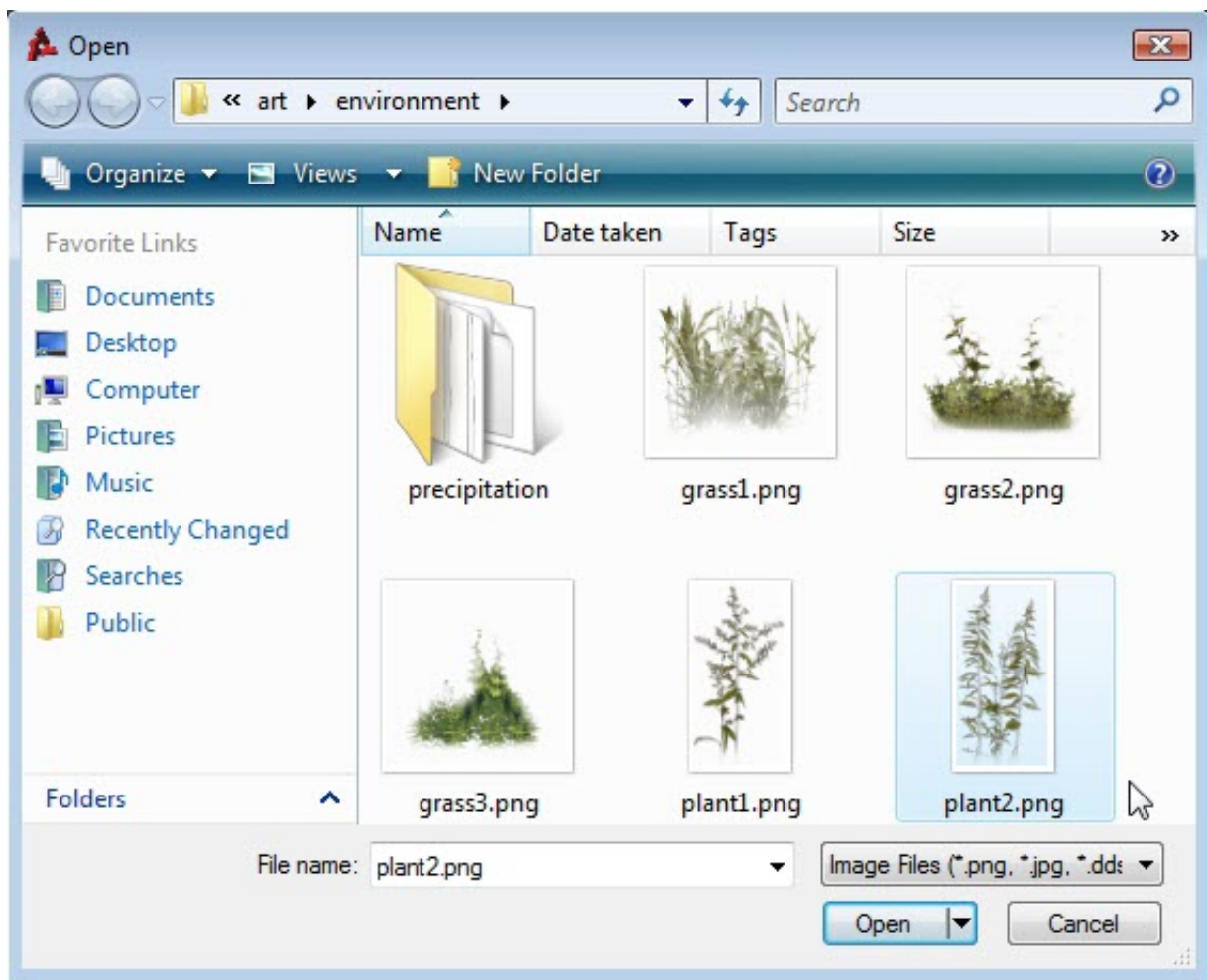
The Material Preview and Material Properties panes should now be visible on the right of the screen. Your Ground-Cover material will already be the active entry. At the top of the Material Properties pane find the Material field and change the value to **Grass**. This will be the name of your new material. Press the Enter key to apply the change, and then click the Floppy Disk icon to save it. **NOTE: You MUST press Enter after typing your material name before clicking the save icon or your new material will not be saved!**



Next, scroll down to the Basic Texture Maps section. Right now the Diffuse Map is assigned to the default “No Material” texture. Click on the preview image or the Edit button:



A file browser should pop up allowing you to select a texture. Navigate to the **game/art/environment** directory. Select the **plant2.png** texture, then click Open.

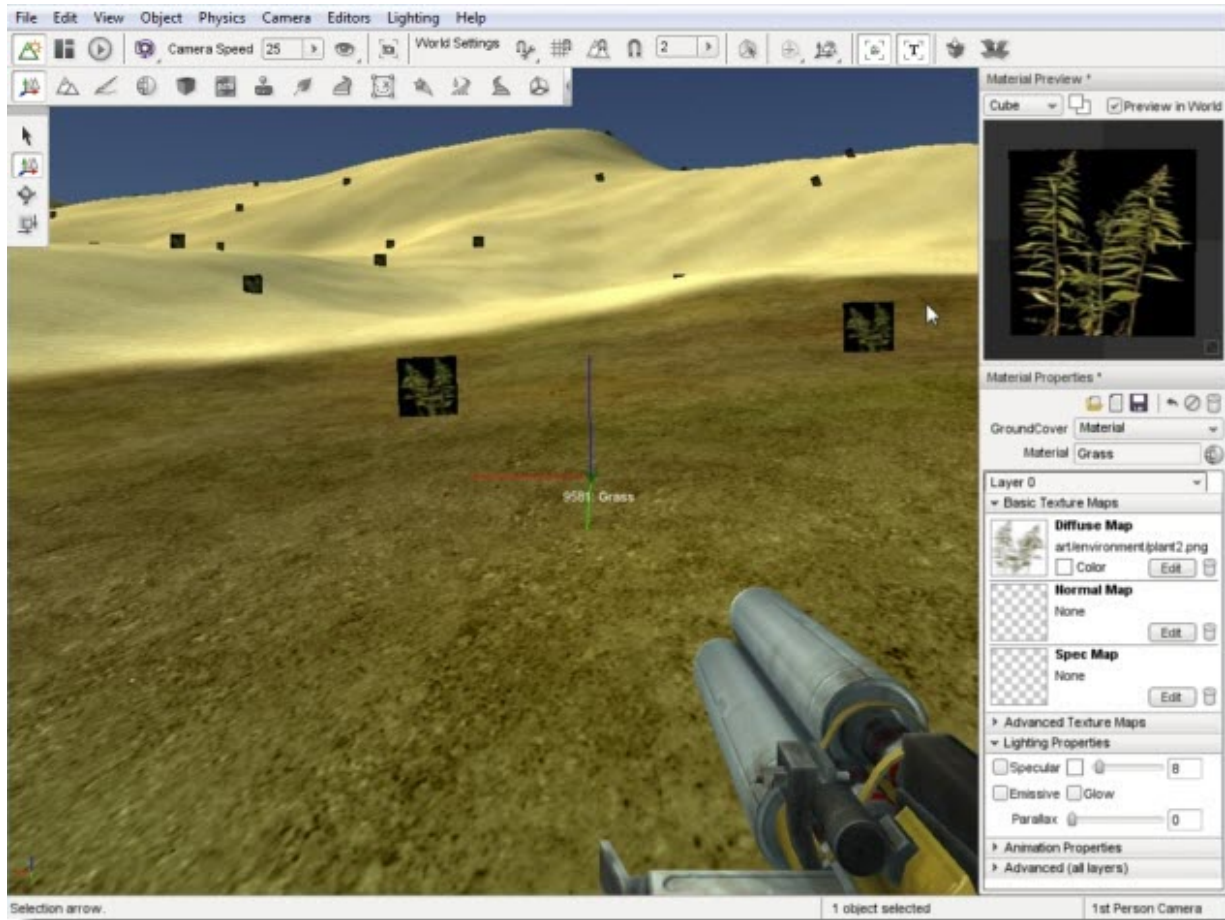


If you are not using the Full Template, but do not have a plant texture, you can use the following image (save to desktop

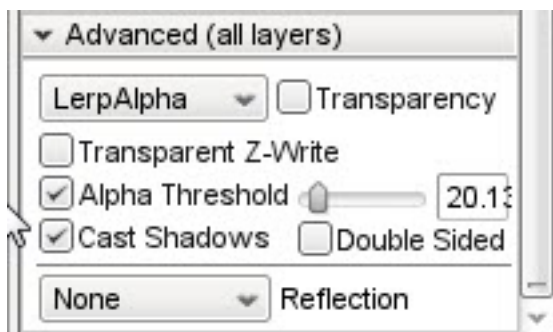
or drag to your folder):



After you load the plant2 texture as the Diffuse Map, your Material Preview will update to show the rendering. The GroundCover in your level will also automatically update. However, you will notice a glaring problem. The material is rendering black where there should be transparency:

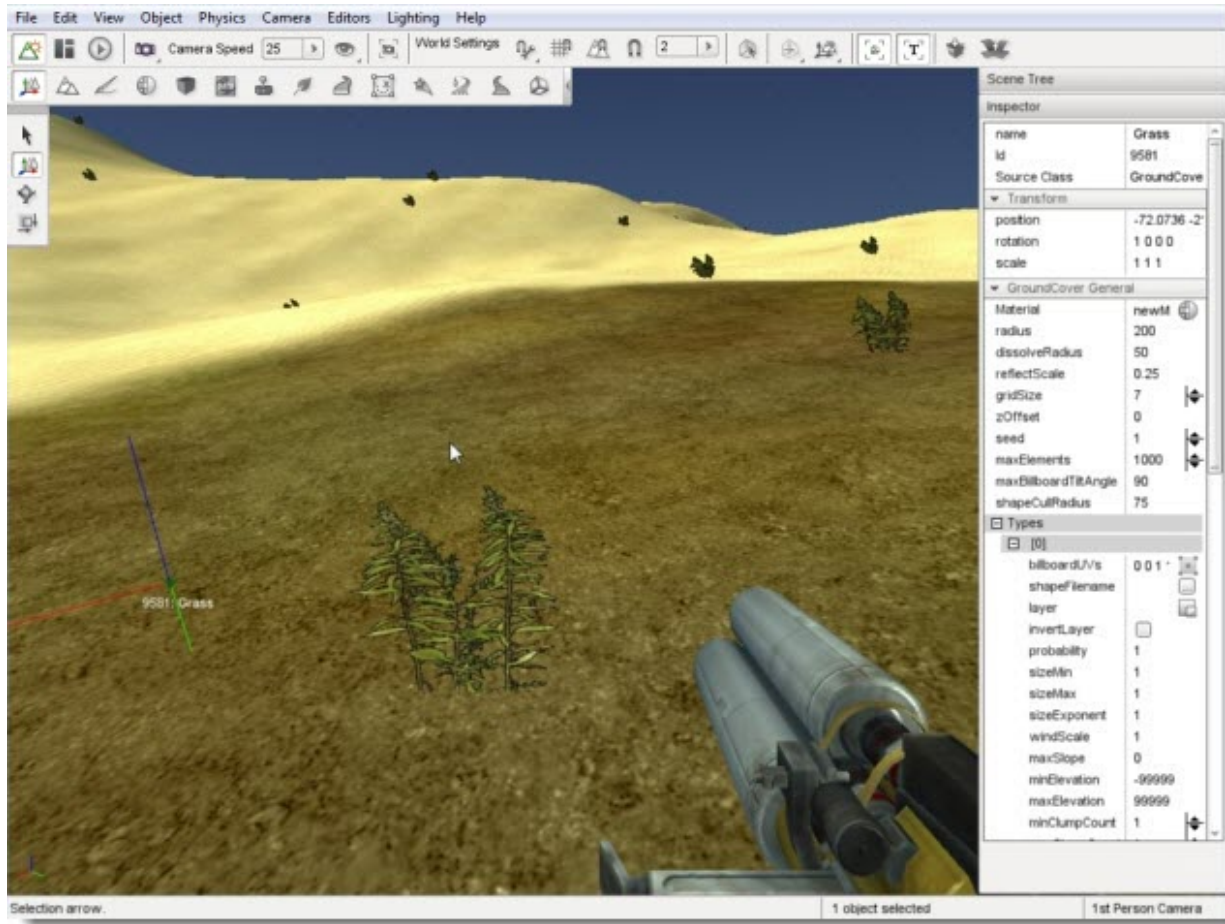


To fix this, scroll down to the Advanced (all layers) section of the Material Editor. Check the Alpha Threshold box, then set the value to something close to 20.13 (or whatever looks best to you):



**SAVE YOUR MATERIAL AGAIN** by clicking the floppy disk icon at the top of the Material Properties pane. Once you are finished with your material, switch back to the Object Editor (F1). Your GroundCover should now be rendering the plant material you have selected. Now is a good time to also save your level.

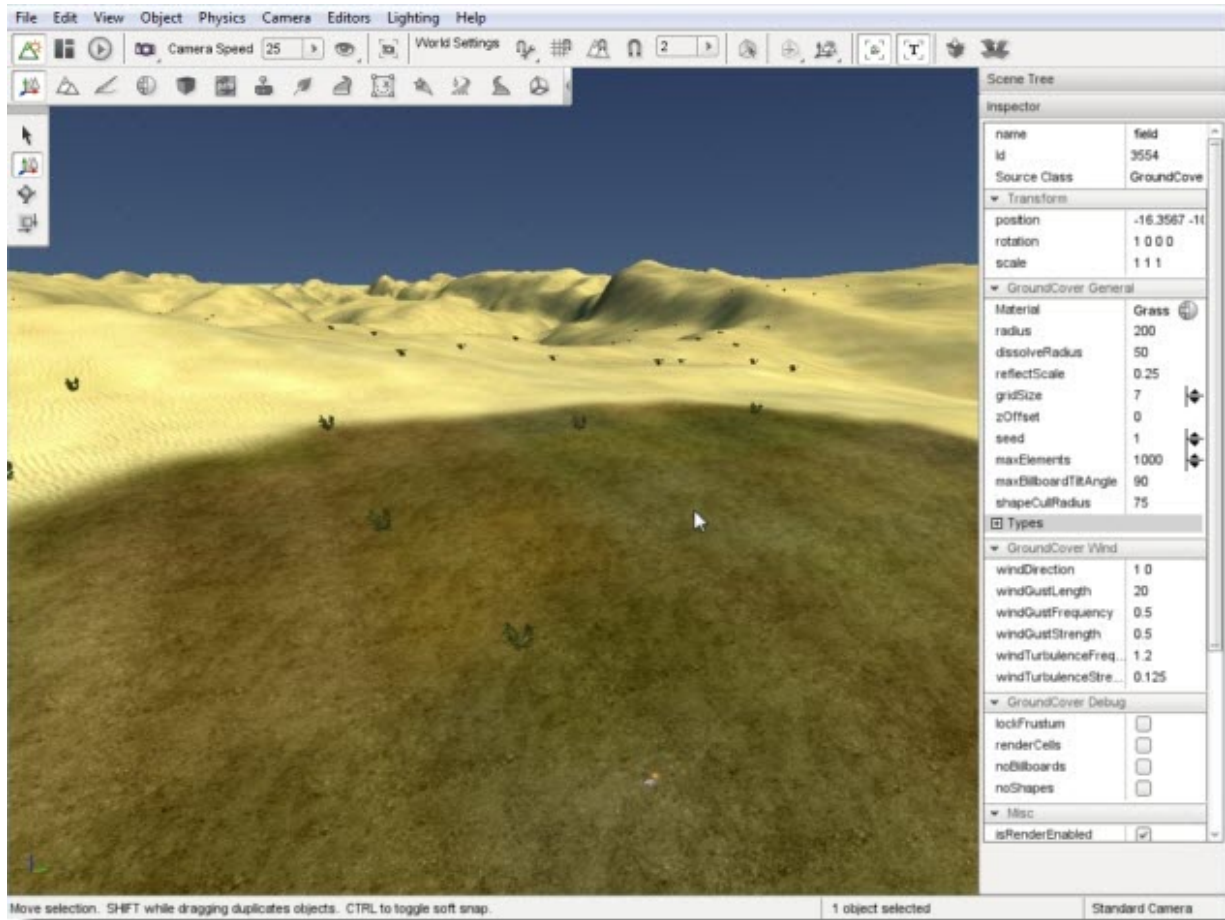




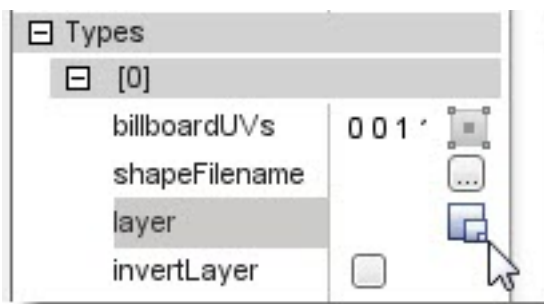
### 14.4.5 Assigning Terrain Material

Currently, the GroundCover is placing the grass material on all of the terrain:



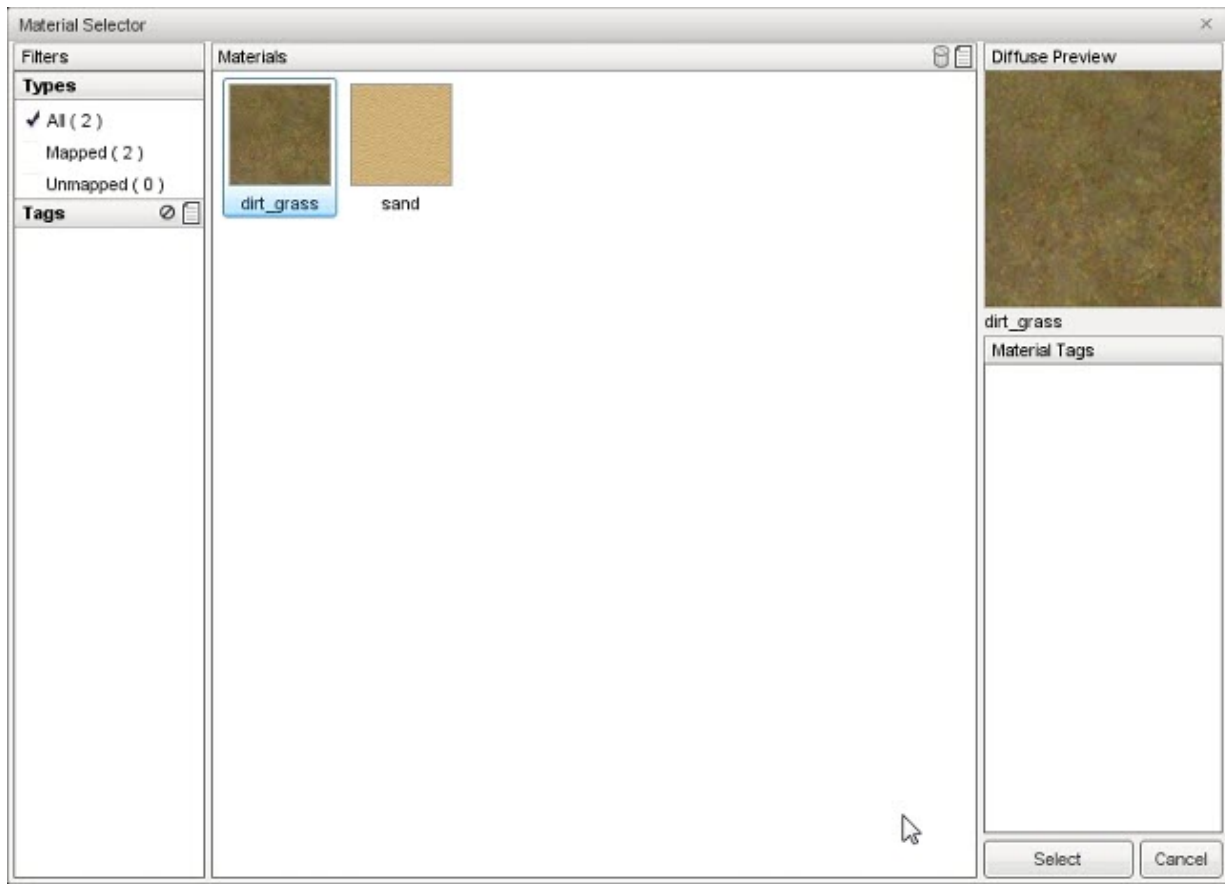


To limit the placement of GroundCover to a specific region, such as the area we painted previously, you must set the terrain layer for the Ground Cover. In the Scene Tree pane, click the Scene tab and select your grass GroundCover object. Scroll down to the GroundCover General set of fields. GroundCover General contains a sub-section of properties, listed under Types. Types is an array where each entry controls a section of the GroundCover. Expand the Types field by clicking on the + icon.

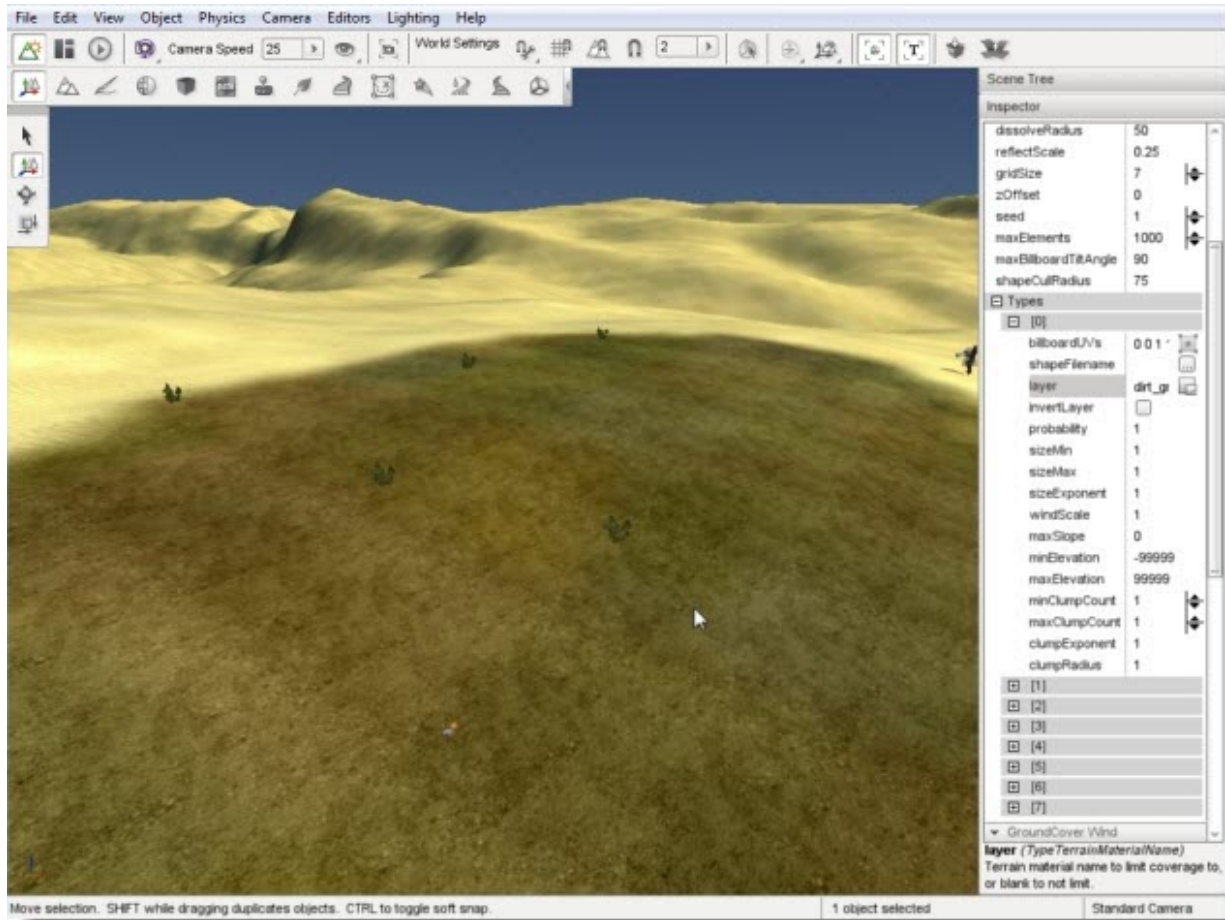


The GroundCover is a single object that is covering the entire terrain. The object itself is comprised of eight sections, Types[0] through Types[7]. Each section can be told what, where, and how to render a material or shape. You can feasibly have the GroundCover object rendering a unique material or shape on eight different terrain layers.

With the above information in mind, it is time to assign the GroundCover to terrain materials. Scroll through the properties until you get to Types[0]. Click on the box icon in the layer field. The Material Selector for terrains should appear. Select the material you used earlier to paint the patch, such as the dirt\_grass shown here:



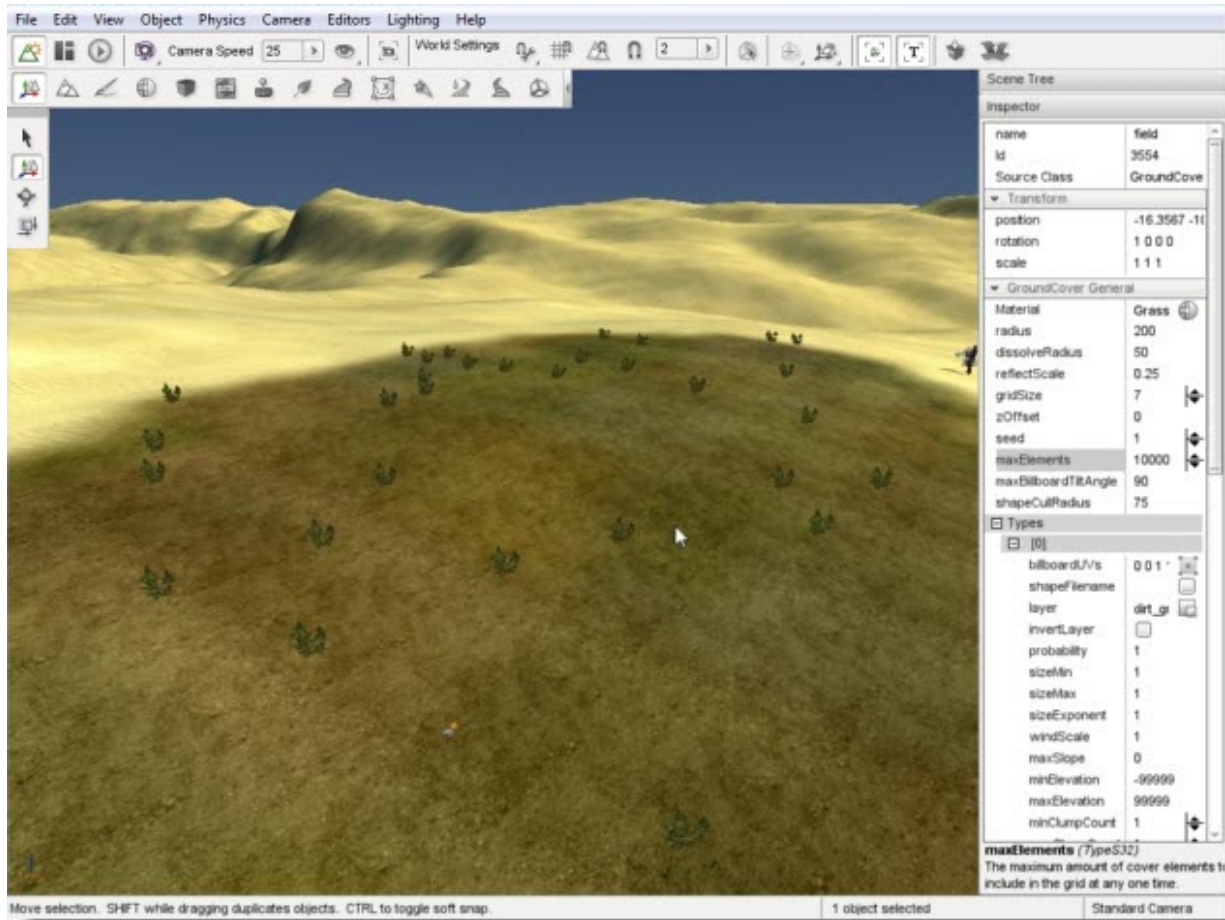
Click the Select button, the GroundCover will stop placing the plant shapes on the entire terrain. It should now only be placing the foliage on the patch that you painted:



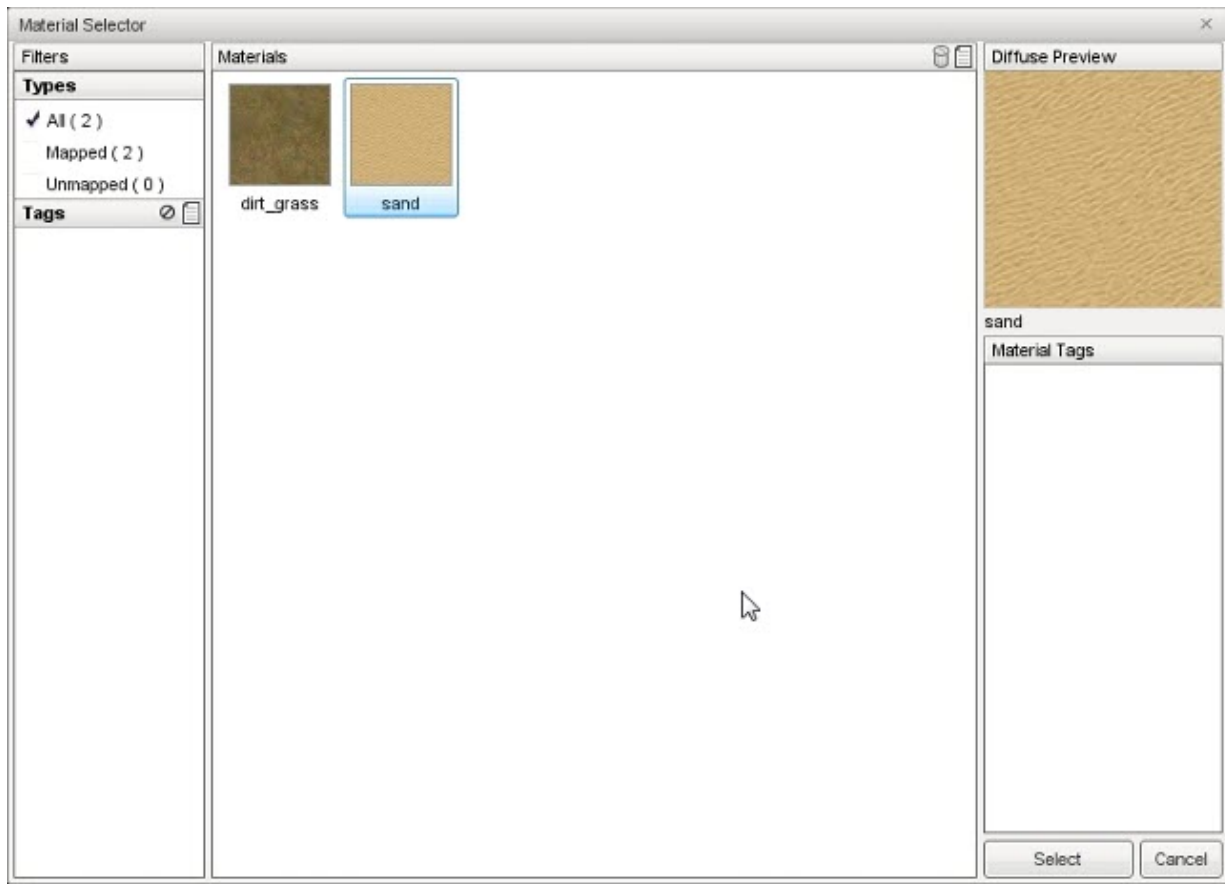
If you are having a difficult time seeing this change, locate the **maxElements** field and increase the value dramatically:



The GroundCover should now be rendering with quite a few more shapes on the isolated terrain material. The higher the `maxElements` value, the more shapes will be rendered:

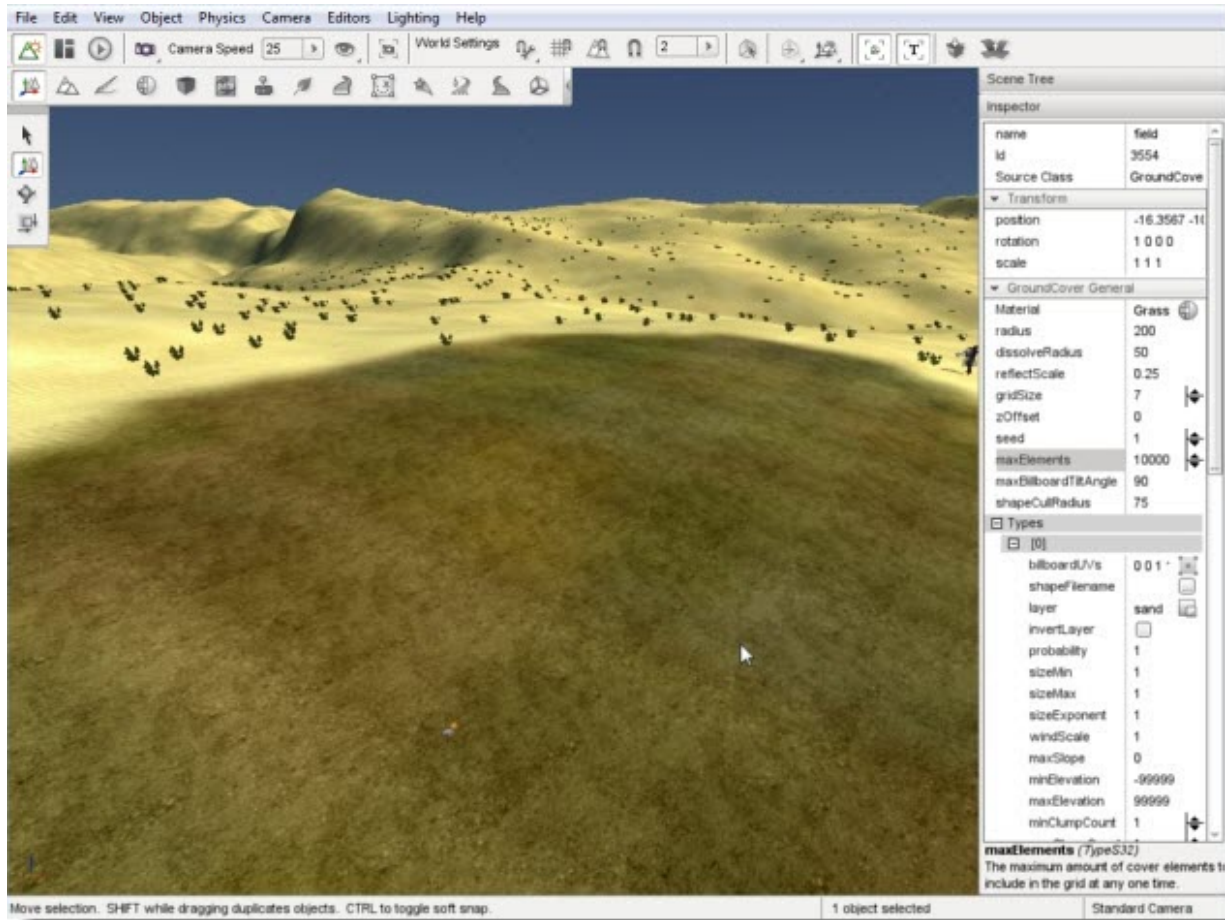


Just for the sake of testing, go back to the **Types[0]->layer** property. Click on the box icon to open the Material Selector, then choose the sand texture (or whatever your main material is):



Click the Select button to apply the material change. The plant billboard should now only be rendering on the main terrain material, avoiding all others including the patch of you painted earlier.





Before proceeding, go ahead and switch the **Types[0]->layer** field back to the isolated terrain material. Next, we will go through the more basic customizations of a GroundCover object.

## 14.4.6 Basic Modifications

### Size Variations

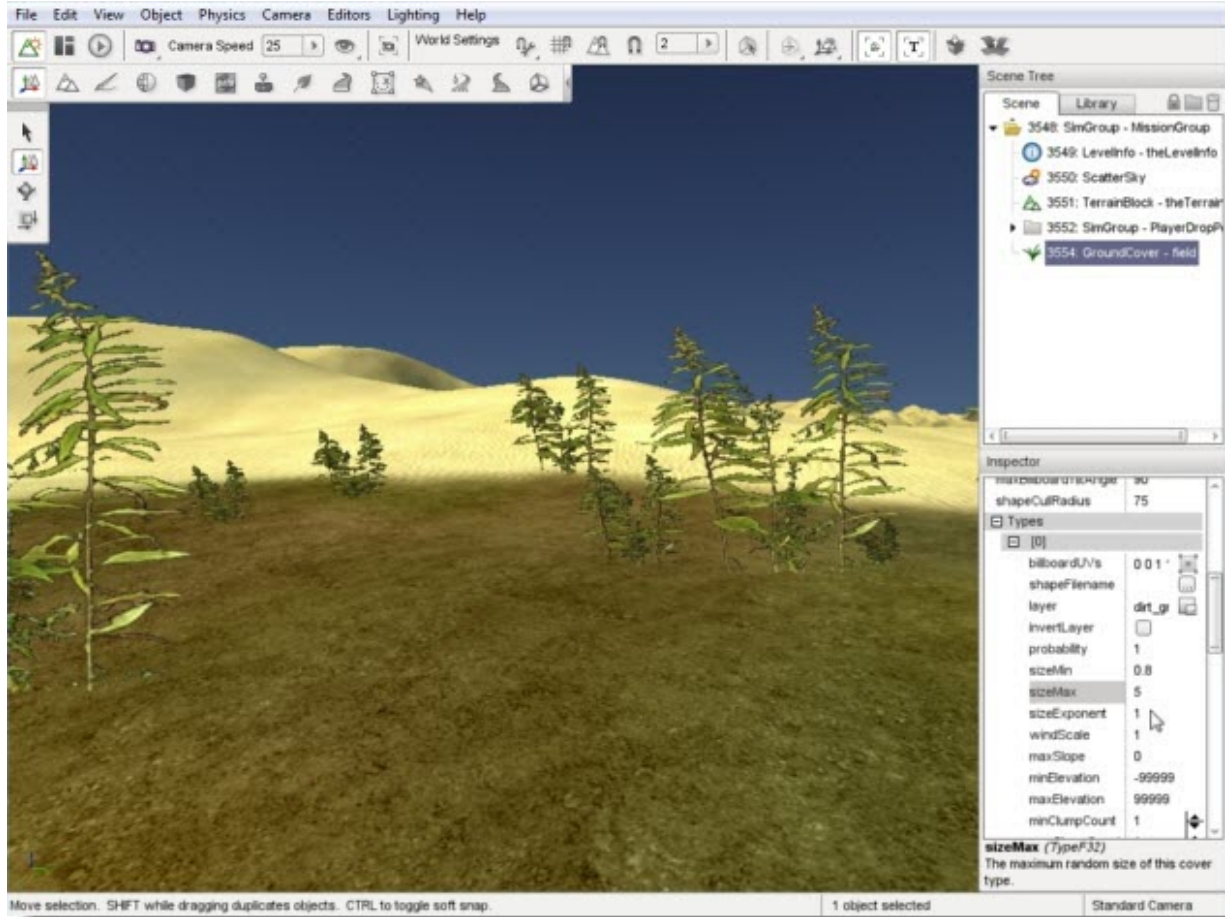
Before we get into the really advanced changes, we will focus on editing a single section of the GroundCover: **Types[0]**. Right now, the GroundCovering should be rendering a single type of material within an isolated section of the terrain. The following changes will affect this section only.

Expand Types[0], then scroll down to the **sizeMin** and **sizeMax** values. The default values should be 1 and 1, which means every shape will be rendered at the same size. Go ahead and change that by setting the sizeMin to 0.8 and sizeMax to 5.



There should now be a dramatic staggering in the size of the shapes. Some will be huge, while others will look smaller than they originally did. This is a great way to increase the realism of your GroundCover:



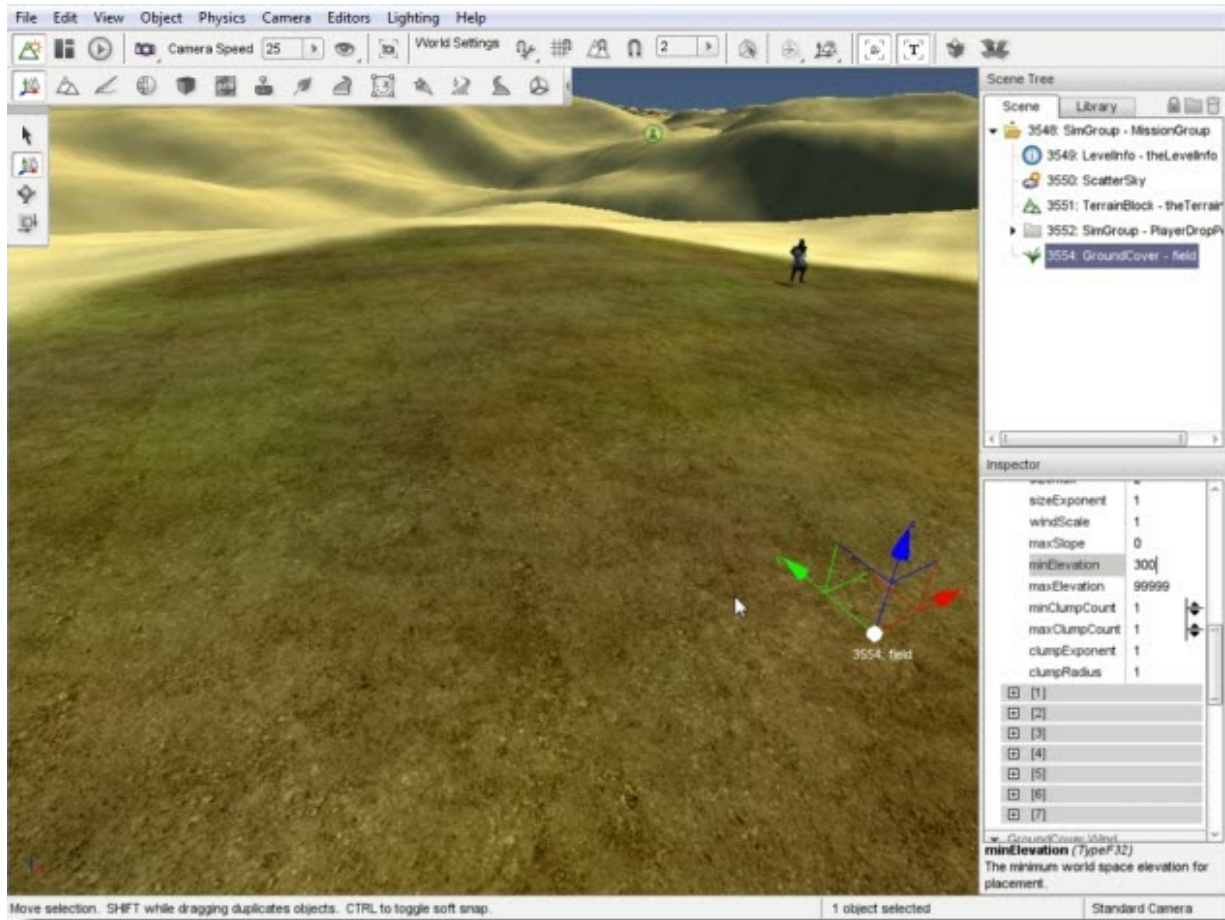


These sizes might be too extreme, so go ahead and balance them a little bit more so we can easily view the rest of our edits:

sizeMin	0.8
sizeMax	2

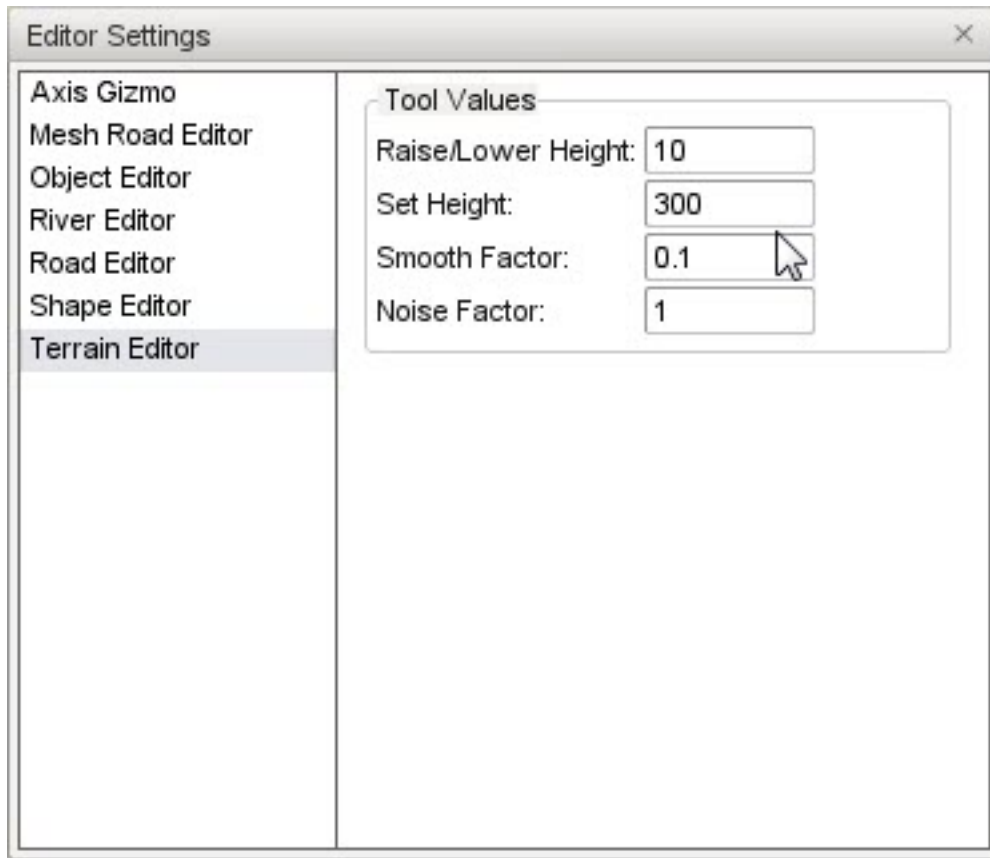
## Elevation Limitations

The **minElevation** and **maxElevation** properties can be used to limit the GroundCover object to only generate shapes and materials within a range of terrain elevations. But, changes to these properties are difficult to see if your level is not set up to handle the parameters. We will walk through an example to illustrate. Go ahead and set the minElevation to 300. After doing so the GroundCover will disappear:

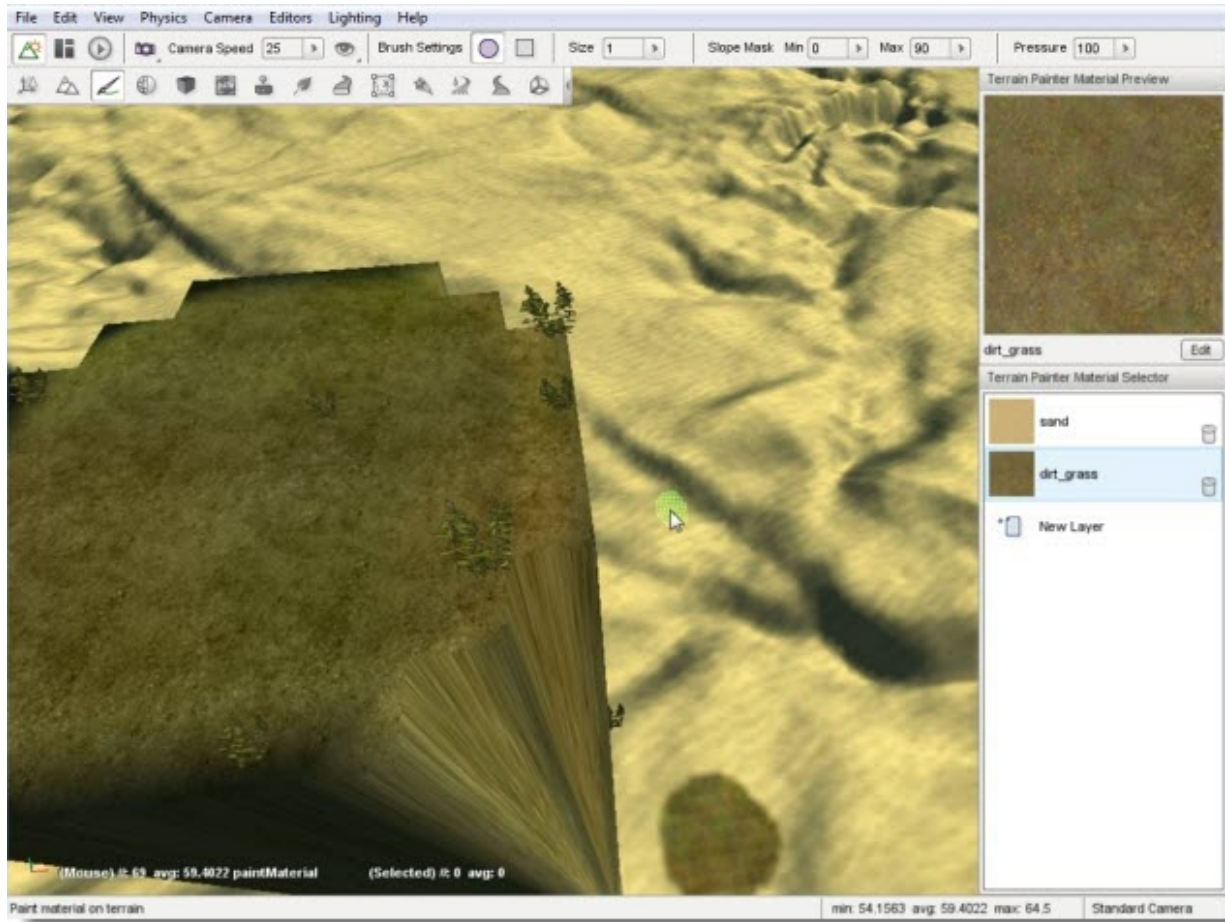


The elevation properties control the height range within which the GroundCover object will place shapes in your level based on the elevation of the terrain at each and every point. When the minSize was increased from -9999 to 300, that decreases the range within which shapes will be generated. The GroundCover object should now only be placing objects on the terrain where the terrain's elevation is 300 meters and higher. The default elevation of TerrainBlock is higher than 300 meters, so it is too low for the GroundCover object to have placed shapes on it.

Perform a quick adjustment to illustrate this behavior.



Open the Terrain Editor (F2), then switch to the Set Height tool .. image:: img/SetHeightIcon.jpg. On the toolbar at the top of the screen, find the Height field and change it to 300. Select a portion of the terrain that should have the GroundCover, then click on it. The terrain should instantly shoot up to 300 . If you move your camera to the top, you should now be able to see your GroundCover:



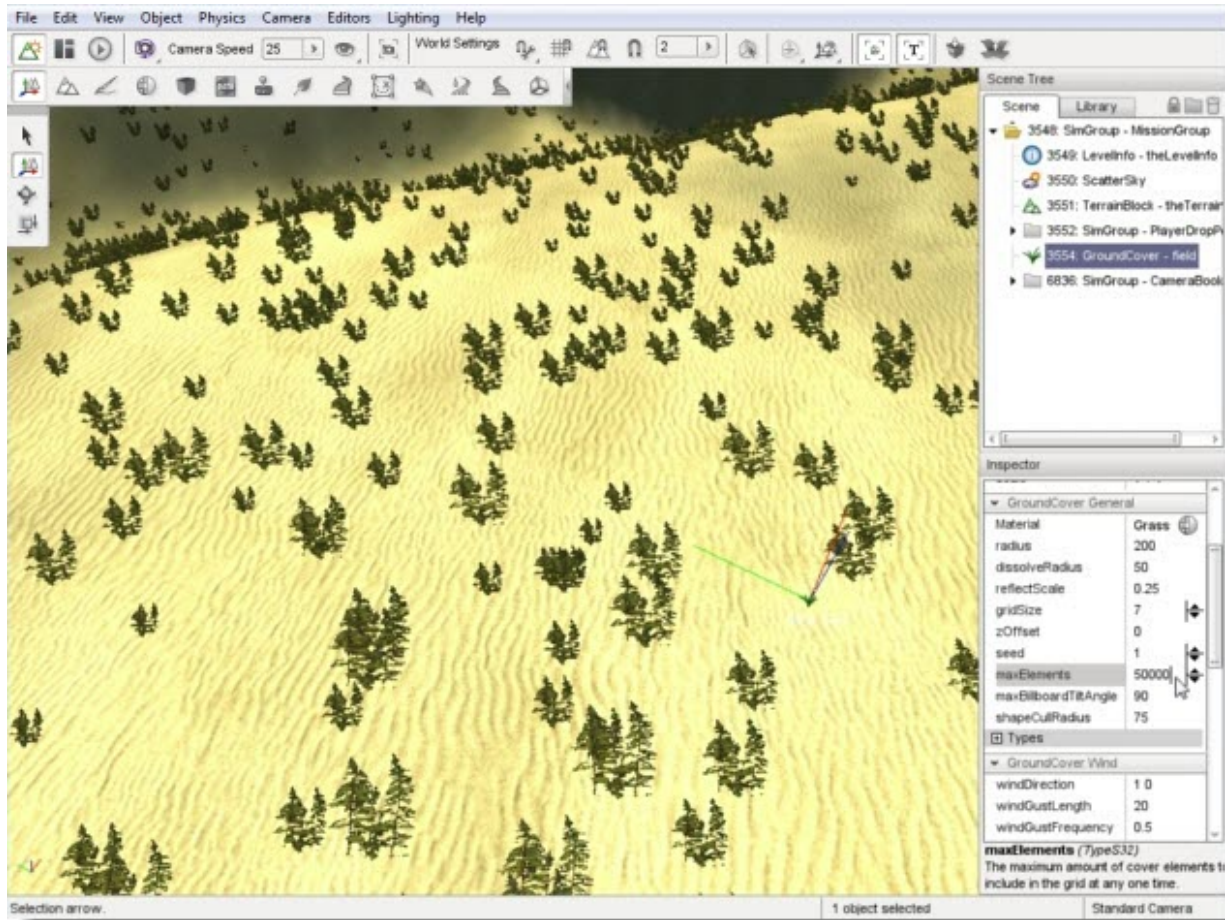
Go ahead and change the **minElevation** field back to -99999, so the grass returns to where it was originally

## Clumping

The clumping ability of GroundCover is another way to add to the realism or diversity of your level. The clumping properties (**minClumpCount**, **maxClumpCount**, and **clumpRadius**) will cause the objects in your current Types[x] entry to group together in designated patterns.

To fully demonstrate this, make sure you can clearly see lots of objects being placed by the GroundCover. For example, I have switched to the sand terrain material (since I am using a dark shape) and increased the count to a huge number:

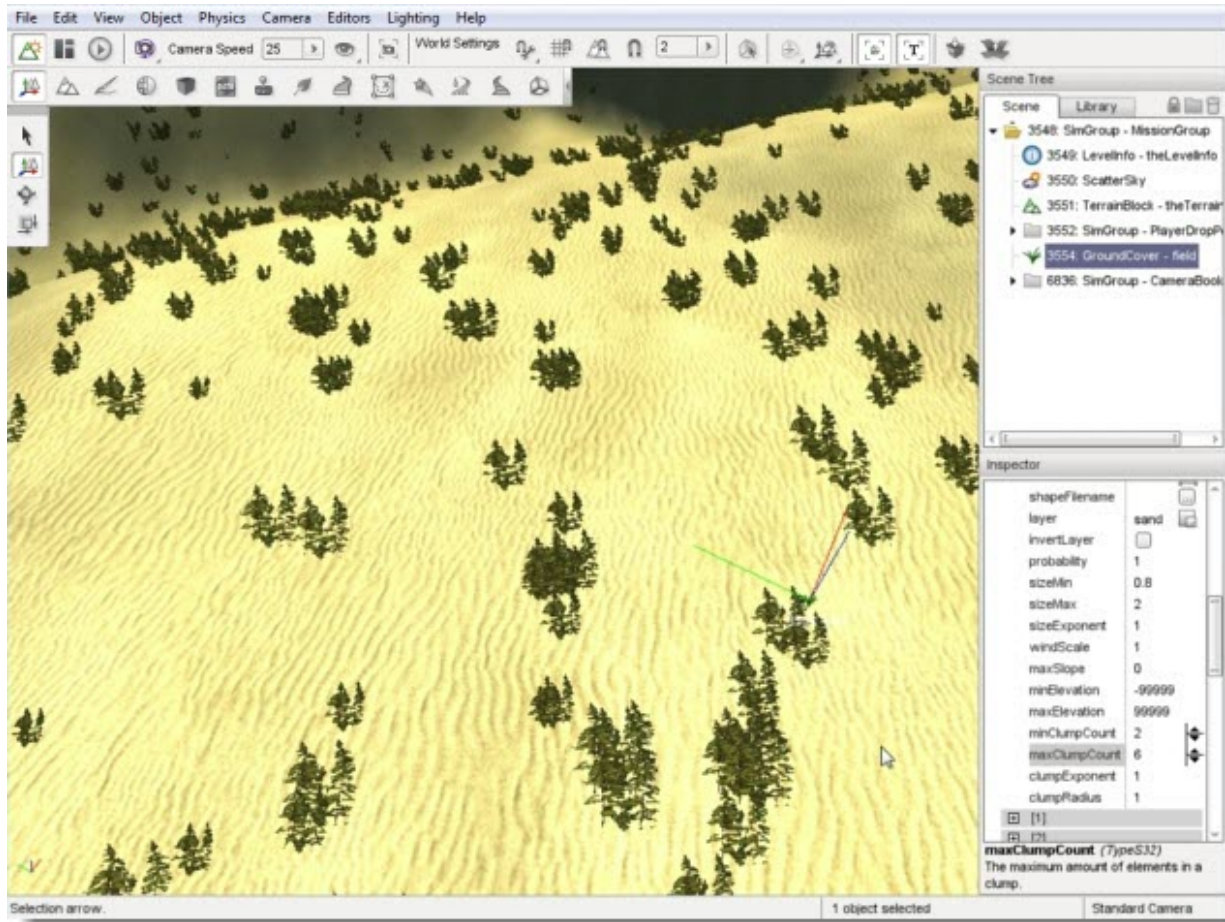




The default values for the clump properties are all 1. This means the system will place a single shape within a 1 meter spacing. Set your GroundCover properties to the following:

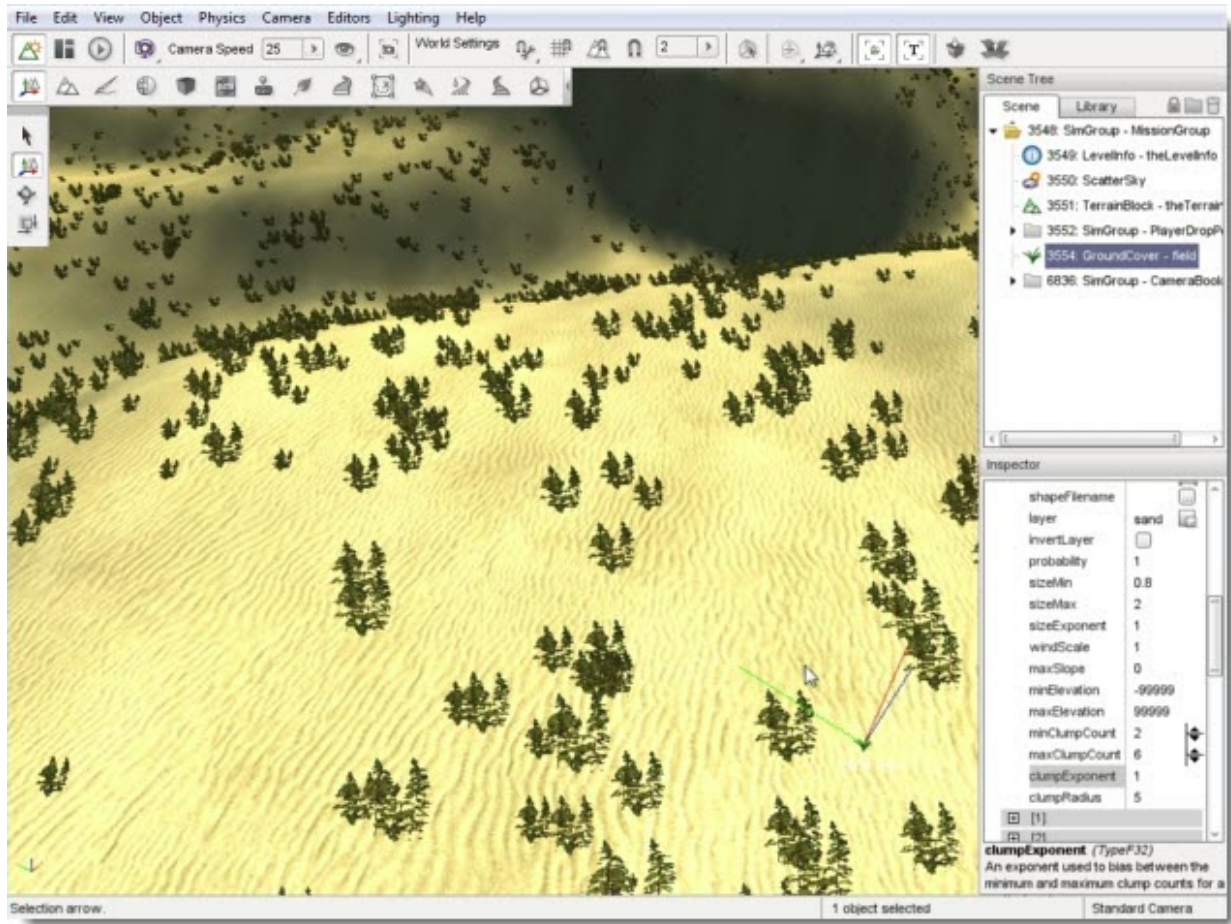
minClumpCount	2	▲▼
maxClumpCount	6	▲▼
clumpExponent	1	
clumpRadius	1	

The system will now procedurally place between two and six objects within a one meter space, creating clusters of shapes around the terrain:



If there is too much clustering or your objects are unrealistically overlapping, you may need to increase the clumpRadius. Go ahead and set the clumpRadius to 5. There will still be clustering, but now the objects should space out more evenly:





These were a few of the basic modifications you can make to a GroundCover object. There are a few additional properties which will be covered in later sections. For now, take some time to experiment with the values mentioned above. When you are ready proceed to the next section.

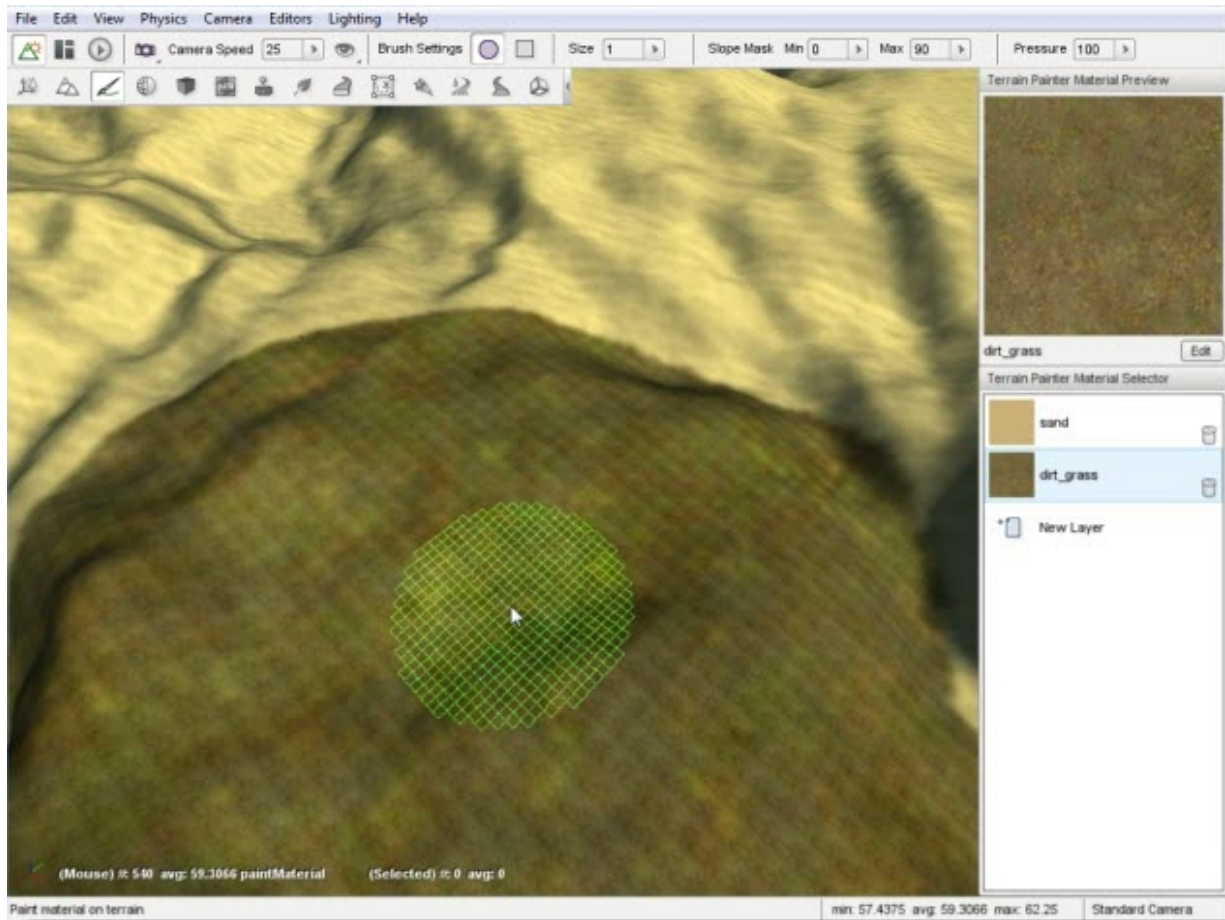
### 14.4.7 3D Shapes

Up till now, the shapes that the GroundCover has been placing have been rendered using billboards. If you are unfamiliar with this term, a billboard is a way of faking the appearance of a 3D object by using a flat image that is rotated as the camera moves so that it is always facing the camera. The material we have been using is a simple 2D image with transparency, representing a plant.

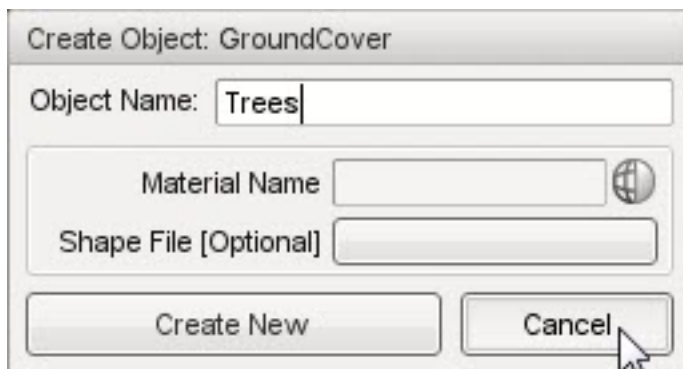
By just displaying this simple image and always rotating it to face the camera we avoid having to create an actual model of a plant with a trunk, limbs, and hundreds of leaves. Displaying and rotating a simple image in this manner uses much less processing power than rendering a full 3D model and all its details. Multiply that performance savings times the number of shapes that are generated by a GroundCover object and the result is more power for other aspects of your game, while still producing an adequate depiction of ground foliage.

However, if you wish to automate the placement of real 3D objects in your level, you can still do so using the GroundCover object. In this next example, we will replicate a 3D model of a tree rather than a billboard image of a tree. Keep in mind that creating a large forest using this method is not the best approach. It is merely an example using a 3D shape which you should have available to you since it is included in the Full template. Creating such a forest for use in a real level is best left up to the [Forest Editor](#), since it is a more powerful tool designed to do just that. If you are not concerned with precision and collision, then the GroundCover object might work just fine.

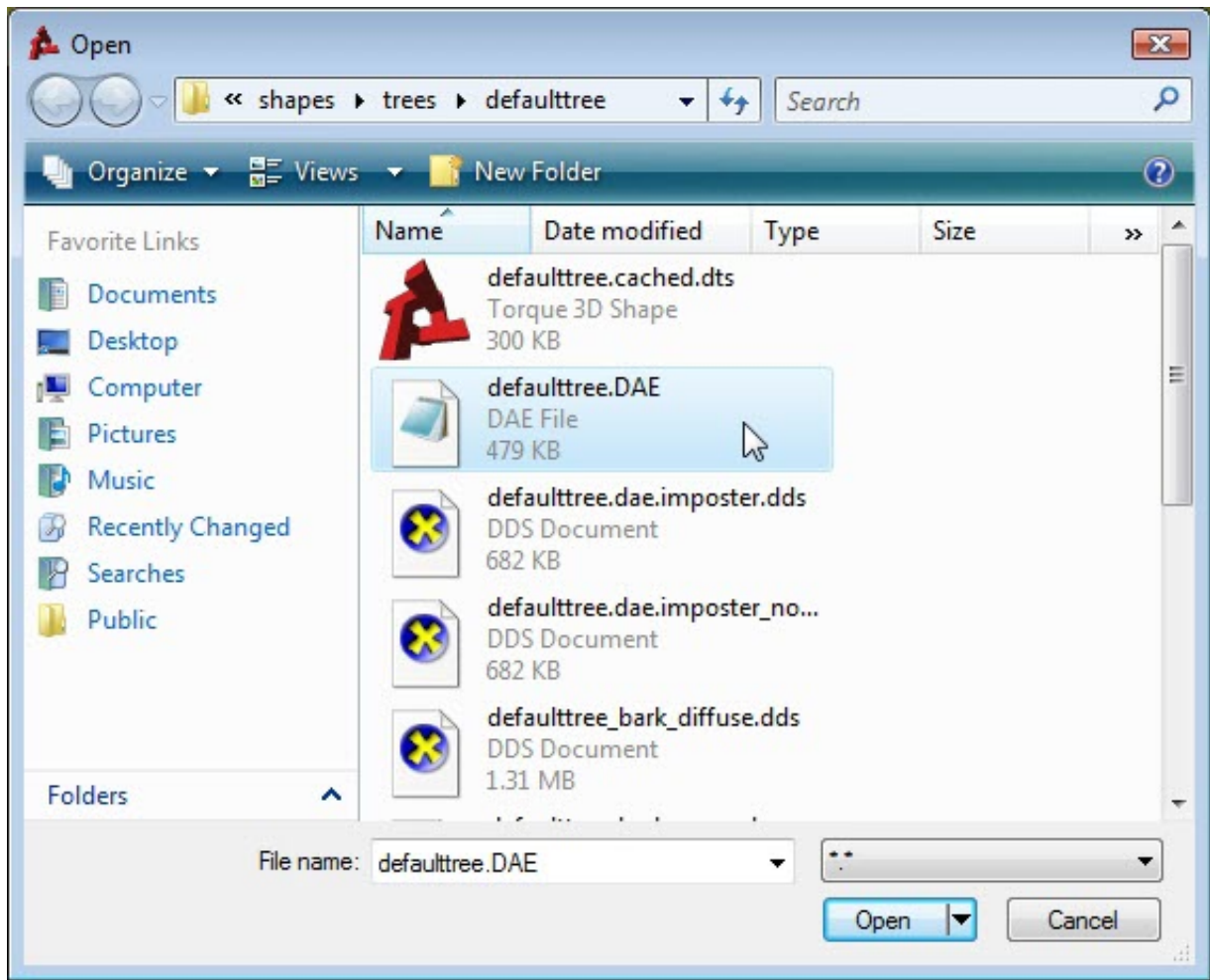
Start by deleting any existing GroundCover objects in your scene. Using the same procedure as before, switch to the Terrain Painter tool and paint a much larger portion of the terrain with your secondary material:



Now add some trees using a Ground Cover object. Switch to the Object Editor tool; select the Library tab; select the Level sub-tab; double-click the environment folder; locate the GroundCover object and double-click it. The Create Object dialog box will appear:

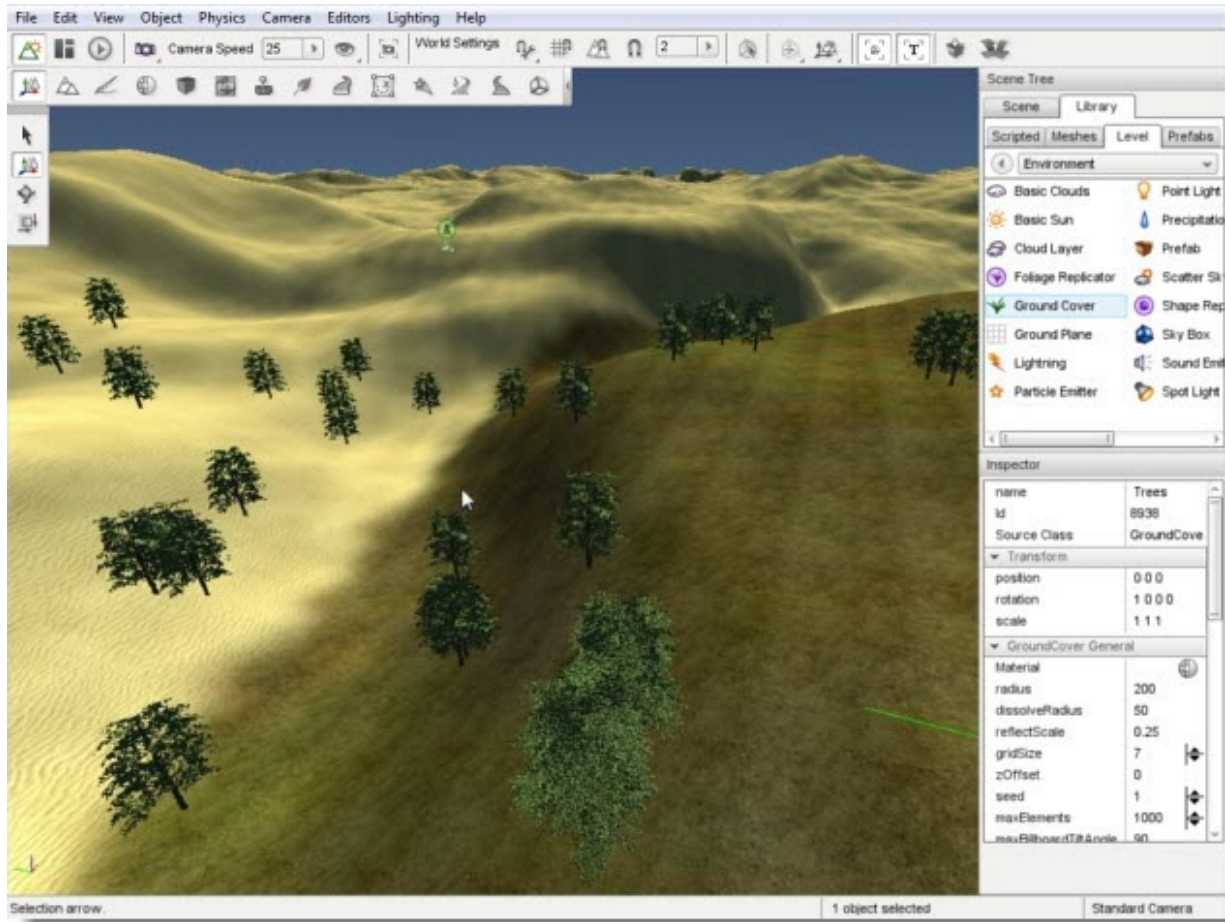


Enter a name for your ground Cover object, then click on the Shape File [Optional] field which will open a file browser. Navigate to the **game/art/shapes/trees/defaulttree** directory within your project. Select the **defaulttree.DAE** file, then click the Open button.

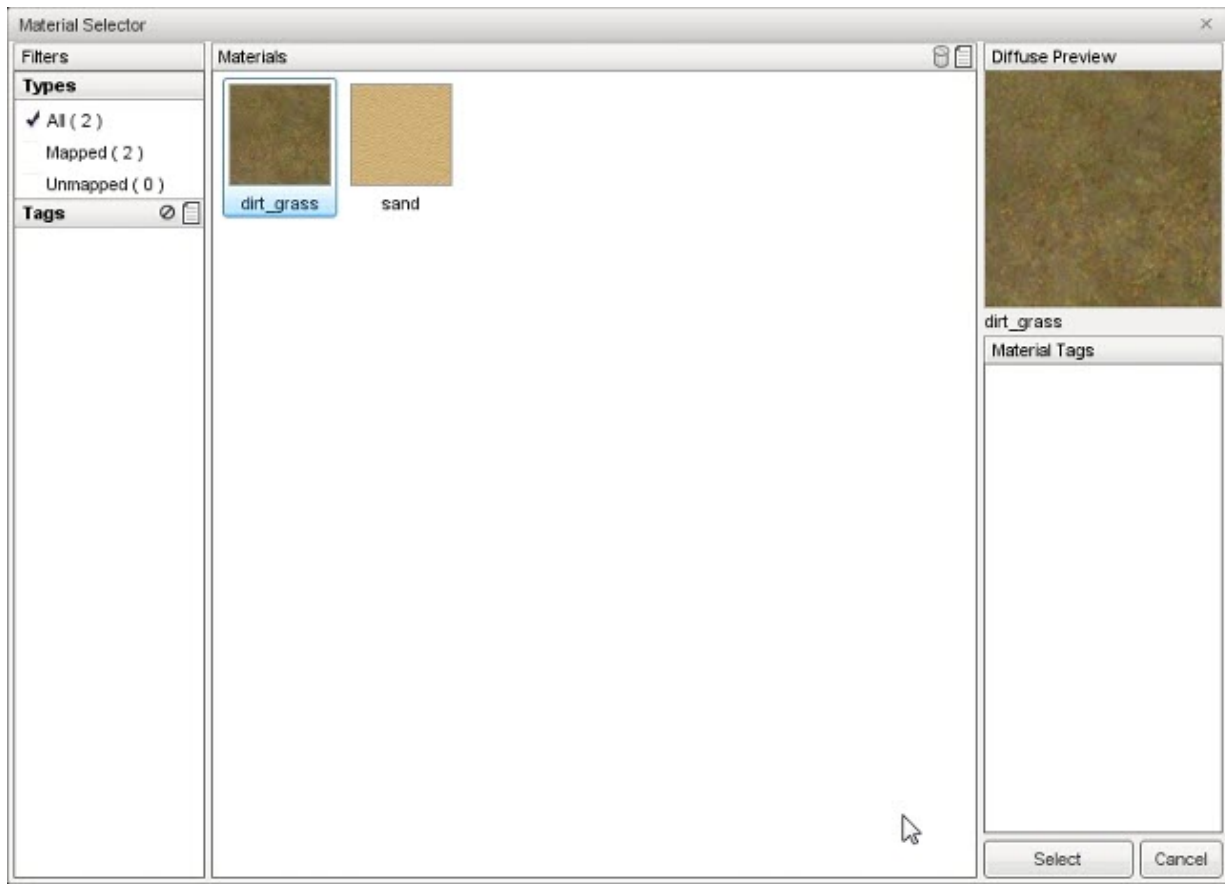


Your selection will be placed in the Shape File [Optional] field and will be the shape that the Ground Cover object will replicate. Click the Create New button to add your new GroundCover object to the scene. Unlike material billboards, the GroundCover object should be rendering full 3D models. Without designating a terrain material, the trees should be located on your entire terrain:





Scroll down to the **Types[0]->layer** property. Click on the blue box to open up the Material Selector. Once the dialog appears, select the alternative material (such as the dirt\_grass):

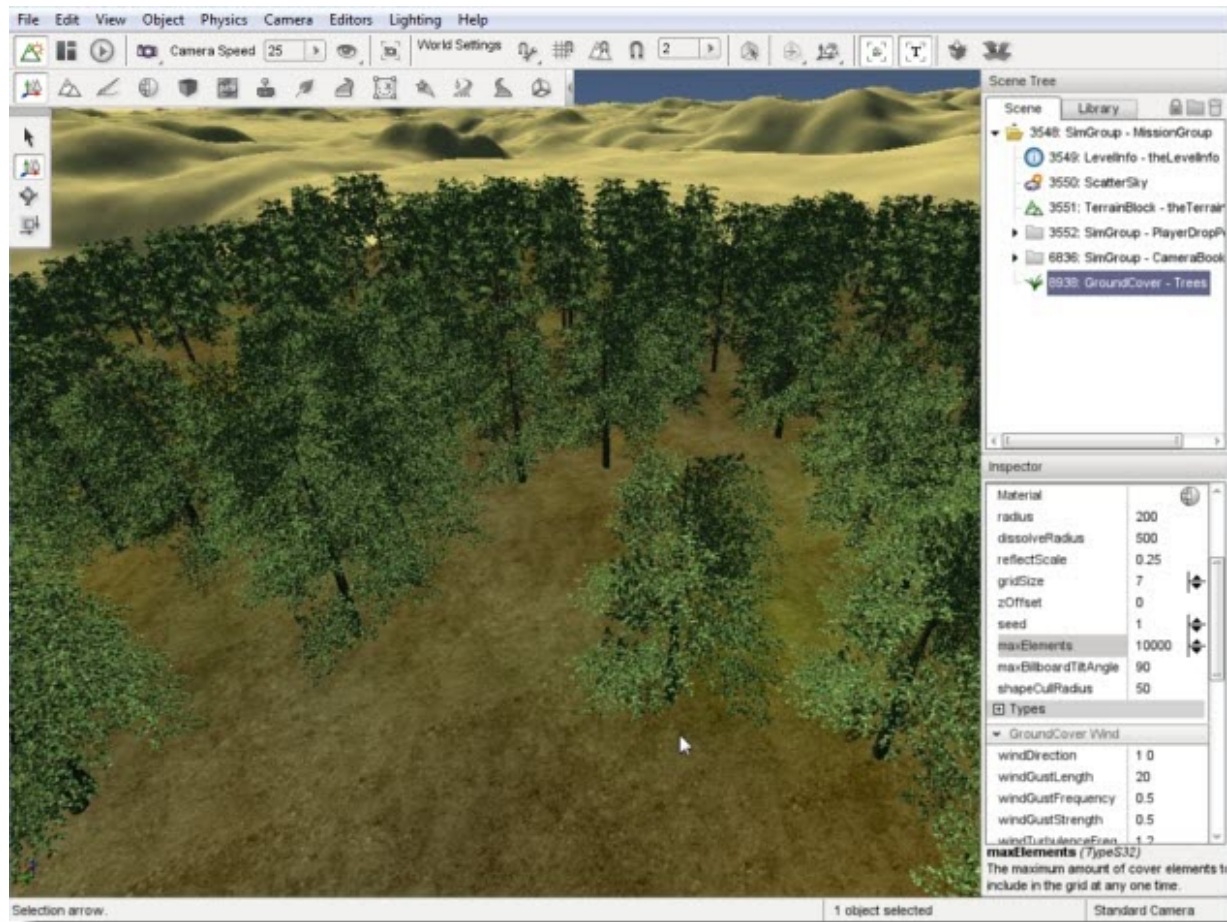


After you click the Select button, the trees will only be placed on the terrain material you picked. In this case, the trees will only be located on the grassy terrain:



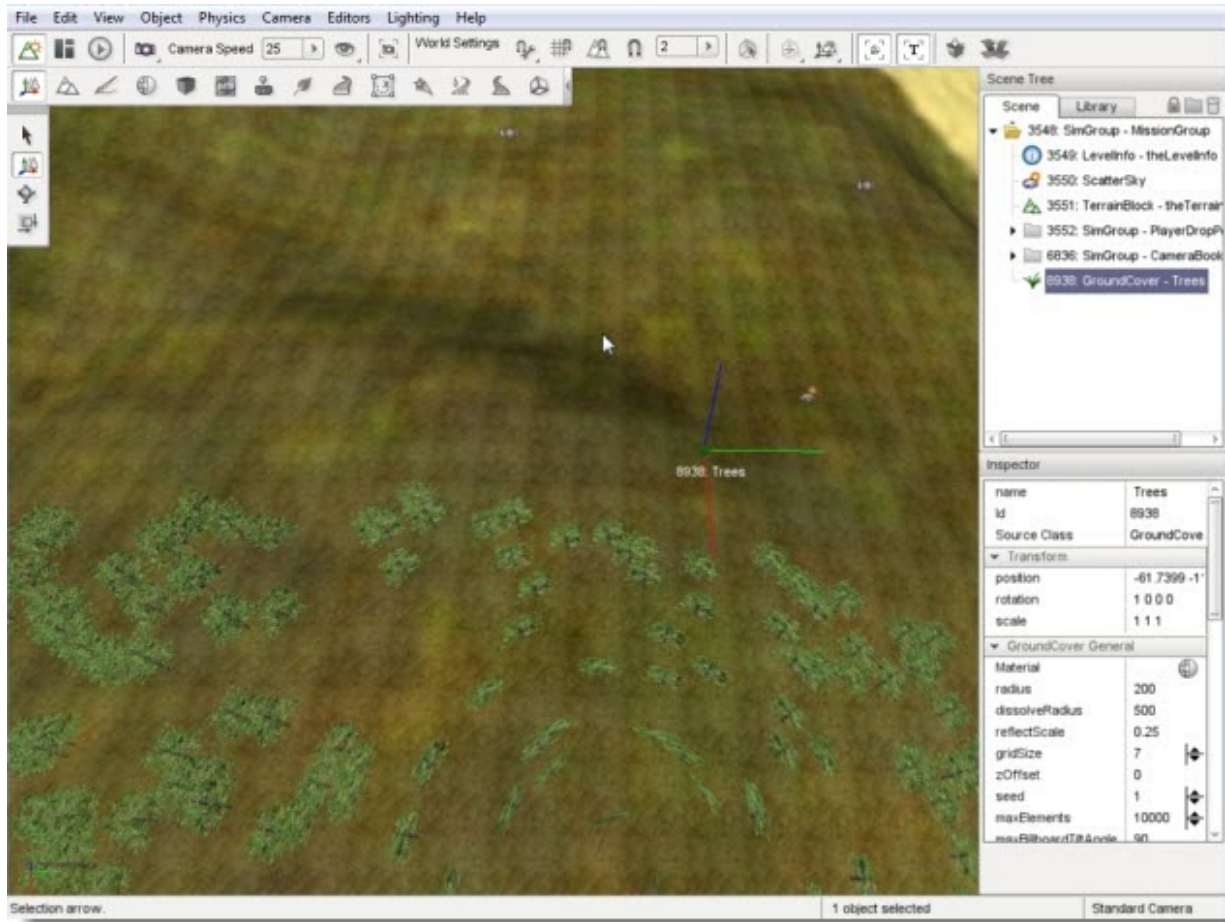
The adjusting properties will affect 3D models the same as the billboards we were editing earlier. For example, you can increase the `maxElements` to a very high number to simulate a forest only on the grassy terrain:



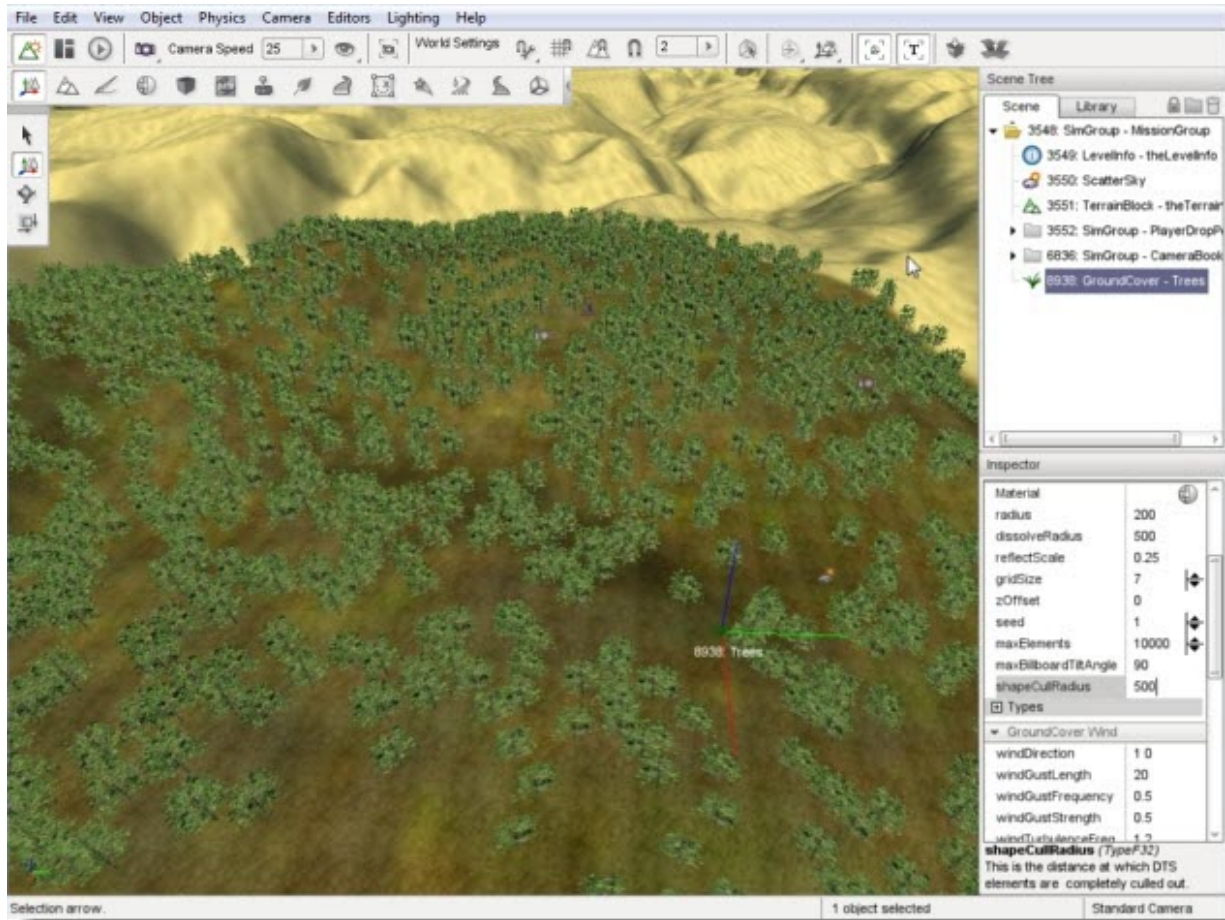


There is one property that is very specific to 3D models. You might notice that the trees disappear very quickly when you move away from them. This is controlled by the **shapeCullRadius** property. This property is the distance at which 3D shapes are completely prevented from rendering.

Low values might be OK for first person views, but at a higher range the results will not appear as you might expect. For example, this terrain looks void of trees in the upper half when in fact they are there but they are not being displayed. They are not displayed because they are further away from the camera than the **shapeCullRadius** distance of the Ground Cover Object:



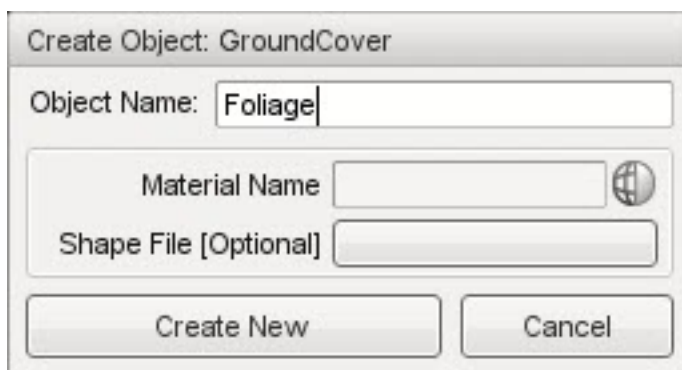
If you increase the value of `shapeCullRadius`, you will be able to see 3D shapes further out from where the camera is located. Try setting the value to something like 500. If you want your whole terrain to be within the area covered by the `shapeCullRadius`, then you must enter a value at least as great as the `Resolution` property of the `TerrainBlock` when you created it:



You should now have a basic understanding of how the GroundCover system works. However, instead of moving on to the next guide we are going to dive deeper into the features that make Torque 3D GroundCover so powerful and flexible. Continue reading to learn a few more advanced modifications.

### 14.4.8 Advanced Modifications

In the same level you have been using, delete all GroundCover objects. If you have already discarded the level, then create a new terrain and make sure you have at least two separate terrain materials being rendered. Create a new GroundCover object, filling out only its name. Leave the other fields blank:

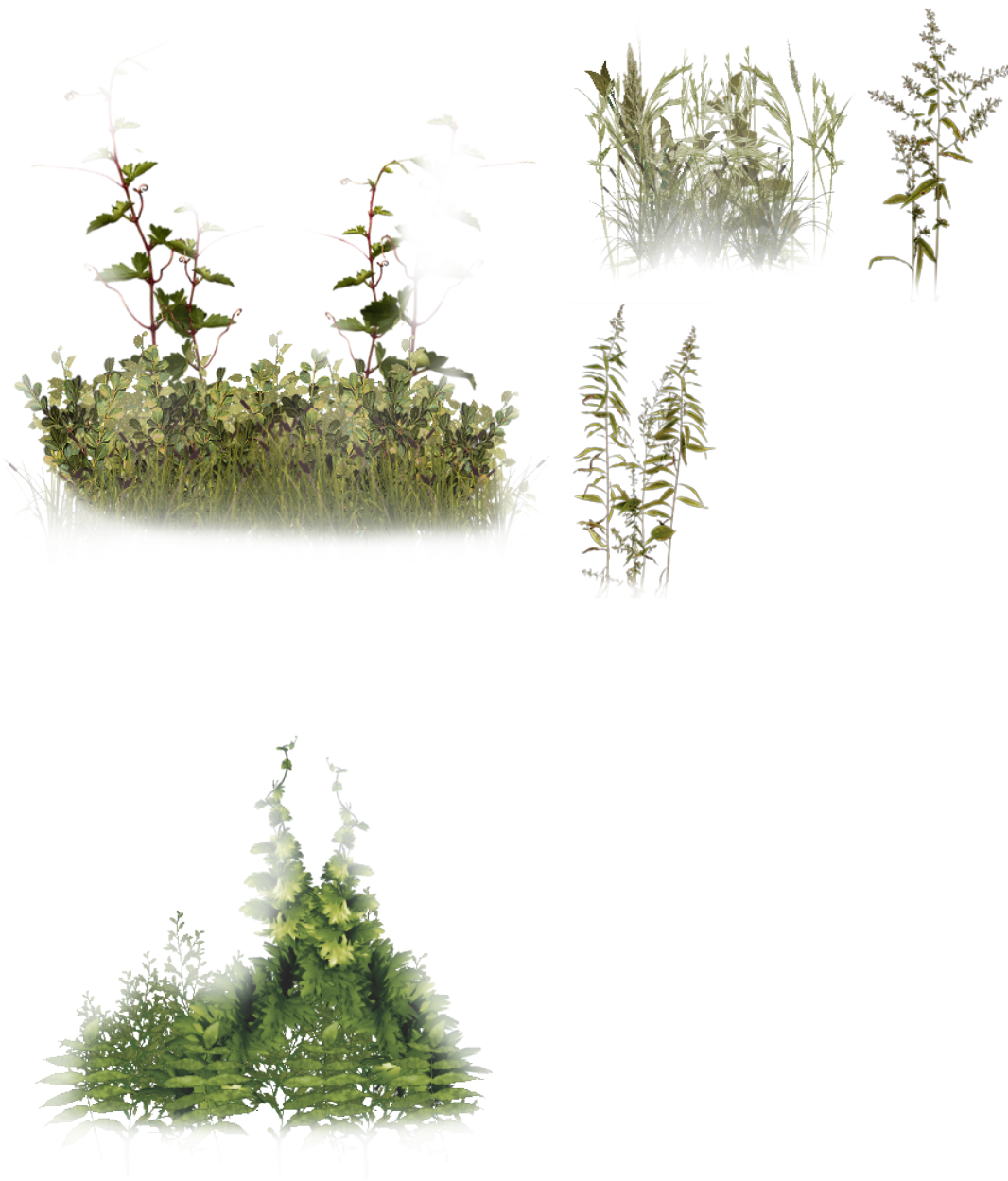




Click the Create New button to add the new GroundCover object to the level. Next, we will create a new material for this GroundCover. However, this will be a special image. Instead of a single material representing an individual plant, we are going to create a material that represents five plants.

Save the following image to your **game/art/environment/** directory:

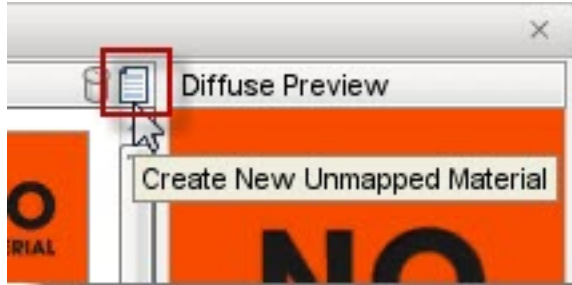
**foliage.png**



Notice how this new image file contains several plant images, stitched together in a single 1024x1024 png file. The

GroundCover can easily separate these individual plant images in order to use them all, but it needs a material to do so. Select your new GroundCover object. Scroll through the properties until you get to the GroundCover General section. Click on the Material Selector icon .. image:: img/GCMatPropIcon.jpg

Now we're going to go through the process of creating a new material like we did before - you know the drill. When the Material Selector appears, click on the "Create New Unmapped Material" button. Then click the Select button.

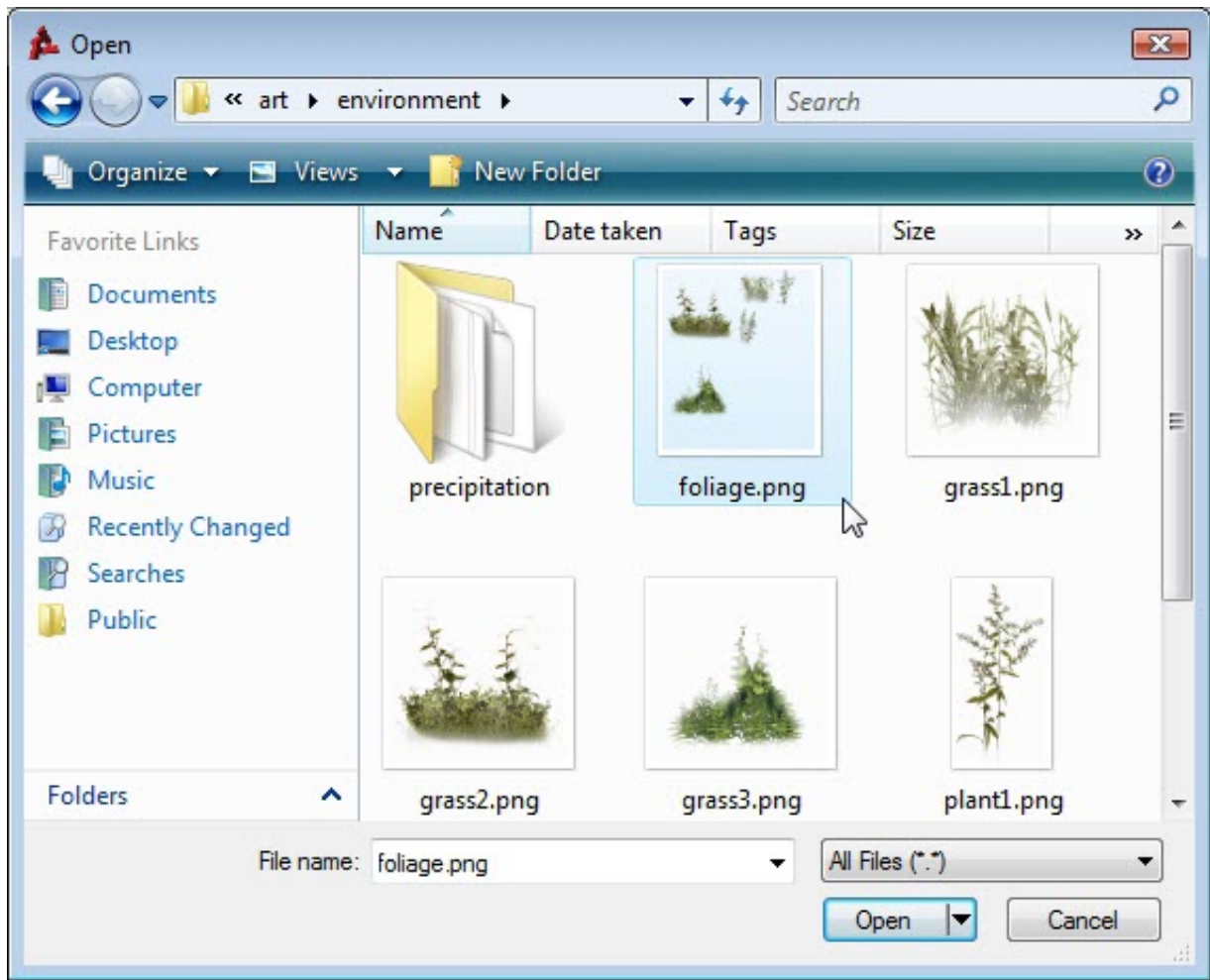


At this point, the new material has been applied to the GroundCover. We have a material, but it has not been assigned a texture so once again you will see the special "No Material" texture. Go ahead and click on the Material Editor icon to activate the tool:

#### Activate Material Editor

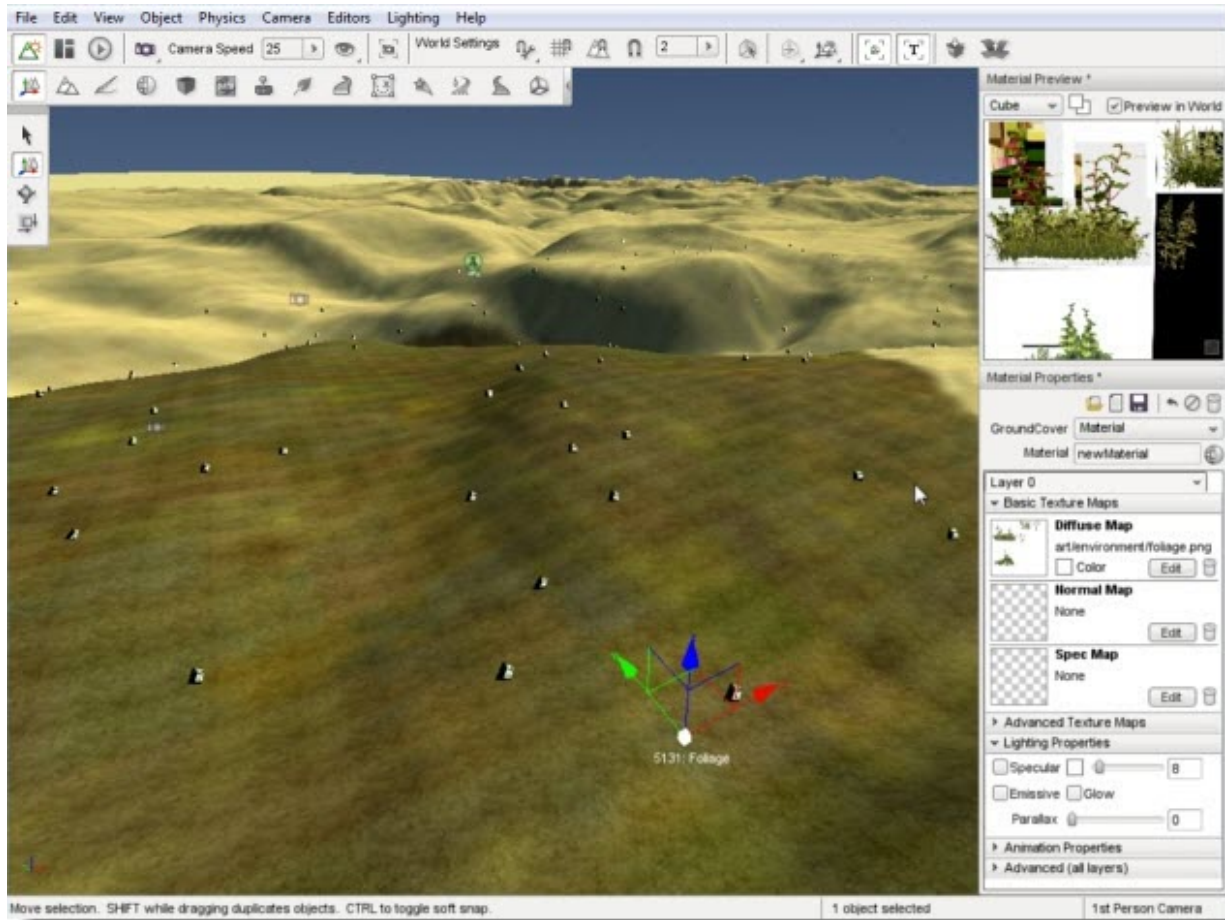


When the Material Editor opens, your GroundCover material will already be the active entry. Change the Material name to **Foliage**, hit Enter, then click the Floppy Disk icon to save. **NOTE: You MUST press the Enter key before clicking the save icon or your new material WILL NOT be saved!** Next, scroll down to the Basic Texture Maps section. Right now the Diffuse Map is assigned to the default "No Material" texture. Click on the preview image or the Edit button. A file browser should pop up allowing you to select a texture. Navigate to the **game/art/environment** directory. Select the **foliage.png** texture that you just placed there, and then click Open.

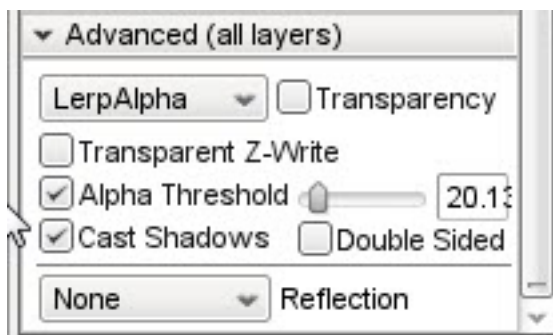


Your initial Foliage material is going to look very odd. This is due to the Alpha Threshold being disabled which causes the black areas to show as black rather than be transparent:





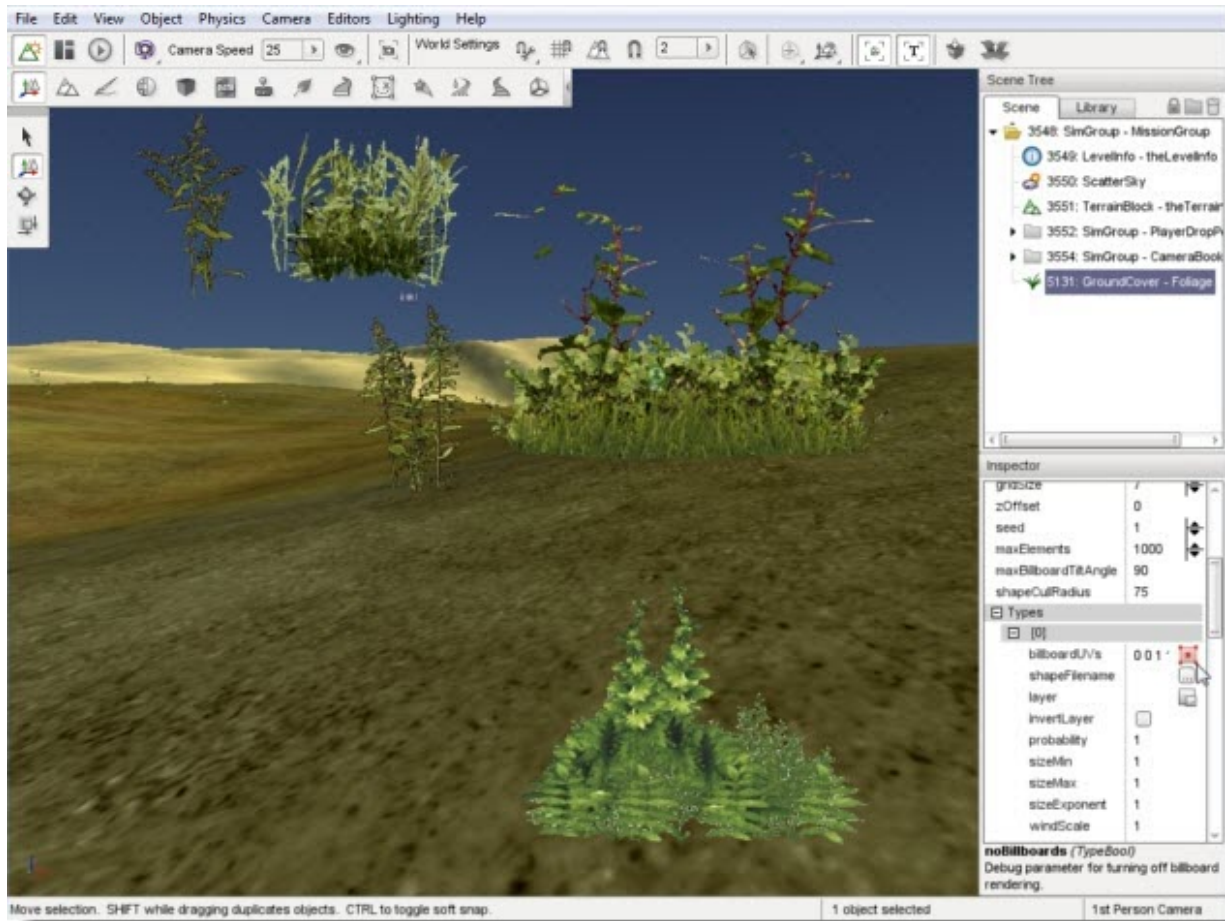
To fix this, scroll down to the **Advanced (all layers)** section of the Material Editor. Check the **Alpha Threshold** box, then set the value to something close to 20.13 (or whatever looks best to you):



**Save your material** by clicking the floppy icon again. Once you are finished with your material, it should resemble the following in the preview:



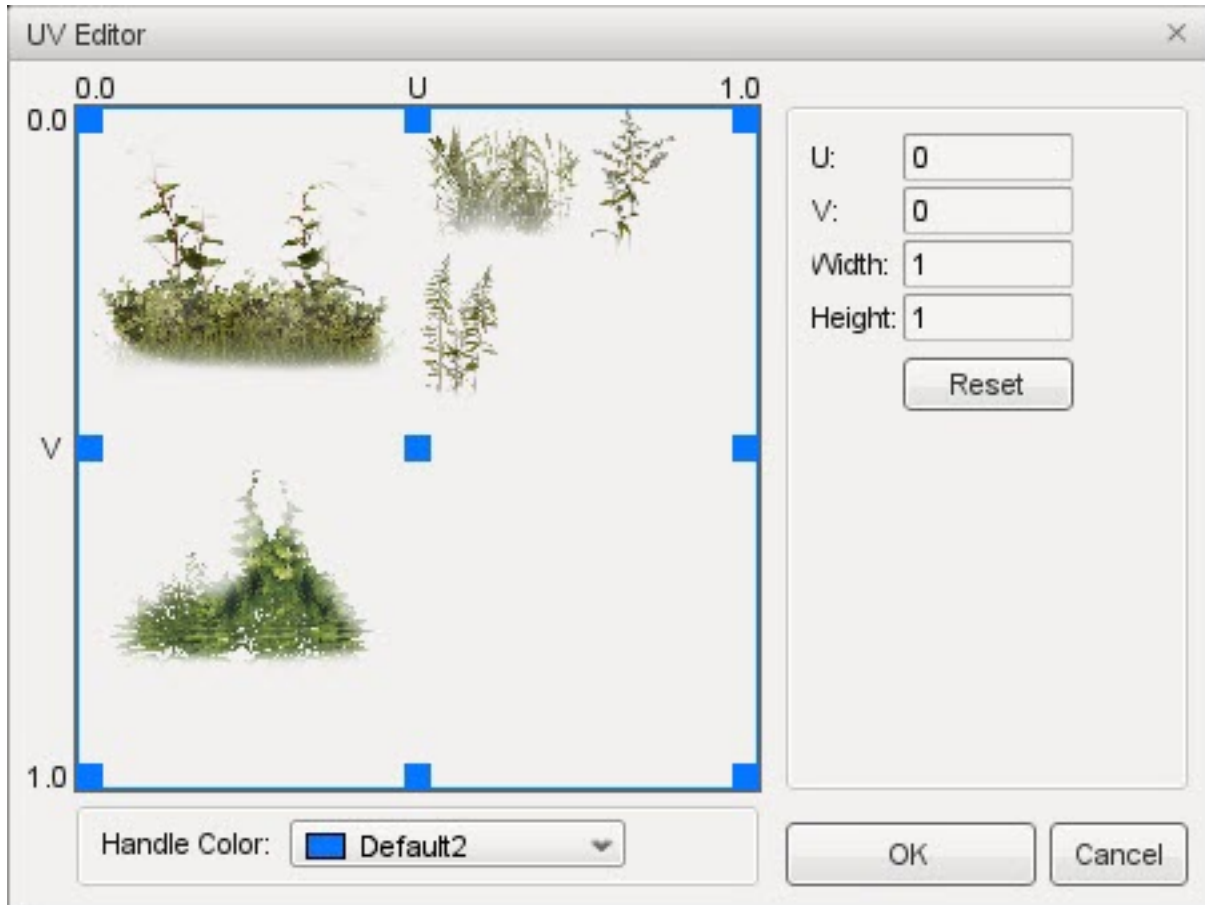
Now is a good time to also save your level if you want. When you are ready switch back to the Object Editor using F1. Your GroundCover should now be rendering the combined plant material.



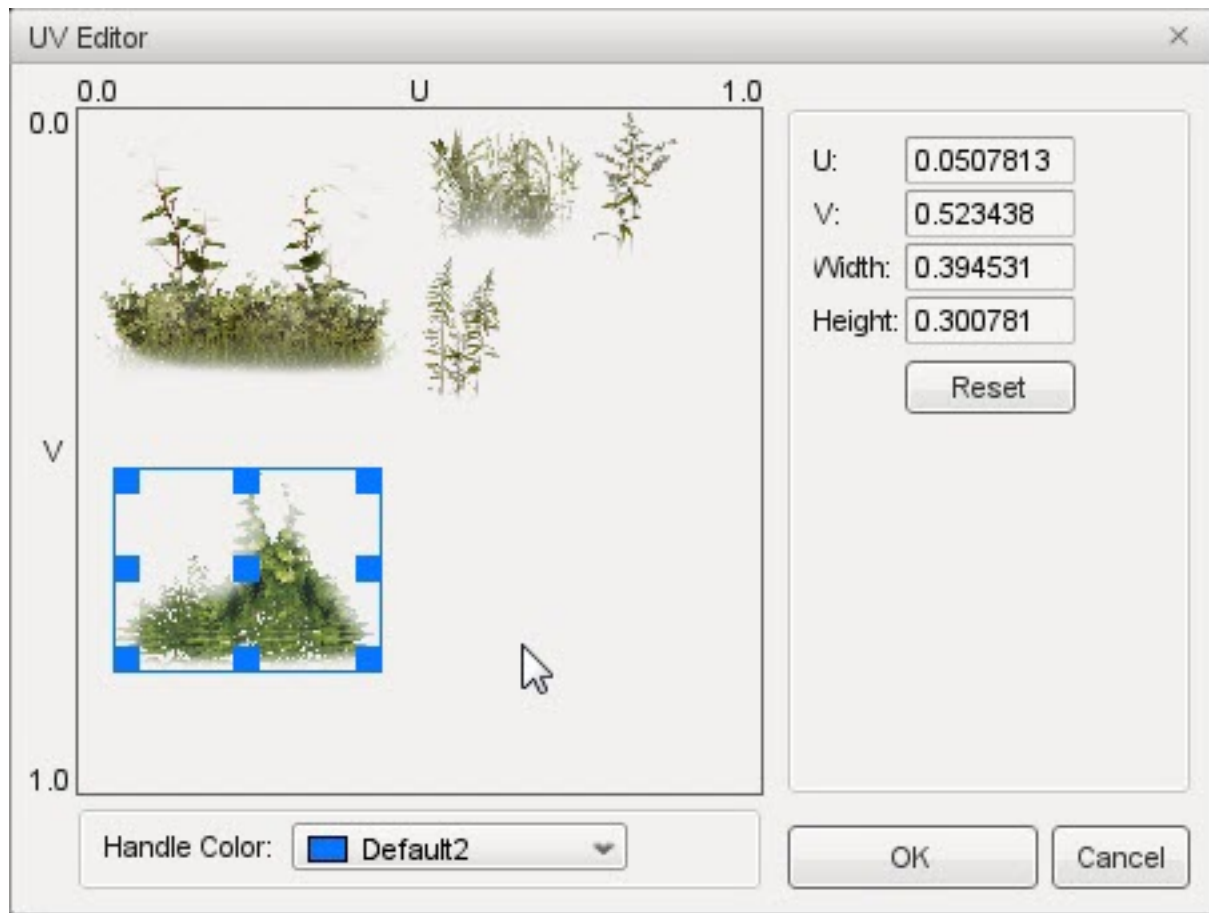
Within each Types[x] entry, we can use the UV Editor to designate which part of the material to use for the billboard rendering. The UV Editor is located under the Types section of the properties, in the billboardUVs field. Click on the square icon to activate the UV Editor:



The UV Editor is extremely simple to use. When the dialog appears, you will be presented a box comprised of nine points. You can drag the edges of the box around to isolate specific parts of a material:

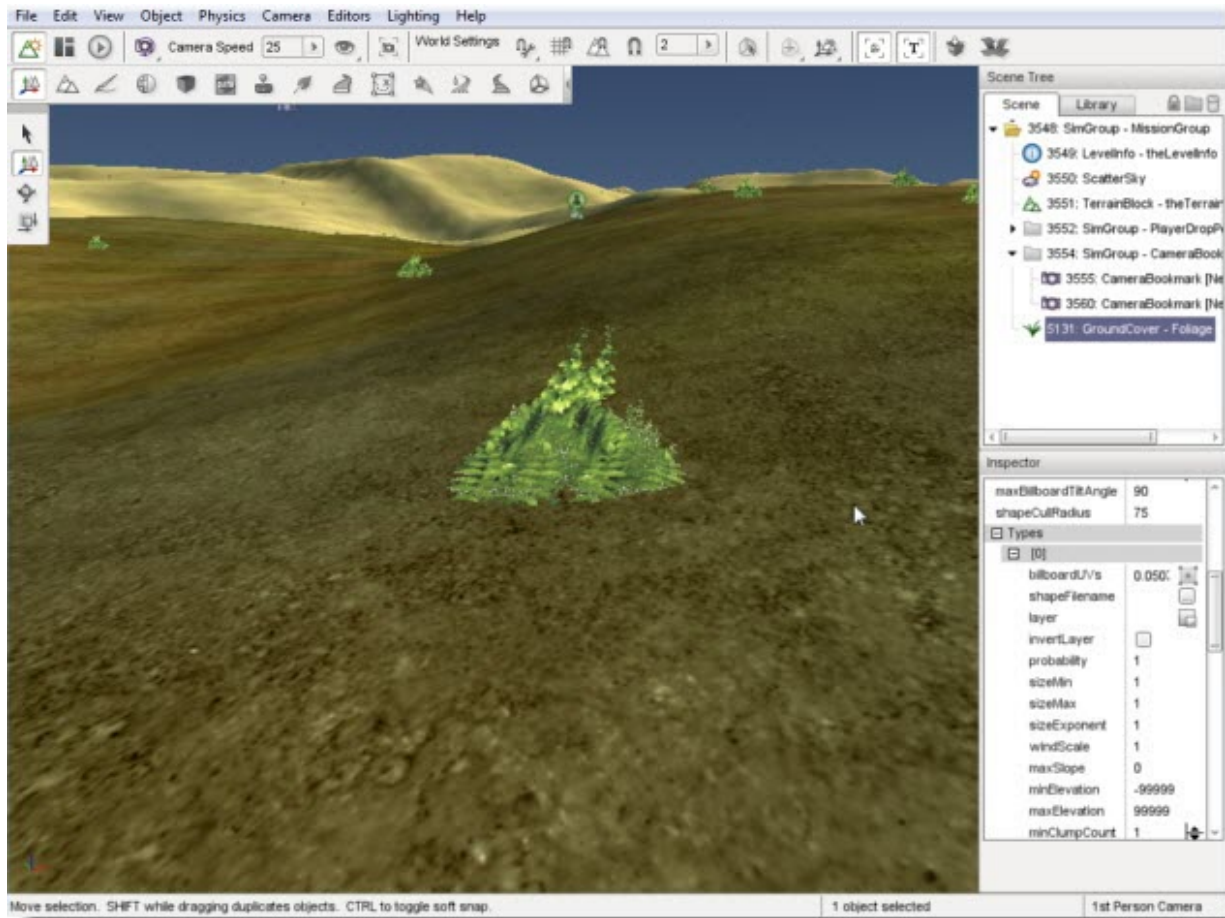


With the foliage material, this is exactly what we need to draw a single plant for this layer. Choose one of the plants you wish to render on the grass, such as the one in the bottom left. Click and drag the blue boxes until just that plant is enclosed in the square:



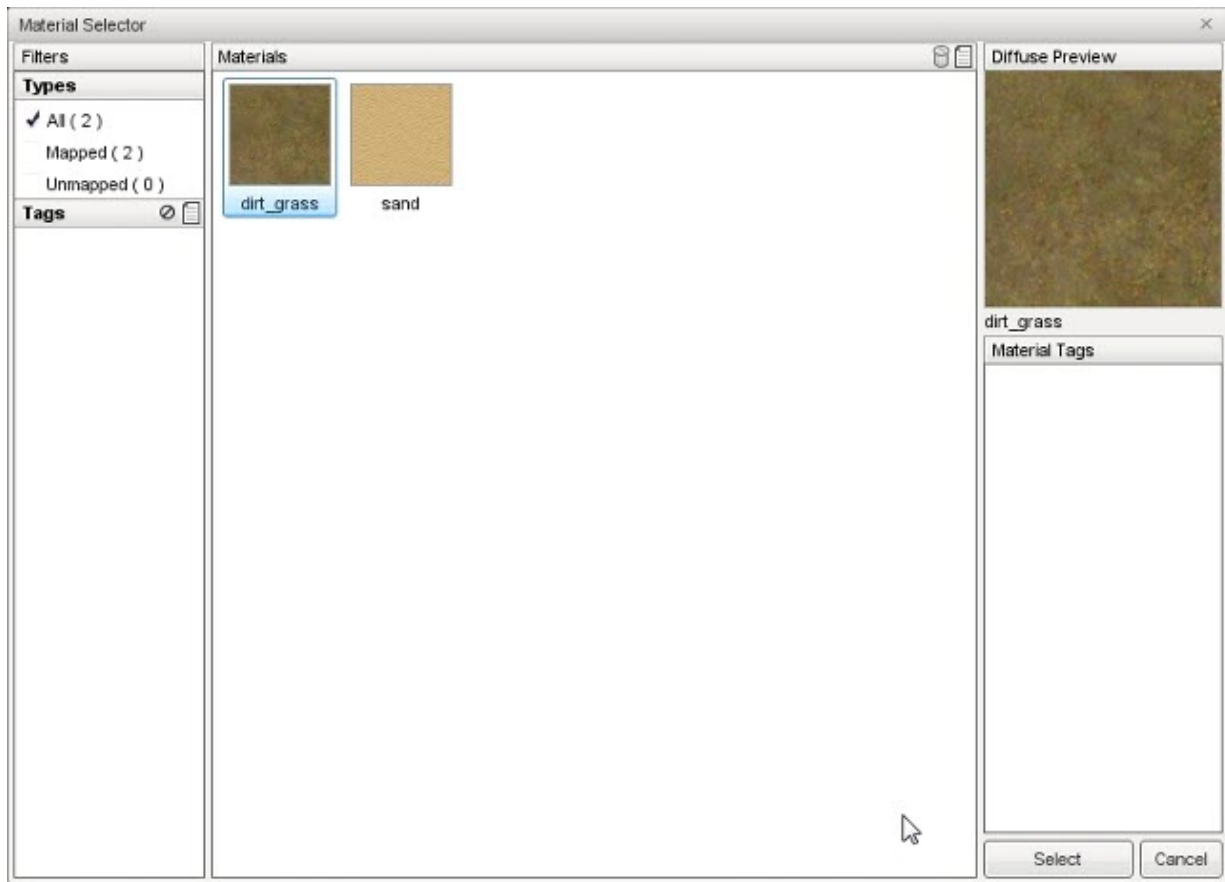
Once you are certain of your placement, click the OK button. The Types[0] layer will now only be rendering the section of the material that you surrounded. Rather than create five separate GroundCover objects, you can use a single material and one GroundCover object with each layer displaying a different plant, that is, section of the image.



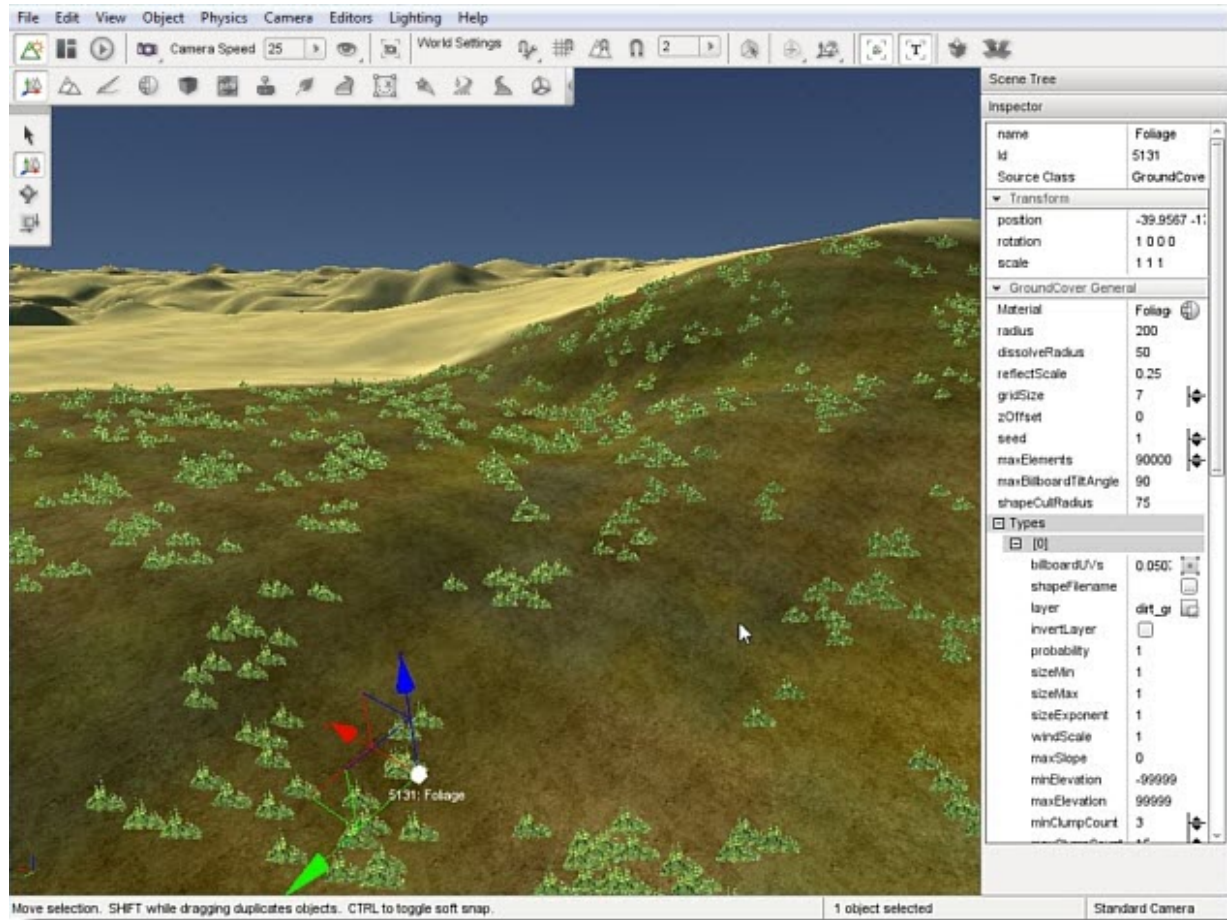


In this manner we can also isolate the individual plant types to specific terrain textures. In **Types[0]>layer**, click the box icon to bring up the Material Selector. Select a terrain material to limit this plant to:

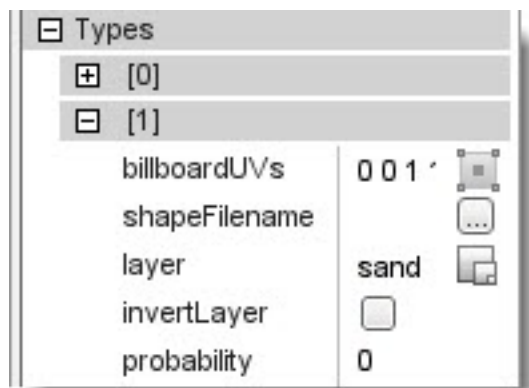




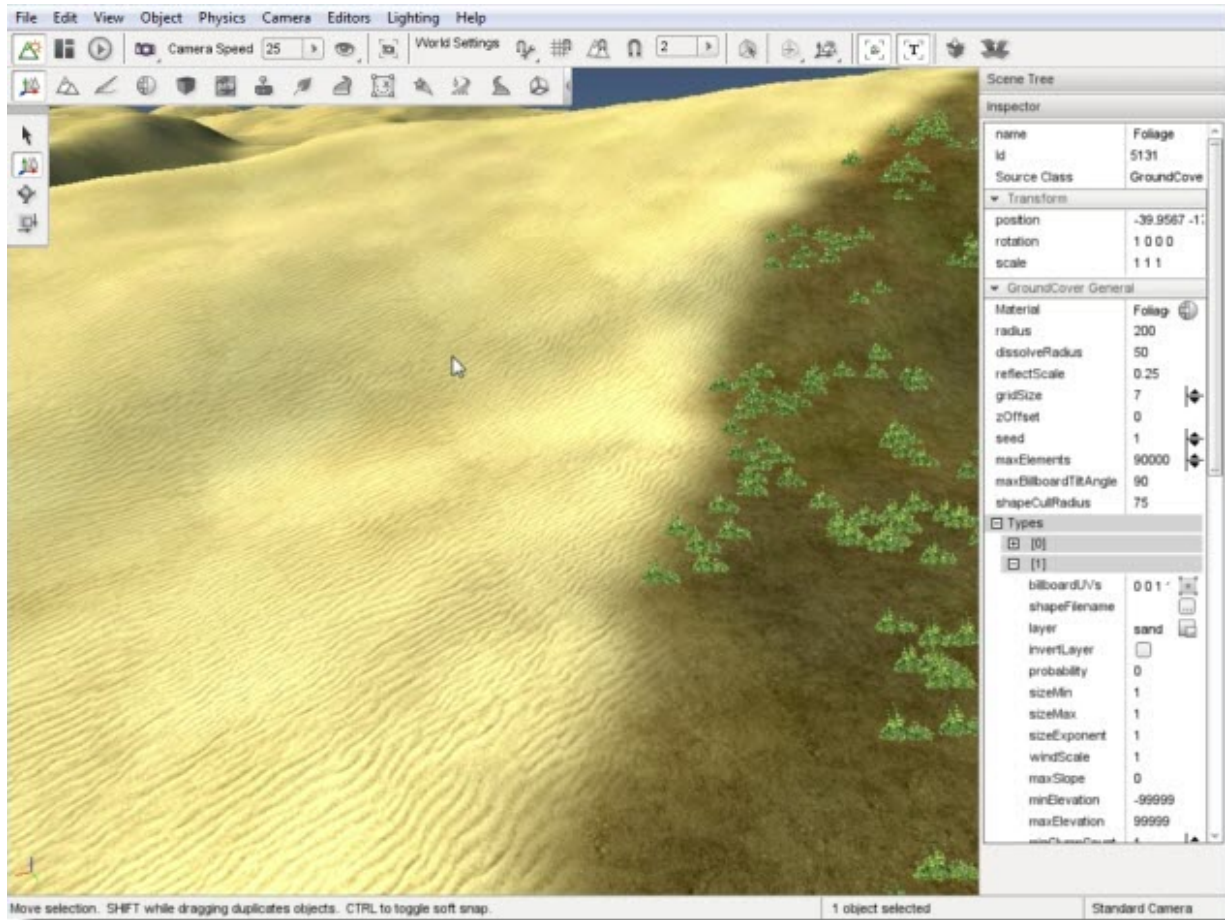
Go ahead and make a few more edits to `Types[0]`, such as adjusting the clumping properties. Ultimately, you are aiming to create something that appears varied, as it would in nature, rather than a systematic placement of objects:



For the next step, collapse the [0] entry and expand **Types[1]**. For this section of the GroundCover, we are going to render another piece of the foliage from that same texture on a separate terrain material. Go ahead and assign **Types[1]->layer** to another material that is in your level:

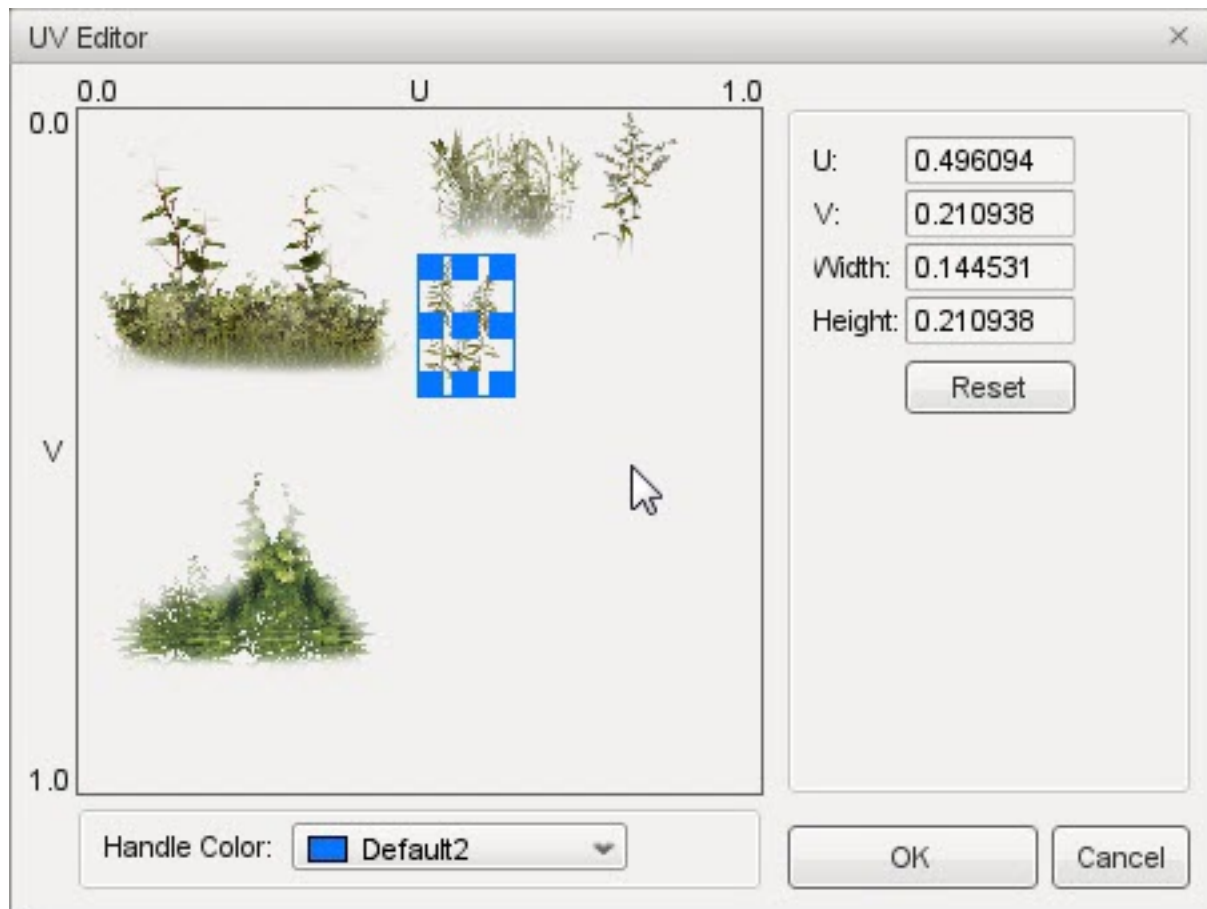


After you have assigned the terrain material, you might notice nothing is rendering. This behavior is controlled by the **probability** property:



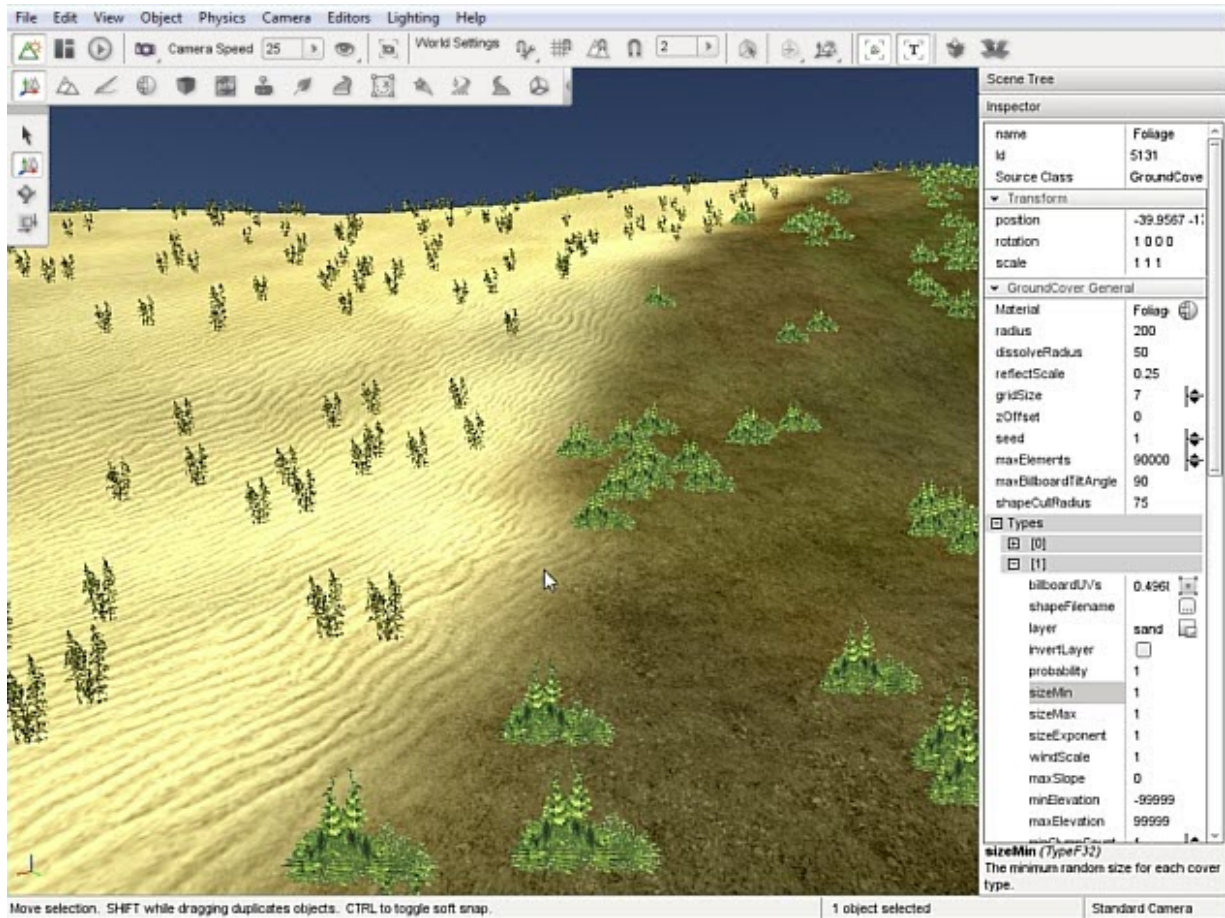
This property is a decimal value, which represents the likelihood (percentage) of an shape being placed by the Ground-Cover object. A value of 0.0 results in a 0 percent change, 0.5 is a 50% chance, and 1.0 is 100% chance that an object will be placed. By default, the probability is set to zero for all Types except Types[0].

Go ahead and set **Types[1]->probability** to 1, which will fully activate the placement of the foliage material. We need to make two more modifications. First, activate the UV Editor for this entry. Drag the UV boundaries to a less green image:

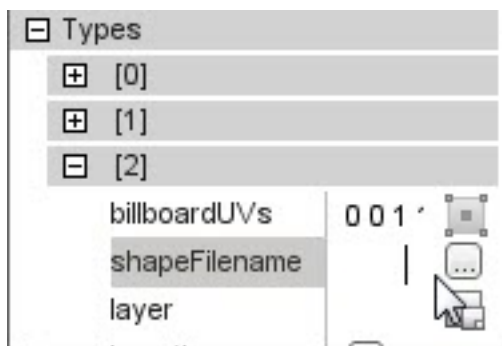


Next, set `Types[1]->layer` to another material in your level to isolate the placement of our new billboards. The final result should be `Types[0]` rendering a one type of plant on one material, and `Types[1]` rendering a different plant on the other material:



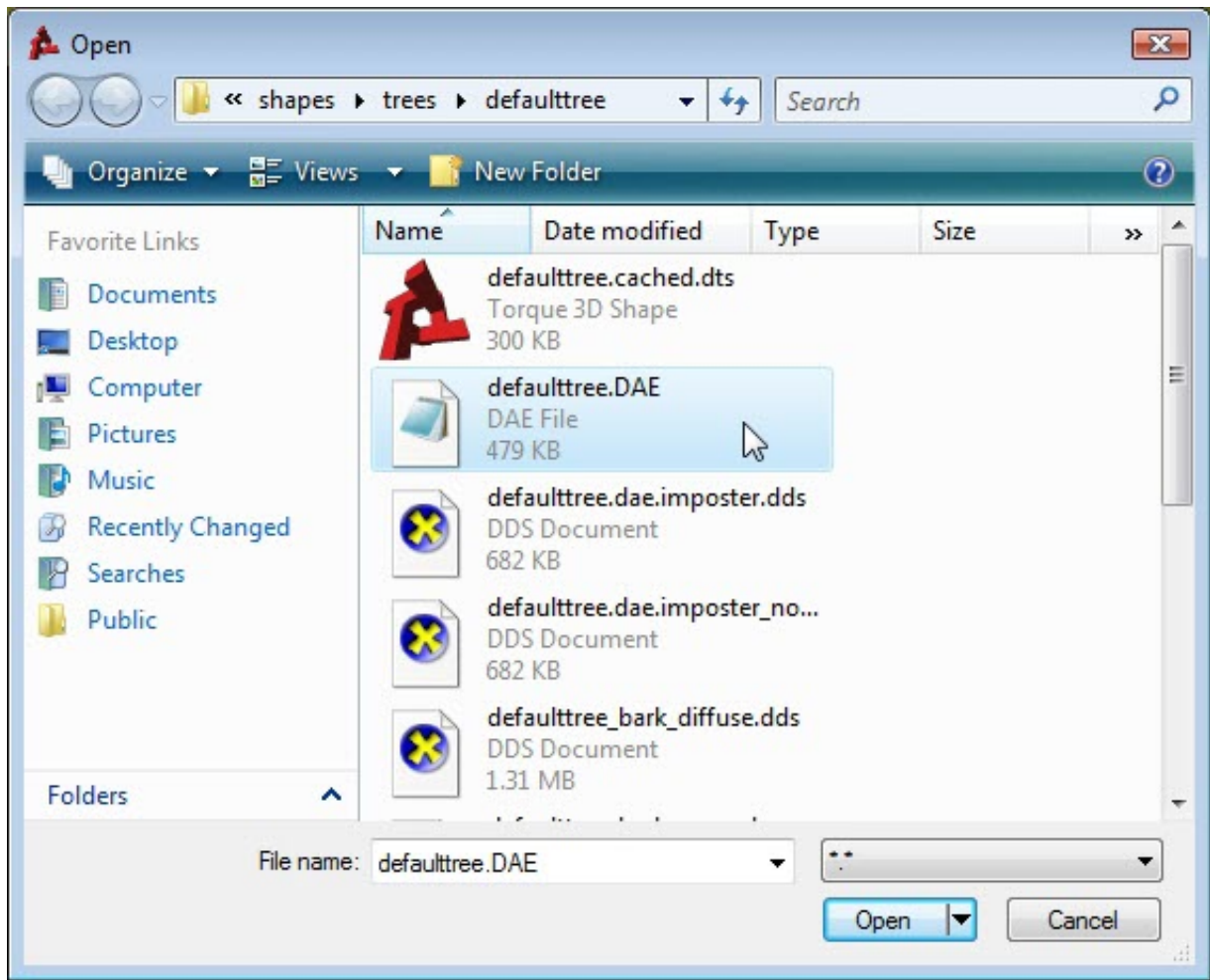


Separating billboards from within a single material is not the only function you can do when modifying the Types section. It is possible to mix in 3D shapes as well. Collapse Types[0] and Types [1], then expand Types [2]. Locate the Types[2]>shapeFilename property - click the box next to it:

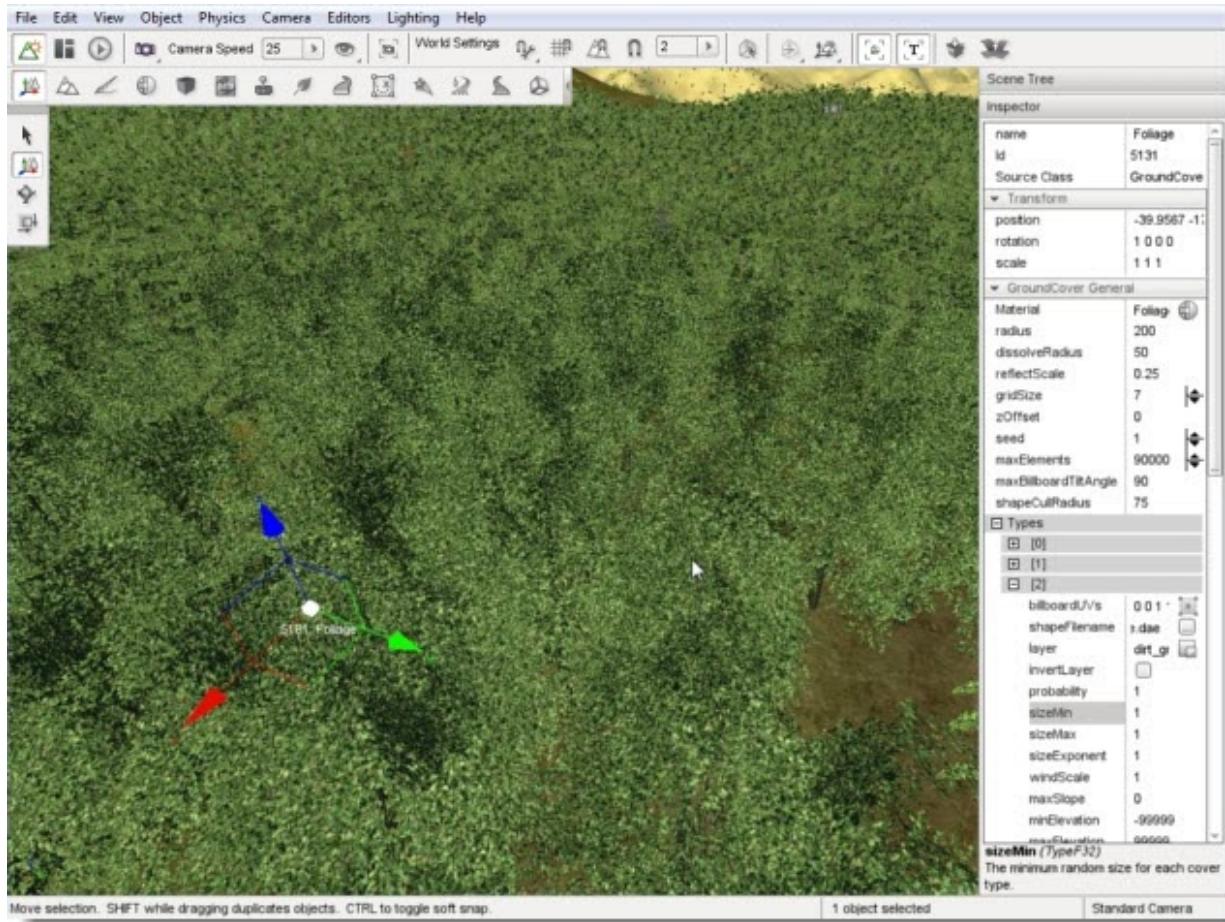


Navigate to the **game/art/shapes/trees/defaulttree** directory. Select the **defaulttree.DAE** file, then click open.



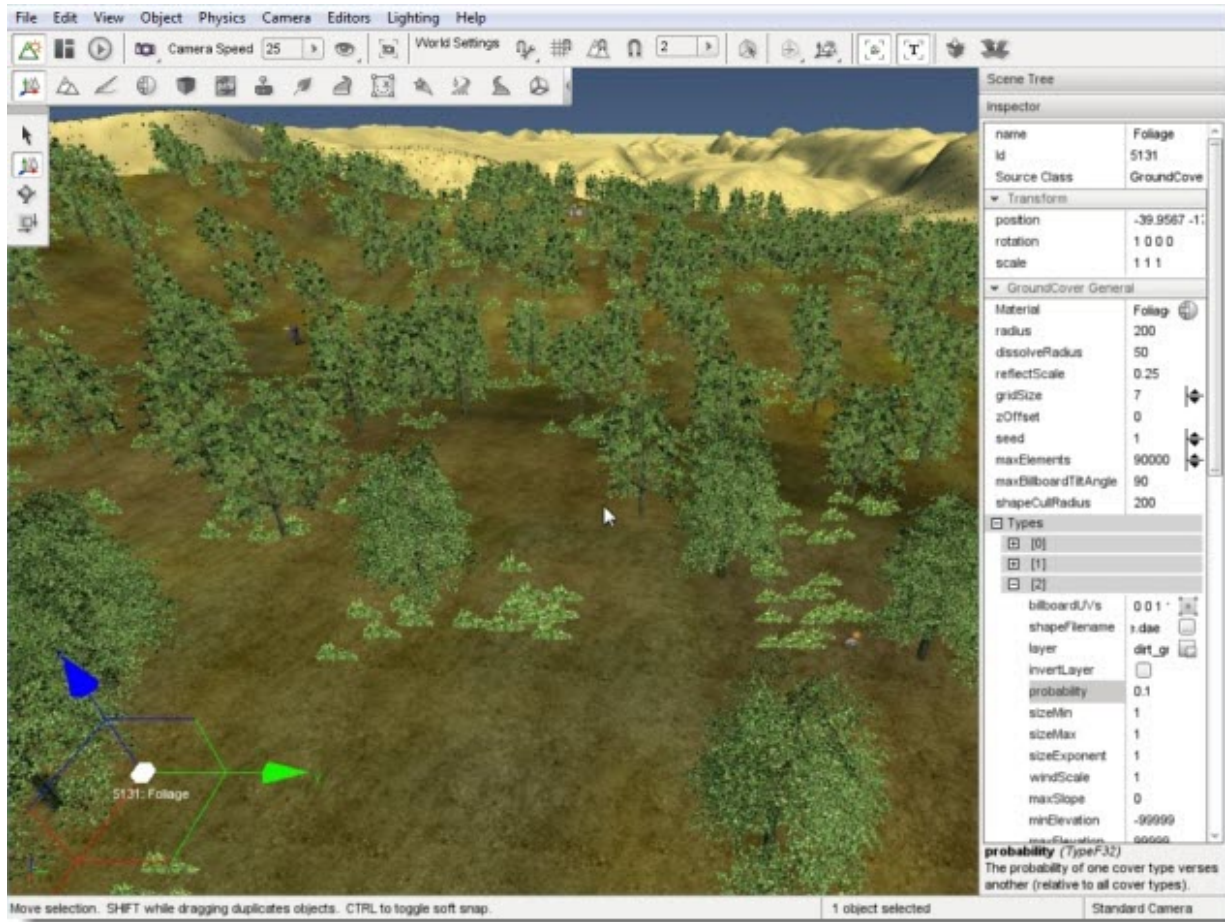


After you click open, the 3D shape will be set. Next, assign **Types[2]->layer** to the grass terrain material and set **Types[2]->probability** to 1. This will result in many trees rendering on the grass terrain:



For our example, there are way too many trees. Under the GroundCover General section we have the maxElements property set to 90,000. If we adjust this number, we will have less trees being placed. However, this also affects our other layers. This can be a problem.

Rather than fiddling with the maxElements property, we can adjust Types[2]->probability. Set the value of probability to 0.1, which will cause the GroundCover to place quite a few less trees:



As you can see, the GroundCover is extremely powerful. With a single object, we were able to place multiple types of foliage (billboards), separate the foliage based on terrain material, and even mix in 3D models:





### 14.4.9 Conclusion

The GroundCover object can add an immense amount of ambiance to your level. It is one of the most powerful and flexible Torque 3D objects. Continue experimenting by trying different types of settings, art, and level arrangements.





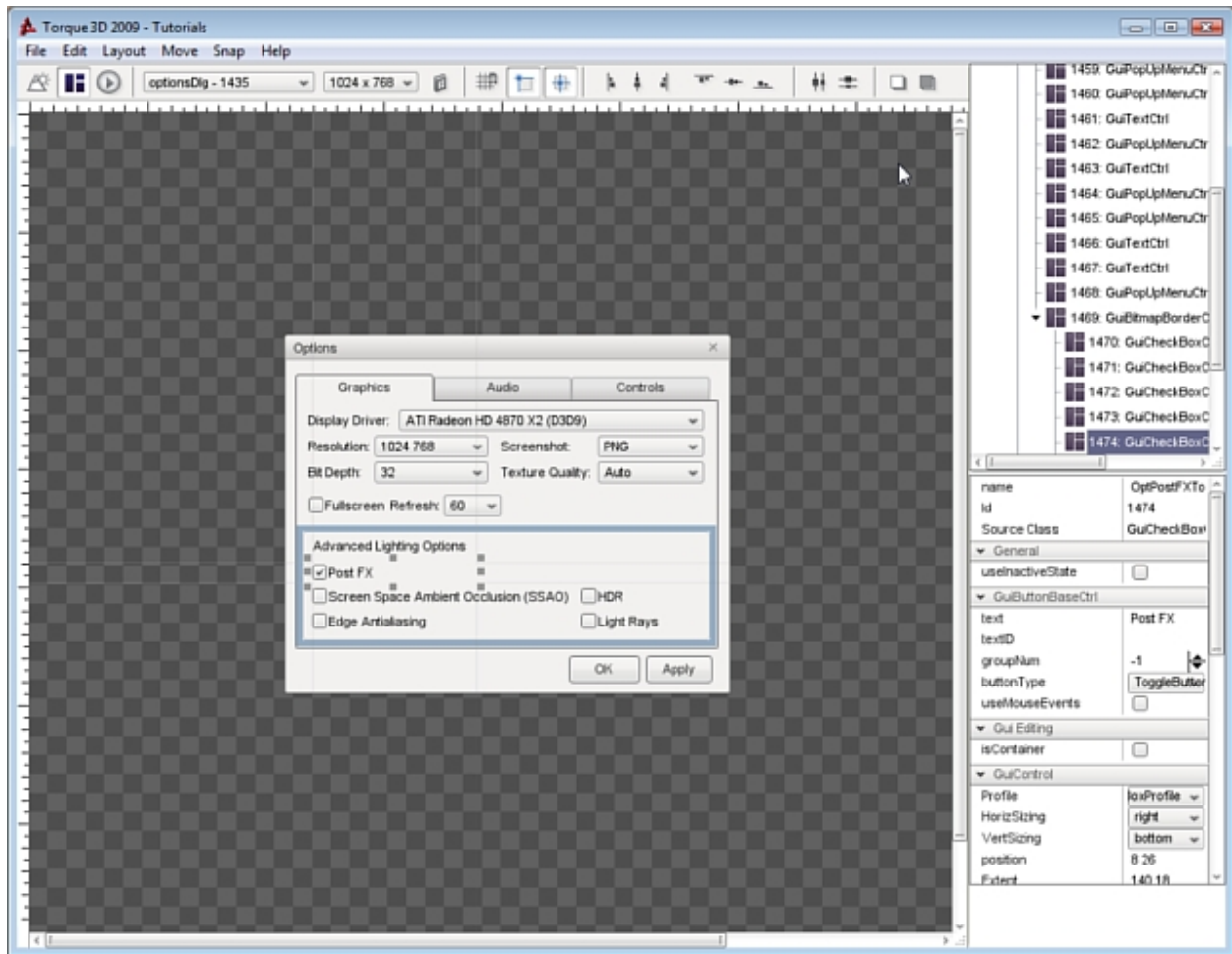
## CHAPTER 15

---

### Overview of GUI Editor

---

“GUI” stands for Graphical User Interface. It is the summation of all the controls (windows, buttons, text fields, etc.) that are used to interact with a game and its settings. Most interfaces in games consist of buttons to launch or join a game session, editing devices to change user preferences, options to change screen resolutions and rendering options, and elements which display game data to the user as they are playing.



GUI creation and design is extremely important to game development. Many decent games have been crippled by inaccessible GUIs, which is why having a built in GUI editor can be a blessing. The Torque 3D editor provides drag and drop functionality, with minimal fill in the blank requirements.

Torque 3D features a WYSIWYG GUI Editor, which allows you to create, edit, and test your GUI in game with maximum fidelity. 90% of your GUI creation can be done in the editor, leaving 10% for scripting advanced functionality.

GUIs are saved as a script (.gui), which allows you to further tweak values using your favorite text editor. Additionally, you can declare variables and define functions at the end of a GUI script, which will not be written over when modifying the GUI using Torques editor.

Multiple controls which can be combined to make up a single interface. Each control is contained in a single structure, which can be embedded into other GUI elements to form a tree. The following is an example of a GUI control which displays a picture:

```
// Bitmap GUI control
new GuiBitmapCtrl() {
    profile = "GuiDefaultProfile";
    horizSizing = "width";
    vertSizing = "height";
    position = "8 8";
    extent = "384 24";
    minExtent = "8 8";
    visible = "1";
    helpTag = "0";
```

(continues on next page)

(continued from previous page)

```
    bitmap = "art/gui/images/swarmer.png";  
    wrap = "0";  
};
```

Once the above GUI is active in your interface, it will display the following:









## CHAPTER 16

---

### Tutorials

---

## 16.1 Creating a New GUI - TODO

### 16.1.1 Introduction

### 16.1.2 Setting Up

### 16.1.3 Our First GUI

### 16.1.4 First Control

### 16.1.5 Text Control

### 16.1.6 Dynamic Text

### 16.1.7 Conclusion

## 16.2 Adding Controls - TODO

### 16.2.1 Introduction

### 16.2.2 Setting Up

### 16.2.3 Adding Controls

### 16.2.4 Bring the GUI to Life

### 16.2.5 A New Window

### 16.2.6 Load GUI From Key Press

### 16.2.7 Conclusion

## 16.3 Mini Console - TODO

This is a simple tutorial to create a mini Console in a window, complete with clear button. You will use a small amount of script to make a toggle key for the mini-console and also look at how to use the text entry box.

Suggested Reading:

- GUI Editor Overview
- GUI Interface
- TorqueScript Reference
- GUI Tutorial 1 (Creating a New GUI)
- GUI Tutorial 2 (Adding Controls)

Covered in this tutorial:

- How to setup a small console
- Set up a clear button
- Bind this dialog to a key
- Learn how to use Text entry
- Take a Look at profiles and what they are

## 16.3.2 Setting Up

By now you should be use to the setup procedure. Open your tutorial project and head to the GUI editor or alternatively use the Torque 3D Toolbox approach. Create a new GUI using the following:

New GUI Name: **miniConsole**

Gui Class: **GuiControl**

Select the GuiControl and set its property profile to **GuiModelessDialogProfile**.

**NOTE:** The GuiModelessDialogProfile sets our GUI so that the background does not prevent mouse events from reaching objects beneath it. Simply put, the user can click through the clear parts.

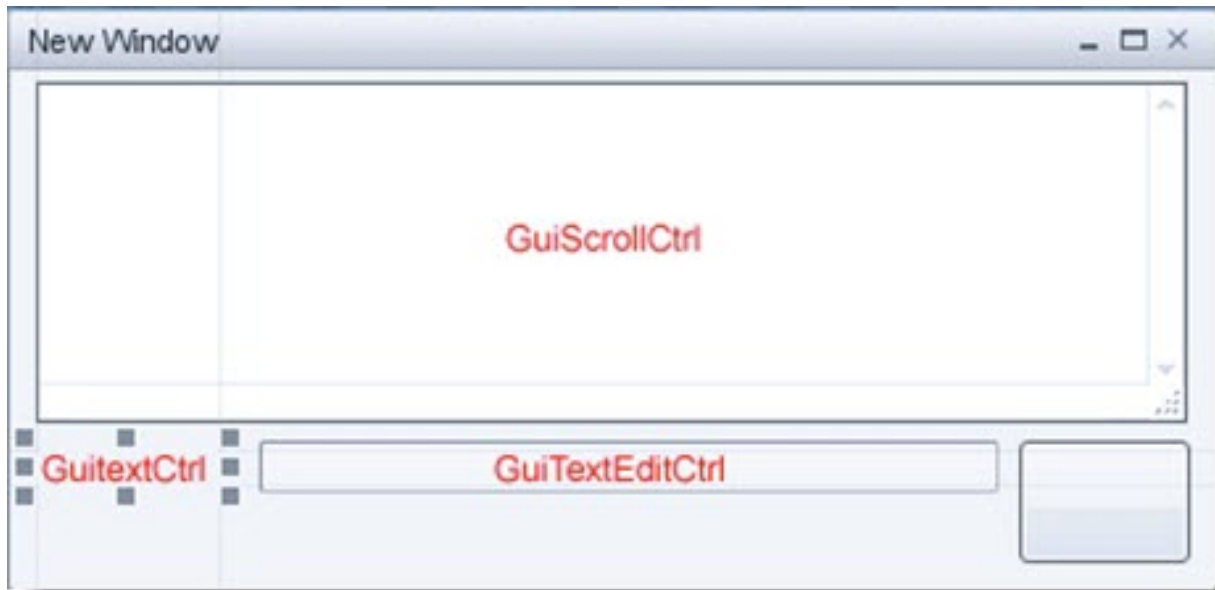
Select a **GuiWindowCtrl** from the library, add it to the editor work area, and resize so that it can contain the console.

## 16.3.3 Add the Controls

**STEP 1:** The next task is adding the controls to our window. Select the GuiWindowCtrl control from the controls list so that it is highlighted and add the following controls:

- **GuiScrollCtrl**
- **GuiTextCtrl**
- **GuiTextEditCtrl**
- **GuiButtonCtrl**

Move and resize the controls so that they match the image below (text in red is to show where each control is):



Save your GUI into your project **game/art/gui** folder with the name of **miniConsole.gui**. Next we need to set some control properties as follows:

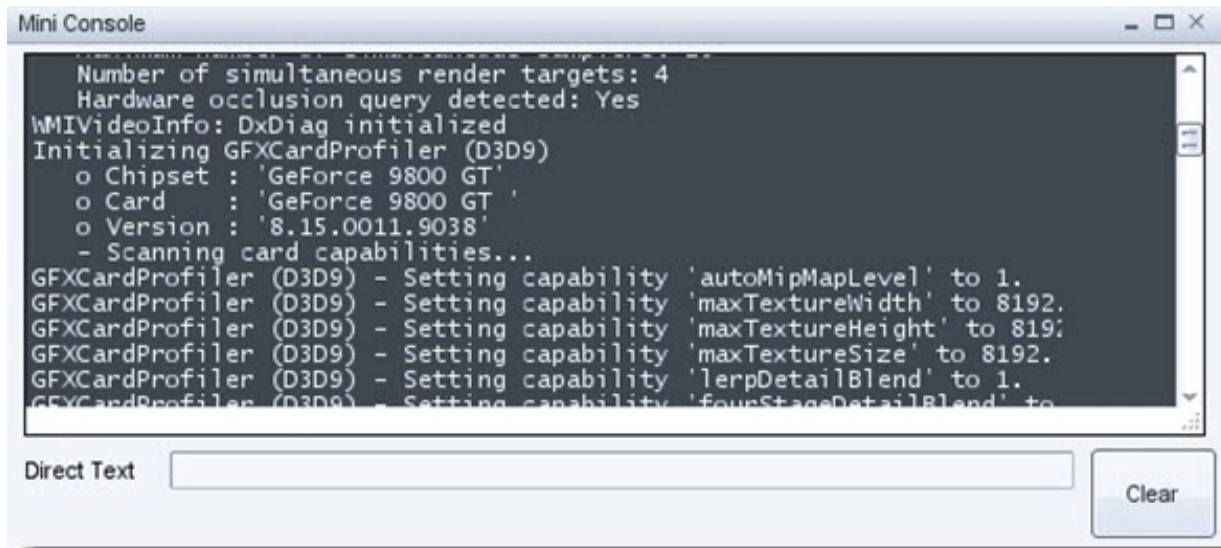
**GuiWindowCtrl** property text: **Mini Console** **GuiTextCtrl** property text: **Direct Text** **GuiButtonCtrl** property text: **Clear**

**STEP 2:** Select the **GuiScrollCtrl** control as we are going use it as a container for our console. With the scroll control selected add a **GuiConsole** from the Editor category. In the controls list, the new console control should be nested inside the scroll control. Do not worry about resizing the console control. The scroll box will handle its size.



This will enable us to scroll through the console as it is larger than our window. If you preview your GUI (F10) you will notice that the color is wrong for the console and also the window may be a little too narrow. We can rectify this by changing the **GuiScrollCtrl** profile to **ConsoleScrollProfile** and resize the window and scroll control horizontal width.

You may have to move the other GUI components so that it looks tidy again.



Do not forget to save often!

### 16.3.4 The GuiTextEdit Control

What we are going to do now is make our text edit control send its contents to the console when we press enter. Basically echo to the console what we type into the text edit control. First set the Clear button property to:

button property Command: `cls()`;

**cls()**

Use the cls function to clear the console output.

Syntax:

```
cls()
```

**Returns** No return value.

Examples:

```
cls();
```

Now when we press this button the console will be cleared. We can now test the GUI by closing the GUI Editor, press F10 and try it out.

GuiTextEditCtrl property name: **txtDirect**

GuiTextEditCtrl property AltCommand: **echo(txtDirect.getValue());**

**echo** (string, all)

Sends output to the console

Syntax:

```
echo(string text, all [...]);
```

**Parameters**

- **text** – Text sent to console
- **[ . . . ]** – Optional value, of any type, that will be appended to the text

**Returns** No return value.

Examples:

```
// Print "Hello World" in the console
echo("Hello World");
```

You may have noticed that this time our command was placed into the **AltCommand** property, the reason for this is so that the control waits until we press enter to send the command string, instead of sending the command on each letter entered.

It would be good that when we pressed enter that the text edit control emptied itself, to save us having to highlight and delete the text ourselves next time we want to enter a new word. Change the property as follows:

GuiTextEditCtrl property AltCommand: **echo(txtDirect.getValue()); txtDirect.setValue("");**

Now when you enter a word and press return the word is displayed in the console and the text is removed from the edit box. Now is a good time to save.

### 16.3.5 Commanding the Console

To do this we are going to need a new text edit control and a text label. Select our **GuiTextCtrl** and **GuiTextEditCtrl**, then copy and paste. Move the new copy beneath the first. Change the copied controls properties as follows:

GuiTextCtrl property text: **Command**

GuiTextEditCtrl property name: **txtEnterCommand**

GuiTextEditCtrl property AltCommand: **eval(txtEnterCommand.getValue()); txtEnterCommand.setValue("");**

The first command in the script sends the contents of the text edit control txtEnterCommand to the console to be executed. The following then clears the text ready for the next command to be entered.

**eval** (script)

Use the eval function to execute any valid script statement

Syntax:

```
eval(string script);
```

**Parameters** **script** – A string containing a valid script statement. This may be a single line statement or multiple lines

**Returns** const char\* Returns the result of executing the script statement.

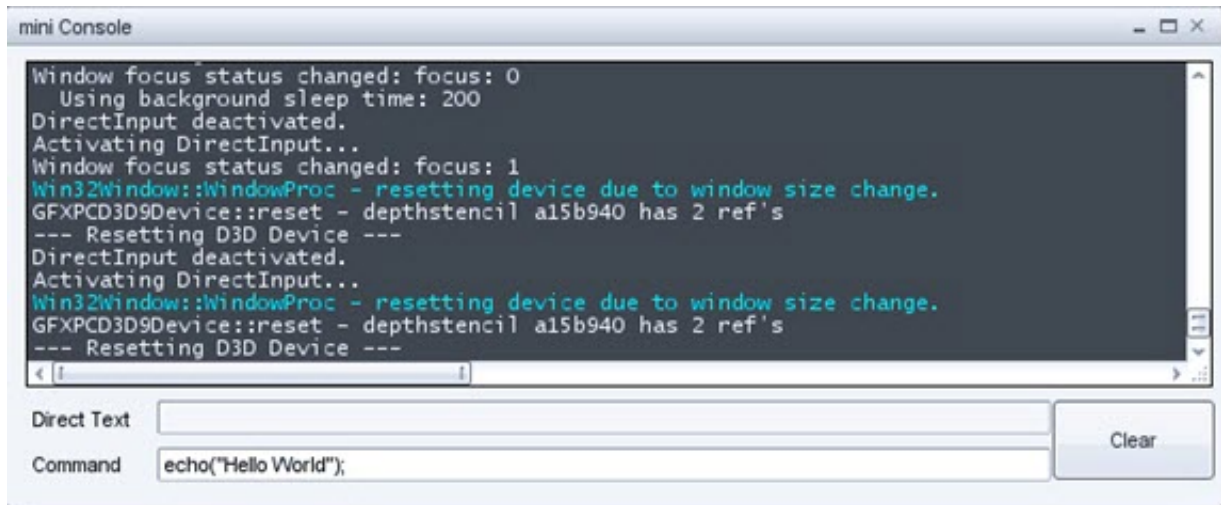
Examples:

```
eval("game/scripts/client/test.cs");
```

**Note:** If you choose to eval a multi-line statement, be sure that there are no comments or (\) comment blocks (\*\*) embedded in the script string.

Your mini Console should now look a little like this.





### 16.3.6 Activating the Console

**STEP 1:** We are going to call our new console from Ctrl + ~ (tilde) to keep inline with the main console. Open the file `scripts/client/default.bind.cs` in Torsion or another text editor. Head to the end of this file and add the following:

```
function callMiniConsole(%val )
{
    if(%val )
    {
        if ( miniConsole.isAwake() )
        {
            // close the mini console.
            Canvas.popDialog( miniConsole );
        }
        else
        {
            //open the mini console
            Canvas.pushDialog( miniConsole );
        }
    }
}

GlobalActionMap.bind(keyboard, "ctrl tilde", callMiniConsole);
```

This function allows us to use the same key press to open / close the dialogue by checking the GUI status **guiControl.isAwake**.

**STEP 2:** Next open the file `game/scripts/client/init.cs` and look for the `// Execute the GUI scripts and functions` section and add the following:

```
exec("art/gui/miniConsole.gui");
```

Now run your project and press the Ctrl + ~ (tilde) key to see your mini Console.

**STEP 3:** One last edit to be made with our GUI, to enable the window close icon:

GuiWindowCtrl property `closeCommand`: **Canvas.popDialog(miniConsole);**

Remember to save before testing the close icon.

**NOTE:** If you want to use this as a project console you may remove the direct text control leaving the command text and text edit. This would be more useful as the first Text edit control was for instruction purposes only.

### 16.3.7 Conclusion

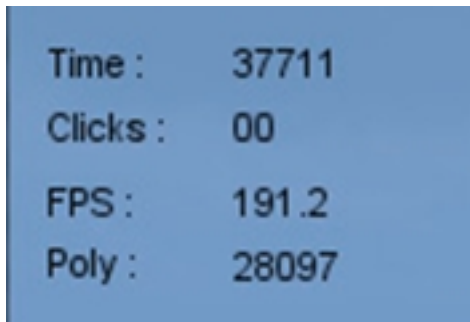
In this tutorial, you learned the following concepts:

- How to setup a small console
- Set up a clear button
- Bind this dialog to a key
- Learn how to use Text entry
- Take a look at profiles and what they are

The next tutorial will show you how to display a GUI while playing the game, which is the foundation for creating a HUD (Heads Up Display).

## 16.4 Simple HUD

### 16.4.1 Introduction



With this tutorial you will learn some simple scripts that will be used to display the game FPS and polygon count as a overlay or HUD (Heads Up Display). We will also cover how to capture a mouse event and put it to good use while a schedule timer is used to constantly update the text displayed on our screen.

Suggested Reading:

- GUI Editor Overview
- GUI Interface
- TorqueScript Reference
- GUI Tutorial 1
- GUI Tutorial 2
- GUI Tutorial 3

Covered in this tutorial:

- Display text on the Game screen
- Show how to detect a mouse click
- Simple use of the schedule timer

- Show game fps and poly count
- Locate and Edit the playGui

### 16.4.2 Setting Up

Open up your tutorial project and start the GUI editor as we have in the previous tutorials in the series. This time we are going to edit the main playGui.gui. This GUI is the main play screen overlay where your in game HUD for ammo and other player feedback is displayed.

The main GUI control in this case is not GuiControl that we used in a dialog. This one uses GameTSCtrl, which is a GUI for rendering 3D scenes. Head to the file menu File->Open From File and point your file browser to **data/FPSGameplay/scripts/gui**. Load the file **playGui.gui**.

This GUI may have a few controls already but we shall ignore them for now.

### 16.4.3 Adding Text Controls

**STEP 1:** For this project we are only going to need to display text so add a new GuiTextCtrl from the control Library. Set its property to:

```
property text: Time
```

Next resize it to fit the text content. Ensure this control is selected, then copy and paste a new control to the right of it and set this controls property to:

```
property text: 00
property name: lblTime
```

**STEP 2:** Select both of the text controls then copy and paste them three more times. Set their properties as follows:

**Below Time:**

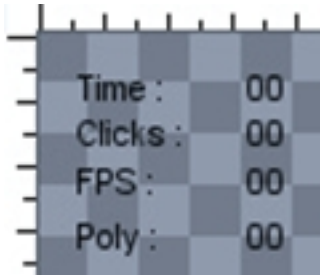
```
Left Text property text: Clicks:
Right Text property text: 00
Right Text property name: lblClicks
```

**Below Clicks:**

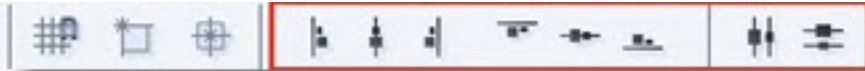
```
Left Text property text: FPS
Right Text property text: 00
Right Text property name: lblFps
```

**Below FPS:**

```
Left Text property text: Poly
Right Text property text: 00
Right Text property name: lblPoly
```



If you need aid in lining your text controls, the editor has a few tools to help you. The selected control can be moved / nudged by pressing the up/down/left/right keys. The tool bar also has a few icons for aligning up the controls:



Select all the left text controls and press the **Align left** icon:



This will make their left edges line up nice and neat. If you want the text controls to be evenly spaced vertically, press the **Distribute Vertically** icon:

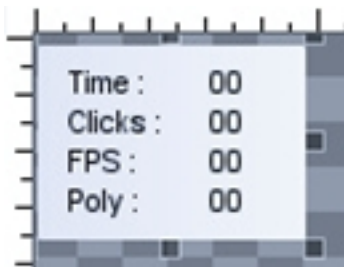


Get to know these layout helpers as they will save you time in a larger GUI. Remember to make a save!

**Optional:** If you think the black text will not show up in your project you could always add a `GuiPanelCtrl`, resize it, and send to the back by pressing the tool bar **Send To Back** icon.



It should look like this:



Save and close down your project.

**HINT:** If you can not see all the numbers or text in a `GuiTextCtrl`, remember to resize the control until you can see all of its contents.

## 16.4.4 Configuring Text Output

Now our text controls are in place we need a little bit of script to bring it to life.

STEP 1: Open your **data/FPSGameplay/scripts/client/playGui.cs** file in your favorite editor and at the end of the first function (function PlayGui::onWake(%this)) add this line:

```
schedule(100,0, updateDisplay);
```

This method is a type of timer. When the specified amount of time has passed, a call is made to the function.

**schedule( waitTime , objID or 0, functionName, arg0, ... , argN )**

Use the function to schedule *functionName* to be executed with optional arguments at time *waitTime* (specified in milliseconds) in the future. This function may be associated with an object ID or not. If it is associated with an object ID and the object is deleted prior to this event occurring, the event is automatically canceled.

**Syntax:**

```
schedule(U32 waitTime, SimObject* objID, string functionName, arg0, ... , argN );
```

### Parameters

- **waitTime** – The time to wait (in milliseconds) before executing **functionName**.
- **objID** – An optional ID to associate this event with.
- **functionName** – An unadorned (flat) function name.
- **arg0...argN** – Any number of optional arguments to be passed to **functionName**.

**Returns** S32 Returns a non-zero integer representing the event ID for the scheduled event.

Example:

```
// Call the updateDisplay function in 100 milliseconds
schedule( 100,0, updateDisplay );
```

This is how the function should look with our new line of code at the end:

```
function PlayGui::onWake(%this)
{
    // Turn off any shell sounds...
    // sfxStop( ... );

    $enableDirectInput = "1";
    activateDirectInput();

    // Message hud dialog
    if ( isObject( MainChatHud ) )
    {
        Canvas.pushDialog( MainChatHud );
        chatHud.attach(HudMessageVector);
    }

    // just update the action map here
    moveMap.push();

    // hack city - these controls are floating around and need to be clamped
    if ( isFunction( "refreshCenterTextCtrl" ) )
```

(continues on next page)



(continued from previous page)

```

        schedule(0, 0, "refreshCenterTextCtrl");
        if ( isFunction( "refreshBottomTextCtrl" ) )
            schedule(0, 0, "refreshBottomTextCtrl");

        schedule(100,0, updateDisplay); //our new schedule
    }

```

**STEP 2:** At the end of the script page add the following new function:

```

function updateDisplay()
{
    lblTime.setValue((getRealTime()/1000));
    schedule(100,0, updateDisplay);
}

```

Here we have made another call to the schedule which will give us a loop timed at 100ms updating any statements in this function. The first line sets the **GuiTextCtrl** named **lblTime** text content to **getRealTime()** which is the current time in milliseconds.

**STEP 3:** Next we will set up the text to display the current fps and poly count of the scene. This is done by looking at the global variables **\$fps::real** and **\$GFXDeviceStatistics::polyCount**. Add the following lines of script after the time statement:

```

lblFps.setValue($fps::real);
lblPoly.setValue($GFXDeviceStatistics::polyCount);

```

Your function should now look like this:

```

function updateDisplay()
{
    lblTime.setValue((getRealTime()));
    lblFps.setValue($fps::real);
    lblPoly.setValue($GFXDeviceStatistics::polyCount);
    schedule(100,0, updateDisplay);
}

```

Save your script and run your project. All being well and you have no errors, your display should show the time in ms, fps and poly count. Next we need to count the mouse clicks. Close down your project so we can add some more script.

**STEP 4:** To count our mouse clicks we need a way to tell the system that our mouse button has been pressed. We do this in the same way as we did for checking for a pressed key, with the bind method. Open your **data/FPSGameplay/scripts/default.keybinds.cs** file for editing. At the end of this file add the following:

```

function mouseFire()
{
    // add the value of one to our "click" text control
    lblClicks.setValue(lblClicks.getValue()+1);
}

moveMap.bind(mouse, "button0", mouseFire);

```

We have set the mouse button0 to call mouseFire function every time its pressed by adding it to the bind method. Run the project and click the left mouse button. The mouseFire() function sets the text controls body text every time its called, this counts the mouse down and mouse up events.

To stop this and only count the mouse down even we need to modify the function slightly. Close your project and edit the file as follows:

```
function mouseFire(%val)
{
    if(%val)
    {
        //mouse down
        // add the value of one to our "click" text control
        lblClicks.setValue(lblClicks.getValue()+1);
    }else
    {
        //mouse up
    }
}
```

Now the mouseFire function only counts the mouse down event. Save your files, run the project and test the mouse down counting.

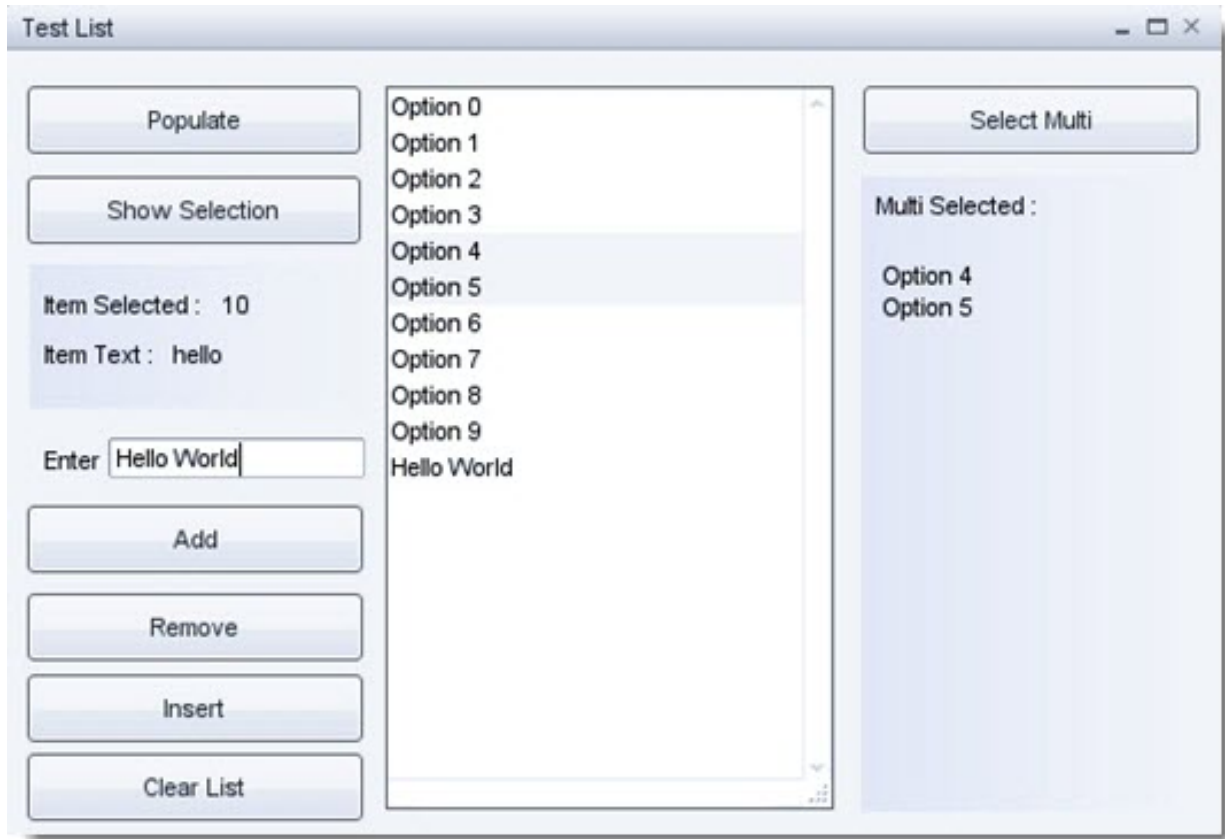
## 16.4.5 Conclusion

In this tutorial, you learned the following concepts:

- Display text on the Game screen
- Show how to detect a mouse click
- Simple use of the schedule timer
- Show game fps and poly count
- Locate and Edit the playGui

## 16.5 Advanced Dialogs

### 16.5.1 Introduction



This project will cover the list control, how to add, remove and insert items in to the list. We will make an interactive demo showing how and where to put the scripts for this project to function. Find out how to make selections, multi-selections and display the results.

Suggested Reading:

- GUI Editor Overview
- GUI Interface
- TorqueScript Reference
- GUI Tutorial 1 (Creating a New GUI)
- GUI Tutorial 2 (Adding Controls)
- GUI Tutorial 3 (Mini Console Tutorial)
- GUI Tutorial 4 (Simple HUD)

Covered in this tutorial:

- Introduce the List Control
- Single and multi-selection
- How to populate your list

- Adding an Item
- Removing an Item
- Inserting an Item

## 16.5.2 Setting Up

Open your tutorial project and start the GUI Editor. Create a new Gui called **testList** using the GUI Class type **GuiControl**. From the controls library add a **GuiWindowCtrl** to the editor work space. Resize so as to fit a few buttons and a list similar to the above image.

GuiWindow property text: **My List**

Now we are ready to add our controls to the new window.

## 16.5.3 Adding Controls

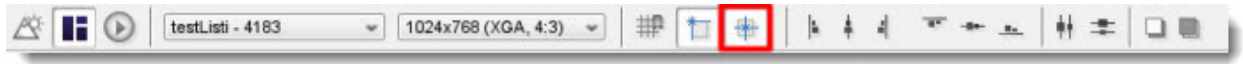
**STEP 1:** First we will add our buttons to the window. Select the **GuiWindowCtrl** control and add a **GuiButtonCtrl** from the library. Reduce its width a little, then copy / paste another 6 copies and move them so that you have something similar to the following image.

**Hint:** **GuiButtonCtrl** is located in the **Library > Buttons** category.



Remember to use the toolbar align tools to help keep you GUI layout uniform and neat.

**HINT:** You may want to turn on Toggle Center Smart Snapping from the tool bar to help align the controls.



**STEP 2:** Next we need to set the button properties;

**button 1** property text: **\*\*Populate\***

**button 2** property text: **Show Selection**

**button 3** property text: **Add**

**button 4** property text: **Remove**

**button 5** property text: **Insert**

**button 6** property text: **Clear List**

**button 7** property text: **Select Multi**

Save your new GUI to **game/art/gui** as **testList.gui**

**STEP 3:** Next we need to add 2 **GuiPanelCtrl** controls and a few **GuiTextCtrl** controls. Set new controls' properties as follows:

**text 1** property text: **Item Selected** **text 2** property text: **Item Text**

**text 3** property text: **00** **text 3** property name: **lblItemSelected**

**text 4** property text: **00** **text 4** property name: **lblItemText**

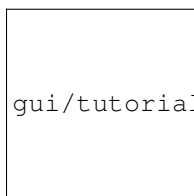
**text 5** property text: **Multi Selected**

How it looks so far:



**Remember to save often.**

**STEP 4:** Now to add the last few controls. From the Library, create a **GuiScrollCtrl** and place it in the center space. Add another **GuiTextCtrl**, a **GuiTextEditCtrl** under the item Text section and a **GuiMLTextCtrl** under the Multi Selected text.



gui/tutorials/5\_MLTextCtrl.jpg

**Properties:**

**GuiTextCtrl** property text: **Enter**

**GuiTextEditCtrl** property name: **txtEnter**

**GuiMLTextCtrl** property name: **lblMLSelected**

**STEP 5:** Select the **GuiScrollCtrl** so that it is highlighted and add a **GuiListBoxCtrl** to the scroll control, this scroll control must become the container for the List box.

**GuiListBoxCtrl** property name: **lstTestList**

That takes care of all our controls. Save your GUI and close down the project. Time to get into some script.



## 16.5.4 Adding Functionality

**STEP 1:** Create a new script file in game/scripts/gui and name it testList.cs and open it in your favorite script editor.

NOTE: The list control is indexed from 0 for the first entry.

The list control exposes a few methods for us to use in populating and controlling list content. We are going to use the following:

- `GuiListBoxCtrl.addItem( itemContent )`
- `GuiListBoxCtrl.deleteItem( itemNumber )`
- `GuiListBoxCtrl.insertItem( itemContent, itemNumber )`
- `GuiListBoxCtrl.getItemText( itemNumber )`
- `GuiListBoxCtrl.clearItems()`
- `GuiListBoxCtrl.getSelCount()`
- `GuiListBoxCtrl.getSelectedItem()`
- `GuiListBoxCtrl.getSelectedItems()`

Copy the following script to your new testList.cs:

```
function testList::addItem()
{
    lstTestList.addItem(txtEnter.getValue());
}

//Insert An Item at the selection
function testList::insertItem()
{
    lstTestList.insertItem(txtEnter.getValue(),lstTestList.getSelectedItem());
}

//Remove a selected Item
function testList::removeItem()
{
    lstTestList.deleteItem(lstTestList.getSelectedItem());
}

//Fill list with content
function testList::populate()
{
    for(%i = 0;%i < 10;%i++)
    {
        lstTestList.addItem( "Option " @%i);
    }
}

//show selected content
function testList::getSelectedContent()
{
    %item = lstTestList.getSelectedItem();
    lblItemSelected.setValue(%item );
    lblItemText.setValue(lstTestList.getItemText(%item ));
}

//Clear the list of items
```

(continues on next page)

(continued from previous page)

```
function testList::clearList()
{
    lstTestList.clearItems();
}

//Display multiselectd items
function testList::multiSelect()
{
    //number of selected items
    %count = lstTestList.getSelCount();

    //returns a space delimited list of all the selected items indexes in the list
    %options = lstTestList.getSelectedItems();

    // parse selected items list
    for(%item = 0;%item <%count;%item++)
    {
        %option = getWord(%options,%item);
        %t = lstTestList.getItemText(%option);
        %text =%text @%t @ "\n";
        lblMLSelected.setValue(%text );
    }
}
```

**STEP 2:** Next we need to add our new script to the engine, open the `game/scripts/client/init.cs` and add the following under the section named `// Execute the GUI scripts and functions`:

```
exec("scripts/gui/testList.cs");
```

We also need to add our gui to this file. Under `// Load up the shell GUIs` add the following:

```
exec("art/gui/testList.gui");
```

Save of your files and run your project. Open the `testList` GUI once again.

### 16.5.5 Scripting Hooks

For the final part of this tutorial we need to add the ability to call our new functions from the respective buttons. Set the buttons properties as follows:

button “Populate” property Command: `testList.populate()`;

button “Show Selection” property Command: `testList.getSelectedContent()`;

button “Add” property Command: `testList.addItem()`;

button “Remove” property Command: `testList.removeItem()`;

button “Insert” property Command: `testList.insertItem()`;

button “Clear List” property Command: `testList.clearList()`;

button “Select Multi” property Command: `testList.multiSelect()`;

Remember to save your GUI. Preview your GUI and try out the buttons, enter some text into the text edit box and try add, insert etc. To multi-select hold down shift while selecting list items, then press the Select Multi button.

## 16.5.6 Conclusion

In this tutorial, you learned the following concepts:

- Introduce the List Control
- Single and multi-selection
- How to populate your list
- Adding an Item
- Removing an Item
- Inserting an Item

In the next tutorial we will create an advanced graphical representation of a GUI.

## 16.6 Simple Inventory GUI

### 16.6.1 Introduction



For the final tutorial in the series, we will make a simple inventory GUI. We will look at how to setup the graphics for your background and buttons. A small script will be written to change the contents of the list for each of the buttons pressed and set up the title text.

Suggested Reading:

- GUI Editor Overview
- GUI Interface
- TorqueScript Reference
- GUI Tutorial 1 (Creating a New GUI)
- GUI Tutorial 2 (Adding Controls)
- GUI Tutorial 3 (Mini Console Tutorial)
- GUI Tutorial 4 (Simple HUD)
- GUI Tutorial 5 (Advanced Dialogs)

Covered in this tutorial:

- Learn how to implement a background image
- Make an image into a button.
- Lists in use

## 16.6.2 Setting Up

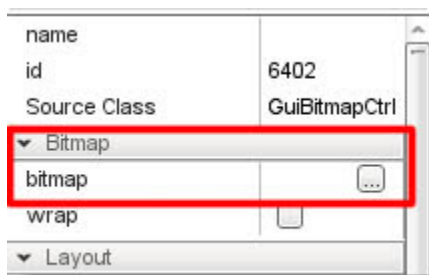
To complete this tutorial you will need some image files. You can either create your own or use the images from this tutorial:

Get the Tutorial 6 Images here: ([gui/tutorials/Tutorial6\\_images.zip](#))

If you have downloaded the **Tutorial6\_images.zip** please place the new folder “images” into **game/art/**. Open your tutorial project and start the GUI Editor. Create a new GUI and call it **InventoryGui** of GUI class type **GuiControl**.

## 16.6.3 Bitmap Controls

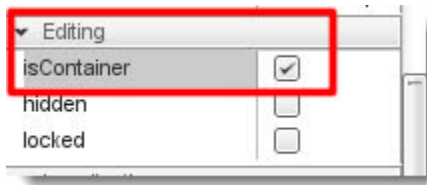
**STEP 1:** From the **Library > Images** category and add a **GuiBitmapCtrl** to the editor work space. Drag the new control into the center of your editor. From within the **GuiBitmapCtrl** section of the properties panel you will notice the square box to the right of the bitmap property.



When clicked, this box will open a file browser. Locate your **game/art/images** folder and load the file *backgnd.png*.

GuiBitmapCtrl Property bitmap: `art/images/backgnd.png`

While we are also in the property section, we will need to set this control as a container. By being a container this control can become a parent control for the buttons and text controls of our inventor gui:



**STEP 2:** You will need to resize the bitmap control by dragging on the sizing handles, do so until you have something looking similar to this:



**NOTE:** The Close button is embedded into the background image.

Now would be a good time to make a save if you have not done so already. Save the GUI as **inventoryGui.gui**.

**STEP 3:** Making sure you have the **GuiBitmapCtrl** selected, add a **GuiTextCtrl** from the **Library > Text** category to act as our inventory title. Place this in the space at the top of our inventory, stretch it, and set its properties to the following:

**GuiTextCtrl** property name: **lblInvTitle**

**GuiTextCtrl** property text: **Inventory**

You will notice that the text is very hard to read. We are going to adjust its color with a control profile. Instead of choosing one from the many provided, this time we are going to make our own.

## 16.6.4 GUI Profile

Setting up your own profile is very easy. Head to the folder **game/art/gui** and open the file **defaultGameProfiles.cs** in Torsion or another text editor.

If **defaultGameProfiles.cs** does not exist in your project you can add the following code to the end of **art/gui/gameProfiles.cs** and then at the top of **scripts/client.cs** add the line:

```
exec("~/art/gui/gameProfiles.cs");
```

To make our Inventory look good we are going to create three new profiles. In the **defaultGameProfiles.cs** add the following script at the bottom of the file:

```
singleton GuiControlProfile (InvList)
{
    opaque = true;
    fontType      = "Arial";
    fontSize      = 16;
    fillColor     = "150 150 158"; //selection color
    fontColor     = "255 255 255";
    justify       = "left";
};

singleton GuiControlProfile (InvScroll)
{
    // make transparent
    opaque = false;
};

singleton GuiControlProfile (InvGui)
{
    // make transparent
    opaque = false;

    //Font
    fontType      = "Arial";
    fontSize      = 18;

    //Set font color - R G B (range 0 -255 )
    fontColor     = "200 200 200";
    justify       = "center";

    //Draw a border
    border        = 1;
    border        = false;
};
```

You will notice the new profile names that will be displayed in the GUI controls profile property:

```
singleton GuiControlProfile( profile name ){ ... }
```

*Hint: When naming variables, controls, profiles, etc., try to be descriptive of its purpose. You can only use the label once otherwise you will have errors due to conflicting naming. For example, calling our profile **inventoryGui** would*



*have caused a conflict with the name of the GUI itself.*

To give you an idea of which properties are available, here is a table with explanations:

Note: certain GuiControl classes use this bitmap `[[#Parsing_the_Bitmap]` parsed into multiple pieces. |

`border` | integer | 1 | For most controls, if border is greater than 0, a border will be drawn. | Some controls use this member to draw different types of borders. |

`borderThickness` | integer | 1 | Thickness of the control's border. |

`borderColor` | Color4I | 040 040 040 100 | The color to use for the border of the control. |

`borderColorHL` | Color4I | 128 128 128 255 | The color to use for the border of the control when the control is highlighted. |

`borderColorNA` | Color4I | 064 064 064 255 | The color to use for the border of the control when the control is not active. |

`canKeyFocus` | bool | false | Whether this control can become the focus for keyboard events (key presses). |

`cursorColor` | Color4I | 000 000 000 255 | The color of the insertion point (blinking I-beam) cursor in an EditText control. |

`fillColor` | Color4I | 211 211 211 255 | The color for the interior of the control. |

`fillColorHL` | Color4I | 244 244 244 255 | The color for the interior of the control when the control is highlighted. |

`fillColorNA` | Color4I | 244 244 244 255 | The color for the interior of the control when the control is not active. |

`fontCharset` | enum | ANSI | The output encoding of the font to use. One of ANSI,?? |

`fontSize` | integer | 14 | The size of the font in points. |

`fontType` | string | Arial | The name of the font, along with other modifiers, like "bold". |

`fontColor` | Color4I | 000 000 000 255 | Color of the font. |

`fontColors` | Color4I | 000 000 000 255 | Unknown. |

`fontColorHL` | Color4I | 032 100 100 255 | Color of the font when the control is highlighted. |

`fontColorSEL` | Color4I | 200 200 200 255 | Color of the font when the control or the text field is selected. |

`fontColorNA` | Color4I | 000 000 000 255 | Color of the font when the control is not active. |

`fontColorLink` | Color4I | 000 000 000 000 | Font color for a hyperlink. |

`fontColorLinkHL` | Color4I | 000 000 000 000 | Font color for a highlighted hyperlink. |

`fontColors_0..9` | Color4I | 000 000 000 255 | Different members of this array of font colors is devoted to particular GuiControl classes. | GuiMLTextProfile, GuiConsoleProfile, and others. |

`justify` | left, center, right | right | Justification of the text of the control. |

`textOffset` | Vector2I | 0 0 | Offset, in points, of the text of the control. |

`Modal` | bool | false | Whether the control should make the UI modal (prevent the user from doing anything outside of the control). |

`mouseOverSelected` | bool | false | Whether the control should be "selected" when the mouse hovers over it. |

`numbersOnly` | bool | false | Whether the text of the control should be restricted to numerical characters only. |

`opaque` | bool | false | Whether the control should be opaque. |

`tab` | bool | false | Whether the user can switch focus to this object by using the tab key. |

`returnTab` | bool | false | Whether a tab-event should be simulated when the return key is pressed. (Used in the EditTextEdit profile.) |

`profileForChildren` | object | none | When the control is used as a container for other controls, this field specifies the profile to use for those child controls. | Profiles that specify this field: EditorListBoxProfile EditorPopupMenu EditorPopupMenuLarge | EditorTabBook GuiFormProfile GuiPopUpMenuDefault GuiPopUpMenuEditProfile T2DDatablockDropDownProfile. |

`soundButtonDown` | AudioProfile | none | Sound profile to use for the sound to produce when the button (control) is pressed. |

`soundButtonOver` | AudioProfile | none | Sound profile to use for the sound to produce when the mouse hovers the

control. |

### 16.6.5 More Visual Editing

Open your project up again and load the **inventoryGui.gui** file. Select the title control **GuiTextCtrl** set its profile property:

**GuiTextCtrl** Property profile: **InvGui**

The **GuiTextCtrl** now uses the profile we created in script. By changing the profile the title text can now be seen. Before we create our bitmap buttons we need to add scroll and list controls to display our items.



With the **GuiBitmapCtrl** selected, add a **GuiScrollCtrl** control to the large pane. With the **GuiScrollCtrl** selected, add a **GuiListBoxCtrl**. The control list should look like this:



**Hint:** **GuiScrollCtrl** can be found in the **Library > Containers** category, **GuiListBoxCtrl** can be found in the **Library > Lists** category.

Next we need to change a few properties to get it all looking nice:

**GuiScrollCtrl** property profile: **InvScroll**

**GuiListBoxCtrl** property name: **lstInventory**

**GuiListBoxCtrl** property profile: **InvList**

**GuiListBoxCtrl** property fitParentWidth: **checked**

Now is a good time to save.

### 16.6.6 Bitmap Buttons

Before we add functionality via script we need to add one last set of controls: our bitmap buttons.

**STEP 1:** With the **GuiBitmapCtrl** control selected add a new **GuiBitmapButtonCtrl** and place it in the left hand panel of our GUI.

**Hint:** **GuiBitmapButtonCtrl** can be found in the **Library > Buttons** category.

Set the new controls property: (remember you can click the file box to access the browser)

**GuiBitmapButtonCtrl** property Bitmap: **game/art/images/aRaLogoIcon.png**

Resize the control as in the image below:



Press F10 to preview your GUI and try the button.

**STEP 2:** The button does not work! This is because we have specified that the control has a single image for all of its states, thus no change when it is clicked. To remedy this ensure the property looks like the following:

**GuiBitmapButtonCtrl** property Bitmap: **game/art/images/aRaLogoIcon** ( notice there is no file extension )

Now a little note on the bitmap button control. You may have noticed that the image was entered as **game/art/images/aRaLogoIcon** with no extension ( \*.png ). This is very important and not an error, but part of the button multi image system.

Each image needs to be named in the correct format for the button system to recognize the image as part of a multiple image set. Have a look at the images in the game/art/gui/images folder.

**Normal** - aRaLogoIcon.png

**Hover** - aRaLogoIcon\_h.png

**Down** - aRaLogoIcon\_d.png

If you want to have multiple images for normal / hover / down, you just add the file path and the file name excluding the file extension ( \*.png ). The engine will look for the correct images. Save and preview (F10) your GUI to try out the button.

**STEP 3:** Next, select the new button and copy / paste three more. Arrange the buttons under each other.

**HINT:** If you select multiple controls you can then resize them all at the same time.

**HINT:** Remember you can use the icons in the tool bar to **line up** and **distribute** the controls evenly.

**HINT:** The tool bar icons will display a tool hint if you hover the mouse over them.

Set the properties of the new buttons:

**Button 1** property name: **btn1**

**Button 2** property name: **btn2**

**Button 3** property name: **btn3**

**Button 4** property name: **btn4**



**STEP 4:** The last control we need is the close button at the bottom of the inventory. With the `GuiBitmapCtrl` selected add one more `GuiBitmapButtonCtrl`. Set its properties to the following:

Button close property name: **btnClose**

Button close property Bitmap: **game/art/images/inventoryClose** (again leave off the file extension)

Resize the control so that it fits neatly over the background image of the close button. Use F10 to preview your GUI to make sure all the button images change with the mouse events.

### 16.6.7 Adding Functionality

Now that all of the controls are in place, we need to make it all do something. Making sure you have saved your GUI, close down Torque 3D and open your favorite script editor. Create a new script file in **scripts/gui/** called



**InventoryGui.cs.**

The new script will add functionality to the buttons which, when pressed, will change the text in the title and place the contents of an array into the list box. Add the following code to your new script file:

```
//
//global arrays for initial content to be displayed
//

$FOOD      = 0;
$SPELLS    = 1;
$WEAPON    = 2;
$ARMOUR    = 3;

$aInv[$FOOD,0] = "Bread x 1";
$aInv[$FOOD,1] = "Apple x 1 ";
$aInv[$FOOD,2] = "Pie x 2";

$aInv[$SPELLS,0] = "Fall From Grace";
$aInv[$SPELLS,1] = "Ice Call";
$aInv[$SPELLS,2] = "Water Wish";
$aInv[$SPELLS,3] = "Fire Storm";
$aInv[$SPELLS,4] = "Healing Heart";

$aInv[$WEAPON,0] = "Sword of Truth";
$aInv[$WEAPON,1] = "Chain Axe";
$aInv[$WEAPON,2] = "Dagger";
$aInv[$WEAPON,3] = "Elf Staff";
$aInv[$WEAPON,4] = "Ork Hammer";

$aInv[$ARMOUR,0] = "Light Mail";
$aInv[$ARMOUR,1] = "Light Shield";
$aInv[$ARMOUR,2] = "Cursed Gloves";
$aInv[$ARMOUR,3] = "Invisiblity Cloak";

//
//Give each of our buttons a function to call
//

function inventoryGui::btn1()
{
    //set the title text
    lblInvTitle.setValue("FOOD");

    //clear the list box of previous content
    lstInventory.clearItems();

    //iterate through our array adding items to our list
    for(%i = 0;%i < 4;%i++)
    {
        lstInventory.addItem( $aInv[ $FOOD,%i ] );
    }
}

function inventoryGui::btn2()
{
    lblInvTitle.setValue("SPELLS");
    lstInventory.clearItems();
```

(continues on next page)

(continued from previous page)

```

        for(%i = 0;%i < 5;%i++)
        {
            lstInventory.addItem( $aInv[ $SPELLS,%i ]);
        }
    }

function inventoryGui::btn3()
{
    lblInvTitle.setValue("WEAPONS");
    lstInventory.clearItems();
    for(%i = 0;%i < 5;%i++)
    {
        lstInventory.addItem( $aInv[ $WEAPON,%i ]);
    }
}

function inventoryGui::btn4()
{
    lblInvTitle.setValue("ARMOUR");
    lstInventory.clearItems();
    for(%i = 0;%i < 4;%i++)
    {
        lstInventory.addItem( $aInv[ $ARMOUR,%i ]);
    }
}

```

Next, open the **game/scripts/client/init.cs** file and add the following under the section *// Execute the GUI scripts and functions*:

```
exec("scripts/gui/InventoryGui.cs");
```

**Save your script files now.**

## 16.6.8 Scripting Hooks

We are going to make our final edits in the editor. Run the GUI Editor and load your inventoryGui. Each button now needs to be linked to our new functions. Set the following properties to our buttons:

**btn1** property command: **inventoryGui.btn1()**;

**btn2** property command: **inventoryGui.btn2()**;

**btn3** property command: **inventoryGui.btn3()**;

**btn4** property command: **inventoryGui.btn4()**;

**btnClose** property command: **Canvas.popDialog(inventoryGui);**

Save your final edits and test out your new inventory system.

## 16.6.9 Conclusion

In this tutorial, you learned the following concepts:

- Learn how to implement a background image
- Make an image into a button.

- Lists in use

This brings the GUI Editor tutorial series to a close. You should now have enough information to begin constructing your own custom GUIs, specific to your game. You can download the images, GUIs, and scripts created for these tutorials by [CLICKING HERE](#). (gui/tutorials/zip/GUITutorial\_Series\_One.zip)

Special thanks to Dave Young (aka NightHawk) of aRa Software Tutorials for writing the original guides.

### File List

- **game/art/**
  - HelloWorld.gui
  - InventoryGui.gui
  - miniConsole.gui
  - newWindow.gui
  - playGui.gui
  - testList.gui
  - testWindow.gui
  - gameProfile.cs
  - Torque-3D-logo.png
- **game/art/images/**
  - araLogoIcon.png
  - araLogoIcon\_d.png
  - araLogoIcon\_h.png
  - backgnd.png
  - inventoryClose.png
  - inventoryClose\_d.png
  - inventoryClose\_h.png
- **game/scripts/client/**
  - default.bind.cs
  - init.cs
- **game/scripts/gui**
  - InventoryGui.cs
  - playGui.cs
  - testList.cs
  - testWindow.cs

## 17.1 Torque Art Primer - TODO

### 17.1.1 Introduction

File Formats

Coordinate System

Normal/Bump Maps

### 17.1.2 3-Space Features

COLLADA and DTS

Level-of-Detail (LOD)

Bounds

Collision Geometry

Billboards

Imposters

Mounting

### 17.1.3 Animation Concepts

Threads

Ground Transforms

Triggers

Blends

## 17.1.4 Materials

Material Mapping

Material and Skin Swapping

Skinning

Animated Materials

UV animation

Image Sequence animation

## 17.1.5 Conclusion

# 17.2 Torque Character Primer

## 17.2.1 Introduction

### Character Setup Pipeline

Characters in T3D can be setup to use different weapon animations and share those animations between different skinned meshes with the same skeleton hierarchy. So a standard T3D character would have COLLADA files for the character's skinned mesh and skinned skeleton as well as for the character's animations with just the skeleton for each weapon pose. The animations can be exported individually or combined in one .dae file that is split up through the shape editor.

### Third Person Weapon Player Animation Names

Back: (Loops)	Run backward
Crouch_Back: (Loops)	Crouch walk backward
Crouch_Forward: (Loops)	Crouch walk forward
Crouch_Root: (Loops)	Crouch idle
Crouch_Side: (Loops)	Crouch move right
Death#:	Where ‘#’ is can be a number for multiple death sequences that will be picked randomly. So Death1, Death2, etc.
Fall: (Loops)	Character is falling
Head: (Blend)	Usually a 9 frame animation that only affects the neck and head and works in conjunction with the “Look” animation. Frame 1 is looking straight up, frame 5 is looking straight forward and 9 is looking down
Jump:	Character jumping
Land:	Character landing
Look: (Blend)	Usually a 9 frame animation that only affects the spine and works in conjunction with the “Head” animation. Frame 1 is looking straight up, frame 5 is looking straight forward and 9 is looking down
Reload: (Blend)	Reloading the weapon
Root: (Loops)	More commonly known as the idle animation in most parts of the industry. Just the character standing and breathing
Run: (Loops)	Character running
Side: (Loops)	Character side stepping to the right. This animation will be played in reverse when moving to the left
Sitting: (Loops)	Character sitting in a vehicle
Swim_Back: (Loops)	Swimming backward
Swim_Forward: (Loops)	Swimming forward
Swim_Idle: (Loops)	Treading water
Swim_Left: (Loops)	swimming left
Swim_Right: (Loops)	Swimming right

After that you add the animations through the shape editor by going to sequence tab (labeled “Seq”). Click on the new sequence icon and a file browsing dialog will open. Select the sequence COLLADA file you want. Now define the time range that you want by changing the numbers at the beginning and end of the timeline. Complete this process for each sequence that you wish to add.

Optionally you can define an object called “BOUNDS” that can be used to define the object’s origin (as opposed to your 3d content application’s origin) and is used to define the speed that the object is intended to be moving at. Such as during a run sequence, if you had a character running forward at 1 meter a second in your 3d application’s scene and had the “BOUNDS” object follow your character then when you bring it into Torque 3D and assigned to the player’s run animation then the playback speed of the animation can be adjusted depending on the speed that the



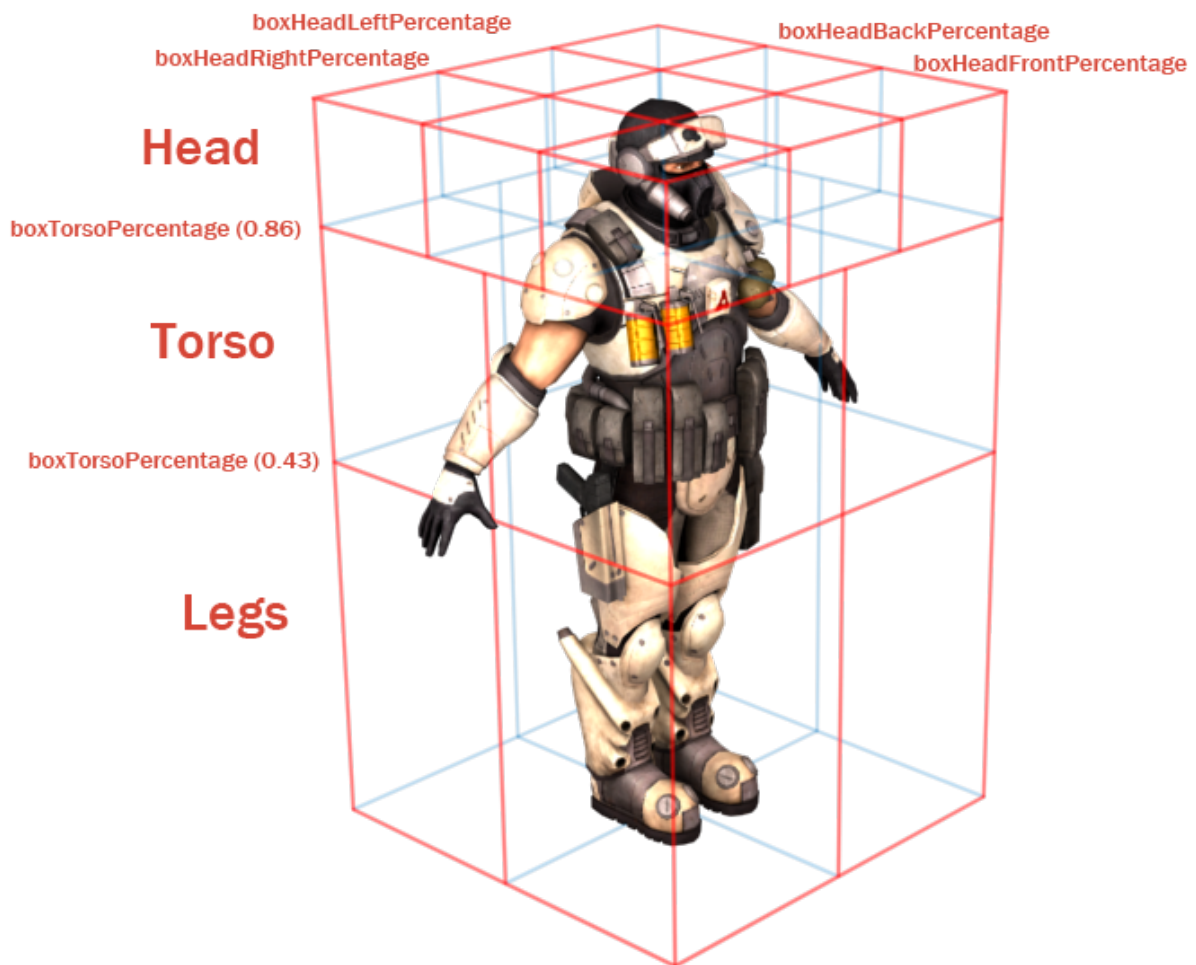
player is moving through the game world. What this means is that animations don't have any sliding issues caused by the character's in game speed not matching with the designed animation speed.

### Hitboxes and Damage Location specifications

Currently, the player's hitbox defined by their bounding box. In order to get damage locations we have cut the player's world box up into pieces as defined by the following sections in the Player's datablock:

- boundingBox
- boxHeadPercentage
- boxTorsoPercentage
- boxHeadLeftPercentage
- boxHeadRightPercentage
- boxHeadBackPercentage
- boxHeadFrontPercentage
- boxHeadFrontPercentage

The player's boundingBox determines the length in each dimension the bounding box should encompass. From the standard player datablock, its sections would look like the following:



It may be easiest to come up with these numbers by taking a render of the player, and using an imaging program to determine what percentage of the player makes up their legs/head/torso.

In order to take advantage of these damage locations we use the function `Player::getDamageLocation()`. One of the best places to call this is from the `Projectile::onCollision` or `Armor::Damage()` as we'll have the position of the projectile, the player to call `getDamageLocation()` and a good place to modify the damage if we wanted to do extra damage on a headshot or less damage to the legs.

`GetDamageLocation()` will return a string using the defined boundingbox percentages from the player datablock. In C++ it more or less transforms the projectile's location from world space to object space, multiplies the player's bounding box by its percentages, then checks to see if the hit location is greater than, or less than the bounding box dimension multiplied by the percentage. For example, if the bounding box's dimension was 1, it will multiply 1 by 0.43, and check if the bullet's location is less than or equal to this value. If it's less than the torso, then it counts as a leg shot. The system then does the same test in the other dimensions to see if the front/back or left/right was hit.

The string returned will be one of the following:

Possible locations:

- legs
- torso
- head

Head modifiers:

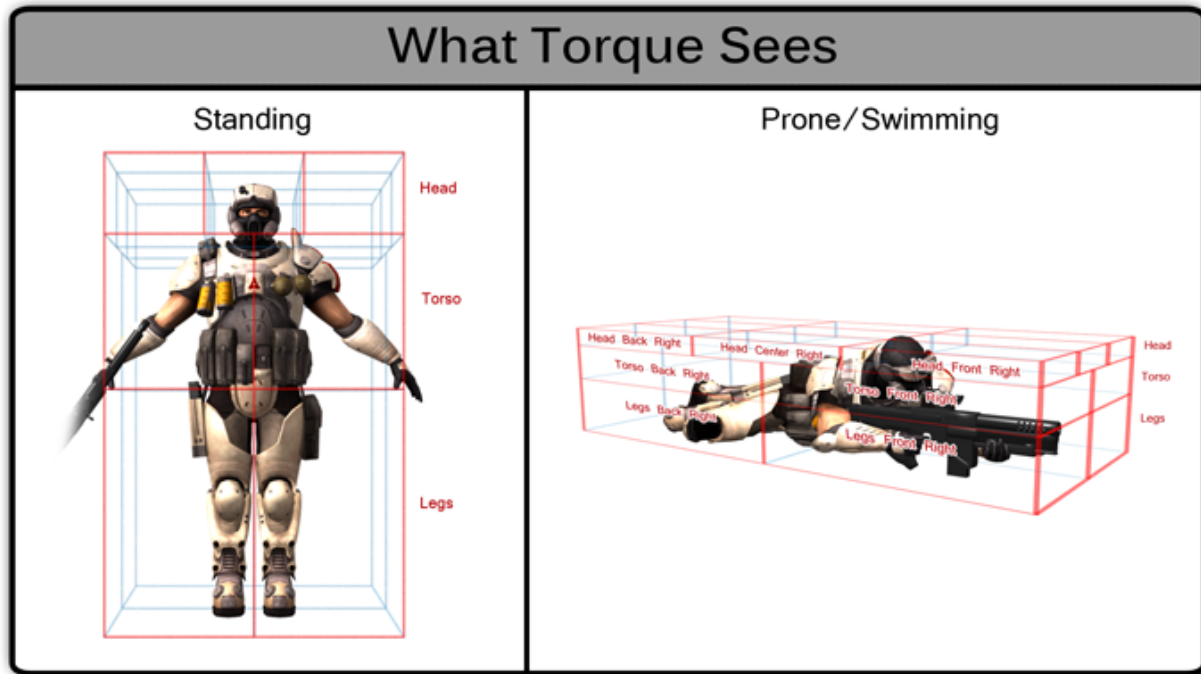
- left\_back
- middle\_back
- right\_back
- left\_middle
- middle\_middle
- right\_middle
- left\_front
- middle\_front
- right\_front

Legs/Torso modifiers:

- front\_left
- front\_right
- back\_left
- back\_right

For example, a perfect headshot would be "head\_middle\_front", head\_middle\_back, or "head\_middle\_middle". A shot to the front left leg would be "legs\_left\_front".

As we can see in the picture, there are situations where we can register a hit from the headbox that actually wouldn't "hit" the player. Such as shooting a bullet in the `boxHeadRightPercentage` area from the front of the player would fly over the player shoulder, but register a hit in the engine. It may be necessary to do some math to see if the bullet will actually pass through the center of the player's head box to get realistic results.



## 17.2.2 Conclusion

This article has described the basic animation setup for a custom character's standard "third person" rig. With this information you will be able to animate a custom character so that it will interact with the standard Torque 3D character animation system. Additionally, we've covered the requisite information for reading basic hit locations from a Torque 3D player.



## CHAPTER 18

---

### Formats

---

#### 18.1 COLLADA - TODO

##### 18.1.1 Introduction

##### 18.1.2 COLLADA for Torque 3D

##### 18.1.3 Troubleshooting

##### 18.1.4 Appendix 1: Material Settings

##### 18.1.5 Appendix 2: Supported Extensions

##### 18.1.6 Appendix 3: Supported COLLADA Elements

#### 18.2 DTS Format - TODO

##### 18.2.1 Introduction

##### 18.2.2 DTS Format

##### 18.2.3 Sequences

##### 18.2.4 Bitsets

##### 18.2.5 Data Buffers

##### 18.2.6 Meshes

#### 18.3 DSQ Format - TODO

##### 18.3.1 Introduction

##### 18.3.2 DSQ Format

## 19.1 DAE to DTS

The `dae2dts` tool is a command-line application used to convert COLLADA (DAE) models to Torque shape (DTS) or animation (DSQ) files. The tool is invoked from the windows command prompt as follows:

```
dae2dts [options] daeFilename
```

Where `daeFilename` is the path to the COLLADA (.dae) file to convert (the DAE file does not need to be in the same folder as the `dae2dts` tool). The tool exit code is zero on success or non-zero on failure, making it suitable for use within a larger batch conversion process. The following options are available:

- **-config `cfgFilename`** Set the conversion configuration filename. Reserved for future use; not currently supported.
- **-output `dtsFilename`** Set the output DTS filename. If not specified, the output will have the same base name as the input DAE file, but with DTS (or DSQ) extension.
- **-dsq** If set, all sequences in the shape will be saved to DSQ files (one for each sequence) instead of being embedded in the DTS file. The generated DTS file will not contain any animation sequences.
- **-dsq-only** Same as `-dsq`, but no DTS file will be saved (handy for animation only input files).
- **-compat** If set, the tool will attempt to write to DTS v24 for compatibility with non-T3D engines (TGE, TGEA, ShowToolPro etc). By default, the `dae2dts` tool outputs DTS v26 files which can only be loaded in T3D based products (T3D can also load DTS v24 files). DTS v24 supports only a single UV set, around 11000 maximum triangles per mesh, and does not support vertex colors. Also, COLLADA models that use autobillboards may not be saved correctly to DTS v24.
- **-diffuse** If set, the tool will use each material's diffuse texture as the material name (instead of the COLLADA `<material>` name) for compatibility with simple TGE materials.
- **-materials** If set, the tool will generate a `materials.cs` script in the output folder to define Materials used in the shape.
- **-verbose** If set, the tool will output progress information



### 19.1.1 Examples

A COLLADA model, `player.dae`, contains 3 animation clips: `root`, `run`, `shoot`:

```
# Convert to 'player.dts' (all animations are embedded in the DTS file)
dae2dts player.dae

# Convert to 'orc.dts', and generate 'orc_root.dsqu', 'orc_run.dsqu' and
# 'orc_shoot.dsqu' (all files compatible with TGE, TGEA and ShowToolPro)
dae2dts --compat --dsq --output orc.dts C:\shapes\player.dae

# Extract animations only and store in 'player_root.dsqu', 'player_run.dsqu' and
# 'player_shoot.dsqu'
dae2dts --dsq-only player.dae
```

### 19.1.2 Materias

A major difference between a COLLADA and a DTS model is in the naming of materials. DTS files only support a single material name, which historically (TGE) has been used to reference the diffuse texture. For example, a DTS material that uses the texture `wood.jpg` would be called `wood`. When Torque loads this model, if there are no script Materials defined that mapTo `wood`, the engine looks for a JPG, PNG, or BMP file with that name in the same directory as the model.

However, in a modelling application (and in Torque3D!), a material encompasses much more than just the diffuse texture. When a model is exported to the COLLADA file format, the name of the material in the modelling app is stored in the `<material>` tag in the COLLADA file.

The default behavior of the `dae2dts` tool is to use the COLLADA `<material>` element name as the material name in the DTS file, which means the model will not display correctly in Torque unless `materials.cs` has been setup correctly. There are two approaches to solve this issue for `dae2dts` converted models:

1. Name the material in the 3D app the same as the diffuse texture file, or use the `-diffuse` option to do it automatically at conversion time. This is the recommended approach for engines that do not support scripted Materials (TGE, ShowToolPro).
2. Create a `materials.cs` file in the same directory as the model (or have the tool create it for you), and define a script Material object that maps to each COLLADA material name. This is the recommended approach if your engine supports scripted Materials as it will give you greater flexibility in naming your materials.

### 19.1.3 Futher Reading

See the *\*Torque COLLADA Loader\**(Artist Guide/Formats/COLLADA) documentation for more details about how Torque loads COLLADA files, and for which COLLADA features are supported.

TSShapeConstructor allows you to modify the loaded COLLADA model in-memory before it is saved to DTS or DSQ. The TSShapeConstructor script can be created by hand, or by using the T3D Shape Editor tool (available in the T3D demo for non-T3D licensees). See TSShapeConstructor in the Torque 3D - Script Manual for more details.

## **19.2 Milkshape - TODO**

### **19.2.1 Introduction**

### **19.2.2 Main Dialog**

### **19.2.3 Meshes**

### **19.2.4 Materials**

### **19.2.5 Sequences**

### **19.2.6 Comment Strings**

### **19.2.7 Additional Information**

### **19.2.8 Change Log**



## 20.1 Adding Objects to level

### 20.1.1 Introduction

3D models, referred to as shapes in these tutorials, make up most of the objects in your game. This includes players, items, weapons, vehicles, props, buildings, and so on. Currently Torque 3D supports three model formats: COLLADA, DTS, and DIF.

**COLLADA:** Short for **C**OLLA**B**orative **D**esign **A**ctivity. COLLADA is emerging as the format for interchanging models between DCC(digital-content-creation) applications. The file format is .dae (**d**igital **a**sset **e**xchange). The data is stored in an open standard XML schema, which means it can be read and tweaked manually if need be.

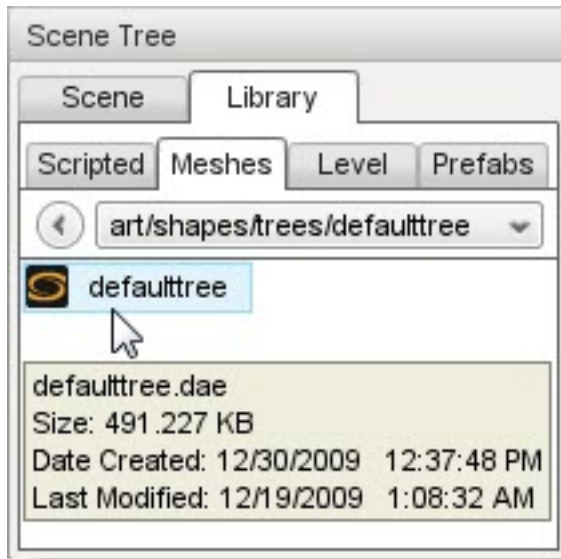
**DTS:** Short for **D**ynamics **F**ree **S**pace, is the native, binary file format used by Torque to store shape (geometry, LOD, bone, and animation) data. DTS exporters exist for several 3D modeling packages such as 3ds Max, Maya, XSI, Blender, and Milkshape3D.

**DIF:** Short for **D**ynamix **I**nterior **F**ile, this is another proprietary format developed during the Tribes days and has survived through Torque. DIFs (also called Interiors) are primarily used for buildings or other enclosing structures. While the binary space partition (BSP) functionality is useful, using DTS or COLLADA files with Polysoup collision enabled is the preferred method and will save you time on asset generation.

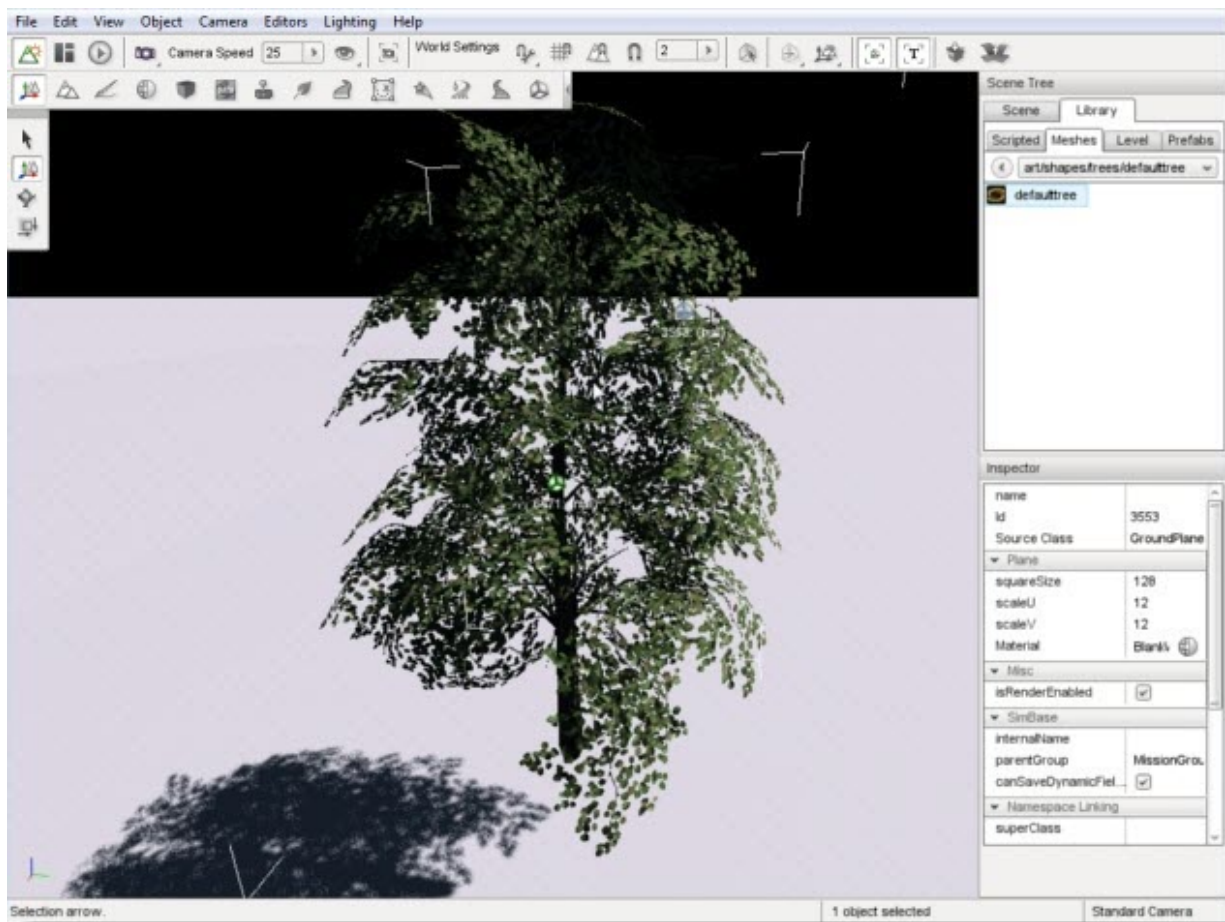
### 20.1.2 Adding A COLLADA Model

The ability to load and render COLLADA models (.dae) is a new Torque 3D feature. The process of adding a COLLADA shape is identical to adding a DTS. You will first need to know where your .dae file is located.

Again, you will open the Library->Meshes tab. Navigate to the directory containing your COLLADA model (.dae). If you hover over the item, you will get a brief file description.



Go ahead and double-click on the object. The file should load extremely fast, but you may not be able to see it right away. Pull your camera up and away from its current location to see the giant shape which has been added.

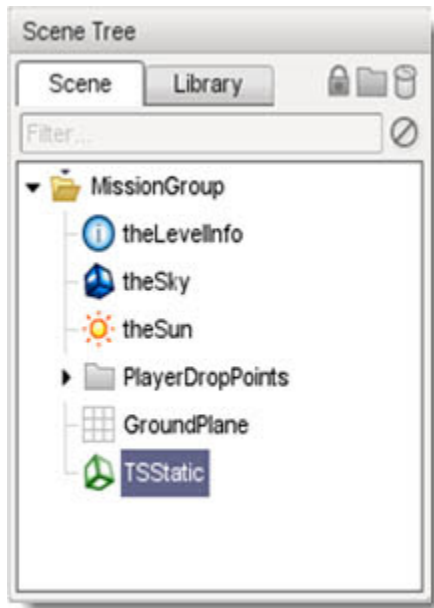


All of the geometry and texture information is readily available in this single format. What's more, nearly every major 3D modeling application is able to export directly to the COLLADA format so you can import similarly complex

objects as you see here.

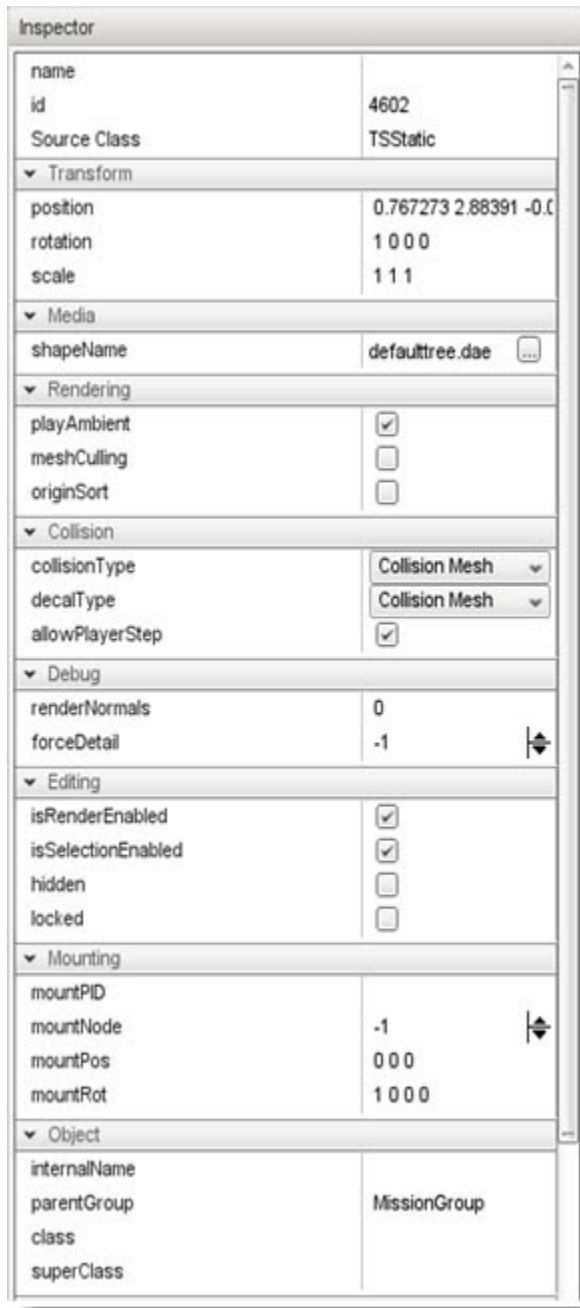
### 20.1.3 Shape Properties

Now that we have added an object to the Scene, lets look at properties that the object has. Click on the Scene tab, then select the TSStatic object.



Object properties are displayed in the pane below:





### 20.1.4 Conclusion

While asset creation is the first step, importing 3D models into your level is a major milestone. The flow goes both ways, however. Once you have added a model to your scene, it can dynamically reflect any changes made to the actual file containing the data. If you re-export an existing COLLADA model from an application, your game object will automatically update to show the changes.

The rest is simply level design. How, where, and why you place objects is up to you.

## 20.2 Adding A New Player - TODO

### 20.2.1 Introduction

One of the first things you will want to do after you have become familiar with T3D is to change the player character model. This can simply be achieved by using Torque 3D's powerful World Editor and its provided game editing tool set.



In this Tutorial we are going to take a step by step look at how to change the default player character from the Gideon model to the BoomBot model. These steps can then be easily applied to your own rigged character model for use in your T3D project. We will explore how to change the players characteristics through the use of the datablock properties and how to spawn this new player into the game , and all this can be done without touching any scripting thanks to T3D's great game editing WYSIWYG tool set.

#### Topics we will look at in this tutorial:

- Create a new Player Datablock
- Changing the player model
- Changing Player properties
- Spawning a new player

### 20.2.2 Setup

We will start off by opening the Deathball Desert Torque 3D project from the provided toolbox, but first I would advise you to back up this example project.

1. The example project is located in the Torque 3D Pro 1.1 Examples folder. Make a backup copy of the FPS Example folder by copying it to a backup folder.
2. Launch the T3D tool box.

## **20.2.3 The Datablock Editor**

### **Player Datablock**

## **20.2.4 Player Properties**

### **Change the Character Model**

## **20.2.5 Spawning**

## **20.2.6 Customizing our Character**

### **Custom 3rd Person Camera**

### **Power up our Character**

### **Activate Weapons**

## **20.2.7 Conclusion**

# **20.3 Terrain - TODO**

## **20.3.1 Introduction**

In this tutorial, we are going to create a lush valley using sample assets provided by Sickhead Games. For this guide, the terrain will be created by importing a heightmap, opacity maps, and creating new materials.

## **20.3.2 Setup**

## **20.3.3 Heightmap, Opacity Layer, Terrain Textures**

## **20.3.4 Importing A Heightmap**

## **20.3.5 Conclusion**

# **20.4 Adding Wind effects**

## **20.4.1 Introduction**

The color of a vertex in a model allows the artist to specify how a plant model will behave once it is brought into Torque 3D. This also covers how a plant will respond to a wind emitter. Here is a breakdown of how the colors affect a model:

1. **Bending of branches:** controlled by the amount of red on a vertex. Normally the ends of branches are fully red, and the spot where they reach the trunk would be fully black (or colorless, depending on how you are painting... see my method below), with a smooth gradation between.
2. **Branch group instancing:** So that groups of branches sway independently, they need to have varying amounts of green. Usually done by selecting a clump of branches/fronds and filling with a random shade of green (anywhere from pure green to black).
3. **Flutter of leaves:** flutter is controlled by the amount of blue on a vertex. Normally, the ends of fronds are fully blue so they flutter fully, but I've found that in a lot of cases (in trees, mostly) I can just fill the entire frond with blue.
4. **Vertical bend:** the vertical bend of a tree does not have to be painted... that is calculated automatically by its height

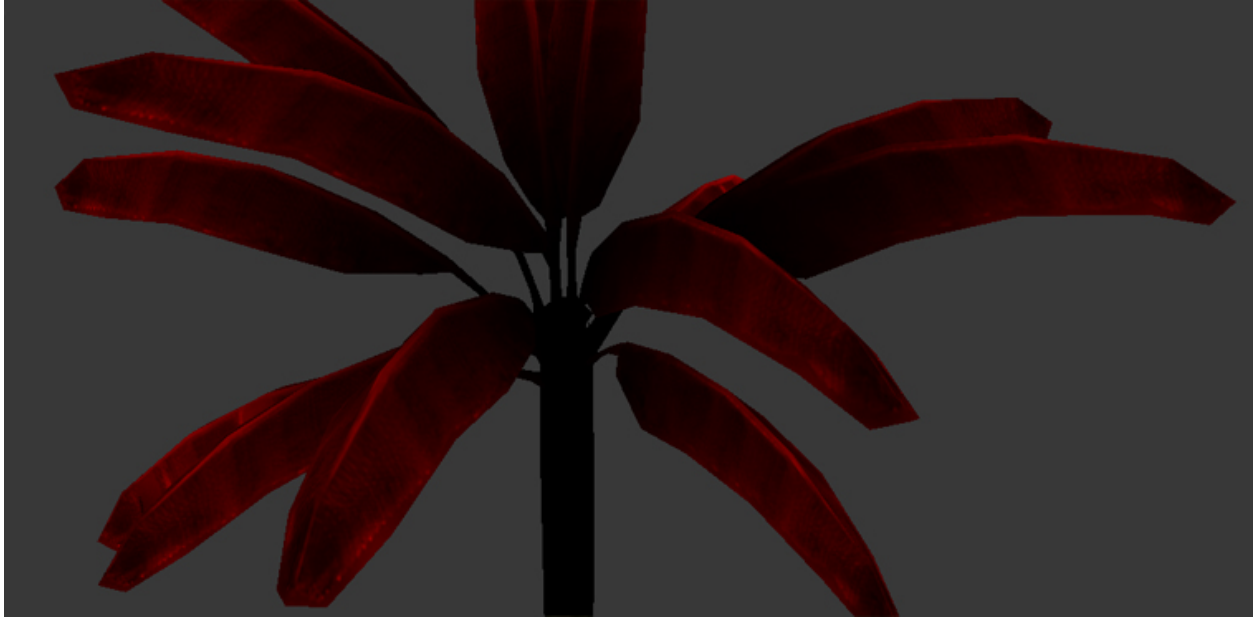
### 20.4.2 Vertex Painting Your Model

The following images represent the tree model with the separate vertex coloring steps:

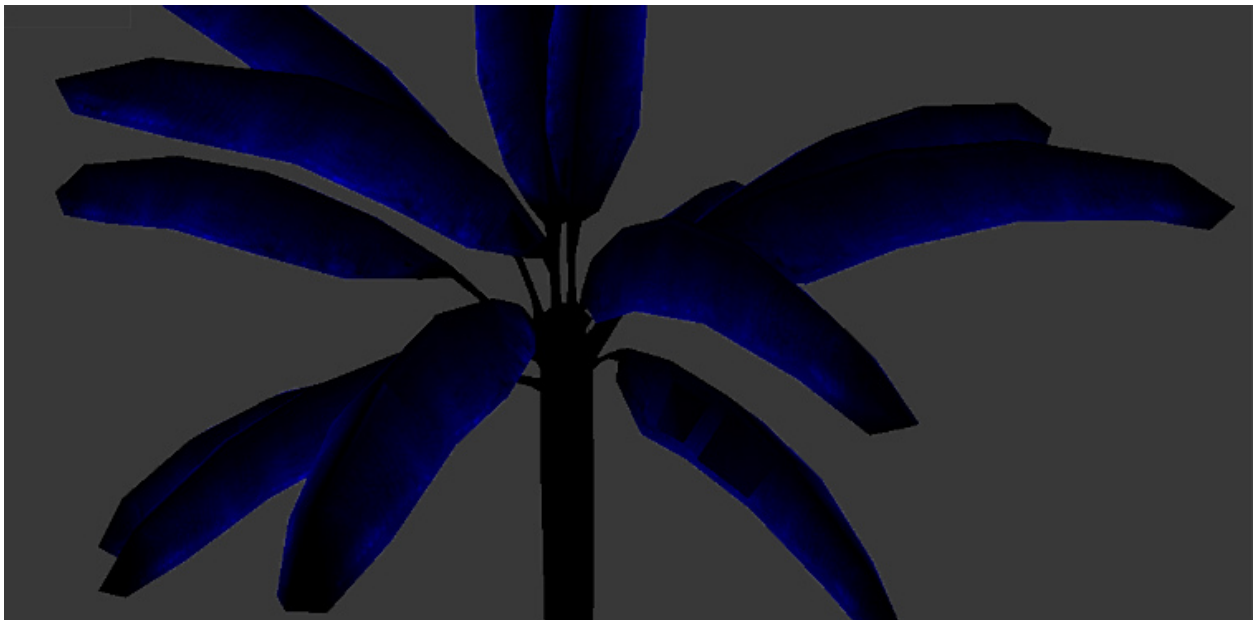
The tree without any vertex coloring:



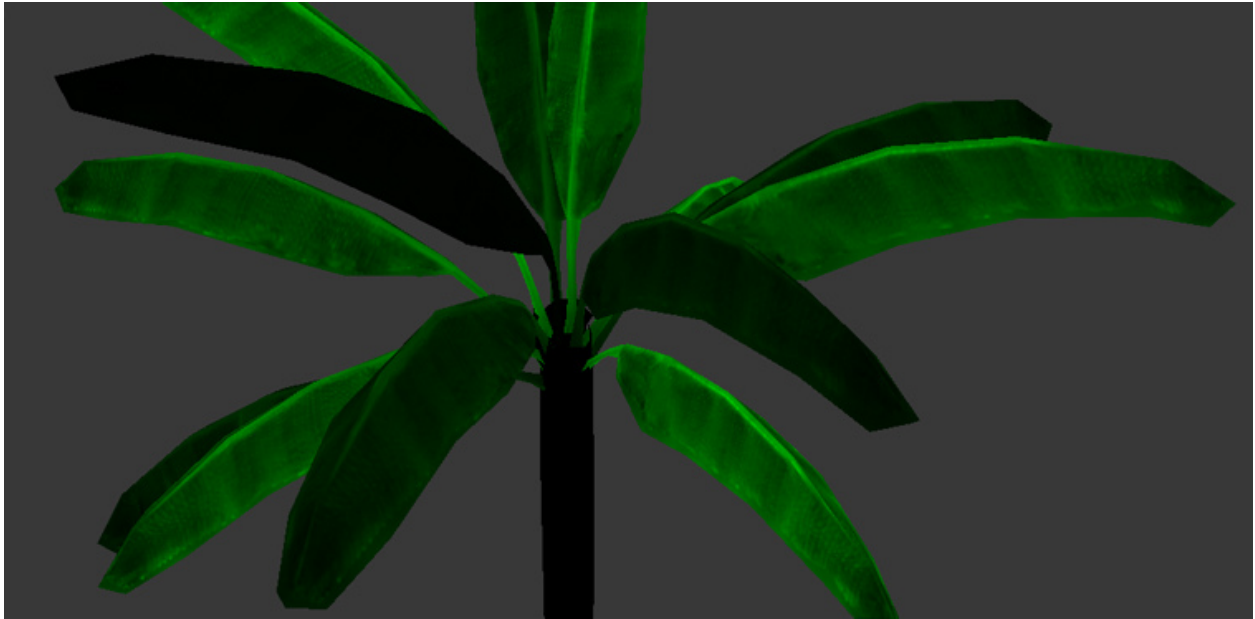
The red coloring only, to designate the upward/downward bend of the branches. Notice how the base of the branches are black and the ends of the branches are red:



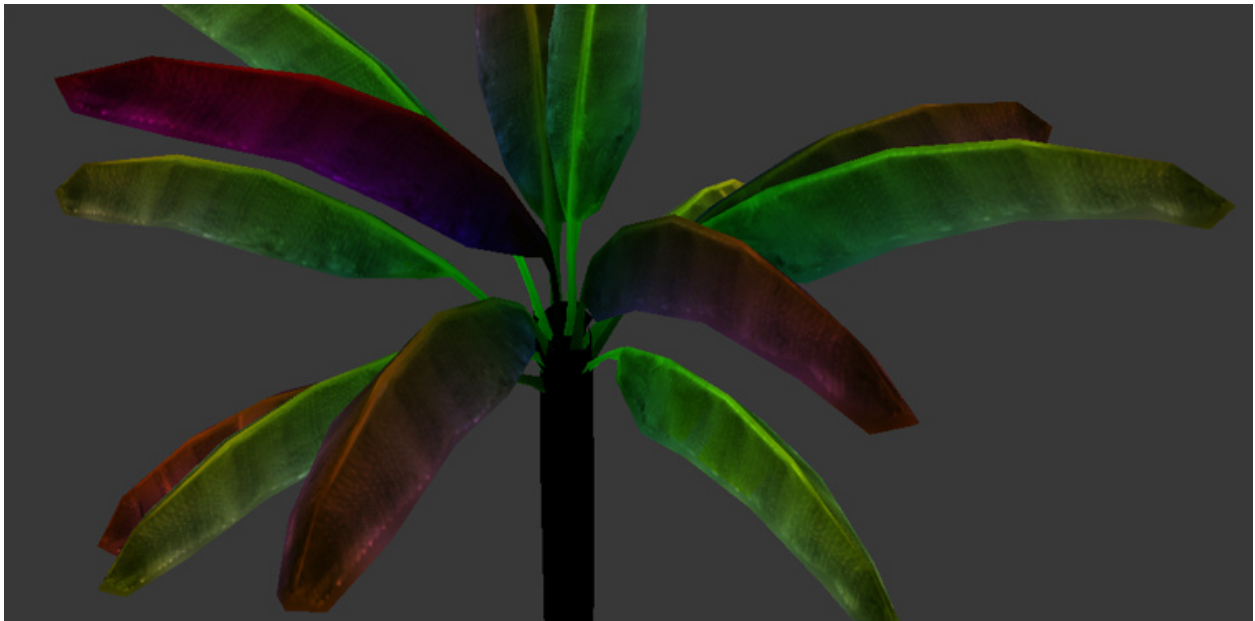
The blue coloring only, to designate leaf fluttering. Notice how only the edges of the blades are colored:



The green coloring only, to designate individual instances of branch bend. Each branch has its own shade of green:



Finally, all the colors mixed together with an additive blend. Notice that the trunk has no coloring, since it will neither flutter like a leaf nor wave like a branch. The bend along the tree's height is done without the use of vertex coloring:



Here is a video of this in action:

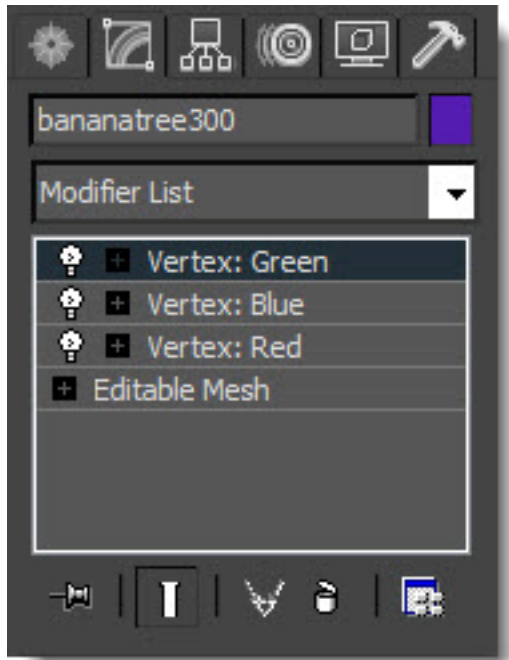
<https://www.youtube.com/watch?v=q8cjthzGcf0> (video/Forest Editor Wind Effects.mp4)

How you actually apply these color sets varies between modeling apps. The following is an example from 3D Studio Max:

1. Stack three separate Vertex Paint modifiers on the tree model.
2. Working from the bottom, one modifier at a time, fill the vertices with black, then paint one of the color sets listed above to each of the three modifiers. In the end, it doesn't matter in what order the colors are created or stacked.



3. After all three have been painted their own color set, set the blend mode of the top two Vertex Paint modifiers to “additive”.
4. Export your model as COLLADA with these modifiers on and you should have the vertex coloring you need to get wind effects on your meshes. Do not collapse the stack.



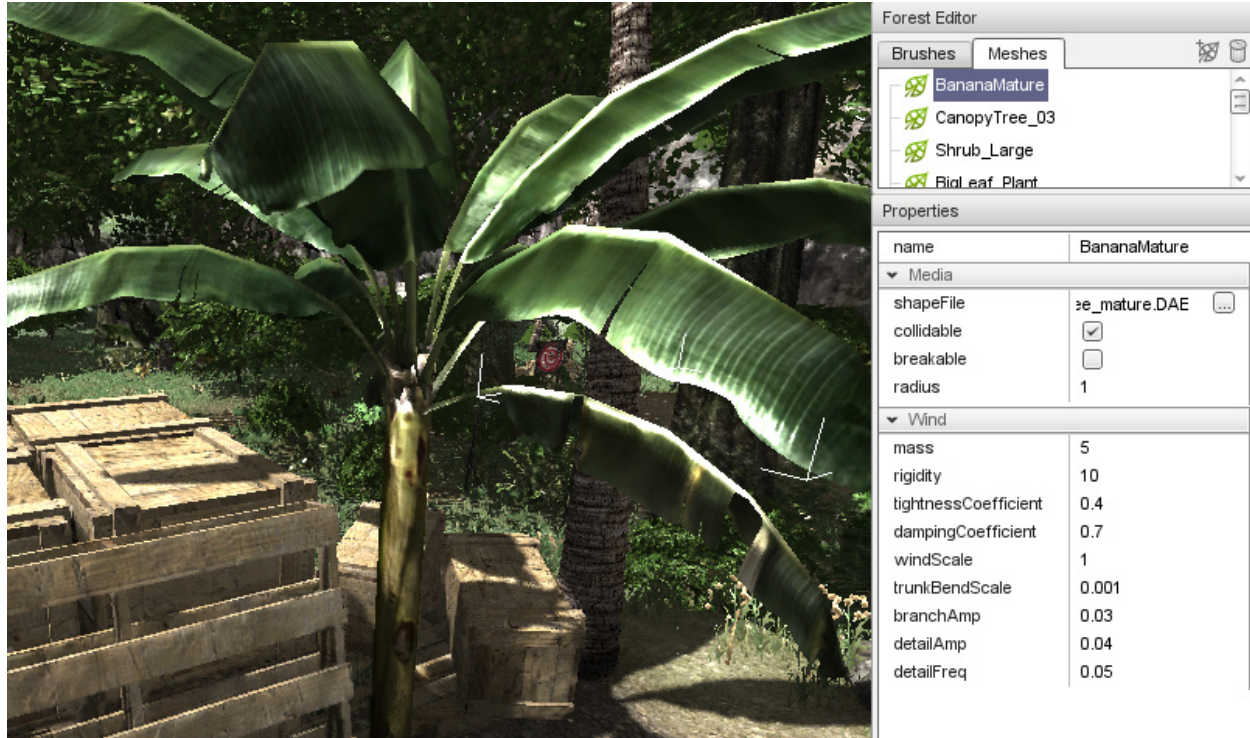
If you need to go back and tweak individual color sets/modifiers in Max, you can put their blending modes back to “normal” and turn them on and off to isolate an individual color set. **DO NOT** eyeball the colors. Dial them in. Red means pure red, or R:100%, G:0%, B: 0%.

### 20.4.3 Exporting Your Mesh

The DTS format does not support vertex color data, which means you will need to export your models as COLLADA files. A commonly used plug-in is OpenCOLLADA, which is a free exporter for 3D Studio Max.

Note: At the time this article was written, 3DS Max’s built-in COLLADA(.dae) exporter was not exporting vertex coloring correctly. OpenCOLLADA exports both vertex coloring and User Properties correctly. Just be aware that it appends material names with “-material”.

## 20.4.4 Setting Up A Mesh In Torque 3D



Once a mesh is brought into the Forest Editor (and a global wind emitter is placed in the scene), the mesh parameters must be set properly for your plant to come alive correctly. Definitions for mesh parameters:

- **shapeFile** - Path to mesh
- **collidable** - Indicates whether or not the mesh uses collision
- **breakable** - (not implemented)
- **radius** - The canopy radius in meters. This keeps plants from being placed too close to one another and having overlapping canopies. This value may need to be adjusted on larger plants when placing smaller plants under their canopy (placing bushes under a tree, for instance). This has no in-game functionality and only matters when placing trees in your environment.
- **mass** - Mass will generally affect how springy a plant is in response to wind or an explosion. Think of it as a weight on the end of a stick. Most of my trees are set to around 5, smaller plants down to 0.5.
- **rigidity** - How much the plant resists the wind force. Most of my trees are within 10-20, but some small grasses go down to 1.
- **tightnessCoefficient** - How much the plant resists bending. Between 0 and 1.
- **dampingCoefficient** - Slows down the sway of the plant so it won't whip back and forth forever. Between 0 and 1.
- **windScale** - Relative scale at which this plant is affected by wind. Between 0 and 1, but will almost always be set around 1.
- **trunkBendScale** - The amount that a plant will sway from top to bottom. This is very sensitive. I don't have any vegetation set higher than 0.03, and most are under 0.01.
- **branchAmp** - Amount that a branch will sway up and down. Start well below 1.
- **detailAmp** - The amount of leaf flutter. Start well below 1, in the 0.05 range.

- **detailFreq** - The speed of leaf flutter. Start well below 1, in the 0.05 range.

It may seem like a lot of these appear to be doing the same thing, but there are subtle differences. The numbers shown in the example images are just starting places. Your plants may differ due to the way you've colored their vertices or set the global wind emitter.

### 20.4.5 Setting the View Distance of Wind Effects

Plants in the distance will not display wind effects. This can be changed by simply adjusting the variable assigned to this range: `$pref::windEffectRadius = 30`. That sets it to 30 meters around the camera. The default setting is 25 meters. Obviously this will affect performance, but you can cater it to your specs. Make sure to save your level after doing this.

### 20.4.6 Conclusion

This guide covered the basics of how Torque 3D utilizes vertex painted objects. Artists are given a lot of control over forest editing. Vertex painting is an extremely important step and should be considered when developing your forest assets. In addition to the global wind emitter, you can use a local wind emitter around anything that may create a localized wind situation

- waterfalls
- helicopter rotors
- big fans
- jet engine exhaust
- etc



## 20.5 Destructible Objects MAX - TODO

### 20.5.1 Introduction to Destructible Objects

### 20.5.2 Setting Up Collision

### 20.5.3 Hierarchy For Meshes With Multiple LODs

### 20.5.4 Aligning the Pivot Points

### 20.5.5 Zero Transforms

### 20.5.6 Setting up the physicsShape.cs file

### 20.5.7 Building a destructible object with more than one damage states

### 20.5.8 Null LODs

### 20.5.9 Conclusion

## 20.6 Destructible Objects XSI - TODO

### 20.6.1 Introduction to Destructible Objects

### 20.6.2 Setting Up Collision

### 20.6.3 Exporting from Softimage

### 20.6.4 Hierarchy For Meshes With Multiple LODs

### 20.6.5 Aligning the Pivot Points

### 20.6.6 Freeze Transforms

### 20.6.7 Setting up the physicsShape.cs file

### 20.6.8 Building a destructible object with more than one damage states

### 20.6.9 Null LODs

### 20.6.10 Conclusion

## 20.7 PostFX Color Correction - TODO

### 20.7.1 Required Knowledge

- You need to be familiar with Torque 3D. Specifically loading a level for game play, taking an in game screen shot and locating the screen shot file location for editing, saving, and loading.

- You need to have a basic understanding of Adobe Photoshop or another image editing software package.

### **20.7.2 Color Correction**

Color Correction is a powerful new tool to help fine tune the overall color temperature in a Torque 3D scene. These adjustments are performed in an image editing program such as Photoshop that allows for individual channel (R, G, B ) tuning. Interesting variants can easily be achieved for a scene with some simple steps.

### **20.7.3 Workflow**





## CHAPTER 21

---

### Importing Assets

---

#### 21.1 Introduction

Test



## 22.1 Introduction to TorqueScript

### 22.1.1 What is TorqueScript?

TorqueScript (TS) is a proprietary scripting language developed specifically for Torque technology. The language itself is derived from the scripting used for Tribes 2, which was the base tech Torque evolved from. Scripts are written and stored in .cs files, which are compiled and executed by a binary compiled via the C++ engine (.exe for Windows or .app OS X). The CS extension should not be confused with C# files.

The CS extension stands for “C Script,” meaning the language resembles C programming. Though there is a connection, TorqueScript is a much higher level language and is easier to learn than standard C or C++.

### 22.1.2 Basic Usage

Like most other scripting languages, such as Python or Java Script, TorqueScript is a high-level programming language interpreted by Torque 3D at run time. Unlike C++, you can write your code in script and run it without recompiling your game.

All of your interfaces can be built using the GUI Editor, which saves the data out to TorqueScript. The same goes for data saved by the World Editor or Material Editor. Most of the editors themselves are C++ components exposed and constructed via TorqueScript.

More importantly, nearly all of your game play programming will be written in TorqueScript: inventory systems, win/lose scenarios, AI, weapon functionality, collision response, and game flow. All of these can be written in TorqueScript. The language will allow you to rapidly prototype your game without having to be a programming expert or perform lengthy engine recompilation.

### 22.1.3 Scripting vs Engine Programming

As mentioned above, TorqueScript is comprised of the core C++ objects needed to make your game. For example, you will use the PlayerData structure to create player objects for your game. This structure was written in C++:

```
struct PlayerData: public ShapeBaseData {
    typedef ShapeBaseData Parent;
    bool renderFirstPerson;    ///< Render the player shape in first person

    mass = 9.0f;               // from ShapeBase
    drag = 0.3f;               // from ShapeBase
    density = 1.1f;            // from ShapeBase
}
```

Instead of having to go into C++ and create new PlayerData objects or edit certain fields (such as mass), PlayerData was exposed to TorqueScript:

```
datablock PlayerData(DefaultPlayerData)
{
    renderFirstPerson = true;

    className = Armor;
    shapeFile = "art/shapes/actors/gideon/base.dts";

    mass = 100;
    drag = 1.3;
    maxdrag = 0.4;

    // Allowable Inventory Items
    maxInv[Pistol] = 1;
    maxInv[PistolAmmo] = 50;
};
```

If you want to change the name of the object, the mass, the inventory, or anything else, just open the script, make the change, and save the file. When you run your game, the changes will immediately take effect. Of course, for this example you could have used the in-game Datablock Editor, but you should get the point. TorqueScript is the first place you should go to write your game play code.

### 22.1.4 Getting Started

Like the rest of the documentation, the TorqueScript guides should be read in order (from top to bottom in the table of contents). This means you should start by reading the Syntax Guide. If you have never written TorqueScript before, **DO NOT SKIP** the Syntax Guide.

The docs in the **Simple Tutorials** will provide you with working code meant to show off syntax and basic TorqueScript structures. This is where you will create, edit, debug, and execute your first scripts.

Finally, the **Advanced Tutorials** section will walk you through complex functionality and algorithms. These tutorials make full use of Torque 3D's editors, networking structure, input system, and TorqueScript. These examples will even get into game play mechanics.

If you simply need a quick reference while writing scripts, you can read through the **TorqueScript Reference Guide**.

## 22.2 TorqueScript Editors - TODO

TorqueScript files are essentially text files. This means you have several editors to choose from. Some users prefer to use the stock OS text editors: Notepad on Windows or Text Edit on OS X. Others will use their programming IDEs (Interactive Development Environments), such as Visual Studio for Windows or Xcode on OS X. Third party applications are the most popular choice:

### 22.2.1 On Windows

**Torsion** - Torsion is undeniably the best TorqueScript IDE was developed by Torque veterans Sickhead Games. If you are developing on Windows, this is the first thing you should purchase after Torque 3D.

Torsion is a powerful development environment for creating TorqueScript based games and mods.

No other editor offers this level of quality and functionality:

- Integrated “One Click” script debugging.
- Full control over script execution via step and break commands.
- Advanced editor features like code folding, line wrapping, auto-indent, column marker, automatic bracket matching, and visible display of tabs and spaces.
- Go to line and text searching.
- ScriptSense updated dynamically as you type.
- Customizable syntax highlighting for TorqueScript.
- Unlimited undo/redo buffer.
- Code browser window for exploring both engine exports and script symbols in your project.

Source Code: <https://github.com/SickheadGames/Torsion/>

Download: <https://github.com/Torque3D-Resources/Torsion/releases/download/v1.1.392-noKey/Torsion.zip>

**Notepad++** - <https://notepad-plus-plus.org/> - This is a free (as in “free speech” and also as in “free beer”) source code editor and Notepad replacement that supports several languages. Syntax Highlighting: <http://greek2me.webs.com/Downloads/Random/Torquescript%20Syntax%20Highlighting.xml> - Forum: <https://forum.blockland.us/index.php?topic=152179.0>

### 22.2.2 On OS X

**Xcode** - <https://developer.apple.com/xcode/> - Xcode is Apple’s premier development environment for Mac OS X. If you plan on modifying Torque 3D’s source code, you will need this anyway. Most developers at GarageGames use Xcode to modify their scripts on a Mac.

**Text Edit** - This is the OS X default text editor. With no bells or whistles, this is not the best editor you can use on OS X, but it is free and ships with the OS.

**Smultron** - <https://www.peterborgapps.com/smultron/> - Smultron is a text editor written in Cocoa for Mac OS X Leopard 10.5 which is designed to be both easy to use and powerful.

### 22.2.3 Cross-platform

**TIDE** - <http://torqueide.sourceforge.net/> - Torque Integrated Development Environment (TIDE) is a free, cross-platform IDE for Torque Game Engine scripting by Paul Dana and Stefan Moises. It is implemented in Java as a plug-in suite for the jEdit text editor and contains plug-ins for syntax highlighting, function browsing, script debugging, etc.

**Atom** - <https://atom.io> - Is a text editor that’s modern, approachable, yet hackable to the core? a tool you can customize to do anything but also use productively without ever touching a config file.

**UltraEdit** - <https://www.ultraedit.com/> - UltraEdit is a powerful disk-based text editor, programmer’s editor, and hex editor that is used to edit TorqueScript, HTML, PHP, JavaScript, Perl, C/C++, and a multitude of other coding/programming languages.



**TODO** - <http://wiki.torque3d.org/introduction:scripting-ides>

## 22.3 Torsion TorqueScript IDE - TODO

Introducing Torsion, a powerful development environment for creating TorqueScript based games and mods.

Created by dedicated Torque developers, Torsion will maximize your productivity when working on your project based on Torque game engines (like Torque 3D). Unlike other editors, Torsion solely targets TorqueScript development to ensure a focused tool without features for other engines getting in your way.

If you haven't tried Torsion and have been using TorqueScript you've been missing out.

Users familiar with other modern development environments will feel right at home using Torsion. Torsion has everything one would expect in a modern IDE:

- Project centric MDI design with a familiar and intuitive user interface.
- Advanced editor features like code folding, line wrapping, auto-indent, column marker, automatic bracket matching, and visible display of tabs and spaces.
- Customizable syntax highlighting for TorqueScript.
- Unlimited undo/redo buffer.
- Goto line and text searching.
- ScriptSense updated dynamically as you type.
- Smart project tree view allows for file manipulation and automatically updates as new files are added.
- Code browser window for exploring both engine exports and script symbols in your project.
- Integrated "One Click" script debugging.
- Full control over script execution via step and break commands.
- Conditional breakpoints.
- Call stack and watch windows allow you to fully inspect and change the running game state.
- Remote console for executing commands in your running game.
- Complete editor state saved between project sessions including open files, breakpoints, and variable watches.
- Can be easily copied and run from a flash drive.
- Written entirely in C++ making it fast and efficient with a small memory footprint.

Source Code: <https://github.com/SickheadGames/Torsion/>

Download: <https://github.com/Torque3D-Resources/Torsion/releases/download/v1.1.392-noKey/Torsion.zip>

## 22.4 Syntax Guide

### 22.4.1 The Basics

#### Main Rules

Like other languages, TorqueScript has certain syntactical rules you need to follow. The language is very forgiving, easy to debug, and is not as strict as a low level language like C++. Observe the following line in a script:

```
// Create test variable with a temporary variable
%testVariable = 3;
```

The three most simple rules obeyed in the above code are:

1. Ending a line with a semi-colon ;
2. Proper use of white space.
3. Commenting.

The engine will parse code line by line, stopping whenever it reaches a semi-colon. This is referred to as a **statement terminator**, common to other programming languages such as C++, JavaScript, etc. The following code will produce an error that may cause your entire script to fail:

```
%testVariable = 3
%anotherVariable = 4;
```

To the human eye, you are able to discern two separate lines of code with different actions. Here is how the script compiler will read it:

```
%testVariable = 3%anotherVariable = 4;
```

This is obviously not what the original code was meant to do. There are exemptions to this rule, but they come into play when multiple lines of code are supposed to work together for a single action:

```
if(%testVariable == 4)
    echo("Variable equals 4");
```

We have not covered conditional operators or echo commands yet, but you should notice that the first line does not have a semi-colon. The easiest explanation is that the code is telling the compiler: “Read the first line, do the second line if we meet the requirements.” In other words, perform operations between semi-colons. Complex operations require multiple lines of code working together.

The second rule, proper use of whitespace, is just as easy to remember. Whitespace refers to how your script code is separated between operations. Let’s look at the first example again:

```
%testVariable = 3;
```

The code is storing a value 3 in a local variable %testVariable. It is doing so by using a common mathematical operator, the equal sign. TorqueScript recognizes the equal sign and performs the action just as expected. It does not care if there are spaces in the operation:

```
%testVariable=3;
```

The above code works just as well, even without the spaces between the variable, the equal sign, and the 3. The whitespace rule makes a lot more sense when combined with the semi-colon rule and multiple lines of code working together. The following will compile and run without error:

```
if(%testVariable == 4) echo("Variable equals 4");
```

## Comments

The last rule is optional, but should be used as often as possible if you want to create clean code. Whenever you write code, you should try to use **comments**. Comments are a way for you to leave notes in code which are not compiled into the game. The compiler will essentially skip over these lines.

There are two different comment syntax styles. The first one uses the two slashes, `//`. This is used for single line comments:

Example:

```
// This comment line will be ignored
// This second line will also be ignored
%testVariable = 3;
// This third line will also be ignored
```

In the last example, the only line of code that will be executed has to do with `%testVariable`. If you need to comment large chunks of code, or leave a very detailed message, you can use the `/*comment*/` syntax. The `/*` starts the commenting, the `*/` ends the commenting, and anything in between will be considered a comment:

Example:

```
/*
While attending school, an instructor taught a mantra I still use:

"Read. Read Code. Code."

Applying this to Torque 3D development is easy:

READ the documentation first.

READ CODE written by other Torque developers.

CODE your own prototypes based on what you have learned.
*/
```

As you can see, the comment makes full use of whitespace and multiple lines. While it is important to comment what the code does, you can also use this to temporarily remove unwanted code until a better solution is found:

Example:

```
// Why are you using multiple if statements. Why not use a switch $?
/*
if(%testVariable == "Mich")
    echo("User name: ", %testVariable);

if(%testVariable == "Heather")
    echo("User Name: ", %testVariable);

if(%testVariable == "Nikki")
    echo("User Name: ", %testVariable);
*/
```

## 22.4.2 Variables

### Usage

A variable is a letter, word, or phrase linked to a value stored in your game's memory and used during operations. Creating a variable is a one line process. The following code creates a variable by naming it and assigning a value:

```
%localVariable = 3;
```

You can assign any type value to the variable you want. This is referred to as a language being **type-insensitive**. TorqueScript does not care (insensitive) what you put in a variable, even after you have created it. The following code is completely valid:

```
%localVariable = 27;
%localVariable = "Heather";
%localVariable = "7 7 7";
```

The main purpose of the code is to show that TorqueScript treats all data types the same way. It will interpret and convert the values internally, so you do not have to worry about typecasting. That may seem a little confusing. After all, when would you want a variable that can store a number, a string, or a vector?

You will rarely need to, which is why you want to start practicing good programming habits. An important practice is proper variable naming. The following code will make a lot more sense, considering how the variables are named:

```
%userName = "Heather";
%userAge = 27;
%userScores = "7 7 7";
```

TorqueScript is more forgiving than low level programming languages. While it expects you to obey the basic syntax rules, it will allow you to get away with small mistakes or inconsistency. The best example is variable **case sensitivity**. With variables, TorqueScript is not case sensitive. You can create a variable and refer to it during operations without adhering to case rules:

```
%userName = "Heather";
echo(%Username);
```

In the above code, %userName and %Username are the same variable, even though they are using different capitalization. You should still try to remain consistent in your variable naming and usage, but you will not be punished if you slip up occasionally.

## Variable Types

There are two types of variables you can declare and use in TorqueScript: *local* and *global*. Both are created and referenced similarly:

```
%localVariable = 1;
$globalVariable = 2;
```

As you can see, local variable names are preceded by the percent sign %. Global variables are preceded by the dollar sign \$. Both types can be used in the same manner: operations, functions, equations, etc. The main difference has to do with how they are **scoped**.

In programming, scoping refers to where in memory a variable exists during its life. A local variable is meant to only exist in specific blocks of code, and its value is discarded when you leave that block. Global variables are meant to exist and hold their value during your entire programs execution. Look at the following code to see an example of a local variable:

```
function test()
{
    %userName = "Heather";
    echo(%userName);
}
```

We will cover functions a little later, but you should know that functions are blocks of code that only execute when you call them by name. This means the variable, %userName, does not exist until the test() function is called.

When the function has finished all of its logic, the `%userName` variable will no longer exist. If you were to try to access the `%userName` variable outside of the function, you will get nothing.

Most variables you will work with are local, but you will eventually want a variables that last for your entire game. These are extremely important values used throughout the project. This is when global variables become useful. For the most part, you can declare global variables whenever you want:

```
$PlayerName = "Heather";

function printPlayerName()
{
    echo($PlayerName);
}

function setPlayerName()
{
    $PlayerName = "Nikki";
}
```

The above code makes full use of a global variable that holds a player's name. The first declaration of the variable happens outside of the functions, written anywhere in your script. Because it is global, you can reference it in other locations, including separate script files. Once declared, your game will hold on to the variable until shutdown.

### 22.4.3 Data Types

TorqueScript implicitly supports several variable data-types: numbers, strings, booleans, arrays and vectors. If you wish to test the various data types, you can use the **echo(...)** command. For example:

```
%meaningOfLife = 42;
echo(%meaningOfLife);

$name = "Heather";
echo($name);
```

The echo will post the results in the console, which can be accessed by pressing the tilde key (~) while in game.

#### Numbers

TorqueScript handles standard numeric types:

```
123      (Integer)
1.234    (floating point)
1234e-3  (scientific notation)
0xc001   (hexadecimal)
```

#### Strings

Text, such as names or phrases, are supported as strings. Numbers can also be stored in string format. Standard strings are stored in double-quotes:

```
"abcd"   (string)
```

Example:

```
$UserName = "Heather";
```

Strings with single quotes are called “tagged strings”:

```
'abcd' (tagged string)
```

Tagged strings are special in that they contain string data, but also have a special numeric tag associated with them. Tagged strings are used for sending string data across a network. The value of a tagged string is only sent once, regardless of how many times you actually do the sending.

On subsequent sends, only the tag value is sent. Tagged values must be de-tagged when printing. You will not need to use a tagged string often unless you are in need of sending strings across a network often, like a chat system.

Example:

```
$a = 'This is a tagged string';
echo(" Tagged string: ", $a);
echo("Detagged string: ", detag($a));
```

The output will be similar to this:

```
Tagged string: 24
Detagged string:
```

The second echo will be **blank** unless the string has been passed to you over a network.

## String Operators

There are special values you can use to concatenate strings and variables. Concatenation refers to the joining of multiple values into a single variable. The following is the basic syntax:

```
"string 1" operation "string 2"
```

You can use string operators similarly to how you use mathematical operators (`=`, `+`, `-`, `*`). You have four operators at your disposal:

Op- era- tor	Name	Exam- ple	Explanation
@	String	<code>\$c @ \$d</code>	Concatenates strings <code>\$c</code> and <code>\$d</code> into a single string. Numeric literals/variables convert to strings.
NL	New Line	<code>\$c NL \$d</code>	Concatenates strings <code>\$c</code> and <code>\$d</code> into a single string separated by new-line. <b>Note:</b> such a string can be decomposed with <code>getRecord()</code>
TAB	Tab	<code>\$c TAB \$d</code>	Concatenates strings <code>\$c</code> and <code>\$d</code> into a single string separated by tab. <b>Note:</b> such a string can be decomposed with <code>getField()</code>
SPC	Space	<code>\$c SCP \$d</code>	Concatenates strings <code>\$c</code> and <code>\$d</code> into a single string separated by space. <b>Note:</b> such a string can be decomposed with <code>getWord()</code>

The @ symbol will concatenate two strings together exactly how you specify, without adding any additional whitespace:

*Note: Do not type in OUPUT: \_\_\_\_\_. This is placed in the sample code to show you what the console would display.*

Example:



```
%newString = "Hello" @ "World";  
echo(%newString);
```

OUTPUT:  
HelloWorld

Notice how the two strings are joined without any spaces. If you include whitespace, it will be concatenated along with the values:

Example:

```
%newString = "Hello " @ "World";  
echo(%newString);
```

OUTPUT:  
Hello World

Example:

```
%hello = "Hello ";  
%world = "World";  
  
echo(%hello @ %world);
```

OUTPUT:  
Hello World

The rest of the operators will apply whitespace for you, so you do not have to include it in your values. Example:

```
echo("Hello" @ "World");  
echo("Hello" TAB "World");  
echo("Hello" SPC "World");  
echo("Hello" NL "World");
```

OUTPUT:  
HelloWorld  
Hello World  
Hello World  
Hello  
World

## Booleans

Like most programming languages, TorqueScript also supports Booleans. Boolean numbers have only two values—true or false:

```
true      (1)  
false     (0)
```

Again, as in many programming languages the constant “true” evaluates to the number 1 in TorqueScript, and the constant “false” evaluates to the number 0. However, non-zero values are also considered true. Think of booleans as “on/off” switches, often used in conditional statements. Example:

```
$lightsOn = true;
```

(continues on next page)

(continued from previous page)

```
if($lightsOn)
    echo("Lights are turned on");
```

## Arrays

Arrays are data structures used to store consecutive values of the same data type:

```
$TestArray[n]    (Single-dimension)
$TestArray[m,n]  (Multidimensional)
$TestArray[m_n]  (Multidimensional)
```

If you have a list of similar variables you wish to store together, try using an array to save time and create cleaner code. The syntax displayed above uses the letters ‘n’ and ‘m’ to represent where you will input the number of elements in an array. The following example shows code that could benefit from an array. Example:

```
$firstUser = "Heather";
$secondUser = "Nikki";
$thirdUser = "Mich";

echo($firstUser);
echo($secondUser);
echo($thirdUser);
```

Instead of using a global variable for each user name, we can put those values into a single array. Example:

```
$userNames[0] = "Heather";
$userNames[1] = "Nikki";
$userNames[2] = "Mich";

echo($userNames[0]);
echo($userNames[1]);
echo($userNames[2]);
```

Now, let’s break the code down. Like any other variable declaration, you can create an array by giving it a name and value:

```
$userNames[0] = "Heather";
```

What separates an array declaration from a standard variable is the use of brackets []. The number you put between the brackets is called the **index**. The index will access a specific element in an array, allowing you to view or manipulate the data. All the array values are stored in consecutive order.

If you were able to see an array on paper, it would look something like this:

```
[0] [1] [2]
```

In our example, the data looks like this:

```
["Heather"] ["Nikki"] ["Mich"]
```

Like other programming languages, the index is always a numerical value and the starting index is always 0. Just remember, index 0 is always the first element in an array. As you can see in the above example, we create the array by assigning the first index (0) a string value (“Heather”).

The next two lines continue filling out the array, progressing through the index consecutively:

```
$userNames[1] = "Nikki";  
$userNames[2] = "Mich";
```

The second array element (index 1) is assigned a different string value (“Nikki”), as is the third (index 2). At this point, we still have a single array structure, but it is holding three separate values we can access. Excellent for organization.

The last section of code shows how you can access the data that has been stored in the array. Again, you use a numerical index to point to an element in the array. If you want to access the first element, use 0:

```
echo($userNames[0]);
```

In a later section, you will learn about looping structures that make using arrays a lot simpler. Before moving on, you should know that an array does not have to be a single, ordered list. TorqueScript also support multidimensional arrays.

An single-dimensional array contains a single row of values. A multidimensional array is essentially an array of arrays, which introduces columns as well. The following is a visual of what a multidimensional looks like with three rows and three columns:

```
[x] [x] [x]  
[x] [x] [x]  
[x] [x] [x]
```

Defining this kind of array in TorqueScript is simple. The following creates an array with 3 rows and 3 columns:

```
$testArray[0,0] = "a";  
$testArray[0,1] = "b";  
$testArray[0,2] = "c";  
  
$testArray[1,0] = "d";  
$testArray[1,1] = "e";  
$testArray[1,2] = "f";  
  
$testArray[2,0] = "g";  
$testArray[2,1] = "h";  
$testArray[2,2] = "i";
```

Notice that we are now using two indices, both starting at 0 and stopping at 2. We can use these as coordinates to determine which array element we are accessing:

```
[0,0] [0,1] [0,2]  
[1,0] [1,1] [1,2]  
[2,0] [2,1] [2,2]
```

In our example, which progresses through the alphabet, you can visualize the data in the same way:

```
[a] [b] [c]  
[d] [e] [f]  
[g] [h] [i]
```

The first element [0,0] points to the letter ‘a’. The last element [2,2] points to the letter ‘i’.

## Vectors

“Vectors” are a helpful data-type which are used throughout Torque 3D. For example, many fields in the World Editor take numeric values in sets of 3 or 4. These are stored as strings and interpreted as “vectors”:

```
"1.0 1.0 1.0"    (3 element vector)
```

The most common example of a vector would be a world position. Like most 3D coordinate systems, an object's position is stored as (X Y Z). You can use a three element vector to hold this data:

```
%position = "25.0 32 42.5";
```

You can separate the values using spaces or tabs (both are acceptable whitespace). Another example is storing color data in a four element vector. The values that make up a color are “Red Blue Green Alpha,” which are all numbers. You can create a vector for color using hard numbers, or variables:

```
%firstColor = "100 100 100 255";
echo(%firstColor);

%red = 128;
%blue = 255;
%green = 64;
%alpha = 255;

%secondColor = %red SPC %blue SPC %green SPC %alpha;
echo(%secondColor);
```

## 22.4.4 Operators

Operators in TorqueScript behave very similarly to operators in real world math and other programming languages. You should recognize quite a few of these from math classes you took in school, but with small syntactical changes. The rest of this section will explain the syntax and show a brief example, but we will cover these in depth in later guides.

### Arithmetic Operators

These are your basic math ops.

Operator	Name	Example	Explanation
*	multiplication	<code>\$a * \$b</code>	Multiply <code>\$a</code> and <code>\$b</code> .
/	division	<code>\$a / \$b</code>	Divide <code>\$a</code> by <code>\$b</code> .
%	modulo	<code>\$a % \$b</code>	Remainder of <code>\$a</code> divided by <code>\$b</code> .
+	addition	<code>\$a + \$b</code>	Add <code>\$a</code> and <code>\$b</code> .
-	subtraction	<code>\$a - \$b</code>	Subtract <code>\$b</code> from <code>\$a</code> .
++	auto-increment (post-fix only)	<code>\$a++</code>	Increment <code>\$a</code> .
--	auto-decrement (post-fix only)	<code>\$b--</code>	Decrement <code>\$b</code> .

**Note:** `++$a` is illegal. The value of `$a++` is that of the incremented variable: auto-increment is post-fix in syntax, but pre-increment in semantics (the variable is incremented, *before* the return value is calculated). This behavior is unlike that of C and C++.

**Note:** `--$b` is illegal. The value of `$a--` is that of the decremented variable: auto-decrement is post-fix in syntax,

but pre-decrement in semantics (the variable is decremented, *before* the return value is calculated). This behavior is unlike that of C and C++.

---

## Relational Operators

Used in comparing values and variables against each other.

### Relations (Arithmetic, Logical, and String)

Operator	Name	Example	Explanation
<	Less than	<code>\$a &lt; \$b</code>	1 if \$a is less than \$b
>	More than	<code>\$a &gt; \$b</code>	1 if \$a is greater than \$b
<=	Less than or Equal to	<code>\$a &lt;= \$b</code>	1 if \$a is less than or equal to \$b
>=	More than or Equal to	<code>\$a &gt;= \$b</code>	1 if \$a is greater than or equal to \$b
==	Equal to	<code>\$a == \$b</code>	1 if \$a is equal to \$b
!=	Not equal to	<code>\$a != \$b</code>	1 if \$a is not equal to \$b
!	Logical NOT	<code>! \$a</code>	1 if \$a is 0
&&	Logical AND	<code>\$a &amp;&amp; \$b</code>	1 if \$a and \$b are both non-zero
	Logical OR	<code>\$a    \$b</code>	1 if either \$a or \$b is non-zero
\$=	String equal to	<code>\$c \$= \$d</code>	1 if \$c equal to \$d.
!=	String not equal to	<code>\$c != \$d</code>	1 if \$c not equal to \$d.

## Bitwise Operators

Used for comparing and shifting bits.

Operator	Name	Example	Explanation
~	Bitwise complement	<code>~\$a</code>	flip bits 1 to 0 and 0 to 1
&	Bitwise AND	<code>\$a &amp; \$b</code>	composite of elements where bits in same position are 1
	Bitwise OR	<code>\$a   \$b</code>	composite of elements where bits 1 in either of the two elements
^	Bitwise XOR	<code>\$a ^ \$b</code>	composite of elements where bits in same position are opposite
<<	Left Shift	<code>\$a &lt;&lt; 3</code>	element shifted left by 3 and padded with zeros
>>	Right Shift	<code>\$a &gt;&gt; 3</code>	element shifted right by 3 and padded with zeros

*References: You might find these two guides useful for learning about bitwise operations:*

1. [https://en.wikipedia.org/wiki/Bitwise\\_operation](https://en.wikipedia.org/wiki/Bitwise_operation)
2. [http://www.cprogramming.com/tutorial/bitwise\\_operators.html](http://www.cprogramming.com/tutorial/bitwise_operators.html)

## Assignment Operators

Used for setting the value of variables.

### Assignment and Assignment Operators

Operator	Name	Example	Explanation
=	Assignment	\$a = \$b;	Assign value of \$b to \$a <b>Note:</b> the value of an assignment is the value being assigned, so \$a = \$b = \$c is legal.
op=	Assignment Operators	\$a op= \$b;	Equivalent to \$a = \$a op \$b, where op can be any of: * / % + - &   ^ << >>

## Miscellaneous Operators

General programming operators.

Operator	Name	Example	Explanation
? :	Conditional	x ? y : z	Evaluates to y if x equal to 1, else evaluates to z
[]	Array element	\$a[5]	Synonymous with \$a5
( )	Delimiting, Grouping	t2dGetMin(%a, %b) if ( \$a == \$b ) (\$a+\$b) * (\$c-\$d)	Argument list for function call Used with if, for, while, switch keywords Control associativity in expressions
{ }	Compound statement	if (1) { \$a = 1; \$b = 2; } function foo() { \$a = 1; }	Delimit multiple statements, optional for if, else, for, while Required for switch, datablock, new, function
,	Listing	t2dGetMin(%a, %b) %M[1,2]	Delimiter for arguments. <b>Note:</b> there is no “comma operator”, as defined in C/C++; \$a = 1, \$b = 2; is a parse error
::	Namespaces	Item::onCollision()	This definition of the onCollision() function is in the Item namespace
.	Field/Method selection	%obj.field %obj.method()	Select a console method or field
//	Single-line comment	// This is a comment	Used to comment out a single line of code
/* */	Multi-line comment	/*This is a multi-line comment*/	Used to comment out multiple consecutive lines /* opens the comment, and */ closes it

### 22.4.5 Control Statements

TorqueScript provides basic branching structures that will be familiar to programmers that have used other languages. If you are completely new to programming, you use branching structures to control your game’s flow and logic. This section builds on everything you have learned about TorqueScript so far.

#### if, else

This type of structure is used to test a condition, then perform certain actions if the condition passes or fails. You do not always have to use the full structure, but the following syntax shows the extent of the conditional:



```
if(<boolean expression>){
    pass logic
}
else
{
    alternative logic
}
```

Remember how boolean values work? Essentially, a bool can either be true (1) or false (0). The condition (boolean) is always typed into the parenthesis after the “if” syntax. Your logic will be typed within the brackets {}. The following example uses specific variable names and conditions to show how this can be used:

Example:

```
// Global variable that controls lighting
$lightsShouldBeOn = true;

// Check to see if lights should be on or off
if($lightsShouldBeOn)
{
    // True. Call turn on lights function
    turnOnLights();

    echo("Lights have been turned on");
}
else
{
    // False. Turn off the lights
    turnOffLights();

    echo("Lights have been turned off");
}
```

Brackets for single line statements are optional. If you are thinking about adding additional logic to the code, then you should use the brackets anyway. If you know you will only use one logic statement, you can use the following syntax:

Example:

```
// Global variable that controls lighting
$lightsShouldBeOn = true;

// Check to see if lights should be on or off
if($lightsShouldBeOn)
    turnOnLights(); // True. Call turn on lights function
else
    turnOffLights(); // False. Turn off the lights
```

## switch and switch\$

If your code is using several cascading if-then-else statements based on a single value, you might want to use a **switch** statement instead. Switch statements are easier to manage and read. There are two types of switch statements, based on data type: numeric (switch) and string (switch\$).

Switch Syntax:

```
switch(<numeric expression>)
{
    case value0:
        statements;
    case value1:
        statements;
    case value3:
        statements;
    default:
        statements;
}
```

As the above code demonstrates, start by declaring the switch statement by passing in a value to the `switch(...)` line. Inside of the brackets `{}`, you will list out all the possible cases that will execute based on what value being tested. It is wise to always use the default case, anticipating rogue values being passed in.

Example:

```
switch($ammoCount)
{
    case 0:
        echo("Out of ammo, time to reload");
        reloadWeapon();
    case 1:
        echo("Almost out of ammo, warn user");
        lowAmmoWarning();
    case 100:
        echo("Full ammo count");
        playFullAmmoSound();
    default:
        doNothing();
}
```

`switch` only properly evaluates numerical values. If you need a switch statement to handle a string value, you will want to use **switch\$**. The `switch$` syntax is similar to what you just learned:

Switch\$ Syntax:

```
switch$ (<string expression>)
{
    case "string value 0":
        statements;
    case "string value 1":
        statements;
    ...
    case "string value N":
        statements;
    default:
        statements;
}
```

Appending the `$` sign to `switch` will immediately cause the parameter passed in to be parsed as a string. The following code applies this logic:

Example:

```
// Print out specialties
switch($userName)
```

(continues on next page)

(continued from previous page)

```
{
    case "Heather":
        echo("Sniper");
    case "Nikki":
        echo("Demolition");
    case Mich:
        echo("Meat shield");
    default:
        echo("Unknown user");
}
```

## 22.4.6 Loops

As the name implies, this structure type is used to repeat logic in a loop based on an expression. The expression is usually a set of variables that increase by count, or a constant variable changed once a loop has hit a specific point.

### For Loop

for Loop Syntax:

```
for(expression0; expression1; expression2)
{
    statement(s);
}
```

One way to label the expressions in this syntax are (startExpression; testExpression; countExpression). Each expression is separated by a semi-colon.

Example:

```
for(%count = 0; %count < 3; %count++)
{
    echo(%count);
}
```

OUTPUT:

```
0
1
2
```

The first expression creates the local variable `%count` and initializing it to 0. In the second expression determines when to stop looping, which is when the `%count` is no longer less than 3. Finally, the third expression increases the count the loop relies on.

### While Loop

A while loop is a much simpler looping structure compared to a for loop.

while Loop Syntax:

```
while(expression)
{
    statements;
}
```

As soon as the expression is met, the while loop will terminate:

Example:

```
%countLimit = 0;

while(%countLimit <= 5)
{
    echo("Still in loop");
    %count++;
}
echo("Loop was terminated");
```

## 22.4.7 Functions

Much of your TorqueScript experience will come down to calling existing functions and writing your own. Functions are a blocks of code that only execute when you call them by name. Basic functions in TorqueScript are defined as follows:

```
// function - Is a keyword telling TorqueScript we are defining a new function.
// function_name - Is the name of the function we are creating.
// ... - Is any number of additional arguments.
// statements - Your custom logic executed when function is called
// return val - The value the function will give back after it has completed.
↳Optional.

function function_name([arg0], ..., [argn])
{
    statements;
    [return val;]
}
```

The **function** keyword, like other TorqueScript keywords, is case sensitive. You must type it exactly as shown above. The following is an example of a custom function that takes in two parameters, then executes code based on those arguments.

TorqueScript can take any number of arguments, as long as they are comma separated. If you call a function and pass fewer parameters than the function's definition specifies, the un-passed parameters will be given an empty string as their default value.

Example:

```
function echoRepeat (%echoString, %repeatCount)
{
    for (%count = 0; %count < %repeatCount; %count++)
    {
        echo(%echoString);
    }
}
```

You can cause this function to execute by calling it in the console, or in another function:

```
echoRepeat("hello!", 5);

OUTPUT:
"hello!"
"hello!"
```

(continues on next page)

(continued from previous page)

```
"hello!"  
"hello!"  
"hello!"
```

If you define a function and give it the same name as a previously defined function, TorqueScript will completely override the old function. This still applies even if you change the number of parameters used; the older function will still be overridden.

## 22.4.8 Console Methods

Console Methods are C++ functions that have been exposed to TorqueScript, which are attached to specific objects.

## 22.4.9 Console Functions

Console Functions are written in C++, then exposed to TorqueScript. These are global functions you can call at any time, and are usually very helpful or important. Throughout this document, I have been using a ConsoleFunction: `echo(...)`. The `echo` function definition exists in C++:

C++ `echo`:

```
ConsoleFunction(echo, void, 2, 0, "echo(text [, ... ])")  
{  
    U32 len = 0;  
    S32 i;  
    for(i = 1; i < argc; i++)  
        len += dStrlen(argv[i]);  
  
    char *ret = Con::getReturnBuffer(len + 1);  
    ret[0] = 0;  
    for(i = 1; i < argc; i++)  
        dStrcat(ret, argv[i]);  
  
    Con::printf("%s", ret);  
    ret[0] = 0;  
}
```

Instead of having to write that out every time, or create a TorqueScript equivalent, the `ConsoleFunction` macro in C++ exposes the command for you. This is much cleaner, and more convenient. We will cover all the `ConsoleFunctions` later.

## 22.4.10 Objects

The most complex aspect of TorqueScript involves dealing with game objects. Much of your object creation will be performed in the World Editor, but you should still know how to manipulate objects at a script level. One thing to remember is that everything in TorqueScript is an object: players, vehicles, items, etc.

Every object added in the level is saved to a mission file, which is written entirely in TorqueScript. This also means every game object is accessible from script. First, we will study the syntax of object creation.

## Syntax

Even though objects are originally created in C++, they are exposed to script in a way that allows them to be declared using the following syntax:

Object Definition:

```
// In TorqueScript
%objectID = new ObjectType(Name : CopySource, arg0, ..., argn)
{
    <datablock = DatablockIdentifier;>

    [existing_field0 = InitialValue0;]
    ...
    [existing_fieldN = InitialValueN;]

    [dynamic_field0 = InitialValue0;]
    ...
    [dynamic_fieldN = InitialValueN;]
};
```

### Syntax Breakdown:

- **%objectID** - Is the variable where the object's handle will be stored.
- **new** - Is a key word telling the engine to create an instance of the following **ObjectType**.
- **ObjectType** - Is any class declared in the engine or in script that has been derived from **SimObject** or a subclass of **SimObject**. **SimObject**-derived objects are what we were calling "game world objects" above.
- **Name** (optional) - Is any expression evaluating to a string, which will be used as the object's name.
- **CopySource** (optional) - The name of an object which is previously defined somewhere in script. Existing field values will be copied from **CopySource** to the new object being created. Any dynamic fields defined in **CopySource** will also be defined in the new object, and their values will be copied. Note: If **CopySource** is of a different **ObjectType** than the object being created, only **CopySource**'s dynamic fields will be copied.
- **arg0, ..., argn** (optional) - Is a comma separated list of arguments to the class constructor (if it takes any).
- **datablock** - Many objects (those derived from **GameBase**, or children of **GameBase**) require datablocks to initialize specific attributes of the new object. Datablocks are discussed below.
- **existing\_fieldN** - In addition to initializing values with a datablock, you may also initialize existing class members (fields) here. Note: In order to modify a member of a C++-defined class, the member must be exposed to the Console. This concept is discussed in detail later.
- **dynamic\_fieldN** - Lastly, you may create new fields (which will exist only in Script) for your new object. These will show up as dynamic fields in the World Editor Inspector.

The main object variants you can create are **SimObjects** without a datablock, and game objects which require a datablock. The most basic **SimObject** can be created in a single line of code:

Example:

```
// Create a SimObject without any name, argument, or fields.
$exampleSimObject = new SimObject();
```

The `$exampleSimObject` variable now has access to all the properties and functions of a basic **SimObject**. Usually, when you are creating a **SimObject** you will want custom fields to define features

Example:



```
// Create a SimObject with a custom field
$exampleSimObject = new SimObject()
{
    catchPhrase = "Hello world!";
};
```

As with the previous example, the above code creates a `SimObject` without a name which can be referenced by the global variable `$exampleSimObject`. This time, we have added a user defined field called “catchPhrase.” There is not a single stock Torque 3D object that has a field called “catchPhrase.” However, by adding this field to the `SimObject` it is now stored as long as that object exists.

The other game object variant mentioned previously involves the usage of datablocks. Datablocks contain static information used by a game object with a similar purpose. Datablocks are transmitted from a server to client, which means they cannot be modified while the game is running.

We will cover datablocks in more detail later, but the following syntax shows how to create a game object using a datablock.

Example:

```
// create a StaticShape using a datablock
datablock StaticShapeData(ceiling_fan)
{
    category = "Misc";
    shapeFile = "art/shapes/undercity/cfan.dts";
    isInvincible = true;
};

new StaticShape(CistFan)
{
    dataBlock = "ceiling_fan";
    position = "12.5693 35.5857 59.5747";
    rotation = "1 0 0 0";
    scale = "1 1 1";
};
```

Once you have learned about datablocks, the process is quite simple:

1. Create a datablock in script, or using the datablock editor
2. Add a shape to the scene from script or using the World Editor
3. Assign the new object a datablock

## Handles vs Names

Every game object added to a level can be accessed by two parameters:

- **Handle** - A unique numeric ID generated when the object is created
- **Name** - This is an optional parameter given to an object when it is created. You can assign a name to an object from the World Editor, or do so in TorqueScript using the following syntax:

Example:

```
// In this example, CistFan is the name of the object
new StaticShape(CistFan)
{
    dataBlock = "ceiling_fan";
```

(continues on next page)

(continued from previous page)

```

position = "12.5693 35.5857 59.5747";
rotation = "1 0 0 0";
scale = "1 1 1";
};

```

While in the World Editor, you will not be allowed to assign the same name to multiple, separate objects. The editor will ignore the attempt. If you manually name two objects the same thing in script, the game will only load the first object and ignore the second.

## Singletons

If you need a global script object with only a single instance, you can use the singleton keyword. Singletons, in TorqueScript, are mostly used for unique shaders, materials, and other client-side only objects.

For example, SSAO (screen space ambient occlusion) is a post-processing effect. The game will only ever need a single instance of the shader, but it needs to be globally accessible on the client. The declaration of the SSAO shader in TorqueScript can be shown below:

```

singleton ShaderData( SSAOShader )
{
    DXVertexShaderFile      = "shaders/common/postFx/postFxV.hlsl";
    DXPixelShaderFile       = "shaders/common/postFx/ssao/SSAO_P.hlsl";
    pixVersion = 3.0;
};

```

## Object Fields

Objects instantiated via script may have data members (referred to as Fields)

## Methods

In addition to the creation of stand-alone functions, TorqueScript allows you to create and call methods attached to objects. Some of the more important ConsoleMethods are already written in C++, then exposed to script. You can call these methods by using the dot (.) notation.

Syntax:

```

objHandle.function_name();

objName.function_name();

```

Example:

```

new StaticShape(CistFan)
{
    dataBlock = "ceiling_fan";
    position = "12.5693 35.5857 59.5747";
    rotation = "1 0 0 0";
    scale = "1 1 1";
};

// Write all the objects methods to the console log
CistFan.dump();

```

(continues on next page)

(continued from previous page)

```
// Get the ID of an object, using the object's name
$objID = CistFan.getID();

// Print the ID to the console
echo("Object ID: ", $objID);

// Get the object's position, using the object's handle
%position = $objID.getPosition();

// Print the position to the console
echo("Object Position: ", %position);
```

The above example shows how you can call an object's method by using its name or a variable containing its handle (unique ID number). Additionally, TorqueScript supports the creation of methods that have no associated C++ counterpart.

Syntax:

```
// function - Is a keyword telling TorqueScript we are defining a new function.
// ClassName:- Is the class type this function is supposed to work with.
// function_name - Is the name of the function we are creating.
// ... - Is any number of additional arguments.
// statements - Your custom logic executed when function is called
// %this- Is a variable that will contain the handle of the 'calling object'.
// return val - The value the function will give back after it has completed.
↳Optional.

function ClassName::func_name(%this, [arg0], ..., [argn])
{
    statements;
    [return val;]
}
```

At a minimum, Console Methods require that you pass them an object handle. You will often see the first argument named %this. People use this as a hint, but you can name it anything you want. As with Console functions any number of additional arguments can be specified separated by commas.

As a simple example, let's say there is an object called Samurai, derived from the Player class. It is likely that a specific appearance and play style will be given to the samurai, so custom ConsoleMethods can be written. Here is a sample:

Example:

```
function Samurai::sheatheSword(%this)
{
    echo("Katana sheathed");
}
```

When you add a Samurai object to your level via the World Editor, it will be given an ID. Let's pretend the handle (ID number) is 1042. We can call its ConsoleMethod once it is defined, using the period syntax:

Example:

```
1042.sheatheSword();

OUTPUT: "Katana sheathed"
```

Notice that no parameters were passed into the function. The `%this` parameter is inherent, and the original function did not require any other parameters.

## 22.4.11 Conclusion

This guide covered the basics of TorqueScript syntax. Compared to other languages, such as C++, it is easier to learn and work with. However, no one is expected to become a TorqueScript master over night, or even in a week. You will most likely need to refer back to this documentation several times for reminders.

**Option 1:** Take some time to read through the **Torque 3D Script Manual**. This manual contains the most valuable information you can find about TorqueScript and Torque 3D's API. This will be your "go to" source for documentation on functions, variables, and classes that make up Torque 3D.

**Option 2:** Move on to the **Simple Tutorials** section, which will continue walking you through the basics of TorqueScript. These tutorials will provide you with sample code to read over. You will also get sample scripts at the end of each tutorial to test out.

## 22.5 Quick Reference - TODO

### 22.5.1 Language Features

TorqueScript is a typeless scripting language, with similarities in syntax to C/C++. In TorqueScript, you will find that most C/C++ operators work in the familiar way (with important exceptions, as noted here). Besides a subset of C/C++, TorqueScript provides:

- Case-insensitive symbols; keywords *false* and *FALSE* are identical.
- Auto creation and destruction of local/global variables and their storage.
- String concatenation, comparison, and auto-string-constant creation.
- Function packaging.

### 22.5.2 Variable Names

### 22.5.3 Constants

### 22.5.4 Operators

Arithmetic Operators	Explanation	Operator	Name	Example
<code>*</code>	multiplication	<code>\$a * \$b</code>	Multiply	<code>\$a</code> and <code>\$b</code> .
<code>/</code>	division	<code>\$a / \$b</code>	Divide	<code>\$a</code> by <code>\$b</code> .
<code>%</code>	modulo	<code>\$a % \$b</code>	Remainder of	<code>\$a</code> divided by <code>\$b</code> .
<code>+</code>	addition	<code>\$a + \$b</code>	Add	<code>\$a</code> and <code>\$b</code> .
<code>-</code>	subtraction	<code>\$a - \$b</code>	Subtract	<code>\$b</code> from <code>\$a</code> .
<code>++</code>	auto-increment	<code>\$a++</code>	Increment	<code>\$a</code> .

(post-fix only)

`--` auto-decrement `$b--` Decrement `$b`.

(post-fix only)

Operator	Name	Example	Explanation
<code>&lt;</code>	Less than	<code>\$a &lt; \$b</code>	1 if <code>\$a</code> is less than <code>\$b</code>
<code>&gt;</code>	More than	<code>\$a &gt; \$b</code>	1 if <code>\$a</code> is greater than <code>\$b</code>
<code>&lt;=</code>	Less than or Equal to	<code>\$a &lt;= \$b</code>	1 if <code>\$a</code> is less than or equal to <code>\$b</code>
<code>&gt;=</code>	More than or Equal to	<code>\$a &gt;= \$b</code>	1 if <code>\$a</code> is greater than or equal to <code>\$b</code>
<code>==</code>	Equal to	<code>\$a == \$b</code>	1 if <code>\$a</code> is equal to <code>\$b</code>
<code>!=</code>	Not equal to	<code>\$a != \$b</code>	1 if <code>\$a</code> is not equal to <code>\$b</code>
<code>!</code>	Logical NOT	<code>!\$a</code>	1 if <code>\$a</code> is 0
<code>&amp;&amp;</code>	Logical AND	<code>\$a &amp;&amp; \$b</code>	1 if <code>\$a</code> and <code>\$b</code> are both non-zero
<code>  </code>	Logical OR	<code>\$a    \$b</code>	1

if either \$a or \$b is non-zero \$= String equal to \$c \$= \$d 1 if \$c equal to \$d. !\$= String not equal to \$c !\$= \$d 1 if \$c not equal to \$d. =====

**Bitwise Operators** ===== Operator Name Example  
 Explanation ===== ~ Bitwise complement ~\$a flip  
 bits 1 to 0 and 0 to 1 & Bitwise AND \$a & \$b composite of elements where bits in same position are 1 | Bitwise OR  
 \$a | \$b composite of elements where bits 1 in either of the two elements ^ Bitwise XOR \$a ^ \$b composite of  
 elements where bits in same position are opposite << Left Shift \$a << 3 element shifted left by 3 and padded with  
 zeros >> Right Shift \$a >> 3 element shifted right by 3 and padded with zeros =====

**Assignment and Assignment Operators** =====  
 ===== Operator Name Example Explanation =====  
 ===== = Assignment \$a = \$b; Assign value of \$b to \$a

**Note:** the value of an assignment is the value being assigned, so \$a = \$b = \$c is legal.

op= Assignment Operators \$a op= \$b; Equivalent to \$a = \$a op \$b, where op can be any of: \* / % + - & | ^  
 << >> =====

**String Operators** ===== Operator Name Example  
 Explanation ===== @ String \$c @ \$d Concatenates  
 strings \$c and \$d

into a single string. Numeric literals/variables convert to strings.

**NL New Line \$c NL \$d Concatenates strings \$c and \$d** into a single string separated by new-line. **Note:** such a  
 string can be decomposed with getRecord()

**TAB Tab \$c TAB \$d Concatenates strings \$c and \$d** into a single string separated by tab. **Note:** such a string  
 can be decomposed with getField()

**SPC Space \$c SCP \$d Concatenates strings \$c and \$d** into a single string separated by space. **Note:** such a  
 string can be decomposed with getWord()

Operator	Name	Example	Explanation
?	Conditional	x ? y : z	Evaluates to y if x equal to 1, else evaluates to z
[]	Array element	\$a[5]	Synonymous with \$a5
()	Delimiting, Grouping	t2dGetMin(%a, %b)	Argument list for function call

if ( \$a == \$b ) Used with if, for, while, switch keywords

(\$a+\$b) \* (\$c-\$d) Control associativity in expressions

{ } Compound statement if (1) { \$a = 1; \$b = 2; } Delimit multiple statements, optional for if, else, for,  
 while

function foo() { \$a = 1; } Required for switch, datablock, new, function

, Listing t2dGetMin(%a, %b) Delimiter for arguments.

**Note:** there is no “comma operator”, as defined in C/C++; \$a = 1, \$b = 2; is a parse error

%M[1,2]

:: Namespace Item::onCollision() This definition of the onCollision() function is in the Item names-  
 pace . Field/Method selection %obj.field Select a console method or field

%obj.method()

// Single-line comment // This is a comment Used to comment out a single line of code /\* \*/ Multi-line  
 comment /\*This is a a Used to comment out multiple consecutive lines

```
multi-line /* opens the comment, and */ closes it  
comment*/
```

**break**

**case**

**continue**

**datablock**

**default**

**else**

**FALSE**

**for**

**function**

**if**

**new**

**package**

**parent**

**return**

**switch**

**switch\$**

**TRUE**

**while**





## 23.1 Echo Examples

### 23.1.1 Syntax Review

The **echo(...)** method syntax is easy to memorize and is an extremely valuable debugging command:

**echo** (string, all)

Sends output to the console.

**Parameters**

- **string** – Text sent to console
- **all** – Optional value, of any type, that will be appended to the text

**Returns** No return value.

Examples:

```
// Print "Hello World" in the console
echo("Hello World");
```

### 23.1.2 Example

Start by running a Torque 3D project. Once the game is up, open the console by pressing the tilde (~) key. In the console, type the following:

**Example 1:**

```
echo("Hello World");

OUTPUT: Hello World
```

Now, let's make use of the second parameter. Passing in a value for the second argument will append it to your text:

### Example 2:

```
echo("Hello World", 3);  
  
OUTPUT: Hello World3
```

Notice how there is no space between World and 3. The optional text is appended exactly how you type it. If you want, you can include your own white space to format the output:

### Example 3:

```
echo("Hello World: ", 5);  
  
OUTPUT: Hello World: 5
```

As you can see, the colon and space are included in the output. 5 is still appended, but does not ignore the whitespace. In addition to **echo(...)**, there are two other output functions you will find useful. Their syntax and functionality are nearly identical to echo, but the output is different.

The two functions I'm referring to are **warn(...)** and **error(...)**. You can post a message in the console and log the same way you echo:

### Example 4:

```
warn("Be careful. Something bad might happen");  
  
error("Something has gone horribly wrong");  
  
OUTPUT:  
Be careful. Something bad might happen (teal color)  
Something has gone horribly wrong (red color)
```

You can use these functions to output multicolored text to the console, which will help you identify problems with your scripts.

## 23.1.3 Creating the Script

There is no real reason to have a script full of echo statements. You will want to use echo(...) while debugging your other functions. However, as an example, you can create a script consisting only of output statements.

First, we need to create a new script:

1. Navigate to your project's **game/scripts/client** directory.
2. Create a new script file called "Output.cs". In Torsion, right click on the directory, click the "New Script" option, then name your script. On Windows or OS X, create a new text file and change the extension to .cs
3. Open your new script using a text editor or Torsion.

Add the following code to the script:

### Output.cs:

```
//-----  
// Torque 3D  
// Copyright (C) GarageGames.com 2000 - 2009 All Rights Reserved  
//-----
```

(continues on next page)

(continued from previous page)

```
// Create a nice border effect around these echos, makes it easier to find
echo("*****");
echo("*****");

// Standard use
echo("Hello");
echo("World");
echo("Hello World");

// With escape commands
echo("H\\ne\\nl\\nl\\no\\nW\\no\\nr\\nl\\nd\\n");

// Appending
echo("Hello World", 1);
echo("Hello World ", 2);
echo("Hello World: ", 3);

// Warning
warn("Warning! Watch for teal text");

// Error
error("Something has gone horribly wrong");
echo("*****");
echo("*****");
```

Save the script now.

### 23.1.4 Testing the Script

Open `game/scripts/client/init.cs` and locate the `initClient()` function. At the end of that function, execute your new script by typing the following:

```
exec("./Output.cs");
```

Run your game, then open the console by pressing tilde (~). Look for the long string of asterisks (\*), and you will find your echo statements. **Note:** you may need to scroll up to find the echo statements.

### 23.1.5 Conclusion

Use `echo(...)`, `warn(...)`, and `error(...)` as often as you can. They can be very helpful when debugging your scripts. The rest of the TorqueScript documentation will use these functions to demonstrate functionality and give you cues on how things are being run.

## 23.2 Creating Functions

### 23.2.1 Syntax Review

Depending on the type of function you are writing or calling, the syntax will vary.

**Stand Alone Function Declaration:**

**function** **function\_name**([arg0],...){...}

Declaration of function

Syntax:

```
function function_name([arg0],...)
{
    statements;
    [return val;]
}
```

### Parameters

- **function** – Is a keyword telling TorqueScript we are defining a new function.
- **function\_name** – Is the name of the function we are creating.
- **[arg0]** – Argument. Value passed into function for use.
- **...** – Any number of additional arguments.
- **statements** – Your custom logic executed when function is called.

**Returns** **return val** *Optional*. The value the function will give back after it has completed.

Examples:

```
// Print 0 to value of %repeatCount in console
function echoRepeat (%echoString, %repeatCount)
{
    for (%count = 0; %count < %repeatCount; %count++)
    {
        echo(%echoString);
    }
}
```

## 23.2.2 Calling Functions

Once a function is written, you can call it from script or the console. You only need to know the name of the function and its parameters, if it has any. The **echo** function is the easiest method to start with. It is a stock ConsoleFunction which accepts up to 2 parameters:

**Example:**

```
// Print "Hello World" in the console
// Only passing in 1 argument
echo("Hello World");
```

The echo command can actually make use of 2 arguments, depending on your goal:

**Example:**

```
// Print "HelloWorld" in the console
// Passing in 2 arguments

%hello = "Hello";
%world = "World";

echo(%hello, %world);
```

If your function does not use arguments, you do not have to type anything in the parenthesis:

**Example:**

```
// Function declaration
function CreateLevels()
{
    echo("Levels Created");
}

// Calling the function
CreateLevels();
```

The last way to call a method is invoking a member function. You can call the member functions of an object, such as a Player, using a scoping symbol:

**Example:**

```
// Player function "doSomething"
// %this - The Player class/object
// %action - String to print out
function Player::doSomething(%this, %action)
{
    echo(%action);
}

// Create a player object
%myPlayer = new Player(){...};

// Call "doSomething" member function
%myPlayer.doSomething("Dance");
```

### 23.2.3 Creating the Script

Now that you know how to declare and call functions, we can create a few examples from scratch. First, we need to create a new script:

1. Navigate to your project's **game/scripts/client** directory.
2. Create a new script file called "sampleFunctions". In Torsion, right click on the directory, click the "New Script" option, then name your script. On Windows or OS X, create a new text file and change the extension to .cs
3. Open your new script using a text editor or Torsion.

Before writing any actual script code, we should go ahead and tell the game it should load the script. Open **game/scripts/client/init.cs**. Scroll down to the **initClient** function. Under the **// Client scripts** section, add the following:

Execute our new script:

```
exec("./sampleFunctions.cs");
```

Now, let's write an extremely simple function that prints a message to the console. The **echo(...)** function already performs this, but we are going to create a more intuitively named method to work with. Type the following in the script:

```
// Print a message to the console
// Kind of repetitiously redundant
```

(continues on next page)



(continued from previous page)

```
// %message - The message to print
function printMessage(%message)
{
    echo(%message);
}
```

To test your new script:

1. Save
2. Run your game
3. Open the console by pressing the tilde (~) key
4. Type the following, pressing enter after each line:

```
printMessage("Of melodies pure and true,");
printMessage("Sayin, this is my message to you-ou-ou");
```

Fairly straight forward. From here on, it will be assumed you know how to save your script, run the game, and call functions in the console. Next, let's create a function that takes multiple parameters. Write the following code in your script:

```
// Print two separate strings to the console
// Equally redundant in equality
// %part1 - First part of message
// %part2 - Second part of message
function printAdvancedMessage(%part1, %part2)
{
    echo(%part1, %part2);
}
```

Run the game and type the following in the console:

```
printAdvancedMessage("Singin: dont worry about a thing,", "\ncause every little thing_
↳gonna be all right");
```

In a single function call, the above code will write out two separate lyrics on different lines. Every game always has at least one initialization function. Some even have multiple inits. We can write a function that creates and initializes a few game specific variables. Note, that the variables used here are completely new and not used by stock Torque 3D projects:

```
// Change global game variables to default values
function resetGameVariables()
{
    // Game's name
    $GameName = "Blank";

    // Player's name
    $PlayerName = "Player";

    // Game play type
    $GameType = "Default";
}
```

The above code simply declares three global variables and sets them to default values. Every time this function is called, the same logic will execute. If you were to call this in the console, you will not see anything for output. Let's add a function to do this:

```
// Print our game's information to the console
function printGameInformation()
{
    echo("Game Name: ", $GameName);
    echo("Player's Name: ", $PlayerName);
    echo("Game Type: ", %gameType);
}
```

Save your new script and run the game. In the console, you will need to call the init function before the print function. Invoke the functions in this order:

```
resetGameVariables();
printGameInformation();
```

Instead of manually setting each variable in the console, we can write a “set” function for our game variables. Add the following to your script:

```
// Set the global game variables
// %gameName - Game's name
// %playerName - Player's name
// %gameType - Game play type
function setGameVariables(%gameName, %playerName, %gameType)
{
    $GameName = %gameName;
    $PlayerName = %playerName;
    $GameType = %gameType;
}
```

Now, you can set your game variables to whatever you wish through a single function call:

```
setGameVariables("Ars Moriendi", "Mich", "Survival Horror");

printGameInformation();

resetGameVariables();

printGameInformation();
```

We will get into creating member functions in a later section of the script documentation. For now, you should know enough about functions to move on.

## 23.2.4 Conclusion

A large portion of your Torque 3D development will occur in TorqueScript. 90% of that will be writing functions to handle your game play and other logic. With TorqueScript, it is easy to create and use functions without having to recompile the engine.

The information you learned in this doc will be used throughout the rest of the documentation, so make sure you are comfortable in your knowledge of functions. Continue reading to learn more advanced logic and math operators.

You can download the entire script from this lesson [HERE](#). Save the script as you would any other text file from a website:

```
//-----
// Torque 3D
// Copyright (C) GarageGames.com 2000 - 2009 All Rights Reserved
```

(continues on next page)

(continued from previous page)

```
//-----  
  
// Print a message to the console  
// Kind of repititiously redundant  
// %message - The message to print  
function printMessage(%message)  
{  
    echo(%message);  
}  
  
// Print two separate strings to the console  
// Equally redundant in equality  
// %part1 - First part of message  
// %part2 - Second part of message  
function printAdvancedMessage(%part1, %part2)  
{  
    echo(%part1, %part2);  
}  
  
// Change global game variables to default values  
function resetGameVariables()  
{  
    // Game's name  
    $GameName = "Blank";  
  
    // Player's name  
    $PlayerName = "Player";  
  
    // Game play type  
    $GameType = "Default";  
}  
  
// Set the global game variables  
// %gameName - Game's name  
// %playerName - Player's name  
// %gameType - Game play type  
function setGameVariables(%gameName, %playerName, %gameType)  
{  
    $GameName = %gameName;  
    $PlayerName = %playerName;  
    $GameType = %gameType;  
}  
  
// Print our game's information to the console  
function printGameInformation()  
{  
    echo("Game Name: ", $GameName);  
    echo("Player's Name: ", $PlayerName);  
    echo("Game Type: ", $GameType);  
}
```

## 23.3 Math Examples

### 23.3.1 Syntax Review

## Variable Types

There are two types of variables you can declare and use in TorqueScript: local and global. Both are created and referenced similarly:

```
%localVariable = 1;
$globalVariable = 2;
```

As you can see, local variable names are preceded by the percent sign (%). Global variables are preceded by the dollar sign (\$). Both types can be used in the same manner: operations, functions, equations, etc. The main difference has to do with how they are **scoped**.

In programming, scoping refers to where in memory a variable exists during its life. A local variable is meant to only exist in specific blocks of code, and its value is discarded when you leave that block. Global variables are meant to exist and hold their value during your entire programs execution.

## Operators

### Arithmetic Operators

Operator	Name	Example	Explanation
*	multiplication	\$a * \$b	Multiply \$a and \$b.
/	division	\$a / \$b	Divide \$a by \$b.
%	modulo	\$a % \$b	Remainder of \$a divided by \$b.
+	addition	\$a + \$b	Add \$a and \$b.
-	subtraction	\$a - \$b	Subtract \$b from \$a.
++	auto-increment (post-fix only)	\$a++	Increment \$a.
--	auto-decrement (post-fix only)	\$b--	Decrement \$b.

**Note:** ++\$a is illegal. The value of \$a++ is that of the incremented variable: auto-increment is post-fix in syntax, but pre-increment in semantics (the variable is incremented, *before* the return value is calculated). This behavior is unlike that of C and C++.

**Note:** --\$b is illegal. The value of \$a-- is that of the decremented variable: auto-decrement is post-fix in syntax, but pre-decrement in semantics (the variable is decremented, *before* the return value is calculated). This behavior is unlike that of C and C++.

### Relations (Arithmetic, Logical, and String)

Operator	Name	Example	Explanation
<	Less than	<code>\$a &lt; \$b</code>	1 if \$a is less than \$b
>	More than	<code>\$a &gt; \$b</code>	1 if \$a is greater than \$b
<=	Less than or Equal to	<code>\$a &lt;= \$b</code>	1 if \$a is less than or equal to \$b
>=	More than or Equal to	<code>\$a &gt;= \$b</code>	1 if \$a is greater than or equal to \$b
==	Equal to	<code>\$a == \$b</code>	1 if \$a is equal to \$b
!=	Not equal to	<code>\$a != \$b</code>	1 if \$a is not equal to \$b
!	Logical NOT	<code>! \$a</code>	1 if \$a is 0
&&	Logical AND	<code>\$a &amp;&amp; \$b</code>	1 if \$a and \$b are both non-zero
	Logical OR	<code>\$a    \$b</code>	1 if either \$a or \$b is non-zero
\$=	String equal to	<code>\$c \$= \$d</code>	1 if \$c equal to \$d.
!\$=	String not equal to	<code>\$c !\$= \$d</code>	1 if \$c not equal to \$d.

### Assignment and Assignment Operators

Operator	Name	Example	Explanation
=	Assignment	<code>\$a = \$b;</code>	Assign value of \$b to \$a <b>Note:</b> the value of an assignment is the value being assigned, so <code>\$a = \$b = \$c</code> is legal.
op=	Assignment Operators	<code>\$a op= \$b;</code>	<b>Equivalent to <code>\$a = \$a op \$b</code>, where op</b> <b>For example:</b> <code>\$a *= \$b;</code> is the same as <code>\$a = \$a * \$b;</code>

## 23.3.2 Creating the Script

First, we need to create a new script:

1. Navigate to your project's **game/scripts/client** directory.
2. Create a new script file called "maths". In Torsion, right click on the directory, click the "New Script" option, then name your script. On Windows or OS X, create a new text file and change the extension to .cs.
3. Open your new script using a text editor or Torsion.

Before writing any actual script code, we should go ahead and tell the game it should load the script. Open **game/scripts/client/init.cs**. Scroll down to the **initClient** function. Under the `// Client scripts` section, add the following:

**Execute our new script:**

```
exec("./maths.cs");
```

Standard arithmetic operators are the easiest to script. Start by adding this function to your new script:

```
// Print the sum of %a and %b
function addValues(%a, %b)
{
    %sum = %a + %b;
```

(continues on next page)

(continued from previous page)

```
    echo("Sum of " @ %a @ " + " @ %b @ ": ", %sum);
}
```

This simple function takes in two numerical arguments. A new variable, %sum, holds the result of adding the two arguments together. Finally, an echo(...) statement is formatted to print the original values (%a and %b) and the sum (%sum of the two).

To test your new script:

1. Save the script
2. Run your game
3. Open the console by pressing the tilde (~) key
4. Type the following, pressing enter after each line:

```
addValues(1,1);
addValues(2,3);
addValues(-3,2);
```

Your console output should look like this:

```
Sum of 1 + 1: 2
Sum of 2 + 3: 5
Sum of -3 + 2: -1
```

As you can see, you can use positive or negative numbers. You can also use floating point (decimal) values if you wish. Add the following script code to test the other basic arithmetic operations:

```
// Print the difference between %a and %b
function subtractValues(%a, %b)
{
    %difference = %a - %b;

    echo("Difference between " @ %a @ " - " @ %b @ ": ", %difference);
}

// Print the product of %a and %b
function multiplyValues(%a, %b)
{
    %product = %a * %b;

    echo("Product of " @ %a @ " * " @ %b @ ": ", %product);
}

// Print the quotient of %a and %b
function divideValues(%a, %b)
{
    %quotient = %a / %b;

    echo("Quotient of " @ %a @ " / " @ %b @ ": ", %quotient);
}

// Print remainder of %a divided by %b
function moduloValue(%a, %b)
{
```

(continues on next page)



(continued from previous page)

```

    %remainder = %a % %b;

    echo("Remainder of " @ %a @ " % " @ %b @ ": ", %remainder);
}

```

You will use the same process of scripting, saving, running the game, and calling the functions via the console that has been previously discussed above. Another way of manipulating values involves more complex operators. Standard additions, subtraction, etc, use two operators: assignment (=) and arithmetic (+,-,\*,etc).

You can increase or decrease the value of a variable by using the auto-increment and auto-decrement operators. As soon as the operation completes, the variable is permanently changed. You do not need to use an assignment operator in this case. Use the following script code to test it out:

```

// Print the increment of %a
function incrementValue(%a)
{
    %original = %a;
    %a++;

    echo("Single increment of " @ %original @ ": ", %a);
}

// Print the decrement of %a
function decrementValue(%a)
{
    %original = %a;
    %a--;

    echo("Single decrement of " @ %original @ ": ", %a);
}

```

As you can see, the original value of %a had to be stored before the increment/decrement operation was applied. The ++ and -- automatically adjust the variable for you. Another non-basic manipulation involves combining the assignment operator with an arithmetic operator:

```

// Print the result of a+=b
function addToValue(%a, %b)
{
    %original = %a;
    %a += %b;

    echo("Sum of " @ %original @ " += " @ %b @ ": ", %a);
}

```

In the above example, the + and = are combined together for a single operation. In simple terms, %a += %b can be verbalized as “A equals itself plus B.” Unlike the addValue(...) function written earlier, a third variable is not used in this equation. This operation can be applied to the other arithmetic operators.

The last topic we will cover in this guide is comparison operators. As the name implies, these operators will compare two values together and produce a boolean (1 or 0) based on the results. Add the following function to see the first example:

```

// Compare %a to %b, then print the relation
function compareValues(%a, %b)
{
    if(%a > %b)

```

(continues on next page)

(continued from previous page)

```
    echo("A is greater than B");
}
```

The above code is very straight forward. The values of `%a` and `%b` are compared to each other to see which is higher. Test the comparison code in the console using the following:

```
compareValues(2,1);
compareValues(3,2);
compareValues(1,2);
compareValues(0,0);
```

The output should be the following:

```
A is greater than B
A is greater than B
<no output>
<no output>
```

The first two calls will prove the comparison as “true”, and print out the message. The comparison results to false on the last two calls, so nothing will be printed. The rest of the function showing off the comparison operators can be copied over what you currently have:

```
// Compare %a to %b, then print the relation
function compareValues(%a, %b)
{
    // Printing symbols just as a decorator
    // Makes it easier to isolate the print out
    echo("\n=====");

    // Print out the value of %a and %b
    echo("\nValue of A: ", %a);
    echo("Value of B: ", %b);

    if(!%a)
        echo("\nA is a zero value\n");
    else
        echo("\nA is a non-zero value\n");

    if(!%b)
        echo("B is a zero value\n");
    else
        echo("B is a non-zero value\n");

    if(%a && %b)
        echo("Both A and B are non-zero values\n");

    if(%a || %b)
        echo("Either A or B is a non-zero value\n");

    if(%a == %b)
        echo("A is exactly equal to B\n");

    if(%a != %b)
        echo("A is not equal to B\n");

    if(%a < %b)
```

(continues on next page)

(continued from previous page)

```

    echo("A is less than B");
else if(%a <= %b)
    echo("A is less than or equal to B");

if(%a > %b)
    echo("A is greater than B");
else if(%a >= %b)
    echo("A is greater than or equal to B");

// Printing symbols just as a decorator
// Makes it easier to isolate the print out
echo("\n=====");
}

```

I have added “decorator text” to help separate console output and make the output easier to read. Notice that each operation uses an `if(...)` statement to compare. Remember, the `if(...)` code is based on checking for a 1 (true) or 0 (false) value. This is all a comparison operation will return.

### 23.3.3 Conclusion

The guide covers the basic arithmetic operators you will use in TorqueScript. For now, read back over the script code you have been provided with. Study the comments and `echo(...)` commands, and feel free to test out new operators.

You can download the entire script from this lesson [HERE](#). Save the script as you would any other text file from a website:

```

//-----
// Torque 3D
// Copyright (C) GarageGames, LLC 2011 All Rights Reserved
//-----

// Print the sum of %a and %b
function addValues(%a, %b)
{
    %sum = %a + %b;

    echo("Sum of " @ %a @ " + " @ %b @ ": ", %sum);
}

// Print the difference between %a and %b
function subtractValues(%a, %b)
{
    %difference = %a - %b;

    echo("Difference between " @ %a @ " - " @ %b @ ": ", %difference);
}

// Print the product of %a and %b
function multiplyValues(%a, %b)
{
    %product = %a * %b;

    echo("Product of " @ %a @ " * " @ %b @ ": ", %product);
}

```

(continues on next page)

(continued from previous page)

```
// Print the quotient of %a and %b
function divideValues(%a, %b)
{
    %quotient = %a / %b;

    echo("Quotient of " @ %a @ " / " @ %b @ ": ", %quotient);
}

// Print remainder of %a divided by %b
function moduloValue(%a, %b)
{
    %remainder = %a % %b;

    echo("Remainder of " @ %a @ " % " @ %b @ ": ", %remainder);
}

// Print the increment of %a
function incrementValue(%a)
{
    %original = %a;
    %a++;

    echo("Single increment of " @ %original @ ": ", %a);
}

// Print the decrement of %a
function decrementValue(%a)
{
    %original = %a;
    %a--;

    echo("Single decrement of " @ %original @ ": ", %a);
}

// Print the result of a+=b
function addToValue(%a, %b)
{
    %original = %a;
    %a += %b;

    echo("Sum of " @ %original @ " += " @ %b @ ": ", %a);
}

// Compare %a to %b, then print the relation
function compareValues(%a, %b)
{
    // Printing symbols just as a decorator
    // Makes it easier to isolate the print out
    echo("\n=====");

    // Print out the value of %a and %b
    echo("\nValue of A: ", %a);
    echo("Value of B: ", %b);

    if(!%a)
        echo("\nA is a zero value\n");
    else
```

(continues on next page)

(continued from previous page)

```
    echo("\nA is a non-zero value\n");

    if(!%b)
        echo("B is a zero value\n");
    else
        echo("B is a non-zero value\n");

    if(%a && %b)
        echo("Both A and B are non-zero values\n");

    if(%a || %b)
        echo("Either A or B is a non-zero value\n");

    if(%a == %b)
        echo("A is exactly equal to B\n");

    if(%a != %b)
        echo("A is not equal to B\n");

    if(%a < %b)
        echo("A is less than B");
    else if(%a <= %b)
        echo("A is less than or equal to B");

    if(%a > %b)
        echo("A is greater than B");
    else if(%a >= %b)
        echo("A is greater than or equal to B");

    // Printing symbols just as a decorator
    // Makes it easier to isolate the print out
    echo("\n=====");
}

function compareStrings(%string1, %string2)
{
    // Print out the values of %string1 and %string2
    echo("\nValue of String 1: ", %string1);
    echo("Value of String 2: ", %string2);

    if(%string1 $= %string2)
    {
        echo("\nString 1 and String 2 contain identical text");
    }

    if(%string1 != %string2)
    {
        echo("\nString 1 and String 2 contain different text");
    }
}
```

## 23.4 String Manipulation

### 23.4.1 Syntax Review

Text, such as names or phrases, are supported as strings. Numbers can also be stored in string format. Standard strings are stored in double-quotes:

```
"abcd" (string)
```

Example:

```
$UserName = "Heather";
```

Strings with single quotes are called “tagged strings.”:

```
'abcd' (tagged string)
```

Tagged strings are special in that they contain string data, but also have a special numeric tag associated with them. Tagged strings are used for sending string data across a network. The value of a tagged string is only sent once, regardless of how many times you actually do the sending.

On subsequent sends, only the tag value is sent. Tagged values must be de-tagged when printing. You will not need to use a tagged string often unless you are in need of sending strings across a network often, like a chat system.

There are special values you can use to concatenate strings and variables. Concatenation refers to the joining of multiple values into a single variable. The following is the basic syntax:

```
"string 1" operation "string 2"
```

You can use string operators similarly to how you use mathematical operators (`=`, `+`, `-`, `*`). You have four operators at your disposal:

```
@      (concatenates two strings)
TAB     (concatenation with tab)
SPC     (concatenation with space)
NL      (newline)
```

### 23.4.2 Creating the Script

First, we need to create a new script:

1. Navigate to your project’s **game/scripts/client** directory.
2. Create a new script file called “stringManip”. In Torsion, right click on the directory, click the “New Script” option, then name your script. On Windows or OS X, create a new text file and change the extension to `.cs`.
3. Open your new script using a text editor or Torsion.

Before writing any actual script code, we should go ahead and tell the game it should load the script. Open **game/scripts/client/init.cs**. Scroll down to the **initClient** function. Under the `// Client scripts` section, add the following:

Execute our new script:

```
exec("./stringManip.cs");
```

In the new script, define three global variables at the very top as shown in the following code:



```
$PlayerName = "Player";  
$GameName = "Default";  
$BattleCry = "Hello World";
```

These are the strings that will be manipulated in this script. To test one of the variables, write the following function:

```
// Print player name string  
function printPlayerName()  
{  
    echo($PlayerName);  
}
```

The `printPlayerName()` function simply prints out the string value held by `$PlayerName` to the console. To test your new script:

1. Save the script
2. Run your game
3. Open the console by pressing the tilde (~) key
4. Type the following, and press enter:

```
printPlayerName();
```

The output is extremely basic. All you will see is the string held by the variable. We can perform some string manipulation to print out something more descriptive. Change the function code to the following:

```
// Print player name string  
function printPlayerName()  
{  
    // Concatenate "Player's Name" with the variable  
    // Containing the name  
    echo("Player's Name: " @ $PlayerName);  
}
```

Now, when you call the function you will see the following output:

```
Player's Name: Default
```

This kind of string formatting and manipulation will make debugging and management a lot easier. Add the following code to achieve the same affect for the `$GameName` variable:

```
// Print game name string  
function printGameName()  
{  
    // Concatenate "Game Name" with the variable  
    // Containing the name  
    echo("Game Name: " @ $GameName);  
}
```

We will do something slightly different with the battle cry. You can store the result of a string manipulation in a variable before you use it. This will come in handy for saving permanent changes for strings and numbers. Use the following code to create a new function:

```
// Print battle cry string  
function printBattleCry()  
{
```

(continues on next page)

(continued from previous page)

```
// Concatenate the string in $PlayerName
// with the static string yelled: "
$message = $PlayerName @ " yelled: \";

// Concatenate the value of %message with
// the string in $BattleCry and the " symbol
// Store the results in the %message variable
$message = %message @ $BattleCry @ "\"";

// Print the new string after it
// has been manipulated
echo(%message);
}
```

The `printBattleCry()` function starts by defining a new local variable (`%message`) and assigning it the value of the `$PlayerName` concatenated with a static string. The second line concatenates the new `%message` variable with the contents of `$BattleCry`, and wraps the quotation mark around the actual phrase. In the same line, the `%message` variable is replaced with itself + the concatenated string.

Let's go ahead and create a function to print all of the variables out with a little decoration. Add the following to your script:

```
// Print all the game strings using a single function
function printGameStrings()
{
    echo("\n*****");
    echo("*          GAME STATS          *");
    echo("*****\n");

    echo("Game Name: " @ $GameName);
    echo("Player's Name: " @ $PlayerName);
    echo($PlayerName @ " battle cry: \" @ $BattleCry @ "\"");
}
```

When you call this function in the console, you will get the following output:

```
*****
*          GAME STATS          *
*****

Game Name: Default
Player's Name: Player
Player battle cry: "Hello World"
```

So far we have been concatenating and printing out strings. You can also assign string values using the assignment operator (`=`), and compare string values using the string equality operator (`$=`).

The following function uses the operators to adjust the game string variables:

```
// Set game strings with other strings
// %playerName will be assigned to $PlayerName
// %gameName will be assigned to $GameName
// %battleCry will be assigned to $BattleCry
function setGameStrings(%playerName, %gameName, %battleCry)
{
    // Check to see if the two strings are identical
    // If so, do nothing and print a message.
```

(continues on next page)

(continued from previous page)

```
// Otherwise, assign the new string
if($PlayerName $= %playerName)
    echo("New player name is identical. Doing nothing");
else
    $PlayerName = %playerName;
}
```

The above function takes in three variables containing strings, one of which is used initially. The first if(...) check compares \$PlayerName to %playerName. If the two are identical, the assignment of a new value will not occur. A message will be printed to console instead.

You can also apply the logical NOT (!) operator to a comparison to achieve the opposite test:

```
// Check to see if the two strings are different
// If so, assign the new string
// Otherwise, do nothing and print a message.
if($GameName != %gameName)
    $GameName = %gameName;
else
    echo("Game name is identical. Doing nothing");
```

In this check, if the two strings are NOT the same, then the new value assignment will occur. Otherwise, a message is printed to the console. You can go ahead and add the last portion of the code handling the %battleCry assignment:

```
// Check to see if the two strings are identical
// If so, do nothing and print a message.
// Otherwise, assign the new string
if($BattleCry $= %battleCry)
    echo("Battle cry is identical. Doing nothing");
else
    $BattleCry = %battleCry;
```

### 23.4.3 Conclusion

This guide covered the most popular operators used for string manipulation: concatenate (@), assignment (=), string equality (\$=), and string inequality (!\$=). Outside of simply printing to the console, during development you will be manipulating strings that directly affect game play, interface messages, and the saving of important data.

You can download the entire script from this lesson [HERE](#). Save the script as you would any other text file from a website:

```
//-----
// Torque 3D
// Copyright (C) GarageGames, LLC 2011 All Rights Reserved
//-----

$PlayerName = "Player";
$GameName = "Default";
$BattleCry = "Hello World";

// Print player name string
function printPlayerName()
{
    // Concatenate "Player's Name" with the variable
    // Containing the name
```

(continues on next page)

(continued from previous page)

```

    echo("Player's Name: " @ $PlayerName);
}

// Print game name string
function printGameName()
{
    // Concatenate "Game Name" with the variable
    // Containing the name
    echo("Game Name: " @ $GameName);
}

// Print battle cry string
function printBattleCry()
{
    // Concatenate the string in $PlayerName
    // with the static string yelled: "
    %message = $PlayerName @ " yelled: \"";

    // Concatenate the value of %message with
    // the string in $BattleCry and the " symbol
    // Store the results in the %message variable
    %message = %message @ $BattleCry @ "\"";

    // Print the new string after it
    // has been manipulated
    echo(%message);
}

// Print all the game strings using a single function
function printGameStrings()
{
    echo("\n*****");
    echo("*          GAME STATS          *");
    echo("*****\n");

    echo("Game Name: " @ $GameName);
    echo("Player's Name: " @ $PlayerName);
    echo($PlayerName @ " battle cry: \"" @ $BattleCry @ "\"");
}

// Set game strings with other strings
// %playerName will be assigned to $PlayerName
// %gameName will be assigned to $GameName
// %battleCry will be assigned to $BattleCry
function setGameStrings(%playerName, %gameName, %battleCry)
{
    // Check to see if the two strings are identical
    // If so, do nothing and print a message.
    // Otherwise, assign the new string
    if($PlayerName $= %playerName)
        echo("New player name is identical. Doing nothing");
    else
        $PlayerName = %playerName;

    // Check to see if the two strings are different
    // If so, assign the new string
    // Otherwise, do nothing and print a message.

```

(continues on next page)

(continued from previous page)

```
if($GameName != %gameName)
    $GameName = %gameName;
else
    echo("Game name is identical. Doing nothing");

// Check to see if the two strings are identical
// If so, do nothing and print a message.
// Otherwise, assign the new string
if($BattleCry != %battleCry)
    echo("Battle cry is identical. Doing nothing");
else
    $BattleCry = %battleCry;
}
```

## 23.5 Looping Structures

### 23.5.1 Syntax Review

There are two types of used in TorqueScript: for and while loops. A for loop repeats a statement or block of code for a set number of iterations. A while loop on the other hand repeats a statement or block of code as long as an expression given to the while loop remains true.

#### for Loop Syntax:

```
for(expression0; expression1; expression2)
{
    statement(s);
}
```

One way to label the expressions in this syntax are (startExpression; testExpression; countExpression). Each expression is separated by a semi-colon.

#### while Loop Syntax:

```
while(expression)
{
    statements;
}
```

As soon as the expression is met, the while loop will terminate.

### 23.5.2 Example

#### For Loop:

```
for(%count = 0; %count < 3; %count++)
{
    echo(%count);
}
```

OUTPUT:  
0

(continues on next page)

(continued from previous page)

```
1
2
```

**While Loop:**

```
%countLimit = 0;

while(%countLimit <= 5)
{
    echo("Still in loop");
    %count++;
}
echo("Loop was terminated");

OUTPUT:
Still in loop
Still in loop
Still in loop
Still in loop
Still in loop
Still in loop
Loop was terminated
```

### 23.5.3 Creating the Script

First, we need to create a new script:

1. Navigate to your project's **game/scripts/client** directory.
2. Create a new script file called "loops". In Torsion, right click on the directory, click the "New Script" option, then name your script. On Windows or OS X, create a new text file and change the extension to .cs.
3. Open your new script using a text editor or Torsion.

Before writing any actual script code, we should go ahead and tell the game it should load the script. Open **game/scripts/client/init.cs**. Scroll down to the **initClient** function. Under the **// Client scripts** section, add the following:

Execute our new script:

```
exec("./loops.cs");
```

We will start with a very basic loop. Add the following to your script:

```
// Print 0 -> %count in the console
function printNumbers(%count)
{
    for(%i = 0; %i < %count; %i++)
    {
        echo(%i);
    }
}
```

The above function takes in a single argument (**%count**). The **for(...)** loop uses three expressions. The first is declaration expression. This is the setup for the loop. In this example, the **iterator** is defined. The iterator is the variable that will change each loop.



The second expression sets up the condition that will cause the loop to terminate. In the above code, when the iterator is no longer less than the `%count` variable, the loop will end. Finally, the third expression is the logic that occurs after each loop. In our example, we increment the count of our iterator.

The main logic is enclosed in brackets after the loop declaration. In above code, the iterator (`%i`) is printed to the console each loop. To test your new script:

1. Save the script
2. Run your game
3. Open the console by pressing the tilde (~) key
4. Type the following, and press enter:

```
printNumbers(10);
```

Your output should look like the following:

```
0
1
2
3
4
5
6
7
8
9
```

As expected, the iterator is printed to the console then incremented by 1. Notice that it stops when it gets to 9, even though 10 was passed in. Look at the second expression's logic again:

```
%i < %count;
```

When `%i` reaches 10, then it is equal to the `%count` passed in which is also 10. 10 is not less than 10. As soon as that expression failed, the loop terminated. To get the full ten count, modify the function to use a different logic check:

```
function printNumbers(%count)
{
    for(%i = 0; %i <= %count; %i++)
    {
        echo(%i);
    }
}
```

Now, when you call the following code in the console:

```
printNumbers(10);
```

Your output should be:

```
0
1
2
3
4
5
6
```

(continues on next page)

(continued from previous page)

```
7
8
9
10
```

You can apply different modifiers to your iterator. You do not always have to use an incremental counter. Add the following function to your script:

```
// Print %startCount -> 0 in the console
function countdown(%startCount)
{
    for(%i = %startCount; %i >= 0; %i--)
    {
        echo(%i);
    }
}
```

Save and run. Now you can see a countdown from a base number, as the following shows:

```
countdown(5);
```

**Output:**

```
5
4
3
2
1
0
```

An important keyword to remember when working with `for(...)` loops is **continue**. The `continue` keyword will cause a loop to immediately skip to the next iteration, similar to how the `return` keyword works in a function. Add the following function to see it work:

```
// Print 0 -> %count, except %skipNumber, in the console
function skipCount(%count, %skipNumber)
{
    for(%i = 0; %i <= %count; %i++)
    {
        if(%i == %skipNumber)
            continue;

        echo(%i);
    }
}
```

In the above code, when the iterator (`%i`) exactly matches the `%skipNumber` variable, the loop immediately goes to the next iteration. This ignores the `echo(...)` command on the next line. Try calling this in the console:

```
skipCount(5, 4);
```

The output should be:

```
0
1
2
```

(continues on next page)

(continued from previous page)

```
3
5
```

Instead of terminating soon as the iterator reached 4, a `continue` keyword was used to skip to the next loop iteration. If a less complex loop is desired, the **while(...)** structure will be handy.

Add the following function to your script:

```
// Increase %count incrementally until it is no
// longer less than %breakNumber
function whileExample(%count, %breakNumber)
{
    // While the count is less than the breaknumber
    while(%count < %breakNumber)
    {
        // Print the count
        echo(%count);

        // Increase the count
        %count++;
    }
}
```

In this new function, the loop will check the expression in the parenthesis each time it completes an iteration. The body of the loop, contained in the brackets, simply prints the `%count` variable and then increases. You must be careful with loops, especially **while(...)** structures. The wrong use of variables can result in an infinite loop which will freeze your game.

**Break** is another keyword that affects looping structures. It will immediately terminate the loop. The following function shows proper use of a while loop avoiding infinite cycling:

```
// Increase %iterator until it is equal to
// %conditional. When it is, break out of
// the infinite loop
function breakOut(%iterator, %conditional)
{
    // If iterator is less than conditional
    // we will be stuck in an infinite loop
    // Error out and exit function.
    if(%iterator > %conditional)
    {
        error("Iterator is greater than conditional, try again");
        return;
    }

    // Loop infinitely until a condition is met
    while(true)
    {
        // Condition has been met, break out.
        if(%iterator == %conditional)
            break;

        echo(%iterator);

        %iterator++;
    }
}
```

Before the loop even starts, an `if(...)` check is made to make sure the variables used by the loop will insure a proper break. The goal of the loop is to continue iterating until the `%iterator` variable is equal to the `%conditional`.

The **`while(true)`** syntax creates the “infinite” loop. However, it will not loop infinitely since a **`break`** keyword is used. Once the `%iterator` is equal to the `%conditional`, a break is called. Otherwise, the `%iterator` is printed to the console and then increased.

To see the output, call the following in the console (pressing enter after each line):

```
breakOut(10,1);
breakOut(10,10);
breakOut(0, 10);
```

#### Output:

Iterator is greater than conditional, try again:

```
0
1
2
3
4
5
6
7
8
9
```

The first call gives you the error message. The second call immediately causes the loop to terminate since the two variables are already equal. The last call provides the proper output of the function.

The last concept we will cover is **nested loops**. These are loops within other loops. For the next example, the terminology should be addressed first. The first loop is identical to the structures you have created in the past.

The nested loop is declared inside the first loop. Remember, it is important to be smart about your variable names. You can name your iterators anything you want, such as using `%iterator` instead of `%i`. If you go with the longer name, then it would make sense to name your second iterator something like “`%iteratorTwo`”.

The naming convention for loop iterators is preferential. The use of `%i` typically stands for iterator. In quite a few programming primers (such as the ones this writer has read), the second iterator is often named `%j`. For these simple examples, you can get away with this. In more complex or critical loops, you might want to name your iterators based on what the loop does.

Add the following function:

```
// Run a nested loop
// Print messages, color based on level
function nestedLoops()
{
    // Max iteration for first loop
    %firstCount = 10;

    // Execute first loop %firstCount times
    for(%i = 0; %i < %firstCount; %i++)
    {
        // Print in teal
        warn("Running main loop: " @ %i);
    }
}
```

Run the function in the console, and you should see the following printed in a teal color:

```
Running main loop: 0
Running main loop: 1
Running main loop: 2
Running main loop: 3
Running main loop: 4
Running main loop: 5
Running main loop: 6
Running main loop: 7
Running main loop: 8
Running main loop: 9
```

For the nested loop, we will stick with a pattern. A second count variable should be declared, and the nested loop should perform a similar operation. Modify the function to use this pattern:

```
// Run a nested loop
// Print messages, color based on level
function nestedLoops()
{
    // Max iteration for first loop
    %firstCount = 10;

    // Max iteration for nested loop
    %secondCount = 2;

    // Execute first loop %firstCount times
    for(%i = 0; %i < %firstCount; %i++)
    {
        // Execute nested loop %secondCount times
        for(%j = 0; %j < %secondCount; %j++)
        {
            // Print in red
            error("Running nested loop: " @ %j);
        }
        // Print in teal
        warn("Running main loop: " @ %i);
    }
}
```

Run this function again to see the new output:

```
Running nested loop: 0
Running nested loop: 1
Running main loop: 0
Running nested loop: 0
Running nested loop: 1
Running main loop: 1
Running nested loop: 0
Running nested loop: 1
Running main loop: 2
Running nested loop: 0
Running nested loop: 1
Running main loop: 3
Running nested loop: 0
Running nested loop: 1
Running main loop: 4
Running nested loop: 0
```

(continues on next page)

(continued from previous page)

```
Running nested loop: 1
Running main loop: 5
Running nested loop: 0
Running nested loop: 1
Running main loop: 6
Running nested loop: 0
Running nested loop: 1
Running main loop: 7
Running nested loop: 0
Running nested loop: 1
Running main loop: 8
Running nested loop: 0
Running nested loop: 1
Running main loop: 9
```

Your console output will be color-coded. The main loop output should still be teal, and the nested loop output should be red. Here is the breakdown of the full loop:

```
#. First loop starts
#. Main iterator (%i) starts at 0
#. Nested loop starts
#. Second iterator (%j) starts at 0
#. Print second iterator (0)
#. Increment second iterator
#. Print second iterator (1)
#. End nested loop
#. Print first iterator
#. Increment first loop
#. Go back to step 3, repeat until first loop ends
```

Based on the default values, the nested loop will execute 10 times. Its iterator will reset each time the first loop iterates. Try adjusting the `%firstCount` and `%secondCount` variables to see the varying outputs if you are still trying to understand the concept.

## 23.5.4 Conclusion

This guide covered the basics of looping structures. You will use these often when you need to accomplish repetitive tasks or iterate through lists. Remember the following:

1. If you perform a task more than twice, you might want to use a loop
2. Be smart when naming your iterators and other variables
3. Always perform safety checks for infinite loops

You can download the entire script from this lesson [HERE](#). Save the script as you would any other text file from a website:

```
//-----
// Torque 3D
// Copyright (C) GarageGames, LLC 2011 All Rights Reserved
//-----

// Print 0 -> %count in the console
function printNumbers(%count)
{
```

(continues on next page)



(continued from previous page)

```

    for(%i = 0; %i <= %count; %i++)
    {
        echo(%i);
    }
}

// Print %startCount -> 0 in the console
function countdown(%startCount)
{
    for(%i = %startCount; %i >= 0; %i--)
    {
        echo(%i);
    }
}

// Print 0 -> %count, except %skipNumber, in the console
function skipCount(%count, %skipNumber)
{
    for(%i = 0; %i <= %count; %i++)
    {
        if(%i == %skipNumber)
            continue;

        echo(%i);
    }
}

// Increase %count incrementally until it is no
// longer less than %breakNumber
function whileExample(%count, %breakNumber)
{
    // While the count is less than the breaknumber
    while(%count < %breakNumber)
    {
        // Print the count
        echo(%count);

        // Increase the count
        %count++;
    }
}

// Increase %iterator until it is equal to
// %conditional. When it is, break out of
// the infinite loop
function breakOut(%iterator, %conditional)
{
    // If iterator is less than conditional
    // we will be stuck in an infinite loop
    // Error out and exit function.
    if(%iterator > %conditional)
    {
        error("Iterator is greater than conditional, try again");
        return;
    }

    // Loop infinitely until a condition is met

```

(continues on next page)

(continued from previous page)

```

while(true)
{
    // Condition has been met, break out.
    if(%iterator == %conditional)
        break;

    echo(%iterator);

    %iterator++;
}

// Run a nested loop
// Print messages, color based on level
function nestedLoops()
{
    // Max iteration for first loop
    %firstCount = 10;

    // Max iteration for nested loop
    %secondCount = 2;

    // Execute first loop %firstCount times
    for(%i = 0; %i < %firstCount; %i++)
    {
        // Execute nested loop %secondCount times
        for(%j = 0; %j < %secondCount; %j++)
        {
            // Print in red
            error("Running nested loop: " @ %j);
        }

        // Print in teal
        warn("Running main loop: " @ %i);
    }
}

```

## 23.6 Array Manipulation

### 23.6.1 Syntax Review

Arrays are data structures used to store consecutive values of the same data type. Arrays can be single-dimension or multidimensional:

```

$TestArray[n]      (Single-dimension)
$TestArray[m,n]    (Multidimensional)
$TestArray[m_n]    (Multidimensional)

```

If you have a list of similar variables you wish to store together, try using an array to save time and create cleaner code. The syntax displayed above uses the letters ‘n’ and ‘m’ to represent where you will input the number of elements in an array. The following example shows code that could benefit from an array:

To learn more, read the full array syntax.

## 23.6.2 Example

### Example:

```
$userNames[0] = "Heather";
$userNames[1] = "Nikki";
$userNames[2] = "Mich";

echo($userNames[0]);
echo($userNames[1]);
echo($userNames[2]);
```

## 23.6.3 Creating the Script

First, we need to create a new script:

1. Navigate to your project's **game/scripts/client** directory.
2. Create a new script file called "arrays". In Torsion, right click on the directory, click the "New Script" option, then name your script. On Windows or OS X, create a new text file and change the extension to .cs.
3. Open your new script using a text editor or Torsion.

Before writing any actual script code, we should go ahead and tell the game it should load the script. Open **game/scripts/client/init.cs**. Scroll down to the **initClient** function. Under the **// Client scripts** section, add the following:

### Execute our new script:

```
exec("./arrays.cs");
```

This new script is going to work with two different arrays: \$names and \$board. \$names is a single dimensional array containing strings. \$board is a two dimensional array, also containing strings. The two are not related, but show off different uses of arrays. Let's create the initialization function:

```
// Set up all of the arrays
// with default values
function initArrays()
{
    // Initialize single dimensional array
    // containing a list of names
    $names[0] = "Heather";
    $names[1] = "Nikki";
    $names[2] = "Mich";

    // Initialize two dimensional array
    // containing symbols for a
    // tic-tac-toe game

    // Row one values
    $board[0,0] = "_";
    $board[0,1] = "_";
    $board[0,2] = "_";

    // Row two values
    $board[1,0] = "_";
    $board[1,1] = "_";
```

(continues on next page)

(continued from previous page)

```
$board[1,2] = "_";

// Row three values
$board[2,0] = "_";
$board[2,1] = "_";
$board[2,2] = "_";
}
```

The above code defines the two arrays (\$names and \$board). \$names is given three strings representing people's names. \$board is setup like a tic-tac-toe board. It will be making use of "X"s and "O"s, but for now the blank value is "\_".

Instead of manually calling this function every time the game is run, we can call the function on game initialization. Open **game/scripts/client/init.cs**. Scroll down to the **initClient** function. Under this code:

```
exec("./arrays.cs");
```

Add the following:

```
initArrays();
```

Save the arrays.cs and init.cs scripts. Since there is nothing to see yet, create a function that will print out the values of \$names:

```
// Print out all the values
// in the $names array
function printNames()
{
    // Print each name using
    // hard coded values (0,1,2)
    echo("0:" @ $names[0]);
    echo("1:" @ $names[1]);
    echo("2:" @ $names[2]);
}
```

To test your new script:

1. Save the script
2. Run your game
3. Open the console by pressing the tilde (~) key
4. Type the following, and press enter:

```
printNames();
```

The output is extremely basic. All you will see is the strings held by the array, by index:

```
0: Heather
1: Nikki
2: Mich
```

This is a good start, but what if the array has 1000 elements? An optimization for this function would be to make use of a looping structure. Modify the printNames() function to use the following code:

```
function printNames()
{
```

(continues on next page)

(continued from previous page)

```
// Iterate through the names
// array and print the values
for(%i = 0; %i < 3; %i++)
    echo(%i @ ": " @ $names[%i]);
}
```

Instead of having three (or 1000) echo statements, you only have to script two lines. The above code iterates through the elements of the \$names array using a for(...) loop. To change an individual element, add the following function to your script:

```
// Change the value of an array item
// %id = index to change
// %name = the new value
function setNames(%id, %name)
{
    // Our array only contains three elements:
    // [0] [1] [2]
    // If anything other than 0, 1, or 2 is
    // passed in, inform the user of an error
    if(%id > 2 || %id < 0)
    {
        error("Index " @ %id @ " out of range");
        error("Please use 0 - 2 as the %id");
    }
    else
        $names[%id] = %name;
}
```

To use this function, run the game and open the console. The first variable determines which array index is changing, and the second variable is the new string (name) to use. Example usage:

```
setNames(0, "Brad");
```

If you try to pass in any other numbers besides 0, 1, or 2, you will get an error message letting you know you have tried to access outside of the array bounds. Moving on, the script needs functions for printing, manipulating, and testing the \$board array.

To print out just the values in order, add the following function:

```
// Print out the the values
// in the $board array
function printBoardValues()
{
    // %i loops through rows
    for(%i = 0; %i < 3; %i++)
    {
        // %j loops through columns
        for(%j = 0; %j < 3; %j++)
        {
            // Print the value of the [%i,%j]
            echo "[" @ %i @ "," @ %j @ "]: " @ $board[%i, %j]);
        }
    }
}
```

The above code uses the concept of nested loops. Nested loops are simply loops within other loops. Notice there are two for(...) structures set up. This allows the iteration of each row and column, which is necessary with a

two-dimensional array. Calling this function will result in the following output:

```
[0,0]: _
[0,1]: _
[0,2]: _
[1,0]: _
[1,1]: _
[1,2]: _
[2,0]: _
[2,1]: _
[2,2]: _
```

As you can see, the function prints the current index and the value it contains. Being a tic-tac-toe board, it might help to visualize the board based on value locations. The following function will print the values of \$board in a relative format:

```
// Print tic-tac-toe board
// in a relative format
function printBoard()
{
    // Print out an entire row in 1 echo
    echo($board[0,0] @" "@ $board[0,1] @" "@ $board[0,2]);
    echo($board[1,0] @" "@ $board[1,1] @" "@ $board[1,2]);
    echo($board[2,0] @" "@ $board[2,1] @" "@ $board[2,2]);
}
```

The initial output without changing the values will look like this:

```
_ _ _
_ _ _
_ _ _
```

If you have never played tic-tac-toe, each player takes a turn putting an X or O in one of the board positions. When three X's or O's are lined up, a player wins. The alignment can be three in a row, three in a column, or three diagonally. We can simulate this game play, but we will only work with rows.

We are going to change this function a few times, but we will start with the shell:

```
// Set a specific value in the array
// to an X or O
function setBoardValue(%row, %column, %value)
{
    // Make sure "X" or "O" was passed in
    if(%value != "X" && %value != "O")
    {
        echo("Invalid entry:\nPlease use \'X\' or \'O\'");
        return;
    }
}
```

The user will input a row index (%row), a column index (%column), and a value (%value) represented by an "X" or "O" string. If anything other than a capital X or capital O are passed in, the function will throw an error message and exit. If the function gets past the check, the value is assigned:

```
// Set a specific value in the array
// to an X or O
function setBoardValue(%row, %column, %value)
{
```

(continues on next page)



(continued from previous page)

```
// Make sure "X" or "O" was passed in
if(%value != "X" && %value != "O")
{
    echo("Invalid entry:\nPlease use \'X\' or \'O\'");
    return;
}

// Set the board value
$board[%row, %column] = %value;
}
```

Save the script and run. Call the following functions, in order, to see the results:

```
printBoard();
setBoardValue(0,0,"X");
setBoardValue(0,1,"O");
printBoard();
```

Your output should look like the following:

```
- - -
- - -
- - -

X O _
- - -
- - -
```

To reset back to the default values, you can create a function that iterates through the array:

```
// Set all values of $board
// array back to "nothing"
// In this case, nothing is _
function resetBoard()
{
    // %i loops through rows
    for(%i = 0; %i < 3; %i++)
    {
        // %j loops through columns
        for(%j = 0; %j < 3; %j++)
        {
            // Set value to _
            $board[%i, %j] = "_";
        }
    }
}
```

Now, any normal game will have a victory condition. Enable to win, a row must contain three of the same value. Creating a function for this is quite simple using array access and string comparisons:

```
// Compare the values of each array
// item in a row
// If row contains the same values
// Return true for a victory
// Return false if values are different
function checkForWin()
```

(continues on next page)

(continued from previous page)

```

{
    // Make sure at least the first symbol is X or O
    // Then compare the three values of a row

    // Row 1
    if($board[0,0] != "_" && $board[0,0] $= $board[0,1] && $board[0,1] $= $board[0,2])
        return true;

    // Row 2
    if($board[1,0] != "_" && $board[1,0] $= $board[1,1] && $board[1,1] $= $board[1,2])
        return true;

    // Row 3
    if($board[2,0] != "_" && $board[2,0] $= $board[2,1] && $board[2,1] $= $board[2,2])
        return true;

    return false;
}

```

The `checkForWin()` function will return true if any of the three `if(...)` statements pass. If there is no win condition, the function will return false. In a previous guide, you learned about the `$=` operator. Alternatively, you can use a function to compare two strings: `strcmp(...)`.

The `strcmp(...)` function takes in two string, compares the two, then return a 1 or 0 based on the comparison. If the two strings are the same, it will return a 0. If the two strings are different, it will return a 1.

Example:

```

%string1 = "Hello";
%string2 = "Hello";
%string3 = "World";

// Returns 0
strcmp(%string1, %string2);

// Returns 1
strcmp(%string1, %string3);

```

We can replace the `$=` operators in the `checkForWin()` function using a different set of operators. Comment out the first chunk of code, and replace it with the following:

```

function checkForWin()
{
    // Make sure at least the first symbol is X or O
    // Then compare the three values of a row
    //if($board[0,0] != "_" && $board[0,0] $= $board[0,1] && $board[0,1] $= $board[0,
↪2])
        //return true;
    //
    //if($board[1,0] != "_" && $board[1,0] $= $board[1,1] && $board[1,1] $= $board[1,
↪2])
        //return true;
    //
    //if($board[2,0] != "_" && $board[2,0] $= $board[2,1] && $board[2,1] $= $board[2,
↪2])
        //return true;
}

```

(continues on next page)

(continued from previous page)

```

    if($board[0,0] != "_" && !strcmp($board[0,0], $board[0,1]) && !strcmp($board[0,1],
↪ $board[0,2]))
        return true;

    if($board[1,0] != "_" && !strcmp($board[1,0], $board[1,1]) && !strcmp($board[1,1],
↪ $board[1,2]))
        return true;

    if($board[2,0] != "_" && !strcmp($board[2,0], $board[2,1]) && !strcmp($board[2,1],
↪ $board[2,2]))
        return true;

    return false;
}

```

Let's break down the if(...) statements to see what is going on:

```
if($board[0,0] != "_" &&)
```

The first part checks to see if the row contains a blank entry ("\_"). If this is true, then there is no point checking for anything else. The row does not have three similar values, so the function can move on to check the rest of the rows:

```
!strcmp($board[0,0], $board[0,1])
```

If the first check succeeds, the values of the row's first and second column are compared. If they are the same, a 0 is returned. Instead of catching the return value in a variable and testing it, we can just use the logical NOT (!) operator.

If the first two columns are the same, we can just compare the third column to one of the others. There is no point in making three string comparisons:

```
&& !strcmp($board[0,1], $board[0,2])
```

There are most likely more optimized ways to check for this kind of situation, but the above code demonstrates multiple syntactical approaches and comparisons. We can now have a way to check for a victory condition. Go back into the setBoardValue(...) function and add the win check:

```

function setBoardValue(%row, %column, %value)
{
    // Make sure "X" or "O" was passed in
    if(%value != "X" && %value != "O")
    {
        echo("Invalid entry:\nPlease use \'X\' or \'O\'");
        return;
    }

    // Set the board value
    $board[%row, %column] = %value;

    // Check to see if we have the same
    // three values in a row
    if(checkForWin())
    {
        // Entire row matched
        // Print a victory message
        echo("\n*****");
        echo("*      Win Condition!      *");
    }
}

```

(continues on next page)

(continued from previous page)

```

        echo("*****\n");

        // Print the board
        printBoard();

        // Reset the game
        echo("\nResetting board");
        resetBoard();
    }
}

```

Remember, the `checkForWin()` functions returns a true if the game has been won. The first portion of the code prints a message about the victory. After that, the board is printed to show what row won, and then resets the game.

While this version of the game is very rudimentary, you should be able to expand it by checking for columns and diagonals. There is plenty of room for optimization and more functions to make the game easier. However, this is not necessary to learning a powerful game engine like Torque 3D.

### 23.6.4 Conclusion

This guide covered the concept of arrays, both single and multi-dimensional. Lessons from past guides were also used: string comparisons, logical operators, function declaration and calling, loop structures, etc.

You can download the entire script from this lesson [HERE](#). Save the script as you would any other text file from a website:

```

//-----
// Torque 3D
// Copyright (C) GarageGames, LLC 2011 All Rights Reserved
//-----

// Set up all of the arrays
// with default values
function initArrays()
{
    // Initialize single dimensional array
    // containing a list of names
    $names[0] = "Heather";
    $names[1] = "Nikki";
    $names[2] = "Mich";

    // Initialize two dimensional array
    // containing symbols for a
    // tic-tac-toe game

    // Row one values
    $board[0,0] = "_";
    $board[0,1] = "_";
    $board[0,2] = "_";

    // Row two values
    $board[1,0] = "_";
    $board[1,1] = "_";
    $board[1,2] = "_";

    // Row three values

```

(continues on next page)

(continued from previous page)

```

    $board[2,0] = "_";
    $board[2,1] = "_";
    $board[2,2] = "_";
}

// Print out all the values
// in the $names array
function printNames()
{
    // Iterate through the names
    // array and print the values
    for(%i = 0; %i < 3; %i++)
        echo(%i @ ": " @ $names[%i]);
}

// Change the value of an array item
// %id = index to change
// %name = the new value
function setName(%id, %name)
{
    // Our array only contains three elements:
    // [0] [1] [2]
    // If anything other than 0, 1, or 2 is
    // passed in, inform the user of an error
    if(%id > 2 || %id < 0)
    {
        error("Index " @ %id @ " out of range");
        error("Please use 0 - 2 as the %id");
    }
    else
        $names[%id] = %name;
}

// Print out the the values
// in the $board array
function printBoardValues()
{
    // %i loops through rows
    for(%i = 0; %i < 3; %i++)
    {
        // %j loops through columns
        for(%j = 0; %j < 3; %j++)
        {
            // Print the value of the [%i,%j]
            echo "[" @ %i @ ", " @ %j @ ": " @ $board[%i, %j]);
        }
    }
}

// Print tic-tac-toe board
// in a relative format
function printBoard()
{
    // Print out an entire row in 1 echo
    echo($board[0,0] @ " " @ $board[0,1] @ " " @ $board[0,2]);
    echo($board[1,0] @ " " @ $board[1,1] @ " " @ $board[1,2]);
    echo($board[2,0] @ " " @ $board[2,1] @ " " @ $board[2,2]);
}

```

(continues on next page)

(continued from previous page)

```

}

// Set a specific value in the array
// to an X or O
function setBoardValue(%row, %column, %value)
{
    // Make sure "X" or "O" was passed in
    if(%value != "X" && %value != "O")
    {
        echo("Invalid entry:\nPlease use \'X\' or \'O\'");
        return;
    }

    // Set the board value
    $board[%row, %column] = %value;

    // Check to see if we have the same
    // three values in a row
    if(checkForWin())
    {
        // Entire row matched
        // Print a victory message
        echo("\n*****");
        echo("*      Win Condition!      *");
        echo("*****\n");

        // Print the board
        printBoard();

        // Reset the game
        echo("\nResetting board");
        resetBoard();
    }
}

// Set all values of $board
// array back to "nothing"
// In this case, nothing is _
function resetBoard()
{
    // %i loops through rows
    for(%i = 0; %i < 3; %i++)
    {
        // %j loops through columns
        for(%j = 0; %j < 3; %j++)
        {
            // Set value to _
            $board[%i, %j] = "_";
        }
    }
}

// Compare the values of each array
// item in a row
// If row contains the same values
// Return true for a victory
// Return false if values are different

```

(continues on next page)



(continued from previous page)

```

function checkForWin()
{
    // Make sure at least the first symbol is X or O
    // Then compare the three values of a row
    //if($board[0,0] != "_" && $board[0,0] $= $board[0,1] && $board[0,1] $= $board[0,
↪2])
        //return true;
        //
    //if($board[1,0] != "_" && $board[1,0] $= $board[1,1] && $board[1,1] $= $board[1,
↪2])
        //return true;
        //
    //if($board[2,0] != "_" && $board[2,0] $= $board[2,1] && $board[2,1] $= $board[2,
↪2])
        //return true;

    if($board[0,0] != "_" && !strcmp($board[0,0], $board[0,1]) && !strcmp($board[0,1],
↪ $board[0,2]))
        return true;

    if($board[0,0] != "_" && !strcmp($board[1,0], $board[1,1]) && !strcmp($board[1,1],
↪ $board[1,2]))
        return true;

    if($board[0,0] != "_" && !strcmp($board[2,0], $board[2,1]) && !strcmp($board[2,1],
↪ $board[2,2]))
        return true;

    return false;
}

```

## 23.7 Switch Statements

### 23.7.1 Syntax Review

There are two types of switch statements used in TorqueScript. **switch(...)** is used to compare numerical values and **switch\$(...)** is used to compare strings.

Standard switch statements use numerical values to determine which case to execute:

**switch Syntax:**

```

switch(<numeric expression>)
{
    case value0:
        statements;
    case value1:
        statements;
    case value3:
        statements;
    default:
        statements;
}

```

Switch statements requiring string comparison use the switch\$ syntax.

**switch\$ Syntax:**

```
switch$ (<string expression>)
{
    case "string value 0":
        statements;
    case "string value 1":
        statements;
    ...
    case "string value N":
        statements;
    default:
        statements;
}
```

## 23.7.2 Creating the Script

First, we need to create a new script:

1. Navigate to your project's **game/scripts/client** directory.
2. Create a new script file called “switch”. In Torsion, right click on the directory, click the “New Script” option, then name your script. On Windows or OS X, create a new text file and change the extension to .cs.
3. Open your new script using a text editor or Torsion.

Before writing any actual script code, we should go ahead and tell the game it should load the script. Open **game/scripts/client/init.cs**. Scroll down to the **initClient** function. Under the **// Client scripts** section, add the following:

Execute our new script:

```
exec("./switch.cs");
```

The first function we are going to write will take in a numerical argument. This number will be checked for a specific value, and a message will be printed based on the value comparison. Create the following function in your script:

```
// Print a message to a console based on
// the amount of ammo a weapon has
// %ammoCount - Ammo count (obviously)
function checkAmmoCount(%ammoCount)
{
    // If the ammo is at 0, we are out of ammo
    // If the ammo is at 1, we are at the end of the clip
    // If the ammo is at 100, we have a full clip
    // If the ammo is anything else, we do not care
    if(%ammoCount == 0)
        echo("Out of ammo, time to reload");
    else if(%ammoCount == 1)
        echo("Almost out of ammo, warn user");
    else if(%ammoCount == 100)
        echo("Full ammo count");
    else
        echo("Doing nothing");
}
```

To test your new script:

1. Save the script

2. Run your game
3. Open the console by pressing the tilde (~) key
4. Type the following, press enter after each line:

```
checkAmmoCount(0);  
checkAmmoCount(1);  
checkAmmoCount(100);  
checkAmmoCount(44);
```

Your console output should be the following:

```
Out of ammo, time to reload  
  
Almost out of ammo, warn user  
  
Full ammo count  
  
Do nothing
```

Instead of using four separate if/else checks, we can use a single switch statement to handle all of the cases. Change the `checkAmmoCount(...)` function to use the following code:

```
// Print a message to a console based on  
// the amount of ammo a weapon has  
// %ammoCount - Ammo count (obviously)  
function checkAmmoCount(%ammoCount)  
{  
    // If the ammo is at 0, we are out of ammo  
    // If the ammo is at 1, we are at the end of the clip  
    // If the ammo is at 100, we have a full clip  
    // If the ammo is anything else, we do not care  
    switch(%ammoCount)  
    {  
        case 0:  
            echo("Out of ammo, time to reload");  
        case 1:  
            echo("Almost out of ammo, warn user");  
        case 100:  
            echo("Full ammo count");  
        default:  
            echo("Doing nothing");  
    }  
}
```

The switch is declared using the `switch(%ammoCount){...}` syntax. The test value is kept in the parenthesis, and the cases are defined in the brackets. Each case you wish to check for is defined by the keyword `case`, the value, and a colon (case: 0).

You can write as few or as many lines of TorqueScript code between cases as you need to handle each numerical value. The **default** keyword is used when you want to handle a value that does not have a defined case. Without the default case, any other value besides was is defined as a case will be ignored.

If you test the function as you did previously, you should get the same result:

```
checkAmmoCount(0);  
checkAmmoCount(1);
```

(continues on next page)

(continued from previous page)

```
checkAmmoCount(100);
checkAmmoCount(44);
```

**Result:**

```
Out of ammo, time to reload

Almost out of ammo, warn user

Full ammo count

Do nothing
```

Testing strings in switch statements requires a small syntactical change. There are multiple ways to perform a string comparison. Write the following function in your script:

```
// Check to see if a person's name is
// a known user
// %userName - String containing person's name
function matchNames(%userName)
{
    if(!strcmp(%userName, "Heather"))
        echo("User Found: " @ %userName);
    else if(%userName $= "Mich")
        echo("User Found: " @ %userName);
    else if(%userName $= "Nikki")
        echo("User Found: " @ %userName);
    else
        echo("User " @ %userName @ " not found");
}
```

The above code defines a function which takes in a string as an argument, then performs three separate string comparison to find a result. The first `if(...)` check uses the **strcmp** function to check the `%userName` variable against a static string ("Heather").

The two other checks use the basic `$=` string equality operator. Finally, an `else` statement exists to inform the system that no user was found. Run the script and type the following to test the function:

```
matchNames("Heather");
matchNames("Mich");
matchNames("Nikki");
matchNames("Brad");
```

**Output:**

```
User Found: Heather
User Found: Mich
User Found: Nikki
User Brad not found
```

Instead of four separate `if/else` string comparison statements, a single switch can clean the code up greatly. Replace the `matchNames(...)` function with the following:

```
// Check to see if a person's name is
// a known user
// %userName - String containing person's name
```

(continues on next page)

(continued from previous page)

```
function matchNames(%userName)
{
    switch$(%userName)
    {
        case "Heather":
            echo("User Found: " @ %userName);
        case "Mich":
            echo("User Found: " @ %userName);
        case "Nikki":
            echo("User Found: " @ %userName);
        default:
            echo("User: " @ %userName @ " not found");
    }
}
```

Just like the switch statement used in the `checkAmmoCount(...)` function, the above code starts with the `switch$` keyword. This is followed by the string we are testing, held in the parenthesis. Instead of numerical values, the case keywords are followed by a strings.

In the above example, the case statements are comparing the test (`%userName`) against string literals. String literals are raw text displayed in code between quotations. If you have variables that contain a string value to test against, you can use those instead.

As with a numerical switch statement, you can write your logic in between the case statements.

### 23.7.3 Conclusion

This guide covered the basics of the switch and `switch$` statement structures. When you need to perform one or two logical checks, you will use the basic control statements such as `if(...)`, `if else(...)`, and `else`. When you need a complex control statement handling multiple outcomes based on a value, try using a switch statement instead.

You can download the entire script from this lesson [HERE](#). Save the script as you would any other text file from a website:

```
//-----
// Torque 3D
// Copyright (C) GarageGames, LLC 2011 All Rights Reserved
//-----

// Print a message to a console based on
// the amount of ammo a weapon has
// %ammoCount - Ammo count (obviously)
function checkAmmoCount(%ammoCount)
{
    // If the ammo is at 0, we are out of ammo
    // If the ammo is at 1, we are at the end of the clip
    // If the ammo is at 100, we have a full clip
    // If the ammo is anything else, we do not care
    switch(%ammoCount)
    {
        case 0:
            echo("Out of ammo, time to reload");
        case 1:
            echo("Almost out of ammo, warn user");
        case 100:
            echo("Full ammo count");
```

(continues on next page)

(continued from previous page)

```
        default:
            echo("Doing nothing");
    }
}

// Check to see if a person's name is
// a known user
// %userName - String containing person's name
function matchNames(%userName)
{
    switch$(%userName)
    {
        case "Heather":
            echo("User Found: " @ %userName);
        case "Mich":
            echo("User Found: " @ %userName);
        case "Nikki":
            echo("User Found: " @ %userName);
        default:
            echo("User " @ %userName @ " not found");
    }
}
```

## 23.8 Vectors - TODO





## 24.1 Player Class

### 24.1.1 Detailed Description

A client-controlled player character.

The Player object is the main client-controlled object in an FPS, or indeed, any game where the user is in control of a single character. This class (and the associated datablock, `PlayerData`) allows you to fine-tune the movement, collision detection, animation, and SFX properties of the character. Player derives from `ShapeBase`, so it is recommended to have a good understanding of that class (and its parent classes) as well.

### 24.1.2 Movement

The Player class supports the following modes of movement, known as poses:

1. Standing
2. Sprinting
3. Crouching
4. Prone
5. Swimming

The acceleration, maximum speed, and bounding box for each mode can be set independently using the `PlayerData` datablock. The player will automatically switch between swimming and one of the other 4 ‘dry’ modes when entering/exiting the water, but transitions between the non-swimming modes are handled by controller input (such as holding down a key to begin crouching). `$mvTriggerCount3` activates crouching, while `$mvTriggerCount4` activates being prone.

It is important to set the bounding box correctly for each mode so that collisions with the player remain accurate:



When the player changes its pose a new `PlayerData` callback `onPoseChange()` is called. This is being used as `Armor::onPoseChange()` to modify an animation prefix used by `ShapeBaseImageData` to allow the 1st person arms to change their animation based on pose.

Example:

```
function Armor::onPoseChange(%this, %obj, %oldPose, %newPose)
{
    // Set the script anim prefix to be that of the current pose
    %obj.setImageScriptAnimPrefix( $WeaponSlot, addTaggedString(%newPose) );
}
```

Another feature is the ability to lock out poses for the Player at any time. This is done with `allowCrouch()`, `allowSprinting()` etc. There is even `allowJumping()` and `allowJetJumping()` which aren't actually poses but states. So if for some game play reason the player should not be allowed to crouch right now, that can be disabled. All poses can be allowed with `allowAllPoses()` on the Player class.

The pose lock out mechanism is being used by the weapon script system – see `Weapon::onUse()`. With this system, weapons can prevent the player from going into certain poses. This is used by the deployable turret to lock out sprinting while the turret is the current weapon.

Example:

```
function Weapon::onUse(%data, %obj)
{
    // Default behavior for all weapons is to mount it into the object's weapon
    // slot, which is currently assumed to be slot 0
    if (%obj.getMountedImage($WeaponSlot) != %data.image.getId())
    {
        serverPlay3D(WeaponUseSound, %obj.getTransform(
            %obj.mountImage(%data.image, $WeaponSlot);
        if (%obj.client)
        {
            if (%data.description != "")
                messageClient(%obj.client, 'MsgWeaponUsed', '\c0%1 selected.',
                    %data.description);
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        else
            messageClient(%obj.client, 'MsgWeaponUsed', '\c0Weapon selected');
    }

    // If this is a Player class object then allow the weapon to modify allowed_
    ↪poses
    if (%obj.isInNamespaceHierarchy("Player"))
    {
        // Start by allowing everything
        %obj.allowAllPoses();

        // Now see what isn't allowed by the weapon

        %image = %data.image;

        if (%image.jumpingDisallowed)
            %obj.allowJumping(false);

        if (%image.jetJumpingDisallowed)
            %obj.allowJetJumping(false);

        if (%image.sprintDisallowed)
            %obj.allowSprinting(false);

        if (%image.crouchDisallowed)
            %obj.allowCrouching(false);

        if (%image.proneDisallowed)
            %obj.allowProne(false);

        if (%image.swimmingDisallowed)
            %obj.allowSwimming(false);
    }
}

```

## Sprinting

As mentioned above, sprinting is another pose for the Player class. It defines its own force and max speed in the three directions in PlayerData just like most poses, such as crouch. It is activated using \$mvTriggerCount5 by default which is often connected to Left Shift. When used this way you could treat it just like a standard run – perhaps with the standard pose used for a walk in a RPG.

But sprinting is special in that you can control if a player's movement while sprinting should be constrained. You can place scale factors on strafing, yaw and pitch. These force the player to move mostly in a straight line (or completely if you set them to 0) while sprinting by limiting their motion. You can also choose if the player can jump while sprinting. This is all set up in PlayerData.

Just like other poses, you can define which sequences should be played on the player while sprinting. These sequences are:

1. sprint\_root
2. sprint\_forward
3. sprint\_backward

4. sprint\_side
5. sprint\_right

However, if any of these sequences are not defined for the player, then the standard root, run, back, side and side\_right sequences will be used. The idea here is that the ground transform for these sequences will force them to play faster to give the appearance of sprinting. But if you want the player to do something different than just look like they're running faster – such as holding their weapon against their body – then you'll want to make use of the sprint specific sequences.

Sprint also provides two PlayerData callbacks: onStartSprintMotion() and onStopSprintMotion(). The start callback is called when the player is in a sprint pose and starts to move (i.e. presses the W key). The stop callback is called when either the player stops moving, or they stop sprinting. These could be used for anything, but by default they are tied into the ShapeBaseImageData system. See Armor::onStartSprintMotion() and Armor::onStopSprintMotion(). With ShapeBaseImageData supporting four generic triggers that may be used by a weapon's state machine to do something, the first one is triggered to allow weapons to enter a special sprint state that plays a sprint animation sequence and locks out firing. However, you may choose to do something different.

## Jumping

The Player class supports jumping. While the player is in contact with a surface (and optionally has enough energy as defined by the PlayerData), \$mvTriggerCount2 will cause the player to jump.

## Jetting

The Player class includes a simple jetpack behaviour allowing characters to 'jet' upwards while jumping. The jetting behaviour can be linked to the player's energy level using datablock properties as shown below:

**Example:**

```
datablock PlayerData( JetPlayer )
{
    ...

    jetJumpForce = 16.0 * 90;
    jetJumpEnergyDrain = 10;
    jetMinJumpEnergy = 25;
    jetMinJumpSpeed = 20;
    jetMaxJumpSpeed = 100;
    jetJumpSurfaceAngle = 78;
}
```

This player will not be able to jet if he has less than 25 units of energy, and 10 units will be subtracted each tick.

If PlayerData::jetJumpForce is greater than zero then \$mvTriggerCount1 will activate jetting.

## Falling and Landing

When the player is falling they transition into the "fall" sequence. This transition doesn't occur until the player has reached a particular speed – you don't want the fall sequence to kick in if they've just gone over a small bump. This speed threshold is set by the PlayerData fallingSpeedThreshold field. By default it is set to -10.0.

When the player lands there are two possible outcomes depending on how the player is set up. With the traditional method the "land" sequence has the player start from a standing position and animates into a crouch. The playback speed of this sequence is scaled based on how hard the player hits the ground. Once the land sequence finishes playing the player does a smooth transition back into the root pose (making them effectively stand up).

Starting with 1.2 there is a new method of handling landing. Here the “land” sequence starts with the player crouching on the ground and animates getting back up. This has a look of the player hitting the ground from a fall and slowly standing back up. This new method is used when the `PlayerData` `landSequenceTime` field is given a value greater than zero. This is the amount of time taken for the player to recover from the landing, and is also how long the land sequence will play for. As this has game play ramifications (the player may have movement constraints when landing) this timing is controlled by the `dataBlock` field rather than just the length of time of the land sequence.

Also when using the new land sequence the `PlayerData` `transitionToLand` flag indicates if the player should smoothly transition between the fall sequence and the land sequence. If set to false (the default) then there is no transition and the player appears to immediately go from falling to landing, which is usually the case when mirroring real life.

## Air Control

The player may optionally move itself through the air while jumping or falling. This allows the player to adjust their trajectory while in the air, and is known as air control. The `PlayerData::airControl` property determines what fraction of the player’s normal speed they may move while in the air. By default, air control is disabled (set to 0).

## Hard Impacts

When the player hits something hard it is possible to trigger an impact (such as handled by `Armor::onImpact()`). The `PlayerData` `minImpactSpeed` is the threshold at which falling damage will be considered an impact. Any speed over this parameter will trigger an `onImpact()` call on the `dataBlock`. This allows for small falls to not cause any damage.

The `PlayerData` `minLateralImpactSpeed` is the threshold at which non-falling damage impacts will trigger the callback. This is separate from falling as you may not want a sprinting player that hits a wall to get hurt, but being thrown into a wall by an explosion will.

### 24.1.3 Dismounting

It is possible to have the player mount another object, such as a vehicle, just like any other `SceneObject`. While mounted, `$mvTriggerCount2` will cause the player to dismount.

### 24.1.4 Triggering a Mounted Object

A Player may have other objects mounted to it, with each mounted object assigned to a slot. These Player mounted objects are known as images. See `ShapeBase::mountImage()`. If there is an image mounted to slot 0, `$mvTriggerCount0` will trigger it. If the player dies this trigger is automatically released.

If there is an image mounted to slot 1, `$mvTriggerCount1` will trigger it. Otherwise `$mvTriggerCount1` will be passed along to the image in slot 0 as an alternate fire state.

Below is a list of the system triggers and their default purposes:

Trigger	Default mapping
<code>\$mvTriggerCount0</code>	fire
<code>\$mvTriggerCount1</code>	jet, activate slot 1 mounted item or alt-fire slot 0 item
<code>\$mvTriggerCount2</code>	jump, dismount
<code>\$mvTriggerCount3</code>	crouch
<code>\$mvTriggerCount4</code>	prone
<code>\$mvTriggerCount5</code>	sprint

Additionally, there are now four generic triggers available for use in weapon state machines. To change trigger mapping, see the Member Data Documentation at the end of this article - specifically the *Trigger* data members (eg `Player::imageTrigger0`).

### 24.1.5 The Character Model

The following sequences are used by the `Player` object to animate the character. Not all of them are required, but a model should have at least the `root`, `run`, `back` and `side` animations. And please see the section on Sprinting above for how they are handled when not present.

**root** Looping sequence played when player is standing but not moving.

**run** Looping sequence played when player is running forward.

**back** Looping sequence played when player is running backward.

**side** Looping sequence played when player is running sideways (strafing). The sequence should depict the player moving left. If `side_right` is not present, this sequence will be played backwards in its place.

**side\_right** Looping sequence played when player is running sideways right.

**sprint\_root** Looping sequence played when the player is stationary but in a sprinting mode. If not present then the `root` sequence is used.

**sprint\_forward** Looping sequence played when the player is sprinting and moving forward. If not present then the `run` sequence is used.

**sprint\_backward** Looping sequence played when the player is sprinting and moving backward. If not present then the `back` sequence is used.

**sprint\_side** Looping sequence played when the player is sprinting and moving sideways. The sequence should depict the player moving left. If `crouch_right` is not present, this sequence will be played backwards in its place. If not present then the `side` sequence is used.

**sprint\_right** Looping sequence played when the player is sprinting and moving sideways. If not present then the `side_right` sequence is used.

**crouch\_root** Looping sequence played when player is crouched and not moving.

**crouch\_forward** Looping sequence played when player is crouched and moving forward.

**crouch\_backward** Looping sequence played when player is crouched and moving backward.

**crouch\_side** Looping sequence played when player is crouched and moving sideways. The sequence should depict the player moving left. If `crouch_right` is not present, this sequence will be played backwards in its place.

**crouch\_right** Looping sequence played when player is crouched and moving sideways.

**prone\_root** Looping sequence played when player is prone (lying down) and not moving.

**prone\_forward** Looping sequence played when player is prone (lying down) and moving forward.

**prone\_backward** Looping sequence played when player is prone (lying down) and moving backward.

**swim\_root** Looping sequence played when player is swimming and not moving.

**swim\_forward** Looping sequence played when player is swimming and moving forward.

**swim\_backward** Looping sequence played when player is swimming and moving backward.

**swim\_left** Looping sequence played when player is swimming and moving left. The sequence should depict the player moving left. If `swim_right` is not present, this sequence will be played backwards in its place.

**swim\_right** Looping sequence played when player is swimming and moving right.



**fall** Sequence played when player is falling.

**jump** Sequence played when player has jumped while moving.

**standjump** Sequence played when player has jumped from a standing start.

**land** Sequence played when player lands after falling.

**jet** Looping sequence played when player is jetting.

**head** Sequence to control vertical head movement (for looking) (start=full up, end=full down).

**headside** Sequence to control horizontal head movement (for looking) (start=full left, end=full right).

**look** Sequence to control vertical arm movement (for looking) (start=full up, end=full down).

**light\_recoil** Sequence played when the player is firing a light weapon. (Based on ShapeBaseImageData)

**medium\_recoil** Sequence played when player is firing a medium weapon. (Based on ShapeBaseImageData)

**heavy\_recoil** Sequence played when player is firing a heavy weapon (Based on ShapeBaseImageData).

**deathN** Sequence played when player has been killed (a random one of these will play). N is an integer from 1 to 11.

## Mounted Image Controlled 3rd Person Animation

A player's 3rd person action animation sequence selection may be modified based on what images are mounted on the player. When mounting a ShapeBaseImageData, the image's imageAnimPrefix field is used to control this. If this is left blank (the default) then nothing happens to the 3rd person player – all of the sequences play as defined. If it is filled with some text (best to keep it to letters and numbers, with no spaces) then that text is added to the action animation sequence name and looked up on the player shape. For example:

A rifle ShapeBaseImageData is mounted to the player in slot 0. The rifle's datablock doesn't have an imageAnimPrefix defined, so the 3rd person player will use the standard action animation sequence names. i.e. "root", "run", "back", "crouch\_root", etc.

Now a pistol ShapeBaseImageData is mounted to the player in slot 0. The pistol's datablock has imageAnimPrefix = "pistol". Now the "**pistol**" (underscore is added by the system) prefix is added to each of the action animation sequence names when looking up what to play on the player's shape. So the Player class will look for "pistol\_root", "pistol\_run", "pistol\_back", "pistol\_crouch\_root", etc. If any of these new prefixed names are not found on the player's shape, then we fall back to the standard action animation sequence names, such as "root", "run", etc.

In all of our T3D examples the player only mounts a single image. But Torque allows up to four images to be mounted at a time. When more than one image is mounted then the engine adds all of the prefixes together when searching for the action animation sequence name. If that combined name is not found then the engine starts removing prefixes starting with the highest slot down to the lowest slot. For example, if a player is holding a sword (slot 0) and a shield (slot 1) in each hand that are mounted as separate images (and with imageAnimPrefix's of "sword" and "shield" respectively), then the engine will search for the following names while the player is just standing there:

- shield\_sword\_root
- sword\_root
- root

The first one that is found in the above order will be used.

Another example: If the player has a jet pack (slot 3 with a prefix of "jetpack") and two pistols being used akimbo style (slots 1 and 0, both with a prefix of "laserpistol") with slot 2 left open for a helmet (which is skipped as it doesn't have a prefix), then the following search order would be used:

- jetpack\_laserpistol\_laserpistol\_root
- laserpistol\_laserpistol\_root

- `laserpistol_root`
- `root`

Again, the first one that is found is used.

A player's 3rd person animation may also be modified by the weapon being used. In T3D 1.1 there are the three recoil sequences that may be triggered on the 3rd person player by the weapon's state. Starting with T3D 1.2 this becomes more generic (while still supporting the existing recoil sequence). When a `ShapeBaseImageData` state defines a `stateShapeSequence`, that sequence may be played on the player's shape (the new `PlayerData` `allowImageStateAnimation` field must be set to "true" as well). The new `ShapeBaseImageData` state `stateScaleShapeSequence` flag may also be used to indicate if this player animation sequence should have its playback rate scaled to the length of the image's state.

What exactly happens on the player depends on what else has been defined. First, there is the sequence name as passed in from the image. Then there is also the `imageAnimPrefix` as defined by the image. Finally, there is the generic script defined prefix that may be added with `ShapeBase::setImageScriptAnimPrefix()` – we're using this to pass along the current pose, but it could be used for anything. Time for an example. We want to throw a grenade that we're holding (mounted in slot 0). The weapon's state that does this has `stateShapeSequence` set to "throw". The grenade image itself has an `imageAnimPrefix` defined as "fraggrenade". Finally, the player is crouching, so `Armor::onPoseChange()` sets the script prefix to "crouch". The final search order goes like this:

- `fraggrenade_crouch_throw`
- `fraggrenade_throw`
- `crouch_throw`
- `throw`

The first of those sequences that is found is played as a new thread on the 3rd person player. As with recoil, only one of these 3rd person animation threads may be active at a time. If an image in another slot also asks to play a 3rd person sequence, the most recent request is what will play.

## First Person Arms

Games that have the player hold a weapon in a 1st person view often let you see the player's arms and hands holding that weapon. Rather than requiring you to build the art for all possible combinations of character arms and weapons, T3D allows you to mix and match shapes and animation sequences.

1st person arms are an optional client-side only effect and are not used on the server. The arms are a separate shape from the normal 3rd person player shape. You reference the arms using the `PlayerData` "shapeNameFP" array. It is an array as we support up to four mounted images therefore we support up to four arm shapes. However, for T3D 1.2 our examples only make use of a single set of arms for the first mounting slot as our example soldier holds a single weapon at a time.

As the arms are just regular DAE/DTS files they may get their animation sequences from anywhere. For the included 1.2 art path (see the soldier in the template projects) we decided that their sequences should come from the weapons themselves. This means that the weapons include all of the bones/nodes needed to animate the arms, but none of the arm geometry. If you take a look at `art/shapes/actors/Soldier/FP/FP_SoldierArms.cs` you'll see the external animation sequence references for each of the possible weapons.

As each weapon may require its own set of animation sequences (i.e. a different idle sequence for a pistol vs. a rifle) starting with T3D 1.2 a new `ShapeBaseImageData` field now exists: `imagePrefixFP`. If this field is defined for the mounted image then it is added to the sequence name as given in the current weapon state in the form of "prefix\_sequence" (the underscore is added by the system). For example, the Lurker rifle has an `imagePrefixFP` of "Rifle". The Lurker's Ready state calls the idle sequence, so the arms will attempt to play the "Rifle\_idle" sequence and if not found, they will play the "idle" sequence.

The advantage of having the prefix defined within the datablock and not making it part of the sequence names referenced directly in the weapon state machine is that you can do something like this:

Example:

```
datablock ShapeBaseImageData(Pistol1Image)
{
    imageAnimPrefixFP = "Pistol1";
    ...other data here...
    ...weapon state machine here...
};

datablock ShapeBaseImageData(Pistol2Image : Pistol1Image)
{
    imageAnimPrefixFP = "Pistol2";
};
```

You could define a new pistol (Pistol2Image) that uses the exact same state machine as Pistol1Image, but could use a slightly different set of animation sequences with a prefix of “Pistol2”.

As was previously discussed with 3rd person animation above, a script-based modifier may also be added when looking up the sequence name for the arms. This is currently used to pass along the player’s pose so the arm’s idle sequence could have a swimming motion when in the swim pose, for example. And as with images, the arms sequence name look up uses the following order to find a sequence to play, with the first one found being used:

- ShapeBaseImageDataPrefix\_ScriptPrefix\_WeaponStateSequence
- ShapeBaseImageDataPrefix\_WeaponStateSequence
- ScriptPrefix\_WeaponStateSequence
- WeaponStateSequence

Finally, the arms support an “ambient” sequence that may be used for anything and will always play, if it is defined in the arm’s shape.

### Example PlayerData Datablock

An example of a player datablock appears below:

Example:

```
datablock PlayerData(DefaultPlayerData)
{
    renderFirstPerson = false;

    computeCRC = false;

    // Third person shape
    shapeFile = "art/shapes/actors/Soldier/soldier_rigged.dae";
    cameraMaxDist = 3;
    allowImageStateAnimation = true;

    // First person arms
    imageAnimPrefixFP = "soldier";
    shapeNameFP[0] = "art/shapes/actors/Soldier/FP/FP_SoldierArms.DAE";

    canObserve = 1;
    cmdCategory = "Clients";
```

(continues on next page)

(continued from previous page)

```
cameraDefaultFov = 55.0;
cameraMinFov = 5.0;
cameraMaxFov = 65.0;

debrisShapeName = "art/shapes/actors/common/debris_player.dts";
debris = playerDebris;

throwForce = 30;

aiAvoidThis = 1;

minLookAngle = "-1.2";
maxLookAngle = "1.2";
maxFreelookAngle = 3.0;

mass = 120;
drag = 1.3;
maxdrag = 0.4;
density = 1.1;
maxDamage = 100;
maxEnergy = 60;
repairRate = 0.33;
energyPerDamagePoint = 75;

rechargeRate = 0.256;

runForce = 4320;
runEnergyDrain = 0;
minRunEnergy = 0;
maxForwardSpeed = 8;
maxBackwardSpeed = 6;
maxSideSpeed = 6;

sprintForce = 4320;
sprintEnergyDrain = 0;
minSprintEnergy = 0;
maxSprintForwardSpeed = 14;
maxSprintBackwardSpeed = 8;
maxSprintSideSpeed = 6;
sprintStrafeScale = 0.25;
sprintYawScale = 0.05;
sprintPitchScale = 0.05;
sprintCanJump = true;

crouchForce = 405;
maxCrouchForwardSpeed = 4.0;
maxCrouchBackwardSpeed = 2.0;
maxCrouchSideSpeed = 2.0;

maxUnderwaterForwardSpeed = 8.4;
maxUnderwaterBackwardSpeed = 7.8;
maxUnderwaterSideSpeed = 7.8;

jumpForce = "747";
jumpEnergyDrain = 0;
minJumpEnergy = 0;
```

(continues on next page)

(continued from previous page)

```

jumpDelay = "15";
airControl = 0.3;

fallingSpeedThreshold = -6.0;

landSequenceTime = 0.33;
transitionToLand = false;
recoverDelay = 0;
recoverRunForceScale = 0;

minImpactSpeed = 10;
minLateralImpactSpeed = 20;
speedDamageScale = 0.4;

boundingBox = "0.65 0.75 1.85";
crouchBoundingBox = "0.65 0.75 1.3";
swimBoundingBox = "1 2 2";
pickupRadius = 1;

// Damage location details
boxHeadPercentage      = 0.83;
boxTorsoPercentage     = 0.49;
boxHeadLeftPercentage  = 0.30;
boxHeadRightPercentage = 0.60;
boxHeadBackPercentage  = 0.30;
boxHeadFrontPercentage = 0.60;

// Foot Prints
decalOffset = 0.25;

footPuffEmitter = "LightPuffEmitter";
footPuffNumParts = 10;
footPuffRadius = "0.25";

dustEmitter = "LightPuffEmitter";

splash = PlayerSplash;
splashVelocity = 4.0;
splashAngle = 67.0;
splashFreqMod = 300.0;
splashVelEpsilon = 0.60;
bubbleEmitTime = 0.4;
splashEmitter[0] = PlayerWakeEmitter;
splashEmitter[1] = PlayerFoamEmitter;
splashEmitter[2] = PlayerBubbleEmitter;
mediumSplashSoundVelocity = 10.0;
hardSplashSoundVelocity = 20.0;
exitSplashSoundVelocity = 5.0;

// Controls over slope of runnable/jumpable surfaces
runSurfaceAngle = 38;
jumpSurfaceAngle = 80;
maxStepHeight = 0.35; //two meters
minJumpSpeed = 20;
maxJumpSpeed = 30;

horizMaxSpeed = 68;

```

(continues on next page)

(continued from previous page)

```

horizResistSpeed = 33;
horizResistFactor = 0.35;

upMaxSpeed = 80;
upResistSpeed = 25;
upResistFactor = 0.3;

footstepSplashHeight = 0.35;

// Footstep Sounds
FootSoftSound      = FootLightSoftSound;
FootHardSound      = FootLightHardSound;
FootMetalSound     = FootLightMetalSound;
FootSnowSound      = FootLightSnowSound;
FootShallowSound   = FootLightShallowSplashSound;
FootWadingSound    = FootLightWadingSound;
FootUnderwaterSound = FootLightUnderwaterSound;

FootBubblesSound   = FootLightBubblesSound;
movingBubblesSound = ArmorMoveBubblesSound;
waterBreathSound   = WaterBreathMaleSound;

impactSoftSound    = ImpactLightSoftSound;
impactHardSound    = ImpactLightHardSound;
impactMetalSound   = ImpactLightMetalSound;
impactSnowSound    = ImpactLightSnowSound;

impactWaterEasy    = ImpactLightWaterEasySound;
impactWaterMedium  = ImpactLightWaterMediumSound;
impactWaterHard    = ImpactLightWaterHardSound;

groundImpactMinSpeed    = "45";
groundImpactShakeFreq   = "4.0 4.0 4.0";
groundImpactShakeAmp    = "1.0 1.0 1.0";
groundImpactShakeDuration = 0.8;
groundImpactShakeFalloff = 10.0;

exitingWater           = ExitingWaterLightSound;

observeParameters = "0.5 4.5 4.5";
class = "armor";

cameraMinDist = "0";
DecalData = "PlayerFootprint";

// Allowable Inventory Items
mainWeapon = Lurker;

maxInv[Lurker] = 1;
maxInv[LurkerClip] = 20;

maxInv[LurkerGrenadeLauncher] = 1;
maxInv[LurkerGrenadeAmmo] = 20;

maxInv[Ryder] = 1;
maxInv[RyderClip] = 10;

```

(continues on next page)

(continued from previous page)

```

maxInv[ProxMine] = 5;

maxInv[DeployableTurret] = 5;

// available skins (see materials.cs in model folder)
availableSkins = "base DarkBlue DarkGreen  LightGreen  Orange  Red  Teal
Violet  Yellow";
};

```

## Member Function Documentation

`void Player::allowAllPoses()`

Allow all poses a chance to occur.

This method resets any poses that have manually been blocked from occurring. This includes the regular pose states such as sprinting, crouch, being prone and swimming. It also includes being able to jump and jet jump. While this is allowing these poses to occur it doesn't mean that they all can due to other conditions. We're just not manually blocking them from being allowed.

`void Player::allowCrouching(bool state)`

Set if the Player is allowed to crouch.

The default is to allow crouching unless there are other environmental concerns that prevent it. This method is mainly used to explicitly disallow crouching at any time.

**Parameters** `state` – Set to true to allow crouching, false to disable it.

`void Player::allowJetJumping(bool state)`

Set if the Player is allowed to jet jump.

The default is to allow jet jumping unless there are other environmental concerns that prevent it. This method is mainly used to explicitly disallow jet jumping at any time.

**Parameters** `state` – Set to true to allow jet jumping, false to disable it.

`void Player::allowJumping(bool state)`

Set if the Player is allowed to jump.

The default is to allow jumping unless there are other environmental concerns that prevent it. This method is mainly used to explicitly disallow jumping at any time.

**Parameters** `state` – Set to true to allow jumping, false to disable it.

`void Player::allowProne(bool state)`

Set if the Player is allowed to go prone.

The default is to allow being prone unless there are other environmental concerns that prevent it. This method is mainly used to explicitly disallow going prone at any time.

**Parameters** `state` – Set to true to allow being prone, false to disable it.

`void Player::allowSprinting(bool state)`

Set if the Player is allowed to sprint.

The default is to allow sprinting unless there are other environmental concerns that prevent it. This method is mainly used to explicitly disallow sprinting at any time.

**Parameters** `state` – Set to true to allow sprinting, false to disable it.



void `Player::allowSwimming` (bool *state*)

Set if the Player is allowed to swim.

The default is to allow swimming unless there are other environmental concerns that prevent it. This method is mainly used to explicitly disallow swimming at any time.

**Parameters** *state* – Set to true to allow swimming, false to disable it.

bool `Player::checkDismountPoint` (Point3F *oldPos*, Point3F *pos*)

Check if it is safe to dismount at this position.

Internally this method casts a ray from *oldPos* to *pos* to determine if it hits the terrain, an interior object, a water object, another player, a static shape, a vehicle (excluding the one currently mounted), or physical zone. If this ray is in the clear, then the player's bounding box is also checked for a collision at the *pos* position. If this displaced bounding box is also in the clear, then `checkDismountPoint()` returns true.

**Parameters**

- **oldPos** – The player's current position.
- **pos** – The dismount position to check.

**Returns** True if the dismount position is clear, false if not

---

**Note:** The player must be already mounted for this method to not assert.

---

void `Player::clearControlObject` ()

Clears the player's current control object.

**Returns** Control to the player. This internally calls `Player::setControlObject(0)`.

**Example:**

```
%player.clearControlObject();
echo(%player.getControlObject()); //<-- Returns 0, player assumes control
%player.setControlObject(%vehicle);
echo(%player.getControlObject()); //<-- Returns %vehicle, player controls vehicle_
↪now.
```

---

**Note:** If the player does not have a control object, the player will receive all moves from its `GameConnection`. If you're looking to remove control from the player itself (i.e. stop sending moves to the player) use `GameConnection::setControlObject()` to transfer control to another object, such as a camera.

---

int `Player::getControlObject` ()

Get the current object we are controlling.

**Returns** ID of the `ShapeBase` object we control, or 0 if not controlling an object.

string `Player::getDamageLocation` (Point3F *pos*)

Get the named damage location and modifier for a given world position.

**Parameters** *pos* – A world position for which to retrieve a body region on this player.

**Returns** A string containing two words (space separated strings), where the first is a location and the second is a modifier.

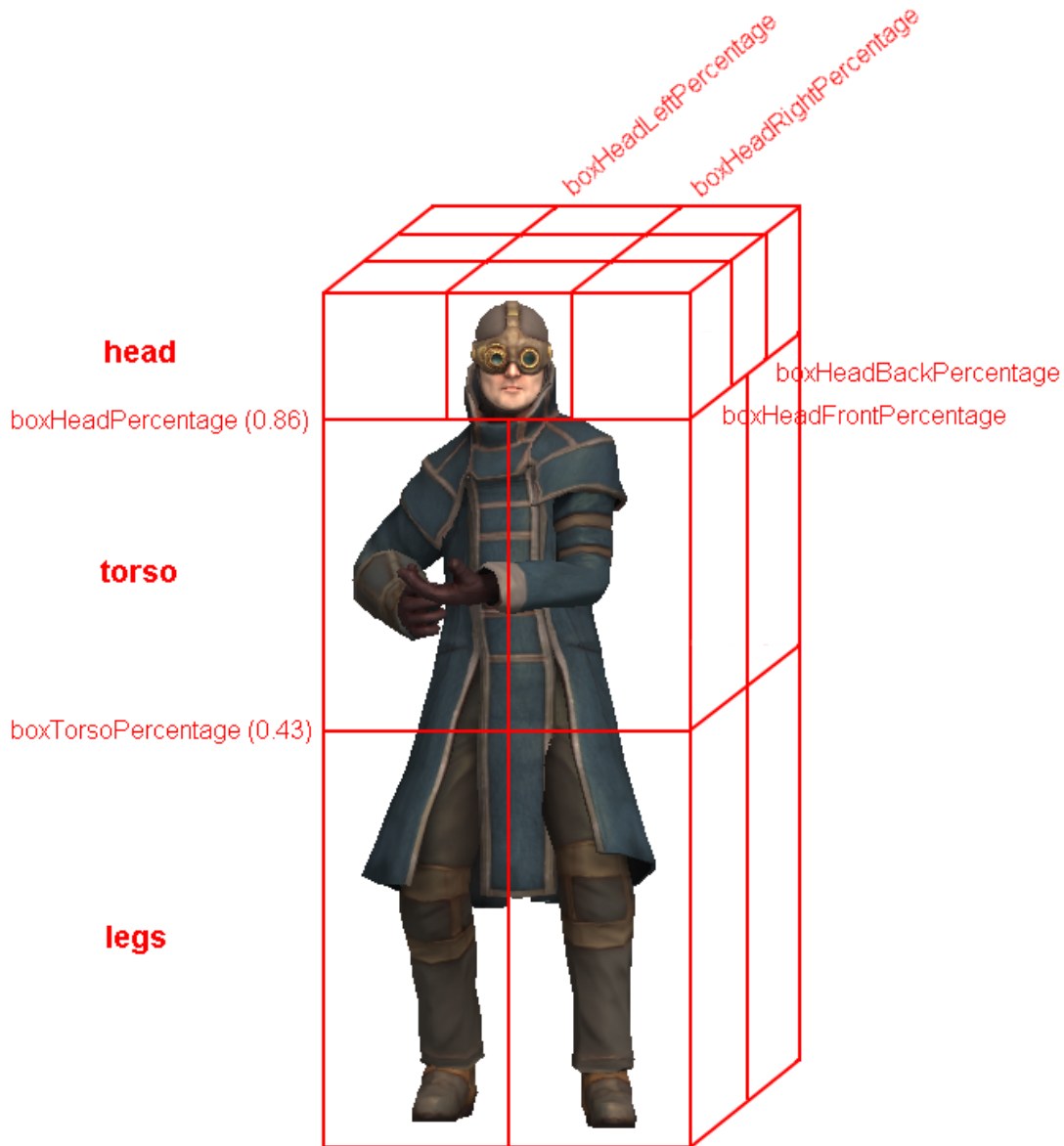
---

**Note:** This method will not return an accurate location when the player is prone or swimming.

---

While you may pass in any world position and `getDamageLocation()` will provide a best-fit location, you should be aware that this can produce some interesting results. For example, any position that is above `PlayerData::boxHeadPercentage` will be considered a ‘head’ hit, even if the world position is high in the sky. Therefore it may be wise to keep the passed in point to somewhere on the surface of, or within, the Player’s bounding volume.

The Player object can simulate different hit locations based on a pre-defined set of `PlayerData` defined percentages. These hit percentages divide up the Player’s bounding box into different regions. The diagram below demonstrates how the various `PlayerData` properties split up the bounding volume:



Possible locations:

- head
- torso
- legs

Head modifiers:

- left\_back
- middle\_back
- right\_back
- left\_middle
- middle\_middle
- right\_middle
- left\_front
- middle\_front
- right\_front

Legs/Torso modifiers:

- front\_left
- front\_right
- back\_left
- back\_right

`int Player::getNumDeathAnimations()`

Get the number of death animations available to this player.

Death animations are assumed to be named death1-N using consecutive indices.

`string Player::getPose()`

Get the name of the player's current pose.

The pose is one of the following:

- Stand - Standard movement pose.
- Sprint - Sprinting pose.
- Crouch - Crouch pose.
- Prone - Prone pose.
- Swim - Swimming pose.

**Returns** The current pose; one of: "Stand", "Sprint", "Crouch", "Prone", "Swim"

`string Player::getState()`

Get the name of the player's current state.

The state is one of the following:

- Dead - The Player is dead.
- Mounted - The Player is mounted to an object such as a vehicle.
- Move - The Player is free to move. The usual state.
- Recover - The Player is recovering from a fall. See `PlayerData::recoverDelay`.

**Returns** The current state; one of: "Dead", "Mounted", "Move", "Recover"

`bool Player::setActionThread(string name, bool hold = false, bool fsp = true)`

Set the main action sequence to play for this player.

**Parameters**

- **name** – Name of the action sequence to set.
- **hold** – Set to false to get a callback on the datablock when the sequence ends (`PlayerData::animationDone()`). When set to true no callback is made.
- **fsp** – True if first person and none of the spine nodes in the shape should animate. False will allow the shape's spine nodes to animate.

**Returns** True if succesful, false if failed.

The spine nodes for the Player's shape are named as follows:

- Bip01 Pelvis
- Bip01 Spine
- Bip01 Spine1
- Bip01 Spine2
- Bip01 Neck
- Bip01 Head

You cannot use `setActionThread()` to have the Player play one of the motion determined action animation sequences. These sequences are chosen based on how the Player moves and the Player's current pose. The names of these sequences are:

- root
- run
- side
- side\_right
- crouch\_root
- crouch\_forward
- crouch\_backward
- crouch\_side
- crouch\_right
- prone\_root
- prone\_forward
- prone\_backward
- swim\_root
- swim\_forward
- swim\_backward
- swim\_left
- swim\_right
- fall
- jump
- standjump

- land
- jet

If the player moves in any direction then the animation sequence set using this method will be cancelled and the chosen motion-based sequence will take over. This makes great for times when the Player cannot move, such as when mounted, or when it doesn't matter if the action sequence changes, such as waving and saluting.

**Example:**

```
// Place the player in a sitting position after being mounted
%player.setActionThread( "sitting", true, true );
```

**bool Player::setArmThread** (string *name*)

Set the sequence that controls the player's arms (dynamically adjusted to match look direction).

**Parameters** *name* – Name of the sequence to play on the player's arms.

**Returns** true if successful, false if failed.

---

**Note:** By default the 'look' sequence is used, if available.

---

**bool Player::setControlObject** (ShapeBase *obj*)

Set the object to be controlled by this player.

It is possible to have the moves sent to the Player object from the GameConnection to be passed along to another object. This happens, for example when a player is mounted to a vehicle. The move commands pass through the Player and on to the vehicle (while the player remains stationary within the vehicle). With setControlObject() you can have the Player pass along its moves to any object. One possible use is for a player to move a remote controlled vehicle. In this case the player does not mount the vehicle directly, but still wants to be able to control it.

**Parameters** *obj* – Object to control with this player.

**Returns** True if the object is valid, false if not

## Member Data Documentation

**int Player::crouchTrigger** [static]

The move trigger index used for player crouching.

**int Player::imageTrigger0** [static]

The move trigger index used to trigger mounted image 0.

**int Player::imageTrigger1** [static]

The move trigger index used to trigger mounted image 1 or alternate fire on mounted image 0.

**int Player::jumpJetTrigger** [static]

The move trigger index used for player jump jetting.

**int Player::jumpTrigger** [static]

The move trigger index used for player jumping.

**float Player::maxImpulseVelocity** [static]

The maximum velocity allowed due to a single impulse.

**int Player::maxPredictionTicks** [static]

Maximum number of ticks to predict on the client from the last known move obtained from the server.

**int Player::maxWarpTicks [static]**

When a warp needs to occur due to the client being too far off from the server, this is the maximum number of ticks we'll allow the client to warp to catch up.

**float Player::minWarpTicks [static]**

Fraction of tick at which instant warp occurs on the client.

**int Player::proneTrigger [static]**

The move trigger index used for player prone pose.

**bool Player::renderCollision [static]**

Determines if the player's collision mesh should be rendered.

This is mainly used for the tools and debugging.

**bool Player::renderMyItems [static]**

Determines if mounted shapes are rendered or not.

Used on the client side to disable the rendering of all Player mounted objects. This is mainly used for the tools or debugging.

**bool Player::renderMyPlayer [static]**

Determines if the player is rendered or not.

Used on the client side to disable the rendering of all Player objects. This is mainly for the tools or debugging.

**int Player::sprintTrigger [static]**

The move trigger index used for player sprinting.

**int Player::vehicleDismountTrigger [static]**

The move trigger index used to dismount player.

## 24.2 Player Datablock

### 24.2.1 Detailed Description

Defines properties for a Player object.

**See also:** `Player`

### 24.2.2 Member Function Documentation

**void PlayerData::animationDone ( `Player *obj*` )**

Called on the server when a scripted animation completes.

**Parameters** `obj` – The Player object

**See also:** `Player::setActionThread()` for setting a scripted animation and its 'hold' parameter to determine if this callback is used.

**void PlayerData::doDismount ( `Player *obj*` )**

Called when attempting to dismount the player from a vehicle.

It is up to the `doDismount()` method to actually perform the dismount. Often there are some conditions that prevent this, such as the vehicle moving too fast.

**Parameters** `obj` – The Player object

**void PlayerData::onEnterLiquid (Player *obj*, float *coverage*, string *type*)**

Called when the player enters a liquid.

**Parameters**

- **obj** – The Player object
- **coverage** – Percentage of the player's bounding box covered by the liquid
- **type** – The type of liquid the player has entered

`void PlayerData::onEnterMissionArea (Player obj)`  
Called when the player enters the mission area.

**Parameters** **obj** – The Player object

**See also:** MissionArea

`void PlayerData::onLeaveLiquid (Player obj, string type)`  
Called when the player leaves a liquid.

**Parameters**

- **obj** – The Player object
- **type** – The type of liquid the player has left

`void PlayerData::onLeaveMissionArea (Player obj)`  
Called when the player leaves the mission area.

:param obj The Player object

**See also:** MissionArea

`void PlayerData::onPoseChange (Player obj, string oldPose, string newPose)`  
Called when the player changes poses.

**Parameters**

- **obj** – The Player object
- **oldPose** – The pose the player is switching from.
- **newPose** – The pose the player is switching to.

`void PlayerData::onStartSprintMotion (Player obj)`  
Called when the player starts moving while in a Sprint pose.

**Parameters** **obj** – The Player object

`void PlayerData::onStartSwim (Player obj)`  
Called when the player starts swimming.

**Parameters** **obj** – The Player object

`void PlayerData::onStopSprintMotion (Player obj)`  
Called when the player stops moving while in a Sprint pose.

**Parameters** **obj** – The Player object

`void PlayerData::onStopSwim (Player obj)`  
Called when the player stops swimming.

**Parameters** **obj** – The Player object



### 24.2.3 Member Data Documentation

**float PlayerData::airControl**

Amount of movement control the player has when in the air.

This is applied as a multiplier to the player's x and y motion.

**bool PlayerData::allowImageStateAnimation**

Allow mounted images to request a sequence be played on the Player.

When true a new thread is added to the player to allow for mounted images to request a sequence be played on the player through the image's state machine. It is only optional so that we don't create a TSThread on the player if we don't need to.

**Point3F PlayerData::boundingBox**

Size of the bounding box used by the player for collision.

Dimensions are given as "width depth height".

**float PlayerData::boxHeadBackPercentage**

Percentage of the player's bounding box depth that represents the back side of the head.

Used when computing the damage location.

**See also:** Player::getDamageLocation

**float PlayerData::boxHeadFrontPercentage**

Percentage of the player's bounding box depth that represents the front side of the head. Used when computing the damage location.

**See also:** Player::getDamageLocation

**float PlayerData::boxHeadLeftPercentage**

Percentage of the player's bounding box width that represents the left side of the head. Used when computing the damage location.

**See also:** Player::getDamageLocation

**float PlayerData::boxHeadPercentage**

Percentage of the player's bounding box height that represents the head. Used when computing the damage location.

**See also:** Player::getDamageLocation

**float PlayerData::boxHeadRightPercentage**

Percentage of the player's bounding box width that represents the right side of the head. Used when computing the damage location.

**See also:** Player::getDamageLocation

**float PlayerData::boxTorsoPercentage**

Percentage of the player's bounding box height that represents the torso. Used when computing the damage location.

**See also:** Player::getDamageLocation

**float PlayerData::bubbleEmitTime**

Time in seconds to generate bubble particles after entering the water.

**Point3F PlayerData::crouchBoundingBox**

Collision bounding box used when the player is crouching.

**See also:** boundingBox

**float PlayerData::crouchForce**

Force used to accelerate the player when crouching.

**DecalData PlayerData::DecalData**

Decal to place on the ground for player footsteps.

**float PlayerData::decalOffset**

Distance from the center of the model to the right foot.

While this defines the distance to the right foot, it is also used to place the left foot decal as well. Just on the opposite side of the player.

**ParticleEmitterData PlayerData::dustEmitter**

Emitter used to generate dust particles.

---

**Note:** Currently unused.

---

**SFXTrack PlayerData::exitingWater**

Sound to play when exiting the water with velocity  $\geq$  exitSplashSoundVelocity.

**See also:** exitSplashSoundVelocity

**float PlayerData::exitSplashSoundVelocity**

Minimum velocity when leaving the water for the exitingWater sound to play.

**See also:** exitingWater

**float PlayerData::fallingSpeedThreshold**

Downward speed at which we consider the player falling.

**SFXTrack PlayerData::FootBubblesSound**

Sound to play when walking in water and coverage equals 1.0 (fully underwater).

**SFXTrack PlayerData::FootHardSound**

Sound to play when walking on a surface with Material footstepSoundId 1.

**SFXTrack PlayerData::FootMetalSound**

Sound to play when walking on a surface with Material footstepSoundId 2.

**ParticleEmitterData PlayerData::footPuffEmitter**

Particle emitter used to generate footpuffs (particles created as the player walks along the ground).

---

**Note:** The generation of foot puffs requires the appropriate triggeres to be defined in the player's animation sequences. Without these, no foot puffs will be generated.

---

**int PlayerData::footPuffNumParts**

Number of footpuff particles to generate each step.

Each foot puff is randomly placed within the defined foot puff radius. This includes having footPuffNumParts set to one.

**See also:** footPuffRadius

**float PlayerData::footPuffRadius**

Particle creation radius for footpuff particles.

This is applied to each foot puff particle, even if footPuffNumParts is set to one. So set this value to zero if you want a single foot puff placed at exactly the same location under the player each time.

**SFXTrack PlayerData::FootShallowSound**

Sound to play when walking in water and coverage is less than footSplashHeight.

See also: footSplashHeight

**SFXTrack PlayerData::FootSnowSound**

Sound to play when walking on a surface with Material footstepSoundId 3.

**SFXTrack PlayerData::FootSoftSound**

Sound to play when walking on a surface with Material footstepSoundId 0.

**float PlayerData::footstepSplashHeight**

Water coverage level to choose between FootShallowSound and FootWadingSound.

See also:

- FootShallowSound
- FootWadingSound

**SFXTrack PlayerData::FootUnderwaterSound**

Sound to play when walking in water and coverage equals 1.0 (fully underwater).

**SFXTrack PlayerData::FootWadingSound**

Sound to play when walking in water and coverage is less than 1, but > footSplashHeight.

See also: footSplashHeight

**float PlayerData::groundImpactMinSpeed**

Minimum falling impact speed to apply damage and initiate the camera shaking effect.

**Point3F PlayerData::groundImpactShakeAmp**

Amplitude of the camera shake effect after falling.

This is how much to shake the camera.

**float PlayerData::groundImpactShakeDuration**

Duration (in seconds) of the camera shake effect after falling.

This is how long to shake the camera.

**float PlayerData::groundImpactShakeFalloff**

Falloff factor of the camera shake effect after falling.

This is how to fade the camera shake over the duration.

**Point3F PlayerData::groundImpactShakeFreq**

Frequency of the camera shake effect after falling.

This is how fast to shake the camera.

**float PlayerData::hardSplashSoundVelocity**

Minimum velocity when entering the water for choosing between the impactWaterMedium and impactWaterHard sound to play.

See also:

- impactWaterMedium
- impactWaterHard

**float PlayerData::horizMaxSpeed**

Maximum horizontal speed.

---

**Note:** This limit is only enforced if the player's horizontal speed exceeds `horizResistSpeed`.

---

See also:

- `horizResistSpeed`
- `horizResistFactor`

**float PlayerData::horizResistFactor**

Factor of resistance once `horizResistSpeed` has been reached.

See also:

- `horizMaxSpeed`
- `horizResistSpeed`

**float PlayerData::horizResistSpeed**

Horizontal speed at which resistance will take place.

See also:

- `horizMaxSpeed`
- `horizResistFactor`

**caseString PlayerData::imageAnimPrefix**

Optional prefix to all mounted image animation sequences in third person.

This defines a prefix that will be added when looking up mounted image animation sequences while in third person. It allows for the customization of a third person image based on the type of player.

**caseString PlayerData::imageAnimPrefixFP**

Optional prefix to all mounted image animation sequences in first person.

This defines a prefix that will be added when looking up mounted image animation sequences while in first person. It allows for the customization of a first person image based on the type of player.

**SFXTrack PlayerData::impactHardSound**

Sound to play after falling on a surface with Material `footstepSoundId` 1.

**SFXTrack PlayerData::impactMetalSound**

Sound to play after falling on a surface with Material `footstepSoundId` 2.

**SFXTrack PlayerData::impactSnowSound**

Sound to play after falling on a surface with Material `footstepSoundId` 3.

**SFXTrack PlayerData::impactSoftSound**

Sound to play after falling on a surface with Material `footstepSoundId` 0.

**SFXTrack PlayerData::impactWaterEasy**

Sound to play when entering the water with velocity < `mediumSplashSoundVelocity`.

See also: `mediumSplashSoundVelocity`

**SFXTrack PlayerData::impactWaterHard**

Sound to play when entering the water with velocity >= `hardSplashSoundVelocity`.

See also: `hardSplashSoundVelocity`

**SFXTrack PlayerData::impactWaterMedium**

Sound to play when entering the water with velocity >= `mediumSplashSoundVelocity` and < `hardSplashSoundVelocity`.

See also:

- mediumSplashSoundVelocity
- hardSplashSoundVelocity

**float PlayerData::jetJumpEnergyDrain**  
Energy level drained each time the player jet jumps.

See also: jetMinJumpEnergy

---

**Note:** Setting this to zero will disable any energy drain

---

**float PlayerData::jetJumpForce**  
Force used to accelerate the player when a jet jump is initiated.

**float PlayerData::jetJumpSurfaceAngle**  
Angle from vertical (in degrees) where the player can jet jump.

**float PlayerData::jetMaxJumpSpeed**  
Maximum vertical speed before the player can no longer jet jump.

**float PlayerData::jetMinJumpEnergy**  
Minimum energy level required to jet jump.

See also: jetJumpEnergyDrain

**float PlayerData::jetMinJumpSpeed**  
Minimum speed needed to jet jump.

If the player's own z velocity is greater than this, then it is used to scale the jet jump speed, up to jetMaxJumpSpeed.

See also: jetMaxJumpSpeed

**int PlayerData::jumpDelay**  
Delay time in number of ticks ticks between jumps.

**float PlayerData::jumpEnergyDrain**  
Energy level drained each time the player jumps.

See also: minJumpEnergy

---

**Note:** Setting this to zero will disable any energy drain

---

**float PlayerData::jumpForce**  
Force used to accelerate the player when a jump is initiated.

**float PlayerData::jumpSurfaceAngle**  
Angle from vertical (in degrees) where the player can jump.

**bool PlayerData::jumpTowardsNormal**  
Controls the direction of the jump impulse.

When false, jumps are always in the vertical (+Z) direction. When true jumps are in the direction of the ground normal so long as the player is not directly facing the surface. If the player is directly facing the surface, then they will jump straight up.

**float PlayerData::landSequenceTime**  
Time of land sequence play back when using new recover system.

If greater than 0 then the legacy fall recovery system will be bypassed in favour of just playing the player's land sequence. The time to recover from a fall then becomes this parameter's time and the land sequence's playback will be scaled to match.

**See also:** `transitionToLand`

**float `PlayerData::maxBackwardSpeed`**

Maximum backward speed when running.

**float `PlayerData::maxCrouchBackwardSpeed`**

Maximum backward speed when crouching.

**float `PlayerData::maxCrouchForwardSpeed`**

Maximum forward speed when crouching.

**float `PlayerData::maxCrouchSideSpeed`**

Maximum sideways speed when crouching.

**float `PlayerData::maxForwardSpeed`**

Maximum forward speed when running.

**float `PlayerData::maxFreelookAngle`**

Defines the maximum left and right angles (in radians) the player can look in freelook mode.

**float `PlayerData::maxJumpSpeed`**

Maximum vertical speed before the player can no longer jump.

**float `PlayerData::maxLookAngle`**

Highest angle (in radians) the player can look.

---

**Note:** An angle of zero is straight ahead, with positive up and negative down.

---

**float `PlayerData::maxProneBackwardSpeed`**

Maximum backward speed when prone (laying down).

**float `PlayerData::maxProneForwardSpeed`**

Maximum forward speed when prone (laying down).

**float `PlayerData::maxProneSideSpeed`**

Maximum sideways speed when prone (laying down).

**float `PlayerData::maxSideSpeed`**

Maximum sideways speed when running.

**float `PlayerData::maxSprintBackwardSpeed`**

Maximum backward speed when sprinting.

**float `PlayerData::maxSprintForwardSpeed`**

Maximum forward speed when sprinting.

**float `PlayerData::maxSprintSideSpeed`**

Maximum sideways speed when sprinting.

**float `PlayerData::maxStepHeight`**

Maximum height the player can step up.

The player will automatically step onto changes in ground height less than `maxStepHeight`. The player will collide with ground height changes greater than this.

**float `PlayerData::maxTimeScale`**

Maximum time scale for action animations.

If an action animation has a defined ground frame, it is automatically scaled to match the player's ground velocity. This field limits the maximum time scale used even if the player's velocity exceeds it.

**float PlayerData::maxUnderwaterBackwardSpeed**

Maximum backward speed when underwater.

**float PlayerData::maxUnderwaterForwardSpeed**

Maximum forward speed when underwater.

**float PlayerData::maxUnderwaterSideSpeed**

Maximum sideways speed when underwater.

**float PlayerData::mediumSplashSoundVelocity**

Minimum velocity when entering the water for choosing between the impactWaterEasy and impactWaterMedium sounds to play.

See also:

- impactWaterEasy
- impactWaterMedium

**float PlayerData::minImpactSpeed**

Minimum impact speed to apply falling damage.

This field also sets the minimum speed for the onImpact callback to be invoked.

See also: ShapeBaseData::onImpact()

**float PlayerData::minJumpEnergy**

Minimum energy level required to jump.

See also: jumpEnergyDrain

**float PlayerData::minJumpSpeed**

Minimum speed needed to jump.

If the player's own z velocity is greater than this, then it is used to scale the jump speed, up to maxJumpSpeed.

See also: maxJumpSpeed

**float PlayerData::minLateralImpactSpeed**

Minimum impact speed to apply non-falling damage.

This field also sets the minimum speed for the onLateralImpact callback to be invoked.

See also: ShapeBaseData::onLateralImpact()

**float PlayerData::minLookAngle**

Lowest angle (in radians) the player can look.

---

**Note:** An angle of zero is straight ahead, with positive up and negative down.

---

**float PlayerData::minRunEnergy**

Minimum energy level required to run or swim.

See also: runEnergyDrain

**float PlayerData::minSprintEnergy**

Minimum energy level required to sprint.

See also: sprintEnergyDrain



**SFXTrack PlayerData::movingBubblesSound**

Sound to play when in water and coverage equals 1.0 (fully underwater).

Note that unlike `FootUnderwaterSound`, this sound plays even if the player is not moving around in the water.

**string PlayerData::physicsPlayerType**

Specifies the type of physics used by the player.

This depends on the physics module used. An example is 'Capsule'.

---

**Note:** Not current used.

---

**float PlayerData::pickupRadius**

Radius around the player to collide with Items in the scene (on server).

Internally the `pickupRadius` is added to the larger side of the initial bounding box to determine the actual distance, to a maximum of 2 times the bounding box size. The initial bounding box is that used for the root pose, and therefore doesn't take into account the change in pose.

**Point3F PlayerData::proneBoundingBox**

Collision bounding box used when the player is prone (laying down).

**See also:** `boundingBox`

**float PlayerData::proneForce**

Force used to accelerate the player when prone (laying down).

**int PlayerData::recoverDelay**

Number of ticks for the player to recover from falling.

**float PlayerData::recoverRunForceScale**

Scale factor applied to `runForce` while in the recover state.

This can be used to temporarily slow the player's movement after a fall, or prevent the player from moving at all if set to zero.

**bool PlayerData::renderFirstPerson**

Flag controlling whether to render the player shape in first person view.

**float PlayerData::runEnergyDrain**

Energy value drained each tick that the player is moving.

The player will not be able to move when his energy falls below `minRunEnergy`.

**See also:** `minRunEnergy`

---

**Note:** Setting this to zero will disable any energy drain.

---

**float PlayerData::runForce**

Force used to accelerate the player when running.

**float PlayerData::runSurfaceAngle**

Maximum angle from vertical (in degrees) the player can run up.

**filename PlayerData::shapeNameFP[4]**

File name of this player's shape that will be used in conjunction with the corresponding mounted image.

These optional parameters correspond to each mounted image slot to indicate a shape that is rendered in addition to the mounted image shape. Typically these are a player's arms (or arm) that is animated along with the mounted image's state animation sequences.

**SplashData PlayerData::Splash**

SplashData datablock used to create splashes when the player moves through water.

**float PlayerData::splashAngle**

Maximum angle (in degrees) from pure vertical movement in water to generate splashes.

**ParticleEmitterData PlayerData::splashEmitter[3]**

Particle emitters used to generate splash particles.

**float PlayerData::splashFreqMod**

Multiplied by speed to determine the number of splash particles to generate.

**float PlayerData::splashVelEpsilon**

Minimum speed to generate splash particles.

**float PlayerData::splashVelocity**

Minimum velocity when moving through water to generate splashes.

**bool PlayerData::sprintCanJump**

Can the player jump while sprinting.

**float PlayerData::sprintEnergyDrain**

Energy value drained each tick that the player is sprinting.

The player will not be able to move when his energy falls below sprintEnergyDrain.

**See also:** minSprintEnergy

---

**Note:** Setting this to zero will disable any energy drain.

---

**float PlayerData::sprintForce**

Force used to accelerate the player when sprinting.

**float PlayerData::sprintPitchScale**

Amount to scale pitch motion while sprinting.

**float PlayerData::sprintStrafeScale**

Amount to scale strafing motion vector while sprinting.

**float PlayerData::sprintYawScale**

Amount to scale yaw motion while sprinting.

**Point3F PlayerData::swimBoundingBox**

Collision bounding box used when the player is swimming.

**See also:** boundingBox

**float PlayerData::swimForce**

Force used to accelerate the player when swimming.

**bool PlayerData::transitionToLand**

When going from a fall to a land, should we transition between the two.

**See also:** landSequenceTime

---

**Note:** Only takes affect when landSequenceTime is greater than 0.

---

**float PlayerData::upMaxSpeed**

Maximum upwards speed.

See also:

- upResistSpeed
- upResistFactor

---

**Note:** This limit is only enforced if the player's upward speed exceeds upResistSpeed.

---

**float PlayerData::upResistFactor**

Factor of resistance once upResistSpeed has been reached.

See also:

- upMaxSpeed
- upResistSpeed

**float PlayerData::upResistSpeed**

Upwards speed at which resistance will take place.

See also:

- upMaxSpeed
- upResistFactor

**SFXTrack PlayerData::waterBreathSound**

Sound to play when in water and coverage equals 1.0 (fully underwater).

Note that unlike FootUnderwaterSound, this sound plays even if the player is not moving around in the water.

## 24.3 Shapebase Class - TODO

### 24.3.1 Detailed Description

### 24.3.2 Control Object

### 24.3.3 Energy/Damage

### 24.3.4 Member Function Documentation

### 24.3.5 Member Data Documentation

## 24.4 Turrets - TODO

### 24.4.1 Detailed Description

### 24.4.2 Overview

#### Scanning

Gained Target

Lost Target

Destroyed

### **24.4.3 Deployable Turret**

### **24.4.4 Example State Machine**

### **24.4.5 Shape File Nodes**

### **24.4.6 Ignore List**

### **24.4.7 Member Function Documentation**

### **24.4.8 AITurretShapeData and Member Data Documentation**

## **24.5 Triggers - TODO**

### **24.5.1 Introduction**

### **24.5.2 Creating Triggers**

### **24.5.3 Using Triggers to Cause Events**

### **24.5.4 Using Trigger Callbacks to Cause Events**

### **24.5.5 Conclusion**

## **24.6 Weapons - TODO**

### **24.6.1 Detailed Description**

### **24.6.2 Weapon Shape Nodes**

### **24.6.3 Weapon Muzzle Flash**

### **24.6.4 First Person Shape (optional)**

### **24.6.5 Animation Sequence Transitions**

### **24.6.6 Animation Sequence Selection**

### **24.6.7 eyeMount Node (optional)**

### **24.6.8 Special State Triggers**

### **24.6.9 Special States**

### **24.6.10 Member Function Documentation**

### **24.6.11 Member Data Documentation**

---

496

## **24.7 Proximity Mines**

### **24.7.1 Detailed Description**

Proximity mines can be deployed using the world editor or thrown by an in-game object. Once armed, any Player or Vehicle object that moves within the mine's trigger area will cause it to explode.

Internally, the ProximityMine object transitions through the following states:

1. **Thrown:** Mine has been thrown, but has not yet attached to a surface
2. **Deployed:** Mine has attached to a surface but is not yet armed. Start playing the armingSound and *armed* sequence.
3. **Armed:** Mine is armed and will trigger if a Vehicle or Player object moves within the trigger area.
4. **Triggered:** Mine has been triggered and will explode soon. Invoke the onTriggered callback, and start playing the triggerSound and triggered sequence.
5. **Exploded:** Mine has exploded and will be deleted on the server shortly. Invoke the onExplode callback on the server and generate the explosion effects on the client.

---

**Note:** Proximity mines with the static field set to true will start in the **Armed** state. Use this for mines placed with the World Editor.

---

The shape used for the mine may optionally define the following sequences:

**armed** Sequence to play when the mine is deployed, but before it becomes active and triggerable (armingDelay should be set appropriately).

**triggered** Sequence to play when the mine is triggered, just before it explodes (triggerDelay should be set appropriately).

**Example:**

```
datablock ProximityMineData( SimpleMine )
{
    // ShapeBaseData fields
    category = "Weapon";
    shapeFile = "art/shapes/weapons/misc/proximityMine.dts";

    // ItemData fields
    sticky = true;

    // ProximityMineData fields
    armingDelay = 0.5;
    armingSound = MineArmedSound;

    autoTriggerDelay = 0;
    triggerOnOwner = true;
    triggerRadius = 5.0;
    triggerSpeed = 1.0;
    triggerDelay = 0.5;
    triggerSound = MineTriggeredSound;
    explosion = RocketLauncherExplosion;

    // dynamic fields
    pickUpName = "Proximity Mines";
    maxInventory = 20;

    damageType = "MineDamage"; // type of damage applied to objects in radius
    radiusDamage = 30;          // amount of damage to apply to objects in radius
    damageRadius = 8;           // search radius to damage objects when exploding
}
```

(continues on next page)

(continued from previous page)

```
    areaImpulse = 2000;           // magnitude of impulse to apply to objects in radius
};

function ProximityMineData::onTriggered( %this, %obj, %target )
{
    echo( %this.name SPC "triggered by " @ %target.getClassName() );
}

function ProximityMineData::onExplode( %this, %obj, %position )
{
    // Damage objects within the mine's damage radius
    if ( %this.damageRadius > 0 )
        radiusDamage( %obj.sourceObject, %position, %this.damageRadius,
            %this.radiusDamage, %this.damageType, %this.areaImpulse );
}

function ProximityMineData::damage( %this, %obj, %position, %source, %amount,
    ↪%damageType )
{
    // Explode if any damage is applied to the mine
    %obj.schedule(50 + getRandom(50), explode);
}

%obj = new ProximityMine()
{
    dataBlock = SimpleMine;
};
```

**See also:** ProximityMineData

## 24.7.2 Member Function Documentation

`void ProximityMine::explode()`  
Manually cause the mine to explode.

# 24.8 Camara Modes - TODO

## 24.8.1 Introduction

## 24.8.2 Camera Modes

## 24.8.3 Toggling Basic Camera Modes

## 24.8.4 Toggling Special Camera Modes

`setOrbitObject(GameBase, Point3F, float, float, [float], [bool], [Point3F], [bool])`

`setOrbitPoint(string, float, float, [float], [Point3F], [bool])`

`setTrackObject(GameBase, [Point3F])`



setFlyMode(void)

setNewtonFlyMode(void)

## 24.8.5 Camera Options

## 24.8.6 Conclusion

# 24.9 RTS Prototype - TODO

## 24.9.1 Introduction

## 24.9.2 Create A New Project

## 24.9.3 Camera Setup

## 24.9.4 Mouse Setup

Mouse Cursor Toggling

Placing Structures Using The GUI

## 24.9.5 Mouse-Driven Input

Player Spawning

Movement

Spawning Enemy Targets

Attacking

Tweaking Attacks

## 24.9.6 Destination Markers

Creating a Material

Creating a Decal

Spawning the Marker

Erasing the Marker

## 24.9.7 Camera Modes

Orbit Camera

Overhead Camera

## **24.9.8 Going More Real-Time Strategy**

## **24.9.9 Conclusion**

# **24.10 Adventure Prototype - TODO**

## **24.10.1 Introduction**

## **24.10.2 Building A Level**

## **24.10.3 Hooking Up The Triggers**

## **24.10.4 Conclusion**

# **24.11 Mission Triggers - TODO**

## **24.11.1 Introduction**

## **24.11.2 Setting Up**

## **24.11.3 Create the New Trigger Datablock**

## **24.11.4 The onEnterTrigger() Callback and Other Support Scripts**

## **24.11.5 Putting It All Together**

## **24.11.6 Conclusion**

# **24.12 TSShapeConstructor - TODO**

## **24.12.1 Introduction**

## **24.12.2 Terminology**

## **24.12.3 Example 1: Adding a Collision Mesh To an Existing Shape**

## **24.12.4 Example 2: Adding a Mesh From an Existing DTS File**

## **24.12.5 Example 3: Auto-loading animations**

## **24.12.6 Example 4: Splitting COLLADA animations**

## **24.12.7 Example 5: Rigid-body Player Character**

## **24.12.8 TSShapeConstructor Commands**

## Miscellaneous

`dumpShape([string])`

`saveShape(string)`

## Nodes

`getNodeCount()`

`getNodeIndex(string)`

`getNodeName(S32)`

`getNodeParentName(S32)`

`getNodeChildCount(string)`

`getNodeChildName(string, S32)`

`getNodeObjectCount(string)`

`getNodeObjectName(string, S32)`

`getNodeTransform(string)`

`setNodeTransform(string, string, [string])`

`renameNode(string, string)`

`addNode(string, string, [string])`

`removeNode(string)`

## Objects

`getObjectCount()`

`getObjectName(S32)`

`getObjectNode(string)`

`setObjectNode(string, string)`

`renameObject(string, string)`

`removeObject(string)`

## Meshes

`getMeshCount(string)`

`getMeshname(string, S32)`

`setMeshSize(string, S32)`

`getMeshType(string)`

`setMeshType(string, string)`

`getMeshMaterial(string)`

`setMeshMaterial(string, string)`

`addMesh(string, string, string)`

`removeMesh(string)`

## AutoBillboards

`addAutoBillboard(S32, S32, S32, S32, S32, bool, F32)`

`removeAutoBillboard(S32)`

## Sequences

`getSequenceCount()`

`getSequenceIndex(S32)`

`getSequenceName(S32)`

`getSequenceSource(S32)`

`getSequenceFrameCount(string)`

`getSequencePriority(string)`

`setSequencePriority(string, F32)`

`getSequenceCyclic(string)`

`setSequenceCyclic(string, bool)`

`getSequenceBlend(string)`

`setSequenceBlend(string, bool, string, S32)`

`getSequenceGroundSpeed(string)`

`setSequenceGroundSpeed(string, string, [string])`

`renameSequence(string, string)`

`addSequence(string, string, [S32], [S32])`

`removeSequence(string)`

`getTriggerCount(string)`

`getTrigger(string)`

`addTrigger(string, S32, S32)`

`removeTrigger(string, S32, S32)`

## 24.12.9 Conclusion

# 24.13 Engine To Script

## 24.13.1 Introduction

This document intends to give an overview and a set of rules for creating and documenting Torque control layer interfaces. There are specific functions used for communication between TorqueScript and the C++ engine. This system makes heavy use of macros and ties in closely with the Console system.

If you plan on extending the engine to support new classes or functionality, you will need to adhere to this guide so as to not break your build. Equally important is the practice of documenting your extensions, so make heavy use of the fields that exist for documentation.

---

**Note:** This document was primarily written for C++ programmers with access to Torque 3D's source code. Without source code you will not be able to directly implement any of the following. However, someone without source access should still be able to harvest important documentation on Torque 3D's control layer interface.

---

## 24.13.2 Creating Call-in Points

A call-in point is an entry point from the control layer into Torque. In simpler terms, these are functions called from TorqueScript. There are two types of call-in functions: global and class

## Global functions

These are stand alone functions that can be called in TorqueScript without belonging to any class or object.

Example:

```
echo("Hello World");
```

You define a global function by using the following procedure:

Include engineAPI.h in your cpp implementation file:

```
#include "console/engineAPI.h"
```

Define the function:

```
DefineEngineFunction( myFunction, myReturnType, ( S32 arg1, F32 arg2, const char*  
↪arg3 ),, "Documentation string" )  
{  
    // Code goes here  
}
```

## Working Example in Torque 3D:

```
DefineEngineFunction(addBadWord, bool, (const char* badWord),,  
    "Add a string to the bad word filter\n"  
    "The bad word filter is a table containing words  
↪which will not be "  
    "displayed in chat windows. Instead, a  
↪designated replacement string will be displayed.\n"  
    "@param badWord Exact text of the word to  
↪restrict.\n"  
    "@return True if word was successfully added,  
↪false if the word or a subset of it already exists in the table\n"  
    "@see filterString\n\n"  
    "@tsexample\n"  
        "// In this game, \"Foobar\" is banned\n"  
        "%badWord = \"Foobar\";\n\n"  
        "// Returns true, word was successfully  
↪added\n"  
        "addBadWord(%badWord);\n\n"  
        "// Returns false, word has already been  
↪added\n"  
        "addBadWord(\"Foobar\");"  
    "@endtsexample\n"  
    "@ingroup Game")  
{  
    TORQUE_UNUSED(badWord);  
    return gBadWordFilter->addBadWord(badWord);  
}
```

## Class methods

These are functions called in TorqueScript from an object associated with a class, such as SFXSource.

Example:

```
%sfxSourceObject.stop()
```

You define a class method by using the following procedure:

Include engineAPI.h in your cpp implementation file:

```
#include "console/engineAPI.h"
```

Define the method:

```
DefineEngineMethod( MyClass, myMethod, myReturnType, ( S32 arg1, F32 arg2, const_
↳char* arg3 ),, "Documentation string" )
{
    // Code goes here
}
```

**Working Example in Torque 3D:**

```
DefineEngineMethod( GuiCanvas, reset, void, (),,
    "@brief Reset the update regions for the canvas.\n\n"
    "@tsexample\n"
    "Canvas.reset();\n"
    "@endtsexample\n\n")
{
    object->resetUpdateRegions();
}
```

Within the function bodies, you can access the given parameters and return a value just like with a normal C++ function. To assign default values to arguments (like in C++: S32 index=-1), you use the macro argument left empty above:

```
DefineEngineMethod( MyClass, myMethod, myReturnType, ( S32 arg1, F32 arg2, const_
↳char* arg3 ), ( -1.f, "foo" ), "Documentation string" )
{
    // Code goes here
}
```

Here, -1.f is the default argument for “arg2” and “foo” is the default argument for “arg3.” **Be aware that the default argument list is matched starting from the end of the argument list (like it happens in C++).**

**Working Example in Torque 3D:**

```
DefineEngineMethod( SFXSource, play, void, ( F32 fadeInTime ), ( -1.f ),
    "Start playback of the source.\n"
    "If the sound data for the source has not yet been fully loaded, there will be a_
↳delay after calling "
    "play and playback will start after the data has become available.\n\n"
    "@param fadeInTime Seconds for the sound to reach full volume. If -1, the_
↳SFXDescription::fadeInTime "
    "set in the source's associated description is used. Pass 0 to disable a fade-
↳in effect that may "
    "be configured on the description." )
{
    object->play( fadeInTime );
}
```



## Notes

- You cannot currently use references (&) as argument types. This is due to the default argument code here and might change in the future.
- All types used in an engine API function/method must have registered console types including an ImplementConsoleTypeCasters instance for the given native C++ type. If this is missing, you will see link-time errors or compile-time errors in the templates.
- **There are two exceptions:**
  - U32 can be used and is treated internally like S32
  - Pointers to SimObject-derived classes can be used freely
- Do not use String for string arguments. Use “const char\*”. You may use Strings as return values, though (for TS: memory will be copied to evaluator stack).
- If you return “const char\*”, the API assumes the lifetime of the string exceeds that of the call-in. If that is not the case, you need to use Con::getReturnBuffer.
- If you return String, the API assumes the String is temporary and will automatically copy it with Con::getReturnBuffer.
- Due to the template trickery involved in the engineAPI macro system, default argument values will be constructed once during global startup (except if the compiler is smart enough to optimize the non-side-effecting constructors away). **This means that any default argument value must not use a feature of Torque that requires global ctors to have executed. Using String::EmptyString is an example of what would not work.**

### 24.13.3 Creating Call-out Points

A call-out point is an exit point from the C++ engine to the control layer (TorqueScript). This is often referred to as a callback. The main purpose of a callback is to trigger a function in script from C++ after an important piece of code has been executed. The following procedure is used to creating a callback:

In your class definition, declare the callbacks for the class with DECLARE\_CALLBACK:

Example:

```
protected:
    /// @name Callbacks
    /// @{

    DECLARE_CALLBACK(void, onAdd, (SimObjectId ID) );

    /// @}
```

In your implementation file, include the engineAPI.h header:

```
#include "console/engineAPI.h"
```

Then, use the IMPLEMENT\_CALLBACK macro for each of the callbacks:

```
IMPLEMENT_CALLBACK( ScriptObject, onAdd, void, ( SimObjectId ID ), ( ID ),
    "Called when this ScriptObject is added to the system.\n"
    "@param ID Unique object ID assigned when created (%this in script).\n"
);
```

The two list arguments to the macro represent the raw argument type list ( type arg1, type arg2... ) as well as the argument call list ( arg1, arg2 ). This is needed by the macros to chain the call along. To trigger a callback in the code, invoke the given callback method by appending `_callback` to its name:

```
bool ScriptObject::onAdd()
{
    if (!Parent::onAdd())
        return false;

    // Call onAdd in script!
    onAdd_callback(getId());
    return true;
}
```

### 24.13.4 Creating Types

Before a native C++ type can be used in the engine API, it must be registered with the console system. How this is done depends on the kind of type. For all registered types, `TYPEID<type>()` returns the numeric type ID of the static type.

#### Creating an Object Type

To define a new object type for use in the control layer API, use the following procedure. For a class `T`, this allows to use pointers to `T` objects to be used in the API. Derive your class directly or indirectly from `SimObject`:

```
class SFXAmbience : public SimDataBlock
{
```

In the class interface, define a public typedef “Parent” as an alias for the parent class:

```
public:
    typedef SimDataBlock Parent;
```

Also, in the public interface, use `DECLARE_CONOBJECT`. Optionally, also supply a description and category:

```
DECLARE_CONOBJECT( SFXAmbience );
DECLARE_CATEGORY( "SFX" );
DECLARE_DESCRIPTION( "An ambient sound environment." );
```

Then, in the implementation file, use `IMPLEMENT_CONOBJECT` or one of its variants (for datablocks and netobjects):

```
IMPLEMENT_CO_DATABLOCK_V1( SFXAmbience );
```

#### Creating an Enumeration Type

To define a new enumeration type for use in the control layer API, use the following procedure. In the file where your native enum type is defined, include the type header:

```
#include "console/dynamicTypes.h"
```

After the enum definition:

```

/// Rolloff curve used for distance volume attenuation of 3D sounds.
enum SFXDistanceModel
{
    SFXDistanceModelLinear,          ///< Volume decreases linearly from min to max,
    ↪where it reaches zero.
    SFXDistanceModelLogarithmic,      ///< Volume halves every min distance steps,
    ↪starting from min distance; attenuation stops at max distance.
};

```

Declare the public type bits:

```
DefineEnumType( SFXDistanceModel );
```

In the corresponding implementation file, add the corresponding implementation detail:

```

ImplementEnumType( SFXDistanceModel,
    "Type of volume distance attenuation curve.\n"
    "The distance model determines the falloff curve applied to the volume of 3D,
    ↪sounds over distance.\n\n"
    "@ref SFXSource_volume\n\n"
    "@ingroup SFX" )
{ SFXDistanceModelLinear, "Linear",
    "Volume attenuates linearly from the references distance onwards to max,
    ↪distance where it reaches zero." },
{ SFXDistanceModelLogarithmic, "Logarithmic",
    "Volume attenuates logarithmically starting from the reference distance and,
    ↪halving every reference distance step from there on. "
    "Attenuation stops at max distance but volume won't reach zero." },
EndImplementEnumType;

```

Note you can declare fields of this type directly:

```

addField( "soundDistanceModel", TYPEID< SFXDistanceModel >(), Offset(
    ↪mSoundDistanceModel, LevelInfo ), "The distance attenuation model to use." );

```

Be aware that the native C++ enum type used in the macros must be global (the resulting type name will be global). To use nested enum types, simply work around this with a typedef:

```

typedef SFXPlayList::ELoopMode SFXPlayListLoopMode;
DefineEnumType( SFXPlayListLoopMode );

```

## Creating a Bitfield Type

To define a new bitfield type for use in the control layer API, follow the instructions for defining an enumeration type with the following macros substituted for their enumeration counterparts:

Declare the type:

```
DefineBitFieldType( MaterialAnimType );
```

Implement the type:

```

ImplementBitFieldType( MaterialAnimType,
    "The type of animation effect to apply to this material.\n"
    "@ingroup GFX\n\n"
    { Material::Scroll, "Scroll", "Scroll the material along the X/Y axis.\n" },

```

(continues on next page)

(continued from previous page)

```

    { Material::Rotate, "Rotate" , "Rotate the material around a point.\n"},
    { Material::Wave, "Wave" , "Warps the material with an animation using Sin,
↪Triangle or Square mathematics.\n"},
    { Material::Scale, "Scale", "Scales the material larger and smaller with a pulsing
↪effect.\n" },
    { Material::Sequence, "Sequence", "Enables the material to have multiple frames of
↪animation in its imagemap.\n" }

```

Immediately after you will end the implementation:

```
EndImplementBitfieldType;
```

## 24.13.5 Documentation

Torque 3D makes it very easy to document the code via strings written directly into the implementations. Documentation strings for engine API items use the JavaDoc tag syntax (@param, @ingroup, etc...).

### Documenting an engine function/method

Place the documentation in the DefineEngineXXX macro. If not containing a @brief, the first line (i.e. up to n) will be taken as the function @brief and automatically split out from the doc string. This allows the coder to omit the @brief in almost all cases as the first line will generally be good enough. Write explicit @briefs where this is not the case. For functions, place them in a group by using @ingroup tag:

#### Example:

```

DefineEngineFunction( strIsMatchExpr, bool, ( const char* pattern, const char* str,
↪bool caseSensitive ), ( false ),
    "Match a pattern against a string.\n"
    "@param pattern The wildcard pattern to match against. The pattern can include
↪characters, '*' to match "
    "any number of characters and '?' to match a single character.\n"
    "@param str The string which should be matched against @a pattern.\n"
    "@param caseSensitive If true, characters in the pattern are matched in case-
↪sensitive fashion against "
    "this string. If false, differences in casing are ignored.\n"
    "@return True if @a str matches the given @a pattern.\n\n"
    "@tsexample\n"
    "strIsMatchExpr( \"f?o*R\", \"foobar\" ) // Returns true.\n"
    "@endtsexample\n"
    "@see strIsMatchMultipleExpr\n"
    "@ingroup Strings" )
{
    return FindMatch::isMatch( pattern, str, caseSensitive );
}

```

### Handling overloaded functions

Overloading is not currently supported by engineAPI. TorqueScript functions that have varying argument lists thus cannot be implemented with engineAPI right now.

Example:

```
GuiControl::setExtent( Point2I p )
GuiControl::setExtent( S32 width, S32 height )
```

These functions must remain implemented with the old ConsoleXXX macros. For documentation, these functions must be split into multiple independent functions/methods for Doxygen. This is achieved using the following procedure:

Document the original function for in-game purposes but @hide it from Doxygen:

```
ConsoleMethod( GuiControl, setExtent, void, 3, 4,
    "( Point2I p | int x, int y ) Set the width and height of the control.\n\n"
    "@hide" )
```

Write an independent ConsoleDocFragment for each variant of the function method:

```
static ConsoleDocFragment _sGuiControlSetExtent1(
    "@brief Resize the control to the given dimensions.\n\n"
    "Child controls will resize according to their layout settings.\n"
    "@param width The new width of the control in pixels.\n"
    "@param height The new height of the control in pixels.",
    "GuiControl", // The class to place the method in; use NULL for functions.
    "void setExtent( S32 width, S32 height );" ); // The definition string.

static ConsoleDocFragment _sGuiControlSetExtent2(
    "@brief Resize the control to the given dimensions.\n\n"
    "Child controls with resize according to their layout settings.\n"
    "@param p The new ( width, height ) extents of the control.",
    "GuiControl", // The class to place the method in; use NULL for functions.
    "void setExtent( Point2I p );" ); // The definition string.
```

## Handling variadic functions

Variadic functions are not currently supported by engineAPI. TorqueScript functions that takes a variable number of arguments cannot be implemented with engineAPI right now.

**Example:**

```
echo( string text... )
```

For now, these functions must remain implemented with the old ConsoleXXX macros. To document them, place the function argument prototype string first in the usage string and then include documentation for other types of functions.

**Class Function:**

```
ConsoleMethod( SimSet, add, void, 3, 0,
    "( SimObject objects... ) Add the given objects to the set.\n"
    "@param objects The objects to add to the set." )
```

**Global Function:**

```
ConsoleFunction( getRandom, F32, 1, 3,
    "( int a=1, int b=0 ) "
    "Get a random number between @a a and @a b.\n"
    "@param a Lower bound on the random number. The random number will be >= @a a.\n"
    "@param b Upper bound on the random number. The random number will be <= @a b.\n"
    "@return A pseudo-random number between @a a and @a b.\n" )
```

## Documenting an engine callback

Place the documentation on the IMPLEMENT\_CALLBACK macro. If not containing a @brief, the first line (i.e. up to n) will be taken as the function @brief and automatically split out from the doc string. This allows to omit the @brief in almost all cases as the first line will generally be good enough. Write explicit @briefs where this is not the case:

```
IMPLEMENT_CALLBACK( GuiControl, onActive, void, ( bool state ), ( state ),
    "Called when the control changes its activeness state, i.e. when going from active_
↳to inactive or vice versa.\n"
    "@param stat The new activeness state.\n"
    "@see isActive\n"
    "@see setActive\n"
    "@ref GuiControl_VisibleActive" );
```

## Documenting an engine class

Place the documentation either in a .txt file in Documentation/scriptDocs/docs or in the C++ implementation files using the ConsoleDocClass macro. Always use @ingroup and @brief:

```
ConsoleDocClass( SFXAmbience,
    "@brief A datablock that describes an ambient sound space.\n\n"

    "Each ambience datablocks captures the properties of a unique ambient sound space._
↳ A sound space is comprised of:\n"

    "- an ambient audio track that is played when the listener is inside the space,\n"
    "- a reverb environment that is active inside the space, and\n"
    "- a number of SFXStates that are activated when entering the space and_
↳deactivated when exiting it.\n"
    "\n"

    "Each of these properties is optional.\n\n"

    "An important characteristic of ambient audio spaces is that their unique nature_
↳is not determined by their location "
    "in space but rather by their SFXAmbience datablock. This means that the same_
↳SFXAmbience datablock assigned to "
    "multiple locations in a level represents the same unique audio space to the sound_
↳system.\n\n"

    "This is an important distinction for the ambient sound mixer which will activate_
↳a given ambient audio space only "
    "once at any one time regardless of how many intersecting audio spaces with the_
↳same SFXAmbience datablock assigned "
    "the listener may currently be in.\n\n"

    "Each SFXAmbience instance will automatically add itself to the global_
↳SFXAmbienceSet.\n\n"

    "At the moment, transitions between reverb environments are not blended and_
↳different reverb environments from multiple "
    "active SFXAmbiences will not be blended together. This will be added in a future_
↳version.\n\n"

    "@tsexample\n"
```

(continues on next page)

(continued from previous page)

```

"singleton SFXAmbience( Underwater ) \n"
"{\n"
"    environment = AudioEnvUnderwater; \n"
"    soundTrack = ScubaSoundList; \n"
"    states[ 0 ] = AudioLocationUnderwater; \n"
"}; \n"
"@endtsexample \n"

"@see SFXEnvironment \n"
"@see SFXTrack \n"
"@see SFXState \n"
"@see LevelInfo::soundAmbience \n"
"@see Zone::soundAmbience \n"
"@ref Datablock_Networking \n"
"@ingroup SFX \n"
"@ingroup Datablocks \n"
);

```

## Documenting a field

Place the documentation directly on the `addField` usage string. The first line (i.e. up to the first `\n`) is taken as the `@brief` and showed in inspectors. The rest is only used for doc output:

```

addField( "coneOutsideVolume", TypeF32, Offset( mConeOutsideVolume,
↳SFXDescription ),
    "Determines the volume scale factor applied the a source's base volume level_
↳outside of the outer cone. \n"
    "In the outer cone, starting from outside the inner cone, the scale factor_
↳smoothly interpolates from 1.0 (within the inner cone) "
    "to this value. At the moment, the allowed range is 0.0 (silence) to 1.0_
↳(no attenuation) as amplification is only supported on "
    "XAudio2 but not on the other devices. \n\n"
    "Only for 3D sound. \n"
    "@ref SFXSource_cones" );

```

## Documenting a variable

Place the documentation on the `Con::addVariable()` call. Use `@ingroup` to properly associate the documentation with a group. The first line of the doc string will be taken as the `@brief`:

```

Con::addVariable( "SFX::numSources", TypeS32, &mStatNumSources, NULL,
    "Number of SFXSources that are currently instantiated. \n"
    "@ingroup SFX" );

```

Make sure the `Con::addVariable()` call is within the engine init phase and will be available after the engine has started up. Don't put the call on conditional paths. If necessary, use the module system (`core/module.h`) for relevant initialization

## Documenting a constant

Place the documentation on the `Con::addConstant()` call. Use `@ingroup` to properly associate the documentation with a group. The first line of the doc string will be taken as the `@brief`:



```
Con::addConstant( "SFX::REVERB_FLAG_DECAYTIMESCALE", TypeS32, &
    ↪sReverbFlagDecayTimeScale, NULL,
    "SFXEnvironment::envSize affects reverberation decay time.\n"
    "@see SFXEnvironment::flags\n\n"
    "@ingroup SFX" );
```

Make sure the `Con::addConstant()` call is within the engine init phase and will be available after the engine has started up. Don't put the call on conditional paths. If necessary, use the module system (`core/module.h`) for relevant initialization.

## Inserting an arbitrary piece of documentation

Arbitrary code documentation can be inserted using the `ConsoleDoc` macro:

```
ConsoleDoc(
    "@defgroup MyGroup\n"
    "@brief Blabla\n\n"
    "My description.\n\n"
    "@ingroup Parent"
);
```

Be aware that only one `ConsoleDoc` instance can be in any given `.cpp` file. To work around this, either place the macro instances in different namespaces or use the `ConsoleDocFragment` class directly:

```
static ConsoleDocFragment _sGuiControlSetExtent1(
    "@fn void GuiControl::setExtent( int width, int height )\n"
    "@brief Resize the control to the given dimensions.\n\n"
    "Child controls will resize according to their layout settings.\n"
    "@param width The new width of the control in pixels.\n"
    "@param height The new height of the control in pixels." );

static ConsoleDocFragment _sGuiControlSetExtent2(
    "@fn void GuiControl::setExtent( Point2I p )\n"
    "@brief Resize the control to the given dimensions.\n\n"
    "Child controls with resize according to their layout settings.\n"
    "@param p The new ( width, height ) extents of the control." );
```

You can also place a fragment inside a class:

```
static ConsoleDocFragment _sFragment(
    "doc",
    "className", //NULL to place globally
    "definition" ); //optional
```

## 24.13.6 Important Notes

Things to watch out for:

- Put two newlines after `@brief`, i.e. “`@brief Textnn`”
- Put two newlines at the end of a paragraph, i.e. “`My paragraph.nn`”
- Put two newlines before `@tsexample`.
- Put two newlines to separate a `@see` from a `@ref`. Otherwise Doxygen will put the `@ref` on the same line as the `@see` which is visually unpleasing.

- Put @ingroup at the end of doc strings.
- Put datablocks in two groups; first in the group that the non-datablock object is in (e.g. PlayerData should be in the same group with Player) and second in the “Datablocks” group
- Use @tsexample and @endtsexample instead of @code and @endcode

## 24.13.7 Conclusion

After reading through this document you should have a solid understanding of how the C++ engine communicates with the script layer. This is how the Torque team extends the engine and generates its official documentation. You should adhere to these standards if you wish to maintain stability while extending the engine.

## 24.14 Projectiles

### 24.14.1 Introduction

This article will discuss the technical aspect of creating custom projectiles for use with custom weapons in your games.

### 24.14.2 Projectiles

Projectiles are what we use in T3D for most bullets, rockets, grenades, and all objects of the like. Typically a projectile is created in the `WeaponImage::OnFire` call like so:

```
// Create the projectile object
%p = new (%this.projectileType)
{
    dataBlock = %this.projectile;
    initialVelocity = %muzzleVelocity;
    initialPosition = %obj.getMuzzlePoint(%slot);
    sourceObject = %obj;
    sourceSlot = %slot;
    client = %obj.client;
};
MissionCleanup.add(%p); // MissionCleanup is a SimGroup.
```

Once created, a projectile will move from its `initialPosition` in the direction of its initial velocity. It will travel along this path until it comes in contact with a `sceneObject`. (This includes the terrain.) where it will either bounce, or explode depending on how the projectile's datablock has been setup. Before the projectile actually explodes, it uses its `onExplode` callback in script before actually the `explode()` code in C++. This gives scriptors a chance to affect the behavior of the projectile before the engine actually processes an explosion, or to do any last minute tricks, hacks, or checks before it explodes. It's important to keep in mind that the `onExplode` callback will only be done server side.

While there are many fields and properties to a Projectile's datablock, there are a couple to keep in mind, and have a firm understanding of. (Those closely related will be grouped together.)

Important Projectile Datablock Fields	
projectileShapeName	Path to the shape to be used for the projectile
explosion	Explosion effect to play at the point of the explosion
decal	Decal to leave on walls/terrain/objects when it explodes
particleEmitter	An effect for the trail as the projectile lies through the air
armingDelay	amount of time (In milliseconds) before a projectile is allowed to explode
lifeTime	amount of time (in milliseconds) before a projectile deletes itself from the scene

**Note:** One thing I've noticed is that somewhere down the line in projectiles, we've relied heavily on the use of dynamic fields to handle things in script that would be assumed to be taken care of by the engine. For example, `directDamage` is in our example datablock, but really is only a dynamic field used later for players hit directly with a rocket.

Most of the processing of a projectile is done from the script side of things. The engine side of things really only updates it's position, calls it's callbacks, networking, exploding when the time is right, etc. Things like damage and pushback must be handled from various callbacks in script.

From a script side of things, when a projectile has been created, the next thing you'll have to do is deal with it's script callback. Depending on the behavior of the projectile, you will be looking to work with `onCollision` (For ballistic-type weapons), or `onExplode` (For grenade-type weapons).

At the very least, we want to call some kind of `damage()` function that will decrement the player's health and update their GUI. In the event of a grenade or rocket, we want to damage the player it hit and likely anything around it. The easiest way to do this is by using a `ContainerRadiusSearch`, passing in the position of the explosion, the radius of the explosion (usually set as a dynamic field on the projectile's datablock), and `$TypeMasks::ShapeBaseObjectType` or `$TypeMasks::PlayerObjectType`. We can do some math on every object the container search returns to determine if we should just outright damage the player, or scale the damage and impulse effects based on their distance from the center of the explosion. (It doesn't make much sense to take full damage when you are on the very edge of an explosion, does it?). You should also make use of `calcExplosionCoverage` during the container call to make sure that objects aren't damaged if they are covered by walls, or other objects. Distance scale can be calculated as:

```
scale = (distanceFromExplosion < explosionRadius / 2)? 1.0 : 1.0
        - ((distanceFromExplosion - explosionRadius / 2) / explosionRadius / 2);
```

### 24.14.3 Conclusion

Having Torque 3D's camera system exposed to script gives you a great deal of power and flexibility in your game. When you have tested the various modes, you can begin to see how this will affect game play

## 24.15 Networking

### 24.15.1 Introduction

Torque 3D was designed around networked games. This tutorial will give an overview of the high level Client-Server Networking concepts utilized by Torque and will show you how to make a network teleport command sequence you can use as a springboard for your own networked game play ideas

What is covered in this tutorial:

- The Client/Server Concept
- Datablocks in Networking
- Network Connection Classes ? Linking Client and Server
- Sending Commands in the Torque Client/Server Model
- Making Your Own Commands

### 24.15.2 The Client/Server Concept

This is a high level overview of the **Client/Server Networking** Model that Torque 3D uses for its networking model. A networked Torque game normally uses only a single **Server** instance, but multiple **Client** instances can connect to the Server at once. When a client joins a server and the mission begins downloading, there is a 3 phase process:

1. Datablocks are sent to the client, such as vehicle and weapon data. Datablocks will be covered in more detail in the **Datablocks in Networking** section of this tutorial.
2. In-Mission Objects are **ghosted** to the client, such as other players. Ghosting will be discussed in The Server section of this tutorial.
3. Finally, the scene is lit on the client, and then game play can begin.

There are 3 types of networking setups an instance of Torque can be in:

- **Dedicated Server** ? The server has no local client; it only connects to external clients. Games like MMOs generally use dedicated servers.
- **Hosted Server** ? The server has a local client that connects directly to the server instance instead of over the network. This is used in both a Single-Player instance of Torque, and when the instance hosting the game also has a player. Many FPS games and LAN party games use this setup, where one player's game is the host and other clients connect to the host.
- **Client Only** ? The game runs in client mode only and joins a game play session by connecting to a server either on a local network or over the internet.

### The Server

The **Server** has multiple responsibilities, and is the central “care-taker” of a networked game. Even when running in a single-player instance, or when the server is not a dedicated server, Torque uses the Client-Server Networking model. In this case, the server connects to a “client” in the same game instance, and sends “short-circuited” commands to the local client.

The Server has the following responsibilities, and may have more, depending on how a specific game is structured:

- **Authoritative** ? Information transmitted from the Server overrides the local Client information in the case of a bad prediction on the part of the Client. This ensures that the local data of the Client does not “drift” from the data the other Clients have.
- **Ghosting** ? The server keeps track of all the “true” objects in play, and “Ghosts” or copies data for each of them to the Clients by using **Scoping**. What this means the server has a master list of objects, and sends updated information for them to a local copy of the object on the client.
- **Scoping** - Scoping helps sort out what each client is aware of so Torque does not have to send updates for objects that don't need to be updated. In order to save bandwidth and minimize network delay, objects are initially ghosted to the client in the second phase of mission startup.
- **Datablocks** ? Datablocks are sent in the first phase of mission start-up. The Server has all of the datablocks for clients to download when they begin a mission. This means that the server only has to send datablocks as

they are needed, and can send new datablocks for different missions. Datablocks are discussed further in the **Datablocks in Networking** section below.

- **Collision & Physics** ? The Server performs collision checks and important physics calculations like rigid-body dynamics that affect game objects.
- **Security** ? The Server can kick bad players and cheaters out of the game, and can be setup to detect hacks being performed by a bad client.
- **Message Routing** ? The server can route messages, such as in-game chats, between clients.

## The Client

The Client also has several important responsibilities, so that it can work well with the server and provide a quality game play experience:

- **Rendering** ? the Client is responsible for rendering the game for the player, and may be responsible for non-important physical simulations, such as cloth simulations and particle effect physics.
- **Sound** ? The Client instance plays back sound effects and music tracks on the local machine and doesn't involve the server with sound playback.
- **Input** ? The Client accepts player input and sends that information to the server.
- **User Preferences** ? Many games store user preferences on the local machine where the client instance runs.
- **Prediction** ? The Client can predict what will happen to game objects in the short term while it waits for the server to synchronize, in order to maintain correct-looking game behavior.
- **Interpolation** ? As part of the prediction process, the Client can determine where it needs to be between where it thought it needed to be and where the server tells it to go.

### 24.15.3 Datablocks in Networking

Datablocks are a useful way to have game objects share common data, such as which model to use, physics properties, ammo type, whatever is relevant to the class the datablock is associated with. This saves Memory and makes it easier to create new types of objects by deriving new datablocks from old datablocks and overriding only the data that is different. Datablocks are declared on the server when a mission is started, and wired over to clients when they begin downloading the mission data. Once downloaded, they cannot be changed:

```
// An Example Datablock
// From art/datablocks/weapons/rocketLauncher.cs for the rocket launcher ammo:

datablock ItemData(RocketLauncherAmmo)
{
    // Mission editor category
    category = "Ammo";

    // Add the Ammo namespace as a parent. The ammo namespace provides
    // common ammo related functions and hooks into the inventory system.
    className = "Ammo";

    // Basic Item properties
    shapeFile = "art/shapes/weapons/SwarmGun/rocket.dts";
    mass = 2;
    elasticity = 0.2;
    friction = 0.6;
```

(continues on next page)

(continued from previous page)

```
// Dynamic properties defined by the scripts
pickUpName = "Rockets";
maxInventory = 20;
};
```

Just like the benefit datablocks provide to memory, they also save bandwidth as the common data they contain is only sent once. Datablocks also make it easier for a client to get mods from the server without updating scripts, as the modded behavior is acquired when the mod datablocks are downloaded.

When the server begins a mission with the client it sends relevant datablocks to the client as part of a multiphase loading process. Datablocks are sent in the first phase of the process, as all in-game objects will need the datablocks to be initialized with the correct settings. Once all datablocks have been downloaded, the server moves on to Phase 2 of the mission downloading process.

For more information on datablocks, you can read: **Datablock Editor** - [TODO Internal Link](#) - World Editor/Editors/DatablockEditor

## 24.15.4 Network Connection Classes ? Linking Client and Server

Torque uses several connection object classes to provide multiplayer networking facilities. The basic functionality is defined in the `NetConnection` and expanded upon with the other connection classes.

### NetConnection

The Base multiplayer networking object class is the `NetConnection` class, which provides the functionality to create connections between two instances of torque (or the same instance if the client is also the server):

```
//an example of NetConnection in the wild?
%connect = new NetConnection(MyNetConnection);
RootGroup.add( MyNetConnection );

%connect.connect(%someAddress); //connect MyNetConnection to %someAddress

//if successful, you are now networked on a basic level
```

The important console method commands to note in the class are:

- **NetConnection.connect( %address )** ? attempts to connect this object to another `NetConnection` object in an instance of Torque running at the network `%address`.
- **NetConnection.connectLocal()** - attempts to connect this `NetConnection` object to the local server when running the client and server in the same instance of Torque. Returns an empty string "" when successful and an error message otherwise.

### GameConnection

The primary subclass of `NetConnection` used by a Torque multiplayer game, utilizes everything that `NetConnection` does but adds game-specific networking functionality on top of that:

```
//making the GameConnection?
%gameCon = new GameConnection(MyGameConnection);
RootGroup.add(MyGameConnection);

MyGameConnection.connect(%someAddress); //connect MyNetConnection to %someAddress
```

The `GameConnection` class enables what is known as the **Control Object**, which can be anything derived from the `ShapeBase` engine class. The client instance of the game tracks control from this object, such as the **Player** or the **Camera** used in editing, and sends it to the server. `GameConnection` has both client and server-side console methods:

```
//setting a control object, we do this on the server side
MyGameConnection.setControlObject(PlayerOne);

//change the view to third-person, on the server side
MyGameConnection.setFirstPerson(false);

//other fun stuff?
```

The `GameConnection` object applies changes to the client control object based on Player input to the game via the **Move** engine structure, which contains positional and rotational changes as well as trigger state changes. The Moves are collected based on time, applied to the client object, and then sent over to the server for processing.

## ServerConnection

This is the named instance of a **GameConnection** object that represents the Client Connection to the Server. It is created on the scripting level:

```
//from core/scripts/server/server.cs
%conn = new GameConnection( ServerConnection );
RootGroup.add( ServerConnection );

%conn.setConnectArgs( $pref::Player::Name );
%conn.setJoinPassword( $Client::Password );
```

## LocalClientConnection

This is the named instance of a `NetConnection/GameConnection` (depending on which class you use) object that is created when it is `connectLocal()` method is performed successfully:

```
//furthering the ServerConnection example from before
%conn = new GameConnection( ServerConnection );
RootGroup.add( ServerConnection );

%conn.setConnectArgs( $pref::Player::Name );
%conn.setJoinPassword( $Client::Password );

// LocalClientConnection is made right here, on the engine level
%result = %conn.connectLocal();

if( %result != "" )
{
    %conn.delete();
    destroyServer();

    return false;
}
```



## Setting Up The Server

- First, we need to set up the **Port** that Torque will be using for communication on the computer. A port is a communication channel a computer uses to filter network traffic. The Stock Example uses port# 28000 by default, set in core/prefs.cs. You can use the script helper function `portInit(%port)`, which will try to find an open port with the console function `setNetPort(%port)`. (**Single-Player games can skip this step**):

```
//an example of setting the port
%myPort = 12345;
//?
if( setNetPort(%myPort) )
{
    //success!
    echo("successfully connected to port:" SPC %myPort);
}
else
{
    //failure
    error("error connecting to port:" SPC %myPort);

    //fallback behavior, maybe try a new port?
}
```

- Next, we need to enable the Torque instance to allow network Connections, we do this by the console function `allowConnections(%enable)`. (**Single-Player games can skip this step**):

```
//activate connections on our selected port
allowConnections(true);
```

- Afterwards, we setup our `ServerGroup`, load up our datablocks, and begin loading the selected mission:

```
//an updated example from core/scripts/server/server.cs?

// Load the level
$ServerGroup = new SimGroup(ServerGroup);

// Load up any core datablocks
exec("core/art/datablocks/datablockExec.cs");

// Let the game initialize some things now that the
// the server has been created
onServerCreated();

loadMission(%level, true); //only true if loading first mission
```

- All of the above is can be performed with the `createServer(%serverType, %level)` script helper function:

```
//do all of the above in one call
if( createServer(%type, %myMission) )
{
    //server for mission created properly
    echo(%type SPC "server created successfully for mission:" SPC %myMission);
}
else
{
    //fallback behavior
    error("error in" SPC %type SPC "server creation process!" SPC %myMission);
}
```

- At this point, if we're running a client on the same instance as the server, we can create our **ServerConnection** object, and connect it to the local server instance:

```
// from server.cs again? with a few extra comments

//create our server connection object
%conn = new GameConnection( ServerConnection );

// RootGroup is the master SimGroup for the entire instance
RootGroup.add( ServerConnection );

%conn.setConnectArgs( $pref::Player::Name );
%conn.setJoinPassword( $Client::Password );

//if you've modified or subclassed GameConnection,
// your additional connection settings might //go here

%result = %conn.connectLocal(); //create the LocalServerConnection
if( %result != "" )
{
    //uh-oh, get rid of the bad connection
    %conn.delete();
    destroyServer();

    return false;
}
```

- We can create the server AND connect locally with the `createAndConnectToLocalServer( %serverType, %level)` script helper function, which also calls the `createServer()` script helper function to make the server:

```
//do all of the above
if( createAndConnectToLocalServer(%type, %myMission) )
{
    //local server & local client connection for mission created properly
    echo(%type SPC "server & client created successfully for mission:" SPC
    ↪%myMission);
}
else
{
    //fallback behavior
    error("error in" SPC %type SPC "server & client creation process!" SPC
    ↪%myMission);
}
```

## Setting Up the Client

It is pretty straightforward, you can look at how the script helper function `connect(%server)` works to see this in action:

```
// Example from the connect(%server) script helper function in :
// core/scripts/client/missionDownload.cs?

function connect(%server)
{

    //First, Create ServerConnection object.
```

(continues on next page)

(continued from previous page)

```

%conn = new GameConnection(ServerConnection);
RootGroup.add(ServerConnection);

// Next, Setup our connection settings,
// such as player name, password,
// and any other game-specific extensions.
%conn.setConnectArgs($pref::Player::Name);
%conn.setJoinPassword($Client::Password);

// Call the GameConnection.connect(%server) method to initiate the connection.

%conn.connect(%server);
}

```

## Using the GameCore Package

A Package is a set of modified scripts that can be loaded “over” preexisting functions, and unloaded to remove the alternate functionality. The **GameCore** package in a stock Torque project sets up the FPS Single and Multiplayer games. You can find the GameCore package functions in `game/scripts/server/gameCore.cs`.

The **GameCore** Package overrides the “blank” functionality associated with many **GameConnection** console callbacks with game-specific behavior, like informing the client it has joined a game with a welcome message, putting in a message when another player enters/leaves the game, how the player is spawned in the world, setting the active inventory, and so on and so forth:

```

//game core example, onConnect override, short version?
function GameConnection::onConnect(%client, %name)
{
    // Send down the connection error info, the client is responsible for
    // displaying this message if a connection error occurs.
    messageClient(%client, 'MsgConnectionError', "",
    ↪$Pref::Server::ConnectionError);

    // Send mission information to the client
    sendLoadInfoToClient(%client);

    //other stuff?

    // Save client preferences on the connection object for later use.
    %client.gender = "Male";
    %client.armor = "Light";
    %client.race = "Human";
    %client.skin = addTaggedString("base");
    %client.setPlayerName(%name);
    %client.score = 0;
    %client.kills = 0;
    %client.deaths = 0;

    // Inform the client of all the other clients
    %count = ClientGroup.getCount();
    for (%cl = 0; %cl < %count; %cl++)
    {
        //?
    }
}

```

(continues on next page)

(continued from previous page)

```

// Inform the client about joining
//?

// Inform all the other clients of the new guy
//?

// If the mission is running, go ahead download it to the client
if ($missionRunning)
{
    %client.loadMission();
}
else if ($Server::LoadFailMsg != "")
{
    messageClient(%client, 'MsgLoadFailed', $Server::LoadFailMsg);
}
$Server::PlayerCount++;
}

```

To make your own game-specific functionality you could re-write the **GameCore** package or make a new package that overrides the functions that the **GameCore** Package defines.

### 24.15.5 Sending Commands in Torque's Client/Server Model

Torque has a very simple setup for sending script commands between the client and server, allowing for a great deal of flexibility in setting up your own commands.

#### Client to Server Commands

The way to send a command from the client to the server is with the `commandToServer(%cmdname, %arglist?)` console function. The `%arglist?` is any number of optional arguments that the function needs to pass to the server command:

```

//commandToServer example, tell the server to start some giant laser mayhem
commandToServer('GiantLaserAttack', %laserPosition, 12, "1.0 0.0 0.75 0.9");

```

For the server to process the command, use the prefix `serverCmd`, followed by the command name. The first argument is the game object id of the client that send the command, followed by the optional arguments given it:

```

//serverCmd example, notice the args match up with our commandToServer() call
function serverCmdGiantLaserAttack(%client, %position, %powerLevel, %laserColor)
{
    if( $GiantLaserActive == false)
    {
        echo("Received GLA from Client:" SPC %client);
        //perform some giant laser mayhem
        beginGiantLaserAttack(%position, %powerLevel, %laserColor);
    }
    //there can only be one giant laser attack at any given time!
}

```

## Server to Client Commands

Sending commands from the server to the client is *almost* the mirror version of sending them from the client to the server. To send a command to the client, use the `commandToClient(%client, %cmdName, arglist..)` console function. You have to specify a client to send the command to. Unlike with `commandToServer()`, there could be multiple clients:

```
//commandToClient example, update our world damage state
commandToClient(%thatClient, 'UpdateGiantLaserWorldDamage', %laserPosition, %radius);

//sending a command to all the clients
%count = ClientGroup.getCount();
for (%i = 0; %i < %count; %i++)
{
    %cl = ClientGroup.getObject(%i);
    commandToClient( %cl, 'UpdateGiantLaserWorldDamage' , %laserPosition,
    ↪%radius);
}
```

And similar to the `serverCmd` prefix, we have the `clientCmd` prefix + `commandName`:

```
//clientCmd example, updating our GiantLaser world damage
function clientCmdUpdateGiantLaserWorldDamage( %position, %radius)
{
    destroyPlayersInArea(%position, %radius);
    destroyVehiclesInArea(%position, %radius);
    destroyPropsInArea(%position, %radius);
    applyWorldDamageDecal(%postion, %radius, "GiantLaser");
    breakAllWindowsInArea(%position, %radius);
}
```

## 24.15.6 Making Your Own Commands

As you learned with the previous section, making client/server commands in Torque is really easy. With this example, you'll learn how to make your own client/server commands.

You'll use a timed command on the client to send a command to the server saying "ready to teleport". The server will teleport the player and send a command to the client to echo a message to the console; Armed with that, you can modify the example and play with making your own commands to make the server and client do whatever you want them to!

You'll be modifying the Stock Torque Full Template for this example, so if you have that ready to go then you're all set.

### Client Setup

First, add a file on the client for our client side commands. Call this file `myClientCommands.cs` and make it in the `game/scripts/client` directory,. Once you've done that, add a few functions to that file:

```
//this will send the command that you're ready to start the teleport procedure
function sendTeleportSignal()
{
    %time = 5000; //a five second delay should be fine

    // call the serverCmdTeleportReady, with a time argument of 5 seconds
    commandToServer('TeleportReady', %time);
}
```

(continues on next page)

(continued from previous page)

```

}

//this clientCmd will report the distance the server teleported the player
function clientCmdAcknowledgeTeleport(%distance)
{
    // tell the user how far up they were teleported
    echo("VOIP! You were teleported! Distance:" SPC %distance);
}

```

## Server Setup

Next, add our server command functions. Make a file in the game/scripts/server directory called myServerScripts.cs, and fill it in with the following code:

```

//this will set us up to teleport the clients in a few seconds
function serverCmdTeleportReady(%client, %time)
{
    schedule(%time, 0, "beginTeleport"); //schedule our teleport
}

//this function will send all the clients the teleport command
function beginTeleport()
{
    %count = ClientGroup.getCount(); //get the count for all of our clients
    for (%i = 0; %i < %count; %i++)
    {
        %cl = ClientGroup.getObject(%i); //get each client
        %dist = 5 + (getRandom() * 5); //generate a random distance between 5_
↪and 10
        %controlObject = %cl.getControlObject(); //nab our control object for_
↪this client
        teleportUp(%controlObject,%dist); //teleport our object
        commandToClient( %cl, 'AcknowledgeTeleport', %dist); //send the_
↪command to the client
    }
}

//teleport our control object up!
function teleportUp(%controlObject, %distance)
{
    %pos = %controlObject.position; //nab position

    %height = getWord(%pos,2); //get the current height

    %height += %distance; //add the distance

    %pos = setWord(%pos, 2, %height); //set the position

    %controlObject.position = %pos; //set the control the position of the object
}

```

### Execution Setup

Now, go to `game/scripts/main.cs`, and add the following under where the normal `init.cs` scripts are executed, so we can execute our new scripts:

```
// Load the scripts that start it all...
exec("./client/init.cs");
exec("./server/init.cs");

//add our own command scripts
exec("./client/myClientCommands.cs"); //exec our client commands
exec("./server/myServerCommands.cs"); //exec our server commands
```

### Mission Setup

Finally, go to `game/core/scripts/client/mission.cs`, and add the following line at the end of the `clientStartMission()` script function, after `$Client::missionRunning = true;`, this is so a client can trigger the teleport sequence whenever they enter the game:

```
//?

// Done.

$Client::missionRunning = true;

sendTeleportSignal(); //start our teleportation sequence
```



```

Using background sleep time: 200
DirectInput deactivated.
Activating DirectInput...
Window focus status changed: focus: 1
Exporting server prefs...
Starting multiplayer mode
Binding server port to default IP
UDP initialized on port 28000
Validation required for shape: art/shapes/actors/Gideon/gideon.dts
Validation required for shape: art/shapes/actors/Gideon/gideon.dts
*** LOADING MISSION: levels/Empty Terrain.mis
*** Stage 1 load
*** Stage 2 load
game -> activatePackages
*** Mission loaded
Connect request from:
Connection established 4090
CADD: 4091 local
*** Sending mission load to client: levels/Empty Terrain.mis
Mapping string: ServerMessage to index: 0
Mapping string: MsgConnectionError to index: 1
Mapping string: MsgLoadInfo to index: 2
Mapping string: MsgLoadDescription to index: 3
Sending heartbeat to master server [IP:92.242.140.1:28002]
Mapping string: MsgLoadInfoDone to index: 4
Mapping string: MsgClientJoin to index: 5
Mapping string: Welcome to the Torque demo app v1. to index: 6
Mapping string: Visitor to index: 7
Mapping string: MissionStartPhase1 to index: 8
*** New Mission: levels/Empty Terrain.mis
*** Phase 1: Download Datablocks & Targets
% - PostFX Manager - Executing core/scripts/client/postFx/default.postfxpreset.cs
% - PostFX Manager - Applying from preset
% - PostFX Manager - PostFX enabled
Mapping string: MissionStartPhase1Ack to index: 0
Validation required for shape: art/shapes/actors/Gideon/gideon.dts
Validation required for shape: art/shapes/actors/Gideon/gideon.dts
Mapping string: MissionStartPhase2 to index: 9
*** Phase 2: Download Ghost Objects
Mapping string: MissionStartPhase2Ack to index: 1
Ghost Always objects received.
Mapping string: MissionStartPhase3 to index: 10
Client Replication Startup has Happened!
fxFoliageReplicator - replicated client foliage for 0 objects
*** Phase 3: Mission Lighting
Mission lighting done
Mapping string: MissionStartPhase3Ack to index: 2
Mapping string: MissionStart to index: 11
Mapping string: SyncClock to index: 12
Mapping string: RefreshWeaponHud to index: 13
Mapping string: swarmer.png to index: 14
Mapping string: reticle_rocketlauncher to index: 15
Mapping string: TeleportReady to index: 3
*** Initial Control Object
Mapping string: setNumericalHealthHUD to index: 16
Mapping string: AcknowledgeTeleport to index: 17
VOIP! You were teleported! Distance: 7.60123

```

And that is all there is to it! Every time a new client Joins the party, everyone gets teleported a random distance. Now that you know how to make your own client and server scripts, you can start working on your own networking ideas.

### 24.15.7 Further Reading

For further reading and a deeper understanding of Torque's Netcode, check out:

- [Older TGEA networking article](#)



## 25.1 Overview

### 25.1.1 Introduction

SFX is the new 2D and 3D audio system for Torque 3D. It was designed to provide the basic features for sound playback across multiple platforms, sound devices, and audio libraries. This guide is a high level overview of the SFX module and what systems it supports.

**Recommendation:** For the best cross-platform support as well as for the best compatibility with future additions to SFX, it is recommended to either use OpenAL or FMOD with SFX. A number of future additions will likely not be supported with other sound APIs.

### 25.1.2 Two Playback Types

Each individual sound can be played back in two ways. The type of playback used by the sound system for a particular sound is determined by the “isStreaming” property of its SFXDescription (SFXInterface.html#SFXDescription)

*Note: With either form of playback, sound data is stored in uncompressed form on the device. For compressed formats, decompression occurs during sound loading. For streamed sounds, this is a continuous process. All sound loading happens on worker threads regardless of how it is played back so it does not tax the main thread.*

#### Buffered

**DEFINITION:** Sample data completely offloaded to a sound device in one contiguous chunk. Ideal for small sounds, such as “place once” effects (e.g. gun fire, collision audio, etc).

#### Advantages

1. Fast seeking
2. No overhead after loading

3. Sound data can be shared between simultaneous playbacks

### Disadvantages

1. May consume a lot of memory on the device (depending on sample data size)
2. May take longer to load (depending on sample data size)
3. May fail to load entirely depending on device (depending on sample data size)
4. Needs entire sound stream to be available before playback can start

### Streamed

DEFINITION: Small buffer allocated on a sound device and data is progressively loaded in the background as playback progresses. Ideal for large sounds (e.g. music) and continuous streams (e.g. voice chat). This feature is fully threaded; playback will continue normally even if Torque 3D is, for example, busy loading a level.

### Advantages

1. Small footprint on sound device
2. Loads quickly
3. Can play sound streams for which only part of the data is available

### Disadvantages

1. Incurs a certain continuous overhead
2. Data for a sound cannot be shared between playbacks
3. Seeking needs to reset stream feed and will generally incur a delay
4. End-of-stream notifications may be imprecise (depends on actual sound device used)

## 25.1.3 3D Sound

Sounds can be positional (3D) or non-positional (2D). Positional sounds are located in Torque's three-dimensional world. Whether a sound is 2D or 3D is determined by the "is3D" property of its SFXDescription. ([SFXInterface.html#SFXDescription](#))

3D sounds will be mixed at varying volumes and distributed dynamically to the sound card's output channels depending on the player's position and velocity in relation to the sound's own position and velocity. This is computed independently for each playing 3D sound.

Each 3D sound has a position, an orientation, and a velocity. To represent 3D sounds by real objects in Torque 3D, use SFXEmitters. ([SFXInterface.html#SFXEmitter](#))

**!! Important !!:** 3D sounds must be single channel (mono). This is a requirement coming from the sound APIs. Currently, 3D sound is completely agnostic to the environment. This means that to the SFX system the entire world is one single empty space and sound will propagate uniformly and unencumbered in all directions.

### Distance Attenuation

Just like sound in the real world, 3D sounds become less audible the farther away you are from their source. Distance attenuation then refers to the process of computing the level of volume that a given 3D sound has at a certain distance away from its origin.

The process of setting up distance attenuation is described in more detail in [Configuring 3D Playback](#). ([SFXInterface.html#Configuring\\_3D\\_Playback](#))

## Sound Cones

Distance attenuation works uniformly in all direction regardless of the actual direction of the sound. Sound cones allow you to add directional volume attenuation to distance attenuation.

There are two cones defined on each 3D sound that affect volume attenuation. Both have a their upwards axis aligned with the sound's direction and their tips at the sound's origin. The width of each cone is defined by an angle. By default this angle is 360 degrees meaning that the sound spreads in all directions.

By narrowing the angle, the cone will get smaller and the sound will spread only across a certain range in its direction. Of the two cones, one is the inner cone and one is the outer cone.

Within the inner cone, the 3D sound will retain full volume as specified in its SFXDescription (attenuated only by the sound channel it is assigned to and, of course, the master volume of the system).

Outside of the outer cone, the 3D sound's volume will be attenuated by the scale factor set by **"coneOutsideVolume"** property of its SFXDescription. Then, within the transitioning zone between the inner cone to outside the outer cone, the volume will gradually shift from an attenuation of 1.0 (inside inner cone) to an attenuation of **"coneOutsideVolume"**.

Note: Cone settings from SFXDescriptions can be overridden for individual SFXSources using the `setCone()` method. ([SFXInterface.html#SFXSource](#))

To gain a better understanding of how sound cones work, read through this MSDN article on sound cones. ([https://msdn.microsoft.com/en-us/library/ee418803\(v=VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ee418803(v=VS.85).aspx))

Doppler Effect Also known as Doppler Shift, this effect is the change in frequency of a wave for an observer moving relative to the source of the waves, similar to how you perceive a loud motorcycle approaching and passing you. The following articles contain a more in depth description:

- Wikipedia Entry ([https://en.wikipedia.org/wiki/Doppler\\_effect](https://en.wikipedia.org/wiki/Doppler_effect))
- Kettering Article (<http://www.acs.psu.edu/drussell/Demos/doppler/doppler.html>)

### 25.1.4 Playback Virtualization

In any given situation, a game may have more sounds playing concurrently than are actually supported by the underlying device. To cope with this, the SFX system uses what is called playback virtualization.

During SFX's update routine, the system will compute effective volumes for all playing sounds. If there are more sounds playing than are supported by the underlying SFX device, the system will reassign voices from the least audible (usually the sounds farthest from the player) sounds to more audible sounds.

A sound source that is playing but loses its voice on the device will transition into virtualized playback mode. In this mode, the sound will continue to have its play cursor advance in real-time but there will not be actual audio output on the device.

Voice distribution is re-evaluated on each SFX update so a given sound that has been transitioned to virtualized playback mode may later regain a voice and transition back to normal playback.

### 25.1.5 Supported Audio Formats

SFX supports WAV and Vorbis across all devices. In addition, devices may implement their own file loading which will take precedence over the built-in loading code. See Supported APIs for more information.

Live Asset Updating is supported by SFX meaning that if a sound file changes on disk, it will automatically be reloaded by the SFX system. All sources playing the sound will temporarily transition to virtualized playback and then transition to normal playback when the file has been reloaded.

## WAV

- **Premise:** Uncompressed format that is most useful for sound production
- **Pro:** Uncompromising sound quality
- **Con:** Consumes lots of disk space

Note: At the moment, none of the enhanced features of WAVs (loops, markers, etc.) are supported.

## Ogg/Vorbis

- **Premise:** High-quality compressed format (usually outperforms rival lossy sound compression formats such as MP3 or WMA)
- **Pro:** Very good quality/size ratio
- **Con:** Compression is lossy (with proper settings, this should not be noticeable in most all game settings)

### 25.1.6 Supported Sound APIs

The SFX system supports several different sound APIs.

*Note: Switching sound devices at runtime will preserve all `SFXProfile` and `SFXSource` states. `SFXSources` that are currently playing will temporarily transition to virtualized playback.*

#### DirectSound

Platform: Windows Description: Standard DirectX audio API.

#### FMOD

**Platforms:** Windows, Mac, XBox, PS3 **Description:** High-quality, highly cross-platform sound API. Must be installed separately. For commercial releases, a license must be obtained.

With FMOD selected, all file loading and streaming will be taken over by the device. If for some reason you want to disabled this feature, set the following global variable in TorqueScript:

```
$pref::SFX::FMOD::noCustomFileLoading = 1)
```

The following formats are supported:

- .aiff
- .asf
- .asx
- .dls
- .flac
- .fsb
- .it
- .m3u
- .mid

- .mod
- .mp2
- .mp3
- .ogg
- .pls
- .s3m
- .vag
- .wav
- .wax
- .wma
- .xm
- .xma (on Xbox only)

*Notes:*

- To download FMOD, visit the FMOD homepage <http://www.fmod.org/>
- To purchase a very friendly priced indie license of FMOD for your game, visit the GarageGames store. <http://www.garagegames.com/products/fmod>
- At the moment, Live Asset Updating is not available with sounds that have been loaded directly through FMOD.
- Currently, using play list files with FMOD is the only way to use play lists with SFX.

## OpenAL

**Platforms:** Windows, Mac **Description:** A cross-platform 3D audio API appropriate for use with gaming applications and many other types of audio applications.

*Note: On Windows, OpenAL must be installed separately. Visit the OpenAL website for more details*

- <http://connect.creativelabs.com/openal/default.aspx>

## XAudio

**Platforms:** Windows, XBox **Description:** InterTrust's fast and robust, multi-platform solution for digital playback. Targets MPEG Audio (MP1, MP2, and MP3) decoding. It is offered in the form of a developer's kit (SDK), which includes the Xaudio decoding engine.

## Null

**Platforms:** Windows, Mac **Description:** Stub device for dedicated servers. Simulates playback. No actual audio output.

## 25.1.7 Conclusion

This document is intended to provide you with the base knowledge of SFX's capabilities.



## 25.2 Interface

### 25.2.1 Overview

SFXSources are the interface to sound playback. They are created through the SFXSystem and control all aspects of playback. The SFX system can be interfaced through a number of console methods.

**Note:** *Do not create SFXSources directly through object declarations (i.e. new or singleton). This is inhibited by the SFX system.*

**SFXSources** are either created from SFXProfiles or directly from SFXStreams. The latter method is only available internally from within the engine and not exposed to script.

**SFXProfiles** are created by the user. They combine sound files with SFXDescriptions that describe how to play back the particular file.

**SFXDescriptions** are datablocks with a series of properties that describe aspects of sound playback.

**SFXSources** create SFXVoices (playback controllers) and SFXBuffers (sample data buffers) on SFXDevices. This happens internally.

**SFXStreams** are used to load sample data into SFXBuffers. For streaming buffers, this is a continuous process. This happens internally.

An **SFXListener** instance is coupled to the SFXSystem and defines the location and velocity of the listener in 3D space. This is used for volume attenuation of 3D sounds. This happens internally. The listener is automatically updated from the game's control object.

### 25.2.2 Channels

To allow volume level adjustment to a whole batch of sounds, sounds are grouped into logical volume channels. Music can thus be separated from sound effects or voices and have its volume adjusted independently.

Volume channels are numbered from 0 onwards. There is a maximum of 32 channels. The core scripts currently define and use three separate channels (defined in **core/scripts/client/audio.cs**):

- \$GuiAudioType = 1;
  - Use for sounds relating to the interface.
- \$SimAudioType = 2;
  - Use for in-game sound effect and environmental sounds.
- \$MessageAudioType = 3;
  - Use for notifications (such as from chat) and possibly voices.
- \$MusicAudioType = 4;
  - Use for background music.

**Note:** *The master volume set on the SFX system simultaneously scales all of the volume channels.*

A set of SFXDescriptions preset to this set of channels is made available through core/scripts/client/audio.cs. These should best be used as copy-sources for custom SFXDescriptions.

- AudioGui
- AudioSim
- AudioMessage

- AudioMusic

### 25.2.3 Descriptions

Playback setups are described in SFXDescriptions. These are defined in **art/datablocks/audioProfiles.cs**. Some useful examples are below:

```
*3D Sounds
  o Single-Shot Sounds
    + AudioDefault3D
      # is3D = true
      # referenceDistance = 20.0
      # maxDistance = 100.0
      # channel = $SimAudioType
    + AudioClose3D
      # is3D = true
      # referenceDistance = 10.0
      # maxDistance = 60.0
      # channel = $SimAudioType
    + AudioClosest3D
      # is3D = true
      # referenceDistance = 5.0
      # maxDistance = 30.0
      # channel = $SimAudioType
  o Looping Sounds
    + AudioDefaultLoop3D
      # is3D = true
      # isLooping = true
      # referenceDistance = 20.0
      # maxDistance = 100.0
      # channel = $SimAudioType
    + AudioCloseLoop3D
      # is3D = true
      # isLooping = true
      # referenceDistance = 10.0
      # maxDistance = 60.0
      # channel = $SimAudioType
    + AudioClosestLoop3D
      # is3D = true
      # isLooping = true
      # referenceDistance = 5.0
      # maxDistance = 30.0
      # channel = $SimAudioType
*2D Sounds
  o Audio2D
    + channel = $SimAudioType

  o AudioStream2D
    + isStreaming = true
    + channel = $SimAudioType

  o AudioLoop2D
    + isLooping = true
    + channel = $SimAudioType

  o AudioStreamLoop2D
    + isLooping = true
```

(continues on next page)

(continued from previous page)

```
+ isStreaming = true
+ channel = $SimAudioType

*Music
  o AudioMusic2D
    + isStreaming = true
    + channel = $MusicAudioType

  o AudioMusicLoop2D
    + isStreaming = true
    + isLooping = true
    + channel = $MusicAudioType

  o AudioMusic3D
    + isStreaming = true
    + is3D = true
    + channel = $MusicAudioType

  o AudioMusicLoop3D
    + isStreaming = true
    + isLooping = true
    + is3D = true
    + channel = $MusicAudioType
```

### 25.2.4 Configuring 3D Playback

There are multiple options available for configuring volume attenuation of 3D sounds on the device. The settings are exposed as script variables and must be set in script (either from the console or add them to **scripts/client/prefs.cs**).

#### **\$pref::SFX::distanceModel**

The distance model determines the rolloff function for 3D sounds, i.e. the way the volume attenuates as you move away from a 3D sound. The chosen model affects distance attenuation of all 3D sounds. Currently, there are two models available:

##### **“linear”**

Starting from a sound’s reference distance, the volume fades linearly towards its set maximum distance at which point it reaches zero.

##### **Notes:**

- Linear rolloff is not supported with DirectSound.
- Linear rolloff is unaffected by the rolloff factor set on the device.

##### **“logarithmic”**

Starting from a sound’s reference distance, the volume halves every reference distance steps. This is the more realistic behavior for distance attenuation. In this model, the maximum distance only determines at which distance volume no longer decreases.

Instead, attenuation simply stops at the set maximum distance. Attenuation can be scaled by the rolloff factor to be faster or slower.

### \$pref::SFX::dopplerFactor

The doppler shift to apply to 3D sounds based on the relative velocity to the listener. Higher values give more pronounced doppler effects. Default is 0.5.

### \$pref::SFX::rolloffFactor

The rolloff factor scales the reference distance of a sound to determine how fast attenuation decreases a sound's volume. At 1.0, every reference distance steps will halve the volume of a sound. At 0.5, every reference distance steps will have the quarter of the previous step's volume.

**Note:** *The rolloff factor only affects logarithmic distance attenuation.*

## 25.2.5 Script Classes

### SFXDescription

#### Description

SFXDescriptions tell the sound system how to play back a sound, i.e. they provide the parameters when setting up playback for a sound on the device.

#### Properties

- **volume [float]:** Base volume level for the sound. This will be the starting point for volume attenuation on the sound. Must be between 0 (min) and 1 (max). Default is 1.
- **pitch [float]:** Pitch scale. This speeds up or slows down playback. Must be greater than 0. Default is 1.
- **isLooping [bool]:** If true, the sound will be played in an endless loop. Default is false.
- **isStreaming [bool]:** If true, the sound will be progressively streamed; if false, the sound will be buffered in whole. Default is false.
- **is3D [bool]:** If true, the sound is positional and mixed according to its spatial properties. Default is false.
- **referenceDistance [float]:** Distance at which volume attenuation begins. Up to this distance, the sound retains its base volume. Also, in the exponential distance model, the reference distance determine how fast the sound volume decreases with distance. Each referenceDistance steps (scaled by the rolloff factor), the volume halves. A rule of thumb is that for sounds that require you to be close to hear them in the real world, set the reference distance to small values whereas for sounds that are widely audible set it to larger values. Only for 3D sounds.
- **maxDistance [float]:** The distance at which attenuation stops. In the linear distance model, the attenuated volume will be zero at this distance. In the logarithmic model, attenuation will simply stop at this distance and the sound will keep its attenuated volume from there on out. Only for 3D sounds.
- **coneInsideAngle [float]:** Specifies the angle of the inside cone in degrees. Valid values are from 0 to 360. Must be less than coneOutsideAngle. Default is 360. Only for 3D sounds.
- **coneOutsideAngle [float]:** Specifies the angle of the outside cone in degrees. Valid values are from 0 to 360. Default is 360. Only for 3D sounds.
- **coneOutsideVolume [float]:** Determines the volume scale factor applied to a source's base volume level outside of the outer cone. In the outer cone, starting from outside the inner cone, the scale factor smoothly interpolates from 1.0 (within the inner cone) to this value. At the moment, the allowed range is 0.0 (silence) to 1.0 (no attenuation) as amplification is only supported on XAudio2 but not on the other devices. Only for 3D sound.

- **channel [int]:** Volume channel that the sound will get assigned to. The base volume level of the sound will be scaled by the channel's volume level. Default is 0.
- **fadeInTime [float]:** Number of seconds to gradually fade in volume from zero when playback starts.
- **fadeOutTime [float]:** Number of seconds to gradually fade out volume to zero before playback stops.
- **streamPacketSize [int]:** Number of seconds of sample data to read per streaming packet. Only for streamed sounds and only when SFX's own streaming system is used.
- **streamReadAhead [int]:** Number of stream packets to keep buffered ahead of time. A number of packets (usually three) will generally be kept immediately on the SFX device for playback. This number controls how many more packets are read ahead of time in addition to this. Higher numbers allow to better cope with lagging stream sources but for good resource consumption, this should be kept reasonably low. Only for streamed sounds and only when SFX's own streaming system is used.

### SFXProfile

#### Description

An SFXProfile combines a sound file with an SFXDescription that tells how to play back the sound. SFXProfiles are datablocks, so when you define server-side SFXProfiles use the **datablock** keyword. For client-side only profiles, use **new** or **singleton**.

For non-streamed sounds, the SFXProfile will keep a reference to the SFXBuffer on the device once the sound has been loaded. This will allow simultaneous playbacks of the same profile to share a single SFXBuffer. A profile's sound will automatically reload when its file on disk changes. This will also update all sources that are currently using the profile.

By default, the sound data contained in an SFXProfile will be loaded into the SFX device when the sound is first played. This will incur a short delay on the first use. To load the sound data ahead of time independent on the first use, set the "preload" property to true. This will cause sound data to be loaded when the profile object is added to the system.

Once loaded, the sound data will remain loaded on the device until either the device is destroyed or the SFXProfile is deleted. Be aware that only non-streamed sounds may be preloaded. Streamed sounds always incur a certain loading delay. To ready a streamed sound for playback, attach and ready an SFXSource before playing the sound.

#### Properties

- **filename [string]:** The name of the file to load. It is usually best to omit extensions with sound filenames. This allows you to easily switch formats without having to change scripts. The SFX system will scan through its list of supported file formats (including extended formats supported by specific devices) to find the full filename.
- **description [object]:** The SFXDescription to use when playing back the sound.
- **preload [bool]:** If true, the sound file will be loaded into the SFX system as soon as the profile object gets added to Torque's object graph. This helps ensure that sounds are immediately ready for playback when the profile is used. Be aware, though, that this will also result in immediate resource consumption on the SFXDevice. Preloading will not cause disruptions in the loading process as sound loading happens in the background. Only non-streamed sounds can be preloaded. This flag is ignored for streamed sounds. To load a streamed sound into ready state before playing, create an SFXSource for it.

#### Methods

**float getSoundDuration():** Return the duration (in seconds) of the sound referenced by the profile.

## SFXSource

### Description

Central controller for sound playback. SFXSources control the playback of a particular SFXProfile or SFXStream. SFXSources can only be created through `sfxCreateSource` or the various `sfxPlay` functions.

SFXSources are explicitly created by the user through SFX functions and are valid until they are deleted. However, to reduce bookkeeping required for single-shot sounds, the system keeps a list of so called “play-once sources.” An SFXSource that is created as a play-once source will only be valid for as long as it is playing. When it has finished playing, it will be automatically deleted during the next SFX update.

Note that while permanent references to play-once sources should not be stored in script, a reference will not become invalid in-midst of script execution.

Markers are used for notifications that are triggered when playback crosses over a certain playback position. Each marker has a name and an associated playback position expressed in seconds. When playback crosses over the position, the “`onMarkerPassed( %source, %markerName )`” callback will be called on the source.

This is useful for synchronizing logic to music and sound effects. Currently, using `setPosition()` on a source will not prevent markers being jumped over from triggering. Instead, these markers will immediately trigger on the next source update.

### Properties

- **statusCallback [string]:** Name of function to call when the status of the SFXSource changes. Must take two parameters “( object %source, string %newStatus )”. Default is empty which deactivates the callback. If this field is set, the source’s `onStatusChange` callback will not trigger.

### Callbacks

These methods may be defined on SFXSources by the user and are called by the system on specific events.

- **onMarkerPassed( object this, string name ):** Called when a notification marker has been passed by the playback cursor. “name” is the name of the marker.
- **onStatusChange( object this, string newStatus ):** Called when the playback status of the source changes. Set `getStatus()` for the possible values for “newStatus”

### Methods

- **bool isReady():** Returns true when the source has successfully loaded all its sound data and is ready for playback.
- **bool isPaused():** Returns true when the source is currently in paused state.
- **bool isPlaying():** Return true when the source is currently playing. Even though a given SFXSource is in playing state, it will not emit sound data on the device if it is in blocked state (awaiting data from its SFXStream) or is in virtualized playback mode.
- **bool isStopped():** Returns true when the source is currently stopped. Sources will start out in stopped state and will transition back into stopped state when they have finished playing.
- **string getStatus():** Return the current playback status of the source. Possible values are:
  - “playing”: source is currently playing

- “stopped”: source is not playing
  - “paused”: source has been paused
- **int getChannel():** Return the volume channel this SFXSource is assigned to.
- **float getDuration():** Return the total playback time of the SFXSource’s associated sound in seconds.
- **setTransform( vector pos, [ vector direction ] ):** Set the 3D position and optionally the direction of the SFXSource.
- **setCone( float innerAngle, float outerAngle, float outsideVolume ):** Set the 3D sound cone of the SFXSource.
- **setVolume( float volume ):** Set the (unattenuated) volume of the SFXSource. Must be between 0 (min) and 1 (max).
- **setPitch( float pitch ):** The frequency shift factor. A pitch of 1 plays back at the default frequency. A pitch of 0.5 of half the default frequency. The default frequency is the frequency of the source SFXStream.
- **float getPosition():** Returns the current position of the play cursor in seconds.
- **setPosition( float pos ):** Set the position of the play cursor in seconds.
- **play( [ float fadeInTime ] ):** Start or resume playback. If “fadeInTime” is given and is greater than 0, then the source will do a volume fade to its assigned volume in “fadeInTime” seconds. “fadeInTime” overrides a setting given in the SFXSource’s SFXDescription. If the sound referred to by the source is not yet fully loaded, there may be a delay before playback actually starts. If there are more active voices than supported by the current SFX device, one SFXSource (this or another one depending on which is deemed least important) will go into virtualized playback mode as a result of calling play().
- **stop( [ float fadeOutTime ] ):** Stop playback. If “fadeOutTime” is given and is greater than 0, then the source will do a volume fade to zero in “fadeOutTime” seconds and then stop. “fadeOutTime” overrides a setting given in the SFXSource’s SFXDescription.
- **pause( [ float fadeOutTime ] ):** Pause playback. If “fadeOutTime” is given and is greater than 0, then the source will do a volume fade to zero in “fadeOutTime” seconds and then pause. “fadeOutTime” overrides a setting given in the SFXSource’s SFXDescription.
- **addMarker( string name, float pos ):** Add a marker called “name” at “pos” seconds into playback. If playback passes the given position, the “onMarkerPassed” callback will trigger.

## SFXEmitter

### Description

An SFXEmitter is a 3D object that emits sound. It has no visible representation (except within the editor), but does have a true location and velocity in 3D world space. Even though an SFXEmitter is an object in 3D space, it need not necessarily emit a 3D sound. It can also emit non-positional 2D sounds.

This is useful for placing background music in a level. There are two ways to set up an SFXEmitter:

- **through a predefined SFXProfile (“profile” property)**
  - plays the sound specified by the profile
  - uses the profile’s SFXDescription
  - some of the emitter’s properties override the settings in the profile’s SFXDescription (see documentation below)
- **directly through a sound file (“fileName” property)**
  - uses the properties on the emitter to set up a custom SFXDescription



Setting a “profile” will take precedence over setting a “fileName”. Currently it is not possible to have an emitter go through a list of sounds (at least not without scripting).

## Properties

- **profile [SFXProfile]:** The sound profile to play on this emitter. Either use this or “fileName” to set the sound to play. This field will take precedence over “fileName”.
- **fileName [FileName]:** The sound file to play on this emitter.
- **playOnAdd [bool]:** If true, playback will be immediately started when the emitter is added to the scene. Applies regardless of whether “profile” or “fileName” is used.
- **isLooping [bool]:** If true, the emitter’s sound will loop infinitely. Only applies when using “fileName”. For “profile”, the profile SFXDescription’s “isLooping” value is used.
- **isStreaming [bool]:** If true, the sound will use streamed playback. Only applies when using “fileName”. For “profile”, the profile SFXDescription’s “isStreaming” value is used.
- **channel [int]:** The volume channel to play the sound in. Only applies when using “fileName”. For “profile”, the profile SFXDescription’s “channel” value is used.
- **volume [float]:** The base (unattenuated) volume at which to play the sound. Applies to both “profile” and “fileName”. Must be between 0 (min) and 1 (max).
- **pitch [float]:** The frequency shift factor at which to play back the sound. Must be greater than 0. Applies to both “profile” and “fileName”.
- **fadeInTime [float]:** Time in seconds to fade volume from zero to full intensity when starting/resuming playback. Must be greater or equal to 0. Only applies when using “fileName”. For “profile”, the profile SFXDescription’s “fadeInTime” value is used.
- **fadeOutTime [float]:** Time in seconds to fade volume out to zero before stopping/pausing playback. Must be greater or equal to 0. Only applies when using “fileName”. For “profile”, the profile SFXDescription’s “fadeOutTime” value is used.
- **is3D [bool]:** If true, the sound will be positional. Only applies when using “fileName”. For “profile”, the profile SFXDescription’s “is3D” value is used.
- **referenceDistance [float]:** The distance at which to start distance-based volume attenuation. Only applies to 3D sounds. Applies to both “profile” and “fileName”.
- **maxDistance [float]:** The distance at which to stop distance-based volume attenuation. Only applies to 3D sounds. Applies to both “profile” and “fileName”.
- **coneInsideAngle [int]:** Inside 3D cone angle in degrees. Must be between 0 (min) and 360 (max). Only applies to 3D sounds. Applies to both “profile” and “fileName”.
- **coneOutsideAngle [int]:** Outside 3D cone angle in degrees. Must be between 0 (min) and 360 (max). Only applies to 3D sounds. Applies to both “profile” and “fileName”.
- **coneOutsideVolume [float]:** Volume scale factor outside 3D cone. Must be between (0) (min) and 1 (max). Only applies to 3D sounds. Applies to both “profile” and “fileName”.

## Methods

- **play():** Sends network event to start playback of the emitter (if not already playing). If called on the client (the ghost), will immediately trigger playback.

- **stop():** Sends network event to stop playback of the emitter (if not already stopped). If called on the client (the ghost), will immediately stop playback.
- **string getPlaybackStatus():** Returns the playback status of the emitter. See `SFXSource.getStatus()`. If called on a server-side `SFXEmitter`, the emitter's client-side object (ghost) object on the local client connection will be queried.
- **bool isInRange():** Returns true if the SFX listener (local connection's control object) is currently within the max range of the emitter.

## 25.2.6 Script Functions

### Device Management

Before sound playback functions can be used, a valid sound device must be created through one of the given sound providers. When using the standard `game/core/` scripts, this is automatically taken care of during engine startup.

- **vector sfxGetAvailableDevices():** Returns a vector that describes each of the available devices. Individual devices are separated by newlines and individual properties of devices are separated by tabs.
- **bool sfxCreateDevice( string provider, string device, bool useHardware, int maxBuffers ):** Try to create a new sound device using the given properties. This function must be called before any of the sound playback functions can be used. If there currently is an active device, it will be deleted automatically. Returns true if the operation succeeded and the device has been created.

Each device entry has the following properties (in the order they appear in the vector):

- **providerName [string]:** The name of the sound provider (e.g. "FMOD")
- **deviceName [string]:** The name of the device made available by the provider.
- **hasHardware [bool]:** If true, the device has support for mixing in hardware.
- **maxBuffers [int]:** Maximum number of concurrent buffers supported by the device, i.e. the maximum number of concurrently audible voices. If this is exceeded, playback virtualization will kick in and distribute the available voices across the playing sounds.

In the `game/core` scripts, sound is automatically set up during startup in the `sfxStartup()` function. Sounds that are playing while the sound device is being changed will be temporarily transitioned to virtualized playback and then resume normal playback once the new device has been created.

- **sfxDeleteDevice():** Delete the currently active sound device and release all its resources. In the core scripts, this is done automatically for you during shutdown in the `sfxShutdown()` function. `SFXSources` that are still playing will be transitioned to virtualized playback mode. When creating a new device, they will automatically transition back to normal playback.
- **vector sfxGetDeviceInfo():** Return information about the currently active sound device. The return value is a tab-delimited string containing the following properties (in the order they appear in the vector):
- **providerName [string]:** Name of the sound provider that supplies the device (e.g. "FMOD").
- **deviceName [string]:** Name of the device on the provider.
- **usesHardware [bool]:** If true, device is set up to use hardware mixing.
- **maxBuffers [int]:** Maximum number of concurrent voices the device is configured to use.

## Configuration

- **float sfxGetMasterVolume():** Return the system master volume. This volume level scales the volume of all independent volume channels simultaneously. Default is 1.
- **sfxSetMasterVolume( float volume ):** Set the system master volume. Must be between 0 (min) and 1 (max). This will affect all currently active sounds.
- **float sfxGetChannelVolume( int channel ):** Return the volume level of the given channel. Will be between 0 and 1. Default is 1.
- **sfxSetChannelVolume( int channel, float volume ):** Set the volume level of the given channel. Must be between 0 (min) and 1 (max). This will affect all sounds currently playing on that channel.
- **string sfxGetDistanceModel():** Return the name of the distance model currently used for distance attenuation of 3D sounds. Currently, this is either “linear” or “logarithmic”. Default is set by “\$pref::SFX::distanceModel”. If unset, the linear distance model is used.
- **sfxSetDistanceModel( string model ):** Set the distance model to use for distance attenuation of 3D sounds. Must be either “linear” or “logarithmic”. This will affect the volume attenuation of all currently active 3D sounds.
- **float sfxGetDopplerFactor():** Return the factor applied to doppler effects on 3D sounds. Default is set by “\$pref::SFX::dopplerFactor”. If unset, default is 0.5.
- **sfxSetDopplerFactor( float factor ):** Set the factor to apply to doppler effects on 3D sounds. Set to zero to turn off pitch shifting caused by the Doppler Effect. The higher the value, the more pronounced the Doppler Effect will be.
- **float sfxGetRolloffFactor():** Get the scale factor applied to distance attenuation curves of 3D sounds in the logarithmic distance model. Default is taken from “\$pref::SFX::rolloffFactor”. If unset, default is 1.0, i.e. no scaling.
- **sfxSetRolloffFactor( float factor ):** Set the scale factor to apply to distance attenuation curves of 3D sounds in the logarithmic distance model. Values greater than 1 cause volume to decrease faster with distance where as values less than 1 cause volume to decrease slower. A value of 0 will disable distance attenuation. Factor must be greater or equal to 0.

## Playback

- **SFXSource sfxCreateSource( SFXProfile profile [, float x, float y, float z ] ):** Create a new SFXSource that plays the given profile. If coordinates are given, the source will be placed at the given position (though only if the given profile contains a 3D sound). The source will initially be in “stopped” state. If the sound contained in the profile has not yet been loaded, this will be initiated in the process.
- **SFXSource sfxCreateSource( SFXDescription description, string filename [, float x, float y, float z ] ):** Create a temporary SFXProfile using the given description and filename and then create a new SFXSource that plays the profile. If coordinates are given, the source will be placed at the given position (though only if the description has is3D set to true). The source will initially be in “stopped” state. Loading of the given file will be initiated in the process.
- **SFXSource sfxPlay( SFXSource source ):** Start playing the given SFXSource. This is the same as calling the play() method on the SFXSource directly. Returns source or null on failure.
- **SFXSource sfxPlay( SFXProfile profile [, float x, float y, float z ] ):** This is the same as calling sfxPlayOnce() with the given parameters.
- **SFXSource sfxPlayOnce(SFXProfile profile [, float x, float y, float z ] ):** Create a play-once SFXSource using the given profile and start playing it. The SFXSource will only be valid for as long as playback is running. If coordinates are given, the SFXSource will be placed at the specified position (only if the profile contains a 3D

sound). The SFXSource will return in “playing” state. Returns a handle for the temporary SFXSource or null on failure.

- **SFXSource sfxPlayOnce( SFXDescription description, string filename [ , float x, float y, float z ] ):** Creates a temporary SFXProfile with the given description and filename and then instantiates a new play-once SFXSource playing the profile. The SFXSource will be valid only for as long as playback is running. If coordinates are given, the SFXSource will be placed at the specified position (only if the given SFXDescription has is3D set to true). The SFXSource will return in “playing” state. Returns a handle for the temporary SFXSource or null on failure.
- **sfxStop( SFXSource source ):** Stop playing the given SFXSource. This is the same as calling the stop() method directly on the SFXSource.
- **sfxStopAll():** Call stop() on all SFXSources that are currently playing.

### Misc

- **sfxDumpSources():** Print a detailed rundown on all currently instantiated SFXSources to the console.

## 25.2.7 Conclusion

This interface guide covers everything you will need to know about using Torque 3D’s stock sound effect system (SFX).

## 25.3 Internals

### 25.3.1 SFXBuffer

Objects of this class hold sound data on the sound device. Buffers will be deleted when no longer used but will also be freed when the device is destroyed. Use StrongWeakRefPtrs to permanently refer to SFXBuffers so that references will null out when the buffer goes away.

Loading and updating of sound buffers happens on dedicated SFX update threads created by the individual devices. For devices that do not create such a thread, updating will happen on the main thread; this, however, is currently only used by the Null device.

Buffers hold raw, uncompressed PCM sample data loaded from SFXStreams. SFXBuffers for normal, buffered playback will be loaded once in entirety and may then be used by an arbitrary number of SFXVoices for concurrent playback.

SFXBuffers for streamed playback will be loaded progressively in the background and are tied to a single unique SFXVoice.

**Important:** Do not use delete directly on an SFXBuffer. Instead call **destroySelf()** on the buffer to delete it. Buffers may have pending asynchronous operations that need to flush out first before the buffer can actually be deleted.

### 25.3.2 SFXVoice

This is an abstract base class for objects on the sound device that the control playback of a particular SFXBuffer. Usually, a device will limit the number of concurrent sound playbacks it supports. SFXVoices follow the same lifecycle as SFXBuffers. An SFXVoice should never be directly deleted by clients. Leave lifetime management to reference-counting.

An SFXVoice promoted to playing state when its attached SFXBuffer has not yet fully loaded will temporarily transition into SFXStatusBlocked to indicate that playback can't progress. This will also happen for streamed sounds when the playback cursor is outrunning the stream feed.

### 25.3.3 SFXStream

Abstract base class for reading raw PCM sample streams. Instances of subclasses will be used for all sound loading that goes through SFX's own loading system (in contrast to sound loading happening directly through the sound API in use). Both buffered and streamed sounds are loaded through SFXStreams.

SFXStreams are used concurrently from multiple threads and are concurrently reference-counted to ensure proper and safe reclamation. For an SFXStream to support seeking in combination with streamed playback, it must implement cloning through the clone() method.

Streamed playback that also loops will require the attached SFXStream to properly reset().

### 25.3.4 SFXFileStream

This is the abstract base class for specific file format loaders.

### 25.3.5 SFXResource

Thin wrapper that represents as sound file on disk. Performs a header read on the file to detect SFXFormat characteristics. Does not keep actual sound data.

### 25.3.6 SFXDevice

Abstract base class for SFX implementations against particular sound APIs. Manages SFXVoices and SFXBuffers as the primary sound device resources. The lifetimes of all sound resources are bound to their respective SFXDevice. Only one SFXDevice will ever be instanced at any one point.

### 25.3.7 SFXProvider

Abstract base class for objects that manage device creation on particular sound APIs. There is one SFXProvider per supported sound API. SFXProviders have no responsibilities besides enumerating, querying, and creating SFXDevices

### 25.3.8 SFXSystem

Singleton class that defines the central hub for the sound system. Exposes the high-level interface of the system. Manages SFXSources and the current SFXDevice instance. Exposes global SFX parameters.

SFXSystem::\_update() is called from the main game loop to periodically update the SFXSources in the system. Play-once source that have stopped playing will purged in the process. Sound loading, however, is usually handled on a dedicated thread rather than on the main thread.

### 25.3.9 Conclusion

You typically will not be working directly with the SFX internals, unless you are heavily extending the SFX system. This guide was meant to show you the major internal classes that drive the SFX system. Continue reading to learn about common tips, troubleshooting, and the conclusion of the SFX system documentation.

## 25.4 Conclusion

### 25.4.1 How Do I...

#### ... convert to/from OGG?

For Windows, a neat little tool is oggdropXPd. It converts to and from OGG using various formats and has an interface as straightforward as it gets.

- <http://www.rarewares.org/ogg-oggdropxpd.php>

#### ... see how the system is used. Is there an example I can look at?

A simple example of how to use the system can be found in the form of the GuiMusicPlayer control found in `core/scripts/gui/guiMusicPlayer.cs`.

#### ... create 5.1 or 7.1 surround sound?

This is done by the mixer of the sound device you are using. This mixer will take all the active sound voice (2D and 3D) and mix their audio signals down into a number of output channels. In plain stereo playback, this will be two channels: a left one and a right one.

With surround systems, a corresponding number of output channels are fed by the mixer. Note, however, that in order to get surround effects in your game, you need to properly set up 3D sounds.

#### ... play background music?

One way is to place an SFXEmitter in your level and let it stream a 2D looping music track. Another way is to manually trigger playback directly from script.

#### ... tune stream buffering?

If you experience lag and interruptions with stream sources, set the `streamPacketSize` and `streamReadAhead` properties in `SFXDescriptions` such that more data is buffered in advanced. Be careful with `streamPacketSize` as it directly affects buffer metrics on the device as well as the time spend on each individual I/O operation.

To directly modify queue length on devices, adjust the following in the engine code:

```
SFXInternal::SFXAsyncQueue::DEFAULT_STREAM_QUEUE_LENGTH
```

This will not affect streaming happening directly on the device (FMOD currently).

#### ... stream custom audio data?

One way is to derive a class from `SFXStream` to use it to write the sample data to the audio buffers. `SFXSystem::createSourceFromStream()` can be used here to create an `SFXSource` from just a plain `SFXStream` and an `SFXDescription`.

Another way is to create and feed an `SFXPacketStream` which consumes raw, uncompressed sample data in discrete packets and writes them to the consumer audio buffer as needed. Use the same `SFXSystem::createSourceFromStream()` method to create an `SFXSource` to control playback.

**... add support for a new audio format?**

Derive a class from `SFXFileStream` and register your extension(s) through `SFXFileStream::registerExtension()` in `SFXSystem::init()`. Look at `SFXWavStream` and `SFXOggStream` for examples.

**... monitor the status of the SFX system?**

Use the “sfx” metrics, for example, by typing ‘`metrics( “sfx” );`’ in the console. This displays various live statistics for the SFX system. To see a detailed listing of all current `SFXSources`, use “`sfxDumpSources`”.

**... play multiple sounds in random/sequential order on an SFXEmitter?**

Currently, this is not possible without custom scripting. The next revision of SFX will, however, include playlist support across all devices.

## 25.4.2 Troubleshooting

**I am hearing 3D sounds outside of their set maximum range. How can I fix this?**

You are probably using the logarithmic distance model. This is where sounds are not cut off at their maximum distance, but rather retain the attenuated volume that is greater than the specified max distance. To make sounds fade out in time, use proper settings for the min and max distance. Also, to speed up or slow down overall volume falloff with distance, increase or decrease the 3D rolloff factor.

**My sounds start with a delay. What is wrong?**

The delay is due to the sound taking a certain time to load. Generally, make sure that either sound data is available before starting playback or that a short delay is not relevant for a given sound.

**I’m seeing an error in the console.log that says “SFXFMODProvider - Could not locate the <fmod.dll>”. What is wrong?**

This only means that the FMOD sound provider cannot find the FMOD DLL. If you do not want to use FMOD, then simply ignore this message. Otherwise, make sure to copy the FMOD DLL (`fmodex.dll` on Windows; `fmodex.dylib` on Mac) to the same folder as your game executable. This message should then disappear the next time you start Torque.

To download FMOD, visit the FMOD homepage. To purchase a very friendly priced indie license of FMOD for your game, visit the Torque store.

1. <http://www.fmod.org/>
2. <http://www.garagegames.com/products/fmod>

**I’m seeing crashes with XAudio. What can I do?** Head over to Torque 3D Private forum and post a bug report. Our SFX developer will be around to look into the issue.

1. <http://www.garagegames.com/community/forums/63>

## 25.4.3 Best Practices

**Preload or not?**

Anything that is used often should be preloaded. Rarely used sounds for which it is okay to start with a short delay need not be preloaded. Streamed sounds cannot be preloaded (create an `SFXSource` if you want to ready them before playing).

**Lifetimes of SFXProfiles**



Scope the lifetimes of SFXProfiles to where they are actually used. Generally, don't put all of the sounds of your game together as one batch of SFXProfiles. Once its sound data has been loaded, an SFXProfile for non-streamed sounds will consume resources on the SFX device.

Ambient sounds, for example, that are not used in a particular level need not have their SFXProfiles loaded. However, sounds that are used in any level in the game are better loaded once at startup. For server-side profiles, use the datablock keyword whereas for client-side profiles, use new or singleton.

### **Stream or not?**

Stream music/ambient sounds, don't stream effects. Generally WAVs over ~700kb and OGGs over 200k should be streamed.

## **25.4.4 Conclusion**

This concludes the SFX documentation for Torque 3D. If you wish to learn more about the system, this is a good time to browse the demos provided with Torque 3D and see how they are used. Additionally, the system is very well commented in the engine code and will contain more detailed information on a per-line basis.

## 26.1 Lighting Overview

### 26.1.1 Introduction

The lighting system in Torque3D is set up to support Shader Model 1.0 as a minimum. While the more advanced features will require Shader Model 3.0, this will provide a balance between users with older and newer graphics cards. The lighting system provides for the use of shadow mapping techniques and deferred rendering.

This guide will give you a detailed analysis of the classes that create the functionality of Torque 3D's lighting system while showing you how some of them are used. In addition to the Source Code Tour ([Lighting Source Code Tour](#) - [TODO Internal Link](#)) and Light Manager Usage ([Using the LightManager Class](#) - [TODO Internal Link](#)), there are detailed function definitions for the following classes.

### 26.1.2 Basic Light Manager

The Basic Light Manager (**class BasicLightManager**) requires that the graphics card have at least Shader Model 1.0. It is an override of the abstract class **LightManager** to provide a simple lighting solution to lower end graphics cards. Although it is targeted to lower end graphics cards, it does still support shadowing techniques.

### 26.1.3 Advanced Light Manager

The Advanced Light Manager (**class AdvancedLightManager**) requires that the graphics card support at least Shader Model 3.0. Much like the **BasicLightManager**, it is an override of the **abstract class LightManager**, however, it will provide a more sophisticated lighting solution for higher end graphics cards. The Advanced Light Manager supports multiple adaptations of shadow mapping techniques such as Dual Paraboloid and Cube Shadow maps.

### 26.1.4 The LightInfo Class

This is the base light information class that will be tracked by the engine. It basically contains a bounding volume and methods to interact with the rest of the system (for example, setting the GFX fixed function lights).

### 26.1.5 The LightInfoEx Class

This is the base class for extended lighting info that lies outside of the normal info stored in `LightInfo`. An example of where this class will be used is for creating `LightMapParams` and `ShadowMapParams`.

### 26.1.6 SceneLightingInterface

The `SceneLightingInterface` object is responsible for returning `PersistChunk` and `ObjectProxy` classes for the lighting system to use. The `SceneLightingInterface` is used by the `AvailableSLInterfaces` to register systems to itself, such as “class `blInteriorSystem`”.

### 26.1.7 AvailableSLInterfaces

`AvailableSLInterfaces` makes use of the `SceneLightingInterface` to register lighting interfaces. By default, the `BasicLightManager` will register an instance of the “`blInteriorSystem`” and “`blTerrainSystem`” in its constructor.

### 26.1.8 The ShadowManager Base Class

The `ShadowManager` is a base class to be extended later by a more sophisticated shadow manager, such as the class `ShadowMapManager`. While the `ShadowManager` does not currently contain any pure virtual functions, creating an instance of `ShadowManager` will not manage any shadows.

### 26.1.9 The ShadowMapManager Class

The `ShadowMapManager` is a full extension of the `ShadowManager` where the `SceneManager` will be notified to add the rendering function (“`_onPreRender`”) to its list. The rendering function will then call the render function from its currently set shadow map pass.

## 26.2 Lighting Source Code Tour

### 26.2.1 Introduction

This document will explain the source code folders and files that make up Torque 3D’s lighting system.

### 26.2.2 The Lighting Core

You can find the core Lighting files in `engine/lighting`. Most of the classes in these files will be derivatives of more sophisticated types. For example, the class `LightManager` from `engine/lighting/lightManager.h` is the derivative of `AdvancedLightManager` from `engine/lighting/advanced/advancedLightManager.h`.

#### Key Classes

- **LightInfoEx** : Located in `engine/lighting/lightInfo.h`

This is the base class for extended lighting info that lies outside of the normal info stored in `LightInfo`. \* **LightInfo** : Located in `engine/lighting/lightInfo.h` This is the base light information class that will be tracked by the engine. Should basically contain a bounding volume and methods to interact with the rest of the system. \* **ISceneLight** : Located in `engine/lighting/lightInfo.h` When the scene is queried for lights, the light manager will get this interface to trigger a register light call. \* **AvailableSLInterfaces** : Located in `engine/lighting/lightingInterfaces.h` List of available “systems” that the lighting system can use. \* **LightManager** : Located in `engine/lighting/lightManager.h` This is the base class for extending a lighting manager to support lighting features. \* **ShadowManager** : Located in `engine/lighting/shadowManager.h` This is the base class for extending a shadow manager to support more advanced shadowing techniques such as shadow maps.

### 26.2.3 Basic Lighting

The basic lighting system that is created in Torque3D requires at least Shader Model 1.0. Much like the advanced lighting code, the basic lighting system is an override of already created classes from the lighting core (`engine/lighting`) that resides in `engine/lighting/basic`.

#### Key Classes

- **BasicLightManager** : Located in `engine/lighting/basic/basicLightManager.h`

The `BasicLightManager` is an override of the **LightManager** that will provide a lighting system for low end cards, it only requires shader model 1.0.

- **BasicSceneObjectLightingPlugin** : Located in `engine/lighting/basic/BasicSceneObjectLightingPlugin.h`

A **BasicSceneObjectLightingPlugin** is an override of the **SceneObjectLightingPlugin** that is used by the **BasicLightManager** during its `_onPreRender` function for updating shadow plugins.

### 26.2.4 Advanced Lighting

Torque3D takes advantage of Shader Model 3.0 to allow advanced lighting techniques. The source code in `engine/lighting/advanced` is an override of already created classes from the lighting core (`engine/lighting`) to implement these techniques.

#### Key Classes

- **AdvancedLightingFeatures** : Located in `engine/lighting/advanced/advancedLightingFeatures.h` The **AdvancedLightingFeatures** class provided static methods for registering and unregistering features to the Feature Manager (**FeatureMgr**). When the function `registerFeatures` is called it will features related to lighting based upon the **GFXFormat** passed into the function. These features are defined in “`Engine/source/materials/materialFeatureTypes.h`” and have the prefix “**MFT\_**”.
- **AdvancedLightManager** : Located in `engine/lighting/advanced/advancedLightManager.h` The **AdvancedLightManager** is an override of **LightManager** that provides a setup for the deferred rendering system. **AdvancedLightManager** requires at least shader model 3.0.

### 26.2.5 Common Lighting Classes

Common is a place where classes that do not particularly fit in one designated area, such as advanced or lighting. They are not considered core classes of the lighting system either. In here you will find classes that will be used to assist in implementing features for the lighting system.

#### Key Classes

- **ShadowBase** : Located in `engine/lighting/common/shadowBase.h`

The **ShadowBase** class is an abstract class containing all pure virtual functions that will be overwritten by the shadow technique using them.

- **BlobShadow** : Located in engine/lighting/common/blobShadow.h

A **BlobShadow** is one of the basic ways you may use the **ShadowBase** class. The **BlobShadow** is just a shadow based upon a radius.

- **LightMapParams** : Located in engine/lighting/common/lightMapParams.h

The **LightMapParams** is an override of the **LightInfoEx** from the lighting core. Most “**LightShadowMap**” based classes will use **LightMapParams** in their render function to determine if it should use light mapped geometry. As **LightMapParams** are a super class of **LightInfoEx**, they will be grabbed by LightInfo’s “**getExtended**” function as a parameter type.

## 26.2.6 Shadow Maps

There are a wide range of shadow maps available in Torque3D, such as the Paraboloid Shadow Map. This folder contains a common base class for many shadow maps to override (**LightShadowMap**) and also an override of the **ShadowManager** class from the lighting core folder that is named “**ShadowMapManager**”.

### Key Classes

- **ShadowMapManager** : Located in engine/lighting/shadowMap/shadowMapManager.h

The **ShadowMapManager** is used to render a shadow map pass via its **onPreRender** function that is triggered by a signal of the **SceneGraph**.

- **LightShadowMap** : Located in engine/lighting/shadowMap/lightShadowMap.h

Represents everything Torque 3D needs to render a shadow map for one light.

- **DualParaboloidLightShadowMap** : Located in engine/lighting/shadowMap/dualparaboloidLightShadowMap.h

The **DualParaboloidLightShadowMap** class uses the Dual Paraboloid shadow mapping technique and is an override of the **ParaboloidLightShadowMap** class, which is an override of the **LightShadowMap**.

## 26.3 Using the LightManager Class

### 26.3.1 Introduction

The **LightManager** is one of the most comprehensive classes in the lighting system. **LightManager** is defined as a singleton and provides the user with the ability to register global lights, compute static lighting, register special types of lights and return the active **LightManager**. The active **LightManager** can be accessed at any time via the static function “**getActiveLM**”, or with the **#define** “**LIGHTMGR**”. Even though it has a wealth of functionality, on its own it can not operate and is the base class to **BasicLightManager** and **AdvancedLightManager**. It contains four key functions that need to be overridden for its subclasses to be initialized:

```
bool isCompatible() const
```

Should return true if this light manager is compatible with the current platform and GFX device:

```
void setLightInfo( ProcessedMaterial *pmat, const Material *mat, const SceneGraphData_
↳ &sgData,
const SceneState *state, U32 pass, GFXShaderConstBuffer *shaderConsts )
```

Sets shader constants / textures for LightInfo’s:

```
void setTextureStage( const SceneGraphData &sgData, const U32 currTexFlag, const U32 ↵
↳textureSlot,
GFXShaderConstBuffer *shaderConsts, ShaderConstHandles *handles )
```

Allows the ability to set textures during the `Material::setTextureStage` call, return true if it has done work:

```
void _addLightInfoEx( LightInfo *lightInfo )
```

Attaches any `LightInfoEx` data for the light manager to the light info object.

### 26.3.2 Example Uses

The `Sun` class will need to have a `LightInfo` object created for it. As a result, in its constructor it will create a `LightInfo` object by calling the static function `createLightInfo(LightInfo *)` to allocate a new `lightInfo` object for its use. Inside of the `LightManager::createLightInfo(LightInfo *)` function the allocated `LightInfo` will then be added to all of the `LightManager`'s currently allocated light objects by traversing the `LightManagerMap` returned by the member function `_getLightManagers()`. Finally, the function will return the pointer to the allocated `LightInfo` object that is created.

After the `Sun` class creates the light info via "`LightManager::createLightInfo()`" the last thing it will do is set the type of light that it is (with `LightInfo::setType()`). It sets the type to "`LightInfo::Vector`", or commonly referred to as "directional".

Other examples of classes using `createLightInfo` are the following. If the type of light is not specified after the `LightInfo` is created, it is "vector" by default.

- class `Projectile`

Sets the type to `LightInfo::Point`

- class `Explosion`

Does not set a `LightInfo` after it is created in the constructor, so by default it uses `Point::Vector`. However, later in the "`Explosion::onAdd`" function it will set the type to a `LightInfo::Point` if the `Explosion`'s internal information calls for it.

- class `ScatterSky`

Will set the type to `LightInfo::Vector`

### 26.3.3 Compatibility

As mentioned in the section above, the `LightManager` contains the pure virtual function "`bool isCompatible()`", which is basically self explanatory. If the subclass detects that the graphics card does not meet the requirements that it needs, it will return false. This compatibility function will be called when a light manager is assigned to the `SceneGraph`. When the `isCompatible` function is called via "`bool SceneGraph::_setLightManager`" and it detects that the function did not return true, it will not assign the `LightManager` to the `SceneGraph`.

### 26.3.4 Activating and Deactivating the LightManager

The `LightManager` will provide the function "activate" that can be overridden by its subclasses, however, the subclasses should call the `LightManager::activate` function from any override that is written. The activate function will ensure that the `SceneManager` passed in is valid, the light manager currently is not active and that no other light manager is active.

### 26.3.5 Using the Active LightManager

The active LightManager is available throughout the code base via the static function “LightManager \*getActiveLM()”, or the most commonly used #define “LIGHTMGR”. The LIGHTMGR definition is used throughout the code base to utilize the fact that the LightManager is defined as a singleton.

#### Example Uses:

```
WaterBlock::setupVertexBuffer(U32, U32, U32)
```

WaterBlock uses LIGHTMGR to retrieve the Sun light and obtain the direction it is pointing at:

```
CloudLayer::renderObject(ObjectRenderInst *, SceneState *, BaseMatInstance *)
```

CloudLayer uses LIGHTMGR during rendering to help obtain the ambient color of the sun:

```
GuiObjectView::renderWorld(const RectI&)
```

GuiObjectWorld uses LIGHTMGR to unregister all of the lights via “LIGHTMGR->unregisterAllLights()”.

### 26.3.6 AdvancedLightManager

The AdvancedLightManager is a singleton, meaning that there is always an instance available to grab and that the class is created at the runtimes initialization. In addition to overriding the pure virtual functions from its base class, LightManager, it will override the activate and deactivate functions. It will also override the way global lights are registered and unregistered. As previously mentioned in the Overview section, the AdvancedLightManager requires the graphics card to support shader model 3.0. Additional functionality over the base LightManager is the ability to look for a LightShadowMap per SimObject (if the object can cast to an ISceneLight properly).

### 26.3.7 BasicLightManager

The BasicLightManager at its base is like the AdvancedLightManager, meaning that it is a singleton, that there is always an instance available, and that the object is created at the runtime’s initialization. In addition to overriding the pure virtual functions from its base class, LightManager, it will override the activate and deactivate functions. It will also create the static method “getShadowFilterDistance” to help filter out shadows. For an example of this, take a look at the function “ProjectedShadow::\_renderToTexture”.

## 26.4 LightManager

#### LightManager Class Reference

**lightScene** (const char \*, const char \*)

Will generate static lighting (aka lightmaps) for the scene if supported by the active light manager. If mode is “forceAlways”, the lightmaps will be regenerated regardless of whether lighting cache files can be written to. If mode is “forceWritable”, then the lightmaps will be regenerated only if the lighting cache files can be written.

Syntax:

```
lightScene(const char* callback, const char* param )
```

#### Parameters

- **callback** – The name of the function to execute when the lighting is complete.



- **param** – Either “forceAlways” or “forceWritable”.

**Returns** **bool** Returns true if the scene lighting process was started.

Examples:

```
// Get the sunlight.
LightInfo *sunLight = mLightManager->getSpecialLight(
↳LightManager::slSunLightType );
```

#### **getSceneLightingInterface()**

Will return the AvailableSLInterfaces for the LightManager, if there is no available lighting system then it will create a new AvailableSLInterfaces and then return that.

Syntax:

```
getSceneLightingInterface()
```

**Returns** **LightInfo \*** The **LightInfo \*** of the special light.

Examples:

```
// From PersistInfo::read
SceneLightingInterfaces sli = LIGHTMGR->getSceneLightingInterface()->
mAvailableSystemInterfaces;
```

#### **\_update4LightConsts(const SceneGraphData &sgData, GFXShaderConstHandle \*, GFXShaderConstHandle \*, GFXShaderConstHandle \*, GFXShaderConstHandle \*, GFXShaderConstHandle \*, GFXShaderConstBuffer \*)**

Will update the shader constants of **GFXShaderConstBuffer \*** depending on the validity of the **GFXShaderConstHandle \***'s passed in.

Syntax:

```
update4LightConsts(const SceneGraphData &sgData, GFXShaderConstHandle
↳*lightPositionSC, GFXShaderConstHandle *lightDiffuseSC, GFXShaderConstHandle
↳*lightAmbientSC, GFXShaderConstHandle *lightInvRadiusSqSC, GFXShaderConstHandle
↳*lightSpotDirSC, GFXShaderConstHandle **lightSpotAngleSC, GFXShaderConstBuffer
↳*shaderConsts)
```

#### **Parameters**

- **sgData** – The scene graph data related to how the lights will be used.
- **lightPositionSC** – The shader constant for the position paramter.
- **lightDiffuseSC** – The shader constant for the diffuse paramter.
- **lightAmbientSC** – The shader constant for the ambient paramter.
- **lightInvRadiusSqSC** – The shader constant for the light inverse radius paramter.
- **lightSpotDirSC** – The shader constant for the spot light direction paramter.
- **lightSpotAngleSC** – The shader constant for the spot light angle paramter.
- **shaderConsts** – The shader consts for the buffer.

**Returns** no return value.

Examples:

```
// From AdvancedLightManager::setLightInfo
// Update the forward shading light constants.
_update4LightConsts( sgData,
    lsc->mLightPositionSC,
    lsc->mLightDiffuseSC,
    lsc->mLightAmbientSC,
    lsc->mLightInvRadiusSqSC,
    lsc->mLightSpotDirSC,
    lsc->mLightSpotAngleSC,
    shaderConsts );
```

**getAllUnsortedLights** (Vector \*)

Will append all of the registered lights to the “Vector **\***” variable passed in.

Syntax:

```
getAllUnsortedLights (Vector *list)
```

**Parameters** **list** – A vector of “LightInfo” pointers that will have the registered lights added to it.

**Returns** no return value.

Examples:

```
// From SceneLighting::light
LIGHTMGR->getAllUnsortedLights (&mLights);
```

**unregisterAllLights** ()

Will clear out all of the special lights and registered lights.

Syntax:

```
unregisterAllLights ()
```

**Returns** no return value.

Examples:

```
// Unregister all the lights in the light manager.
LIGHTMGR->unregisterAllLights ();
```

**unregisterLocalLight** (LightInfo \*)

Empty function.

Syntax:

```
Not used.
```

**Returns** no return value.

Examples:

```
None.
```

**registerLocalLight** (LightInfo \*)

Empty function.

Syntax:

```
Not used.
```

**Returns** no return value.

Examples:

```
None.
```

#### **unregisterGlobalLight** (LightInfo \*)

Will remove the passed in light from the registered lights. If the light passed in is the sun, then it will clear the suns special light also.

Syntax:

```
unregisterGlobalLight (LightInfo *light)
```

**Parameters** **light** – The light to be registered from mRegisteredLights.

**Returns** no return value.

Examples:

```
lightManager->unregisterGlobalLight( mLight );
```

#### **registerGlobalLight** (LightInfo \*, SimObject \*)

If the light is not already registered, then the light will be added to the registered lights. If it already added, a AssertFatal will be thrown.

Syntax:

```
registerGlobalLight (LightInfo *light, SimObject *obj )
```

**Parameters**

- **light** – The light to be registered to mRegisteredLights.
- **obj** – Not used.

**Returns** no return value.

Examples:

```
// From inside of Item::registerLights
lightManager->registerGlobalLight( mLight, this );
```

#### **registerGlobalLights** (const Frustum \*, bool)

Register the lights depending if there is a Frustum or if there is static lighting. If there is no frustum or there is static lighting, then there will be no light culling. If there is a frustum or there is no static lighting, then cull the lights using the frustum.

After the decision for light culling or not, it will have the lights register themselves.

Syntax:

```
registerGlobalLights(const Frustum *frustum, bool staticLighting )
```

**Parameters**

- **frustum** – The frustum to be used for light culling.
- **staticLighting** – Whether or not static lighting is being processed.

**Returns** no return value.

Examples:

```
// Get the lights for rendering the scene.
LIGHTMGR->registerGlobalLights( &sceneState->getFrustum(), false);
```

**setSpecialLight** (*LightManager::SpecialLightTypesEnum*, *LightInfo* \*)

Register the light with the LightManager and set the light to one of the special light types for the LightManager.

Syntax:

```
setSpecialLight (LightManager::SpecialLightTypesEnum type, LightInfo *light )
```

**Parameters**

- **type** – The special light type.
- **light** – The light to apply the type to.

**Returns** no return value.

Examples:

```
// Set the sunlight.
mLightManager->setSpecialLight( LightManager::slSunLightType, theSun );
```

**getSpecialLight** (*LightManager::SpecialLightTypesEnum*, bool)

Will return the special light based upon the type passed in if it is available. If it is not available, and the useDefault is set to true then it will return the result of getDefaultLight(). In the event that the special light is not found and useDefault is false, the function will return false.

Syntax:

```
getSpecialLight (LightManager::SpecialLightTypesEnum type, bool useDefault )
```

**Parameters**

- **type** – The special light type to find.
- **useDefault** – If the special light based up the light is not found, it will return the default light from getDefaultLight(). useDefault is set to true by default.

**Returns** *LightInfo* \* The *LightInfo* \* of the special light.

Examples:

```
// Get the sunlight.
LightInfo *sunLight = mLightManager->getSpecialLight(
↳LightManager::slSunLightType );
```

**getDefaultLight** ()

Will return the sun if it is registered, as it is always the default light. However, if the sun has not been registered yet then it will create a dummy special light.

Syntax:

```
getDefaultLight()
```

**Returns** `LightInfo *` The `LightInfo *` of the special light.

Examples:

```
// Get the default light, which will be the sun if it is registered
mLightManager->getDefaultLight();
```

### **deactivate()**

Will check to see if the `LightManager` has been already been deactivated and that it is the active light manager. If it passes those two checks, it will call its deactivate callback and then unregister all of the lights associated with `LightManager` (via `unregisterAllLights()`, detailed below).

Syntax:

```
deactivate()
```

**Returns** No return value.

Examples:

```
// Deactivate the light manager
mLightManager->deactivate();
```

### **activate(SceneGraph \*)**

Will activate the `LightManager` if it is not already activate (in which case an `AssertFatal` will be thrown) and call the callback for activating the light manager.

Syntax:

```
activate(SceneGraph *sceneManager);
```

**Parameters** **sceneManager** – The `SceneGraph` that will be used to activate rendering passes and the post effect fog if there is a prepass.

**Returns** No return value.

Examples:

```
// From inside of "SceneGraph::_setLightManager( LightManager *lm )" located in
// sceneGraph.cpp
mLightManager->activate( this );
```

### **initLightFields()**

A static function that will traverse the `LightManagerMap` and will call each `LightManager`'s `_initLightFields` function. Since `_initLightFields` is a pure virtual function, it will call the derived classes `_initLightFields`.

Syntax:

```
initLightFields()
```

**Returns** No return value.

Examples:

```
LightManager::initLightFields();
```

**getLightManagerNames** (String \*)

A static function that will create a String containing all of the LightManager names from the LightManagerMap.

Syntax:

```
getLightManagerNames (String *outString);
```

**Parameters** **outString** – The string that will be constructed based upon the available LightManager names from the LightManagerMap.

**Returns** No return value.

Examples:

```
If the LightManagerMap had two LightManager's inside of it, one being
"Advanced Lighting" and the other "Basic Lighting" then outString will be
"Advanced Lighting  Basic Lighting". The spacing between "Advanced Lighting"
and "Basic Lighting" is a tab ("\t").
```

**findByName** (const char \*)

A static function that traverses the LightManagerMap for the LightManager with the name passed in. If no LightManager in the LightManagerMap contains that name, it will return NULL.

Syntax:

```
findByName (const char *name);
```

**Parameters** **name** – The name of the LightManager to find.

**Returns** LightManager\* The LightManager found in the LightManagerMap with the name passed in.

Examples:

```
// Find the LightManager with the name "Advanced Lighting"
LightManager::findByName( "Advanced Lighting" )
```

**\_getLightManagers** ()

Returns the static LightManagerMap.

Syntax:

```
_getLightManagers ()
```

**Returns** LightManagerMap

Examples:

```
// Retrieve the LightManagerMap, example is from LightManagerMap::findByName
LightManagerMap &lightManagers = _getLightManagers();
```

**~LightManager()**

Will safely delete the default light and Available Scene Lighting Interfaces. It will also remove its self from the LightManagerMap.

Syntax:

```
~LightManager()
```

**Returns** No return value.

Examples:

```
This function is called implicitly at the destruction of LightManager.
```

**LightManager(const char \*, const char \*)**

Initializes the class variables to the default values.

Syntax:

```
LightManager(const char *name, const char * id);
```

**Parameters**

- **name** – The name of the LightManager.
- **id** – The id of the LightManager, often an abbreviation of the name.

**Returns** No return value.

Examples:

```
// The LightManager being given the name "Advanced Lighting" and the id of "ADVLM"
LightManager( "Advanced Lighting", "ADVLM" )
```

## 26.5 BasicLightManager

### BasicLightManager Class Reference

**isCompatible()**

Checks to make sure that the graphics card is compatible with the current pixel shader version that is needed. Currently at least 1.0 is needed.

Syntax:

```
isCompatible()
```

**Returns** no return value.

Examples:

```
// Make sure its valid... else fail!
if ( !lm->isCompatible() )
return false;
```



**activate** (SceneGraph \*)

In addition to calling its base classes `active(SceneGraph*)`, it will also set the Scenegraph to enable post effect fog and tell the material manager not to use prepass.

Syntax:

```
activate( SceneGraph *sceneManager); // SceneGraph* The SceneGraph to activate.
```

**Returns** SceneGraph\* The activated SceneGraph object.

**deactivate** ()

Will remove all the objects from the AdvancedLightBinManager and the PrePassRenderBin, then set them to NULL. It will deactivate the Shadow Manager, unregister all the advanced lighting features and then finally send a trigger to let everyone know the LightManager has been deactivated.

Syntax:

```
deactivate()
```

**Returns** no return value.

Examples:

```
if (mLightManager)
    mLightManager->deactivate();
```

**setLightInfo** (ProcessedMaterial \*, **const** Material \*, **const** SceneGraphData&, **const** SceneState \*, U32, GFXShaderConstBuffer \*)

Will make sure that the current lighting constants are initialized, then it will update the lighting constants via `_update4LightConsts::_update4LightConsts`.

Syntax:

```
setLightInfo(ProcessedMaterial *pmat, Material *mat, const SceneGraphData &sgData,
↳ SceneState *state, U32 pass, GFXShaderConstBuffer *shaderConsts )
```

**Parameters**

- **pmat** – Not used.
- **mat** – Not used.
- **sgData** – Used in the call for `_update4LightConsts`. See `LightManager::_update4LightConsts`.
- **state** – Not used.
- **pass** – Not used.
- **shaderConsts** – Is used to check to see if it is the same as the last shader. This check is done because T3D sorts by material and the light manager should get hit repeatedly by the same shader. The advantage of the check is that it gives optimization that will prevent has table look ups. It is also used in the call for `_update4LightConsts`. See `LightManager::_update4LightConsts`.

**Returns** no return value.

Examples:

```
// From inside of ProcessedShaderMaterial::setSceneInfo
LIGHTMGR->setLightInfo( this, mMaterial, sgData, state, pass, shaderConsts );
```

**setTextureStage** (const SceneGraphData &sgData, const U32 currTexFlag, const U32 textureSlot, GFXShaderConstBuffer \*shaderConsts, ShaderConstHandles \*handles)  
Will always return false.

Syntax:

```
setTextureStage(const SceneGraphData &sgData, const U32 currTexFlag, const U32 textureSlot,
↳GFXShaderConstBuffer *shaderConsts, ShaderConstHandles *handles )
```

#### Parameters

- **sgData** – Not used.
- **currTexFlag** – Not used.
- **textureSlot** – Not used
- **shaderConsts** – Not used.
- **handles** – Not used.

**Returns** no return value.

Examples:

```
// From inside of ProcessedCustomMaterial::setTextureStages
lm->setTextureStage(sgData, currTexFlag, i, shaderConsts, handles )
```

## 26.6 AdvancedLightManager

### AdvancedLightManager Class Reference

#### getLightBinManager()

Will return the lightBinManager for this light manager.

Syntax:

```
getLightBinManager();
```

**Returns** AdvancedLightBinManager \* The lightBinManager member variable in Advanced-LightManager.

Examples:

```
lightmgr->getLightBinManager()
```

#### isCompatible()

Checks to make sure that the graphics card is compatible with the current pixel shader version that is needed. Currently at least 3.0 is needed.

Syntax:

```
isCompatible();
```

**Returns** No return value.

Examples:

```
// Make sure its valid... else fail!
if ( !lm->isCompatible() )
    return false;
```

### **activate** (SceneGraph \*)

In addition to calling its base classes active(SceneGraph\*), it will activate the Shadow Manager and create the AdvancedLightBinManager. It will also setup the Render Prepass Manager and register the feature as an AdvancedLightingFeature.

Syntax:

```
activate( SceneGraph *sceneManager);
```

**Parameters** **SceneGraph\*** – The SceneGraph to activate lighting for.

**Returns** SceneGraph \* The SceneGraph that will be used to create lighting features.

Examples:

```
// From the engine function "resetLightManager"
LIGHTMGR->activate( LIGHTMGR->getSceneManager());
```

### **deactivate** ()

Will remove all the objects from the AdvancedLightBinManager and the PrePassRenderBin, then set them to NULL. It will deactivate the Shadow Manager, unregister all the advanced lighting features and then finally send a trigger to let everyone know the LightManager has been deactivated.

Syntax:

```
deactivate()
```

**Returns** No return value.

Examples:

```
if (mLightManager)
    mLightManager->deactivate();
```

### **registerGlobalLight** (LightInfo \*, SimObject \*)

In addition to calling LightManager::registerGlobalLight, it will add the light to the AdvancedLightBinManager member variable if the AdvancedLightBinManager is created and the light type is a LightInfo::Point or LightInfo::Spot.

Syntax:

```
registerGlobalLight(LightInfo *light, SimObject *obj )
```

**Parameters**

- **type** – The light to be registered to mRegisteredLights.
- **light** – Not used.

**Returns** No return value.

Examples:

```
// From inside of Item::registerLights
lightManager->registerGlobalLight( mLight, this );
```

### unregisterAllLights()

In addition to calling LightManager::unregisterAllLights, it will clear the AdvancedLightBinManager if it has been created.

Syntax:

```
unregisterAllLights()
```

**Returns** No return value.

Examples:

```
// Unregister all the lights in the light manager.
LIGHTMGR->unregisterAllLights();
```

### setLightInfo(ProcessedMaterial \*, const Material \*, const SceneGraphData&, const SceneState \*, U32, GFXShaderConstBuffer \*)

Will check to make sure that the SceneGraphData is not PrePassBin, if it is then it will return out immediately. If it is not, then it will update the constants for the GFXShaderConstBuffer passed in.

Syntax:

```
setLightInfo(ProcessedMaterial *pmat, Material *mat, const SceneGraphData &sgData,
→ SceneState *state, U32 pass, GFXShaderConstBuffer *shaderConsts )
```

#### Parameters

- **pmat** – Not used.
- **mat** – Not used.
- **sgData** – Will be used to ensure rendering is not being done from the PrePassBin and also to update light constants by a call to `_update4LightConsts(...)`.
- **state** – While setting information to “shaderConsts” it will be used to obtain the camera’s transform.
- **pass** – Not used.
- **shaderConsts** – Will be used to obtain the LightingShaderConstants and to the call to `_update4LightConsts(...)`. It will also have its “set” function called to set the shader constant for “ViewToLightProj”.

**Returns** No return value.

Examples:

```
// From inside of ProcessedShaderMaterial::setSceneInfo
LIGHTMGR->setLightInfo( this, mMaterial, sgData, state, pass, shaderConsts );
```

**setTextureStage** (const SceneGraphData &sgData, const U32 currTexFlag, const U32 textureSlot, GFXShaderConstBuffer \*shaderConsts, ShaderConstHandles \*handles)

Will assign a Shadowmap if it exists. It will grab the ShadowMap via the LightingShaderConstants obtained via the shaderConsts passed in.

Syntax:

```
setTextureStage(const SceneGraphData &sgData, const U32 currTexFlag, const U32 textureSlot, GFXShaderConstBuffer *shaderConsts, ShaderConstHandles *handles )
```

#### Parameters

- **sgData** – Used to obtain ShadowMapParams.
- **currTexFlag** – Depending on the currTexFlag the texture will be set differently to the GFXDevice.
- **textureSlot** – Not Used.
- **shaderConsts** – Used to obtain the LightingShaderConstants via getLightingShaderConstants(...).
- **handles** – Not used.

**Returns** No return value.

Examples:

```
// From inside of ProcessedCustomMaterial::setTextureStages
lm->setTextureStage(sgData, currTexFlag, i, shaderConsts, handles )
```

**getSphereMesh** (U32&, GFXPrimitiveBuffer \*&)

Will return a vertex buffer handled filled out by a SphereMesh (mSphereGeometry), along with set the variables passed in. If the SphereMesh (mSphereGeometry) is not created by the time this function is called, it will create the sphere mesh (mSphereGeometry) in addition to returning it.

Syntax:

```
getSphereMesh(U32 outNumPrimitives, GFXPrimitiveBuffer *&outPrimitives )
```

#### Parameters

- **outNumPrimitives** – Will be set to the number of polygons for the SphereMesh.
- **outPrimitives** – Will always be set to NULL.

**Returns** GFXVertexBufferHandle<AdvancedLightManager::LightVertex> Used for the vertex buffer, typically for a LightBinEntry.

Examples:

```
// From inside AdvancedLightBinManager::addLight
AdvancedLightBinEntry::LightBinEntry lEntry.vertBuffer = mLightManager->
    getSphereMesh( lEntry.numPrims, lEntry.primBuffer );
```

**getConeMesh** (U32&, GFXPrimitiveBuffer \*&)

Will return a vertex buffer handled filled out by information for a cone, along with set the variables passed in. If the cone geometry (mConeGeometry) is not created by the time this function is called, it will create the cone geometry (mConeGeometry) in addition to returning it.

Syntax:

```
getConeMesh(U32 outNumPrimitives, GFXPrimitiveBuffer *%outPrimitives )
```

#### Parameters

- **outNumPrimitives** – Will be set to the number of polygons for the SphereMesh.
- **outPrimitives** – Will always be set to NULL.

**Returns** GFXVertexBufferHandle<AdvancedLightManager::LightVertex> Used for the vertex buffer, typically for a LightBinEntry.

Examples:

```
// From inside AdvancedLightBinManager::addLight
AdvancedLightBinEntry::LightBinEntry lEntry.vertBuffer = mLightManager->
    getConeMesh( lEntry.numPrims, lEntry.primBuffer );
```

#### findShadowMapForObject (SimObject \*)

Will take in a SimObject\*, then cast it to a ISceneLight\*. If the converted variable is valid (meaning you passed in a valid ISceneLight), then it would get the shadow map available for the light.

Syntax:

```
findShadowMapForObject (SimObject *object)
```

**Parameters** **object** – The SimObject to be converted to a ISceneLight\* to find the ShadowMap.

**Returns** LightShadowMap \* The found shadow map from the ISceneLight casted “object” variable.

Examples:

```
LightShadowMap *lightShadowMap = lm->findShadowMapForObject( object );
```

## 26.7 ShadowManager

### ShadowManager Class Reference

#### activate ()

Called when the shadow manager should become active. It will assign the class variable “SceneGraph\* mSceneManager” to the global “SceneGraph\* gClientSceneGraph” variable.

Syntax:

```
activate()
```

**Returns** no return value.

Examples:

```
// Called from within the AdvancedLightManager::activate, SHADOWMGR is a #define
// of the ShadowMapManager::instance() function.
SHADOWMGR->activate();
```

**deactivate()**

Called when we don't want the shadow manager active (should clean up). As this is basically just a base class, this function does nothing and should be overridden by the super class to clean up its own data.

Syntax:

```
deactivate()
```

**Returns** no return value.

Examples:

```
// Called from within the AdvancedLightManager::deactivate, SHADOWMGR is a #define
// of the ShadowMapManager::instance() function.
SHADOWMGR->deactivate();
```

**canActivate()**

Called to find out if it is valid to activate this shadow system. Currently this function will always return true.

Syntax:

```
canActivate()
```

**Returns** Will always return true (**bool**).

Examples:

```
Currently this function is not called in the base and is left up to the user to
↳ implement.
```

## 26.8 ShadowMapManager

### ShadowMapManager Class Reference

**activate()**

Called when the ShadowMapManager should become active. If the ShadowMapManager is unable to retrieve a SceneManager through the internal method getSceneManager then it will return a console error saying it was unable to activate the ShadowMapManager and return out of the function. If it is able to retrieve the SceneManager then it will notify the SceneManager of its onPreRender function and turn its self active.

Syntax:

```
activate()
```

**Returns** no return value.

Examples:



```
// Called from within the AdvancedLightManager::activate, SHADOWMGR is a #define
// of the ShadowMapManager::instance() function.
SHADOWMGR->activate();
```

**deactivate()**

Called when the ShadowMapManager should be deactivated. The ShadowMapManager will notify the SceneManager to remove its onPreRender function from the pre-render signal, clean up the shadow texture memory and makes its self no longer active.

Syntax:

```
deactivate();
```

**Returns** no return value.

Examples:

```
// Called from within the AdvancedLightManager::deactivate, SHADOWMGR is a #define
// of the ShadowMapManager::instance() function.
SHADOWMGR->deactivate();
```

**setLightShadowMapForLight** (LightInfo \*)

Looks up the shadow map for the light then sets it.

Syntax:

```
setLightShadowMapForLight(LightInfo *light )
```

**Parameters** **light** – Will use this variable to look up the potential ShadowMapParams.

**Returns** no return value.

Examples:

```
// Set light holds the active shadow map.
mShadowManager->setLightShadowMapForLight( sunLight );
```

**setLightShadowMap** (LightShadowMap \*)

Sets the current shadowmap (used in setLightInfo/setTextureStage calls).

Syntax:

```
setLightShadowMap(LightShadowMap *lm )
```

**Parameters** **lm** – The current shadow map that will be assigned to the classes internal “LightShadowMap \*mCurrentShadowMap” variable.

**Returns** no return value.

Examples:

```
// While traversing the shadow maps in "ShadowMapPass::render".
LightShadowMap *lsm = shadowMaps[i];
mShadowManager->setLightShadowMap( lsm );
```

## 26.9 LightInfo

### LightInfo Class Reference

#### **unpackExtended** (BitStream \*)

Will traverse the vector of LightInfoEx pointers (mExtended) and call their individual unpackUpdate function with the passed in stream as a parameter.

Syntax:

```
unpackExtended( BitStream *stream )isCompatible()
```

**Parameters** **stream** – Used to pass to the individual calls of the unpackUpdate through the traversed mExtended.

**Returns** No return value.

Examples:

```
// From LightBase::unpackUpdate  
mLight->unpackExtended( stream );
```

#### **packExtended** (BitStream \*)

Will traverse the vector of LightInfoEx pointers (mExtended) and call their individual packUpdate function with the passed in stream as a parameter.

Syntax:

```
packExtended( BitStream *stream )
```

**Parameters** **stream** – Used to pass to the individual calls of the packUpdate through the traversed mExtended.

**Returns** No return value.

Examples:

```
// From LightNase::packUpdate at lightBase.cpp  
mLight->packExtended( stream );
```

#### **packExtended** (BitStream \*)

Will traverse the vector of LightInfoEx pointers (mExtended) and call their individual packUpdate function with the passed in stream as a parameter.

Syntax:

```
packExtended( BitStream *stream )
```

**Parameters** **stream** – Used to pass to the individual calls of the packUpdate through the traversed mExtended.

**Returns** No return value.

Examples:

```
// From LightNase::packUpdate at lightBase.cpp  
mLight->packExtended( stream );
```

**getWorldToLightProj** (MatrixF \*)

Builds the world to light view projected used for shadow texture and cookie lookups.

Syntax:

```
getWorldToLightProj( MatrixF *outMatrix )
```

**Parameters** **outMatrix** – The generated Matrix, if the type of the light is a spot then this matrix will be created by multiplying the cone projection by the inverse transform of the light. If the type of the light is not a Spot, then the Matrix will be assigned to the inverse of the light.

**Returns** No return value.

Examples:

```
// From AdvancedLightManager::setLightInfo
MatrixF proj;
light->getWorldToLightProj( &proj );
```

**deleteAllLightInfoEx** ()

Deletes all LightInfoEx objects.

Syntax:

```
deleteAllLightInfoEx()
```

**Returns** No return value.

Examples:

```
// From the destructor of LightInfo
deleteAllLightInfoEx();
```

**addExtended** (LightInfoEx \*)

Will add the passed in LightInfoEx \* to the current vector of LightInfo \* (mExtended) if it is not null.

Syntax:

```
addExtended( LightInfoEx *lightInfoEx )
```

**Parameters** **lightInfoEx** – Will be added to mExtended.

**Returns** No return value.

Examples:

```
// AdvancedLightManager::_addLightInfoEx from advancedLightManager.cpp
lightInfo->addExtended( new ShadowMapParams( lightInfo ) );
```

**getExtended** (const LightInfoExType&)

Will return the LightInfoEx \* mExtended based upon the LightInfoExType passed in.

Syntax:

```
getExtended( const LightInfoExType &type )
```

**Parameters** **type** – The type of a LightInfoEx to be used to return the LightInfoExType.

**Returns** `LightInfoEx *` The `LightInfoEx *` from `mExtended` with regards to the type passed in.

Examples:

```
// From CubeLightShadowMap::setShaderParameters at cubeLightShadowMap.cpp
ShadowMapParams *p = mLight->getExtended<ShadowMapParams>();
```

**setGFXLight** (`GFXLightInfo *`)

Sets a fixed function `GFXLight` with the properties on this class.

Syntax:

```
setGFXLight (GFXLightInfo *light )
```

**Parameters** `light` – The light that will have its properties filled in based upon the properties of this class.

**Returns** No return value.

Examples:

```
// From ProcessedFFMaterial::_setPrimaryLightInfo at porcessedFFMaterial.cpp
GFXLightInfo xlatedLight;
light->setGFXLight (&xlatedLight);
```

**set** (`const LightInfo *`)

Copies the data passed in from the `LightInfo` passed in, such as the properties and the contents of `mExtended`.

Syntax:

```
set (const LightInfo *light )
```

**Parameters** `light` – The light in which the information to be set to this class should be obtained from.

**Returns** No return value.

Examples:

```
None.
```

## 27.1 Overview

### 27.1.1 Introduction

Torque 3D's rendering system is a complex set of modules working together to deliver a next-gen appearance. In addition to many stand-alone rendering systems, such as managers and render instances, a core system is GFX. GFX is an abstract graphics layer designed to reside above graphics APIs such as Direct3D and OpenGL. The system utilizes current and next-gen concepts, such as deferred rendering technology, state blocks, shader buffers, and so on.

### 27.1.2 High Level Features

Torque 3D's rendering features advanced technology, similar to what you might find in DirectX

- GLSL and HLSL shader support
- Vertex and primitive buffers
- Post-processing effects
- Deferred lighting
- Compatibility with Windows, Mac OS X
- State blocks
- Shader constant buffers

### 27.1.3 Platform Support

GFX wraps around multiple rendering systems, which are handled automatically for you. When working in the engine, you will want to focus on the source code related to your target platform:

- **Windows** - GFX->D3D9 for rendering while input, window management, and general Windows components are handled by platformWin32, windowManager->win32, and a few other filters which will be detailed in the GFX Engine Tour. **Source Code Tour - TODO //Add Internal Link**
- **Mac OSX** - GFX->gl for rendering while platformMac and windowManager->mac handle input, window management, and other Mac components.

### 27.1.4 Key Concepts

In order to grasp the high level rendering concepts of Torque 3D, you should be familiar with the following:

- **SDK** - A **Software Development Kit** is typically a collection of tools and APIs that focus on developing applications for a particular framework/hardware/OS. An example would be the DirectX SDK, which includes all of the APIs and debugging tools used in developing Windows games.
- **API** - An **Application Programming Interface** is a collection of functions, classes, and systems dedicated to supporting a specific feature. An example would be DirectX's Direct3D, which is used for rendering.
- **DirectX** (<http://msdn.microsoft.com/en-us/directx/default.aspx>) - A collection of APIs which handle rendering, input, audio, and other forms of media interaction. DirectX is commonly what drives game and video programming on Microsoft's operating systems. Examples of DirectX APIs include Direct3D (D3D) and DirectInput. DirectX developers must use the DirectX SDK to develop a Windows game. The SDK contains all of the DirectX APIs including the runtime libraries and source headers.
- **OpenGL** (<http://www.opengl.org/>) - OpenGL stands for Open Graphics Library. This powerful, cross-platform API is used for low-level rendering of 2D and 3D graphics. Supporting OpenGL in Torque 3D is what allows the engine to run on Mac OSX.
- **Shaders** - Shaders are part of the DirectX and OpenGL rendering systems. A shader file contains a set of instructions that get passed to the GPU along with the 3D data it will affect. Both OpenGL and DirectX have their own shader languages, both of which are handled by Torque 3D's GFX. Examples of shaders include motion blur, reflection, bloom, bump mapping, and other advanced rendering effects. Shaders are typically written in a high level shading language based on C programming.
- **GLSL** - OpenGL Shader Language is a high level shading language utilized by OpenGL to create and render shaders in a game.
- **HLSL** (<http://msdn.microsoft.com/en-us/library/bb509561%28VS.85%29.aspx>) - High Level Shader Language is used by DirectX for creating and displaying shaders in a game. Using HLSL, you can create C like programmable shaders for the Direct3D pipeline.
- **Textures and Materials** - A texture is typically an image file mapped to a polygon or shape, which provides color and detail to the model. Torque 3D materials are used to wrap texture and shader information into a single object.
- **Vertex Buffer** - A vertex buffer is an array of vertex data which can exist in system or video memory.

### 27.1.5 Important Links

1. Torque 3D Documentation Page - <http://www.garagegames.com/documentation/torque-3d>
2. OpenGL Home Page - <http://www.opengl.org/>
3. DirectX Home Page - <http://msdn.microsoft.com/en-us/directx/default.aspx>

## 27.1.6 Conclusion

This article is just a high level description of Torque 3D's rendering system. From here you can proceed however you want. However, it is highly recommended you proceed to the next section (Source Code Tour) if you are new to Torque 3D, GFX, or rendering code in general.

## 27.2 Source Code Tour

### 27.2.1 Introduction

This document will explain the source code folders and files that make up Torque 3D's rendering system.

### 27.2.2 GFX Core

You can find the core GFX files in **engine/gfx**. Some of the source files in this system are meant to be sub-classed based on platform. There are currently 4 main gfxDevice classes:

- GFXDevice
- GFXD3D9Device (inside the D3D9 directory)
- GFXGLDevice (inside the gl directory)
- GFXNullDevice (inside the Null directory)

As you can see, each platform has its own GFXDevice. The same goes for other platform dependent files: gfxShader, gfxCubemap, gfxVertexBuffer, and so on. Some of the other source files are stand-alone. For instance, gfxStructs.h contains classes/structs and functions used by all platforms. This usually involves generic information, such as GFX-LightInfo containing light type, position, etc.

### 27.2.3 DirectX (D3D, D3D9)

Because Torque 3D supports cross-platform compatibility, it is important to keep the GFX layers separate. The D3D folder contains only two files: screenshotD3D.h and screenshotD3D.cpp. Just as the name implies, these two files are used in capturing screens while running a game. (Currently D3D9 only).

You will see the major platform specific classes represented here, such as the GFXdevice, TextureManager, VertexBuffer, etc. The functionality remains the same, but the application and execution are specific to the API.

### 27.2.4 OpenGL (gl and ggl)

Much like the D3D sections, the gl folder and its files contain rendering functionality specifically meant to run on a Mac. Once again, the major systems are integrated into the OpenGL layer (StateBlock, GFXDevice, PrimitiveBuffer, etc).

Within the gl directory is another folder: ggl. The source code in this section make up the Torque OpenGL Library. In the code itself, you will find the various OpenGL configurations, bindings, and extension definitions.



### 27.2.5 Null

The Null folder contains `gfxNullDevice.h` and `gfxNullDevice.cpp`. This device layer is used primarily for dedicated servers, which typically do not require any rendering. Dedicated servers usually just process simulation events and relays that information to the clients. There is no real reason to waste processing power and memory rendering objects that no one will see.

### 27.2.6 Sim

The 3 systems found in this folder are: `CubeMapData`, `gfxStateBlockData`, and `debugDraw`.

**debugDraw** does exactly what it sounds like. Once you enable `debugDraw`, you can pass it data (`Point3Fs` and `ColorF`) which will then render points, cubes, and other polygons. For instance, if you pass in the points that make up a Player's bounding box, you will see a box surrounding that player when running the game.

**CubemapData** is a class exposed to TorqueScript, making the class a `ConsoleObject`. A Cubemap is a texture that represents a rendering of the surrounding environment. The easiest example to explain would be a Sky cubemap rendered onto a body of water. The cubemap would consist of various sky images taken from different angles.

**GFXStateBlockData** is extremely important. Since `GFXStateBlocks` are meant to be created in script, a `ConsoleObject` that can hold `StateBlock` description was needed. `GFXStateBlockData` is that `ConsoleObject`. This covered in more detail in `Stateblocks`. (TODO - Internal link)

### 27.2.7 Test

*You can ignore this folder, as it was used during internal testing.*

### 27.2.8 RenderInstance

Another section of engine code that is considered an important part of GFX can be found under `renderInstance` (engine/`renderInstance`). The files found in this directory make up the various render managers. We will cover these in the next GFX document. For now, just remember these classes are responsible for controlling the rendering flow of your game.

### 27.2.9 Conclusion

This guide is another high level walk through of a Torque 3D system. Even though we covered some major files and concepts, you should take the time to read through the code comments left by the engine developers. Some of this will make more sense as you read through the remaining GFX documents.

## 27.3 Render Management

### 27.3.1 Introduction

The purpose of the render manager system is to gather rendering commands submitted from game code and sort them to get proper effects, draw order, and optimal performance from GFX.

### 27.3.2 RenderInst

**RenderInst** is a base structure for more task-specific render managers. The current version of **RenderInst** only contains information on sorting, translucency, and rendering type overrides.

### 27.3.3 ObjectRenderInst

**ObjectRenderInst** is derived from **RenderInst**. It does not actually contain any information about meshes, materials, transforms, etc. However, it makes use of a very important feature: Delegate callbacks. ([RenderDelegate](#) - [TODO Internal Link](#))

### 27.3.4 MeshRenderInst

Derived from **RenderInst**, a **MeshRenderInst** object contains the critical data needed for rendering. It is declared directly below **ObjectRenderInst**. Within this structure is an object's geometry(mesh), lighting information, textures, transforms, and base material.

Some of the most basic classes are used in **MeshRenderInst**:

- **GFXVertexBufferHandleBase** and **GFXPrimitiveBufferHandle** handle the vertex buffer and primitive buffer (respectively)
- World transforms and object-to-world transforms are handled by **MatrixF** pointers
- **LightInfo** pointers retain information for primary and secondary lighting
- **GFXTextureObject** pointers also aid in lighting, as well as texturing the object

### 27.3.5 RenderBinManager

**RenderBinManager** manages and signals a main list of **RenderInst** objects.

**RenderBinManager** contains the variables and functions necessary for adding, processing, sorting, and clearing **RenderInst** objects. To get a closer look at **RenderBinManager**, open `engine/source/renderInstance/renderBinManager.h` and `renderBinManager.cpp`.

Most of the important management functionality is defined in the class, but you should notice a very important chunk of functionality missing: **rendering code**. **RenderBinManager** does have a rendering function:

```
virtual void render( SceneState *state ) {}
```

But as you can see, we are not going to be directly using a **RenderBinManager** for rendering. It's important to know how the class's base functionality works, but we will get to the actual rendering code when we look at **RenderBinManager**'s children.

This class lays the ground work, but the tangible rendering sub-managers derive from **RenderBinManager**: **RenderMeshMgr**, **RenderObjectMgr**, **RenderTranslucentMgr**, and so on. These are detailed further down in the sub-managers section.

### 27.3.6 RenderPassManager

The **RenderPassManager** could be considered the "top manager" when it comes to the rendering system. The responsibilities of this manager include:

- Declaring and organizing the various RIT: "R"ender "I"nstance "T"ypes

- Allocating a render instance for MeshRenderInst, ObjectRenderInst, and so on
- Adding, sorting, and rendering a list of RenderInst's per bin
- Memory allocation and deallocation for the RenderBinManagers
- Adding, sorting, and managing the various RenderBinManagers. The importance of this task is best shown in code

See the code initializing the rendering managers, `initRenderManager.cs`:

```
// In game/core/scripts/client/renderManager.cs:
function initRenderManager()
{
    // If we already have a script version of
    // RenderPassManager (DiffuseRenderPassManager)
    // do not proceed with this function
    assert( !isObject( DiffuseRenderPassManager ), "initRenderManager() -
↳DiffuseRenderPassManager already initialized!" );

    // Create a new RenderPassManager
    new RenderPassManager( DiffuseRenderPassManager );

    // Begin adding sub-managers
    DiffuseRenderPassManager.addManager( new RenderPassStateBin() { renderOrder = 0.
↳001; stateToken = AL_FormatToken; } );

    // We really need to fix the sky to render after all the
    // meshes... but that causes issues in reflections.
    DiffuseRenderPassManager.addManager( new RenderObjectMgr() { bintype = "Sky";
↳renderOrder = 0.1; processAddOrder = 0.1; } );

    DiffuseRenderPassManager.addManager( new RenderObjectMgr() { bintype
↳= "Begin"; renderOrder = 0.2; processAddOrder = 0.2; } );
    // Normal mesh rendering.
    DiffuseRenderPassManager.addManager( new RenderMeshMgr() { bintype
↳= "Interior"; renderOrder = 0.3; processAddOrder = 0.3; } );
    DiffuseRenderPassManager.addManager( new RenderTerrainMgr() {
↳renderOrder = 0.4; processAddOrder = 0.4; } );
    DiffuseRenderPassManager.addManager( new RenderMeshMgr() { bintype
↳= "Mesh"; renderOrder = 0.5; processAddOrder = 0.5; } );
    DiffuseRenderPassManager.addManager( new RenderImposterMgr() {
↳renderOrder = 0.56; processAddOrder = 0.56; } );
    DiffuseRenderPassManager.addManager( new RenderObjectMgr() { bintype
↳= "Object"; renderOrder = 0.6; processAddOrder = 0.6; } );

    DiffuseRenderPassManager.addManager( new RenderObjectMgr() { bintype
↳= "Shadow"; renderOrder = 0.7; processAddOrder = 0.7; } );
    DiffuseRenderPassManager.addManager( new RenderMeshMgr() { bintype
↳= "Decal"; renderOrder = 0.8; processAddOrder = 0.8; } );
    DiffuseRenderPassManager.addManager( new RenderOcclusionMgr() { bintype
↳= "Occluder"; renderOrder = 0.9; processAddOrder = 0.9; } );

    // We now render translucent objects that should handle
    // their own fogging and lighting.

    // Note that the fog effect is triggered before this bin.
    DiffuseRenderPassManager.addManager( new RenderObjectMgr(ObjTranslucentBin) {
↳bintype = "ObjectTranslucent"; renderOrder = 1.0; processAddOrder = 1.0; } );
```

(continues on next page)

(continued from previous page)

```

    DiffuseRenderPassManager.addManager( new RenderObjectMgr()           { bintype_
↪= "Water"; renderOrder = 1.2; processAddOrder = 1.2; } );
    DiffuseRenderPassManager.addManager( new RenderObjectMgr()           { bintype_
↪= "Foliage"; renderOrder = 1.3; processAddOrder = 1.3; } );
    DiffuseRenderPassManager.addManager( new RenderParticleMgr()         { _
↪renderOrder = 1.35; processAddOrder = 1.35; } );
    DiffuseRenderPassManager.addManager( new RenderTranslucentMgr()       { _
↪renderOrder = 1.4; processAddOrder = 1.4; } );

    // Note that the GlowPostFx is triggered after this bin.
    DiffuseRenderPassManager.addManager( new RenderGlowMgr(GlowBin) { renderOrder = 1.
↪5; processAddOrder = 1.5; } );

    // Resolve format change token
    DiffuseRenderPassManager.addManager( new RenderPassStateBin(AL_FormatToken_Pop) { _
↪renderOrder = 1.6; stateToken = AL_FormatToken; } );
}

```

The premise behind this chunk of code is simple. Calling `DiffuseRenderPassManager` “the manager of managers” seems appropriate. As the client is being initialized, `initRenderManager()` is called to create the rendering managers.

Using the `.addManager(...)` function, the `RenderPassManager` stores an internal list of `RenderBinManagers`. We have managers for Sky, Interiors, Lighting, Shadows, and so on.

### 27.3.7 Sub-Managers

As mentioned previously, the actual rendering managers are children of `RenderBinManager`. We are calling them sub-managers, since the `RenderPassManager` manages and maintains them. Each of these rendering sub-managers contains rendering code unique to its purpose, though multiple instantiations do occur to handle our various renderable Torque objects.

Let’s go down a simplified list of these classes and their main purpose:

- **RenderObjectMgr** - This class is used for rendering more than any of the other sub-managers. This manager is responsible for rendering common objects that do not have a standard mesh.
  - Sky
  - Shadows
  - Water
  - Foliage
  - Shapebase
- **RenderMeshMgr** - This class is used for rendering mesh based objects such as interiors, TSMesh, and decals.
- **RenderTerrainMgr** - This class is used for rendering the terrain.
- **RenderRefractMgr** - Stock Torque 3D uses only one `RenderRefractMgr`. The name of the manager describes it well. This manager takes in `RenderInst` elements and checks to see if they have a refraction custom material. If this check succeeds, the element is maintained by the manager and makes use of the refraction rendering code.
- **RenderImposterMgr** - This is a special render manager for processing single billboard imposters typically generated by the `tsLastDetail` class.
- **RenderOcclusionMgr** - Used for performing occlusion queries on the scene.

- **RenderTranslucentMgr** - Stock Torque 3D uses only one RenderTranslucentMgr. This manager is a bit more complex than the previous ones described. A RenderInst element must meet a strict set of requirements to be managed by this class. If you look at RenderTranslucentMgr::addElement(...), you can see there are 3 main if(...) statements checking for translucent properties and appropriate render instance type. The actual render function is quite clean, and you can gain more insight about the class by reading through it.
- **RenderGlowMgr** - Just like the previous two managers, there is only one instance of RenderGlowMgr in stock Torque 3D. The name is pretty self-descriptive. This manager is responsible for accepting RenderInst elements that require rendering with a properly set up glow buffer.

## 27.3.8 Conclusion

The purpose of this document is to provide you with a basic understanding of the rendering management system used by Torque 3D. There is still much to be explained in the way of rendering flow, extending the system, and specific examples.

We've covered the basic renderable object instances, base class render managers, specialized rendering manager classes, and touched on some new subjects such as the RenderDelegate. (TODO - Internal Link)

## 27.4 RenderDelegate

### 27.4.1 Concept

The concept and functionality behind RenderDelgate gives you, the developer, a lot of flexibility when you are creating your own rendering objects. RenderDelgates are based on the system sending a signal, which is caught by an object's RenderDelegate. The RenderDelgate itself calls an object's render function.

Let's take a look at a few simple examples.

### 27.4.2 SkyBox Example

The SkyBox object is a great example of a custom object that requires rendering. The class is derived from SceneObject, which does not have a rendering function. Open **engine/source/environment/skyBox.h**. If you scroll through the SkyBox class, you will find the declaration of its RenderDelegate:

```
/// Our render delegate.
void _renderObject( ObjectRenderInst *ri, SceneState *state, BaseMatInstance *mi );
```

Open skyBox.cpp (same directory), then locate the SkyBox::prepRenderImage(...) function. At the bottom of the function, Sky's RenderDelegate is bound. In the following code, read each line's comment to understand what is happening:

```
// Create a render instance by asking the RenderPassManager for one
ObjectRenderInst *ri = state->getRenderPass()->allocInst<ObjectRenderInst>();

// Bind the SkyBox's rendering function to the renderDelegate
ri->renderDelegate.bind( this, &SkyBox::_renderObject );

// Set the Render Instance Type (RIT)
ri->type = RenderPassManager::RIT_Sky;

// Set the sorting keys
```

(continues on next page)

(continued from previous page)

```

ri->defaultKey = 10;
ri->defaultKey2 = 0;

// Add the render instance to the manager
state->getRenderPass()->addInst( ri );

```

When binding, we are passing in the SkyBox class (this), and its rendering function. Let's say the rendering function had a different name, such as `_renderSky`:

```

ri->renderDelegate.bind( this, &SkyBox::_renderSky);

```

What we are doing is preparing the SkyBox class to receive a render signal and act on it. One of our manager's (which we will discuss in a minute), will parse through all of its contained objects. When it comes across SkyBox, it will send a render signal to the class. It doesn't care about the name of the rendering function, it just tells the object to render with certain information:

```

void SkyBox::_renderObject( ObjectRenderInst *ri, SceneState *state, BaseMatInstance_
↪ *mi )
{
    ...

    GFXTransformSaver saver;
    GFX->setVertexBuffer( mVB );

    MatrixF worldMat = MatrixF::Identity;
    worldMat.setPosition( state->getCameraPosition() );

    SceneData sgData;
    sgData.init( state );
    sgData.objTrans = &worldMat;

    ...

```

### 27.4.3 RenderObjectExample

No doubt, if you are reading through these engine docs you are most likely a programmer. Regardless of your experience, diving head first into Torque 3D's source code can be overwhelming. If you need a very simple RenderDelegate example, you do not have to dig through the entire Player->ShapeBase->etc hierarchy.

Instead, an example class was written specifically for demonstrating a basic RenderDelegate: **RenderObjectExample**. The source files, `renderObjectExample.h` and `.cpp`, are found in **engine/source/T3D/examples**. Every line has been heavily commented to explain the purpose of the class.

RenderObjectExample has even been exposed to script and the World Editor. You can add a RenderObjectExample to your scene while debugging the engine, and go through each line of rendering code step by step.

### 27.4.4 RenderDelegate vs Other Render Methods

A render object (using a RenderDelegate) handles its own rendering by submitting itself as an ObjectRenderInst along with a delegate for its render() function. However, the preferred rendering method in the engine is to submit a MeshRenderInst along with a Material, vertex buffer, primitive buffer, and transform and let the RenderMeshMgr handle the actual rendering.

With that in mind, you may be wondering why you should ever use a `RenderDelegate` when some of the most important classes do not use it: `Player`, `Item`, `ShapeBase`, etc. While writing the actual rendering code can be complex, deciding on which method to use is simple.

When you have access to an object's 3D geometry (mesh) and material (textures and texture properties), you might as well make use of the mesh and shape rendering systems. This includes anything using COLLADA and .DTS.

If your new object does not use 3D geometry, you should look into using the `RenderDelegate` system. More importantly, integration of 3rd party technologies that handle their own rendering, such as `SpeedTree` or `Scaleform`, should definitely use `RenderDelegates`.

- `SpeedTree` - <http://www.speedtree.com/>
- `Scaleform` - <http://web.archive.org/web/20111026114129/http://www.scaleform.com/>

### 27.4.5 Conclusion

This goal of this document was to provide you with an introduction and specific examples of the `RenderDelegate` system. Should you decide to create your own custom classes which require object rendering, please refer back to this doc.

Remember, if your new object has a 3D mesh and makes use of the material system, you are encouraged to follow the example set by `RenderMeshExample` and `RenderObjectExample`. If you are implementing a very custom object, or are integrating a 3rd party product with its own rendering, using a `RenderDelegate` will be much easier.

## 27.5 Stateblocks

### 27.5.1 Concept

The purpose of `GFXStateblocks` is quite simple: an entire rendering state is contained in one object, and you are able to set the rendering state with one call. If you've written rendering code before, you may have written code similar to this before:

```
// Enable Blending
glEnable (GL_BLEND);

// Set The Blend Mode
glBlendFunc (GL_SRC_ALPHA , GL_ONE_MINUS_SRC_ALPHA)
```

What if you need to setup that kind of rendering state more than just once? `GFXStateblocks` allow you to wrap that up into the following:

```
GFX->setStateBlock(myState);
```

There's a little more tech and setup involved, but the above example should be enough to encourage you to read on and use this concept in your own project.

### 27.5.2 Technical Description

Using `GFXStateblocks` allows you to pre-generate your rendering states, which is going to save you time, cleanup your code, apply modern rendering techniques, and give you more control over your game's rendering from the engine **as well as script**. When combined with `CustomMaterials`, the material definitions in script need less custom engine support.



Stateblocks are created by filling out a `GFXStateBlockDesc` structure and calling `GFX->createStateBlock` on it. This returns a `GFXStateBlockRef`, which is a reference-counted `GFXStateBlock`. This is essentially a `GFXResource`, just like a texture. Then you use a stateblock by calling `GFX->setStateBlock` and passing the state block in.

Using a stateblock fully defines a render state, which completely does away with all concepts canonical. This is actually a good thing as this will reduce bugs introduced because a state was not properly cleaned up before exiting.

### 27.5.3 Source Code

It might help if we tour the engine code, so you can get some concrete code samples in front of your eyes. Start by opening `gfxStateBlock.h` and `gfxStateBlock.cpp`, both of which are found in `engine/source/gfx`. Let's browse the header (.h) first. All four major declarations are important here:

```
struct GFXSamplerStateDesc {...};

struct GFXStateBlockDesc {...};

class GFXStateBlock {...};

typedef StrongRefPtr<GFXStateBlock> GFXStateBlockRef;
```

### 27.5.4 GFXSamplerStateDesc

The `GFXSamplerStateDesc` struct contains variables and methods to identify the texture object used for each texture lookup. You'll find texture arguments for color, alpha, min/mag/mip filter, and address modes. A solid example of how this structure is used can be found in the `gfxD3D9StateBlock::activate(...)` function.

Inside this method, a `GFXSamplerStateDesc` variable (`ssd`) is set to one of the stateblocks internal samplers. The function performs a check to see how OpenGL handles the `GL_TEXTURE_2D` variable:

Abbreviated code from `GFXGLStateBlock::activate(...)`, found in `engine/source/gfx/gl/gfxGLStateBlock.cpp`

```
const GFXSamplerStateDesc ssd = mDesc.samplers[i];

switch (ssd.textureColorOp)
{
case GFTOPDisable:
    if(!tex)
        break;
    glDisable(GL_TEXTURE_2D);
    updateTexParam = false;
    break;
}
```

### 27.5.5 GFXStateBlockDesc

`GFXStateBlockDesc` defines a render state, which is then used to create a `GFXStateBlock` instance. If you look through the structure, you will read through some common render state ops and references: blending (source/destination/operation), depth buffer (z buffer enable/bias/definition), color writes (write red/blue/green/alpha), and so on.

A nifty concept applied through `GFXStateBlockDesc` is the combining state block descriptions. This is done so through the `::addDesc(...)` function:

```
/// Adds data from desc to this description, uses *defined
parameters in desc to figure out what blocks of
state to actually copy from desc.
void addDesc(const GFXStateBlockDesc& desc);
```

## 27.5.6 GFXStateBlock

The base GFXStateBlock class looks very bare. It is derived from StrongRefBase and GFXResource. Being a StrongRefBase object, the stateblock will exist as long as the reference exists. When all of the strong references to the stateblock go away, it is permanently deleted. It is treated as a GFXResource since our GFX system needs to track its resources owned by a particular device.

There are only 4 functions in this base class, and they are all virtual. This is due to the fact that we have multiple graphical APIs to support, which means we are going to have GFXStateBlocks for DirectX and OpenGL:

- GFXD3D9StateBlock
- GFXGLStateBlock

Of course, we have our null device stateblock (GFXNullStateBlock), but you will not actually need to use that for any rendering. The other GFX devices will have their own implementation of stateblocks. In fact, let's take a look at the GL implementation. Open engine/source/gfx/gl/gfxGLStateBlock.h. The class we are looking at is **GFXGLStateBlock**.

Within this function we have a working interface (constructor and destructor), a function called by OpenGL to activate the stateblock (::activate), and a tangible GFXStateBlockDesc member variable (mDesc). The activate function definition can be found in **gfxGLStateBlock.cpp**

```
GFXGLStateBlock::activate(const GFXGLStateBlock* oldState)
{
    // Internal code not shown
}
```

It is heavily commented and has a very important warning at the beginning. It is highly recommended you read through the comments and code to get a better understanding of how the stateblock is set up.

## 27.5.7 GFXStateBlockData

Two very important classes we need to take a look at is the GFXStateBlockData class and GFXSamplerStateData class, found in engine/source/gfx/sim/gfxStateBlockData.h/.cpp. The class definitions and their comments are quite descriptive:

```
/// Allows definition of render state via script,
basically wraps a GFXStateBlockDesc
class GFXStateBlockData: public SimObject

/// Allows definition of sampler state via script,
basically wraps a GFXSamplerStateDesc
class GFXSamplerStateData: public SimObject
```

As you read, these classes allow us to create GFXStateBlock and GFXSamplerState descriptions in TorqueScript. This means you are able to utilize a lot more custom shader and rendering work without being as reliant on custom engine code. You should definitely look through the ::initPersistFields() functions for each class. You will notice the various references and ops that make up a render state are exposed.

## 27.5.8 Engine Example

We are going to use a very simple engine example for showing how GFXStateBlocks are created and used. Start by opening *engine/source/interior/interior.h.cpp*. When we use debug rendering, we are able to see certain aspects of an interior that are normally invisible to a player. One such aspect would be the rendering of portals.

In the Interior class, we have defined multiple stateblock references. Let's look at one that affects our portal rendering:

```
GFXStateBlockRef mInteriorDebugPortalSB;
```

Ok! We have our stateblock reference, but it still needs some information. Go into *Interior.cpp*, and scroll down to the *prepForRendering* function around line 343:

```
bool Interior::prepForRendering(const char* path)
{
    // Previous and remaining code not shown

    mInteriorDebugPortalSB = GFX->createStateBlock(sh);
}
```

This line of code shows we have initialized our stateblock reference, but what device does so? We haven't seen any OpenGL or DirectX code anywhere, so how can we know if we are using a GFXGLStateBlock or not? This is where GFX takes control. **GFX->createStateBlock(...)** takes in a fully initialized GFXStateBlockDesc reference.

GFX then proceeds to set up the hash value and search for existing stateblocks. If the stateblock we need does not exist, it calls **createStateBlockInternal**. This function drills into our platform's device, so we have a GFXGLDevice::createStateBlockInternal and a GFXD3D9Device::createStateBlockInternal.

When creating a GFXStateBlock in script, quite a few things happen for you automatically. However, when you are in the engine there are a few "Gotchas" you have to remember. **When you create a GFXStateBlock in the engine, you must call setStateBlock before you use it!**

**Called in Interior::debugRenderPortals():**

```
GFX->setStateBlock(mInteriorDebugPortalSB);
```

A simple, more generic engine example would be this:

```
// Setup code, done once
GFXStateBlockDesc desc;
desc.setBlend(true, GFXSrcAlpha, GFXInvSrcAlpha);
GFXStateBlockRef myState = GFX->createStateBlock(desc);

// Render time code
GFX->setStateBlock(myState);
```

## 27.5.9 Script Example

Earlier, I mentioned using shader data and GFXStateBlocks together in script. We have provided you with a couple of examples. Open *Examples/FPS Example/game/core/scripts/client/postFX.cs*. This file contains multiple GFXStateBlockData and ShaderData definitions. First, we focus on the default stateblock:

```
singleton GFXStateBlockData( PFX_DefaultStateBlock )
{
    zDefined = true;
```

(continues on next page)

(continued from previous page)

```

zEnable = false;
zWriteEnable = false;

samplersDefined = true;
samplerStates[0] = SamplerClampLinear;
};

```

The above code demonstrates how to declare a `GFXStateBlock` in TorqueScript using the exposed Console Object, **GFXStateBlockData**. In this example, we define depth sorting and handle linear clamping. We can then use this stateblock in a `PostEffect` definition:

```

singleton PostEffect( BL_ShadowFilterPostFx )
{
    requirements = "";

    shader = BL_ShadowFilterShader;
    stateBlock = PFX_DefaultStateBlock;
    targetClear = "PFXTargetClear_OnDraw";
    targetClearColor = "0 0 0 0";
    texture[0] = "$inTex";
    target = "$outTex";
};

```

A `GFXStateBlockData` definition can also be used in the creation of a separate stateblock, as shown below:

```

singleton GFXStateBlockData( LightRayStateBlock: PFX_DefaultStateBlock )
{
    samplersDefined = true;
    samplerStates[0] = SamplerClampLinear;
    samplerStates[1] = SamplerClampLinear;
};

```

Essentially, we wanted the exact same rendering state for our two post processing effects. Instead of having to modify our engine code (twice), we can define our stateblock in script (once) and use it multiple times.

## 27.5.10 Conclusion

There is a lot to learn about `GFXStateBlocks`. The intent of this article was to give you a strong introduction on the purpose, engine structure, and examples of how to use this new system. As you develop new shaders, look for ways you can save yourself time and headaches by using `GFXStateBlocks`.

## 27.6 Shader Constant Buffers

### 27.6.1 Concept

Much like state blocks, shader constant buffers help reduce the amount of rendering code you have to write. This concept is quite similar to DirectX 10's version of constant buffers, which share a similar purpose:

Quoted concepts from DirectX HLSL MSDN:

```

A constant buffer, or shader constant buffer, is a
buffer that contains shader constants.

```

(continues on next page)

(continued from previous page)

Conceptually, it looks just like a single-element vertex buffer.

A constant buffer **is** a specialized buffer resource that **is** accessed like a buffer.

A buffer resource **is** designed to minimize the overhead of setting shader constants.

## 27.6.2 Technical Description

A shader constant buffer is just a block of memory (or an object) that contains the constants that a shader needs to have bound. Specifically, they buffer a collection of string/value pairs that are sent to a shader. They are allocated by the shader itself because the shader may have information about layout that no other part of the system will know about.

You can look up handles by shader constant name. This removes the need for a static shader constant-to-slot mapping and also allows us to do some CPU side optimizations. Under the hood, the string/value pair is mapped to a block of memory that can be handed to a shader with one call.

## 27.6.3 Source Code

We are going to take a light tour of the engine code that makes up the `GFXShaderConstBuffer` system. Let's start by opening `engine/source/gfx/gfxShader.h`. Toward the top of this header, you will see a struct and two base classes that make up this system:

```
/// Instances of this struct are returned GFXShaderConstBuffer
struct GFXShaderConstDesc {...};

/// This is an opaque handle used by GFXShaderConstBuffer
/// clients to set individual shader constants.
/// Derived classes can put whatever info they need into here,
/// these handles are owned by the shader constant buffer
/// (or shader). Client code should not free these.
class GFXShaderConstHandle {...};

/// GFXShaderConstBuffer is a collection of
/// string/value pairs that are sent to a shader.
/// Under the hood, the string value pair is
/// mapped to a block of memory that can
/// be blasted to a shader with one call (ideally)
class GFXShaderConstBuffer {...}
```

It should be obvious by the amount of pure virtual function declarations that `GFXShaderConstHandle` and `GFXShaderConstBuffer` are meant to have children. Both `GFXD3D9` and `GFXGL` will get their own versions of shader constant buffers. In this file, there is not much you can learn from `GFXShaderConstHandle`. However, you might want to scan `GFXShaderConstBuffer`.

It is derived from `StrongRefBase` and `GFXResource`. Being a `StrongRefBase` object, the stateblock will exist as long as the reference exists. When all of the strong references to the stateblock go away, it is permanently deleted. It is treated as a `GFXResource` since our `GFX` system needs to track its resources owned by a particular device.

The virtual function declarations should give you insight on how we are setting up the future child functionality. We have almost twenty overloaded functions named `set(...)`. Each one takes in a `GFXShaderConstHandle*` and a constant. The handles contains the name of the constant, which should be a name contained in the array returned in `getShaderConstDesc`.

It is very important that you remember what `GFXShaderConstBufferRef` is:

In `engine/source/gfx/gfxShader.h`:

```
typedef StrongRefPtr<GFXShaderConstBuffer> GFXShaderConstBufferRef;
```

`GFXShaderConstBufferRef` is like a pointer (reference) to `GFXShaderConstBuffer`, with a few key differences. A few paragraphs back I mentioned GFX tracking its resources and references, and this is one of the ways it does so. This is a **referenced counted**, object template pointer class (quite descriptive!). Heavy emphasis on the **referenced counted**.

If you open `engine/source/gfx/gl/gfxGLShader.h`, you can see how we have set up our OpenGL version of a constant shader buffer.

**We have our child class declaration:**

```
class GFXGLShaderConstBuffer: public GFXShaderConstBuffer{...}
```

**The GL version contains an activation function:**

```
/// Called by GFXGLDevice to activate this buffer.
void activate();
```

**The class even keeps track of the shader that creates the buffer:**

```
WeakRefPtr<GFXGLShader> mShader;

/// Return the shader that created this buffer
virtual GFXShader* getShader() { return mShader; }
```

Of course, the multiple virtual void `set(...)` functions get defined as well, but in the `gfxGLShader.cpp` file. The base relationship you should remember is a GFX shader creates and uses a shader constant buffer, while the constant buffer keeps track of its owner and sets the actual constants.

## 27.6.4 Engine Example

As for an engine example, I'll start with a generic chunk of code:

**We will set up our constant buffer and handle once:**

```
// Setup code
GFXShaderConstBufferRef myBuff = shader->allocConstBuffer();
GFXShaderConstHandle* myHandle = shader->getShaderConstHandle("$diffuseColor");
```

**Now you can set the constant buffer:**

```
// Render code
myBuff->set(myHandle, myConst);
GFX->setShaderConstBuffer(myBuff);
```

Now that you see the basic code concept, let's examine an existing constant buffer in the engine. The `CloudLayer` class handles its shader constant buffer internally. Open `engine/source/environment/cloudLayer.cpp`. Scroll down until you see the following function:

```
bool CloudLayer::onAdd() {...}
```

Further into the function, around line 92, you can see where the internal GFXShaderConstBufferRef (mShaderConsts) is allocated:

```
// Create ShaderConstBuffer and Handles
mShaderConsts = mShader->allocConstBuffer();
```

The GFXShaderConstHandle pointers are internal members belonging to the CloudLayer class:

```
**In engine/source/environment/cloudLayer.h**::
```

```
GFXShaderConstHandle *mModelViewProjSC;  GFXShaderConstHandle *mAmbientColorSC;
GFXShaderConstHandle *mSunColorSC;  GFXShaderConstHandle *mSunVecSC;  GFXShader-
ConstHandle *mTexOffsetSC[3];  GFXShaderConstHandle *mTexScaleSC;  GFXShaderConstHandle
*mBaseColorSC;  GFXShaderConstHandle *mCoverageSC;  GFXShaderConstHandle *mEyePos-
WorldSC;
```

Back in the cloudLayer.cpp, these handles are set after the buffer has been allocated:

```
mModelViewProjSC = mShader->getShaderConstHandle( "$modelView" );
mEyePosWorldSC = mShader->getShaderConstHandle( "$eyePosWorld" );
mSunVecSC = mShader->getShaderConstHandle( "$sunVec" );
mTexOffsetSC[0] = mShader->getShaderConstHandle( "$texOffset0" );
mTexOffsetSC[1] = mShader->getShaderConstHandle( "$texOffset1" );
mTexOffsetSC[2] = mShader->getShaderConstHandle( "$texOffset2" );
mTexScaleSC = mShader->getShaderConstHandle( "$texScale" );
mAmbientColorSC = mShader->getShaderConstHandle( "$ambientColor" );
mSunColorSC = mShader->getShaderConstHandle( "$sunColor" );
mCoverageSC = mShader->getShaderConstHandle( "$cloudCoverage" );
mBaseColorSC = mShader->getShaderConstHandle( "$cloudBaseColor" );
```

The actual setting of the shader data, constant buffer, and stateblock does not happen until further down in the source file. If you scroll down to around line 264, you will find the rendering function for CloudLayer:

```
void CloudLayer::renderObject( ObjectRenderInst *ri, SceneState *state,
↳BaseMatInstance *mi ){...}
```

On line 276, GFX takes over and performs the “set” code:

```
GFX->setShader( mShader );

// HERE WE SET THE SHADER CONSTANT BUFFER
GFX->setShaderConstBuffer( mShaderConsts );

GFX->setStateBlock( mStateblock );
```

## 27.6.5 Script Example

Using shader constant buffers in TorqueScript is a little different than in the engine code. At this time, most game developers know about the SSAO (Screen Space Ambient Occlusion) rendering technique. Torque 3D has a SSAO solution, which is defined in TorqueScript.

SSAO is a PostEffect, so it must be defined as such. Locate and open **Examples/FPS Example/game/core/scripts/client/postFx/ssao.cs**. The effect is declared using the following code (reduced to just show the declaration):



```
singleton PostEffect( SSAOPostFx ){...};
```

In the CloudLayer example, I mentioned the class contained internal GFXShaderConstHandle pointers as member variables. In TorqueScript, SSAO uses scoped global variables:

```
// The small radius SSAO settings.
$SSAOPostFx::sRadius = 0.1;
$SSAOPostFx::sStrength = 6.0;
$SSAOPostFx::sDepthMin = 0.1;
$SSAOPostFx::sDepthMax = 1.0;
$SSAOPostFx::sDepthPow = 1.0;
$SSAOPostFx::sNormalTol = 0.0;
$SSAOPostFx::sNormalPow = 1.0;

// The large radius SSAO settings.
$SSAOPostFx::lRadius = 1.0;
$SSAOPostFx::lStrength = 10.0;
$SSAOPostFx::lDepthMin = 0.2;
$SSAOPostFx::lDepthMax = 2.0;
$SSAOPostFx::lDepthPow = 0.2;
$SSAOPostFx::lNormalTol = -0.5;
$SSAOPostFx::lNormalPow = 2.0;
```

These variables are used when setting the buffer. As a script object, SSAOPostFx can have member functions. An important function to define for PostEffect objects is **setShaderConsts(%this)**. If a PostEffect object, such as SSAOPostFx, has this function defined, it will be called by the engine automatically.

If you scroll down to the function, you can see how it sets the shader constant buffer:

```
function SSAOPostFx::setShaderConsts(%this )
{
    %this.setShaderConst( "$overallStrength", $SSAOPostFx::overallStrength );

    // Abbreviate is s-small l-large.

    %this.setShaderConst( "$sRadius",      $SSAOPostFx::sRadius );
    %this.setShaderConst( "$sStrength",    $SSAOPostFx::sStrength );
    %this.setShaderConst( "$sDepthMin",    $SSAOPostFx::sDepthMin );
    %this.setShaderConst( "$sDepthMax",    $SSAOPostFx::sDepthMax );
    %this.setShaderConst( "$sDepthPow",    $SSAOPostFx::sDepthPow );
    %this.setShaderConst( "$sNormalTol",    $SSAOPostFx::sNormalTol );
    %this.setShaderConst( "$sNormalPow",    $SSAOPostFx::sNormalPow );

    %this.setShaderConst( "$lRadius",      $SSAOPostFx::lRadius );
    %this.setShaderConst( "$lStrength",    $SSAOPostFx::lStrength );
    %this.setShaderConst( "$lDepthMin",    $SSAOPostFx::lDepthMin );
    %this.setShaderConst( "$lDepthMax",    $SSAOPostFx::lDepthMax );
    %this.setShaderConst( "$lDepthPow",    $SSAOPostFx::lDepthPow );
    %this.setShaderConst( "$lNormalTol",    $SSAOPostFx::lNormalTol );
    %this.setShaderConst( "$lNormalPow",    $SSAOPostFx::lNormalPow );

    %blur =%this->blurY;
    %blur.setShaderConst( "$blurDepthTol", $SSAOPostFx::blurDepthTol );
    %blur.setShaderConst( "$blurNormalTol", $SSAOPostFx::blurNormalTol );

    %blur =%this->blurX;
    %blur.setShaderConst( "$blurDepthTol", $SSAOPostFx::blurDepthTol );
```

(continues on next page)

(continued from previous page)

```

%blur.setShaderConst( "$blurNormalTol", $SSAOPostFx::blurNormalTol );

%blur =%this->blurY2;
%blur.setShaderConst( "$blurDepthTol", $SSAOPostFx::blurDepthTol );
%blur.setShaderConst( "$blurNormalTol", $SSAOPostFx::blurNormalTol );

%blur =%this->blurX2;
%blur.setShaderConst( "$blurDepthTol", $SSAOPostFx::blurDepthTol );
%blur.setShaderConst( "$blurNormalTol", $SSAOPostFx::blurNormalTol );
}

```

## 27.6.6 Conclusion

The intent of this document was to provide you with a strong introduction to GFX shader constant buffers. There are various examples scattered throughout the code, so you might want to spend some more time browsing through the code and comments.

Should you decide to create your own custom classes with renderable objects, remember to check this document again and see how you can use constant buffers in your code. The optimization is well worth the learning curve.

## 27.7 Interface

### 27.7.1 Introduction

The following categories represent the GFX interface in TorqueScript.

### 27.7.2 Core Global Variables

default values

- **\$pref::Video::displayDevice = "D3D9";**
  - User's preference for device
- **\$pref::Video::disableVerticalSync = 1;**
  - Toggles adapter to wait for vsync
- **\$pref::Video::mode = "1024 768 false 32 60 0";**
  - Contains settings for video resolution, screen mode (full/windowed), bit depth, refresh rate, and anti-aliasing
- **\$pref::Video::screenShotSession = 0;**
  - Number attached to **screenshot\_** file when taking an actual screenshot from within the game.
- **\$pref::Video::screenShotFormat = "PNG";**
  - The file format of a screenshot taken in game, either .jpg or .png

## 27.7.3 Script Classes

### Render Managers

For more detailed information, refer back to the Render Management Guide. (TODO - Internal Link)

#### RenderObjectMgr

This class is used for rendering more than any of the other sub-managers. This manager is responsible for rendering common objects that do not have a standard mesh.

- Sky
- Shadows
- Water
- Foliage
- Shapebase

#### RenderMeshMgr

This class is used for rendering mesh based objects such as interiors, TSMesh, and decals.

#### RenderTerrainMgr

This class is used for rendering the terrain

#### RenderRefractMgr

Stock Torque 3D uses only one RenderRefractMgr. The name of the manager describes it well. This manager takes in RenderInst elements and checks to see if they have a refraction custom material. If this check succeeds, the element is maintained by the manager and makes use of the refraction rendering code.

#### RenderImposterMgr

This is a special render manager for processing single billboard imposters typically generated by the tsLastDetail class.

#### RenderOcclusionMgr

Used for performing occlusion queries on the scene.

#### RenderTranslucentMgr

Stock Torque 3D uses only one RenderTranslucentMgr. This manager is a bit more complex than the previous ones described. A RenderInst element must meet a strict set of requirements to be managed by this class. If you look at RenderTranslucentMgr::addElement(...), you can see there are 3 main if(...) statements checking for translucent properties and appropriate render instance type. The actual render function is quite clean, and you can gain more insight about the class by reading through it.

## RenderGlowMgr

Just like the previous two managers, there is only one instance of RenderGlowMgr in stock Torque 3D. The name is pretty self-descriptive. This manager is responsible for accepting RenderInst elements that require rendering with a properly set up glow buffer.

## GFXStateBlockData

For more detailed information, refer back to the Stateblocks Guide.

## Description

Allows definition of render state via script, basically wraps a GFXStateBlockDesc

## PostEffect

### Description

Class used for defining post-processing effects, such as depth of field, SSAO, and light rays.

### Methods

- *\*void reload()*: Reloads the effect shader and textures.
- *\*void enable()*: Enables the effect.
- *\*void disable()*: Disables the effect.
- *\*bool toggle()*: Toggles the effect state returning true if we enable it.
- *\*bool isEnabled()*: Returns true if the effect is enabled.
- *\*void setShaderConst()*: Sets the shader constant buffer for this effect
- *\*F32 getAspectRatio()*: Returns width over height aspect ratio of the backbuffer.
- *const char dumpShaderDisassembly()*: Dumps this PostEffect shader's disassembly to a temporary text file. Returns the fullpath of that file if successful.
- *\*void setShaderMacro( string key, [string value]*: Add/set a shader macro.

## GFXSamplerStateData

### Description

Allows definition of sampler state via script, basically wraps a GFXSamplerStateDesc

## GFXInit

### Description

Interface for tracking GFX adapters and initializing them into devices.

## Methods

- *\*float* getSoundDuration(): Return the duration (in seconds) of the sound referenced by the profile.
- *\*S32* getAdapterCount(): Return the number of adapters available.
- *const char* getAdapterName(int id): Returns the name of a given adapter.
- *\*const char\** getAdapterType(int id): Returns the type (D3D9, D3D8, GL, Null) of a given adapter.
- *\*F32* getAdapterShaderModel(int id): Returns the SM supported by a given adapter.
- *\*S32* getDefaultAdapterIndex(): Returns the index of the adapter we'll be starting up with.
- *\*S32* getAdapterModeCount(int id): Gets the number of modes available on the specified adapter. id Index of the adapter to get data from. Return the number of video modes supported by the adapter, or -1 if the given adapter was not found.
- *const char* getAdapterMode(int id, int modeId): Gets information on the specified adapter and mode. id Index of the adapter to get data from. Param modeId Index of the mode to get data from. Return (string) a video mode string given an adapter and mode index.
- *\*void* createNullDevice(): Create a NULL device.

### 27.7.4 Conclusion

This interface guide covers everything you will need to know about using Torque 3D's stock render system (GFX) in TorqueScript.

## CHAPTER 28

---

### License

---

Copyright (c) 2012 GarageGames, LLC

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.





## Symbols

`_getLightManagers` (C++ function), 560  
`_update4LightConsts` (C++ function), 555  
`~LightManager` (C++ function), 560

## A

`activate` (C++ function), 559, 561, 564, 567, 568  
`addExtended` (C++ function), 571

## C

`canActivate` (C++ function), 568  
`cls` (C++ function), 331

## D

`deactivate` (C++ function), 559, 562, 564, 568, 569  
`deleteAllLightInfoEx` (C++ function), 571

## E

`echo` (C++ function), 331, 417  
`eval` (C++ function), 332

## F

`findByName` (C++ function), 560  
`findShadowMapForObject` (C++ function), 567

## G

`getAllUnsortedLights` (C++ function), 556  
`getConeMesh` (C++ function), 566  
`getDefaultLight` (C++ function), 558  
`getExtended` (C++ function), 571  
`getLightBinManager` (C++ function), 563  
`getLightManagerNames` (C++ function), 560  
`getSceneLightingInterface` (C++ function), 555  
`getSpecialLight` (C++ function), 558  
`getSphereMesh` (C++ function), 566  
`getWorldToLightProj` (C++ function), 570

## I

`initLightFields` (C++ function), 559  
`isCompatible` (C++ function), 561, 563

## L

`LightManager` (C++ function), 561  
`lightScene` (C++ function), 554

## P

`packExtended` (C++ function), 570  
`Player::allowAllPoses` (C++ function), 477  
`Player::allowCrouching` (C++ function), 477  
`Player::allowJetJumping` (C++ function), 477  
`Player::allowJumping` (C++ function), 477  
`Player::allowProne` (C++ function), 477  
`Player::allowSprinting` (C++ function), 477  
`Player::allowSwimming` (C++ function), 477  
`Player::checkDismountPoint` (C++ function), 478  
`Player::clearControlObject` (C++ function), 478  
`Player::getControlObject` (C++ function), 478  
`Player::getDamageLocation` (C++ function), 478  
`Player::getNumDeathAnimations` (C++ function), 480  
`Player::getPose` (C++ function), 480  
`Player::getState` (C++ function), 480  
`Player::setActionThread` (C++ function), 480  
`Player::setArmThread` (C++ function), 482  
`Player::setControlObject` (C++ function), 482  
`PlayerData::onEnterLiquid` (C++ function), 483  
`PlayerData::onEnterMissionArea` (C++ function), 484  
`PlayerData::onLeaveLiquid` (C++ function), 484  
`PlayerData::onLeaveMissionArea` (C++ function), 484

`PlayerData::onPoseChange` (C++ *function*), 484  
`PlayerData::onStartSprintMotion` (C++  
*function*), 484  
`PlayerData::onStartSwim` (C++ *function*), 484  
`PlayerData::onStopSprintMotion` (C++ *func-*  
*tion*), 484  
`PlayerData::onStopSwim` (C++ *function*), 484  
`ProximityMine::explode` (C++ *function*), 498

## R

`registerGlobalLight` (C++ *function*), 557, 564  
`registerGlobalLights` (C++ *function*), 557  
`registerLocalLight` (C++ *function*), 556

## S

`set` (C++ *function*), 572  
`setGFXLight` (C++ *function*), 572  
`setLightInfo` (C++ *function*), 562, 565  
`setLightShadowMap` (C++ *function*), 569  
`setLightShadowMapForLight` (C++ *function*),  
569  
`setSpecialLight` (C++ *function*), 558  
`setTextureStage` (C++ *function*), 563, 566

## U

`unpackExtended` (C++ *function*), 570  
`unregisterAllLights` (C++ *function*), 556, 565  
`unregisterGlobalLight` (C++ *function*), 557  
`unregisterLocalLight` (C++ *function*), 556