

The Ultimate Guide to Torque 3D
Volume I: By Robert C. Fritzen

|

The Ultimate Guide to Torque 3D
Edition I
Written for Torque 3D 3.5.1 by GarageGames



By: Robert C. Fritzen

Introduction

So, here I am again, doing another work for Torque 3D, even after the numerous “I’m Done!” topics I have posted on the GarageGames website, however this one will be my final entry to the engine itself, and it will probably serve to contain the greatest value of all of the works I have done for Torque 3D in the past.

Basically, imagine holding a key of knowledge and this key opens the door of pure information behind the engine, and all of its functioning. That’s what I’m offering up here, an insight into the game engine that launched me into computer programming and game design and development.

I’ve done a few write up documents in the past with some of my packs I have released, namely, a 30 page work on how to properly design a First Person Shooter, and giving you the general concepts and rules as well as some code examples to show you some basics behind the thought process. I’ve also taken some time to write a long tutorial series on third person weapon modelling, which was a fun experience, as I was still learning the topic myself, and while I got some criticism for that work, this one, I doubt will get me the same level of criticism.

I’m going to take you on a “guided tour” of the engine, explaining how the internal portion of the engine functions, showing you how to create compact, and powerful scripts to deploy a game the way you want to make it, and maybe if I’m lucky, push a few individuals to my level of knowledge where all they need is a spark of inspiration to create the next amazing piece of work for the community.

Table of Contents

Chapter 1: Welcome to Torque!	1
Chapter 2: Setting Up, Getting the Files & Compiling.....	2
Git Introduction.....	2
Getting the Files.....	3
The Torque 3D Project Manager.....	3
Compiling the Engine	5
Chapter 3: The World Editor	7
The World Editor Toolbar	9
The Quick Settings Bar	11
The World Editors	12
Object Editor (F1).....	12
Terrain Editor (F2).....	15
Terrain Painter (F3)	17
Material Editor (F4).....	19
Sketch Tool (F5).....	21
Datablock Editor (F6)	22
Decal Editor (F7).....	23
Forest Editor (F8)	24
Mesh Road Editor (F9)	26
Mission Area Editor.....	29
Particle Editor.....	29
River Editor.....	31
Road Editor (Decal Road Editor)	31
Shape Editor.....	32
First Steps.....	33
Design Rules	33
Design Concepts.....	34
Adding Terrain.....	34
Boundaries	36
Terrain Noise / River Sections.....	36
Coloring The Terrain.....	38

Adding an Environment	40
Purging that Black Abyss: Skies	40
Clouds, Precipitation, Storm Objects	41
Water	44
Forest	45
Adding some Objects	47
Bridges	47
Spawn Points.....	48
SimGroups: A World Editor Perspective	50
Naming & Level Parameters: What is theLevelInfo?	51
Additional Topics.....	52
Barriers Revisited	52
Cleaning Up the Empty Zones	54
Lights	54
Physical Zones	56
Chapter 4: The GUI Editor	57
The GUI Editor Toolbar	58
The Quick Settings Bar	60
The Sidebar & Options	60
Inspector (GUI).....	60
Library	61
Profiles	62
Building a GUI... From Start to Finish	63
Getting Started.....	64
The Resolution Problem.....	64
Containers and Windows, Screen Sizing	66
Buttons and Button Variants	68
Text Instances	70
Text Input	72
Images	72
Editing an Existing GUI	74
Chapter 5: Introduction to Computer Programming	75

Brief Introduction.....	75
Variables, Definitions, and Methods.....	76
Scope.....	76
What is Syntax.....	77
Error Checks, your new Best Friend.....	79
Being an Organized Coder.....	79
Developing By Process	81
Chapter 6: TorqueScript 101.....	82
Tribal-IDE.....	82
Introduction	83
Variables	83
Keywords.....	84
Keyword: break.....	84
Keywords: case, or, switch, and switch\$	85
Keyword: continue	86
Keywords: datablock & singleton	86
Keyword: default.....	88
Keywords: else & else if & if.....	88
Keywords: false & true.....	89
Keyword: for.....	89
Keyword: function.....	89
Keyword: new	90
Keyword: package.....	91
Keyword: parent	91
Keyword: return	92
Keyword: while	92
Operators	92
Arithmetic Operators	93
Assignment Operators	93
Bitwise Operators	93
Comparison Operators (Logical Operators)	94
Miscellaneous Operators	95

Functions & Parameters.....	96
Basics of Functions.....	96
Parameters.....	97
Calling Functions, Returning Values.....	99
Conditionals	101
If, Else If, Else Blocks	101
Complex Logical Operations	104
Switch Blocks.....	106
Conditional Operator	107
Error Checks	108
Loops.....	109
While Loops.....	109
For Loops.....	110
How Scripts are Loaded	111
Client-Side Versus Server-Side	112
Final Notes	112
Chapter 7: Intermediate TorqueScript, Digging Deeper.....	114
Arrays	114
Defining an Array	114
Working with Arrays	114
The Index of an Array.....	116
Multi-Dimensional Arrays	117
Topics with Global Variables.....	118
Constant Values	118
Data Enumerations	119
Global Array Instances	120
Timers, Indirect Function Calls.....	120
Schedule & Cancel.....	120
More on Indirect Function Calls.....	122
Topics with String type Variables.....	123
String Keywords	123
Words, Fields, and Records.....	124

Basic String Functions	127
Comparison Functions	129
Extraction Functions	129
Closing Notes	130
Chapter 8: Game Objects, Classes, Packages.....	131
Chapter Introduction	131
Introduction to Objects in Torque	131
Creating an Object	132
Object ID's and Names.....	133
Object Parameters and the Access Operator	133
Classes, Object Functions (Direct & Indirect), and the Dump Command	136
Defining a Class, what is a Super-Class?	136
Multiple Inheritance	137
Direct Inheritance	138
Introduction to Object Methods, Namespace Operator, Direct Calls to Object Methods	139
Indirect Object Method Calls	142
The 'Dump' Command	143
Datablocks.....	144
Rules.....	144
How to Define a Datablock	144
SimGroups Revisited: The TorqueScript Perspective.....	147
SimGroup & SimSet Defined	147
Creating a SimGroup & SimSet	147
Adding and Removing Objects from Groups	148
Listing Functions	150
Packages, the Parent Keyword, and Function Overloading.....	152
Introduction to Packages	152
The Parent Keyword & Overloading Functions.....	157
Closing Notes	160
Chapter 9: Mathematics for Video Games: The Basics.....	161
Welcome Back!	161
Modulation	161

Working with Exponents in Computers	162
Raising to a Power, Taking a Root.....	162
Logarithms	162
Powers of 2	163
Basic Mathematical Functions	163
Floor	163
Ceil.....	163
Rounding	164
Absolute Values	164
Minima & Maxima Functions.....	164
Clamping	164
Linear Interpolation	164
Trigonometry: Back with a vengeance!	164
Pi.....	165
The Unit Circle.....	165
Radians & Degrees	166
Trigonometric Functions, Relations	166
Introduction to Vectors, Positions	167
Vector Mathematics	169
Unit Vectors & Vector Normalization	169
Vector Addition.....	170
Vector Subtraction	171
Vector Dot Product, Vector Scaling Product.....	171
Vector Cross Product	173
Basic Matrix Operations for TorqueScript	174
Transforms	175
Matrix Vector Transformations.....	175
Multiplying	175
Bézier Curves, Determining a Path	176
Mathematical Definition	176
Linear, Quadratic, and Cubic Definitions	177
Control Points of the Curve.....	177

Writing a Bézier Curve in TorqueScript.....	178
Mathematical Uncertainty, AKA: Randomization.....	180
Numerical Seeding	181
How to beat Pseudo-random.....	182
How to solve Mathematical Problems in Programming	182
Some Final Mathematical Rules.....	182
Numerical Integration	183
Differentiation.....	185
Table of Differentiation Rules	187
From the Programming Perspective	188
Closing Notes	189
Chapter 10: GUI's Again, Wiring Things Together	190
Chapter Introduction	190
The Command Parameter in more Detail.....	190
Revisiting the Command Parameter.....	190
The eval Function	191
How to 'actually' use the control parameter.....	192
GUI Functions.....	192
Global GUI Functions	192
GUIControl Functions.....	196
GUIControl Callback Functions	197
Final Notes on Engine-Defined GUI Functions	200
Creating Custom GUI Functions	200
The GUI	200
Strategy of Approach	201
Application of the Strategy: Let's Write the GUI!	202
Additional Wiring Topics	205
Wiring: Dialog to Dialog	205
Wiring: Control to Dialog	207
Concluding Notes on Wiring	211
Final Remarks.....	211
Chapter 11: Advanced TorqueScript Topics: Entering the Game World	213

Chapter Introduction	213
Game Objects, Revisiting Objects	213
Object Inheritance Tree	213
Revisiting Datablocks	214
Object Functions Revisited.....	214
Scripted Events, and Enhancing our Map	215
The Trigger Object.....	215
When & Where to use This.....	217
Proper Trigger Placement.....	217
Players.....	219
The Player Datablock	219
Spawning a Player Object	227
Functions & Callbacks for Player.....	229
Items	231
The Item Datablock	231
Functions and Callbacks for Item	233
Weapons	234
Intro to Weapons	235
Weapon Properties (Datablock Fields)	235
The Weapon State System.....	238
Functions for Weapon Images	247
How to create various types of Weapons.....	250
Projectiles.....	262
The Projectile Datablock	262
Functions for Projectiles	264
Additional Scripting Topics.....	264
Object Typemasks, First Applications of Bitwise Operators	265
Raycasts & Container Searches.....	267
AI Players.....	271
Relating AIPlayer to Player.....	271
How to spawn an AI Player	271
Functions for AIPlayer Instances.....	272

AI Path Object, Routes	274
Setting up a Path.....	274
How to get an AIPlayer to follow the Path.....	277
Introduction to Recast	280
Using the Recast System in T3D.....	280
Strategies to create New Objects	283
Revisiting Object Inheritance	283
Getting Everything Together.....	285
Final Remarks.....	285
Chapter 12: Introduction to Networking In Torque.....	286
Chapter Introduction	286
Client versus Server, Instances	286
Instances	286
Client & Server Loading.....	287
Client Object: The GameConnection Class.....	287
What is a client object?.....	287
The Connection Process.....	288
Functions for the GameConnection Class.....	289
Client & Server Commands	292
CommandToClient and CommandToServer	292
Tags	293
Function Syntax.....	293
Introduction to the Ghosting System.....	295
What is a Ghost?	295
Ghost ID's	295
A little more on Scoping.....	296
Resolving Ghosts	297
Finishing Up.....	298
Chapter 13: C++ for Absolute Beginners.....	299
Chapter Introduction	299
Getting started in Visual Studio	299
Solutions & Projects	300

C++ Basics: Getting Started with Data Types	302
Data Types.....	302
Signed & Unsigned Types.....	305
Constant Variable Types	306
Entry Point.....	307
C++ Basics: Functions	308
C++ Functions, Parameters, and Predefined Values.....	308
Function Stubs, Predefining Functions (Prototypes), And Header Files	310
Inline Functions.....	312
Keywords of C++	313
Reviewing the Old	313
New Stuff	314
Conditionals and Loops in C++.....	322
If and Else, back once again	323
Switch.....	324
For Loops.....	325
While Loops.....	326
Do-While Loops.....	327
Final Remarks.....	328
Chapter 14: Diving into the Core of C++	329
Chapter Introduction	329
Arrays in C++	329
Reviewing, More of the Same	329
Multi-Dimensional Arrays	330
For-Loop Review	331
New Array Concepts: Memory & Function References	332
Arrays as seen by Computers.....	332
Passing Arrays as Function Parameters	333
References	335
The Reference-Address Operator (&)	335
Introduction to Pointers.....	337
Pointer Introduction: Variables.....	337

The Dereference Operator.....	339
Comparing Pointers	342
Pointer Variables.....	344
Returning Pointers from Functions.....	346
Double-Pointers	348
The Generic Pointer, Type-Casting, and Templates.....	349
The Generic Pointer	349
Type-Casting.....	350
Templates, Revisiting the Generic Pointer.....	351
Memory Tools	354
Memory Allocation	354
Memory Reallocation.....	357
Free & Delete	359
Lists, Vectors	360
Introduction: Linked List	361
Structures.....	362
Structure Functions, a Preview of Chapter 16	364
Linked List: Examples	367
Introduction: Vectors.....	369
Vectors: Examples.....	370
Namespaces	373
Intro to Namespaces, Revisiting the using Keyword	373
Multiple-Layered Namespaces	374
Exception Handling	377
The throw Keyword.....	377
Try-Catch Blocks.....	378
The Standard Template Library.....	380
Final Remarks.....	383
Chapter 15: Opening Up the Engine, Combining C++ and TorqueScript	384
Chapter Introduction	384
C++ Data Type Trickery (Typedefs)	384
C++ Macros & Conditional Inclusions	386

Conditional Inclusions	389
How to add your own C++ Files to Torque	391
Creating new TorqueScript Functions in C++.....	393
Returning Data	394
Calling Pre-Written Functions	395
Calling a TorqueScript Function in C++	396
Pointer Variables in C++ Defined TorqueScript Functions.....	397
Entry Points	399
Final Remarks.....	400
Chapter 16: The Key of C++, Object-Oriented Programming	401
Chapter Introduction	401
C++ Header Files	401
Header Files: Introduction	401
Header Files: Format.....	402
The #include Statement and your Header Files.....	402
Introduction to Classes, Objects	404
Classes: Introduction.....	404
Classes: Access Level Modifiers	405
Template Classes.....	411
Specialized Templates.....	412
Multiple T-Variables, Partial Template Specialization	415
Operator Overloading, this Keyword	417
Which Operators can be Overloaded?.....	417
Operator Overloading Examples, The this Keyword.....	418
Inheritance	422
Single Class Inheritance.....	423
Multiple Inheritance	424
Polymorphism, Virtual Functions, and Abstract Classes.....	426
Pointer to Base Class.....	427
Virtual Functions.....	428
Pure Virtual Functions, Abstract Classes.....	430
Final Remarks.....	433

Chapter 17: Classes and Objects for Torque.....	434
Chapter Introduction	434
Creating a new Object Type.....	434
Creating a New Game Object: The Questions	435
Getting Started: Building an Object in C++ for Torque	435
Object Fields, Implementing TorqueScript Variables	440
Networking Your Object	446
Enumerations.....	446
Getting your new Object “Network Ready”.....	448
BitStreams, Introduction to Networking.....	451
Rendering in C++.....	454
Shape Rendering: Model to Game.....	454
Mesh Rendering: Geometry to Game.....	460
Beam Rendering: Lasers, Lightning, Etc.	466
Rendering: Quick Conclusion Points	471
Collision in C++.....	471
Bounding Boxes.....	472
Advanced Collision Notes.....	474
Processing Time Events.....	475
Final Remarks.....	476
Chapter 18: Some Final Goodies.....	477
NetEvents, sending data between server & client without objects	477
NetEvent: Basic Example.....	478
NetEvent Macros	480
Ghosting in C++	481
Functions for Ghosting.....	481
Ghosting Guidelines	483
Getting an external C++ Library into the Engine.....	484
What is a Library?	484
Using a Preprocessor to include a Library.....	485
Configuring MSVS to include a Library.....	485
Resolving Linker Errors.....	486

Entry Point Revisit	487
General Guidelines for Engine Modifications	489
Final Remarks.....	490
Chapter 19: Packaging for a Release.....	492
Configuring the Shipping Build.....	492
Compiling TorqueScript	493
Removing Source Assets	495
Final Steps	495
Chapter 20: Conclusion, Final Remarks	497
Thanks!.....	497
And that's all... or is it?.....	497
Appendix I: Torque-Script Quick Reference.....	498
Appendix II: Quick C++ Reference For Torque	502
Appendix III: Binary Numbers & Operations.....	507
Appendix IV: Online Resources.....	508

Chapter 1: Welcome to Torque!

So, let's kick things off here. First and foremost, welcome to Torque 3D! Torque is a game engine technology that has been around for quite a period of time and each new version has brought a set of features and tools built for one purpose, to help you design a 3D world environment that numerous players could join together and explore together. Now, before we continue with this document, I want to bring to attention some common misconceptions about this game engine that have unfortunately been painted into the minds of many by the other game engines out there. The first big one is that Torque can only make FPS games, and that is completely untrue. Torque is built with FPS games in mind, however, you will learn (hopefully) here how to adapt the codebase of the engine to create games of different genres and play styles. The second misconception is that Torque is incapable of generating 'X' graphics. I'm not sure who came up with this one, but, graphics and game engines rarely touch each other aside from a few lines of DX / OpenGL code, the graphics department, aka the art department is responsible for creating those stunning visual effects in your games. And the last big misconception that has been painted is that Torque is slow. While this will seem true to you while developing a game, you need to realize, when you're making a game project, most of the time, you will be in a debugging mode of some form, which gives you editor toolsets and debugging symbols (we'll get there), and these things actually put quite a performance load on your computer causing the misconception of being "slow". When you actually finish a project, you'll use what is called a Shipping Build to optimize everything, and you'll notice things running much more smoothly.



Now, if I haven't bored you half to sleep with my introductory paragraph there, I'd like to give you a quick overview of what we'll be covering. I'm going to start at a "beginners" level and work my way up slowly in difficulty, covering all forms of topics regarding the engine. This should serve to you as an amazing resource in TorqueScript as well as C++, since we'll be using both of those here. I'm also going to walk you through some things such as creating GUI dialogs, wiring objects together, client/server transmissions and much more. So, this will be the "shortest" chapter here, let's get going!

Chapter 2: Setting Up, Getting the Files & Compiling

Alright, so before you can do any work with Torque, you actually need, Torque. Torque 3D is now completely free technology, released under the MIT license, which basically states that you can do pretty much anything you want with the engine, and even that would be an understatement. Anyways, navigate to the GitHub page here: <https://github.com/GarageGames/Torque3D>. So, the first thing you're going to learn from me is how to use Git.

Git Introduction

Git is a source code management and distributed revision control service. In layman terms, this is basically a piece of software that stores change information, or basically, the local history of a file, for all of the files in a project, and stores this in a central file called a repository. Git also contains a few extra tools along the side to help you do some pretty cool things when working in development teams and groups for projects.

For our purposes here, you won't need to learn much right now, but I have a little bit towards the end that will come back to the topic of Git for when you actually have code you're ready to publish. Anyways, for the current moment, we have a few options here, and a few things that need explaining.

First, we need to talk about branches. So, as I stated before, imagine Git as a central database containing the history of every file in the engine. A branch is the current line of changes between "versions" of your program. So, let's use the default Git model to explain this topic. Picture two branches, one of them named 'Master', and the other 'Development'. The 'Master' branch contains the state of the files relating to the version you currently have released to the public, or basically the current version of the engine. The 'Development' branch, on the other hand, contains a state of files that exists after the state of 'Master', or basically, what we're currently working on. To imagine this from a conceptual model, think of a tree, you have the "Root", or the main part of the tree, which would be the 'Master', and then branches extending from the tree, or 'Development', basically saying that Development bases from Master. In the reality of Git, you can have even more branches expanding from these branches, or use a completely different model, but for the purposes of Torque, we have Master and Development.

Next, we'll talk about the files themselves. So, as I stated before, Git stores the history of files. Now, it wouldn't be efficient to store the full history of every single file, every time a change is made, there would be millions upon millions of entries into the database; so instead, Git uses the concept of 'Commits'. A commit, is basically a compilation of every change up to the current point in time from the previous commit. When using Git, these commits are pushed to the storage location, which is then parsed into the file history.

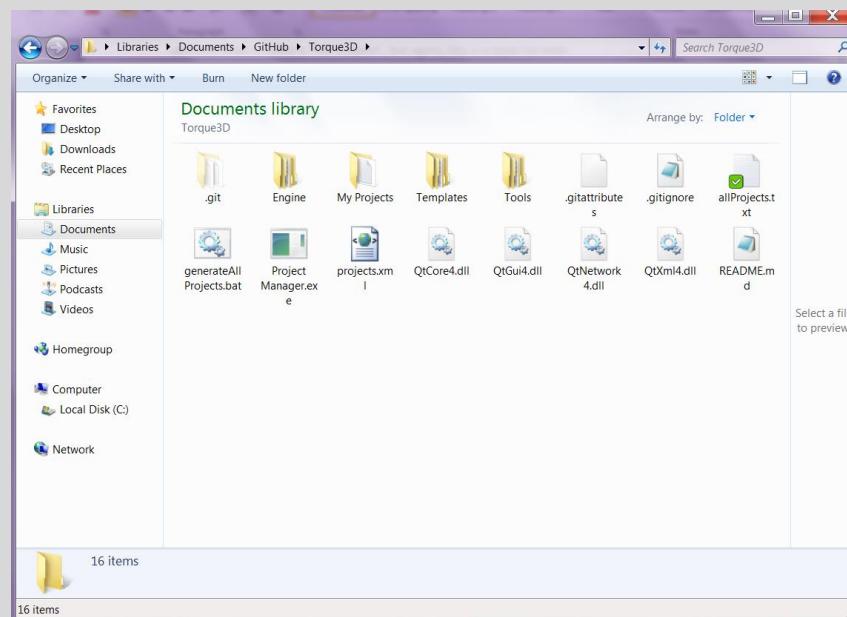
Now we'll combine the topics. Each branch has a layer of associated commits attached to it. The moment a branch is created, it takes the exact state of the files as they exist now (the current commit)

as the “base” of the branch, at which point it begins to expand outwards in time. Doing so can lead to a branch being “ahead” or “behind” a certain number of commits at any point in time. At some point however in our model, ‘Development’ needs to join together with ‘Master’ again to become the new official version of the engine. This is called a merge, and we’ll get to that later on.

Getting the Files

For now, we’re ready to grab Torque 3D. On that page, you’ll see the main readme listed as well as the source content of the engine. This readme contains some valuable information regarding setting up a project and how to compile everything, but don’t worry, you’ve got me to cover that information for you as well. What we want from this page is one of two things, you’ll either want to download a ZIP archive of the current codebase, or if you’re feeling brave enough to enter the world of computers, download the GitHub client, create an account, and Clone the repository into your computer (which also means downloading the code).

Now, we’re going to need a piece of software that is responsible for compiling C++ code. The one we’re using here is called Microsoft Visual Studio 2010 (<http://www.visualstudio.com/en-us/downloads#d-2010-express>). Make sure to register the software once you get it, they’ll give you a “product key”, even though the software is 100% free to use, once everything is installed and downloaded, you’ll want to set up Torque 3D. If you downloaded a zip archive, this means unzipping the files to a folder somewhere easy to remember, and if you cloned the repository from GitHub, simply go to that folder where the files were saved.



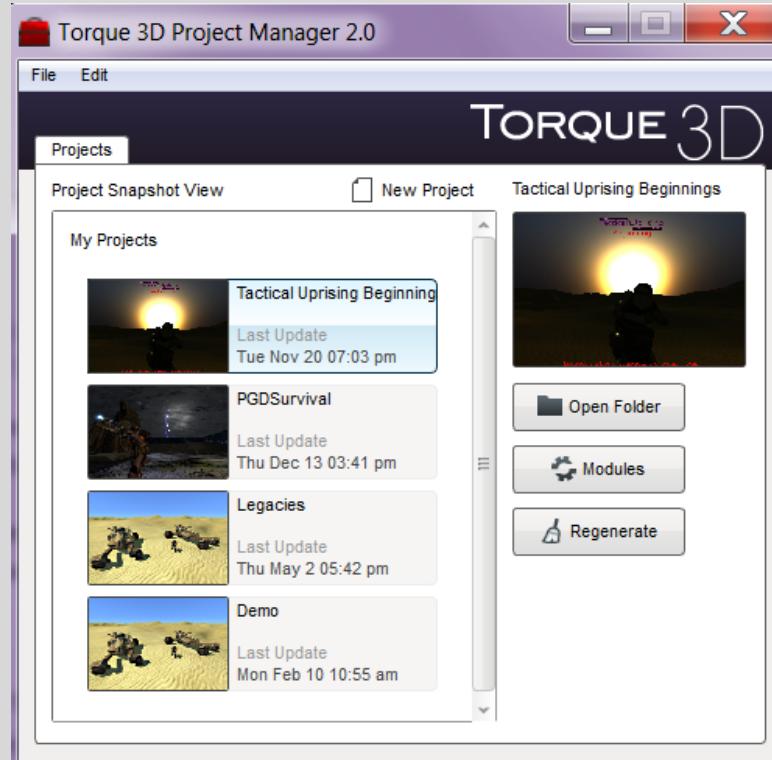
What You Should Have

The Torque 3D Project Manager

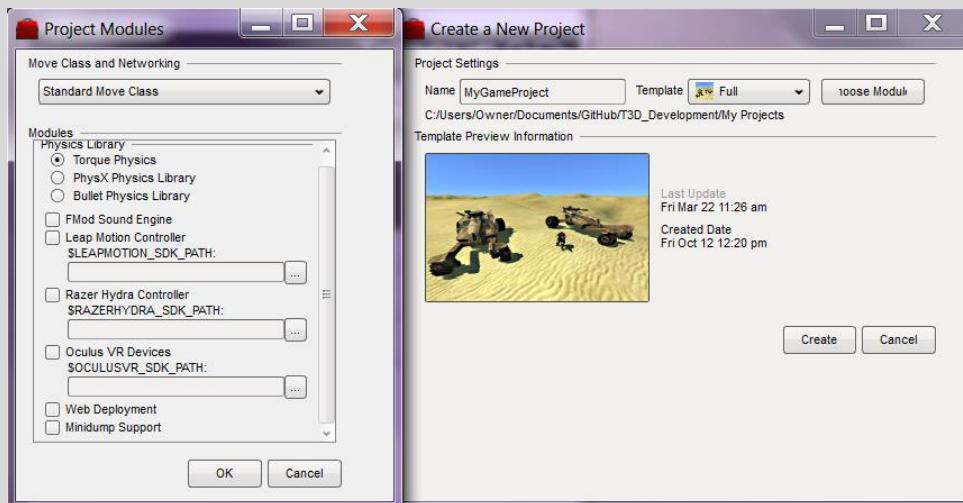
Now, since most people are visually oriented, I’m going to help out a lot of people here by skipping the instructions to manually create a new project in favor of using a tool created by GarageGames to make life easier for us all. <http://mit.garagegames.com/T3DProjectManager-2-0.zip>. All

you need to do is download the file and extract the contents to your Torque3D folder (for example, the one above in the picture).

Once you merge in the files to the Torque3D folder, run the file Project Manager.exe to get a new window that manages your projects:



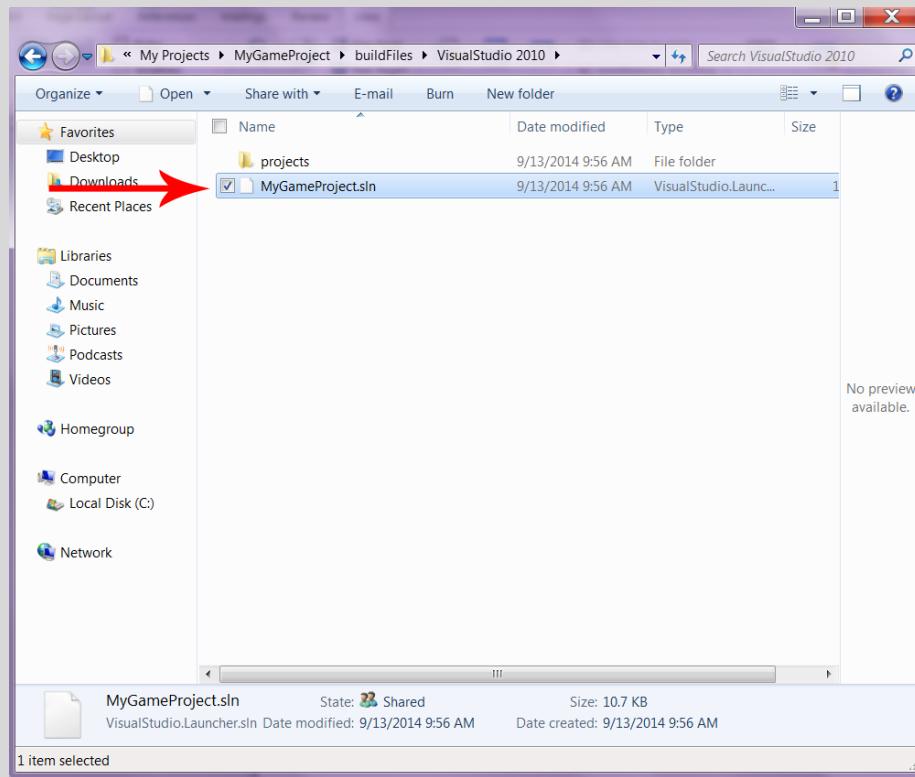
You'll want to click the button that says New Project, give your project a name and set it to the Full Template. You shouldn't need to adjust the modules the project uses unless you want to change your Physics library to use another instance, or enable devices such as the Leap Motion, and the Oculus Rift.



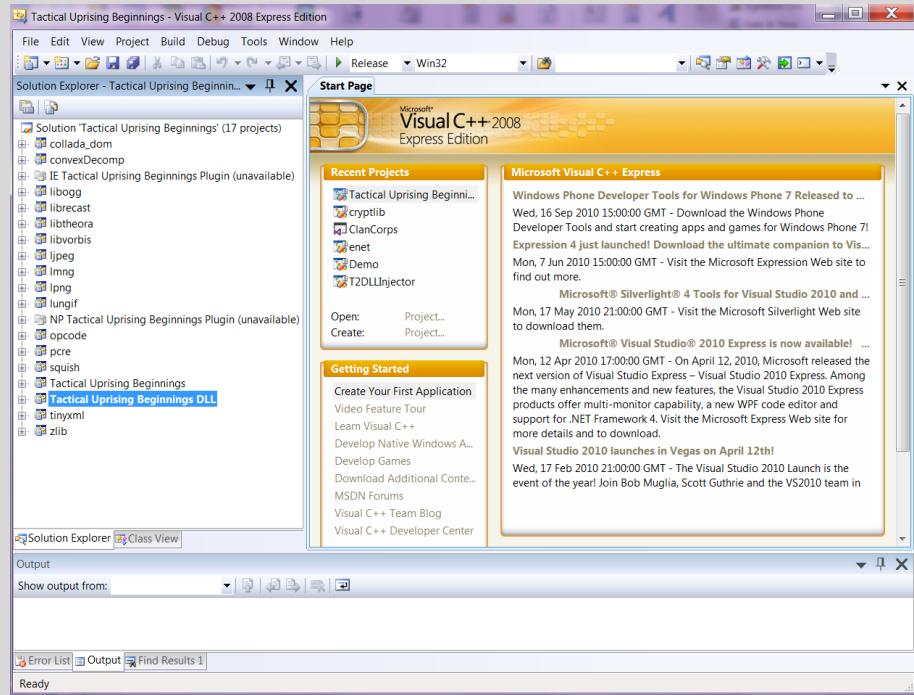
Once you set it all up, click the Create button to generate your project's files. The process of project generation will take 2 – 5 minutes depending on your computer, and any additional modules you may have selected. Once it's done click the Ok and Finished buttons to exit the Project Generator. Once it's generated, you'll see your project in the list of projects in the Project Manager, select it and click the Open Folder button.

Compiling the Engine

Inside that folder you will see a few folders, for now we want the folder named buildFiles, and then open the Visual Studio 2010 folder. You will see a .sln file in there, when you select this file, Microsoft Visual Studio 2010 will open.



You'll get a window that is somewhat similar to this one (I'm using MSVS 2008, so there will be some slight differences):



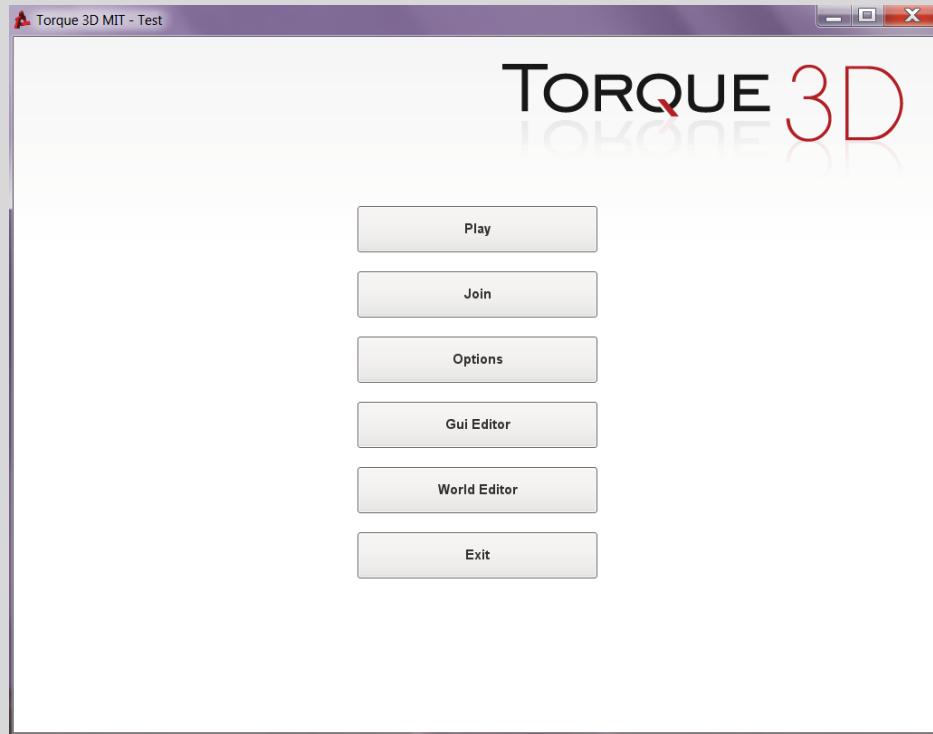
The first thing you need to do (if this is not already true on your version of the engine) is right click the projects with the prepend tags IE and NP and select the **unload project** option, since these features are no longer supported by the engine. Before you can actually compile the engine however, you need to set up Microsoft Visual Studio to be loaded with the necessary C++ files that are required to compile the engine. To perform this setup process, follow the guide for your version of Visual Studio here: <http://docs.garagegames.com/torque-3d/official/index.html?content/documentation/Setup/Overview.html>

Once you complete the setup for the files, select the dropdown box that says the word Debug on it and select the option that says Release, and then hit the F7 key on your keyboard to initialize the build process. This will take anywhere between 10 – 30 minutes depending on your computer’s memory capacity. Once it’s finished, you can press Ctrl + F5 to launch the game project. This process you have just completed is called compiling the engine. Basically, it takes C++ source code and converts it into machine code, which is code that a computer recognizes. There are numerous settings and options in Visual Studio, but we’ll cover Visual Studio later on when we get into the advanced topics and C++ in general. For now, everything is ready to go, and you’re now all set to begin working with Torque 3D!

Chapter 3: The World Editor

So, now that the engine has been compiled, you're ready to get working on your first game project. The thing to note here is that when you compiled the engine, you actually directly compiled your game's project for the first time, in the event you add or change C++ code in the engine, you'll need to perform the compile step again for the engine to notice the changes. However, this chapter is not about C++, or even programming for that manner, we're going to talk about a very useful and powerful tool in the engine called the World Editor. As you've probably taken home from that little name there, the World Editor is responsible for creating the World; however, more common developers will recognize this as the tool to create their levels.

If you haven't done so yet, launch the engine executable you just compiled from the prior chapter. Once you get through the load screens you'll reach the main menu of the game, which will appear as the following:



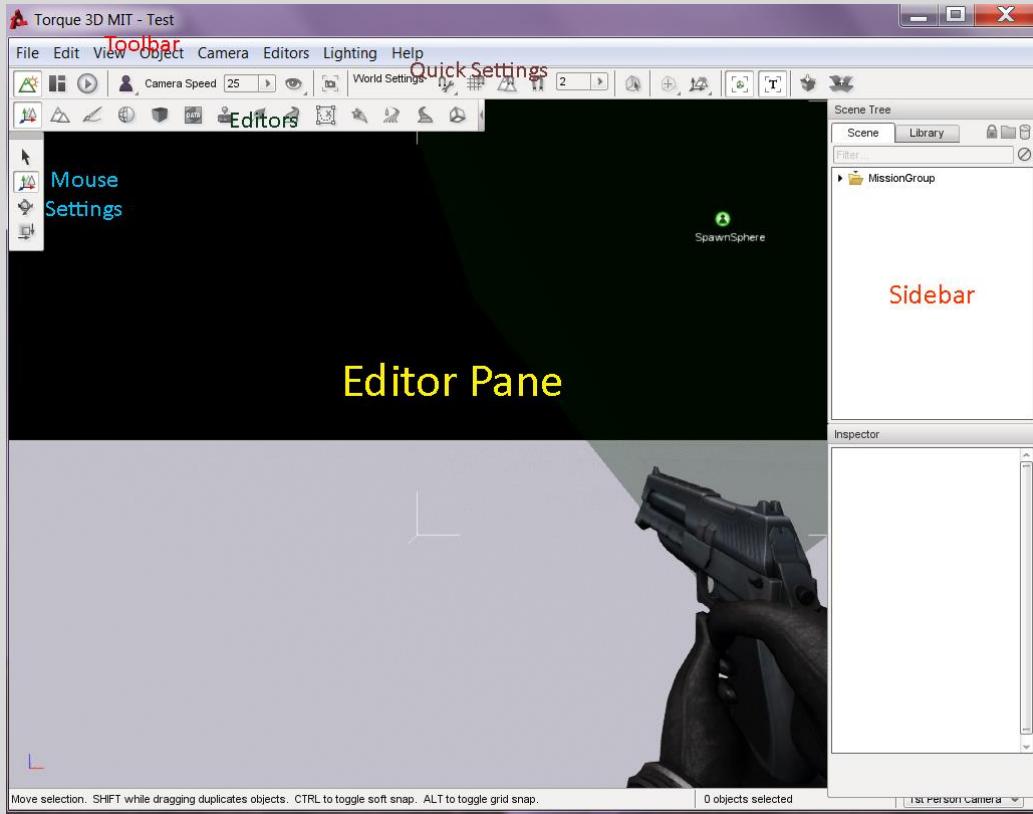
We'll modify the look of this menu in the next chapter, but for now, let's go ahead and start up the world editor. There are three ways to load the world editor from the main menu. The first and obvious way is to click the big button that says World Editor, and then load the level you'd like to edit. The second way is to actually directly load the level by means of the Play button, and press F11 once in the game. And the last way is to press F11 at this menu and select the level you want to edit. So, select the World Editor, then select the Empty Room map and click the go button to load the map.



This is the map we want

You'll likely notice the loading of the engine for the first time takes a bit, so while that's going on, let me explain how the loading process works for Torque 3D. The first thing the engine does is it loads the mission file itself, this detects which objects and resources are needed by the engine to be loaded, this includes the loading process for all of your model files, sound files, and texture files. For model files, the engine loads models (.DAE, .DTS) and converts them all to a format it understands, which is the .DTS format, or Dynamix Three Space. This model format has been used by every iteration of the Torque Game Engine since its original release. Sound & Texture files are left as they are. Once this is handled, the engine then loads the resources and objects needed by all of the missions, such as your players, weapons, etc. Like the process beforehand, this includes loading all of the necessary asset files the engine needs to create these in-game objects. The next thing the engine does is receive a list of datablocks from the server. A datablock is a persistent network object that cannot be manipulated once it is received, but we'll cover these later on. Once all of the datablocks are loaded, the server sends the initial state packet to the player so they know where everything is on the map, and then the game begins for the player.

Once the loading is completed, you should spawn in a big empty world with a black sky and the World Editor will load for you. If everything went well for you, you should end up with a screen that looks like the one I have presented below:



Now, the world editor itself is split into 14 different modes, each of these modes are responsible for allowing you to manipulate parts of the map, and others are responsible for controlling your assets. To make things a bit easy, I'll cover each one of these individual editors in a sub-chapter. First, let's cover the toolbar at the top of the editor so you can learn some keyboard shortcuts and how these options work.

The World Editor Toolbar

The toolbar for the world editor consists of the options along the top of the screen.



We'll start at the left and work our way to the right here. In the File menu, we have most of our common options for running the level. The first option is *New Level*, which as it states, creates a brand new level file. In the case of the engine, all this does is load up the mission file you're currently in, since *Empty Level* is the "base" starting point level of the engine. The next option is *Open Level*, and this has a keyboard shortcut of (Ctrl + O). This option allows you to select a mission file and load it in the editor. The next two options are there for saving the level. The first is the generic *Save Level* option, which has a keyboard shortcut of (Ctrl + S). Use this option to quickly save the level to the current file. If you want to create a new file, use the *Save Level As* option. The next two options are for Torsion, which is a Torque Script editor for Torque 3D, since this is a generic overview of the engine without addons, we'll ignore

these two options. The next option is *Create Blank Terrain*, which allows you to create a flat terrain object for your map. We'll get to that option in a little while when I cover the terrain editor. The next two options are very useful options to import and export what are called heightmaps. These work on the premises of your standard GIS contour maps where certain colors resemble higher spots and others resemble lower spots. Those two options are a little out of the scope of this guide; however you're more than welcome to thoroughly explore them on your own time. The next option is the *Export to COLLADA* option, which exports the entire map to a COLLADA (.DAE) model file that you may then edit in a program like Blender. The next option is *Add FMOD Designer Audio*. This option is for developers who have a license to the FMOD sound library and would like to use it in their project. You'll use this option to add the sound files and specify their parameters here. And lastly, we have options to exit the world editor, exit the current level, and to exit the game application.

The next menu is the Edit menu. In this menu, you have some standard option editing tools such as *Undo* (Ctrl + Z), *Redo* (Ctrl + Y), *Cut* (Ctrl + X), *Copy* (Ctrl + C), *Paste* (Ctrl + V), *Delete* (Del), *Deselect* (X), and *Select*. The remaining tools are editor settings and a few managers to manipulate parameters and settings of some of the engine's core features.

Under the next menu, View, we have two options to control what you see in the editor. This can be useful from time to time to debug some things on your level that may be causing some problems.

The next menu is the Object menu. This menu controls editing and feature mechanics on the individual objects in your level. The first two options are the lock (Ctrl + L) and unlock (Shift + Ctrl + L) options which are used to prevent the editor from being able to manipulate an object. When you're mass selecting objects, this is a great tool to use to omit certain objects from the selection. After these objects, you have the hide (Ctrl + H) and show (Shift + Ctrl + H) options which can turn objects invisible and visible in the editor. The next two options are for aligning an object to the selected parameter. The next four tools have to do with manipulating the object's transformation, or the world position and rotation of the object. The first option, *Reset Transforms* (Ctrl + R), resets the object to its original transformation. The next two options are there to reset just the rotation and the scale of the object without touching other transformation parameters. The last option is *Transform Selection* (Ctrl + T) and is used to manipulate the transformation parameters of the object. Next up is the *Drop Selection* (Ctrl + D) command, which is used to drop the object at the specified position. This is a good tool to test gravity parameters as well as mechanics such as fall damage. The next two commands have to do with Prefabs, which are out of scope of this guide, feel free to explore if you want. Finally, we have the mount and unmount commands which can be used to test the mounting of objects on other objects.

The next menu is the Camera menu. This menu has options and settings for your camera, which is what you are using to see into the world itself. The first two options in the menu allow you to toggle between a world camera, which allows you to move around in a no-clip style camera, and the player camera, which is attached to the player object in either first person or third person. The *Toggle Camera* (Alt + C) option is an extremely useful tool that allows you to quickly move between the world camera and the player camera. The option *Place Camera at Selection* (Ctrl + Q) allows you to have the camera jump to the selected object instance, and the option *Place Camera at Player* (Alt + Q) places your

camera on the associated player object. The last option in this group *Place Player at Camera* (Alt + W) will teleport your player object to the location of your camera. The next two options will fit the camera to the current selected object (F) and the selected object and orbit setting (Alt + F). The next group of settings relates to the speed of your camera instance. There are 7 choices each having their own keyboard shortcut (Shift + Ctrl + 1 – 7) with 1 being the slowest and 7 being the fastest, use these options when moving around in your world camera. The next selection is the View tool, which uses options that are similar to a typical 3D modelling program. Select an object, then one of the options to move the camera to that view perspective (Alt + 1 - 8). The last three options are used to set camera bookmarks for the current level instance to quickly jump the camera to the position and rotation of the bookmark. You can add (Ctrl + B), manage (Shift + Ctrl + B) and jump to a bookmark.

The next menu is the Editors menu. I'm not going to cover this one here, as all this does is switch between the 14 modes of the world editor, and I'll cover all of them in more detail below. Just know that you can use F1 – F9 to move between the first 9 modes of the editor.

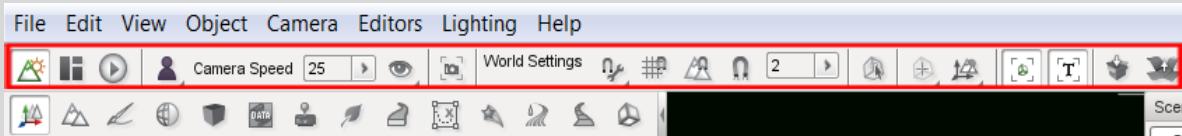
The next menu is the Lighting Menu. This menu controls how light is handled in your world instance. There isn't much to discuss here so I'll be quick about it. The first option is *Full Relight* (Alt + L). In prior versions of Torque (TGE / TGEA), you'd use this option to generate a lightmap for your current level. You only need this if you're in Basic Lighting to generate the lightmap for the level, if you're in Advanced Lighting, this option may be ignored. ShadowViz is a bit beyond the perspective of this guide, so you can explore that on your own time. And then lastly, you have the options to toggle between Basic and Advanced lighting.

The final menu is the Help menu, which is your quick shortcut list to things such as the Torque 3D documentation, the online documentation, and the engine's forums.

So that covers all of the toolbar options for the World Editor. The next thing we need to talk about for the editor is the quick settings bar.

The Quick Settings Bar

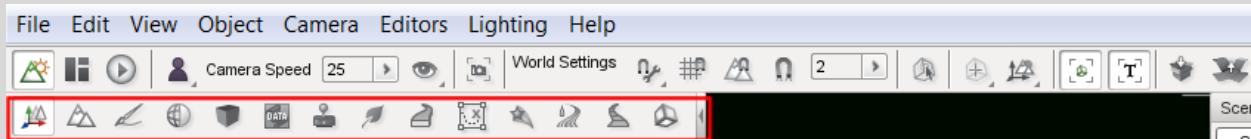
The Quick Settings Bar is the list of options and controls that can be quickly accessed by a mouse click for each of the 14 different modes. This bar will change for each of the modes you select, so be sure to keep an eye on that.



Since this bar changes with each of the 14 modes of the editor, I'm not going to discuss them in this part of the chapter; I'll discuss them as they are relevant for each of the editors. So let's start talking about these editors now.

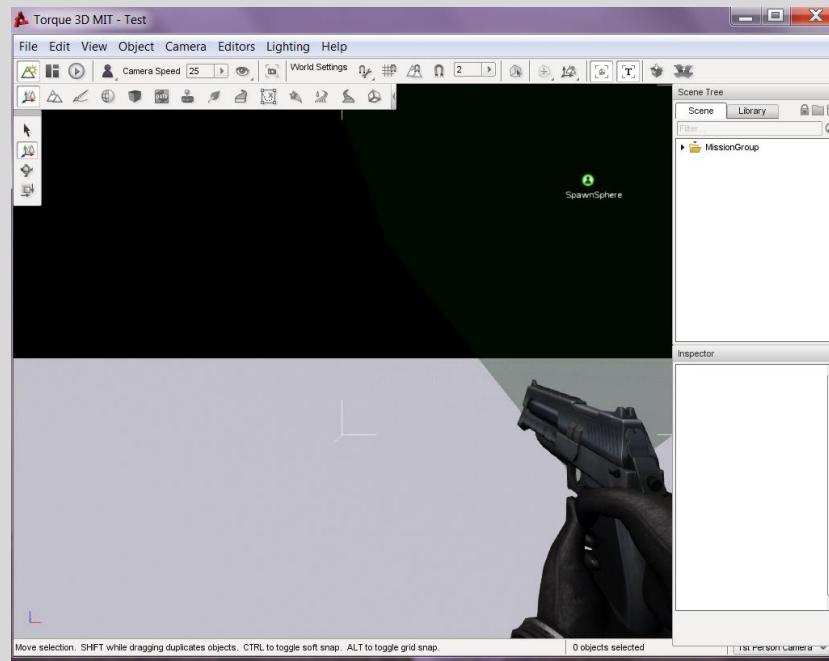
The World Editors

The Torque 3D World Editor is split into 14 different modes. Each of these modes contains a set of tools and category of options that are used to manipulate parts of the level, and to edit individual assets of your game, such as models, and particle effects.



Object Editor (F1)

The Object Editor is the first editor available in the World Editor. This is the editor you will likely spend most of your time with when you're working on a level. This editor's main purpose is to select objects and change properties of the object on the fly. When you select the object editor, this is what the editor will look like:



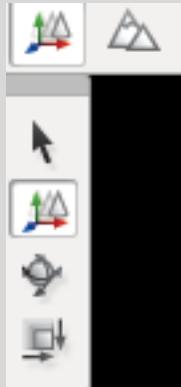
The first thing we'll discuss is the options for the quick settings bar for the Object Editor. Here is how it will be displayed to you:



Now, from left to right, we'll discuss the options here. The first is a non-changing button instance that may be ignored. It's basically a toggle button for the world editor itself that remains active. The second button opens the GUI Editor, and we'll cover the GUI Editor in Chapter 4. The last button there closes the world editor so you may return to your game instance. The next option is the camera

toggle button, which is a little fancier than the one we discussed in the Camera options before, this has a list of buttons you may select from. The next item on the toolbar is a scrollbar to control the speed of your camera; you can also type a number between 1 and 200 in the box. The eye icon is the option to toggle visibility of objects based on type and settings, this is mainly an editor debugging tool to locate faults in your map. The camera icon surrounded by a box is the *Fit Camera to Selection* option we briefly discussed earlier. The next five settings deal with object snapping parameters. This allows you to ‘Snap’ an object to numerous different settings when manipulating the transform of the object in the editor itself. Beyond these snapping options are a few basic object manipulation parameters. I’m not going to discuss them here as they should be rather straightforward, and as mentioned earlier, this guide doesn’t talk about Prefabs.

The next thing to talk about in the object editor is the mouse control pane, or basically what your actions with the mouse will do when it’s working in the editor pane.



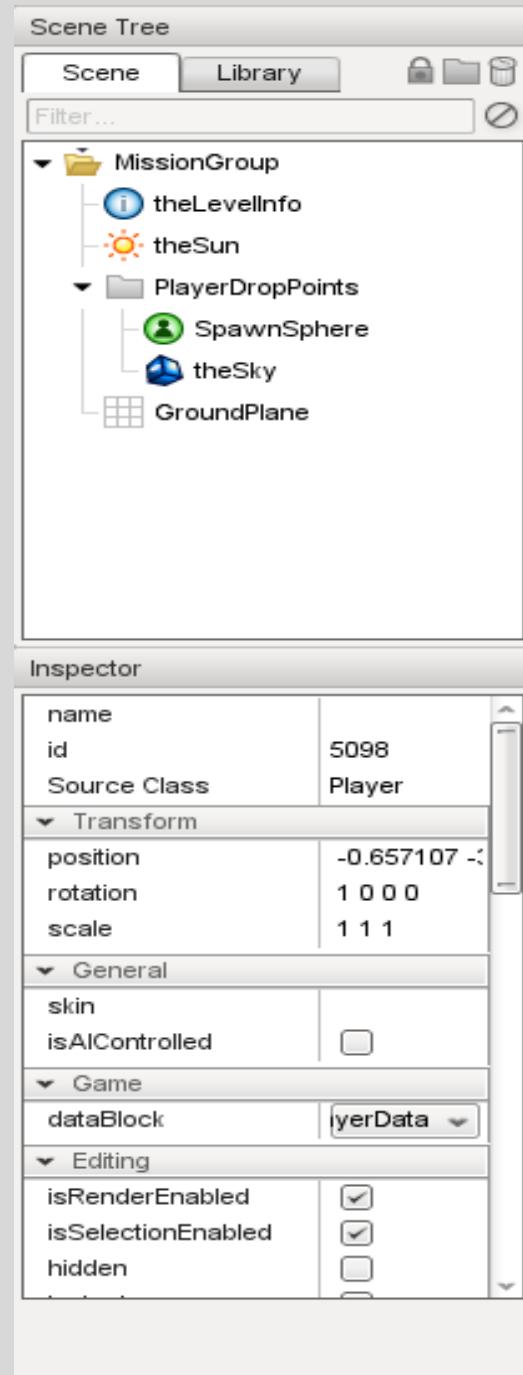
This is the mouse control pane for the Object Editor of the World Editor. You can use the keyboard shortcuts of 1 – 4 to quickly select these commands to work with. The first control is the Selector control. You can use this to directly select objects without manipulating their transformation parameters. The second option is the Move control which allows you to edit the position of an object. The third option is the Rotate control which allows you to edit the rotation of an object, and the last option is the Scale control which allows you to scale the object along the specified axis. When you select an object in the editor pane, it will change cursor selectors based on these settings here.

On the right side of the editor pane, we have the editor sidebar. For the case of the Object Editor, we have two tools of interest. The things of interest here are your Scene Tree and Object Inspector tools. The Scene Tree allows you to directly select objects that are placed by means of the editor, or are flagged as “part of the map”, objects such as players, weapons, and projectiles will not appear here unless placed by the editor. This is also where you can select an object based on the object itself to place on the map. The Inspector is a one stop shop for quickly editing an object’s fields and properties without needing to touch a single line of code. Now, let’s have a closer look at these two extremely useful tools and go into a bit more detail on how to use them to the best of their abilities.

Let's start by going into more depth for the Scene Tree. As you can see here, everything is placed into Groups of objects, in our case, the base is called MissionGroup. You can create, lock, and remove these groups with the icons to the right of the Library tab. From here you can also quick-search objects by filtering them by name. For now, I'm not going to go into these items in detail, just know that selecting an object brings up its details in the Inspector below.

From the Library tab, you can select objects that are saved in the engine based on their type and categories and then double click to spawn them. These objects will automatically be added to the MissionGroup allowing for direct editing through the Inspector. Some objects will require some input parameters and a popup box will appear for you to fill in their needed parameters in order to spawn the object.

The next topic is the Object Inspector. The Object Inspector tool is a very powerful tool that may be used to directly edit the fields and properties of an object and quickly see the changes directly in the editor. When you select an object in the editor, the inspector will fill with the properties of the object, for more information, we'll cover how this is handled in Chapters 17 and 20 when we dig deep into the actual mechanics of the engine. One of the more powerful tools of the inspector is to manipulate a Dynamic Field, or basically a field that you define for the object. When we get into scripting and writing scripts for object, you'll understand this section a little more and why it's important as a quick debugging tool to ensure your code is doing what it needs to be doing.



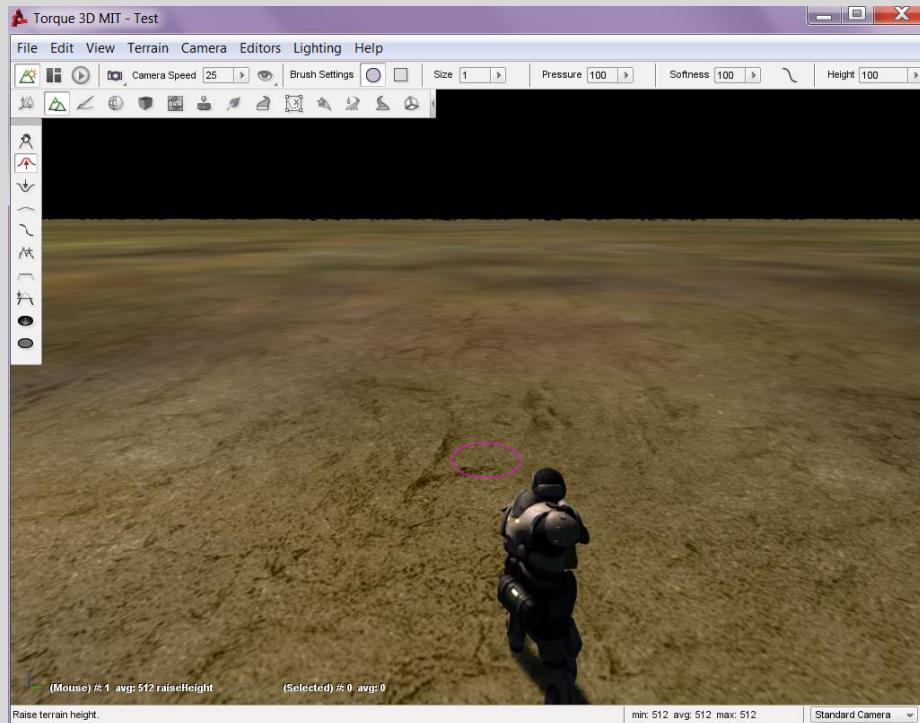
Now that we've finished up with the sidebar, let's talk about the important part of the Object Editor, which is the Editor Pane. When you're in the Object Editor, the editor pane will show you your world and there will be icons over the world objects showing you where objects are positioned on the map. You can left click any object to select it, right click to rotate your camera around, and even mass select objects by clicking a point, then holding the mouse and dragging a selection box around the

objects you want to manipulate. When your camera is in player mode, your player can move normally around the map with the editor open, which is a good way to debug objects and regions of the map to ensure they are behaving as intended. When you are in the world mode for your camera, the movement keys make the camera move in no-clip mode through the world for quick movement and manipulation of the level.

So, that's about all you need to know about the Object Editor to get your feet wet in Torque 3D. We'll come back to this editor in a little while when we actually start playing around with these tools, but for now, let's talk about the next editor.

Terrain Editor (F2)

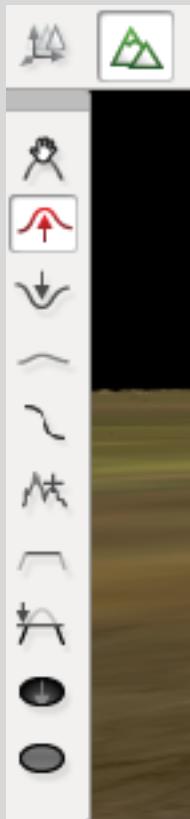
The Terrain Editor is the next editor you have access to. In order to load this editor, your map must have a terrain object to work with. You can either select this option to allow the terrain editor to create it for you, or you may select the *Create Blank Terrain* option in the File menu of the editor toolbar. When you select this option, it will ask you for some options. The first is the Name. This is mainly used by the Object Editor and the Inspector, it just has to be something unique (IE: Not already used by the mission). The next choice is the Material option, which defines what the terrain actually looks like, this is a texture. You can load in your custom settings and options with the Terrain Material Editor, and we'll cover that in a little while. The Resolution of the terrain is a parameter of the texture of the image; a bigger number means a bigger terrain and so forth. Lastly, you can either choose to create a big flat terrain, or select the Noise option to pre-load in some randomly generated hills, mountains, etc. Once you have a terrain, you can select the Terrain Editor.



For the purpose of this tutorial, I went ahead and deleted the ground pane object in the Object Editor so only the terrain would be visible here. As we did last time with the Object Editor, let's start by talking about the Quick Settings bar.



We covered the first six last time when we were in the Object Editor and therefore don't need to cover them again. The first two icons in our interest are the brush settings options, which are the Circle Brush (V) and the Box Brush (B), which controls what type of editor brush you will have in the Editor Pane. The next option is the size option, which is very self-explanatory. This is the size of your brush. The Pressure of the brush is how much of the specified operation is applied per second. A Higher number here means options like Raise Terrain will be applied much more quickly. The next option is called Softness, and how it works is defines how much of the specified terrain curve must be applied during an operation. A Larger value here means the curve will be sharper, and a smaller value means the curve will be smoother. The curve icon allows you to manipulate the actual curve itself. I highly advise actually experimenting around with this tool, as you can create some drastic terrain effects with it. The last option has to do with the Set Height tool, and this specifies the height value to use. The Terrain Editor only has one other important list of commands, and those are located in the mouse control pane. These are tools of the Terrain Editor used in the editor pane.

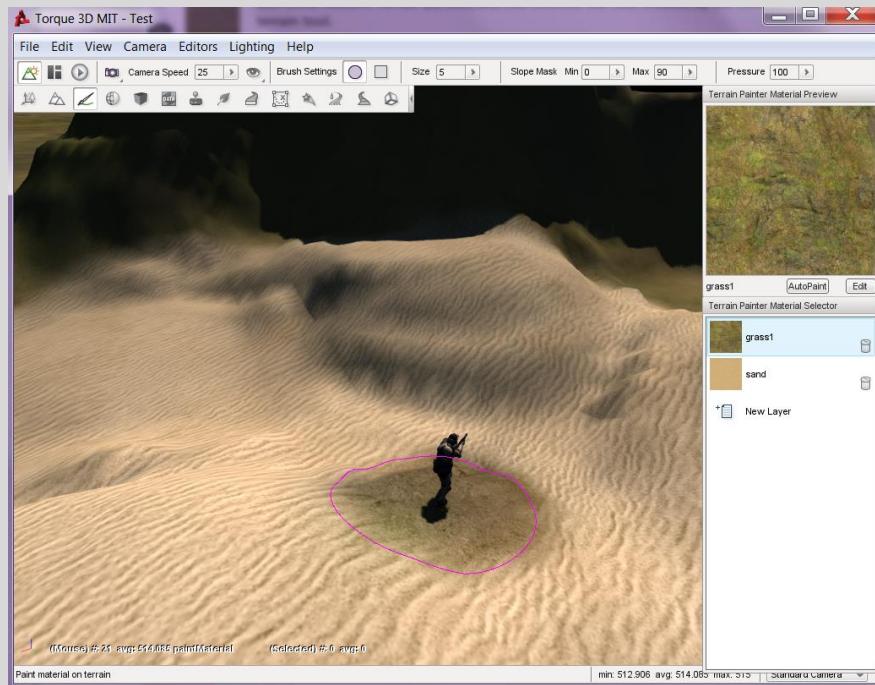


The first tool in this list is the Grab Terrain (1) tool. When selected you can left click and hold terrain and move your mouse around to manipulate the terrain either up or down. The second tool is the Raise Height (2) tool, which as it states will raise the terrain. You can click and hold on a spot to continually raise the terrain to the map's height limit. The Lower Height (3) tool follows this, and does the opposite of the Raise Terrain tool. The Smooth (4) tool is the next tool, and how it works is you click and hold over terrain and then drag over another terrain area to smooth the terrain. A lower Softness value will apply less, and vice versa. The Smooth Slope (5) tool is the next tool and this tool is much more effective to smooth out steep slopes created by the raise and set height tools so players may walk on them. The Paint Noise (6) tool follows this tool, and when you use this tool, it will randomly apply raise and lower to every grid point in your brush. The Flatten (7) tool works like the Smooth tool on maximum, immediately turning a surface flat. After the flatten tool is the Set Height (8) tool, which instantly moves all of the terrain in the brush to the specified height in the quick settings bar. This will create very steep slopes that are likely impassable by players, so be sure to smooth it out with the smooth slope tool. The last two tools are special tools. The first tool is the Clear Terrain (9) tool, and this allows you to axe out part of the terrain and make it completely open. When placing objects that have underground segments, use this tool on the terrain where the entrance is. The last tool is the Restore Terrain (0) tool, and this reverses the hole created by the clear terrain tool.

The editor pane for the Terrain Editor is extremely easy to work with. When you move your camera and mouse around, you will see a purple cursor over the selected terrain and all you need to do is click to apply the changes set by your mouse control pane and the quick settings bar. The terrain editor is likely one of the first stops in your level editing process as it will help you set up the overall look of the map before you populate it with objects. Now, let's talk some more about the terrain, but how we can change it from just being a solid single-texture block.

Terrain Painter (F3)

The Terrain Painter is the third editor in the World Editor. This editor, like the Terrain Editor requires that there be a terrain object on the map. It will have you create one if you're missing it. This editor is used to texture parts of the terrain using different textures to create more realistic looking terrain. You can specify these textures with the Terrain Material Editor, and we'll cover that in a short period of time.



One thing you'll notice right away is there is no mouse control pane on the side, and the quick settings bar is almost identical to the one in the Terrain Editor with the exception of the Slope Mask parameter which is used to determine the minimum and maximum slope range where the terrain brush will be applied (for slopes), setting the minimum value to anything greater than 0.0 will result in flat surfaces being ignored by the terrain painter. With that in mind, the only topic of interest here is the sidebar.

On the sidebar for the Terrain Painter, you'll see a little preview image of the current material selected for the painter. These materials are controlled by a tool called the Terrain Material Editor, and we'll get there in just a moment, but before we do that, let's talk briefly about some of the controls of

the sidebar. There are two buttons you can click. The first one is the AutoPaint tool, which appears like this:

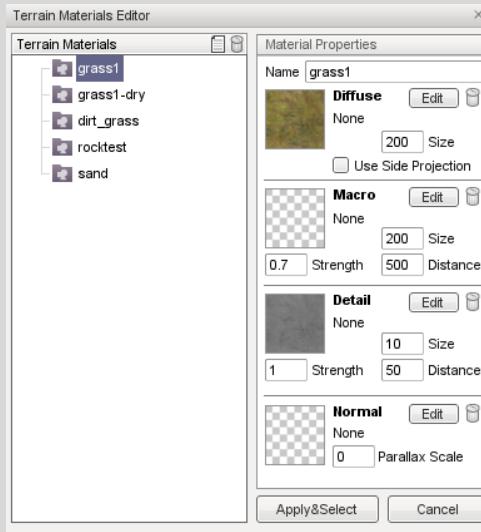


The controls on this tool are extremely easy to use. You define two height parameters, a minimum and a maximum, which uses the heights of the terrain as specified by the Z-Transformation coordinate of the terrain object (Object Inspector) plus or minus the amount of height on any given point in the terrain. And the slope parameters define the minimum and maximum slope required for a specific point of the terrain to be painted by the tool. If the points are inside both ranges when the Generate button is pressed, it will be painted with the selected material.

And if you click the Edit button, you'll open the Terrain Material Editor. Before we move onto that topic, let's quickly discuss the pane below these buttons. When you use the terrain painter for a level, it defines each individual material as a layer (kind of like how photoshop works). You can then click on these layers to select the material to paint to the terrain, add a new layer, or remove a layer from the terrain. Do note that when a layer is removed, its associated painted regions are replaced with the next available material, and if no materials are available, an orange grid material (default) is selected. So, with those topics covered, it's time to talk about how we actually get these materials.

The Terrain Material Editor

The Terrain Material editor is a little subset editor located within the Terrain Painter editor. These materials are treated much differently than default engine materials which we will cover in the Material Editor in a short period of time, but they use similar concepts to get the job done.



The previous picture shows the Terrain Material editor when you open it up. On the left side, you have a list of the specified materials in the engine. You can either add a new material, or delete an existing material through this dialog. When you select one of the materials in the list, the right side of the editor shows the individual material maps for the selected terrain material. For these materials, there are four different texture maps to be aware of. The first section is the *Diffuse* section.

A Diffuse map is a texture map that contains the base color and appearance of a material. The size option controls the physical size (or world-space size) in meters, of the material as it appears on the map. A smaller number will result in a higher texture quality. The option to use side projection on the material will use a blending technique to replace the standard top-down approach which can cause the material to appear to be stretched from the view perspective.

For most purposes and the purpose of this guide, we're going to ignore the Macro option, just know that it's there to apply a secondary blending of the diffuse map. You can treat the main diffuse map as the primary coloring of the material, and the Macro as an overlay, for example, if I wanted to create what appeared to be a mossy terrain over a rock surface, I would use a rock diffuse map, and some lightly grassy patches for the Macro option.

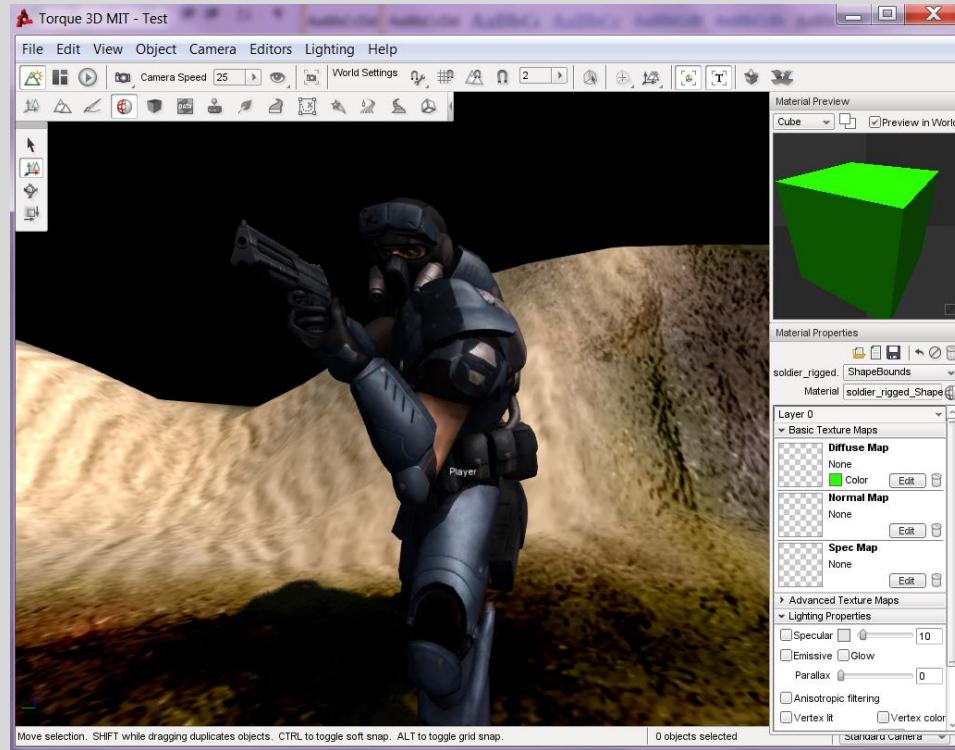
The Detail map is used to give the terrain a crisper look. This is a more advanced rendering technique used by many game engines out there today. It's pretty much a greyscale image with darker regions highlighting areas of more "important" textures; the engine then puts higher quality into these regions and lower quality into lighter colored regions. The size option specifies how close the camera needs to be to the terrain in order to produce this effect. The strength option is an overall multiplier to the effect where higher numbers result in greater contrasting resolutions, and the distance option is used to determine how much "bolding" or sharper detail contrasts appears on the default diffuse map.

The Normal map is used to "fake" changes in altitude on the terrain. It works similar to the Detail map, but it treats brighter colors as higher areas and darker colors as lower areas. You can use a normal map to make a terrain appear to have small ridges and troughs in the appearance such as a beach, or a sand dune, without actually manipulating the terrain itself. The Parallax Scale option is used to apply a scaling parameter to the effect of the normal map.

Whenever you make changes to any of the options, be sure to click the apply & select option to save the changes to the engine itself. Also, be aware that materials are global, so changing a material used for one map, may have consequences in another if you're not careful.

Material Editor (F4)

The next editor to talk about is the Material Editor. This editor has quite a few advanced topics in it, so we won't be covering too much here. But I'll explain most of it to show you how it works so you can at least get off the ground and running with it when the time comes to actually use the Material Editor.



Now, the first thing to notice about the material editor, is that it shares both the mouse controls and the quick settings bar options as the Object Editor, so we don't need to discuss anything new there. The important part of the Material Editor is the sidebar.

So, what is this editor used for? The Material Editor is used to create, and quickly edit texture maps that are used by game objects, and to be quite specific, the models that are associated with them. This editor takes texture maps, and binds them to the UV-Map generated by the model to actually make the texture appear on the model in the game. There are three key maps of interest, the *Diffuse*, the *Normal*, and the *Specular (Spec)* maps. The controls on this editor that we are interested are nearly identical to the Terrain Material Editor, but there are no fine tuning options, just the image selectors.

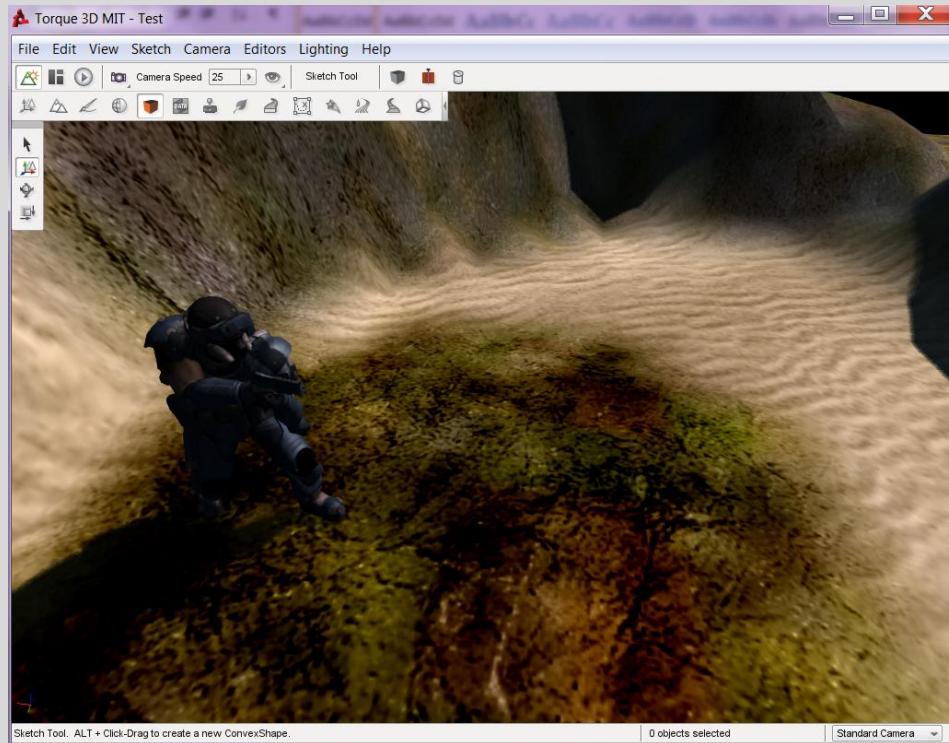
Like before, the Diffuse map is used to define the color and appearance of a texture on the object, and also like before, the Normal map is used to fake height effects on an object. This is a more advanced technique to lower what is known as a poly-count on the model, or basically how detailed a model is to the engine. Things with higher poly-counts will look a lot better on your screen, but it will eat your computer's performance like no tomorrow.

The only change here is what is called a Specular map. A Specular map is an advanced detailing map to define how "shiny" an object appears. It uses another bright/dark color scale where brighter areas appear shinier in light and darker areas don't have the luster effect. You will notice under the lighting properties area some more fine-tuned controls for the specular maps, and I invite you to actually try it out on an existing material some time to see how it works in the engine. The only thing of importance I have to tell you is to be cautious when naming materials, both in the engine, and on your models. I highly advise that you keep a naming scheme in mind so that model files don't end up fighting

over what material belongs to a certain model, it can cause some problems in the end. But for now, that's really all of the basics you need to know about the Material editor, let's move on to another important tool.

Sketch Tool (F5)

The next editor is a very small tool called the Sketch Tool. This is a tool that can create very quick and effective Convex Shapes by means of dragging your mouse around. This is a great tool to quickly draw up some shapes that can be used in the engine, apply some basic materials to it, and bypass the modelling process.

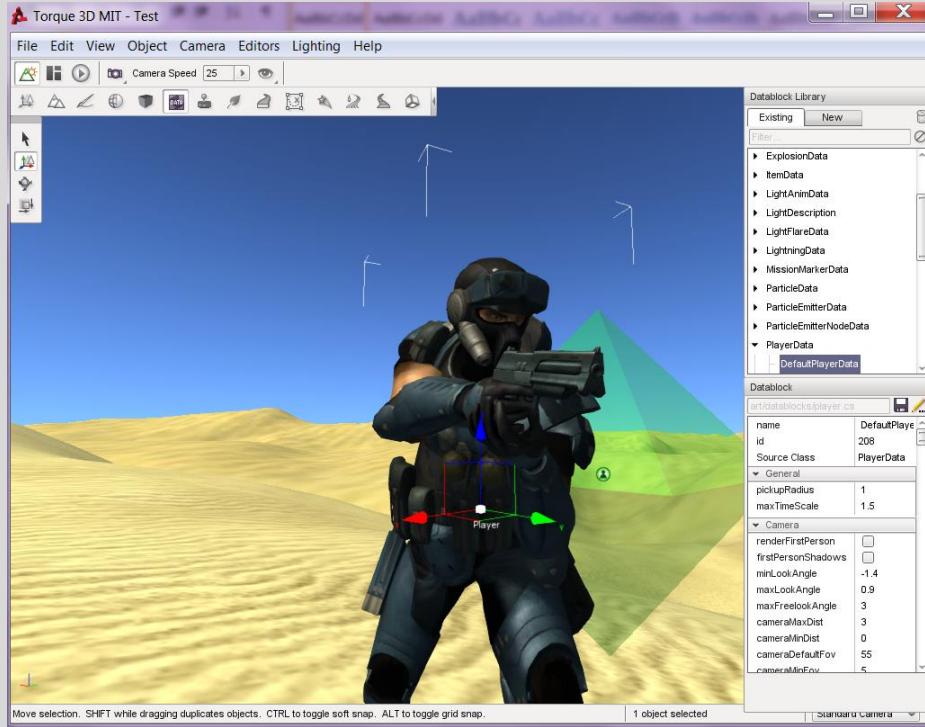


The Sketch Tool has a very basic overview in the editor, not even having a sidebar for editing. The only important part here is on the quick settings bar with three new options after our standard settings. The first button is a box icon, which actually spawns a Box Convex object in the world to edit with this tool. The second option is currently disabled in Torque 3D pending an update to fix the tool, but when it's done, it will give the option to split a selected part of the convex in half. The last option simply deletes the Convex.

The important part of our little tool however, comes in the little add-on to the toolbar in the Sketch option. The first option is the Hollow option, which hollows out the convex shape into six different convex shapes, allowing you to manipulate the individual parts of the convex. The other option is the re-center tool, which brings everything back to a common center point. By combining the Sketch tool with the Object Editor, you can create very useful objects on the map such as world barriers and quick structures.

Datablock Editor (F6)

The Datablock Editor is the sixth editor on our guide. As I briefly stated earlier, a datablock is a persistent piece of information that is stored on the server and cannot be changed in-game. This editor will allow you to create and manipulate these blocks of information. While this is a great tool to learn about these blocks, I prefer writing them by hand, and I'll show you how that's done in a little while, but for now, let's walk through this editor.

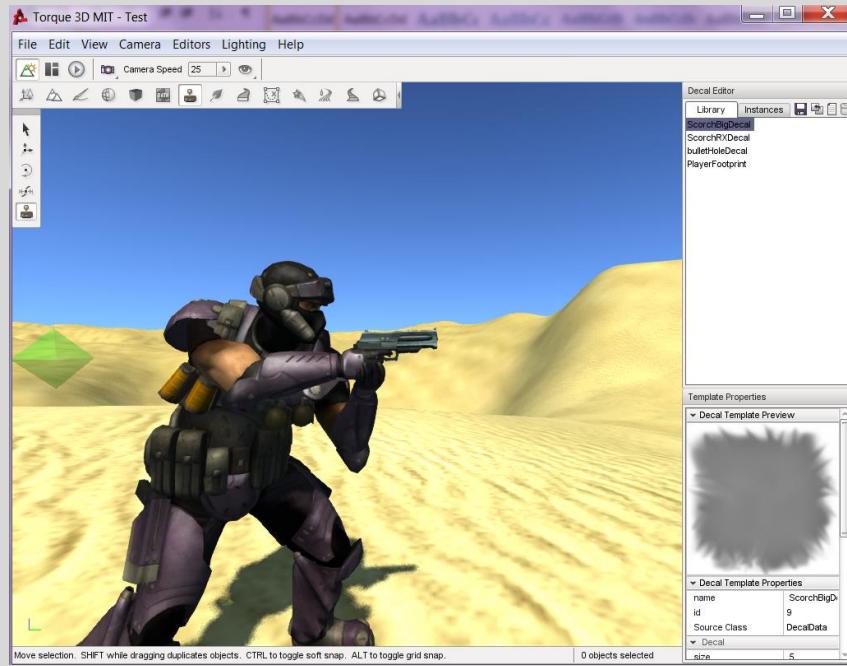


The Datablock Editor does not have any special quick settings bar options or anything next on the mouse control pane, so let's discuss the sidebar, where the actual editor is located. There are two options for the datablock editor, existing and new. Selecting the existing option will allow you to edit properties of existing datablocks. This is how you change the parameters of your player objects, the weapons a player can use, and more. When you select the new option, you can create a brand new datablock by selecting the category of interest and then using the editor to actually make the datablock.

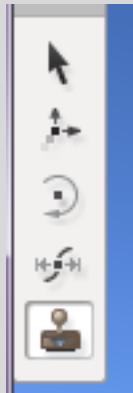
The second part of the datablock editor has a portion that should remind you of the Object Inspector we covered a while back with the Object Editor. This is where all of the datablock's individual fields are stored, and any custom set dynamic fields towards the bottom. There are two important things to note about datablocks. The first is that your game instance can only have a pre-determined set number of them (~2048), and that no two datablocks may share the same name in the engine. Just be sure to save your changes as you move along with the save options in the editor, otherwise you'll lose your work when you exit the editor.

Decal Editor (F7)

Next up is the Decal Editor. This editor is used for painting decals on the map, or basically 2D texture instances that can be painted on 3D surfaces.



The Decal Editor doesn't have any unique settings on the quick settings bar, and all of the tools you need to work with this editor are controlled either on the mouse settings or the sidebar. Let's start with the mouse settings for the Decal Editor.



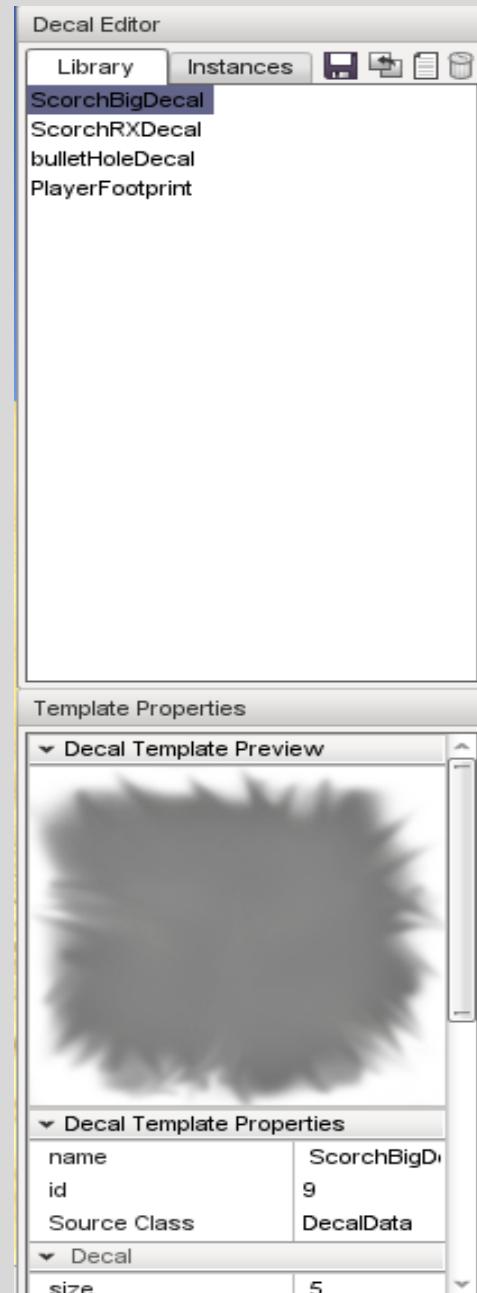
There are five options for the Decal Editor in the mouse settings. The first option is the Select Decal (1) option, which allows you to click a decal in the editor pane to select it. Once a decal is selected, you can either use the Move Decal (2) option, the Rotate Decal (3) option, or the Scale Decal (4) option to apply transformation modifiers to a decal instance placed on the map. The last tool in the mouse settings list is the Add Decal (5) option, which is actually used to spawn a decal instance on the map where you click. The instance that is spawned by this tool depends on the selected options and settings in the sidebar of the Decal Editor.

As mentioned above, most of the settings and options for the Decal Editor are located in the editor sidebar. This is where you can load in textures to become decals, or manipulate existing decals with some tools and settings to make them appear different in the world.

The sidebar for the Decal Editor is responsible for actually selecting and manipulating the decal instance you are currently working with in the Editor Pane. On the top bar we have the Library tab, which contains a list of decals you may use, the Instances tab, which has a list of all active decals on the map. Afterwards, we have the save decal button for when you edit in the Library. The remap option follows which can re-map decals with invalid datablocks to a valid instance. And lastly we have the new decal and the delete decal options.

At the bottom of the list, is a set of controls and options for working with Decal instances. First, there's a decal preview image so you can see what will be placed in the world. Below that are your standard engine options for naming, identification and class. Under those, are the decal options, where you may select a material to use, as well as specify the decal's size and lifetime parameters. The rendering options contains a set of internal commands and settings used to specify how important a decal instance is when a render pass is performed on the game instance. The Texturing options contains some commands and options for "animated" decal instances that store multiple images in one using "frames", or image instances that are spaced exactly the same and located within another image. For Example, 5 frames of 20x20, would be in a 100x20 image. The last tab of texture coordinates are slightly out of the range of this guide, so we won't cover that topic.

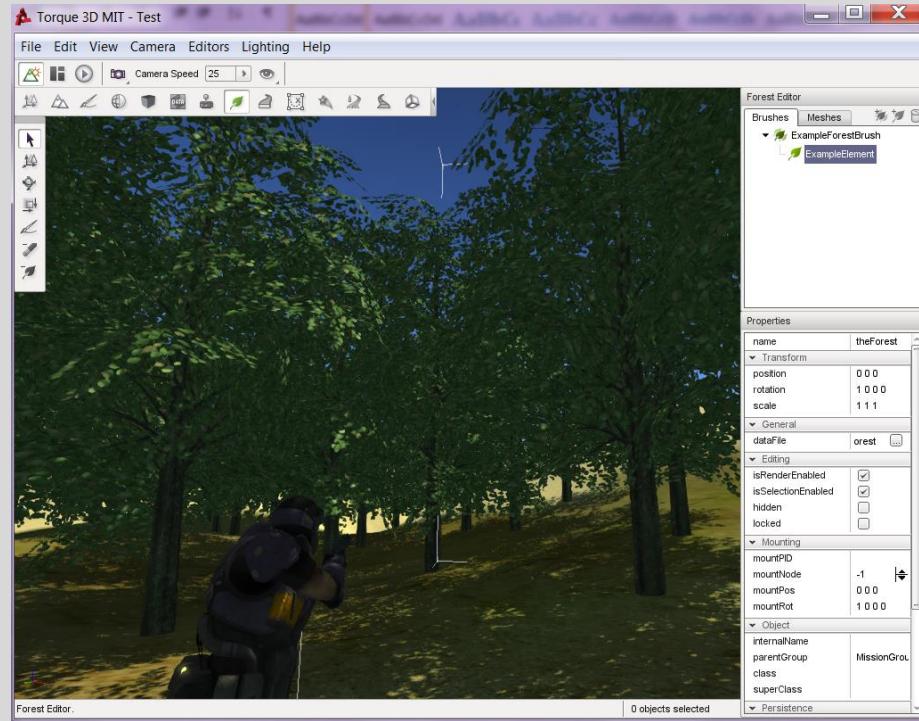
When working from the Instances tab, a list will populate based on any decal instances placed on the map. You can directly select a decal by simply clicking the item in the list and then either use the mouse settings bar to select tools to manipulate a decal instance, or simply delete or replace a decal instance that is placed on the map.



So, now that we've talked about the Decal Editor, and how to use it in Torque 3D, we can move onto the next editor, which is the Forest Editor.

Forest Editor (F8)

The Forest Editor is probably right up with the Terrain Editor in terms of the fastest way to change how a level instance looks. This is a way to rapidly paint object instances to a level, but with one big catch. All of these objects are stored in one single Forest Object in the world, instead of occupying numerous objects worth of space.



Like the editors before, the Forest Editor does not have any special new options on the Quick Settings bar, as all of the tools are controlled by the mouse settings or the sidebar of the editor. The purpose of the Forest Editor, is exactly as it sounds, to rapidly create a “Forest”, or in a more generalized term, vegetation. So, let’s start talking about the tools here.



Under the mouse settings options, we have some important tools to work with. The first option is the Select Item (1) option which allows you to select individual forest objects once they are placed and directly manipulate the individual object. The second option is the Move Item (2) option, and then we have the Rotate Item (3) option, and lastly, the Scale Item (4) option to manipulate transformation properties. When you click on the last three tools, some brush settings you have seen before will appear on the Quick Settings bar, and they’ll perform the exact same way as you have learned. The Paint (5) option will paint random forest objects in the bounds of the brush area. The Erase (6) option will erase forest objects in the bounds of the brush area. Lastly, we have the Erase Selected (7) option to delete individual objects located in your forest.

By combining these tools and settings you can quickly build forests, fields, and much more containing large groups of trees, brushes, or **any** object for that manner!

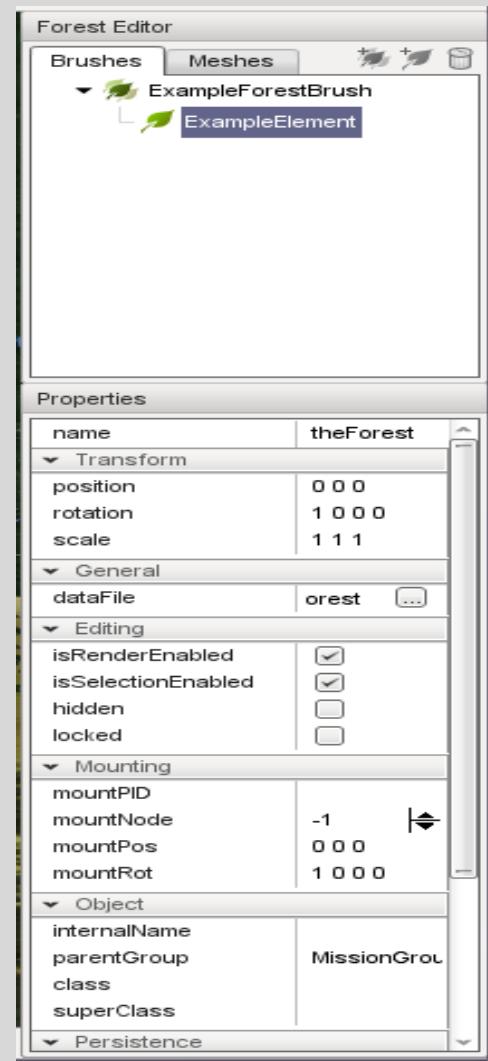
As to be expected however, these mouse tools won’t work alone. The actual options and settings for the Forest Editor can be located in the sidebar, where you can manipulate the entire forest, or add new instances to be painted, and much more. So let’s talk about the sidebar now!

The Forest Editor sidebar is split into two important categories. There are Brushes and Meshes. A Brush contains a list of meshes to be painted when using the mouse controls, and the meshes contains a list of objects that are within a brush. In the Brushes tab, you have a list of “Brush Groups”, which contains a list of Meshes to be painted under a drop-down arrow. You can add and remove mesh instances, or a new brush group with the options on the far right of the tabs.

Under the Brushes tab is the properties menu, which controls the entire forest as it exists, or, you can select a single forest object and directly manipulate it like you would with the Object Editor here. When you make a forest selection, the properties tab will switch to the Brush itself to control aspects of the probability of that brush being used in the set, and more options.

Under the Meshes tab is the Mesh Properties menu, which controls things such as the shape of the mesh, and how the object reacts to things in the world, such as the wind and other objects. From this menu, you can also create a new instance to use for the painting tool.

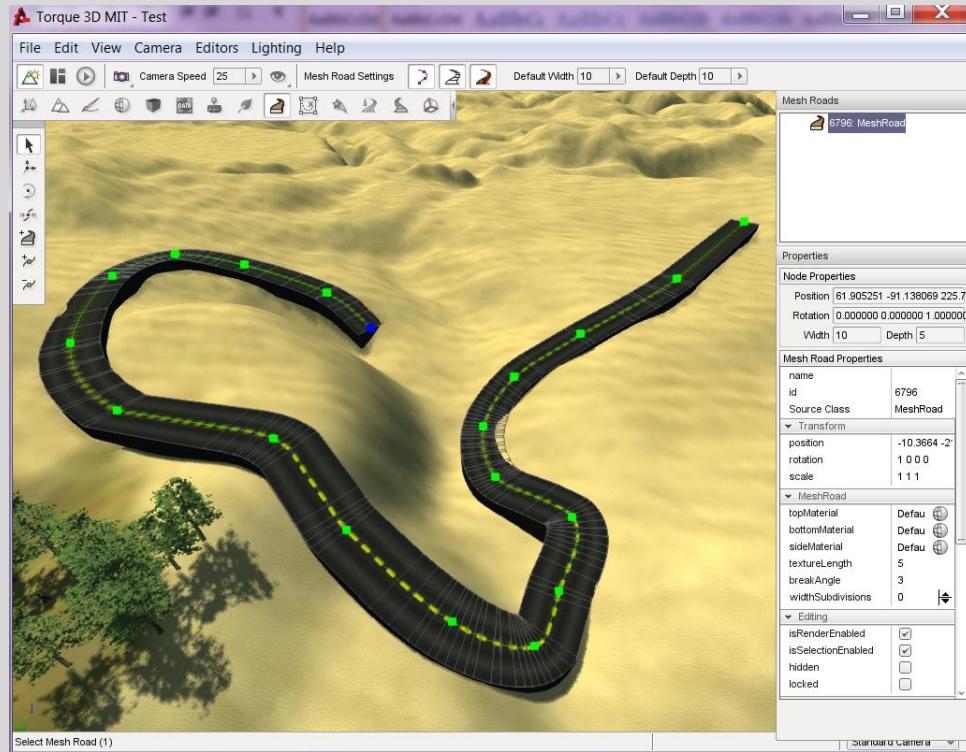
By moving between the options within the sidebar of the forest editor, you can create numerous different forest types and looks, and even use a single shape with scaling and rotational effects to make something that appears to be completely unique, but actually is a large group of the same objects.



By using the many different properties and settings of the Forest Editor, you can create vast jungle settings, or even use the forest editor to paint large quantities of rocks or other objects in specified areas of your map. The best way to learn this tool is to experiment with it, there's plenty you can learn from it!

Mesh Road Editor (F9)

Probably one of the most unique editors in the World Editor is the Mesh Road Editor. People making Racing Games, or games that revolve around Vehicles will likely spend a good deal of their time in this editor. With the Mesh Road editor, you can take a road shape object, and create roads of varying shapes and sizes to stretch around your world. The editor contains a list and set of unique tools and settings to control things such as bends and turns, as well as elevation and more fun settings.



There are actually two different “road” editors to be known about in Torque 3D. The Mesh Road and the Decal Road. We’ll get to the latter in a bit, but let’s briefly discuss what this editor actually does. The Mesh Road editor takes a “road” mesh and then bends it according to a Bezier Curve (Chapter 9) or for the sake of the engine’s terminology, a Spline, which is created by using a series of points you create. The road then has a texture painted over it to make it appear as a road. The mesh road takes terrain height into account based on the points created, so the road may actually “float” over the terrain in some points if the settings are proper for it. This effect may help when creating things such as bridges, but in other cases may counter the desired effect. You’ll need to experiment with the editor to attain your goal for a road instance.



The Mesh Road editor has a few options for the Quick Settings bar. The three selectable items will toggle off/on things that are rendered by the editor, such as the Bezier Curve of the road instance, the mesh grid of the road, and the texture of the road, and the Default Width and Default Depth control the width of the road mesh, and the height of the mesh object relative to its current Bezier Point.

Most of the editing you will do in the Mesh Road editor is controlled by the Mouse Settings and the Sidebar options. Let’s talk about what these options do and how you can use them to your advantage now.



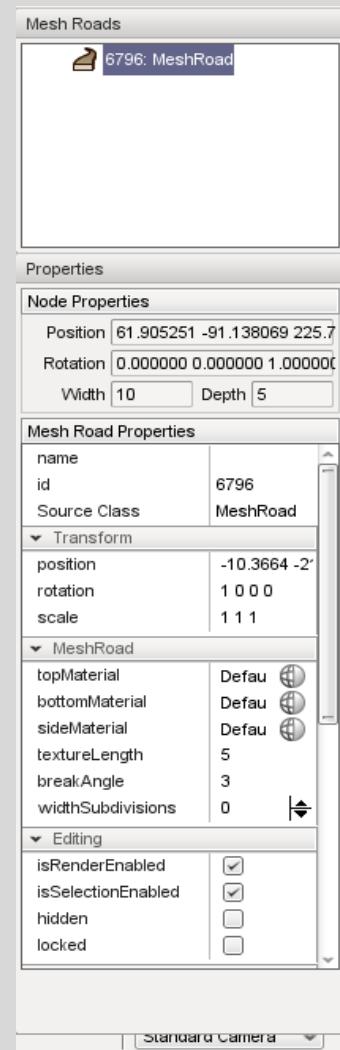
The mouse settings for the Mesh Road Editor mainly contribute to Selecting the road, and editing individual aspects of it. The Select Mesh Road (1) option allows you to select a road instance or an individual road control point to manipulate. The Move Point (2), Rotate Point (3), and Scale Point (4) options allow you to edit properties of individual control points, which will in turn edit the road between the point prior to the selected one, and after the selected one using the Bezier algorithm. The Create Road (5) option allows you to create a new road instance where you click. You can continue to click points until you are done, at which point you may either press ESC or click another option to stop creating a road. Lastly, we have the Insert Point (+) and Remove Point (-) options for quickly inserting or removing points from the selected road instance to quickly add or remove segments from the road instance.

The remainder of this editor is controlled by the sidebar, which is used to define properties of the road itself, as well as a few minor options.

The sidebar of the Mesh Road editor doesn't have too much going on, yet it does serve as importance for how the road behaves when created, as well as how each individual point acts compared to the remainder of the Spline. The first option is a list of road instances on the map you may select from. When you select an item from this list, the last node it automatically selected to be edited. Below the list is the Node Properties tab, which controls each individual node in the list. By selecting a node in the editor pane, the items in the properties menu will edit themselves to be further adjusted by you.

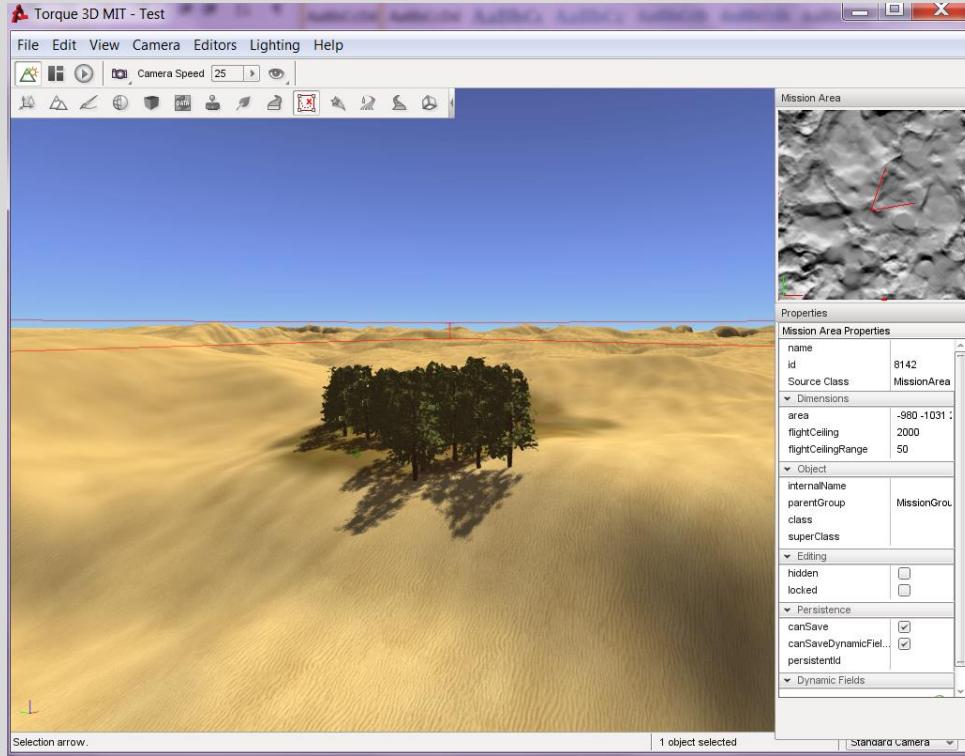
Below that, is the Mesh Road properties, where you have your standard list of options for objects in Torque 3D. Under the Mesh Road list, we have a list of special options. The *topMaterial* is the material that is rendered on the road as viewed from a top-down perspective. The *bottomMaterial* option is the material that is rendered on the bottom side of the road, and the *sideMaterial* option is the material rendered on the sides of the road mesh. The *textureLength* option controls how "long" of the road needs to be painted with the material. For the generic texture, a larger number here will correspond with the lines being much longer on the road. The *breakAngle* option is an angle in degrees that the road will actually turn in the event that the threshold of the engine is broken by the Bezier Curve instance. Lastly, the *widthSubdivisions* option will subdivide the curve segments the specified number of times when it generates the vertices of the curve.

Play and explore with the effects of all of these options to further learn how to use them in the engine. Mastering these tools will help you create roads and paths for your players to follow in the world so they don't get lost.



Mission Area Editor

The next editor on our list is the Mission Area Editor. There really isn't much to discuss with this editor as it is essentially a glorified version of the Object Editor's Inspector tool, but built specifically to work with a MissionArea object. Do note, that you need to have a MissionArea object spawned in order to use this editor, and it will not prompt to create one if you don't have it.



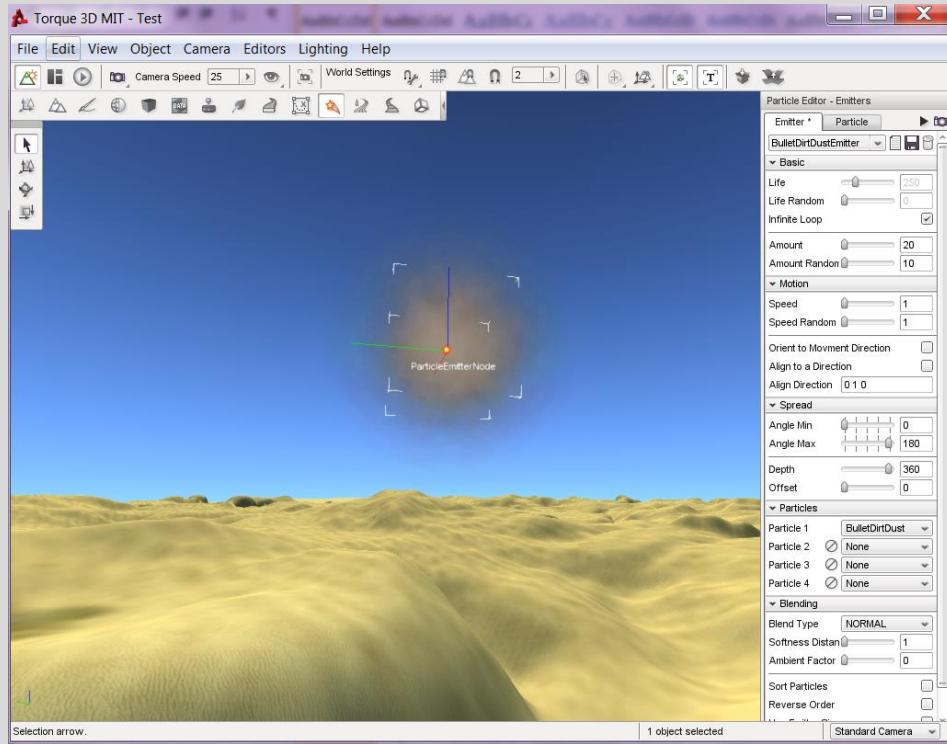
A Mission Area is a specified boundary to your map. Players can still freely enter and exit this area and it won't do anything special unless edited to do so (Chapter 11). Some addons to the engine use the MissionArea object to grasp the level boundaries to perform special tasks, and it is at least recommended to have a MissionArea object on all of your maps, but not required.

Particle Editor

The Particle Editor is a special editor that is designed for creating special effects in game. With this editor, you can use textures to create various different effects such as fires, explosions, smoke, and much more. The grasp of the particle system in the engine is a very wide topic for discussion and there have been numerous additions, and edits to the system to contradict writing a full chapter on it, so we'll just go over the basics so you can at least get going to the point of experimentation to handle the remainder of the learning.

As for the premises of how the system works for the engine, it is split into a particle and an emitter. A Particle is a single instance of textures and settings to actually render an effect in the engine, and the Emitter is an object which creates particles and applies motions and rotations to the individual particles. Unlike some other game engines out there, the system does not have direct particle collision

support just yet, however that has been a topic of discussion for a feature update mechanic to come to the engine at a future point.



This editor is almost exclusively handled by the sidebar here, since the other editor tools and settings mirror the Object Editor. The editor itself is split into two category tabs, the Emitter tab and the Particle tab, each of which is responsible for editing the two responsible objects needed for a rendered effect instance.

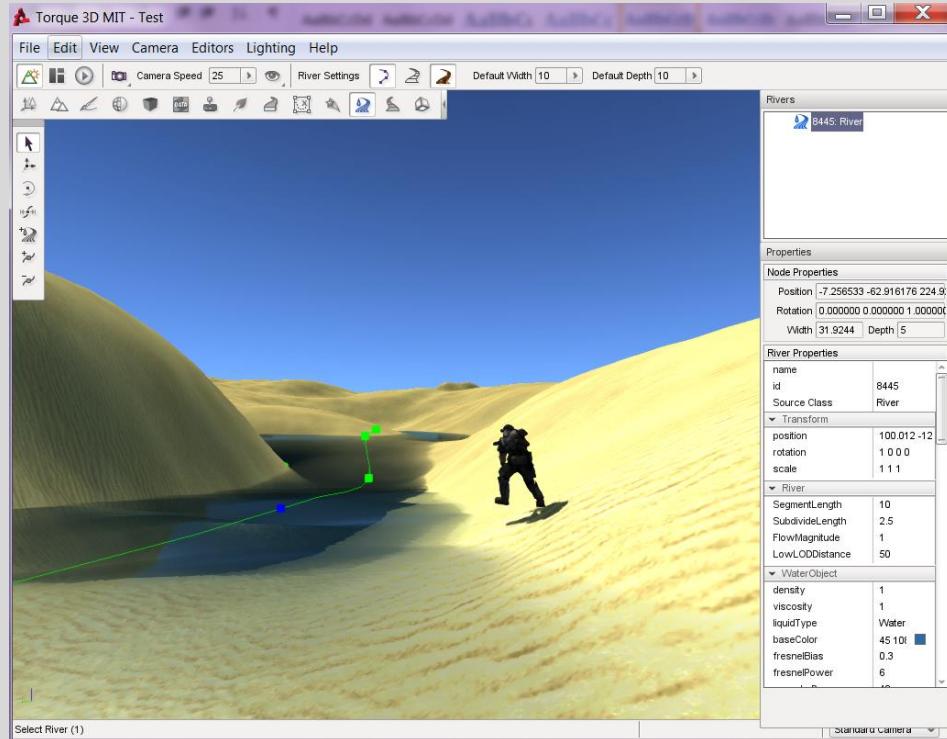
As noted before, this guide will not go into extreme detail regarding the Particle System and what everything does here as each of these settings has a related effect to the next. The best way to learn this system is by trial & error, and by reading and observing other people's particle effects to see how each of these parameters affect the rendering of the world effect. Also, you need to keep in mind that certain objects that are placed in the world can also apply a "wind" effect to particles on the local scale which may also apply an effect to the emitter. We will however, go over some of the important concepts here.

In the Emitter Tab, the life option controls the overall lifetime (in milliseconds) of your particle. The Life Random option applies a small addition or subtraction on the bounds of this option to the lifetime number, and infinite loop keeps the particle alive indefinitely. Amount & Amount Random control how much of the particle is created by the emitter. You need to be careful here not to create a "Virtual FPS Killing Machine" here, or basically a particle effect that eats your performance when you look at it. The motion and spread options control the overall movement of a particle once it is created. The Particles option allows you to specify multiple particle effects to be bound to an emitter, and Blending controls how they blend with each other when coherently spawned.

Under the Particle Tab, you can directly edit the textures and coloration and sizes of these textures as it related to an individual particle. You can also apply effects such as spin, and state how forces such as gravity and drag effect your particles.

River Editor

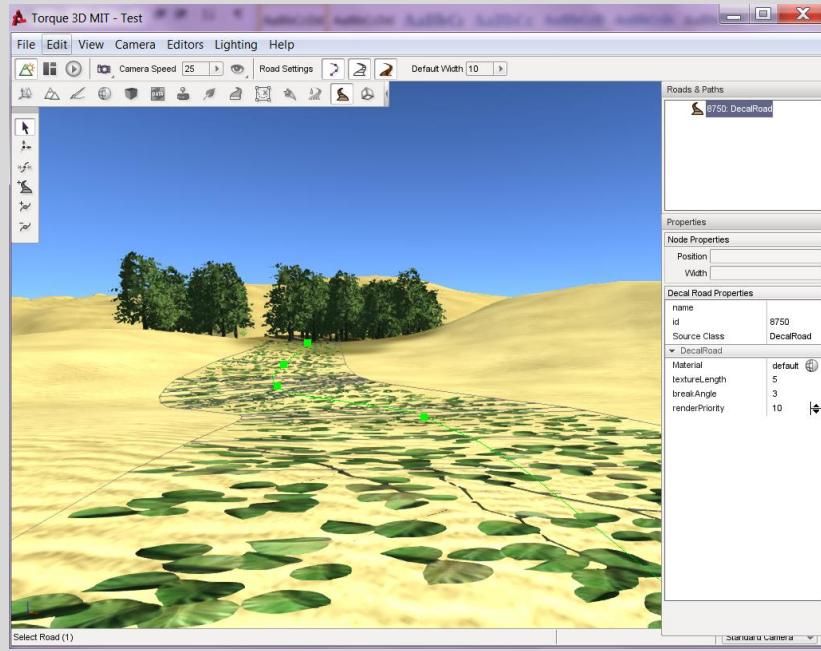
The River Editor is the next editor in the list. It behaves almost identically to the Mesh Road Editor in terms of functioning; however instead of spawning a mesh, it spawns a river, or basically a small block of water.



Since the functioning is essentially identical, we won't actually cover anything new here, just know that the sidebar has water properties so you can adjust how the water behaves for your world instance.

Road Editor (Decal Road Editor)

You're essentially doing the same thing with the Decal Road Editor as the River Editor and the Mesh Road Editor in terms of functioning.



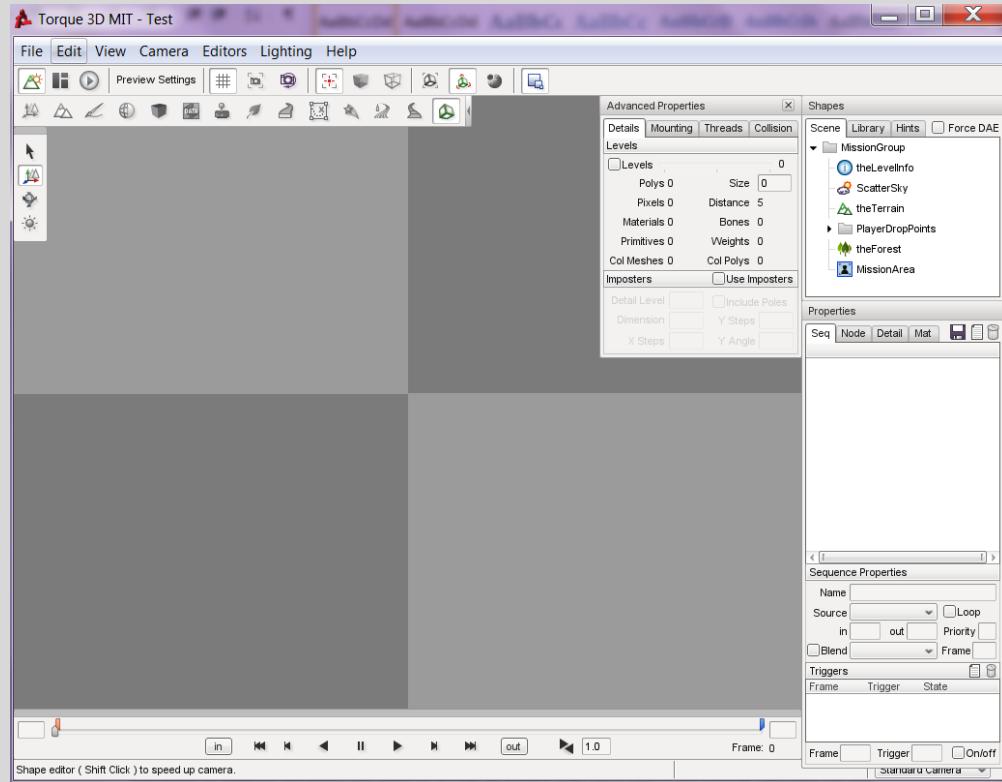
The only difference to discuss here is how the decal road differs from the mesh road. Recall that terrain heights were directly influenced to create things like bridges using the mesh road. That does not apply here. Essentially, for the Decal Road Editor, it will paint directly on the surface of the terrain here. When you want to create things like dirt paths, or even the effect of a forest or jungle floor with leaves, you can use the Decal Road Editor to quickly accomplish this.

Shape Editor

The final editor is the Shape Editor. This is a unique editor in the engine that was bundled in the World Editor to remove the need to create a third editor category in the engine. The Shape Editor is responsible for loading in models and art assets and “readying” them to be used in Torque. Essentially, this is where you bind textures to objects, load in the animation sequences and name them to be used in the engine, and to finalize everything before it can be safely used. When you save the changes in this editor, it creates a .DTS model which the engine recognizes and a source script file containing the scripted information related to the model itself.

The scope of the shape editor is very massive, and it would have numerous sub-topics to actually discuss along with the necessary steps to load in new models and sequences to the engine. That is beyond the scope of this guide, as it fits more directly with that of your artist’s job. If you are however, an artist reading this guide for the sake of that topic, direct your attention to the Engine’s Documentation regarding the editor here:

<http://docs.garagegames.com/torque-3d/official/index.html?content/documentation/World%20Editor/Editors/ShapeEditor.html>



The Shape Editor

So, that's all of the editors in the World Editor! That's quite a bit of information to actually take in, but don't fret. You'll usually end up using only a few of these to actually create stunning levels and wonderful environments for your players to enjoy.

Now, let's actually step out of the editors and actually use them to create a new level!

First Steps

So, now that we've covered all of the editors located in the World Editor, let's actually walk you through the process to creating a brand new level. In terms of actually making a full level for a game, we'll cover some more tactics and specialized objects when we get to Chapter 11, but this section of the chapter should be viewed as a generalized guide on making a level for your game, and an overview of the process of level creation.

Design Rules

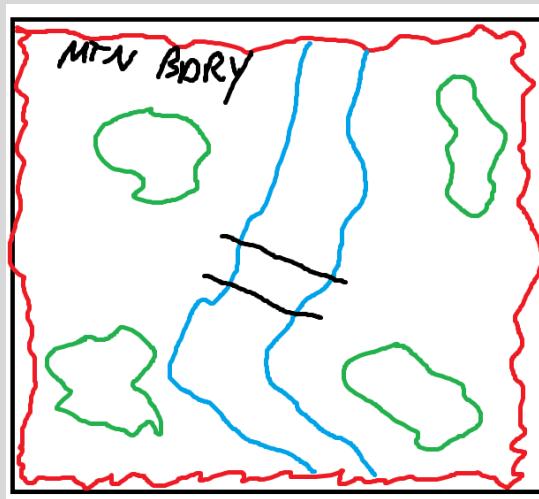
The first big rule is to realize that every game is different, and every level for every game will be formatted in a very different way. For example, a role-playing game will likely be more of a "world" versus a level, where there is a starting point and an ending point that need to be reached, or for something even more advanced, numerous locations on the map where the world translates from one zone to the next. Now, consider a First Person Shooter title, where a multiplayer arena needs to have elements of balance between the sides of a map to ensure that both teams are evenly matched for the scenario presented before them. Just remember, a world for a game is where the players will be

interacting with each other, and the environment they are in. Here are a few things to always consider when making a level for your game:

- How big does the map need to be?
- What kind of environment does my map have?
- How many players does my map need to support?
- What kind of terrain effects will my level have? Does my level need a terrain object?
- Are there any Dynamic Objects my map needs? (CH. 11 Topic)

Design Concepts

Once you answer these basic questions, you can move onto the first step of level design, which is to create a concept for the level. This can either be a concept diagram that you personally draw out, or you can do it in a program like Photoshop or Paint even. This is just for you to have a general overview of what a map needs to have.

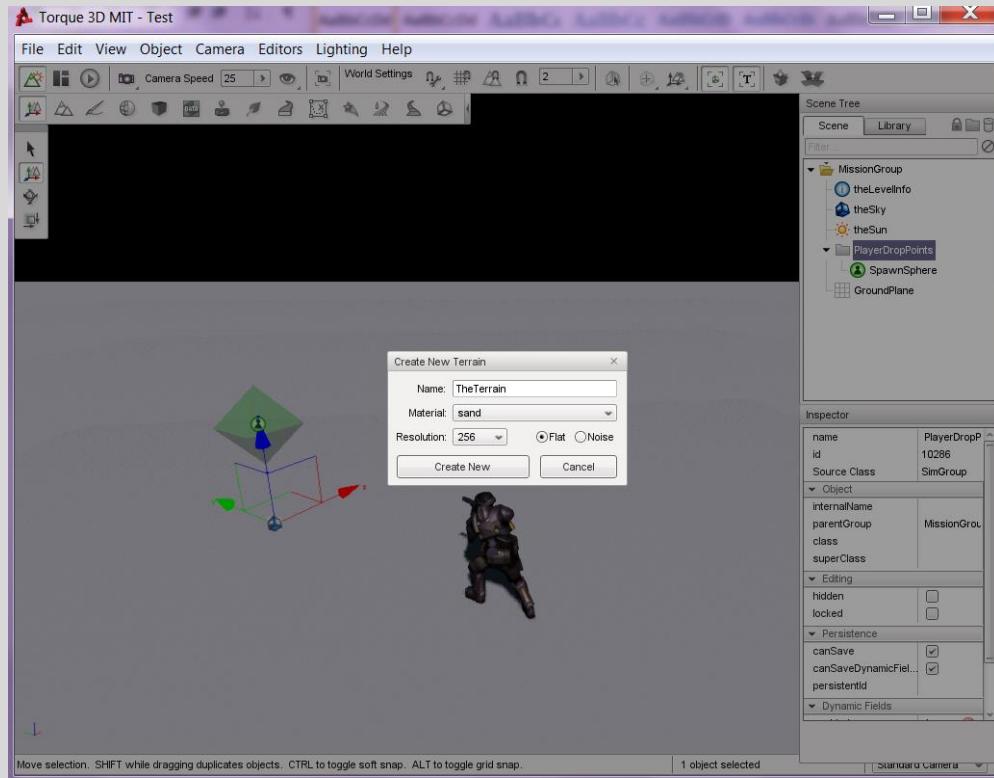


Concept Diagrams

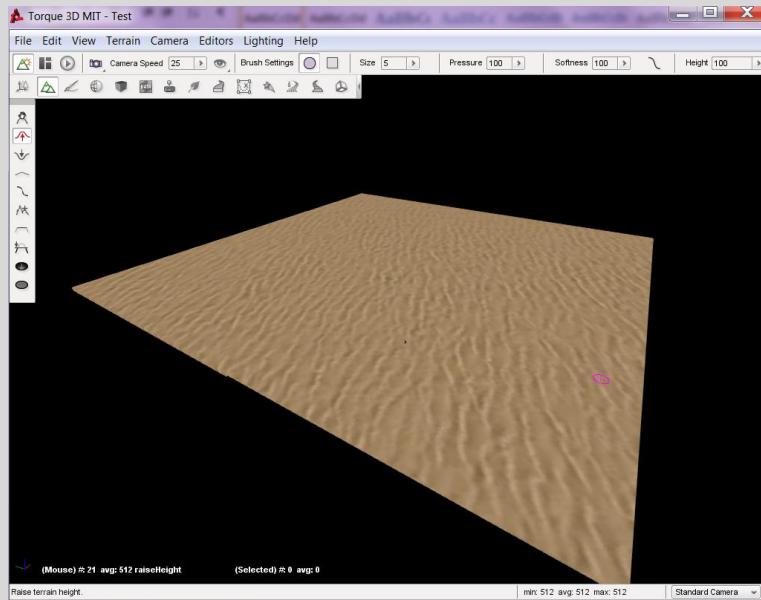
Obviously, you can be much more detailed than this, but basically some kind of an overview to go from will help you out in the long run to make sure that your idea flows along with the overall concept of the map. Once you've got a basic concept in place, you can actually get started.

Adding Terrain

First thing to do is to create a new level, which will open up the Empty Room map. When you open this map up, it will already have some generic objects to use. What I usually prefer to do is set my camera to the world camera so I can quickly pan around the map and find what I need for editing purposes. The first thing we need to do for this new map is to add a terrain object to the map. So open up the terrain editor and select the option to create a flat terrain:

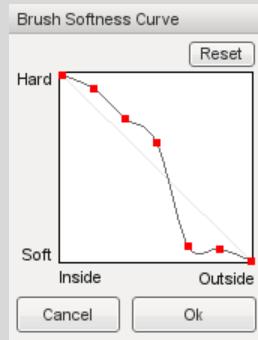


This will spawn a large flat sand terrain on the map. There's just one problem here however, and that is the fact that we already have a special object on this map. That big white plane we're standing on is actually a special object called a Ground Pane. Open up the Object Editor, and delete this object. Your character should now fall down to the small sand terrain you have just created. Now, if we zoom out and switch over to the terrain editor we can start working on the outer boundaries of our map.

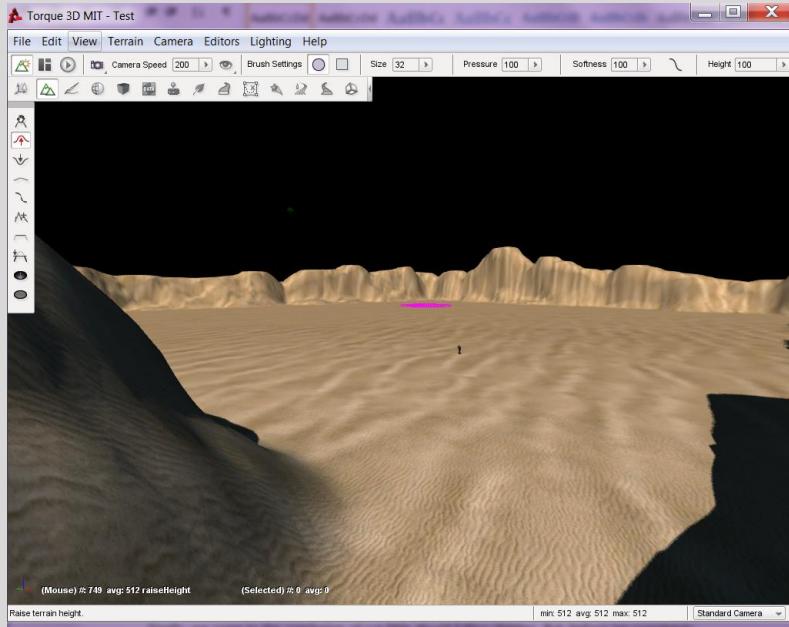


Boundaries

So, according to our little concept diagram, we need to create a mountain boundary to the map. In games, a Mountain Boundary is basically a zone of extremely elevated terrain that surrounds the far outside of the map, so a player cannot cross it and fall off the edge of the terrain. There are a few other types of boundaries we could use, and we'll talk about those later on. The easiest way to create a large mountain boundary is to either use the Set Height tool to a high value, and then build a boundary, or to use the Brush Softness Curve tool to create a terrain style that players would have a hard time crossing and use the raise terrain tool to accomplish the effect.



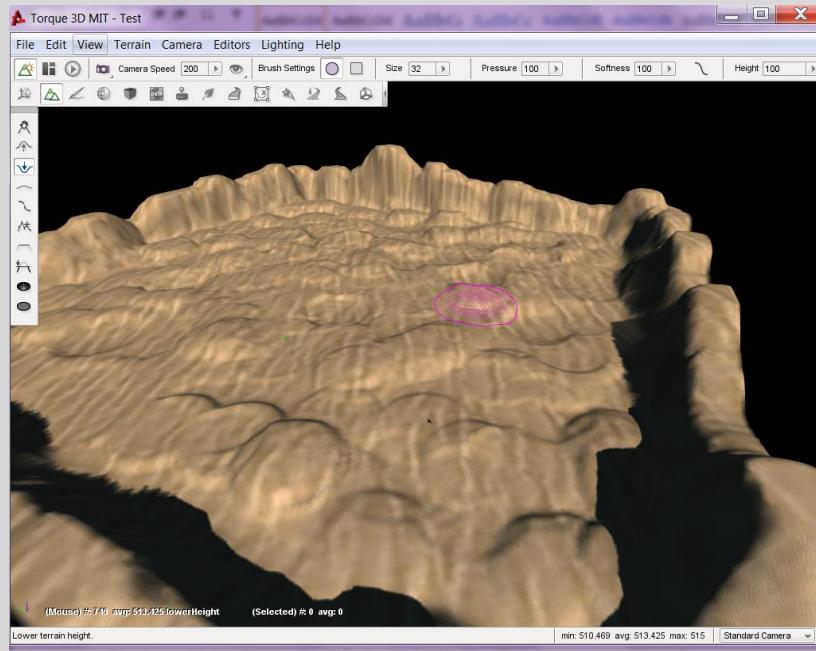
Once you have the curve set, apply with vigorous force to the outer boundary of the map the Raise Terrain tool to create a large ridge of unpassable terrain.



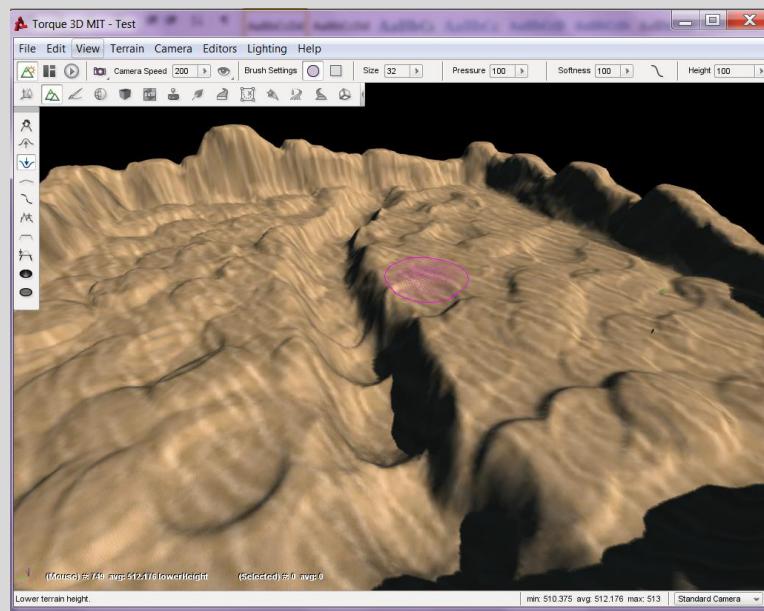
Terrain Noise / River Sections

The steep sections of the terrain should provide more than enough of a blocking force to prevent the player from being able to walk up the mountain and off of the map. Be sure to switch between the world camera and the player camera and actually test your created map's edges to ensure that players cannot exit the designated mission bounds. Now, a large flat section to any map would bore

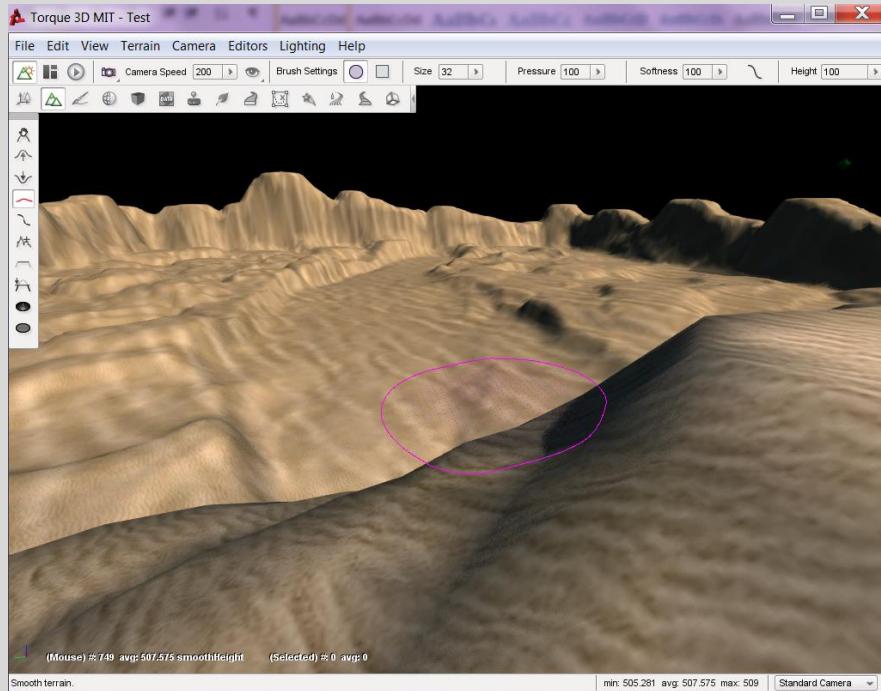
most players eyes, so let's mix things up by adding some segments of higher and lower terrain. Experiment with the curve smoothness for the terrain as well as different segments to get something like this:



Now, if you properly recall from our little concept diagram, we're to have a river that cuts through the middle of the map. Now, you could use the river tool to accomplish this, but I'm going to show you another way to make water for the map. Let's start by cutting the river section into the terrain itself.



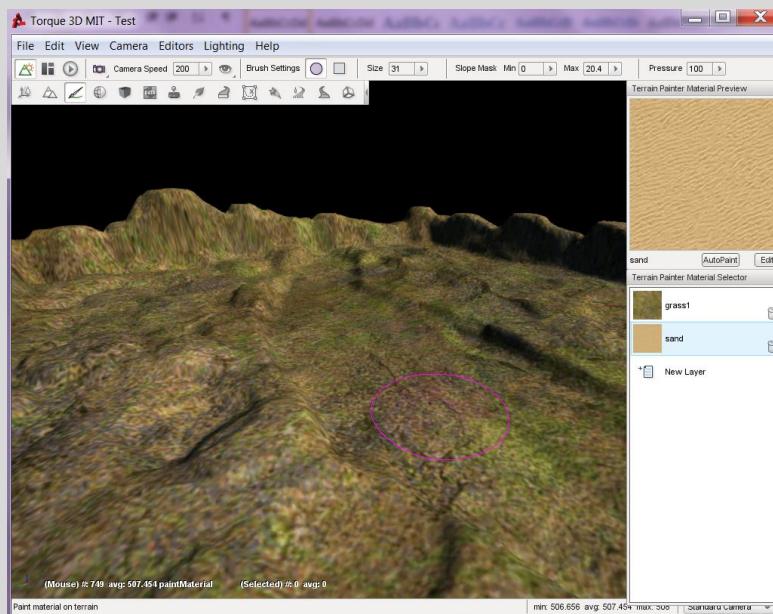
One problem though, if a player falls into this chasm, it's game over for them. So, let's fix that by using the Smoothen tool along the edges of the river chasm, and in the river chasm itself.



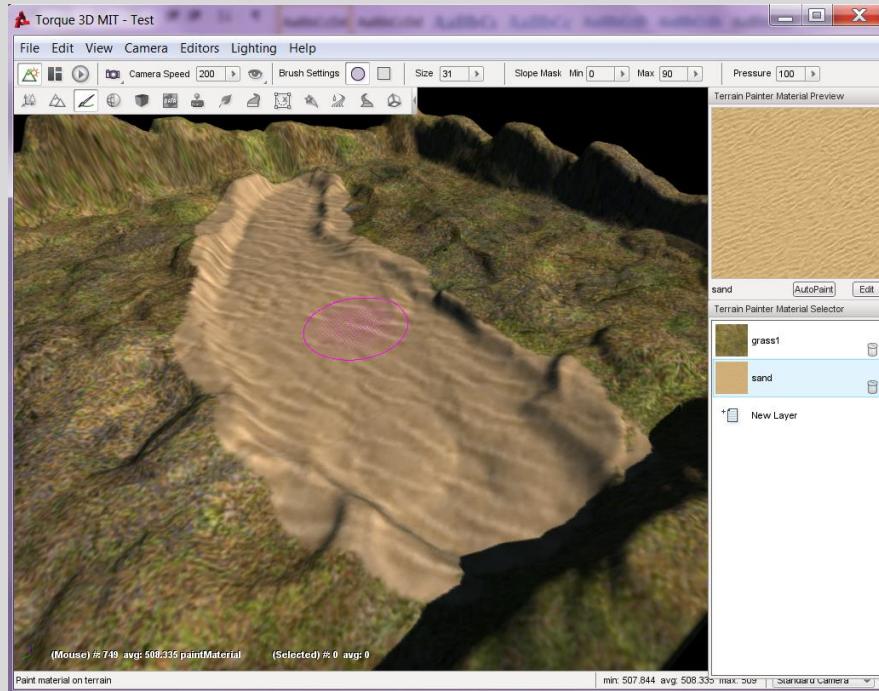
That's Better!

Coloring The Terrain

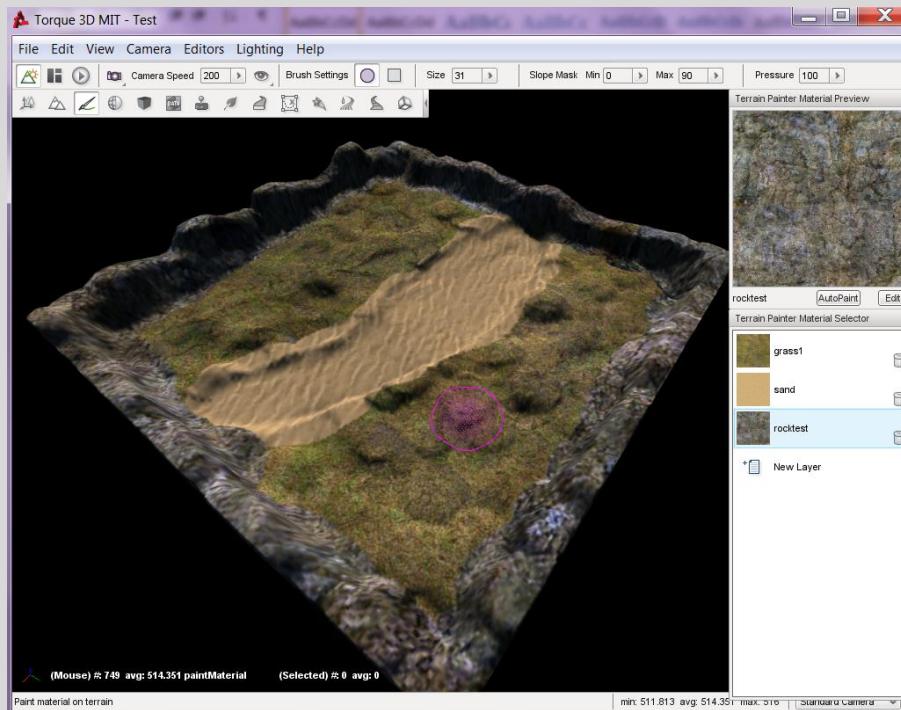
So now we have the generalized terrain concept done. But there's just one problem for me here. The entire map is a giant desert, and I didn't want a desert. Let's fix this with the Terrain Painter tool. So the first thing I do here is replace the sand layer by double clicking it with a Grass-1 Layer, and then add a new sand layer to the map so I can paint the areas I want to have sand with sand.



Next, I paint the entire river area, and the surrounding terrain with the sand brush to make it look more like the area around a river would look like.



So that's nice and all, but now I have large grassy mountains. So, let's fix that by adding a third layer and paint the mountain boundaries with the rocktest layer.



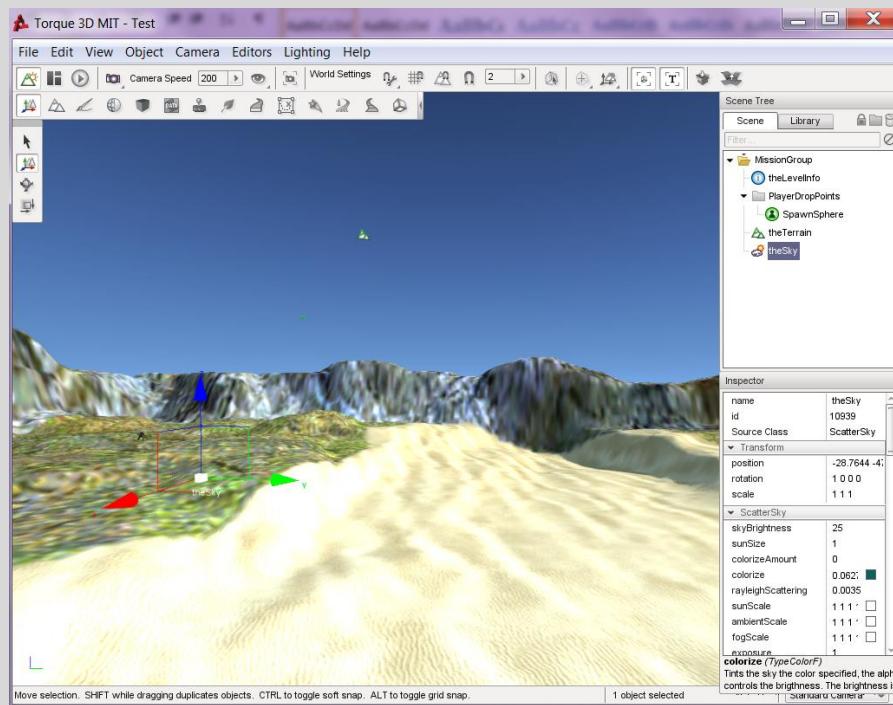
So that should just about cover everything we need for our terrain. Feel free to make further adjustments to the terrain before moving on.

Adding an Environment

Now that we have the terrain for the map done, we need to actually give the world a bit of an environment so the player actually feels like they're walking in a world rather than some random box. Open up the Object Editor and click the Library tab on the right side. Then under the level tab, you'll see an Environment tab. Click in there to open a large list of environmental objects to work with.

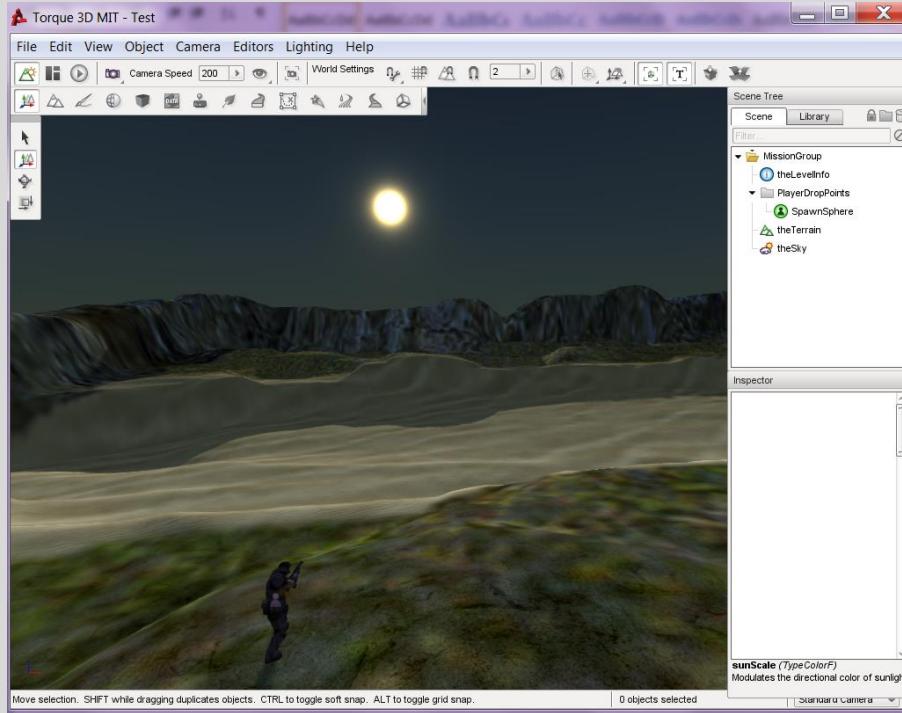
Purging that Black Abyss: Skies

The first thing we want to do, is give our world a sky, because frankly, I'm tired of looking at the black pane of oblivion. To do so, from the object inspector, delete the two objects named ***TheSky***, and ***TheSun***. You will then want to select from the Environment list the **ScatterSky** object, and spawn one in the map. Once you do that, your world should light up!



The ScatterSky object is a special object that works for Torque 3D. The two objects you deleted prior to spawning this object built a sky using the old SkyBox method of Torque Game Engine (TGE) and Torque Game Engine Advanced (TGEA), this new ScatterSky object does both of these, and can do some very cool things as well. So experiment with those options! You've got a lot to pick from here. And don't worry about the black sections on the bottom of the pane, they're simply blank areas and we can fill that in later.

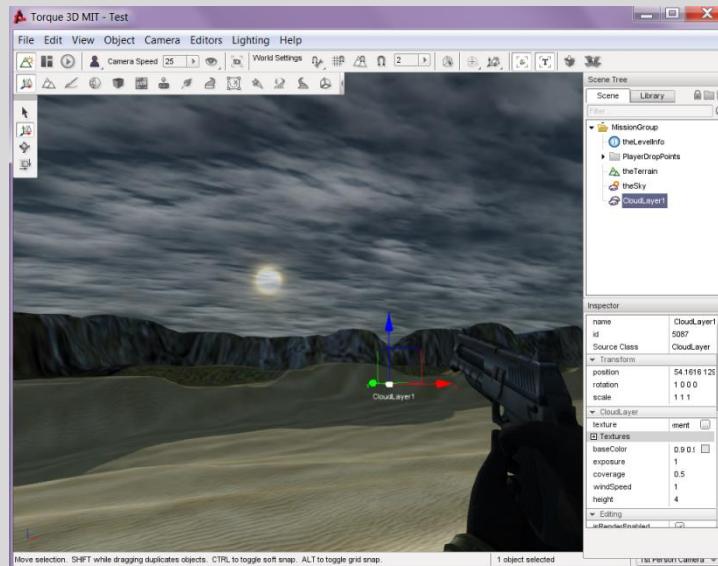
After playing around a bit with the lighting parameters and other sky parameters in the Object Editor, I create a sky that looks a bit like this for my level:



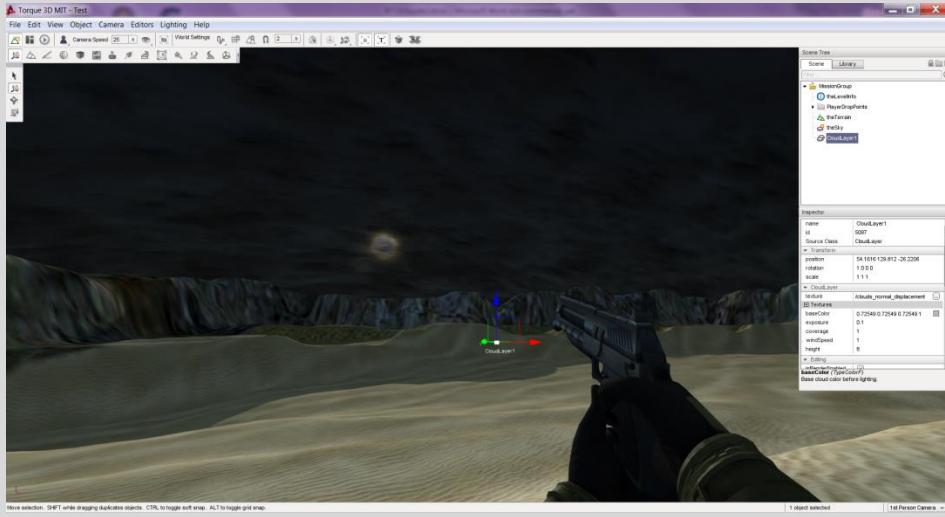
So now, while still talking about the Sky, let's say I want to have a bit of a storm going on for this area at the time of the map. To do that, we're going to need a few objects.

Clouds, Precipitation, Storm Objects

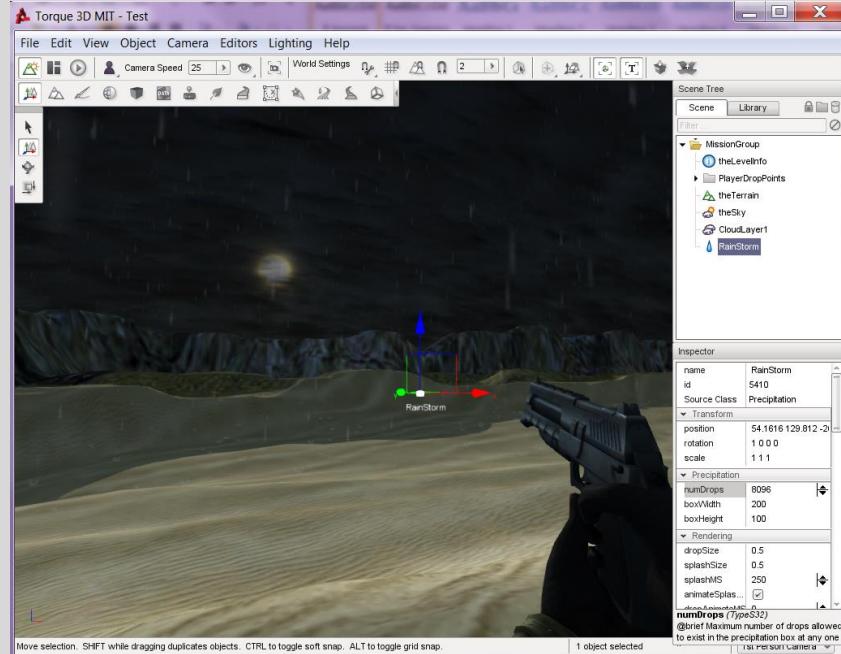
Now, let's talk about the sky we currently have set up. The ScatterSky object handles both the sun and overall mission lighting effects. To actually add some more things to the sky, we'll need to make use of a few more objects. The first new object we want to add is called a Cloud Layer. Like all of the other objects we have placed to this point, you'll find it in the Environment folder. Just leave the texture as the default setting, name it something, and place it on the map.



The Cloud Layer object creates an infinite plane of clouds that stretches across the whole sky. You can also set parameters to control the speed of the clouds, the height of the clouds relative to other sky objects, and even more. After messing with the parameters of the CloudLayer object, I get the following result:

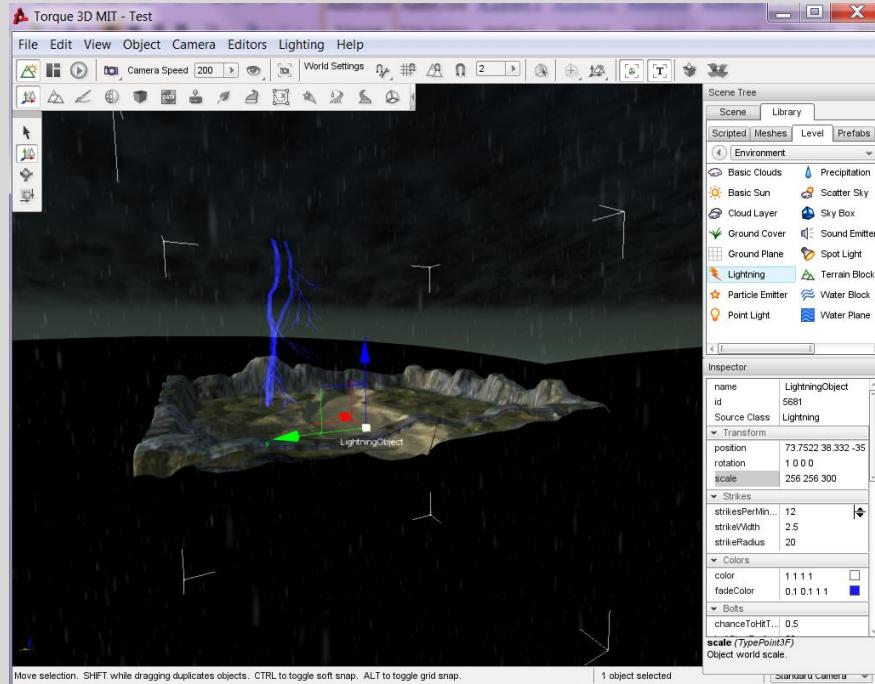


So, now that we have our storm clouds, let's actually add some rain to the map. You'll find the Precipitation object in the same category as before, so add it to the map now. Once you select the HeavyRain datablock and name it, you'll notice a few rain drops begin to fall on you. But, this is hardly a storm, so find the **numDrops** parameter, and up it to something like **8096**, and you'll get a more noticeable effect:



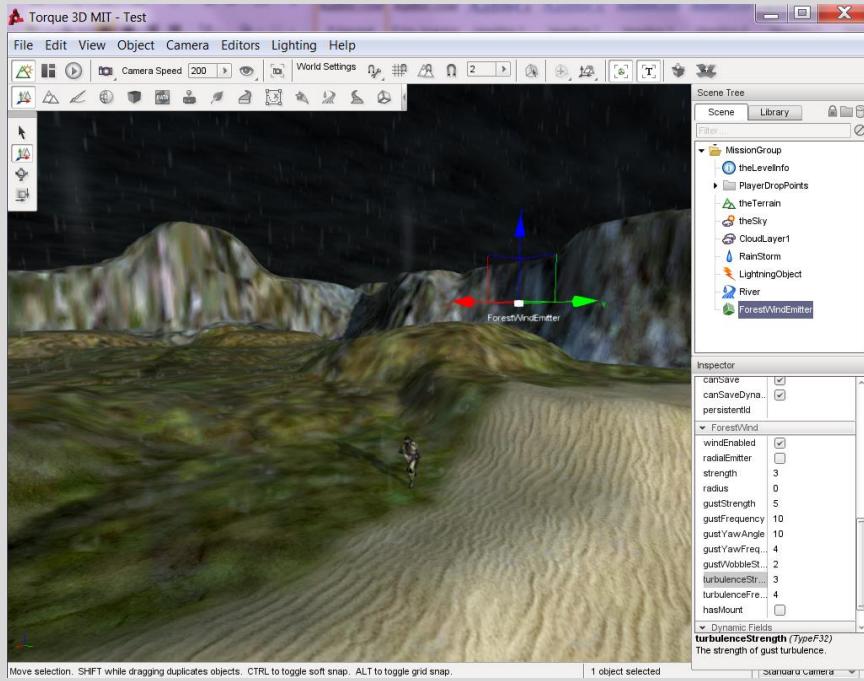
I'm still not completely convinced that we have a "storm", so let's make even more rain fall, but by changing something else. Find the **boxWidth** parameter, and cut it in half to **100**, the effect should be almost instantaneously noticeable. Feel free to adjust the other parameters as you feel the need to.

The next thing we want to add is some lightning to our little storm. Now, Lightning is a special object in the sense that it can actually damage players when they get struck. And, we can control that by using the object's scale parameter to create a box section where lightning will hit. I want the entire map to be valid for strikes, so I just scale the box to cover the whole map.



Feel free to adjust the remaining lightning parameters to create the storm you want to have for the map. But that covers the effects of the sky on your world. Now we have a more dynamic map environment, where players need to be careful of the storm when they navigate through the mission.

But, there is still some room for one more little toy in our group of options. Find the Wind Emitter object in the Environment tab and place it in the map. This will add a wind storm effect which will apply to all emitters, and even precipitation. This will make the map feel more "alive" in a sense of it not just being a persistent downward rain.

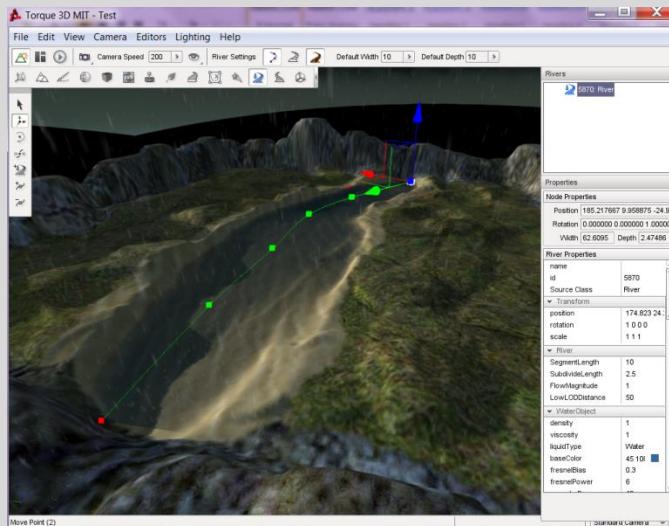


Water

Now, if we return to our little design concept for the map, you'll recall that I have a river cutting through the middle of the map. Right now, all we have is a sandy trench cutting down the middle. There are two ways we can approach the concept of the river for this map, and I'll talk about both methods here.

River Editor

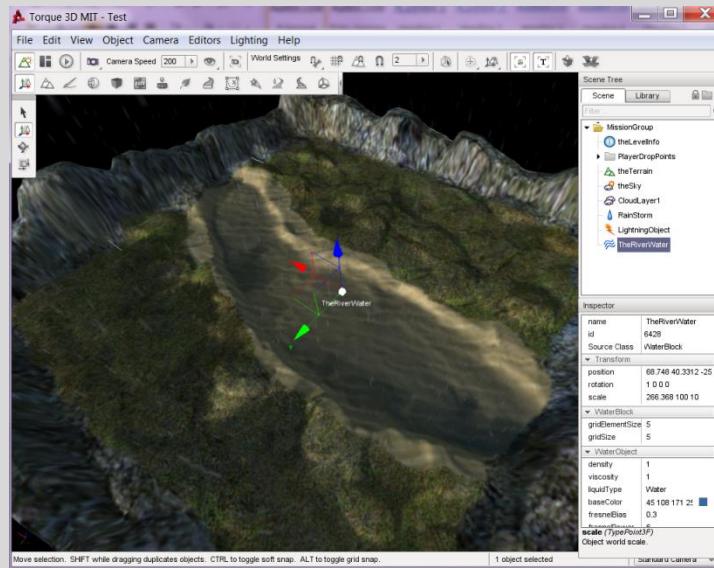
The first and obvious choice here is to use the river editor to create the river. This has an advantage of being able to control segments and where specifically they need to be placed. For this case, you can simply use the river editor to place a Spline down the sandy area, and then adjust the transformation properties of each individual node so that it perfectly fits for the river:



Obviously, you need to keep in mind that the terrain here also has “low” areas where water can leak into which may flood part of the map that you didn’t intend to be flooded by water. For most cases, when you want to make water that is zoned to curved areas like a river, you’ll want to use the River Editor to accomplish this.

Water Block

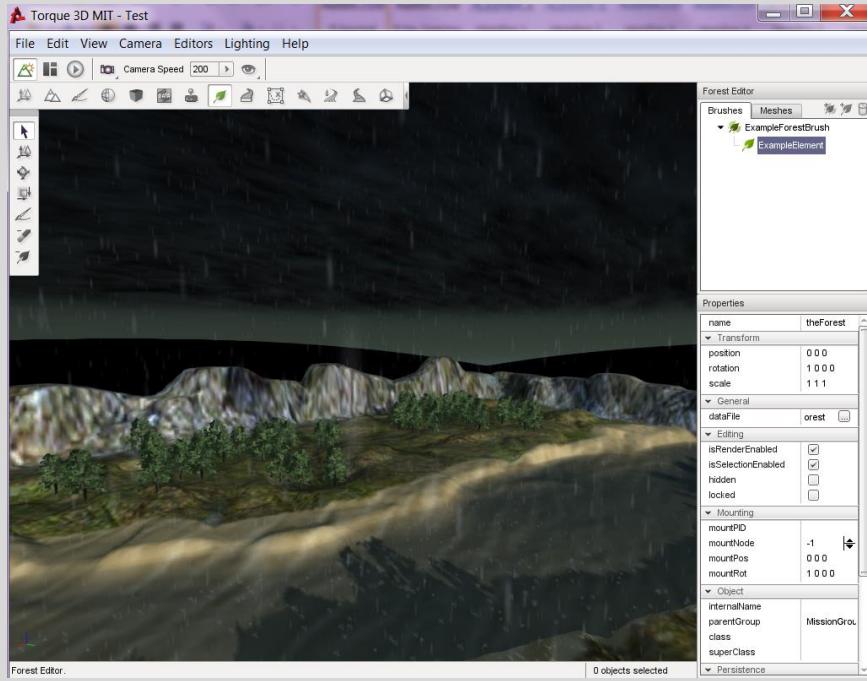
The second option here is located in the Environment category again, and it is the Water Block object. A Water Block is essentially a cube shaped box with water that fills it, and you can scale the cube to fit any rectangular or square shaped zone on your map, if it’s inside the box, there will be water there. This is a great option when dealing with things like lakes and small oceans.



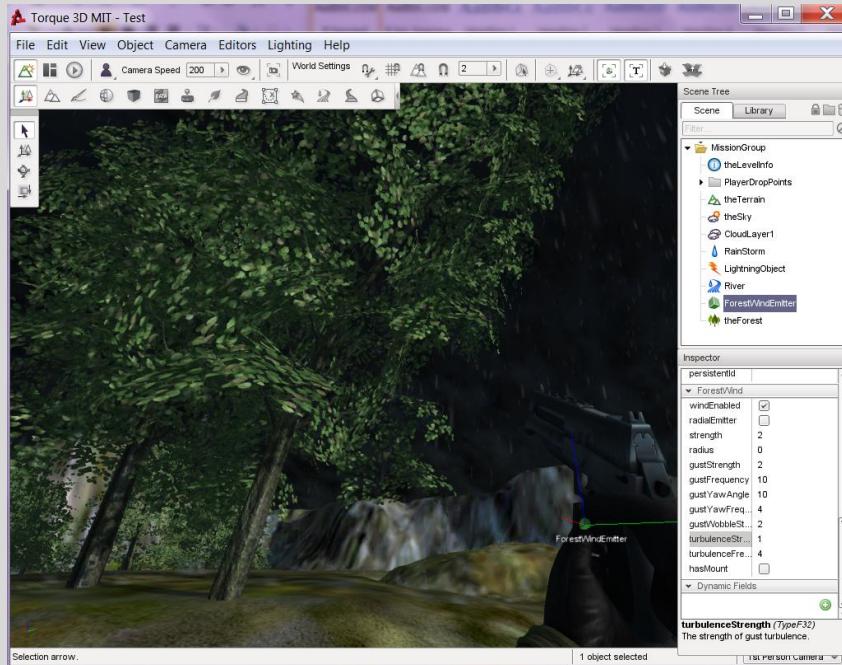
Again, this has some cons when dealing with things like rivers with turning sections. If you happen to have a low-lying area that unfortunately fits in the same space, it will also unintentionally have water inside of it. For our purposes, we’ll stick with the River Editor version, but just know that this is an option for you as well.

Forest

The next thing on our design concept to take care of is the forested areas marked by the green zones on the diagram. To do this, we’ll open up the Forest Editor and paint the zones we wanted with some trees. This is a fairly easy task with the forest editor. Basically open it up, select the brush paint tool and click a few times in the map in the areas where we want some trees. The editor will ask you to place a Forest Object on the map before you start painting. Allow it to do so and you’ll be all set to quickly paint the trees in the map.



However! We do have one little tiny problem right away. Pan your camera over to the trees, and you might notice something funny if you've been following along with the environment settings up to now. The trees are bending really far, which means the wind setting might be a bit too high, so let's scale the ForestWindEmitter effect down a bit.



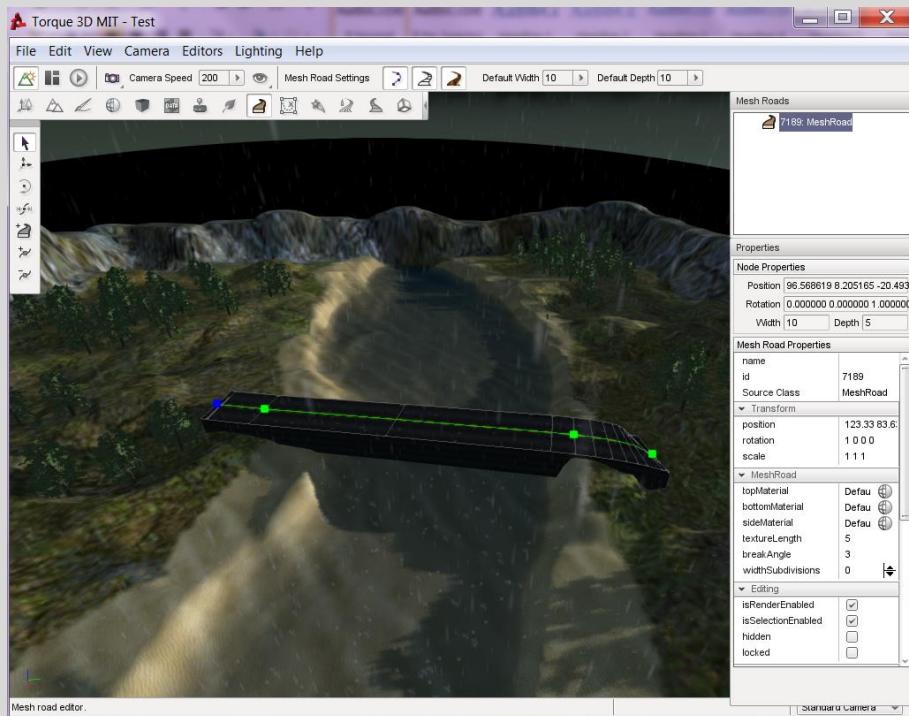
Now we have an active forest that is properly affected by the wind storm on the map for our players to run through!

Adding some Objects

Up to this point, we've been working on the Environment of the map, the important layout concepts. Now let's actually work on some common objects we'll be working with in our game itself. However, before we do that, let's add the final object on our little concept.

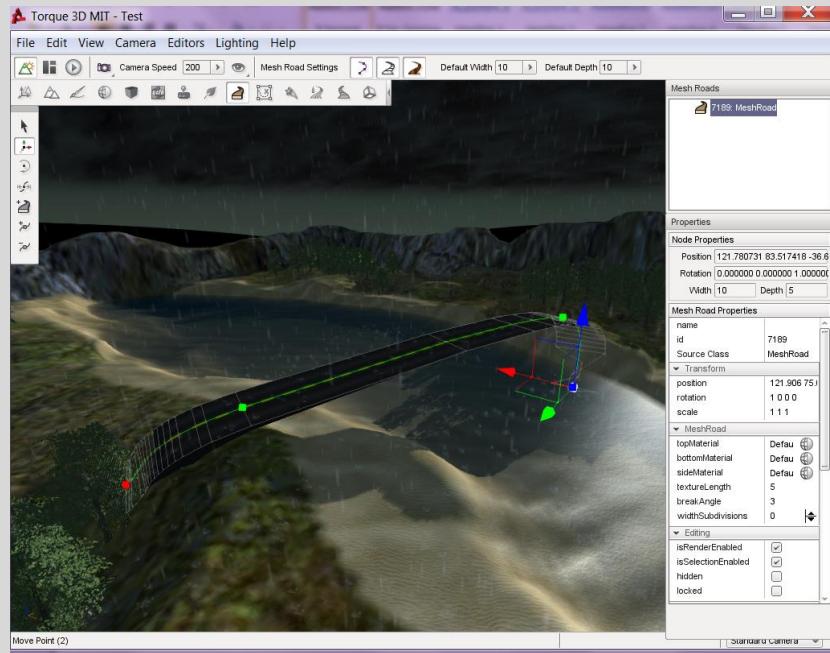
Bridges

Bridges basically allow players to traverse over water, or areas that would kill a player on your map (Chapter 11). There's plenty of ways to accomplish a good bridge, but let's use one of the other editors so you can learn some new tricks. Let's use the Mesh Road Editor to create a small bridge from one side of the map to the other. To do this, start by making four points as shown here:



Now, select the two inner points, and set the Depth parameter to 0.5. The under-section will now disappear allowing players to navigate across the bridge as well as under the bridge. Then, select the two outer nodes and move them underground to complete the structure.

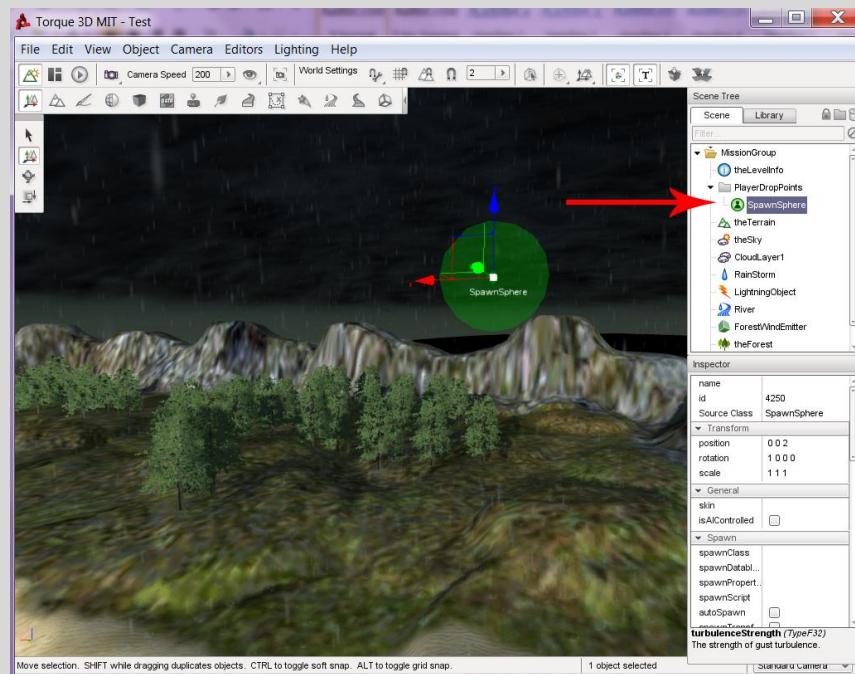
Feel free to add more points, or adjust how they move to create additional segments as you'd like for your map instance. When I was finished with my Mesh Road instance, I had something that looked a bit like this:



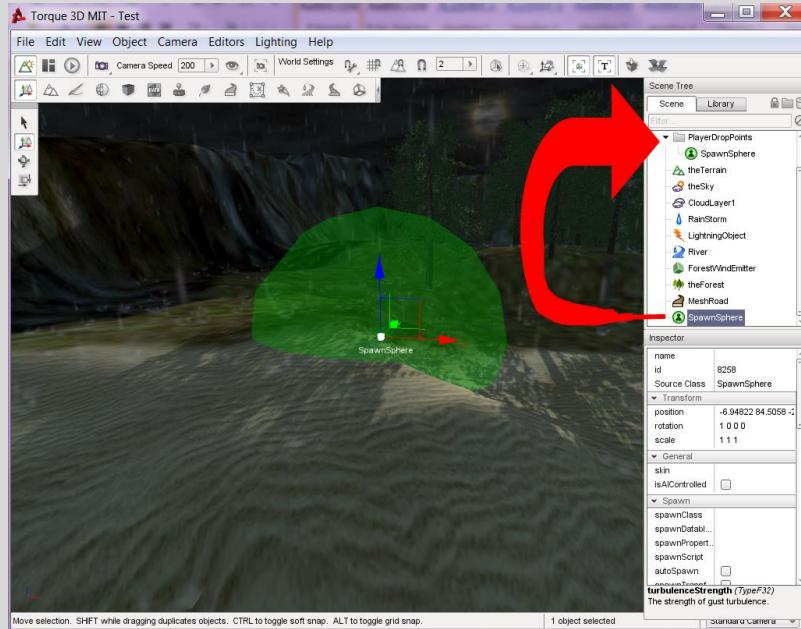
However, now you have something that looks like, well, a road that's out of place. To fix this, change the topMaterial to something like **Structure_plate**, to have a small metal bridge.

Spawn Points

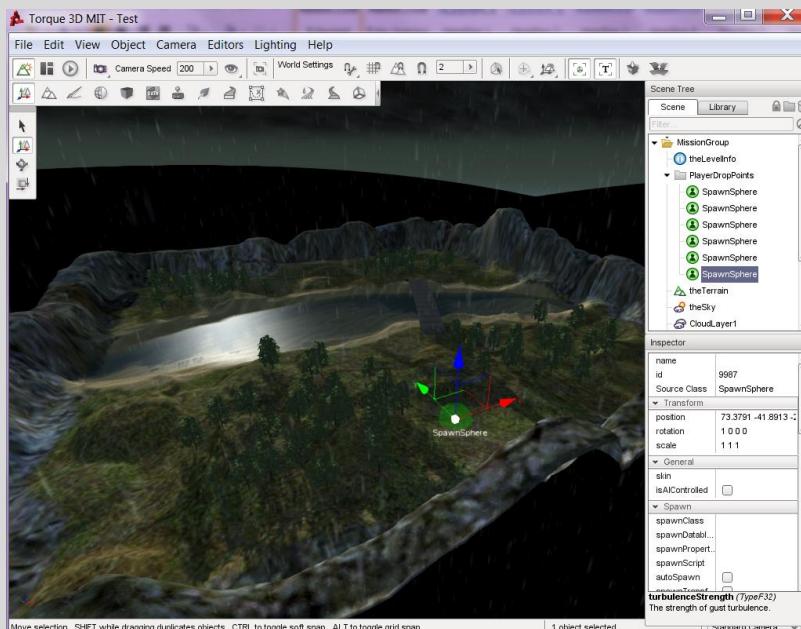
When making a map, you need to decide where your players need to spawn, and to do that we'll need the Spawn Sphere object. How many you have on your map will depend on the game, as well as what type of map it is. The easiest way to work here is to copy the existing one for the amount you need.



Simply select the existing spawn sphere and copy it (Ctrl + C) and then paste it (Ctrl + V) and move it to the desired location. By default, the game will add it to the standard MissionGroup instance. We want all spawn spheres to be in the PlayerDropPoints folder, so drag and drop the new instance to the folder:



Alternatively, you can right click the Folder named PlayerDropPoints and select the option that says *Add New Objects Here*. When you do this, the folder will change color to Yellow and any new objects you spawn will be placed in there. Continue to add spawn points to the folder until you have the desired number, and don't forget to have the original spawn point moved to be on the map itself. Once you're done, reset the active folder to **MissionGroup**.

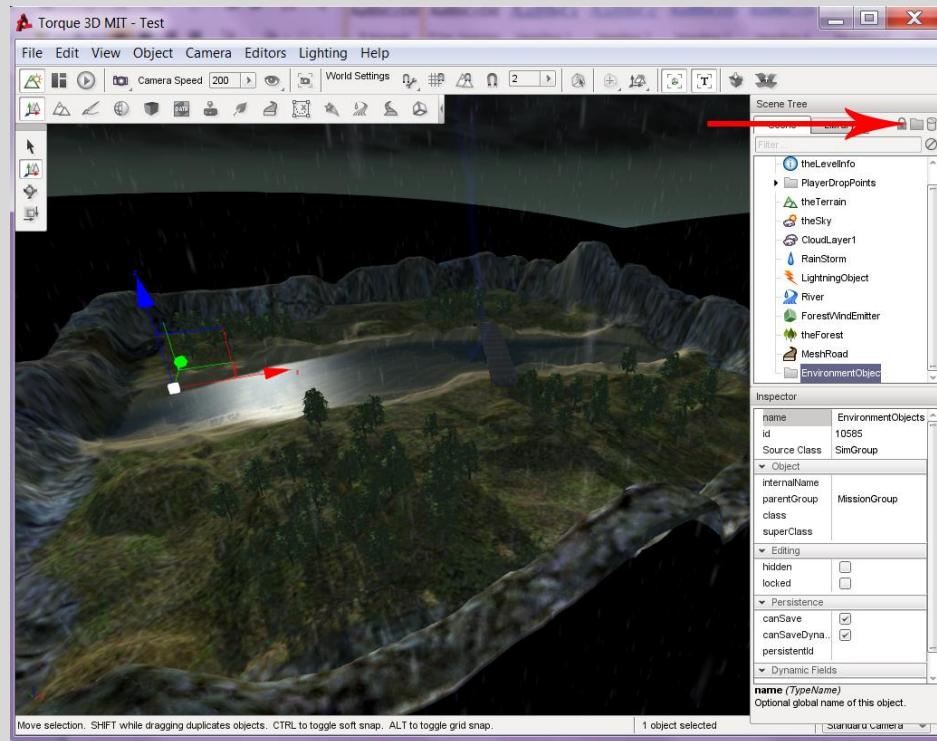


Once you've placed all of the spawn points you want on the map, you can collapse the PlayerDropPoints folder in the editor.

SimGroups: A World Editor Perspective

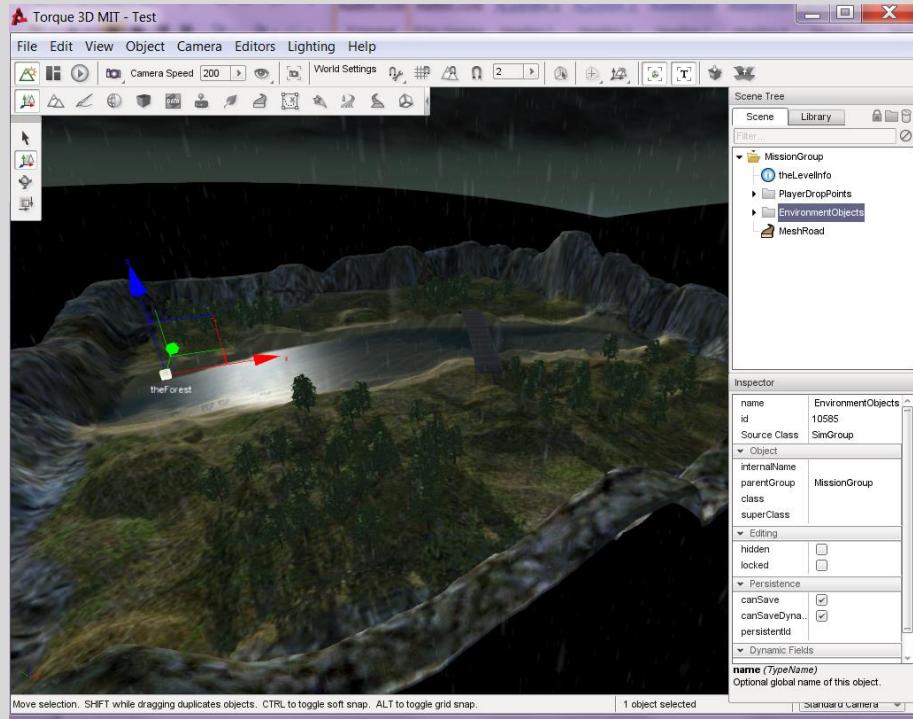
So, what are these little Folders? Well, this is actually something called a SimGroup. I'll have loads more on what a SimGroup is later on when we get into the Scripting side of the Engine, but essentially think of it as a group of objects that are stored together, which is why it's related to a Folder in the editor. It's usually a good practice to sort your objects into SimGroups to find the things you want to edit a little bit easier. You can also have SimGroups within SimGroups for complex object structures. A perfect example is the fact that we've been doing everything in a SimGroup, the MissionGroup.

Let's start with a basic example to show you how this works. There's a Folder icon in the Object Editor, click that to create a SimGroup. Make sure that it's in the MissionGroup folder, and then name it **EnvironmentObjects**.



To add objects to a SimGroup, you can either drag and drop them into the folder, or right click the Folder and select the *Add New Items Here* option. So, move **theTerrain**, **theSky**, **CloudLayer1**, **RainStorm**, **LightningObject**, **River**, **ForestWindEmitter**, and **theForest** to this new group. You can also hold Shift to select multiple objects at once.

You can then close the Folder for the new EnvironmentObjects and see how using SimGroups can help organize your space for the better:



A Rule I generally follow for making a level is to create a SimGroup for each individual category of object that is placed on the map. For Example:

- PlayerDropPoints
- EnvironmentObjects
- WeaponPickups
- Buildings
- ItemPickups
- MiscObjects

Naming & Level Parameters: What is theLevelInfo?

Now before we completely finish, we need to talk about one more item on the list, which is how to actually set up the level for your game. This basically fits the topics of level naming and parameterization. This is actually very easy to do, so let's quickly walk you through it.

Up to this point, you may have noticed the little object named **theLevelInfo** just sitting there in the MissionGroup. This is a level specific object that each level in the engine must have. It contains properties and settings for the entire map, such as Fog parameters, The Blank Color, and why we're here, the name and description of the level. So, scroll on down and name your level. Feel free to adjust the other map properties as well to give your map some more ambience. So with that, we're done with our first level! Be sure to save it otherwise you'll lose your work. You can find the completed version in the Chapter 3 folder included with this guide for a resource in case you've forgotten anything, or if you'd like to see some additional pointers to how to do things in the World Editor.

Additional Topics

Although we're done with the level itself, that doesn't mean our little journey through the world editor is completely done. There are still a few things to cover for future knowledge, and to show you some very awesome tips and tricks to use for your level design.

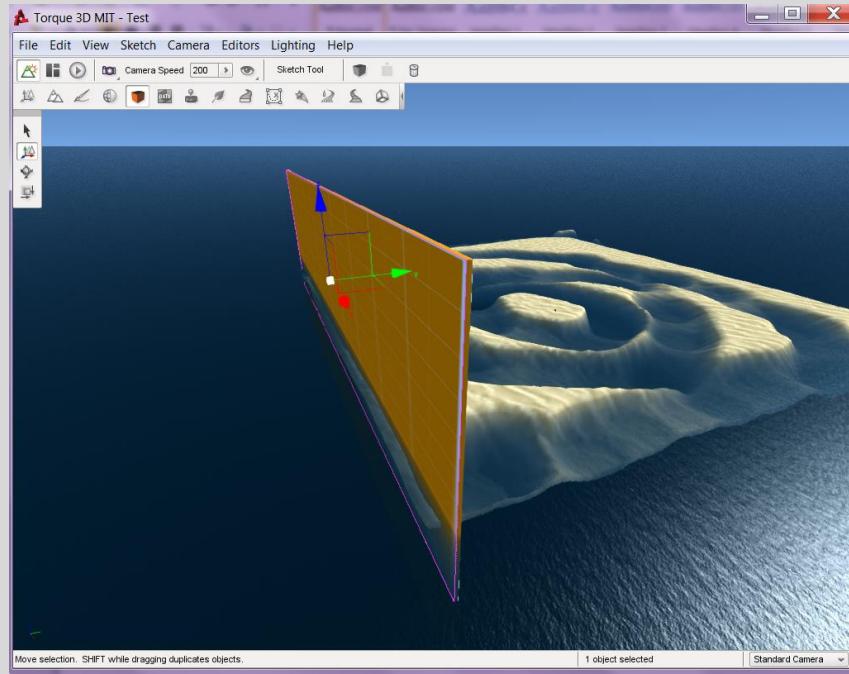
Barriers Revisited

Alright, so you know that we don't want players to fall off of the map into a deep dark pit of infinite falling, and by using a steep mountain slope, we can stop players from going out of the map. But now, let's take a completely different case. What if my map looks a bit like this?

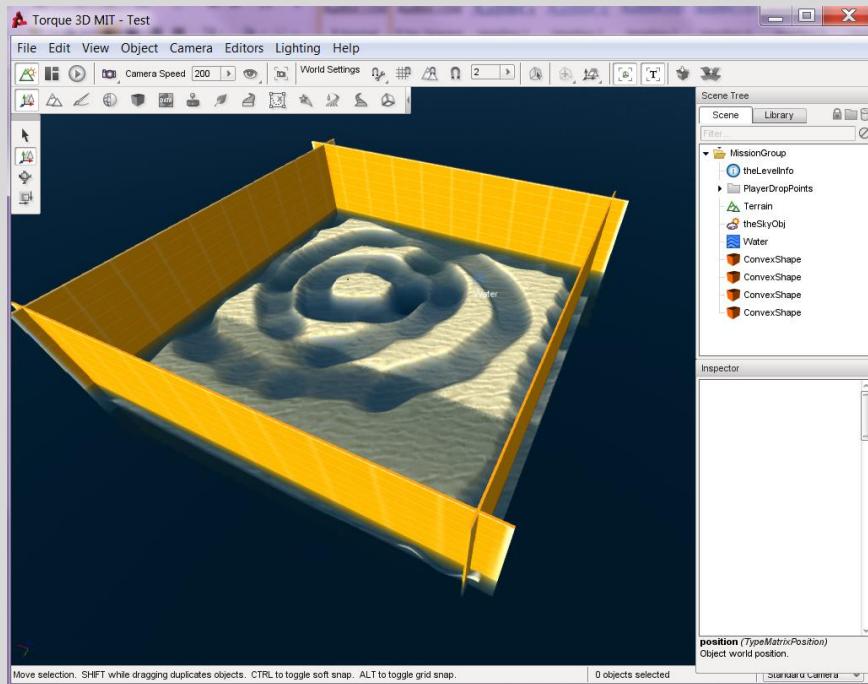


So, what do we do now? A large mountain range here would just look absolutely silly, since I like my large ocean plane. So let's talk about another way to create a map barrier, and that is something you've probably encountered before numerous times in other video games. And that is the concept of an invisible barrier that your player cannot pass.

Setting one of these up is surprisingly simple, and we can use two tools in the editor to accomplish this task. You'll want to grab the Sketch Tool here, and spawn a single block instance to work with. Once you do this, scale the X/Y and Z parameters to create a giant wall shape. It will have an orange texture, but don't worry about that for now. Once you do this, move it to the outer edge of the map where the terrain is about to end:



Copy/Paste and rotate to surround the map with these wall instances. Depending on the game, you may or may not need to place a roof over the top of the map instance. Something very important to know, is that these are literal walls, which means even projectiles and other objects will not be allowed to pass through the walls, so keep that in mind for your application purposes. For rotating, use "0 0 -1 90" to create a 90 degree rotation to shift between sides:



Now, to make the walls invisible, all you need to do is click each of the convex shapes and mark the checkbox that says *isRenderEnabled* to the off position and then de-select the object. The walls will then turn invisible in game, and will be impassable to your players.

Cleaning Up the Empty Zones

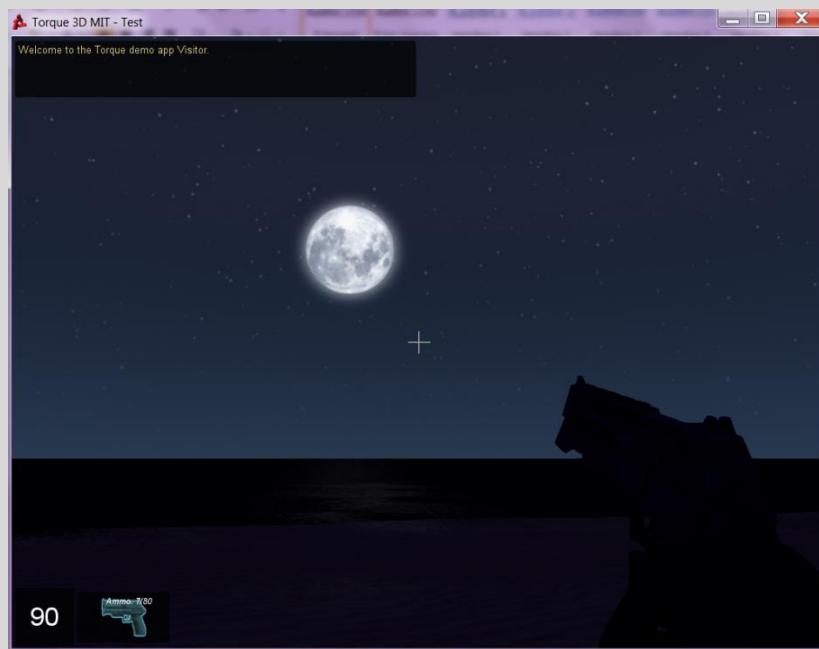
You may have noticed that my little island map example there doesn't have the black sections on the map signifying missing object areas. There is one way you can get rid of the black abyss of the missing zones on the map that is extremely simple to implement.

Under the environment tab, you'll see an object called the **Water Plane**. This is essentially an infinitely large section of water where the Z coordinate specifies the "top" of the water. For maps like the one we made earlier, you can simply create a water plane and move it completely below the terrain. For maps like my island map, I use this object as both the filling object as well as the water on the map.

You can also do things such as playing with the Fog Parameters to "fog" out the clear sections, or even cleverly adjust the Canvas Clear Color so it "fits" with the rest of your sky instance.

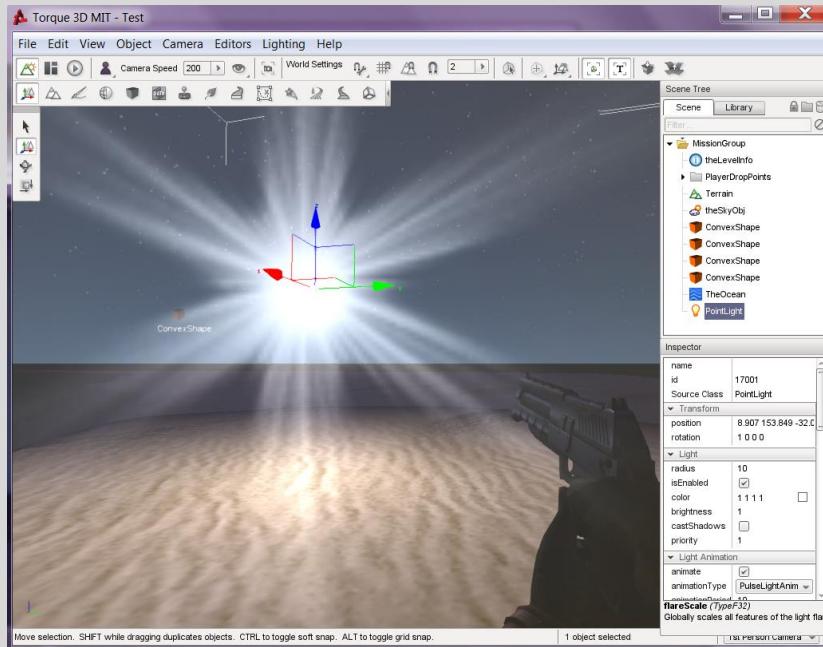
Lights

So, let's go back to our little island example, and then let's change the time of day to nighttime. This is what the map look like right when I do that (For reference: You can switch a map to nighttime in a map with a ScatterSky object by setting the Sun Elevation to a negative number):



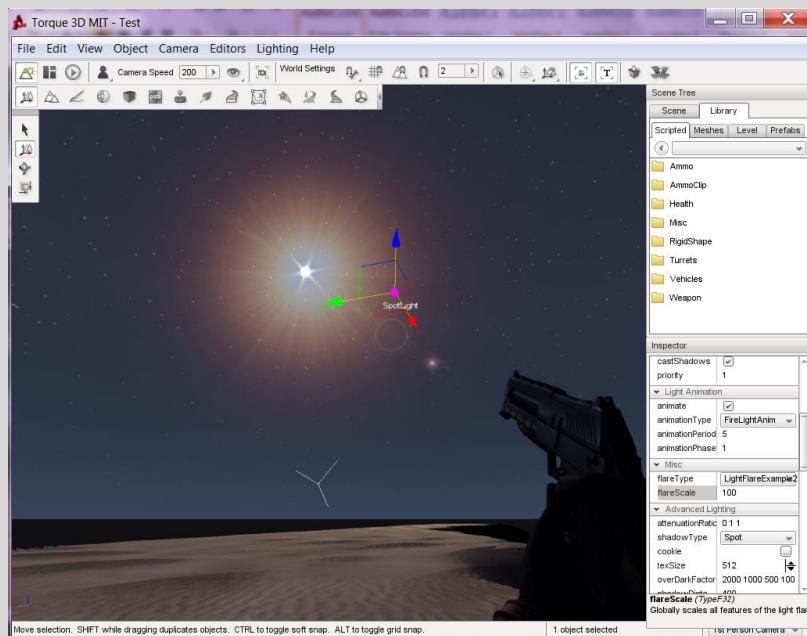
That's nice and all but there's only one problem, I can't see anything on the map anymore. So, in order to fix this problem, we need to provide the player with some lights to work with. The engine has a few different light source objects to work with; however, we're going to only talk about the Point Light and the Spot Light.

A Point Light is a single point that emits a radial source of light. You can specify all of the standard lighting parameters for the point light. For example, here's a flare instance that is at its peak intensity for the small area:



You should use point lights in areas of the map that are darkened to help players see where they are going. Also, point lights are great when you have things like lamp instances that need light to shine from the lamp into the world.

A Spot Light instance, projects a cone from the source point in a forward vector pointing outwards. Anywhere that is inside the cone will be lit by the light.

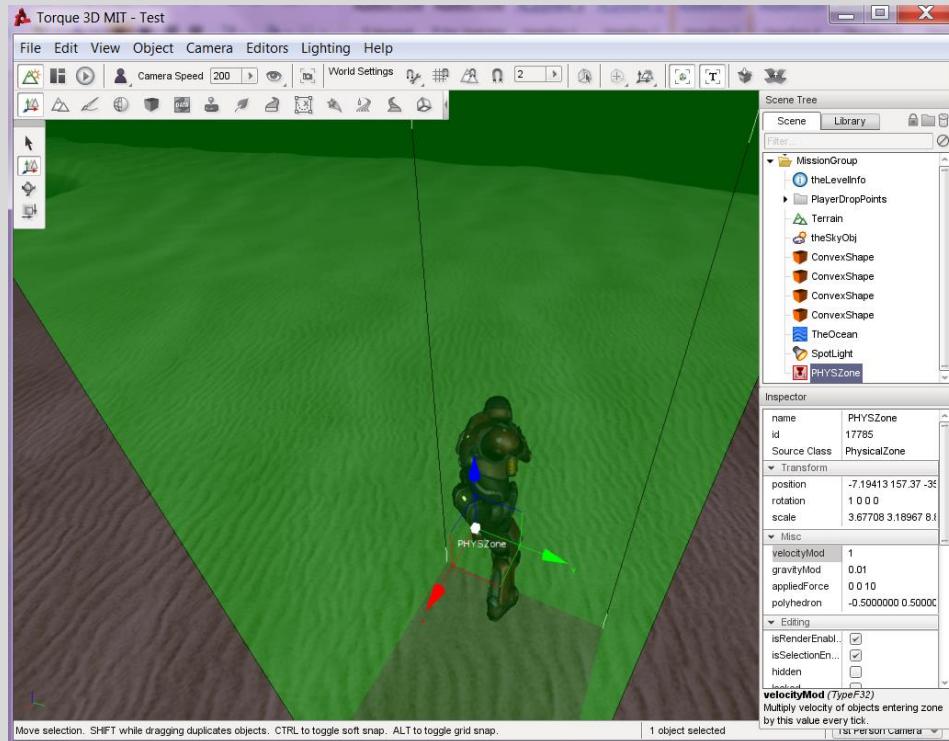


Spot Lights can also have the standard lighting parameters and are good for instances such as watchtowers or large aerial sources of light that project down to the entire map below, such as a close star, or a meteor.

Physical Zones

Our final topic is that of the Physical Zone object. A Physical Zone is a special object that manipulates the overall movement parameters of an object that enters the field. For example, you can adjust the effect of gravity to a player that enters it to have no gravity, or very large gravity values. The Physical Zone can also apply a force to an object inside of it, or manipulate the overall speed of an object inside of the field.

You can find the Physical Zone object under the Level/Level tab in the Library. Here is an example of an active Physical Zone suspending a player in a floating state:

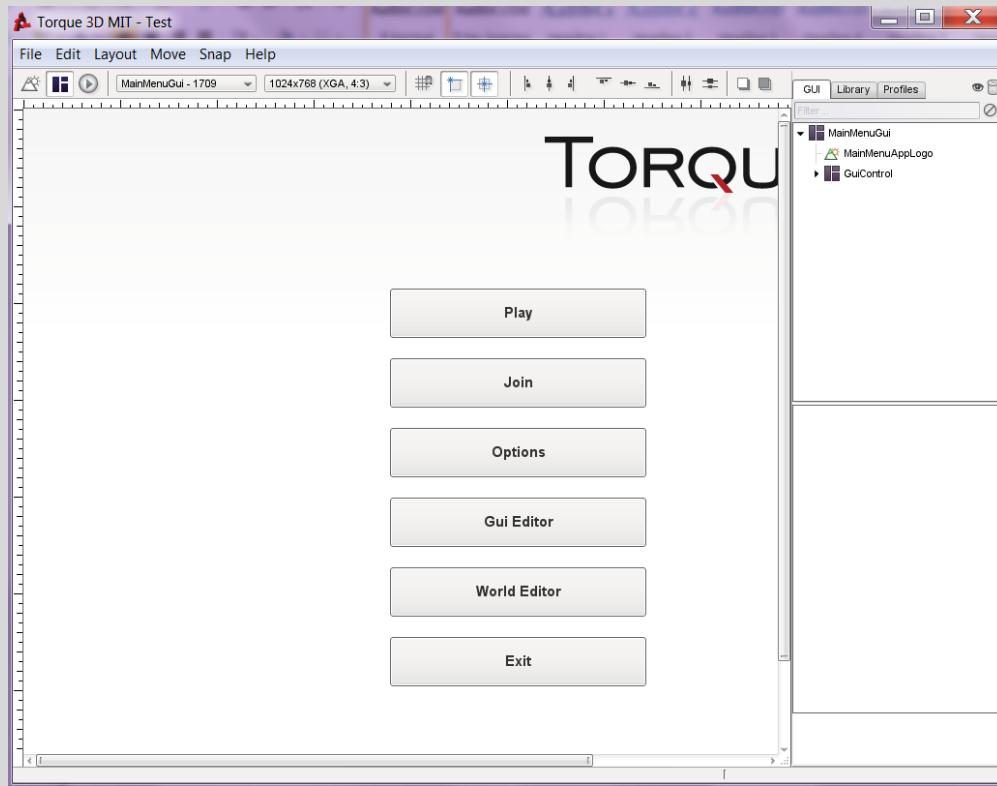


By combining a Physical zone with something like a particle emitter or other effects, you can create objects such as gravity lifts and portable launchers for your game objects.

Chapter 4: The GUI Editor

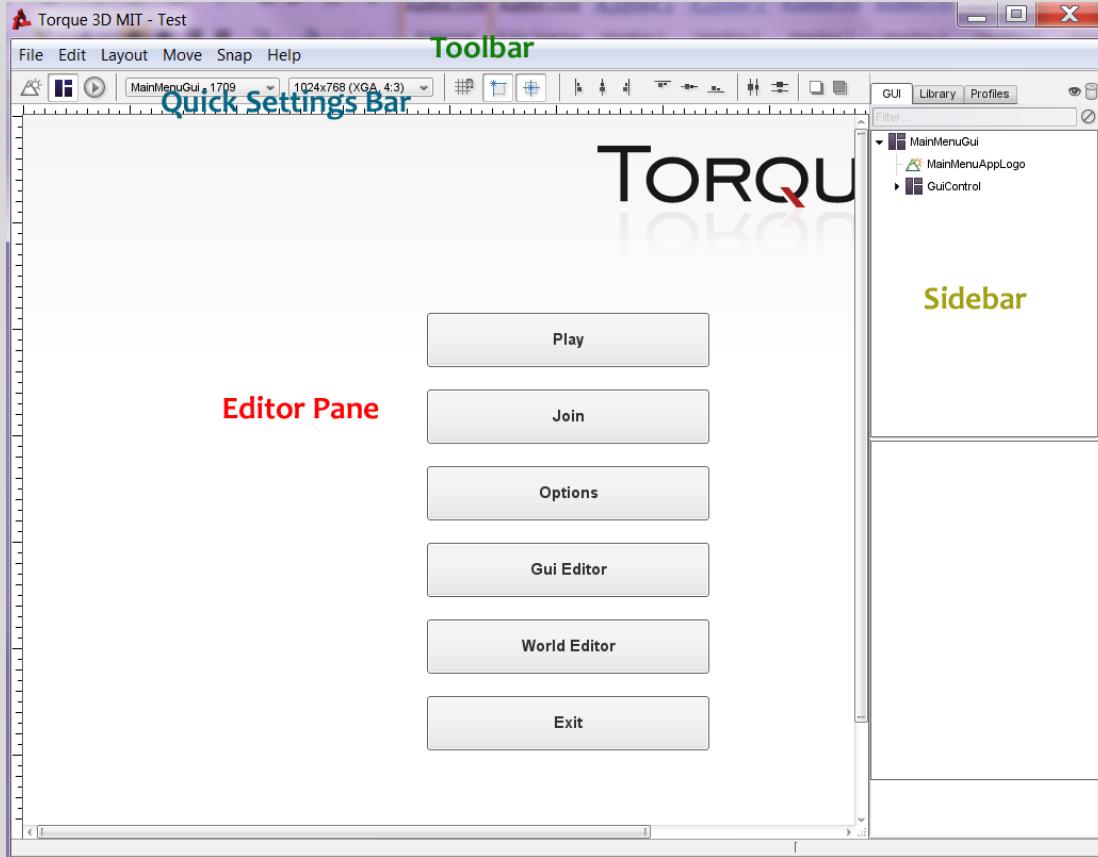
The next topic of the guide is to talk about Torque 3D's other in-game editor system, the GUI Editor. A GUI or a Graphical User Interface, is a 2D Screen element that is viewed by the player when in the game instance, in layman's terms, this is what is shown on the screen inside the Torque 3D window. The GUI Editor will allow you to edit these instances and create new elements to be displayed on the screen when and where you want them.

So, go back to the main menu of the Engine Application. From here you can either click the button that says GUI Editor, or press F10 on your keyboard to launch the editor. By default, it will select the current GUI you are viewing, which in our case will be the MainMenuGui.



From an editor perspective of things, the GUI Editor is much easier to work with compared to the World Editor since there isn't as much to worry about, however, there are still some design challenges and tough aspects to learn from this tool, and that is the purpose of this guide. As we did before, we're going to walk through the editor itself before we actually start working with it.

Let's start by again identifying the aspects of the GUI editor. There are some similarities and differences between the World Editor and the GUI editor, but it should be rather easy to pick up learning one once you've got the other down.



So these are the sections of the GUI Editor. So like we did with the World Editor chapter, let's go into more detail regarding each of these sections now.

The GUI Editor Toolbar

The first part of the GUI Editor we're going to talk about is the Editor Toolbar.



Like we did last time, we'll work from left to right here through the menus. The *File* menu contains generic options and settings for saving and loading your work while in the editor. The first option is the New Gui (Ctrl + N) command which empties your current canvas to start work on a new GUI. You will also be given a prompt to select the base dialog to start from, for most cases, you will be fine leaving it as GUIControl. The next command is the Open (Ctrl + O) command which allows you to open a .GUI file into the editor to make changes to a GUI that is not loaded in the engine yet. For all of the GUIs that are already loaded, there is the option on the quick settings bar to switch between the GUIs. The Save command (Ctrl + S) is self-explanatory; all changes to the existing GUI will be changed in the existing file. The Save As command (Ctrl + Shift + S) allows you to save your current GUI to a new file. The Save Selected As (Ctrl + Alt + S) command allows you to save only the selected portions of the GUI to a new file. The next command is the Revert GUI command, which works as a full undo of all of your

changes that are unsaved, and reloads the file. Afterwards is the Add GUI from File command which loads the aspects of a .GUI file into the GUI you are currently editing. The following option has to do with a third party program, and is beyond the scope of this guide, so the last two commands are Close Editor (F10) which closes the GUI editor, and the Quit command which exits the application.

The *Edit* menu is used to control aspects of editing single controls or working with the overall editor in terms of quick changes. The first five commands are standard editing commands: Undo (Ctrl + Z), Redo, Cut (Ctrl + X), Copy (Ctrl + C), and Paste (Ctrl + V). After those, we have Select All (Ctrl + A) and Deselect All (Ctrl + D) which are self-explanatory. The next command is the Select Parent (Ctrl + Alt + Up Arrow) which selects the GUI control that is above the current one in hierarchy. This is followed by Select Child (Ctrl + Alt + Down Arrow), which selects the GUI control below the current one in hierarchy. The next two options are Add Parent(s) to Selection (Ctrl + Alt + Shift + Up Arrow), and Add Children to Selection (Ctrl + Alt + Shift + Down Arrow), which are used to move GUIs into a hierachal tree structure. Afterwards is the generic Select command which is used to select/deselect individual GUI elements. The next two commands are Lock/Unlock Selection (Ctrl + L) which locks an element from being edited, and the Hide/Unhide Selection (Ctrl + H) which toggles visibility of a GUI element. The next two commands are Group Selection (Ctrl + G) and Ungroup Selection (Ctrl + Shift + G) which merges GUIs together into group instances. The Full Box Selection option allows the editor to select every GUI instance that is touching any point inside the selection box, even if it's not covering the entire control. And lastly, the Grid Size (Ctrl + ,) option, allows you to manipulate the size of the editor's grid for Snapping controls.

The *Layout* menu is used to adjust the location placement of GUI controls and the overall application settings of the GUI Editor. First off, is the Align Left (Ctrl + Left Arrow) command which tells the editor to align all elements to the left of the placed position. The Center Horizontally command sets the alignment to the middle, and the Align Right (Ctrl + Right Arrow) command aligns everything to the right. The next three commands deal with vertical alignment and are Align Top (Ctrl + Up Arrow), Center Vertically, and Align Bottom (Ctrl + Down Arrow). The next two commands will apply spacing between controls either horizontally or vertically when being placed. The Fit into Parent(s) command will adjust the positioning of the selected control(s) to fit into the parent control(s). The next two options are Fit Width into Parent(s) and Fit Height into Parent(s) which adjusts the width and height of the selected controls to fit into their parent control(s). Lastly, we have Bring to Front and Send to Back which adjusts the render order of the controls so certain GUI elements have render priority over others.

The next command bar is the *Move* bar, which contains some quick move commands for your controls. They are specified as Nudge Left (Left Arrow), Right (Right Arrow), Up (Up Arrow), and Down (Down Arrow) to slightly move the selected control(s). Then, there is Big Nudge Left (Shift + Left Arrow), Right (Shift + Right Arrow), Up (Shift + Up Arrow), and Down (Shift + Down Arrow) which applies a larger move to the selected control(s).

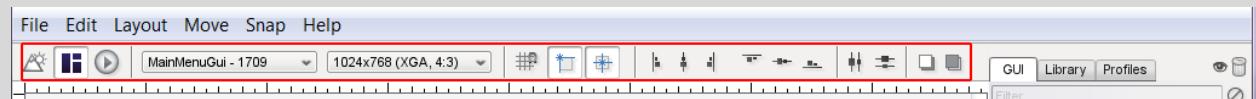
The last command bar of our interest is the *Snap* bar, which allows the editor to snap controls to certain aspects to give a better fit and layout of the GUI itself. The Snap Edges (Alt + Shift + E) command will snap the selected control to edges of other controls when editing. The Snap Centers (Alt + Shift + C) command will snap the selected control to the centers of other controls. The Snap to Guides command

(Alt + Shift + G) will allow the editor to snap the control to the guides on the editor that are placed by the user. The Snap to Controls (Alt + Shift + T) command allows edited controls to be snapped to other controls. The Snap to Canvas command allows the control to be snapped to the edges of the editor canvas, and the Snap to Grid command will allow you to snap controls to the editor's grid when it is enabled. The last two commands allow you to visually see Guides and to remove all of the guides that are on the editor.

There is also a *Help* command bar in the Toolbar of the editor, but it only contains some quick links to view the documentation and other relevant help pages.

The Quick Settings Bar

The next thing we're going to talk about is the Quick Settings bar for the GUI Editor. Unlike the World Editor, this is a constant bar that doesn't change under any circumstances, so we can talk about it right here.



The first three options should look familiar to the World Editor, since they're the same. They simply toggle between the editors and the game instance itself. The next is a dropdown box that contains a list of all of the **loaded** GUI instances (Chapter 10) to select from in the editor. Afterwards is the dropdown box with a list of resolution options for the GUI Editor. The next three options are for snapping controls, they enable in order, Snap to Grid, Snap to Edge, and Snap to Center. The next six options control alignment options for your GUI Instances, and they are as follows: Align Left, Center Align (Horizontally), Align Right, Align Top, Center Align (Vertically), Align Bottom. Finally, we have two button controls to send a GUI control to the front, and to the back.

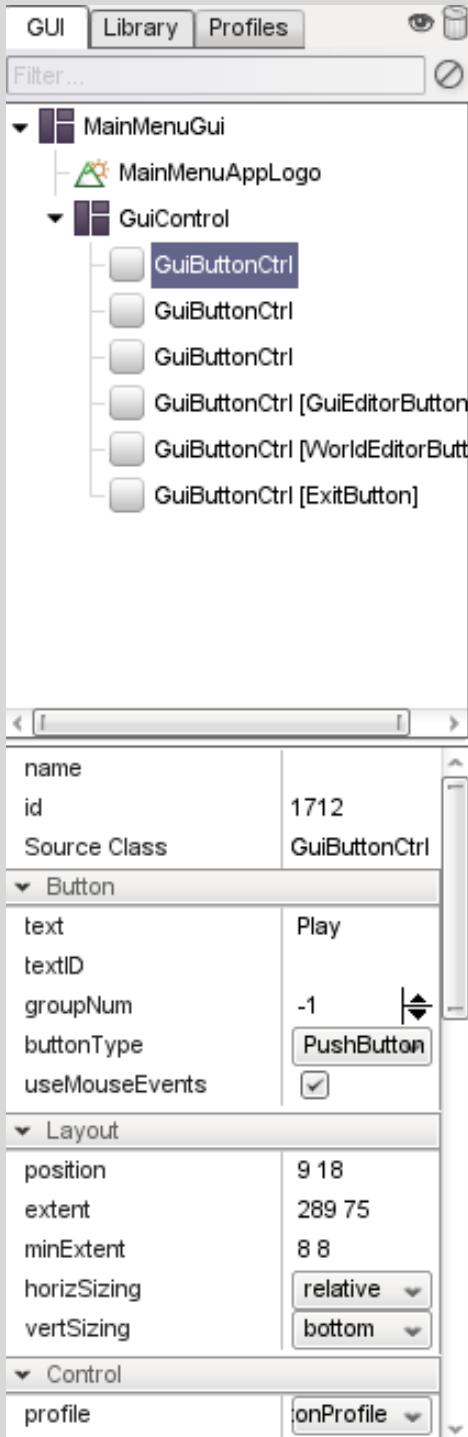
And that's all there is for the GUI Editor in the quick settings bar. Most of the time you'll only need to use a few of these tools since the toolbar can handle most of the other controls of this editor instance.

The Sidebar & Options

Unlike the World Editor, the GUI Editor's sidebar is actually used to perform a wide variety of tasks related to working with this editor. Most of your time will be used working with this part of the editor. The GUI Editor's Sidebar is split into three separate tabs, each with their own commands and options. The tabs are as follows: GUI, Library, and Profiles. To help you learn each individual section, this guide has split them all into separate sub-chapters, starting with the GUI tab.

Inspector (GUI)

The GUI Tab, or the Inspector, is used to manipulate properties and settings of each of the individual controls in the GUI Editor.



This is the Inspector of the GUI Editor, or the GUI Tab. This is very similar to the Object Editor of the World Editor, it lists all of the controls in the GUI as well as their individual settings and properties. You can select an individual component in the list and then manipulate the properties and settings in the section located below the list.

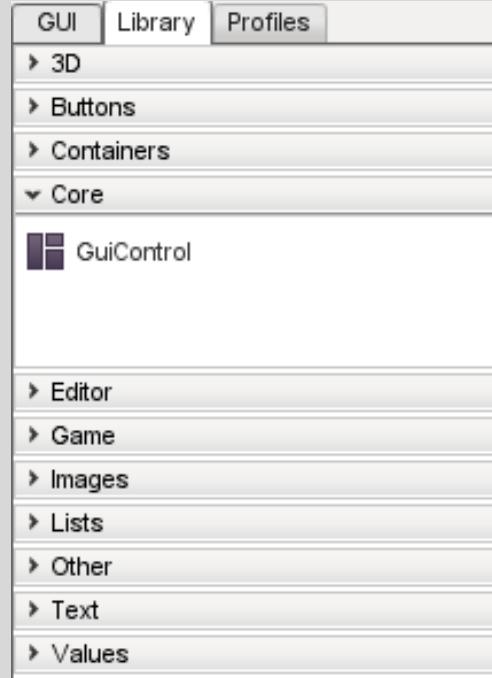
The two buttons on the right allow you to toggle the visibility of an individual control and remove a control from the GUI. You may also filter the list of your GUIs by typing in the name of the object you want to edit, or by typing in the name of the control itself, for example: **GuiButtonCtrl**.

In the inspector itself, you'll have all of the general properties and settings that are common to all GUI instances such as the name, identification number, and source class of the GUI instance, and then you'll have controls and settings that are uniquely specific to each GUI instance. This is where you can set up unique looks and properties of each of your controls in the GUI itself. Finally, we have an additional group of generic properties and settings used for layout parameters, positioning, and extent of the control, which basically specifies how large of a space the control holds. The horizSizing and vertSizing parameters are special settings used to specify how a GUI behaves on differing screen sizes. We'll cover this topic in much more detail later on when we actually make our own GUIs, but just know it's important. Finally, we have the control section which contains important settings such as the profile instance of a GUI as well as the specified command the GUI executes when it is "activated", such as a button press event for a button. We'll cover that topic in Chapter 10 of this guide, so you can safely ignore it for now.

Library

The next section of the editor is the Library. This is where every GUI instance in the engine will be located. Basically, you select the control you want to place in your GUI from the library and it places it on the screen so you can edit it.

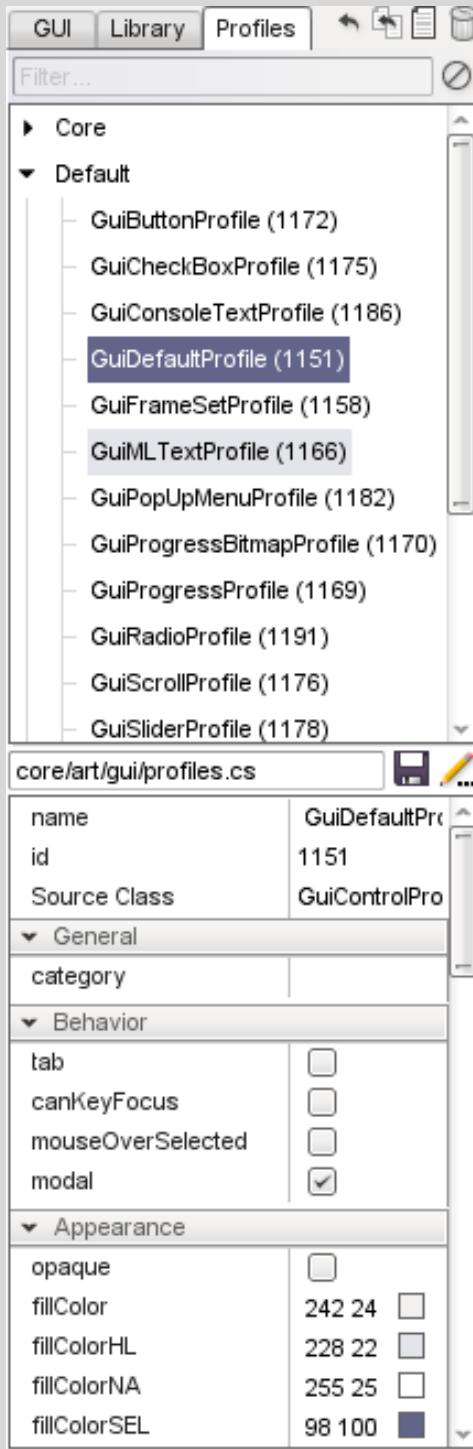
The library of the GUI editor is split into sections, based on what type of control you have. In the engine's definitions, each coded GUI is set to be in a section, and when the editor is opened, it is populated from that list of GUIs. To add a control to the screen, you can either double click the control of interest to place it at the origin, or click and hold over the name and drag it to the desired location on the screen. Either of these two will place the instance on the screen to be edited in the Editor Pane, or the GUI Inspector tab.



Profiles

The final section of the GUI Editor is the profiles tab. A profile is a list of default properties and settings that are applied to the GUI Instance. This covers things like fonts for text GUIs, modulation of color for button instances, and much more.

This editor will allow you to both manipulate existing profiles in the engine, as well as define a new profile to be saved to a script file for future use by the editor or your game instance. We'll cover profiles in much more detail when we get to Chapter 10, and we'll also show you how to make sure that your profiles are loaded in the correct order so the GUI instances can actually use these profile definitions when they're supposed to be activated.



The four buttons along the top bar of this profile editor are as follows: The first button reverts the profile to the last saved file instance, the second button creates a new profile by copying the selected one, the third button creates a brand new profile, and the final button deletes the selected profile. Like the Inspector, you can search for a specific profile by typing the name into the box.

Inside the editor itself, you'll see a few categories to choose from, like a GUI control, the profiles are syphoned off into individual groups and then placed in a list so you can select the instance you want to manipulate. Once you select an instance it will populate the inspector section below.

Also like the general inspector of the GUI editor or the World editor, you'll find common settings and properties loaded into the inspector. You can manipulate any existing profile or create a brand new profile to work from in this section. Be sure to click the save button whenever you're done making changes or you'll lose your work with the profiles because the profile editor is a separate entity compared to the GUI editor, you'll need to make sure your progress is not lost.

Building a GUI... From Start to Finish

So that's all you need to learn about the editor itself. As originally stated, in most senses, the GUI editor is a much more easy system to work with compared to the World Editor, however there are still some challenges you'll need to learn to overcome when working in the editor. We'll get to those

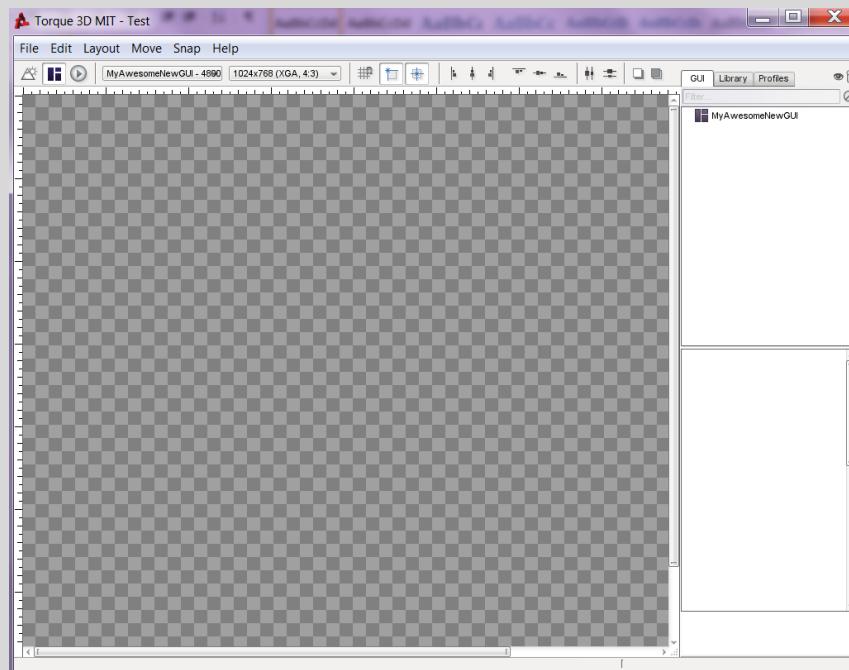
individual challenges and solutions here in a bit, but let's start by introducing how to actually work with the GUI Editor by an example of creating a new GUI.

Getting Started

The first thing you want to do is create a new GUI in the editor, and give it a relevant name. Leave the source class as GuiControl and click the create button:



Once you click the create button, the GUI Editor will empty and have a checkerboard plane and then only one object instance in the editor list:



Now before we start throwing objects in there, let's talk about the first major challenge of creating a GUI instance. This is the challenge of differing resolutions for target platforms.

The Resolution Problem

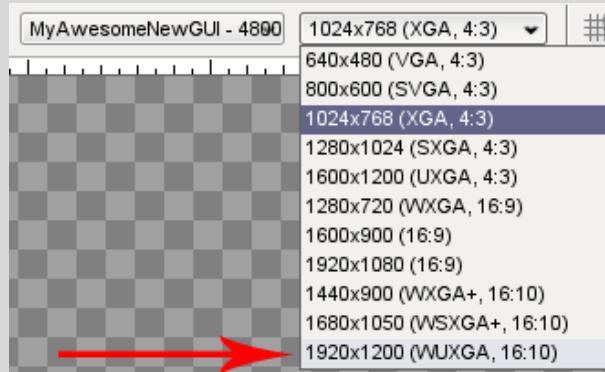
When you deploy a game instance, unless you're deploying to one specific device that has the same properties as all others (such as a handheld game platform), you'll be dealing with different resolutions. When you play a computer game, most people will have a monitor that has a different overall screen size compared to your screen size. If you don't account for this, the GUI on a different

person's computer could look messed up compared to your computer. There's a few easy ways to avoid this problem, and we'll cover all of them now.

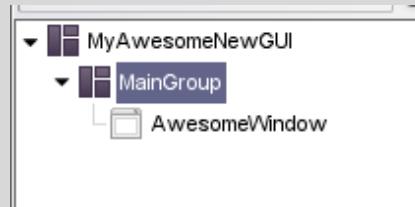
Editing Rules to Prevent Resolution Errors

This first thing we want to do is to teach you how to prevent this problem. To do this, there are a few rules you should always follow when creating a GUI in the GUI Editor.

- 1) Always edit under the largest screen resolution available. By doing this, you can size the GUI Elements down later on to fit the screen, this doesn't work the other way around.



- 2) Contain all of your elements in relevant sub-groups. By doing this, we can take advantage of a little trick to size the group which will also size the elements in the group.



- 3) Make use of the sizing parameter **Relative** when using horizSizing and vertSizing to force the engine to fit the GUI element to its relative resolution.

name	MainGroup
id	4973
Source Class	GuiControl
Layout	
position	0 0
extent	1920 1200
minExtent	8 2
horizSizing	relative
vertSizing	relative
Control	

As long as you follow these rules when editing your GUIs you'll ensure to avoid the dreaded resolution problem when you go to deploy your game later on.

Fixing a Resolution Error

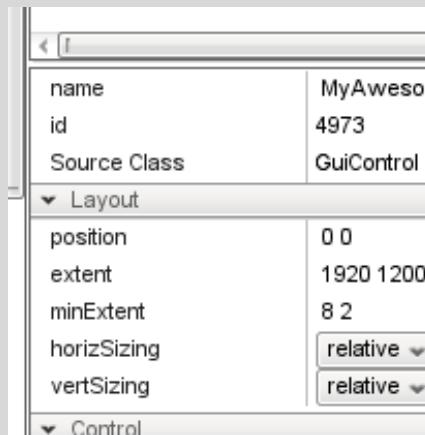
In the event you forget to do one of these things and then become struck by the Resolution Error you can still recover your work on the GUI. All you need to do is re-open the editor and the relevant GUI under the parameters you originally edited on. For example, if I made a GUI in fullscreen but am now in Windowed mode, you'll need to close the editor, go back into fullscreen and then reload the GUI in the editor in fullscreen mode. Then, you can apply the rules mentioned above to the GUI to fix it.

Containers and Windows, Screen Sizing

So, now that you know how to prevent the Resolution Error from occurring let's get started on our GUI. The first thing you want to do is spawn a GuiContainer instance, you can find this in the Library under the Containers tab. Before we continue you need to learn some naming conventions. As you will learn in the coming chapters, and I may have briefly mentioned this in Chapter 3, no two objects can have the same name. To make sure this is never the case, you should adapt this naming convention. When you name a control in your GUI, start by retyping the original name of the GUI, in my case **MyAwesomeNewGUI**, then type an underscore (_), followed by the desired name of the control.

With the GuiContainer spawned, name it as the primary group, for me:

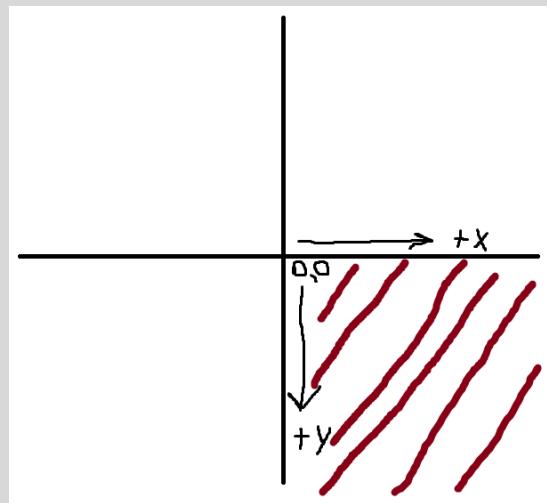
MyAwesomeNewGUI_MainGroup. Since we're following the rules to prevent Resolution errors, set both horizSizing and vertSizing to **Relative**. Then, set the position of the control as **0 0**, and the extent to match that of the resolution in the drop down box.



This will set up a group instance. A GuiContainer or a GuiControl object can be used to set up a grouping instance, or basically a place where you can create other GUI Elements to fit on the screen. By setting the container to have a relative size, when the GUI is opened on a computer, the container will scale up and down based on the resolution, and all of the GUI elements on the screen that are inside this container will also scale to fit into the GUI.

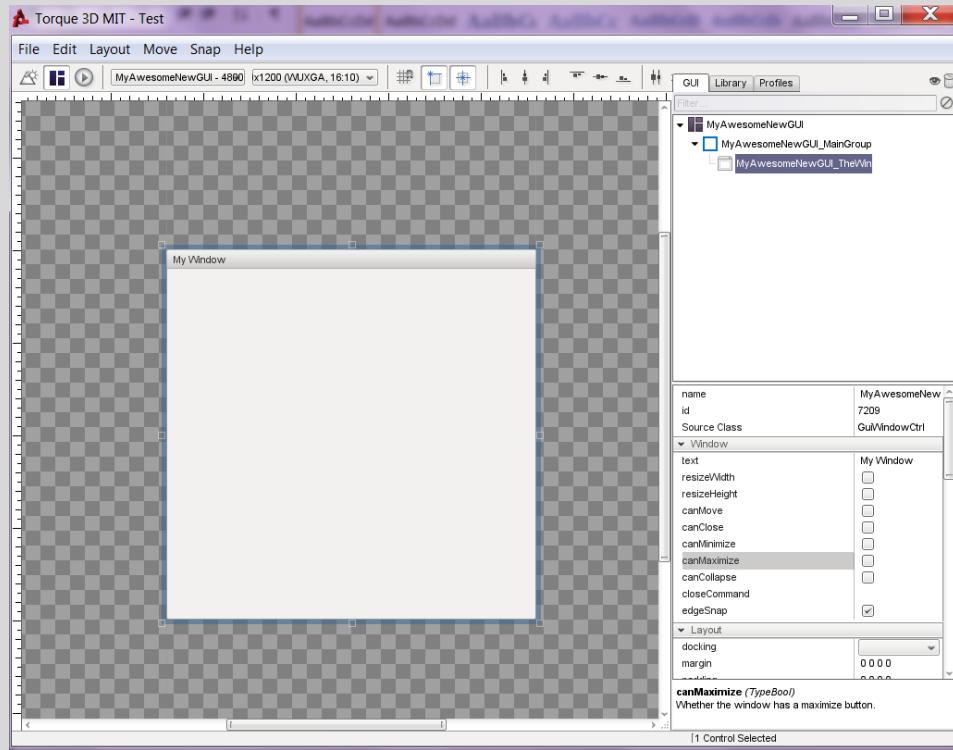
When we place objects inside a container, they become “relative” location and scale wise to the container, compared to the GUI itself, so for example, if inside my container instance, I create a 400 x 400 window, and then put a button in the window, a button at 0 0, may not match the actual position 0 0 on the screen, however it will be at 0 0 relative to the window instance. You need to be mindful of this fact when you create GUI instances and try to manually assign a position to the individual element, it will have a relative position to the GUI object that is the parent of the control.

Now, let's actually keep working here. Let's make a 400 x 400 window in the middle of the screen. However, there is one thing you must be absolutely mindful of when working with screen positions in computers, and that is the axis reversal of 'y':



The area with the brown lines represents the screen in our case. So, smaller values of Y actually represent a higher position on your screen, so be sure to keep that in mind when working with the GUIs here.

Before we continue, you'll need some basic math. My maximum resolution is 1920 x 1200, so the first thing I do is determine the middle of the screen, or basically half of these numbers, which comes out to 960 and 600, now to place the GUI itself in the middle of the screen, we need to subtract half of the desired size from the “extent” of the GUI, or basically the size. Our size is 400 x 400, so I need to subtract 200 from those two numbers, so the position of the new window will be 760 and 400, and the extent will remain as 400 400.



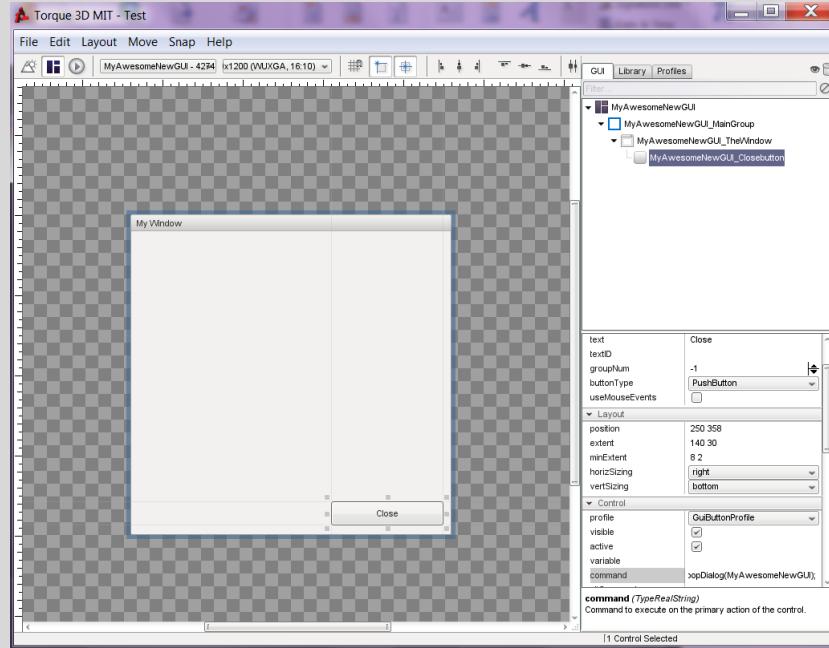
You may need to move your editor pane into view if you have a very large resolution, like I had to, but the control will now be situated in the middle of the screen. Like we did before, set both horizSizing and vertSizing to relative, and then feel free to edit the individual settings and parameters of the window to suit your needs.

Buttons and Button Variants

So let's talk about buttons first, since everyone loves to click on buttons in their GUIs. A button instance in the engine can come in a few forms, a click button, a radio button, and a checkbox button. The latter two require an additional step to set up, but let's start with the click button.

Click Button

A click button is a simply button with one operation, you click on it, and something happens when you click it. Let's add one to the GUI to work as our close window command. So in the library, open up the buttons section and drag in a **GuiButtonCtrl** to the dialog. Name it **MyAwesomeNewGUI_closeButton** or something similar, following our naming schemes. In the inspector for this GUI, you'll see an item in the list called **command**. We'll have much more on this in Chapter 10 when we cover more advanced GUI stuff, but for now, fill in the field: `Canvas.popDialog(MyAwesomeNewGUI);` replacing `MyAwesomeNewGUI` with the name of your dialog. You can mess with the other parameters of the button as you feel fit.



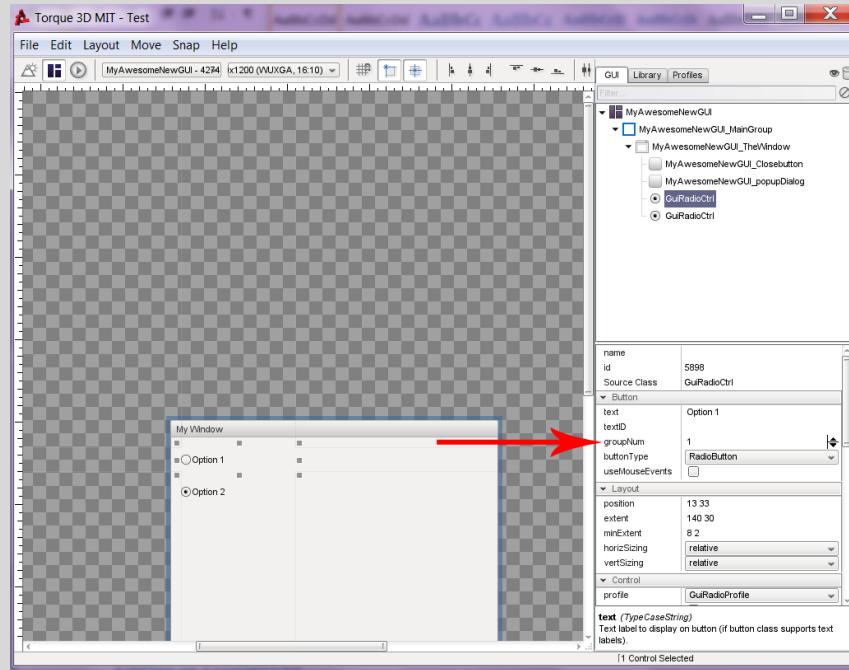
Again, be sure to set those scaling elements to relative here to ensure the positioning isn't messed up.

Now, let's make something else with the click button. Create another button and name it `x_popupButton` (replacing `x` with the name of the gui), and set the command to the following: `MessageBoxOK("Hey There!", "Hello World!");` Then when you click on the button, you get this:



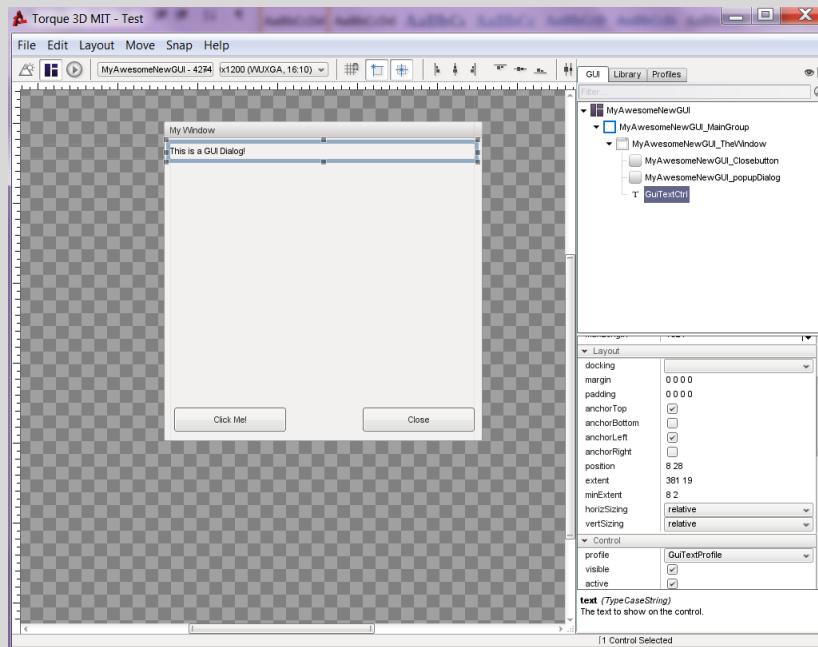
Checkboxes and Radio Buttons

These two button instances have the same properties as the regular button counterparts and are almost entirely identical in process to set up, but there's only one more thing you need to ensure to do, and that is to set them to be in the same group number so the relevant properties of the radio and checkbox buttons can be applied to them. Here's an example:



Text Instances

The next topic is that of placing text on your dialogs. This is something that almost every dialog has, explaining what needs to be done, or telling a user what they need to type or click during the dialog, or even to explain something to the user. There are two controls of interest for this topic, the **GuiTextCtrl** and the **GuiMLTextCtrl**, both located under the Text section of the library. The **GuiTextCtrl** offers the easiest deployment, but the least amount of customizability for the controls. You simply place the control on your dialog, type the text into the section, and move it to fit in your GUI, for most text instances, you don't even need to name it, just give it the "relative" scaling, and you're done!



The other text control is the GuiMLTextCtrl, and this control has much more flexibility and options in it, because it allows the use of the TorqueML Markup Language. The GuiTextCtrl on the other hand has to use the GuiProfiles to adjust the text, and even then, it can only have one setting per control. Here's the TDN Documentation on TorqueML:

GUI/TorqueML

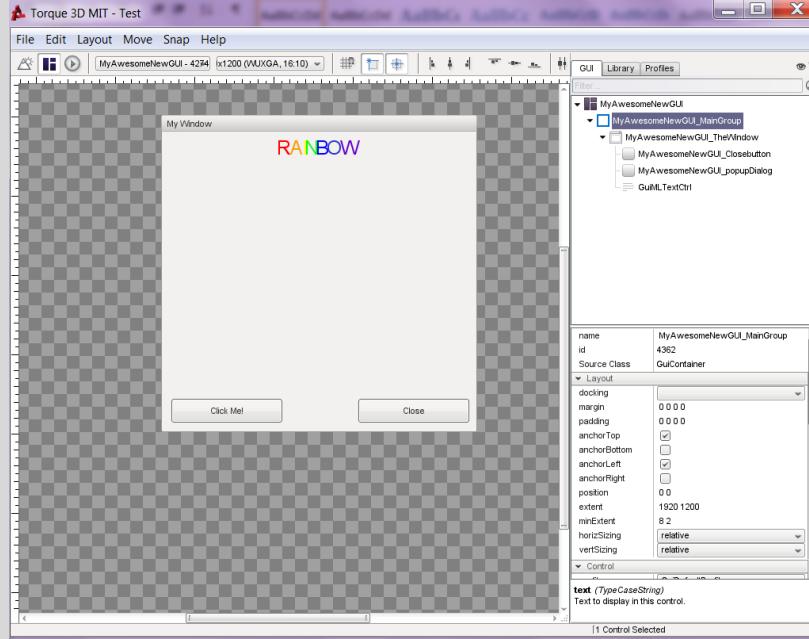
TorqueML is a markup language which allows one to dynamically change the style and formatting of text in certain Gui Controls. See the .hfi files included with the distribution for examples of usage.

Note: When specifying a color, you will use the format `RRGGBB[AA]`, where RR is a 2-digit hexadecimal number signifying the Red, and where Green and Blue are specified similarly. The optional AA signifies the "alpha": an alpha of "00" is transparent, "FF" opaque.

The following markup tags are recognized:

- `` Sets the current font to the indicated name and size. Example: `<font:Arial Bold:20>`
- `<tag>ID>` Set a tag to which we can scroll a `GuiScrollViewCtrl` (parent control of the GuiMLTextCtrl)
- `<color:RRGGBB>` Sets text color. Example: `<color:#ff0000>` will display red text.
- `<linkcolor:RRGGBBAA>` Sets the color of a hyperlink.
- `<linkcolorHL:RRGGBBAA>` Sets the color of a hyperlink that is being clicked.
- `<shadow:x:y>` Add a shadow to the text, displaced by (x, y).
- `<shadowcolor:RRGGBBAA>` Sets the color of the text shadow.
- `<bitmapfile>` Displays the bitmap image of the given file. Example: `<bitmap:demo/client/ui/separator>`
- `<push>` Saves the current text formatting so that temporary changes to formatting can be made. Used with `<pop>`.
- `<pop>` Restores the previously saved text formatting. Used with `<push>`.
- `
` Produces line breaks, similarly to `
`. However, while `
` keeps the current flow (for example, when flowing around the image), `<sbreak>` moves the cursor position to a new line in a more global manner (forces our text to stop flowing around the image, so text is drawn at a new line under the image).
- `<just:left>` Left justify the text.
- `<just:right>` Right justify the text.
- `<just:center>` Center the text.
- `content` Inserts a hyperlink into the text, which will open the user's browser. In the URL, the leading "http://" is understood. Example: `<a:www.garagegames.com>Garage Games Website`. When the user clicks on the hyperlink text, `GuiMLTextCtrl::onUrl(%this, %text)` function is called. Your implementation can do whatever you want, including but not limited to opening the user's browser and navigating to the url.
- `<margin:width>` Sets the left margin.
- `<margin%:width>` Sets the left margin as a percentage of the full width.
- `<margin:width>` Sets the right margin.
- `<margin%:width>` Sets the right margin as a percentage of the full width.
- `<clip:width>content</clip>` Produces the content, but clipped to the given width.
- `<div:bool>` Use the profile's fillColorHL to draw a background for the text.
- `<tab:#:#:#:#>` Sets tab stops at the given locations.
- `
` Forced line break.

You can place these directly in the text field of the control to do some very nifty things, for example:

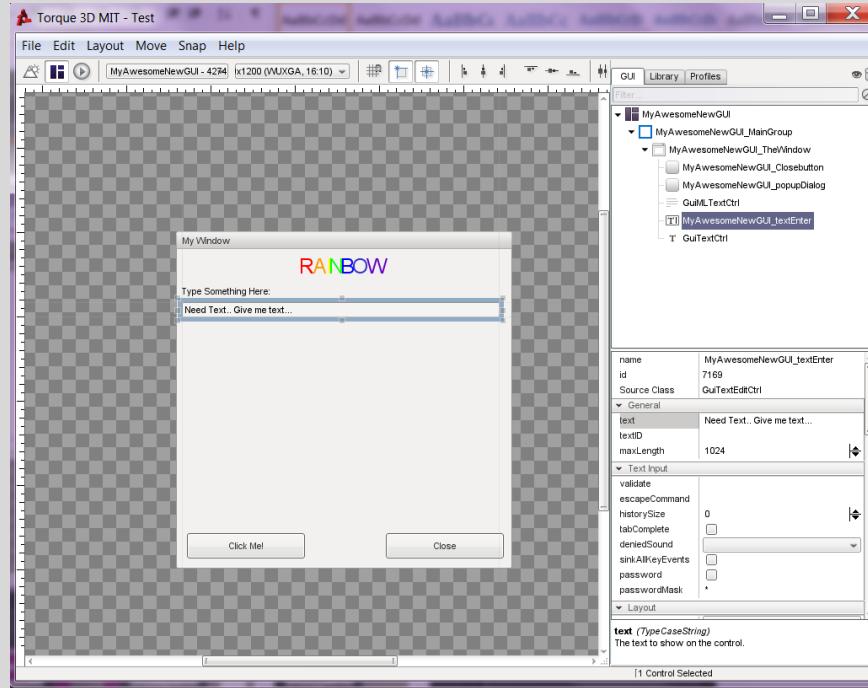


This uses the following markup string:

```
<Font:Arial:30><Just:Center><Color:FF0000>R<Color:FF9900>A<Color:FFFF00>I<Color:00FF00>N<Color:0000FF>B<Color:3333CC>O<Color:6600CC>W
```

Text Input

The next common thing that people want in their GUIs is the ability for their users to type some information in and then have it be used in the game itself. This uses what is called as a **GuiTextEditCtrl**. All you need to do for this is give it a name, and place it on the screen. Make sure to apply to relative sizing to it to ensure it fits in the screen.



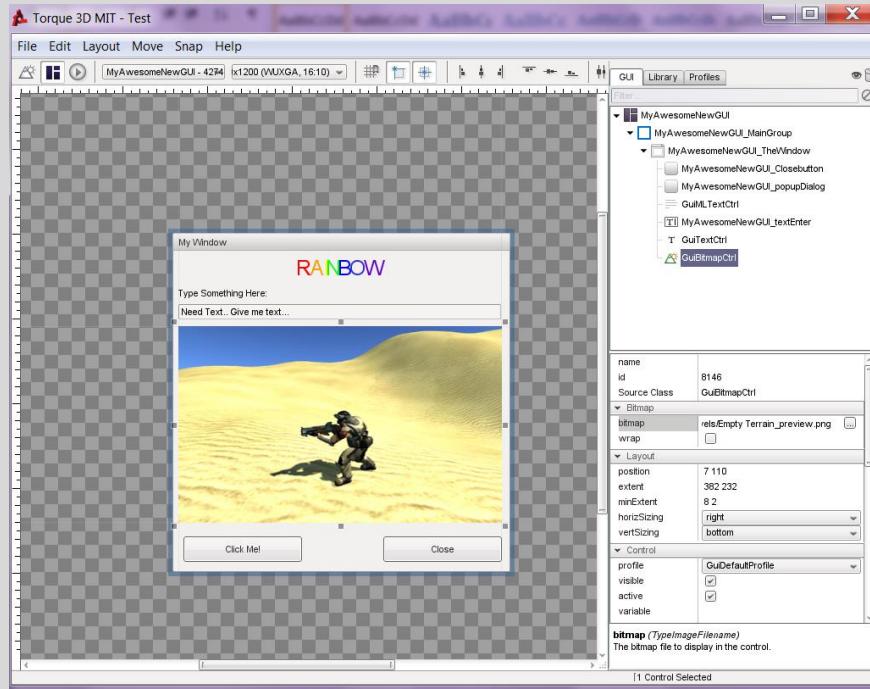
Now, let's do something else cool here. How about we change our "Click Me!" button to read off what you just typed in. So, click the button, and change the command to:

```
MessageBoxOK("Hey There!", "You Typed "@MyAwesomeNewGUI_textEnter.getText());
```

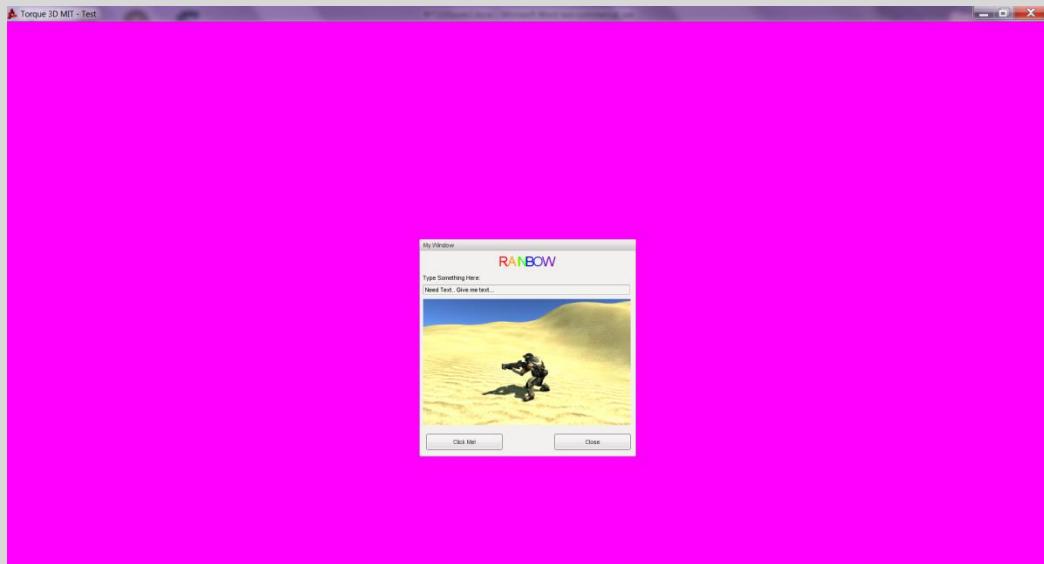
Replacing `MyAwesomeNewGUI_textEnter` with the name of your text entry control.

Images

Finally, let's talk about putting pictures in your control. Under the images tab of the library, you'll see an item named **GuiBitmapCtrl**, you can place this on your gui and then select an image in the inspector to display. Again, remember to keep the relative sizing on, and you'll be good to go!



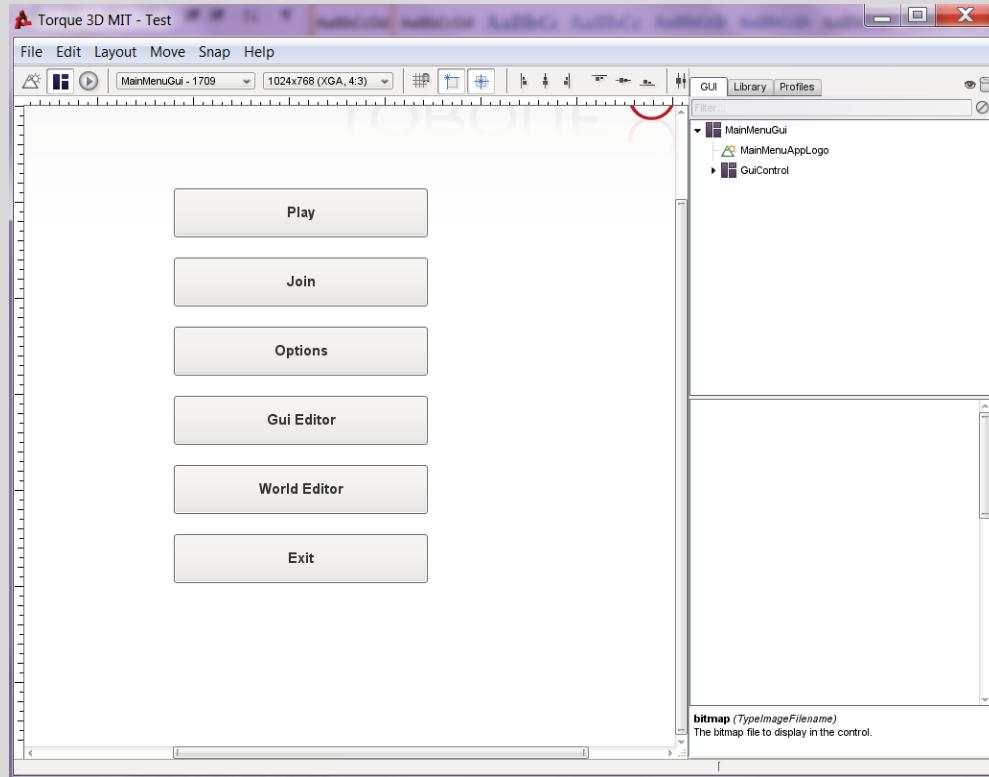
When you're all done, press F10 to show what the GUI would look like to you in the engine itself.



If everything looks good, go ahead and save the GUI. We'll talk about how to actually load this into the engine at startup every time in Chapter 10, but for now, that should cover the basics of actually creating a new GUI instance.

Editing an Existing GUI

So now that you know the basics of the GUI Editor, let's take that knowledge and use it to edit a GUI that's already in the engine, how about your main menu. So click the dropdown box and select the MainMenuGui option from the list, or any other gui you want to edit:



I'm not going to talk about much here since this is technically your project, but I will give you some basic rules about editing GUI instances that are already in the engine.

- Always ensure to follow the existing container sizes, if it's using relative, you also need to be using relative.
- Check for a "command" field, if you strip out something that performs a required operation, you need to replace it with something that will keep the existing dialog functioning properly.
- Be careful when removing or changing named instances, some of them are attached to the engine via scripts. When we get to Chapter 10, you'll understand this a little more to be able to detect anything that may have a relation to a script.

So, I hope this little journey through the game's editors has proven to be useful for you to teach you how to set up the game inside the engine by means of worlds and guis. Please be sure to go through and explore these editors to see what kind of cool things you can do. At this point in time, we leave the world of "easy" behind, and begin our journey into the more complicated side of game development, the side of Computer Programming.

Chapter 5: Introduction to Computer Programming

Brief Introduction

So far, I have exposed you to two of the powerful development tools in Torque 3D, the World Editor, and the GUI Editor. While a lot of your work developing a game is done in game through editors like the ones I have shown you there, another large portion of the work is actually done behind the scenes through actual programming. For a large portion of the remainder of this guide, I will now take you to the fun part of game development, which is actually making things work the way you want to. Before we dive into programming however, there are some general things I need to teach you about programming that will help you learn it much more quickly, and to help make it stick.

First off, computer programming is definitely not a task to be taken lightly. True, some of your programs you make will be extremely simple and require only a few lines of development, one of the hardest things to realize about programming is that there is no absolute correct answer to a problem, only a direction you place from the start of a problem to the end of a problem. For example, I could do something extremely crazy for a simple math problem like:

```
function basicMath() {  
    %one = 1;  
    %two = 2;  
    return %one + %two;  
}
```

```
function basicMath() {  
    return 1 + 2;  
}
```

These two different code blocks have the exact same outcome. Someone who ran both of these functions would be unable to tell the difference between the two without actually reading your source code. The point to get at here is that there are unlimited ways to come across an answer, and sometimes, the easiest solution to a problem can be overly complicated by some very simple mistakes. One of the hardest lessons you will learn is that computers are very unforgiving in terms of simple mistakes; it will do exactly what you tell it to do, even that little computation error you never saw in your code.

But before we talk about TorqueScript I'm going to introduce you to subjects and topics that are common to every single programming language out there. The reason people will say that once you learn how to program in one language, the others will easily follow, is because they're all similar in the finishing goal. They just have some slight differences.

Variables, Definitions, and Methods

So, let's start our little journey into computer programming. The first program almost anyone will try to create is a basic "Hello World" program, or basically a program that prints some text to a console or the screen to be viewed. Our first steps will begin by teaching you some very important concepts of programming. To start things out, we're going to talk about how data is stored in the computer. While most people will simply jump to the conclusion of memory, you need to remember that we're talking about things from a programming perspective now. And yes, while the correct answer for a generic question would be memory, the actual answer to my question is a *Variable*. A variable, is just like those lovely X's and Y's you saw in your mathematics courses all of those years ago, it basically stores something, whether it's a number, some text, or even something as big as an object reference, variables are one of the most important concepts of any programming language. Each programming language has its own way of handling variables, and I'll be sure you explain how they work when the time is right.

The next thing to talk about is the definitions of these variables. In programming languages, there are numerous ways to actually get data into a variable, usually something very silly and simple such as:

```
%x = 1;
```

This little fragment of code can be sufficient to place some information into a variable. But consider some operations such as loading information from a file, or even when this information is passed from each individual operation in the code. You can never guarantee anything to just be there all the time. And then, each programming language has its own rules for variable definitions. For example, TorqueScript treats entries as strings, so the above could be interpreted as a numerical 1, or the text "1". All in all, you need to remember how to define variables as according to the rules of the language you're programming in.

Finally, let's talk about the operations themselves. Contrary to your nightmares of programming being one massive text file with a large method to handle everything, programming has actually evolved into something much more useful. You can break operations into methods that may be repeated as many times as you want, classify methods to be held to a certain object, or even keep them hidden for debugging purposes. Each language has its own set of rules for function/method definitions, and we'll be sure to cover those in this guide when we get there.

Scope

The next important topic I want to nail out of the way right now is the concept of Scope. Scope basically tells your program where a certain variable is valid to be used at. I'll explain this much more in the coming chapters, but be aware that there are both local and global variables in the world of computer programming. A local variable means the variable is confined to the calling function, or method and it can only be used within the body of the function it belongs to. A global variable, is a variable that can be defined anywhere in the code, and once it's defined, any method or code block in

the entire program has access to that variable. I'll cover this example a little more in Chapter 6, but here's what you need to know about scope from a program example:

```
$myGlobal = 1;  
  
%myLocal = 1; //<- Bad  
  
function tests() {  
    echo("My Global: @"$myGlobal);  
    echo("My Local: @"%myLocal); //<- This is invalid and will crash.  
}  
  
  
function test2() {  
    %myLocal = 1; //<- Ok!  
    echo("My Global: @"$myGlobal);  
    echo("My Local: @"%myLocal); //<- This is valid and will be fine!  
}
```

In the above examples, if you tried to call `tests()`, you would likely get a crash from an out-of-scope error. Calling `test2()` would validate the output as you would expect, because `%myLocal` is located within the body of `test2()` and is valid. For the first example, having a local variable outside of a function is never advised, as it can lead to crashing, but from the context of `tests()`, `%myLocal` is a non-existent variable.

What is Syntax

The first major thing you will need to learn for any programming language is the syntax of the language, or basically, the rules of the programming language. Like any spoken or written language in the world, there are rules such as grammar that need to be followed, computer programming is no different. While each language may specify its own syntax to follow, they are generally the same overall with a few key differences. Since we're focusing on TorqueScript and C++, this will be simple to talk about since they are very similar to each other. I'll cover C++ with much more detail in Chapter 13, and we'll go over the C++ syntax then, but for now, I'll teach you generic syntax from the TorqueScript perspective.

So far, I have taught you a few things, and actually, some of those things are related to this topic, for example, the concept of Scope is a syntax rule for programming, which states that all local variables must be defined within, and before the variable is used. You also know that global variables may be defined anywhere and can be used anywhere, but now, let's talk about some of the basic rules you should start putting in your head.

For TorqueScript and C++, any line that is not a: comment, function or class definition, or a conditional statement, must end with a semicolon (;). This little guy will likely soon become the bane of your existence. I cannot stress how many times I've gotten people asking me why something isn't working, only to discover they forgot a semicolon on a line of code. Basic rule to follow by, if the computer is saying there's a syntax error, the first thing to check for is a missing semicolon.

Next up, every block of code must begin with an opening brace ({) and end with a closing brace (}). You should get into the habit of checking to ensure that the total number matching opening and closing braces ends at zero. Basically, for each opening brace, add one, and for each closing brace, subtract one, this rule also applies to parenthesis for statements, there must be the same amount of opening and closing parenthesis. For class definitions, this rule is expanded just a little bit, and it states that the class definition begins with the opening brace, and ends with a closing brace with a semicolon appended to it (};).

Speaking of parenthesis, the rule of mathematical operations has not changed from the real world and computers, and if you're getting a logic error (a human generated error that basically means the result is not as expected), then you're likely forgetting this order and should enclose the block in parenthesis such that the operation proceeds as expected. While we're still talking about math, don't just write a piece of code stating:

```
%myNumber = 10 ^ 2;
```

And expect the computer to spit out 100 from the above line of code. The reason is because you will soon learn about a computer operation called *Bitwise XOR*, which has the exact same operator. For mathematics in computers (I've got a whole chapter on it later), you'll need to learn how to do things a little differently.

The last thing to talk about is how Strings and Characters work in computer programming. Because you're likely not interested in the deep stuff, I'll just hit you with it now. Computers treat everything as numbers, remember all those binary jokes? This even applies to characters and strings, which by the way is just an array of characters. And these characters follow what is known as the ASCII table, which translates numerical values to their character and binary representations. So just be sure to be careful when working with characters, the main reason here is the existence of escape characters. This will likely get you the first time you try to output "\\" on the screen and to your astonishment only get "\" as a result, this is because '\' is the universal character entry to an escape character, which either accepts a second character, or a two digit hex code to convert to the ASCII representation of the string. In terms of programming, just know when you define a character, to enclose single quotes around the string:

```
char myChar = 'a';
```

And that when you define a string that you need to enclose it in double quotes:

```
%myString = "Hello World";
```

So, I hope this little prelude chapter helps you out a little bit before we dive into the juicy material coming up for the remainder of this guide. I know that taking your first steps into any programming language is no easy task to accomplish, but practice makes perfect. Please be sure to realize, you will make mistakes, and sometimes they will be rather big mistakes that require a lot of time

an effort to fix, but learning to fix these mistakes will help you better your skills as a programmer in the long run.

Error Checks, your new Best Friend

One of the biggest problems to most pieces of code I read out there is that there is never the assumption of “What If” being made. For example, let’s say I have a method called dividelt() and it accepts two numbers to perform a simple division. From a programmer’s perspective, they’re just assuming right away that the answer and numbers will always be legal, this is a very dangerous mindset as it can create a world of problems from memory leaks, to crashes, even to the dreaded blue screen of death if you’re not careful enough.

```
function dividelt(%num, %den) {  
    return %num / %den;  
}
```

So, look again at the little math function. Now, the rules of division are very straight-forward to what happens if the denominator is zero, and you don’t want to be throwing things like Not a Number or Infinity to a computer’s math processor, so what we’re going to do is implement an error check. Error checks will become one of your most powerful tools of function security to prevent the data from becoming corrupted, or even being invalid in nature. By implementing checks to ensure that there is data being sent to the function, objects are valid and other things like that, you can ensure that the method always performs to the highest level of efficiency. So, with that in mind, let’s revise the example above, again, don’t worry about the code itself, just try to think of what it does.

```
function dividelt(%num, %den) {  
    if(%num == "" || %den == "" || %den == 0) {  
        error("error: invalid parameter sent to dividelt().");  
        return 0;  
    }  
    return %num / %den;  
}
```

The general rule of thumb with this is to keep reliability and safety in mind for functions that require the data to be constrained by values, or to be set to a value when reliability of the sent values cannot be guaranteed by the program.

Being an Organized Coder

One of the big things you should quickly pick up when you get into development is the process of being a consistent and organized computer programmer. By keeping your code neat and organized, it will be much easier for others to read and for you to quickly identify potential problems in your code. Here’s my general stance on programming style (You don’t need to take this as the style that needs to be followed, this is just my example for you to see here, the goal is to establish a set of guidelines you follow when writing code):

- Document each code file with a comment signature on the top of the file. Ex:

```
/*
 * MathFunctions.cs
 * By: Robert C. Fritzen - © 2014
 * Basic mathematical functions and operations to be used in the program
 */
```

- Each declaration and conditional line ends with the bracket on the same line. Ex:

```
function divideIt(%num, %den) { //Good
    if(%num == "" || %den == "" || %den == 0) { //Good

function myFunction()
{ //Bad!
```

- Tab once per each bracket, and return a tab to close a bracket. Ex:

```
function myFunction() { //<- Note the bracket. Next line begins one tab.
    if($GVar == 1) { //<- Again, another bracket, another tab
        echo("Hello!");
    } //<- End of a block, return one tab...
} //<- End of function, return tab to the original level of the remainder of the file.
```

- Name your functions & variables accordingly, constants should be in uppercase, or be prepended with a _. Ex:

```
// Function Names \\

//Bad. Function name is very generic and doesn't make sense.

function doMath(%num1, %num2) {
    return %num1 + %num2;
}

//Good!

function addTwoNums(%num1, %num2) {
    return %num1 + %num2;
}
```

```
// Constants \\

//Bad. Undefined and explains nothing.

$noise = 1.0;

//Good!

$_GlobalVolume = 1.0;
```

- Use comments to explain your code. Some programs such as Doxygen use your comments to generate documentation of your code, so be sure to explain your code in detail! For more information on Doxygen and how to format for it, see this article:

<http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html>

```
//addTwoNums([int], [int]) – Add two numbers together, returns an integer.

function addTwoNums(%num1, %num2) {

    return %num1 + %num2;

}
```

And the list could go on, but the important thing is to pick up a style, and to keep with it for the entire project. It will help you, and anyone who needs to help with your project when they see a consistent and easy to read and navigate code-base.

Developing By Process

In the end, there is one big takeaway point from this chapter, and that is to devise a strategy to approach your programming problems. When I was writing my Radar Pack for Torque 3D, I had countless pages of notes and diagrams that showed me what I needed the result to be, and how I wanted to approach each program associated problem. Am I saying that you need to have hundreds of pages of a development journal for each program you make? Absolutely not! But it never hurts to have diagrams, or flowcharts to work from to show you what needs to get done. One of the biggest debugging tools is to know what the answer needs to be before it is actually received by the computer. If you know the starting point, and the ending point, you can work from both directions and isolate the problem in your code.

By creating something to work from, such as a chart or a diagram, or even by laying out the points of what your code needs to accomplish before you actually write it, you can create a mindset in yourself that states, I know how to do this, I just need to put the pieces together now, because in the end, programming is just like solving a big puzzle, or some kind of math problem. You know what the end result is supposed to be, you just need to put the pieces together to find a solution to the problem.

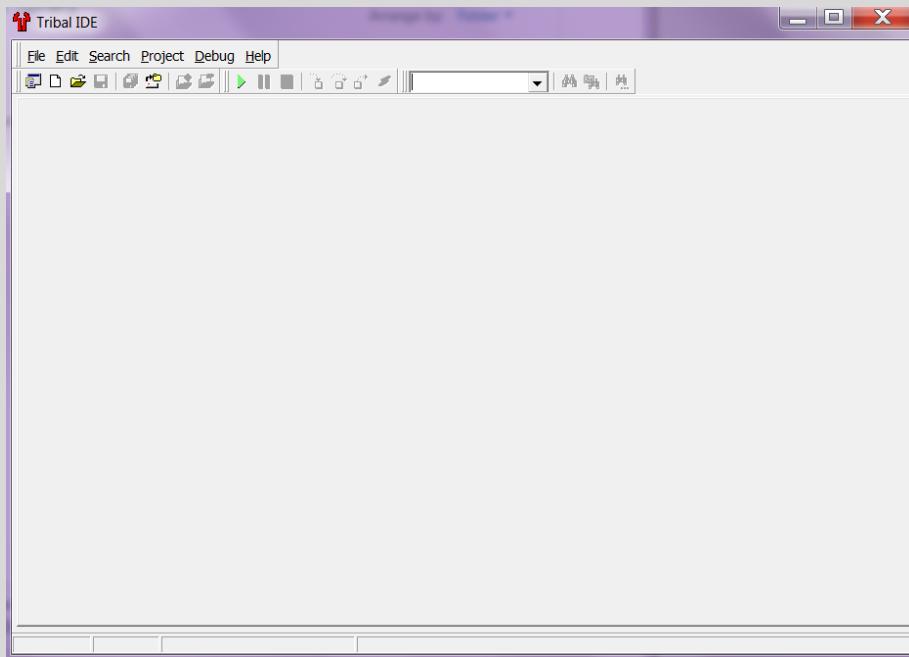
Anyways, I hope my little chapter here on the basics will at least get you into the proper mindset to programming, because from this point forward, we're going to hit the text blocks and dive head first into the wonderful world of computer programming!

Chapter 6: TorqueScript 101

Alright, so now that you understand (I hope), some of the overall basics of computer programming, we can actually begin to do just that. Before we jump into anything TorqueScript related, I'm going to give you something. Included with this little guide in the *Files* folder is a program called Tribal IDE. This is a free program that was designed to develop scripts for Tribes 2, which was a game built on the original version of the Torque Game Engine, however this editor still works fine for other Torque games, so it's only fitting that we use it for the purposes of the guide.

Tribal-IDE

So, go ahead and fire it up, you'll get a screen that looks like this (If you get the data error, you can safely ignore it):



The script files for the Torque Game engine use the .cs extension, so try not to confuse it with a C# source file, they are in very little ways similar, if not at all. Torque also uses two other formats, .gui for GUI files (Chapter 4) and .mis for Mission Files (Chapter 3), when working with Tribal IDE, you can directly edit all three of these formats.

There isn't too much that you really need to learn with this script editor for Torque 3D. Variables are colored based on their type, and some important Torque keywords are also highlighted for your reading convenience. For people looking to dive a little deeper into the editor, you can try to set up the debugger tool (Tribal was made for Tribes 2, a pre-TGE 1.0 game) to work with Torque 3D to give you some extra tools to help out in the process, but for the most part, all you really need to work with here is the text editing pane.

Introduction

So, now that we're set to start coding in TorqueScript, let's actually introduce the language to you so you can begin to understand some basic principles of the language. First off, TorqueScript is a Torque game engine unique scripting language with C-Style syntax. One of the biggest advantages to this scripting language is the direct lack of a need to use any form of data types, everything is directly converted to the proper type when handling information by default, and all information is parsed by the engine's console as a string instance. Most of your work in TorqueScript is going to be to master a few really basic concepts of programming and expand your knowledge from constructing new tools and functions to pull knowledge from.

In TorqueScript there are no advanced memory operations or working with things like pointers (Chapter 14) and references to memory locations, but you still have some very powerful options when working with objects and references made to objects by means of either the Ghosting system or the numerical id system, both of which, we'll get to at a later point in time.

So with the basic information about TorqueScript out of the way, let's get started with our journey!

Variables

The first thing you're going to learn about in programming is how to create a variable. In any computer programming language, a variable, is a stored location in memory where information of a specific data type (C++ / Chapter 13), or a universal data type (TorqueScript / Now) is stored for use at either the current moment, or a moment in a future point of time. You can only use a variable after it has been defined, attempting to use a variable before it is defined can lead to some potential problems such as crashing or invalid output. Variables come in two forms, local variables, and global variables. A local variable is a variable that can only operate within the block of code it is defined in, and a global variable is a variable that may operate within the entire program instance once it is defined.

Defining a variable in TorqueScript is extremely simple. To define a local variable, you use the percent symbol (%) followed by the name of the variable, and to define a global variable, you use the dollar symbol (\$) followed by the name of a variable. Variables can only use alpha-numeric characters and the underscore (_) character, and variables cannot start with a number. Here are some examples of local and global variables. If you name two local variables the same name in your code, the second instance will define the value over the first, Global Variables behave the same way. You'll want to pay close attention to the fact that each line ends with a semicolon (;), we'll talk more about that when we get to syntax in a bit, but for now, just know the importance of it. We'll also talk about comments (//) here in just a bit as well.

```
//Local Variables are defined with the % character.  
%myString = "Hello World";  
%myVar1 = 1;  
%myChar2 = 'a';  
%123 = 123; //-- Bad!
```

```
//Global Variables are defined with the $ character.  
  
$myString = "Hello World";  
  
$_stillOk = true; //<-- This is still ok, you can start with a _, but not with numbers  
  
$globalVar1 = 1;
```

Also recall from the previous chapter, the concept of variable scope, as it will be essential to your learning of any programming language. For a quick refresher in the topic, remember, that you should never define a local variable outside the body of any function, and the local variable is only valid in the block of code (the brackets containing the variable) in which it is defined. I'll have quite a few examples of scope here in a bit.

So, that's all you need to know about the concept of Variables, for now. We'll be coming back to this for quite some more information, especially how to manipulate these variables in a short period of time. The next topic we need to discuss is the keyword topic, which are important words of text the language has.

Keywords

In a computer programming language, a keyword, is a word the language knows to interpret as a specific action to perform. A good portion of these keywords have an additional function to it, which we will cover in more depth as we move along, but you need to know about these for the time being. Keywords are case-sensitive when you write them in the scripting language so something like **return** will differ from **Return**, which will have no effect but to likely generate a syntax error. Note, that all keywords in Torque are used in lowercase.

One important thing to realize for TorqueScript is that keywords are not reserved. Therefore one could in practice; create a function with the same name as a keyword. This is bad practice, but allowed by the language. So, let's talk about these keywords now.

Keyword: break

The first keyword we're going to talk about is the **break** keyword. This keyword is used exclusively for loops (Covered later on), and unlike in languages such as C/C++, this has no effect on **switch**. When this keyword is used in a loop, it immediately stops the execution of the loop, basically the equivalent of setting the loop condition to **false**. Here is an example of the **break** keyword:

```
$LoopCounter = 0;  
  
function myLoopExample() {  
    while(true) {  
        $LoopCounter++;  
        if($LoopCounter > 10) {  
            break; //<-- Break Keyword, Ends the loop.  
        }  
    }  
}
```

Keywords: case, or, switch, and switch\$

The **case** keyword is used in conjunction with the **switch** or **switch\$** keyword which is a secondary form of a conditional statement compared to keywords such as **if** and **else**. When you use the **case** keyword in TorqueScript it accepts a single parameter to be associated with followed by a colon (:) to mark the end of the parameter. Unlike languages such as C++, where a **break** is required to end a **case** block, TorqueScript requires no extra syntax, it either looks for the next **case**, **default**, or the end of the **switch** block to tell where a **case** ends and begins. Here is an example of these keywords:

```
function basicConditional(%number) {  
    switch(%number) {  
        case 1:  
            //do code here when %number is 1  
        case 2:  
            //do code here when %number is 2  
        case 3:  
            //do code here when %number is 3  
        default:  
            //call code here when none of the above values are matching  
    }  
}
```

When using **case** in conjunction with a string literal **switch\$** statement, you must enclose your parameters in quotation marks to mark the difference to the engine:

```
case "example":
```

One last neat thing to talk about with the **case** keyword is a special option when making statements by using the word **or** in between your arguments to accept multiple parameters for the same case statement. Here's an example:

```
function basicConditional(%number) {  
    switch(%number) {  
        case 1 or 2 or 3:  
            //do code here when %number is 1, 2, or 3  
        default:  
            //call code here when none of the above values are matching  
    }  
}
```

A general rule of thumb when using the switch and case statements is to place the case values in an order of most likely to least likely due to the way the statement is parsed. Placing the highest probable condition first will usually result in faster execution times.

Keyword: continue

The **continue** keyword is used exclusively for loops, like the **break** operator. When you call the **continue** statement in a loop, it skips the current iteration of the loop and begins the loop again on the next value. Here is an example of the **continue** keyword:

```
function myLoopExample() {
    for(%i = 0; %i < 10; %i++) {
        if(%i == 5) {
            continue; //Skip over the number 5.
        }
        echo("Loop at @"%i);
    }
}
```

Keywords: datablock & singleton

The **datablock** keyword is a special keyword that is unique to Torque 3D. It begins the definition of what is known as a **datablock** (Chapter 7 / Chapter 8). For knowledge you will learn later, a datablock is basically a persistent object that is exclusively controlled by the server and is transmitted one time to the clients when they join to have knowledge of what must always remain the same (Think: Player objects, projectiles, etc.). Datablock definitions are the same as object definitions, and thus they share the same syntax. Here is an example of a **datablock** definition:

```
datablock TypeIdentifier (DatablockName [: InheritedBlock]) {
    properties...
};
```

What you want to note is the little piece in the brackets [] there. This portion of the datablock definition will not always be in place, but it allows you to copy the properties of an existing datablock into a new instance, by means of something called inheritance (Chapter 8). As for the TypeIdentifier, this will be the name of the datablock type you are defining. By default, Torque3D offers the following types of datablocks:

- **AI{TurretShape}Data:** Used by AI controlled turret instances
- **CameraData:** Used by camera instances
- **DebrisData:** Used by debris, which are individually spawned objects
- **DecalData:** Used to define decal instances
- **ExplosionData:** Used to define explosions, which are either directly spawned by projectiles, or by you
- **FlyingVehicleData:** Used to define properties and settings for a flying vehicle object
- **ForestItemData:** Used to define objects and their settings to be used by the forest editor
- **GameBaseData:** Generic datablock inherited by most visual objects seen in game
- **HoverVehicleData:** Used to define properties and settings for a hovering vehicle object
- **ItemData:** Used to define properties and settings of items

- LightAnimData: Used to define light instances that change with time
- LightDescription: Used to define overviewed properties applies to individual light datablocks
- LightFlareData: Used to define lights with camera flaring effects, such as a lens flare
- LightningData: Used to define lightning properties and settings for storm objects
- MissionMarkerData: Used to define properties of objects such as waypoints, and markers
- ParticleData: Used to define individual properties and elements of a single particle effect
- ParticleEmitterData: Used to define properties and settings of a rendered particle effect
- ParticleEmitterNodeData: Used to define emitter properties for a spawned particle instance
- PathCameraData: Used to define properties and settings for a camera instance that follows a path
- PhysicsDebrisData: Used when the physics module is enabled, debris instance with physics
- PhysicsShapeData: Used when the physics module is enabled, shape instance with physics
- PlayerData: Used to define properties and settings for a player object
- PrecipitationData: Used to define properties and settings for precipitation
- ProjectileData: Used to define properties and settings for projectile instances
- ProximityMineData: Used to define properties and settings for a proximity mine object
- ReflectorDesc: Used to define properties and settings for objects with light reflection properties
- RigidShapeData: Generic datablock used for shape instances with rigid collision properties
- SFXAmbience: Sound block for ambient world effects
- SFXDescription: Used to define properties of sound for SFX blocks
- SFXEnvironment: Sound block for environmental world effects
- SFXPlayList: Used to define a list of sounds to randomly select when this sound block is called
- SFXProfile: Generic sound block with properties and settings
- SFXState: Used to define the overall properties of the SFX sound when enabled
- ShapeBaseData: Very generic level object datablock, can be used to create almost any object type
- ShapeBaseImageData: Very generic level image datablock, used to define images that are mounted on another object instance
- SplashData: Used to create a water splash effect instance
- StaticShapeData: Very generic level object datablock, used to create non-moving object instances
- TriggerData: Used to define properties of a map trigger instance
- TSForestItemData: Used to define individual forest objects to be spawned by the forest editor
- TurretShapeData: Used to define properties of a turret object
- VehicleData: Generic datablock inherited by all vehicle type datablocks
- WheeledVehicleData: Used to define properties and settings of a wheeled vehicle object
- WheeledVehicleSpring: Used to define a spring instance for a wheeled vehicle, like a buggy
- WheeledVehicleTire: Used to define the properties of the tires on a wheeled vehicle

In addition to the standard **datablock**, Torque 3D offers the **singleton** keywords, which are usually kept in exclusiveness to objects such as material instances and audio settings. This was done for two

important reasons. The first of the two reasons is that Torque 3D has a maximum allowance of around 2048 datablock instances to be active at any given time. The second of the two reasons is that not all blocks of data need to be server exclusive, for example, in the event of materials and audio effects, you could get away with using a singleton instead, because they won't affect actual gameplay versus aesthetic looks of the game, and singletons don't have a limit on the amount you may use. The **singleton** keyword has the exact same syntax as the **datablock** keyword, with the exception of replacing the word **datablock** with the word **singleton**.

Keyword: default

The **default** keyword is used in conjunction with the **switch** or **switch\$** keyword which is a secondary form of a conditional statement compared to keywords such as **if** and **else**. Unlike **case**, **default** has no additional parameters and is simply the fall-through parameter in the event the value sent to **switch** does not match any of your **case** values. Here is an example of **default**:

```
function basicConditional(%number) {  
    switch(%number) {  
        case 1:  
            //do code here when %number is 1  
        case 2:  
            //do code here when %number is 2  
        case 3:  
            //do code here when %number is 3  
        default:  
            //call code here when none of the above values are matching  
    }  
}
```

Keywords: else & else if & if

The **if** and **else** keywords are a common conditional keyword used by most, if not all computer programming languages. These keywords have two purposes. When used in conjunction with **if** to form the **else if** statement, this is used to create multiple conditions, and when using just **else**, this keyword establishes a generic overviewed condition. Here is an example of these keywords:

```
function basicConditional(%number) {  
    if(%number == 1) {  
        //do code here when %number is 1  
    }  
    else if(%number == 2) {  
        //do code here when %number is 2  
    }  
    else {
```

```
//call code here when none of the above values are matching
}
}
```

In TorqueScript, **else if** is a conditional statement which means it requires a set of parenthesis with a statement inside that needs to be tested. If the condition of the statement is true, then the block of code inside the **else if** is executed. When using the **else** keyword, if neither the if statement, or any of the **else if** statements match to trigger a true case, then the code inside the **else** statement is executed.

We'll cover **if**, **else if**, and **else** a little later on here in this chapter.

Keywords: false & true

The **false** keyword is a programming standard stating a Boolean false value, or a statement which is not true. In a numerical sense, **false** is the equivalent of 0. The **true** keyword is the opposite, resembling a Boolean value of true, or in the terms of the engine, any value that is not 0.

Keyword: for

The **for** keyword is used to initialize what is known as a **for** loop, or a counter-oriented looping statement. The **for** keyword accepts three parameters and is used to create a conditional oriented statement which is evaluated once each time before a loop is executed. Here is an example of the **for** keyword:

```
function basicForLoop() {
    for(%i = 0; %i < 10; %i++) {
        echo("Now at @"%i);
    }
}
```

For further assistance using a **for** loop, here is the generic syntax for the statement:

```
for (Initializer; Conditional; Change of Variable per Loop)
```

We'll cover the **for** statement, as well as other loop types in a little bit.

Keyword: function

The **function** keyword is probably one of the most important keywords you will need to learn for your journey through TorqueScript. This keyword allows you to declare and define a method, or a segment of code that is set to be executed from start to finish when it is called, by any means. Functions come in two major forms of importance, object functions, and non-object functions. Here is an example of each type.

Object Function:

```
function Player::killMe(%this) {
    %this.applyDamage(9999);
}
```

Non-Object Function:

```
function addTwoNumbers(%num1, %num2) {  
    return %num1 + %num2;  
}
```

Now, let's discuss each of these in more detail. First, we'll talk about the object function. Object functions are used to call code directly on an object instance. These functions have direct access to all of the properties and settings of the individual object, which by definition, **MUST** be stored in the first parameter of the function. I'll define what a parameter is in a little bit, but for now, just think of a parameter as a variable that is owned by the function you're in. Here is the general syntax of the object function:

```
function ObjectType::FunctionName(%this, %arg1, %arg2, ..., %argN) {  
    //Code Block  
}
```

Now, your function won't look exactly the same as this, for an easy example, just look above. See how the only parameter in killMe is %this, or the instance of the object. In fact, I didn't even need to use %this. I could have called it %player, or %object. Just know that in an object function, the first parameter is the object you're calling. I'll go into much more detail regarding object functions in chapter 8 when we actually talk about objects, but for now, let's focus on the non-object function.

A non-object function is a general function that can be called at pretty much any time, and can do just about anything, and work with just about any object or variable, as long as it is in the parameter list of the function, or is stored in a global variable. Here's the syntax of a non-object function:

```
function FunctionName(%arg1, %arg2, ..., %argN) {  
    //Code Block  
}
```

Remember, that each function statement must have a bracket after the closing parenthesis to begin the code block, and a closing bracket at the end of the function to end the code block. Forgetting to do this will cause a syntax error to occur. Also, know that no two functions may share the same name. The latest function to be defined will have precedence over the other one. As for object functions, be cautious with naming them. While you can define object functions with the same name as long as the object class is different, if they have any form of inheritance towards one another, you will have issues.

We'll go into functions and how to create them in very great detail here in just a bit, and actually, for a very good portion of this entire TorqueScript part of the guide, so hold tight.

Keyword: new

The **new** keyword is used in conjunction with spawning an object instance in the world when this statement is called. This code is treated as a definition statement, so it must be completely closed by means of a semicolon following the closing brace on the statement. Here is an example of the **new** keyword:

```
function spawnObject() {  
    %newObj = new Player()  
    position = "0 0 0";  
    rotation = "0 0 0 0";  
    datablock = DefaultPlayerData;  
}  
  
MissionCleanup.add(%newObj);  
  
return %newObj;  
}
```

We'll cover plenty of examples with this keyword in Chapter 8 when we cover object definitions in TorqueScript.

Keyword: package

The **package** keyword declares a special segment of code that is contained within a grouping of code called a **package**. The overall concept here is similar to C++ Namespaces, where a code block may contain uniquely identified code, the main difference here is that a package may be activated and deactivated on the fly, and similar functions contained within a package take precedence over previously defined functions. Here is an example of the **package** keyword. (We'll cover this topic further in Chapter 8)

```
function helloWorld() {  
    echo("hello world!");  
}  
  
package hwOverride {  
    function helloWorld() {  
        echo("Hello From Inside the Package!");  
    }  
}  
  
helloWorld();  
activatePackage(hwOverride);  
helloWorld();
```

Keyword: parent

The **parent** keyword is a special keyword that is used for inherited methods. When you derive a function or an object from an existing function or object, all of the prior defined functions and properties are made available to the new function or object. You can use the parent keyword to access these previously defined functions. The **parent** keyword can also be used in conjunction with a package to call a method outside of the package. Here is an example of the **parent** keyword.

```
datablock ItemData(MyNewItem) {  
    //...  
};
```

```
function ItemData::onAdd(%this, %obj) {
    //Do Stuff...
}

function MyNewItem::onAdd(%this, %obj) {
    parent::onAdd(%this, %obj); //Call ItemData::onAdd()...
    //Do More Stuff...
}
```

Keyword: return

The **return** keyword is a special keyword that can be used inside a function for two purposes. The first purpose is to terminate the execution of a function at the specified point, and the other purpose is to return a value to whatever called the function. Here is an example of the **return** keyword:

```
function dividelt(%num, %den) {
    if(%num == "" || %den == "" || %den == 0) {
        error("error: invalid parameter sent to dividelt().");
        return 0;
    }
    return %num / %den;
}
```

Keyword: while

The **while** keyword is used to initialize a **while** loop, which is a conditionally executed looping statement. The loop will continue to run while the condition is true. This looping statement is used frequently, but when creating one you must be careful not to create an infinite loop, otherwise the program will halt in the loop itself. Here is an example of the **while** keyword:

```
function basicWhileLoop() {
    %number = getRandom(1, 100);
    while(%number != 50) {
        echo("Generated A: @" + %number);
        %number = getRandom(1, 100);
    }
}
```

Operators

In any computer programming language, you will be dealing with operators to manipulate variables. Think of an operator as something you would do in a mathematics course, such as addition, or subtraction. Most of these operators from mathematics apply here, however, there are quite a few

special operators that are unique to programming, and you need to learn what they do. To make things easy, we'll separate them into categories. For all of these, assume we have two variables, %a and %b.

Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations on variables.

Operator Name	Operator Symbol	Example	Short Explanation
Multiplication	*	%a * %b	Multiply two numbers together
Division	/	%a / %b	Divide one number from another
Modulation	%	%a % %b	Fetch the remainder of the division between %a and %b
Addition	+	%a + %b	Add two numbers together
Subtraction	-	%a - %b	Subtract one number from another
Increment	++	%a++	Add one to the number
Decrement	--	%a--	Subtract one from the number
Totalizer	+=	%a += %b	Add a number to the value of a variable
Deduction	-=	%a -= %b	Subtract a number from the value of a variable
Multiplier	*=	%a *= %b	Multiply a number to the value of a variable
Divider	/=	%a /= %b	Divide a number from the value of a variable
Modulo Assign	%=	%a %= %b	Modulate a number from the value of a variable

Assignment Operators

Assignment operators are used to apply a value to a variable.

Operator Name	Operator Symbol	Example	Short Explanation
Assignment	=	%a = %b	Set the value of a variable

Bitwise Operators

Bitwise operators are used to adjust values at the byte level. When working in computers, you need to think of values on the binary level. So for example, our variables are 32-bit numeric, so imagine a string of 32 zeros and ones. For binary, we read from right to left and each successive value increases the overall power of the number or in a mathematical sense, it would be 2 raised to the power of the index of the table, where the first value is 0. To help imagine this, let's assume we have an eight bit number. So if I call my value: 00000001 in binary, this would translate to the number 1. To further imagine this concept, think of this table:

BIN VAL	1	1	1	1	1	1	1	1
NUM	128	64	32	16	8	4	2	1

For an eight bit number, setting all of the values to a 1, will give you a result of 255. When you set the first bit to 0, the number is zero, and when you set it to 1, the value is 1. Now, consider the last value, the 128 slot. If I set that value to 1, we add 128 to the overall result, and if I set it to 0, we simply ignore it, and the value would read: 127. So, the overall concept of binary should be fairly simple to grasp with this example. Imagine binary numbers to be a flag of values referring to off (0) and on (1). When the value is on (1), you add the relevant value to the overall result, and when the flag is off (0), you skip that number and move to the next one. So, let's take our example table above, flip the direction so it reads what we're used to and play the example out:

BIN VAL	1	1	1	1	1	1	1	1
NUM	1	2	4	8	16	32	64	128
Result	1	3	7	15	31	63	127	255

Simple, right? Now, for a 32 bit number, we extend the table out to 32 positions, increasing the NUM value by a power of 2 each time. For more examples of binary numbers and operations with them, see Appendix III. So, with that in mind, let's learn some operators.

Operator Name	Operator Symbol	Example	Short Explanation	Binary Example
Left Shift	<<	%a << %b	Shift the bits of %a, %b positions to the left	00001000 << 1 = 00010000
Right Shift	>>	%a >> %b	Shift the bits of %a, %b positions to the right	10000000 >> 1 = 01000000
Bitwise And	&	%a & %b	Set the flag of values where they match with the number 1 to 1, and where they don't match to 0	10011001 & 00001001 = 00001001
Bitwise Or		%a %b	Set the flag of values where either are 1 to 1. If they are both 0, the number remains 0.	10000100 00111000 = 10111100
Bitwise XOR (Exclusive Or)	^	%a ^ %b	Sets non-matching values to 1 and matching values to 0.	10000100 ^ 00111000 = 10111100
Bitwise Complement	~	~%a	Flip the value of all bits in the string	~00010001 = 11101110
Bitwise Assignment	X= (Where X is <<, >>, &, , ^, or ~)	%a X= %b	Apply the bitwise operation specified to %b and add the result to %a.	10010000 >>= 2 will set %a = 00100100

Comparison Operators (Logical Operators)

Comparison operators, sometimes called logical operators are used to perform comparisons of values between one another and then return true or false based on the result.

Operator Name	Operator Symbol	Example	Short Explanation
Equal To	==	%a == %b	Test if the two values are equal

Not Equal To	<code>!=</code>	<code>%a != %b</code>	Test if the two values do not match
String Equal To	<code>\$=</code>	<code>%a \$= %b</code>	Return true if two strings match
String Not Equal To	<code>!\$=</code>	<code>%a !\$= %b</code>	Return true if two strings don't match
Greater Than	<code>></code>	<code>%a > %b</code>	Return true if <code>%a</code> is greater than <code>%b</code>
Greater Than or Equal To	<code>>=</code>	<code>%a >= %b</code>	Return true if <code>%a</code> is greater than or equal to <code>%b</code>
Less Than	<code><</code>	<code>%a < %b</code>	Return true if <code>%a</code> is less than <code>%b</code>
Less Than or Equal To	<code><=</code>	<code>%a <= %b</code>	Return true if <code>%a</code> is less than or equal to <code>%b</code>
Logical And	<code>&&</code>	<code>%a && %b</code>	Return true if <code>%a</code> and <code>%b</code> both evaluate to true (non-zero)
Logical Or	<code> </code>	<code>%a %b</code>	Return true if <code>%a</code> or <code>%b</code> evaluate to true (non-zero)
Logical Not	<code>!</code>	<code>!%a</code>	Return true if <code>%a</code> evaluates to false (zero)

Miscellaneous Operators

You've learned most of the operators now; however there are a few more that can be used in Torque to perform some important tasks and jobs when working with complex object structures, conditionals, and strings.

Operator Name	Operator Symbol	Example	Short Explanation
Group, Parenthesis	<code>()</code>	<code>(%a == %b (%a != %b))</code>	Compound statements in the order that they need to be executed. Required syntax statement for control blocks & function definitions
Block	<code>{ }</code>	<code>If(%a == 1) { } else { }</code>	Defines code block sections, required for function. Optional for loops and conditionals, but enforces one line statements
Array	<code>[]</code>	<code>%a[0] = 1;</code>	Defines array elements (Chapter 7)
Listing, Variable Separator	<code>,</code>	<code>getWord(%a, 2)</code>	Separates variable parameters, list parameters
Access	<code>.</code>	<code>%a.position</code>	Accesses an object's definition or field (Chapter 8)
Conditional	<code>? :</code>	<code>%a == 1 ? true : false</code>	Defines a conditional statement
Namespace	<code>::</code>	<code>parent::onAdd(%this, %obj)</code>	Access a function within the specified namespace / object
Single Line Comment	<code>//</code>	<code>//Hello!</code>	Single Line comment, not executed by compiler. Used to document
Block Comment	<code>/* */</code>	<code>/* This area is ignored */</code>	Used to define large areas of code or comments to be ignored by the compiler
String Concatenation	<code>@</code>	<code>%a @ " " @ %b</code>	Add a string to the final result

Object Inheritance	:	new ScriptObject(A : B) {	Inherit the properties of B into A
String Space	SPC	%a SPC %b	Insert a space character at the specified point in a closed string
String Tab	TAB	%a TAB %b	Insert a horizontal tab character at the specified point in a closed string
String Newline	NL	%a NL %b	Insert a newline break character at the specified point in a closed string

Functions & Parameters

Now that you've got the basics down we can start our journey in programming itself. So with Tribal IDE open go ahead and create a blank script file (Ctrl + F10). You'll notice in Tribal IDE, that the line numbers are displayed on the left side of the screen and otherwise what would appear as a basic text editor to you. That's something you're going to learn quickly, is that writing code is more of a textual process, where-as in game things will be more visual. Each script you write is stored in a TorqueScript file, or a .CS file. There is no limit to the amount of scripts an individual project may have, and it is recommended that you keep things nice and organized based on what they do.

So, the first topic, and probably the most important topic for TorqueScript is the concept of writing functions. A function is basically a list of instructions that the engine will follow from start to finish. Each function must be uniquely named, and follows the same variable naming rules you saw earlier in this chapter. For our purposes right now, we're going to start with basic non-object functions and work our way up in complexity as we move through the guide.

Basics of Functions

Let's start with something very basic and common to most computer programming languages, the Hello World program. This will only take three lines of code to accomplish, so let's start by teaching you the proper syntax of a function definition. Each function definition begins with the **function** keyword. After the **function** keyword and one space, you provide the name of your function. After you write the name, create a parenthesis and an opening brace, space down one line and create a closing brace. [() { }]:

```
function helloWorld() {  
}
```

What we have here is what is commonly referred to as a function stub, or basically a template function that does nothing right away. This is a great way to deploy a list of functions you know you will need to complete a task, and then be able to simply fill in code as you move along.

For the hello world program, I will teach you a few tools here for displaying text in the console window of your application (~ key). These are pre-defined engine functions that can be used to display text messages to the console window when needed. They follow: **echo**, **warn**, and **error**. All three of these functions are defined in the following style:

```
echo(%message);
```

The “style” referred to here, is the accepted **parameter** the function will take. In our necessary example case here, the function echo accepts a string **parameter**. A **parameter** is basically a variable defined to a function. They are treated as **local variables** (Recall the concept of scope) that are defined right at the beginning of the function before your first line of code in the function, so for the echo function, the %message parameter is defined right away, and the remainder of the function is actually used to print the message to the console. So, with that in mind, let’s finish up our hello world function. Add another space in the function block and add the line **echo(“hello world!”);** to the code:

```
function helloWorld() {  
    echo("hello world!");  
}
```

Note how I space the echo statement three lines in (or one tab) following the formatting rules I defined back in chapter 5. Now, if you saved this script and loaded it up in the engine, you could call **helloWorld();** in the console to get a nice pretty output to the console window. But, we’re not at actual script loading just yet, so just know for now that it would do that. If you’re somewhat of a more visual learner, fire up the game and at the main menu, press the ~ key to open the console, then type this in:

```
function helloWorld() { echo("hello world!"); }
```

After you type that in, type **helloWorld();** in the console, and you’ll get the output message.

Parameters

So, this is the general syntax of a function body, you define the name, then you define the block of code to go with the function itself. So now, let’s expand the knowledge a little bit and talk about those parameters. So, we know that **echo** accepts a **%message** parameter, but what if we wanted to have a parameter for our little hello world example program to add some more text to the function call itself? This is a very easy thing to do! To help, let’s look at the full syntax of the **function** keyword:

```
function fcnName(%arg1, %arg2, %arg3, ..., %argn) {  
    //function body  
}
```

Notice the difference between the syntax here and our function definition? We have nothing inside the parenthesis, which is basically stating that our function does not accept **parameters**. Be careful when you read the syntax above, just because it has , ..., does not mean to actually place that in the code body, that is just there to make a point of variable separation. So, let’s expand our little hello world example now. So, let’s say we want to tag on another message after the “Hello World!” prints. To do this, we need to add one parameter to the function. Remember parameters are essentially local variables, but defined in function definitions.

```
function helloWorld(%moreText) {  
    echo("hello world!");  
}
```

Be sure that the parameter you make follows the variable naming rules, otherwise you'll have some problems. So, this is nice and all, but it doesn't actually do anything quite yet. I could call **helloWorld("More Text")**; and it would still print "hello world!" in the console. So, let's fix that up by using the string concatenation operator in our echo statement. Now, to use the string concatenation operator, you need to first ensure the string is closed, for example: "hello world!" is not a **closed** string until it hits that second quotation mark. Once you have a closed string, you can place the operator (@) after the quotation mark and place the second closed string instance there. So, let's place %moreText after the "hello world!" text:

```
function helloWorld(%moreText) {  
    echo("hello world!" @ %moreText);  
}
```

Now, repeating that call from earlier: **helloWorld("More Text")**; we get the output: "hello world!More Text". That looks a little silly though. There's no space between the "hello world!" and "More Text". There's two easy ways we can fix this without changing the parameter to " More Text". The first is to concatenate a space between the two strings:

```
function helloWorld(%moreText) {  
    echo("hello world!" @ " " @ %moreText);  
}
```

And the other is to concatenate the special "space" mark word between the text like so:

```
function helloWorld(%moreText) {  
    echo("hello world!" @ SPC @ %moreText);  
}
```

The **SPC** word simply means to add a space in the text at the point. There is also the **TAB** word which adds a tab at the specified point. Using either of those two and sending the call **helloWorld("More Text")**; now results in: "hello world! More Text" being outputted. So, now you have the grasp on one parameter. Let's try two.

So now, let's use our newly founded knowledge of writing functions to write a basic mathematical function to add two numbers together and then print the result to the console. To do this, we're going to need to create a variable inside the function. So let's start the same way as before. Write the two lines with the word function, the name, the parenthesis, and the braces. But this time, add the two parameters right away:

```
function addTwoNumbers(%num1, %num2) {  
}
```

Notice here, how the comma separates the two parameters but that there isn't one after the second parameter. This validates the point made above earlier about how syntax is specific to the function itself, and simply showing a ,... ,%argn is there to help you understand the format, not the actual case for your specific function. Now, back to our function, the goal of the function is to add two numbers together which is quite simple. But for the case of learning, I'll show you how to set this up in an efficient form. We'll start by making a **%result** variable, which will store the final result of the addition of the two numbers, and then we'll print the result. By telling what we're doing beforehand, we can explain a step-by-step process.

```
function addTwoNumbers(%num1, %num2) {  
    echo("We're going to add "@%num1@" and "@%num2); //-- String Concatenation Rules... Remember them!  
    %result = %num1 + %num2;  
    echo("The Answer is: @"%result);  
}
```

So hopefully this example has demonstrated how to set up functions, how to set up a local variable inside a function, and you should be getting pretty good at those string concatenation rules by now. Before we move on, let's talk more about defining variables inside functions just to make sure you understand everything here. Recall from a while back the difference between local variables and global variables. For functions, the most of your variable definitions will be of the local form. The rules for local variables are that they **1**, cannot be used before they are defined, and **2**, cannot be used beyond the scope of their definition. Going back to our function example now, the parameters **%num1**, and **%num2** are treated as local variables for the entire function of **addTwoNumbers**. What this means, is that their scope is the entirety of **addTwoNumbers**, they can be used anywhere inside the function. And unless they come undefined into the method, they are defined. What this means is that our new local variable, **%result** can safely use both **%num1**, and **%num2**. As for the variable **%result**, it can only be used by functions and methods **AFTER** the line **%result = %num1 + %num2;** which our example here is clearly showing.

So, one last thing to quickly talk about here before moving on and that is what happens if someone sends too few parameters, or sends one of them undefined. For our example here, that means the variable will be undefined, and hence invalidate the **%result** variable. To protect them method from a problem like this, we'll use what is known as an error check, which is a conditional statement to test the validity of the variables before the function can execute, but we'll get there in just a bit.

Calling Functions, Returning Values

Functions play one more important role in the engine, and this is to return a value to the method or source that called the function. For example, let's say we have the **addTwoNumbers** method up above be called from a **generalMath** function that needs a bunch of other functions. Well, by

returning a value we can define a new variable inside the calling function to accept a value from the method we're calling as a new variable. Here's an example:

```
function generalMath(%num1, %num2) {  
    echo("GeneralMath()... Time to do some mathematics");  
  
    %addResult = addTwoNumbers(%num1, %num2);  
  
    %subResult = subTwoNumbers(%num1, %num2);  
  
    echo("Addition Result: @"%addResult@"\nSubtraction Result: @"%subResult);  
}
```

So, let's first talk about the concept of **calling functions**. When you **call** a function, you tell the computer to jump over to the requested function, run it, and then come back to the original function. In the engine there are two ways to call a function, a **direct call**, and an **indirect call** (Ch. 7). For now, we're going to work with the direct function call. To directly call a function, you simply type the name of the function, and provide it with the parameters you want to send to it, or in our case:

```
addTwoNumbers(%num1, %num2);
```

This is called a direct call because the engine will immediately jump to the function and run it directly from the function you're calling it from. In fact, when you did the same in the console window earlier by typing **helloWorld("More Text")**; you were actually performing a direct call to **helloWorld**. This is because the console window input is actually controlled by a function, and when you sent the input to that function, it directly called the method. Need more examples for a direct call? Well, just look at all of the code samples we have written so far. When you call the **echo** function, you're actually directly calling the **echo** function and providing it with the **%message** parameter.

So, let's expand things a little bit here by introducing the concept of **return values**. All of the functions you write can be told to **return** a value to the calling method. By doing this, you can send a value back to the calling method so it can work directly with it in its own code. So, let's expand our **addTwoNumbers** example code above to include a **return** statement. To do this, we leverage the fact that we already created a **%result** variable earlier:

```
function addTwoNumbers(%num1, %num2) {  
    echo("We're going to add @"%num1@" and @"%num2); //-- String Concatenation Rules... Remember them!  
  
    %result = %num1 + %num2;  
  
    echo("The Answer is: @"%result);  
  
    return %result;  
}
```

The bolded code above shows what we added to the method. All we did was place a **return** statement in the code, and now whenever we call this method through the console, or by a direct

method call, it will return the %result variable to the calling method to be used. To actually use the value, we go back to the example provided at the beginning of this section:

```
function generalMath(%num1, %num2) {  
    echo("GeneralMath()... Time to do some mathematics");  
    %addResult = addTwoNumbers(%num1, %num2);  
    %subResult = subTwoNumbers(%num1, %num2);  
    echo("Addition Result: @"%addResult@"\nSubtraction Result: @"%subResult);  
}
```

So, note that the direct call to **addTwoNumbers** sends the method %num1, and %num2 from **generalMath**, and then when we hit the return statement in **addTwoNumbers**, it sends it back to **generalMath**. We then store that value in a new variable, **%addResult**, which now has the scope of allowance beyond the statement.

As for the **subTwoNumbers**, that is for you to code. It's time to actually apply what you've learned so far and just write up the method using the examples above. It should be a very simple task. And the last thing we're going to briefly mention here is in that echo statement. You'll notice the little \n in there. That is a new line terminator sequence, or basically a shortcut to creating a new line break at that point. You can use that as a shortcut to not having to write multiple echoes. So go ahead and play around with those functions and methods, or even add some more functioning to it. We'll have plenty more on functions in the coming chapters when we get a little deeper into the engine's toolbox itself, but for the time being we can move on to our next topic.

Conditionals

Now that we've got the important concept of functions down, we can move onto the next important topic of scripting, and that is the topic of conditional statements. When it comes down to thinking terms, a conditional is basically a statement of choice, a piece of code that is evaluated when a certain condition is met. There's a few ways we can accomplish this concept, so let's take a good look at them.

If, Else If, Else Blocks

Now, the very first thing to realize is that computers work in a sense of what is true and what is not true. When you work with any form of a conditional, the computer will **ONLY** execute that which is true, and it will only execute the **FIRST** case that is **COMPLETELY** true in a block of statements. So, the first thing we need to look at is how to actually use a conditional statement, and to do that, we need to go up to our table of Comparison Operators. These operators will be of utmost importance when you write a conditional statement. They are basically logical in a sense of trying to find out what is true and what is not true.

To actually make a conditional statement however, there are a few tools at our disposal. The first tool is something called an **if-else if-else block**. At an exact minimum, this conditional needs an **if**

statement, you can get away with using **if/else** as well, but, you cannot have an **if** and an **else if** without an **else**, and you cannot have an **else if** or an **else** without an **if** statement. So, before we begin, let's look at the overall syntax example:

```
function basicConditional() {  
    if(statement) {  
        //code to execute  
    }  
    else if(statement) {  
        //code to execute  
    }  
    else {  
        //code to execute  
    }  
}
```

This is a complete sample containing all three statements here; each conditional here requires some form of logical statement that it needs to test. If the statement it tests validates to true, the code will execute. If the statement in the **if** or the **else if** are not true, the code in the **else** statement will execute. So, let's expand our little math example above. We'll add a parameter at the beginning to detect what operation needs to be performed, and then to perform the operation requested. To do this, we'll have a numerical parameter named **%operation**. If this number is 1, we'll do addition, if it's 2, we'll do subtraction, and if it's neither 1 or 2, we'll push an error message. You'll need the equality test logical operator (**==**) to do this, and here is what it looks like:

```
function generalMath(%operation, %num1, %num2) {  
    echo("GeneralMath()... Time to do some mathematics");  
    if(%operation == 1) {  
        %addResult = addTwoNumbers(%num1, %num2);  
        echo("Addition Result: @"%addResult);  
    }  
    else if(%operation == 2) {  
        %subResult = subTwoNumbers(%num1, %num2);  
        echo("Subtraction Result: @"%subResult);  
    }  
    else {  
        error("Invalid option, use (1) for addition and (2) for subtraction.");  
    }  
}
```

```
}
```

```
}
```

Now when we run this code, we need to provide a value to the **%operation** variable. You'll notice I have also moved the echo statements to be inside their relevant block, because now, the scope of the result variables has changed to be inside the if/else if statements, and both will not be valid outside of the block since you will only have one of the two actually defined. Now, we can expand this method just a bit. Let's say you'll only ever need to add or subtract. Well, you're not limited to three if block statements here, you can compact this down to an if/else block.

```
function generalMath(%operation, %num1, %num2) {  
  
    echo("GeneralMath()... Time to do some mathematics");  
  
    if(%operation == 1) {  
  
        %addResult = addTwoNumbers(%num1, %num2);  
  
        echo("Addition Result: @"%addResult);  
  
    }  
  
    else {  
  
        %subResult = subTwoNumbers(%num1, %num2);  
  
        echo("Subtraction Result: @"%subResult);  
  
    }  
}
```

Now, if the **%operation** variable is set to 1, it will perform addition, otherwise the function will perform subtraction. We can still expand this method even further. Assume we have two more math methods: **multiplyTwoNumbers()**, and **divideTwoNumbers()**, and let's assume we change **%operation** to be a string instead of a number, so we need the String Equal To operator (**\$=**). This is how you'd expand the conditional block:

```
function generalMath(%operation, %num1, %num2) {  
  
    echo("GeneralMath()... Time to do some mathematics");  
  
    if(%operation $= "+") {  
  
        %addResult = addTwoNumbers(%num1, %num2);  
  
        echo("Addition Result: @"%addResult);  
  
    }  
  
    else if(%operation $= "-") {  
  
        %subResult = subTwoNumbers(%num1, %num2);  
  
        echo("Subtraction Result: @"%subResult);  
  
    }  
}
```

```
else if(%operation $= "*") {  
    %mulResult = multiplyTwoNumbers(%num1, %num2);  
    echo("Multiplication Result: @"%mulResult);  
}  
  
else if(%operation $= "/") {  
    %divResult = divideTwoNumbers(%num1, %num2);  
    echo("Division Result: @"%divResult);  
}  
  
else {  
    error("Invalid option, provide a mathematical operator (+, -, *, /) to run this function");  
}  
}
```

What you need to pull from here is the equality testing operators from the Comparison Operators table above and how to actually set up these logical statements. But, we can still go even further than that, and make our statements even more logically complex to have access to more advanced functioning.

Complex Logical Operations

We can take things a step further with the logical testing. To do this, we'll use three more operators that are at your disposal, the **logical and (&&)**, the **logical or (||)**, and the **logical not (!)** operators. These three operators revolve around that core idea I talked about at the beginning of this sub-section on conditionals, they only care about what is true and what is not true. The logical and statement is used to specify that two conditional statements must both be true in order for the overall statement to be true, for example:

```
function advancedConditional() {  
    %myNumber = getRandom(1, 100);  
    if(%myNumber > 25 && %myNumber < 75) {  
        echo("Number is greater than 25, but less than 75");  
    }  
}
```

In this example, the number must fit both conditions for the statement to execute. The next one is the logical or statement, which specifies that only one of the two stated conditions must be met for the answer to evaluate to true. If I revise the above example:

```
function advancedConditional() {  
    %myNumber = getRandom(1, 100);
```

```
if(%myNumber > 25 || %myNumber < 75) {  
    echo("Where is this number?");  
}  
}
```

So, here's an interesting question for you now. What happens when I call this function now? The correct answer is that the echo statement will print every time because numbers between 1 and 100 will always either be greater than 25, or less than 75. There are two more topics to discuss here. The first is the logical not statement, and then complex logical groups. The logical not basically flips the notion on true and false, and says if the condition is not true, then the statement is true:

```
function advancedConditional() {  
    %myNumber = getRandom(1, 100);  
    if(!(%myNumber > 25 && %myNumber < 75)) {  
        echo("Where am I now?");  
    }  
}
```

You'll notice with the above example, I enclosed the logical and statement in parenthesis, so the computer will evaluate that first, and it will either read **true** or **false**. The logical not statement then flips the result from **true** to **false**, and vice versa. So the correct answer here is the number is either less than 25 or greater than 75.

The final topic for discussion here is that of complex logical groups. You can have multiple logical blocks within one big if statement, and as long as the overall result is true, so will the statement, so let's take a peek:

```
function advancedConditional() {  
    %myNumber = getRandom(1, 100);  
    if((%myNumber > 25 && %myNumber < 75) && (%number != 50)) {  
        echo("We got it!");  
    }  
}
```

See what's going on there? There are three conditional statements that are evaluated here. To make things easy, let's assume we generate **%myNumber = 44**. Here is how the computer would read this in order:

```
if((44 > 25 && 44 < 75) && (44 != 50)) {
```

```
if((true && true) && (true)) {  
  
if((true) && (true)) {  
  
if(true && true) {  
  
if(true) {
```

You can see there, how the statement works its way down through the defined order of execution here to come to the answer we need. Basically put, you can extend your logical conditions as far out as they need to go. Complex logical structures can ensure that your problem that needs to be solved will always fit the correct answer that you need to derive from the function. As long as the final condition breaks down to **if(true)** the specified code block will execute.

Switch Blocks

Now that some comprehension of the scripting language should be developing, let's slightly tick up the difficulty and lower the amount of discussion a tad bit to ensure we're advancing at a good pace. The next piece of information is the teaching of the **switch** statement. This is another conditional, however it works in a slightly different fashion compared to the if blocks above. Instead of defining a statement, we send the variable in question to the switch statement and case out possible answers:

```
function switchTest() {  
  
%myNumber = getRandom(1, 100);  
  
switch(%myNumber) {  
  
case 1 or 2 or 3 or 4 or 5:  
  
echo("Our number is between 1 and 5...");  
  
case 20:  
  
echo("Our number is 20.");  
  
default:  
  
echo("Didn't get anything...");  
  
}
```

You should be able to understand what's going on here pretty quickly. The first line generates a number between 1 and 100, then we test the number to see if it matches a value in the switch statement. If the value matches one of our case statements, we print it out, otherwise the **default** statement is called. This is essentially another way to get past some basic if/else blocks by directly testing possible answers.

We could take our larger block code from earlier from the function **generalMath** and swap that out with a switch statement, but since it's using a string literal, we need to use the keyword **switch\$** in this example:

```
function generalMath_Conditional(%operation, %num1, %num2) {  
    echo("generalMath_Conditional()... Time to do some mathematics");  
  
    switch$(%operation) {  
        case "+":  
            %addResult = addTwoNumbers(%num1, %num2);  
            echo("Addition Result: @"%addResult);  
  
        case "-":  
            %subResult = subTwoNumbers(%num1, %num2);  
            echo("Subtraction Result: @"%subResult);  
  
        case "*":  
            %mulResult = multiplyTwoNumbers(%num1, %num2);  
            echo("Multiplication Result: @"%mulResult);  
  
        case "/":  
            %divResult = divideTwoNumbers(%num1, %num2);  
            echo("Division Result: @"%divResult);  
  
        default:  
            error("Invalid option, provide a mathematical operator (+, -, *, /) to run this function");  
    }  
}
```

Basically put, you can use **switch** and **switch\$** to quickly evaluate a conditional statement instead of needing to write large blocks of if / else statements to accomplish the same goal.

Conditional Operator

The last type of conditional is something called the conditional operator. This operator directly performs an **if/else** check directly on the provided values to the operator. There really isn't too much to talk about here, so let's provide some examples:

```
function conditionalOperator() {  
    %myNumber = getRandom(1, 100);  
  
    %myNumber == 50 ? echo("It's 50") : echo("It's not 50");  
  
    %myNumber > 25 ? (%myNumber < 75 ? echo("yes") : echo ("No.")) : echo("Nope.");  
}
```

The conditional operator is the combination of the ? and : symbols. The syntax is as follows:

```
Statement ? True Condition : False Condition;
```

You can also see by the second example of the operator, that you can have **nested** conditional operators. The second example code is actually a copy of the logical and example where we tested if the number was between 25 and 75.

Before we finish up here, you should be aware that it's entirely possible to have what is known as a **nested** conditional statement using all three of the types of conditionals you have learned here. For example, you can have secondary if/else-if/else blocks within any of the if statements, you could also perform switch() on a second variable in a case of the switch of a first variable, and you just saw an example of the nested conditional operator. Be sure to explore, you only learn by trying!

Error Checks

Our final topic in this section takes us back to the ending of the prior section on functions. You may recall I mentioned the concept of using an error check to ensure the function's input was valid. In order to actually do this, we needed to know how to use a conditional, and now that you know how to use these statements, we can actually go back and fix our math example above. So, I took the liberty of writing the division function I mentioned for you to save some time:

```
function divideTwoNumbers(%num1, %num2) {  
    echo("We're going to divide "@%num1@" and "@%num2); //<-- String Concatenation Rules... Remember them!  
    %result = %num1 / %num2;  
    echo("The Answer is: @"%result);  
    return %result;  
}
```

There's only one problem here. What would happen in the event I forgot to send **%num2**, or for some reason, I send the number **0** to **%num2**. I'm sure you're well aware of the fact that you can't divide by zero (Well, you can, but in most cases it's either infinity or not a number), so we need to make sure that when the function is called, that the output is valid, to do this, we implement an error check on the function to ensure the input values are correct!

```
function divideTwoNumbers(%num1, %num2) {  
    if(%num1 $= "" || %num2 $= "" || %num2 == 0) {  
        error("Cannot perform this division...");  
        return 0;  
    }  
    echo("We're going to divide "@%num1@" and "@%num2); //<-- String Concatenation Rules... Remember them!  
    %result = %num1 / %num2;  
    echo("The Answer is: @"%result);  
    return %result;  
}
```

Basically, I took the knowledge of the if statement from earlier, added a few logical or tests to ensure that all three conditions are safe to use, and if any one of the input statements are invalid, to break out of the function and return 0. You'll want to get into the habit of implementing error checks in your code. From a security perspective you should always assume that the user will try to do something at some point to break your code, and by implementing an error check, we can ensure that it won't do anything too bad when that happens.

Loops

Our next topic encompasses a part of computer programming that will be used in numerous applications when developing games, and internal functioning, and that is the concept of looping structures. A Loop is basically a block of code that executes over and over again until the condition of the loop is no longer true. The reason I'm teaching this topic after the conditionals section is because the structure of a loop uses the same condition logic as before. In TorqueScript, there are two types of loops you can use, a While loop and a For loop.

While Loops

A while loop is a pre-check loop, which means the condition of the loop is evaluated before the loop is actually performed. These loops are the most common types of loops deployed in programming and are very easy to use. Here is the following structure of a while loop:

```
while(condition) {  
    //code here  
}
```

However, while this loop is easier to deploy, it is also more prone to the problem of an infinite loop lock, or a loop that can't stop due to the condition always evaluating to true. To ensure your looping structures are safe and don't cause a deadlock of the program, you need to ensure that the loop either has a built in error check to break out of the loop (using the **break** keyword), or that you always know what the looping result is meant to be and know that it won't enter a lock.

Let's look at a few cases of looping structures to teach you how to properly use a while loop now. The first example here is actually something that would be more suited to the second type of loop, but know that you can do it this way as well:

```
function counterLoop_While() {  
    %i = 1;  
    while(%i <= 10) {  
        echo("Counter now at "@%i);  
        %i++;  
    }  
}
```

Let's look at the example here. The first thing we do, is initialize a local variable named `%i` to the number 1, then we begin a looping structure. The while loop tests if the variable `%i` is less than or equal

to 10, if it matches the condition, it enters the loop and runs the code inside the loop. The first thing that will happen is the code will print: “Counter now at 1”, and then increment the variable %i one, which will set the number to 2. The code then goes back to the while statement and evaluates the code again, this time %i is 2, so the loop will run again, and again, until %i is equal to 11, then it will stop and exit the code and move on through the remainder of the code, which will do nothing since the while block is the only thing there.

So now, let’s go back a bit to the advanced conditional stuff above, where we tested ranges of numbers at a time. Let’s say you wanted to keep running that code until you got a number that fit the range. Well, by using a while loop, you can do this:

```
function generatorWhile() {  
    %myNumber = getRandom(1, 100);  
  
    echo("Number is "@%myNumber);  
  
    while(%myNumber > 25 && %myNumber < 75) {  
  
        echo("The number is where we don't want it, let's generate another!");  
  
        %myNumber = getRandom(1, 100);  
  
        echo("Number is now "@%myNumber);  
  
    }  
  
    echo("We got the number we wanted: "@%myNumber);  
}
```

With this code example, the loop will continue to run until it gets a number less than 25, or greater than 75. But! You also need to notice the potential that the generator will meet the condition right away, in this event the loop will **not** execute. That is because this is a pre-check loop, which means the condition is tested before we enter the loop itself. So you need to remember that a while loop won’t always execute, but when it does, to be careful of creating a locking structure (infinite loop). Now, let’s look at the other type of loop.

For Loops

A for loop, like the while loop is another type of pre-check loop which means the condition will be evaluated before the code will run, however, the main purpose of a for loop is to be a counter based loop. This looping structure has three statements allowed to be evaluated at once before the loop itself is actually ran. Here is the syntax of a for loop:

```
for(expression 1; expression 2; expression 3) {  
    //code here  
}
```

A majority of the for loops you will deploy will use a very similar form and will almost always take on the form shown below, however, there are numerous different approaches and applications of a

for loop. Basically, the rule of thumb here is if you know how many times you need the loop to run, use a for loop. Let's look at the counter loop again, but using a for loop this time around:

```
function counterLoop_For() {  
    for(%i = 1; %i <= 10; %i++) {  
        echo("Counter now at \"%@i\";  
    }  
}
```

Here's how this works, the for loop begins right away and sets %i = 1 with the first expression. This expression, is only executed **once**, and that is right away. The second expression is the condition of the loop, similar to that of the while loop, which again will test if %i is less than or equal to 10. The last expression is the modifier of %i to be done with each successive iteration of the loop, or in our case, to increase %i by one each time through.

Loops are a very important focus of any programming language, as it takes the need to run a function over and over again to achieve a result out of the equation, this in turn leads to better overall performance and run times. You may not see the full application and importance of a loop right now, but as we progress through this guide, their importance will become clearer.

How Scripts are Loaded

So, what we have done so far in our work is to teach the basics of the scripting language, and to show you how to set things up. But there's just one problem, you haven't been able to actually test your code for yourself to see if it's working or not, and it's time to change that notion and show you how to actually get your scripts to be loaded by the engine.

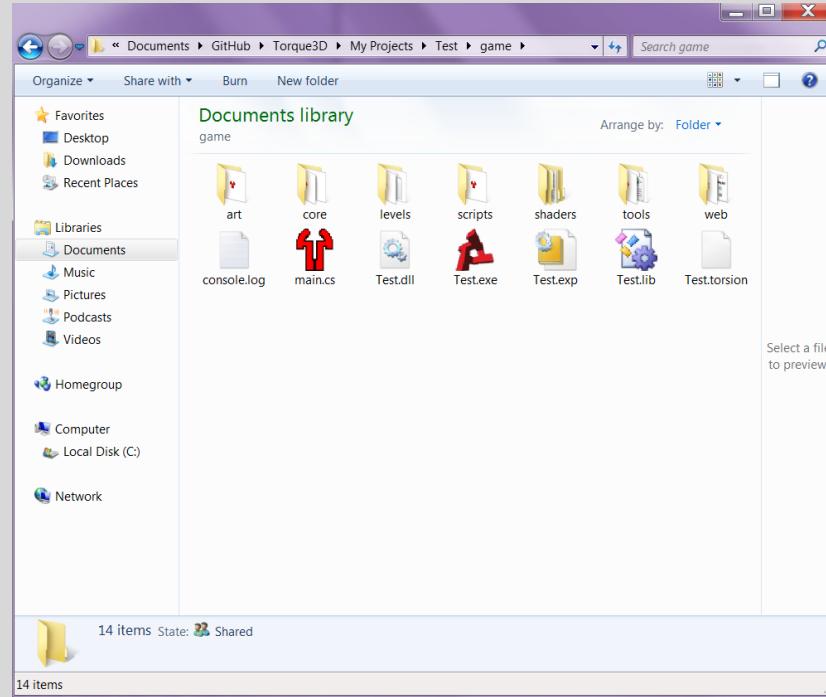
The concept of script loading is actually handled by another function in the engine named **exec**. This function accepts a file path as a variable, and when it's called, it loads the file and evaluates the content of it into the engine's script parsing system. If the file is error free, it will load into the console & engine, otherwise it will print "Syntax Error" and the relevant line number in the console window.

You probably still have one little problem in your head. How can we use **exec** if the file isn't even loaded yet? The answer is surprisingly easy to comprehend. The engine, by default looks for one file at the loading step, **main.cs**, which is located in the same directory as the engine executable itself. The engine loads this file first, which contains some default settings and parameters. **main.cs** then loads the client and server versions of **main.cs**, which load other relevant files, and so on.

For our purposes right now, there are two ways to load a script file. The first is to edit one of these existing files to point to our file, or we can directly load the file in the engine through the console. If you've been following the code samples of this chapter and have them all in a file that is saved somewhere in your engine, you can load the console (~) and type **exec("path/to/file.cs")**; replacing the path with the path to the file, relative to the game's executable file. If you want to use the other option of editing an existing file however, you need to be mindful of a concept we'll be covering later in this guide in Chapter 12.

Client-Side Versus Server-Side

When you open the game's folder with the executable file, this is likely what you'll see in the folder along with the executable:



We're interested in that scripts folder. Inside there, you'll see another main.cs file, as well as three folders: client, gui, and server. I'll be going into very great detail about the client and server folders later on, but for now, you need to understand that scripts are loaded on either your client, or the server instance you're playing on. When it comes to things like GUIs on your screen, or PostFX rendering tools, these are handled on the client. But things like object instances, game settings, and world controls are handled on the server. This concept is the concept of a client-side script and a server-side script.

For our code examples so far, you'll want them on the client since they involve using the client's console dialog (~). Things we do here in a bit will use the server folder instead. If you want to use a single script file and load it on the client, place your new .cs file in the /client/ folder and then open the **init.cs** file. You'll notice near the bottom some additional exec commands that use syntax like **exec("./client.cs");** Notice the ./ in the path? This means to point to the relative current directory. Because exec uses the executable as the CWD (Current Working Directory) you either need to point the full path from the executable to the script, or you can use the ./ shortcut to set the CWD to the folder in which the **exec** statement is being called from, in this case /client/. So go ahead and add a exec line to the **init.cs** file using a similar syntax in the statement to load your script when the game loads.

Final Notes

Over the course of this chapter, we've covered all of the basics of using TorqueScript, and more specifically, computer programming in general. Please be sure to re-read any of the sections you may not have completely understood before moving on because from this point forward, this guide is going

to assume you have these basics down and in the books so the more advanced concepts can be taught to you.

If you're struggling with any of these concepts, it is highly recommended that you take a peek at the official engine's TorqueScript basics guide. (<http://docs.garagegames.com/torque-3d/official/index.html?content/documentation/Scripting/Overview/Introduction.html>) Some of these concepts will be taught to you in the coming chapters, but for the most part what you should understand by this time will be in there.

I hope you're picking apart what I have been teaching you so far, because now we're going to step things up in difficulty, and complexity in terms of code. For any references you need, you can find all of the code examples we did in this chapter in the files folder of this guide.

Chapter 7: Intermediate TorqueScript, Digging Deeper

Welcome Back! You've made it this far without closing the guide for headache purposes so let's move on! This chapter will expand on the topics covered in Chapter 6 in terms of technicality. If you skipped over that final notes page, please be sure you understand all of those concepts before coming through these next topics, because I'm going to assume you know how to do all of that stuff in Chapter 6 here, also, you're pretty much going to be expected to know what I mean when I say things like: "Print that variable, or print those variables, or write this function", things like that should be absolutely known by this point. But don't fret, because there are still some new things to learn, and I will cover those with more detail to ensure you understand how they work. So, now, let's dig deeper into the scripting!

Arrays

The first new topic to talk about is the concept of an **array**. Arrays are a type of variable that contain multiple values of a similar type all under one variable access name. Some people also refer to arrays as a list of values, but for our purposes, we will stick with the official designation **array**. To use an array in TorqueScript you need to use the array operator (`[]`). Unlike in programming languages such as C++, arrays in TorqueScript are always dynamically sized in nature, and require no prior definition.

Defining an Array

To actually define an array, you start with the similar variable naming convention with either the local or global variable definition, followed by the name; however, before you apply the application operator, you define the array operator, and provide it with an index value to specify what slot of the array you want the value to go in. Arrays begin at index 0 and increment by one.

```
%arrayVar[0] = 1; //Example of index '0'  
%arrayVar[1] = 2; //Example of index '1'
```

TorqueScript actually interprets this as two separate variables, one named **arrayVar0** and the other Named **arrayVar1**. However, using arrays instead of naming variables with an appended numerical value has some advantages. Before we move on however, you should also note that you can go backwards as well with that example and notion:

```
%arrayVar0 = 1; //Example of index '0'  
%arrayVar1 = 2; //Example of index '1'  
echo(%arrayVar[0] @ SPC @ %arrayVar[1]); //Outputs "1 2"
```

Working with Arrays

The real power of arrays however comes into effect when you combine it with looping structures to access multiple values within the array at a time. This is actually one of the big common practices of using arrays in any programming language, is to combine the for loop with the array to perform multiple operations at any given time.

To actually do this however, you need to have prior knowledge of the amount of elements in the array itself. The best way to accomplish this is by means of a global variable that stores the counter, or if the array is being populated and manipulated in the same function, a local variable. Let's look at some examples of this in action:

```
function arrayExample(%numberOfElements) {  
    //Start by populating the array  
    for(%i = 0; %i < %numberOfElements; %i++) {  
        %myArray[%i] = getRandom(1, 100);  
    }  
    //Print them... but backwards! Quick review: Does %i above go out of scope at this location? Answer: No, if you printed %i right  
    // here it would match %numberOfElements  
    for(%i = %numberOfElements; %i >= 0; %i--) {  
        echo("Array Index:@%i@: Value: @%myArray[%i]");  
    }  
}
```

So what we've created here is a basic function that accepts a number parameter to declare how many elements to create. The function then creates those elements and then prints them to the console in the reverse order in which they were created. Since we both created and used the array in the same function, we get away with using a local variable to store the counter (It's actually the parameter of the function in this case). Notice in this example, how we use the counter variable in the loop **%i** to access the array index. This is why you combine loops with arrays to obtain the desired result of working with them.

Next, let's look at a case where the index of the array has to be stored outside of the function itself by means of a global variable.

```
$EnemyTypes = 5;  
function fetchHighestValue() {  
    for(%i = 0; %i < $EnemyTypes; %i++) {  
        %enemyScore[%i] = getRandom(100, 5000);  
    }  
    //Fetch the highest.  
    %highest = 0;  
    %highIndex = -1;  
    for(%i = 0; %i < $EnemyTypes; %i++) {  
        if(%enemyScore[%i] > %highest) {  
            %highest = %enemyScore[%i];  
            %highIndex = %i;  
        }  
    }  
}
```

```
echo("Highest Enemy Score is: "@%highest@" located at index "@%highIndex);  
}
```

Again, we have the array being generated inside the function, but this time we have the stored number of elements being kept outside of the function itself in the global variable. We generate some numbers here to put in the elements and then use the second loop to pull out the highest value from those scores. We'll have much more discussions related to arrays when we cover more details on those Global Variables, but for now, let's expand your knowledge a little bit with some new insight on the index parameter.

The Index of an Array

So far you have seen some basic numerical applications of arrays. But there is so much more you can do with arrays than you think. Before we get there however, let's talk a bit more about that index parameter of the array. So far you have seen examples of arrays that use this format:

```
%arrayVariable[%numericalIndex] = %value;
```

But! This isn't the only format of an array that is allowed for the index parameter. Let's take an example here. We have a function that spawns some kind of enemy, and we need to tell the game some parameters about the enemy to apply. Now, it would be quite annoying to do something like:

```
%enemyParams[0] = 100; // [0] - Health  
%enemyParams[1] = 10; // [1] - Damage
```

So, instead, let's show you something else you can do here in place of what you've seen so far:

```
%enemyParams["health"] = 100; //Health  
%enemyParams["damage"] = 10; //Damage
```

That's right! You can actually use other variable types as the index parameter. TorqueScript simply converts the above to things like: **%enemyParamshealth** and **%enemyParamsdamage**. So the same backwards rules as earlier apply here just like they did with the numerical array, however just stick with the standard array syntax in your code, it reads much more easily than the long name format. To access these values in the code, you'd simply do something similar to the array format we did earlier, so for example, a print statement of these new array parameters would look like this:

```
echo("Enemy HP: "@%enemyParams["health"]@"\nEnemy Damage:"@%enemyParams["damage"]); //Remember what \n does?
```

There are many cases in the base engine itself with the **Full** engine template where both string and numerical index values are used here. And, you can still use the loops here, you'll just need to define a second array that converts the names to an "index", starting at 0, and then use that variable to convert to an index, and back from the index. Oh! And one last thing before we move onto the final section of the array sub-chapter. Don't think you only need to use numbers as the value of the array itself, I could do something like this for example:

```
%enemyNames[0] = "Bob";  
%enemyNames[1] = "Joe";
```

Multi-Dimensional Arrays

The last topic of the array sub-chapter is a concept called **Multi-Dimensional Arrays**. So far, we've been working with a single-dimensional array, or a 1-D array. You can expand your array to contain multiple dimensions (2-D, 3-D, etc.) as needed by the code. In other programming languages such as C++, each additional dimension added would equate to raising the total number of elements by a power of 2. In TorqueScript, this simply becomes another access type. Here's a basic example of a simple two dimensional array:

```
%my2DArray[0][0] = 0;  
%my2DArray[0][1] = 1;  
%my2DArray[1][0] = 2;  
%my2DArray[1][1] = 3;  
echo("Let's access element [1][0]: "@%my2DArray[1][0]);
```

Starting a multi-dimensional array is as simple as that. With each additional array operator appended to the end of the variable name, you add another level to the array variable. TorqueScript interprets that variable as: **%my2DArray1_0** for **%my2DArray[1][0]** as an example. Let's expand on the previous with some looping to print everything.

```
function basic2DArrayLoop() {  
    %my2DArray[0][0] = 0;  
    %my2DArray[0][1] = 1;  
    %my2DArray[1][0] = 2;  
    %my2DArray[1][1] = 3;  
    for(%i = 0; %i <= 1; %i++) {  
        for(%j = 0; %j <= 1; %j++) {  
            echo("Printing %my2DArray at ["">@%i@"][@%j@"]: "@%my2DArray[%i][%j]);  
        }  
    }  
}
```

What I used here is called a nested for loop to access all of the elements of this array, obviously when you use this, you're directly assuming the array itself to be consistent on sizes of %i & %j, to further expand that to accept arrays of all sizes of %j you'd introduce a simple error check that looks like this:

```
function another2DArrayLoop() {  
    %highestJ = 2;  
    %my2DArray[0][0] = 0;  
    %my2DArray[0][1] = 1;  
    %my2DArray[0][2] = 2;  
    %my2DArray[1][0] = 2;  
    %my2DArray[1][1] = 3;
```

```
for(%i = 0; %i <= 1; %i++) {  
    for(%j = 0; %j <= %highestJ; %j++) {  
        if(%my2DArray[%i%j] $= "") {  
            break; //<- This cancels out the %j loop, but leaves the %i loop running.  
        }  
        echo("Printing %my2DArray at [%@%i@%][%@%j@%]: %@%my2DArray[%i][%j]);  
    }  
}  
}
```

Lastly, as previously discussed in the index section, array index values are non-restrictive based on type. You can even mix types between different indices:

```
%enemyParams[0]["name"] = "Joe";  
%enemyParams[0]["health"] = 100;  
%enemyParams[0]["damage"] = 10;  
%enemyParams[1]["name"] = "Bob";  
%enemyParams[1]["health"] = 250;  
%enemyParams[1]["damage"] = 20;
```

The same access rules as mentioned before would apply in this case. By combining these concepts together, you can trim a lot of unnecessary code out of your project by compacting large variable structures into an array instance which can then be used anywhere else in your code, or by means of a simple looping structure.

Topics with Global Variables

The next topic to cover in this guide is the concept of a Global Variable. We've used a few Global Variables back in Chapter 6, and a few cases with the arrays, but now we need to actually have a nice discussion about what you can do with these. A Global Variable is a variable that is unique and can be used on the full instance of the application. By instance, I refer to the client instance or the server instance (Chapter 12). A Super-Global Variable or in C++ terms a **define** is a variable that can be used on the entire game application, client and server instances. Global variables have numerous applications and usages in Torque-Script, so let's talk about those.

Constant Values

The first big application of a Global Variable is to setup a constant value definition. This would be something like the volume of your game's application, or for gameplay mechanics, the maximum score a player or team needs to reach in in order to win, or even things like a time limit for the game. Global Variables allow you to set a variable to a value, and call it up again at any time once it has been defined. To define a global variable, you use the dollar sign (\$) symbol, and then follow the same conventions of the local variable definitions. Here is an example of the Global Variable in action with some basic counter type functions:

```
$myConstantNumber = 27;  
$numberOfCalls = 0;  
  
function globalVariableBasics() {  
    $numberOfCalls++;  
  
    echo("The global variable $myConstantNumber is: @"$myConstantNumber@", you have called this function @"$numberOfCalls@"  
times.");  
}
```

Now call this function a few times. You should see the first number (**\$myConstantNumber**) always print out the number 27 (unless you change it), and the counter increase each time you run the function. Global Variables, unlike their local counterparts do not go out of scope once the function has run. Instead, they are kept in memory and can be accessed at any time and changed at any time as needed by the purpose of the application.

Data Enumerations

One of the big uses for Global Variables is the concept of **enumerating** a specific value to a named variable. We'll get to this topic with much more detail in the C++ chapters, but for TorqueScript, there isn't too much you need to worry about. Basically, the concept here is you take a Global Name, and assign it to a number. Then, you can use that flag in conditional statements in place of throwing around numbers at all times which could become suspect to corruption or change beyond the values you specify as enumerated variables. Here's an example (Try to ignore the **namespace operator** (::), we'll cover that in more detail in the next chapter):

```
$Statuses::Fire = 0;  
$Statuses::Ice = 1;  
$Statuses::Electric = 2;  
  
function setStatusEffect() {  
    %effectToApply = getRandom(0, 2);  
    switch(%effectToApply) {  
        case 0:  
            %e = $Statuses::Fire;  
        case 1:  
            %e = $Statuses::Ice;  
        case 2:  
            %e = $Statuses::Electric;  
    }  
    return %e;  
}
```

By doing this, you can ensure by means of forcing the variable to be set by the **\$Statuses** variable, that the desired effect always fits in the numerical value range you have specified. In our case here, it will always be between 0 and 2.

Global Array Instances

As you likely guessed, you can also set up an array on a Global Variable instance, and you use the exact same syntax you were just taught, but replace the local variable definition with a global one instead:

```
$enemyParams[0][“name”] = “Joe”;  
$enemyParams[0][“health”] = 100;  
$enemyParams[0][“damage”] = 10;  
$enemyParams[1][“name”] = “Bob”;  
$enemyParams[1][“health”] = 250;  
$enemyParams[1][“damage”] = 20;
```

If you were to load this script, the entire application on the instance (client/server) would be able to use the \$enemyParams array to pull information out for a specific enemy instance. This is actually the best way to define constant values for game objects, is by means of globally defined arrays on the server. Feel free to experiment with all of these applications of global variables, you will find them to have countless uses and to be one of the more valuable tools in your programming arsenal.

Timers, Indirect Function Calls

At this point in time, I have now taught you all but one of the “core” concepts of the engine. We’ll get to that final topic in Chapter 8, but now this is where the primary aspect of Chapter 7 (Intermediate TorqueScript) comes into play. So, from this point forward, you should have an understanding of all of the basic terminology of the engine and how to set up the basic things. Be sure to go back and read over anything that didn’t make sense beforehand.

The next topic we’re going to cover is the topic of **Timers** or basically setting up an event that is run after a specified amount of time. This will lead into the topic of Indirect Function Calls. You may recall in the prior chapter I talked about how we’ve only been using a direct function call, now we’ll expand that knowledge a bit. But before we get there, we’ll start by introducing the **schedule** function.

Schedule & Cancel

This is going to be your first taste of a topic we’re going to begin to discuss in Chapter 8, and that is to use variables to set up references to objects or functions. A reference by definition is a variable that holds an identifier to something in memory, be it a function, and object, or any instance in your program. We’ll be using this concept to employ a timed function call. This introduces what is known as an **indirect function call** or basically a function call that cannot be linked directly to another function, or by something you’ll understand, a function that cannot return a value to the calling function.

To begin a timed function call you use a function defined by the engine called **schedule**. Here is the syntax of the schedule command:

```
%scheduleIDReference = schedule(MSTime, ObjID, ‘functionName’, %arg1, %arg2, ..., %argN);
```

So, let's go over each point of this call so you understand what's going on here. Now, the first thing is the %scheduleIDReference, this is a variable reference to the schedule instance itself. You do not need to actually include this in the schedule call, but if you plan on using the **cancel** command, you'll need this. Next is the actual function name **schedule**. The presented syntax here is the standard non-object schedule call, we'll get to the other version in Chapter 8. The first parameter in the function is the time (in Milliseconds [1000 = 1 second]) until the function is called. The next parameter is a reference ID to the object in question. For most cases, you'll leave this as **0**. The third parameter is the name of the function you're calling, enclosed in single quotes (this is what is known as a **tag** [Chapter 12]). Following that is your standard function parameter list (recall the whole %arg1, %arg2, ..., %argN talk we had back in Chapter 6). You won't necessarily need all of these, only the amount your function requires.

So, let's look at some examples of the **schedule** command here. First, we'll go back and expand on our old hello world example by making it delayed:

```
function sayHello(%moreText) {  
    echo("Hello Computer...");  
    schedule(2500, 0, 'helloWorld', %moreText);  
}  
  
function helloWorld(%moreText) {  
    echo("hello world!" @ " " @ %moreText);  
}
```

When you call `sayHello()`, the console will print “Hello Computer...”, and then after two and a half seconds, it will print “hello world!”. The `%moreText` parameter is shared between both functions, so if you add a text parameter to `sayHello()`, the `helloWorld()` indirect call will receive the variable and print it along with Hello World.

Now, let's introduce a “timer loop”. This is a loop of forms, but not enough to be designated as one. Essentially, you call the same function over and over again until a condition is met.

```
function timerLoop(%counter) {  
    if(%counter > 10) {  
        echo("I'm done!");  
        return;  
    }  
    echo("Counter At @" + %counter);  
    schedule(1000, 0, 'timerLoop', (%counter + 1));  
}
```

This function will run once every second until the number is greater than 10, then the function will stop. To initiate it, simply call timerLoop and send a numerical variable less than 10 to the function.

So, this would be great, but what if you send it -1000, you'd have a function running for hours. Now, let's introduce the **cancel** command. The cancel function accepts a schedule reference id as a parameter, and when it's called, it will stop a schedule. Let's for example re-do the above example using the cancel command now:

```
function timerLoop(%counter) {  
    if(%counter < -100 && isEventPending($schedRef)) {  
        echo("Number is too far away, killing schedule.");  
        cancel($schedRef);  
        return; //<- Don't forget this return, otherwise the function will continue!  
    }  
    if(%counter > 10) {  
        echo("I'm done!");  
        return;  
    }  
    echo("Counter At @" + %counter);  
    $schedRef = schedule(1000, 0, 'timerLoop', (%counter + 1));  
}
```

So, let's discuss this addition. Now, there's a check to see if the sent number is less than -100, but what's that second part? The function **isEventPending** is a timer related function that accepts a schedule reference id as a variable. If there is a schedule on the ID and it has not yet been called, this will return true. So basically, if there is a schedule pending, and the number is less than -100, the schedule and loop will be terminated. The only other change made was the addition of the **\$schedRef** variable to the schedule call to actually grab and store that reference ID.

More on Indirect Function Calls

All of the work we've done here classifies as an **indirect function call** because a scheduled command can be terminated at any time by **cancel**. If this is the case, you won't get a return call expected and will trigger an error / crash in the calling function. This is why you cannot use the **return** statement in an indirectly called function to return a value to the calling function, you can only use it to stop the function at the specified point. To return a value to the calling function in a scheduled function, you need to use a global variable, or as you'll learn in Chapter 8, an object parameter.

Indirect function calls have many uses, and you'll see that when you try to set up mechanics in your game such as events or things that need to happen at specified time intervals, that they will become essential to your game instance.

Topics with String type Variables

Now, we're going to dive away from those topics into something a little more useful, and that is String Data. As you've seen already, there are a few different forms of variable types you can use, but the most important of all of them in this engine is the String Type, and that is because TorqueScript treats all variables as Strings. So, let's expand on your knowledge of this data type a bit.

String Keywords

Space

As you may have seen used a few times in functions, there are some special keywords you can use when working with strings to add contents to a string. Let's go over these keywords. The first is the shortcut to add a space to a string, which has the notion of **SPC**, here's an example from earlier:

```
function helloWorld(%moreText) {  
    echo("hello world!" @ SPC @ %moreText);  
}
```

This shortcut can be used in variable definitions, just as well as echo statements, so let's say you wanted to add a space between two numerical variables without needing to differentiate them:

```
%twoNumbers = 1 SPC 2;
```

Go ahead and use the `echo()` command on the `%twoNumbers` variable and see what you get. The space between these two numbers equates to what is known as a **word** to strings, we'll get to what you can do with that in just a bit.

Tab

The next keyword you have access to is the **TAB** keyword, which does exactly as it says, it adds a horizontal string tab at the specified location. Here's a quick sample:

```
function helloWorld(%moreText) {  
    echo("hello world!" @ TAB @ %moreText);  
}
```

Now if you call this function, you'll notice instead of a space, there's a carrot key (^) in its place. In TorqueScript this equates to a **field** separator for strings, we'll get there in a bit too. However, there's also a tab shortcut when you're inside string data itself:

```
%tabList = "one\ttwo\tthree";
```

Notice in the string we use the `\t` to designate an internal string tab. You can use this if you're inside a string and need to add a tab at the specified point without needing to concatenate the TAB keyword to the points.

Newline

Finally, we have the newline break. The keyword for newline is **NL**, and the separation between newline breaks in the engine is called a **record**. All it does is insert a new line at the specified point. If you have multiple lines of information to print to the console, this is how you do it:

```
function helloWorld(%moreText) {  
    echo("hello world!" @ NL @ %moreText);  
}
```

We used the other shortcut a few times already, so let's bring up one of those old examples:

```
echo("Enemy HP: @"%enemyParams["health"]@"\nEnemy Damage:"@%enemyParams["damage"]); //Remember what \n does?
```

The **\n** terminator is the internal string shortcut to the newline break, so instead of needing to concatenate the **NL** keyword, you can simply inset the **\n** terminator sequence to the desired point.

Words, Fields, and Records

With these in mind, let's expand on the use of the above topic. The main reason I introduced the concept of spacing, tabbing, and newline breaks to you in strings is because of the three notions of **words**, **fields**, and **records** in strings. Torque treats the space, tab, and newline characters as token characters that can be used to strip out parts of the string between these two points. This is the first of what is called an **extraction function**. I'll teach you a few more of those later on, but this part is about the concept of words and fields.

Words

A word to Torque is any string of characters that ends with a space token. They can be treated as an array of characters that put together a string. For example the following string: "This is a string" would have 4 words, and it could be treated as an array like so:

```
%str[0] = "This";  
%str[1] = "is";  
%str[2] = "a";  
%str[3] = "string";
```

The engine would then assemble the string by placing a space character after the end of each array element where the index is less than the maximum index. The primary use of the **word** element however comes with five brand new functions for you to learn. These functions are: **getWord**, **getWords**, **setWord**, **removeWord**, and **getWordCount**.

The **getWord** function is used to strip out a single one of the "Array element" instances from the string. Essentially it would be like calling **%word = %str[2]** for example, however this notion believes that you are sending a pure string instance that is not split into arrays, this function basically takes the necessary work of making the array elements out of it. Here's an example of the **getWord** function in action:

```
function getWordExample() {  
    %string = "This is a nice long string instance";  
    %word4 = getWord(%string, 4); //<- "long";  
    echo("The fourth \"word\" in \"@%string@\" is \"@%word4@\" //<- The \\\" termination sequence allows you to insert a \" character in a  
string without needing to close it.  
}
```

Notice the parameters of the function, and how the numerical value parameter works like an array index (where the value starts at 0). Next up is the extended version of the above function, `getWords`, which allows you to access multiple word instances at once.

```
function getWordsExample() {  
    %string = "This is a nice long string instance";  
    %firstThree = getWords(%string, 0, 2);  
    echo("The first three \"words\" in \"@%string@\" are \"@%firstThree@\";  
}
```

This one's a little more interesting. The first two parameters are the same as the first function, the string instance, and the targeted index, but this is for the starting location. The third parameter here isn't the number of words to go out, but instead, the end index. So for our case here, we want words 0, 1, and 2, so I set the value of the end index to 2.

The next word function is called `setWord`, and this is used to replace a word in a string with another word instance. Here's how you use it:

```
function setWordExample() {  
    %string = "This is a nice long string instance";  
    %newStr = setWord(%string, 3, "very"); //<- Replace the word nice with the word very.  
    echo("The new string is: \"@%newStr@\";  
}
```

Easy enough to understand again, this time, the third parameter of this function is used to tell what word (or words) you want to replace at the desired point in the string. Next up is another easy function, `removeWord`, which is used to remove a single word from the string:

```
function removeWordExample() {  
    %string = "This is a nice long string instance";  
    %newStr = removeWord(%string, 3); //<- Remove the word nice from the string.
```

```
echo("After removing word 3, we have: @"%newStr);
}
```

Finally we have the `getWordCount` function, which returns the number of word indices in the string. When you want to iterate through a word delimited string in a `for` loop, you can use the `getWordCount` function to receive the number of elements to run through:

```
function getWordCountExample() {
    %string = "This is a nice long string instance";
    for(%i = 0; %i < getWordCount(%string); %i++) {
        echo("Word @"%i@": @"getWord(%string, %i);
    }
}
```

So by combining these functions in various forms and looks, you can deploy powerful tools that work with string data to pull out certain parts of the string you're interested in, and remember, **ALL** variable data in torque is treated as a string, so these functions are valid on all of your variables.

Fields

A **field** is another token delimited string instance, but this one breaks at a TAB instance. It has the same parameters and functions of the word, however you simply replace **word** with **field**: `getField`, `getFields`, `setField`, `removeField`, `getFieldCount`. When you want your tokens to have spaces in between the information, use the field functions instead. You can also combine words and fields in your functions, take a peek at this example:

```
function fieldsExample() {
    %string = "String One\tString Two\tString Three";
    %field1 = getField(%string, 1); //<- String Two
    %first2 = getFields(%string, 0, 1);
    %newField1 = setField(%string, 1, "String #2");
    %minus2 = removeField(%string, 2);
    %fCount = getFieldCount(%string);
    for(%i = 0; %i < %fCount; %i++) {
        echo("Field @"%i@": @"getField(%string, %i));
        for(%j = 0; %j < getWordCount(getField(%string, %i)); %j++) {
            echo("Word @"%j@": @"getWord(getField(%string, %i), %j));
        }
    }
}
```

```
}
```

```
}
```

Records

The highest tier of string extraction however comes with the topic of a record. This has the same properties as before, where your function names have the same naming, but in this case replace field with record: **getRecord**, **getRecords**, **setRecord**, **removeRecord**, **getRecordCount**. The separator character here is the newline break, you shouldn't need an example here with the knowledge of the formatting and parameterization of words and fields, so we should be good here!

So there you have it, the topic of words, fields, and records. You will find that you can do some impressive coding feats by means of using words and fields in your code to solve complex problems that may require string data.

Basic String Functions

The important thing here is to remember that a string is basically an array of characters. So, the engine provides you with a set of tools and functions to help you do some operations with these strings and characters. So let's talk about these tools now.

Strasc

The first thing I want to show you is how strings and characters are interpreted by a computer. You may recall from a while back, I mentioned how numerical data is translated via the ASCII table to your characters that make up a string. Now, let's go backwards. The strasc function accepts a single character as a parameter and will spit out the numerical ASCII value to you.

Strformat

For the more programming related folk, you may know of a really basic C++ function called printf. This function allows you to place a formatted string instance as a variable with some tokens in the formatting to be used as variables. Strformat is the Torque scripting language version of this function. Here's an example:

```
%hexStr = strformat("The Hex of 255 is: %x", 255); //Convert 255 to its HEX variant
```

Strlen

This function will probably become one of your most employed tools when working with strings. Strlen accepts a string as a parameter and will return the numerical length of the string when counting each character in the string.

Strstr

The Strstr function is used to locate a string instance within another string instance. It has two parameters, the first being the full string, and the second being the string you are looking for. It will return the numerical start position of the string if it found it, or -1 if it didn't find the string you're looking for in the large string instance. Here's an example:

```
function strStrExample() {  
    %string = "This is a nice long string instance";  
    %find = strstr(%string, "nice"); //<- returns 10.  
    echo("The start of \"nice\" in \"%string@\" is at: \"%find\"");  
}
```

Strpos

Strpos works on the exact same premises of Strstr, however, it has a final parameter added to the first two to define the starting position of the string to look from. Remember, a String is an array of characters, so the first character of the string has the index value of 0.

Ltrim, Rtrim, and Trim

These next three functions are used to remove **trailing** whitespace from your string. Trailing whitespace is basically excess spaces and tabs located outside of any letters or numbers in your string instances. For example “_this is a string_”, the trailing whitespace here is underlined in red. **Ltrim** allows you to only remove the trailing whitespace on the left end of the string, **rtrim** removes it from the right, and **trim** removes all of the trailing whitespace.

Strlwr, Strupr

These next two functions are used to make an entire string all lowercase or all uppercase, they accept a string as the parameter input. When you’re using a direct comparison function and don’t need to worry about case-sensitivity of the string, be sure to use one of these two functions first to ensure both strings are either uppercase or lowercase.

Strchr, Strrchr, Strchrpos, Strrchrpos

Strchr is used to find the first occurrence of a specified character in a string, and Strrchr is used to find the last. They both use the same set of parameters in the definitions, where the first parameter is the string you’re searching in, and the second parameter is the character you’re looking to find in the string instance.

The two functions Strchrpos and Strrchrpos do the same as their smalled named counterparts however they have a third parameter to specify the starting index of the string to look at.

All of these functions will either return the numerical position of the string the character was found at, or a -1 if nothing was found in the string.

Isspace, Isalnum

Finally, we have the two functions isspace and isalnum. The isspace function is used to detect if the specified character in a string is a whitespace character (Space, Tab, Newline). The syntax of the function accepts the string you’re looking at, and the index of the string to check. Isalnum has the same syntax, however, that function checks if the character you’re looking at is an alpha-numeric (0-9, a-z, A-Z) character.

There are a few more functions in the engine beyond the scope of this guide, but if you're interested in learning more about these tools at your disposal, be sure to read over the engine's scripting documentation regarding string functions.

Comparison Functions

Next up, we're going to talk about functions that are used to compare strings with one another. You have a few tools at your disposal to accomplish these tasks. Always be sure to remember the fact that a string is an array of characters when using these functions. This is very important, and you need to understand this fact otherwise your resulting answers might not come out as you expect them to.

\$= Operator

The first comparison tool, I actually taught you back in Chapter 6. That is the string equal to operator or the **\$=** operator. This is a direct comparison function you can use to directly check if one string is equal to another string. If you need help using this operator, read the relevant logical operators section in Chapter 6.

Strcmp, Stricmp, Strnatcmp, Strinatcmp

These four functions make up the collection of string comparison functions in the engine. They all have the same parameters, and the two parameters are the two strings you are comparing with one another. The only difference is how these two functions perform the comparison operation. The return value of these functions work by returning a 0 when the two strings match the condition completely, return a value less than 0 if the character that violates the difference has a smaller code in the first string than the second string, and a value greater than 0 if the code is larger in the first string than the second one, the code here referring to the ASCII value of the character.

Strcmp is a complete equality check, it will only flag a match if both strings are **completely** equal to one another, which means the strings are case-sensitive. Stricmp will flag a match if the strings are equal to each other, but this function does not require the case of the characters to match. The last two functions work the same as their similarly named counterparts, but they use what is called the natural order comparison, which is useful when comparing numerical strings. Instead of flagging a difference like "10" and "2" as less than zero because 1 is less than 2, it will actually treat the entire string as a number, hence, it will compare 10 to 2, so the return value will be greater than 0.

Startswith, Endswith

These two functions are used to test if a string begins or ends with the specified string. It has three parameters, and you only need to send two of them to the function. The first parameter is the string you're checking, the second parameter is the string you're checking against (if it starts or ends with this), and finally, you can send a third parameter (true or false [default]) to test for case sensitivity of the desired string. This function returns true or false based on the result.

Extraction Functions

Finally, we have the string extraction functions, which are used to either remove certain aspects of a string from the instance, or to pull out parts of the string as a second string result.

Stripchars

The stripchars function is used to remove certain characters from a string instance. You can have as many characters you want in the targeted for removal parameter, if a character is in the remove list, and it's present in the string it will be removed. Here's an example:

```
function stripCharsExample() {  
    %string = "TooManyChars";  
  
    %new = stripChars(%string, "aAeEiloOuU"); //Removes Vowels: "TMnyChrs"  
  
    echo("If I remove the vowels from \"%@%string@\", I get \"%@%new\"");  
}
```

Strreplace

The strreplace function is used to replace all instances of a string with another inside a string instance. There are three parameters to this function, the first parameter is the desired string to perform the replace operation on, the second parameter is the string instance you want to replace, and the final parameter is the string instance you want to replace it with.

getSubStr

The final function on our list is getSubStr. This function is used to extract a **substring** from a function, or basically a portion of the string. There are three parameters to this function, the first parameter is the string instance you want to pull a substring from. The second parameter is the starting position (index) of the string to pull from, and the final parameter is the number of characters to go out. Alternatively you can leave this parameter empty or send it a -1 to remove all remaining characters from the string.

Closing Notes

So, you should be starting to see some patterns here. Once you get the basics down, things aren't that hard to manage in terms of scripting. Now, we're going to introduce the final core concept of the engine's scripting language to you, and then everything should be starting to click together so we can introduce some more complicated control structures and coding pieces. This next chapter is where we'll introduce objects and classes to you, and actually step out of the console, and into the game world.

To help you along the road in the engine, you should now have enough comprehension to fully tackle the engine's scripting documentation to get a more complete list of the functions available to you in the engine. There are a few more minor things to teach at the start of Chapter 8, but the basics, you should now completely understand.

Chapter 8: Game Objects, Classes, Packages

You should now have a complete understanding of the basics of TorqueScript, which holds a great level of importance moving from this point forward. I slowly started to break you away from teaching the language core concepts in favor of teaching important functions and how to use them late in the prior chapter, and that will continue in this chapter, however, there's one final important language concept I need to teach you to do, and that is how to work with game objects, classes, and the concept of packages.

Chapter Introduction

The concept of objects in most programming languages comes from their source of being what is called an object-oriented programming language, or a language where the core sources of functioning comes from a set list of individual classes and elements of those classes. Each object in the engine has a list of variables that are directly attached to the object; therefore, an object can be transmitted across sources such as functions and the networking system of the engine with multiple pieces of data attached to it.

This chapter will focus on the primary teaching of using objects in the engine, and how to properly set them up. I'm also going to introduce some more Torque Related topics of importance for you to know and understand, finally, we'll cover some topics relating to computer programming and Torque to get you prepared for the C++ chapters coming up here in a little while. Once you're done with this chapter, you'll have a complete understanding of all of the basics of the engine's programming language, and we'll be ready to move onto topics that expand on these concepts to actually build your game.

Introduction to Objects in Torque

As with almost everything else you've done in the engine in terms of scripting up to this point, Objects are not too much more difficult to understand. They too, are attached to variables, such as local and global variables, but there are a few key differences that objects have in relation to your standard variable definitions:

- Objects are permanent instances, each having an ID associated to it, this ID can be called to pull up the object any time after it is created, but not after it is deleted.
- Objects can also have a name associated to it, and this name can be used to pull up the object instance in code at any time.
- Objects have a set of pre-defined functions attached to them, and can be programmed by you to have more functions.
- You can attach a variable of any type and name (so long as it meets the naming criterion (CH.6)) to the object and use it at any time you have access to the object.

We'll get to all of these topics as we move along, but first, let's talk about how objects are actually created in the engine.

Creating an Object

In Torque, all objects have a **class** that is associated to it. You briefly got a glimpse at some of these in Chapters 3 and 4. That's right! All of the objects you placed on the map, or in your GUIs are actually defined as a class instance, and each class instance has its own set of properties and settings on it. The first type of object (class) instance we're going to work with is called a **ScriptObject**. This is the most basic type of object instance in the engine, and its only purpose is to serve as an object instance that exists in the scripting language. To create an object in the engine, you need to use the **new** keyword and provide it with some variables and definitions, let's look at the syntax first, and then an example:

```
%objRef = new ObjectClass(ObjectName) {  
    //Parameters  
};
```

This is the most basic form of object creation here. Notice the semicolon on the end of the closing bracket, when you work with an object, class, or package definition, the ending bracket must have one of these on it. I also underlined the object reference variable part, because you don't even need to include that in the creation, it's just good practice to include it.

Now, let's actually have a basic example function to show you how to work with an object in the engine, again, we'll use the **ScriptObject** class here:

```
function basicObjectExample() {  
    %newObj = new ScriptObject() {  
        class = "ScriptObject";  
        number = 1;  
    };  
    echo("The ID of this object is "@%newObj);  
}
```

So, let's talk about what's going on here. First off, in the object definition block, we're creating an un-named **ScriptObject** instance. You can do this, because the reference ID is being stored in the **%newObj** variable. If you were not going to provide the reference ID variable, then you should provide an object name, you could still make do without it, but it's not good practice because then you'd have a "floating" object instance, that you couldn't delete in memory, which is bad. The very first line in the example there isn't even needed, but I'm providing it to prepare you for an upcoming topic. That line defines the class of the object as a **ScriptObject**. This may seem confusing to you right now, since we're already creating a **ScriptObject** instance, but I'll explain further here in a bit. Finally, I'm creating an object parameter named **number** in this object instance, and setting it to 1. The last thing this function does is print the ID of the object to your console. When you create an object, if you print the object reference variable to the console, it will print the ID of the object.

Object ID's and Names

Before we move on, let's talk about something important regarding objects, and that is their identification numbers. If you were to run the above method, I could not tell you the exact output of the console, only because each game instance almost always has a different amount of object instances created at startup, and then when the game is running, every object in the game is treated as an object instance. For a great example, run that above code 2 or 3 times, you'll notice the ID of the object changes with each successive run of the function, **HOWEVER**, this is the important part, you have not actually removed the existing objects, they're still there as well! The function above only creates the object instances.

Let's talk about object names now. I'm going to provide you with a code sample below, I'll explain everything afterwards, but just have a look at this.

```
function namedObjectExample() {  
    if(isObject(mySampleObject)) {  
        echo("Detected existing object, deleting.");  
        mySampleObject.delete();  
    }  
    new ScriptObject(mySampleObject) {  
        number = 1;  
    };  
    %id = nameToID("mySampleObject");  
    echo("The ID of this object is @%id");  
}
```

So we did a few things differently this time around, let's talk about the different things in here. First off, is the error check right at the beginning of the function. The **isObject** method tests if a named object (or ID) instance exists in the game instance and returns true if it does, and false if it does not. If the object does exist, we print a message out and then the next line, **mySampleObject.delete()**; deletes the object. We'll get to the **.delete()**; part here in just a bit, but for now, you just need to know that deletes the object. We then proceed to create a new object, no reference variable, but with a name this time and give it the same number parameter. Finally, we have the **nameToID** command, which can take a given object name instance, and pull its ID out for you. Remember, no two objects in the engine can share the same name (Chapter 3), so only one instance will be returned by the function, or -1 if no object with that name is found.

Object Parameters and the Access Operator

So now you know how to create an object. Let's talk about actually using the object in say some functions. Before we get there however, I'm going to expand on our first example, to show you how to

print the number variable inside the newly created object. Doing so requires something called the **access operator**.

```
function basicObjectExample() {  
    %newObj = new ScriptObject() {  
        class = "ScriptObject";  
        number = 1;  
    };  
    echo("The ID of this object is "@%newObj@\nThe number parameter is "@%newObj.number);  
}
```

The important part here is bolded. The access operator (.) allows you to access an object's parameters and functions wherever needed. This operator actually works in two ways. You can use it to "access" or read the value from the object, or "modify" or change the value stored in the object:

```
function applySettingExample(%newNumber) {  
    %newObj = new ScriptObject() {  
        class = "ScriptObject";  
        number = 1;  
    };  
    %newObj.number = %newNumber;  
    echo("The ID of this object is "@%newObj@\nThe number parameter is now "@%newObj.number);  
}
```

Finally, you can use the access operator to create new object parameters or variables to store in the object, here's another example:

```
function newParameterExample(%newName) {  
    %newObj = new ScriptObject() {  
        class = "ScriptObject";  
    };  
    %newObj.myName = %newName;  
    echo("The ID of this object is "@%newObj@\nMy name is "@%newObj.myName);  
}
```

So, with this basic overview of information regarding creating an object and using values stored inside the object, let's expand on the knowledge here along with some prior knowledge to teach you how to use objects in functions. So, let's do another math example:

```
function objectMath(%objectsToSpawn) {  
  
    for(%i = 0; %i < %objectsToSpawn; %i++) {  
  
        %newObj[%i] = new ScriptObject() {  
  
            class = "ScriptObject";  
  
        };  
  
        %newObj[%i].number = getRandom(0, 1000);  
  
        echo("Math Object "@%i+1@": ID:"@%newObj[%i]@", Number:"@%newObj[%i].number);  
  
    }  
  
    echo("Spawned "@%i+1@" objects., Performing addition.");  
  
    %result = 0;  
  
    for(%i = 0; %i < %objectsToSpawn; %i++) {  
  
        %result += %newObj[%i].number;  
  
    }  
  
    echo("The Answer Is "@%result);  
  
}
```

This time, we combined a lot of the older concepts with some new stuff here. First off, we created an array of object references, and then we looped through those instances to pull out the number we generated and printed our result to the console. Before we move on, let's talk about one last thing, and that is using objects in multiple functions. So, you may have heard me mention this many times before, but all parameters in functions are treated as local variable instances, even your object references can be sent as parameters to functions, so by sending object references to functions you can use them in functions. Here's an example expanding on the prior.

```
function objectMathExampleTwo(%objectsToSpawn) {  
  
    for(%i = 0; %i < %objectsToSpawn; %i++) {  
  
        %newObj[%i] = new ScriptObject() {  
  
            class = "ScriptObject";  
  
        };  
  
    }  
  
    echo("Spawned "@%i+1@" objects., Performing addition.");  
  
    %total = 0;  
  
    for(%i = 0; %i < %objectsToSpawn; %i++) {  
  
        performObjectAddition(%newObj[%i]);  
  
        %total += %newObj[%i].number;  
  
    }
```

```
echo("Addition complete, result is: @"%total);
}

function performObjectAddition(%objRef) {
    %objRef.number = getRandom(0, 1000);
}
```

So we've got some more stuff going on this time. The objects are created right away, but we don't assign a numerical parameter to it, instead, that is handled by a second function defined below. So pay close attention here, when we send the object to the **performObjectAddition** method, we use the variable **%newObj[%i]**, because this is the defined reference variable in the method **objectMathExampleTwo**, however, once the object reference is inside the body of **performObjectAddition**, it now is stored in a non-array variable instance of **%objRef**, allowing us to modify the original object defined in **objectMathExampleTwo** in the function **performObjectAddition**. Once the number is applied in the method, we can then access it in the original function to add up a total number.

The takeaway point from this final example is that you need to always be mindful of what your variables are named and what functions they belong to. Always keep your scope rules in check when working with objects, it will make programming with them much easier to manage.

Classes, Object Functions (Direct & Indirect), and the Dump Command

Our next segment will explore the definition of the **class** as it relates to Torque. Classes in programming are the primary source of control when it comes to objects, they define the functions and properties of each individual object, and the object is simply the created instance of a class. There are a few things to learn in regards to this topic before we get to the programmatical side of things.

Defining a Class, what is a Super-Class?

So, a little while back I gave you a little bit of example code and told you to not worry about the line of code that had the word **class** in it, now I'll show you how to actually use this keyword. So, let's go back to the first example, and make a little change.

```
function basicClassExample() {
    %newObj = new ScriptObject() {
        class = "MyClass";
    };
    echo("The ID of this object is @"%newObj);
}
```

So, what exactly did this do? Well, from your perspective, absolutely nothing but from the perspective of the engine, you've introduced a brand new class instance named **MyClass**. You may think

for a moment that this is a bad thing because now the object won't do anything right, but actually, if you were to re-do all of the above examples with the class set to MyClass, you'd notice no change in the output of the code in terms of functionality. And that is due to the introduction of a new concept called **inheritance**, or basically, we're inheriting the members of another class instance, called a **super-class**. In this case, the super-class of our example here is the **ScriptObject** class. Basically put for an easy to understand example, the **class** is the instance of code for your object, and the **super-class** contains any and all other needed instances of code to derive your new class from. I say any and all because say you inherited a game object class. That object would then inherit its source class, which would inherit its source class and so on and so forth until you got to the highest object tier in the engine, a concept called **multiple inheritance**.

Multiple Inheritance

The concept of multiple inheritance has pretty much just been explained to you, so let's actually build on it with a few examples. We'll start by introducing a simple class instance with a few parameters.

```
function basicMultipleInheritanceExample_One() {  
  
    %newObj = new ScriptObject() {  
  
        class = "MyClass";  
  
        superclass = "ScriptObject"; //<- Demonstration purposes only.  
  
        numberVal = 10;  
  
        name = "Bob";  
  
        objType = "Basic";  
  
    };  
  
}
```

So here's a basic class example with some parameters attached to it. By default, this class instance will already pull the associated fields from the **ScriptObject** class into this new class instance. By definition, that makes the **ScriptObject** class the super-class of our new class here. The line setting the superclass in this case is redundant, but it's there to show you this point. Now, let's build on this example by creating a second class instance, but inherit from the one we created here.

```
function basicMultipleInheritanceExample_Two() {  
  
    %newObj = new ScriptObject() {  
  
        class = "MyClass";  
  
        superclass = "ScriptObject"; //<- Demonstration purposes only.  
  
        numberVal = 10;  
  
        name = "Bob";  
  
        objType = "Basic";  
  
    };  
  
}
```

```
%newObjTwo = new ScriptObject() {  
  
    class = "MySecondClass";  
  
    superclass = "MyClass";  
  
    numberVal = 20;  
  
    coolField = true;  
  
};  
  
}
```

So a quick discussion here, you can see that we've established a second class instance named **MySecondClass** in this code bit and that it by definition is a **ScriptObject** instance. However, this time it pulls the information from the class named **MyClass**, which is the superclass of **MySecondClass**. This in turn will load all of the information from **MyClass** into **MySecondClass**, and in extension it will detect the superclass setting on **MyClass** is from **ScriptObject**, therefore it will load all of the information from **ScriptObject** into **MySecondClass**. After this line is loaded, it will see **numberVal = 20** and set the field **numberVal** to 20, overriding the default setting in **MyClass**. If you were to grab the fields name and **objType** from **%newObjTwo**, they would match those in **MyClass**. Finally we have a new field that would be allowed for use in **%newObjTwo**, but not in **%newObj**.

Multiple inheritance is a top down structured style of script, it starts at the highest class instance and works its way through all other definitions of superclass until it reaches your class instance, loading all of the fields and methods associated with the class as it moves down the structure.

Direct Inheritance

So that's one way to inherit properties of a class into another class, but we can also directly inherit the properties of a defined object into another object by means of **direct inheritance**. To directly inherit the properties of an object, we use the object inheritance operator (**:**) and this is how you use it. Let's assume we create two instances of the **MyClass** object, but they have different parameters, and I want to load the properties of the first defined instance, into the second one. Here's how to do it:

```
function directInheritanceExample() {  
  
    %newObj = new ScriptObject(ObjectOne) {  
  
        class = "MyClass";  
  
        numberVal = 10;  
  
        name = "Bob";  
  
    };  
  
    //Create the second instance, inheriting the properties of the first.  
  
    %newObjTwo = new ScriptObject(ObjectTwo : ObjectOne) {  
  
        class = "MyClass";  
  
        numberVal2 = 20;  
  
    };
```

```
}
```

```
echo("Inherited Properties: "@%newObjTwo.numberVal@", Name: "@%newObjTwo.name");
```

```
echo("Non-Inherited Properties: "@%newObjTwo.numberVal2);
```

```
}
```

To use the direct inheritance operation, you need to have two named object instances, for example; if I had left the **%newObj** instance without a name, we could not directly inherit the properties into **%newObjTwo**. Ordering wise on the operator sequence, you start by typing the name of the new object first, followed by the direct inheritance operator. Finally, you type the name of the object you're inheriting the properties from. This can be used on any object instance in the engine as long as the object you're inheriting from is defined **before** the object you are attempting to create is.

Introduction to Object Methods, Namespace Operator, Direct Calls to Object Methods

So far we've set up some basic class instances and worked with some basic functions that create objects and do some basic operations with them. Now, let's talk about what we'd do if we wanted to perform an operation with the direct instance of the class, or say we wanted to create multiple instances of an object, but didn't want to write functions to do the work for each individual instance, this is where the object method comes into play. So, way back in chapter 6, I taught you the basics of creating functions in the engine, we had a long discussion on that topic because it was a very important concept. Same thing here, now we're going to create object instanced functions. Let's start by showing you the syntax.

```
function className::functionName(%this, %arg1, %arg2, ..., %argN) {
```

```
    //Function Body
```

```
}
```

So as you can see here, we've got a few things that are done differently compared to your standard function. The first thing I want to talk about here however is a new operator. The **namespace operator (::)** is used in definition lines to separate a variable or class instance from the name of the variable or method. In the case of the function, you use the namespace operator to differentiate the class instance from the function name. You may also recall from a little example back in Chapter 7 with the global variable, we used the namespace operator to separate the global variable name from the name of the definition instance. Variable instances are allowed to have as many namespace separators as you want, functions are restricted to one. There's also one more application of this operator, and we'll get there in just a bit.

As for the remainder of the function definition for objects, you need to start with the class name instance, or in your case this will either be the setting of the class variable, or for datablock and singleton instances which we'll get to in just a bit, it will be the name of the block. After that, you have the function name, which follows the standard function name rules as before with one little exception. For object functions, you can have shared named functions under the circumstance that it has not been defined before by that instance, or if the className definition of the function does not match the other

function. The variables are easy to understand too, you have the standard arguments, except the very first variable **%this** which is a local variable instance that references the object calling the function. It doesn't necessarily need to be named **%this** but an object function requires an object variable to come first.

Let's look at an example with our first object class instance. We'll create a method to generate a new value for the number variable:

```
function basicObjectMethodExample() {  
  
    %newObj = new ScriptObject() {  
  
        class = "MyClass";  
  
        number = 1;  
  
    };  
  
}  
  
  
function MyClass::generateNewNumber(%this, %minimumValue, %maximumValue) {  
  
    %this.number = getRandom(%minimumValue, %maximumValue);  
  
}
```

Let's focus on the function definition here since we've seen the class already before. So in this example code, you can see we started the function line with the class instance name **MyClass**, and then followed it with the namespace operator and our function name **generateNewNumber**. The variable list includes the object reference **%this**, and then two other variables used in the number generation sequence. Hopefully you can see here how the **%this** variable is used in the object function to serve as the object reference in this case.

But this leaves us with one tiny little question here. How do we go about using this function now? We've done the hard part of writing the bit of code for our application, but how do we actually use this function in the engine now. Well, also like you've learned before, there are two ways of calling a function in the engine. There's a direct object call route and an indirect object call route. Also like before, if you use the direct route, you're allowed to return a value to the calling source, and you can't do so without a global variable for the indirect function call. Let's look at the direct call first in the same example:

```
function basicObjectMethodExample() {  
  
    %newObj = new ScriptObject() {  
  
        class = "MyClass";  
  
        number = 1;  
  
    };  
  
    echo("Generating a new number for object \"@%newObj");
```

```
%newObj.generateNewNumber(0, 1000);

echo("The number is now "@%newObj.number);

}

function MyClass::generateNewNumber(%this, %minimumValue, %maximumValue) {

    %this.number = getRandom(%minimumValue, %maximumValue);

}
```

So, remember that access operator we talked about earlier and how it's used to change variables? Well, it's also used to call functions on objects, which will clarify to you the name example a long time ago where we called **mySampleObject.delete()**; We were calling the global object method delete() on the object named mySampleObject in that case, and in the function example above, we're calling **generateNewNumber** on the %newObj instance. Hopefully, this should begin to clarify everything in terms of how things work in the engine. Object instances revolve around names and identifiers combined with the access operator, just like everything before revolved around functions and variables with a few keywords blended in the mix.

Before we move on, I want to give you a little return type example so you're clear on how that still works for object functions:

```
function basicObjectReturnMethodExample() {

    %newObj = new ScriptObject() {

        class = "MyClass";
        number = 1;
    };

    echo("Generating a new number for object "@%newObj);

    %newNumber = %newObj.generateNewNumberTwo(0, 1000);
    %newObj.number = %newNumber;
    echo("The number is now "@%newObj.number);

}

function MyClass::generateNewNumberTwo(%this, %minimumValue, %maximumValue) {

    %newNumber = getRandom(%minimumValue, %maximumValue);

    return %newNumber; //Alternatively, this could just be 'return getRandom(%minimumValue, %maximumValue);'
}
```

So not much difference in this function definition compared to the other one, you can see I just swapped the ordering to apply the value later on to the function. But, how did I know the .delete()

method existed or how could I be sure that I didn't name my function something that already existed in the engine? We'll get there in just a moment, but let's start with the indirect calls.

Indirect Object Method Calls

So, not too long ago in Chapter 7, I taught you how to use schedules and timers to perform what was called an indirect method call. You learned that indirect methods could not have a return value, and that they used schedule reference identifiers to control timer instances. You can do the same with your object classes, and in fact, there are actually quite a few advantages to using an object with a timer compared to a generic function, let's start with the syntax:

```
%schedRefID = %obj.schedule(timeInMS, 'functionName', %arg1, %arg2, ..., %argN);
```

So, you should see a few similarities here with the original scheduling method. We still have the schedule reference id variable at the front, but this time, there's no object reference identifier, only the object variable and the millisecond timer (Recall that 1000 milliseconds is 1 second). So, let's look at a few examples, and I'll also show you why object schedules are much easier to work with.

```
function timedObjectMethod_Example1() {  
  
    %newObj = new ScriptObject()  
  
    class = "MyClass";  
  
    number = 1;  
  
};  
  
echo("We currently have: @"%newObj.number);  
  
%newObj.schedule(1000, 'generateNewNumberTimer', 0, 1000);  
  
}  
  
  
function MyClass::generateNewNumberTimer(%this, %miniumumValue, %maximumValue) {  
  
    %this.number = getRandom(%miniumumValue, %maximumValue);  
  
    echo("Now we have @"%this.number);  
  
}
```

So you can see in this second example that we print out the number at the creation of the object, and one second later, generate a new number and print that to the console. So, that's a basic schedule command. Now let's spruce things up just a little bit, and I'll show you why objects are superior in schedules:

```
function timedObjectMethod_Example2() {  
  
    %newObj = new ScriptObject()  
  
    class = "MyClass";  
  
    number = 1;
```

```
};

echo("We currently have: @"%newObj.number);

%newObj.checkSchedule = %newObj.schedule(1000, 'generateNewNumberTimer2', 0, 1000);

}

function MyClass::generateNewNumberTimer2(%this, %minimumValue, %maximumValue) {

%this.minVal = %minimumValue;

%this.maxVal = %maximumValue;

%this.number = getRandom(%minimumValue, %maximumValue);

echo("Now we have @"%this.number);

%this.checkSchedule = %this.schedule(1000, 'generateNewNumberTimer2', %this.minVal, %this.maxVal);

%this.schedule(250, 'checkValue');

}

function MyClass::checkValue(%this) {

if(%this.number > (%this.maxVal / 2)) {

cancel(%this.checkSchedule);

}

}
```

So this looks like quite a bit of code, but there's not too much going on here that's new. When we enter the schedule, we apply the minimum and maximum values to the class so it can be tested and reapplied if necessary. But the really cool thing here is that we can actually store the schedule ID in a class variable, so instead of hauling hundreds of global variables around you can just fork it to the class so if the schedule needs to be canceled we can directly do it with the class instance. There's plenty to read through and learn from the above example, so spend some time with and play with the values and behaviors of the function.

The 'Dump' Command

So now you've got the basics of objects down. Before we move on, I want to share with you a useful little tool you can use when programming objects. This is called the dump command and it is used to print all of the object's variables, values, and methods to your console. To use it, all you need to do is load up the class instance in your engine via the exec command, and then open the console when ready and type the following: **className.dump()**; or **objName.dump()**; or **objID.dump()**; to perform the dump command. Depending on the above, you'll either get a generic property list or the properties and methods of the specific object in question.

A rule of thumb when you code object types you are unfamiliar with is to use the dump command first to obtain a list of named properties and methods associated with that object before you start coding a new class to make sure you're not accidentally overwriting a mechanic of the core class before you proceed.

With all of these mechanics taught, you now have all of the tools at your disposal to enter the world of programming a new game. From this point forward, we're only going to focus on new functions and properties you need to learn, but all of the basic topics are now understood. We're now moving on to intermediate and advanced topics.

Datablocks

Now that you have a basic understanding of objects in the engine, we can introduce probably the most important type of object instance in the engine, the **datablock**. A datablock is a persistent data object that stores important information on the server instance that cannot be adjusted by individual clients instances once received by the clients. They are used to define the properties and settings of important game related objects such as players, vehicles, projectiles, weapons, and others. For a list of datablock types, see the **datablock** keyword section in Chapter 6.

Rules

When you create a datablock, you need to adhere to a few rules of deployment as it relates to the rest of the engine. Some of these should already be familiar to you, but some others will be new concepts to follow.

- Make sure datablocks are uniquely named, and that they do not conflict with names of other object instances or datablocks in the engine.
- All datablock instances are required to have an assigned name to them, regardless of type.
- Datablock instances must be defined before objects using that specific block are loaded in the engine.
- Datablock instances that inherit properties from other datablocks must be loaded **after** the parent block is loaded.
- You can have a total of approximately 2048 datablock instances in your game. If you have an excess of datablock instances, consider either consolidating properties of existing blocks, or swapping some block instances to be client oriented by means of the **singleton** keyword.

As long as you follow these rules when creating datablock instances for your game, you should be perfectly fine with the syntax and confliction properties.

How to Define a Datablock

Datablock instances may either be defined in the same file as a script instance, or in a separate file from your script instances, just remember that datablocks need to be loaded before the object code that uses these blocks. In current versions of the engine, the datablocks are stored separately from the rest of the engine in the **/art/datablocks/** folder. So, let's talk about the syntax of the datablock first, and then we'll give a few examples to use for future reference:

```
datablock datablockType(datablockName [: inheritedBlock]) {  
    //Properties  
};
```

So this is the default syntax of defining a datablock, for a singleton instance you just replace the word datablock with the word singleton. The first thing after the keyword is the type of datablock you are defining. The list of block types can be found in Chapter 6. The first thing inside the parenthesis is the name of the datablock you are defining. This follows the same object/variable naming rules of the engine, as well as the rules stated above. The small portion of code that is underlined and inside the brackets [] is optional, and is used to inherit code from an existing datablock. You don't actually include the brackets [] when writing the definition, they're just there to show you the optional aspect, the below examples will clarify. Finally, inside the block we have the properties of the datablock, and then it's closed with the brace and semicolon pair.

That's all there is to defining a datablock in the engine, so now let's look at a few examples of definitions for you to work with.

```
datablock ItemData(AK47) {  
    category = "Weapon";  
    className = "Weapon";  
  
    // Basic Item properties  
    shapeFile = "art/shapes/weapons/AK47/TP_AK47.DAE";  
    mass = 1;  
    elasticity = 0.2;  
    friction = 0.6;  
    emap = true;  
  
    // Dynamic properties defined by the scripts  
    PreviewImage = "AK47.png";  
    pickUpName = "AK47";  
    description = "AK47";  
    image = AK47WeaponImage;  
    reticle = "crossHair";  
    heldWeaponName = "AK47";  
};
```

Datablock instances in the engine are treated just like regular objects. They have some common properties, or properties that are defined by default in a datablock instance (recall the dump command) and then you can define your own list of properties to work with as well.

So, let's provide you with a quick inheritance example for datablocks, and then we can move on to the next topic of discussion:

```
datablock ItemData(AK47) {  
    category = "Weapon";  
    className = "Weapon";  
  
    // Basic Item properties  
    shapeFile = "art/shapes/weapons/AK47/TP_AK47.DAE";  
    mass = 1;  
    elasticity = 0.2;  
    friction = 0.6;  
    emap = true;  
  
    // Dynamic properties defined by the scripts  
    PreviewImage = "AK47.png";  
    pickUpName = "AK47";  
    description = "AK47";  
    image = AK47WeaponImage;  
    reticle = "crossHair";  
    heldWeaponName = "AK47";  
};  
  
//Define inherited block  
datablock ItemData(AK47_VersionTwo : AK47) {  
    // Dynamic properties defined by the scripts  
    PreviewImage = "AK47_V2.png";  
    pickUpName = "AK47 2.0";  
    description = "AK47 2.0";  
    heldWeaponName = "AK47 2.0";  
};
```

So, we've got pretty much the same thing going on in our example here, but the key difference comes with the second definition, which starts by loading the properties of the AK47 datablock into our newly created AK47_VersionTwo block. Afterwards, we overwrite some of the properties of the original block in our second block by simply using the same property names.

So that's all there is to creating datablocks in the engine. Just be sure to adhere to the rules mentioned in section above to ensure you don't conflict with existing objects or trigger syntax errors in your game's scripts.

SimGroups Revisited: The TorqueScript Perspective

So, way back in Chapter 3 when we were working in the World Editor, I had you siphon your objects into special containers called SimGroups that would help you sort objects into containers to make your editing job easier. What you actually did there was create an object instance in the engine, and in extension, your level's source file, because all a mission file does, is create a large script file of objects to create and place on the designated positions on the map. Even those SimGroups you created are engine object instances.

Now, we're going to revisit this topic and introduce some new tools and functions to help you out with some more uniquely created scripts.

SimGroup & SimSet Defined

A SimGroup is basically a container instance that stores objects in a "list" type manner. This is a strictly ruled list, as objects that belong to one SimGroup cannot be in any other group instances in the engine. It also enforces a single-group-membership rule, which means if you attempt to add an object to a SimGroup and it already belongs to another group, it will be removed from its current group instance before being added to the new group. Finally, if the SimGroup is deleted, all objects that are contained in the SimGroup will also be deleted.

But, that's not the only type of grouping instance in the engine. You could also define a **SimSet** instance. A SimSet follows pretty much the exact same functioning of the SimGroup with one key change, the SimSet has no membership rules for objects, so objects that belong to one SimSet, might also be located in another SimSet for example, and when the Set is deleted, the objects are not.

Creating a SimGroup & SimSet

Creating a SimGroup in the engine is surprisingly simple. We basically do some of the stuff we've done before in this chapter:

```
function simGroupExample_One() {  
    %group = new SimGroup() {};  
    return %group;  
}
```

This is probably the easiest of object creation forms you've seen to this point in the example code, one line to define the object, and another to return the instance to the calling function to be used in the code. To do the same for a SimSet, you simple change one thing:

```
function simSetExample_One() {  
    %group = new SimSet() {};  
    return %group;  
}
```

Most times in the engine, people decide to omit the variable definition in place of a named object so it can be used whenever it's needed. This works for things like a list of active player objects, or active mission objects, or anything in your level that needs to be grouped and worked with on the fly. Since these two object types have the exact same architecture in terms of function names and operations with the only exclusion being its rule set, we'll only discuss the **SimSet**. Just know that all of these code pieces can also be applied to a SimGroup as well but the object rules for single-group-membership will be enforced by doing so.

Adding and Removing Objects from Groups

Working with groups and sets in the engine is also very easy. They have some engine coded functions to perform individual tasks on the objects in the list. We'll start by showing you how to add objects to the list.

```
$TestObj[0] = isObject($TestObj[0]) ? $TestObj[0] : new ScriptObject();

$TestObj[1] = isObject($TestObj[1]) ? $TestObj[1] : new ScriptObject();

$TestObj[2] = isObject($TestObj[2]) ? $TestObj[2] : new ScriptObject();

$TestObj[3] = isObject($TestObj[3]) ? $TestObj[3] : new ScriptObject();

function simSetExample_Two() {

    %group = new SimSet() { };

    for(%i = 0; $TestObj[%i] != ""; %i++) {

        %group.add($TestObj[%i]);

    }

}
```

So, I decided to give something new to throw at you this time. I threw this more "advanced" code sample at you to try to get the mind working on your end. Let's look it over first. The first four lines should be very self-explanatory in nature. They're conditionals to test if an object already exists, if it does, it returns the variable that stores the object, and if not, it creates the object instance. The more tricky code piece comes with that for loop. You're used to seeing the format of the first and final parameter, but what is going on with that second parameter? Actually, if you go back to chapter 6 and re-read the for loop syntax, all it accepts are "expressions", and that's what I've given you there. A conditional expression to test if the object in the current iteration exists. The actual code to add the object however is bolded. The .add command of the set list will add the specified object (name, id, variable) to the set.

Remember that for SimGroups it will pull the object out of any existing group instance to add it to another group. But let's say you didn't want that behavior to happen and to simply ignore the object being added if it's already in another group. Well, the engine has a tool to help with that:

```
$TestObj[0] = isObject($TestObj[0]) ? $TestObj[0] : new ScriptObject();
$TestObj[1] = isObject($TestObj[1]) ? $TestObj[1] : new ScriptObject();
$TestObj[2] = isObject($TestObj[2]) ? $TestObj[2] : new ScriptObject();
$TestObj[3] = isObject($TestObj[3]) ? $TestObj[3] : new ScriptObject();

function simGroupExample_Two() {
    %group = new SimGroup() { };
    for(%i = 0; $TestObj[%i] != ""; %i++) {
        if(%group.acceptsAsChild($TestObj[%i])) {
            %group.add($TestObj[%i]);
        }
        else {
            error("Cannot add object \"@$TestObj[%i]@\" on index \"@%i@\", it belongs to another group.");
        }
    }
}
```

The engine function for groups named **acceptsAsChild** will test if an object already belongs to a conflicting group or set instance and return false if it cannot be “safely” added to the group without being removed from another. For SimSet instances, this will always be true since there are no membership rules, but for SimGroups this will depend on if the object belongs to another group instance or not.

Now, let’s say we made a mistake, and object three wasn’t supposed to be added to the group. Well, this next code sample will show you how to “safely” remove an object from the list:

```
$TestObj[0] = isObject($TestObj[0]) ? $TestObj[0] : new ScriptObject();
$TestObj[1] = isObject($TestObj[1]) ? $TestObj[1] : new ScriptObject();
$TestObj[2] = isObject($TestObj[2]) ? $TestObj[2] : new ScriptObject();
$TestObj[3] = isObject($TestObj[3]) ? $TestObj[3] : new ScriptObject();

function simSetExample_Three() {
    %group = new SimSet() { };
    for(%i = 0; $TestObj[%i] != ""; %i++) {
        %group.add($TestObj[%i]);
    }
    //Remove the third object.
```

```
if(%group.isMember($TestObj[3])) {  
    %group.remove($TestObj[3]);  
}  
}
```

There are two new functions here. The **isMember** function will check a group to see if the object instance in question belongs to the group or set that is being requested, and the **remove** command is used to remove the object in question from the group instance.

The last command I want to show you here is a command to remove all of the objects from the current group. That is the clear command:

```
$TestObj[0] = isObject($TestObj[0]) ? $TestObj[0] : new ScriptObject();  
$TestObj[1] = isObject($TestObj[1]) ? $TestObj[1] : new ScriptObject();  
$TestObj[2] = isObject($TestObj[2]) ? $TestObj[2] : new ScriptObject();  
$TestObj[3] = isObject($TestObj[3]) ? $TestObj[3] : new ScriptObject();  
  
function simSetExample_Four() {  
    %group = new SimSet() {};  
    for(%i = 0; $TestObj[%i] != ""; %i++) {  
        %group.add($TestObj[%i]);  
    }  
    echo("Cleaning the set.");  
    %group.clear();  
}
```

So there are the basics of adding and removing objects from the groups and sets. But what exactly do we need to do this for? Let's dig a little deeper into the functioning.

Listing Functions

The actual power to use when it comes to using a group or set instance is the ability to fetch objects from the individual groups and sets. You can place objects in the list, and then either use a function to pull one out of the set, or even loop through the objects in the set to perform actions on them. Before we can go that far though, we actually need to show you the new functions and how to use them.

The two important group functions here we need are **getObject** and **getCount** which are used in conjunction with for loops to get the objects in the list. For example:

```
$TestObj[0] = isObject($TestObj[0]) ? $TestObj[0] : new ScriptObject();
$TestObj[1] = isObject($TestObj[1]) ? $TestObj[1] : new ScriptObject();
$TestObj[2] = isObject($TestObj[2]) ? $TestObj[2] : new ScriptObject();
$TestObj[3] = isObject($TestObj[3]) ? $TestObj[3] : new ScriptObject();

function simSetExample_Five() {
    %group = new SimSet() { };
    for(%i = 0; $TestObj[%i] != ""; %i++) {
        %group.add($TestObj[%i]);
    }
    //Get the objects...
    for(%i = 0; %i < %group.getCount(); %i++) {
        %obj = %group.getObject(%i);
        echo("Object at Index "@%i@": "@%obj");
        //Do work here...
    }
}
```

Use **getCount** to establish how many objects are in your SimSet instance, and then **getObject** with the **index** parameter (remember, it's a list, which will be treated like an array, indices start at 0) to fetch the object instance in question. Now, let's say we have a function that we want to use to pull a single object out of the list to do something to. With the **getRandom** function, you can do just that:

```
$TestObj[0] = isObject($TestObj[0]) ? $TestObj[0] : new ScriptObject();
$TestObj[1] = isObject($TestObj[1]) ? $TestObj[1] : new ScriptObject();
$TestObj[2] = isObject($TestObj[2]) ? $TestObj[2] : new ScriptObject();
$TestObj[3] = isObject($TestObj[3]) ? $TestObj[3] : new ScriptObject();

function simSetExample_Six() {
    %group = new SimSet() { };
    for(%i = 0; $TestObj[%i] != ""; %i++) {
        %group.add($TestObj[%i]);
    }
    //fetch an object.
    %obj = %group.getRandom();
```

```
if(%obj != -1) {  
    %index = %group.getObjectIndex(%obj);  
    echo("Random has pulled the object at index "@%index@" from the group: "@%obj");  
}  
  
else {  
    error("No objects in the group.");  
}  
}
```

So there's a few functions being shown here. The getRandom function will either return the object id of the object it pulled from the list or a -1 if there are no objects in the list. And then, I added the other function getObjectIndex to the code to show you how to pull the index of the object we just got from the group. You could in a sense, combine these two to create a "random deck" of objects, where the object that has just been called up by getRandom / getObjectIndex can be removed from the group.

And there you have it, that's how you use SimGroups and SimSets in engine code. Sometimes you might think that this segment would be pointless, but the day will come when these two special class architectures will come to be extremely useful to you, and it's nice to know how to actually use it before second guessing yourself. So with that out of the way, let's move on!

Packages, the Parent Keyword, and Function Overloading

The last part of our objects chapter will cover something not entirely related to objects, but relevant enough that it should be taught here. This is also the final "base" topic for the scripting language that you should be aware of for the engine, and that is the concept of a package.

Introduction to Packages

The notion you have used so far in the engine has been to employ a function to perform a specific set of actions and then to move on to the next job. And while this aspect is true for most of the code, there are some times when this will not be the case. Let's take one special object for example, the **Game** object. This is a server object that stores parameters and settings for the game mode being ran on the server, and by default this would be a Deathmatch game. Now, let's assume you code up a Team Deathmatch game, the engine already has a default list of functions and they are pre-programmed to deathmatch and you don't want to overwrite them completely? So how do we get around this problem?

This is where the concept of a **package** comes into play. A Package is a segment of code that can be activated and deactivated to safely overwrite existing code blocks without overwriting the existing bits of code. Let's look at the syntax of a package definition, then some examples:

```
package packageName {  
    //Code...  
};
```

So continuing to follow the object type notion, the package definition is closed with a brace-semicolon pair. There are a few rules about packages that you need to know, so let's quickly discuss those.

- Global variables cannot be defined or set within the body of a package, but they can be used in conditional statements.
- Two package instances cannot share the same name.
- Datablocks cannot be defined in the body of packages.
- Packages have stacked activation, so packages that overload the same function will be deactivated all at once in a line (more below).

So there isn't too much you need to be aware of when it comes to coding packages, but sometimes the global variable definition rule will trip up a few people, you'll just need to work around that with some function trickery, or by means of moving your variables to object parameters. So, with that in mind, let's have a peek at some examples:

```
function helloWorld(%moreText) {  
    echo("Hello World! @"%moreText);  
}  
  
function packageDemo1(%moreText) {  
    helloWorld(%moreText);  
    if(!isActivePackage(hwOverload)) {  
        activatePackage(hwOverload);  
    }  
    helloWorld(%moreText);  
    deactivatePackage(hwOverload);  
}  
  
package hwOverload {  
    function helloWorld(%moreText) {  
        echo("Hello World, from inside the package, also: @"%moreText);  
    }  
};
```

So we've gone back to our old hello world example to work with here. First of all, the original definition still stands at the top so we've got Hello World! and then some additional input text from the function definition. Now, let's skip to the bottom to the package definition. You can see how to actually use it here, enclose the code you wish to adjust in the package bracket. This is the only time in the engine, where it is actually "safe" to use the same function names. Finally, let's look at the packageDemo function. I had two instances of the print function run to show you how this works. There are three functions you need to be aware of for packages in the engine. **activatePackage** will enable to code contained in the specified package instance, **deactivatePackage** will disable the selected code and revert back to the original, and the conditional boolean function **isActivePackage** will return true or false depending on the active status of the package in question.

Packages however, are not just limited to basic functions, but they can be used to overload object functions and properties as well!

```
function initMathDemoObject() {  
    %mObj = new ScriptObject()  
    class = "MathObject";  
}  
  
%mObj.num1 = getRandom(1, 1000);  
%mObj.num2 = getRandom(1, 1000);  
  
return %mObj;  
}  
  
function MathObject::performMathematics(%this) {  
    return %this.num3 $= "" ? (%this.num1 + %this.num2) : (%this.num1 + %this.num2 + %this.num3);  
}  
  
function packageDemo2() {  
    %mObj = initMathDemoObject();  
    echo("Math Demo 1: @"%mObj.performMathematics());  
    if(!isActivePackage(mathOverload)) {  
        activatePackage(mathOverload);  
    }  
    echo("Math Demo 2: @"%mObj.performMathematics());  
    %mObj.delete();  
    %mObj = -1; //Set to no object, just to be safe.  
    //Now, generate a second object from inside the package.
```

```
%mObj = initMathDemoObject();

echo("Math Demo 2 [2]: @"%mObj.performMathematics());

deactivatePackage(mathOverload);

echo("Math Demo 1 [2]: @"%mObj.performMathematics());

%mObj.delete();

}

package mathOverload {

    function initMathDemoObject() {

        %mObj = new ScriptObject() {

            class = "MathObject";

        };

        %mObj.num1 = getRandom(1, 1000);

        %mObj.num2 = getRandom(1, 1000);

        %mObj.num3 = getRandom(1, 1000);

        return %mObj;

    }

    function MathObject::performMathematics(%this) {

        return %this.num3 $= "" ? (%this.num1 * %this.num2) : (%this.num1 * %this.num2 * %this.num3);

    }

};
```

So, there's plenty more going on in this example. First, we have an object creation script and an associated function to add the two generated numbers together, but the conditional allows for a third number to be added as well. For the actual third number, you need to go into the package definition, where the object generated in there has three numbers and the function performs multiplication instead of addition. The package demo function first generates an original object (2 numbers) and performs addition. Then the package is enabled and we call the multiply function. After we do that, we delete the original object and generate a new object in its place. The second time around we call multiply first (since the package is still on), then we disable the package and do the add function at the end.

While this second example has a lot going on for it, there really isn't too much that should be complex or new about it since we've talked about all of these topics beforehand. The concept of using a package to overload functions should come very easily to you with this in mind. Now, let's take a new

question to ask. What happens if there are multiple packages activated over the same function, and we activate and deactivate them. Let's look at an example:

```
function baseFunction() {  
    echo("base.");  
}  
  
function packageDemo3() {  
    baseFunction();  
    activatePackage(baseOverload1);  
    baseFunction();  
    activatePackage(baseOverload2);  
    baseFunction();  
    deactivatePackage(baseOverload1); //<- What happens here?  
    baseFunction();  
}  
  
package baseOverload1 {  
    function baseFunction() {  
        echo("package 1.");  
    }  
};  
  
package baseOverload2 {  
    function baseFunction() {  
        echo("package 2.");  
    }  
};
```

So ask the question, what happens when you hit the deactivate line. Packages are loaded in a "stack" style loading system. So basically imagine a tower of plates, and adding a package to this list of the currently active function places a plate on top of the function. If you pull the bottom plate out from the tower, the others will fall. Same notion here, if you disable the "first" package activated, all packages that overload the same function will also deactivate for that function, which is why the function prints "base" when you deactivate the first package, but not the second one.

Now, let's ask another interesting question. What do we do in the case that we write a base function with some functions and settings that we'd like to keep in our overloaded function inside the package. Well, you could just copy the contents of the original function over, but that could be a lot of work, especially for large functions. But thankfully, the engine has a little tool we can use to get around this.

The Parent Keyword & Overloading Functions

With the whole notion of multiple inheritance discussed a little while ago you need to think about the prospects of overloading functions now. For the entire time since Chapter 6, you've been told under no circumstances to use the same name for functions, and then I went ahead in the prior section that it's safe to do it, but only in a package. Well, I'm going to break that little notion, one more time.

In reality, it's perfectly safe in the engine to use the same name for functions at any time, but what happens is as soon as a function is loaded with the same name, it "overwrites" the existing function completely. The reason you need to be careful with this is even internal engine functions can be overwritten by your scripts. For example:

```
function echo(%message) {  
    error(%message);  
}
```

Doing this in a script will overload the echo function and replace all of the echo statements with error statements (White/Blue Text -> Red Text). So, the actual statement should read, it's safe to use the same name for functions, but you should never overload engine functions or functions that have a core mechanic for the application instance.

Now, let's look at Object Functions really quick here. All objects in the engine have a standard set of calls they use in the engine when they are created, updated, and deleted. You could overwrite these functions to add custom behaviors to the function, but then you'd lose the original mechanics of the function, and that could be bad. Thankfully, you can keep the original function intact with a new keyword and sequence. This keyword is the **parent** keyword. The parent keyword is a direct access to the original object's class instance. So for example, a commonly edited function is the onAdd() function for objects, here's how you can safely keep the original functioning.

```
function initMathDemoObject() {  
    %mObj = new ScriptObject()  
    {  
        class = "MathObject";  
    };  
    %mObj.num1 = getRandom(1, 1000);  
    %mObj.num2 = getRandom(1, 1000);  
    return %mObj;  
}
```

```
}
```



```
function MathObject::onAdd(%this) {
    parent::onAdd(%this);
    echo("Adding a MathObject. Num1: @"%this.num1@", Num2: @"%this.num2");
}
```

So, you've seen the original bit already; now let's have a peek at the `onAdd` function. When an object is added to the game instance, it calls a generic `onAdd` method. Some of these have parameters and others do not (see `obj.dump()` for differentiation). What you can do is overload the instance for your object, but keep the original mechanic intact by calling the parent's `onAdd` function. When you use the `parent::` notion however, you need to keep the original parameters, like you'd be directly calling a non-object function. The `parent::` notion is just a shortcut to the original class call, in our case it'd be like writing `ScriptObject::` so be sure to include the full parameter list in the function call.

The `parent` keyword also has one more little trick to it. When you're inside a package definition, you can use the `parent` keyword to access the original function outside of the package, and here's how:

```
function initMathDemoObject() {
    %mObj = new ScriptObject() {
        class = "MathObject";
    };
    %mObj.num1 = getRandom(1, 1000);
    %mObj.num2 = getRandom(1, 1000);
    return %mObj;
}

function MathObject::performMathematics(%this) {
    return %this.num3 $= "" ? (%this.num1 + %this.num2) : (%this.num1 + %this.num2 + %this.num3);
}

function packageDemo2() {
    %mObj = initMathDemoObject();
    echo("Math Demo 1: @"%mObj.performMathematics());
    if(!isActivePackage(mathOverload)) {
        activatePackage(mathOverload);
    }
}
```

```
echo("Math Demo 2: @"%mObj.performMathematics());  
  
%mObj.delete();  
  
%mObj = -1; //Set to no object, just to be safe.  
  
//Now, generate a second object from inside the package.  
  
%mObj = initMathDemoObject();  
  
echo("Math Demo 2 [2]: @"%mObj.performMathematics());  
  
deactivatePackage(mathOverload);  
  
echo("Math Demo 1 [2]: @"%mObj.performMathematics());  
  
%mObj.delete();  
}  
  
  
package mathOverload {  
  
    function initMathDemoObject() {  
  
        %mObj = parent::initMathDemoObject();  
  
        %mObj.num3 = getRandom(1, 1000);  
  
        return %mObj;  
    }  
  
  
    function MathObject::performMathematics(%this) {  
  
        %addResult = parent::performMathematics(%this);  
  
        %multResult = %this.num3 $= "" ? (%this.num1 * %this.num2) : (%this.num1 * %this.num2 * %this.num3);  
  
        return %multResult;  
    }  
};
```

So take in this example for a moment. All you need to look at is the functions inside the package here since the others have remained untouched. For the init function, we call the original function, which generates num1 and num2 and returns the object generated. We capture that in a variable, and then tag on num3 to the function and return that instead. As for the second function, the syntax is there to prove that both object and non-object functions use the same style for parent:: notations, while we don't actually use the %addResult variable, it's there to show you how you would.

So with all of that in mind you should now be off to the races in terms of using the package and parent keywords with your functions, and you now have a complete understanding of inner workings of function definitions in the engine.

Closing Notes

So, we've covered some pretty serious groundwork in this chapter, introducing one of the most important topics in the engine (Objects & Classes) and teaching you how they work, and how to set up their parameters and function instances. We also covered some final engine core mechanics with packages and the parent relation with functions, and I went back and re-covered a Chapter 3 topic, but gave it the TorqueScript perspective instead.

This chapter should leave you with everything you need to actually comb through the engine's scripting documentation with a full understanding of how mechanics and features work. I actually advise you to take a peek through there to add some new functions to your understanding from time to time; you never know when you might find some tools that prove to have utmost importance to you.

Now, we're going to shift topics to something very important for game development in general. Your eyes may catch that twelve letter word (Starting with a 'M') and glare at it with disgust, but unfortunately for you it's time to talk about Mathematics for game development! Horary!

Chapter 9: Mathematics for Video Games: The Basics

Welcome Back!

Ah yes, mathematics! Either the most hated subject, or most loved subject back in school. Surprised to see it here? Well, don't be. Math plays an extremely pivotal role in computers because almost everything in computer programming has something to do with numbers, and how numbers change over time, ergo, Math.

Needless to say however, your study of mathematics in computers will generally come down to a limited number of subjects, so you don't have too much to worry about. But, once you start going from general computer programming into video game development, that's when things start to become a little more complicated, and that's what I'm here for!

So, why did I insert this chapter here when we're just starting to get into the fun stuff of programming? I did this for two reasons. The first reason is because the next chapter is going back into GUI's, and then we're going to go into actual world events in Chapter 11. The problem here is that both of these subjects of programming require mathematics to perform certain operations, both in 2D-Space and more importantly 3D-Space. The other reason is because some of these operations will be coming back later on when we dive into C++, and it helps to have any form of head start you can get when going into a higher level language like C++.

For a quick review, remember that the operational order of computers is the same as it is in real world mathematics: Parenthesis, Exponents, Multiplication, Division, Addition, and Subtraction. Or is it?

Modulation

When I throw this big sub-chapter header at you, you're probably sitting there wondering what I'm babbling off on now, when I'm actually presenting you with something brank spanking new to think over. This is a special mathematical operation called modulation, or in computers, applying the modular operator to a statement. It works by taking the division between two numbers requested, but then returns the remainder of the operation. Here's where it gets good:

```
%modResult = 100 % 3; // = 33 R 1, hence 100 % 3 = 1.
```

Alright, so you're probably screaming "Syntax Error! Syntax Error!" at me right about now, but I'm not even kidding you here. The % symbol is actually the modulation operator **and** the local variable definition operator. Torque is smart enough to differentiate between the two.

The reason I flipped this little thing at you right away is because of the order of operations I mentioned above. The question is: Where does modulation go in the list? The answer to this question is right after Division. While it may not appear to have a major role right away, there are many applications in programming to determining the numerical integer remainder of the division versus getting a decimal result, some are a little beyond the scope of this guide, so we won't go there.

Working with Exponents in Computers

Anyways, with that little topic out of the way we can start to work our way into the engine's mathematics. First off, we need to talk about exponential numbers in the engine, and programming in general due to bitwise operators taking over the standard power symbol. There are a few important functions you need to learn here to accomplish powers in TorqueScript. We'll cover the C++ version of this when we get to Libraries later on.

Raising to a Power, Taking a Root

To raise a number to a power in Torque you use the **mPow** function. This is a very easy function to use since it accepts two floating point numbers (numbers with a decimal point, 32-bit) as parameters, here's a quick sample:

```
function powerExample(%x, %y) {  
    echo("Raising \"%@x@\" to the power of \"%@y@\"");  
    return mPow(%x, %y);  
}
```

So as you can see here, it's easy enough to understand. The `%x` variable is the number you wish to apply the power to, and `%y` is the exponent you are raising it to. If you've had any kind of algebra course, you also know that by using the 1/power relation you can take a numerical root of a number, for example `mPow(2, (1/3))` would give you the cubic root of 2. For a square root however, there's a little shortcut you can use:

```
function squareRootExample(%x) {  
    echo("Taking the square root of \"%@x@\"");  
    return mSqrt(%x);  
}
```

This shortcut comes in handy when working with quadratic style equations or formulae that require the square root of the result.

Logarithms

Logarithms are a topic that is usually covered in a later algebra course or a pre-calculus course but it plays an important role in mathematical applications. By definition a logarithm of a number is the exponent of another value of the logarithm, called a base, must be raised to in order to obtain the value. In standard mathematical applications these logarithms use a base-10 scale in which the base is defined as a common value (10). For most other applications, such as science and for our game-engine applications, we will use Euler's Number ($e \approx 2.71828$) as the base to establish what is called a **natural logarithm** or in mathematical terms **ln(x)**.

To actually perform this operation in the engine, you use the **mLog** function and provide a value to the function to perform this operation on the function.

Change of Base Formula

However, not all of your applications will seek to use this natural logarithm as the base, for example you might actually need the standard base-10 formula instance. But since the engine only provides you with a natural logarithm function, you need to use what is called the change of base formula to actually accomplish this.

The change of base formula is defined as: $\log_b x = \frac{\ln(x)}{\ln(b)}$, or an example you can understand:

```
function logWithBase(%value, %newBase) {  
    return (mLog(%value) / mLog(%newBase));  
}
```

While most applications in mathematics for your game will likely not use this function, you should be aware of it in the event the need for it rises.

Powers of 2

The engine also provides you with a function to determine if the specified numerical value is an exact power of 2 (recall our binary table back in Chapter 6). If the value of x is 2^n when sent to the **mIsPow2** function, it will return true, otherwise it will return false. This can be useful when performing binary operations on certain numerical parameters to ensure you have valid functioning.

Basic Mathematical Functions

Now, let's cover a few more basic mathematical operations that you might find useful from time to time in your applications. These first few are little shortcut tools to help you with numerical rounding when you need to have an integer answer, and then we've got a few more useful mathematical tools you might find useful in your applications.

Floor

The mathematical operation for rounding a number down to the nearest integer no matter the decimal value is called flooring the value. Common rounding rules tells you that any number whose decimal value is at or above .5 should be rounded up, and all others rounded down. This does not apply to the floor command. Any value, even x.999999 will be rounded down when using the floor option. To do this in the engine, you use the **mFloor** function and provide it with a single numerical parameter of the value you wish to round down.

Ceil

The opposite of the floor operation is the ceil operation, which will round any number, even x.000001 up to the next integer value. To use this operation in TorqueScript, you use the **mCeil** function with a single numerical parameter to receive an integer result back.

Rounding

However, if neither of these two functions are what you need and you want to perform a standard numerical rounding of the value where values at or above x.5 round up, and others round down, you can use the **mRound** function.

Absolute Values

The absolute value of a number is a guaranteed **positive** value of the input number. Basically put, if you send a positive number to this function, nothing will happen, but if you send a negative number to this function, it will multiply it by -1 to flip the sign to positive. To do this in the engine, you use the **mAbs** function and provide it with a numerical parameter.

Minima & Maxima Functions

Another common mathematical task is to find the larger and smaller of two different values you have. These are referred to as the minima (smallest) and maxima (largest) of the two values. Torque provides you with two functions to obtain these values. For the minima of two values, use the **getMin** function and provide it with two numerical parameters. For the maxima of two values, you use the **getMax** function and provide it with two numerical parameters.

Clamping

Something that isn't done a lot in mathematics is a task called **clamping** a value, which is a fancy term for saying: constrain a value between two numerical points. The engine has two functions for doing this. To clamp a number between two arbitrarily defined points, you use the **mClamp** function. To clamp the value between 0.0 and 1.0, you use the **mSaturate** function. Here's a short example of mClamp and mSaturate in action:

```
function clampExample(%x, %min, %max) {  
  
    echo("Clamping \"%@x@\" between \"%@min@\" and \"%@max@\"");  
  
    %clamp = mClamp(%x, %min, %max);  
  
    %saturate = mSaturate(%x);  
  
    echo("The clamped value is: \"%@clamp@\"\nThe saturated value is \"%@saturate@\"");  
}
```

Linear Interpolation

Finally, we come to a mathematical operation called interpolation, which is an operation of position and time. The engine provides you with a linear function to interpolate between two points given a normalized time parameter (0 – 1). To use this operation, you use the **mLerp** function. The first two parameters are the starting and ending numerical points of the linear operation, and the third parameter is the normalized time value (between 0.0 and 1.0).

Trigonometry: Back with a vengeance!

For many people, Trigonometry is a mathematical bane of existence having to deal with plenty of tedious relations and operations with trigonometric functions and complex formulae that need to be

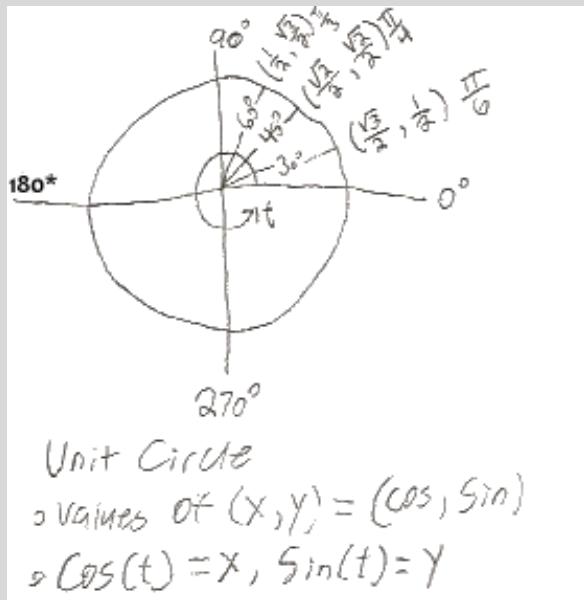
simplified until simple values like 0 or 1 come out. Fortunately for you, this is not the application of Trigonometry as it relates to video game math, however there is still a major important role this mathematical functioning plays, and that is the deterministic angle calculations. Since not all developers have a full background in what's needed here, we'll go over the basics of Trigonometry that you should be aware of when working with this engine.

Pi

This should be a very familiar concept no matter what level of mathematics you have completed. The value of Pi (π) which rounds to 3.14159, is a numerical relation of a circle's circumference to its diameter, which is constant, regardless of the size of the circle. Pi is an infinitely expanding decimal number (irrational & transcendental). The amount of approximation needed for each individual application is completely dependent on the developer. To obtain a value of Pi in the engine, you use the **mPi()** function. There is also the **m2Pi()** function to obtain the value of two times pi. For most applications of trigonometry, this is used to find radian points on the unit circle and to directly convert values of radians and degrees.

The Unit Circle

One of the big important tools of Trigonometry is the Unit Circle, or basically a circle of points where each point extends a line outward in the shape of a circle that has a length of 1. Here's a little drawing of the Unit Circle:



So, we know from here that each point on this circle extends outward to a length of 1. The Pythagorean Theorem states: $a^2 + b^2 = c^2$. So, let's expand on this just a little bit. We'll assume that a is x and b is y. Since the length is constantly 1 the c^2 term becomes 1. Now there's a trigonometric identity out there that states the following: $\sin^2 t + \cos^2 t = 1$. So we assume that each point around the circle (angular point) is a value of t, and then define $\sin(t) = y$ and $\cos(t) = x$ to complete the relation $x^2 + y^2 = 1$. From this, we can work around the circle at each point of t to define the values of x and y

for each coordinate. I have provided in the diagram above enough points to complete the full circle if you wanted to complete it. The unit circle is a very helpful reference to quickly determine values for “common” angles.

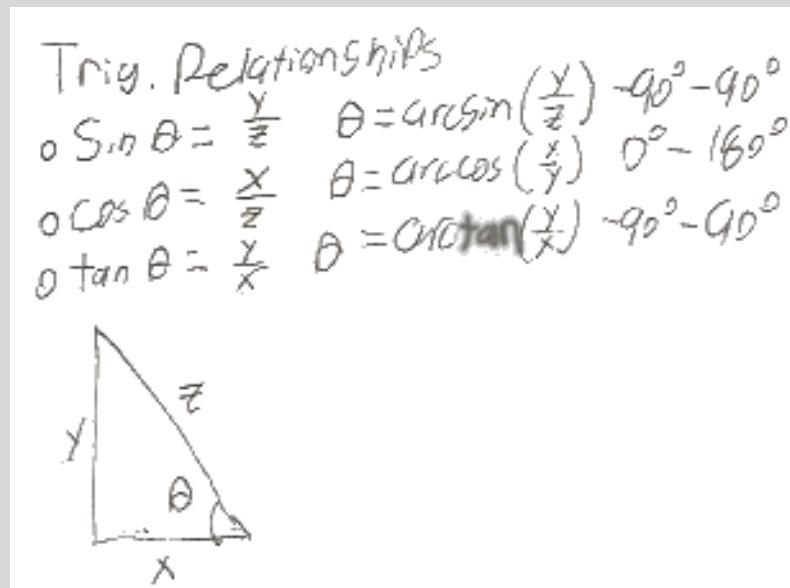
Radians & Degrees

In terms of what we've been seeing so far, you've been working directly with numerical values such as 3.1415 (Pi) and integer constants for variables. All of these values are classified as **radians**, which is a standard unit of measure in angular mathematics. By default, the engine treats all numerical values as radians, which means when you use trigonometric functions in the engine, that you need to be using a radian number to obtain answers. You can however, switch between the measures of **radians** and **degrees** (Another form of angular measurement) by means of two single argument functions:

mDegToRad and **mRadToDeg** to convert your numbers to the proper measure you need. The rule of thumb to follow here is when you're working with complex angle measurements, you might want to be in degrees to solve calculations, but once you work with world space coordinates, you need to be in radians.

Trigonometric Functions, Relations

The actual work of trigonometry however comes from a standard set of functions. The names of these mathematical functions should be familiar: sine, cosine, and tangent. From a mathematical standpoint they define basic relations between the angle and side lengths on a triangle. When working with vectors later on, you'll see how some of these relations are established with the points of these vectors. For some common remembrance rules here, sine is the relationship between the side opposite and the hypotenuse side of the triangle. Cosine relates the side adjacent to the hypotenuse side, and Tangent relates the side opposite to the side adjacent. Here's a two part diagram for the following:



The second part of this figure relates to the inverse trigonometric functions which are prepended with the word “arc” before the same names. These functions are used to back the angle out of the trigonometric function between the specified bounds of the function parameters. When actually

solving for these functions, you should have a well-founded understanding of the unit circle to understand where the values will actually come out as, and if you should add/subtract angular values to obtain the correct answer for the given range of the function. There is one stickler in the bunch of these functions, and that is the arctangent function, which uses a special definition which is common to computer programming called atan2, which is a specialized version of arctangent that has six solution criterions:

Arctangent Relation

$\text{mAtan}(y, x) =$

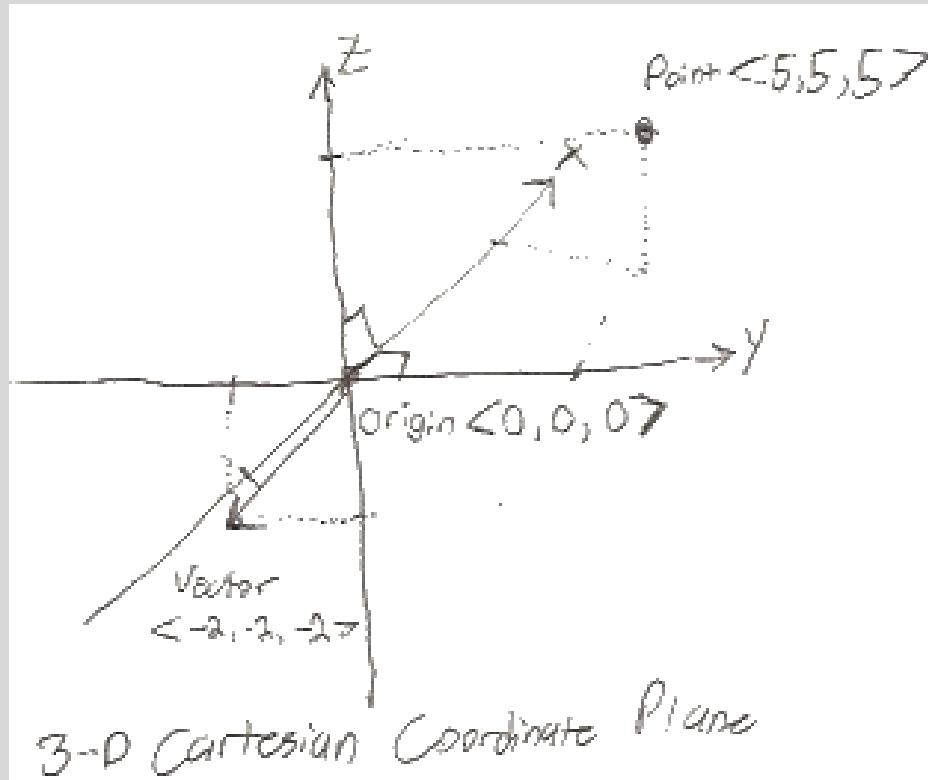
$$\begin{cases} \arctan \frac{y}{x} & x > 0 \\ \arctan \frac{y}{x} + \pi & y \geq 0, x < 0 \\ \arctan \frac{y}{x} - \pi & y < 0, x < 0 \\ \frac{\pi}{2} & y \geq 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{NaN} & y = 0, x = 0 \end{cases}$$

In TorqueScript, the actual functions accept a single parameter (with the exception of the above) and are as follows: **mSin**, **mCos**, **mTan**, **mASin**, **mACos**. Remember that values are expected in radian format when sent, and you'll receive a radian answer. For **mATan** you need to send two values, the first relating to the rise of the angle (y), the second relating to the run of the angle (x), and the answer will come out based on the following input above. One last thing to note is that not all mathematical operations refer to the inverse functions as $\arcsin(x)$. Sometimes you'll see notation like: $\sin^{-1} x$ to indicate arcsine, and so on for the others.

This is really all you need to know about Trigonometry as an overall subject for the engine, just know the functions, what they do, and when you're supposed to use them and you should be good to go in terms of calculating values.

Introduction to Vectors, Positions

The next topic in our chapter covers a very important segment of discussion in the engine, relating to the topics of positions and vectors. This is where I'm going to teach a bulk of the mathematical topics to you as it relates to game development, as most of your work will be dealing with the change of position with time, which uses vector quantities to solve. In mathematical terms, a **vector** is any quantity with a magnitude and a directional component, and it takes the standardized definition as follows: $\vec{A} = a\hat{i} + b\hat{j} + c\hat{k}$. The components i, j, and k, are referring to what is called a **unit vector** which is a vector with a length of 1. The notation itself refers to the Cartesian Coordinate Plane, where i refers to x, j to y, and k to z. Here's a diagram to help you out:



Vectors can be defined in the standard notation above, or in component form, as shown by the diagram where in our case we have a vector quantity where $\vec{A} = < -2\hat{i}, -2\hat{j}, -2\hat{k} >$. When you work with vectors and positions in Torque, they are converted to string notation. For example, our vector there would be translated to: $\vec{A} = "-2 -2 -2"$. Notice the spaces between the individual components there, and you also don't need to include the unit vector notation. While we're still here on this topic let's also talk about world-space coordinates, or **positions**. Basically a position in the engine takes the exact same form as a vector, where the coordinates of X, Y, and Z are separated in an individual string instance with spaces. To perform operations that work with moving positions around by means of standard mathematical operations, you can either use the string tools (**Chapter 7 – getWord**) or use some of these new functions I will teach you to accomplish this. To construct a vector given three points (x, y, z) you could use a function like this:

```
function vectorConstruct(%x, %y, %z) {
    if(%x == "" || (%y == "" && %z == ""))
        error("Cannot construct vector, need at least 3 coordinates.");
    return "0 0 0";
}
return %x @ " " @ (%y == "" ? %z : (%y @ (%z == "" ? "" : "%z")));
}
```

Vector Mathematics

This next part of the guide will walk you through some of the common applications of vectors and operations you can perform on vectors. We'll also show you how to perform these operations in TorqueScript. For each of the below examples, assume we have two individual vectors with the following quantities: $\vec{A} = A_x\hat{i} + A_y\hat{j} + A_z\hat{k}$ and $\vec{B} = B_x\hat{i} + B_y\hat{j} + B_z\hat{k}$. For common vector practice, know that the **magnitude** or the length of a vector is defined as: $\|\vec{A}\| = \sqrt{(A_x\hat{i})^2 + (A_y\hat{j})^2 + (A_z\hat{k})^2}$.

You can find this value in the engine with the **vectorLen** function. Here's an example:

```
function vectorLengthExample() {  
    %vector = "5 5 5";  
    %len = vectorLen(%vector);  
    echo("The magnitude of \"%@vector@\" is \"%@len@\"");  
}
```

And to find the distance between two vectors (also good for positions) you use the **vectorDist** function:

```
function vectorDistExample() {  
    %vector[0] = "5 5 5";  
    %vector[1] = "10 20 30";  
    %dist = vectorDist(%vector[0], %vector[1]);  
    echo("The distance between these vectors is \"%@dist@\"");  
}
```

Now that we've got these basics out of the way, let's talk about some common vector mathematical functions.

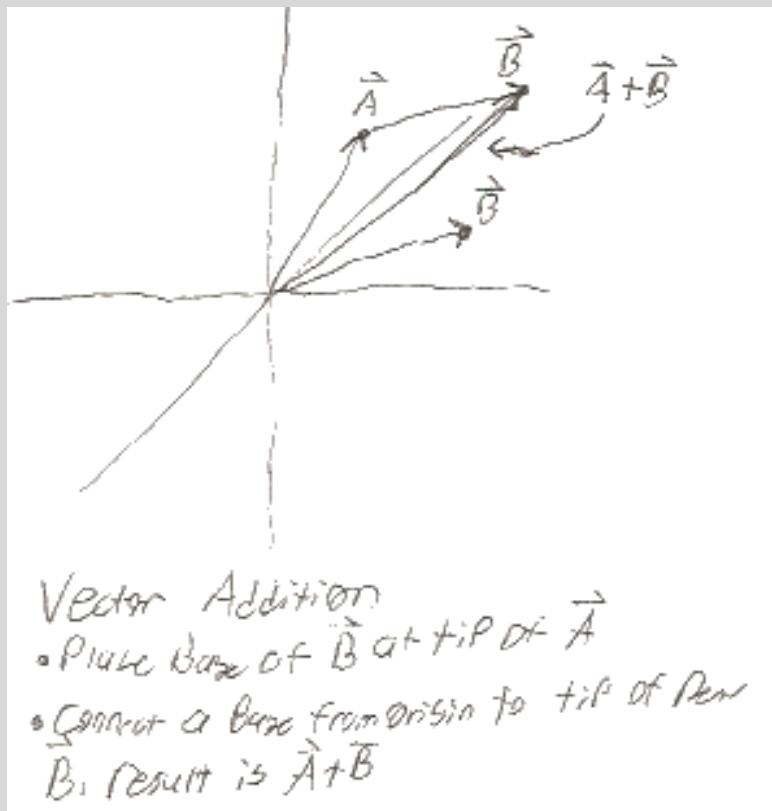
Unit Vectors & Vector Normalization

Before we jump into standard operations with vectors, let's talk a little more about that **unit vector** we saw a little while ago. As I stated just a short time ago, a unit vector is a vector quantity with a length of 1, what the diagram doesn't show you however, is that a unit vector doesn't necessarily need to point in the direction of an axis. The axis vectors (\hat{i} , \hat{j} , and \hat{k}) are simply common notation for unit vectors pointing along their respective axis. The actual definition of a unit vector is as follows: $\hat{\vec{A}} = \frac{\vec{A}}{\|\vec{A}\|}$, which can be expanded to write: $\hat{\vec{A}} = \frac{A_x\hat{i} + A_y\hat{j} + A_z\hat{k}}{\|\vec{A}\|}$. The actual task of converting a vector into this form is called **normalizing** a vector, which basically means that you're converting the vector to mainly serve as a directional pointer instance. This is sometimes also notated as \hat{A} , removing the arrow bar from the vector. To perform the normalize operation in Torque, you use the **vectorNormalize** function and provide it with a single vector instance:

```
function vectorNormalizeExample() {  
    %vector = "5 5 5";  
    %unitVector = vectorNormalize(%vector);  
    echo("The unit vector of \"%@vector@\" is \"%@unitVector@\"");  
}
```

Vector Addition

One of the most commonly used vector functions is basic addition of two vector quantities. This is a cumulative operation so $\vec{A} + \vec{B} = \vec{B} + \vec{A}$. To actually perform this operation, you simply add up the components of the vectors: $\vec{A} + \vec{B} = (A_x + B_x)\hat{i} + (A_y + B_y)\hat{j} + (A_z + B_z)\hat{k}$. From a visual standpoint, you can see where the addition vector will point by means of moving the two vectors into the following form:



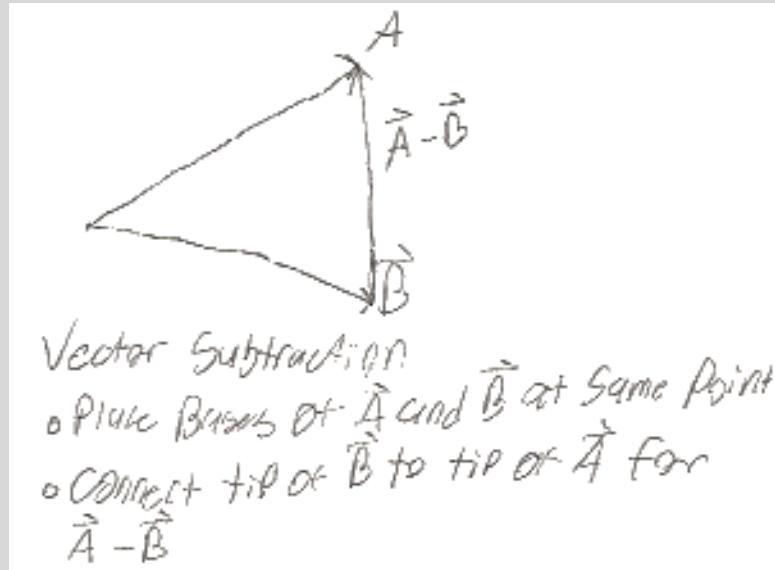
You'll likely use addition in instances where you want to manipulate positions, or if you have an object that needs to be moving at a specific angle away from the target object. To perform the vector addition in the engine, you use the **vectorAdd** function, and provide it with two vector parameters:

```
function vectorAddExample() {  
    %vector[0] = "5 5 5";  
    %vector[1] = "10 20 30";
```

```
%result = vectorAdd(%vector[0], %vector[1]);
echo("The result of adding "@%vector[0]@" to "@%vector[1]@" is: "@%result");
}
```

Vector Subtraction

The opposite operation is vector subtraction. This is a non-cumulative function so they cannot be interchanged, however you can accomplish the same operation by means of: $\vec{A} - \vec{B} = -(\vec{A}) + \vec{B}$. The actual component operation looks very similar to addition, with the obvious change of operators: $\vec{A} - \vec{B} = (A_x - B_x)\hat{i} + (A_y - B_y)\hat{j} + (A_z - B_z)\hat{k}$. You can also visualize the vector subtraction operation on a coordinate plane diagram by the following:



The most common applications of vector subtraction in the engine are again to work with position elements, and this time, it can also be used to find the path that must be followed by an object to intercept another object. In the engine's terms, you'll use the **vectorSub** function to accomplish vector subtraction:

```
function vectorSubExample() {
    %vector[0] = "5 5 5";
    %vector[1] = "10 20 30";
    %result = vectorSub(%vector[0], %vector[1]);
    echo("The result of subtracting "@%vector[1]@" from "@%vector[0]@" is: "@%result");
}
```

Vector Dot Product, Vector Scaling Product

Multiplication with vectors in mathematics isn't as easy as your standard multiplication because there are three different types of multiplication that can be performed on a vector. The **scalar product** is

used to simply “magnify” an existing vector and it returns a vector as a result of the calculation. The **dot product** of a vector is used to obtain a single number (also called a scalar) as a result of multiplying with components, and the **cross product** of a vector is used to obtain a vector result that is normal to the plane created by the two vectors that are being multiplied together.

The dot and scalar products are relatively simple to understand, and therefore don’t require any fancy diagrams to work from. We’ll start with the simple **scalar product** which just magnifies a current vector instance. Here is the formula for this operation: $S * \vec{A} = S * (A_x\hat{i} + A_y\hat{j} + A_z\hat{k}) = S * A_x\hat{i} + S * A_y\hat{j} + S * A_z\hat{k}$. To do this in Torque, you use the **vectorScale** command:

```
function vectorScaleExample() {  
    %vector = "5 5 5";  
    %result = vectorScale(%vector, 5);  
    echo("The result of performing the vector scale of 5 * \"%@vector@\" is: \"%@result@\"");  
}
```

The dot product has a few more applications that can prove to be useful when working with vectors, including some angle relationships as well. The **dot product** returns a single number result when performed on two vectors, and it takes the following formula: $\vec{A} \cdot \vec{B} = A_xB_x + A_yB_y + A_zB_z$. To use the dot product in the engine, you use the **vectorDot** function on two vectors:

```
function vectorDotExample() {  
    %vector[0] = "5 5 5";  
    %vector[1] = "10 20 30";  
    %result = vectorDot(%vector[0], %vector[1]);  
    echo("The dot product of \"%@vector[0]@\" and \"%@vector[1]@\" is: \"%@result@\"");  
}
```

While it seems like a simple operation, the dot product also has a very important geometrical relationship with two vectors to determine the angle between two vectors. By definition of the dot product, we know that: $\vec{A} \cdot \vec{B} = \|\vec{A}\| \|\vec{B}\| \cos \theta$, where θ is the angle between the two vectors. With some simple rearrangement, we get: $\frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|} = \cos \theta$, so $\theta = \cos^{-1} \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|}$. There are a few other useful applications for the dot product of two vectors, but you are welcome to explore online references or mathematics books regarding vectors and their properties on your own time to find these answers. Just for a quick little reference, you can write the above function as:

```
function vectorAngleExample() {  
    %vector[0] = "5 5 5";  
    %vector[1] = "10 20 30";
```

```
%dot = vectorDot(%vector[0], %vector[1]);
%mag[0] = vectorLen(%vector[0]);
%mag[1] = vectorLen(%vector[1]);
%angle = mACos((%dot) / (%mag[0] * %mag[1])); //What angular notation is this in?
echo("The angle between \"%@vector[0]@\" and \"%@vector[1]@\" is \"%@angle\"");
}
```

I should note that this function does not correct for vectors whose angles lie beyond the bounds of the arccosine function, I leave that to your own time and effort to solve.

Vector Cross Product

Finally, we have the **vector cross product**. The cross product is used to obtain a vector instance that is normal (90 degrees from the existing) to the plane created by the two vectors. The direction of the normal vector is determined by the **right hand rule** which states that the direction can be easily determined by using your right hand to point in the direction of the first vector, and then curl the fingers to the second vector and the thumb will point in the direction of the cross-product. The actual definition of the cross product follows similar rules of a **determinant** calculation on a 3D Matrix instance. For actual calculations however, we can write the cross product as the following: $\vec{A} \times \vec{B} = (A_y B_z - A_z B_y)\hat{i} + (A_z B_x - A_x B_z)\hat{j} + (A_x B_y - A_y B_x)\hat{k}$.

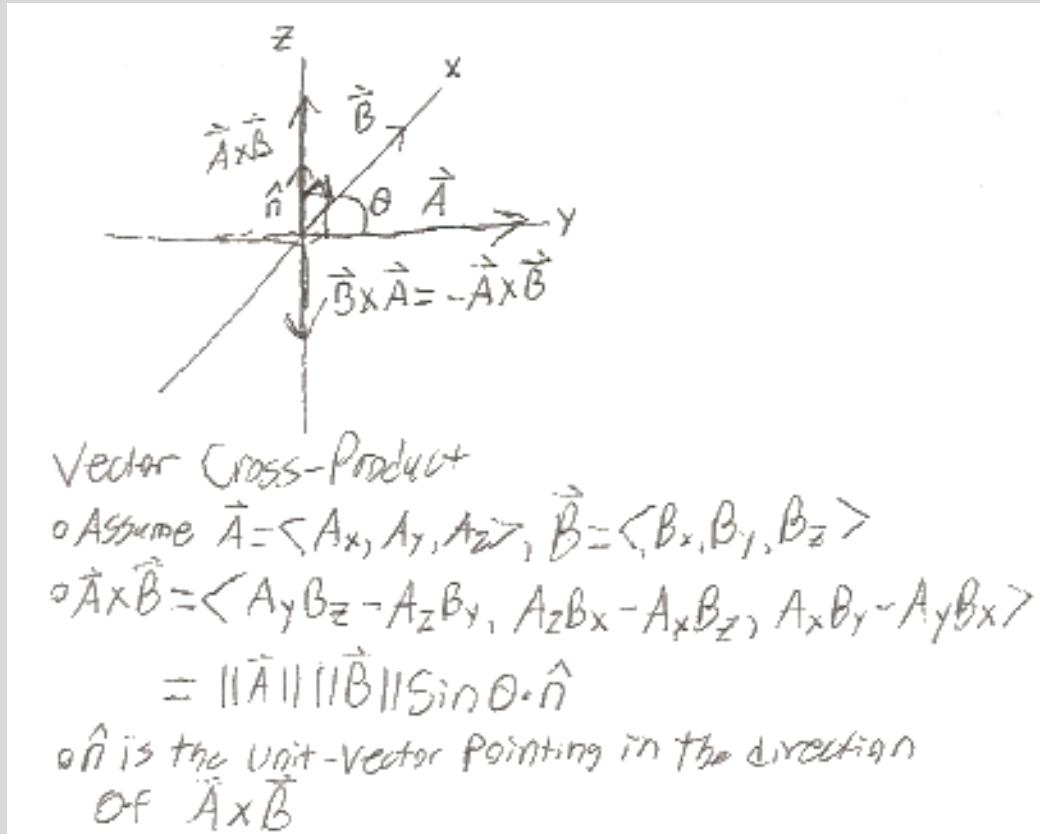
The cross product of a vector can also be used to establish a geometric relation with the angle of the vector, however the application of the dot product version of this calculation is usually simpler due to this calculation's requirement of a correction parameter that deals with the unit vector instance of the cross product. The definition of the cross product relation between the vector and the angle can be established by means of the following definition of the cross product: $\vec{A} \times \vec{B} = \|\vec{A}\| \|\vec{B}\| \sin \theta \hat{n}$, where θ is the angle between the vectors and \hat{n} is the unit vector of the vector created by the cross product. By means of some simple rearrangement of the terms in this equation, we can back out the unit vector definition as follows: $\frac{\vec{A} \times \vec{B}}{\|\vec{A}\| \|\vec{B}\| \hat{n}} = \sin \theta$. This still leaves the problem of having two vector instances to solve for a scalar, so we can perform a simplification here to get: $\frac{\|\vec{A} \times \vec{B}\|}{\|\vec{A}\| \|\vec{B}\|} = \sin \theta$. From here, it becomes another inverse sine relationship: $\theta = \sin^{-1} \frac{\|\vec{A} \times \vec{B}\|}{\|\vec{A}\| \|\vec{B}\|}$. Again, you need to be careful when using the inverse sine function here because of the boundaries of this function which can potentially lead to two possible angle outcomes in your solution that will need to be corrected. I leave the actual writing of this function to you for practice on TorqueScript with your own spare time.

To perform the vector cross product in the engine you use the **vectorCross** function and provide it with two vector instances, here's an example:

```
function vectorCrossExample() {
    %vector[0] = "5 5 5";
```

```
%vector[1] = "10 20 30";
%result = vectorCross(%vector[0], %vector[1]);
echo("The cross product of "@%vector[0]@" and "@%vector[1]@" is: "@%result);
}
```

And here's a diagram to show you how to establish the cross product relationship with two vector instances, as well as the directions they establish in these examples:



Basic Matrix Operations for TorqueScript

The next topic of mathematics we're going to cover deals with the concept of matrices. A Matrix in mathematics is an "array" of numbers and operators consisting of rows and columns. This topic should come quite easily to you now with your concept of arrays in TorqueScript. For most matrices, you can treat it as a 2D array instance where the first index refers to the row, and the second index refers to the column of the array. In the engine however, the matrices can be established in a 3x3 standard matrix, or a position matrix (4x4). For the standard matrix format, you provide it a string of 9 numbers

separated by spaces (words) and the engine will translate it as: $M = \begin{bmatrix} W_0 & W_1 & W_2 \\ W_3 & W_4 & W_5 \\ W_6 & W_7 & W_8 \end{bmatrix}$, where W_n refers

to the word at the specified word index. For a positional matrix, you provide 4 numbers separated again

by a space. This type of matrix however will be translated as follows: $M = \begin{bmatrix} 0 & 0 & 0 & W_0 \\ 0 & 0 & 0 & W_1 \\ 0 & 0 & 0 & W_2 \\ 0 & 0 & 0 & W_3 \end{bmatrix}$. The

operations provided by the engine for matrices are used to define positions, rotations, and to deal with advanced operations with vector instances.

Transforms

One of the important topics of the Matrix definition is the formatting of results of these types of calculations. The engine translates results of matrices into what is called a **transform**, which is a combination of position and rotation values for the object. You might have seen the word transform fly around a few times already if you've been poking around in the world editor or used the dump command on a game object class. This is the actual formatting of an object's current position and rotation in the world, and it's formatted like so: $\vec{T} = "P_x\ P_y\ P_z\ O_x\ O_y\ O_z\ O_a"$, a string of seven numbers. The first three numbers refer to the position coordinates of the object, the last four are orientation angles, where the first three values of these last four are orientations with respect to the x, y, and z axis's and the final number being the orientation angle.

The engine doesn't actually perform too many calculations as it relates to operations with direct matrices; however the usage of a transform in the engine is heavily used with these styles of functions, so let's get to talking about what kind of functions you'll be using here:

Matrix Vector Transformations

So we had a brief taste of Euler's work earlier with the logarithms, and now we're going to introduce the concept of an Euler Vector to establish a Transformation Vector. In our programming sense, an Euler Vector is simply a rotational analysis vector, where the X, Y, and Z components refer to the amount of magnitude the overall vector is being pushed in that direction. The application of the Euler Vector is to convert it into a format that is recognized by the engine's transformation vectors, or in more common terms: **yaw** (O_x), **pitch** (O_y), and **roll** (O_z), and the respective angular application of these vectors (O_a). These terms are then subjected to an angular axis analysis to convert the values into a proper transformation vector (7 term **transform** variable). This function in the engine is called **MatrixCreateFromEuler**, and to use it, you send it an Euler Vector with X, Y, and Z values to receive a transform in return.

Multiplying

While this result is nice and all, it's simply a rotational analysis. You may be wondering what use this could possibly have for a game, so let's talk about the actual analysis. Let's assume for a minute that our Euler Vector consists of a "spread" value, where we generate a random angle on each of the three axis's, and then constrain it to a specified range. The returned value, will actually consist of a transformation vector that is pointing in the direction of the spread analysis. But, we can further capture this result by multiplying it against an actual vector to get a world space vector instead. To perform this operation you use the **MatrixMulVector** function. If you need to grasp this concept a little further, let's combine the topics into an example:

```
function matrixAnalysisExample(%spreadForce) {  
    %fVec = "10 0 0";  
  
    %x = (getRandom() - 0.5) * m2Pi() * %spreadForce;  
    %y = (getRandom() - 0.5) * m2Pi() * %spreadForce;  
    %z = (getRandom() - 0.5) * m2Pi() * %spreadForce;  
  
    %mat = MatrixCreateFromEuler(%x SPC %y SPC %z);  
    %newvector = MatrixMulVector(%mat, %fVec);  
}
```

From this little example, you can establish something very common to FPS games, weapon fire spread; where a higher value of %spreadForce will result in larger angles being generated.

While there are many applications out there for advanced mathematical concepts using matrices, such as tensor vector applications for advanced forcing mechanics for physics applications, most of them can be accomplished using some of the more simpler functions and mechanics the engine has to offer you.

Bézier Curves, Determining a Path

When you come to working with games, one of the big topics I get asked about a lot is how to generate a path between two points. This question depends on quite a few questions, like what type of curve the path needs to contain, and how many data points do you need to create this curve instance? For most applications of this question, the answer lies in something called a Bézier Curve. A Bézier Curve is a mathematical application that allows you to connect a set of **control points** with a line instance of a specified order (linear, quadratic, cubic, etc).

Mathematical Definition

The Bézier Curve can be defined for any order or magnitude for a curve instance and it follows the following explicit mathematical formulation: $B(t) = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i P_i$. Some of these terms may seem a little “new” to you, so let’s talk about this in more detail. The first thing is that “scary looking” symbol with the parameters on it. This is called a summation in mathematics. You’ve already used this in Torque, this is the mathematical equivalent of a **for loop**. The bottom term, $i = 0$ is the initial value of the looping parameter, and the top value, n , is the ending value of the loop. In our case, n refers to the order of magnitude of your curve. The next term to talk about here is $\binom{n}{i}$ which is a special definition which equates to: $\binom{n}{i} = \frac{n!}{i!(n-i)!}$, the ! symbol in mathematics is the factorial symbol which means to multiply together all numbers starting with the first number down to 1, or: $n! = (n) * (n - 1) * (n - 2) * ... * 1$. Finally we have the P_i term which specifies the **control point** at the specified index. What we have created here is a form of **interpolation** which is a type of formulation that picks the best point over a set period of time, or in our formula, the t variable, which is constrained between 0 and 1.

Linear, Quadratic, and Cubic Definitions

By taking the above definition of the Bézier Curve, and then applying it to Linear, Quadratic, and Cubic orders, we obtain the following definitions of the curve. For the Linear case:

$B(t) = \sum_{i=0}^1 \left(\frac{1!}{i!(1-i)!} \right) (1-t)^{1-i} t^i P_i$, and then we add up all of the summation terms to get:

$B(t) = \binom{1}{1} (1-t)^1 t^0 P_0 + \binom{1}{0} (1-t)^0 t^1 P_1$, which simplifies to the following: $B(t) = P_0 + t(P_1 - P_0)$ or as it is more commonly expressed: $B(t) = (1-t)P_0 + tP_1$, which is the definition of the Linear Bézier Curve. For the Quadratic form of the curve: $B(t) = \sum_{i=0}^2 \left(\frac{2!}{i!(2-i)!} \right) (1-t)^{2-i} t^i P_i$ equates as:

$B(t) = \binom{2}{2} (1-t)^2 t^0 P_0 + \binom{2}{1} (1-t)^1 t^1 P_1 + \binom{2}{0} (1-t)^0 t^2 P_2$, and is simplified to the following:

$B(t) = (1-t)[(1-t)P_0 + tP_1] + t[(1-t)P_1 + tP_2]$. Or written in its standard form, the Quadratic curve formula simplifies to: $B(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2$. You should start notice a pattern developing for an expanding order of magnitude for each of these formulations. Instead of writing out the full derivation of the cubic formula, here is the standard equation for the cubic Bézier curve definition: $B(t) = (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)t^2 P_2 + t^3 P_3$.

So, to sum things up here, we have the linear definition as: $B(t) = (1-t)P_0 + tP_1$, the quadratic definition as: $B(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2$, and the cubic definition as: $B(t) = (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)t^2 P_2 + t^3 P_3$.

Control Points of the Curve

Now that you have the standard definition of the Bézier curve formulas at your disposal, let's talk about those P_n variables. These variables refer to the **control points** of the Bézier curve instance, or basically specified instances where the curve will "bend" or connect. For the curve instances, the first point and the last point are the endpoints of the curve (start & end), for linear instances, these will simply connect a line by means of Linear Interpolation (see the **mLerp** function), but once you begin to go into higher orders, the mechanics to create the curve are handled by a **spline**, or basically a culling term which will be the control points that are not the endpoints, where the amount of these points are counted as $N = n-1$, where n is the degree of the curve, and N is the culling points.

But how do you actually generate these points? First and foremost, let's correct a very common misconception about the above formula. Most people will think that this curve is useless to them because they're used to a $y(x)$ style of terminology, where changes in a curve formula relate to both x , and y , so we can only solve for one axis, and this is true, but not entirely true. The style of function presented above is called a **parametric function**, or basically a function of one axis that changes with time ($t = 0 - 1$), ergo, this function needs to be performed on all axis's you need to solve for. With that problem fixed, let's talk about control points, and actually this one's completely at your disposal here. The only points that are "fixed" in terms of needing to pay close attention to, are the end points of the Bézier curve. The remaining point(s) however, can be calculated at your own line of thought as needed by the function instance. For example, a quadratic curve that needs to arc upward over the starting points can be created by taking the mid-point of the starting and ending values, and then raising the 'Z' term by a specific amount. Since this is an interpolation function, the formula will find a "best-fit" curve to go around the spline term, or in your instance the midpoint of the quadratic curve instance. Obviously

you need to be cautious of placing the curve directly at the mid-point to prevent multiple-solution style instances from developing, but overall, you're in charge of how this works.

Writing a Bézier Curve in TorqueScript

This last part of the sub-chapter will start to introduce you to a topic we'll also be covering later on, and that is the process of actually transforming mathematical formulations into computer code. So, let's start by transforming the linear function into a code segment. Note that you could just as easily use **mLerp** for this, as it's the exact same thing, but this is for demonstration purposes:

```
function linearBezierExample(%start, %end, %quality) {  
  
    //We use %start and %end as position variables, to solve a Bezier curve, we need to strip out the components.  
  
    //The quality variable is used to determine how many parametric terms to solve for.  
  
    %sX = getWord(%start, 0);  
    %sY = getWord(%start, 1);  
    %sZ = getWord(%start, 2);  
  
    %eX = getWord(%end, 0);  
    %eY = getWord(%end, 1);  
    %eZ = getWord(%end, 2);  
  
    for(%i = 0; %i < %quality; %i++) {  
  
        %t = (%i * 1.0) / (%quality * 1.0); //Multiply the variables by 1.0 to convert them to decimals  
  
        %x = (1 - %t) * %sX + %t * %eX;  
        %y = (1 - %t) * %sY + %t * %eY;  
        %z = (1 - %t) * %sZ + %t * %eZ;  
  
        %pos[%i] = %x SPC %y SPC %z;  
    }  
  
    //Print out the positions.  
  
    for(%i = 0; %i < %quality; %i++) {  
        echo("Point \"%@%i+1@\": \"%@%pos[%i]\"");  
    }  
}
```

Feel free to run this script with a set of parameters, and change them around a bit. You'll notice that the results of this formula will match the **mLerp** function when provided with the same values. I'll give you the quadratic example, but the cubic formulation I leave to you to explore and solve on your own time for practice. You can see here that I've named my variables with similar names as the actual Bézier curve variables to help with definition styles. So, let's look at the quadratic example now for another conversion example:

```
function quadraticBezierExample(%start, %end, %quality) {  
    %sX = getWord(%start, 0);  
    %sY = getWord(%start, 1);  
    %sZ = getWord(%start, 2);  
    %eX = getWord(%end, 0);  
    %eY = getWord(%end, 1);  
    %eZ = getWord(%end, 2);  
  
    //Solve for the midpoint.  
  
    %midL = vectorLen(%start, %end) / 2; //Length between start and end divided by 2  
    %dir = vectorNormalize(vectorSub(%end, %start)); //Unit vector pointing in the direction from start -> end  
    %midPt = vectorAdd(%start, vectorScale(%dir, %midL)); // Add the start to the scaled (unit vector * length) to get mid point  
  
    //Strip points.  
  
    %mX = getWord(%midPt, 0);  
    %mY = getWord(%midPt, 1);  
    %mZ = getWord(%midPt, 2) + 5.0; //Add 5.0 to create a curve that "bends" up.  
  
    for(%i = 0; %i < %quality; %i++) {  
        %t = (%i * 1.0) / (%quality * 1.0); //Multiply the variables by 1.0 to convert them to decimals  
  
        %x = mPow((1 - %t), 2) * %sX + 2 * (1 - %t) * %t * %mX + mPow(%t, 2) * %eX;  
        %y = mPow((1 - %t), 2) * %sY + 2 * (1 - %t) * %t * %mY + mPow(%t, 2) * %eY;  
        %z = mPow((1 - %t), 2) * %sZ + 2 * (1 - %t) * %t * %mZ + mPow(%t, 2) * %eZ;  
        %pos[%i] = %x SPC %y SPC %z;  
    }  
  
    //Print out the positions.  
    for(%i = 0; %i < %quality; %i++) {  
        echo("Point \"%@%i+1@\": \"%@%pos[%i]\"");  
    }  
}
```

So, you'll notice here I did some vector trickery to fetch the midpoint, so let me explain how that works. First, what I did was establish the length between the two vectors and then divided the number by two to get the length from the starting point to the mid-point. Next, I fetched the unit vector between the starting point and the end point by performing vector subtraction on end – start to get the vector pointing in the direction it needs to with a unit vector length of 1, basically converting it to a direction vector. Finally, to get the mid-point, I add the starting vector to the unit vector times the length to the mid-point to get the mid-point position. From there, it's a repeat exercise of the previous

example converting the quadratic formulation into a script. Be sure to remember the order of operations and functions like **mPow** and **mSqrt** when working with powers and roots.

So, this should give you a good idea of how to define Bézier curves for any order of magnitude and how to convert them into a TorqueScript function to be used in the engine. Before we finish up here, let's talk about one last little piece of mathematics to use in the engine.

Mathematical Uncertainty, AKA: Randomization

Finally, we're going to actually define something you've seen quite a bit already. The **getRandom** function is used to generate random numbers between specified points, and over intervals [0-1]. This function actually has three different "modes" of operation, and is dependent on how many parameters you send to the function.

The first case of this function occurs by sending no parameters, and when you do this the function will generate a random **floating point number** between 0.0 and 1.0. A floating point number is a fancy term for a 32-bit decimal number. The first format uses the following style of code:

```
function getRandomExample_one() {  
    %rand = getRandom();  
    echo("Generated :"@%rand);  
}
```

This is a great tool when you are generating a random time interval for a parametric function, or if you need to generate a random multiplier between 0 and 1.

The next case of the function occurs when you send a single integer parameter to the function. This will generate a random number between zero and the number you sent to the function. To use this style of the **getRandom** function, you do the following:

```
function getRandomExample_two(%max) {  
    %rand = getRandom(%max);  
    echo("Generated the following number in the [0-{@%max@}] range:"@%rand);  
}
```

This form of the function is a good tool to use when you want to fetch a value above the number zero, but also includes the number zero in the possible numerical generation sequence.

The final form is the one you've seen the most so far in our previous examples, and it is used when you send two parameters to the function. The generator will generate a value in the range of the two numbers sent to it.

```
function getRandomExample_three(%min, %max) {  
    %rand = getRandom(%min, %max);
```

```
echo("Generated the following number in the [%min@ - %max@] range:@%rand);  
}
```

Use this form of the function when you know the range of values you want to generate in, and that range either includes negative numbers, or does not include the number zero, otherwise, you should use the one parameter form.

So, you may think this is all I have for this little sub-chapter section, but you'd be wrong. There's actually a good amount to talk about when it comes to the properties of uncertainty in mathematical applications, such as the examples you have seen in the past. But one of the big topics here is the actual method behind the madness (pun not intended) and that is the concept of numerical seeding.

Numerical Seeding

To actually initialize a random number generator in this case, we need what is called a random seed. This seed is then used in the pseudorandom number generator (PRNG) code to generate a sequence of numbers that is then used by the code bit itself to actually "give" the results of calling a getRandom function. The most commonly used "seed" in a computer application is the current UTC time string since its constantly changing.

The actual differences in random number generators comes with the formulation that generates the numerical string that the numbers draw from, the generator itself by definition is therefore predictable and not truly "random" in that sense. But, it is close enough to be counted as a random number when called in the engine, so what is the key that's used by Torque? This also uses the UTC time string that is pulled directly at the initialization of the engine instance. You can actually set the seed to a fixed value or have it pull the current time when calling the function to adjust this value. The function demo below shows you how:

```
function getRandomExample_four(%min, %max) {  
    %cSeed = getRandomSeed();  
    echo("The current seed is: %@cSeed@, setting this to 10.");  
    setRandomSeed(10);  
    %rand1 = getRandom(%min, %max);  
    echo("With a seed of 10, we generated: %@rand1@");  
    //set the seed to the current UTC time  
    setRandomSeed(-1);  
    %rand2 = getRandom(%min, %max);  
    echo("With the current time seed, we generated:@%rand");  
}
```

How to beat Pseudo-random

One very easy solution you can use to “beat” pseudo-random is right there in your face from the last section. If the needs of your program are heavily reliant of numbers that are “always” random, you could implement a function override on the getRandom function call to update the seed to the current UTC time before actually generating the number. Here’s a quick sample of that:

```
package randomGenerator_Update {  
  
    function getRandom(%min, %max) {  
  
        setRandomSeed(-1);  
  
        parent::getRandom(%min, %max);  
  
    }  
  
}  
  
if(!isActivePackage(randomGenerator_Update)) {  
  
    activatePackage(randomGenerator_Update);  
  
}
```

So I hope this quick section on random numbers and the seeding algorithm behind it helps you to improve your overall usage of the function when working with it in the engine.

How to solve Mathematical Problems in Programming

The last segment of our mathematics journey will bring together math and script/code so you can understand the process of problem solving when it comes to working with the engine. First and foremost, let’s discuss some simple rules to ensure you always get the result you need.

- Name variables accordingly, for formulation purposes, you may want to keep them as the same names as their mathematical counterparts.
- Use parenthesis to keep blocks of code that require answers in the specified order to be performed in that order.
- Use the simplest form of the math equation to get your answer, this will save on processing and make logic errors less likely.
- Don’t be afraid to separate operations to fit in multiple function declarations if it helps make the problem easier to solve.
- Use echo statements in all steps of the calculations in you’re getting logic errors to pin the location of the error to make it easier to fix.
- Use error checks in places they need to be used (For example: if it’s possible to have missing parameters or if a number might be zero during a division step).

Some Final Mathematical Rules

As long as you use these basic rules when writing your functions, you’ll be sure to keep the most out of the mathematical processor for the engine. Now, there are a few topics to cover before we end this chapter as some mathematical applications, specifically advanced ones might use terminology and

symbolism you haven't seen before, so let's go over the most common ones and methods you can use to solve for them.

Numerical Integration

Integration is an advanced mathematical topic usually taught in an introductory calculus course in college. This process is commonly used for two purposes. First, it is used to reverse the operation known as taking a derivative of an equation. And the other is the one we're interested in, which is the method for finding the area underneath the specified bounds of a curve. Integration in this form, called a definite integral falls under this form: $N = \int_a^b f(x) dx$, where 'a' and 'b' are the starting and ending coordinates of 'x' (we use x because the **differential** is **dx**). Since integration is a higher topic for a calculus course, I won't actually discuss it here, but instead, I'll go over some methods to actually solve these styles of equations by approximation methods. These won't give you the exact answer, but it will give a result that is close enough for application purposes.

Midpoint (Rectangle) Approximation

One of the simplest forms of numerical integral approximation comes in the form of the midpoint rule, or as it's more commonly known, the rectangle rule. The way to imagine this rule is if you have your curve. This rule stretches a set number of same-sized rectangles to the "best-fit" locations to calculate the area of the curve. We call it the rectangle rule because of that, and the midpoint rule, because the middle-point on the top or bottom of the rectangle touches the curve. For each rectangle that is above the axis of interest, we add to the total, and for the ones below, we subtract from the total. Let's look at an example of this. The formula for the midpoint rule is specified as follows:

$N = \int_a^b f(x) dx \approx (b - a)f\left(\frac{a+b}{2}\right)$. This may seem a bit confusing, so let's try a basic example so you can see how this works. Assume I have a line equation $f(x) = x$, and I want to find the area under that line between the points of 0 and 10 on the 'x' axis. Personally having taken a calculus course I can tell you by applying the integration on this, we get: $N = \int_0^{10} x dx = \left[\frac{1}{2}x^2\right]_0^{10} = 50$. But, let's apply the midpoint rule here instead. $N = \int_0^{10} x dx \approx (10 - 0)f\left(\frac{0+10}{2}\right) = 10 * 5 = 50$. The reason the answers match here is because the equation uses a linear interpolation method, and we just used it on a linear equation. But if you go to higher orders, then you'll start to introduce error into the equation. For example, let's try x^2 . $N = \int_0^{10} x^2 dx = \left[\frac{1}{3}x^3\right]_0^{10} = 333.\overline{33}$. Now let's use the midpoint rule on this equation: $N = \int_0^{10} x^2 dx \approx (10 - 0)f\left(\frac{0+10}{2}\right) = 10 * \left(\frac{10}{2}\right)^2 = 10 * 25 = 250$. So here, you can see a pretty significant error. The reason is that the rectangle method encompasses large areas outside the bounds of the equation. To get around this, we need to use a different approximation method.

Trapezoid Rule

The second most commonly used rule for numerical integration is called the Trapezoid rule, which as you can imagine, uses trapezoids to approximate the area under the curve. The formulation for the trapezoid rule is almost identical to the midpoint rule with one change: $N = \int_a^b f(x) dx \approx (b - a)\frac{f(a)+f(b)}{2}$. Let's see how this works on our quadratic example. $N = \int_0^{10} x^2 dx \approx 10 * \frac{0+100}{2} = 500$. So as you can see here, there's still a level of error in our equation, but you might notice that our answer

lies near the middle of these two rules. So, if we could just combine these rules in a way, there might be a route to obtain a more accurate solution.

Composite Method

Well, actually there is a way to combine these two rules. The derivation of a further rule (discussed next) actually leads to a composite definition of the two discussed rules. So, let's quickly walk through this. Assume the following: $R = (b - a)f\left(\frac{a+b}{2}\right)$, and $T = (b - a)\frac{f(a) + f(b)}{2}$, we can solve for the overall error of these two approximation methods as $E(R) = -\frac{1}{24}(b - a)^3 \frac{d^2f(a)}{dx} + O((b - a)^4)$, and $E(T) = \frac{1}{12}(b - a)^3 \frac{d^2f(a)}{dx} + O((b - a)^4)$, where the notation of $O(x)$ refers to a term that is asymptotically proportional to x . You should also be aware that these two $O(x)$ terms are not equal to one another in these cases. But the important part of these formulations is that these error terms can be canceled when combining these two formulas together with a weighted average rule, which equates to a form of **Simpson's Rule** and is defined as: $N = \frac{2R+T}{3}$, which is the composite method. We already solved for both the M & T terms for our x^2 example, so by using those results we get $N = \frac{2(250)+500}{3} = 333.\overline{33}$, which matches the answer perfectly.

Simpson's Rule

The last method to apply to numerical integrations, and the one you should stick with for all of your applications is a rule known as **Simpson's Rule**. This approximation rule uses the following formulation: $N = \int_a^b f(x) dx \approx \frac{b-a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right]$, which is based on a quadratic interpolation method. Let's apply this rule to our quadratic example x^2 . $N = \int_0^{10} x^2 dx \approx \frac{10}{6} [0 + 4 * (25) + 100] = 333.\overline{33}$. As I briefly discussed earlier, our formula here is actually a more common form of the composite rule of the midpoint and trapezoid methods. This method is a highly accurate for numerical integration, however there is still the potential for error. While usually small, some applications of this method would be deemed to be inefficient to solve, and therefore you need to use a more advanced form of **Simpson's Rule** to help combat this error. This next rule is known as **Simpson's 3/8 Rule** and it uses a cubic interpolation instead of a quadratic interpolation. This formula will generate an approximation that is almost more than twice as accurate as the standard approximation method above. This approximation formula is defined as $N = \int_a^b f(x) dx \approx \frac{(b-a)}{8} \left[f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b) \right]$.

There are additional formulations out there that are even more accurate than these two forms of **Simpson's Rule**, however most game applications will be safe to get away with the 3/8 rule for the highest level of approximation. If your level of mathematics has reached a point of at least a Calculus 1 course in college to understand differentials and integrations, I highly recommend you apply the Newton-Cotes formulation to attain an interpolation method to a higher order for the level of approximation needed by your application.

Differentiation

So now that you understand the basics of integration, let's talk about the reversed operation. In most calculus courses you learn about differentiation first, since its notation is used prior to integration, however we were only interested in the approximation methods, and therefore it was valid to discuss it first.

The topic we're going to talk about now is the topic of a differential, or basically an infinitesimal value of change of a function with relation to another variable. As you can imagine, it can be extremely useful to understand how something changes over time, and mathematics has quite a few tools at your disposal in order to understand this, unlike the integration rules though, there are no "easy" means of approximation without actually using the rules of differentiation, so let's talk about the most common ones here.

Differentiation Rules

Before we can actually talk about the rules, let's talk about the standard notation of a **derivative** of a function. There are two common forms of this. The first form is by means of the mathematical **prime** symbol, which takes the following notation: $f'(x)$, which means to take the derivative of the function with respect to the variable x . The other notation used by derivatives is something you've seen a basic form of not too long ago called the differential form, and it appears as so: $\frac{d}{dx}(f(x))$, where the denominator refers to the variable being differentiated, and the $f(x)$ term is what is being differentiated. So with that in mind, let's begin.

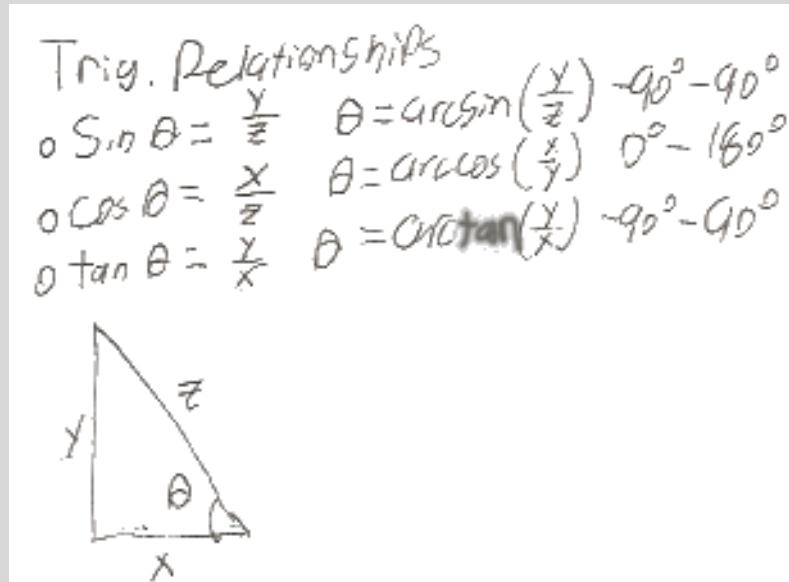
These first set of rules are very simple rules to follow and should very quickly become "on the top of your head" style answers. The first rule is called the **constant rule** and it is defined for single numbers without a variable attached to it. This rule states: $\frac{d}{dx}(c) = 0$, where c is any number. The next rule is the **power rule** and is used for basic function differentiations. This rule is defined as: $\frac{d}{dx}(x^n) = nx^{n-1}$. There are three quick examples I want to give when referring to the power rule so you understand how it works. The first is a basic one: $\frac{d}{dx}(x^2) = 2x^1 = 2x$. The next is a constant and a variable: $\frac{d}{dx}(3x^2) = 3 * 2x^1 = 6x$. Finally a form of a linear variable definition: $\frac{d}{dx}(7x) = 7 * 1x^0 = 7 * 1 = 7$. The second and third examples demonstrate the **constant multiple rule** which is defined as: $\frac{d}{dx}(cx^n) = cnx^{n-1}$. But we know that functions are more complex than just one term at most times. The **sum** and **difference rules** are very simple and take the exact same form: $\frac{d}{dx}[f(x) \pm g(x)] = \frac{d}{dx}[f(x)] \pm \frac{d}{dx}[g(x)]$. Basically put, if you have a function of multiple terms, each term needs to have the derivative taken.

The next set of rules is where things start to become a little more "tricky" in nature for functions. The **product rule** is used when two non-simplifiable terms are multiplied together and is defined as $\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}(g(x)) + g(x)\frac{d}{dx}(f(x))$. For example, assume we have a function $f(x) = (10x + 5x^2)(32 - 4x)$ and want the derivative of this function. The long route would be to

factor out the terms, but instead we can simply perform the product rule to obtain: $\frac{d}{dx}[f(x)] = (10x + 5x^2)(-4) + (32 - 4x)(10 + 5x) = (-40x - 20x^2) + (320 + 160x - 40x + 20x^2) = 80x + 320$. The **quotient rule** works in a similar fashion and is defined as: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}(f(x)) - f(x)\frac{d}{dx}(g(x))}{[g(x)]^2}$.

This rule looks very complex at first, but functionally, it has the exact same traits as the product rule.

Earlier in the chapter you learned about the trigonometric functions. Those can be differentiated as well and take on the **trigonometric function rules** of derivatives, and are defined as the following: $\frac{d}{dx}(\sin(x)) = \cos(x)$, $\frac{d}{dx}(\cos(x)) = -\sin(x)$, $\frac{d}{dx}(\tan(x)) = \sec^2(x)$, $\frac{d}{dx}(\sec(x)) = \sec(x)\tan(x)$, $\frac{d}{dx}(\csc(x)) = -\csc(x)\cot(x)$, and $\frac{d}{dx}(\cot(x)) = -\csc^2(x)$. You might notice a few “head-scratching” things with these rules, and those are your inverse trigonometric functions. To explain these, I start by reminding you that: $\tan(x) = \frac{\sin(x)}{\cos(x)}$. If you remember back when I introduced the trigonometric rules, there was a diagram that went with it that looked a bit like this:



Well, in actual mathematics, you can pull three more trigonometric relations out for x, y, and z by using the inverted relations of the functions. These are the **secant**, **cosecant**, and **cotangent** relationships. In this diagram, they are defined as: $\sec \theta = \frac{z}{y}$, $\csc \theta = \frac{z}{x}$, and $\cot \theta = \frac{x}{y}$. Therefore by mathematical definitions, these functions are defined as: $\sec x = \frac{1}{\sin x}$, $\csc x = \frac{1}{\cos x}$, and $\cot x = \frac{1}{\tan x} = \frac{\cos x}{\sin x}$.

The final important rule to teach here is the **chain rule** of differentiation which comes into effect when you have functions within functions, and is defined as: $\frac{d}{dx}[f(g(x))] = \frac{d}{dx}[f(g(x))] \frac{d}{dx}[g(x)]$. Since this rule has a high level of importance to differentiation, let's do a few examples to ensure you understand how it works. $\frac{d}{dx}[(3x - 2)^2] = 2(3x - 2)^1 * (3) = 6(3x - 2) = 18x - 12$. In the example there, you can see the first thing we did was apply the **power rule** to the entire parenthesis term,

treating $(3x - 2)^2$ as a $f(x)$ term, so we took the derivative of the entire term, leaving the inside term alone. This term therefore becomes: $2(3x - 2)^1$, then we applied the derivative to the inside term which simply becomes 3. Finally, we simplified everything down to get our final result. Now, let's consider another example $\frac{d}{dx} [\sin(3x)] = \cos(3x) * 3 = 3\cos(3x)$. Again, this is a simple thing to understand once you have some practice. Take the derivative of the outside term, leaving the inside alone, and then perform the derivative to the inside and multiply it to the top result. Now let's try something a little more tricky: $\frac{d}{dx} [\sqrt{\sin(x)}] = (\sin(x))^{\frac{1}{2}} = \frac{1}{2}(\sin(x))^{-\frac{1}{2}} \cos(x) = \frac{1}{2} * \frac{\cos(x)}{\sqrt{\sin(x)}} = \frac{\cos(x)}{2\sqrt{\sin(x)}}$. And actually, that really isn't tricky. What you need to remember is something I briefly mentioned a while back, and that is you can treat roots as a $1/n$ power. So we convert the square root to a $\frac{1}{2}$ term and then apply the derivative to the $\frac{1}{2}$ power. From there it's another set of rinse, wash, and repeat examples where you take the derivative of the outside, and then the inside and simplify it. Before we finish up, let's look at one last case. $f(x) = \sin^2(4x^2)$. So what do we do here? Actually, this is an example of a **repeated application of the chain rule**, where you have multiple functions inside functions. Let's split this into separate functions by first using a different notation: $f(x) = (\sin(4x^2))^2$. Let's assume to start inner functions at the letter u and keep going. So our forms work as so: $f(x) = u^2$. Then we have $u(x) = \sin(v)$. And finally we have $v(x) = 4x^2$. And with that, it makes the job a lot easier. $\frac{d}{dx} [\sin^2(4x^2)] = 2 \sin(4x^2) * \frac{d}{dx} [\sin(4x^2)]$. Now we perform the second differentiation: $2 \sin(4x^2) * \frac{d}{dx} [\sin(4x^2)] = 2 \sin(4x^2) (\cos(4x^2)) * \frac{d}{dx} [4x^2]$. Finally, we can do the last application and then simplify: $2 \sin(4x^2) (\cos(4x^2)) * \frac{d}{dx} [4x^2] = 2 \sin(4x^2) (\cos(4x^2))8x$. Therefore our answer is: $\frac{d}{dx} [\sin^2(4x^2)] = 16x\sin(4x^2)\cos(4x^2)$.

That should give you a basic overview of the basic rules of differentiation. I do realize calculus is a very challenging subject, and you shouldn't be expected to know these rules when working with games, but just be aware that some tough mathematical problems are out there, and you will need to know how to solve them, and more specifically, how to convert them to code. If you have trouble grasping these rules, I highly suggest using forms of online solvers like the one here:

<http://www.derivative-calculator.net/> to handle the work for you, but at least knowing these rules to do things yourself is a good step. If you wanted to see if I did my work right or not, feel free to validate it with the online calculator, here's the formula in their syntax: $(\sin((4x^2)))^2$

Table of Differentiation Rules

We did cover quite a few rules up there, and I left out some because they were irrelevant or not needed for discussion. For your reference, here's a list of common derivative rules as you may see them. For these rules, assume that u and v are referring to any function of x , and the notation u' refers to the derivative of the function u . All of the following account for the **chain rule** as well.

Rule Name	Derivative
Constant Rule	$\frac{d}{dx}(c) = 0$
Constant Multiple Rule	$\frac{d}{dx}(cu) = cu'$

Power Rule	$\frac{d}{dx}(u^n) = nu^{n-1}u'$
Add & Subtract Rules	$\frac{d}{dx}(u \pm v) = u' \pm v'$
Product Rule	$\frac{d}{dx}(uv) = uv' + vu'$
Quotient Rule	$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{vu' - uv'}{v^2}$
Absolute Value Rule	$\frac{d}{dx}(u) = \frac{u}{ u } (u'), \text{ where } u \neq 0$
Natural Logarithm Rule	$\frac{d}{dx}(\ln(u)) = \frac{u'}{u}$
Euler's Number Rule	$\frac{d}{dx}(e^u) = e^u u'$
Sine Rule	$\frac{d}{dx}(\sin(u)) = \cos(u)u'$
Cosine Rule	$\frac{d}{dx}(\cos(u)) = -\sin(u)u'$
Tangent Rule	$\frac{d}{dx}(\tan(u)) = \sec^2(u)u'$
Secant Rule	$\frac{d}{dx}(\sec(u)) = (\sec(u) \tan(u))u'$
Cosecant Rule	$\frac{d}{dx}(\csc(u)) = -(csc(u) \cot(u))u'$
Cotangent Rule	$\frac{d}{dx}(\cot(u)) = -\csc^2(u)u'$
Arcsine Rule	$\frac{d}{dx}(\sin^{-1}(u)) = \frac{u'}{\sqrt{1-u^2}}$
Arccosine Rule	$\frac{d}{dx}(\cos^{-1}(u)) = \frac{-u'}{\sqrt{1-u^2}}$
Arctangent Rule	$\frac{d}{dx}(\tan^{-1}(u)) = \frac{u'}{1+u^2}$
Arcsecant Rule	$\frac{d}{dx}(\sec^{-1}(u)) = \frac{u'}{ u \sqrt{1-u^2}}$
Arccosecant Rule	$\frac{d}{dx}(\csc^{-1}(u)) = \frac{-u'}{ u \sqrt{1-u^2}}$
Arccotangent Rule	$\frac{d}{dx}(\cot^{-1}(u)) = \frac{-u'}{1+u^2}$

From the Programming Perspective

So now that you have an understanding of all of the mathematical principles needing to create advanced operations and calculations using programs, you're ready to actually dive into the code itself. The important part about bringing together your mathematical formulations and the higher order work in mathematics is meant to bring the answer to the simplest form possible, such that the computer can solve it with as little processing as it needs. By performing complex operations and then bringing it down to a level of easy solvability, you will ensure you can do cool mathematical calculations when working with vectors on objects while not blowing up the processor with lengthy formulas to solve at every

second. This is what you need to accomplish, and I hope you've learned enough here to be able to do this.

Closing Notes

So that does it for the “scary” math chapter in this guide. I hope you take as much as you need from this section of the guide. The next few chapters will revisit the editors and expand on materials inside the editors by combining the game instance and objects within the game, with your newfound scripting and mathematical knowledge. Remember, if you ever need help solving a complex math problem, there’s plenty of online resources that are in place to help you, and be sure to read through the engine’s documentation to see what mathematical functions are available for you.

Chapter 10: GUI's Again, Wiring Things Together

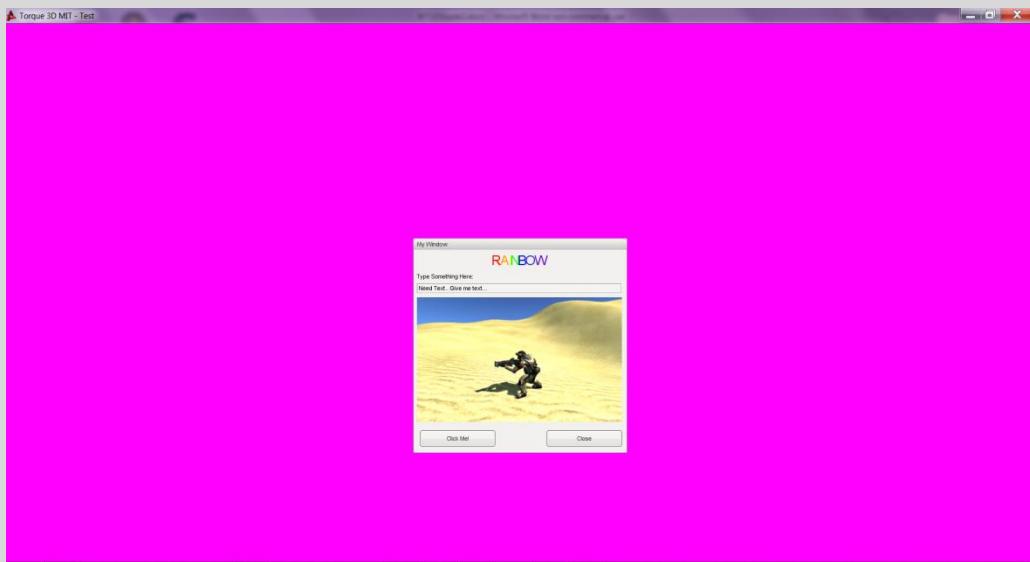
Chapter Introduction

Well now that we've gotten through the mathematics stuff, we can move onto our next topic. We're going to revisit the topic of GUIs and the GUI Editor, but now we're going to take things to the advanced level by adding our knowledge of TorqueScript to the mix to actually get the GUI to handle tasks in game. This chapter will teach you how to get controls to perform custom functions, how to create functions for each individual control, and how to actually wire things together to create useful and user-friendly dialogs.

For this chapter, you're going to want to have the GUI Editor (Chapter 4) in full knowledge to understand how to design GUIs to be able to create similar things, and you should have the knowledge from Chapters 6 – 8 completely down, with some aspects of the math stuff known as well. So if you're lacking in a bit of knowledge, you might want to go back to those chapters and get the fine details down, or use some of the Torque 3D documentation to help you along there as well.

The Command Parameter in more Detail

So, way back in Chapter 4, we played around with the GUI Editor in the engine and created a sample dialog that looked a bit like this:



And this is a good start for a GUI control. But, it really doesn't have any custom functioning to it and it doesn't do anything special other than display a message box when you click a button, or close the GUI itself when you click the other message. We're going to change this now.

Revisiting the Command Parameter

So back in Chapter 4 in the “Buttons and Button Variants” section, I briefly introduced you to the **command** parameter on the GUI controls, and then I gave you a few different things to fill in there to give the buttons something to do. The **command** parameter in an actual definition is an input function which tells the engine to evaluate the specified command when the GUI performs a specific action. That

specific action varies from control to control, for instance the click buttons will run the **command** parameter when you actually click the button, and the radio buttons will run it when the button is selected, and so on. One of the key things to know about GUIs is that each individual control is treated as an object instance (Chapter 8), so as long as the control has a unique name to it, you can access it directly in your functions by means of the access operator (.). So, let's quickly review the "command" parameters of the control we made earlier now that we actually have TorqueScript knowledge to use. We had two buttons in our example control. We'll start with the Close button, which allowed you to close the current GUI Dialog. You typed in a command instance that I specified to you and the GUI editor assigned it to the command parameter. In terms of what the engine saw when you did that, it turned your string into this:

```
MyAwesomeNewGUI_Closebutton.command = "Canvas.popDialog(MyAwesomeNewGUI);";
```

I'll get to what that actually does in the sense of the engine here in just a bit, but you may notice that the function we're calling is contained inside a string instance. Our other button has an even better case of the string function. The button we had with the text "Click Me", we had the control print out a message box with the text that was typed into the text input control. We accomplished that by setting the following:

```
MyAwesomeNewGUI_popupDialog.command = "MessageBoxOK(\"Hey There!\", \"You Typed  
\"@MyAwesomeNewGUI_textEnter.getText());";
```

Actually, what you did was type that into the GUI Editor, but in a sense of the TorqueScript side, this is what you called in the engine. You might also notice a new escape sequence in there, the \" sequence. Here's how that works, if you remember back to Chapters 6 and 7, I told you that all string instances needed to be "closed", which means that each quotation mark needs a closing pair. Well in some instances of the engine, functions are placed in strings, so you'd need to be able to open a second layer of string within the first, this is what the \" sequence allows you to do. This sequence places a new quotation mark at the specified point, inside the current string without closing it. But why is this important at all? Well, what happens when the **command** parameter is run is a special engine function is executed, and that is the **eval** function.

The eval Function

This is a topic I originally had planned for Chapter 7, but decided to hold it to this point for relevance. The **eval** function is used to as it sounds; evaluate a string of code in the engine. This function is how the console (~) and all of your GUI Controls work. It is a one parameter function that accepts a string input of code to be evaluated in the engine, taking this form: **eval(%code);**

The reason I'm giving this to you now is because for a good deal of functions, where string type variables or input is used, you will need to use the \" terminator sequence to get the quotation marks into the string and close it properly. When you type into a box in something like the GUI editor to input a function, the engine will convert your quotation strings to the proper format in order to run it correctly. Here's an example of how to use the eval function in the engine:

```
function evalExample() {  
    eval("schedule(1000, 0, \"echo\", \"Hello World!\");");  
}
```

So again, pay close attention to usage of the quotation mark escape sequence that is used to create closed string instances inside of another string instance. Failing to generate the proper string closings will result in a syntax error when the code is called.

How to 'actually' use the control parameter

So, now we know what makes the control parameter work when it's called. But how do we actually go about using this in our own GUIs? Well, as I stated before, the control parameter is executed when a specific action of the GUI control is performed, such as a button click or a change in a value of the GUI. This in turn fired off the **eval** command which as you now see, can be used to change global variables, or even fire off another function. So, what can we do with this in mind? Well, we're going to get to making control functions here in just a bit, but keep in mind two very important concepts. The first is that GUI controls can be named instances, so you could use the object-access relationship (`name.function()`), and secondly that **eval** can both change variables and run function instances.

Now that you've got the control parameter's inner workings understood, let's actually do something with this parameter. But before we start writing custom GUI functions, let's start by talking about some of the existing ones.

GUI Functions

Before we learn how to write GUI control functions, which by the way is simply a trip down memory lane to Chapter 8 with a few new concepts, let's talk about some of the functions the engine provides to you for your GUI controls. Do realize that there are hundreds upon hundreds of actual functions for all of the GUI controls in the engine, so we're only going to cover the important ones. For a full list, you should review the Torque 3D engine documentation.

Global GUI Functions

The first group of functions classifies as **global** functions, or basically a set of functions designed to be used to affect the overall status of the GUI. Not all surprisingly, these commands revolve around yet another engine object, called the **GuiCanvas**. The GuiCanvas object is basically a fancy object name for the application's window instance. Obviously this means you could easily call `GuiCanvas.dump()`; in your console to get a list of functions for this object instance, but we're actually going to cover all of them in here.

A quick note regarding these, they all take the format `GuiCanvas.X()`, where X is the name of the command below. I'll provide details and examples as necessary.

setContent

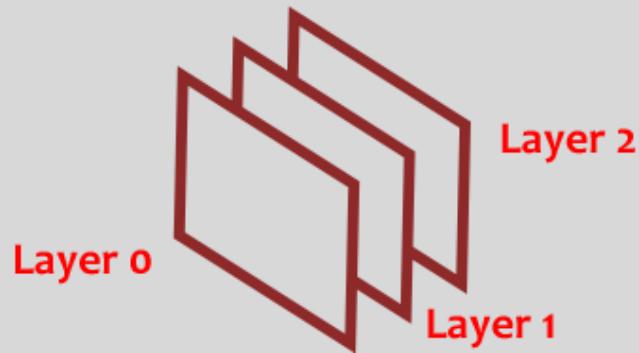
The first function we're going to learn about is the **setContent** command. There are a few functions for the GuiCanvas that function like this command, so I'll explain the differences between

them here. The `setContent` function is used for **full screen** GUI instances, such as your in-game GUI (`PlayGui`). This command accepts one parameter, and that parameter is the name or ID of a `GUIControl` to display in the window. When you call this command, the new GUI will cover all existing GUI instances on the screen. Ideally, you'll want to use this for instances such as in-game full-window dialogs, or the main menu instances, or a credits screen. Basically, any dialog that needs to cover the entire game application window, use the `setContent` function. For example, here's how you'd use this command on our GUI from Chapter 4:

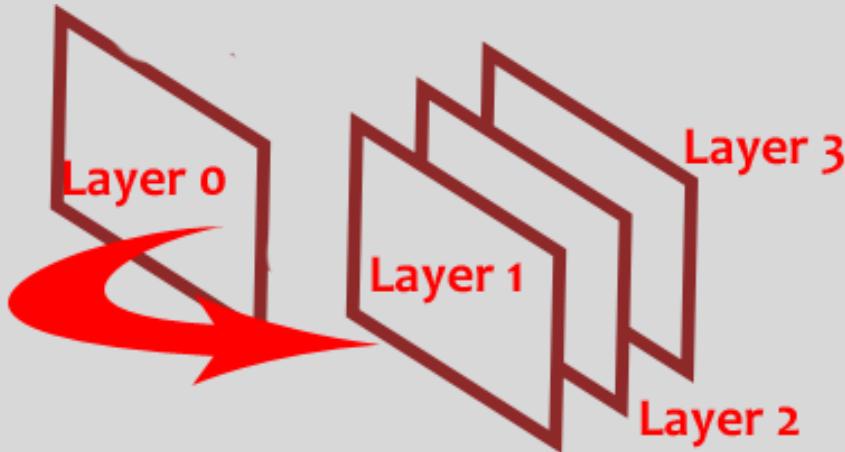
```
function displayOurGUI() {  
    Canvas.setContent(MyAwesomeNewGUI);  
}
```

pushDialog

The next command we're going to talk about is the **`pushDialog`** command. Like `setContent`, this command is used to place a GUI Control instance on the screen. Unlike `setContent` however, this command is used to push any style of control to the screen. You can have many controls on the screen at any given time, and they will be rendered according to their **layer**, or basically the priority of rendering. This works like your computer's windows, where the current window takes priority, and then the last used one, and so on. Layers are numbered and start at zero. This image should help you understand:



The **`pushDialog`** function has three arguments, but you only need to use one. The first argument, like the prior function is the name or ID of the GUI Control you'd like to place on the screen. The second argument is a numerical value saying which layer you'd like to place your new control on, and the last argument is a Boolean that says if the new dialog needs to go in the center of the screen. For 95% of the instances you use this command, the last two parameters can be sent as empty for the following result:



Notice how when you use the **pushDialog** command, all prior GUI instances on the screen are moved back a layer, and the new dialog becomes layer zero. Remember, this only occurs when you only send the name or ID of the GUI, otherwise you'll provide the layer number. So, to do this to our Chapter 4 GUI, you'd write this:

```
function displayOurGUI_pushDialog() {  
    Canvas.pushDialog(MyAwesomeNewGUI); //pushDialog has 3 parameters in total, if we used all three here it would look like this:  
    // Canvas.pushDialog(MyAwesomeNewGUI, 0, false);  
}
```

popDialog

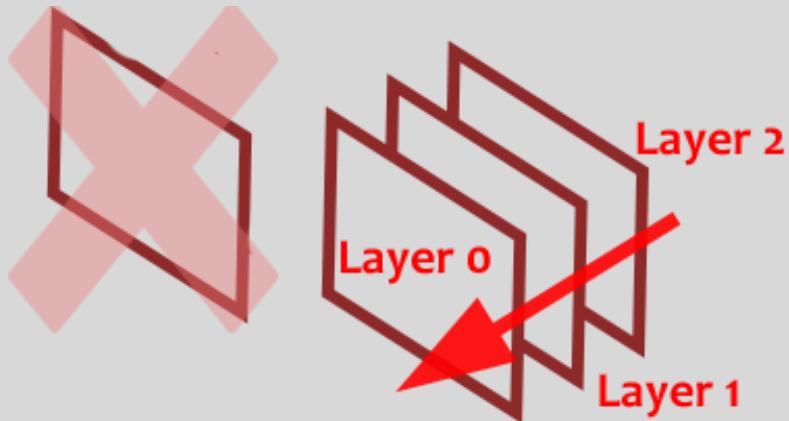
Next up is the **popDialog** command which is used to remove a GUI control instance from the screen once it's pushed onto it. This is a very easy command to use, all you do is provide it with the name of the GUI control you'd like to remove from the screen. For example, our dialog from Chapter 4 would look like this:

```
function removeOurGUI() {  
    Canvas.popDialog(MyAwesomeNewGUI);  
}
```

This should look very familiar to you, because this is the **exact same** command we used on our close button in the GUI.

popLayer

The **popLayer** command is similar to the **popDialog** command, however this command uses the concept of the numerical layers of your GUI controls to remove a GUI control. To use this function you can either send it a numerical parameter containing the layer you want to remove from the screen, or you can leave it blank to remove layer zero. When you do that, all prior layers on the screen move to the "next open" index value, like so:

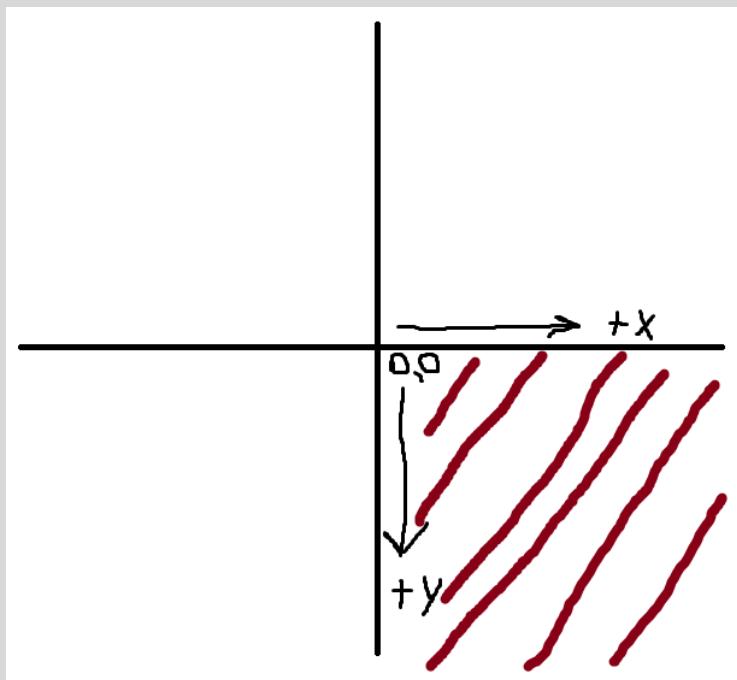


cursorOn & cursorOff

These two functions are used to enable and disable the mouse cursor when you move the cursor across the GUI. Alternatively, you can bypass the need for this function by using the **nocursor** property on a GUI control to enable the mouse when the specific control is active. This is mainly used for in-game GUI dialogs.

getCursorPos & setCursorPos

These two functions are used to obtain and set the position of the GUI cursor on the screen. These functions are not relative to the current control, but to the entire screen, and be sure to remember how GUI's are “formatted” position wise in computers. This diagram will help you remember that:



The `getCursorPos` function does not have any arguments and returns a two dimensional position string (Chapter 9). The `setCursorPos` function accepts a two dimensional position string as a single argument to use that function, if you don't send a position, it will place the cursor at the position "0 0".

getExtent

The **`getExtent`** function is used to obtain the size of your game window. Recall the concept of an extent versus the size of a window from Chapter 4, this will return a two dimensional position string where the first value is the x-extent and the second value is the y-extent. Also be sure to remember as mentioned above how the y-axis is flipped in screen space coordinates. You do not need to send any parameters to this function.

setWindowTitle

While you won't likely need to touch this function, it's used to change the title of the game's window instance. The current use of this function can be found in the game's `main.cs` function in the top directory with the game executable. The current form of this function uses the global variable `$appName` to set the title of the window. This global variable can also be found in the same `main.cs` file. To use this function, you send one string variable indicating the name you would like to set.

GUIControl Functions

The other topic of GUI functions we're going to cover is the functions used by the game class **`GUIControl`**, which is the base class instance for all GUI objects you can place on the screen; therefore these functions are valid for all of your controls. Again, there are a lot of functions that are in here, but we'll cover the important ones you'll probably need the most. For the full list, refer to the engine documentation.

One of the differences here between the `GUICanvas` functions and the `GUIControl` functions is that instead of calling a constant name convention like `GuiCanvas.x()`; here you'll need to actually use the name or ID of the control you want to use the function on, followed by the name here, IE: `Name.X()`;

getParent

The **`getParent`** function is used to obtain the GUI control instance that is "above" the control you use this function on in the hierarchy. This function has no parameters. For instance, in our Chapter 4 example, the close button control would have this output:

```
function fetchParentOfCloseButton() {  
    %parent = MyAwesomeNewGUI_Closebutton.getParent();  
    echo("The Parent control is: "@nameToID(%parent).getName());  
}
```

isActive

The **`isActive`** function is a quick boolean function to test if the GUI control in question is "active", which means the control is currently being used by the application instance. This function has no parameters.

isVisible

The **isVisible** function is another quick boolean check function used to tell the scripting engine if the control in question is visible to the user or not. This function has no parameters.

setVisible

The **setVisible** function is used to tell the engine to either make a control visible or make a control invisible to the user. If a control is invisible, it cannot be used (for instance, an invisible button control cannot be clicked). This function accepts a single boolean as a variable, where true will make the control visible and false will make the control invisible.

isAwake

The **isAwake** function is used to test if a specific GUI element is “awake”, or currently part of the active GUI instance on the screen. In terms of what we’ve covered before, this means if the control is part of the layer zero GUI on the screen. This function has no parameters.

getPosition

The **getPosition** function is used to tell you where the specific GUI control is located relative to the control’s parent object on the screen. If the control has no parent object in your GUI, it will be relative to the GUI Canvas object. This function has no parameters and returns a two dimensional string parameter containing the x and y coordinates on the screen.

getCenter

The **getCenter** function is used to tell you where the “center point” of the specific GUI control is located. Again, this is relative to the parent in terms of positioning on the screen. There are no parameters to this function and it returns a two dimensional string parameter containing the x and y coordinates on the screen.

setCenter

The **setCenter** function is used to update the position of the specified GUI control by moving the center point of the specific control. This function does not update the extent, or size of the control it only moves it. This function accepts two integer parameters. The first is the x-coordinate you would like to have the center of the GUI control become, and the second is the y-coordinate you would like. This function does not return a value.

setPosition

Like the setCenter function, the **setPosition** function is used to update the position of the specified GUI control. This function also uses the two integer parameter format. The primary difference here is that instead of moving the center point, you’re moving the screen position, or in a term you’ll understand easily, it’s the top-left most point of your GUI control.

GUIControl Callback Functions

So those are your most common GUI control functions that you’ll be using when working with GUI elements in your code. Now we’re going to introduce a slightly new topic here. This is the concept of a **callback function**. A callback function is simply a pre-defined function instance that is in place and is

automatically called by the engine when a certain action is performed on the object. What you do, is override this function by writing your own instance of it, and then when the engine triggers the callback, your code will be called in this instance.

Again, each individual control has numerous instances of callback functions that could potentially be overridden by you, but we're only going to cover the ones defined for GUIControl since all other GUI element instances use these. Since this is a new concept, we'll cover a few examples in the listings below so you can hopefully understand how this works. So for the examples, we're going to use our Chapter 4 close button as an example, which is named **MyAwesomeNewGUI_CloseButton**.

onAdd

Our first callback function is a very common function you'll actually see a few times in the coming chapter as well. The **onAdd** callback is sent to the engine when an object is spawned, or for a more technical term, when the object becomes registered in the engine (is assigned an ID). This callback has no additional parameters besides the object variable enforced due to the object function (Chapter 8). Here's a small sample:

```
function MyAwesomeNewGUI_CloseButton::onAdd(%this) {  
    echo("Close button has been added, ID: "@%this);  
}
```

So as you can see by our first example, a callback function is essentially a pre-defined object function that has built in engine wiring to be called when certain events happen. You can take advantage of these functions to wire in additional functions and tasks for your GUI elements. Now let's talk about the other callbacks.

onRemove

The **onRemove** callback is the opposite of the **onAdd** callback, and is called right before the object is deleted in C++ (which is a fancy way of saying deleted from existence). You can use this callback to do clean-up tasks and delete any additional objects that need to be deleted when this control dies, or clean some settings that are no longer relevant. Here's a sample of this callback:

```
function MyAwesomeNewGUI_CloseButton::onRemove(%this) {  
    echo("Close button being deleted, ID: "@%this);  
}
```

onWake

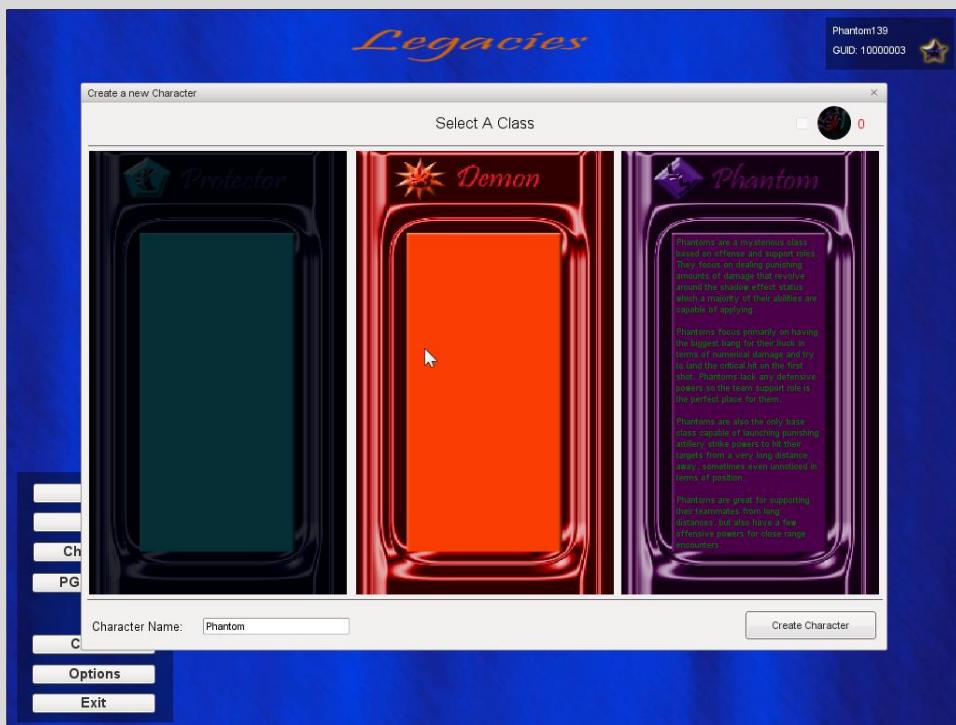
The **onWake** callback is activated when the GUI control instance becomes "active", or is pushed to the screen. This function will be directly called as a result of a call to **pushDialog** or **setContent** on the parent control that holds the specific element you have this callback function assigned to. You could also write this callback to the parent holder control (**MyAwesomeNewGUI**) to write some initialization tasks to perform when the GUI window is opened every time. There are no special parameters for the **onWake** callback function besides the standard object parameter.

onSleep

This is the opposite of the onWake callback. The **onSleep** callback is initialized when the specific control no longer is active, or basically has been removed from the screen. This function will trigger as a result of the **popDialog** or the **setContent** functions. You can use this call to do some post-close tasks on your GUI instance. **Closing a GUI dialog in the engine does NOT delete the object instance. It will remain alive to be used until a call to the object's delete function is performed. A closed dialog simply exists in the form of "sleep" until it is pushed back onto the screen.** There are no special parameters for this callback besides the standard object parameter.

onMouseEnter

The **onMouseEnter** callback is triggered when the mouse cursor moves into the selected GUI control instance. You need to have the **useMouseEvents** parameter set to true on the control for this callback to be triggered, but it has numerous useful tools to be able to do some very neat things. For a quick example, here's something I did using some mouse events for a small project of mine:



This callback has no special parameters attached to the function aside from the standard object parameter instance.

onMouseLeave

The **onMouseLeave** callback is used when the mouse cursor leaves the GUI control instance. Like the **onMouseEnter** callback, you need to have the **useMouseEvents** parameter set to true on the selected control instance in order to trigger this callback, and there are no special parameters associated with this callback besides the object parameter.

onVisible

The **onVisible** callback is used when the GUI control in question changes its visibility status (this works either way [visible -> invisible or invisible -> visible]). This function has a boolean parameter associated with it to indicate what the status is becoming. Since this is the first time dealing with a special parameter, let's look at an example:

```
function MyAwesomeNewGUI_CloseButton::onVisible(%this, %newVisStatus) {  
    echo("Close button switching visibility to: @"%newVisStatus);  
}
```

So, as you can see from this sample code here, the object parameter still remains (since this is "still" an object function [Chapter 8]), but then we added the second callback parameter to this instance to indicate what the visibility of the control is now being switched to.

onActive

The **onActive** callback is used when a GUI control in question changes its "active" status in the engine via (**setActive**). This callback is triggered when a control either becomes active or inactive, and the callback has a special boolean parameter associated with this status.

Final Notes on Engine-Defined GUI Functions

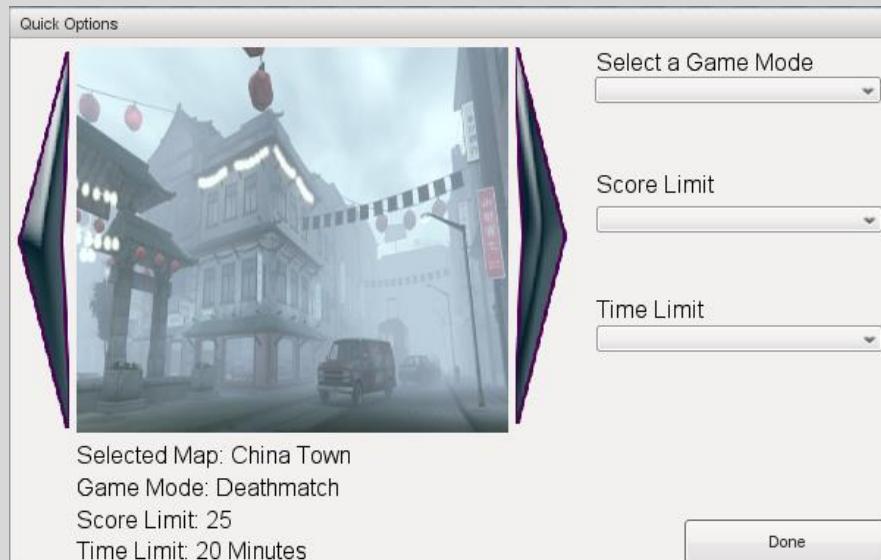
So as I have mentioned countless times already, these are only some of the common functions you will encounter the most in your programming tasks associated with the GUI system in the engine. There are hundreds upon hundreds of other control functions and callbacks available to you in the engine. To actually get to use these, you'll need to use the dump command on a GUI control instance in question to see its function list (callbacks are in here too!), or refer to the engine's documentation for more information regarding the controls and how to use them in the engine. Also a little thing you might be noticing is 95%+ of the callback functions start with the word **on**, so keep that little trick in mind if you need help differentiating callbacks and regular functions.

Creating Custom GUI Functions

Now that we've covered the topic of general functions for GUI's, we can do a quick exploration of another topic needed by GUIs in your programming, and that is the task of defining custom functions for your GUI instances. For the most part, this is a quick review section of the Chapter 8 topic of object functions since that's all we're doing at this point. You should be able to put this notion together in your head by now: "Writing code in the engine for objects is simply a combination of using existing functions with a few lines of custom defined code to do your needed work".

The GUI

So, let's do one nice example to help you understand how this works in terms of the engine's code system. Assume I have created a mission & option selection GUI that is used to select the map and a few quick options for the users, and it looks a bit like this:



Please note that this is a custom dialog, you'll need to make your own

So as you can see by the dialog we have a few things going on here. First of all there are three drop-down box controls that can be populated with potential options, there is an image control with the preview of the map in question, and then two arrows to move the map selection left and right. Finally, we have some text controls to display a textual version of our selections and a done button to apply these settings.

Strategy of Approach

To help you understand this topic, I'll teach you my approach to tackle this GUI's programming so you can hopefully understand how a new challenge can actually turn into a few very easy problem solving steps. So let's assume I have created this dialog in my GUI editor, and have saved it and now I need to write some functions for the GUI to actually get it to do what I need it to do. Here's my style to how I go about this:

1. What does this GUI need to do?
 - I ask myself, what needs to happen at each point in the GUI, when it's opened up, when a user makes a selection, and when the dialog is closed.
2. What does each control need to do?
 - I ask myself, what each control needs to accomplish in order to serve the purpose of the GUI as defined in #1.
 - I try to get an initial list of all of the functions I will need to accomplish my goal.
3. Pseudocode
 - Fancy term for logical structure that isn't actually code, but looks like it. This can contain things like variable names and what these variables need to have in them, but without actually writing the code.
 - I create some small bits of code, including some function stubs and write the needed callback functions for my dialog, without actually filling anything in there. I also write function stubs for bits of code that may exist at some point.

4. Actual Code

- I then fill in the blanks with the actual code needed by the dialog, referring to tools like the engine's references, and the dump command for any functions I may have missed in step #2.

5. Testing

- I then load the code into the engine and test the dialog. I try all of the possible options to ensure each control is now performing as it needs to.
- I get a list of broken items and code that needs to be fixed.

6. Conclusion

- I then repeat steps #3 - #5 until all items in the GUI are functioning as intended.

Application of the Strategy: Let's Write the GUI!

So let's use this strategy to actually approach the GUI above now. So the very first thing I do here is ask myself "What do I need this GUI to do?". This is a very easy step in which you basically do a complete overview of what needs to happen. For instance of this control, the answer is simple: "This is a GUI in which the user must be able to select a mission and some options for their game". This completes step #1, at this point in time if I haven't already done so, I would fire up the engine and hop into the GUI editor and put the GUI together.

Now comes the next step, which is asking what each control needs to do. So, let's just say I have the GUI named **MPSelectDialog** and the controls follow the naming scheme as defined in Chapter 4. In which case I then answer the question for each individual control, and start to gather a list of functions I will need for this control:

- MPSelectDialog: Store the settings of the user for use in the game. (onAdd, onWake, onSleep)
- MPSelectDialog_GameDD: Contain a list of possible game mode options for the user to select (add, onSelect, getSelected, setSelected, getText)
- MPSelectDialog_ScoreDD: Contain a list of possible score limit options for the user to select (add, onSelect, getSelected, setSelected, getText)
- MPSelectDialog_TimeDD: Contain a list of possible time limit options for the user to select (add, onSelect, getSelected, setSelected, getText)
- MPSelectDialog_MapPreview: Contains the map preview image to display to the user (setBitmap)
- MPSelectDialog_LeftButton / MPSelectDialog_RightButton: Allows the user to select different maps in the list of available maps (N/A uses **command** parameter)
- MPSelectDialog_Text[Map,Game,Score,Time]: Contains a textual representation of the current selection (setText)
- MPSelectDialog_DoneButton: Used to apply all of the current settings and close the GUI instance (N/A uses **command** parameter)

So as you can see from the above list, I have compiled a list of all of the controls on the screen in this GUI. Now comes step #3, which is to write the psuedocode for the dialog. This varies for each

individual user so I won't actually write it. Just know that this is basically there to help you see a path from an idea to the actual code. Some people don't need this step and will simply dive into writing code, which is also ok to do. I just prefer the psuedocode route to actually help me make sure I know I'm getting everything I need to do, done.

So after that step is complete, we move into step #4, which is to actually write the code for the GUI. This is a very in-depth process here, so I'll walk you through how it works. What I like to do is a top-down approach for things like this. Or basically, start at the highest GUI object in the tree and move down the list. So in our GUI here, we're going to start with **MPSelectDialog** itself, and work our way down into each of the individual controls.

For the MPSelectDialog, the important task happens when the GUI is loaded into the engine right away. The GUI needs to have everything ready to go for the users as soon as they open the dialog for the first time, so this function will be responsible for setting up the other dialog options:

```
function MPSelectDialog::onAdd(%this) {  
    //set initial option values.  
  
    %this.selectedMap = "Chinatown";  
  
    %this.selectedMode = "Deathmatch";  
  
    %this.selectedTime = "20";  
  
    %this.selectedScore = "25";  
  
    //Populate GUI dropdown objects  
  
    MPSelectDialog_GameDD.add("Deathmatch", 0);  
    MPSelectDialog_GameDD.add("Team Deathmatch", 1);  
    MPSelectDialog_GameDD.add("CTF", 2);  
  
    MPSelectDialog_ScoreDD.add("25", 0);  
    MPSelectDialog_ScoreDD.add("50", 1);  
    MPSelectDialog_ScoreDD.add("Unlimited", 2);  
  
    MPSelectDialog_TimeDD.add("20 Minutes", 0);  
    MPSelectDialog_TimeDD.add("30 Minutes", 1);  
    MPSelectDialog_TimeDD.add("Unlimited", 2);  
  
    //Set GUI dropdowns to their initial values  
  
    MPSelectDialog_GameDD.setSelected(0);  
    MPSelectDialog_ScoreDD.setSelected(0);  
    MPSelectDialog_TimeDD.setSelected(0);  
  
    //Set the image.  
  
    MPSelectDialog_MapPreview.setBitmap("Levels/Chinatown.png");
```

```
//Set text fields.  
  
MPSelectDialog_TextMap.setText("Selected Map: Chinatown");  
  
MPSelectDialog_TextGame.setText("Game Mode: Deathmatch");  
  
MPSelectDialog_TextScore.setText("Score Limit: 25");  
  
MPSelectDialog_TextTime.setText("Time Limit: 20 Minutes");  
  
}
```

That may look like a hefty amount of code, but it's actually not really all much. We're just setting some default values and giving settings to the individual controls. I'm not going to cover those GUI control functions for the controls you haven't seen before; this is an exercise in learning how to find the functions, and how to use them. Just know what they do, and why they're being used there. Also, I won't cover the **onWake** and **onSleep** functions here since we're not actually wiring this specific dialog into the game itself, but know this is where you'd transmit the information to the GUI to update the fields when it's opened, or to the server when the GUI is closed.

Instead, I'll walk you through some more callback and function examples as noted by our new dropdown dialogs (`GuiPopUpMenuCtrl`). This control is best used when you have many different static options that can be selected from, and you want control over the ability to quickly change them at any given point (For example, say a user makes a change to another setting, you could change the options in the dropdown list).

```
function MPSelectDialog_GameDD::onSelect(%this) {  
  
    //Called when a user changes the dropdown selected option  
  
    %selected = %this.getText();  
  
    %row = %this.getSelected();  
  
    //Do stuff...  
}
```

This should give you a basic idea of how to work the drop-down box dialogs. I actually went into the game and pulled the list of functions to find the `onSelect` callback, as well as the `getText` and `getSelected` functions.

For the remainder of the dialog, specifically the two buttons, you can simply write a function and bind it to the command parameter of the button so when the user clicks the left or right arrows the dialog adjusts the current selected mission to either select the next mission or the previous mission. A good exercise to try is to write one function to do both, and find out how to use the command parameter to handle both cases. The hint I'll give here is to remind you that the command parameter runs the `eval` function, so have a peek back in that section. You should also pay close attention to the **done** button we created. Remember for that, I wrote in the `onAdd` function to the dialog that we were

storing all of the parameters of the options to the **%this** variable (`MPSelectDialog`), so make sure you're using that properly.

The last two steps in the GUI creation process are to test the dialog, and fix any problems in your existing functions. So be sure to test all of the options, and all of the different possibilities for the options to make sure the dialog is performing as expected. If a problem comes up, you can use the echo command to print out the parameters to see what your variables are being sent as and locate the problem that way.

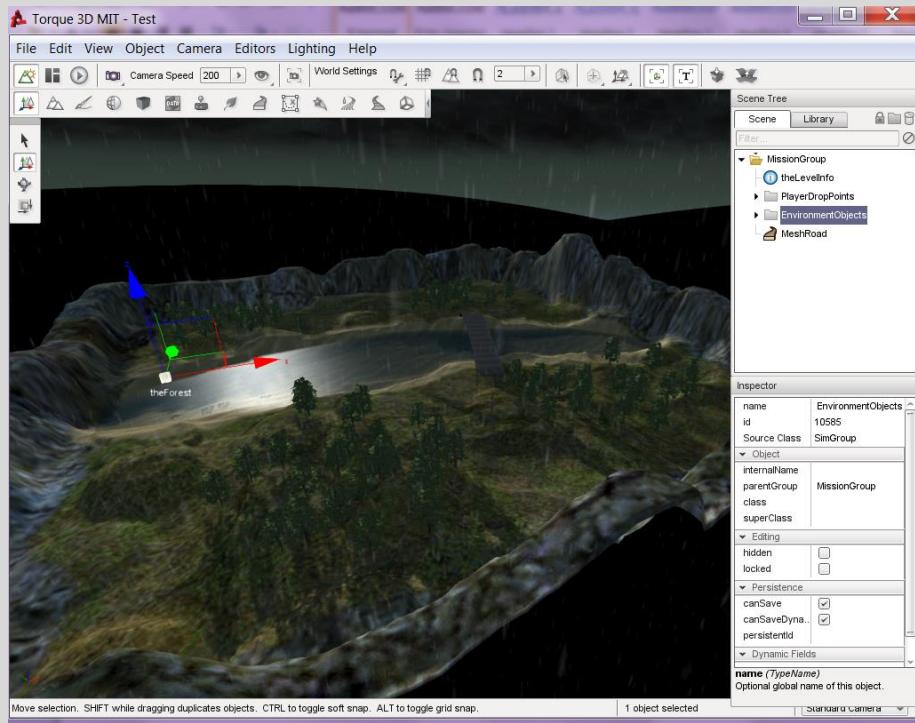
Additional Wiring Topics

The last thing I want to talk about in the GUI chapter is how to "wire" your dialogs together with proper functioning. The concept of **wiring** means that all of your functions tie into another function, or into another dialog. This is how you actually make more advanced dialogs that can do so much more for you in the engine. You may actually find this hard to believe, but the World Editor, is actually a highly complex GUI dialog with countless wired connections between multiple dialogs, and the game instance itself.

Wiring: Dialog to Dialog

What you should have established by this time in the guide, is how each individual object in the engine is treated as either a numerical id reference, or a named instance which you can access (Chapter 8) to use functions, and properties. GUI dialogs are no different, and their functions and properties can be manipulated on the fly, as long as the dialog remains "alive", or basically, has not been deleted via the delete command.

So, let's take a quick trip down memory lane to chapter 3, specifically the SimGroup section of chapter 3 where we placed SimGroup instances in other SimGroups. So imagine for one minute, that the game instance works like your computer. The files are stored on some disk drive, and you need to map out a path to the file, for example: `C:/Users/Documents/ThisFile.txt`. In the engine, a SimGroup is essentially a folder instance, which is why it has the folder icon in the world editor. And objects stored within are actually pathed under a style as a folder map. This can be recognized by the **nameToID** function we've used before. So let me bring back a Chapter 3 image to jog the memory:

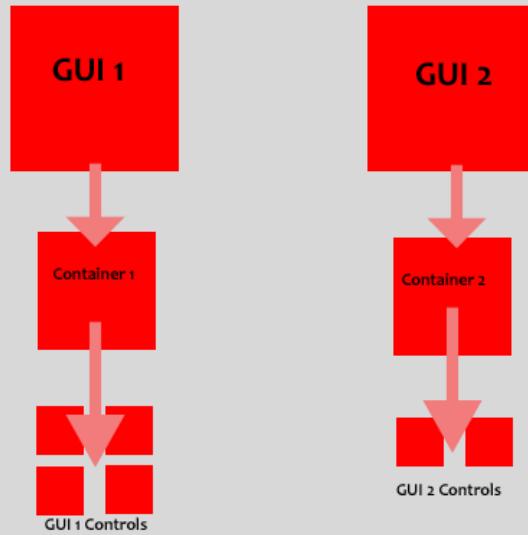


If you remember what we just got done with here, we had moved all of the objects that we spawned into SimGroup instances. Now, let's say I wanted to access the lightning object in the folder without directly just calling **LightningObject.blah()**, well you can do that by fetching it's ID in the path form:

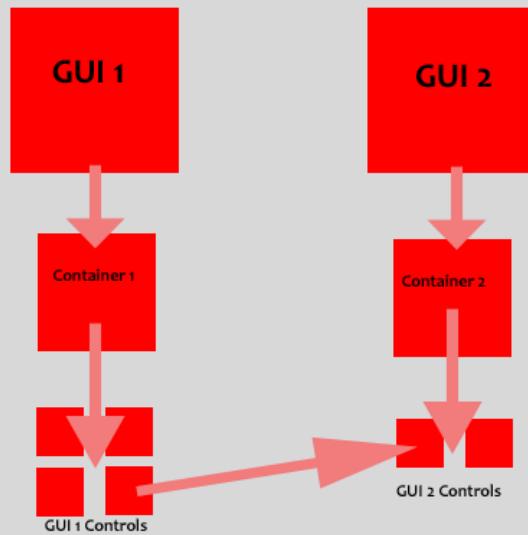
```
function fetchLightning() {
    %stormID = nameToID("MissionGroup/EnvironmentObjects/LightningObject");
    //Do Stuff
}
```

So you can see here that each SimGroup instance is mapped as a folder instance and when we reach the desired object within, we simply don't include another forward slash mark. Now the topic of using nameToID on SimGroup instances is irrelevant, but the concept here is not. What I'm getting at is each GUI is part of an **object tree** or basically one object with many children objects. Each object can be uniquely accessed and used regardless of its position in these tree instances.

If the concept of dialog to dialog wiring is still confusing, let's look at an image for an example of what I'm referring to. Let's assume we have two GUI control instances with a container setup and a assortment of controls stored inside each individual GUI instance:



The concept of wiring together two dialogs would be to do something in one GUI that has a direct effect to the other. So let's say by a button in GUI 1, we'd affect an image in GUI 2 like so:

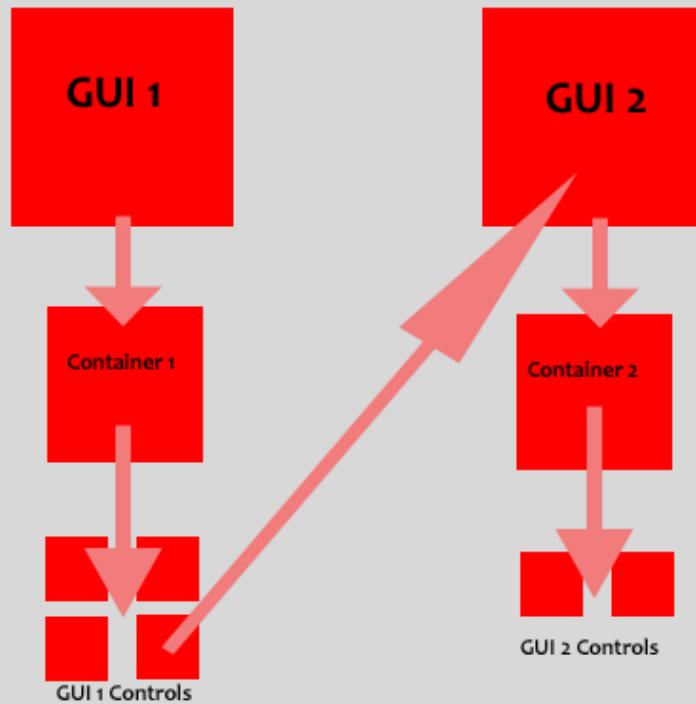


From a functional standpoint, you'd have the callback function of the GUI 1 control directly use a function from the GUI 2 control; this is a dialog to dialog wire. All you're doing here is a repeated example of Chapter 8's object functions and using the access operator to create the wiring points of the dialogs.

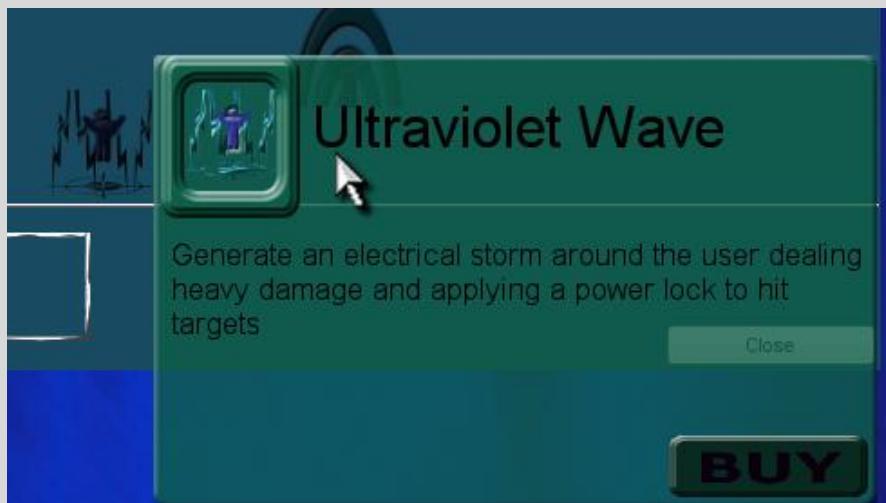
Wiring: Control to Dialog

The other case of GUI control wiring very frequently used is the concept of wiring an individual GUI control to a GUI dialog. This is used for things like hovering mouse controls, where you hover over one GUI control to spawn another dialog instance. To create a wired instance like so, you need to cleverly make use of the mouse event callbacks as provided to you earlier in this chapter, as well as

some mathematics to determine the correct position to spawn the new dialog instance. From a flow diagram process, this wire looks like the following:



Since this is a guide on the engine and scripting, I'll show you how to successfully create this type of a wire. For the first part, you'll need two dialogs, the primary dialog, and the dialog that will appear on the screen. Here's an example from a project of mine:



How it works is simple, the user moves their mouse over one of the image icons in the primary dialog, and after a second if the mouse is still there, the secondary dialog spawns. If the mouse leaves the icon but does not enter the secondary dialog, it disappears. Likewise, once the mouse enters the

second dialog, it enters an “active” state. If the mouse leaves the dialog when in the “active” state, the dialog disappears. So, we use a few mouse event functions to do this, like so:

```
function LegaciesBitmapButton::onMouseEnter(%this) {  
    $currentControl = %this;  
  
    if(!skillTreeDlg.storeLocked) {  
  
        //This is a power/sub-power button, push the item tooltip  
  
        if(!%this.isLocked) {  
  
            %this.hoverBuyTooltip = schedule(500, 0, pushStoreTooltip, %this);  
        }  
    }  
}  
  
function LegaciesBitmapButton::onMouseLeave(%this) {  
  
    $currentControl = "";  
  
    if(isEventPending(%this.hoverBuyTooltip)) {  
  
        cancel(%this.hoverBuyTooltip);  
    }  
}
```

So a few things to note about this function. I assigned a “lock” variable to detect the active state I mentioned above. This is also a very stripped down version of my actual code I used where the button’s internal name (`getName()`) is used to determine which control instance to spawn, I leave the tricky things like that to your own exercise, I also stored the control instance in a global variable, for the spawn function to use. Now, here’s how I handled spawning the dialog:

```
function StoreHoverTooltip::onWake(%this) {  
    skillTreeDlg.makeFirstResponder(true);  
  
    %this.sleepSched = Canvas.schedule(1500, popDialog, %this);  
}  
  
function StoreHoverTooltip::onSleep(%this) {  
    skillTreeDlg.makeFirstResponder(false);  
}  
  
//Mouse Events  
function StoreTooltipBox::onMouseEnter(%this) {  
    cancel(StoreHoverTooltip.sleepSched);  
  
    skillTreeDlg.storeLocked = true;  
}
```

```
function StoreTooltipBox::onMouseLeave(%this) {
    StoreHoverTooltip.sleepSched = Canvas.schedule(100, popDialog, StoreHoverTooltip);
    skillTreeDlg.storeLocked = false;
}

function pushStoreTooltip(%button) {
    %cursorPos = Canvas.getCursorPos();
    %screenPos = Canvas.getWindowPosition();
    %screenRes = Canvas.getExtent();
    %boxExtent = StoreTooltipBox.extent;
    //Secondary Items:
    %posAdd["ItemOfInterest"] = "18 19";
    %posAdd["BuyItemButton"] = "285 210";
    %posAdd["ItemTitle"] = "87 22";
    %posAdd["ItemDescription"] = "10 101";
    //Determine the best location to place the box on the screen
    %xCursor = getWord(%cursorPos, 0);
    %yCursor = getWord(%cursorPos, 1);
    %xSPos = getWord(%screenPos, 0);
    %ySPos = getWord(%screenPos, 1);
    %xScreen = getWord(%screenRes, 0);
    %yScreen = getWord(%screenRes, 1);
    %xBox = getWord(%boxExtent, 0);
    %yBox = getWord(%boxExtent, 1);
    //Some basic math.
    %xPos = (%xCursor-%xSPos) + 5;
    %yPos = (%yCursor-%ySPos) + 5;
    if((%xPos + %xBox) > (%xScreen)) {
        %xPos = %xScreen - (%xBox+5);
    }
    if((%yPos + %yBox) > (%yScreen)) {
        %yPos = %yScreen - (%yBox+5);
    }
    //Set the box position.
    StoreTooltipBox.position = %xPos SPC %yPos;
    ItemOfInterest.position = vectorAdd(StoreTooltipBox.position, %posAdd["ItemOfInterest"]);
    BuyItemButton.position = vectorAdd(StoreTooltipBox.position, %posAdd["BuyItemButton"]);
    ItemTitle.position = vectorAdd(StoreTooltipBox.position, %posAdd["ItemTitle"]);
}
```

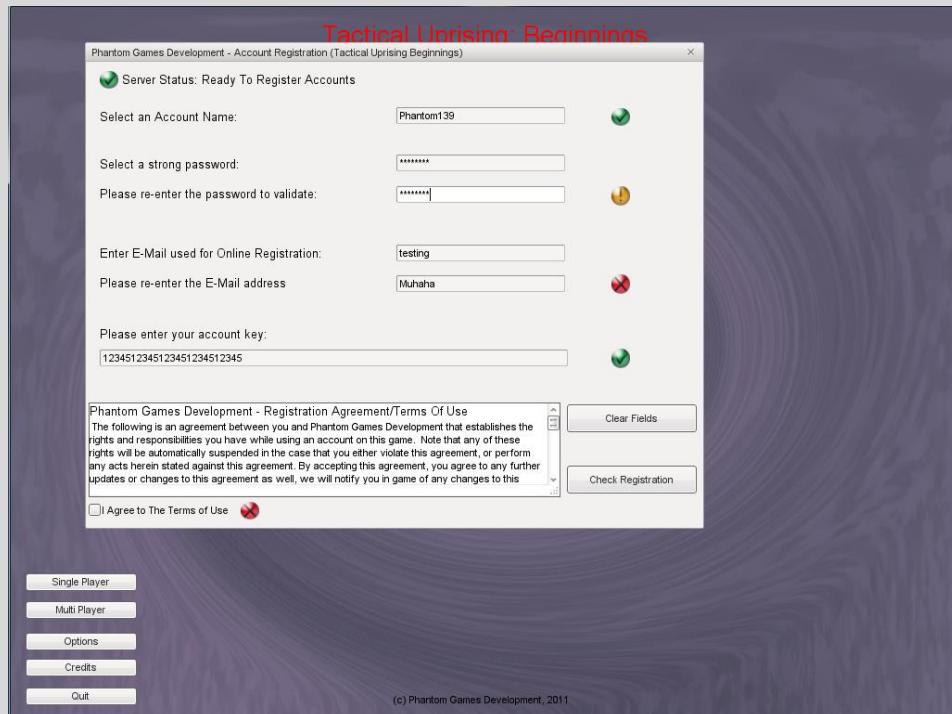
```
ItemDescription.position = vectorAdd(StoreTooltipBox.position, %posAdd["ItemDescription"]);  
//Set the internal dialog stuff here.  
// ... Omitted Code ..  
//And lastly push the dialog itself.  
Canvas.pushDialog(StoreHoverTooltip);  
}
```

So, this should hopefully teach you some new functions, and how to properly use the canvas size functions and adapt them into position based functions used to wire control to dialog type instances.

Concluding Notes on Wiring

GUI wiring is a complex, yet important part of creating user friendly dialogs that will ensure the players of your game or users of your application have an easy, stress-free time learning them and quickly learning to use them to make the most of their game experience. Most of the time, the wiring process will only take up a few lines of code, and other times (as shown above), they can take large amounts of code to create complex, but helpful dialogs.

The best piece of advice here is to keep things simple, or to a point where you and your friends can easily navigate the dialog without issues. Always place help dialogs, or tooltips on the more complicated things, oh, and things like this are nice too:



Final Remarks

So I hope you've learned a few new things with this short chapter on how to set up more advanced GUI dialogs. Be sure to remember that if you ever need help wiring a dialog together, to refer

to the engine's documentation on how each control works, and when the callbacks of the individual controls are used, that way you'll be sure to know that you're using the functions correctly, and that the dialogs will function as you have intended them to function.

Now we're going to take another step forward in terms of scripting, and move back into the object side and gameplay side of the engine and take our "advanced topics of 'x' in torquescript" to the in-game process.

Chapter 11: Advanced TorqueScript Topics: Entering the Game World

Chapter Introduction

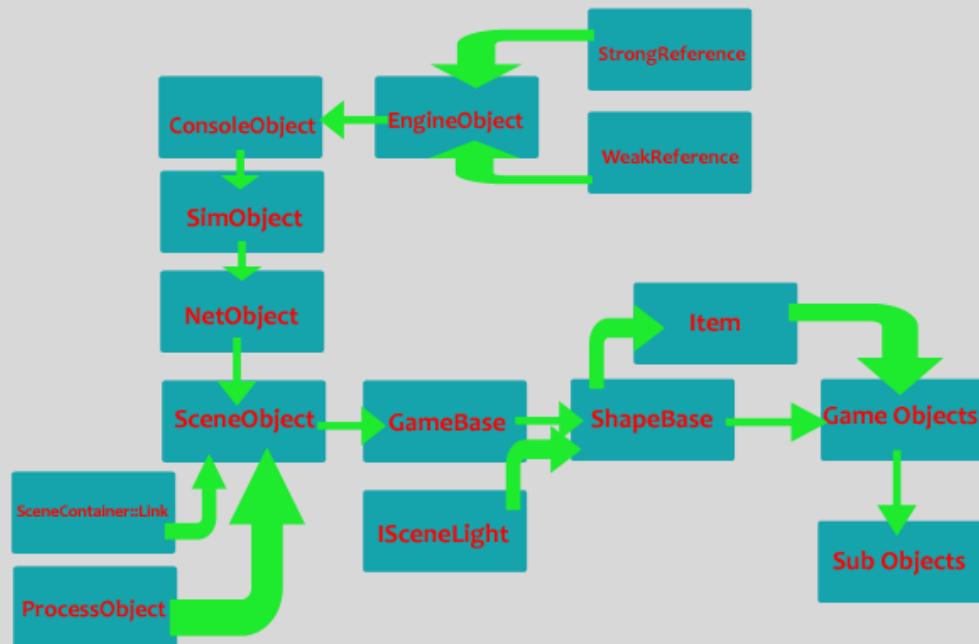
If chapter 10 was the engine scripting link to chapter 4, then chapter 11 must be the engine scripting link to chapter 3, and you'd be right to assume that notion. This chapter will bring together the concepts of Chapter 3, 6, 7, 8, and 9 into the core knowledge that is scripting for your world itself. This covers the topics of things like world events, scripted objects, and giving objects an actual purpose in your world. We'll be using concepts from all of the listed chapters above so make sure you've got those down and ready to use here. This chapter has numerous topics to discuss and there is a lot of content here so make sure to absorb what you can from here, because this is where you'll actually put the important pieces of your game together.

Game Objects, Revisiting Objects

Our first stop in this chapter is to introduce the in-game object itself as it pertains to the engine. The engine itself is massive in nature and has hundreds of object instances in the engine, each with its own importance and functioning, but what's even more complex in nature is how these object instances are linked together to form the engine's inheritance tree. This section is mainly going to be a quick review of chapter 8 as well as a few new things just so that you're ready to go for this chapter's new content.

Object Inheritance Tree

You'll see this diagram later, and I'll go much more into detail regarding it at this point, but for the time being, you should know the inheritance tree of **all** game objects in the engine, or in terms we've seen before, what classes inherit the functions of other classes:



If you remember back in Chapter 8 when we discussed object and class inheritance, this is where it plays a key role for this chapter. The tree here shows you what Game Objects have access to when they're being used in the engine. For the actual purposes of this current chapter, we're only interested in a few pieces of this tree, specifically **SimObject**, **SceneObject**, **GameBase**, and **ShapeBase**. The classes of the engine related to objects in the game are all derivations of these four classes. Even in the prior chapter when we covered the GUI stuff, all of the GUI controls are derivations of the **SimObject** class. Likewise with the GUI controls there are hundreds upon hundreds of active functions available to your objects in the engine, and infinitely larger possibilities by creating your own function instances. The scope of the guide would be way too massive to write a detailed description of how each function works in the engine, and this chapter would likely exceed a thousand pages to do so, just to put things into perspective, so we're not going to go there.

At this point in time however, you should have a full understanding of how to write object functions anyways and be familiar with the concepts taught back in the prior five chapters, as you'll likely be using most, if not all of them at some point during your in-game programming experiences. The final take-away point before we dive further is that in-game programming is a very time consuming process, and it takes a lot of practice and experience of learning by process and mistakes to actually get good at it, so don't be intimidated by exceptionally large code files, or if a problem seems too complex to solve, you'll be able to get it with trial and error, and by applying the numerous practices we covered before.

Revisiting Datablocks

Another topic we talked about in chapter 8 was the topic of datablocks, and even earlier than that in chapter 6, I gave you the large list of datablock types to use in the engine. You'll get used to working with datablocks very frequently because all in-game objects that are influenced by players in some way or form have a parented datablock. The obvious exclusions here are mission direct objects that are "static", or do not change over time, they do not require a datablock object. We'll talk a little more about datablocks in chapter 12, but be aware that they are persistent engine objects, or basically an object whose properties cannot be changed by an individual client, but instead holds a server controlled precedence over all objects that use the block.

The most common applications of datablocks are to define objects such as players, vehicles, weapons, projectiles, items, and sounds. All of the properties of these individual object classes can be viewed through the dump command on the datablock instance, or by means of reading the engine's reference documentation. Also remember that the engine will only allow you to have around 2048 datablocks at one time, and that no two datablock instances can share the same name with each other or with any other object instances in the engine.

Object Functions Revisited

The last stop on the review segment of this chapter will be a quick refresher on object functions and their respective syntax. Recall from chapter 6, I presented the function definition syntax as follows:

```
function ObjectType::FunctionName(%this, %arg1, %arg2, ..., %argN) {  
    //Code Block  
}
```

Where the variable **%this** was a local reference to the object being sent to the function and then that you could define any number of parameters for the object. Also, recall from the chapter 8 section on inheritance and the parent keyword that you could access the super-class functioning of the class instance by means of:

```
function initMathDemoObject() {  
    %mObj = new ScriptObject() {  
        class = "MathObject";  
    };  
  
    %mObj.num1 = getRandom(1, 1000);  
    %mObj.num2 = getRandom(1, 1000);  
  
    return %mObj;  
}  
  
function MathObject::onAdd(%this) {  
    parent::onAdd(%this);  
  
    echo("Adding a MathObject. Num1: @"%this.num1@"", Num2: @"%this.num2");  
}
```

This would keep the original functioning of the **onAdd** function as defined by the **ScriptObject** class in our instance here.

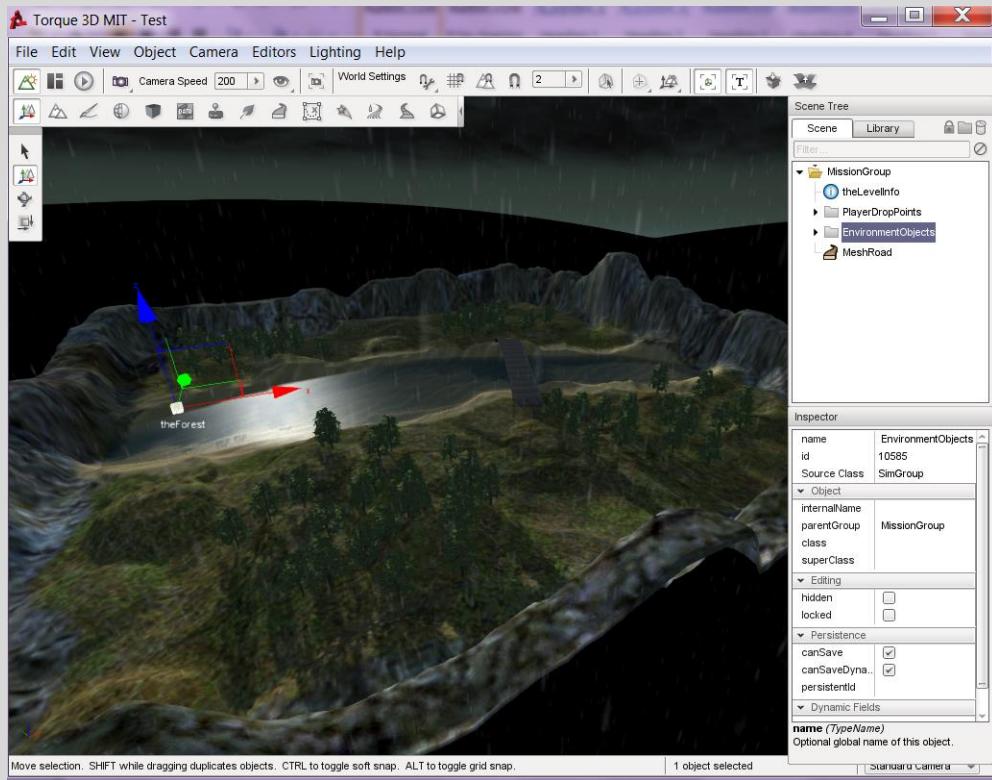
I'm hoping this review segment here is enough to jog your memory just enough so that you're remembering the main concepts from earlier. If not, please go back to the prior chapters and look over the necessary sections again to ensure you're ready for the new content here.

Scripted Events, and Enhancing our Map

Our first stop in the topics of new is how to enhance your worlds with something called a **scripted event**. A scripted event is basically a programmer defined event that happens in the game world when a certain action is performed by a player. This can range from pressing a button, or doing something like stepping into a new combat zone. We'll mainly be focusing on the second case with this sub-chapter, so let's get started.

The Trigger Object

So, way back in chapter 3 we walked through a start to finish process on some very basic level design tips, tricks, and techniques, and then we actually went about creating a brand new level for our players to walk through. If you remember, it looked a bit like this:



Now the time has come to upgrade our little mission editor experiment with some brand new scripted events to make the level much more interesting in nature. To do this, we're going to take advantage of a special object class that exists in the engine called the **Trigger**. The trigger object is a special object in the engine used to detect when an object enters and exits a pre-defined space. From the standpoint of the editor, it's a simple box object with some parameters to size the box and move it around, but the real importance of this object actually lies in the scripted side of things. The trigger relies on three callback functions, which are defined as so:

```
function triggerClass::onEnterTrigger(%this, %triggerObj, %objEntering) {
    //Do code here
}

function triggerClass::onLeaveTrigger(%this, %triggerObj, %objLeaving) {
    //Do code here
}

function triggerClass::onTickTrigger (%this, %triggerObj) {
    //Code called every %triggerObj.tickPeriodMS.
```

```
//Asset functions for getting objects: %triggerObj.getNumObjects() & %triggerObj.getObject(%i)  
}
```

With these functions at your disposal you can use a trigger object in your mission to cause an event to happen when a player enters or leaves a trigger. And by event, this means you call a function that causes other things to happen in your world instance.

When & Where to use This

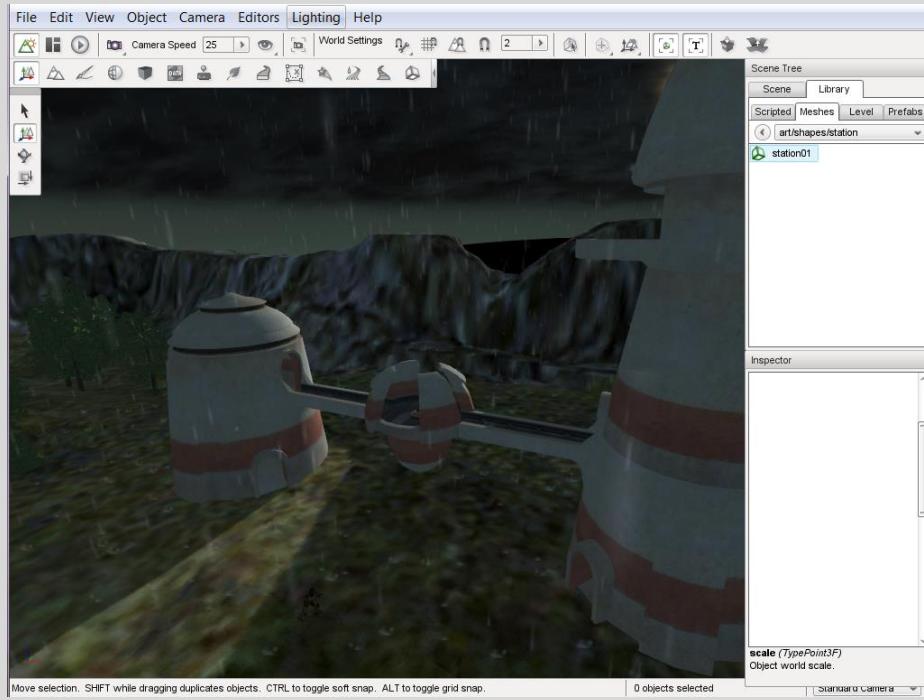
So the big question is then, when and where do I actually use this functioning in my game? One of the biggest examples of using a trigger to cause a scripted event is in something along the lines of a singleplayer mission instance. When the player enters a certain area, you may want to cause something to happen, like triggering an explosion, or spawning a large group of enemies in the general area. The trick here however, is to make sure that the event is only triggered for the one time, and you can either make it per-player based, or per-trigger based, here's how:

```
function triggerClass::onEnterTrigger(%this, %triggerObj, %objEntering) {  
  
    //Per-Player Based.  
  
    if(!%objEntering.hasTriggered[%triggerObj]) {  
  
        //Do the event.  
  
        %objEntering.hasTriggered[%triggerObj] = true;  
  
    }  
  
    //Per-Trigger Based.  
  
    if(!%triggerObj.hasTriggered) {  
  
        //Do the event.  
  
        %triggerObj.hasTriggered = true;  
  
    }  
}
```

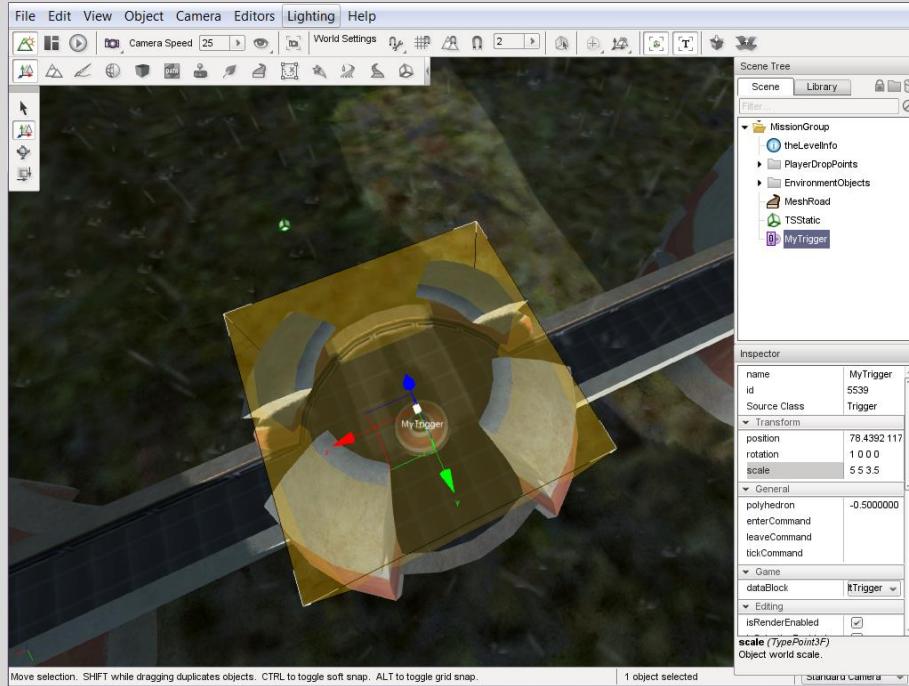
Make sure to obviously use one or the other, unless you need the event to be triggered once for each player, and then once for the first time anyone enters it, but you should be able to pick apart the logic here by now.

Proper Trigger Placement

About the trickiest part of the trigger object setup here, is that you need to ensure that the trigger object is placed in the correct location to prevent it from being passed by your players (unless this behavior is intended). So, let's assume for a moment that back in my example map in chapter 3, I have placed a station object on the map, and I want to trigger an event when someone touches the middle of the station:



So in the editor, under the level tab of the library, you'll see the Trigger object, so what I do is I place the object in the center of the middle piece, and then scale it so it fits over the entire part of the map I want to be affected by this trigger:



The thing to note here is that the object datablock is the **defaultTrigger** datablock, so it will base the functioning from that class. You'd likely want to write your own datablock for your trigger classes. Just remember the discussion about the limit on datablocks. If you're good enough, you'd be able to get

away with one datablock for your entire game instance. As for a little “hint” on how to do this, just remember that lovely **eval** command from chapter 10, and how objects can have any parameter associated to them you want them to have. Just be careful when you place your trigger objects, make sure they cover the entire area you want to be affected, and that they also do not touch places that should not trigger the event, you may need to make use of rotation and scaling of the map to do this, also use the terrain to your advantage as well to accomplish the affect desired. If the trigger cannot be safely applied there still, use the polyhedron variable to change the shape of the trigger from a rectangle to another shape. You’ll need some mathematics to accomplish this.

Either way, there are many opportunities to use the trigger object to bring some very unique gameplay experiences to your worlds, and I hope you use them whenever, and wherever you can find a use for them, they’re there to make your world feel more alive, which is the ultimate point of merging scripts with the world editor!

So now that we’ve touched the waters of combining the two instances together, let’s take our first leap into the waters and actually learn how to make objects for the game in script, and then use them in the world.

Players

Whether you’re creating the next first person shooter blockbuster hit title, the next role playing masterpiece, or just any general game that involves a character, you’ll likely be using the **Player** class to accomplish a great deal of your work. The player class basically defines a humanoid instance object that binds player input to movement controls to navigate your worlds created in the world editor. There are many properties available in the player’s datablock instance to change individual parameters of the player object. So let’s start by talking about the player datablock.

The Player Datablock

You should have full knowledge by now of how to actually set up the datablock for an object. If you’re a little lacking on the topic, refer to the datablocks section in Chapter 8, or even better, just open up the datablock file for a player and learn by example. This section will teach you the available fields to the player datablock so you know how to use them and what type of value to set it to. Also remember that the Player class directly inherits from the ShapeBase class, so it will also have all of the datablock properties of the ShapeBase class available. See the documentation for reference on those fields.

Field Name	Value Type	Description
pickupRadius	Floating Point Number	The radius around a player in which item pickups are allowed.
maxTimeScale	Floating Point Number	The maximum scale of time in which player animations can be done
renderFirstPerson	Boolean	A flag to tell the engine if it needs to render the player object in first person mode
firstPersonShadows	Boolean	A flag to tell the engine if it needs

		to render shadows in first person mode.
minLookAngle	Floating Point Number	The lowest angle in which a player may look (radians)
maxLookAngle	Floating Point Number	The highest angle in which a player may look (radians)
maxFreelookAngle	Floating Point Number	The highest left and right angles in which a player may look to in freelook mode (radians)
maxStepHeight	Floating Point Number	The maximum height in which a player may “step up” by colliding with a vertical stair.
runForce	Floating Point Number	The force used to accelerate the player object when running.
runEnergyDrain	Floating Point Number	The energy value drained each tick that a player is moving. If this is not 0 and the energy runs out, the player cannot move.
minRunEnergy	Floating Point Number	The minimum energy required for a player to run.
maxForwardSpeed	Floating Point Number	The maximum speed a player can run when moving forward.
maxBackwardSpeed	Floating Point Number	The maximum speed a player can run when moving backwards.
maxSideSpeed	Floating Point Number	The maximum speed a player can run when strafing to the left and right.
runSurfaceAngle	Floating Point Number	The maximum angle from vertical in which a player can run (degrees).
minImpactSpeed	Floating Point Number	The minimum speed needed to apply impact damage to the ground.
minLateralImpactSpeed	Floating Point Number	The minimum speed needed to apply impact damage to lateral surfaces.
horizMaxSpeed	Floating Point Number	The maximum speed a player may move horizontally
horizResistSpeed	Floating Point Number	The speed in which resistance to horizontal movement will take place.
horizResistFactor	Floating Point Number	The factor of horizontal resistance to apply once horizResistSpeed has been exceeded.
upMaxSpeed	Floating Point Number	The maximum speed a player may move vertically

upResistSpeed	Floating Point Number	The speed in which resistance to vertical movement will take place.
upResistFactor	Floating Point Number	The factor of vertical resistance to apply once upResistSpeed has been exceeded.
jumpForce	Floating Point Number	The force in which a player is accelerated by jumping.
jumpEnergyDrain	Floating Point Number	The amount of energy that is removed by jumping.
minJumpEnergy	Floating Point Number	The minimum amount of energy needed to perform a jump.
minJumpSpeed	Floating Point Number	The minimum speed a player needs to be moving in order to jump.
maxJumpSpeed	Floating Point Number	The maximum speed a player can be moving to jump.
jumpSurfaceAngle	Floating Point Number	The maximum angle from vertical in which a player is allowed to jump from (degrees).
jumpDelay	Integer	The amount of ticks before the jump is done from when the signal is received.
airControl	Floating Point Number	The amount of movement control a player has in the air as a multiplier of horizontal movement.
jumpTowardsNormal	Boolean	A special flag where false results in always +Z jumps and true results in jumps towards the direction of the ground normal as long as the player is facing the surface.
sprintForce	Floating Point Number	The amount of force applied to a player when sprinting.
sprintEnergyDrain	Floating Point Number	The amount of energy drained by sprinting.
minSprintEnergy	Floating Point Number	The minimum amount of energy needed to sprint.
maxSprintForwardSpeed	Floating Point Number	The maximum speed a player may sprint when moving forward.
maxSprintBackwardSpeed	Floating Point Number	The maximum speed a player may sprint when moving backwards.
maxSprintSideSpeed	Floating Point Number	The maximum speed a player may sprint when strafing left and right.
sprintStrafeScale	Floating Point Number	The amount to scale the strafing vector by when sprinting.
sprintYawScale	Floating Point Number	The amount to scale yaw

		(up/down) movements while sprinting.
sprintPitchScale	Floating Point Number	The amount to scale pitch (left/right) movements while sprinting.
sprintCanJump	Boolean	A flag to indicate whether or not a player may jump while sprinting.
swimForce	Floating Point Number	The amount of force to accelerate a player when swimming.
maxUnderwaterForwardSpeed	Floating Point Number	The maximum speed a player may swim forwards.
maxUnderwaterBackwardSpeed	Floating Point Number	The maximum speed a player may swim backwards.
maxUnderwaterSideSpeed	Floating Point Number	The maximum speed a player may swim left and right.
crouchForce	Floating Point Number	The amount of force to accelerate a player to move when crouching.
maxCrouchForwardSpeed	Floating Point Number	The maximum speed a player may move forwards while crouching.
maxCrouchBackwardSpeed	Floating Point Number	The maximum speed a player may move backwards while crouching.
maxCrouchSideSpeed	Floating Point Number	The maximum speed a player may move left and right while crouching.
proneForce	Floating Point Number	The amount of force to accelerate a player to move when in prone.
maxProneForwardSpeed	Floating Point Number	The maximum speed a player may move forwards while in prone.
maxProneBackwardSpeed	Floating Point Number	The maximum speed a player may move backwards while in prone.
maxProneSideSpeed	Floating Point Number	The maximum speed a player may move left and right while in prone.
jetJumpForce	Floating Point Number	The amount of force to accelerate a player upwards when they use the jet.
jetJumpEnergyDrain	Floating Point Number	The amount of energy to drain from the player when they jet.
jetMinJumpEnergy	Floating Point Number	The minimum amount of energy a player needs in order to jet.
jetMinJumpSpeed	Floating Point Number	The minimum speed a player needs to be moving in order to jet.
jetMaxJumpSpeed	Floating Point Number	The maximum upwards speed a player may obtain by jetting.
jetJumpSurfaceAngle	Floating Point Number	The maximum vertical angle in which a player may initiate a jet jump (degrees).

fallingSpeedThreshold	Floating Point Number	The downward speed in which a player is considered to be falling.
recoverDelay	Integer	The amount of time needed (in ticks) for a player to recover from a fall.
recoverRunForceScale	Floating Point Number	The scale to apply to player movement while recovering from a fall.
landSequenceTime	Floating Point Number	The time of the landing sequence to play when recovering from a fall.
transitionToLand	Boolean	When switching between falling and landing, should the engine apply a transition effect?
boundingBox	3-Point Value ("x y z")	The size of the box (from center) to use for the player's collisions.
crouchBoundingBox	3-Point Value ("x y z")	The size of the box (from center) to use for the player's collisions while crouching.
proneBoundingBox	3-Point Value ("x y z")	The size of the box (from center) to use for the player's collisions while in prone.
swimBoundingBox	3-Point Value ("x y z")	The size of the box (from center) to use for the player's collisions while swimming.
boxHeadPercentage	Floating Point Number	The percentage of the player's bounding box (at the top) that is represented as the "head" of a player, this is mainly used for damage calculations.
boxTorsoPercentage	Floating Point Number	The percentage of the player's bounding box (at the middle) that is represented as the "torso" of a player.
boxHeadLeftPercentage	Floating Point Number	The percentage of the player's bounding box from the head location to the left counted as the head of a player.
boxHeadRightPercentage	Floating Point Number	The percentage of the player's bounding box from the head location to the right counted as the head of a player.
boxHeadFrontPercentage	Floating Point Number	The percentage of the player's bounding box from the head location to the front counted as the head of a player.
boxHeadBackPercentage	Floating Point Number	The percentage of the player's

		bounding box from the head location to the back counted as the head of a player.
footPuffEmitter	Emitter Datablock	The particle emitter to use when a player walks along the ground.
footPuffNumParts	Integer	The amount of particles to create with each step.
footPuffRadius	Floating Point Number	The radius to use for the foot puff emitter.
dustEmitter	Emitter Datablock	The particle emitter to use to create dust particles when the player moves (Be aware that this feature is disabled in T3D 3.5.1)
decalData	Decal Datablock	The datablock of the decal to place on the ground where a player walks.
decalOffset	Floating Point Number	The distance from the center of the model to the right foot, used to calculate where the decals are placed.
footSoftSound	String (SFX File Path)	The sound to play when a player walks on a “soft” surface (material footstepSoundID is set to 0)
footHardSound	String (SFX File Path)	The sound to play when a player walks on a “hard” surface (material footstepSoundID is set to 1)
footMetalSound	String (SFX File Path)	The sound to play when a player walks on a “metal” surface (material footstepSoundID is set to 2)
footSnowSound	String (SFX File Path)	The sound to play when a player walks on a “snow covered” surface (material footstepSoundID is set to 3)
footShallowSound	String (SFX File Path)	The sound to play when a player walks in water where the coverage is less than the footstepSplashHeight variable
footWadingSound	String (SFX File Path)	The sound to play when a player walks in water when the coverage is greater than the footstepSplashHeight variable, but not completely covered (1.0).
footUnderwaterSound	String (SFX File Path)	The sound to play when a player walks in water and is completely covered by water (1.0).

footBubblesSound	String (SFX File Path)	Additional sound to play along with footUnderwaterSound
movingBubblesSound	String (SFX File Path)	This sound plays alongside the above two, and even if the player isn't moving.
waterBreathSound	String (SFX File Path)	This sound plays along with movingBubblesSound.
impactSoftSound	String (SFX File Path)	The sound to play when a player takes impact damage from a "soft" surface (material footstepSoundID is set to 0).
impactHardSound	String (SFX File Path)	The sound to play when a player takes impact damage from a "hard" surface (material footstepSoundID is set to 1).
impactMetalSound	String (SFX File Path)	The sound to play when a player takes impact damage from a "metal" surface (material footstepSoundID is set to 2).
impactSnowSound	String (SFX File Path)	The sound to play when a player takes impact damage from a "snow covered" surface (material footstepSoundID is set to 3).
impactWaterEasy	String (SFX File Path)	The sound to play when a player strikes water with a speed less than mediumSplashSoundVelocity
impactWaterMedium	String (SFX File Path)	The sound to play when a player strikes water with a speed greater than mediumSplashSoundVelocity but less than hardSplashSoundVelocity
impactWaterHard	String (SFX File Path)	The sound to play when a player strikes water with a speed greater than or equal to hardSplashSoundVelocity
exitingWater	String (SFX File Path)	The sound to play when the player object leaves water moving at a speed greater than or equal to exitSplashSoundVelocity.
splash	Splash Datablock	The datablock of a splash effect to create when a player enters water.
splashVelocity	Floating Point Number	The minimum speed a player needs to enter water at in order to create a splash.
splashAngle	Floating Point Number	The maximum angle in which a player may enter water from to create a splash (degrees).

splashFreqMod	Floating Point Number	A modifier of speed that is used to determine how many particles are spawned by the splash.
splashVelEpsilon	Floating Point Number	The minimum speed a player needs to enter water to create splash particles.
bubbleEmitTime	Floating Point Number	The amount of time in seconds until bubble particles are created once the player enters water.
splashEmitter	Emitter Datablock	The particle emitter that is spawned by splashing when the thresholds above a broken.
footstepSplashHeight	Floating Point Number	The height used to determine the footstep sounds when walking in water.
mediumSplashSoundVelocity	Floating Point Number	The speed used to determine if the player should create an easy or medium entry sound into water.
hardSplashSoundVelocity	Floating Point Number	The speed used to determine if the player should use a medium or a hard entry sound into water.
exitSplashSoundVelocity	Floating Point Number	The minimum speed for existing water in order to trigger the sound for leaving the water.
groundImpactMinSpeed	Floating Point Number	The minimum speed a player needs to strike the terrain in order to be considered an impact.
groundImpactShakeFeq	3-Point Value ("x y z")	The speed modifier of the camera shake to apply to the player that takes in impact with the ground.
groundImpactShakeAmp	3-Point Value ("x y z")	The amount of camera shake to apply for impacting the ground.
groundImpactShakeDuration	Floating Point Number	The amount of time to shake the camera due to a ground impact in seconds.
groundImpactShakeFalloff	Floating Point Number	The factor of the amount of reduction as the time of shaking approaches groundImpactShakeDuration.
physicsPlayerType	String	The type of physics module used by the Player Object. (Be aware that this feature is disabled in T3D 3.5.1)
imageAnimPrefixFP	String	Optional prefix to attach to all mounted images and their animation sequences when in first

shapeNameFP (Array)	String (File Name)	person (This by default is FP_)
imageAnimPrefix	String	Optional prefix to attach to all mounted images and their animation sequences when in third person.
allowImageStateAnimation	Boolean	Allow images to request a separate animation sequence beyond that of the player's sequence.

So as you can see here, there's a ton of fields for a Player object to use. You should take the time to actually comb through this and pick out what you need and what you don't need. Most player instances only use about half of these fields, however the engine's template provides you with an example of how all of these fields are used by default. Just remember the rules of datablocks and how to use them when it comes to making a player datablock instance. Here's a sample of a player datablock instance (without the fields):

```
datablock PlayerData(DefaultPlayerData) {
    //Fields
}
```

Now that you know how to set up a player datablock instance as well as the fields that you have access to for your player datablocks, let's actually talk about spawning a player instance and the kind of things you can do with a player object in the engine.

Spawning a Player Object

If you go back to Chapter 8, I talked about how you create objects in the engine, specifically how to do it with the engine's scripting language. Those points hold almost identically true here, where the overall syntax of spawning a player instance would be done the same way as almost any other object in the engine, however there are a few parameters that you should use when spawning players. I'll give the most generic form here:

```
%player = new Player() {
    datablock = "DefaultPlayerData";
};
```

Doing this alone will spawn a base player instance under the **DefaultPlayerData** datablock, or basically the player object instance will inherit the properties of the datablock instance. However, if you used this alone, there's a very good possibility you'd never see the player instance since you haven't told it where it needs to spawn at (Leading to a spawn at 0 0 0). There's two ways to do this, the first is to apply a **position** parameter to the object instance when you spawn it, and the other is to use the **setTransform** command after you spawn the player object.

Let's look at both cases here to help you spawn a player instance in the correct location; we'll start with the parameter example:

```
function spawnAPlayer(%position) {  
    %player = new Player() {  
        datablock = "DefaultPlayerData";  
        position = %position;  
    };  
    MissionCleanup.add(%player); //<- Game objects belong in MissionGroup/MissionCleanup.  
    return %player;  
}
```

Most objects that appear in the game instance can be treated with this logic, where you can define a position parameter to specify where you want the object to spawn in the world. Also, take note of the line where the player instance is added to the **MissionCleanup** group. This is a special SimGroup instance where game objects that need to be deleted when the mission ends goes. The **MissionGroup** SimGroup also behaves with this same logic.

Now let's look at the other case, where we spawn the player and then move it into position using the **setTransform** command:

```
function spawnAPlayer2(%position, %rotation) {  
    if(%rotation $= "") {  
        %rotation = "0 0 0 0";  
    }  
    %player = new Player() {  
        datablock = "DefaultPlayerData";  
    };  
    %player.setTransform(%position SPC %rotation);  
    MissionCleanup.add(%player); //<- Game objects belong in MissionGroup/MissionCleanup.  
    return %player;  
}
```

So just a bit more to cover here; remember from chapter 9 that a **Transformation Matrix** consists of both a position and a rotation factor, so we covered that in the **setTransform** function by splitting the variables. Also, you don't need to provide a rotation variable, we'll set it to zero rotation if you don't send one in this case.

Either way you choose to spawn your player objects, this is how you do it, and these examples will set the baseline for spawning other types of objects in your game's scripts, so you'll see things like this later on in this chapter. Now let's talk about the functions and callbacks available to your player object instances.

Functions & Callbacks for Player

Like all of the prior object classes in the engine we've worked with, the player class has a set of pre-defined functions and callbacks that allow you to do some pretty interesting things with them. Also, be mindful of the chart I presented at the beginning of this chapter, and realize that functions and callbacks of the derivation classes of Player will also be included in the Player class.

I'll talk about a set of commonly used functions first and then we'll get into the commonly used callbacks as well. Remember, there are many more functions available to choose from, but they're not listed here due to the amount of them. Refer to the engine's documentation if you're interested in learning the other functions.

getPose

The **getPose** function is used on an individual player object instance to return a string containing the current "pose" of the object. This will either be: "Stand", "Sprint", "Crouch", "Prone", or "Swim". There are no variables on this function:

```
function findPoseOf(%player) {  
    return %player.getPose();  
}
```

getState

The **getState** function is a very useful function to return a string containing the current status of a player as it relates to the world simulation. This is mainly used to handle things like telling if the player is alive or dead, or recovering from a fall. There are four possible states that can be returned by this object function: "Dead", "Mounted", "Move", "Recover". You don't need to send any parameters to the function, and it only returns a string containing the result.

setControlObject

The **setControlObject** function is used to pass the control movements of the player object through the player and down to a secondary object, for instance, a vehicle when the player mounts to it. To use this function, you pass an object reference to the function and then when the player object is issued a move command it will pass through the player and down to the object you specified here.

```
function makeObjectMine(%player, %target) {  
    %player.setControlObject(%target);  
    echo("Player '@%player@" now controls "@%target");  
}
```

getControlObject

The **getControlObject** function is used in conjunction with the **setControlObject** function. When the player instance is currently controlling another object instance, this function will return an object reference of the object that is being controlled by the specified player instance. There are no parameters for this specific function instance.

clearControlObject

The **clearControlObject** function is used to remove control from an object and give it back to the player that was controlling it. When you want your player instance to stop controlling an object and re-assume control of the player object, you can either use this command on the player, or pass the player reference object to the **setControlObject** function. There are no parameters for the **clearControlObject** function.

So those are a few of the functions that player objects will commonly be using in your game instances. Remember, there are many more available due to the derivation structure of the engine, and that you should refer to references such as the engine documentation or the object's dump command to view the available functioning for player objects.

Next up, let's look at some of the common player object callbacks to use. Unlike the other class instances you've worked with, the Player has its callbacks defined on the datablock instance to be used for per-class instancing. So when you work with these callbacks you'll define them on the datablock, which for all of our examples so far, has been **DefaultPlayerData**. You can also extend the notion to all player datablocks by using **PlayerData**. Now let's look at the callbacks.

onPoseChange

The first callback we're going to look at is the **onPoseChange** callback which is initiated when a player switches between one of the five poses mentioned back when I introduced the **getPose** function. Like all of these callbacks the first variable is the datablock, and then the second is the player object itself. Let's look at the format, and then I'll introduce this callback:

```
function PlayerData::onCallbackX(%this, %player, ..., %argN) {  
    //Do Something.  
}
```

So all of the callback functions for the player class will use this format to define and be used in the engine's scripting language. Now, let's look at the **onPoseChange** definition:

```
function PlayerData::onPoseChange(%this, %player, %oldPose, %newPose) {  
    echo("Player \"%@%player@\" is changing from \"%@%oldPose@\" to \"%@%newPose@\"");  
}
```

doDismount

This is one of those rare callback functions that don't specifically use the word "on" in the beginning, but it has an important role. When a player object dismounts from any object instance, this callback is triggered, however it's usually wise to ignore this one as it only uses the player instance when being fired off, and does not reference the object that is being dismounted from. There are no extra arguments for this callback function, only the datablock **%this** and the player variable.

onEnterLiquid

This callback is fired off when a player enters a liquid surface (Water, Lava, Etc) and is a scripted callback of some common functioning that the engine uses internally. This callback has two extra

parameters aside from the datablock and the player object. The first parameter is a coverage variable with a numerical range of 0.0 to 1.0, where 0.0 is no coverage and 1.0 is complete submersion. The second parameter is a string containing the type of the liquid the player is entering.

onLeaveLiquid

This callback is fired when a player exits any liquid surface. Like **onEnterLiquid** it has an internal callback as well that is handled by C++, and this is the scripted entry to the setting if you're going to need it for a player object. There is one extra parameter and it is a string containing the type of the liquid the player is exiting.

onEnterMissionArea

This callback is fired on a player instance when it enters the **MissionArea** object. If there is no object instance of this type, the callback won't be triggered. This callback is used in conjunction with the **onLeaveMissionArea** callback and it has no extra parameters aside from the two we've been talking about for the duration of this section.

onLeaveMissionArea

This callback is fired on a player instance when it leaves the **MissionArea** object. If there is no object instance of this type, the callback won't be triggered. This callback is used in conjunction with the **onEnterMissionArea** callback and it has no extra parameters aside from the two we've been talking about for the duration of this section.

And that covers everything we need to talk about for the Player class. This is a class you'll be using a lot in most of your game applications (unless you're making a Racing Game, or a Camera World Instance) so make sure you know how to use it. If at any time you get lost or cannot find the function of interest, review this chapter, or refer to the engine's documentation to see what is available to you. Now, let's introduce another important class.

Items

The next class I want to introduce to you is the **Item** class. **Item** is a highly versatile class that has many object derivations and references. The **Item** class defines basic collision and rendering properties along with interaction functions between players and the item class itself. This class is mainly used to define things like object pickups, but is so versatile that it is used to create objects like the **AITurret** class instance.

You will likely use this object class to create various item pickups for your player in a game, or even use it to create basic renderable & collidable shape instances for your players to navigate though in your world instance.

The Item Datablock

Like we did with the player class, let's start by exploring the datablock of the **Item** class. You'll find that this one's much less in-depth because it mainly serves as a template class to the other classes that are derivations of **Item**. But also be mindful, that like Player, this class inherits a boat load of other properties from the structured tree above **Item**.

Field Name	Value Type	Description
friction	Floating Point Number	A factor of how much movement is lost to friction against surfaces when moving.
elasticity	Floating Point Number	A factor of how “elastic” or how much the object “bounces” upon impact with surfaces.
sticky	Boolean	A flag that says if the Item instance will “Stick” to StaticShape instances when it collides with them.
gravityMod	Floating Point Number	A factor of how much gravity affects this object. 1.0 is normal gravity.
maxVelocity	Floating Point Number	A cap on the speed this object is allowed to move through the world.
lightType	Enumerated Value	What type of light is emitted by this object instance (Values: NoLight , ConstantLight , PulsingLight)
lightColor	3-Point Value (“r g b”)	What color needs to be emitted from this object instance (only if lightType is not NoLight)
lightTime	Integer	How frequent the pulses of light are if lightType is set to PulsingLight.
lightRadius	Floating Point Number	How much radius the light from this object needs to cover.
lightOnlyStatic	Boolean	If this is set to true, the light will only emit if the item instance is static (IE: Not Moving)
simpleServerCollision	Boolean	This is a special setting that should be set to false for item pickup instances, and true to world-object derivations that require advanced collisions (Projectile Hits, Etc).

So as you can see this time around, there isn’t as much that needs to be covered, but there’s still plenty of settings here that are important to the object instance itself. There are plenty of other fields out there that can be used, and you should refer to the engine’s documentation, or the datablock editor to get the full list of these fields as well as a description of the field. Defining an Item datablock is also very easy to do, like the Player class:

```
datablock ItemData(MyItemDatablock {  
    //Fields  
}
```

Since you should now be able to spawn objects of any type without needing another example (Refer to the Player section if you need more help on this) on how to properly place it in the world, let's talk about some of the functions that the Item class has for your use.

Functions and Callbacks for Item

The **Item** class doesn't define many methods and callbacks, because it's mainly a template class definition where you write your own functions for it and build off of the existing code. We'll talk about the important ones here.

isStatic

The first function we're going to introduce is a boolean check function called **isStatic**. This will return true if the object has the static flag enabled, or the object is defined to be stationary. There are no special parameters on this function and it will return true or false.

isAtRest

Next up is the **isAtRest** function which also returns true or false if the object is not moving, however this function also applies to non-static item instances, or basically item instances that may move, but are possibly not moving at the time of the function call. This function has no additional parameters and will return true or false.

isRotating

Item instances can be allowed to continually rotate by setting the field **rotate** to true, or the instance can have a rotational force applied to it by other means. This function will capture both instances and return true or false based on whether or not the item instance is rotating. You don't need to send any parameters to this function.

getLastStickyPos

This is a special function designed for objects with the **sticky** flag enabled. When an object collides with a **StaticShape** instance and sticks to it, this function can capture the position of the item instance. You don't need to send any additional parameters to this function and it will return a string containing the position of the item.

getLastStickyNormal

You can use the **getLastStickyNormal** function in conjunction with the **getLastStickyPos** function in combination with a raycast (See below) to obtain the object that you're sticking to. This function will return a 3-D vector instance of the normal of the stuck surface on the item instance. Like its counterpart, there are no parameters needed and it will return a string containing the vector instance.

So those are the basic functions you'll need to know for the **Item** class, now let's introduce the function callbacks for this class to give you some additional power to your instances.

onStickyCollision

This callback will only be triggered if the item instance has the **sticky** flag set to true, and the item object collides with a **StaticShape** instance and sticks to it. This function has one additional parameter containing the ID of the item object.

```
function Item::onStickyCollision(%this, %itemID) {  
    //Do Something.  
}
```

onEnterLiquid

Like the **Player** class, the **Item** class also has a callback for entering a liquid surface, and the parameters are very similar as well:

```
function Item::onEnterLiquid(%this, %itemID, %coverage, %liquidType) {  
    //Do Something.  
}
```

Recall that coverage is a factor between 0.0 and 1.0 and is treated as a percentage of how much the item instance is covered.

onLeaveLiquid

Also to be expected then, is the **onLeaveLiquid** callback when the **Item** instance leaves any liquid surface:

```
function Item::onLeaveLiquid(%this, %itemID, %liquidType) {  
    //Do Something.  
}
```

So there you have the callbacks and functions for the **Item** class instance. While not as numerous as the **Player** class, you'll still find many uses in your game applications for the item class, and we'll come back to it in just a little while to introduce a new concept. If you need more resources or help with the **Item** class, see the engine's documentation, or refer to the datablock editor and dump commands on the properties of the object instances.

Now, let's move onto one of the more "important" objects in games of the shooter genre, which are the numerous little toys that your players will be using.

Weapons

Our next stop for the game world covers the topic of Weapons for your game. This is a very generic topic in general since Weapon instances can either be held weapon instances, or mounted turret barrel instances. They both base from the same class instance which is **ShapeBaseImage** or to make it simpler, **ShapeBase**. Since this class is technically speaking, the largest in the engine in terms of in-game functioning, we're not going to go into extreme depths on it, however, there are some important parts of ShapeBase we need to talk about weapons.

Intro to Weapons

So, as you have seen above, the Weapons of the engine use the ShapeBaseImage class, which is a subset of the ShapeBase class. In T3D 3.5.1, the weapon system makes use of three item datablocks, and one image datablock for the weapon itself. The item datablocks are the weapon item instance (what players pick up to get the gun), the ammo for the weapon (held ammo), and the ammo clip (stockpiled ammo) for the weapon. For weapons, you only need to define the image and the weapon item instance. The ammo and the clip are optional depending on what type of weapon you're creating.

Weapon instances have two model instances associated with them, there's the first person perspective model (**shapeFileFP**) and the base model (third & first person [if **shapeFileFP** is undefined]) (**shapeFile**). There are two important pieces for a weapon's image datablock. Both of these are treated as fields in the datablock of the weapon, but they have important roles. The first group we're going to talk about is the properties of the weapon in terms of general roles (think datablock fields as we covered above), and then there is the **state system** of the weapon, which controls how the weapon behaves when it's being operated. So, let's get started

Weapon Properties (Datablock Fields)

So like all other game objects with player interactions, they have a datablock behind them, a weapon instance is no different. And there are a wide number of properties you can set on the datablock instance for a weapon.

Field Name	Value Type	Description
emap	Boolean	A flag to tell the image if it needs to use environmental mapping or not.
shapeFile	String (File Name)	The model to use for the weapon in third person, and first person if shapeFileFP is not defined.
shapeFileFP	String (File Name)	The model to use for the weapon in first person. Note, if you use this parameter you must either define eyeOffset , or useEyeNode otherwise shapeFile will be used.
imageAnimPrefix	String	A string to prepend to the animation sequences for third person animations.
imageAnimPrefixFP	String	A string to prepend to the animation sequences for first person animations.
animateAllShapes	Boolean	When you define multiple shape instances for one instance, should they all synchronize together when animated?
animateOnServer	Boolean	Should the weapon image animation be controlled by the

		server instance? You'll usually want to set this to true if useEyeNode is true, or if the muzzle point of the weapon is animated at any time.
scriptAnimTransitionTime	Floating Point Number	The amount of time to perform a transition animation when moving between sequences controlled by script.
projectile	Projectile Datablock	The projectile instance that is fired from this weapon image.
cloakable	Boolean	Is this image allowed to be cloaked?
mountPoint	Integer	The mount node to attach this image to when it becomes mounted to a player, a turret, etc.
offset	3-Point Value ("x y z")	How much to offset the image instance from the position of the mounted node. By default this is 0 0 0.
rotation	4-Point Value ("x y z a")	How much rotation to offset the image instance, relative to the mounted node instance. By default this is 0 0 0 0.
eyeOffset	3-Point Value ("x y z")	When in first-person mode, how much do we offset the model from the eye perspective. By default this is 0 0 0.
eyeRotation	4-Point Value ("x y z a")	When in first-person mode, how much do we rotate the model relative to the mounted node instance from the eye perspective. By default this is 0 0 0 0.
useEyeNode	Boolean	Mount the instance using the model's eyeNode point and place the camera at the image's eye node. If in FP mode, the image will mount eyeMount for camera placement.
correctMuzzleVector	Boolean	When in first-person mode, should we adjust the aim vector to the eye's LOS vector?
correctMuzzleVectorTP	Boolean	When in third-person mode, should we adjust the aim vector to the eye's LOS vector?
firstPerson	Boolean	Should we render this image in first person?

mass	Floating Point Number	The mass of the weapon/image instance.
usesEnergy	Boolean	Does this weapon drain energy from the holder's energy reserves instead of using ammo?
minEnergy	Floating Point Number	The minimum energy needed to fire the weapon when usesEnergy is set to true.
accuFire	Boolean	Should the weapon aim be converged to the crosshair GUI on the player's screen? (Be aware that this feature is disabled in T3D 3.5.1)
lightType	Enumerated Value	What type of light is emitted by this object instance (Values: NoLight , ConstantLight, SpotLight, PulsingLight, WeaponFireLight)
lightColor	3-Point Value ("r g b")	What color needs to be emitted from this object instance (only if lightType is not NoLight)
lightDuration	Integer	How frequent the pulses of light are if lightType is set to PulsingLight or WeaponFireLight.
lightRadius	Floating Point Number	How much radius the light from this object needs to cover.
lightBrightness	Floating Point Number	How "bright" or how intense is the light instance.
shakeCamera	Boolean	Does the camera instance need to shake when this weapon is fired?
camShakeFreq	3-Point Value ("x y z")	The frequency of the camera shake effect applied to each individual axis.
camShakeAmp	3-Point Value ("x y z")	The amplitude of the camera shake effect applied to each individual axis.
casing	Debris Datablock	The debris datablock to spawn when the weapon is fired (Used to spawn shell casings when a weapon is fired).
shellExitDir	3-Point Value ("x y z")	The vector (from the node) from which shell casings are fired out of the weapon.
shellExitVariance	Floating Point Number	The amount of "randomness" to apply to the exit vector of shell casings (Degrees).
shellVelocity	Floating Point Number	The speed at which shell casings

		are launched out of the weapon when spawned.
computeCRC	Boolean	If set to true, the server will validate that the client's image instance matches that of the server when spawned.
maxConcurrentSounds	Integer	The amount of sounds that can concurrently play from a weapon at once (A value less than or equal to zero means infinite).
useRemainderDT	Boolean	Allow multiple timeout transitions to occur in the same tick?

Like with the **Player** datablock and the **Item** datablock, you don't necessarily need to define all of these parameters, but only a certain number of them need to be defined. Also recall that there are other fields that are inherited from other datablocks that exist above **ShapeBase**. For the full list and descriptions, refer to the documentation of the engine, or see the datablock editor when trying to create a **ShapeBaseImageData** instance.

The Weapon State System

The second part of making a weapon datablock is mastering what is known as the **state system** for weapons. The state system is a special category of array instanced fields consisting of 31 indices that can be defined for a weapon to tell the weapon what to do at a certain point in time. These states define how a weapon behaves when a trigger is pulled, and what happens when the trigger is pulled during that time. This also pieces and wires together what happens after a trigger pull, or when a weapon is currently doing something, and for how long the weapon is stuck in a state.

Weapon states are defined alongside the datablock fields above as they exist within the same bounds. A common coding practice for weapons is to define the above fields first, and then define an array instance containing the values mentioned here. I'll first go over all of the types of states, and cover what values can be set within the state, and then we'll move onto functions for weapons and how to bind these states to actually tell the weapon to do what you want it to do.

How you define a **state** for a weapon is to gather the components you want to be together into an array index of the same type and then use the following fields below to accomplish this, I'll have an example afterwards to demonstrate.

State Name	Value Type	Description	Values
stateName	String	What is the name of this state?	String
stateTransitionOnLoaded	String	What state do we move to when the weapon image loaded state becomes "loaded"?	String
stateTransitionOnNotLoaded	String	What state do we move to when the weapon image loaded state becomes	String

			“empty”?
stateTransitionOnAmmo	String	What state do we move to when the weapon image ammo state is true?	String
stateTransitionOnNoAmmo	String	What state do we move to when the weapon image ammo state is false?	String
stateTransitionOnTarget	String	What state do we move to when the weapon image gains a target?	String
stateTransitionOnNoTarget	String	What state do we move to when the weapon image loses a target?	String
stateTransitionOnWet	String	What state do we move to when the weapon image enters the water?	String
stateTransitionOnNotWet	String	What state do we move to when the weapon image exits the water?	String
stateTransitionOnMotion	String	What state do we move to when the object holding this image moves?	String
stateTransitionOnNoMotion	String	What state do we move to when the object holding this image stops?	String
stateTransitionOnTriggerUp	String	What state do we move to when the object holding this image pushes the fire button down?	String
stateTransitionOnTriggerDown	String	What state do we move to when the object holding this image releases the fire button?	String
stateTransitionOnAltTriggerUp	String	What state do we move to when the object holding this image pushes the alternate fire button down?	String
stateTransitionOnAltTriggerDown	String	What state do we move to when the object holding this image releases the alternate fire button?	String
stateTransitionOnTimeout	String	What state do we move to when the timeout of the current state expires?	String
stateTransitionGeneric0In	String	What state do we move to when the object holding this image pushes trigger0 in?	String

stateTransitionGeneric0Out	String	What state do we move to when the object holding this image releases trigger0?	String
stateTransitionGeneric1In	String	What state do we move to when the object holding this image pushes trigger1 in?	String
stateTransitionGeneric1Out	String	What state do we move to when the object holding this image releases trigger1?	String
stateTransitionGeneric2In	String	What state do we move to when the object holding this image pushes trigger2 in?	String
stateTransitionGeneric2Out	String	What state do we move to when the object holding this image releases trigger2?	String
stateTransitionGeneric3In	String	What state do we move to when the object holding this image pushes trigger3 in?	String
stateTransitionGeneric3Out	String	What state do we move to when the object holding this image releases trigger3?	String
stateTimeoutValue	Floating Point Number	How long of a transition time exists between this state and the next?	Floating Point Number
stateWaitForTimeout	Boolean	If set to false and a condition is met prior to stateTimeoutValue expiring, the transition will occur.	Boolean
stateFire	Boolean	The first state with this set to true is defined as the state in which the weapon fires.	Boolean
stateAlternateFire	Boolean	The first state with this set to true is defined as the state in which the weapon fires it's alternate.	Boolean
stateReload	Boolean	The first state with this set to true is defined as the reload state.	Boolean
stateEjectShell	Boolean	If true, the weapon will fire off a shell casing when this state is active.	Boolean
stateEnergyDrain	Floating Point Number	How much energy is drained from the object that enters this state.	Floating Point Number
stateAllowImageChange	Boolean	If set to false, the object is not allowed to mount images while this state is active.	Boolean

stateDirection	Boolean	Which direction to play the animation for this state?	True – forwards false – backwards
stateLoadedFlag	Enum	What status of “loaded” to apply to the image?	IgnoreLoaded, Loaded, NotLoaded
stateSpinThread	Enum	How fast should the “spin” animation play on the image while this state is still active?	Ignore, Stop, SpinUp, SpinDown, FullSpeed
stateRecoil	Enum	How much recoil should be applied to the image when this state is active?	NoRecoil, LightRecoil, MediumRecoil, HeavyRecoil
stateSequence	String	What animation should be played on entry to this state?	String
stateSequenceRandomFlash	Boolean	Does this state play a weapon flash randomly when it’s active?	Boolean
stateScaleAnimation	Boolean	Do we scale the time of the animation to cover the timeout value of the state?	Boolean
stateScaleAnimationFP	Boolean	Do we scale the time of the first person animation to cover the timeout value of the state?	Boolean
stateSequenceTransitionIn	Boolean	Do we transition to the animation when we enter this state?	Boolean
stateSequenceTransitionOut	Boolean	Do we transition to the animation when we exit this state?	Boolean
stateSequenceNeverTransition	Boolean	Are animation transitions never allowed during this state? (This is usually used during the fire state)	Boolean
stateSequenceTransitionTime	Floating Point Number	The time we take to perform a transition for the animation on this state.	Boolean
stateShapeSequence	String	The animation to play on the object mounting this image.	String
stateScaleShapeSequence	Boolean	Should stateShapeSequence be scaled in time to match the timeout of the current state?	Boolean
stateSound	String (SFX File Path)	What sound needs to be played when this state is active?	String
stateScript	String	What object function needs to be called when this state is	String

		activated. The format is: ImageData::X(%this, %obj, %slot)	
stateEmitter	Particle Emitter Datablock	What particle emitter needs to be spawned on the weapon when this state is entered?	Datablock
stateEmitterTime	Floating Point Number	How long does this emitter need to remain active in this state?	Floating Point Number
stateEmitterNode	String	Which weapon node do we attach the particle emitter to?	String
stateIgnoreLoadedForReady	Boolean	If this is set to true, and both ready and loaded transitions are also true, the ready transition will take precedence over loaded.	Boolean

As you can see from this table, there is a wide variety of options at your disposal for states for your image instances. Learning how to properly use the state system does take time and practice, but once you master it, you'll be able to create almost any type of weapon you need for your game. Let's look at a quick example:

```
datablock ShapeBaseImage(MyShotgunBlock) {
    //blah blah fields...
    //

    // Initial start up state
    stateName[0]          = "Preactivate";
    stateTransitionOnLoaded[0] = "Activate";
    stateTransitionOnNoAmmo[0] = "NoAmmo";

    // Activating the gun. Called when the weapon is first
    // mounted and there is ammo.
    stateName[1]          = "Activate";
    stateTransitionGeneric0In[1] = "SprintEnter";
    stateTransitionOnTimeout[1] = "Ready";
    stateTimeoutValue[1]     = 1.0;
    stateSequence[1]        = "switch_in";
    stateSound[1]           = DispositionSwitchinSound;

    // Ready to fire, just waiting for the trigger
    stateName[2]          = "Ready";
    stateTransitionGeneric0In[2] = "SprintEnter";
    stateTransitionOnMotion[2] = "ReadyMotion";
}
```

```
stateTransitionOnTimeout[2] = "ReadyFidget";
stateTimeoutValue[2] = 10;
stateWaitForTimeout[2] = false;
stateScaleAnimation[2] = false;
stateScaleAnimationFP[2] = false;
stateTransitionOnNoAmmo[2] = "NoAmmo";
stateTransitionOnTriggerDown[2] = "Fire";
stateSequence[2] = "idle";

// Same as Ready state but plays a fidget sequence
stateName[3] = "ReadyFidget";
stateTransitionGeneric0In[3] = "SprintEnter";
stateTransitionOnMotion[3] = "ReadyMotion";
stateTransitionOnTimeout[3] = "Ready";
stateTimeoutValue[3] = 1;
stateWaitForTimeout[3] = false;
stateTransitionOnNoAmmo[3] = "NoAmmo";
stateTransitionOnTriggerDown[3] = "Fire";
stateSequence[3] = "idle_fidget1";
stateSound[3] = DispositionCockSound;

// Ready to fire with player moving
stateName[4] = "ReadyMotion";
stateTransitionGeneric0In[4] = "SprintEnter";
stateTransitionOnNoMotion[4] = "Ready";
stateWaitForTimeout[4] = false;
stateScaleAnimation[4] = false;
stateScaleAnimationFP[4] = false;
stateSequenceTransitionIn[4] = true;
stateSequenceTransitionOut[4] = true;
stateTransitionOnNoAmmo[4] = "NoAmmo";
stateTransitionOnTriggerDown[4] = "Fire";
stateSequence[4] = "run";

// Fire the weapon. Calls the fire script which does
// the actual work.
stateName[5] = "Fire";
stateTransitionGeneric0In[5] = "SprintEnter";
```

```
stateTransitionOnTimeout[5]    = "Reload";
stateTimeoutValue[5]          = 0.33;
stateFire[5]                  = true;
stateRecoil[5]                = "";
stateAllowImageChange[5]       = false;
stateSequence[5]              = "fire";
stateScaleAnimation[5]        = true;
stateSequenceNeverTransition[5] = true;
stateSequenceRandomFlash[5]   = true;    // use muzzle flash sequence
stateScript[5]                = "onFire";
stateEmitter[5]               = GunFireSmokeEmitter;
stateEmitterTime[5]           = 0.025;
stateSound[5]                 = DispositionFireSound;

// Play the reload animation, and transition into
stateName[6]                  = "Reload";
stateTransitionGeneric0In[6]    = "SprintEnter";
stateTransitionOnNoAmmo[6]      = "NoAmmo";
stateTransitionOnTimeout[6]     = "Cocking";
stateWaitForTimeout[6]          = "0";
stateTimeoutValue[6]            = 0.0;
stateAllowImageChange[6]        = false;

// Cock the shotgun
stateName[7]                  = "Cocking";
stateTransitionGeneric0In[7]    = "SprintEnter";
stateTransitionOnNoAmmo[7]      = "NoAmmo";
stateTransitionOnTimeout[7]     = "Ready";
stateTimeoutValue[7]            = 0.75;
stateWaitForTimeout[7]          = true;
stateSequence[7]                = "fire_cocking";
stateEjectShell[7]              = true;
stateAllowImageChange[7]         = false;
stateSound[7]                   = DispositionCockSound;

// No ammo in the weapon, just idle until something
// shows up. Play the dry fire sound if the trigger is
// pulled.
```

```
stateName[8]          = "NoAmmo";
stateTransitionGeneric0In[8] = "SprintEnter";
stateTransitionOnMotion[8] = "NoAmmoMotion";
stateTransitionOnAmmo[8]  = "ReloadClip";
stateTimeoutValue[8]     = 0.1; // Slight pause to allow script to run when trigger is still held down from Fire state
stateScript[8]           = "onClipEmpty";
stateTransitionOnTriggerDown[8] = "DryFire";
stateSequence[8]         = "idle";
stateScaleAnimation[8]   = false;
stateScaleAnimationFP[8] = false;

stateName[9]          = "NoAmmoMotion";
stateTransitionGeneric0In[9] = "SprintEnter";
stateTransitionOnNoMotion[9] = "NoAmmo";
stateWaitForTimeout[9]   = false;
stateScaleAnimation[9]   = false;
stateScaleAnimationFP[9] = false;
stateSequenceTransitionIn[9] = true;
stateSequenceTransitionOut[9] = true;
stateTransitionOnAmmo[9]  = "ReloadClip";
stateTransitionOnTriggerDown[9] = "DryFire";
stateSequence[9]         = "run";

// No ammo dry fire
stateName[10]          = "DryFire";
stateTransitionGeneric0In[10] = "SprintEnter";
stateTimeoutValue[10]     = 1.0;
stateTransitionOnTimeout[10] = "NoAmmo";
stateScript[10]           = "onDryFire";

// Play the reload clip animation
stateName[11]          = "ReloadClip";
stateTransitionGeneric0In[11] = "SprintEnter";
stateTransitionOnTimeout[11] = "Ready";
stateWaitForTimeout[11]   = true;
stateTimeoutValue[11]     = 3.0;
stateReload[11]           = true;
stateSequence[11]         = "reload";
```

```
stateShapeSequence[11]      = "Reload";
stateScaleShapeSequence[11]   = true;
stateSound[11]               = DispositionReloadSound;

// Start Sprinting
stateName[12]                = "SprintEnter";
stateTransitionGeneric0Out[12] = "SprintExit";
stateTransitionOnTimeout[12]   = "Sprinting";
stateWaitForTimeout[12]       = false;
stateTimeoutValue[12]         = 0.5;
stateWaitForTimeout[12]       = false;
stateScaleAnimation[12]       = false;
stateScaleAnimationFP[12]     = false;
stateSequenceTransitionIn[12] = true;
stateSequenceTransitionOut[12] = true;
stateAllowImageChange[12]     = false;
stateSequence[12]              = "sprint";

// Sprinting
stateName[13]                = "Sprinting";
stateTransitionGeneric0Out[13] = "SprintExit";
stateWaitForTimeout[13]       = false;
stateScaleAnimation[13]       = false;
stateScaleAnimationFP[13]     = false;
stateSequenceTransitionIn[13] = true;
stateSequenceTransitionOut[13] = true;
stateAllowImageChange[13]     = false;
stateSequence[13]              = "sprint";

// Stop Sprinting
stateName[14]                = "SprintExit";
stateTransitionGeneric0In[14]  = "SprintEnter";
stateTransitionOnTimeout[14]   = "Ready";
stateWaitForTimeout[14]       = false;
stateTimeoutValue[14]         = 0.5;
stateSequenceTransitionIn[14] = true;
stateSequenceTransitionOut[14] = true;
stateAllowImageChange[14]     = false;
```

```
stateSequence[14] = "sprint";  
};
```

This is another very large amount of text to look over, so let's quickly talk about what's going on here with this code. The initial state of the weapon is called right when the player receives the `onMount` call. The state quickly checks for ammo, and then forks the weapon off to the proper state. If the weapon has ammo, it moves to state 1, which tells the weapon it's ready for use, and to play the load animation. Once the timeout expires on this state, we move into state 2, which is the **ready** state, or basically, the gun is sitting here and can now fire when the player pushes the button.

There are a few idle animation sequences attached to the weapon on the first person that can be triggered every 10 seconds or so for the next few states. Then we move into the Fire state, which calls the script to fire the weapon, and then perform another action. The next action is dependent on if the gun still has ammo, or if there are more clips in the gun. Finally, we have our states for sprinting with the gun, and reloading the gun.

Just by looking at the example and watching how each state transitions from one to another, you'll see how to quickly set up a weapon exactly how you need it to be, and I'll cover the basic archetypes below in just a bit, but let's talk about some functions at your disposal first.

Functions for Weapon Images

You may be surprised at first, but the entire engine's weapons and image system relies on two functions, that's right only two functions. And they're callback functions. The reason for this is because the weapon system makes use of the state system to define its primary functioning when a weapon is currently doing a specific action. By making use of the **stateScript** field, you can insert additional scripts when a weapon enters a state to perform an action of your own coding.

For instance, most guns in the engine's templates define a **onFire** method and attach it to the **stateScript** of the fire state, from there the script that is written spawns the projectile instance where it needs to be and gives it the necessary velocity and everything else is handled for you!

So, the two function callbacks are **onMount** and **onUnmount**, and they are called when the gun is mounted to an instance, and when it is removed from a player instance. They both appear in the same script format:

```
function ImageData::onMount(%this, %obj, %slot, %dt) {  
    //Do Something  
}
```

The final parameter is usually ignored on the function instance here as it defines how much time is left on the current weapon state. You should know what `%this` is by now. `%obj` is the object that is holding the weapon, and `%slot` is the slot that the weapon instance is being mounted to. A key tip to know here is that **every single state that defines stateScript uses these same function parameters**. That should hopefully keep you out of trouble when writing scripts that are triggered by the state system. The important thing to keep note of is what each parameter is in the function, and what functions that

are granted to you by access to those variables, for instance the %obj variable will either be attached to the player holding the weapon, or the turret the image is mounted to, and so on. By knowing this, you can expand the ability of your weapon image's script to make use of additional functions that exist on the object that is holding the weapon, granting you even more possibilities. So let's quickly take a peek at a typical weapon **onFire** function:

```
function WeaponImage::onFire(%this, %obj, %slot) {  
    // Make sure we have valid data  
    if (!isObject(%this.projectile)) {  
        error("WeaponImage::onFire() - Invalid projectile datablock");  
        return;  
    }  
    // Decrement inventory ammo. The image's ammo state is updated  
    // automatically by the ammo inventory hooks.  
    if ( !%this.infiniteAmmo )  
        %obj.declInventory(%this.ammo, 1);  
  
    // Get the player's velocity, we'll then add it to that of the projectile  
    %objectVelocity = %obj.getVelocity();  
  
    %numProjectiles = %this.projectileNum;  
    if (%numProjectiles == 0)  
        %numProjectiles = 1;  
  
    for (%i = 0; %i < %numProjectiles; %i++) {  
        if (%this.projectileSpread) {  
            // We'll need to "skew" this projectile a little bit. We start by  
            // getting the straight ahead aiming point of the gun  
            %vec = %obj.getMuzzleVector(%slot);  
  
            // Then we'll create a spread matrix by randomly generating x, y, and z  
            // points in a circle  
            %matrix = "";  
            for(%j = 0; %j < 3; %j++)  
                %matrix = %matrix @ (getRandom() - 0.5) * 2 * 3.1415926 * %this.projectileSpread @ " ";  
            %mat = MatrixCreateFromEuler(%matrix);  
  
            // Which we'll use to alter the projectile's initial vector with  
            %muzzleVector = MatrixMulVector(%mat, %vec);  
    }  
}
```

```
}

else {

    // Weapon projectile doesn't have a spread factor so we fire it using
    // the straight ahead aiming point of the gun
    %muzzleVector = %obj.getMuzzleVector(%slot);

}

// Add player's velocity
%muzzleVelocity = VectorAdd(
    VectorScale(%muzzleVector, %this.projectile.muzzleVelocity),
    VectorScale(%objectVelocity, %this.projectile.vellInheritFactor));

// Create the projectile object
%p = new (%this.projectileType) {
    dataBlock = %this.projectile;
    initialVelocity = %muzzleVelocity;
    initialPosition = %obj.getMuzzlePoint(%slot);
    sourceObject = %obj;
    sourceSlot = %slot;
    client = %obj.client;
    sourceClass = %obj.getClassName();
};

MissionCleanup.add(%p);
}

}
```

If you've been poking around some script files, you might notice that this function exists in the stock engine's templates, and that's because this is a general over scoped function that is used to work with **all** weapon instances, reason being is because all held weapons, although they use the ShapeBaseImage block, are still counted as a WeaponImage. You might see some familiarities to some things in the math chapter here, as we're doing things like compensating for player velocity and weapon spread in this code, but the key focus is still the same here. When this code is triggered, we spawn a projectile and decrease the ammo from the gun.

If you wanted to write a custom onFire script, you could still do that, because remember, your individual weapon image is on a higher tier, and therefore you could override the existing onFire method with a custom method of your own design. So, now that you've seen how to write and use functions for weapon images, let's go back to the state system, and show you how to make weapons of different types.

How to create various types of Weapons

So now that you've got a grasp on how the state system works and how we use it to trigger functions of various types, let's actually combine the two topics and show you how to create various types of weapons. Obviously to cover every single weapon would take an infinite amount of time because there are so many weapons plus the infinite possibility of new weapons, so instead I'm going to cover the basic archetypes or basically categories of weapons to make life easy for us all. My goal here is to provide a couple of good samples and then to let you use these samples to expand into different types.

Fully Automatic Weapon

One of the most common types of weapons made in FPS games are weapons that classify as Fully Automatic (Assault Rifles, Machine Guns, Etc), and they work on the basic purpose that when the fire trigger is held down, that bullets will continue to be fired until the clip is empty. The basis of the states for a fully automatic weapon is simple enough to understand, when the trigger goes down you enter the fire state, and then it moves to a conditional state where you check if the trigger is still active, and ammo is still there and fire again if it is. Here's the basic form of this state:

```
// Initial start up state
stateName[0]          = "Preactivate";
stateTransitionOnLoaded[0] = "Activate";
stateTransitionOnNoAmmo[0] = "NoAmmo";

// Activating the gun. Called when the weapon is first
// mounted and there is ammo.
stateName[1]          = "Activate";
stateTransitionGeneric0In[1] = "SprintEnter";
stateTransitionOnTimeout[1] = "Ready";
stateTimeoutValue[1]     = 0.5;

// Ready to fire, just waiting for the trigger
stateName[2]          = "Ready";
stateTransitionGeneric0In[2] = "SprintEnter";
stateTransitionOnMotion[2] = "ReadyMotion";
stateWaitForTimeout[2]   = false;
stateScaleAnimation[2]   = false;
stateScaleAnimationFP[2] = false;
stateTransitionOnNoAmmo[2] = "NoAmmo";
stateTransitionOnTriggerDown[2] = "Fire";

// Ready to fire with player moving
```

```
stateName[3]          = "ReadyMotion";
stateTransitionGeneric0In[3] = "SprintEnter";
stateTransitionOnNoMotion[3] = "Ready";
stateWaitForTimeout[3]     = false;
stateScaleAnimation[3]     = false;
stateScaleAnimationFP[3]   = false;
stateSequenceTransitionIn[3] = true;
stateSequenceTransitionOut[3] = true;
stateTransitionOnNoAmmo[3]  = "NoAmmo";
stateTransitionOnTriggerDown[3] = "Fire";

// Fire the weapon. Calls the fire script which does
// the actual work.

stateName[4]          = "Fire";
stateTransitionGeneric0In[4] = "SprintEnter";
stateTransitionOnTimeout[4] = "NewRound";
stateTimeoutValue[4]    = 0.06;
stateFire[4]           = true;
stateRecoil[4]          = "";
stateAllowImageChange[4] = false;
stateScaleAnimation[4]   = false;
stateSequenceNeverTransition[4] = true;
stateSequenceRandomFlash[4] = true; // use muzzle flash sequence
stateScript[4]          = "onFire";

// Put another round into the chamber if one is available

stateName[5]          = "NewRound";
stateTransitionGeneric0In[5] = "SprintEnter";
stateTransitionOnNoAmmo[5]  = "NoAmmo";
stateTransitionOnTimeout[5] = "Ready";
stateWaitForTimeout[5]    = "0";
stateTimeoutValue[5]     = 0.07;
stateAllowImageChange[5]  = false;
stateEjectShell[5]        = true;

// No ammo in the weapon, just idle until something
// shows up. Play the dry fire sound if the trigger is
// pulled.
```

```
stateName[6]          = "NoAmmo";
stateTransitionGeneric0In[6] = "SprintEnter";
stateTransitionOnMotion[6] = "NoAmmoMotion";
stateTransitionOnAmmo[6]  = "ReloadClip";
stateTimeoutValue[6]     = 0.1; // Slight pause to allow script to run when trigger is still held down from Fire state
stateScript[6]           = "onClipEmpty";
stateScaleAnimation[6]   = false;
stateScaleAnimationFP[6] = false;
stateTransitionOnTriggerDown[6] = "DryFire";

// No ammo, and sprinting
stateName[7]          = "NoAmmoMotion";
stateTransitionGeneric0In[7] = "SprintEnter";
stateTransitionOnNoMotion[7] = "NoAmmo";
stateWaitForTimeout[7]   = false;
stateScaleAnimation[7]   = false;
stateScaleAnimationFP[7] = false;
stateSequenceTransitionIn[7] = true;
stateSequenceTransitionOut[7] = true;
stateTransitionOnTriggerDown[7] = "DryFire";
stateTransitionOnAmmo[7]  = "ReloadClip";

// No ammo dry fire
stateName[8]          = "DryFire";
stateTransitionGeneric0In[8] = "SprintEnter";
stateTransitionOnAmmo[8]  = "ReloadClip";
stateWaitForTimeout[8]   = "0";
stateTimeoutValue[8]     = 0.7;
stateTransitionOnTimeout[8] = "NoAmmo";
stateScript[8]           = "onDryFire";

// Play the reload clip animation
stateName[9]          = "ReloadClip";
stateTransitionGeneric0In[9] = "SprintEnter";
stateTransitionOnTimeout[9] = "Ready";
stateWaitForTimeout[9]   = true;
stateTimeoutValue[9]     = 3.0;
stateReload[9]           = true;
```

```
stateScaleShapeSequence[9]      = true;

// Start Sprinting
stateName[10]                  = "SprintEnter";
stateTransitionGeneric0Out[10]   = "SprintExit";
stateTransitionOnTimeout[10]     = "Sprinting";
stateWaitForTimeout[10]         = false;
stateTimeoutValue[10]           = 0.5;
stateWaitForTimeout[10]         = false;
stateScaleAnimation[10]         = false;
stateScaleAnimationFP[10]        = false;
stateSequenceTransitionIn[10]   = true;
stateSequenceTransitionOut[10]  = true;
stateAllowImageChange[10]        = false;

// Sprinting
stateName[11]                  = "Sprinting";
stateTransitionGeneric0Out[11]   = "SprintExit";
stateWaitForTimeout[11]          = false;
stateScaleAnimation[11]          = false;
stateScaleAnimationFP[11]         = false;
stateSequenceTransitionIn[11]   = true;
stateSequenceTransitionOut[11]  = true;
stateAllowImageChange[11]        = false;

// Stop Sprinting
stateName[12]                  = "SprintExit";
stateTransitionGeneric0In[12]    = "SprintEnter";
stateTransitionOnTimeout[12]     = "Ready";
stateWaitForTimeout[12]          = false;
stateTimeoutValue[12]            = 0.5;
stateSequenceTransitionIn[12]   = true;
stateSequenceTransitionOut[12]  = true;
stateAllowImageChange[12]        = false;
```

Notice how the fire sequence works for this weapon. When the fire state is entered, it checks the gun for ammo and then places it back in the Ready sequence if the gun still has ammo. The ready sequence then validates if the fire trigger is still held down, and fires the next shot off if it is. This is the basic form of the fully automatic weapon in Torque 3D. Now, let's look at the semi-automatic gun.

Semi-Automatic Weapon

The next type of weapon we're going to look at is the semi-automatic weapon. A semi-auto gun exists to fire bullets one at a time, or once per trigger pull, and holding the trigger down either blocks additional bullets from firing, or just has a delay before the next shot is fired. Most guns of this class exist in the form of shotguns, sniper rifles, and semi-automatic rifles. Let's have a look at the basic state system form:

```
// Initial start up state
stateName[0]          = "Preactivate";
stateTransitionOnLoaded[0] = "Activate";
stateTransitionOnNoAmmo[0] = "NoAmmo";

// Activating the gun. Called when the weapon is first
// mounted and there is ammo.
stateName[1]          = "Activate";
stateTransitionGeneric0In[1] = "SprintEnter";
stateTransitionOnTimeout[1] = "Ready";
stateTimeoutValue[1]     = 1.0;

// Ready to fire, just waiting for the trigger
stateName[2]          = "Ready";
stateTransitionGeneric0In[2] = "SprintEnter";
stateTransitionOnMotion[2] = "ReadyMotion";
stateWaitForTimeout[2]   = false;
stateScaleAnimation[2]   = false;
stateScaleAnimationFP[2]  = false;
stateTransitionOnNoAmmo[2] = "NoAmmo";
stateTransitionOnTriggerDown[2] = "Fire";

// Ready to fire with player moving
stateName[3]          = "ReadyMotion";
stateTransitionGeneric0In[3] = "SprintEnter";
stateTransitionOnNoMotion[3] = "Ready";
stateWaitForTimeout[3]   = false;
stateScaleAnimation[3]   = false;
stateScaleAnimationFP[3]  = false;
stateSequenceTransitionIn[3] = true;
stateSequenceTransitionOut[3] = true;
stateTransitionOnNoAmmo[3] = "NoAmmo";
```

```
stateTransitionOnTriggerDown[3] = "Fire";  
  
// Fire the weapon. Calls the fire script which does  
// the actual work.  
  
stateName[4] = "Fire";  
stateTransitionGeneric0In[4] = "SprintEnter";  
stateTransitionOnTimeout[4] = "NewRound";  
stateTimeoutValue[4] = 0.33;  
stateFire[4] = true;  
stateRecoil[4] = "";  
stateAllowImageChange[4] = false;  
stateScaleAnimation[4] = true;  
stateSequenceNeverTransition[4] = true;  
stateSequenceRandomFlash[4] = true; // use muzzle flash sequence  
stateScript[4] = "onFire";  
  
// Put a new round into the chamber  
  
stateName[5] = "NewRound";  
stateTransitionGeneric0In[5] = "SprintEnter";  
stateTransitionOnNoAmmo[5] = "NoAmmo";  
stateTransitionOnTimeout[5] = "Ready";  
stateWaitForTimeout[5] = "0";  
stateTimeoutValue[5] = 0.33;  
stateAllowImageChange[5] = false;  
  
// No ammo in the weapon, just idle until something  
// shows up. Play the dry fire sound if the trigger is  
// pulled.  
  
stateName[6] = "NoAmmo";  
stateTransitionGeneric0In[6] = "SprintEnter";  
stateTransitionOnMotion[6] = "NoAmmoMotion";  
stateTransitionOnAmmo[6] = "ReloadClip";  
stateTimeoutValue[6] = 0.1; // Slight pause to allow script to run when trigger is still held down from Fire state  
stateScript[6] = "onClipEmpty";  
stateTransitionOnTriggerDown[6] = "DryFire";  
stateScaleAnimation[6] = false;  
stateScaleAnimationFP[6] = false;
```

```
stateName[7]          = "NoAmmoMotion";
stateTransitionGeneric0In[7]  = "SprintEnter";
stateTransitionOnNoMotion[7] = "NoAmmo";
stateWaitForTimeout[7]     = false;
stateScaleAnimation[7]    = false;
stateScaleAnimationFP[7]  = false;
stateSequenceTransitionIn[7] = true;
stateSequenceTransitionOut[7] = true;
stateTransitionOnAmmo[7]   = "ReloadClip";
stateTransitionOnTriggerDown[7] = "DryFire";

// No ammo dry fire
stateName[8]          = "DryFire";
stateTransitionGeneric0In[8]  = "SprintEnter";
stateTimeoutValue[8]     = 1.0;
stateTransitionOnTimeout[8] = "NoAmmo";
stateScript[8]          = "onDryFire";

// Play the reload clip animation
stateName[9]          = "ReloadClip";
stateTransitionGeneric0In[9]  = "SprintEnter";
stateTransitionOnTimeout[9] = "Ready";
stateWaitForTimeout[9]     = true;
stateTimeoutValue[9]      = 3.0;
stateReload[9]           = true;
stateScaleShapeSequence[9] = true;

// Start Sprinting
stateName[10]          = "SprintEnter";
stateTransitionGeneric0Out[10] = "SprintExit";
stateTransitionOnTimeout[10] = "Sprinting";
stateWaitForTimeout[10]     = false;
stateTimeoutValue[10]      = 0.5;
stateWaitForTimeout[10]     = false;
stateScaleAnimation[10]    = false;
stateScaleAnimationFP[10]   = false;
stateSequenceTransitionIn[10] = true;
stateSequenceTransitionOut[10] = true;
```

```
stateAllowImageChange[10]      = false;

// Sprinting
stateName[11]                  = "Sprinting";
stateTransitionGeneric0Out[11]   = "SprintExit";
stateWaitForTimeout[11]         = false;
stateScaleAnimation[11]         = false;
stateScaleAnimationFP[11]        = false;
stateSequenceTransitionIn[11]   = true;
stateSequenceTransitionOut[11]  = true;
stateAllowImageChange[11]        = false;

// Stop Sprinting
stateName[12]                  = "SprintExit";
stateTransitionGeneric0In[12]    = "SprintEnter";
stateTransitionOnTimeout[12]     = "Ready";
stateWaitForTimeout[12]          = false;
stateTimeoutValue[12]           = 0.5;
stateSequenceTransitionIn[12]   = true;
stateSequenceTransitionOut[12]  = true;
stateAllowImageChange[12]        = false;
```

You should notice now that the forms appear almost similar, because the main point to be made here is that the timeout parameter controls the timing between each individual state. Obviously you could add a 13th state to test if you're still holding down the fire button and not to fire again until the trigger is released to make a more realistic "feel" of semi-automatic, but I leave that decision to you in terms of your own weapons.

Now let's look at the energy based weapons in the engine, and show you how to set one of those up.

Energy Weapons

For energy based weapons, we take a step away from the state system, and move back to the datablock parameters for the weapon image, specifically the **usesEnergy** and **minEnergy** parameters which control if the weapon uses energy, and how much energy needs to be drained from the object using the weapon. The actual process here is the exact same for full auto and semi auto because the engine treats having enough energy to fire as **has Ammo** and not enough energy as **noAmmo**. Therefore you can repeat the same states as desired for your weapon instance. Here's an example of an energy weapon datablock:

```
datablock ShapeBaseImageData(EnergyChaingunImage) {
    className = WeaponImage;
```

```
shapeFile = "art/shapes/weapons/EChaingun/EChaingun.dae";
item      = EnergyChaingun;

projectile = EnergyChaingunBullet;
projectileType = Projectile;
usesEnergy = 1;
minEnergy = 10;

emap = true;

casing      = ShellDebris;
shellExitDir  = "0.3 0.5 1.0";
shellExitOffset = "0.15 -0.56 -0.1";
shellExitVariance = 15.0;
shellVelocity   = 4.0;

.
.

};

};
```

So, as you can see in this little example there is no Ammo or Clip datablock associated with the weapon setup here, and we make use of the fields, **usesEnergy** and **minEnergy** to define the “ammo” properties of this weapon. From here forward, you’d define either a full auto mode or a semi auto mode. Or, if you want to be more realistic here with the whole “chaingun” notion, you’d use the next topic, which is a continual fire weapon.

Continual Fire Weapons

The next gun type we’re going to talk about is a continual fire weapon. This weapon comes in two forms, you have a spin-up sequence style weapon, where the gun spins into a ready state and then fires, and then you have a continual fire style weapon, where as long as the fire trigger is held down, the weapon will continually fire until the trigger is released. I’ll cover both of those types here.

To cut down on the space, I’m not going to post the full sequence, but only the important parts of the sequence here.

Spin-Up

To create a spin-up style of weapon, you create a spin-up state as well as a spin-down state. When the fire trigger is initiated, the gun will spin-up to the fire state, and when the trigger is released the gun will spin-down back to the ready state. You can add additional states as needed. Here’s the format of this style of weapon:

```
stateName[1]  = "Ready";
```

```
stateSpinThread[1] = Stop;
stateTransitionOnTriggerDown[1] = "Spinup";
stateTransitionOnNoAmmo[1]    = "NoAmmo";

stateName[2]      = "NoAmmo";
stateTransitionOnAmmo[2] = "Ready";
stateSpinThread[2] = Stop;
stateTransitionOnTriggerDown[2] = "DryFire";

stateName[3]      = "Spinup";
stateSpinThread[3] = SpinUp;
stateTimeoutValue[3] = 0.5;
stateWaitForTimeout[3] = false;
stateTransitionOnTimeout[3] = "Fire";
stateTransitionOnTriggerUp[3] = "Spindown";

stateName[4]      = "Fire";
stateSequence[4]   = "Fire";
stateSequenceRandomFlash[4] = true;
stateSpinThread[4] = FullSpeed;
stateAllowImageChange[4] = false;
stateScript[4]     = "onFire";
stateFire[4]       = true;
stateEjectShell[4] = true;
stateTimeoutValue[4] = 0.02;
stateTransitionOnTimeout[4] = "Fire";
stateTransitionOnTriggerUp[4] = "Spindown";
stateTransitionOnNoAmmo[4] = "EmptySpindown";

stateName[5]      = "Spindown";
stateSpinThread[5] = SpinDown;
stateTimeoutValue[5] = 1.0;
stateWaitForTimeout[5] = true;
stateTransitionOnTimeout[5] = "Ready";
stateTransitionOnTriggerDown[5] = "Spinup";

stateName[6]      = "EmptySpindown";
stateSpinThread[6] = SpinDown;
```

```
stateTimeoutValue[6] = 0.5;  
stateTransitionOnTimeout[6] = "NoAmmo";
```

As you can see here, we create a few new states, one for spinning the gun into the fire state, one for spinning the gun down, and an empty spin-down state, which is used if the gun is forcibly spun-down by running out of ammo. You'll likely use this style of weapon for things like a chaingun, or a large machine gun, where the gun spins into a ready state, and then fires very rapidly until the gun spins back down.

Continual Fire

A continual firing weapon is used for things such as a directed laser weapon where the gun fired constantly while the trigger is held down and stops firing once released. This feature can sometimes be combined with the spin sequences to introduce a charging mechanic (see next section) and usually requires two or three unique states:

```
stateName[3] = "Fire";  
stateEnergyDrain[3] = 3;  
stateFire[3] = true;  
stateAllowImageChange[3] = false;  
stateScript[3] = "onFire";  
stateTransitionOnTriggerUp[3] = "Deconstruction";  
stateTransitionOnNoAmmo[3] = "Deconstruction";  
  
stateName[4] = "NoAmmo";  
stateTransitionOnAmmo[4] = "Ready";  
  
stateName[5] = "Deconstruction";  
stateScript[5] = "deconstruct";  
stateTransitionOnTimeout[5] = "Ready";
```

Creating a continual firing weapon usually involves creating a “deconstruct” phase, which will be called when the trigger is released to delete the beam projectile if you’re using a scripted beam, or to clean up anything else from the firing sequence.

Charge-Up Sequence

Finally, I want to show you how to create a charge-up sequence which can be used for things like large weapons (Railguns) and laser type weapons. This is actually a very simple thing to do by combining existing mechanics:

```
stateName[3] = "Charge";  
stateTransitionOnNoAmmo[3] = "NoAmmo";  
stateTransitionOnTimeout[3] = "Charge";  
stateTimeoutValue[3] = 0.05;
```

```
stateScript[3] = "onCharge";
stateTransitionOnTriggerUp[3] = "Fire";

stateName[4] = "Fire";
stateTransitionOnTimeout[4] = "Reload";
stateTimeoutValue[4] = 0.1;
stateFire[4] = true;
stateRecoil[4] = LightRecoil;
stateAllowImageChange[4] = false;
stateScript[4] = "onFire";

stateName[5] = "Reload";
stateTransitionOnNoAmmo[5] = "NoAmmo";
stateTransitionOnTimeout[5] = "Ready";
stateAllowImageChange[5] = false;
stateSequence[5] = "Reload";
```

So, what we've created here is a sequence that when the fire trigger is pressed, we move into a charge state, and when the trigger is released, we fire the weapon. To actually accomplish this feature, we create two unique functions for the weapon image, a onCharge function and a custom onFire function:

```
function PhotonLaserImage::onCharge(%data, %obj, %slot) {
    if(%obj.chargeCountPhotonLaser $= "") {
        %obj.chargeCountPhotonLaser = 0;
    }
    %obj.chargeCountPhotonLaser++;
    if(%obj.chargeCountPhotonLaser > 100) {
        %obj.chargeCountPhotonLaser = 100;
    }
}
```

What we've created here is a simple counter mechanic, where the weapon's charge count is stored on the weapon sequence every time it's called. Then for the onFire script we do a basic check to see if the maximum charge is met:

```
function PhotonLaserImage::onFire(%data, %obj, %slot) {
    if(%obj.chargeCountPhotonLaser == 100) {
        Parent::onFire(%data, %obj, %slot);
    }
    %obj.chargeCountPhotonLaser = 0;
}
```

So my ultimate goal of this section is that you start to see how weapons blend the concepts of a diverse state system with the object methods and callbacks we've been talking about since chapter 8 to create diverse and uniquely designed and implemented weapons for your game. But the weapon image itself is only one part of the puzzle, now we need to introduce the other piece.

Projectiles

The other piece of the puzzle is the weapon's fired projectile, which is an object that usually exists for a very short time, storing a few fields and holding a few important callbacks that are used to actually do work in the FPS environment. You can create a wide assortment of projectiles that range from simple things like a bullet, all the way to something more complex like a grenade instance or a missile that tracks a target. While some projectiles will be very easy to create and likely only need a datablock definition, others will require the implementation of some of those complex mathematics we saw back in chapter 9 combined with some knowledge from this chapter to create.

The first thing we need to do however is to talk about the projectile datablock and how to set it up.

The Projectile Datablock

Here are the fields for the projectile datablock and a short description of how it works for each individual projectile definition:

Field Name	Value Type	Description
particleEmitter	Emitter Datablock	The particle emitter this projectile should create when not in a liquid.
particleWaterEmitter	Emitter Datablock	The particle emitter this projectile should create when submerged in a liquid.
projectileShapeName	String (File Path)	The model this projectile should use when being rendered.
scale	3-Point Value ("x y z")	The scale to apply to the model compared to its normal size (value of 1.0 is normal)
sound	String (SFX File Path)	The sound the projectile makes while in its flight.
explosion	Explosion Datablock	The explosion effect to be spawned when the projectile collides with any surface (excluding liquid)
waterExplosion	Explosion Datablock	The explosion effect to be spawned when the projectile collides with any surface while also being submerged in a liquid.
splash	Splash Datablock	The splash effect to be spawned when the projectile enters or exits a liquid surface.

decal	Decal Datablock	The decal instance to be spawned at the location of the projectile's impact.
lightDesc	Light Description Datablock	The light instance to attach to the projectile instance while it remains active.
isBallistic	Boolean	Should this projectile be affected by gravity or should it not?
velInheritFactor	Floating Point Number	How much the projectile should be affected by the velocity of the source that created it? A value between 0.0 and 1.0 creates the best effect.
muzzleVelocity	Floating Point Number	The amount of force the projectile receives from leaving the muzzle of a weapon image.
impactForce	Floating Point Number	How much force is created on the object that is struck by this projectile?
lifetime	Integer	The amount of time (in milliseconds) a projectile is allowed to remain active before being deleted.
armingDelay	Integer	The amount of time (in milliseconds) before a projectile becomes "live" and can explode or create an impact.
fadeDelay	Integer	The amount of time (in milliseconds) before the projectile begins to "fade" and eventually disappear, this value must be less than lifetime .
bounceElasticity	Floating Point Number	A scale of post-bounce velocity that is applied to a non-exploding ballistic projectile. This velocity is applied after bounceFriction is taken into affect.
bounceFriction	Floating Point Number	A factor of how much to reduce the post-bounce velocity applied to a non-exploding ballistic projectile.
gravityMod	Floating Point Number	A factor of how much gravity affects the projectile (downward force). Treated as a multiple of the standard gravity of 9.81m/s.

So these are the fields that are readily available to your projectile instances in the engine. There are a few addon packs to the engine out there (some created by myself) that add additional projectile

types with their own uniquely defined fields to affect the properties of the projectile instance as it applies to it, for more reference on those fields, refer to the in-game datablock editor as it pertains to these individual datablock instances.

Functions for Projectiles

Now let's talk about the two callback functions that are available for your use when it comes to the projectile instances.

onExplode

The first of the two callbacks is the **onExplode** callback which exists on the projectile's datablock instance. This callback is obviously triggered when the projectile instance explodes in the world. Here's a basic example including its parameters:

```
function ProjectileData::onExplode(%this, %projectile, %position, %fadeValue) {  
    //Code Here.  
}
```

This kind of functioning can be combined with a container radius search (topic coming up) to apply after-damage effects to objects that are hit by the initial blast of the projectile instance.

onCollision

The other callback is the **onCollision** callback which is triggered for both exploding and non-exploding projectiles when the projectile instance strikes any object surface (excluding water):

```
function ProjectileData::onCollision(%this, %projectile, %hitObject, %fadeValue, %position, %impactNormal) {  
    //Code Here.  
}
```

Again we have some useful parameters that are in place to provide some after-impact scripting such as damage, effects, and much more.

Obviously to create unique projectiles, you'll likely need to use some creative mathematics which require you to write a customized **onFire** function to capture the projectile object reference variable as that is created, and I advise you to experiment with some nifty mathematics, try to create the seeking projectile, or a MIRV style projectile where the initial blast splits into an array of mini-bombs, the possibilities are endless, and by combining these two topics together, you can create a vast array of cool weapons with all forms of unique properties for your players to use in their game experience.

Now, before we move on to the next game object topic, we'll need to cover a few brand new topics in the scripting language to introduce some brand new features and complex mechanics used to perform some very specific tasks as needed by our next object.

Additional Scripting Topics

Before we move on to some of the AI topics of the guide, we need to introduce some advanced scripting topics that are used to detect objects given two points, or a specific position and a radius.

There are two individual operations we will be using to perform these checks and each of them require using a special type of code called a **typemask** which defines an object based on how it exists in the world itself.

Object Typemasks, First Applications of Bitwise Operators

A **typemask** as just defined is used to define an object based on its properties as it pertains to the object world. The engine, being 32-bit in nature, only allows a total of 32 typemasks to exist at one point in time, and a majority of them are currently being used, however current developments for the engine are suggesting that this may change at some point in the future for the better.

These typemasks are defined internally and are associated by means of global variable definitions for your usage in script files. Here are the type definitions:

Table of Typemasks

Typemask Variable	What Objects Are Covered
\$TypeMasks::DefaultObjectType	Any object that is inherited from the SceneObject class can be associated with this typemask.
\$TypeMasks::StaticObjectType	Any object that inherits from SceneObject that is not allowed to move (other than setTransform) in the world.
\$TypeMasks::EnvironmentObjectType	Objects that exist in the environment (Sky, Clouds, Forest) are classified in this group.
\$TypeMasks::TerrainObjectType	Terrain objects are classified as this typemask.
\$TypeMasks::WaterObjectType	Water objects and their derivations are classified as this typemask.
\$TypeMasks::TriggerObjectType	Invisible trigger volume objects are classified as this typemask (See Triggers section from earlier for these objects).
\$TypeMasks::MarkerObjectType	Marker objects such as mission markers and waypoints classify under this typemask.
\$TypeMasks::LightObjectType	Objects that exist as light emitters are classified under this typemask.
\$TypeMasks::ZoneObjectType	An object that manages zone instances (These are automatically instanced by the SceneZoneSpaceManager class instance)
\$TypeMasks::StaticShapeObjectType	Any object that is defined to have one or more solid and renderable geometries that exist as part of the static (non-moving) level are classified here.
\$TypeMasks::DynamicShapeObjectType	Any object that is defined to have one or more solid and renderable geometries that exist as part of the dynamic (moving) level are classified here.
\$TypeMasks::GameBaseObjectType	Any object that derives from the GameBase class is defined to have this typemask.
\$TypeMasks::GameBaseHiFiObjectType	Any object that derives from the GameBase class and is internally defined to use the HiFi Networking module is defined here.

\$TypeMasks::ShapeBaseObjectType	Any object that is derived from the ShapeBase class is defined to have this typemask.
\$TypeMasks::CameraObjectType	Any object that is a camera object has this typemask associated with it.
\$TypeMasks::PlayerObjectType	Players and AIPlayer instances are classified as this typemask.
\$TypeMasks::ItemObjectType	Any object that can be picked up as an item is defined under this typemask.
\$TypeMasks::VehicleObjectType	Any object that exists under one of the Vehicle classes is defined as this typemask.
\$TypeMasks::VehicleBlockerObjectType	Any object that exists to block vehicles is defined as this typemask.
\$TypeMasks::ProjectileObjectType	All projectile instances are classified as this typemask.
\$TypeMasks::ExplosionObjectType	All explosion effect objects are defined to use this typemask.
\$TypeMasks::CorpseObjectType	All player corpse objects are defined to use this typemask.
\$TypeMasks::DebrisObjectType	All debris objects are defined to use this typemask.
\$TypeMasks::PhysicalZoneObjectType	All physical zone instances (Chapter 3) are defined as this typemask.

So as you can see, there's quite a few to select from, but what happens if you want to use multiple typemasks at one time? I gave a little hint at the beginning of this section. The engine defines these typemasks as a 32-bit number, specifically a 2^n value, where n is the number used by each mask, starting at zero and moving up one at a time. This allows us to make use for the first time, of the bitwise operators in the engine.

Using Multiple Typemasks

This brings up a problem when you want to search by a typemask in the event you want to look for multiple instances at one time. But fear not, because these typemasks are defined as numerical bits moving up in a sequence, we can make use of the powerful bitwise operators to use multiple typemasks. Here's how we'll do it.

Let's assume I have two applications to do here. In the first instance, I need to find an object that exists as both a ShapeBaseObjectType and a VehicleObjectType, and then in the second case, I'll look for objects that classify as either of the two. Now common sense from what you've seen would tell you to do something like this:

```
function multipleTypemaskExample1(%object) {
    %typemask = %object.getType();
    if(%typemask == $TypeMasks::ShapeBaseObjectType && %typemask == $TypeMasks::VehicleObjectType) {
        //Do something?
    }
}
```

Well, actually this won't work at all. Because if the object exists as both typemasks, then by definition the binary will be applied as a **bitwise and** of the two instances, so the correct way to search for this specific instance is to do something like this:

```
function multipleTypemaskExample2(%object) {  
    %typemask = %object.getType();  
  
    if(%typemask & ($TypeMasks::ShapeBaseObjectType | $TypeMasks::VehicleObjectType)) {  
        //Do something?  
    }  
}
```

Confused yet? Good, so let's explain what exactly is going on here. Since the typemask itself already exists as a binary value (**%typemask**) we can use the bitwise and operator to compare it to the result of interest instead of the comparison equal to operator. As for the inside of the parenthesis, if you remember, we're validating that the object of interest exists as both typemasks and because bitwise and checks if the bits are similar that would do nothing but to negate the result of interest, therefore we use bitwise or on the inside to set all of the bits we need to 1, then when the two typemasks are validated by the outer bitwise and operator, it will confirm that the typemask exists as both of the two inner typemasks. If the typemasks match, bitwise and will evaluate the binary string to end in a 1, therefore making the result "true", and if they don't match, the binary string will end in a 0, or "false".

Testing for an object that exists under any of the following is just a change of one thing, the outside bitwise and operator to become a bitwise or operator instead.

```
function multipleTypemaskExample3(%object) {  
    %typemask = %object.getType();  
  
    if(%typemask | ($TypeMasks::ShapeBaseObjectType | $TypeMasks::VehicleObjectType)) {  
        //Do something?  
    }  
}
```

The exact same thing as last time happens on the inside parenthesis, but the main difference with this application is that when the outside is validated by the engine it will see if any of the bits needed are 1, and if that is the case, the binary string will evaluate to 1 or "true", and if none of them match, the result will be false.

There are numerous applications for using the typemasks of an object, but the most prevalent application comes from something called a container search function.

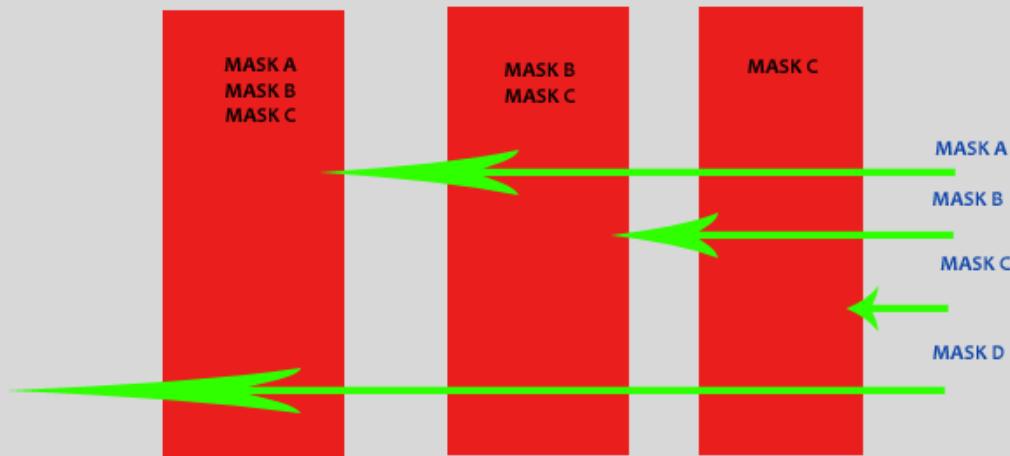
Raycasts & Container Searches

The next topic I'm going to introduce will cover the advanced operation of finding an object based on an input position and another parameter. This is a very useful operation when doing things like checking for a line of sight, to a very basic operation like a radius check to see if an object is inside some

kind of an AoE damage sphere. Either way the two topics here will easily expand your arsenal of tools for the better.

Raycast

The first of the tools is called a **raycast**. A raycast is a point-to-point search where the user defines a starting point, an ending point, and the target mask to search for, and the engine performs a test along the path to see if any objects of the specified type are in the way of the “casted” ray instance, hence the name. To see how this object filtering works, assume we have three objects, each having their own set instances of typemasks, and then assume we do four raycast calls, each having their own individual search masks, this is how it would work:



The above image should demonstrate how objects containing different masks would essentially line up to their individual raycast calls. The syntax and setup of a raycast is easy enough to understand, here's a quick example:

```
function raycastExampleOne(%startPoint, %endPoint, %searchMask) {  
    %raycast = containerRaycast(%startPoint, %endPoint, %searchMask, -1, false);  
    %tObj = getWord(%raycast, 0);  
    if(isObject(%tObj)) {  
        %tPos = getWords(%raycast, 1, 3);  
        %tNorm = getWords(%raycast, 4, 3);  
    }  
}
```

So here's the basic format of the raycast function. The first two parameters of the call are the starting and ending positions, respectively. Next up, you have the search mask you want to look by (Note, you can have multiple masks here, just use the bitwise or (|) to separate the masks). The fourth parameter is optional, and you can place one object id in there to omit that object from the search, and finally the last parameter is a boolean to tell the engine whether or not to use the search on the client end (true), or the server end (false). We'll get to the differences in Chapter 12.

What is returned is based on the output of the function itself. If no object is found, it will return a single value (-1) which can be tripped off by the **isObject** function that I taught back in Chapter 8 (see the above example). If an object is found however, it will return a seven word (Chapter 7) string. The first word will be the object id, the next three words is the position the raycast hit the object found, and the last three words is the position at which the raycast struck the object, normal (perpendicular) to its actual relative world-space position.

There are many very useful applications for a raycast as you can imagine, they can be employed almost everywhere, from weapons all the way to player to player detection and LOS checks between the players. Another very good example is when you want to use a raycast to check if any object is seen out to a certain distance from another object, let's look at how you do that:

```
function raycastExampleTwo(%object, %distanceFromObject, %searchMask) {  
    %startPoint = %object.getEyePoint();  
    %eyeVec = %object.getEyeVector();  
    %endPoint = vectorAdd(%startPoint, vectorScale(%eyeVec, %distanceFromObject));  
    %raycast = containerRaycast(%startPoint, %endPoint, %searchMask, -1, false);  
    %tObj = getWord(%raycast, 0);  
    if(isObject(%tObj)) {  
        %tPos = getWords(%raycast, 1, 3);  
        %tNorm = getWords(%raycast, 4, 3);  
    }  
}
```

And based on the application of your function, you can swap out the **getEyePoint** and **getEyeVector** functions for your own functions if you need them, for example if it's a weapon instance you can instead use the **getMuzzlePoint** and **getMuzzleVector** functions for the weapons.

Container Searches

The other type of object searching that you have at your disposal is a container search. There are two sub-types of the container search at your disposal. The first is the container radius search, which can be used to find all objects of a certain type within a defined radius of a given point. And the other type is the container type search, which will find all objects of the defined type on the map. These two searches are loop-based searches, because they pull each object from the list individually until the list is empty.

To start a radius search, you use the **initContainerRadiusSearch** function. The parameters for this function is the position to search from, the radius you want to search from the position, the target mask, and a boolean as to whether or not to use the client-container to search for objects. The other search begins with the **initContainerTypeSearch** function in which you provide the typemask and the boolean to whether or not to use the client container.

The next part of the container search is the loop-powered functions to actually navigate through the list. How it works is when you use one of the above functions, a temporary list is populated with the objects found by the search, to actually access this list, you use the **containerSearchNext** function. This function call will pull the next item from the list. When the call is triggered again, the object you were on is removed from the list, and the next object is pulled. This repeats until all of the objects have been ran through, and **NULL (0)** will return from the call.

There are two asset functions that are provided to you by the container search functions, and these are only usable by the radius search function. The first of the two is the **containerSearchCurrDist** function which returns the distance of the object the list is currently on from the initialization point of the radius search. The other function is the **containerSearchCurrRadiusDist** function which functions similarly to the other function, but the point in which is used to calculate the distance, is the point on the object that is the closest to the center, instead of the center of the object itself.

So now that you have the list of the functions, let's look at some examples of these functions being used.

```
function radiusSearchExample(%startPoint, %radius, %searchMask) {  
    initContainerRadiusSearch(%startPoint, %radius, %searchMask, false);  
    while(%tObj = containerSearchNext()) {  
        //Do something?  
    }  
}
```

So this is the most generic form of the radius search function, complete with an object fetching loop. Most of your radius searches will be in this form, where you can “check” for objects this way. The other type of search was the container type search, which will find all objects on the map of a specific typemask:

```
function typeSearchExample(%searchMask) {  
    initContainerTypeSearch(%searchMask, false);  
    while(%tObj = containerSearchNext()) {  
        //Do something?  
    }  
}
```

And here is the generic form of the type search function. As you can see both of them make use of a loop to actually go through the list and pull all of the objects from it for your use in other places.

So these are your search tools available for use in the engine. Now that you have these tools at your disposal we can move on to the next topic for the guide, which will cover the AI aspect of your game.

AI Players

AI, or artificial intelligence (not talking Terminator stuff here, unless you're making a Terminator bot) is basically a player that is controlled by the computer (or the server) in which you program instructions into the game and the player executes these instructions. There are two types of AI instances that the engine provides, but we will only focus on one of the two as the other is a more outdated form of AI. The AI type we are going to use is called the **AIPlayer** or basically a player object that has some additional functioning allowing you to program where it moves and what it does.

Relating AIPlayer to Player

First thing's first though, let's actually talk a little bit about the AIPlayer object, and you'll find some vast similarities between it, and the normal player because the AIPlayer class inherits directly from the Player class, which as you now know, means all of the functions and data fields available to the Player class are also available to the AIPlayer class.

There are a few key differences between the two objects however. First and foremost is the fact that the AIPlayer object has no direct methods for reading movement control data as you would usually see in a standard object, this object instead streams a constant line of updates of its position and rotation which is used for more precise movement and to make sure that each player in the game has "lag-free" AI instances moving about the map.

Secondly, this isn't a "do-all" solution to make AI's who will behave like normal player instances in say a multiplayer engagement. You'll soon learn in Chapter 12 about a "Client" instance and how that's different than your player instance. An AIPlayer for instance, is one individual player object, once it dies, the information associated to it is also deleted. So to actually make a "AI Player" for instance in terms of a constant player who can respawn in a multiplayer scenario, you need to create a custom client object that stores the important information of the AI to be kept after the death of the player instance, and to actually respawn it.

How to spawn an AI Player

So now that you know just a little bit about AIPlayer instances, let's actually look at how you spawn them. Now, I would do another nice table for information on the AIPlayer datablock, but it's the same datablock as the Player class, so just refer to that above. As for the AIPlayer itself, you're given three additional fields that are mainly used for movement controls. **mMoveTolerance** is a floating point number that tells the player how "close" it needs to be to its destination in order to be "at it". **moveStuckTolerance** is a floating point number that can be triggered when the AIPlayer is moving, if the player doesn't move at least this amount, it's considered stuck and a callback will be triggered. Finally we have **moveStuckTestDelay** which is an integer value of how many ticks to wait before testing if the player is "stuck". But these are per-player values on an AI instance; let's now look at an example function of how to spawn the AIPlayer itself:

```
function AISpawnExample(%position) {  
    %aiPlayer = new AIPlayer() {  
        datablock = GenericAI;  
    };  
    %aiPlayer.setTransform(%position SPC @"0 0 0");  
    return %aiPlayer;  
}
```

If you call this function, a player object will spawn at the specified position and just sit there all nice and pretty. That's because this AI doesn't have any instructions as to what it needs to do, and until it receives any instructions, it will continue to sit there and do nothing for you. So since we now know enough about programming to know that instructions means function calls, let's actually look at some of the functions the engine has for the AIPlayer object.

Functions for AIPlayer Instances

So in order to actually have the AIPlayer do anything useful, you'll need to use the functions provided to you, or create some new functions based off of these functions. The first half of these will be standard function calls, and the second half will be callbacks. I'll inform you of the change point.

stop

This is a very self-explanatory and easy to use function. Simply call this function to make the AIPlayer stop what it's doing, no parameters and no return values:

```
function AIStop(%ai) {  
    %ai.stop();  
}
```

clearAim

The clearAim function is used to remove the object instance from the AIPlayer which is currently being "aimed" at. Like the stop function, there are no parameters and return values, simply push this function to the object to make it clear the aim instance.

setMoveSpeed

The setMoveSpeed function is used to tell the player how fast (from normal speed) it needs to move. You send a single floating point number (0.0 – 1.0) to this function where the number is a multiplier where 1.0 is normal and 0.0 is no movement as a multiplier of the datablocks defined moveSpeed variable. There is no return value with this function.

getMoveSpeed

The getMoveSpeed function is used to fetch the current multiplier that is set by the setMoveSpeed function. By default the value is 1.0 on an AIPlayer object instance. You don't send any parameters to this function and it returns a floating point number that is between 0.0 and 1.0.

setMoveDestination

The first “important” function of use to you is the **setMoveDestination** function. This is used to tell the AIPlayer where it needs to currently move to. There are two parameters for this function, the first parameter is the position and the second parameter is a boolean value where true means the AI is to slow down when it gets close to the target point, and false means to abruptly stop when it reaches the destination. This function has no associated pathfinding, the bot will run in a straight line from its current position to the target point regardless of objects in the way. This function does not return any values.

getMoveDestination

This function is used to pull the current movement destination from the AIPlayer instance. There are no parameters and it will return the position the AI is moving to, or “0 0 0” if there is no set destination for the AIPlayer instance.

setAimLocation

The **setAimLocation** function is used to tell the AIPlayer instance where it needs to “aim” or look at. You provide this function with a single variable; the position where the AI needs to look and the engine takes care of the rest. There are no return values for this function call.

getAimLocation

The **getAimLocation** pulls the position of what the AIPlayer is currently looking at, you don’t send any parameters to this function and it either returns the position in which was set to be aimed at, or if there is no aim location set, it will return a raycast position value from the AI’s LOS.

setAimObject

The **setAimObject** function is used when the AI needs to aim at a target object. There are two parameters for this function, the first is the object reference variable as to the object that needs to be targeted, and the second optional parameter is an aim offset from the center of the object that the AIPlayer should aim at (FYI, it’s about a “0 0 0.5” offset to make those evil aimbot AI bots ;)). There is no return value for this function

getAimObject

Finally, we have the **getAimObject** function, which is used to return the object the AIPlayer is currently aiming at. If the AI is not aiming at anything, the function will return -1. There are no parameters for this function.

We’ve reached the point now at where we will switch to covering the callback functions for the AIPlayer class. The AIPlayer doesn’t have a unique datablock, but these callbacks exist on the datablock that is defined to be used by an AIPlayer object.

onReachDestination

The first of the callbacks defined for the AIPlayer class is the **onReachDestination** callback which is triggered as a result of the **setMoveDestination** function. When the player gets close enough to the

destination as defined by the **mMoveTolerance** variable, this callback is triggered. The callback variables are the datablock instance, and the AIPlayer instance, here's how it looks:

```
function GenericAI::onReachDestination(%this, %aiObj) {  
    //Do something  
}
```

onMoveStuck

The next callback is the **onMoveStuck** callback which can be triggered when the AIPlayer is moving. If the AI is determined to be stuck by means of the **moveStuckTolerance** and the **moveStuckTestDelay** variables this callback is triggered on the AIPlayer instance. The variables are the same as the **onReachDestination** callback.

onTargetEnterLOS

The next callback is a utility function when the AI is set to aim at an object, this function will be triggered when the AI can see the object it is supposed to be aiming at with no objects blocking the line of sight. This function has the same variables as the other two callbacks we've looked at.

onTargetExitLOS

The last callback function defined by the AIPlayer class is the opposite of the **onTargetEnterLOS** function and this one is triggered when the AI did have a sight on the object it was supposed to be looking at and then lost sight of the target object. This function also has the same variables as the other callbacks we've looked at.

So this covers the object instance itself and the methods available to it. Now, as mentioned beforehand, you actually need to set up the player instance to move and do tasks, so let's take a peek at how you actually go about doing this now.

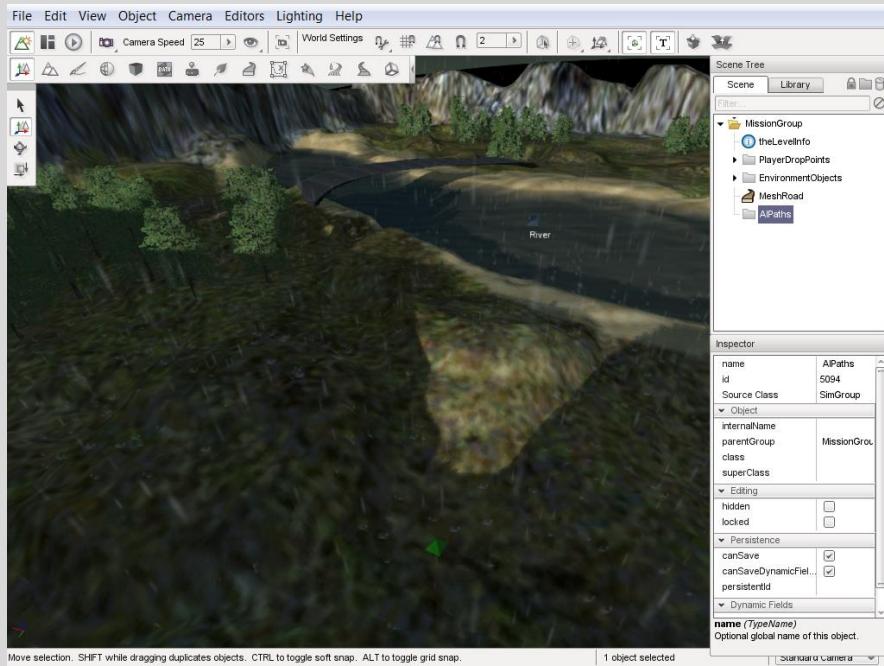
AI Path Object, Routes

The next thing we're going to discuss in the guide is how to actually have our AIPlayer instance do something in the world. Now, if you're using the Full template for the engine then you have access to a few pieces of powerful tools and functions to handle a good portion of this. This part of the Chapter will cover how to set up Path objects, and how to get our AI instances to follow the path.

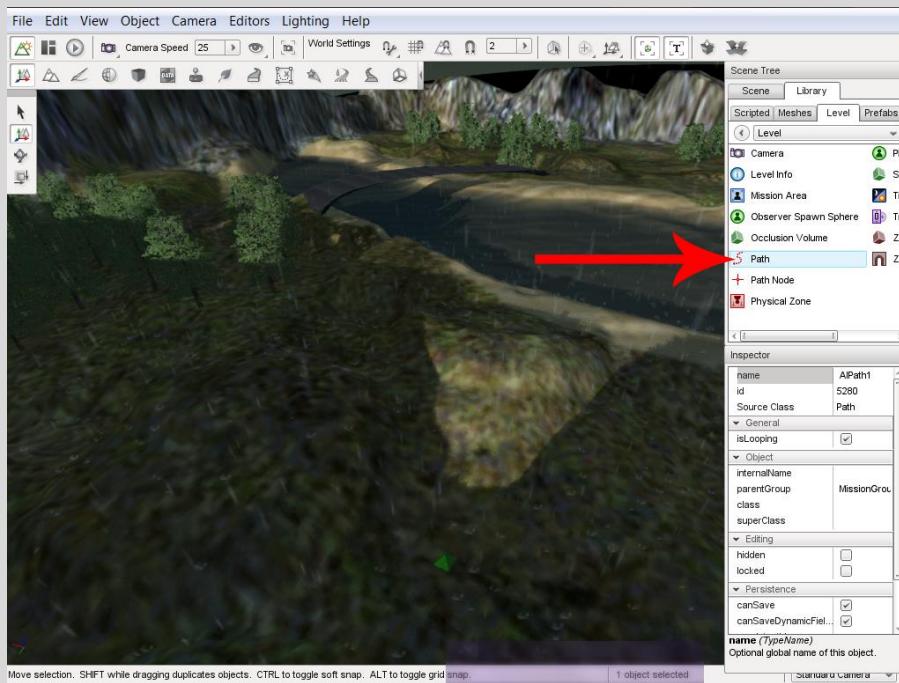
Setting up a Path

We're going to start by introducing the Path object. I frequently use this kind of a setup (pre-defined path) when I want to create pre-scripted AI objects that do the same thing every mission, for instance, your single-player AI that runs out of a specific door to shoot at you, or runs away before exploding, or something along those lines. These are the pre-defined AI instances that will follow the path you make and then stop.

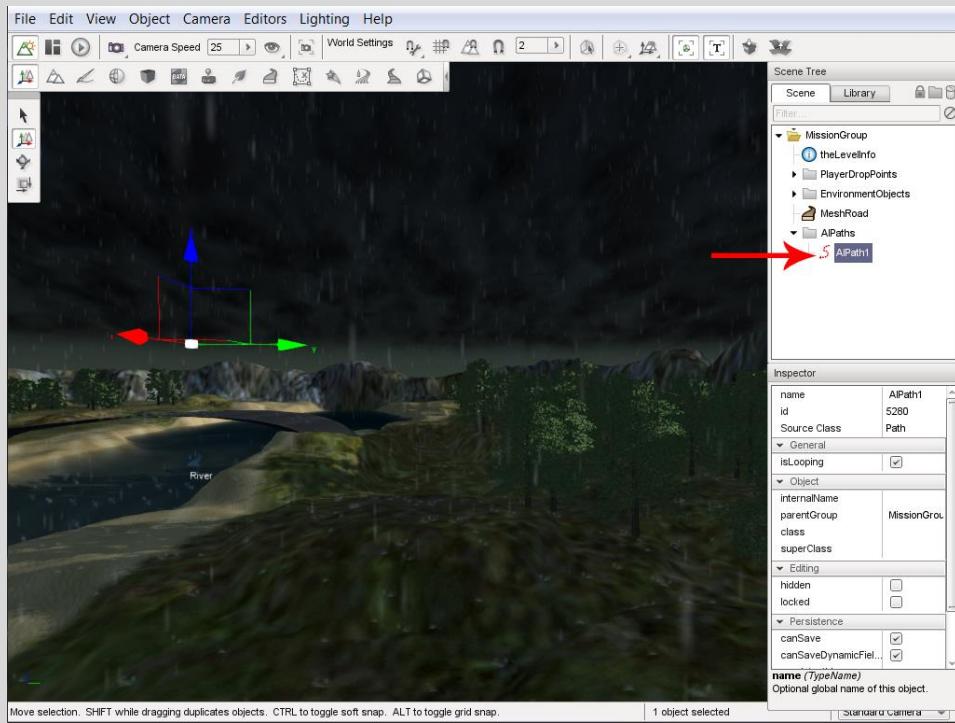
My best approach is to create a SimGroup to store these path instances in. Since we're dealing with AIPlayer objects, I name this specific group 'AIPaths'.



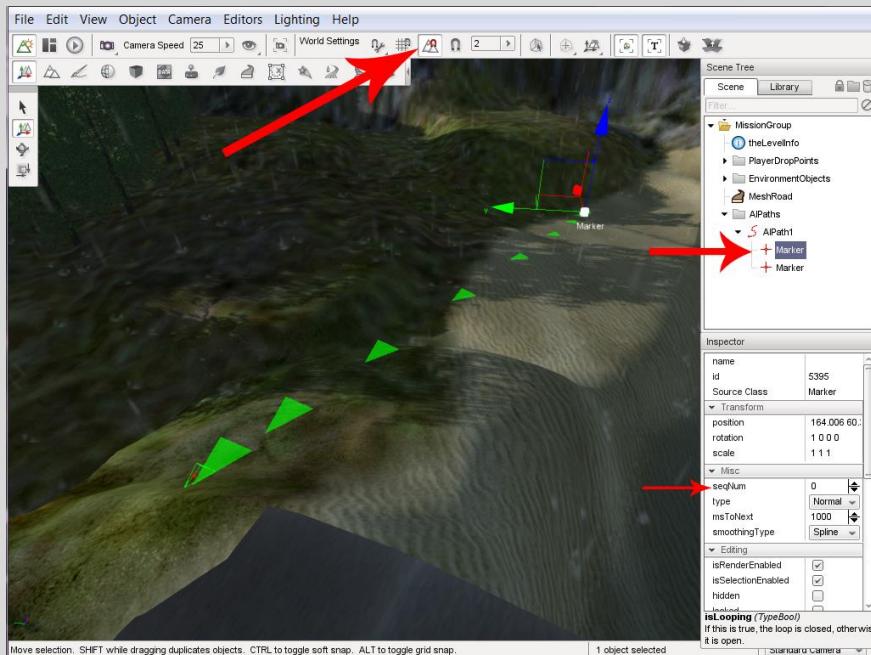
The next step is to actually create the Path object itself. Now, a Path object is actually a unique SimGroup instance that has a few extra fields that are defined on it to make creating Path instances a little easier to work with. So, go ahead and create your path, you'll find it in the Level folder of the Library. I usually number my paths to easily remember them, since it's for an AI instance, I name it accordingly: AIPath1.



The next thing to do should be obvious enough, put the AIPath1 object into the SimGroup you created just a few moments ago, also be sure to de-check the **isLooping** parameter from the path:



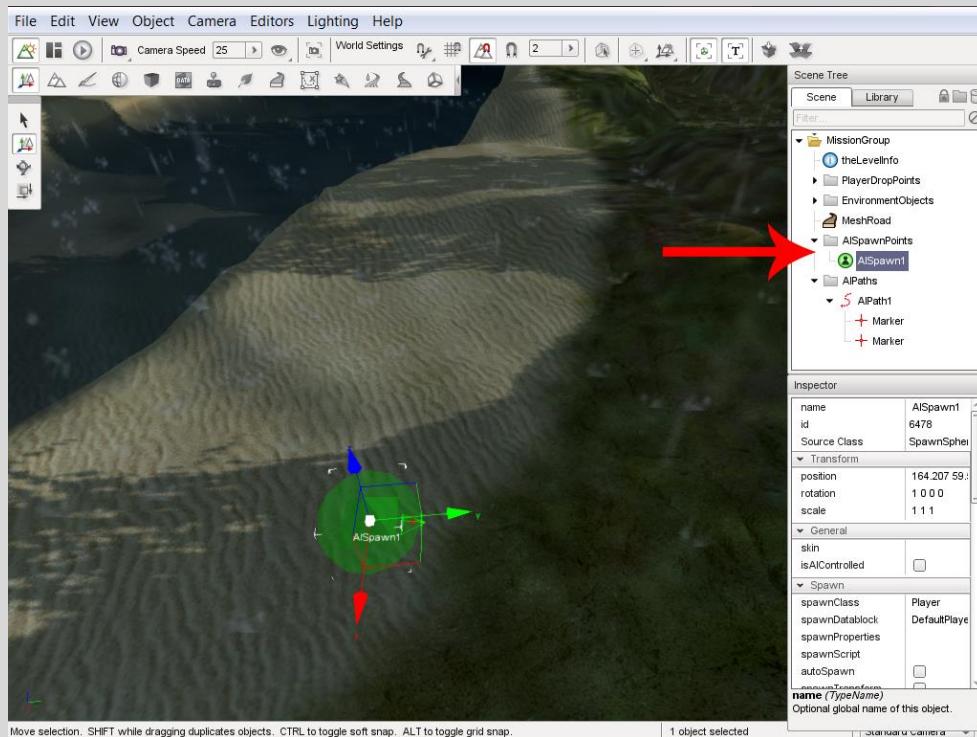
So now that you have the Path instance defined, we actually need to make the individual PathNodes that will be used by the AI instance. If you remember all the way back to Chapter 9, we talked about how to use mathematics to define a curve between two or more points, that's exactly how this function works in the engine.



So there are a few things going on here. My first arrow points to the snap to terrain tool, you'll always want your nodes touching the terrain when making an AI Path, otherwise the AI will not be able

to reach it and become permanently stuck. Next thing is obvious enough, put all of your PathNode objects into the Path Object (Remember, it's essentially a SimGroup). Finally you'll notice the **seqNum** parameter is being pointed to. For your path instances, you'll want the first node to be defined as **0** and then you'll increase that number by one for each additional node you make. The arrows will point to the path instance to show you if you've got it right. Also, a little trick, you can also rotate the arrows to get the AI to be moving in that direction (Unless the AI is aiming at an object).

Finally, we need to actually have our AI spawn near the path, so what I did was create another spawn sphere instance, and named it **AISpawn1**, and placed it in its own SimGroup instance:



Now, once we make our spawning function, we'll have the AI be sitting next to the start of the path and ready to go.

How to get an AIPlayer to follow the Path

So now that we have a path instance, let's actually get our bot to follow the path. But, before we continue, let's start by making our own custom AI datablock, and give it a few functions to use here:

```
datablock PlayerData(TutorialAI : DefaultPlayerData) {
    shootingDelay = 200;
};

function TutorialAI::onReachDestination(%this, %obj) {
    if (%obj.path != "") {
        if (%obj.currentNode == %obj.targetNode) {
```

```
%this.onEndOfPath(%obj, %obj.path);
}

else {
    %obj.moveToNextNode();
}

}

function TutorialAI::onMoveStuck(%this, %obj) {
    //Do Something?
}

function TutorialAI::onEndOfPath(%this, %obj, %path) {
    %obj.aiDecide();
}

function AITutorialDemo(%position) {
    %aiPlayer = new AIPlayer() {
        datablock = TutorialAI;
    };
    %aiPlayer.setTransform(%position SPC @"0 0 0");
    return %aiPlayer;
}
```

So what we have here is a standard set of basic functions for spawning the AI and the necessary callbacks for when we have the AI move along the path, but if you study the code, you'll see that our bot still won't be going anywhere or doing anything just yet, so let's fix that:

```
function TutorialAI::followPath(%this, %path, %node) {
    if (!isObject(%path)) {
        %this.path = "";
        return;
    }
    if (%node > %path.getCount() - 1) {
        %this.targetNode = %path.getCount() - 1;
    }
    else {
        %this.targetNode = %node;
    }
}
```

```
if (%this.path $= %path) {
    %this.moveToNode(%this.currentNode);
}
else {
    %this.path = %path;
    %this.moveToNode(0);
}
}

function TutorialAI::moveToNextNode(%this) {
    if (%this.targetNode < 0 || %this.currentNode < %this.targetNode) {
        if (%this.currentNode < %this.path.getCount() - 1) {
            %this.moveToNode(%this.currentNode + 1);
        }
        else {
            %this.moveToNode(0);
        }
    }
    else {
        if (%this.currentNode == 0) {
            %this.moveToNode(%this.path.getCount() - 1);
        }
        else {
            %this.moveToNode(%this.currentNode - 1);
        }
    }
}

function TutorialAI::moveToNode(%this, %index) {
    // Move to the given path node index
    %this.currentNode = %index;
    %node = %this.path.getObject(%index);
    %this.setMoveDestination(%node.getTransform());
}
```

So now we have some basic functioning available to have our AIPlayer move on command and follow the path until the end, then a callback will be triggered saying the AI has reached the end of the path and now needs to do something. So let's tie everything together now to spawn the AI and to have it follow the path we made:

```
function DoDemoLevelAI() {  
  
    %spawnPosition = getWords(AISpawn1.getTransform(), 0, 2);  
  
    %demoAI = AITutorialDemo(%spawnPosition);  
  
    %demoAI.followPath(AIPath1, -1);  
  
}
```

Call the function, and watch the AI Object spawn in the sphere and the run along your path until it reaches the end, and then stop there. So this is how you create static AI routes that will be persistent every time you run the level. Now you could further expand this functioning to have the AI run to the end of the path, and then start shooting at the nearest player, and now that you have tools such as Raycasting and the ability to make your own weapons, you should have no problem accomplishing this. So, now we know how to make static routes, but let's say you want to have a little more advances in your AI, and want to make a Dynamic route that changes based on the AI's current position, and it's goal, while also taking into effect the map and it's geometries.

Introduction to Recast

Enter Recast, an AI navigation and pathfinding system. The actual codebase was created by Mikko Mononen and has since been released as an open source project available for games and game engines of all shapes and forms. The premises of the system is easy enough, the geometry of the world is loaded into a format that is recognized by the system and a few parameters such as the size and thresholds of movement is sent to the system, and in turn it creates a navigation mesh that can then be used by path objects that are adjusted on the fly by the engine.

The actual work of converting everything and getting it working in the engine is credited to Daniel Buckmaster and a few others who worked on individual components of the system's necessary conversions. Daniel actually created a custom system using this that has since been released as open source to everyone, you can check that out here, while we await the potential of seeing it merged to the actual engine itself in a future version: <https://github.com/eightyeight/walkabout/>

Using the Recast System in T3D

So how do we actually use the Recast system in the engine? The first thing you need to do is to make sure that your engine instance is using the recast module, which can be done so by enabling the navigation module in the project configuration:

```
includeModule( 'navigation' );
```

Once you've enabled the module and rebuilt your solution, you should have access to two new objects in the world editor. The **NavMesh** and **NavPath** objects. The **NavMesh** object is used to actually generate the recast paths by means of converting the world-space geometry into a format recognized by the system, while the **NavPath** object is used to actually take this format, and spit out a path that the AI player will follow.

To actually go the mesh generation properly, make sure the mesh covers all of the map you want to be treated as “walkable space”, then you can tweak the parameters of the mesh to fine tune it to fit your project. The mesh will generate as you make changes, you can turn on the debug rendering tools to see visually the paths generated around your object instances, and even in some cases, on and through them. Once you’ve got it the way you want it to, make sure you set the data file parameter so the information can be quickly saved and re-accessed by the engine for quick use when you actually play your game.

Now, you’ll want to create another AIPlayer instance (datablock and functions), since the possibilities of this system are determinant on your application of it, I’m not actually going to cover the functions to be written for the AI Player, I’ll leave that to you. But since some of these are more advanced in nature, here’s a three part guide written by Steve Acaster on how to set up the AI using Recast navigation: <http://www.garagegames.com/community/resources/view/21604>, do note that some of the functions are a little outdated, so you’ll need to adapt it based on the most current version of the Recast system and the functions that are available to you by means of the **NavMesh** and **NavPath** objects, so let’s talk about those functions now.

NavMesh Functions

The NavMesh object doesn’t define functions that pertain to the AIPlayer itself, but are defined as functions that are needed by the mesh in order to save, load, and otherwise use the mesh instance in the engine.

build

The first function we’re going to talk about is the **build** function, which is used to actually convert the world’s geometry using the Recast system into a format that the engine can use to make paths. There are two parameters on this object function, both are optional parameters. The first parameter is a boolean that states if the building process should occur using a background thread (save performance) or if it should build right now (less performance), set it to true to use the background and false to use the current thread. The second parameter is a boolean that says that when building is done, the information should be saved as a binary file to the **fileName** parameter of the NavMesh object. This function will return a boolean stating if the build was successful or not.

cancelBuild

The next function is the **cancelBuild** function. If you set the current build process to run as a background process, you can use the **cancelBuild** function to stop the current building of the navigation mesh instance. There are no parameters or return values for this function.

buildTiles

The next function is called **buildTiles** which is a function access to an internal method used by the build function. This function accepts a 6 point value string, which is defined as a 3-dimensional box. How you define this is start with the lowest corner of the box and provide the x, y, and z coordinates first and then you define three more values containing the “length” of each side outward from that position. This term is called the **extent** of the box. The function will then build the navigation mesh only in that box. There is no return value for this function.

load

The load function is used to load a pre-built navigation mesh that has been saved to a binary file back into the engine for use. To use this function, the navigation mesh object must be pointing to a file with its **fileName** parameter. There are no additional parameters for this function and it will return true if the load was successful and false if the loading failed.

save

The save function is used to save a built navigation mesh to a binary file object for future use. To use this function, the mesh object must have a **fileName** parameter set. The function has no additional parameters and returns true if the save was successful and false if it failed to save the information.

NavPath Functions

The **NavPath** object is where the magic happens in terms of the dynamic player movement. What you'll want to do is spawn an object for each AIPlayer instance and associate the AI to the path object, then when the AIPlayer dies, you can safely delete the object. Here are the functions available to the NavPath object to help you program your AI instances. To use the path object, you pass the reference object of the **NavMesh** to the **mesh** variable of the **NavPath** and then provide two position variables, **to** and **from** to tell the AI where it needs to start and where it needs to end.

replan

The first function is the **replan** function which is used to generate the path instance between the specified **to** and **from** variables. This process is automatic when the path is spawned, but if you need to rapidly update the planned path, call this function. There are no parameters and the function returns a boolean value based on if the call to the function succeeded or not.

getCount

The **getCount** function is used to return the amount of node objects that were generated along the navigation path instance. You can use this as part of a loop or in conjunction with the followPath function we created earlier. There are no parameters sent to this function and it returns an integer number containing the amount of nodes in the sequence.

getNode

The **getNode** function is used to fetch the position of the node instances generated by the path. While these individual node as are not actually “treated” as a spawned object in the engine it used to be that way, this function has been recently adapted to return just the point, instead of an object. You provide one variable, the index number, and the function returns the position of the node, or “0 0 0” if there was no node in that place.

getLength

The final function is the **getLength** function which is used to return the amount of distance the current path covers. You send no parameters to this function and it returns a floating point number containing the distance of the path.

Conclusion

While this guide covers the basic tools available to you, I highly recommend you to also refer to the engine reference documentation, or the recast source files in your engine **engine/source/navigation/x.h, x.cpp** which contains a list of pre-defined functions that are available to each of the objects that are added. I'll have much more on how to read, and even write these kind of things in Chapter 15, so if you need a quick guide to that, go ahead and skip down to there.

So there you have it, static paths by means of the Path object, and dynamic paths by means of the Recast addon. While each form of the AI routes and movement planning has its own level of coding difficulty associated with it, you should now have the premises of how to use the AIPlayer class to accomplish your AI solutions for your game project.

Strategies to create New Objects

Our last stop for Chapter 11 is going to be a revisiting of a topic we briefly covered all the way back in Chapter 8, but we're going to "upgrade" your knowledge of the topic now with some new concepts. We're going to talk about the process to create a new object instance in the engine using only TorqueScript. Obviously the most optimal way to do this is internally using C++, but since we don't have access to this at the moment, we'll have to wait until Chapter 18 until we actually do that.

Revisiting Object Inheritance

So way back in Chapter 8, I introduced you to the concept of **Object Inheritance**, which is the process of copying an existing object or class into a new class and "inheriting" all of the properties of the existing object into the new object. If you recall we had two types of inheritance that we covered. The first of which was the class inheritance by means of the **superclass** parameter:

```
function basicMultipleInheritanceExample_Two() {  
    %newObj = new ScriptObject()  
    class = "MyClass";  
    superclass = "ScriptObject"; //<- Demonstration purposes only.  
    numberVal = 10;  
    name = "Bob";  
    objType = "Basic";  
};  
%newObjTwo = new ScriptObject()  
class = "MySecondClass";  
superclass = "MyClass";  
numberVal = 20;  
coolField = true;  
};  
}
```

Our second example used the **inheritance operator** (`:`) to directly inherit an existing object into another one, and we've used this one quite a few times to load datablocks into other datablocks:

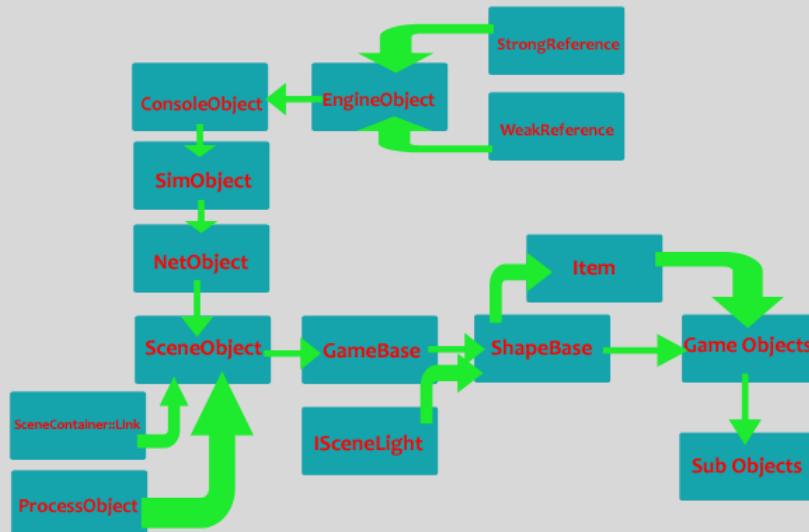
```
function directInheritanceExample() {
    %newObj = new ScriptObject(ObjectOne) {
        class = "MyClass";
        numberVal = 10;
        name = "Bob";
    };

    //Create the second instance, inheriting the properties of the first.
    %newObjTwo = new ScriptObject(ObjectTwo : ObjectOne) {
        class = "MyClass";
        numberVal2 = 20;
    }

    echo("Inherited Properties: "@%newObjTwo.numberVal@", Name: "@%newObjTwo.name");
    echo("Non-Inherited Properties: "@%newObjTwo.numberVal2);

}
```

The key part of this review is the fact to remember that inherited objects draw in the functions and properties of all of the classes that exist above it in the inheritance tree, you may recall that we began this chapter by introducing the large object tree that exists in this engine:



What you need to be aware of is that when you write your functions, that you need to be aware of existing functions and properties that exist due to inheritance alone, and to write your object functions based on the properties of the current object, and not the parent classes. Using the **dump**

command on your object instance will help you to identify existing parameters, functions, and other properties of your class.

Getting Everything Together

Once you've found the object to inherit, you'll write a function or a class definition that actually performs the inheritance and sets up the new class by means of the; you guessed it, the **class** variable. From that point, you then have access to the ability to define object functions under the new class definition that are unique to that class instance and will help to make in-game functioning much easier.

The important thing to recognize though is the right place and time to actually use this functioning. For instance, I'd create a SinglePlayerMission object instance to store the information of the player's progress in a single-player campaign mission when the game begins and it sees that we're in the single-player mode. It's all a challenge of simply identifying what needs to be done, and where the proper code needs to be inserted to create new objects and instances.

What I hope you've learned over these past few chapters is that a majority of the engine is actually run internally, and what we have are just entry points and callbacks to add some additional functioning to the engine itself. The scripting language, while powerful, cannot actually accomplish every task in the world as it pertains to your game, but it can get you moving in the right direction.

Final Remarks

Well, that was a long one! So, you should now have essentially all of the tools at your disposal to actually create your game worlds, and give them more life than what the world editor could give you alone. We also now have some new tools of the trade to actually create the player instances, the weapons they hold, and some artificial intelligence that will either be working with you, or against you in your gameplay.

So now that we've finished that, there's one last topic we need to cover before diving into the world of C++, and that is what happens when you put players with other players. Let's introduce the engine's Networking system.

Chapter 12: Introduction to Networking In Torque

Chapter Introduction

Chapter 12 is a very important chapter for this guide. Now that you have what I would call a complete understanding of the engine mechanics and the objects to be able to create your own objects in the engine, we need to turn away from it and move to one of the most important topics of the engine, which is its networking system. Torque brilliantly combines the client and server into your application to allow you to play both singleplayer and multiplayer games with a click of a button and here, you're actually going to learn how this engine's networking works and how to take advantage of the tools available to you for applications like communicating between client and server. This is going to be your last chapter of TorqueScript for a little while, so make sure you've got the programming thing down, because we're going to turn up the difficulty even further after this chapter.

Client versus Server, Instances

We start chapter 12 by finally getting to a topic that I briefly started on a long time ago, but couldn't really discuss due to the scope of knowledge at the time. So far, you've been under the assumption that the game loads your scripts and that all you need to do is pop open the console to run them. This is actually incorrect. Inside the game you have two game **instances**, and at any time one, the other, or both may be running on your machine. If the instance in which your script file is on does not exist at the time, then you're out of luck in terms of running the script.

Instances

So what am I rambling on about now? The main priority of the game is to act as either a **client instance**, or a **server instance**. In some events, when you're playing by yourself, you're running what is called a **listen server**, where your machine runs both jobs simultaneously. If you've done some exploring through the engine's files, you'd have noticed this a while back, how the scripts folder is split into client and server, this is where that topic comes into play.

So, what exactly does each instance do? That's very easy. A client instance is responsible for managing scripts and assets that are specific to each game client, basically objects like the game's GUIs or your keyboard commands are handled on the client instance. Certain texture and shader mappings are also handled on the client instance due to the way the engine handles the connection process (getting there). On the other end, the server instance is responsible for handling the datablocks, and the game's objects and all of the scripts relating to these datablocks and in-game objects.

With this knowledge in mind, you should now be able to place your scripts in the right place. So everything pertaining to objects that exist in the game, belong on the server side of things, and everything relating to GUIs or scripts that are run on the game client, belong in the client side of things. If you're still having trouble determining which end of the game a script file belongs in, you should look for cases that are similar to your script. If in the rare event, a script belongs on both instances, then be sure to load it in the **main.cs** file directly in the scripts folder.

Client & Server Loading

So, that's great and all, but how exactly are these files loaded? So another topic I briefly mentioned was right at the end of chapter 6 where I talked about how scripts were loaded into the engine. The primary script file in the engine is at the base directory in **main.cs**. This is where all of the launch parameters are executed to get the game instance running, but after that we fork it off to client and server where surprise, there's another **main.cs** file waiting for you. If you carefully read the **main.cs** file in the scripts folder, you'll notice right at the end of the initialize script, that we either initialize a client instance, or a server instance. This is where the game forks off individual functioning to the relevant **main.cs** files. For a quick reference, here's how it works:

- All game instances start as such, neither defined as client or server
- If the game instance is not spawning a dedicated server, it becomes a client instance
- Otherwise it becomes a server instance
- Client instances may become listen server instances by means of "hosting" a game instance. This will either be you starting a server from inside the game, or playing a single-player game.

So just because you're flagging the instance as a listen server doesn't mean that your game cannot be singleplayer. The engine just treats the application as so to properly split off functioning to a server or a client so neither will do tasks the other is not supposed to do.

With that in mind, let's actually learn how to use the networking system in the engine to do some pretty cool things.

Client Object: The GameConnection Class

To this point in time, you've been stuck in a very unilateral view of the engine, where objects seem to exist in both a client and a server instance, can be set up to be moved around, or interacted with by a player instance. But now here's the main question, what is actually controlling the player object? Let's introduce that concept now.

What is a client object?

How it actually works is a bit more like this. Every object in the engine that plays in the game is handled by the Sim classes, which is short for Simulation. Beyond that, we have the NetObject class, which grants objects the ability to transmit and receive data between the clients. Now, these individual client objects, exist under a class called **GameConnection**, which is another word for a "Living Player Instance", or just a "Player". While we do have the Player class, this is actually just another game object, specifically what could be called an Actor instance. Each client on the game's Server instance is assigned a GameConnection object to handle all of the input events and all of the transmissions. All of these instances are stored in a **SimGroup** called **ClientGroup** which can be used to loop through all of the connected clients.

It is these client objects, that are assigned control over other objects in the engine, from there the information is transmitted back to the individual clients to actually render the scene as needed from their perspective of things.

The Connection Process

So now that you know what exactly a GameConnection instance is at it pertains to the engine, let's actually talk about how it's created. Before we talk about the step-by-step process of creation, let's introduce what these two instances do for the game itself.

Server Roles

- Authority on Information: Data stored on the server has authoritative override permissions on all information stored on the client instances, this way if a client becomes out-of-synch, the server can override the information on it to bring it back in line.
- Collision & Physics: The server has control over collisions and physics of objects in the world and handles all of the calculations of these aspects.
- Datablocks: The server has all of the datablocks stored on its end to be transmitted for download to all clients that connect to it. These objects are treated as read-only instances for clients and both read and write for server instances.
- Ghosting: The server keeps track of all of the real objects in the game world, and gives copies of these objects as **ghosts** to all of the clients by means of its **scoping** role. The server keeps a master list of all available objects to use.
- Messaging: The server handles all messages sent to it by individual clients and has the authority to route them between clients. This covers commands and events like chat messages.
- Scoping: The server has the authority to have knowledge of the status of each client as it pertains to the simulation so it knows what needs to be transmitted to and from each client, this process saves on bandwidth transmission.
- Security: Finally, the server has authority over all client instances and has the power to adjust, and remove clients from the server instance with a single command.

Client Roles

- Input: The client instance can accept user input and transmit it to the server.
- Interpolation: As part of the **prediction** aspect of the client, the client is allowed to determine where things need to be between the time of updates from the server.
- Prediction: The client is allowed to predict what will happen to each object between server updates by means of the information it knows about the object (datablocks).
- Preferences: The client is allowed to hold important information that is to be kept common through all running instances.
- Rendering: The client is responsible for generating the game's scene and smaller physical calculations such as cloths, fluids, and particles.
- Sound: The client is responsible for handling SFX events as they are sent to the client instance.

The Connection Process, Step by Step

So now that you know the roles for the server and the client, let's show you how the connection process works, step by step.

- 1) Server Initialization: The first step in any connection process is to have the target server of interest running, and allowing connections to be made to the server.

- 2) Connect Event: The client interested in joining sends a connection request packet event to the targeted server, this event contains the connection request string which contains the information on the client such as the version of the game, and the password (if the server has one), the client also sends their authentication information packet (If applicable).
- 3) GameConnection::readConnectRequest(): The packet is then processed on the server end in the GameConnection class, which is the point at which the client's object is actually created on the server. This function then parses the connection request event to make sure that the fields are correct, and if there is a password, to make sure the password is correct. If there are any problems, the server generates a CHR_X error and sends it back to the client.
- 4) onConnect(): The executable then hands the functioning down to TorqueScript where the new client object and their fields are added to the server's information. If administrator privileges need to be granted, this is usually where this is done.
- 5) Mission Start Phase 1: Once the client is successfully connected to the server, the server initializes Phase 1 of information transmission, where the mission information and the datablock information and object target information is sent down to the client.
- 6) Mission Start Phase 2: The client then sends an acknowledgement packet to the server saying it has completed Phase 1, the server then begins Phase 2, where all of the Ghosting information is sent down to the client to begin the sequence of known versus unknown objects and to bring the client into synchronization with the game state.
- 7) Mission Start Phase 3: Finally, the client sends another acknowledgement packet to the server. The client then validates the integrity of all of the information sent to it by the server and performs the lighting operation on its end to begin the rendering sequence. After a few final checks, the client enters the game.

Beyond the connection process, the server then handles packet updates on objects within the scope of the client to make sure the client remains in synch with the server. The roles mentioned above are held constant. So, this is how the connection process works in the engine, now there's obviously some more fine details within each step, but the overall premises is that of the seven step process above. Now, let's talk about some of the functions available to the server when it comes to managing the individual client objects within the game.

Functions for the GameConnection Class

The GameConnection class, as mentioned earlier is another fancy term for a client instance, or basically a real-life player instance that is connected to the server. The server has an authoritative role over all of these object instances, therefore there are some functions that are readily available to be used to control the gameplay aspect of each individual client in the game. Now, there are a few functions that are in the list that would negatively impact the overall gameplay if improperly used, so I'm going to omit those from the guide to make sure you don't risk that problem.

setControlObject

So if you remember back a little while ago in Chapter 11, I gave you a similarly named function for the Player class. Well, that function was just a routing tool, this is the primary control function for the engine, as it tells the client what object it has control over at any given time. If you gave the client

control over a Player instance, you could use the player object's **setControlObject** function to pass the control information to another object without relinquishing control of the player object itself. However, using this function overrides every other control information for a specific client. To use this function, you pass it a reference object variable of the object you would like to control, and it returns a boolean result (true if successful, false if not). Here's an example of using this function:

```
function thisIsMine(%client, %targetObj) {  
    return %client.setControlObject(%targetObj);  
}
```

Now, since this is the first time you're learning this concept, all of these functions must be executed on the **server end** since they control the gameplay itself. Remember, the client only controls things such as rendering and sound, so that will be things like GUIs and SFX events, everything else goes on the server.

getControlObject

Likewise with the function we used before, there is also a **getControlObject** function to fetch the object that is currently being controlled by a client instance. You don't send any parameters to this function and it returns a reference object id as a variable.

isAIControlled

This is a special function and you likely won't be using this ever, unless you're on an older generation of Torque (TGE, TGEA). This function returns true if the specified client instance is an **AIConnection** object and not a **GameConnection** object. There are no special variables and you receive a boolean result.

play2D

This function is used to play a SFX event on the specified client instance, since it's treated as a 2D sound effect, there is no special positioning variable, this function will just tell the client to play the specified sound profile. To use this function you send it the datablock name of the SFXProfile you'd like to play on the client. The function returns true if it successfully played and false if not.

play3D

This function is the counterpart to **play2D** in which the client instance will play a world-sound event. Unlike play2D, you can specify a target position that the sound instance will originate from, the engine's sound libraries will generate a more realistic feel to the sound by dampening and sending the audio cue to the correct side of the speaker based on the information. You send two variables to this function, the first is the datablock name of the SFXProfile to play, and the second variable is a transformation matrix ("x y z rx ry rz ra") (Chapter 9) of where the sound needs to originate from and which way the forward vector of the sound points. The function returns true if the sound playing was successful and false if it was not.

setControlCameraFOV

This function is used to say how wide of a viewing angle the client's camera is supposed to cover. By default this is around 45 degrees, but you can use any value between 1.0 and 179.0, do note

that all decimal values will be removed when actually used internally. To use this function you send one variable, the new FoV in degrees and there is no return value on the function.

getControlCameraFOV

This function is used in conjunction with the above to fetch the current FoV used by the client's camera instance. There are no parameters needed for this function and it returns a decimal value of the current viewing angle of the client.

getDamageFlash

This function is used to tell the server how much damage flash (red flash) is occurring on the specified client, where 0.0 means none, and 1.0 means the maximum. There are no parameters to this function and it returns a decimal number.

getWhiteout

This function, like **getDamageFlash** is used to tell the server how much "flash" is occurring on the client. However this is for the whiteout flash number. There are no parameters needed and the function returns a decimal number.

setBlackout

The next function is used to have the specified client's screen either fade to or from black. There are two parameters on this function and no return value. The first parameter is a boolean where true means fade to black, and false means fade from black. The second parameter is an integer value stating how long (in milliseconds) until the screen either fades to or from black.

delete

While every object in the engine has a **delete** function coded into it, the client object has an extra string parameter attached to it. When you use this function on a client, that client will be kicked from the game. Optionally you can attach a string message to this function as a reason for the disconnect on the client.

getServerConnection

This is a special function that can be used on the client side to fetch the connection object to the server. This is used to access a limited number of client specific functions that are related to the **GameConnection** and **ServerConnection** classes. There are no parameters to the function and it will return a reference id as a result. You can then use the **dump** command to fetch a list of functions this object may use.

setCameraObject

This function is used to set the camera object associated with the client. If the client is not connected to a specific object instance by means of the **setControlObject** function, then you can use this function to attach the client's view to a camera instance. You pass the reference id of the camera object, and the function returns a boolean based on the success or failure of the function attaching the view instance to the camera.

getCameraObject

This function is used to fetch the reference id of the camera object that belongs to this individual client object. You do not pass any variables to the function and it returns the id of the camera object, if there is one.

clearCameraObject

This function is used to remove the camera object reference attached to this specific client instance. There are no function parameters and no return values from this function.

isFirstPerson

This function is used to check if a specified client is currently in the first person view or the third person viewing perspective. There are no special parameters for the function and it returns true if the player is in the first person, and false if it is not.

setFirstPerson

Finally, we have the ***setFirstPerson*** function, which is used to change the current viewing perspective of the specified client instance. You pass a boolean to the function where true means to switch to first person, and false means to switch to third person. There is no return value for this function instance.

Client & Server Commands

So there's the setup for how the server can control the client. But now, if you go back to the roles section I told you about, I said that the server had the role of **Messaging** where it could route event messages to and from the server and between clients. This in turn means that you can write functions that send events to the server, or to other clients even. Now I'm going to show you how to do this.

CommandToClient and CommandToServer

This process works by taking advantage of two function archetypes, and they are the **commandToClient** and **commandToServer** functions. Here's the syntax of these two functions for your viewing:

```
commandToClient(%targetClient, %FunctionTag, %arg1, ..., %argN);
commandToServer(%FunctionTag, %arg1, ..., %argN);
```

The first function is a function that can only be called on the server instance, you pass the reference of the client object you would like to send an event to, and then send it the event tag and the arguments of the event function. The second function is called on the client, when you want to reply to the server, or request the server to perform an action. You send the event tag and the arguments of the event function.

But there's just one question that comes about this, what exactly is an **event tag**? Well, let's introduce that to you.

Tags

Tags are special string instances that are stored in a special table of compressed information, called the **String Table**. This is a quick reference for instances that are frequently repeated and instead of sending the lengthy string data every single time, it is compressed into an index value (integer) and that is sent instead, and the target instance is then able to decompress the instance using the same **String Table**. To make sure this works, the tables are synchronized between the server and the client during the connection process so they share the same information, and any new information is added via the first transmission which contains the plain string, and an index value to associate it to.

To actually form a tagged string in the engine, you surround it in single quotes instead of double quotes, for example:

```
function tagExample1() {  
    %basicString = "This is a string";  
    %basicTaggedString = 'This is a string';  
    echo("Str: @%basicString@", Tagged: "@%basicTaggedString");  
}
```

If you run this function, you'll notice a message indicating that the string has been added to the table on an index, and when the string is printed, you'll see the first string print, and then that index value print for the second string. Now, you can also compress and decompress tagged strings by means of two simple function calls:

```
function tagExample2() {  
    %basicString = "This is a string";  
    %basicTaggedString = addTaggedString(%basicString);  
    echo("Str: @%basicString@", Tagged: "@%basicTaggedString");  
    %removedTag = getTaggedString(%basicTaggedString);  
    echo("Decompressed: @%removedTag");  
}
```

This basic form of message compression is used to minimize the amount of bandwidth that is needed by the Torque Networking System to ensure the engine is optimized for information transfer between individual clients. So now that you know about tags, let's put them to use for the server and client commands.

Function Syntax

So the first thing we need to show you how to do, is how to actually set up the event functions that are going to be used by the client and the server. The engine differentiates these functions by means of prepending the function names with words.

To create a server event function, you prepend the function name with **serverCmd**, and to create a client event function, you prepend the function name with **clientCmd**. These cannot be object functions, and the server function requires a client variable as the first variable. So, let's look at a basic example function:

```
//Server Functions (Put these on the server side):

function networkExample1(%targetClient) {
    if(%targetClient.stopSendingEvents) {
        %targetClient.stopSendingEvents = 0;
        return;
    }

    commandToClient(%targetClient, 'TestEvent1', "Hello World");
    schedule(2500, 0, networkExample1, %targetClient);
}

function serverCmdRecieveEvent1(%client, %message) {
    echo("Client '@%client@" says: "@%message");
}

function serverCmdStopBuggingMe(%client) {
    %client.stopSendingEvents = 1;
}

//Client Functions (Put these on the client side):

function stopMessages() {
    commandToServer('StopBuggingMe');
}

function clientCmdTestEvent1(%message) {
    echo("Got '@%message@" from the server, responding.");
    commandToServer('RecieveEvent1', "I got it!");
}
```

This here is a basic form of how to set up a network event. The server fires off the **networkExample1** function and selects a client to message. It then pushes the **TestEvent1** function to the client, which picks up the message and sends **RecieveEvent1** back to the server. Once the first function is called, the server will continue to fire messages every two and a half seconds until the client uses the **stopMessages** function, which pushes the event **StopBuggingMe** to the server which applies a variable to the client object which turns off the **networkExample1** function.

This example should demonstrate how to properly set up server to client and client to server transmissions in the engine's scripting system, and how to create variable definitions in those functions as well (but you should be good there too by now). There's one final thing to cover in our last journey through TorqueScript and that is some more information on the system we discussed a little while ago in this chapter.

Introduction to the Ghosting System

To ensure that the game connections in the engine run smoothly and without much lag (allowing for large 64+ player games), the Torque networking system introduced the concepts of **ghosting** and **scoping**. I briefly defined both of these earlier when we talked about the roles of the client and the server, but now I want to further discuss the ghosting part of it, as there are some useful applications that you can create through this system.

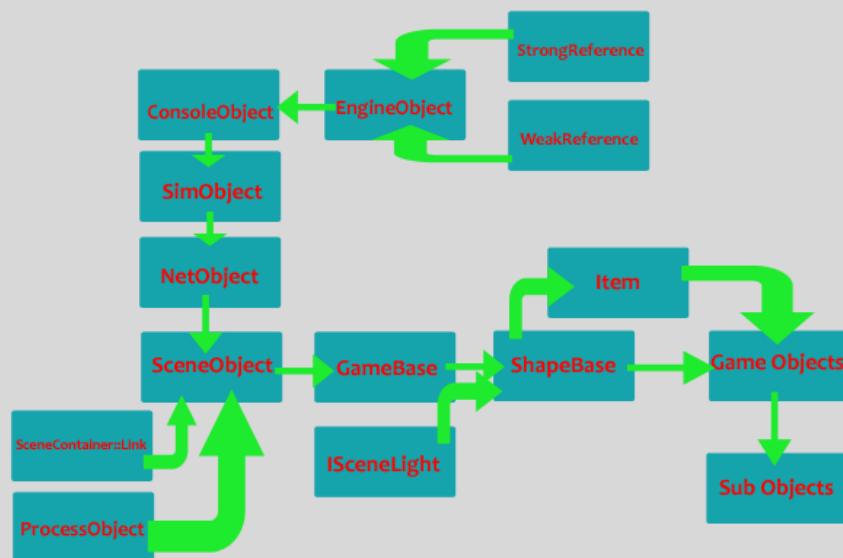
What is a Ghost?

So, obviously the concept of a ghosting system means that we have ghosts as well. And no, these aren't the kind of things that will reach a materialized hand through your computer and try to do things, the actual definition of a **ghost** as it relates to the engine is a copy of an object that is sent down to an individual client to be used as their version of the object in their instance. Why do we do this? So remember that the client and the server run on two separate instances (except in the case of a listen server), so each instance will have their own object reference ID numbers, obviously if the client sees an object and the server sees the same object, they'll exist under two different ID's and then problem's will happen when one instance tries to do something with the other's same instance.

To fix this problem, the server holds the authority on all of the objects and then transmits copies of the objects down to the individual clients to be stored on the client as a ghost. The server, meanwhile, stores a table of the client's identification numbering information as the ghosting information, and this is what we're interested in for this part of the guide.

Ghost ID's

So, a ghost identification number, as you likely guessed runs independent from, you also guessed it, the object numbering. So now we've got two sets of numbers to worry about here. What we have to deal with is the fact of how to translate these ghost identification numbers into object reference numbers, or basically reference variables of the objects on either the client or the server. To help with this, I'm actually going to bring back a diagram that you're now going to see for the third time:



Yep, that's right, the object inheritance tree makes a third return to the guide, and here's why it's coming back again. In the tree, there is one very extremely ultra-important class that we need to worry about: **SimObject**. And now, I'll actually explain both of the numbering systems. So, as you've already seen from our many examples of the time before, that when you create an object in the engine, it stored a reference number as an object ID, and that these ID numbers can be translated to an object reference variable. We've also gone through the examples of Groups and Sets to show you how the SimObject can be organized when creating instances on your maps. Well, here's a little news flash for you to comprehend for a moment, the **GameConnection** class we were talking about a little while back, actually inherits directly from the **SimGroup** class (via **NetConnection**), which leads to **SimSet** and then to **SimObject**. Alright, so that shouldn't be too hard to comprehend since we assign object ID's to client objects as well, but where exactly are these ghost numbers coming from? This is actually a system that is derived inside of the **NetConnection** class as packet information. You know this already because I told you earlier that the server via the scoping protocol sends all of the client instances, ghosting information when it needs to.

Confused yet? Don't be, just realize that the Ghosting system exists directly inside each individual **NetConnection** object, and that **GameConnection** is in fact a **NetConnection** instance, therefore all client objects have their own ghost system installed in it. Since the ghosting system is not directly associated with the **SimObject**, but instead packet information that is unique to each individual **NetConnection** object, the numbering system is unique to the ghosting system. That's how you have two different numerical systems to worry about.

So, how then does it actually work? Easy, when the server scopes an object to be flagged for transmission (usually when the object comes into "view" of the client) it transmits the object's information down to the client as a ghost copy. The client receives this information and then places it in a table of objects, let's just call this the **ghost table** for now, and then sends a response to the server saying that it got the information, and it also sends up a numerical id or the index on the table (**array**) that it stored the object in, and the server keeps this handy in the event of a lookup call. The cycle is then endlessly repeated as needed when the client needs something from the server until the client instance disconnects from the server, and dumps the table. So that's all the ghost id is, it's an index of the table, or if you've been following us for the whole ride, the index of the array for the ghosted objects on the client or server end. Just know that the ghost instance itself stores two numbers, the reference id of the object, and the ghost index for the table.

A little more on Scoping

So now you know about one end of the spectrum, let's talk about the other end of it, which is the **scoping** side of it. To do this, on the ghosting system, the engine installs flags that can be triggered on objects containing relevant information regarding their current state and whether or not they need to be transmitted to a client. Since it exists as part of each ghosted object instance, it also exists for each individual client instance. There are nine flags that can be placed on an object, like the typemask system, these are installed based on bitwise numbers, therefore multiple flags can be placed on an object at any given time. Here's the list of the scoping flags and here's what they do:

Valid is the flag on bit zero (2^0) and this is the generic state flag saying that the individual ghost instance is valid and can be resolved. **InScope** is the flag on bit one (2^1) and this flag states that an object has come into scope (is in view and needs to be updated by the client) and is ready for ghosting.

ScopeAlways is on bit two, and is a special flag for an important object instance (like other client objects) saying that this object needs to be ghosted on each individual engine tick regardless of being in view or not. **NotYetGhosted** is on bit three, and this flag is assigned to an object that may not be in scope yet, or has not been ghosted to an individual client. This object will not be resolvable until it comes into scope and had been ghosted. **Ghosting** is on bit four, and this is assigned to an object that has been flagged to be in scope and is now in the process of receiving a ghost id. **KillGhost** is on bit five, and this flag is assigned to an object that is being deleted, while not completely deleted yet, but is in the process of being removed from the table. **KillingGhost** is on bit six, and this flag is assigned during the deletion process of an object. **ScopedEvent** is on bit seven, and this is a special flag that is assigned to a net event saying that the context of the event must be in scope of the client to be received. Finally we have **ScopeLocalAlways** on bit eight, which is only used for listen server instances, where the client who is also the server, must have constant updates of all objects.

For your applications you don't need to know these flags, but just know they work internally to set the scoped state to actually ghost individual objects on the clients. We'll cover this a little bit more in Chapter 18 when we do the internal ghosting, but for now, just know how this works to bring objects into scope when they need to be and to be placed locally on a client instance to be used as a reference ghost.

Resolving Ghosts

So, how exactly then do we resolve the ghosts? Well, if you scroll just a little ways back up to the functions for the **GameConnection** class, there's one that we're going to need: **getServerConnection**. This function when used on the client end can translate the current connection to a server into a **NetConnection** object, specifically, the object of the client itself. On the server end, you have direct access to **NetConnection** through all of the client objects, remember **GameConnection** inherits from **NetConnection**. And with that in mind, let me add three new functions to your arsenal:

getGhostID

And we start with the **getGhostID** object function, which is used to translate an object instance into the ghost id instance. You obviously call this function on the **NetConnection** you're interested in, and it accepts one object parameter, the reference ID of the "real" object. This function will either return an integer of the ghost ID, or -1 if there was no object found (It doesn't exist or hasn't been ghosted yet), here's a quick sample:

```
function fetchClientGhostID(%targetClient, %realObject) {  
    return %targetClient.getGhostID(%realObject);  
}
```

resolveObjectFromGhostIndex

Besides having a long name, the **resolveObjectFromGhostIndex** is the function for the **server instance** to convert a ghost id into an actual object reference ID. You call this function on the server end, and attach the client variable you want to resolve from to it. This function accepts one integer as a variable, and that is the ghost ID on the specified instance. This function will either return the id of the resolved object, or -1 if the object could not be found. Here's an example of this in action:

```
function resolveClientGhost(%targetClient, %clientGhostID) {  
    return %targetClient.resolveObjectFromGhostIndex(%clientGhostID);  
}
```

resolveGhostID

And this is the counterpart to the above. The **resolveGhostID** function is called on the client end to fetch its ghosted object instance from the specified ghost ID. You can use this function to fetch the client's local instance of the specific object, but keep in mind the rules of ghosting and how the server maintains authority over objects. Here's an example of this function:

```
function resolveGhost(%clientGhostID) {  
    return ServerConnection.resolveGhostID(%clientGhostID);  
}
```

Finishing Up

So there you have it, that's how you can fetch ghost instances and how to resolve them into the actual object instances that exist on the client or the server. While you may never actually touch the ghosting system as it pertains to gameplay scripting, there may come a time when you'll need this system to resolve objects as it pertains from client to client, or if you need to do something on the client end that requires the current object information it possesses, although a simple client to server to client message sequence would also work for that.

The main goal of this chapter was to introduce you to the systems that run the engine's networking and to show you some of the basic tools that are offered to you from this, and finally, we put to bed the whole notions of client instances, server instances, and client objects.

You've now completed all of your "TorqueScript Training" you'll be getting from this guide. The only step that remains for you to do is to get out there and actually try it for yourself. Go out there and craft some amazing levels, GUIs, and then wire them together using compact and powerful scripts to give your users the most fascinating experience they will have in your game world. The possibilities are endless, and the experience could not be any more fun or rewarding when you feel the success of creating something.

So, if you feel like you're ready, stick around because it's time to step away from the "easy" stuff and dive head first into the more complex coding and the inner-workings of the engine. Yes! It's time to learn the C++ side of things!

Chapter 13: C++ for Absolute Beginners

Chapter Introduction

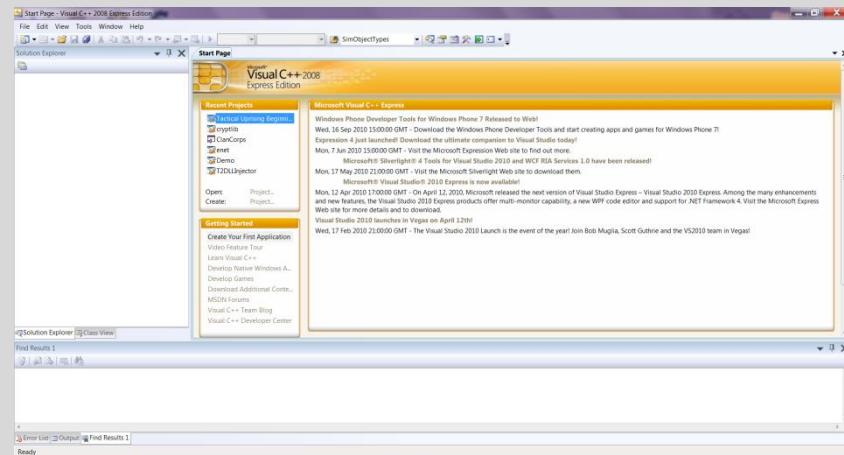
Alright! Is your head hurting yet? Alright good, cause it's about to get much worse. Why you ask? Well, because it's now time to turn the difficulty level up a bit here. So, as I originally mentioned way back in Chapter 6, TorqueScript is a scripting language with a C-Style approach. So, let's go back and revisit that statement. First off, you've been working with a scripting language, which is really not that difficult to comprehend. Scripting Languages are usually much more "forgiving" in terms of mistakes and you also have the whole in-engine compile thing which is nice. The second part of that is the text "C-Style", which means, you have been doing computer programming, but at the same time, you haven't been.

So, what we're about to do is open up a brand new book, per say. What you've learned up to this point has been really easy, now, things are going to get harder. This chapter will teach you the absolute basics of the C++ programming language. So, what's the difference? A lot actually, a programming language, unlike the scripting language is actually responsible for converting your code into machine code, which is then translated into an executable file, or a program. You're going to get a lot thrown at you very quickly here, so please read and re-read if you feel like you're getting lost.

The first thing you need to understand here is that you will notice quite a few things that are similar to what you have been doing, and then you'll also see things in here that are much different than what you have been doing. The first thing we need to talk about is a major difference in how C++ is built compared to TorqueScript.

Getting started in Visual Studio

Unlike TorqueScript, C++ is a language that requires code changes to be **compiled** again into the file that becomes the executable for your project. C++ is a low-level system development language. Basically compared to a higher-level language like TorqueScript that can be adjusted on the fly where you can view your changes in real-time, C++ requires you to re-build the entire program even if you change a single character in your code. In order to compile our code, we're going to use Microsoft Visual Studio, which is a C++ compiler and editor program.



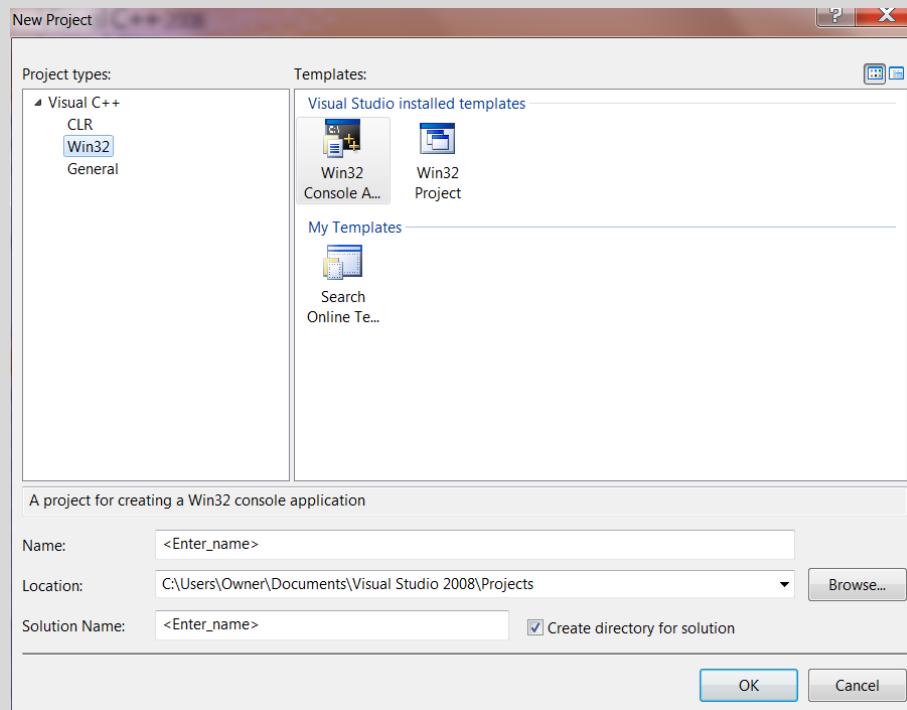
I'll be using the 2008 version of VS for the purposes of the tutorial, but you'll be perfectly fine using 2008 or 2010. There are a few differences between the two, but they accomplish the same things. So now that we've got into the editor itself (you should have completed the registration back in Chapter 2) we can get started.

Solutions & Projects

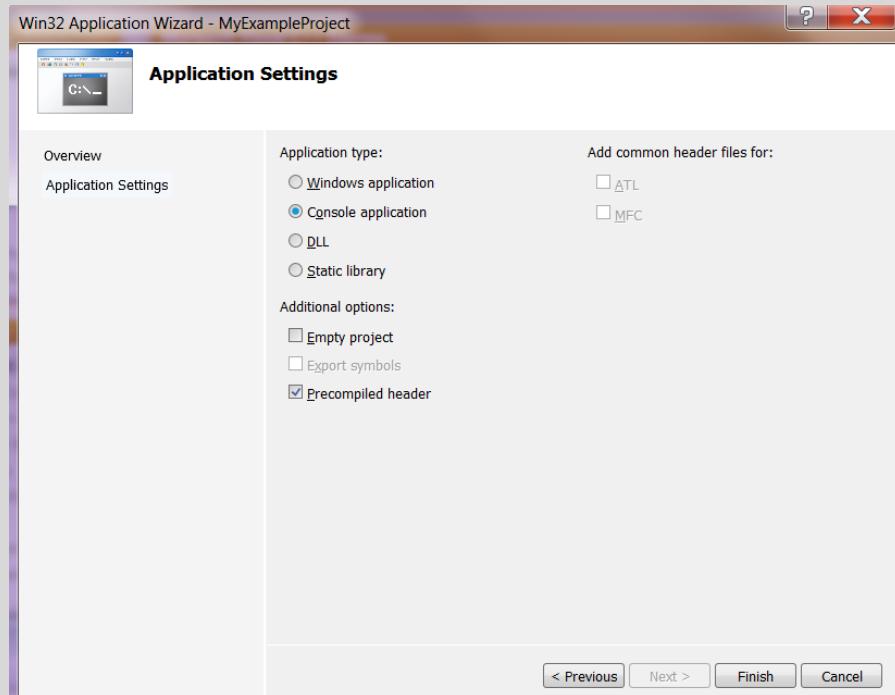
One of the tougher tasks when it comes to C++ versus TorqueScript is the size of the overall task in general. There is so much more going on in C++ at a time compared to TorqueScript so organization of the entire project is the key to success. To help you with this, Visual Studio sorts your "projects" into two categories which are the **solution** and the **project**. The solution is the overall program you are **building** (more on this in a bit) and it brings all of the **projects** together. A project therefore is an individual grouping of code that is built. Also unlike in TorqueScript, the order in which you build the solution does matter. C++ builds projects by compiling the code, then linking it using any additional libraries and headers you're using before creating the executable or library file you're interested in. Therefore, you need to have the functions you're using available for the engine to read from before you actually use it (unlike in TS, where you could call something even if you defined it in a completely different file). Basically, imagine C++ as one massive datablock system, you can't use something, until it has been defined.

Creating a Solution & Project

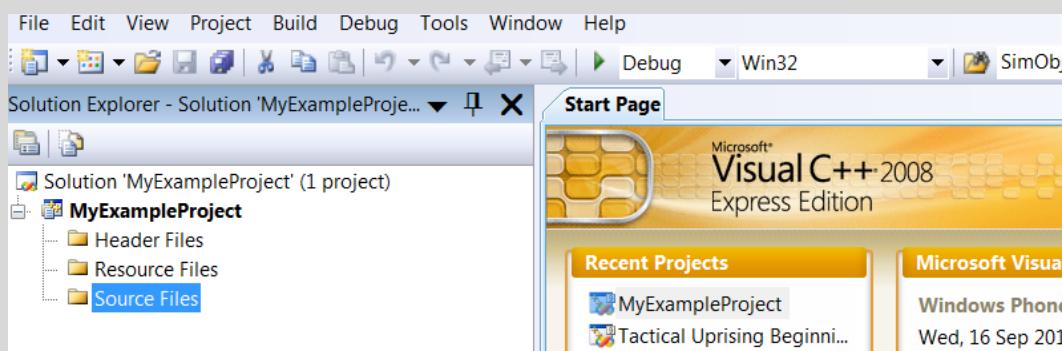
So the first thing we're going to do here is to create a basic solution example. This is very easy, just click the file button, then navigate to new and project. We'll get to the actual solution thing here in just a moment. You'll get a window a bit like this when you click that button:



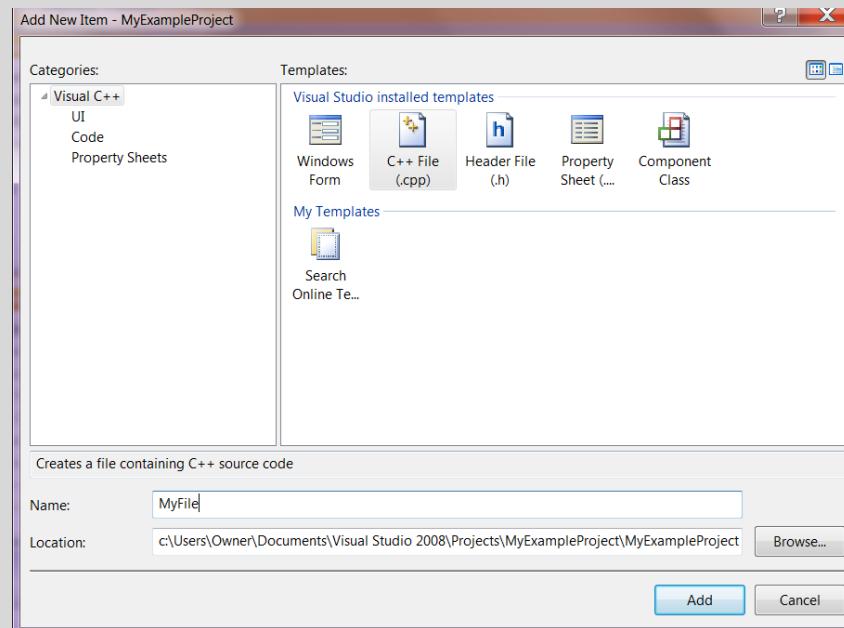
For the time being, you can stick with the Win32 Console Application. A Win32 Project deals with some more advanced C++ code (beyond the scope of our guide) and is used to start a much larger scale project that usually involves dialogs and windows. Torque 3D fits under the category of the second, but since the code is already there, you don't need to worry too much about it. Go ahead and give your project a name and leave the **Create directory for solution** button checked. This will make sure you create both a project and a solution. Click Ok, and the next to get to this window:



This gives you some more advanced options as to **entry points** and what type of thing you want to actually build (.exe, .dll, .lib). For our purposes here, leave it as a console application, uncheck the box that says Precompiled header, and check the empty project box and then click the Finish button. On the side of your screen, you can now open the solution explorer and you'll see something that looks like this:



Finally, we'll actually need a file to work with, so right click the Source Files folder and click Add and then New Item to get a selection window that looks a bit like this:



We want a C++ file, so select that button, and give your file a name then add it to the solution, which will open your new file up in the editor. Now we can get started with learning C++!

C++ Basics: Getting Started with Data Types

When it comes to C++, there's a key difference between what we were doing in TorqueScript, and what we will be doing in C++, and that difference is the data type. Every variable in C++ must have a data type associated with it. There's a small list of them to choose from, and other data types can be defined by additional header files, or even created by you for your own purposes. Let's start by looking at the data types and giving a short explanation of them.

Data Types

In C++, a data type defines the properties of a specific variable, or function by means of telling the computer what it can expect to be held in the specific location in which the variable or function is being held in memory. We'll talk a lot more on the topic of memory in the coming chapters, but you need to understand that since C++ is a lower level language than TorqueScript, that it involves a higher level of complexity and access on a computer instance.

Also, like before we have a convention of naming for variables in C++. They are very similar to the ones we discussed before for TorqueScript, basically variables must begin with either a letter or an underscore, cannot contain unknown characters, and cannot share a name with one of C++'s unique identifiers. There's also the new rule of two variables cannot share the same name if they are being initialized within the same scope of a function body.

Now, let's look at the primary forms of variable types, and what they can do for you in C++.

Variable Name	C++ Keyword	Bit Size	Description
Character	char	8	A single ASCII character or a terminated character

			sequence
Wide Character	wchar_t	16	The largest of character sequence sizes available.
Short Integer	short	16	The smallest of integer sequence sizes available.
Integer	int	16 - 32	Standard integer value, size of the variable is ranged based on value
Long Integer	long	32	Standard integer value, forced to 32 bits
64-Bit Integer *1	long long	64	The largest of integer sequence sizes available.
Floating Point Number	float	32	The lowest of decimal precision available for C++.
Double Floating Point Number	double	32 - 64	Higher precision available compared to <i>float</i> but it is ranged based on the value in the variable.
Long Double Floating Point	long double	64	Offers the highest precision for decimal point numbers in C++.
Boolean	bool	8	A boolean flag, holds true or false
Void Type	void	0	An empty data type (mainly used for return flags on functions)

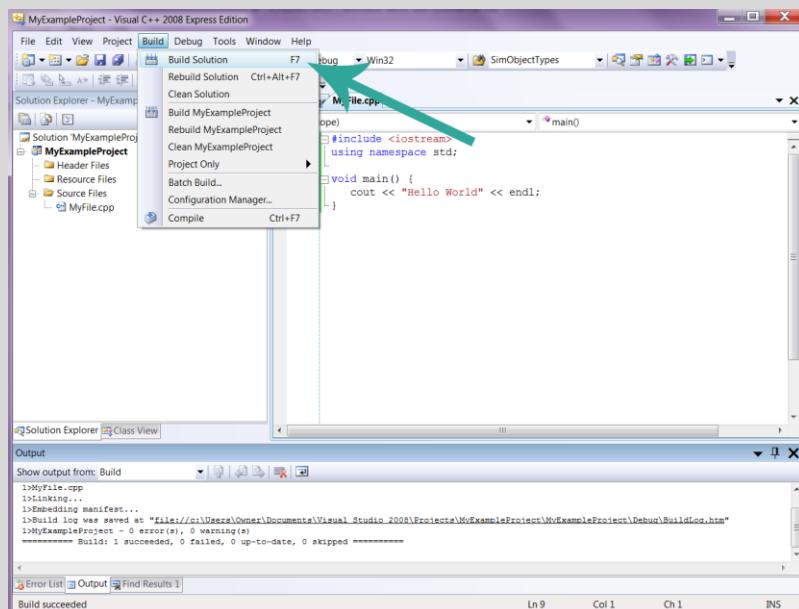
*1: This data type can be initialized differently based on the compiler, for instance in MSVS you can also use `_int64`.

With these basic data types in mind, let's have a look at a starting example to get us moving in C++. In the file we just created earlier, go ahead and put the following in it:

```
#include <iostream>
using namespace std;

void main() {
    cout << "Hello World!" << endl;
}
```

Save it, and then build the entire solution using the Build Solution option to compile the code into a basic executable:



If everything compiled without error, you'll see the bottom line say: "Build: 1 successful, 0 failed...", which indicates there were no errors in the code. To run the code from here, go over to the debug tab and click the button that says **Start Without Debugging**. You'll get a console window with the words "Hello World" printed inside of it. For now, pay no mind to the first two lines and the **void main()** line, I'll get to those briefly.

While we've got our basic code up and running, let's apply what we've just learned in terms of variable types to this code. Now, we're going to expand the sample code to read as follows:

```
#include <iostream>
using namespace std;

void main() {
    //Character Types
    char a = 'a';
    wchar_t b = 'b';
    //Numerical Types
    short c = 1;
    int d = 2;
    long e = 3;
    long long f = 4;
    float g = 1.0;
    double h = 2.0;
    long double i = 3.0;
    //Others
    bool j = true;
    /* Now that we've defined our types, let's print them out! */
    cout << "Character Types:" << endl << "A: " << a << ", " << sizeof(char) * 8 << endl;
    cout << "B: " << b << " [" << (char)b << "]", " << sizeof(wchar_t) * 8 << endl;
    cout << "Numerical Types:" << endl << "C: " << c << ", " << sizeof(short) * 8 << endl;
    cout << "D: " << d << ", " << sizeof(int) * 8 << endl;
    cout << "E: " << e << ", " << sizeof(long) * 8 << endl;
    cout << "F: " << f << ", " << sizeof(long long) * 8 << endl;
    cout << "G: " << g << ", " << sizeof(float) * 8 << endl;
    cout << "H: " << h << ", " << sizeof(double) * 8 << endl;
    cout << "I: " << i << ", " << sizeof(long double) * 8 << endl;
    cout << "Other Types:" << endl << "J: " << j << ", " << sizeof(bool) * 8 << endl;
}
```

Remember!!! In C++, whenever you make any changes to code, you need to rebuild the entire solution before it is allowed to run with the new changes.

So let's quickly talk about what we're actually doing here. Most of this stuff should actually come as a semi-review, semi-new stuff section where we're re-introducing variables and how to use them in the code. The bit of new stuff comes with the little bit in the actual printing code where I use the **sizeof** keyword, and multiply it by 8. This is to show you how the table above is correct in the amount of bits being used by each of the data types. The **sizeof** command is used to tell you the amount of **bytes** something uses, and since a byte consists of 8 bits, we multiply it by 8 to determine the amount

of bits being used. The little difference in the line with letter **B** is called a **type-cast** where we convert the **wchar_t** standard output (ASCII Number) to a character. I'll have more on type-casting later on in chapter 14. Other than that, you're pretty much seeing a good starting point with the above example of basic C++ code.

I'm hoping that you're starting to pick up on habits that we've been working with in TorqueScript from the past few chapters here too. You should begin to notice some similarities already for example, how lines that aren't function definitions end with a semicolon, and how all of the strings need to be closed strings, just like before as well.

The primary goal of the above example is to show you how to define a variable type in a function and how to assign it a value right away. You could however, just create the variable, and assign it later, for example:

```
#include <iostream>
using namespace std;

void main() {
    //Character Types
    char a;
    a = 'a';
    /* Now that we've defined our types, let's print them out! */
    cout << "Character Types:" << endl << "A: " << a << ", " << sizeof(char) * 8 << endl;
}
```

That would also be perfectly legal in C++. Now you may be wondering, now that I've talked about the variables, where are the operators? Well, I don't need to teach you operators, because they're exactly the same as the TorqueScript operators. This may confuse you at first due to these **cout** lines having the bitwise left shift operator to split the string sections. You don't need to be confused, because in C++, operators can be overridden when used by a class instance, which is what's happening here with the **cout** class. There are a few new operators in C++ that you'll need to learn, but I'll have more on those in the coming chapters.

So before we move on to talk about that **void main** line, let's introduce a few more topics that relate to the subject of data types.

Signed & Unsigned Types

Now that you know about the basic types of C++, let's introduce the next concept with certain types we just introduced. Before we do this, remember back just a moment ago when we discussed the table involving the amount of bits a variable can use. If you also go all the way back to chapter 6 when I introduced you to the bitwise operators, I told you about binary numbers and how they stream together in flags of zeros and ones to make other numbers. This holds the exact same concept; basically for example a 64-bit integer is a sequence of 64 binary flags. How this works out numerically for our perspective is like so:

Bit Amount	Maximum Value
8 Bits	256

16 Bits	65536
32 Bits	4294967296
64 Bits	18446744073709551616

So the higher you go in terms in bit count, the larger your values can reach. But now we need to ask ourselves a question, what about negative numbers? Our next topic mainly deals with the numerical data types, however the character types can also have these same rules enforced on them. By default, if you declare these data types in their pure form (as we did above), they are by nature **signed** variables, which means the data range is split in half to accommodate for both positive and negative values. So by default, you cannot for instance create an integer variable, and give it the value of four million, when you try to print it, you'd have a negative number, because the variable would reach to the other end to try to piece it back into range. To get it to reach the full extent, you need to create an **unsigned** variable, which enforces a one-way range of the variable, basically an unsigned integer would be allowed to go from zero to the four million mark shown above. To use these two types in C++, you prepend the variable name with **signed** or **unsigned**. Remember, that by default all variables are signed, so you don't actually need to put the word signed in front of it, unless you are explicitly defining the variable as a signed variable. Here's another table to recap:

Variable Type	Bit Count	Range
signed char	8	-128 – 127
unsigned char	8	0 – 255
signed short	16	-32768 – 32767
unsigned short	16	0 – 65535
signed int	32	-2147483648 – 2147483647
unsigned int	32	0 – 4294967295
signed long long	64	-9223372036854775808 – 9223372036854775807
unsigned long long	64	0 – 18446744073709551615

The important thing to remember for this is to know and understand where your numerical results will end up in and to ensure that you have sufficient storage available for it. Now, just because you can, don't start under the ideology of making all of your number variable 64-bit integers, because remember when you have to deal with games, and specifically networking in games, you want to minimize your data transfer and processing amount to ensure the game moves smoothly. Not also saying it's a bad thing to use 64-bit integers, but to exercise control when using large variable types. It's all about efficiency when it comes to working with C++, and sometimes bigger isn't always better.

Constant Variable Types

The last thing we need to introduce with basic data types is the concept of **constant** variables. A constant variable, just like the name suggests is a variable whose value remains constant, or does not change during the course of a program. For now, you'll hold this knowledge as truth, but it'll change a bit a little further down the line. Basically you can use a constant variable to store information that will not change, but may be needed quite a bit in your program. A common instance is the value of Pi (3.14159), which you may store in a constant double for instance, and then pull out without the need to

constantly re-type the numbers again and again. To create a constant variable, you prepend the data type with the keyword **const**.

So that covers the basics of the topic of data types and variables in C++. We'll expand on these topics a little more in chapter 14, but for the time being, let's move on to the next topic in our discussion, which deals with that **void main** line we saw back in our earlier example.

Entry Point

So what exactly did you see there? That was your first C++ function declaration. And it's an important one at that. The line **void main** is the function's pre-defined entry point. Since we're dealing with a console application, the compiler knows that it needs to look for a function that is named **main** at the top layer of the program as the **entry point**, or basically the first thing to be ran when the program is initialized. When dealing with more advanced setups like windows applications, or Torque for example, which is a cross-platform engine, there are multiple different entry points to the program that could be used. For example, a Windows application will look for a **WinMain** method, the important thing to know here is what method serves as the entry point, and how you're supposed to set it up.

For console applications, there are two accepted formats. There is the void format, and the integer format. The void format (what we've been using) runs the code once, and when it's done, you just close the console window and that's it. Now let's say you're creating an auto-patcher program of some form, and the application that runs this patcher needs to know if everything succeeded. This is where the integer format comes into play. Basically the integer format is required to return a number at the end, where 0 means success and anything else means failed, obviously you could code it differently based on what you need it to do. Here's how you'd do that:

```
#include <iostream>
using namespace std;

int main() {
    return 0;
}
```

This is the most basic form of a console application program. But, we're not completely done with these yet. As I talked about a few times, when you build a C++ codebase, you actually create a standard executable file (.exe), and you should know by now from experience that these files can accept a target parameter line that allows you to define additional parameters for the code to use at the launch, or in other terms, at its entry point. To catch this, we have to add a little bit of a cryptic mess to the main function:

```
#include <iostream>
using namespace std;

int main(int argc, const char **argv) {
    for(int i = 0; i < argc; i++) {
        cout << "Parameter " << i << ":" << argv[i] << endl;
    }
    return 0;
}
```

So, just like that, you get to have a little up and close preview of two topics that are coming up very shortly, for loops and arrays. The array part is a little more complicated this time around, but for the most part you just need to know that this is how you get command line parameters into the code. You can just as easily copy and paste that cryptic two parameter line into the void main method and it'll work just fine too. So while we're moving along nicely with parameters and functions, why don't we start talking about those again.

C++ Basics: Functions

And now we're going to get back into the important topic of functions. Just like before in TorqueScript, you'll find that breaking apart a larger task into smaller ones with functions will make the overall job of C++ become much more manageable. We have to deal with the topic of data types now when it comes to function definitions, but after the first half of this guide throwing complex function creation with TorqueScript at you, this part should be a breeze.

The important part about C++ remember, is that everything has a data type. So that means that your functions too, will have a data type associated with them. So let's talk about creating functions now.

C++ Functions, Parameters, and Predefined Values

To start this section, let's take a few steps back and look again at our main example:

```
#include <iostream>
using namespace std;

int main() {
    return 0;
}
```

We're going to focus primarily on the line that says `int main()`. When you define a function in C++, the first thing you need to declare is a return type for the function. A **return type** tells the compiler that the value that comes out of this function will be of a set type. So in our example above, the return type of the main function is **int**. The next thing you define is the name of the function, so in the above example the function name is **main**.

Now in TorqueScript, sometimes we had functions that did not return values. You can do the same in C++, just by using the **void** return type:

```
#include <iostream>
using namespace std;

void main() {
    cout << "Hello World!" << endl;
}
```

That's pretty much all of the basics just like we covered in TorqueScript! But before we move on, let's talk about another key part of functions, parameters. So just like in TorqueScript, you can have arguments on your functions, and it's really easy to make them in C++:

```
#include <iostream>
using namespace std;
```

```
int addEm(int a, int b) {
    return a + b;
}

void main() {
    cout << "The Result Is: " << addEm(10, 20);
}
```

The only difference here is again, the data types of the function. You'll need to include the data types of your parameters in these function definitions as well as in the names of your parameters. I also show you here how to call a function in C++. Unlike in TS, there are no such things as indirect calls, there are only direct calls. You can even store the values in a variable and use it later:

```
#include <iostream>
using namespace std;

int addEm(int a, int b) {
    return a + b;
}

void main() {
    int result = addEm(10, 20);
    cout << "The Result Is: " << result;
}
```

Function calls work exactly the same in TS as they do in C++. You pass the name of the function in the code and then provide the parameters of the function to the call. C++ however, has a cool feature that you can't do in TorqueScript, and that is the concept of **predefined parameters**. Now if you remember a long while ago, I said that all variables in TS were of a string type, so you could put anything in them and use it. So by that very logic you could have a function that set parameters to a default value if you didn't fill it in with a simple string check. In C++, if you don't send a parameter to a function, you'll get an error, so to get around this problem, you can use a **predefined parameter**. There is only one rule to using this, and that is that predefined parameters must be defined in reverse order. That is to say, the first parameter cannot have a predefined value unless the last one does, and so on and so forth. If you have a function with five parameters for example and you want the third one to have a default value, then the fourth and fifth must also have a value. To make a predefined value for a parameter, you simply set the parameter to that value directly in the definition line like so:

```
#include <iostream>
using namespace std;

int addEm(int a, int b = 20) {
    return a + b;
}

void main() {
    int result = addEm(10);
    cout << "The Result Is: " << result;
}
```

Now if the code calls the addEm function, you can either provide a second variable, or leave it blank (like in the above example) and C++ will set the value to 20 when it runs. So the long story short,

comes down to the next code block. Here's how you go about defining a function in C++, and if you want to use a PDV (**predefined value**) on it, here's how to do that as well:

```
returnType functionName(type a, .., type n-1 = PDV, type n = PDV) {  
    [If returnType != void, return result;]  
}
```

So there you have it, that's the process to cut down on your space used up in the main function right there. Just like last time, you'll want to break your work up into numerous functions to handle smaller tasks in an optimal manner. We'll come back to this again quite a few times as we move along. This is a very key concept to C++ right here, so if you don't have this down, please re-read and understand this section.

Now, my example above with the adding function is nice and all, but wouldn't you want to rearrange it to have the main function come first?

```
#include <iostream>  
using namespace std;  
  
void main() {  
    cout << "The Result Is: " << addEm(10, 20);  
}  
  
int addEm(int a, int b) {  
    return a + b;  
}
```

Something like this for instance? Well, go ahead and try to compile this, and see what happens. You'll get a missing symbol error. So what exactly is going on here and how do we fix it?

Function Stubs, Predefining Functions (Prototypes), And Header Files

So if you tried to compile the above example, you'll get a missing symbol error. So what gives? Let's talk about how C++ works before we move on. If you go back to the beginning of this chapter, I told you that C++ is a **low-level** language, which basically means that this functions on the lowest level of your computer, which would be the manipulation of memory. About the only lower language you could go to and still be logically functional in writing code would be **assembly** (And I'm Not Going There). So what happens when you try to compile a code is that the computer generates a list of instructions, and then it runs through a step called the **link** process, or the **linker**. How to understand this is easy. When the compiler runs through, the instructions it generates are in machine code (.OBJ Format), and that is something we cannot read. What the linker does, is it resolves these references in the machine code to a function list that is accessible to the program itself (which is the executable file). The key word in the previous sentence is the word **list**. Now, the error message you'd be getting due to the above code, would be a missing symbol, which basically means that when the programs runs the main method above, it doesn't see the addEm function, because to main, that method does not exist.

To cut the long story short, C++ is an ordered language, which means in order to use a function, it must exist in some way or form before it is called. But obviously I wouldn't give you a massive paragraph on how the language works without telling you a way to get around this, right? Well, you are correct! There is a way to get around this problem, and that is to **predefine** the function. So what on

earth am I going on about now? Well it's quite simple actually, what you do is you tell the computer that a function with a certain name, certain parameters, and a certain return type will exist at some point, and then the linker will establish the references later. How do we do this in code sense? With one line:

```
#include <iostream>
using namespace std;

int addEm(int a, int b = 20);

void main() {
    cout << "The Result Is: " << addEm(10);
}

int addEm(int a, int b = 20) {
    return a + b;
}
```

I even took it a step further and gave you the predefined value as well. All you do, is you create a copy of your actual function line, but in place of the opening brace that you would usually have to start the function's body, you simply close it off with a semicolon. This will give the linker the instruction to put that function in the global list, and then when the compiler hits the actual function it will resolve the instruction to the correct point in the code. This same line creates what we call a **prototype function**, or basically a function that exists in the code, but has no instructions. If you actually define the function later in the code, then it points the instruction to the correct definition. If not, then get ready for more errors.

Well this has been great and all; we've gotten some of the wonderful basics down. But you still haven't told us what those two very first lines are in every single one of the examples. Well, it's time for you to find out. The first line is an include statement. This tells C++ to load up the `<iostream>` file, and dump all of its contents right into the code there. Now, the `<iostream>` is short for Input/Output Stream, and it is used to receive and send messages to the console. That line in the `main` function accesses a class in `<iostream>` called `cout`. The left shift operator is overridden by the `cout` class (We'll get there later) to basically act as a one-shot call to the print function that exists within the `cout` class.

Now, I'm not to the point to introduce classes and objects again yet, we still have some other topics to cover for C++ before we go there, but just know that `cout` is a class instance. But what exactly is `#include`? This is a **preprocessor directive**, which is basically one of a select few special keywords that are initiated using the `#` symbol. [Preprocessor directives only cover one line of code and are never ended in a semicolon](#). We'll cover a few of these through the guide, and the first one you're going to learn is the INCLUDE directive. This basically, as mentioned above dumps the content from the file you're calling into the code. This is how you "load" a file into C++. What's loaded here is what is called a **header file** (`.h / .hpp`), which usually only contains class definitions (Chapter 16) and function stubs (Just Covered That). I'll go into much more details on these header files a little later, but before we finish up here, I'm going to explain one little thing. You may have noticed that in all of the include statements used so far, that there is no file extension on `<iostream>`. This is because of the symbology around the name `iostream`. When you place GT/LT symbols around the name, you tell C++ that you're loading in a system file, which basically means that the compiler will check its settings and see if the file exists in one

of the pre-defined locations. You'll only really use this for platform files, and occasionally on libraries that will tell you how to set them up. For header files you make, you'll do something like this:

```
#include "myHeader.h"
```

So you can see there that the header you make is surrounded by double quotes. This tells the compiler to start at the current working directory to look for the header file. This is a relative path declaration, so make sure to include folder paths when using the quoted form. When we get to Chapter 16, I'll help you to understand this concept better. So back to our old example, what's that second line? Well, I'll tell you more about that one in Chapter 14.

Inline Functions

Our next part of function making goes into the topic of **inline functions**. When it comes to writing a program in C++, the topics of performance and efficiency are two of the top things when it comes to your user experience. As I briefly mentioned earlier, the linker resolves references to the machine code. So imagine for a minute that your program has hundreds upon thousands of functions in its machine code, you'll begin to see how function lookup when it comes to resolving references can start to become hampered in the performance department. To help solve this problem, C++ has what are called **inline functions**. An inline function is one that exists directly at execution time, basically put when you define a function to be inline, the compiler will treat the definition as a stub function and when the function is actually called, it will replace the contents of the stub function with your actual code to save on performance.

So, why then don't we just make every function inline? Because an inline function occupies space in the program itself (executable file), and if you start throwing thousands of inline functions at it, you'll notice a very large increase in the size of your executable, not to mention that this alone can degrade the performance of your machine. Also, the **inline** keyword is more or less a recommendation that you make to the compiler. The compiler can still choose to simply ignore your request altogether and compile it as a normal function.

The best practice when it comes to making inline functions is to only use it on small functions, usually ones that encompass around or less than 20 lines that will be frequently executed. By using the **inline** keyword on functions that are very frequently called, you will maximize its performance for the program itself in the longer term, even if you need to sacrifice some disk space on the executable end of the program. In the coming chapters I'll teach you about C++ macros and define statements. What you'll need to take away from this point, is that by making a function inline, you're essentially making a macro version of a function. I'll hit this point again when we actually get to that topic.

So how do we create an inline function? Very easily, you just add one word to the function's definition line:

```
#include <iostream>
using namespace std;

inline int addEm(int a, int b = 20);
```

```
void main() {
    cout << "The Result Is: " << addEm(10);
}

inline int addEm(int a, int b = 20) {
    return a + b;
}
```

So there you have it! That's all of the basics you need to know about creating functions and working with parameters in C++. We'll obviously expand on this topic as we move along further in the guide, but for the time being, this is what you should absolutely know. Now let's move into our next topic.

Keywords of C++

Next up on the block is the **keywords** of C++. So just like in TorqueScript we have a unique set of words that are used to perform tasks. In C++ however, these keywords are explicitly defined, so you cannot overload any of these keywords at any time. Also unlike TorqueScript the keywords are case-sensitive so for example **return** will do the standard function of **return**, whereas **Return** will give you an error unless it is defined as a function.

Reviewing the Old

So let's start our keyword topic by reviewing the keywords we learned in TorqueScript to see the direct comparisons made in C++.

Keyword	TS Function	C++ Function
break	Stop the execution of a loop	Stop the execution of a loop, end a case statement in a switch operation
case	A case statement used by switch	A case statement used by switch
continue	Move to the next iteration of a loop	Move to the next iteration of a loop
default	The default statement of switch	The default statement of switch
else	Conditional, used for else if & else	Conditional, used for else if & else
false	Boolean Value	Boolean Value
for	Counter Loop	Counter Loop
if	Conditional used for else if & if	Conditional used for else if & if
new	Create a new object instance	Create a new object pointer in memory
return	Return a value, stop a function execution	Return a value, stop a function execution only if the return type is void
switch	Conditional Statement, Can use any data	Conditional Statement, Can only use standard C++ data types
true	Boolean Value	Boolean Value
while	Conditional Loop	Conditional Loop

So as you can see in this table here, most of the old TS keywords make a return to C++, or vice versa however you want to see it. There are many similarities and a few differences, and I'll cover all of the differences when we get to those keywords later on. You'll also notice here that some of the TS keywords do not appear in C++, and others behave completely different, such as the **switch** keyword, which can be used as **switch** or **switch\$** in TS to handle all forms of data, but can only be used as **switch** in C++ to handle **int**, **float**, and **char** data types. C++ cannot do strings in the **switch** statements.

New Stuff

Now, let's look at the keywords that are completely new to you in the context of programming in C++ compared to doing TorqueScript.

Keyword	Function	Example
__asm	Begins the definition of an inline assembly block (unused by this guide).	<pre>#include <stdio.h> char text[] = "Hello World"; char fmt[] = "%s\n"; void main() { __asm { lea eax, text push eax lea eax, fmt push eax call DWORD ptr printf pop ebx pop ebx } }</pre>
bool	Boolean type-definition.	bool myBoolean = true ;
catch	Catch statement of the try-catch block, defines a list of exceptions to be tested for.	<pre>#include <iostream> using namespace std; void main() { try { throw 15; } catch (int eNum) { cout << "Caught Error: " << eNum; } }</pre>
char	Character type-definition.	char fmt[] = "%s\n";
class	The beginning of a C++ class definition. By C++ definitions all members are given the private level modifier unless explicitly defined by the programmer.	<pre>#include <iostream> using namespace std; class myClass { public: void hi(); }; void myClass::hi() { cout << "Hi!" << endl; } void main() { myClass *c = new myClass(); c->hi(); }</pre>

const	Constant definition, can be used to identify a constant variable or a constant function definition.	<code>const char *text = "Hello World";</code>
const_cast	Expression to perform a constant type-cast of a variable.	<pre>#include <iostream> using namespace std; void main() { const char *text = "Hello"; char *newText; newText = const_cast<char*>(text); cout << newText << endl; }</pre>
do	The declaration of a do-while loop.	<pre>#include <iostream> using namespace std; void main() { int num = 1; do { num++; cout << "Number is now: " << num << endl; }while(num < 1); }</pre>
double	Double type-definition.	<code>double val = 1.0;</code>
dynamic_cast	Expression to safely convert pointers and class references along the inheritance hierarchy of the specific object.	<pre>#include <iostream> using namespace std; struct Base { int a; }; struct Child : public Base { int a; int b; }; void main() { Child *c = new Child(); if(Base *b = dynamic_cast<Base*>(c)) { cout << "Conversion Successful!" << endl; } }</pre>
enum	Declaration of an enumerated data type.	<pre>#include <iostream> using namespace std; struct Base { enum Info { One = 1, Two = 2, }; }; void main() { cout << "Base One: " << Base::Info::One; }</pre>
explicit	Forces class constructors and operators to not allow implicit type conversions.	<pre>#include <iostream> using namespace std; struct Base {</pre>

		<pre> explicit Base(int a) { num = a; } int num; }; void main() { Base b(1); cout << "Number: " << b.num; } </pre>
extern	Access external variables or external language features.	<pre> //FILE 1 #include <iostream> using namespace std; int gInt = 12; //FILE 2 #include <iostream> using namespace std; void main() { extern int gInt; cout << "The Number Is: " << gInt; } </pre>
float	Floating Point Number type-definition.	<pre> float val = 1.0f; </pre>
friend	Defines a class body to have friend level access to another, which grants access to protected and private level methods	<pre> #include <iostream> using namespace std; class B; class A { friend B; private: int a; public: A() : a(1) {} }; class B { private: int a; public: B(A x) : a(x.a) { cout << "Stored: " << a; } }; void main() { A a; B b(a); } </pre>
inline	Defines the compiler to mark a function to be inline.	<pre> #include <iostream> using namespace std; inline int addEm(int a, int b = 20); void main() { cout << "The Result Is: " << addEm(10); } inline int addEm(int a, int b = 20) { return a + b; } </pre>

		}
int	Integer type-definition.	int val = 1;
long	Long type-definition.	long val = 1;
mutable	Applies the mutable modifier to a class member. This allows the value to be modified through a constant level function or object.	<pre>#include <iostream> using namespace std; class MyClass { public: const struct vals { int x; mutable int y; vals() : x(1), y(1) {} }; vals x; MyClass() {} }; void main() { MyClass c; c.x.y = 4; cout << "Values of class: " << c.x.x << ", " << c.x.y << endl; }</pre>
namespace	Used to define a new namespace or in conjunction with the using keyword to include the specified namespace in the code.	<pre>#include <iostream> using namespace std; namespace MyCode { void HelloWorld() { cout << "Hello World!" << endl; } }; void main() { MyCode::HelloWorld(); }</pre>
operator	Starts the definition of a class based operator overload method.	<pre>#include <iostream> using namespace std; struct MyStruct { int num; MyStruct() : num(0) {} MyStruct& operator++() { num++; cout << "Number is now " << num << endl; return *this; } }; void main() { MyStruct s; for(int i = 0; i < 10; i++) { s++; } }</pre>
private	Defines a section of a class or struct to be marked with the private level access modifier.	<pre>#include <iostream> using namespace std; class MyClass {</pre>

		<pre> private: int a; }; void main() { MyClass mC; cout << "This function cannot print mC.a because it is a private member."; } </pre>
protected	Defines a section of a class or struct to be marked with the protected level access modifier.	<pre> #include <iostream> using namespace std; class MyClass { protected: int a; }; void main() { MyClass mC; cout << "This function cannot print mC.a because it is a protected member."; } </pre>
public	Defines a section of a class or struct to be marked with the public level access modifier.	<pre> #include <iostream> using namespace std; class MyClass { public: int a; }; void main() { MyClass mC; mC.a = 5; cout << "Value: " << mC.a << endl; } </pre>
reinterpret_cast	Used to reinterpret memory at the bit level to a different data type without guaranteeing the same bit order. NOTE: You should avoid use of reinterpret_cast at all costs in favor of using static_cast unless if you're working with differing API's and the type cast is required.	<pre> #include <iostream> using namespace std; void main() { int* a = new int(27); void* b = reinterpret_cast<void*>(a); int* c = reinterpret_cast<int*>(b); cout << "A: " << (*a) << endl; cout << "B: " << (b) << endl; cout << "C: " << (*c) << endl; } </pre>
short	Short type-definition.	short val = 1;
signed	Apply the signed type modifier to a variable which allows values to use half of its bit range for positive and the other half for negative values. By Default all variables are signed so using this keyword is redundant.	signed short val = 1;

sizeof	Fetch the size (in bytes) of a specific variable, or instance.	<pre>#include <iostream> using namespace std; void main() { char a; a = 'a'; cout << "Character Types:" << endl << "A: " << a << ", " << sizeof(char) * 8 << endl; }</pre>
static	Define a variable or function to be static, or to be available directly at runtime. By applying the static modifier to a variable, you initialize it under global program scope.	<pre>#include <iostream> using namespace std; static int myValue = 5; void main() { cout << "My Value: " << myValue; }</pre>
static_cast	Defines a type-conversion that allows implicit and user-type conversions to be made.	<pre>#include <iostream> using namespace std; void main() { int number = 43; char c = static_cast<char>(number); cout << number << " is now: " << c << endl; }</pre>
struct	The beginning of a C++ structure definition. By C++ definitions all members are given the public level modifier unless explicitly defined by the programmer.	<pre>#include <iostream> using namespace std; struct myClass { void hi(); }; void myClass::hi() { cout << "Hi!" << endl; } void main() { myClass *c = new myClass(); c->hi(); }</pre>
template	Start of a template definition, which is usually done through a class instance where multiple forms of data types are sent through to compound functioning through one instance.	<pre>#include <iostream> using namespace std; template<typename T> class Z { public: Z(T t) { a = t; } T get() { return a; } private: T a; }; void main() { Z<int> myClass1(15); Z<char> myClass2('a'); cout << "Value of 1 is: " << myClass1.get() << endl; cout << "Value of 2 is: " << myClass2.get() << endl; }</pre>

		}
this	This pointer, used inside a class definition to access the self-object that is calling a specific class method.	<pre>#include <iostream> using namespace std; struct MyStruct { int num; MyStruct() : num(0) {} MyStruct& operator++() { num++; cout << "Number is now " :>< num << endl; return *this; } }; void main() { MyStruct s; for(int i = 0; i < 10; i++) { s++; } }</pre>
throw	Used to throw an exception in the specified block of code. You can throw a direct exception, or even standard values.	<pre>#include <iostream> using namespace std; void main() { try { throw 15; } catch (int eNum) { cout << "Caught Error: " << eNum; } }</pre>
try	The start of a try-catch block in a code. This block will have potentially fatal code that needs to be ran through where exceptions may be caught and handled properly without crashing the program.	<pre>#include <iostream> using namespace std; void main() { try { throw 15; } catch (int eNum) { cout << "Caught Error: " << eNum; } }</pre>
typedef	A shortcut command to alias a symbol to a common C++ type, or a specified user definition.	<pre>#include <iostream> using namespace std; void main() { typedef int i; i num = 1; cout << "Num is: " << num; }</pre>
typename	Used in conjunction with a template definition, this is used as an alias for the word type, basically meaning that any data type is accessible at this point.	<pre>#include <iostream> using namespace std; template<typename T> class Z { public: Z(T t) { a = t; } T get() { return a; } }</pre>

		<pre> private: T a; }; void main() { Z<int> myClass1(15); Z<char> myClass2('a'); cout << "Value of 1 is: " << myClass1.get() << endl; cout << "Value of 2 is: " << myClass2.get() << endl; } </pre>
union	A declaration of a union instance, which is a special C++ class-like instance where it may only hold one non-static member at a time and the amount of used storage is based on the size of the largest member.	<pre> #include <iostream> using namespace std; union myUnion { int a; unsigned short b; unsigned char c; }; void main() { myUnion u = {0xFFFFFFF}; cout << "A is active: " << u.a << endl; u.b = 0x0011; cout << "B is active: " << u.b << endl; cout << "C? " << u.c << " That is: " << (short)u.c << endl; } </pre>
unsigned	Apply the unsigned type modifier to a variable which allows values to use the full bit range of the type over the positive interval.	<pre> unsigned int a = 0xFFFFFFF; </pre>
using	The directive to include the specific namespace or identifier in the specified code.	<pre> #include <iostream> using namespace std; namespace MyCode { void HelloWorld() { cout << "Hello World!" << endl; } }; void main() { using namespace MyCode; HelloWorld(); } </pre>
virtual	Defines a class function instance to be virtual, which is to say its behavior can be overridden by any derivative class instances.	<pre> #include <iostream> using namespace std; class Base { public: virtual void printme() { cout << "Hello!" << endl; } class Derived : public Base { </pre>

		<pre> /* Uncomment for override effect. public: void printme() { cout << "Hello From Two!" << endl; } }; void main() { Base b; b.printme(); Derived d; d.printme(); } </pre>
void	The void type definition which is used by a pointer to specify any data type, or the void return type which defines a function to not need a return value.	<pre> #include <iostream> using namespace std; void main() { cout << "Hello World!" << endl; } </pre>
volatile	Defines a class member to be volatile, which is to say that all effects of reading, writing on the variable is a visible side effect of a single-threaded program. That is to say that this variable is directly affected by multiple threads.	volatile int a = 100;
wchar_t	Definition of the wide-character data type	wchar_t a = 'a';

So as you can clearly see by this table, there's quite a bit that one could actually learn through programming in C++. You should also know right away, that there are even more keywords than the ones I presented to you here, however they for the most part, are highly technical in nature and way beyond the scope of our guide. Take the time to look through the above table, and the provided samples to understand what each of these keywords are, and what they are used for. I'll explain the usage of these keywords in places you haven't seen them before as we move through this guide, but the context of this guide is to serve as a learning tool, and that moves in the forward direction.

So, now that you know how to write functions in C++, and know all the keywords, what are we going to do next? Well step up the complexity of course, and we'll do that by bringing back our old friend the conditional statements!

Conditionals and Loops in C++

So just like in our Chapter 6 scripts, we're eventually going to have to write a bit of code where what happens next will be based directly on the result of something else, which is to say we're going to have to use a conditional statement to determine something. The great news for you, is that this section will be a breeze and review mostly, because the applications of conditionals work exactly the same in C++ as they do in TS with a few minor changes which I'll go over in just a bit.

If and Else, back once again

We'll start by revisiting the simplest form of conditionals, which is the if statement and its variants. Just like in TorqueScript when you make an if statement, the block contained within the brackets will be executed if the condition evaluates to the true statement. For example:

```
#include <iostream>
using namespace std;
static int myValue = 5;
void main() {
    if(myValue == 5) {
        cout << "It's Five!" << endl;
    }
    else {
        cout << "It's Not Five!" << endl;
    }
}
```

This should be starting to ring some bells in terms of what we were doing a while back in Chapters 6 through 12. And you can still do more complex statements using the logical operators that you learned back in Chapter 6 as well:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
void main() {
    srand(time(0));
    int random = rand() % 100 + 1;
    if(random > 25 && random < 75) {
        cout << "It's Between 25 and 75!" << endl;
    }
    else {
        cout << "Maybe Next Time!" << endl;
    }
}
```

In this example we also do some things with random numbers as well. If you remember back in Chapter 9, I told you how computers don't actually generate random numbers, but instead use a pseudorandom formula based on a seed value to pull numbers from a long string. The **srand** function here performs the seeding and the number we're providing to the seed is the current UTC time **time(0)**. To use these functions however, you need to pull in two extra header files, **<stdlib.h>** for **rand** and **srand** and **<time.h>** for **time**. The random variable will have a number between 1 and 100 stored in it, and then our conditional statement will be evaluated.

Also like in TorqueScript, the **else if** conditional is available to you:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
void main() {
    srand(time(0));
    int random = rand() % 100 + 1;
```

```
if(random > 25 && random < 75) {
    cout << "It's Between 25 and 75!" << endl;
}
else if(random >= 75) {
    cout << "It's Big!!!" << endl;
}
else {
    cout << "Maybe Next Time!" << endl;
}
```

And that about covers our review exercise in terms of the conditional statements **if**, **else if**, and **else**. If you need further review or more examples on the topic, go back to the section on conditional programming using these keywords in Chapter 6.

Switch

Our next review topic is the conditional statement **switch** which acts as a tree of results for a statement, and then you map out the potential results in **case** statements. Unlike in TorqueScript however, C++ does not offer a direct **string** data type. You'll learn soon in Chapter 14 that you either need to use a character array, or a STD class instance to create a string instance. This means that the **switch\$** command you learned in TorqueScript is invalid in C++, instead you need to use the conditional **if** statement in combination with the **strcmp** function (**There are no \$= and !\$= operators in C++**). Therefore the switch statement can only test for numerical, single-character, and boolean data. Here's an example of switch for C++:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
void main() {
    srand(time(0));
    int random = rand() % 5 + 1;
    switch(random) {
        case 1:
            cout << "It's One!" << endl;
            break;
        case 2:
            cout << "It's Two!" << endl;
            break;
        case 3:
            cout << "It's Three!" << endl;
            break;
        case 4:
            cout << "It's Four!" << endl;
            break;
        case 5:
            cout << "It's Five!" << endl;
            break;
        default:
            cout << "It's None Of Them!" << endl;
            break;
    }
}
```

One of the key differences you'll notice right away is that each case statement ends with a break statement. That is required in C++, otherwise the case will fall through to the next one, and the next until it does hit a break statement, however, you can use this behavior to your advantage if you want to stack cases together to make a more complex structure:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
void main() {
    srand(time(0));
    int random = rand() % 5 + 1;
    switch(random) {
        case 1:
        case 2:
        case 3:
            cout << "It's Less Than Or Equal To Three!" << endl;
            break;
        case 4:
        case 5:
            cout << "It's Greater Than Three!" << endl;
            break;
        default:
            cout << "It's None Of Them!" << endl;
            break;
    }
}
```

That's really all you need to know about switch and case for C++, like I said before, you can use numerical, single-character, and boolean data on the case statements in C++, but not string data.

Also a little note before we move onto loops, if you remember back in Chapter 6, I taught you about the conditional operator (?:), this also works in C++, so if you want to use that to perform a conditional operation to set to a variable, you can do that as well.

For Loops

Now we're back to looping structures again. Just like before you have the **for** and **while** loops at your disposal, but this time you also have access to a new type of loop, the **do-while** loop. We'll get to that in just a bit, but let's start with the counter oriented **for loop**.

Just like in Chapter 6, a for loop is usually used when you have a set value that you want to use for the amount of times to perform the loop operation, however you can also establish a more conditional approach, such as testing array values (We'll do this in Chapter 14). All in all, the format you learned in Chapter 6 still applies, but now you'll use a data type for the initializer:

```
for(expression 1; expression 2; expression 3) {
    //code here
}
```

So, take that format of the loop, and apply what you learned from earlier to establish a counter oriented loop:

```
#include <iostream>
using namespace std;
void main() {
    for(int i = 0; i < 10; i++) {
        cout << "Now At " << i << endl;
    }
}
```

Run this code and watch the numbers zero through nine print on the screen with the text **Now At** prepending it. You could also for instance, use a for loop to make a guessing game where the user has three chances to get the value correct:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
void main() {
    srand(time(0));
    int random = rand() % 100 + 1;
    int chances = 3;
    char strGuess[4];
    cout << "Guess The Number..." << endl;
    for(int i = 0; i < chances; i++) {
        cin.getline(strGuess, 4);
        if(atoi(strGuess) == random) {
            cout << "You Got It!" << endl;
            break;
        }
        else {
            cout << "Nope..." << endl;
        }
    }
}
```

So, since we're doing a few new things here, let's tell you what's going on in this bit of code. First and foremost, the variable **strGuess[4]** is a character array (We'll get there soon), and it's used to hold the user's input on the manner. To catch user input in the console window, we use the **cin.getline()** function. The arguments of that function are a character array to store the result in, and the amount of size that can be caught (in character count). Since the number is between 1 and 100, we only need a length of 4 to store the character (Remember, strings end with a NULL terminator). Finally, we have a function called **atoi** which is used to convert characters into numbers. Essentially if you type the string "123" it converts it to the numerical value 123. Everything else in this code is a repeat of what you've already seen.

While Loops

Next up, we have the **while** loop, which also functions in the exact same manner as it does in TorqueScript. You provide a conditional to the while statement, and it will run the loop code so long as the condition remains true:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
void main() {
    srand(time(0));
    int random = rand() % 100 + 1;
    cout << "Random is " << random << endl;
    while(random < 90) {
        random = rand() % 100 + 1;
        cout << "Random is " << random << endl;
    }
    cout << "All Done!" << endl;
}
```

Just remember to avoid the **infinite loops** that were also a problem in TorqueScript. If you create a C++ program that enters an infinite looping structure without a breaking point, you'll hog up the CPU and crash.

Do-While Loops

So, now you're re-learned how to do loops. Except there's always one problem that bugged you in Torque at least it may at some point in time, in a way that you'll need to find a workaround of sorts. Whenever you use a while loop, it always runs the conditional first and then performs the looping structure. This is called a **pre-test loop**. The for loop also works in the exact same manner. Eventually however, you may come across a case where you might need to actually run the loop code first, and then check the condition, this guarantees the loop to run at least once. This kind of loop is called a **post-test loop**, and in C++ terminology, this is called the **do-while loop**.

Creating a do-while loop is essentially just as easy as any other type of loop out there, except there's one little change to syntax you'll learn to accomplish this, here's an easy example:

```
#include <iostream>
using namespace std;
void main() {
    int num = 1;
    do {
        num++;
        cout << "Number is now: " << num << endl;
    }while(num < 1);
}
```

So take a look at this code for a minute. You'll see the while condition at the bottom. This loop will run so long as num is less than one, which would be a problem for a standard while loop because the variable starts as the number one. However, if you go ahead and run this code, you'll actually get the statement "Number is now: 2" to print, and that is because the do-while loop will check the condition after running the looping statement. The change to the syntax here is in the final line, you'll see that right after the closing brace, we place our while statement, and then complete the entire line with a semicolon, basically treating the do-while loop as a class or object structure.

Just like with while loops, you need to be careful when it comes to the infinite loop problem, especially more-so with the do-while loop, because the condition is ran after the looping structure. If you don't double check your code, you may find your program running away with your performance faster than you can stop it.

Final Remarks

And there you have it! See that wasn't too terribly difficult to grasp. You'll notice when you work in C++ that there are many striking familiarities to your work in TorqueScript, and vice versa. That's because once you understand the general workflow to programming, you can essentially pick up any language just by learning the key concepts of the language.

Now, that's great and all for some very simple workflow and script-like programs, but C++ was built to be used to create application interfaces and true application programming interfaces (APIs). And to understand how to bask in the glory of creating something bigger, we'll need to dive deeper into the understanding of the language.

Chapter 14: Diving into the Core of C++

Chapter Introduction

So that wasn't too bad to get things started, right? And you should notice that, aside from core language concepts, the majority of programming is exactly the same across scripting languages and programming languages. There will be many similarities and then some will have some key differences, but in the end the method comes down to the exact same key concepts or variables, functions, and keywords.

But this is where the guide will start to take a turn for the more advanced side of programming. In this chapter I'll dive you head first into the deeper aspects of C++, mainly those that deal with memory, and the concepts to manipulate it for your own advantage. The stuff you'll learn in this chapter will help you to build your understanding of how programming works with computers, and it will prepare you to actually open up the hood of the engine called Torque 3D in the next few chapters. So with that in mind, let's hit the ground running!

Arrays in C++

But wait... what? Arrays? Surely there's a mistake, right? Actually, there is no mistake here, there's a really important reason I held arrays for this chapter compared to giving it to you in the previous chapter. And I'll get to that in just a bit, but let's dust off the old stuff first to make sure you still understand the concept.

Reviewing, More of the Same

So, let's take a step back to Chapter 7 for a minute. You might remember that I said this: Arrays are a type of variable that contain multiple values of a similar type all under one variable access name. Well, let's just say I only gave you half of the story back then. In truth, the definition of an array is a consecutive block of pre-defined memory that stores values of a similar data-type. And the methodology to creating arrays in C++ is almost the same:

```
#include <iostream>
using namespace std;
void main() {
    char myName[8] = "Robert";
    cout << myName << endl;
}
```

You'll see the similar bracket syntax for arrays. So let's play around with this example here for a bit. There's two ways you can define an array instance in C++. The first of which, I just showed you above, but you can also do dynamic allocation this way:

```
#include <iostream>
using namespace std;
void main() {
    char myName[] = {'R', 'o', 'b', 'e', 'r', 't', '\0'};
    cout << myName << endl;
}
```

If you're wondering why I added the \0 to the end, just remember that all strings end with NULL terminator sequences, and that the \0 is a shortcut tag in ASCII to the NULL terminator. Run both of those code instances and you'll get the same output.

Now let's try something else, if you remember back in TorqueScript, you could at any time expand upon an array by just defining a new index, or add to the existing array by simply doing something like:

```
%arr[] = 5;
```

There's just one little problem. This is a **language feature** of TorqueScript, meaning that this does not apply to C++. In C++ arrays are statically defined in memory, basically once you define it, the only way you can expand upon it, is to use memory functions to expand it, and at that point, you're better off using a **list** or a **vector** to accomplish your task. What I'm trying to say here is that when you define an array, you either must provide the amount of space you're planning on using in a **direct numerical form** or you must use the closed bracket notation (example two) and manually define the values.

When I use the term **direct numerical form**, what I mean here is that you must send a constant variable to the parameter. You cannot for instance do this:

```
#include <iostream>
#include <stdio.h>
using namespace std;
void main() {
    char myName[rand()]; //<- Error, Cannot allocate constant size 0.
}
```

Arrays must either be defined using a constant, or a static number value that does not change over the lifetime of the program. Accessing an array value in C++, is also the exact same as you would do it in TorqueScript:

```
#include <iostream>
#include <stdio.h>
using namespace std;
void main() {
    char myName[] = "Robert";
    cout << myName << endl;
    cout << "The Third Letter Is: " << myName[2] << endl;
}
```

This should also remind you here that arrays still use index values compared to numerical positions, so the starting index is zero. Now, I'll get to that first print line in just a bit to explain why that works to print the entire array, but let's move on to the next array review topic.

Multi-Dimensional Arrays

You might also remember that after I taught you how to write a basic array, I showed you how to expand on it, by adding additional “dimensions” to the array. Well, you can do the exact same thing in C++, so long as all of the array indices follow the definition rules:

```
#include <iostream>
using namespace std;
void main() {
    char names[2][8] = {"Robert", "Tom"};
    cout << "Names: " << names[0] << ", " << names[1];
}
```

By this example here, you can see how the dimension values work as a top-down approach, so the first dimension will be the overall “string” instance, and the second dimension defines the amount of space each string can occupy. You can also use the blank size notation on the array, but only on the first index:

```
#include <iostream>
using namespace std;
void main() {
    char names[] [8] = {"Robert", "Tom"};
    cout << "Names: " << names[0] << ", " << names[1];
}
```

And if you wanted to access an individual letter in the names? Well, you just add another identifier to the call:

```
#include <iostream>
using namespace std;
void main() {
    char names[] [8] = {"Robert", "Tom"};
    cout << "Names: " << names[0] << ", " << names[1] << endl;
    cout << "Third letter of name 1: " << names[0][2] << endl;
}
```

You should be starting to pick back up now on the mechanics of arrays from the program standpoint. You define sizes of the amount of data you need to store, and then populate it by defining values to it. Also, just because my above examples only use the **char** datatype, don’t assume that’s the only form an array can take. You can define any variable or even class object to be an array instance, so long as you follow the correct notation.

For-Loop Review

So before we move onto showing you the new things to learn about arrays when it comes to C++, let’s look at one final review topic for arrays, and that is processing through their contents using a loop. Now, back in TorqueScript all variables were treated as strings and arrays did not have fixed size definitions, so your looping structures on them would be very simple to manage. In C++ however, we have fixed boundaries, so you cannot (without some advanced programming which we will not cover) for instance use an expression to fetch the amount of index values used by your array for processing, instead you must rely on fixed sizes:

```
#include <iostream>
using namespace std;
void main() {
    int values[4] = {10, 20, 30, 40};
    int result = 0;
    for(int i = 0; i < 4; i++) {
        cout << "Adding " << values[i] << " to the result..." << endl;
    }
}
```

```
        result += values[i];
    }
    cout << "The Result is: " << result << endl;
}
```

And the same rule applies as well to multiple dimension arrays, just remember that arrays work from a top-down perspective, so the first loop will match up to the first dimension, and the second loop to the second dimension, and so on:

```
#include <iostream>
using namespace std;
void main() {
    int values[2][4] = {10, 20, 30, 40, 10, 20, 30, 40};
    int result = 0;
    for(int x = 0; x < 2; x++) {
        for(int y = 0; y < 4; y++) {
            cout << "Adding " << values[x][y] << " to the result..." <<
endl;
            result += values[x][y];
        }
    }
    cout << "The Result is: " << result << endl;
}
```

The goal here is to understand that when you create an array in C++, that unless you're using a list or vector, that the size instances will be constant in nature, and therefore processing must also match or be less than this constant size. Since C++ does not provide bounds checks on array instances, you could potentially step out of the array bounds and into a restricted portion of memory, causing access violation crashing on your program instances. So, now let's actually talk more about how that works as we introduce you to some new concepts.

New Array Concepts: Memory & Function References

So now that you have a basic understanding of how to actually set up and process arrays in C++, let's talk about why the topic was held off for this chapter. As I just got done explaining, you need to understand that C++ does not provide **bounds checking** which basically means that even though your array instances are of static sizes, C++ will allow a call to a potentially undefined index. Is this an error, yes, but because of the concept of dynamically allocated arrays, C++ must allow this behavior. To explain this further, let's talk more about arrays from the technical standpoint.

Arrays as seen by Computers

So when you define an array in a program, what you are telling the computer to do is to reserve a consecutive block of memory to store values in. Since you know that all variables access memory already from our prior studies, then the next concept should be fairly easy to understand. Let's assume for a minute then, that you're creating an array of integers with the size of 6. You would do something like this:

```
int values[6];
```

What this would do is pick an open spot in memory and then allocate the space for six elements. How does it know how much space to allocate though? Easy, you already told it. It's just a simple mathematical calculation the computer does by knowing the amount of storage you want, and the amount of storage a single integer needs, which you already know from Chapter 13 is 16 to 32 bits. For safety purposes, we'll assume it to be 32 bits, or 4 bytes. The computer then creates a block of memory like so:

values[0]				values[1]				values[2]				values[3]				values[4]				values[5]			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7

Logically from the perspective of this table, the top row indicates the name of the value as you would access it from a program's standpoint. The bottom row then, states the location you would need to access in memory to access it manually. I highlighted the starting value of each of the indices to show the next point. So let's assume that the computer has assigned this array to memory location 0xAEBC1000. To actually access these elements manually, you would need to add the green row's memory address to the starting point. So for example, if I wanted to access values[4], I would access the memory at location 0xAEBC1010.

What you need to realize from this aspect of arrays is that when you start creating arrays of larger and larger sizes that you begin to occupy more of the computer's available memory. For each dimension you add to the array, you are essentially raising that amount exponentially, you don't want to throw a 4-D data cube at the memory and expect to be able to do hardcore processing for example. The goal when it comes to working with arrays is to be compact, but at the same time to be smart about it. Don't let this axe all of your 2D arrays now in favor of two 1D arrays, you'll still occupy the same memory space, but the goal here is to simply say, be smart about your usage of high density arrays in your programs. Now I'll have a lot more on the whole aspect of memory and using it here as it pertains to this section in just a little bit, but for the time being, let's introduce one more important topic for arrays in C++.

Passing Arrays as Function Parameters

One of the big new concepts of arrays is dealing with them in function calls. If you remember in TorqueScript, all variables were either local or global and had an associated name instance. As far as passing variables, you just had to pass the name to the function and you were all set to go. In C++, we're at a bit of a loss of the simplicity of variable arguments, because everything is data type enforced and therefore we need to pass the actual type of variable along with it.

To handle an array as a variable, you just need to make sure you define to the compiler that the argument being accepted by a function is an array instance:

```
#include <iostream>
using namespace std;

int addEmUp(int [], int);
```

```
void main() {
    const int SIZE = 5;
    int numbers[SIZE] = {10, 20, 30, 40, 50};
    int result = addEmUp(numbers, SIZE);
    cout << "Added result is: " << result << endl;
}

int addEmUp(int nums[], int elements) {
    int total = 0;
    for(int i = 0; i < elements; i++) {
        total += nums[i];
    }
    return total;
}
```

Here's a basic example of array parsing to functions. I also did something a little different in this example as well in the function prototype line; you'll see that I did not include variable names. In C++, this is perfectly legal because there is no function body presented here, so when the actual code is loaded in through the override, the parameter names are properly enforced as well.

As for the actual function, I included a size parameter to be used with the for loop because if you remember from a little while ago, C++ doesn't have any means to test the size of dynamic arrays (outside of using more advanced classes). You might be thinking now that the array instance here is not dynamic, and you'd be correct, but remember to the computer, it doesn't have a forward knowledge that the size of numbers will be a constant before it hits the function we wrote, and therefore we must provide a size variable.

Now let's take another step back to one of our old examples and modify it to use functions as well. This time, we'll update the example with the two dimensional array instance. In C++, due to the way computers store multi-dimensional array instance in memory, you must provide the size of the array's bounds for each dimension after the first in order for the definition to be legal, and here's what I mean by that:

```
#include <iostream>
using namespace std;

const int COLS = 4;
const int ROWS = 2;
int addEmUp(int [][]COLS], int);

void main() {
    int values[ROWS][COLS] = {{10, 20, 30, 40}, {10, 20, 30, 40}};
    int result = addEmUp(values, ROWS);
    cout << "The Result is: " << result << endl;
}

int addEmUp(int nums[][][COLS], int rows = ROWS) {
    int result = 0;
    for(int x = 0; x < rows; x++) {
        for(int y = 0; y < COLS; y++) {
```

```
        cout << "Adding " << nums[x][y] << " to the result..." <<
endl;
        result += nums[x][y];
    }
}
return result;
}
```

So just like last time, we skipped out on the names in the prototype, but the key difference this time is that we provided the size of the column element of the array instance to the function definition so the compiler knows that it must expect a column amount of 2 for this particular instance. And that pretty much covers the topic of arrays as it pertains to what we know for now. I'll have much more on this in just a little bit, but for the time being, just understand how to use them in C++, and how the computer treats a definition as it pertains to memory instances.

References

Now it's time to dive into some more tricky things in C++. Now that you know how computers treat array instances in memory, let's actually start working with manipulation of data as it pertains to function definitions and variable instances. What you need to understand for now is that to this point in your C++ programs, you have been passing to a function by the value of a parameter, basically put, when you've been calling a function you gave it a variable name, and then the function you were working with had the ability to modify the variable in the function, but as soon as you left the function, what happened? The value of the function returned back to the original value. This is called **passing by value**, and it's the programmatical term of a read-only variable.

Now don't be mistaken, function most certainly can also write to variables, in fact we've been doing this for a long time already, that's the purpose of the return keyword. Essentially, when a function exits its body by means of a return keyword and a value is sent back to the calling source, you can write the result to the point of the call, which basically means that the function can both read and write to a variable instance. But now we have a little bit of an issue. Let's say I want to write a function instance that needs to have write permissions to multiple variables at once. How do I go about doing this?

To accomplish the feat of writing to multiple variable instances at a time, you need to perform the task of **passing by reference**, which is to say to grant the program read and write access to a particular variable.

The Reference-Address Operator (&)

In order to pass to a function by reference, we need to use the reference-address operator in the definition line to tell the compiler to give the variable reference status. Here's a really basic example of a pass by reference function:

```
#include <iostream>
using namespace std;

void doubleMe(int &);

void main() {
```

```
int number = 5;
cout << "The Number is: " << number << endl;
doubleMe(number);
cout << "The Number is now: " << number << endl;
}

void doubleMe(int &num) {
    num *= 2;
}
```

So what we've done here is created a simple function that doubles the number sent to it. You'll notice in the above example the new syntax of a reference variable when it comes to the function itself, how we used the operator in the definition line. How this works is basically what we just learned about in the previous section. Remember how I said that each variable is stored in memory using a block the size of the variable? Well in our case the variable we made looks a bit like this:

number			
0000	0001	0002	0003

Where once again the green block is the amount we need to add to memory to reach the access point of the variable, since this isn't an array, we just pass the address to the computer. So, assuming we have an address of 0xAF32CD08, then all we've done is accessed this point in the computer to modify the number stored inside of it. This is essentially all we've done by using the reference-address operator. Instead of passing a value to the compiler, which is recognized in C++ by the lack of the 0x in front of the value, the computer knows that when it has the 0x prepending term, that it reads the location in memory, this in turn gives write access to the variable. You can really easily see this in action by using the "address" part of the term reference-address operator:

```
#include <iostream>
using namespace std;

void doubleMe(int &);

void main() {
    int number = 5;
    cout << "The Number is: " << number << endl;
    doubleMe(number);
    cout << "The Number is now: " << number << endl;
    cout << "We read and wrote to: 0x" << &number << endl;
}

void doubleMe(int &num) {
    num *= 2;
}
```

When you run this second example, you'll get the numerical doubling, and then you'll actually see the computer print the location in memory where this variable is stored by using the reference-address operator. We prepend the 0x to the print statement, because the computer will not print the 0x in the print statement.

You can use as many reference variables as you want or need in your specific function example, however it would be much more beneficial at that point to instead use the next topic in its place. What we've done with this section was introduce the concept of working with variables directly through memory, and now let's dig right into one of C++'s most important concepts.

Introduction to Pointers

Our next topic deals with one of the most important aspects of learning C++, which is the topic of **pointers**. So as you just learned with the topic of references, that by using a reference variable, you can access the direct point in memory in which a variable is stored to handle both reading and writing of a variable. So why exactly do we need this? If you think of the life of a program, when you create a variable within the scope of a local function, it lives for the duration of the function and is then removed. While it's true that you can simply establish all of your long-term variables in the main method of the program, the amount of variables would very quickly take off and you would lose track of them all too quickly to keep track of. By using a **pointer** you essentially slate a point down in memory which is used by the particular variable, and can then be accessed again and again by reaching to the address of the variable. We call these pointers because of that very nature, instead of storing a single value, they point to the address in memory in which the value is stored, sound familiar?

You could by that very logic put the notion that a pointer variable is simply an array instance of a variable, however they are a little more in depth than just that. Now we can take an even closer step back to what we were just doing with passing by reference, remember what happened when you used a reference variable in a parameter? You were simply pointing to the location in your computer's memory in which the variable was located. By using a pointer variable, we can simplify the concept.

Pointer Introduction: Variables

Let's start our journey through pointers by looking at a very basic example of a pointer variable in C++:

```
#include <iostream>
using namespace std;

void main() {
    int *num;
    int myNumber = 27;

    num = &myNumber;
    cout << "Number is: " << myNumber << " and it's stored at: 0x" << num
    << endl;
}
```

First off, you should notice the new notation and the pointer operator (*). The variable num isn't actually an "integer" variable per-say, or it's not the integer data type. Instead we call this a pointer to int, or basically an address location in which an integer is stored. Since we store an address in the location of this variable, we can use the reference-address operator to fetch the address of the variable, and throw it at the pointer.

So now let's back up to a few of our examples from the prior chapter. Do you remember all of those **char *** and **const char *** definitions we used a while back?

```
#include <iostream>
using namespace std;

void main() {
    char *myText = "Hello World";
    cout << "Printing: " << myText << " at, 0x" << &myText << endl;
}
```

So what's happening at this block of code? Well, instead of simply defining a single character variable, we're actually defining an array here. This very reason here, is why arrays were held off for this chapter, to connect them to pointers. Don't believe me? Run this code and watch what happens:

```
#include <iostream>
using namespace std;

void printIt(char []);

void main() {
    char *myText = "Hello World";
    printIt(myText);
}

void printIt(char text[]) {
    int len = strlen(text);
    for(int i = 0; i < len; i++) {
        cout << "Char " << i+1 << ":" << text[i] << endl;
    }
}
```

And there you have it, you'll clearly see that the code will treat the myText pointer defined in the main method as an array instance when parsed to the function. This happens because when you declare the pointer instance, you actually allocate the location in memory in which the pointer variable itself is stored. If we actually look at the assign operator (=), you'll see that it actually is just a data access method, or a hard location in memory, passed by value. The pointer variable, is basically a reference variable, so it has both read and write access at the same time. Therefore, when you actually perform the operator, the pointer expands in memory as a static array definition, the computer thinks you're actually doing this:

```
char myText[] = "Hello World";
```

You still need to be very cautious when using pointers and arrays side-by-side, because even though you have this ability now, you still need to remember that C++ has no array bounds checking, it's possible that you could accidentally step outside the bounds of the pointer itself and grab a piece of memory outside of your program's access region.

The Dereference Operator

Now that you know how to create pointer variables, and assign them to points in memory, or define an array instance over the pointer location, now let's actually do some more cool things with them, like pulling values right out of the memory locations themselves.

In order to do this task, you'll need to use something called the **dereference operator**. Basically, if a pointer is a reference to a location in memory, then the value of the pointer is the dereferenced location in memory. Let's do this with our first example again:

```
#include <iostream>
using namespace std;

void main() {
    int *num;
    int myNumber = 27;

    num = &myNumber;
    int val = *num;
    cout << "Our pointer is at: 0x" << num << ", and the value is: " << val
    << endl;
}
```

And now let the confusion set in. So, not only is the asterisk symbol used for multiplication in C++, it's also used to define a pointer variable, and now it's also the dereference operator. However, there's nothing to get worried over. C++ is a smart language, even though it will let you get away with some very silly things like bounds checking, it knows when you're trying to multiple variables versus trying to dereference one.

Anyways, let's explain what's happening with this example, I'll even use a little table diagram this time:



So let's explain how this works. We start by defining a pointer variable called **num**. When this definition happens, there is no storage of the pointer, basically it's in a void position in the computer, only the program knows of its existence. Meanwhile, we create a variable called **myNumber** and give it a

value. So now, somewhere in temporary memory, is a variable called myNumber and it's storing a value of 27. Our next command tells the pointer named num to point to the location of myNumber and copy the contents of the memory stored there into its own position. Now, the pointer num is storing the data that was kept in myNumber. How this is done is by pointing to the original position of the variable myNumber and storing that in the pointer itself. Finally, we create a third variable named val, and in order to fetch the actual value of the pointer num, we dereference it into our final variable. To see the effect on memory, let's expand our example a bit:

```
#include <iostream>
using namespace std;

void main() {
    int *num;
    int myNumber = 27;
    cout << "Created variable myNumber (" << myNumber << ") at: 0x" <<
&myNumber << endl;

    num = &myNumber;
    cout << "Pointed num (0x" << num << ") to myNumber" << endl;
    int val = *num;
    cout << "Created variable val at: 0x" << &val << " and dereferenced
num: " << val << endl;
}
```

When you run this code, you'll notice that the memory locations of myNumber and num match exactly, as they should, and that val has its own position in memory when it accepts the dereferenced value of the num pointer.

So to put things into easy terminology now, pointers access and store memory, and the dereference operator takes a pointer to return the stored value in that particular position in memory. If you're thinking like I hope you are about that last statement, you should also be putting arrays back into thought now. Yep, that's essentially correct, when you're accessing array elements, you're essentially using a dereference operator on the array itself:

```
#include <iostream>
using namespace std;

void main() {
    int *num;
    int myArray[] = {10, 20, 30, 40, 50};

    num = myArray;
    cout << "The array is stored at: 0x" << num << endl;
    cout << "We're dereferencing value 2: " << num[2] << endl;
    cout << "Or we can do this: " << myArray[2] << " which is at: 0x" <<
&myArray[2] << endl;
}
```

So, you see how in the second print statement, we treat the pointer as an array and access the third element of an array? This works because when you're accessing array elements, you're

dereferencing the memory position of the array itself. Run the program, and you'll see the memory address of myArray[2] is actually 0008 spots forwards compared to the starting point of the array itself.

Now, we see in the above example that an array is treated just like a pointer definition, so what happens when we try to dereference the array?

```
#include <iostream>
using namespace std;

void main() {
    int *num;
    int myArray[] = {10, 20, 30, 40, 50};

    num = myArray;
    cout << "Dereference num: " << *num << endl;
    cout << "Defeference myArray: " << *myArray << endl;
}
```

When you run this code, you'll see that when you dereference both the array and the pointer, you simply get the code to print out the first element of the array itself. Why is this the case?

Remember, the dereference operator **accesses the first point in the variable's memory**. Because an array starts at a point and then expands outwards from that location, you're essentially trying to write:

```
cout << "Dereference num: " << num[0] << endl;
```

With the line of code to dereference in its position. In just a little bit, I'm going to introduce the concept of a vector, which is basically a dynamically expanding array instance. To capture the elements as the array expands, it actually dereferences the values at the specified points in the array. But how do we know where we need to dereference? This is extremely simple to do because if you remember the original definition of the pointer, it points to the location in memory **where the specific data type** will be located. Hence, the computer **MUST** know the size of the variable. Bingo, now we know the amount of space our variable will occupy, and now it's just simple math:

```
#include <iostream>
using namespace std;

void main() {
    int *num;
    int myArray[] = {10, 20, 30, 40, 50};

    num = myArray;
    for(int i = 0; i < 5; i++) {
        cout << "Accessing memory at 0x" << &(*(num + i))
            << ":" << *(num + i) << endl;
    }
}
```

Since the computer knows the type of the pointer is an integer, it knows that the amount of memory used by the integer variable in the array element will be sizeof(int), therefore when we simply add the pointer to the loop element, we're essentially telling the computer add the memory position of the array, to i*sizeof(int), which will allow the computer to reach into each memory spot used by the

array. The print statement here uses the exact same code to access the position, and access the location in memory if you look closely. All I did for the memory was add the reference-address operator to prepend the statement.

Comparing Pointers

Now that you have the basics of pointers in hand, let's look at our operators as they pertain to pointer instances. So, many of you out there may be thinking a little forward now, and I do applaud this kind of thinking, that because we can store an entire array in a pointer, then we can magically compare the entire array using our operators like so:

```
#include <iostream>
using namespace std;

void main() {
    int *num1, *num2;
    int myArray1[] = {10, 20, 30, 40, 50};
    int myArray2[] = {10, 20, 30, 40, 50};

    num1 = myArray1;
    num2 = myArray2;
    cout << "Are they equal?" << endl;
    if(num1 == num2) {
        cout << "Yes!" << endl;
    }
    else {
        cout << "No!" << endl;
    }
}
```

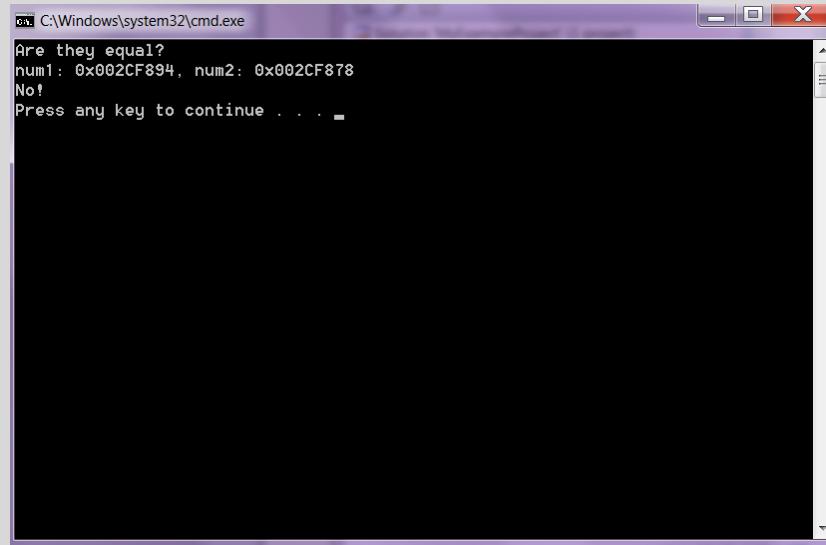
If you ran this program, you'd quickly find out that this is not the case, and the explanation is very simple. Remember what a pointer is, it's a variable with a stored data type that points to a location in memory. Back up to that statement again, what does it store? A memory address. So what exactly is your computer looking at here? Run this example:

```
#include <iostream>
using namespace std;

void main() {
    int *num1, *num2;
    int myArray1[] = {10, 20, 30, 40, 50};
    int myArray2[] = {10, 20, 30, 40, 50};

    num1 = myArray1;
    num2 = myArray2;
    cout << "Are they equal?" << endl;
    cout << "num1: 0x" << num1 << ", num2: 0x" << num2 << endl;
    if(num1 == num2) {
        cout << "Yes!" << endl;
    }
    else {
        cout << "No!" << endl;
    }
}
```

All we've done here is added the memory addresses to our code, and when you run it now, you'll get a window that looks a bit like mine:



And that right there is why your pointers will never be equal to one another in terms of matching with two arrays, because they exist in two different memory locations. You can use all of your standard logical operators to try to compare the pointers, so have at it there. If you want to actually compare the "contents" of the pointer, then you need to use the dereference operator to fetch what's actually stored within the pointer and compare it then. How would we therefore go about doing the pointer check on the array? We add a looping structure and do it again:

```
#include <iostream>
using namespace std;

void main() {
    bool fullMatch = true;
    int *num1, *num2;
    int myArray1[] = {10, 20, 30, 40, 50};
    int myArray2[] = {10, 20, 30, 40, 50};

    num1 = myArray1;
    num2 = myArray2;
    cout << "Are they equal?" << endl;
    for(int i = 0; i < 5; i++) {
        if(*(num1 + i) == *(num2 + i)) {
            continue;
        }
        else {
            fullMatch = false;
            break;
        }
    }
    if(fullMatch) {
        cout << "Yes!" << endl;
    }
    else {
```

```
        cout << "No!" << endl;
    }
}
```

Feel free to experiment a bit more with this example and the prior ones to learn more about how you can compare pointers and their contents. You should even expand your knowledge a bit and try switching up the data types on the pointers as well.

Pointer Variables

At this point, you should have the fundamentals of pointers understood. Just remember that a pointer is a pointer-to-type variable when you are using it, and the word pointer means that the variable is pointing to a location in memory for future use in the code. Now that we've got that part understood, let's enhance our C++ knowledge by using pointers in functions.

The key to understanding pointer variables is to remember at all times what a pointer really is, it's an address in memory, and so whenever you pass to a pointer function, you need to pass the address to that function. Let's go back to our numerical doubling example, but make some changes to it to demonstrate:

```
#include <iostream>
using namespace std;

void doubleMe(int *);

void main() {
    int number = 5;
    cout << "The Number is: " << number << " and is stored at 0x" <<
&number << endl;
    doubleMe(&number);
    cout << "The Number is now: " << number << endl;
}

void doubleMe(int *num) {
    *num *= 2;
}
```

Now when you run this code, you get the exact same result as the old pass-by-reference example, ergo when you use a pointer as a variable, you are passing by reference, right? Well, not exactly. It's true that the behavior of this code is treated exactly like the pass by reference behavior, but this happens for one reason, because you're manipulating the actual location of the variable in memory. By accessing the direct location of the value in memory, versus passing a copy of the value, you have both read and write permissions on the original value, and not just holding a copy of it in temporary storage. And remember, we use the address-reference operator on the function call because the function is expecting a pointer.

What this should paint in your head is a picture of the operators now, the address-reference operator “pointerizes” the variable, and the dereference operator “de-pointerizes” the variable, per-say.

Now, let's do something even more interesting using the pointers. Instead of starting with a defined value from main, let's actually create a value in another function, and then use it in the main call itself.

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

void generateNumber(int *);

void main() {
    int myArray[10];
    cout << "Fetching an array from memory..." << endl;
    cout << "What did we get?" << endl;
    generateNumber(myArray);
    for(int i = 0; i < 10; i++) {
        cout << "Value #" << i+1 << ":" << myArray[i]
            << " (" << &myArray[i] << ")" << endl;
    }
}

void generateNumber(int *arrayMe) {
    srand(time(0));
    for(int i = 0; i < 10; i++) {
        *(arrayMe + i) = rand() % 100 + 1;
    }
}
```

Run this program, and you'll see how there will be random numbers between 1 and 100 populating the array structure. This works because just like in the last example, we're essentially giving write access to the variable myArray when we pass it as a pointer to the generateNumber function, therefore the function is free to manipulate the values as it sees fit, and then by the time we get back to the main function, the array is full of values and they can safely be printed to the console window.

Before we move on, let's talk about one more thing which may come back to cause some C++ headaches in the future unless you gain mastery over it, and that is the topic of constant variables. Remember back in Chapter 13, I introduced variable modifiers that you could place before the data type to adjust the behavior of a variable as it pertains to the overall program. One of the more restrictive forms of these type modifiers is the constant modifier, which basically states that the contents of the variable are constant and cannot be adjusted by the code. Just like all other data types however, you can also establish a pointer to a constant data variable.

When you work with a constant variable type, you need to use a pointer to that constant variable. The most common type you'll see when working with C++ is the **const char *** type, which is essentially the closest thing you'll get to a string without using any external classes. The reason constant character arrays are so useful as string definitions is because no outside functions can come in and manipulate the memory of the string once it's set. This means when you want to perform

operations such as copying the contents of one string into another, it's best to treat the string you're copying from as a constant as to ensure that the original string remains untouched. Here's an example:

```
#include <iostream>
using namespace std;

void copyStr(const char *, char *);

void main() {
    char theName[80];
    copyStr("Robert", theName);
    cout << "Name: " << theName << endl;
}

void copyStr(const char *src, char *trg) {
    int len = strlen(src);
    for(int i = 0; i < len; i++) {
        trg[i] = src[i];
    }
    trg[len] = '\0';
}
```

Now you could have just as easily used the `strcpy()` function provided by C++ here, but this is a basic example of how something like this works. Just remember that when working with arrays of any form that you need to ensure the code has enough room to use for the memory allocation as to not go beyond the bounds. We'll get to a fix for this problem in just a bit, but we have a few more topics to cover before that.

Returning Pointers from Functions

Our next topic has some more advanced aspects to it, but some important aspects as it relates to topics we'll be getting at in the future. We've worked with pointers so far, we've initialized them, modified them, messed with their values, and now have even copied the values from one to the other. So what haven't we done yet? How about directly creating a pointer and then returning it to the calling function.

Now you might be inclined to simply copy one of these old bits of code I wrote and simply slap a return statement on there and think that you're golden, but there's just one little tiny problem. A pointer that is for instance defined through memory in a local scope will fall out of scope once the function ends, resulting in an access violation in your program. What do I mean by this? If you try to run the following code, you'll have problems:

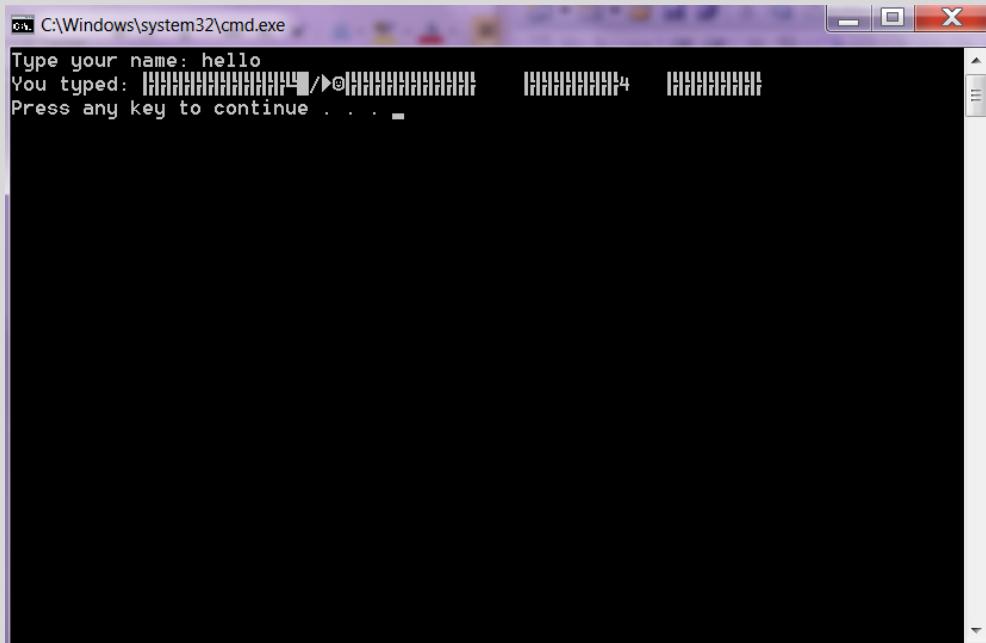
```
#include <iostream>
using namespace std;

char *getText();

void main() {
    char *name = getText();
    cout << "You typed: " << name << endl;
}
```

```
char *getText() {  
    char text[80];  
    cout << "Type your name: ";  
    cin.getline(text, 80);  
    return text;  
}
```

What happens when you run this code? Once you type the name in and hit enter, WOW what a mess!



So what exactly happened here? The problem with your code there is once you enter the `getText` function, the program's scope moves within the bounds of that function. Therefore the array we created in the function `getText` falls out of scope once we leave the function, you're essentially passing a copy of the text at that point, which existed in temporary memory. The pointer points there, and when you try to access it, you've got a garbled mess of nothingness.

Fixing this problem is surprisingly simple, and there's two ways you can accomplish it. The first way is to pass the variable you are planning on returning as a pointer of the same type, like so:

```
#include <iostream>
using namespace std;

char *getText(char *);

void main() {
    char name[80];
    getText(name);
    cout << "You typed: " << name << endl;
}

char *getText(char *text) {
```

```
    cout << "Type your name: ";
    cin.getline(text, 80);
    return text;
}
```

By doing this, you're granting the `getText` method full write access to the `text` variable in the function. It's true that at this point, you're better off simply using a void function with no return statement, but this does fix the issue.

The other way is to actually solve the problem implemented by the local scoping of the variable by creating the actual memory space in the function definition for your program to use. To accomplish this form of the fix, we bring back the **new** keyword:

```
#include <iostream>
using namespace std;

char *getText();

void main() {
    char *name = getText();
    cout << "You typed: " << name << endl;
}

char *getText() {
    char *text = new char[80];
    cout << "Type your name: ";
    cin.getline(text, 80);
    return text;
}
```

You'll learn a lot more about `new` in Chapter 16, but for the time being, just know that the `new` keyword creates the specified object or type instance and returns a pointer to the location in memory that was used. Since the local code this time around actually creates the memory space used by the variable and then stores the result, you're free to return the pointer to the `main` method and actually use it.

The key point to remember here is that even pointers are still treated as variables, and as such the concept of variable scope does apply. Just like in TorqueScript, variables that are not "global" cease to exist outside of their predefined scope. The rule to follow here is simply put to ensure variables, especially pointers that need to exist beyond the scope of their defining function, must be created inside memory before you proceed to use it beyond the scope of the defining function. We'll cover this much more in Chapter 15 when we talk about how to properly use memory in the engine, but you need to be aware of this problem when you try to create a function instance that returns a pointer of a variable type.

Double-Pointers

Our final topic for pointer introduction comes in the form of introducing multiple pointers. To this point, we've been using a single pointer to fetch the address of a variable instance in memory and then to manipulate it. We've made the direct association to array instances using pointers, showing how

by manipulating direct address references, you can access the individual points of a variable in memory. So now let's expand on this topic and introduce multiple pointers, specifically double pointers.

As you now know, a pointer is basically a variable that points to a particular location in memory. By this logic then, a double pointer instance is a pointer variable which points to the location of a pointer in memory. To this point in time, you've only seen one instance of a double pointer, so let's recall it now:

```
#include <iostream>
using namespace std;

int main(int argc, const char **argv) {
    for(int i = 0; i < argc; i++) {
        cout << "Parameter " << i << ":" << argv[i] << endl;
    }
    return 0;
}
```

If you look at the argv variable in the above example, you'll see the double pointer notation is just adding another asterisk to the variable itself. This variable definition is specifically saying to pass to this function call a pointer to a constant character pointer, or to pass the address of the constant character pointer of argv.

```
#include <iostream>
using namespace std;

void main() {
    char n[][4] = {"You", "Me"};
    cout << "0: " << *n << ", 1: " << *(n+1) << endl;
    cout << "Double-Dereference: " << **(n+1) << endl;
    cout << "Fetching Second Letter: " << (*n)[1] << endl;
}
```

For most cases in C++, you'll likely never work beyond a single level of pointer instances. The most extreme cases of double pointer usage comes when working with object lists or high element count array instances where direct memory access is preferred over standard array processing.

The Generic Pointer, Type-Casting, and Templates

Now that we've established our introduction to working with pointer instances, let's move a little deeper into working with pointers. This section will teach you about a special type of pointer instance, and how to use this pointer to adapt a code to accept numerous data types in one location, and then I'll introduce C++'s template keyword which is the standard for using any data type at a specified location.

The Generic Pointer

Gaining mastery in C++ is essentially how you learn to adapt to a numerous set of available tools and tricks to get your program completed. One of the more "sneaky" tricks is the **void pointer** which is also sometimes referenced as the **generic pointer**. To this point, you have only seen one application of the keyword **void**, and that is to tell a function that it has no return value. There's another application of

the keyword, and that is the void pointer. This is a special pointer instance that can accept any value or object to store inside of it. This allows for a huge amount of template variety functions, where a generic pointer can be defined to tell a function it can accept numerous values.

```
#include <iostream>
using namespace std;

void main() {
    int v1;
    double v2;
    char v3;

    void *voidPtr;
    voidPtr = &v1;
    cout << "Address 1: 0x" << voidPtr << endl;
    voidPtr = &v2;
    cout << "Address 2: 0x" << voidPtr << endl;
    voidPtr = &v3;
    cout << "Address 3: 0x" << voidPtr << endl;
}
```

This is a great tool to use as you can imaging, and right now you're probably thinking to heck with C++'s data types and to just use void pointers everywhere with the dereference operator. There's just one little problem to stop that notion dead, and that is the problem that you cannot use the dereference operator on a void pointer. You need to remember that a pointer is simply a variable that points to a location in memory. When we defined pointer variables a little while back, they were always given a specific type, and therefore the compiler knew what was stored inside. In this instance here however, the compiler doesn't know what is kept inside of the variable. To actually dereference a void pointer, you need to **type-cast** the variable.

Type-Casting

Back in Chapter 13, I gave a slight introduction of this topic in a few of the examples, now we'll actually dive into it. The concept of **type-casting** is basically to take a variable of one data type and convert it to another data type. There are two primary types of casting and you've seen both of them used already. The first type is called **implicit** casting, which happens when you for instance send a variable of a "similar" data type to a function. The compiler does the best it can to handle the conversion itself. This is a frequent form of casting, and it's mainly used when you typecast for instance a char to a const char, or an int to a float, basically as long as the types are similar in nature, they can be implicitly casted. To look at an example of implicit casting, let's revisit our string copy method example from a little while back with one change:

```
#include <iostream>
using namespace std;

void copyStr(const char *, char *);

void main() {
    char theName[80];
    char *copyFrom = "Robert";
```

```
    copyStr(copyFrom, theName);
    cout << "Name: " << theName << endl;
}

void copyStr(const char *src, char *trg) {
    int len = strlen(src);
    for(int i = 0; i < len; i++) {
        trg[i] = src[i];
    }
    trg[len] = '\0';
}
```

You can see in the updated example here that we're defining the string that is being copied from (src) as the data type `char *`, which obviously doesn't match `const char *`, to fix this problem the compiler will implicitly type-cast the `char *` to a `const char *` before executing the function call.

The other type of casting is called **explicit** casting, which is where you tell the compiler the data type that it needs to cast to. To accomplish this form of type-casting there are a number of keywords, and syntax types you can use. The most common form of type-casting under this definition is the **static cast** which is used to directly convert one data type to another if it is possible. To accomplish this form of casting, there are two options:

```
#include <iostream>
using namespace std;

void main() {
    int v1 = 10;
    double v2 = static_cast<double>(v1);
    double v3 = (double)v1;

    cout << "v1: " << v1 << ", v2: " << v2 << ", v3: " << v3 << endl;
}
```

The first option is to use the **static_cast** keyword, followed by a pair of GT/LT symbols containing the type you wish to convert to. The other way to accomplish this is to stick the desired data type inside a pair of parenthesis next to the variable you're interested in converting. The only other static casting method you need to be concerned about at the moment is the **const_cast** keyword, which allows you to type-cast up or down a constant variable definition, however most of the time, you can just let the compiler implicitly typecast these forms of variables for you.

You will frequently come across cases when programming in C++ where type-casting will be needed in order to accomplish a task. Just know the two forms in which type-casting is used and how to handle both cases of these type-casts.

Templates, Revisiting the Generic Pointer

Now that you know a little bit about the generic pointer, and type-casting let's go ahead and apply both of the concepts to what we're really after here, which is a tool to pass any form of data through a function instance. There is still one little hurdle to overcome when passing a void pointer through a function, and that is somehow communicating with the destination function to tell it what

exactly the pointer is holding in order to safely dereference the pointer instance. There are numerous ways you could actually accomplish this job, they include things like setting a flag variable, using a string to tell it, and even an enumeration (Chapter 16). However, the method I find to be the easiest, is to use what is called a **template function**.

In C++, a **template function** is a function that pre-defines a set of template variables that can then be used throughout the body of the definition. The syntax of a template is very easy to understand:

```
template <typename T1, ..., typename N-1, typename N> function() {  
    //Body  
}
```

You start with the keyword `template`, and then enclose in a pair of LT/GT symbols a list of `typename` variables, the `typename` keyword here tells the compiler that this specific template variable can be of any data type. You could easily replace that with your standard data-type definition as well. After the closing symbol, you define a function as you normally would. Here's a basic example of a print function:

```
#include <iostream>  
using namespace std;  
  
template <typename T> void printIt(T toPrint) {  
    cout << "Printing: " << toPrint << endl;  
}  
  
void main() {  
    int v1 = 10;  
    double v2 = 5.5;  
    const char *v3 = "Robert";  
  
    printIt<int>(v1);  
    printIt<double>(v2);  
    printIt<const char *>(v3);  
}
```

This example shows you how to define a template function and how to call a template function. You'll see the familiarities in the definition line beyond the `typename` definition, the letter `T` in the variable `toPrint` tells the compiler that the `toPrint` variable is of type `T`, which is specified by you when you actually make the call to the function (Enclosed in the LT/GT symbols). If you wanted to expand upon this function, you could then very easily modify it to accept all three of these variables at once:

```
#include <iostream>  
using namespace std;  
  
template <typename T1, typename T2, typename T3> void printIt(T1 v1, T2 v2,  
T3 v3) {  
    cout << "Printing: " << v1 << ", " << v2 << ", " << v3 << endl;  
}  
  
void main() {  
    int v1 = 10;
```

```
    double v2 = 5.5;
    const char *v3 = "Robert";

    printIt<int, double, const char *>(v1, v2, v3);
}
```

Also, never be afraid to expand on your code here, and use advanced tools such as pointers and references in your template variables. You can even define return types under the definitions provided:

```
#include <iostream>
using namespace std;

template <typename T1, typename T2, typename T3> T1 printIt(T1 v1, T2 *v2, T3 v3) {
    cout << "Printing: " << v1 << ", " << *v2 << ", " << v3 << endl;
    return v1;
}

void main() {
    int v1 = 10;
    double v2 = 5.5;
    const char *v3 = "Robert";

    int result = printIt<int, double, const char *>(v1, &v2, v3);
    cout << "Result: " << result << endl;
}
```

Obviously this above code would have catastrophic consequences if you sent a type other than int to the first parameter, but you get the point of this specific case. You are free to use the newly defined T1, T2, and T3 as replacements for standard C++ type keywords. I'll have more on this in the next chapter, but for now just understand that using template functions is one way to use new data type definition lines.

With this in mind, let's now back up to our generic pointer example. So beforehand, the compiler would have no knowledge of what was contained within the generic pointer, and therefore dereferencing it would be impossible, but if you send the pointer through a template function for example, you can then use the template itself to retrieve the data type:

```
#include <iostream>
using namespace std;

template <typename T> void passRef(void *);
void printIt(char *);
void printIt(int *);

void main() {
    char *myName[] = {"Robert"};
    int nums = 10;

    void *ptr;

    ptr = myName;
    passRef<char*>(ptr);
```

```
    ptr = &nums;
    passRef<int*>(ptr);
}

template <typename T> void passRef(void *ptr) {
    T v = static_cast<T>(ptr);
    cout << "Printing: " << *v << endl;
}
```

You can see in the above example how we now send the pointer type directly to the function. You need to be careful when reading this code and thinking there may be a problem in the template call. Just because you see a T without a pointer, doesn't mean that the variable isn't already a pointer. Go back and actually look at the call if you think this is the case, you'll notice that T becomes both char ** and int * when being passed through. Once the conversion is handled, then it's just a simple dereference of the pointer as we have done numerous times already.

By combining these tools in your code, you can adapt the needs of the project to suit the code whenever and wherever it is needed. With that in mind, let's move onto our next topic in this chapter.

Memory Tools

Our next topic aims to introduce the process in which memory is allocated, reallocated, and deleted in a program by means of C++. Once you complete this section, you should have another set of tools at your disposal to safely create pointers to be returned inside of a function definition. We'll cover this topic a bit more in Chapter 15 when we cover TorqueScript function making, but for now I'll cover the basics.

Memory Allocation

If you ever choose to study C++, or dive deeper into coding in C++, one of the common things you'll hear about is an argument on using **new** and **malloc**. C++ introduced the **new** keyword in a way to allocate a specified amount of memory for an array instance or to create a pointer instance while calling the constructor (Chapter 16) at the same time. The **malloc** operation on the other hand is used to define a block of memory with the specified size that can then be type-casted to what you need and worked with. Many people out there would then say it's a bad thing to bring these two together because each is to be used in their own language; I kindly disagree with these individuals.

While it's true that using the **new** keyword is a safer form to allocation, it really restricts you to defining an array instance, or a class instance. Remember as I said before, data types sometimes comply to ranges of sizes, and those vary based on the compiler and the computer running the program instance. That's why sometimes you may need to break out of the C++ standard and use the **malloc** and **calloc** functions.

malloc & calloc

So, let's actually use the function here. Before we start, I'll tell you about these two functions. **Malloc** is a function designed to do one thing, allocate a block of memory of the specified size, it does not assign a value to the function and it does not provide a means to order the memory. **Calloc** on the other hand will find a block of the specified size and order it for your purposes and then assign the block

to fill with zeros, this will ensure any overhead isn't wasted. As you can imagine though, this does take a little more performance to accomplish, so as I said earlier in this guide, your programming choice once again will come down to a performance situation, where do you choose to sacrifice a bit of safety for faster processing, or do you choose to take a performance hit to prevent bugs and errors?

Once you make that decision, you actually need to use the function in your program. Both of these functions will either return a void pointer to the space reserved, or a NULL pointer if it could not allocate the space, here's how you'd run this function on one of our earlier examples:

```
#include <iostream>
using namespace std;

char *getText();

void main() {
    char *name = getText();
    cout << "You typed: " << name << endl;
}

char *getText() {
    char *text = (char *)malloc(80);
    if(text != NULL) {
        cout << "Type your name: ";
        cin.getline(text, 80);
        return text;
    }
    return "NULL PTR";
}
```

The function above will define a block of memory 80 bytes long for you to fill with character data. Since malloc returns a void pointer, we type-cast it to a character pointer so the compiler knows what's being held within. The conditional we use in this function verifies that the memory safely initialized and prevents us from trying to work with a block of memory that is not available to us. The only difference between malloc and calloc on the program level is the size parameters used. If we wanted to rework the example to use calloc, we'd make a few adjustments:

```
#include <iostream>
#include <stdlib.h>
using namespace std;

char *getText();

void main() {
    char *name = getText();
    cout << "You typed: " << name << endl;
}

char *getText() {
    cout << "Type your name: ";
    char entry[256];
    cin.getline(entry, 256);
    int len = strlen(entry);
```

```
char *text = (char *)calloc(len, sizeof(char));
if(text != NULL) {
    strcpy(text, entry);
    return text;
}
return "NULL PTR";
```

You'll also notice that we need to use the stdlib.h header file if we want to use the calloc function in our program. Again we do the null pointer test on the allocated block for safety reasons.

There are both advantages and disadvantages of using malloc versus the new keyword. As I said before, the advantage is the ability to have user defined size allocations for memory blocks, this opens up more powerful tools such as dynamic array allocation (remember earlier, that we could soon), you can even expand your array to be completely dynamically sized, just like it is in TorqueScript. The key disadvantage of using the malloc function in C++ is that it does not have built in tests for the memory instancing on the order of the new keyword, where a C++ built in exception is thrown if there are any problems with the allocation. New also calls the constructor of a class or data-type instance allowing you to handle post-initialization for all instances of that type defined in your code.

Just remember that malloc defines memory on the byte level, so if you plan on defining an array instance, that you need to accommodate for both the size of the elements, as well as the size of the array. The easiest way to ensure you're getting the number correct is to allocate using the maximum desired number of array elements times the size of the data type that will be occupying the space.

The new keyword

The other choice of memory allocation comes in the form of the new keyword, which creates a pointer to the specified data type, or a pointer to the specified object instance. We'll cover the latter of the two in Chapter 16, but for now we're going to focus primarily on the creation of memory elements using the new keyword. We accomplished this task a little while back in the exact same example I was pulling from, here's a recap of it:

```
#include <iostream>
using namespace std;

char *getText();

void main() {
    char *name = getText();
    cout << "You typed: " << name << endl;
}

char *getText() {
    char *text = new char[80];
    cout << "Type your name: ";
    cin.getline(text, 80);
    return text;
}
```

You can see in the example that we've created memory for a character array of 80 elements. The new keyword essentially uses the malloc command to create the space in memory and then test the allocation to ensure that it succeeded. Since we're creating an array instance, the keyword creates a character pointer to store the address of the array in and then returns it which is ultimately stored in our text pointer.

You can also use the new keyword to directly initialize a pointer of a data type. For instance, let's say we have a numerical value that we need to store inside a pointer for later use. We can easily accomplish this by means of the new keyword:

```
#include <iostream>
using namespace std;

int *fetch();

void main() {
    int *v = fetch();
    if(v != NULL) {
        cout << "Got a pointer at 0x" << v << ", it has: " << *v << endl;
    }
}

int *fetch() {
    int *value = new int(27);
    if(value != NULL) {
        return value;
    }
    return NULL;
}
```

Run the program, and you'll get the address of the initialized variable, as well as the number 27. So you'll notice here that the initializer of the int data type does not specify to the compiler that you want an int occupying 27 bytes, instead it places the value 27 inside the integer.

So as I said before, using the new keyword has the advantage of directly defining and calling the constructor of whatever you are creating, and it provides some additional protection on the memory being created by means of internal error checks. You lose out on a bit of freedom in terms of dynamic sizing, but I'll show you a little trick around this in a bit.

For the long haul, you'll likely be sticking with the new keyword in C++, however don't ever be afraid to step out of the box to use the malloc or calloc keywords in particular places where they might be a necessary component of your project. So now that you know how to safely create memory in a program, let's show you how you can manipulate the sizes of previously defined blocks of memory to gain more dynamic size freedom in your programs.

Memory Reallocation

So now that you know how to create memory blocks in C++, and have a little bit of understanding on the process of memory allocation, let's show you how to **reallocate** memory in C++. By using this process you can essentially re-size blocks of memory to either contain sizing for more or

less usage of it. To get a concept in an “Array” term, this essentially would allow you to expand or contract the array’s size to get the most (or least) out of the array instance you’ve defined. Contrary to a thought that may have entered your mind, don’t start nil-defining arrays using the [] notation and expect to access array elements that still don’t technically exist. To actually use these portions of an array in C++, you’ll need to use the **pointer-dereference method** we demonstrated a little while ago.

Before we get there however, let’s actually introduce the method behind the scenes of memory reallocation and teach you how to use it.

The realloc function

This behavior is accomplished using the realloc function in C++. To use the realloc function, we use a similar format to that of the malloc function, where the return type is a void pointer that will be type-casted to that of the memory space you plan on using. The arguments are simple to learn as well. The first is a void pointer of the pointer whose memory size you wish to alter, and the second is the new size in bytes you wish to adjust the memory to. If the new size is equal to or smaller than the current size, the address will not be modified, and if the size is larger, the address may stay the same, or move based on the available storage needed by the new pointer definition. Here’s an example of how to use the realloc function:

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void main() {
    cout << "Allocate Array" << endl;
    int myNums[] = {10, 20, 30, 40, 50};
    int newCount = 6;
    int *nums = NULL;
    int *newPtr = (int *)realloc(nums, newCount * sizeof(int));
    for(int i = 0; i < 5; i++) {
        *(newPtr + i) = myNums[i];
    }
    *(newPtr + 5) = 60;
    cout << "Print" << endl;
    for(int i = 0; i < 6; i++) {
        cout << i+1 << ":" << *(newPtr + i) << endl;
    }
}
```

So there’s quite a bit going on in this function, so let’s break it down. First and foremost, we have an array instance with five numbers in it. And we need to either expand the array, or we need to expand a space of memory to occupy more numbers. What we did in this example was define a NULL integer pointer and then expanded it to hold six numbers instead of five, and then we filled the new pointer using the contents of the old array, plus added our next number to the list. Finally, we printed the instance.

Now someone will look at this example ad say, clearly you’re not expanding the existing array and you’d be correct. In the above example we’re not expanding the existing array, instead we’re

expanding a space of existing memory to occupy the new space. The concept of expanding an array itself in C++ is simply not possible, since the contents of the array instance are statically defined on the heap of your computer. Attempting to reallocate this memory will result in heap corruption, and will crash your program. To “safely” reallocate an array instance, you need to cast it to a pointer and then reallocate the pointer of your array pointer. I’ll cover this a little bit more when we get to vectors here in just a little bit. So, expanding on our prior example, you get this:

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void main() {
    cout << "Allocate Array" << endl;
    int myNums[] = {10, 20, 30, 40, 50};
    int newSize = 10 * sizeof(int);
    int *nums = (int *)malloc(5 * sizeof(int));
    for(int i = 0; i < 5; i++) {
        *(nums + i) = myNums[i];
    }
    void **arrayPtr = (void **)(&nums);
    cout << "Safely Re-Allocating Array" << endl;
    *arrayPtr = *arrayPtr ? realloc(*arrayPtr, newSize) : malloc(newSize);
    nums = (int *)(*arrayPtr);
    for(int i = 5; i < 10; i++) {
        *(nums + i) = (i+1)*10;
    }
    cout << "Print...." << endl;
    for(int i = 0; i < 10; i++) {
        cout << "Number " << i+1 << ":" << *(nums + i) << endl;
    }
}
```

So again, we have our array instance and we copy the contents of our array over to the new pointer we defined. We then acquired the pointer of our newly defined pointer and passed that to our conditional under the realloc function. This test basically ensures that the contents of the old array were indeed made and to simply allocate new space for it if it was not copied over. Finally, we add our new numbers into our expanded pointer and print the contents.

The concept to learn from this section is that while realloc does not directly create dynamically allocated arrays, it can be used as a resize tool that can be converted into this feature. I’ll show you in just a little while how the realloc function can be combined with the other tools you’ve learned here to actually create arrays that function the way they did in TorqueScript.

Free & Delete

Our last segment on memory for this chapter comes with this section. There are two more function/keyword style things you need to learn about. These are the **free** and **delete** tools. The **free** function was used back in C to clear out allocated memory. What I never explained beforehand was that during the life of a program, memory that is allocated for use in the program is never deleted, so when you use the new and malloc functions that memory persists until the program closes. You’ll want to get

into the habit of cleaning up memory that you are no longer using. For the most part, you won't need to touch these functions outside of the occasional call to a pointer that is being removed. There isn't that much to cover here, so I'll make it short and quick.

Whenever you use the malloc or calloc functions to create memory, they should have an equivalent **free** call when you are done using it. Items created with the new keyword should be cleaned up with the **delete** keyword. Here's a full program example of using these two tools:

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void main() {
    int *v1 = new int(27);
    int *v2 = new int[3];
    for(int i = 0; i < 3; i++) {
        *(v2 + i) = i * 10;
    }
    int *v3 = (int *)malloc(sizeof(int));
    *v3 = 10;
    cout << "Using the variables..." << endl;
    cout << "v1: " << *v1 << endl << "v2: ";
    for(int i = 0; i < 3; i++) {
        cout << *(v2 + i) << " ";
    }
    cout << endl << "v3: " << *v3 << endl;
    cout << "Deleting/Freeing" << endl;
    delete v1;
    delete[] v2;
    free(v3);
    cout << "Complete..." << endl;
}
```

You should also notice one thing about this program instance, only pointers are being freed and deleted, and that is because of the scope concept from all the way back in Chapter 5. Remember, that once a variable falls out of scope, it no longer exists so in C++ non-pointer variables fall out of scope once the function is exited, and therefore you do not need to delete them. Also, mind the delete notation when trying to delete an array pointer, if you do not use that notation you'll only delete the first element, leaving the others behind which may cause some issues if you're not careful about it.

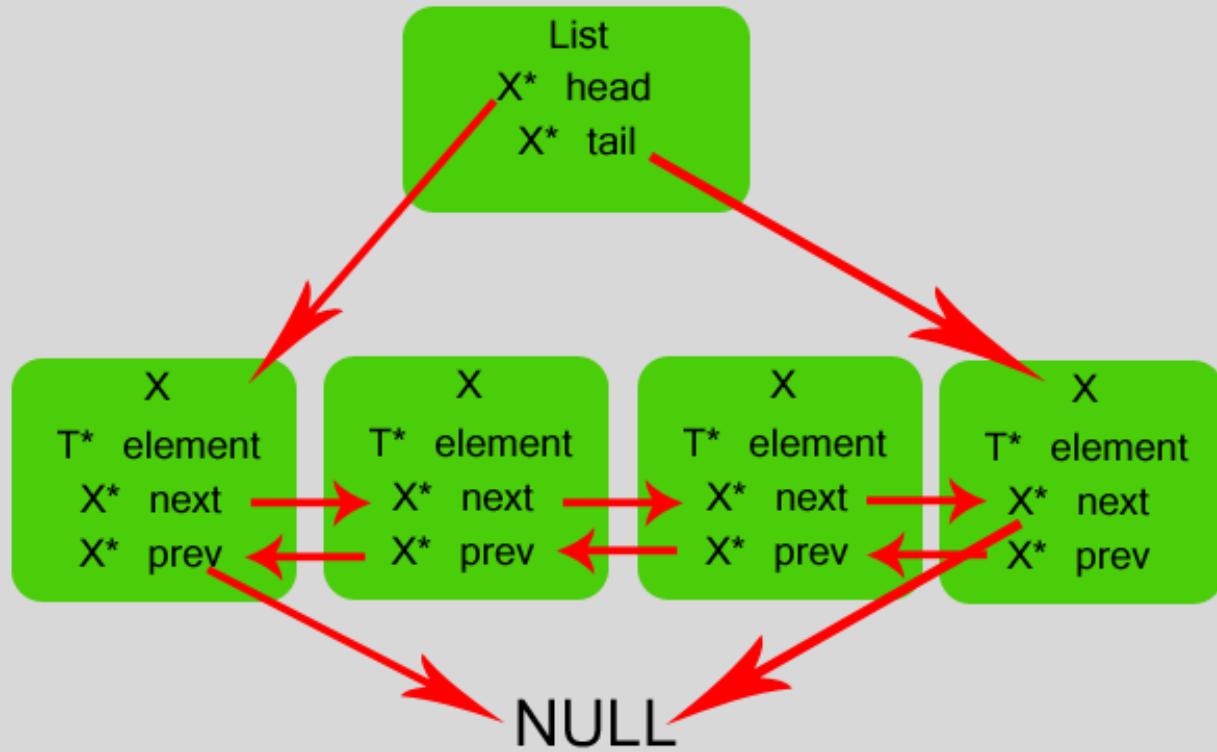
Lists, Vectors

Now that we have completed our long section of the chapter on pointers and memory, we can start to actually move into applications of these tools as they pertain to C++. The next two tools I'm going to teach you about are **Linked Lists** and **Vectors**. For the purpose of this chapter I'll teach you about the Standard Template Library's version of these tools, but you should also be aware that there is an internal definition of both of these tools available through Torque and when working inside the engine, you should replace all definitions with the Torque versions.

Introduction: Linked List

The first of these tools I'm going to introduce is the **linked list**. A linked list is a sequential list of containers that allow insert and delete operations anywhere in the sequence and iteration through the list in either of the two directions. Usage of a linked list has advantages and disadvantages over the second container I'll teach you about. The primary advantage of a linked list is the performance gain when it comes to inserting, extracting and moving elements within the container when you have a prior knowledge of the iterator of the element. On the other hand, the disadvantage is that in order to do any extraction of an unknown element, you need to iterate through the full list in order to find what you're looking for, which is to say that there is no direct access on the individual elements within the list. Also, when it comes to memory storage, the linked list uses up additional memory in order to make the linking references between the elements.

To get an idea of the standard implementation of a linked list, you can use the following picture as a reference of the setup:



You can see in the diagram above that each element in the list contains the data you want to hold within the list, as well as a reference between the next and previous elements contained within the list itself. On the top of the list we have a second instance that stores the first and last element addresses for quick access to the list itself. You may now be thinking, but wait a minute each of those individual boxes contains multiple references and types. How would I even manage to do that in a program, especially trying to keep control of each of those pointers?

Structures

There are two possible routes to go with the answer here. You could implement a class to store the elements inside (Chapter 16), or we could go a little simpler and implement a **structure**. From the technical standpoint, both a class and a structure do the same thing, but in implementation, you don't need to worry too much about structure definitions because the members are all public by default (Again, Chapter 16 topic).

Just like before in Torque when we introduced the concept of working with objects, the same applies to this section. Each structure you create in C++ is treated as an object, therefore it exists in memory either locally in scope, or globally when defined as a static memory location.

Defining a structure is also very easy, take a look at this next example on how we accomplish that task:

```
#include <iostream>
using namespace std;

struct testStruct {
    int num1;
    int num2;
    int num3;
};

void main() {
    testStruct a;
    a.num1 = 10;
    a.num2 = 20;
    a.num3 = 30;

    cout << "The numbers are: " << a.num1 << ", " << a.num2 << ", " <<
a.num3 << endl;
}
```

A word of caution here, avoid thinking of the pre-define notation for structures like you did for your functions back in Chapter 13, C++ requires foreknowledge of the contents of the structure before you can actually use it, so the following would be invalid to C++:

```
#include <iostream>
using namespace std;

struct testStruct;

void main() {
    testStruct a;
    a.num1 = 10;
    a.num2 = 20;
    a.num3 = 30;

    cout << "The numbers are: " << a.num1 << ", " << a.num2 << ", " <<
a.num3 << endl;
}
```

```
struct testStruct {
    int num1;
    int num2;
    int num3;
};
```

So let's go back and look at the first example again, you'll see that a few old formats make a return. First of all, when you define a structure itself, it behaves like our old object definitions in Torque, so don't forget the semi-colon at the end of the closing brace of the structure definition. Also, our old **object-access** operator makes a return in the main function, you'll see here how we use it to access the contents of the structure, and print them in the print statement.

Now looking to the new stuff here, obviously don't forget about the data types for C++, you'll see each member in the structure contains a data type. Then, in the main function when we create the structure, the notation is very simple. The data type of the structure is simply the name of the structure, and then you provide a variable name for the access to use.

You can also define a structure as a pointer instance, however if you do that, we make one change to our standard notation for object access:

```
#include <iostream>
using namespace std;

struct testStruct {
    int num1;
    int num2;
    int num3;
};

void main() {
    testStruct *a = new testStruct();
    a->num1 = 10;
    a->num2 = 20;
    a->num3 = 30;

    cout << "The numbers are: " << a->num1 << ", " << a->num2 << ", " << a-
>num3 << endl;
    delete a;
    cout << "Done!" << endl;
}
```

You'll notice the new arrow operator we're using in the above example, this is called the **object-dereference** operator. I'll talk about this one a lot more in Chapter 16, but just know that if you need to access an item within a pointer to a class or a structure, that you need to use the object-dereference operator instead of the object-access operator. It's essentially the same concept as before with a pointer instance, in order to access the pointer's value itself, you need to dereference the pointer instance. Here, in order to access the structure's contents itself, you need to dereference the structure.

Now let's upgrade our structures by introducing the concept of adding functions to the structures, this will allow us to complete the toolset we need to actually define and implement the linked list standard.

Structure Functions, a Preview of Chapter 16

Our next topic will also share a lot of similarities with the object programming we did on the Torque side of things, so instead of spending a long time talking about it, let's jump right into an example:

```
#include <iostream>
using namespace std;

struct testStruct {
    int num1;
    int num2;
    int num3;

    void setThemUp();
};

void testStruct::setThemUp() {
    num1 = 10;
    num2 = 20;
    num3 = 30;
}

void main() {
    testStruct a;
    a.setThemUp();
    cout << "Numbers: " << a.num1 << ", " << a.num2 << ", " << a.num3 <<
endl;
}
```

Right away you'll notice we have the same namespace operator to separate the class/structure name with the name of the function call. Also, just like in all of our function calls beforehand, you still need to define a return type instance for the function itself. Finally, when you're inside the body of a class function, you have direct access to all of the variables contained within the class definition itself; this will have a much greater impact to your programs in Chapter 16. To actually call functions for objects, again we use the object-access, or object-dereference operator depending on if the object is a pointer or not and then call the function directly. You can also use variables, so don't get concerned about those:

```
#include <iostream>
using namespace std;

struct testStruct {
    int num1;
    int num2;
    int num3;

    void setThemUp();
    void fetch(int *, int *, int *);
};
```

```
};

void testStruct::setThemUp() {
    num1 = 10;
    num2 = 20;
    num3 = 30;
}

void testStruct::fetch(int *a, int *b, int *c) {
    *a = num1;
    *b = num2;
    *c = num3;
}

void main() {
    testStruct *a = new testStruct();
    a->setThemUp();
    int a1;
    int b1;
    int c1;
    a->fetch(&a1, &b1, &c1);
    cout << "Numbers: " << a1 << ", " << b1 << ", " << c1 << endl;
    delete a;
}
```

However, in C++ one of the really useful tools of an object is the **constructor** and **destructor** of the object. A **constructor** is a special function that is called on an object instance when it is created, and a **destructor** is a special function that is called when an object is destroyed. So instead of using a **setThemUp** function, we can just predefined it in the constructor:

```
#include <iostream>
using namespace std;

struct testStruct {
    int num1;
    int num2;
    int num3;

    testStruct();
    ~testStruct();
    void fetch(int *, int *, int *);
};

testStruct::testStruct() : num1(10), num2(20) {
    num3 = 30;
    cout << "I have been created..." << endl;
}

testStruct::~testStruct() {
    cout << "I have been deleted..." << endl;
}

void testStruct::fetch(int *a, int *b, int *c) {
    *a = num1;
    *b = num2;
```

```
*c = num3;  
}  
  
void main() {  
    testStruct *a = new testStruct();  
    int a1;  
    int b1;  
    int c1;  
    a->fetch(&a1, &b1, &c1);  
    cout << "Numbers: " << a1 << ", " << b1 << ", " << c1 << endl;  
    delete a;  
}
```

A fair warning, I did throw a bit of new stuff at you here as well along with the constructor & destructor of the function. The notation of those two functions are very easy, it is type-less in the return type and simply mirrors the name of your object. You'll notice that weird notation in the constructor definition line itself, let's talk about that first:

```
testStruct::testStruct() : num1(10), num2(20) {
```

What I've done here is essentially what is pre-defining the values. If you remember in the prior chapter, we had the ability to do a similar thing with the function arguments, this is essentially how you can mirror that effect with variables inside a class. To show you that you can do either the above, or by including the standard line in the body of the structure, I provided both in the above example. You could also get even fancier here and provide arguments to the constructor to define the starting values as you need them:

```
#include <iostream>  
using namespace std;  
  
struct testStruct {  
    int num1;  
    int num2;  
    int num3;  
  
    testStruct(int, int, int);  
    ~testStruct();  
    void fetch(int *, int *, int *);  
};  
  
testStruct::testStruct(int n1, int n2, int n3) : num1(n1), num2(n2), num3(n3)  
{  
    cout << "I have been created..." << endl;  
}  
  
testStruct::~testStruct() {  
    cout << "I have been deleted..." << endl;  
}  
  
void testStruct::fetch(int *a, int *b, int *c) {  
    *a = num1;  
    *b = num2;  
    *c = num3;
```

```
}
```

```
void main() {
    testStruct *a = new testStruct(10, 20, 30);
    int a1;
    int b1;
    int c1;
    a->fetch(&a1, &b1, &c1);
    cout << "Numbers: " << a1 << ", " << b1 << ", " << c1 << endl;
    delete a;
}
```

When we get to Chapter 16, we'll have some more advanced cases where you may have variable amount of arguments that can be passed and multiple constructors that will be chosen based on those parameters, but for the time being, this is all you'll need for the basic understanding of object programming in C++. Now that you understand this concept, let's get back to the Linked List definition and show you how to use it.

Linked List: Examples

To use a linked list in C++, we're going to make use of the `std::list` class which is part of the Standard Template Library. When you work in Torque however, if you want to use a linked list, make use of the `Torque::List` class.

For our first example, we're going to populate a linked list with 15 integers, and then print them out in sequential order. This is one of the standard applications of a list, to insert elements for future use and then fetch them as needed:

```
#include <iostream>
#include <list>
#include <stdlib.h>
#include <time.h>
using namespace std;

void main() {
    srand(time(0));
    std::list<int> list1;
    int num, i;
    for(i = 0; i < 15; i++) {
        num = rand() % 1000 + 1;
        list1.push_front(num);
    }
    cout << "Inserted " << i << " numbers... reading..." << endl;
    for(std::list<int>::iterator it = list1.begin(); it != list1.end();
it++) {
        cout << "Read: " << *it << endl;
    }
}
```

You'll notice the template syntax returns here, because the Standard Template Library defines the linked list as a template class instance to allow you to use any object or data type in the list instance itself. The `push_front` function inserts an element at the beginning of the list, any existing elements

move back one index in the line, so in reality here, we're actually reading out the numbers in the reverse order they were placed in. To reverse this behavior, replace the `push_front` call with the `push_back` function instead to have the elements be placed at the back of the list as they go into it.

This is a great example to begin with, but how about we use lists for a more advanced task, and combine some of our newfound knowledge to our own personal advantage here. In this next example, I'll create a volume structure, this will define three randomized properties and then declare the volume of it, we'll then use the list to find all volumes above a certain volume, below one, and then add them all together for a total volume:

```
#include <iostream>
#include <list>
#include <stdlib.h>
#include <time.h>
using namespace std;

struct Volume {
    double len, wid, hei;
    char name[32];

    Volume(int index) {
        len = 1.0 + ((double)rand() / RAND_MAX) * (25.0 - 1.0);
        wid = 1.0 + ((double)rand() / RAND_MAX) * (25.0 - 1.0);
        hei = 1.0 + ((double)rand() / RAND_MAX) * (25.0 - 1.0);
        sprintf(name, "Volume%i", index+1);
    }

    double fetchV() { return len * wid * hei; }
    char *fetchN() { return name; }
};

void main() {
    srand(time(0));
    Volume *v;
    std::list<Volume*> list1, list2, list3;
    for(int i = 0; i < 10; i++) {
        v = new Volume(i);
        list1.push_back(v);
        cout << "Created new volume: " << v->fetchN() << ", " << v->fetchV() << endl;
    }
    //Thresholds
    double threshold = 500.0, total = 0.0;
    for(std::list<Volume*>::iterator it = list1.begin(); it != list1.end(); it++) {
        v = *it;
        if(v->fetchV() >= threshold) {
            list2.push_back(v);
        }
        else {
            list3.push_back(v);
        }
        total += v->fetchV();
    }
}
```

```
    cout << "Volumes Above Threshold (" << threshold << ")" :> endl;
    for(std::list<Volume*>::iterator it = list2.begin(); it != list2.end();
it++) {
    v = *it;
    cout << v->fetchN() << " at " << v->fetchV() << endl;
}
cout << "Volumes Below Threshold (" << threshold << ")" :> endl;
for(std::list<Volume*>::iterator it = list3.begin(); it != list3.end();
it++) {
    v = *it;
    cout << v->fetchN() << " at " << v->fetchV() << endl;
}
cout << "Total Volume: " << total << endl;
}
```

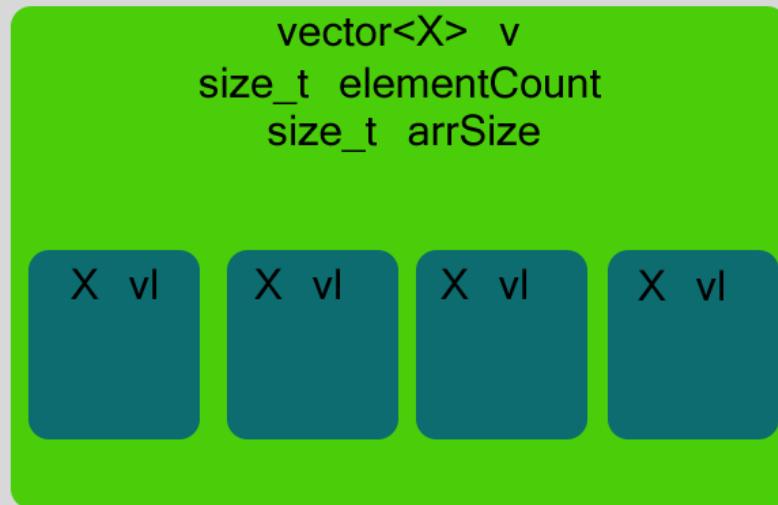
The above example will break you out a bit from the comfort of simplistic programming as we start to introduce some more complex coding examples, since we're dealing with pointers in the above program we also need to ensure to properly dereference them when trying to access the individual elements, which is why I kept **Volume *v** handy during the execution of the program.

The goal here is to teach you a basic understanding of how to establish and iterate through a list format. By the time you get finished with Chapter 16, you should be more than equipped to actually come back here and write your own implementation of a Linked List, it's really not all that difficult and as you saw, there are quite a few uses for the list toolset.

Now let's introduce the topic you've been hearing about a lot in this chapter, and that is the concept of a Vector.

Introduction: Vectors

The other important tool we're going to introduce here is the Vector, which in easier terminology that you'll understand is a dynamically allocated array instance. A Vector allows you to define an array of any type without setting a size limit, and then it expands or shrinks based on what you need and when you need it.



Unlike the linked list, there are no references between the individual elements, and most implementations provide an operator override to treat the class as an array itself. By this definition the Vector allows direct access to the elements contained within the instance. This is the primary advantage of a vector over a list, direct access to elements allow you to quickly retrieve the single element you need from the vector instead of having to iterate through the list. The only main disadvantage is the dynamic sizing of the array, which might sound weird to you, but whenever the instance itself needs to expand in size to accommodate for a growing number of elements, the realloc function is called and as I said earlier, this leads to processing of the computer's available memory to locate the adequate spacing for the increase in size.

You should always weigh the benefits of each tool to the cons of using the tool over the other when picking the application. For about 85% of the applications out there, using a **vector** will be the best option. The remaining 15% deals with extremely large object instances or very large numbers of instances of moderate to large size, where the benefit of using realloc fall short of sacrificing a small bit of extra space for the nodes of the list.

Vectors: Examples

Just like last time, using one of these tools is really easy, you just need to define a container instance and you're off to the races:

```
#include <iostream>
#include <vector>
using namespace std;

void main() {
    std::vector<int> myVec;
    for(int i = 0; i <= 10; i++) {
        myVec.push_back(i * 10);
    }
    int total = 0, itC = 0;
    cout << "Adding up the vector" << endl;
    for(std::vector<int>::iterator it = myVec.begin(); it != myVec.end();
    it++, itC++) {
        total += myVec[itC];
    }
    cout << "The total is: " << total << endl;
}
```

You'll notice a little new switch up in the standard for loop syntax here. We've added what looks like a fourth parameter, but if you look a bit closely at the last parameter, they're separated with a comma and therefore they are treated as the same. Why we did this was to remove the need for a dereference in favor of demonstrating the array notation ability of the vector class.

Just to show you that both the linked list and vector classes are capable of doing the similar tasks, here's the volume example that we made just a little bit ago in vector form:

```
#include <iostream>
#include <vector>
#include <stdlib.h>
```

```
#include <time.h>
using namespace std;

struct Volume {
    double len, wid, hei;
    char name[32];

    Volume(int index) {
        len = 1.0 + ((double)rand() / RAND_MAX) * (25.0 - 1.0);
        wid = 1.0 + ((double)rand() / RAND_MAX) * (25.0 - 1.0);
        hei = 1.0 + ((double)rand() / RAND_MAX) * (25.0 - 1.0);
        sprintf(name, "Volume%i", index+1);
    }

    double fetchV() { return len * wid * hei; }
    char *fetchN() { return name; }
};

void main() {
    srand(time(0));
    Volume *v;
    std::vector<Volume*> list1, list2, list3;
    int arrEle = 0;
    for(int i = 0; i < 10; i++) {
        v = new Volume(i);
        list1.push_back(v);
        cout << "Created new volume: " << v->fetchN() << ", " << v->fetchV() << endl;
    }
    //Thresholds
    double threshold = 500.0, total = 0.0;
    for(std::vector<Volume*>::iterator it = list1.begin(); it != list1.end(); it++, arrEle++) {
        v = list1[arrEle];
        if(v->fetchV() >= threshold) {
            list2.push_back(v);
        }
        else {
            list3.push_back(v);
        }
        total += v->fetchV();
    }
    cout << "Volumes Above Threshold (" << threshold << ") :" << endl;
    for(std::vector<Volume*>::iterator it = list2.begin(); it != list2.end(); it++) {
        v = *it;
        cout << v->fetchN() << " at " << v->fetchV() << endl;
    }
    cout << "Volumes Below Threshold (" << threshold << ") :" << endl;
    for(std::vector<Volume*>::iterator it = list3.begin(); it != list3.end(); it++) {
        v = *it;
        cout << v->fetchN() << " at " << v->fetchV() << endl;
    }
    cout << "Total Volume: " << total << endl;
}
```

If you look closely at this example, I did both the dereference method and the array access method to show you that they both still function exactly the same as they did before.

The key to using both of these tools is to remember the ordering on the vectors and lists is based on the order you push elements to the list or vector, and that accessing elements depends on which of the two tools you are using.

The last thing I want to show you how to do before we leave this section is how to remove elements from the list instances. You can do one of three things when erasing from these two instances. The first is to delete the very first node from the list, this is done using the `pop_front` function. The second is to delete the last node from the list by using the `pop_back` function, and finally you can use the `erase` function and specify an iterator position to delete from. The only thing you need to be aware of is that vector's do not have the `push_front` or `pop_front` functions, but can accomplish these by combining other functions, here is an `erase` example:

```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <time.h>
using namespace std;

void main() {
    //Insert 30 numbers
    std::vector<int> myList;
    std::vector<int>::iterator it;
    for(int i = 0; i < 30; i++) {
        myList.push_back(rand() % 1000 + 1);
    }
    //Delete the first number
    myList.erase(myList.begin());
    //Delete the last number
    myList.pop_back();
    //Delete the 10th number
    it = myList.begin();
    advance(it, 9);
    myList.erase(it);
    cout << "Printing the list: ";
    for(it = myList.begin(); it != myList.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
```

I hope this short segment on lists and vectors has shown you how we can use the memory tools that you now know to create dynamic spaces in our programs to allow for some more advanced functioning, especially when it comes to dealing with object instances, or long lists of values that need to be stored for later use.

If you're up to the challenge, you can try to take the knowledge you've gained from this chapter and mimic the functioning of the linked list using the tools I have taught you to this point. Obviously,

some tools taught later on in the guide will make this job even easier yet, but it's a great C++ learning exercise, and you'll only get better at it by trying new things out.

That about wraps up all of the serious headache material for this chapter, I'll conclude with a few short topics now to give you a few extra tools to work with, and then we'll actually open up the engine in Chapter 15.

Namespaces

Our next topic is a short one on organization techniques when programming in C++. C++ provides a special toolset called **namespaces** to allow you to break segments of code off from the global call stack into separate entities. We've been using this tool the entire time we've been programming in C++ so far, you just haven't really noticed. To this point, all of our programs have had one line in the opening few statements:

```
using namespace std;
```

Now it's time you actually get to learn a bit about this line, and how we can use this in our programs to organize our coding for the better.

Intro to Namespaces, Revisiting the using Keyword

The closest form of work we've had to the namespaces of C++ to this point would be the package tool in TorqueScript. If you recalled from a while back packages were used to declare functions and class methods outside of the global call stack but when invoked would come into place and override any existing methods. A namespace in C++ only brings the functions out of the global call stack.

To define a namespace in C++, all you need to do is use the `namespace` keyword like so:

```
#include <iostream>
using namespace std;

namespace MyNamespace {
    int getNumber() {
        return 27;
    }
}

void main() {
    int num = MyNamespace::getNumber();
    cout << "Number is: " << num << endl;
}
```

What this does is take the function `getNumber` out of the global call stack and moves it into a separate location, or a namespace. This namespace we named **MyNamespace**. In order to get inside the namespace in our actual function call, we make a call to the namespace itself, and then separate it with the **namespace operator** (`::`) before making the actual function call afterwards.

But that's not the only way we can access the contents of a namespace, we have been accessing the **standard template library's** default namespace **STD** for the longest of time by implementing the

using keyword. This has allowed us to strip out unnecessary calls to **std::x** in our program. Without that line, our above program would look like this:

```
#include <iostream>

namespace MyNamespace {
    int getNumber() {
        return 27;
    }
};

void main() {
    int num = MyNamespace::getNumber();
    std::cout << "Number is: " << num << std::endl;
}
```

We've only used the **using** keyword for our own personal benefit to this point, and we'll continue to do that as necessary. Just know and understand that when you make use of the **using** keyword, that you move the specified namespace onto the global call stack for the scope of the file, so for instance if we have two files and one of them has the using keyword, that file will have the contents of the namespace on the global call stack (can be used anywhere in the file). But if the other file does not have it, then that file will not be able to do that.

We can also clean up our example above with the **using** keyword:

```
#include <iostream>
using namespace std;

namespace MyNamespace {
    int getNumber() {
        return 27;
    }
};
using namespace MyNamespace;

void main() {
    int num = getNumber();
    cout << "Number is: " << num << endl;
}
```

Remember, C++ compiles from top to bottom so the namespace MUST be defined prior to using it otherwise it will not exist to the compiler. The above example demonstrates that once you tell the file that you are using a particular namespace, that it becomes accessible to the entire file from that point forward. However, if you place the using statement inside a function call, then it follows the standard rules of scope, and the using statement will fall out of scope at the conclusion of the function.

Multiple-Layered Namespaces

The last thing to talk about is how you can layer namespaces together to create more complex organization structures in your code projects. This can be a great tool to employ if you use it properly, however if you lose track of your namespacing, it can turn into more of a problem than anything else

when you try to access bits of your code locked away in a namespace. The best rule of thumb is to keep functions of the same family together that way access is not an issue. Here's an example of a multi-layer namespace:

```
#include <iostream>
using namespace std;

namespace Tools {
    namespace Math {
        int addEm(int a, int b) {
            return a + b;
        }
    };
    namespace Str {
        const char *getName() {
            return "Math Program";
        }
    };
}

void main() {
    const char *progName = Tools::Str::getName();
    int result = Tools::Math::addEm(10, 20);

    cout << "Program " << progName << " says 10 + 20 is " << result <<
endl;
}
```

Again this is a fairly straight-forward example to play with. Now let's up the complexity a bit. Let's say we want to fetch the function Str::getName() from inside the Math namespace **without** adjusting the locations of existing calls. How would we do this?

```
#include <iostream>
using namespace std;

namespace Tools {
    namespace Str {
        const char *getName();
    };
    namespace Math {
        int addEm(int a, int b) {
            return a + b;
        }
        void doProgram() {
            int res = addEm(10, 20);
            const char *pName = ::Tools::Str::getName();
            cout << "Program " << pName << " says 10 + 20 is " << res
<< endl;
        }
    };
    namespace Str {
        const char *getName() {
            return "Math Program";
        }
    };
}
```

```
};

void main() {
    Tools::Math::doProgram();
}
```

There's only two changes to this program to accomplish that desired behavior. First and foremost, all you need to do is pre-define the function call for `getName` inside the `Str` namespace, nothing new, we've been doing this since Chapter 13. The big second change however comes inside the `doProgram` function we added. You'll notice the namespace operator being placed in a weird location, right at the beginning. This is a special notation in C++ that tells the compiler to back all the way out to the global call stack, from there we can go back into the `Tools` namespace, and into the `Str` namespace to fetch the desired function call.

Finally, you can use the `using` keyword to specify that you only want to access one of the internal layered namespaces. This is a nifty tool that we can use on our first example to cut the `Math` line down a bit:

```
#include <iostream>
using namespace std;

namespace Tools {
    namespace Math {
        int addEm(int a, int b) {
            return a + b;
        }
    };
    namespace Str {
        const char *getName() {
            return "Math Program";
        }
    };
}

void main() {
    using namespace Tools::Math;
    const char *progName = Tools::Str::getName();
    int result = addEm(10, 20);

    cout << "Program " << progName << " says 10 + 20 is " << result <<
endl;
}
```

By using this feature, you can specify to break into only certain parts of a layered namespace without bringing the rest of it onto the global call stack. This can be an extremely helpful tool when you start getting into large libraries, or complex code portions with functions of matching names. [Be Cautious! C++ will allow you to have functions of the EXACT SAME name in separate namespaces, but bringing those functions into the global call stack together can have undesired consequences.](#)

So that covers our topic of namespaces in C++. You should now have all of the tools you need from this section to start writing more organized code blocks in your complex programs. This toolset will

also allow you to write “varying” forms of function definitions and then to bring in the correct function at the correct time in your programs. Now let’s move onto our next topic.

Exception Handling

Our next C++ topic has to deal with something all programs can have issues with from time to time, and that is the very topic of **issues**, or from the technical standpoint, **exceptions**. An exception is a signal to the computer than an unexpected event has happened, and the program needs to shut down to prevent damage from happening to the computer due to the event. Let’s just say without this tool, we’d be dealing with Blue Screens quite frequently, especially in our programs that manipulate memory.

In C++, an exception can occur for a multitude of reasons, one of which can actually be controlled by you, the programmer. Let’s say you have a function that accepts some user input and the input comes back bad, to the point where it may crash the calls to the system. You can manually **throw** an exception to the computer to tell it to close your program immediately, your other option at this point is to test for the exception or **try** to work with a block of code, and then **catch** and handle the exception. With this thought line, we can begin our learning of the **try-catch** blocks.

The throw Keyword

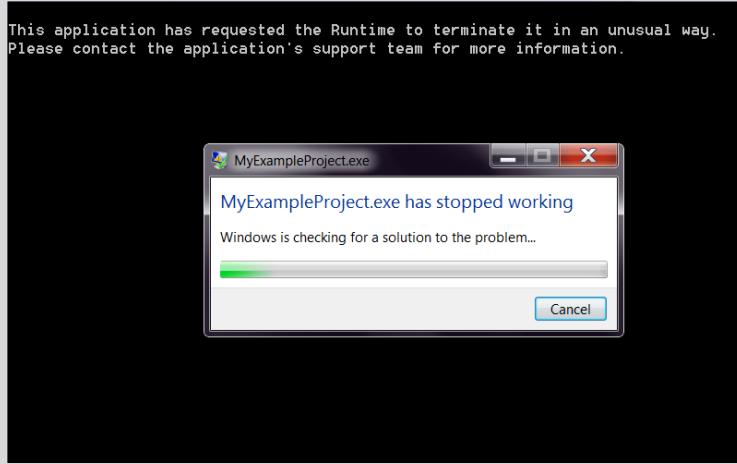
Before we get there however, let’s actually introduce the manual exception throwing, or in short terminology the **throw** keyword. The throw keyword allows you to manually stop a program by throwing an exception to the computer. You can throw any piece of information to the engine, and if the exception is not caught by the code block, the program will exit. Here’s an example of an uncaught statement:

```
#include <iostream>
using namespace std;

int divideIt(int num, int den) {
    if(den == 0) {
        throw "Divide By Zero";
    }
    return num / den;
}

void main() {
    int result = divideIt(10, 0);
    cout << "The division is: " << result << endl;
}
```

When you try to run this program, it will set off the throw statement. Since the statement is not caught the program will be forced to exit and you’ll get an error that looks a bit like this:



In our example program above, we threw a character array, or a character pointer, therefore you will need to catch a character exception in order to block this. You can use the throw statement to throw any piece of valid information, which means you can throw numbers, or even if you're getting more advanced an object instance. I'll show you what I mean about that in just a bit, but let's actually handle the exception now.

Try-Catch Blocks

So we've successfully created a way to forcibly halt the program when it's running. But let's say you actually want the program to continue running, and maybe just print a message to the user saying that a specific operation failed to complete and to keep going. In order to do that, we're going to have to catch the exception, and to do this, you'll need to implement a **try-catch** block.

Doing this is a very easy task, you just create a try statement and place the code that has a chance to fail inside the blocks, and then on the outside of the block, declare one or more catch statements to block a list of the specified instructions. Here's how we'd re-write our above example:

```
#include <iostream>
using namespace std;

int divideIt(int num, int den) {
    if(den == 0) {
        throw "Divide By Zero";
    }
    return num / den;
}

void main() {
    try {
        int result = divideIt(10, 0);
        cout << "The division is: " << result << endl;
    }
    catch(char *exception) {
        cout << "The above operation failed: " << exception << endl;
    }
}
```

You need to be careful when using this tool, because if an exception “slips the net” you’ll still crash. For example, if we throw a numerical code to the same block, the program will crash:

```
#include <iostream>
using namespace std;

int divideIt(int num, int den) {
    if(den == 0) {
        throw -1; //Boom! Crash!
    }
    return num / den;
}

void main() {
    try {
        int result = divideIt(10, 0);
        cout << "The division is: " << result << endl;
    }
    catch(char *exception) {
        cout << "The above operation failed: " << exception << endl;
    }
}
```

If you have a more complex version of a code instance, you can for instance try to catch multiple exceptions:

```
#include <iostream>
using namespace std;

int divideIt(int num, int den) {
    if(den == 0) {
        throw -1;
    }
    return num / den;
}

void main() {
    try {
        int result = divideIt(10, 0);
        cout << "The division is: " << result << endl;
    }
    catch(char *exception) {
        cout << "The above operation failed: " << exception << endl;
    }
    catch(int eCode) {
        cout << "The above operation failed with error code: " << eCode
<< endl;
    }
}
```

When you run the above program this time, it will work perfectly fine and abort with the error code message instead of the operation failed by character pointer message. Just remember that in order to block an exception you need to catch the correct error instance.

If you want to do more tricky things, you can catch a class instance (or a structure). Since you don't know how to use a class yet (Chapter 16), we'll use a struct instead, just know that this method will also work fine with a class instance:

```
#include <iostream>
using namespace std;

struct MyException {
    int errorCode;
    char *message;

    MyException(int a, char *c) : errorCode(a), message(c) { }
    const char *fetch() {
        char *lMsg = new char[256];
        sprintf(lMsg, "The program failed with error code %i (%s)", errorCode, message);
        return lMsg;
    }
};

int divideIt(int num, int den) {
    if(den == 0) {
        throw MyException(0, "Divide By Zero");
    }
    return num / den;
}

void main() {
    try {
        int result = divideIt(10, 0);
        cout << "The division is: " << result << endl;
    }
    catch(MyException e) {
        cout << e.fetch() << endl;
    }
}
```

So in the above example, you'll see that we can create an exception object using the `throw` statement and then access the members inside the object from the `catch` statement. By doing this you can create a global exception class to handle all of your potential errors and then deal with it inside one single `catch` statement instead of trying to worry about multiple errors.

The Standard Template Library

Our last stop in this chapter is a short bit on the **standard template library**. This is a C++ library that has been incorporated with features of the C++ Standard Library; therefore these functions and features are standardized for the language to work on all platforms. We've been using this library so far in all of our programs, all of the functions and features within the `std` namespace (short for Standard) are part of the standard template library.

Since the library itself is massive and well beyond the full scope of our guide here, I'm just going to provide you a list of all of the C++ Standard Template Library and the C++ Standard Library header

files. Whenever you use one of these header files, you can be assured that the contents of that file are global to all platforms and require no compatibility. For a list of the functions and features within each of these headers, I'll leave that to your favorite web browser and search tool.

Name	Header File	Short Description
Basic C++ Utilities		
C-Standard Lib	<cstdlib>, <stdlib.h>	General purpose utilities & functions
Signal Library	<csignal>	Signal tools & macros
Set & Jump Library	<csetjmp>	Macro tools for jumping to execution context.
Standard Arguments	<cstdarg>	Handling of variable length argument lists
Runtime Info	<typeinfo>	Runtime type information utilities & functions
Bitset Class	<bitset>	std::bitset class
Function Objects	<functional>	Object class for working with standard algorithms
Utility Library	<utility>	Functions & utilities to work with the standard library.
Time Classes	<ctime>, <time.h>	Defines standard time classes
Standard Definitions	<cstddef>	Defines type definitions for NULL, size_t, and others
C++ Memory Utilities		
Low Level Memory	<new>	Provides tools and functions for advanced low-level memory operations
High Level Memory	<memory>	Provides tools and functions for advanced high-level memory operations
C++ Numerical Data Limits		
Limits of Integral Data	<climits>	Limits of basic numerical types (int)
Limits of Floating Data	<cfloat>	Limits of basic numerical types (float, double)
Numerical Limits	<limits>	Standardized limits method of <climits> and <cfloat>
C++ Error & Exception Handling		
Exception Utilities	<exception>	Standard C++ exception types & handles
Exception Objects	<stdexcept>	Exception objects & classes
Assertion	<cassert>	Conditional macros to test the condition and assert an error otherwise
Error Number	<cerrno>	Macro definitions containing the latest error number

C++ Character & String Tools		
Character Examination Class	<code><cctype></code>	Functions and utilities to determine the contents of a character
Wide Char Examination	<code><cwctype></code>	Functions and utilities to determine the contents of a wide character
String Handling	<code><cstring></code>	Functions and utilities for string data
Wide Character Tools	<code><cwchar></code>	Functions and utilities for wide character and multi byte types
String	<code><string></code>	Defines std::string and std::basic_string types
Container Classes		
Vector	<code><vector></code>	Defines std::vector class
Deque	<code><deque></code>	Defines std::deque class
List	<code><list></code>	Defines std::list class
Set & Multiset	<code><set></code>	Defines std::set and std::multiset classes
Map & Multimap	<code><map></code>	Defines std::map and std::multimap classes
Stack	<code><stack></code>	Defines std::stack class
Queue & Priority Queue	<code><queue></code>	Defines std::queue and std::priority_queue classes
Container Functions	<code><algorithm></code>	Algorithms & utilities that operate on the container classes
Iteration Tools	<code><iterator></code>	Defines the iteration class object for container classes
C++ Numerical Tools & Functions		
Common Math Lib	<code><cmath></code>	Defines basic mathematical functions
Complex Numbers	<code><complex></code>	Defines the complex number type
Value Array	<code><valarray></code>	Defines a class for representing arrays of multiple values
Numerical Operations For Containers	<code><numeric></code>	Numeric operations for values that are stored in container classes
C++ Tools for Input & Output		
Pre-Define IOS Classes	<code><iosfwd></code>	Forward declarations for all IO classes (You shouldn't need this ever)
IOS Base	<code><ios></code>	Define the base class for all input and output operations
Input Stream	<code><istream></code>	Define the std::istream base class for all input operations

Output Stream	<code><ostream></code>	Defines the std::ostream base class for all output operations
Input/Output Stream	<code><iostream></code>	Defines a multi-function class to handle input & output operations
File Stream	<code><fstream></code>	Defines the std::fstream class to handle file manipulation
String Streams	<code><sstream></code>	Defines the numerous string stream classes for advanced string type manipulations
IO Formatting	<code><iomanip></code>	Tools and functions for manipulating input and output of the stream classes
Stream Buffer	<code><streambuf></code>	Defines the stream buffer class
C-Style Input/Output	<code><cstdio></code>	Defines C-Style classes used for input and output
Miscellaneous Tools & Operations		
Localization	<code><locale></code>	Localization utilities & functions
C-Style Localization	<code><clocale></code>	Defines C-Style classes used for localization functioning.

This list only contains the relevant headers to this point, C++11 added quite a few more to this list, but for the most parts you won't be touching those in your work inside the engine. For a more detailed list and header information for the standard template library, I'll direct you to the following web resource: <http://en.cppreference.com/w/cpp/header>.

Final Remarks

I hope your headache level isn't too high after this chapter. It's a very important chapter filled with a ton of important C++ concepts that we'll be using from this point forward so if there was anything that didn't really make sense, please be sure to go back and look it over again, or go online and look up further examples of what was provided here.

Now that you have pretty much the core concepts of C++ understood with the exception of a few things which we'll be getting to in the next few chapters, you are now ready to open up the engine itself and to begin working under the hood to implement new features and mechanics to set your game apart from the others made in this engine.

Chapter 15: Opening Up the Engine, Combining C++ and TorqueScript

Chapter Introduction

With that out of the way, we can now step out of the world of console windows (for a bit) and jump into Torque itself. This chapter will introduce a few more C++ things to you, but in a way that ties in with the overall theme of this chapter, which is to open the hood of the engine we call Torque 3D and take you on your first steps of the guided tour of the inside. This chapter will teach you two brand new C++ concepts and then we'll move onto our first steps in bringing together what we've learned in C++, with what we accomplished a while back in the TorqueScript chapters. Finally, I'll take you through the processing structure of the engine showing you how to properly use memory in your segments you add, and how the engine launches (Entry Points).

It sounds like quite a bit, but this will be a fairly short chapter there's a lot to talk about, but the information you gain from here will be extremely valuable, and not all that complicated, so let's get going!

C++ Data Type Trickery (Typedefs)

We'll start by learning a new concept that the engine uses throughout. Torque is a massive project and there are loads of files to work with. If you've been following along with C++ since Chapter 13, go ahead and close the example project you made and then open up the Torque project. This is the same project that you built back in Chapter 2, so open up the .sln file for that. Navigate in the solution explorer to the DLL project, and then open the source file **platform/types.h**. We'll be using this file to teach two new concepts before we move forward.

The first part of this file contains a block comment with the license of the engine (MIT) and then a cryptic looking block that looks like this:

```
#ifndef _TORQUE_TYPES_H_
#define _TORQUE_TYPES_H_
```

We'll come back to this block definition format in Chapter 16. The topic for this section deals with the majority of this file, which is called **type definitions**. If you scroll down you'll see the following lines:

```
typedef signed char           S8;      ///< Compiler independent Signed Char
typedef unsigned char         U8;      ///< Compiler independent Unsigned Char

typedef signed short          S16;     ///< Compiler independent Signed 16-bit
short
typedef unsigned short         U16;     ///< Compiler independent Unsigned 16-bit
short

typedef signed int             S32;     ///< Compiler independent Signed 32-bit
integer
typedef unsigned int           U32;     ///< Compiler independent Unsigned 32-bit
integer

typedef float                 F32;     ///< Compiler independent 32-bit float
typedef double                F64;     ///< Compiler independent 64-bit float
```

A type definition or a **typedef** for short is the practice of renaming a data type to another in C++ for quick and easy access. To make more sense of things, the following block turns the data type **char** into **S8** in the engine for instance, so instead of needing to type **char** every time, you can substitute **S8** in the place of it. This engine is constructed in a way that these type definitions will be pulled into all of the engine's files. When you go about adding your own files to the engine, you'll need to incorporate this file as well to pull in most of the features of the engine. Before we move forward, I want to show you one thing outside of Torque for this section, and that is how you can use a **typedef** to cut down on a large block of unnecessary code, for instance let's go back to our iterator example from the previous chapter and make a change:

```
#include <iostream>
#include <list>
#include <stdlib.h>
#include <time.h>
using namespace std;

struct Volume {
    double len, wid, hei;
    char name[32];

    Volume(int index) {
        len = 1.0 + ((double)rand() / RAND_MAX) * (25.0 - 1.0);
        wid = 1.0 + ((double)rand() / RAND_MAX) * (25.0 - 1.0);
        hei = 1.0 + ((double)rand() / RAND_MAX) * (25.0 - 1.0);
        sprintf(name, "Volume%i", index+1);
    }

    double fetchV() { return len * wid * hei; }
    char *fetchN() { return name; }
};

void main() {
    typedef std::list<Volume *> vList;
    typedef vList::iterator vIt;

    srand(time(0));
    Volume *v;
    vList list1, list2, list3;
    for(int i = 0; i < 10; i++) {
        v = new Volume(i);
        list1.push_back(v);
        cout << "Created new volume: " << v->fetchN() << ", " << v->fetchV() << endl;
    }
    //Thresholds
    double threshold = 500.0, total = 0.0;
    for(vIt it = list1.begin(); it != list1.end(); it++) {
        v = *it;
        if(v->fetchV() >= threshold) {
            list2.push_back(v);
        }
        else {
            list3.push_back(v);
        }
    }
}
```

```

        total += v->fetchV();
    }
    cout << "Volumes Above Threshold (" << threshold << ")" :>< endl;
for(vIt it = list2.begin(); it != list2.end(); it++) {
    v = *it;
    cout << v->fetchN() << " at " << v->fetchV() << endl;
}
cout << "Volumes Below Threshold (" << threshold << ")" :>< endl;
for(vIt it = list3.begin(); it != list3.end(); it++) {
    v = *it;
    cout << v->fetchN() << " at " << v->fetchV() << endl;
}
cout << "Total Volume: " << total << endl;
}

```

You'll see in the above that we type defined the repetitive iterator as well as the definition of the list itself, then when we needed to use it we could use our shortcut name instead of needing to re-type the entire line. Like always, just remember your scope rules when it comes to using this tool as it will only be legal within the scope of the definition line.

C++ Macros & Conditional Inclusions

The second tool you need to learn about is the **macro** tool. In C++ a macro is a pre-defined **inline** function that can be either run at your call, or be used to define a parameter to the global scope. To do this, you use the **preprocessor directive #define**. This is the second time we've used a preprocessor directive now. Our first instance was the **#include** statement which you learned is used to load in an external header file to the global scope.

Just remember, that as a preprocessor directive, it occupies a single line of code, and is **NOT** terminated with a semicolon, adding a semicolon here is actually a syntax-error to C++, so don't do it. I'll show you how a macro can occupy multiple lines in just a bit, but let's start with some examples:

```

#include <iostream>
using namespace std;

#define ARRAY_SIZE 10

void main() {
    int nums[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for(int i = 0; i < ARRAY_SIZE; i++) {
        cout << "Number " << i << ":" << nums[i] << endl;
    }
}

```

The above example should explain what I mean in terms of the one-line directive statements. We assign a global parameter to the value of 10 and then have access to it throughout the remainder of the file from the point the definition is made. You can define everything from values, to strings, even micro function definitions by using the macro tool. We'll get to those in just a moment, but before we continue let's talk about how to clear out a definition.

Sometimes in a program you'll reach a point where you no longer want a specific macro to be valid, or you need to clear out the space for potentially a replacement with another macro definition. To do this, you can use the **#undef** preprocessor directive.

```
#include <iostream>
using namespace std;

#define ARRAY_SIZE 10

void main() {
    int nums[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    #undef ARRAY_SIZE //<- ARRAY_SIZE no longer valid.
    int size = 10;
    for(int i = 0; i < size; i++) {
        cout << "Number " << i << ":" << nums[i] << endl;
    }
}
```

This program will un-define our macro right after the array is created, so in order to use the size of the array in the loop, we need to either directly paste it into the code, or simply define a new variable to take its place. So now that we have some basic macro definitions down, let's replace a basic function with a macro. Remember, a macro function is essentially the exact same thing as an **inline** function, so the best place to use macro functions is for small and compact quick hitting functions, you don't want to overload the program with function definitions that are entirely macro based, at that point you should just be sticking with the **inline** keyword.

```
#include <iostream>
using namespace std;

#define maxima(a, b) a > b ? a : b

void main() {
    int one = 1;
    int two = 2;
    int max = maxima(one, two);
    cout << "The maxima of " << one << " and " << two << " is: " << max <<
endl;
}
```

So the above example contains the basic setup for a macro function. Right away you should notice a potential issue when using a macro function, and that is the variables and return types are typeless, so essentially you could very quickly run into some unintended behaviors if the user for example types something the compiler doesn't like.

Before we move onto the next step, there is one thing to clarify for you. When you write a "definition", that is to say a macro function for instance, all the compiler is doing when the macro call is made is essentially copying and pasting the contents of the definition block into the specified area, so for instance the compiler sees what we just did as the following image:

```
#include <iostream>
using namespace std;

#define maxima(a, b) a > b ? a : b

void main() {
    int one = 1;
    int two = 2;
    int max = maxima(one, two); → int max = (one > two ? one : two);
    cout << "The maxima of " << one << " and " << two << " is: " << max <<
endl;
}
```



The contents of the macro are placed in parenthesis to show the point here. So essentially you're making a shortcut call again, just like the **typedef** we made earlier, but with the change of allowing variable definitions as well. By this very logic, you could legally include semi-colons in this form of a define if you were for instance writing a variable defined function instance that needs semicolons.

With that in mind, let's actually use a macro to define a function instance to work with in C++, I'll also show you a few little tricks with this next example:

```
#include <iostream>
using namespace std;

#define DefineTypedMax(type, name) \
    type GetMax##name(type a, type b) { \
        if(a >= b) { \
            return a; \
        } \
        return b; \
    }

DefineTypedMax(int, Numbers);

void main() {
    int a = 1, b = 2;
    cout << "Max is: " << GetMaxNumbers(a, b) << endl;
}
```

The first glance of this may cause a bit of headaches, but work through it one piece at a time and you'll see that it's not too terrible. The macro itself is calling for two arguments, a data type and a name for the function instance. **The single character (\) is used to concatenate a line to the macro, allowing you to use multiple lines for one macro definition. The final line of the macro does not contain this character telling the compiler to end the definition there.** So work it out one line at a time and you'll see how this works.

The first line of the macro after the define is the function definition line, imagine replacing the word **type** with **int** and **name** with **Numbers**. You'll see what C++ will see when you do that:

```
int GetMax##Numbers(int a, int b) {
```

The symbol (##) when used in a macro concatenates two string instances together, so the text reads GetMaxNumbers after that command is parsed. From that point forward, the rest is just a standard function definition. The nice thing about tools like this, is you can then proceed to define multiple instances of the function by only using the macro, so for instance you could then define one for characters, doubles, floats, and so forth all with one macro.

Conditional Inclusions

The next little trick you're going to learn about has some importance now, and even more so in the coming chapters. So far, you've learned two preprocessor directives, now I'm going to add a few more to your collection. These are called the **conditional inclusion** statements. They are used to run conditional checks on macro definitions on the global scope level.

These are really easy to pick up and use because they follow a near identical setup to your standard conditional statements. For example, a basic if structure appears like so:

```
#include <iostream>
using namespace std;

#define USE_MY_CODE
#ifndef USE_MY_CODE
    int GetMaxNumbers(int a, int b) {
        return a > b ? a : b;
    }
#else
    #error "Cannot compile without code"
#endif

void main() {
    int a = 1, b = 2;
    cout << "Max is: " << GetMaxNumbers(a, b) << endl;
}
```

So first of all a new little thing for you to see about macros is that even if you don't specify a parameter it still flags it to the compiler as defined, this is a useful shortcut to know, especially in the next Chapter. The first new preprocessor directive to learn about in the above is the **#ifdef** directive which is short-hand for "if defined". The next preprocessor we have is the **#else** command which simply enough is used to end an if statement and move to the else block. The **#endif** preprocessor is used to close out a conditional inclusion block, **even if you only are using the if statement, you still need to use the #endif preprocessor.** For a quick reference the **#error** preprocessor is used to send an error to the compiler preventing the code from successfully building, use this if a flag setup is improperly defined. But let's say that now we want to have an **else if** clause. In this case we need to manipulate our first block a bit to achieve this behavior:

```
#include <iostream>
using namespace std;

#define USE_MY_CODE
#define REVERSE
#if defined(USE_MY_CODE) && !defined(REVERSE)
```

```
int GetMaxNumbers(int a, int b) {
    return a > b ? a : b;
}
#elif defined(USE_MY_CODE) && defined(REVERSE)
int GetMaxNumbers(int a, int b) {
    return a > b ? b : a;
}
#else
#error "Cannot compile without code"
#endif

void main() {
    int a = 1, b = 2;
    cout << "Max is: " << GetMaxNumbers(a, b) << endl;
}
```

This time the print statement will print the incorrect answer, but you'll notice that is the intended behavior. What happens this time around is we have multiple definition tests to use so instead of using the **#ifdef** preprocessor we use the standard **#if** preprocessor and combine it with the C++ function **defined**. To use an “else-if” clause you use the **#elif** preprocessor, from that point forward, everything else is the same. You could just as easily nest two **#ifdef** statements together to achieve the same functioning like so:

```
#include <iostream>
using namespace std;

#define USE_MY_CODE
#define REVERSE
#ifndef USE_MY_CODE
#define REVERSE
    int GetMaxNumbers(int a, int b) {
        return a > b ? b : a;
    }
#else
    int GetMaxNumbers(int a, int b) {
        return a > b ? a : b;
    }
#endif
#else
#error "Cannot compile without code"
#endif

void main() {
    int a = 1, b = 2;
    cout << "Max is: " << GetMaxNumbers(a, b) << endl;
}
```

So I hope this basic introduction of macro definitions and conditional inclusion statements will help you along through some more advanced C++ functioning. As I stated, when we get a little deeper into Chapters 16 and 17 we'll come back to these tools to handle our class definitions and bringing them into the global scope of the program.

How to add your own C++ Files to Torque

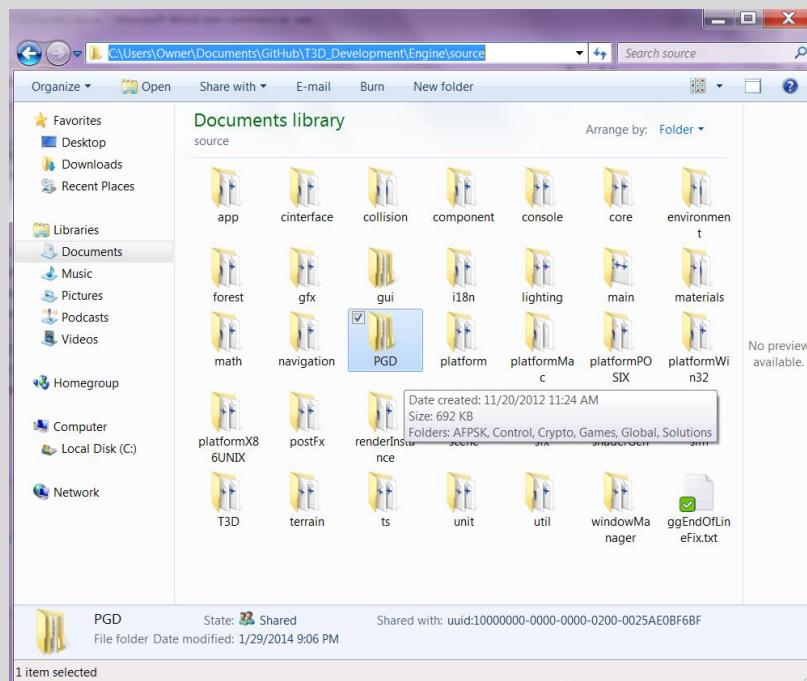
Our next stop is your first step into making changes to the Torque 3D engine itself. We're going to get you ready to start creating your own solutions to implement into the engine. Before we just jump right into this task however, there's a few things you need to ask yourself:

1. What do you need this file to accomplish?
2. What kind of functions, classes, etc. will be defined by this file?
3. What headers / libraries do I need to access to use this file?

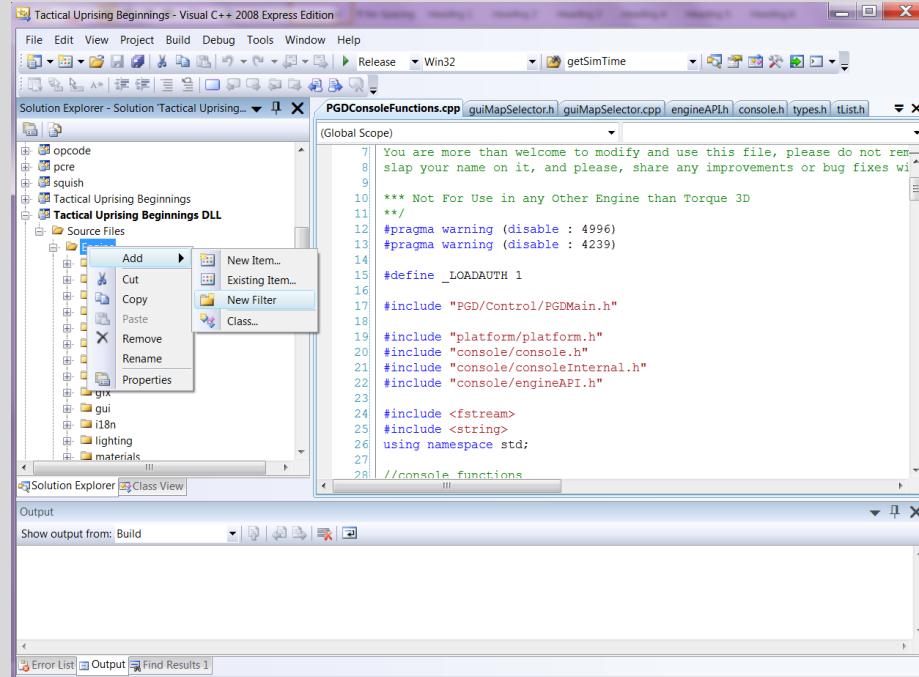
To actually work with this section, we'll prepare to add a file that has the sole purpose of adding custom TorqueScript functions to the engine which is our ultimate goal in this chapter. In the future however, when you actually go to make custom engine files, you'll want to pull in necessary headers and other files as needed by the end goal. The best route of action here is to find a file that does the most similar job as what you intend to do, and start with those headers as the functions needed will likely be the same. From there you can work in additional header files as necessary.

As I said earlier in this chapter, most if not all of the files in the engine in some way, shape or form access **platform.h** at some point. This is mainly used to bring in the individual data types and define platform specific engine functioning you'll need. I'll talk a lot more about the **#include** preprocessor in the next chapter, but you'll need it here in order to use the functioning of the engine.

Let's start by creating our new file. I usually work by creating a new folder for my individual files to store them in. This is a two part process. You'll want to navigate to your engine's files itself in Windows Explorer or whatever form of file exploring solution you use, and create a new folder in the source folder:



Once you create the folder, then re-open your engine solution in Visual Studio, and do the same on that end, ensuring the folder name is exactly the same (In case you forgot how, right click the **Engine** folder in the .DLL project and select the Add->New Filter option):



Once you have the folder installed in the engine and the file system, you can start adding files to the folder in the project and it will be placed in the proper location (in both places). Remember, when C++ compiles your code it loads in all of the C++ files from top to bottom, so header files are loaded in as well. The way Visual Studio handles this is it compiles the projects in the specified order (This makes sure that all of the external functions are brought in first), and then it compiles the individual engine files. Any files you add to the engine will be the last to be compiled as they are read in as external to the core.

So let's begin. Go ahead and right click the newly added folder and add a new C++ file. Name it whatever you want. For the purposes of the guide, mine is named MyNewEngineFile.cpp. Since we're going to be adding new TorqueScript functions we need to pull in the necessary header files used by the engine to handle the type definitions as well as the definition of TS functions:

```
/*
    MyNewEngineFile.cpp
    Custom Functions
*/
#include "platform/platform.h"
#include "console/console.h"
#include "console/consoleInternal.h"
#include "console/engineAPI.h"
```

Once you have these header file in place, you're all set to start programming for the Engine itself!

Creating new TorqueScript Functions in C++

So now we're ready to start writing custom functions for the engine itself. To actually create custom functions, we make use of a macro definition that Torque 3D defines in the `engineAPI.h` header file. Also, like I said earlier Torque also makes use of the modified type definitions for the data types. In Torque 3D you can use the following shortcuts:

C++ Data Type	Torque Shortcut
<code>char</code>	S8
<code>unsigned char</code>	U8
<code>short</code>	S16
<code>unsigned short</code>	U16
<code>int</code>	S32
<code>unsigned int</code>	U32
<code>float</code>	F32
<code>double</code>	F64
<code>char</code>	UTF8
<code>unsigned short</code>	UTF16
<code>unsigned int</code>	UTF32

So feel free to substitute away for the following type definitions. You still need to include additional modifiers when using these types like const and the pointers (since they don't typedef pointer types).

Now to actually define a function in the engine we need to use the `DefineEngineFunction` macro definition. This is the most complex macro definition you'll learn to date so instead of starting with an example, we'll start with the syntax:

```
DefineEngineFunction(FunctionName, ReturnType, (Args),  
(Predefined_Backwards_Args), StrDoc) {  
    //Body  
}
```

There are a few rules to the defined functions. First and foremost no two functions can share the same name (should be obvious by now). The return type for the function **MUST** be a standard type (int, bool, float, double, const char *, char). The argument list and the predefined arguments **MUST** be enclosed in parenthesis, also be cautious to note that the predefined arguments are reversed, so for instance if you have 3 arguments, the first predefined argument will fill in argument #3. If you don't want to predefine arguments, you can skip the parameter entirely. Finally we have a string documentation parameter where you can explain the function and its usage to the coder. Here's a look again at the definition without the predefined case:

```
DefineEngineFunction(FunctionName, ReturnType, (Args),, StrDoc) {  
    //Body  
}
```

Now that we have the basic format in syntax down, let's actually write a basic function. This will be a repeat of our very first TorqueScript function back in Chapter 6, but converted to C++ form:

```
DefineEngineFunction(helloWorld, void,,, "Prints out the Hello World!") {
    Con::printf("Hello World!");
}
```

Yep, there it is again! Our very first function we wrote in TorqueScript, but now in C++ form. So what exactly are we doing in this function? Well first and foremost, you'll notice the void return type so we don't need to return any values here. There are no arguments or predefined parameters so both of those are left empty. The line inside the code block is effectively the exact same **echo** function we used way back in Chapter 6. In fact, here's that definition for your liking:

```
DefineEngineFunction(echoClone, void, (const char *str, const char *fmt),
(""), "We just rewrote echo, because reasons...") {
    Con::printf(str, fmt);
}
```

So what exactly is **Con::printf**? Well for starters you should be able to notice the namespace operator right away and pick apart that **Con** is a namespace, in fact that's the console namespace where all of the important tools related to the Console and the TorqueScript language are stored. The printf function is an exact replica of the C++ standard printf function, but altered to be used inside of the Torque console instead of the C++ console window. I could go into variadic functions but that would be a little beyond the scope of the guide. For a reference that you can look into at your own time, feel free to read up (<http://en.cppreference.com/w/cpp/utility/variadic>), essentially it's a function that can have varying number of arguments.

For the reference of this guide, just know that when you need to print something to the Torque console window in C++ use the **Con::printf** function. So, as of now we've just done some very basic stuff so let's step up the difficulty and functioning a bit to do some more cool things.

Returning Data

Our next step is to create functions that provide results to TorqueScript. Most of the functions we used, be it mathematical or hand written returned a result to the user so they could use it in additional functions. The internal functions can do the same thing, except the difference is the result is calculated in the much faster world of C++ before sent to the user in TorqueScript. Writing a function in this form is just as easy as before, we just need to alter the macro line to tell the system that it should expect a value:

```
DefineEngineFunction(maxima, S32, (S32 a, S32 b),, "[int, int] Returns the
max number") {
    S32 maxima = a > b ? a : b;
    return maxima;
}
```

In the above code, you'll see that we are now using the S32 return type which tells the macro to expect an integer result. We then accept two numerical parameters to the function and run our same check as before. This function however isn't "safe" because if the user forgets to type a value, it will crash the engine. To prevent this, we redefine both of the values in this example function:

```
DefineEngineFunction(maxima, S32, (S32 a, S32 b), (0, 0), "[int, int] Returns  
the max number") {  
    S32 maxima = a > b ? a : b;  
    return maxima;  
}
```

To continue along with this current line of thought, let's imagine now that there are potentially "fatal" values that could be sent. Now in C++, you would then just proceed to use the standard **try-catch** block to handle this problem, but in TorqueScript if you define a function to return a value, then it **MUST** return a value. To handle this, the best course of action is to return a constant value that can then be caught by the TorqueScript end to flag the error:

```
DefineEngineFunction(divideIt, S32, (S32 a, S32 b), (1, 1), "[int, int]  
Perform Division") {  
    if(b == 0) {  
        Con::errorf("divideIt(): Cannot perform division by zero,  
returning 0.");  
        return 0;  
    }  
    return a / b;  
}
```

You can see in the above case that if the user tries to divide by zero, that will not be a good thing to do, so instead we just return the number zero and run the **errorf** command (Same as printf, red text) to alert the user that an error occurred. Just remember the rule that TorqueScript functions defined in C++ have to return standard C++ data types and you'll be all set.

Calling Pre-Written Functions

This is nice and all, but to this point we haven't had one case where the function code was written outside the macro block. And don't be concerned, this is a perfectly legal thing to write in C++, you can have your function body outside the macro, in fact in Chapter 17 we'll dedicate pretty much the entire chapter worth of function writing to this behavior. All you need to follow is the standard rule of global scope for C++, where as long as the function is defined prior to the macro, you can safely use it. Here's an example using some code we wrote a while back:

```
namespace Tools {  
    namespace Math {  
        int addEm(int a, int b) {  
            return a + b;  
        }  
    };  
    namespace Str {  
        const char *getName() {  
            return "Math Program";  
        }  
    };  
};  
  
const char *fetchMathResult(S32 a, S32 b) {  
    using namespace Tools::Math;  
    const char *progName = Tools::Str::getName();
```

```
S32 result = addEm(a, b);

UTF8 *str = new UTF8[256];
dSprintf(str, 256, "%s says %i+%i is: %i", progName, a, b, result);
return str;
}

DefineEngineFunction(fetchMathResult, const char *, (S32 a, S32 b), (1, 1),
"[int, int] External function example") {
    const char *res = fetchMathResult(a, b);
    Con::printf("Result: %s", res);
    return res;
}
```

So easy enough, it's perfectly safe to write a function outside of the macro call and use it inside of the macro call, just make sure the function exists before the macro and you'll be good to go. You'll notice a new-ish looking definition of the C++ **sprintf** function above. The **dSprintf** function is the Torque internal definition of this function. Torque redefines these C++ functions to be cross-platform in nature and the functions that require this have the letter **d** added to the beginning and the second letter capitalized. A full list of these functions is available in the **core/string/stringFunctions.h** file. But as you can see, the external definition example is a great tool when you want to for instance, use an external library in the engine and have some of the library functions be available to TorqueScript.

Calling a TorqueScript Function in C++

So now let's imagine for a moment that you'd like to go the other way around, where you have a function that is defined in TorqueScript that needs to be called by another written in C++. This is also possible. You make use of the **evaluatef** function included in the console namespace:

```
/*
Assume we have the following function:
    function doTSMath(%a, %b, %c) {
        return %a + %b + %c;
    }
*/
DefineEngineFunction(CallTSMath, S32, (S32 a, S32 b, S32 c), (0, 0, 0),
"Calls a function inside the engine") {
    const char *result = Con::evaluatef("doTSMath(%i, %i, %i)", a, b, c);
    S32 intResult = dAtoi(result);
    return intResult;
}
```

You don't need to include the semicolon at the end of the string since the engine will automatically add it if it believes it needs to. The formatting here is exactly the same as a printf/sprintf function call so you can have as many variables as you need. Lastly, you caught a glimpse of **atoi** a little while back, **dAtoi** is the Torque definition of the function and is used to convert string data integers to standard C++ integers, just like the definition of **dSprintf** above that we talked about.

Just remember that the error checking rule works both ways around, so for instance if your TS function can throw an invalid value check, then you need to handle it on the C++ end as well. Let's provide an example of that using our division function rewritten in TS for C++ now:

```
/*
Assume we have the following function:
    function doDivisionTS(%a, %b) {
        if(%b == 0) {
            error("Cannot divide by zero.");
            return 0;
        }
        return %a / %b;
    }
*/
DefineEngineFunction(CallTSMath, S32, (S32 a, S32 b), (1, 1), "Calls a
function inside the engine") {
    const char *result = Con::evaluatef("doDivisionTS(%i, %i)", a, b);
    S32 intResult = dAtoi(result);
    if(intResult == 0) {
        Con::errorf("CallTSMath(): Divide by zero error caught on C++,
forwarding 0");
        return 0;
    }
    return intResult;
}
```

While either of the two error checks in this case would work fine, it's usually a good habit to get into the nature of catching errors on both ends of the spectrum to prevent crashing. The last thing to remember with this section is to be mindful of the return types in the functions you're calling in the Console namespace. Some will return **const char *** values, others will return different types. It's important to get into the habit of reading up on the functions you're calling and seeing the data type that it's expecting back, this will make you a better programmer in the long run.

Pointer Variables in C++ Defined TorqueScript Functions

Our next stop is to bring pointers back into the equation. To this point all we've done is messed around with the standard data types and some of our old examples, but we really haven't done anything too advanced as it pertains to C++. One of the key things you may get to work with at some point is a function that needs to define a block of memory while inside a TorqueScript function. The issue here lies once again with the scope problem, only this time you need to remember one little thing about TorqueScript. Functions can be indirect, unlike in C++ where a result is required at the time of execution. This can lead to some memory related problems when using the **new** keyword or the **malloc** function.

To get around this issue, we use what is called the **return buffer** in the engine. The return buffer is a stack of memory that exists in the engine's console module and is effectively the large amount of space that the console holds during the calculations and method calls that are being used. This stack in that same nature is a **dynamically allocated** space of memory that grows and compacts based on the amount of storage that is needed by the engine at the time of the operation. Since it exists within the engine's console itself, it is safe to be used through indirect methods of TorqueScript.

For instance, let's now go back to some of our pointer related examples that required the malloc and new keywords and provide the correct replacements as needed by the engine to be used internally:

```
DefineEngineFunction(reprintName, const char *, (const char *name),, "Prints
out the name created in local memory") {
    char *buffer = Con::getReturnBuffer(dStrlen(name) + 1);
    dMemcpy(buffer, name);
    Con::printf("The name is: %s", buffer);
    return buffer;
}
```

This function takes the length of the string (plus one for the null terminator) and allocates a space on the return buffer for the result of the function call. This is effectively the same operation as your **malloc** or **new** call from the previous chapter. Once the memory block is created, we copy the string into it by means of the standard C++ operation **strcpy**. Finally we can print it out and safely return it to the user. By using the return buffer, you can ensure that your pointers never fall out of scope leaving your program with an access violation issue. Whenever you plan on writing TorqueScript functions that allocate pointer variables within the body of the function, you should use the return buffer.

Now, there's one point you need to be careful with and that is the fact that the return buffer exists as a **char *** variable type. This means that if you want to store other data types in the buffer, you need to first convert them to a **char *** type. Let's look at a repeat example from one of our integer pointer functions in the previous chapter now:

```
DefineEngineFunction(reprintNumber, int, (const char *name),, "Prints out the
number created in local memory") {
    int *value = new int(1207);
    String numStr = String::ToString("%i", *value);
    char *buffer = Con::getReturnBuffer(numStr.length() + 1);
    dMemcpy(buffer, numStr.c_str());
    Con::printf("The Number is: %i", dAtoi(buffer));
    return dAtoi(buffer);
}
```

You'll remember a while back that we created a function that fetched an integer pointer and returned it to the main method. Now we're doing the same thing here, but de-pointerizing the value after its usage and sending the result to the user. To properly store the number in the buffer, we make use of the Torque **String** class to convert the number to a dynamically expanding string, which can then be safely loaded onto the return buffer by converting the string to a **const char *** through the **c_str** method on the String class. Once we've converted the result to a recognizable format, we run it through the string copy method again to load it onto the buffer, and then we can freely use it.

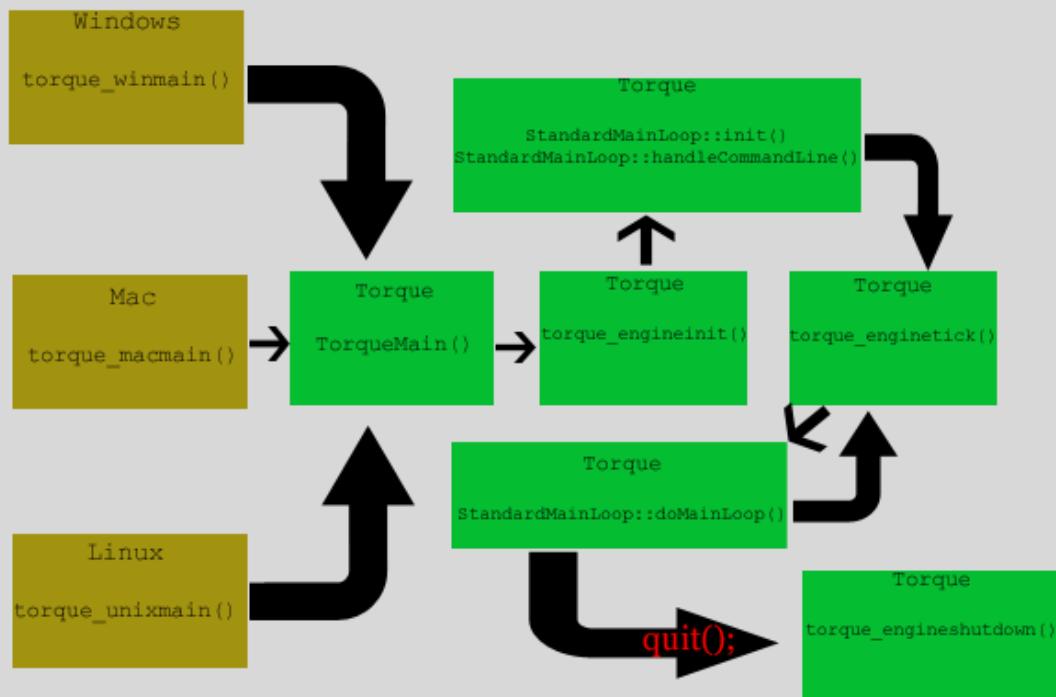
The important rule and repeating concept of this chapter is to nail down the concept of global scope and local scope. As long as you understand where your variables are “legal” and where they are no longer defined, you should be in great shape to handle more and more complex data structures as you proceed through these final few chapters.

In Torque, as long as you stick with the return buffer when using pointers in your defined functions, you shouldn't have any issues with pointers falling out of scope. We'll come back to this topic a bit more in the next two chapters when we start working with object functions, but for the time being, just know and understand how to use the return buffer.

Entry Points

Our final stop of this chapter is a short discussion on the loading sequence of the Torque 3D game engine. This topic will hold some merit for you in the later chapters if you plan on loading in external libraries into the engine that require a persistent data structure (object).

To this point in time, we've worked with one single form of entry point in C++, which is the **main** function definition. This is true whenever you're working with the **console application** style of program. However, it's a standard practice to run the main method even outside of this style of application. Torque is no different, the only key change here is that the main method doesn't open the console window, instead it forks the call to the platform specific initialization, which handles the cross-platform (the very few platforms) code before coming back to the **MainLoop** of the engine. For the purposes of this guide, the platform related initialization does not hold importance to what we're working with. If you're seeking more advanced programming challenges, then by all means dig deeper into the code base and into the individual platform folders to see how the engine kicks everything off. To help you understand this loading sequence here's a diagram of the flow of the engine:



Each of the platforms defines a main method. The engine through platform flagging picks out the main method for the correct platform and then runs it. The main method's only purpose for the individual platforms is to parse the command line parameters (since each platform handles it

differently). These then are sent to the global TorqueMain method which kicks off the engine startup sequence by sending the command line parameters to **torque_engineinit()**. This function is responsible for starting up the engine's main loop and actually parsing the command line parameters. Once the main loop completes, the engine moves onto the actual engine loop sequence which runs the main loop over and over again until the user sends the quit command to the engine which shuts it down.

For the purpose of the guide, the entry point that we are concerned about is the **StandardMainLoop::init()** function. You'll find this in the [app/mainLoop.cpp](#) file. When you scroll down to the method above, you'll see the order of initialization for each of the individual modules in the engine. Whenever you decide to start writing your own module for instance, you should keep in mind what your module needs in order to run when placing your code in the list.

Final Remarks

And that's it! See this chapter wasn't all too hard. You've learned how to manipulate data types in C++, how to define macros and conditional inclusions (Which will be very important in the next chapter), and then we went out of our way to learn how to write functions in C++ and to bring them into TorqueScript so we could marry the advanced functioning of C++ with the processing of TorqueScript. Finally we talked about how to safely use memory in our function definitions and the various points of entry for the engine.

Our next step we're taking is going to be the largest step of the guide. The next two Chapters contain the **most important** bits of information you will learn from me as it pertains to C++, Torque, and your programming skills in general. Chapter 16 will introduce the power of C++ by teaching you the concept of class definitions, and Chapter 17 will combine that concept with Torque to show you how to define custom class and object instances in the engine. It's a massive juggernaut of words and learning, so let's get started!

Chapter 16: The Key of C++, Object-Oriented Programming

Chapter Introduction

Welcome to one of the most important chapters in the guide. Chapter 16 will cover the topic of object oriented programming as it pertains to C++, which is the process of creating and working with class instances in your programs. While the prior chapter took us inside the engine for a change, we'll be stepping outside of it again for this chapter so we can teach the new and important concepts needed. We'll start by doing a brief overview again of the topic of C++ global scope, and how a new type of file will be used for class instances, and then we'll actually get started with teaching how to write and work with objects in C++. Finally, we'll wrap up the chapter with topics on class inheritance and pointer instances.

So with the basic introduction of this chapter out of the way, let's get started on our work here!

C++ Header Files

We're going to start this chapter off with a brief review of C++'s global scope and how it pertains to our new topic of **header files** in C++. Back in Chapter 13, I introduced how C++ is a top-down language in terms of compilation, and how in order for any instance to exist to C++, it must be defined prior to its use in the codebase. This functioning is called global scope and is how C++ reads in class and function definitions.

Header Files: Introduction

Now imagine you're working on a very large project in C++. As you'll soon learn a class definition contains members and functions. Now imagine this is for instance a template class that is used by a large majority of your other C++ files in your project. You'll begin to see how re-defining the class over and over again would be an extremely tedious not to mention, a redundant task to undertake. Therefore, to combat this issue in C++ we use what is called a **header file**. A header file is a definition file, where functions and class instances are defined to your program. If you go all the way back to the early parts of Chapter 13, you'll come across this example program:

```
#include <iostream>
using namespace std;

int addEm(int a, int b = 20);

void main() {
    cout << "The Result Is: " << addEm(10);
}

int addEm(int a, int b = 20) {
    return a + b;
}
```

What we did in our example here was to **pre-define** the function instance by means of a **function prototype**. The easiest connection to make here is that a header file is a file that contains a

large list of **prototypes**, and usually prototypes a full class instance. By doing this, you can really easily include a class instance in multiple files without needing to redefine it over and over again.

Header Files: Format

So introductions aside, let's get started with the programming which is what you're really after here. To define a header file in C++ we need to abide by a specific format. Remember, C++ is a global scope oriented language which means you can't have multiple definitions of the same class running around in your code. To ensure this is not the case, we'll insert a **macro** into the header file to ensure the first time the file hits the global scope is also the last time it hits the global scope. Here is the most common format I use for my header files; you are free to use your own if you want:

```
/*
* MyFile.h
* (C) Name, Date
* Project Name
* License (If Any)
*
* Short Explanation of File
*/
#ifndef _MYFILE_H_
#define _MYFILE_H_

/*
    Make definitions here
*/

#endif // _MYFILE_H_
```

There's one new thing in this code, and that's the **#ifndef** conditional inclusion. I briefly mentioned that conditional inclusions will play a key role in this chapter, and this concept is the primary reason. The **#ifndef** conditional inclusion will be called if the tested variable **is not** defined. The first thing we do if we make it inside this block is to define the variable under the same name. This effectively makes sure that the only time this file's contents hits the global scope is the first time it's included in the project. Once we're inside the block, we can safely make all of the necessary definitions.

The **#include** Statement and your Header Files

The last thing to talk about as it pertains to header files is how to use them in your project. Unlike the .cpp files, a header file isn't by default pulled into the compilation list. In order to use a header file in your project you must define the point where you want to **include** it in your project, and this is where our old friend the **#include** preprocessor directive comes into play. To this point in time we've been surrounding our header files in greater than/less than symbols. This is a special format that the include statement uses that tells the compiler to look in the defined header locations for this file. This is typical when dealing with libraries, and whenever you want to use a standard template library header file. For our purposes though, we will alter the statement to work for our purposes. Let's revisit that prototype example now, but make use of a header file to accomplish this. **For all purposes in this chapter, I will be using the exact same header file named MyFile.h, if you change the name of this file please alter your code in the examples so your file is not affected by the change.**

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef __MYFILE_H__
#define __MYFILE_H__

    int addEm(int a, int b = 20);

#endif //__MYFILE_H__
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    cout << "The Result Is: " << addEm(10);
}

int addEm(int a, int b) {
    return a + b;
}
```

So let's look at what we've done. This time around we moved our function prototype to the header file of our project. We also moved the inclusion of the iostream header to our new header file as well. On the C++ file end, we use quotations to surround the name of the file which tells the compiler to look in the current working directory of the project. In a default C++ project this will match the current directory of the file you are in, however in Torque 3D this is changed to be the base of the **source** folder.

What I hope you learn quickly from this example is that you can use a header file to pull in additional files and namespace inclusions as needed by the contents of what the source files that need the header will be. When we get a little deeper into this chapter and more so into Chapter 17, you'll see that multiple files will call the same header file, if there are headers and libraries outside of the define block, those will also be brought in at the location of the include statement.

C++ works during the compilation project by creating object file instances (.o) which are then linked together into the executable by the linker. All of these references are resolved to the global scope allowing your program to have access where needed, so long as you followed the guidelines of the global scope. If something falls out of scope during compilation due to an invalid link, you'll get a **linker error** instead of a **compile error**. When this is the case, you'll want to check the compile order of your program and adjust your include statements as necessary.

And that's all you need to learn about header file as it pertains to this guide. There are many more things you could do with header files in general, but the primary focus is what is shown here. Feel free to dive deeper into the C++ Standard Online Reference if you need additional help with header files

Introduction to Classes, Objects

So with that out of the way, let's get to the primary topic of this chapter, which is the concept of object-oriented programming in C++. You got a small taste of this back in Chapter 14 when we talked a little bit about **structures** and how to define them, but I never really got deep into the focus or the mechanics of it as it pertains to the language itself. Now we're going to dive into this topic will full interest and discuss in full detail how it all works.

Classes: Introduction

In C++, a class instance is an expanded form of a data structure, where instances can have both members and functions. Unlike the prior form of structures we defined, a class instance by defaults initializes all members and functions to be **private** which means that only the class itself has access to read and modify the information held within. An **object** as it pertains to C++ is the initialized instance of a class.

You can write a class instance in either a .cpp or a .h file, however if you write the class instance in a .cpp file it will only remain legal according to scope within the contents of that file, which is why it is recommended that all of your class instances be defined within a header file.

To get started with the topic of classes, we're going to revisit one of our old examples from Chapter 14, which would be the structure example. This time around however, we're going to adjust the structure to be a class instance. There are a few changes to be noted here, and I'll explain them to you as we move through this section.

MyFile.h

```
#ifndef __MYFILE_H__
#define __MYFILE_H__

class myClass {
public:
    myClass();
    ~myClass();
    void set(int, int, int);
    void fetch(int *, int *, int *);

private:
    int num1;
    int num2;
    int num3;
};

#endif //__MYFILE_H__
```

MyFile.cpp

```
#include "MyFile.h"
#include <iostream>
using namespace std;

void main() {
```

```
myClass m;
m.set(10, 20, 30);
int *a = new int();
int *b = new int();
int *c = new int();
m.fetch(a, b, c);
cout << "Numbers: " << *a << ", " << *b << ", " << *c << endl;
}

myClass::myClass() : num1(0), num2(0), num3(0) {
}

myClass::~myClass() {

void myClass::set(int a, int b, int c) {
    num1 = a;
    num2 = b;
    num3 = c;
}

void myClass::fetch(int *a, int *b, int *c) {
    *a = num1;
    *b = num2;
    *c = num3;
}
```

Alright, so we've got a lot of good stuff going on here. First let's look at the header file to our new class definition to learn some new syntax. You'll notice one brand new thing right away and that would be the introduction of the keywords **public** and **private**. We'll get there in just a brief moment, for now just understand that those are what are called access level modifiers and they manipulate where data can be used within a class. Switching over to the C++ file, you'll notice the very same syntax being used here as the examples we did in the **structure functions** section of Chapter 14. You provide the return type first (unless it's a constructor/destructor) and then the name of the class, followed by the namespace operator, and then a standard function definition line.

Just remember that the contents within the header file are essentially function prototypes, they provide the C++ file with foreknowledge of the functions held within the specified class instance. Also, if you look at the constructor of our example class, you'll see that familiar value pre-define notation which automatically sets the values of the numbers within the class to zero. Now let's talk about those access level modifiers and what they're used for.

Classes: Access Level Modifiers

When you define a class instance in C++, one of the reasons for doing so is to protect the data contained within an individual class from unauthorized access. While the information can still be mined through external decompilation methods, the primary reason for incorporating data in a class is to ensure that the program doesn't try to use the data in a way it's not supposed to.

In C++, there are three **access level modifiers** that can be applied to sections within a class instance. These modifiers are: **public**, **private**, and **protected**.

The **public** access level modifier is used whenever the data or functions held within this section of a class can be freely used anywhere within the scope of the program. Generally you want to have any function that you want your program to be able to use from your specific class instance to be held under the public access level modifier. When you define a **structure** in C++, all members and functions contained by default are set to be under the public access level.

The **private** access level modifier is used whenever the data or functions held within this section of a class must remain within the confines of the specific class instance. The most common practice is to keep all values of a class held under a private access level modifier and then write public level functions to retrieve these values. This also ensures that no outside function can adjust these values without you knowing about it. The only way a **private** member or function can be accessed is if the body of the calling scope lies within the same class, or has been granted **friend-level access** to the specific class. By default in C++, all **class** members and functions contained by default are set to be under the private access level.

Finally, we have the **protected** access level modifier. This modifier provides almost identical settings as the **private** modifier with one key change. This is that classes that are inherited from your current class instance have access to protected members within the parent class. There are no places in C++ where the protected access level modifier is enabled by default, and you can usually skip it unless you are writing a base class instance where multiple derivation classes will be derived from your base class. We'll come back to this modifier level in a little while.

To actually demonstrate these access level modifiers, we'll go to our next example. This time we're going to have two class instances. Each of them will have something the other wants, but stored outside of its normal reach:

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef _MYFILE_H_
#define _MYFILE_H_
    class Volume; //<- Predefine for Area

    class Area {
        public:
            Area();
            void assignFrom(int l, int w);
            void assignFrom(Volume v);
            int calculate();

            int fetchL() { return length; }
            int fetchW() { return width; }

        private:
            int length;
            int width;
    };

```

```
class Volume {
public:
    Volume();
    void assignFrom(int l, int w, int h);
    void assignFrom(Area a, int h);
    int calculate();

    int fetchL() { return length; }
    int fetchW() { return width; }
    int fetchH() { return height; }

private:
    int length;
    int width;
    int height;
};

#endif // _MYFILE_H_
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    cout << "Area & Volume Example" << endl;
    Area a;
    Volume v;
    a.assignFrom(10, 10);
    cout << "Area of " << a.fetchL() << "x" << a.fetchW() << " is: " <<
a.calculate() << endl;
    v.assignFrom(a, 10);
    cout << "Volume of " << v.fetchL() << "x" << v.fetchW() << "x" <<
v.fetchH() << " is: " << v.calculate() << endl;
    cout << "Adjust Volume..." << endl;
    v.assignFrom(5, 7, 9);
    cout << "Volume of " << v.fetchL() << "x" << v.fetchW() << "x" <<
v.fetchH() << " is: " << v.calculate() << endl;
    a.assignFrom(v);
    cout << "Area of " << a.fetchL() << "x" << a.fetchW() << " is: " <<
a.calculate() << endl;
}

// AREA
Area::Area() : length(0), width(0) { }

void Area::assignFrom(int l, int w) {
    length = l;
    width = w;
}

void Area::assignFrom(Volume v) {
    length = v.fetchL();
    width = v.fetchW();
}
```

```
int Area::calculate() {
    return length * width;
}

// VOLUME
Volume::Volume() : length(0), width(0), height(0) { }

void Volume::assignFrom(int l, int w, int h) {
    length = l;
    width = w;
    height = h;
}

void Volume::assignFrom(Area a, int h) {
    length = a.fetchL();
    width = a.fetchW();
    height = h;
}

int Volume::calculate() {
    return length * width * height;
}
```

Aside from being one of the longest C++ programs we've done to this point, we do a few new tricks with this example program. First of all, you'll notice in the header file that we're "prototyping" the class named **Volume** before the definition of **Area**. We do this because the **Area** class uses the **Volume** class and header files obey the same compile order rule as cpp files. Also, this example does offer one brand new thing to which you have thought was an error so far. If you read closely, you'll notice that not only do two functions both use the function named **assignFrom**, but each of them have **two definitions of this function**. Now before you scream at me saying this long code is broken, go ahead and actually run it.

The notion I gave you before that two functions cannot share the same name, is incorrect. But not completely so. The actual rule is that two functions cannot share the same name, if and only if these two functions have matching parameters. Basically, I cannot define the **assignFrom** function twice if they both use two integer parameters. However, if you look closely, the functions differ in parameters because they use a combination of integers and object instances.

Next up, you need to pay close attention to the access level that each member of these classes holds. The integers stored inside each class instance here is set to the private level, which means they can only be read and written to by functions contained within that class. Since the **assignFrom** method exists within the class itself, we can write to those members. But what about reading from these classes? That's where the **fetchL**, **fetchW**, and **fetchH** functions come into play. These functions I defined within the class themselves since they are simplistic in nature. Since the function exists within the boundaries of the class instance, we have access to read the member and return it to the calling source.

Before we move on, I want to show you one more access level modifier. This last access modifier is used to work around the private and protected access levels for other class instances when needed.

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef __MYFILE_H__
#define MYFILE_H
    class Volume; //<- Predefine for Area

    class Area {
        public:
            Area();
            void assignFrom(int l, int w);
            void assignFrom(Volume v);
            int calculate();

            int fetchL() { return length; }
            int fetchW() { return width; }

        private:
            int length;
            int width;
            friend class Volume;
    };

    class Volume {
        public:
            Volume();
            void assignFrom(int l, int w, int h);
            void assignFrom(Area a, int h);
            int calculate();

            int fetchL() { return length; }
            int fetchW() { return width; }
            int fetchH() { return height; }

        private:
            int length;
            int width;
            int height;
            friend class Area;
    };

#endif //__MYFILE_H__
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    cout << "Area & Volume Example" << endl;
    Area a;
    Volume v;
    a.assignFrom(10, 10);
    cout << "Area of " << a.fetchL() << "x" << a.fetchW() << " is: " <<
a.calculate() << endl;
```

```
v.assignFrom(a, 10);
    cout << "Volume of " << v.fetchL() << "x" << v.fetchW() << "x" <<
v.fetchH() << " is: " << v.calculate() << endl;
    cout << "Adjust Volume..." << endl;
    v.assignFrom(5, 7, 9);
    cout << "Volume of " << v.fetchL() << "x" << v.fetchW() << "x" <<
v.fetchH() << " is: " << v.calculate() << endl;
    a.assignFrom(v);
    cout << "Area of " << a.fetchL() << "x" << a.fetchW() << " is: " <<
a.calculate() << endl;
}

// AREA
Area::Area() : length(0), width(0) { }

void Area::assignFrom(int l, int w) {
    length = l;
    width = w;
}

void Area::assignFrom(Volume v) {
    length = v.length;
    width = v.width;
}

int Area::calculate() {
    return length * width;
}

// VOLUME
Volume::Volume() : length(0), width(0), height(0) { }

void Volume::assignFrom(int l, int w, int h) {
    length = l;
    width = w;
    height = h;
}

void Volume::assignFrom(Area a, int h) {
    length = a.length;
    width = a.width;
    height = h;
}

int Volume::calculate() {
    return length * width * height;
}
```

In this example we made one little change. You'll notice two new lines in the header file. The last line of each class instance now applies what is called **friend** to the other class instance. What this does is applies what is called the friendship modifier to the other class. This allows the other class to have access to private and protected members within the current class. If you look in the cpp file we changed the **assignFrom** members to directly access the members of the other class now because it is

perfectly legal to do so. The reason we kept the fetch methods is because the **main** function does not have friend level access to our two class instances and therefore cannot access the members.

So now that we have the basics of writing C++ classes down, let's move onto our next topic for classes in C++.

Template Classes

Next up is an expansion of two prior topics we've covered before. Template functions and class instances. A template class is frequently used when the features of the class are generic in nature, which is to say they provide a level of functionality to all forms of data types out there. Most of the standard template library classes are template classes because they work with all forms of data types out there.

To define a template style class, we do the exact same thing we did before with our template structure, however we switch the instance to a class definition. This also means that we need to include some access level modifiers to ensure we can use them in our code itself:

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef __MYFILE_H__
#define __MYFILE_H__

template <class T> class Container {
public:
    Container(T in) : element(in) { }

    T fetch() { return element; }
    void increment() { element++; }
    void decrement() { element--; }
private:
    T element;
};

#endif // __MYFILE_H__
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    Container<int> myNumber(20);
    Container<char> myLetter('q');

    cout << "Contents Are: " << myNumber.fetch() << ", " <<
myLetter.fetch() << endl;
    myNumber.decrement();
    myLetter.increment();
    cout << "Contents Are Now: " << myNumber.fetch() << ", " <<
myLetter.fetch() << endl;
}
```

One of the big changes we made here aside from using a class to define our template instance was in the definition of the template itself. You'll notice how we define the template instance as **class T**. What this does, unlike using the **typename** keyword, is allows your template to use both standard C++ data types and your class instances through the template instance.

Specialized Templates

One of the problems we overlooked last time we talked about templates was the possibility of introducing an unintended syntax or logic error to the program by means of using a template class instance. Imagine for a moment we send a **char *** or an unknown class instance to our template container above and then try to use the increment and decrement commands on the container instance. There would be unintended and perhaps disastrous consequences to the program if you tried to do this. One way we can help to mitigate potential flaws in the code is to use what is called a **specialized template**. A specialized template instance is a predefined form of a template class instance; basically it tells the compiler that when it receives 'x' class, to use 'y' behavior in its place. Here's how you deploy a specialized template instance:

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef _MYFILE_H_
#define _MYFILE_H_

template <class T> class Container {
public:
    Container(T in) : element(in) { }

    T fetch() { return element; }
    void increment() { element++; }
    void decrement() { element--; }
private:
    T element;
};

template <> class Container<char *> {
public:
    Container(char *in) : element(in) { }

    char *fetch() { return element; }
    void increment() { printf("Cannot use increment on char\n"); }
    void decrement() { printf("Cannot use decrement on char\n"); }
private:
    char *element;
};

#endif // _MYFILE_H_
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    Container<int> myNumber(20);
    Container<char> myLetter('q');
    Container<char *> myWord("Hello");

    cout << "Contents Are: " << myNumber.fetch() << ", " <<
myLetter.fetch() << ", " << myWord.fetch() << endl;
    myNumber.decrement();
    myLetter.increment();
    myWord.increment();
    cout << "Contents Are Now: " << myNumber.fetch() << ", " <<
myLetter.fetch() << ", " << myWord.fetch() << endl;
}
```

For the highest level of program security you could reverse this logic and write template specializations for each of the class instances you want to function properly and then have the generic template contain the error back out methods. Let's show off one more example of templates to provide focus on writing the functions in the C++ file this time instead of containing them in the header itself so you can see how that is done:

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef _MYFILE_H_
#define _MYFILE_H_

template <class T> class Container {
public:
    Container(T in) : element(in) { }

    T fetch();
    void increment();
    void decrement();
private:
    T element;
};

template <> class Container<char *> {
public:
    Container(char *in) : element(in) { }

    char *fetch();
    void increment();
    void decrement();
private:
    char *element;
};

#endif // _MYFILE_H_
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    Container<int> myNumber(20);
    Container<char> myLetter('q');
    Container<char *> myWord("Hello");

    cout << "Contents Are: " << myNumber.fetch() << ", " <<
myLetter.fetch() << ", " << myWord.fetch() << endl;
    myNumber.decrement();
    myLetter.increment();
    myWord.increment();
    cout << "Contents Are Now: " << myNumber.fetch() << ", " <<
myLetter.fetch() << ", " << myWord.fetch() << endl;
}

//Container<class T>
template <class T> T Container<T>::fetch() {
    return element;
}

template <class T> void Container<T>::increment() {
    element++;
}

template <class T> void Container<T>::decrement() {
    element--;
}

//Container<char *>
char *Container<char *>::fetch() {
    return element;
}

void Container<char *>::increment() {
    cout << "Cannot perform ::increment() on char *" << endl;
}

void Container<char *>::decrement() {
    cout << "Cannot perform ::decrement() on char *" << endl;
}
```

For the generic template functions in the above example, you'll notice a semi-familiar template notion to the functions we worked with back in Chapter 14. The change to the old syntax comes with the class name separated with the namespace operator. You'll see how we surround the template variables in greater than/less than symbols here.

For the specialized template, you skip out on the template notation because there are no template variables being defined for this particular instance, and in place of the class name template variables, we provide in place the specialized variable type.

Multiple T-Variables, Partial Template Specialization

In C++, you don't want to limit your programs in any way or form. Complexity can sometimes add very advanced features and tools to your program and if these features can cut down on the time needed to incorporate other features into your project then you should definitely go for it.

You can very easily expand your template classes to accept multiple parameters at once, and you can even mix around template specializations and generic template instances to fine tune your class to perform exactly as you intended it to. We'll look at one good lengthy example to talk about this and then we'll move on to our next topic in classes.

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef __MYFILE_H__
#define __MYFILE_H__

template <typename T1, typename T2, typename T3, typename T4>
class Container {
public:
    Container(T1 v1, T2 v2, T3 v3, T4 v4) {
        e1 = v1;
        e2 = v2;
        e3 = v3;
        e4 = v4;
        printf("Initialized using generic format.\n");
    }

    ~Container() {
        printf("Generic Poof!\n");
    }

    T1 fetch1() { return e1; }
    T2 fetch2() { return e2; }
    T3 fetch3() { return e3; }
    T4 fetch4() { return e4; }
private:
    T1 e1;
    T2 e2;
    T3 e3;
    T4 e4;
};

template <typename T1, typename T2>
class Container<char *, T1, int, T2> {
public:
    Container(char *v1, T1 v2, int v3, T2 v4);

    ~Container() {
        printf("Partial Spec. Poof!\n");
    }

    char *fetch1();
```

```
    T1 fetch2();
    int fetch3();
    T2 fetch4();
private:
    char *e1;
    T1 e2;
    int e3;
    T2 e4;
};

#endif // _MYFILE_H_
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    cout << "Init Templates..." << endl;
    Container<int, int, int, int> *c1
        = new Container<int, int, int, int>(10, 20, 30, 40);
    Container<char *, double, int, char> *c2
        = new Container<char *, double, int, char>("Hello", 2.0, 10,
'r');
    cout << "Container 1: " << endl;
    cout << c1->fetch1() << ", " << c1->fetch2() << ", " << c1->fetch3() <<
", " << c1->fetch4() << endl;
    cout << "Container 2: " << endl;
    cout << c2->fetch1() << ", " << c2->fetch2() << ", " << c2->fetch3() <<
", " << c2->fetch4() << endl;
    delete c1;
    delete c2;
}

//Container functions
template <typename T1, typename T2>
Container<char *, T1, int, T2>::Container(char *v1, T1 v2, int v3, T2
v4) {
    e1 = v1;
    e2 = v2;
    e3 = v3;
    e4 = v4;
    cout << "Initialized using partial specialization template" << endl;
}

template <typename T1, typename T2> char *
Container<char *, T1, int, T2>::fetch1() {
    return e1;
}

template <typename T1, typename T2> T1
Container<char *, T1, int, T2>::fetch2() {
    return e2;
}

template <typename T1, typename T2> int
Container<char *, T1, int, T2>::fetch3() {
```

```
    return e3;
}

template <typename T1, typename T2> T2
Container<char *, T1, int, T2>::fetch4() {
    return e4;
}
```

This very lengthy example makes use of what we call **partial template specialization**, which is the process in which some of the template variables are defined, while others remain as template variables. I've also brought back the pointer instances for this particular example to show you how you can define a pointer instance of a template class and how to delete them (no change). The rest of this example pretty much covers stuff you've already seen in the prior chapters and this section, so read over it, re-read it, and play around with this example until you've got a firm grasp on template classes, template functions, and specialization.

I leave any further use of template class creation to your imagination. As I stated back in Chapter 14, you now have all the tools at your disposal to re-create the Standard Template Library's linked list implementation, so if you're up for a little challenge, go for it!

Operator Overloading, this Keyword

Our next topic for classes deals with something I talked about way back in Chapter 13 when you were first learning about **std::cout**. You may or may not remember, but I briefly mentioned how **std::cout** was a class instance and it was **overloading** the bitwise right-shift operator to the compiler. In this section I'll teach you how classes can overload the standard C++ operators and I'll also teach you about the **this** keyword which is a special keyword used in class programming to obtain a pointer instance to the current class.

Which Operators can be Overloaded?

First and foremost, let's talk about the method of overloading operators in C++. When you overload an operator in C++, what you are doing is changing the behavior of the standard C++ functioning of the operator. Before we actually introduce this concept as a whole, let's start by showing you which operators can be overloaded. And for that, here's a table containing the full list of overloadable operators:

+	-	*	/	=	<	>	+=	-=	*=
/=	<<	>>	<<=	>>=	==	!=	<=	>=	++
--	%	&	^	!		~	&=	^=	=
&&		%=	[]	()	,	->*	->	new	delete
new[]	delete[]								

So as you can see there is a large list of operators that can be overloaded in C++. Some of these will be familiar to you, others you may or may not have seen before, even in TorqueScript. The purpose of this section is not to explain what each of these operators are for, or their behavior. That is for your own time and alternate learning material to cover.

What this section will cover however is the proper syntax to overload these operators for usage in your own class instances. Each of these operators behaves differently and to safely overload these operators you need to know the correct and acceptable syntax to overload these operators with.

Generally, the syntax of using these operator overloads falls under a set number of guidelines, mainly dealing with whether or not the instance is a member function or a standard function definition, and what operator you are using. For the below table you can make the following assumptions.

1. A, B, C refer to class instance names. The definition of A is noted below. For usage in the function lines you can either refer to it directly as A, or in the forms **const A&**, **const A***.

```
class A {  
  
};
```

2. a, b, c refer to the defined object instances of the global class instances, so for instance:
`A a;`
3. # refers to any of operators listed by the specific section of the table.

Operator(s)	Expression	Member-Function	Non-Member Function *2
+ - * & ! ~ ++ --	#a	A::operator#()	operator#(A)
++ --	a#	A::operator#(int)	operator#(A, int)
+ - * / % ^ & < > == != >= <= <> && ,	a#b	A::operator#(B)	operator#(A, B)
= += -= *= /= %= ^= &= = <<= >>= []	a#b	A::operator#(B)	N/A
()	a(b, c, ..)	A::operator#(B, C, ..)	N/A
->	->	A::operator#()	N/A
TYPENAME *1	(type)a	A::operator TYPENAME()	N/A

*1: TYPENAME refers to any data type in which explicit type conversion will be allowed for your defined class instance.

*2: Not all operators have legal non-member definitions. For the rows with N/A specified there are no legal non-member variants of the specified operator overload.

*3: Not all operators that can be overloaded are shown in this table. For most operators you can assume a standard behavior on the order of the Member-Function definition of row #1, for any conflicts arising due to an operator overload, refer to the online C++ reference for operator overloading to see the correct syntax and usage for your specific overload.

Operator Overloading Examples, The this Keyword

So now that you have the generic format and the operator list in hand, let's actually look at a few examples of the operator overloading in action.

`MyFile.h`

```
#include <iostream>
```

```
using namespace std;

#ifndef _MYFILE_H_
#define _MYFILE_H_

class Numerics {
public:
    Numerics() : one(1), two(2), three(3) { }
    ~Numerics() { }

    void set(int, int, int);
    void fetch(int *, int *, int *);

    Numerics &operator+(const Numerics &);
    Numerics &operator-(const Numerics &);

private:
    int one;
    int two;
    int three;
};

#endif //_MYFILE_H_
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    int a1, b1, c1;
    Numerics a;
    Numerics b;

    a.set(10, 20, 30);
    b.set(40, 50, 60);

    a.fetch(&a1, &b1, &c1);
    cout << "Current A: " << a1 << ", " << b1 << ", " << c1 << endl;
    b.fetch(&a1, &b1, &c1);
    cout << "Current B: " << a1 << ", " << b1 << ", " << c1 << endl;
    //Make A = A + B
    a = a + b;
    a.fetch(&a1, &b1, &c1);
    cout << "Now A: " << a1 << ", " << b1 << ", " << c1 << endl;
    //Revert.
    a = a - b;
    a.fetch(&a1, &b1, &c1);
    cout << "Now A: " << a1 << ", " << b1 << ", " << c1 << endl;
}

//Numerics Class Definition
void Numerics::set(int a, int b, int c) {
    one = a;
    two = b;
    three = c;
}
```

```
void Numerics::fetch(int *a, int *b, int *c) {
    *a = one;
    *b = two;
    *c = three;
}

Numerics &Numerics::operator+(const Numerics &r) {
    one += r.one;
    two += r.two;
    three += r.three;

    return *(this);
}

Numerics &Numerics::operator-(const Numerics &r) {
    one -= r.one;
    two -= r.two;
    three -= r.three;

    return *(this);
}
```

The above example will show you how operators are overridden and then deployed in a C++ program. You can see how our two mathematical operators now work to perform mathematics on the internal values instead of trying to add the objects together (which would be bad). But now there's one line of trickery in the above program that you've not seen yet:

```
return *(this);
```

What exactly is this line doing? That line introduces the **this** keyword to you. The keyword **this** is a special class instance keyword that is used to return a constant pointer to the current class you are working with. Since our function returns a reference and not a pointer, we apply the dereference operator to the keyword to allow the function to return the value. The reason to use this return value in an operator is simple. Once you make a change to the contents of the class instance, you no longer want the old data hanging around, so instead we pass a reference to the new class data instance instead.

While we're still on the same example it might also help to point out that **const Numerics &r** could actually be pointing to a different instance (and it actually does in our example), so how can we access the private members of the other class? Easy, you need to remember that in C++ the compiler treats all classes of the same instance as **direct friends**, therefore we can access the private members of the **&r** class even though we're not directly working with it. If you were however using another class that did not have this relation, you would need to use the friend operator to grant access to the contents of the other class, or insert a fetch method to grab the relevant data.

Now let's look at an example that uses a few other operator types to demonstrate how to use those as well. We'll modify our prior example to read the results into a structure which is then printed via arrays:

MyFile.h

```
#include <iostream>
```

```
using namespace std;

#ifndef _MYFILE_H_
#define _MYFILE_H_

struct Numbers {
    int a;
    int b;
    int c;

    void set(int n1, int n2, int n3);
    void fetch(int n[3]);
};

class Numerics {
public:
    Numerics() : one(1), two(2), three(3) { }
    ~Numerics() { }

    void set(int, int, int);
    void operator<<(Numbers *o);
    Numerics &operator+=(int nums[3]);
private:
    int one;
    int two;
    int three;
};

#endif // _MYFILE_H_
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    int alpha[3] = {5, 5, 5};
    int bravo[3];
    Numerics a;
    Numbers n;

    a.set(10, 20, 30);
    a += alpha;
    a << &n;

    n.fetch(bravo);

    cout << "Numbers: " << bravo[0] << ", " << bravo[1] << ", " << bravo[2]
<< endl;
}

//Numbers Struct Definition
void Numbers::set(int n1, int n2, int n3) {
    a = n1;
    b = n2;
    c = n3;
}
```

```

void Numbers::fetch(int n[3]) {
    n[0] = a;
    n[1] = b;
    n[2] = c;
}

//Numerics Class Definition
void Numerics::set(int a, int b, int c) {
    one = a;
    two = b;
    three = c;
}

void Numerics::operator<<(Numbers *o) {
    o->set(one, two, three);
}

Numerics &Numerics::operator+=(int n[3]) {
    one += n[0];
    two += n[1];
    three += n[2];

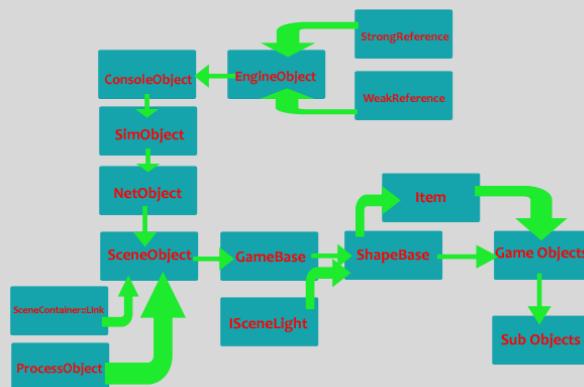
    return *(this);
}

```

You could very easily adjust this example to treat the left shift operator to accept an array as well to place the values of the class into the array. This is just to demonstrate how this works. Also keep in mind of the proper usage of the keyword **this** when working with classes and operator functions in particular.

Inheritance

Next up is the topic of class inheritance. Just like before in TorqueScript the concept of class inheritance is to take the functioning and members of a parent class and copy them down to the next class, giving access to the parent class' functions and members to the new instance as well. We've worked with inheritance twice now. Once in Chapter 8 when we were working with our script objects, and then again in Chapter 11 when we talked about the engine's object tree with this picture:



We'll get back to that picture again very shortly in Chapter 17, but for the time let's get back to the topic of class inheritance. It's a little trickier to "write" in C++, but the concept is exactly the same as before. You write a parent class instance, and then a derivation class which pulls from the parent class.

Single Class Inheritance

The most common type of class inheritance in C++ is **single inheritance**, which is the process of pulling in the members of a single class instance to create a derivation class. Here's an example of this in action, and then we'll talk about the technical stuff:

MyFile.h

```
#include <iostream>
#include <cstdarg>
using namespace std;

#ifndef _MYFILE_H_
#define _MYFILE_H_

class Printer {
public:
    void print(const char *fmt, ...);
};

class Poly : public Printer {
public:
    Poly() : width(1), height(1) { }
    Poly(int w, int h) : width(w), height(h) { }

    int fetchArea();
    void out();
protected:
    int width;
    int height;
};

#endif // _MYFILE_H_
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    cout << "Inheritance Example..." << endl;
    Poly p(10, 20);
    p.out();
}

//Printer Class
void Printer::print(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    vprintf(fmt, args);
    va_end(args);
}
```

```
//Poly Class
int Poly::fetchArea() {
    return width * height;
}

void Poly::out() {
    print("The area of %i*%i is: %i\n", width, height, fetchArea());
}
```

We'll start with the **Printer** class instance. You'll see a cryptic looking notation in C++ that might have you scratching your head for a moment with the **Printer::print** function. This is in fact what we call a **variable argument function**, which is a special function definition that accepts any number of variables. For the purpose of this example, this is a *printf wrapper function*. A **wrapper function** is a function written in any language whose only purpose is to call a secondary function, usually with no additional computation. These functions primarily exist to call library functions as needed by a program.

Our second class definition is where the inheritance stuff kicks off. To create an inherited class you separate your class line with a single colon, and then provide a list of classes you wish to inherit. To give additional functioning, you also provide an access level to the class you are seeking to inherit from. These are the exact same modifiers as earlier. To help you pick the right choice, look at this table:

Access Operation	public	protected	private
Members of the same class instance	Yes	Yes	Yes
Members of the derived class instance	Yes	Yes	No
Non-Members, Outside Functions	Yes	No	No

From that point forward, everything else is exactly the same as it pertains to the class definitions. What we did by inheriting the **Printer** class on our **Poly** class was grant access to the **print()** method on our **Poly** class.

I'm not going to go into great detail of the **va_** functions seen in the **print()** method, as I said before the last time this topic came up, you're welcome to visit the online C++ references to learn about variable argument functions, there are plenty of great online resources and tutorials available for those types of functions. What we are interested in for the above example is the **Poly::out()** function. Since we've inherited the **Printer** class, we can directly call the **print()** method we created in the **Printer** class in our **Poly** class. By granting public inheritance to our class, we have direct access to the parent class members and functions.

Multiple Inheritance

When you begin to dive deeper into the engine however, you'll find multiple cases where inheriting one single class, just won't be enough for the application needed for a class instance. This is perfectly fine, because in C++ you can inherit multiple class instances to derive a new class. The notation

is pretty much identical to before, however we just make a comma separated list of the classes we want to inherit from.

With that in mind, let's modify our previous example by creating a third class instance called Triangle. We know triangles are polygons, but the calculation of the area is slightly different, so we'll need a new method for calculating it. We also still need the Printer class to allow our Triangle to output, so we'll inherit multiple classes to accomplish this feat.

MyFile.h

```
#include <iostream>
#include <cstdarg>
using namespace std;

#ifndef __MYFILE_H__
#define __MYFILE_H__

class Printer {
public:
    void print(const char *fmt, ...);
};

class Poly {
public:
    Poly() : width(1), height(1) { }
    Poly(int w, int h) : width(w), height(h) { }

    int fetchArea();
protected:
    int width;
    int height;
};

class Triangle : public Poly, public Printer {
public:
    Triangle() : Poly() { }
    Triangle(int w, int h) : Poly(w, h) { }

    int fetchArea();
    void out();
};

#endif //__MYFILE_H__
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    cout << "Multiple Inheritance Example..." << endl;
    Triangle t(10, 20);
    t.out();
}
```

```
//Printer Class
void Printer::print(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    vprintf(fmt, args);
    va_end(args);
}

//Poly Class
int Poly::fetchArea() {
    return width * height;
}

//Triangle Class
int Triangle::fetchArea() {
    return (int)(0.5 * width * height);
}

void Triangle::out() {
    print("The area of this triangle (%i, %i) is: %i.\n", width, height,
fetchArea());
}
```

So there are a few new things to talk about in this particular example. Let's start with the header again. You'll notice I removed the inheritance route to Printer on the Poly class. This is simply because we don't need to inherit the same class twice. Yes, you could either do it using the prior example's way of inheriting Printer on the Poly class and then using single inheritance on the derived classes, or use multiple inheritance to do it all at once, the choice is yours to make.

You'll also notice that in the constructor methods on the Triangle class, that we're actually calling the constructor of the Poly class. We do this to properly initialize the parent class' members. We can't initialize them in our own class because they don't exist there. Also, this is a good time to introduce the **protected** access level. I briefly mentioned it a little while back with a limited amount of detail, so let's explain it a little more here. **Protected** works on the same basis as **private** with a key difference. If you inherit a class instance, the derivation class will have access to **protected** members. They still can't use **private** members, but they'll have access to **protected** ones. This is why you usually only use the **protected** access level modifier when the class instance is going to be inherited by another class, otherwise you can use the **private** access level.

Polymorphism, Virtual Functions, and Abstract Classes

Our final stop in Chapter 16 deals with what we call **polymorphism**. This concept is extremely useful when creating a **base class instance** or a class that will be inherited by usually a large number of derivation class instances. The base class instance creates the template of the new class, and the derivation classes follow the template to form a new instance. To successfully learn this topic in C++, we'll pull three prior topics together and teach a few new concepts to you as well.

The actual power of **polymorphism** deals with the act of creating pointers to a base class instance and then having access to two separate classes over an easy set of variable instances.

Pointer to Base Class

Let's start by introducing the concept of polymorphism, or the act of taking a derived class instance, and creating a pointer to a base class. What this functioning allows us to take advantage of is when dealing with multiple derivation class instances that implement varying forms of a function instance. In our case here, we'll create our Poly class again, but this time we'll have both a Triangle and a Rectangle class instance, and both of these classes will require a different form of the fetchArea method.

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef __MYFILE_H__
#define __MYFILE_H__

class Poly {
public:
    Poly() : width(0), height(0) { }

    void set(int w, int h);
protected:
    int width;
    int height;
};

class Triangle : public Poly {
public:
    int fetchArea();
};

class Rect : public Poly {
public:
    int fetchArea();
};

#endif //__MYFILE_H__
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    cout << "Polymorphism Example..." << endl;
    Triangle t;
    Rect r;
    //Pointer-to-base
    Poly *p1 = &t;
    Poly *p2 = &r;
    p1->set(10, 20);
    p2->set(10, 20);
    cout << "Fetching Area..." << endl;
    cout << "Area of Triangle is: " << t.fetchArea() << endl;
    cout << "Area of Rectangle is: " << r.fetchArea() << endl;
```

```
}
```

```
//Poly Class
void Poly::set(int w, int h) {
    width = w;
    height = h;
}

//Triangle Class
int Triangle::fetchArea() {
    return (int)(0.5 * width * height);
}

//Rect Class
int Rect::fetchArea() {
    return width * height;
}
```

This time let's pay attention mainly to the C++ source file here; since the header has things we've already seen quite a few times. The main focus of our code is in the **main** method. We start by creating a Triangle and a Rect object instance. What comes next is the **polymorphism** part of our code. We initialize two pointers to our Poly class by using the reference of our derivation classes. This is legal in C++, because by definition our derived classes are directly compatible with the parent class because they exist along the same inheritance tree. By doing this, we can now use the Poly class' set method to actually set the variables of our derivation classes because they are linked via pointer. From there, it's just using the fetchArea method on the original object instances to print out the values we need to see.

Virtual Functions

Now from a programming standpoint, that's a cool feature to incorporate, but it might get a little repetitive to have to constantly jump back and forth between the pointer and the original object, but what if there was a way to just create our pointer to the base class, and then stay inside the pointer itself? Well, actually there is a way to accomplish this functioning and it's called **virtual functions**.

A **virtual function** is a function that can be defined in a base class, and then be redefined in any inherited classes, all while preserving the properties of the original class by means of linked references. There are two types of virtual functions, there are the standard forms of virtual functions, which we'll talk about here. And then there are the pure virtual functions, which we'll get to right after we finish with our example here.

So let's go back to our prior example now, but this time we'll create a virtual function to calculate the area:

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef _MYFILE_H_
#define _MYFILE_H_
```

```
class Poly {
public:
    Poly() : width(0), height(0) { }

    void set(int w, int h);
    virtual int fetchArea();
protected:
    int width;
    int height;
};

class Triangle : public Poly {
public:
    int fetchArea();
};

class Rect : public Poly {
public:
    int fetchArea();
};

#endif // _MYFILE_H_
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    cout << "Polymorphism Example..." << endl;
    //Pointer-to-base
    Poly *p1 = new Triangle();
    Poly *p2 = new Rect();
    Poly *p3 = new Poly();
    p1->set(10, 20);
    p2->set(10, 20);
    p3->set(10, 20);
    cout << "Fetching Area..." << endl;
    cout << "Area of Triangle is: " << p1->fetchArea() << endl;
    cout << "Area of Rectangle is: " << p2->fetchArea() << endl;
    cout << "Area of Poly P3 is: " << p3->fetchArea() << endl;
}

//Poly Class
void Poly::set(int w, int h) {
    width = w;
    height = h;
}

int Poly::fetchArea() {
    return 0;
}

//Triangle Class
int Triangle::fetchArea() {
    return (int)(0.5 * width * height);
}
```

```
//Rect Class
int Rect::fetchArea() {
    return width * height;
}
```

So, let's look at the header really quickly. You'll notice the change we made in this example was to add a line to the Poly definition **virtual int fetchArea()**. What this line does it tell the compiler that the method **int fetchArea()** is a virtual definition, which means that the method will be established as a template function that will have a default output that can then be overridden by any derivative classes.

As for the C++ definition of the method, we define a generic function that simply returns the number zero if the generic function is called. To test this functioning, we create a third Poly pointer which is simply a Poly class definition. The other change we made here was to directly create our Poly pointers by calling the constructor of the derivative classes, which is another legal definition of **polymorphism** in C++, however, by doing it this way you cannot call the functions of the Rect/Triangle classes, as the pointers exist as Poly definitions.

By using polymorphism and virtual functions, you'll have much more power over your class inheritance, and you'll be able to safely move up and down the inheritance chain of your object instance as needed.

Pure Virtual Functions, Abstract Classes

Our final topic for this section deals with the **pure virtual functions**. A pure virtual function is an abstract function template definition that states that any derivation classes MUST override that function instance in order to be a legal derivation. By the definition of a **pure virtual function**, any class instance that defines at least one pure virtual function is by definition an **abstract class instance**. An **abstract class** is essentially a class template, defining functions and members that can be used by inherited class instances. An **abstract class** cannot be directly instanced as an object as it only exists as a template instance.

Let's rewrite our prior example now, but this time we'll turn our Poly class into an **abstract class** and use **pure virtual functions** on the definition lines:

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef _MYFILE_H_
#define _MYFILE_H_

class Poly {
public:
    void set(int w, int h);
    virtual int fetchArea() = 0;
protected:
    int width;
    int height;
```

```

};

class Triangle : public Poly {
public:
    int fetchArea();
};

class Rect : public Poly {
public:
    int fetchArea();
};

#endif // _MYFILE_H_

```

MyFile.cpp

```

#include "MyFile.h"

void main() {
    cout << "Abstract Class Example..." << endl;
    //Pointer-to-base
    Poly *p1 = new Triangle();
    Poly *p2 = new Rect();
    //Poly *p3 = new Poly(); //<- Illegal
    p1->set(10, 20);
    p2->set(10, 20);
    cout << "Fetching Area..." << endl;
    cout << "Area of Triangle is: " << p1->fetchArea() << endl;
    cout << "Area of Rectangle is: " << p2->fetchArea() << endl;
}

//Poly Class
void Poly::set(int w, int h) {
    width = w;
    height = h;
}

//Triangle Class
int Triangle::fetchArea() {
    return (int)(0.5 * width * height);
}

//Rect Class
int Rect::fetchArea() {
    return width * height;
}

```

You'll notice that by using an **abstract class** we're able to axe out quite a few lines from our prior example. First and foremost, you'll notice the syntax for the **pure virtual function**, is to simply append the **= 0** to the end of the function definition line. That alone tells the compiler that the **fetchArea** method must now be defined on all of the classes which derive the **Poly** class, and it converts the **Poly** class to become an abstract instance.

From that point forward, we've kept pretty much everything else the same. We no longer need to define a generic template of `fetchArea` for `Poly` since it is a pure virtual function and needs no definition. You can also see here that while you cannot directly define a `Poly` class instance, that you can still use other methods and members of the class like any other piece of code you've used to this point in time.

There's one more little trick I want to show you before we wrap things up with Chapter 16 here, and that is another useful application of the special keyword `this`. Remember, the `this` keyword is used to return a **constant pointer** to the object we're directly working with. But what happens when you try to use this keyword on an abstract class? Take a peek at this example:

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef __MYFILE_H__
#define __MYFILE_H__

class Poly {
public:
    void set(int w, int h);
    void printArea();
    virtual int fetchArea() = 0;
protected:
    int width;
    int height;
};

class Triangle : public Poly {
public:
    int fetchArea();
};

class Rect : public Poly {
public:
    int fetchArea();
};

#endif //__MYFILE_H__
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    cout << "Abstract Class Example..." << endl;
    //Pointer-to-base
    Poly *p1 = new Triangle();
    Poly *p2 = new Rect();
    //Poly *p3 = new Poly(); //<- Illegal
    p1->set(10, 20);
    p2->set(10, 20);
```

```
cout << "Fetching Area..." << endl;
p1->printArea();
p2->printArea();
}

//Poly Class
void Poly::set(int w, int h) {
    width = w;
    height = h;
}

void Poly::printArea() {
    cout << "Area of our instance is: " << this->fetchArea() << endl;
}

//Triangle Class
int Triangle::fetchArea() {
    return (int)(0.5 * width * height);
}

//Rect Class
int Rect::fetchArea() {
    return width * height;
}
```

Now wait just a minute, you just said that we can't call a pure virtual function directly from the abstract class itself, right? Yes, that's correct but there's one little tricky thing about the **this** keyword. Remember, it exists as a **constant pointer TO THE CURRENT CLASS INSTANCE**. Therefore, by that definition when we're actually calling the **printArea** method in this example, we're directly referring to the pointer being a pointer to **Triangle** and **Rect**.

You can use this little trick whenever you have a pure virtual function that you want to directly call from inside the abstract class.

Final Remarks

And there you have it! Now you've completed your learning of all of the necessary C++ topics to dive head first into Torque itself and start writing your own custom code and internal engine functioning. There are a few new topics we'll cover in the next chapter, but you now should have a complete understanding of how to create classes, and work with objects in C++.

This is a very important concept to have mastery over when you're working in C++, so be sure to re-read any of the sections you may have had a little trouble understanding, and refer to the online references and C++ documentation if there is anything you're not getting.

Now that we've completed this chapter, let's move on to our final major topic of the guide!

Chapter 17: Classes and Objects for Torque

Chapter Introduction

And at long last, you have reached the culmination of your journey through the Ultimate Guide. This chapter is where it all comes together to teach you probably the most powerful tool available to you through C++ as it pertains to Torque3D, and that is the power of customizable engine object classes. This chapter will show you the ins and outs of creating a brand new object type in the engine, and then we'll bring back TorqueScript one last time to show you how to use your newly created custom objects in the engine.

This chapter brings back topics from all over the guide, mainly from Chapters 5 and on, so if there's anything you don't really think you know completely, be sure to go back and re-read the section again as this Chapter will primarily be focusing on creating new object types in the engine itself.

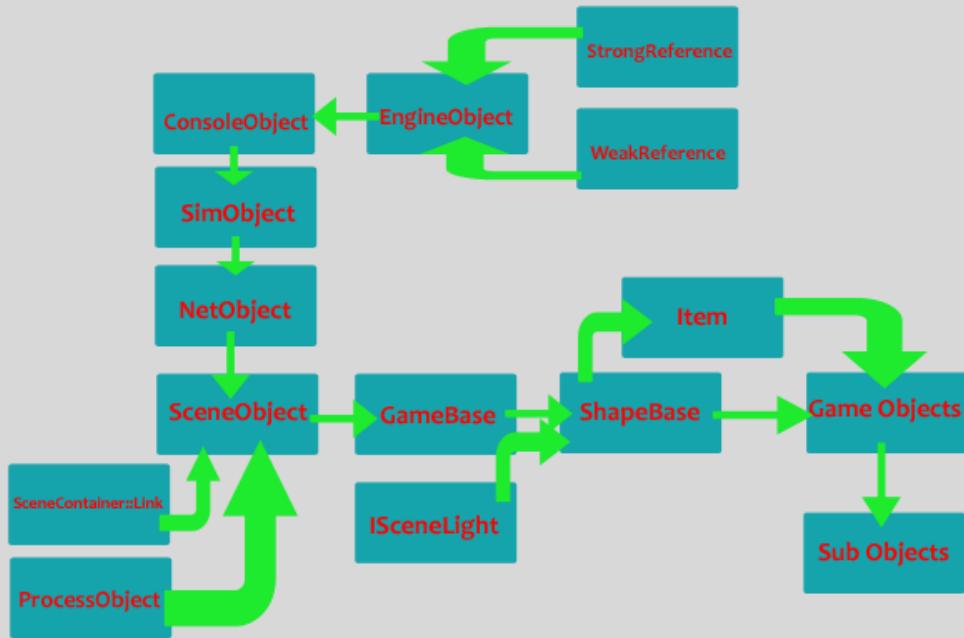
Also a fair warning, there will be no sample files in the folder for this chapter, since we'll mainly be building a new object from start to finish, by teaching you the internal tools available to you for the engine to do so. We're going to focus on the important concepts you'll need to get a new object instance working in the engine. This is where you'll actually get to create a brand new object that you can call your own and work with it in TorqueScript. We'll start by teaching how to actually get a new object into the engine itself, and then we'll cover the key important topics of objects as they pertain to the engine, which encompasses Networking, Rendering, and Collision.

So now that you know what's coming down the pipeline for this chapter, let's get started by teaching you how to create a brand new engine object and then show you how to create an instance of your new object in TorqueScript to be used as you need it.

Creating a new Object Type

So let's get started. You'll want to re-open your Torque3D engine solution for this chapter. Make sure you followed the steps back in Chapter 15 to set up a custom folder to place your new code in because we'll be using that folder for our object instance for this chapter.

Before we get started on the topic of actually coding a new object instance for the engine, I'm actually going to annoy your mind with an image that you've seen quite a few times now, because it's extremely important for the concept of this entire chapter:



Yep, there it is for the fifth time in this guide. However, now that you've learned about classes in C++, this diagram should actually make sense to you now. This is the inheritance tree of the primary class instances in the engine. Whenever you want to create a derivation object instance in the engine, this is the list of the classes from which your new class will have access to by means of inheritance. For the purposes of this chapter, we'll be working from the little box near the end that says **SceneObject**.

Creating a New Game Object: The Questions

Before we actually jump into the programming side of things, we want to quickly sit back and ask a few questions about our new object. First and foremost, what exactly **IS** this object? This is where you ask yourself what this object needs to do, and the kind of coding you'll need to put behind this object in order to get it to function. Secondly, you need to ask yourself what kind of functions and parameters of the object need to be exposed to TorqueScript in order to get it to function, remember all of those `%obj.blah` variables back in Chapter 8? This is where you need to ask yourself what those variables are, and how they will affect your object.

The last question you need to ask yourself is something you'll likely get better at doing with practice for creating objects, and that is the question of what functions and classes need to be inherited in order to get the new object to function correctly. That diagram above will help you to see how deep certain classes go when it comes to inherited functioning, but you also need to be careful because some of the parent classes have private access level sections which will not be available to you. Do your research prior to writing a single line of code, it will help you save time and prevent you from writing redundant code blocks.

Getting Started: Building an Object in C++ for Torque

So now that we've asked the questions, let's actually get started with our new object. For the purpose of this guide, I'm going to keep our example nice and easy. We'll create a simple box object that

has a modifiable color parameter. We'll also provide it collision so other objects will be able to interact with this box we make. We'll get to the coloring and collision stuff in a little while, but for now let's actually talk about the first steps to creating our new object.

As I said before, there won't be any example step files to this Chapter in the files folder, so make sure you're following along with each step as we progress through our object creation here.

When it comes to making anything new in C++, I usually start with the header file, it lets me write a list of template functions and set up the inheritance properties before I write the actual code. This also helps me pick out the header files I will need to include when creating the actual source code. So, instead of bantering away, let's actually start writing some code.

```
#ifndef EXAMPLEOBJECT_H
#define EXAMPLEOBJECT_H

#ifndef _SCENEOBJECT_H_
    #include "scene/sceneObject.h"
#endif

#endif
```

Here's how I start an object instance. Now, you might ask why I pick the SceneObject class when there are items further down the inheritance tree. The answer is easy for now. Because we're not worrying about things like rendering and collision just yet, we don't need to dive very deep into the tree. We just need to make sure the object can be networked and that part is covered. Also, you'll notice that we also have a check inclusion for sceneObject.h, this is to save on some executable space. We don't need to include the header twice.

Now that we have our first header instance picked out, let's expand on our code base and actually create our class instance:

```
#ifndef EXAMPLEOBJECT_H
#define EXAMPLEOBJECT_H

#ifndef _SCENEOBJECT_H_
    #include "scene/sceneObject.h"
#endif

class MyExampleObject : public SceneObject {
    typedef SceneObject Parent;
public:
    MyExampleObject();
    virtual ~MyExampleObject();

    //Add this object instance to the engine
    DECLARE_CONOBJECT(MyExampleObject);

    //Virtual method overrides
    static void initPersistFields();
    bool onAdd();
    void onRemove();
```

```
};  
  
#endif
```

And now you should be seeing some familiar sights from our previous chapter. As you can see, we're directly inheriting the `SceneObject` class into our new object class instance. The very first line inside the class itself is a shortcut type-definition that may prove to be very useful to your coding down the line, should you need to access any piece of code stored in the `SceneObject` instance. Most object definitions in the engine employ this shortcut, and there is no reason for you not to use it as well!

Next up, we have our standard constructor and destructor for objects in Torque 3D. We label the destructor as virtual so we don't need to write an override method for it in the actual C++ code. It's just there to be there, nothing more.

Following those two lines is a new macro for your dictionary of Torque macros in C++, which is the **DECLARE_CONOBJECT** macro. This macro is responsible for converting a C++ class name into a TorqueScript class name. This allows you do the following in TorqueScript:

```
function spawnMyObject() {  
  
    %newObj = new MyExampleObject()  
    class = "MyExampleObject";  
  
};  
  
}
```

Finally we have three methods that are overridden by our new class that exist as virtual functions within the `SceneObject` class. These are mainly scripting callbacks, and we'll cover those three in just a moment.

With the header file basics in place, let's now move over to the C++ side and talk about how to implement these few starting methods.

```
#include "platform/platform.h"  
#include "PGD/exampleObject.h"  
  
#include "math/mathIO.h"  
#include "scene/sceneRenderState.h"  
#include "console/consoleTypes.h"  
#include "core/stream/bitStream.h"  
#include "materials/materialManager.h"  
#include "materials/baseMatInstance.h"  
#include "renderInstance/renderPassManager.h"  
#include "lighting/lightQuery.h"  
#include "console/engineAPI.h"
```

So, as you can see. We start with quite a list of header files for our example object. Let's actually walk through these headers. First and foremost, you should automatically recognize `platform.h`, we talked a bit about this file back in Chapter 15. This file defines the properties and data types for the

engine and also keys in the preludes for the macro definitions. The second header we load in is the one we just made a bit ago. I called mine exampleObject.h and I put it inside the custom folder I created back in Chapter 15, again go back and recap that early section if you haven't done this already.

Following those two headers are the necessary C++ components we'll need as we work through the rest of this chapter. The first file is the math system in the engine. We'll use this mainly in calculation methods dealing with our object's world box and its properties. Following that is the scene render state manager, which as you probably guessed from the name is used for the rendering system. The console types header contains a few macro definitions that are used in conjunction with the custom defined engine types from platform.h. The bit stream header is used by the networking side of the engine and handles the data transmission stream that moves between clients and server. The next three files are the rendering pass and material headers which are used to actually handle the definition of the rendering of our particular object instance. Finally we have the lighting system's input header and the engine's application programming interface header, which defines our object macros.

There are thousands of available header files in the engine, just remember the prior section on choosing what headers you need based on what your object will need to know in order to function properly. Most of the time, there will be something in the engine that functions similarly to what you want to achieve. Start from that object, and build up the new object instance you want to work with.

Now that we have the framework headers in place, let's add the methods we've created to our new object's C++ file.

```
#include "platform/platform.h"
#include "PGD/exampleObject.h"

#include "math/mathIO.h"
#include "scene/sceneRenderState.h"
#include "console/consoleTypes.h"
#include "core/stream/bitStream.h"
#include "materials/materialManager.h"
#include "materials/baseMatInstance.h"
#include "renderInstance/renderPassManager.h"
#include "lighting/lightQuery.h"
#include "console/engineAPI.h"

MyExampleObject::MyExampleObject() {
    mTypeMask |= StaticObjectType | StaticShapeObjectType;
}

MyExampleObject::~MyExampleObject() {
}

void MyExampleObject::initPersistFields() {
    Parent::initPersistFields();
}

bool MyExampleObject::onAdd() {
    if(!Parent::onAdd())
        {
```

```
        return false;
    }
    resetWorldBox();
    addToScene();
    return true;
}

void MyExampleObject::onRemove() {
    removeFromScene();
    Parent::onRemove();
}
```

So we have a few things to talk about. First and foremost let's explain what we're doing with the constructor. Remember, we're directly inheriting our new object instance from the `SceneObject` class, which inherits from higher level engine class instances. If you go all the way back to Chapter 11 you may remember when we were talking about container searches, I introduced the concept of a **type mask**, which is the specific explanation of a certain object based on its properties.

The `SceneObject` class by default applies the **DefaultObjectType** to an object instance. All we did in the constructor was to apply the **StaticObjectType** and the **StaticShapeObjectType** masks to our object as well. This will be used when the object actually exists in the game world to explain certain properties of the object to the server and the client.

Since we do not define any memory based pointers used in our object instance, we don't need to do anything in our destructor, so that is left empty (We'll come back to this in a little while).

The **initPersistFields** method is used to define properties of the object that can be manipulated from inside the engine (TorqueScript), for now we don't have any object fields and all of the existing important fields are defined by the `SceneObject` class, therefore we can simply call the parent method and move on.

The **onAdd** method is a precursor internal call to the TorqueScript version of this method, what you should essentially do in this function is define any values you did not set in the constructor and then make the default object initialization calls (**resetWorldBox**, **addToScene**, etc), this topic is generally open minded, so feel free to do some research into other object implementations to get an idea of what needs to be initialized in this method.

And finally, we have our **onRemove** method, which is the precursor internal call to the TorqueScript version. De-initialization can usually be safely handled in the object destructor, however there may be some isolated cases where you may need to use this method. Be sure the object is removed from the scene a-la **removeFromScene**. If you're not sure if this call is made, go up the object inheritance tree to see if this call is made, otherwise you'll need to do it here.

At this point in time, you've successfully created a brand new object type that you can now use inside your engine. It won't do anything as it exists right now, and you can't network to other players with it, but this is the general framework of getting things going for this type of code. Now let's build on

your brand new object by expanding on our current code and preparing for the rendering phase of the object instance.

Object Fields, Implementing TorqueScript Variables

Creating object fields is another important part of creating a new object instance for the engine, this allows you to define internal properties of your object that the developer can manipulate to change the behavior of your object instance. As mentioned a little bit ago, we'll primarily be working with the **initPersistFields** method with this section, however we'll come back to this again in the object's networking when it comes to making the fields be manipulatable across the network stream.

As earlier mentioned, the goal of this section is a renderable box instance with a color field that can be modified on the fly by the user, so we're going to create a color parameter for our box instance. There are two ways you can build a field into an object instance, and it deals with the **access level** setting of the parameter. If you define the parameter to be **public** or **private**, then you can use the generalized **addField** method, however if you define the field to be **protected**, then you must use the **addProtectedField** method, let's show examples of each to demonstrate how this is done.

First let's start with the basic case of the public/private field instance. This is the most typical case of a field instance, however if you plan on making object instances that can be inherited from then you may find yourself using protected. We'll start with the header:

```
#ifndef EXAMPLEOBJECT_H
#define EXAMPLEOBJECT_H

#ifndef _SCENEOBJECT_H_
    #include "scene/sceneObject.h"
#endif

class MyExampleObject : public SceneObject {
    typedef SceneObject Parent;
public:
    MyExampleObject();
    virtual ~MyExampleObject();

    //Add this object instance to the engine
    DECLARE_CONOBJECT(MyExampleObject);

    //Virtual method overrides
    static void initPersistFields();
    bool onAdd();
    void onRemove();

private:
    ColorF mColorMod;
};

#endif
```

The only change we made to the header was to add the new field to the private section of the class instance. For your own personal reference, the type **ColorF**, is an internally defined Torque class instance that defines a 0.0 to 1.0 floating point color scale for RGBa.

A naming convention used in the engine is to prepend the name of all internal C++ fields with the letter ‘m’. Whether or not you follow this convention, is your call. Now let’s look at the source changes we made:

```
#include "platform/platform.h"
#include "PGD/exampleObject.h"

#include "math/mathIO.h"
#include "scene/sceneRenderState.h"
#include "console/consoleTypes.h"
#include "core/stream/bitStream.h"
#include "materials/materialManager.h"
#include "materials/baseMatInstance.h"
#include "renderInstance/renderPassManager.h"
#include "lighting/lightQuery.h"
#include "console/engineAPI.h"

MyExampleObject::MyExampleObject() {
    mTypeMask |= StaticObjectType | StaticShapeObjectType;
    mColorMod.set(1.0f, 1.0f, 1.0f, 1.0f);
}

MyExampleObject::~MyExampleObject() {

}

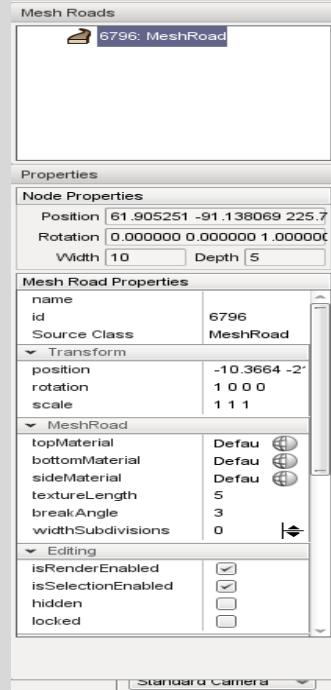
void MyExampleObject::initPersistFields() {
    addGroup( "Rendering" );
   addField( "colorMod", TypeColorF, Offset( mColorMod, MyExampleObject ),
        "The color modulation of the mesh instance." );
    endGroup( "Rendering" );

    Parent::initPersistFields();
}

bool MyExampleObject::onAdd() {
    if(!Parent::onAdd()) {
        return false;
    }
    resetWorldBox();
    addToScene();
    return true;
}

void MyExampleObject::onRemove() {
    removeFromScene();
    Parent::onRemove();
}
```

You'll see that the only changes we made were in the constructor and the `initPersistFields` method. The constructor pre-defines the color modulation we created to a pure white color. Now, let's look at our changes to the `fields` method. First and foremost, when you create object fields, you can use **groups** to separate your newly defined fields into visible categories for the editor.



For a reminder of what those fields are, let's take a peek all the way back at the World Editor chapter (Chapter 3) and revisit a picture from that chapter. You'll see under the **Mesh Road Properties** section there are dropdown boxes named **Transform**, **MeshRoad**, and **Editing** that are visible.

These instances are actually the groups that we're talking about above when it comes to defining your own custom fields. When it comes to defining an object instance in C++, it will place them in the order in which they are defined, so when you see in the code above that we define our custom group and fields first, they will be visible first in the dropdown list, followed by the groups and fields that are defined in the parent class, **SceneObject**, which will then be followed by the fields defined in the remainder of the object inheritance tree up to the point where the very first instance of the `initPersistFields` method is actually defined in the C++ code instance.

Keep this trick in mind when creating your new objects, it will help you when your code and fields are more organized together.

Now let's actually look at the method that is responsible for mapping the internal C++ field to a TorqueScript version, the **addField** method. This method has four parameters that need to be set when called. The first is a string instance containing the desired TorqueScript name of the field. The second is a flag instance telling the engine what the field must contain. For your own reference, here is a list of the flags and their description:

Flag	Description	C++ Data Type
TypeString	A basic string instance	const char *
TypeCaseString	A case-sensitive string instance	const char *
TypeRealString	A pure String instance	Torque::String
TypeCommand	A string instance that maps to a command	Torque::String
TypeFilename	A string instance that maps to a file path	Torque::String
TypeStringFilename	A string instance to a relative or absolute path filename instance	Torque::String
TypePrefabFilename	A string instance that maps to a Torque Prefab file	Torque::String
TypeImageFilename	A string instance that maps to a image file	Torque::String
TypeShapeFilename	A string instance that maps to a shape instance file	Torque::String
TypeS8	A single signed character instance	S8 (char)
TypeS32	A signed integer instance	S32 (int)

TypeS32Vector	A vector of integers	Torque::Vector<S32>
TypeF32	A signed floating point number	F32 (float)
TypeF32Vector	A vector of floating point numbers	Torque::Vector<F32>
TypeBool	A boolean variable	bool
TypeBoolVector	A vector of booleans	Torque::Vector<bool>
TypeFlag	A Torque Flag instance (numerical 1 = true, 0 = false)	S32 (int)
TypeColorF	A color instance RGBa (0.0 – 1.0)	Torque::ColorF
TypeColorI	A color instance RGBa (0 – 255)	Torque::ColorI
TypeSimObjectName	A mapping to a SimObject instance	Torque::SimObject *
TypeParticleParameterString	A string instance used for the particle system, you shouldn't use this.	const char *
TypeMaterialName	A string mapping to a material instance	Torque::String
TypeTerrainMaterialIndex	A numerical mapping used by the engine's terrain system, you shouldn't use this.	S32 (int)
TypeTerrainMaterialName	A string mapping used by the engine's terrain system, you shouldn't use this.	Torque::String
TypeCubemapName	A string instance used to map to a cubemap instance, used for rendering	Torque::String
TypeRectUV	A internal class representation of a rectangle instance (uses X, Y, W, H)	Torque::RectF
TypeUUID	A internal representation flag for the engine's UUID system	Torque::UUID
TypePID	A internal representation flag for the engine's PID system	Torque::SimPersistID *
TypeSimPersistID	Duplicate of the TypePID	Torque::SimPersistID *

This list of flags is contained within the **ConsoleTypes.h** header, in the event you need to add more to the engine to get a custom feature working, that's where you need to look.

The third field in the **addField** method is an **offset** which is a bitwise representation of multiple fields compacted into one single readable field. In the case of our method here, it's an offset of the name of the C++ internal field to the C++ internal class name. Finally, the last field for our method contains a user written description of the newly defined field that will be visible to the editor when a user clicks on the field.

Always remember though that if you're making fields and they are contained within a group instance that you **MUST** call the **endGroup** function before you move on, or end the function.

Now that you have a working example of a public or a private object field instance, let's look at the protected field instance. Again we'll start with the header of the code:

```
#ifndef EXAMPLEOBJECT_H
#define EXAMPLEOBJECT_H

#ifndef _SCENEOBJECT_H_
#include "scene/sceneObject.h"
#endif
```

```
class MyExampleObject : public SceneObject {
    typedef SceneObject Parent;
public:
    MyExampleObject();
    virtual ~MyExampleObject();

    //Add this object instance to the engine
    DECLARE_CONOBJECT(MyExampleObject);

    //Virtual method overrides
    static void initPersistFields();
    bool onAdd();
    void onRemove();

    void setColorMod(ColorF col) { mColorMod = col; }

protected:
    ColorF mColorMod;

private:
    static bool _setColorMod(void *object, const char *index, const
char *data);
};

#endif
```

Nothing major at all changed here, all we did was make our field fit into the protected section of the class. However, you'll notice something new this time around. We've also had to create a new function definition under the private section of our class instance. We also created a header defined method to update the current color modulation variable. Why we need to do this is the internal definition of the **addProtectedField** method, which we'll talk about in just a moment after you see the updated C++ file:

```
#include "platform/platform.h"
#include "PGD/exampleObject.h"

#include "math/mathIO.h"
#include "scene/sceneRenderState.h"
#include "console/consoleTypes.h"
#include "core/stream/bitStream.h"
#include "materials/materialManager.h"
#include "materials/baseMatInstance.h"
#include "renderInstance/renderPassManager.h"
#include "lighting/lightQuery.h"
#include "console/engineAPI.h"

MyExampleObject::MyExampleObject() {
    mTypeMask |= StaticObjectType | StaticShapeObjectType;
    mColorMod.set(1.0f, 1.0f, 1.0f, 1.0f);
}

MyExampleObject::~MyExampleObject() {
```

```
void MyExampleObject::initPersistFields() {
    addGroup( "Rendering" );
    addProtectedField("colorMod", TypeColorF, Offset(mColorMod,
MyExampleObject), &_setColorMod, &defaultProtectedGetFn,
        "The color modulation of the mesh instance" );
    endGroup( "Rendering" );

    Parent::initPersistFields();
}

bool MyExampleObject::onAdd() {
    if (!Parent::onAdd()) {
        return false;
    }
    resetWorldBox();
    addToScene();
    return true;
}

void MyExampleObject::onRemove() {
    removeFromScene();
    Parent::onRemove();
}

bool MyExampleObject::_setColorMod(void *object, const char *index, const
char *data) {
    MyExampleObject* so = static_cast<MyExampleObject*>(object);
    if (so) {
        ColorF col;
        Con::setData(TypeColorF, &col, 0, 1, &data);
        so->setColorMod(col);
    }
    return false;
}
```

Feel free to read over the newly added **_setColorMod** function, however it's just a simple function which fetches the object instance and calls the header written function, so we're primarily going to focus on the **addProtectedField** function call. The first three variables match exactly what we've already done before with **addField**, the changes come into play afterwards. You'll see a new notation the likes of which, you haven't seen yet. This is actually a **pointer to a function**, or more commonly used as a **callback function**.

We have two of these instances in the **addProtectedField** function declaration. The first call contains the set method for the custom field, and the second one contains the read method for the custom field. 95% of the time you can use the internal Torque defined **defaultProtectedGetFn** call to handle this, the compiler will tell you otherwise in which case a basic function returning the desired variable will suffice. Finally we have the description again.

Hopefully now, you're starting to see how Torque defined functions, and macros come together with everything you've learned in the prior chapters for the process of creating custom objects inside

the engine. This notion will continue through the remainder of this chapter as we continue to build upon our object example here.

So now that you've successfully created a basic object, let's actually let your object exist in a multiplayer environment by introducing the concept of networking an object instance for the engine.

Networking Your Object

When it comes to making a game engine, one of the things you're likely going to want to knock out sooner rather than later is the networking side of the engine, even if you're making a single player game because wiring everything up later requires a ton of effort to complete. As we've completed the basics of our new object, we're going to follow the exact same notion by wiring our new object up to the Torque networking system to allow other clients to actively see how changes affect our object.

Before we get started with this section however, I want to teach you one more C++ tool at your disposal, and this one will prove to be extremely useful in the development of class instances.

Enumerations

One of the last C++ tools I'm going to teach you is the C++ concept of **enumerations**. To this point in time, you've been using explicitly defined data types in both TorqueScript and C++, where all of the data types were either defined in concept by the engine, or by the compiler. An enumeration however defines a completely custom definition of data and the values that it may contain.

Enumerations are most commonly used as a tool for flagging certain fields as they can contain multiple definitions under one type instance.

To create an enumeration, you use the **enum** keyword in C++, followed by a list of data and their respective value set. If you do not provide a value set, the compiler will automatically assign numerical values to your data.

Here's a basic example of an enumeration in action from the perspective of a class instance, however you are free to use them anywhere in your code so long as they maintain scope:

MyFile.h

```
#include <iostream>
using namespace std;

#ifndef _MYFILE_H_
#define _MYFILE_H_

class MyClass {
public:
    enum Colors {
        RED = 0,
        GREEN = 1,
        BLUE = 2,
    };

    void set(Colors c);
```

```
        Colors fetch();
        char *toText(Colors x);

    private:
        Colors mC;
};

#endif // _MYFILE_H_
```

MyFile.cpp

```
#include "MyFile.h"

void main() {
    MyClass M;
    M.set(MyClass::RED);

    cout << "Color is: " << M.toText(M.fetch()) << endl;
}

//MyClass
void MyClass::set(MyClass::Colors c) {
    mC = c;
}

MyClass::Colors MyClass::fetch() {
    return mC;
}

char *MyClass::toText(Colors x) {
    char *res = new char[16];
    switch(x) {
        case RED:
            res = "Red";
            break;

        case GREEN:
            res = "Green";
            break;

        case BLUE:
            res = "Blue";
            break;
    }
    return res;
}
```

You can see that in our above example we define a custom enumerated field named Colors with a few options that can be used by the code. Our main method applies the color to the class instance and then prints out a text version of the color by retrieving the internal value that we stored in our class variable.

Enumerations are very easy, yet very powerful tools that you have at your disposal in C++. Whenever you have a class instance where flagged variables are possible, you may want to consider

using enumerations as a route to prevent users from setting unknown or invalid flags on your code, or if you want to define custom type information, you can achieve this through the enumeration tool as well. Now let's get back to the topic at hand.

Getting your new Object "Network Ready"

The reason I introduced the topic of enumerations before starting is because the very first change to our code will actually apply an enumerated variable to our class instance. To get an object "Network Ready" as it pertains to the engine, we need to make use of the topics we talked about back in Chapter 12, which are scoping and ghosting.

If you remember all the way back there, I told you that the engine flags objects that need to be updated to prevent unnecessary transfers over the network to conserve bandwidth and keep speeds running smoothly. These individual scoping flags as you will learn here, can be customized by you to tell the network transmission stream to only update certain aspects of the object as they are needed.

Let's start by updating our header file to contain the new lines of code to prepare our object for the networking of the engine:

```
#ifndef EXAMPLEOBJECT_H
#define EXAMPLEOBJECT_H

#ifndef _SCENEOBJECT_H_
#include "scene/sceneObject.h"
#endif

class MyExampleObject : public SceneObject {
    typedef SceneObject Parent;
    enum MaskBits {
        TransformMask = Parent::NextFreeMask << 0,
        UpdateMask    = Parent::NextFreeMask << 1,
        NextFreeMask  = Parent::NextFreeMask << 2
    };

public:
    MyExampleObject();
    virtual ~MyExampleObject();

    //Add this object instance to the engine
DECLARE_CONOBJECT(MyExampleObject);

    //Virtual method overrides
    static void initPersistFields();
    virtual void inspectPostApply();
    bool onAdd();
    void onRemove();

    //Custom Method
    void setTransform(const MatrixF &mat);

    //Networking
    U32 packUpdate(NetConnection *conn, U32 mask, BitStream *stream);
    void unpackUpdate(NetConnection *conn, BitStream *stream);
```

```
    private:  
        ColorF mColorMod;  
};  
  
#endif
```

So as you can see, we started by creating an enumeration called **MaskBits**. If you actually go and read the SceneObject class instance, you'll see that this class also defines an enumeration called MaskBits. The special thing about enumerations is that instead of being blocked from defining a new version on the derived class, it will actually extend the current enumeration through the new class, so we'll have access to both the enumerated variables of our defined instance here, as well as the enumerations set in any parent classes. As for the enumerations themselves, we define a **TransformMask** to handle updates to the object position and rotation as well as an **UpdateMask** to handle updates to our custom fields on the object (Color). The **NextFreeMask** is used on the parent class to define the next open bit per the enumeration behavior, we follow this standard here in the event you wish to create a derivation from this class.

You'll also notice a new virtual method has been added to our list. The **inspectPostApply** method is used here when the object's custom fields are updated. We'll use this method to send a message to the network to ask for an update.

Next up, we added a **setTransform** method to our object. This will allow the user to move the object in the game world (We'll network that property to show you how this is done). And finally we added two more methods to our object that are defined as virtual by means of the NetObject class which are **packUpdate** and **unpackUpdate**. We'll get to these in just a minute.

Now let's take a peek at the source file to see what changes we need to make on that end to get ready for networking.

```
#include "platform/platform.h"  
#include "PGD/exampleObject.h"  
  
#include "math/mathIO.h"  
#include "scene/sceneRenderState.h"  
#include "console/consoleTypes.h"  
#include "core/stream/bitStream.h"  
#include "materials/materialManager.h"  
#include "materials/baseMatInstance.h"  
#include "renderInstance/renderPassManager.h"  
#include "lighting/lightQuery.h"  
#include "console/engineAPI.h"  
  
MyExampleObject::MyExampleObject() {  
    mNetFlags.set( Ghostable | ScopeAlways );  
    mTypeMask |= StaticObjectType | StaticShapeObjectType;  
    mColorMod.set(1.0f, 1.0f, 1.0f, 1.0f);  
}  
  
MyExampleObject::~MyExampleObject() {
```

```

}

void MyExampleObject::initPersistFields() {
    addGroup( "Rendering" );
    addField( "colorMod", TypeColorF, Offset( mColorMod, MyExampleObject ),
        "The color modulation of the mesh instance." );
    endGroup( "Rendering" );

    Parent::initPersistFields();
}

void MyExampleObject::inspectPostApply() {
    Parent::inspectPostApply();
    setMaskBits(UpdateMask);
}

bool MyExampleObject::onAdd() {
    if(!Parent::onAdd()) {
        return false;
    }
    resetWorldBox();
    addToScene();
    return true;
}

void MyExampleObject::onRemove() {
    removeFromScene();
    Parent::onRemove();
}

void MyExampleObject::setTransform(const MatrixF & mat) {
    Parent::setTransform(mat);
    setMaskBits(TransformMask);
}

U32 MyExampleObject::packUpdate(NetConnection *conn, U32 mask, BitStream
*stream) {

}

void MyExampleObject::unpackUpdate(NetConnection *conn, BitStream *stream) {
}

```

So there's actually quite a bit of stuff going on here that has been changed. First and foremost, you'll notice that we apply some data to **mNetFlags** in our constructor. This is used to identify the object's scoping rules to the engine.

Next up is our newly added **inspectPostApply** method. As I mentioned in the header text, this is a method defined virtually on the SceneObject class, but I held off for now to show you the network flag update here. The command of interest in our case is the **setMaskBits** command. This is used to update the current **MaskBits** of the object. Remember those enumerations we just got done defining? This is where they come into play. When you use the **setMaskBits** command, the networking side of the engine

picks up on that and then runs an instance of it before disabling the flag for the next cycle of the engine. I did the same thing in the **setTransform** method to demonstrate how this works.

Both of these methods are already defined on the class instance above, but we'll catch them in our custom class for the purposes of the guide.

Finally we have two methods which, hold on a second. They're empty. And I did this on purpose, because before we talk about the pack and unpack methods, we need to introduce a brand new topic, so let's get that out of the way.

BitStreams, Introduction to Networking

The thing that's keeping us from a perfectly working networked object instance is a little class called the BitStream. A BitStream is a streamlined class instance that derives in Torque from a Stream class instance. This class is the so called "Network Transmission Stream" that we've been talking about since Chapter 12.

The BitStream class can take in data from a class instance and stream it across another class called a NetConnection, which is an individual client instance, which you may also recall from Chapter 12. Both of these classes interact together to form the Torque networking system. So now let's revisit our code, but actually put in the pieces we took out:

```
#include "platform/platform.h"
#include "PGD/exampleObject.h"

#include "math/mathIO.h"
#include "scene/sceneRenderState.h"
#include "console/consoleTypes.h"
#include "core/stream/bitStream.h"
#include "materials/materialManager.h"
#include "materials/baseMatInstance.h"
#include "renderInstance/renderPassManager.h"
#include "lighting/lightQuery.h"
#include "console/engineAPI.h"

MyExampleObject::MyExampleObject() {
    mNetFlags.set( Ghostable | ScopeAlways );
    mTypeMask |= StaticObjectType | StaticShapeObjectType;
    mColorMod.set(1.0f, 1.0f, 1.0f, 1.0f);
}

MyExampleObject::~MyExampleObject() {

}

void MyExampleObject::initPersistFields() {
    addGroup( "Rendering" );
   addField( "colorMod", TypeColorF, Offset( mColorMod, MyExampleObject ),
        "The color modulation of the mesh instance." );
    endGroup( "Rendering" );

    Parent::initPersistFields();
}
```

```
void MyExampleObject::inspectPostApply() {
    Parent::inspectPostApply();
    setMaskBits(UpdateMask);
}

bool MyExampleObject::onAdd() {
    if(!Parent::onAdd()) {
        return false;
    }
    resetWorldBox();
    addToScene();
    return true;
}

void MyExampleObject::onRemove() {
    removeFromScene();
    Parent::onRemove();
}

void MyExampleObject::setTransform(const MatrixF & mat) {
    Parent::setTransform(mat);
    setMaskBits(TransformMask);
}

U32 MyExampleObject::packUpdate(NetConnection *conn, U32 mask, BitStream
*stream) {
    //First, allow the parent classes the write access.
    U32 retMask = Parent::packUpdate(conn, mask, stream);
    //TransformMask
    if(stream->writeFlag(mask & TransformMask)) {
        mathWrite(*stream, getTransform());
        mathWrite(*stream, getScale());
    }
    if(stream->writeFlag(mask & UpdateMask)) {
        stream->writeFloat(mColorMod.red, 8);
        stream->writeFloat(mColorMod.green, 8);
        stream->writeFloat(mColorMod.blue, 8);
        stream->writeFloat(mColorMod.alpha, 8);
    }
    return retMask;
}

void MyExampleObject::unpackUpdate(NetConnection *conn, BitStream *stream) {
    //Have the parent read method complete first
    Parent::unpackUpdate(conn, stream);
    //TransformMask
    if (stream->readFlag()) {
        mathRead(*stream, &mObjToWorld);
        mathRead(*stream, &mObjScale);
        setTransform( mObjToWorld );
    }
    //UpdateMask
    if (stream->readFlag()) {
        mColorMod.red = stream->readFloat(8);
        mColorMod.green = stream->readFloat(8);
        mColorMod.blue = stream->readFloat(8);
    }
}
```

```
    mColorMod.alpha = stream->readFloat(8);  
}  
}
```

So those two methods actually contain quite a bit of code, and with a very good reason. Remember we only want to transmit information that we need to transmit across, and the smaller we can transmit across, the better.

This can be a really tough task to actually gain mastery over when you're first starting out with networking object instances, but you'll get better and more performance friendly as you move along. When you use the **packUpdate** method, this is the process of the server sending down the information of the object in its current state to a client instance.

We start this process by allowing any parent classes, such as SceneObject to handle any information updates that it needs to write to the networking stream before we actually start writing our own. You'll notice afterwards we use an if flag to test the mask if it's active in this current run through and if it is indeed active, to write it in a flag form to the networking stream. This is the standard way of doing things, and it works wonders so stick with it.

When you write complex mathematical classes such as boxes, or rotational data, you'll usually want to use the **mathWrite** method that is included with the **BitStream** class instance. When you write standard C++ data types, or basic information that can be compacted down to a standard C/C++ type, use the BitStream's internal write methods. To determine the size you need to write, that's the tough challenge you'll face. You want to make sure you cover the maximum value the information can contain, but you don't want to go overboard otherwise you're wasting bandwidth and performance.

Use the C++ data type size table back in Chapter 13 to help you determine what the absolute maximum these numbers should be to get an idea of where to start.

Next up is the **unpackUpdate** method provided to you by the NetObject class. This method is called on the client end when it receives a network update to store the information on its ghost copy of an object instance (We'll get to revisiting Torque's ghosts in a bit). **IT'S EXTREMELY IMPORTANT THAT WHEN YOU UNPACK INFORMATION, THAT IT MATCHES EXACTLY WHAT IS PACKED FROM THE PACKUPDATE() METHOD, FAILURE TO DO SO WILL CAUSE CRASHING!!!** Warning aside, what this means is that when you write something to be packed in the **packUpdate** method, then you must also have a read counterpart in the **unpackUpdate** method. To see how this is done in the code, read it above, you'll notice that we unload the parent first (because we packed the parent first), and then we test for the flag that we set if the update was needed for the **TransformMask**. If this is the case, we then call the **mathRead** methods to unpack the math classes. If we need to run through the **UpdateMask**, we use the standard BitStream read methods, and use the same size we packed it as to read it out from.

Again, this can be quite a challenge to overcome, especially if you're new to this whole process, but once you start doing this more often you'll get the hang of it very quickly.

And in this point in time, you now have a custom object instance that can be spawned in your game world, and even exist in multiplayer environments with networked data. But there's just one little thing about our object that we're missing, and that would be the visual aspect of it. So why don't we fix that problem now.

Rendering in C++

Our next stop in the guide is to talk about the visual side of your objects. Mainly how to go about rendering your object instances. The overall topic of rendering as it relates to game engines is extremely massive in scope and far beyond the range of one guide alone. Instead, we're going to focus primarily on the key topics of how to render your object instances in the engine, we'll cover a few great examples of rendering and provide code samples to get those moving in the right direction for you.

Before we move on through this part of the chapter, I highly recommend you have a strong foundation with Chapter 9, which is the included chapter on video game mathematics, as rendering can sometimes require complex mathematical operations, and a strong basis on C++ class functioning as there are multiple ways to accomplish the end goal using a few of the internal tools provided to you.

There are three topics of rendering we're going to cover in this guide. For further learning, I advise you to go through the GarageGames forums, particularly covering the older engines as they use more advanced rendering techniques. The three types of rendering we're going to discuss are **Shape Rendering** which is the rendering of a loaded model instance, **Mesh Rendering** which is the rendering of a mathematically defined geometrical shape instance, and **Beam Rendering** which is used to render rectangular beams.

Shape Rendering: Model to Game

Our specific class instance won't actually use this particular technique, but if you would like to replace our colored box with a model instance (Although at that point you could just as easily use a **StaticShape** instance, because it would be the same) then this section of the guide is for you.

To actually get a shape instance to render we need to adjust our class a little, mainly to add some new data to store to our header class, as well as our source code. For the actual rendering methods themselves, we'll get there in just a bit.

For each of these rendering systems, we need a quick and basic understanding of how Torque renders a scene in your instance. For the overall perspective of the rendering system, the first step is the binding of standard library functions into engine parsable functions. These standard libraries are DirectX and OpenGL (However, full GL support is current unsupported). From that point, the engine converts these functions into an internal graphics library (called gfx) which is used to define standard rendering tools and functions.

These functions are then used in a few instances within the engine to establish what is called the render pass manager. This class handles the calls to the individual rendering instances to draw the scene to the client instance. The pass manager itself doesn't actually render the scene, however it establishes

the calls needed to perform the rendering to the scene. We'll take advantage of this setup to create our own rendering instances for our custom objects.

So now that you know how that works, let's talk about shape rendering, or what is commonly referred to in engine lingo as TS (Three Space) rendering. To render an instance of this form you need to store the shape's file on your class instance so the engine can actually load the instance to be used. To handle the shape where we don't need to load the file for each tick we'll store two shape instances on the object as well. One instance will be responsible for actually handling the load process when a file name is parsed, and the other will store the generated shape object instance on the object to be referenced with each rendering pass.

```
#ifndef EXAMPLEOBJECT_H
#define EXAMPLEOBJECT_H

#ifndef _SCENEOBJECT_H_
    #include "scene/sceneObject.h"
#endif
#ifndef _TSSHAPEINSTANCE_H_
    #include "ts/tsShapeInstance.h"
#endif

class MyExampleObject : public SceneObject {
    typedef SceneObject Parent;
    enum MaskBits {
        TransformMask = Parent::NextFreeMask << 0,
        UpdateMask    = Parent::NextFreeMask << 1,
        NextFreeMask   = Parent::NextFreeMask << 2
    };
public:
    MyExampleObject();
    virtual ~MyExampleObject();

    //Add this object instance to the engine
    DECLARE_CONOBJECT(MyExampleObject);

    //Virtual method overrides
    static void initPersistFields();
    virtual void inspectPostApply();
    bool onAdd();
    void onRemove();

    //Custom Method
    void setTransform(const MatrixF &mat);

    //Networking
    U32 packUpdate(NetConnection *conn, U32 mask, BitStream *stream);
    void unpackUpdate(NetConnection *conn, BitStream *stream);

    //Rendering
    void createShape();
    void prepRenderImage(SceneRenderState *state);

private:
```

```
        String mShapeFile;
        TSShapeInstance *mShapeInst;
        Resource<TSShape> *mShape;
    };

#endif
```

So let's have a look at what we changed here. First of all, we added a new header file to be loaded alongside the scene object header. This file loads in additional headers as needed by the Three Space object rendering system, including the render pass manager handles.

You'll notice that we added two methods under a rendering section. The `createShape` method will be used to actually perform the steps needed to test if a model file needs to be loaded, and actually load the instance into our object itself. The `prepRenderImage` function is a virtual function which actually works with the render pass manager to handle rendering of an object instance.

We've removed our color modulation field as we will not need it for a shape rendering pass, and in its place have added three new fields. Again, I mentioned what these variables will be used for in the paragraph prior to the code.

Now let's look at the source code, just a fair warning here, it's quite long so be sure to read carefully:

```
#include "platform/platform.h"
#include "PGD/exampleObject.h"

#include "math/mathIO.h"
#include "scene/sceneRenderState.h"
#include "console/consoleTypes.h"
#include "core/stream/bitStream.h"
#include "materials/materialManager.h"
#include "materials/baseMatInstance.h"
#include "renderInstance/renderPassManager.h"
#include "lighting/lightQuery.h"
#include "console/engineAPI.h"

MyExampleObject::MyExampleObject() {
    mNetFlags.set( Ghostable | ScopeAlways );
    mTypeMask |= StaticObjectType | StaticShapeObjectType;
    mShapeInst = NULL;
}

MyExampleObject::~MyExampleObject() {
}

void MyExampleObject::initPersistFields() {
    addGroup( "Rendering" );
    addField( "shapeFile", TypeStringFilename, Offset( mShapeFile,
MyExampleObject ),
        "The file path to the model instance we would like to render." );
    endGroup( "Rendering" );
}
```

```
    Parent::initPersistFields();  
}  
  
void MyExampleObject::inspectPostApply() {  
    Parent::inspectPostApply();  
    setMaskBits(UpdateMask);  
}  
  
bool MyExampleObject::onAdd() {  
    if(!Parent::onAdd()) {  
        return false;  
    }  
    resetWorldBox();  
    addToScene();  
    createShape();  
    return true;  
}  
  
void MyExampleObject::onRemove() {  
    removeFromScene();  
    if(mShapeInstance) {  
        SAFE_DELETE(mShapeInstance);  
    }  
    Parent::onRemove();  
}  
  
void MyExampleObject::setTransform(const MatrixF & mat){  
    Parent::setTransform(mat);  
    setMaskBits(TransformMask);  
}  
  
U32 MyExampleObject::packUpdate(NetConnection *conn, U32 mask, BitStream  
*stream) {  
    //First, allow the parent classes the write access.  
    U32 retMask = Parent::packUpdate(conn, mask, stream);  
    //TransformMask  
    if(stream->writeFlag(mask & TransformMask)) {  
        mathWrite(*stream, getTransform());  
        mathWrite(*stream, getScale());  
    }  
    //UpdateMask  
    if(stream->writeFlag(mask & UpdateMask)) {  
        stream->write(mShapeFile);  
        //Create the shape instance on the server end  
        createShape();  
    }  
    return retMask;  
}  
  
void MyExampleObject::unpackUpdate(NetConnection *conn, BitStream *stream) {  
    //Have the parent read method complete first  
    Parent::unpackUpdate(conn, stream);  
    //TransformMask  
    if (stream->readFlag()) {  
        mathRead(*stream, &mObjToWorld);  
        mathRead(*stream, &mObjScale);  
        setTransform( mObjToWorld );  
    }
```

```
        }
    //UpdateMask
    if (stream->readFlag()) {
        stream->read(&mShapeFile);
        if(isProperlyAdded()) {
            createShape();
        }
    }
}

void MyExampleObject::createShape() {
    if(mShapeFile.isEmpty()) {
        return;
    }
    if (mShapeInstance && mShapeFile.equal(mShape.getPath().getFullPath(),
String::NoCase)) {
        //We don't need to reload the same file if we already have one.
        return;
    }
    if(mShapeInstance) {
        SAFE_DELETE(mShapeInstance);
    }
    mShape = NULL;
    mShape = ResourceManager::get().load(mShapeFile);
    if(!mShape) {
        //Did loading fail?
        Con::errorf("MyExampleObject::createShape() - Unable to load shape:
%s", mShapeFile.c_str());
        return;
    }
    if(isClientObject() && !mShape->preloadMaterialList(mShape.getPath()) &&
NetConnection::filesWereDownloaded()) {
        //Load the materials, if possible...
        mShape = NULL;
        return;
    }
    //Finish up...
    resetWorldBox();
    setRenderTransform(mObjToWorld);
    mShapeInstance = new TSShapeInstance(mShape, isClientObject());
}

void MyExampleObject::prepRenderImage(SceneRenderState *state) {
    if (!mShapeInstance) {
        //No shape? Stop us dead here...
        return;
    }
    //Handle camera distance calculations for the RPM
    Point3F cameraOffset;
    getRenderTransform().getColumn(3, &cameraOffset);
    cameraOffset -= state->getDiffuseCameraPosition();
    F32 dist = cameraOffset.len();
    if (dist < 0.01f) {
        dist = 0.01f;
    }
    //Set up the LOD for the shape and verify it.
```

```
F32 invScale = (1.0f / getMax(getMax(mObjScale.x, mObjScale.y),  
mObjScale.z));  
mShapeInstance->setDetailFromDistance(state, dist * invScale);  
    if (mShapeInstance->getCurrentDetail() < 0) {  
        return;  
    }  
    GFXTransformSaver saver;  
    //set up the render state  
    TSRenderState rdata;  
    rdata.setSceneState(state);  
    rdata.setFadeOverride(1.0f);  
    //Set up lights  
    LightQuery query;  
    query.init(getWorldSphere());  
    rdata.setLightQuery(&query);  
    //Calculate and set up rendering transforms  
    MatrixXF mat = getRenderTransform();  
    mat.scale(mObjScale);  
    GFX->setWorldMatrix(mat);  
    //Grant animation access  
    mShapeInstance->animate();  
    //And render...  
    mShapeInstance->render(rdata);  
}
```

So there is a ton of stuff going on here, so let's break it down piece by piece so you understand how this all works. First and foremost, note the adjustments from our prior code to the new code so you can see how we changed the color modulation field to be replaced with the three new fields. We also had to establish a NULL to one of the pointers when the instance is first spawned to prevent premature calls to the render method.

As for the calls we make to the `createShape` function we added, we only do so when the shape file variable is populated to make sure we don't call it unnecessarily. The rest of the code itself remains unchanged aside from the two new methods we added, so let's talk about those two now.

The first function we added was called `createShape`. This is a custom method we wrote for our object (not virtual) to handle the loading of the model's shape file into a resource parameter which is then used to create a shape instance that the engine recognizes to be rendered. We start this function with a few checks, first of all to make sure there is a file we want to load, and secondly to make sure we're not repeating a load call on the same file. Once this is validated we clean up the old resources to make way for the new ones. From there we perform some tests to make sure the shape loaded, and that we can safely preload the object's materials if they exist. If all of these tests were successful, we can then perform some cleanup operations and actually use our shape instance.

You'll notice whenever you do rendering in the engine that a recurring theme will be one or more preparation functions, next to a virtual function in one of the parent classes, usually `prepRenderImage`. Now let's actually talk about this function and explain its role in the rendering process.

The prepRenderImage function is defined on the SceneObject class, and this function is triggered by the object instance when the engine sends a signal to the object asking for the render states on this object to actually render to the scene. For our purposes, this is the function where we actually call final setup operations and make the call to the render instance to draw the object to the scene.

Now let's look at what we did for this example. You'll notice that we block the call right away if there is no shape instance on the object to prevent a potential crash scenario from occurring. From there the render pass manager expects some information about the location of the client's camera instance so it can properly draw the object, so we fetch and store that information. We then establish the LOD (Level of Detail) information for our model and also store that on the pass manager's info structure. Once we have that information, we fetch the information on the rendering state, which includes the light data and rendering transformations. We then establish model animation information and then signal the RPM to make the rendering call.

When it comes to rendering shape instances, it doesn't get any easier than this process right here. There are some other tools at your disposal with the RPM to mess with the light information as well as the object's materials, but most of the work you'll do will be in the preparation functions you will write. Be sure to read up on other object types in the engine and look to cases that closely match what you're trying to do for help on the rendering process.

Mesh Rendering: Geometry to Game

The second type of rendering that you may use for object instances is called **mesh rendering**, or the process of creating an internal geometric shape and then parsing that information to the game's rendering systems to draw that geometry to the scene. These mesh instances can have both material and color modulation applied to it to give it a unique look in the world space, however most of the time when you use this tool, it will be for debugging purposes or a very low-scale object instance that doesn't require much work to complete.

For our particular object example when we move forward, we'll be using this technique for our object instance.

When you want to perform a mesh render in the engine there are a few parameter's you'll need to store on your object instance. First, you'll need to store a few GFX Buffer instances to handle the primitive storage. A **primitive** as it pertains to the engine is a basic geometrical shape, be it a triangle or a rectangle. These buffers will be formed through a calculation method that we write to transform a client perspective into our object's "look" from their perspective. We'll also use our color modulation parameter here to actually render a rectangle with a specific color instance on it.

With this in mind, let's go back to our networked source code, and re-write the header to be ready for the mesh rendering instance:

```
#ifndef EXAMPLEOBJECT_H
#define EXAMPLEOBJECT_H
```

```
#ifndef _SCENEOBJECT_H_
    #include "scene/sceneObject.h"
#endif
#ifndef _GFXSTATEBLOCK_H_
    #include "gfx/gfxStateBlock.h"
#endif
#ifndef _GFXVERTEXBUFFER_H_
    #include "gfx/gfxVertexBuffer.h"
#endif
#ifndef _GFXPRIMITIVEBUFFER_H_
    #include "gfx/gfxPrimitiveBuffer.h"
#endif

class MyExampleObject : public SceneObject {
    typedef SceneObject Parent;
    enum MaskBits {
        TransformMask = Parent::NextFreeMask << 0,
        UpdateMask    = Parent::NextFreeMask << 1,
        NextFreeMask   = Parent::NextFreeMask << 2
    };
public:
    MyExampleObject();
    virtual ~MyExampleObject();

    //Add this object instance to the engine
    DECLARE_CONOBJECT(MyExampleObject);

    //Virtual method overrides
    static void initPersistFields();
    virtual void inspectPostApply();
    bool onAdd();
    void onRemove();

    //Custom Method
    void setTransform(const MatrixF &mat);

    //Networking
    U32 packUpdate(NetConnection *conn, U32 mask, BitStream *stream);
    void unpackUpdate(NetConnection *conn, BitStream *stream);

    //Rendering
    void createGeometry();
    void prepRenderImage(SceneRenderState *state);
    void render(ObjectRenderInst *ri, SceneRenderState *state,
BaseMatInstance *overrideMat);

private:
    ColorF mColorMod;
    GFXStateBlockRef mStateBlockRef;
    GFXVertexBufferHandle<GFXVertexPNT> mVertexBuffer;
    GFXPrimitiveBufferHandle           mPrimitiveBuffer;
};

#endif
```

So again, pay close attention to which headers are being pulled into our own header file here. We need to include support for the GFX Vertex and GFX Primitive buffers, which is what these two header files will do for us here. We then add our three new rendering methods to the class instance, a custom **createGeometry** function, which will be used to run the mathematical calculations needed to populate our two buffers, and of course we have the standard **prepRenderImage**, which is the same rendering function we used in the prior example. The big change this time around is we're going to hard-code the **render** method so we can fit in a few more bits of information to the actual render call to get our color modulation working.

Now, when it comes to rendering a mesh, there is a LOT of mathematics involved, as you need to set up the shape to be supported by primitives, instead of a standard model file, which uses tools such as nodes and verticies to draw geometrical properties. To handle this form of a rendering pass, we'll create four arrays in our code.

The first array will establish the cube's edge points, or the far edges of the cube. A tip to know is that we can start with a very basic 1x1x1 cube, and use Torque's extent property to modify the shape later on.

The second array we will create will establish the normal for each of the standard UV faces of the cube. This will be used primarily during the render pass when we set up the individual verticies for the cube to define the normal for each vertex we create during the calculation method.

The third array we'll use for this function is used to calculate texture coordinate conversions for our cube. Although it won't play much of a role for our particular case, if you choose to add a material property to the mesh, you'll need these coordinates during the render pass to properly place the UV Mapped texture onto the cube instance. Since we're not using materials, we'll omit this array from our code.

Finally, we'll create a special array instance which will hold array indices to our other three array instances. This will establish the cube's faces or the individual primitives that create a side to the cube instance.

These three arrays will work together to create a list of GFXVertexPNT objects (Vertex Points, Verticies) which will then be fed into the primitive buffer to actually create the shape itself as it would appear to our clients in the engine. The render pass manager will accept this format and a few other parameters to actually render our shape instance to the screen.

So now that you know what the overview is for the process of calculating the geometry for the cube instance, let's show you the full source code for the mesh render. Again, this is a very long piece of code so ready carefully!

```
#include "platform/platform.h"
#include "PGD/exampleObject.h"

#include "math/mathIO.h"
#include "scene/sceneRenderState.h"
```

```
#include "console/consoleTypes.h"
#include "core/stream/bitStream.h"
#include "materials/materialManager.h"
#include "materials/baseMatInstance.h"
#include "renderInstance/renderPassManager.h"
#include "lighting/lightQuery.h"
#include "console/engineAPI.h"

MyExampleObject::MyExampleObject() {
    mNetFlags.set( Ghostable | ScopeAlways );
    mTypeMask |= StaticObjectType | StaticShapeObjectType;
    mColorMod.set(1.0f, 1.0f, 1.0f, 1.0f);
}

MyExampleObject::~MyExampleObject() {
}

void MyExampleObject::initPersistFields() {
    addGroup( "Rendering" );
   addField( "colorMod", TypeColorF, Offset( mColorMod, MyExampleObject ),
        "The color modulation of the mesh instance." );
    endGroup( "Rendering" );

    Parent::initPersistFields();
}

void MyExampleObject::inspectPostApply() {
    Parent::inspectPostApply();
    setMaskBits(UpdateMask);
}

bool MyExampleObject::onAdd() {
    if(!Parent::onAdd()) {
        return false;
    }
    resetWorldBox();
    addToScene();
    return true;
}

void MyExampleObject::onRemove() {
    removeFromScene();
    Parent::onRemove();
}

void MyExampleObject::setTransform(const MatrixF & mat) {
    Parent::setTransform(mat);
    setMaskBits(TransformMask);
}

U32 MyExampleObject::packUpdate(NetConnection *conn, U32 mask, BitStream
*stream) {
    //First, allow the parent classes the write access.
    U32 retMask = Parent::packUpdate(conn, mask, stream);
    //TransformMask
    if(stream->writeFlag(mask & TransformMask)) {
```

```
    mathWrite(*stream, getTransform());
    mathWrite(*stream, getScale());
}

if(stream->writeFlag(mask & UpdateMask)) {
    stream->writeFloat(mColorMod.red, 8);
    stream->writeFloat(mColorMod.green, 8);
    stream->writeFloat(mColorMod.blue, 8);
    stream->writeFloat(mColorMod.alpha, 8);
}
return retMask;
}

void MyExampleObject::unpackUpdate(NetConnection *conn, BitStream *stream) {
    //Have the parent read method complete first
    Parent::unpackUpdate(conn, stream);
    //TransformMask
    if (stream->readFlag()) {
        mathRead(*stream, &mObjToWorld);
        mathRead(*stream, &mObjScale);
        setTransform( mObjToWorld );
    }
    //UpdateMask
    if (stream->readFlag()) {
        mColorMod.red = stream->readFloat(8);
        mColorMod.green = stream->readFloat(8);
        mColorMod.blue = stream->readFloat(8);
        mColorMod.alpha = stream->readFloat(8);
    }
}

void MyExampleObject::createGeometry() {
    static const Point3F cubePoints[8] = {
        Point3F( 1.0f, -1.0f, -1.0f), Point3F( 1.0f, -1.0f, 1.0f),
        Point3F( 1.0f, 1.0f, -1.0f), Point3F( 1.0f, 1.0f, 1.0f),
        Point3F(-1.0f, -1.0f, -1.0f), Point3F(-1.0f, 1.0f, -1.0f),
        Point3F(-1.0f, -1.0f, 1.0f), Point3F(-1.0f, 1.0f, 1.0f)
    };
    static const Point3F cubeNormals[6] = {
        Point3F( 1.0f, 0.0f, 0.0f), Point3F(-1.0f, 0.0f, 0.0f),
        Point3F( 0.0f, 1.0f, 0.0f), Point3F( 0.0f, -1.0f, 0.0f),
        Point3F( 0.0f, 0.0f, 1.0f), Point3F( 0.0f, 0.0f, -1.0f)
    };
    static const U32 cubeFaces[36][3] = {
        { 3, 0, 0 }, { 0, 0, 0 }, { 1, 0, 0 },
        { 2, 0, 0 }, { 0, 0, 0 }, { 3, 0, 0 },
        { 7, 1, 0 }, { 4, 1, 0 }, { 5, 1, 0 },
        { 6, 1, 0 }, { 4, 1, 0 }, { 7, 1, 0 },
        { 3, 2, 1 }, { 5, 2, 1 }, { 2, 2, 1 },
        { 7, 2, 1 }, { 5, 2, 1 }, { 3, 2, 1 },
        { 1, 3, 1 }, { 4, 3, 1 }, { 6, 3, 1 },
        { 0, 3, 1 }, { 4, 3, 1 }, { 1, 3, 1 },
        { 3, 4, 2 }, { 6, 4, 2 }, { 7, 4, 2 },
        { 1, 4, 2 }, { 6, 4, 2 }, { 3, 4, 2 },
        { 2, 5, 2 }, { 4, 5, 2 }, { 0, 5, 2 },
        { 5, 5, 2 }, { 4, 5, 2 }, { 2, 5, 2 }
    };
    // Fill the vertex buffer
```

```

GFXVertexPCN *pVert = NULL;
mVertexBuffer.set(GFX, 36, GFXBufferTypeStatic);
pVert = mVertexBuffer.lock();
Point3F halfSize = getObjBox().getExtents() * 0.5f;
for (U32 i = 0; i < 36; i++) {
    const U32& vdx = cubeFaces[i][0];
    const U32& ndx = cubeFaces[i][1];
    pVert[i].point = cubePoints[vdx] * halfSize;
    pVert[i].normal = cubeNormals[ndx];
    pVert[i].color = mColorMod;
}
mVertexBuffer.unlock();
//Apply the blend for our coloration
GFXStateBlockDesc desc;
desc.setBlend( true, GFXBlendSrcAlpha, GFXBlendOne );
mStateBlockRef = GFX->createStateBlock(desc);
}

void MyExampleObject::prepRenderImage(SceneRenderState *state) {
    if (mVertexBuffer.isNull()) {
        //Create the buffer list, if empty.
        createGeometry();
    }
    ObjectRenderInst *ri = state->getRenderPass()->
>allocInst<ObjectRenderInst>();
    ri->renderDelegate.bind(this, &MyExampleObject::render);
    ri->type = RenderPassManager::RIT_Object;
    ri->defaultKey = 0;
    ri->defaultKey2 = 0;
    state->getRenderPass()->addInst(ri);
}

void MyExampleObject::render(ObjectRenderInst *ri, SceneRenderState *state,
BaseMatInstance *overrideMat) {
    if (overrideMat) {
        return;
    }
    if (mVertexBuffer.isNull()) {
        return;
    }
    GFXTransformSaver saver;
    MatrixF objectToWorld = getRenderTransform();
    objectToWorld.scale(getScale());
    GFX->multWorld(objectToWorld);
    GFX->setStateBlock(mStateBlockRef);
    GFX->setupGenericShaders(GFXDevice::GSModColorTexture);
    GFX->setVertexBuffer(mVertexBuffer);
    GFX->drawPrimitive(GFXTriangleList, 0, 12);
}

```

This time around we're using three methods to perform the full rendering process, so let's discuss how this works. As mentioned before this code block, the **createGeometry** method is used to actually create a list of vertices and then to populate some GFX buffers with this information to actually create the mesh instance as it would usually appear in model form (primitives).

You'll also notice that we handle the coloration of our mesh instance through this method by using what is called a `GFXStateBlockDesc` object to blend the alpha coloration into the mesh. This allows us to create transparent mesh instances as well as ones that are opaque.

From there, we turn our attention to the new `prepRenderImage` method which now uses a function reference pointer through a new class called the `ObjectRenderInst`. This particular class instance is used to render individual objects by any of the rendering methods we are covering, as well as a few others which are beyond the scope of this guide. For a full list of the render instances, refer to the engine documentation or have a read of the Render Pass Manager source code.

Basically, the point of this function for this case is to create the buffers if they are currently unpopulated, then to establish the object rendering instance and bind it to the final rendering function, and to describe the type and rendering keys to be used for this particular case.

Finally, we have our hardcoded `render` method, which for our object here quickly gathers the calculations we did in the shape rendering example, and stores it to the pass manager and then performs the rendering call by using the primitives we created in the `createGeometry` function.

You could have easily just used the `prepRenderImage` function in this case as well, however there would still be the standard transform and definition calls we made in the prior example, as well as the functioning we just called in our new render function. Either way, you can learn by doing either case here, and I invite you to explore using both cases for a learning exercise.

You can use the mesh rendering method to create all various forms of geometrical shapes, but just remember that the more complex the shape is, the more complex the rendering of it will also be. Be sure to read the code and see how each of the individual vertices and faces are calculated through the `createGeometry` method if you're planning on creating more diverse and more complex forms of shapes and meshes for your particular project.

Beam Rendering: Lasers, Lightning, Etc.

Our final form of rendering we're going to discuss is called **beam rendering**, which is the process of defining two arbitrary points and connecting it with a beam instance. When you make a game instance, and you've got to draw lasers, or map out lines on the screen from an overhead perspective to provide a few examples, this is the form of rendering you will use to accomplish that.

We're going to keep the color parameter for our particular case here, and we're also going to add two new modifiable parameters to our object to define a starting and an ending point for our beam to be rendered across.

When you want to do beam rendering, we're essentially going to do the same as mesh rendering, but step it down another level by directly calling the primitive methods themselves to build the beam along the vectors we establish (Re-read the section on vectors in Chapter 9 if you're unfamiliar with this topic).

We'll start again with our networked object header and construct the beam rendering example object here. We'll add the new fields we want to use and work from there:

```
#ifndef EXAMPLEOBJECT_H
#define EXAMPLEOBJECT_H

#ifndef _SCENEOBJECT_H_
    #include "scene/sceneObject.h"
#endif
#ifndef _GFXSTATEBLOCK_H_
    #include "gfx/gfxStateBlock.h"
#endif

class MyExampleObject : public SceneObject {
    typedef SceneObject Parent;
    enum MaskBits {
        TransformMask = Parent::NextFreeMask << 0,
        UpdateMask    = Parent::NextFreeMask << 1,
        NextFreeMask   = Parent::NextFreeMask << 2
    };

public:
    MyExampleObject();
    virtual ~MyExampleObject();

    //Add this object instance to the engine
    DECLARE_CONOBJECT(MyExampleObject);

    //Virtual method overrides
    static void initPersistFields();
    virtual void inspectPostApply();
    bool onAdd();
    void onRemove();

    //Custom Method
    void setTransform(const MatrixF &mat);

    //Networking
    U32 packUpdate(NetConnection *conn, U32 mask, BitStream *stream);
    void unpackUpdate(NetConnection *conn, BitStream *stream);

    //Rendering
    void prepRenderImage(SceneRenderState *state);
    void render(ObjectRenderInst *ri, SceneRenderState *state,
BaseMatInstance *overrideMat);

private:
    ColorF mColorMod;
    GFXStateBlockRef mStateBlockRef;
    Point3F mStartPos;
    Point3F mEndPos;
};

#endif
```

So this time around, we only need to store the state block reference. This is because we have no preparation methods for our beam instance, we're going to directly make calls to the internal primitive builder methods in order to construct our A to B beam instance. We've also added the starting and ending points to our object's variable list, and we've kept the color modulation. Once again, this is a very lengthy bit of code so be sure to read each line to see how it works together:

```
#include "platform/platform.h"
#include "PGD/exampleObject.h"

#include "math/mathIO.h"
#include "scene/sceneRenderState.h"
#include "scene/sceneManager.h"
#include "gfx/gfxTransformSaver.h"
#include "gfx/primBuilder.h"
#include "console/consoleTypes.h"
#include "core/stream/bitStream.h"
#include "materials/materialManager.h"
#include "materials/baseMatInstance.h"
#include "renderInstance/renderPassManager.h"
#include "lighting/lightQuery.h"
#include "console/engineAPI.h"

MyExampleObject::MyExampleObject() {
    mNetFlags.set( Ghostable | ScopeAlways );
    mTypeMask |= StaticObjectType | StaticShapeObjectType;
    mStartPos.set(0.0f, 0.0f, 0.0f);
    mEndPos.set(0.0f, 0.0f, 0.0f);
    mColorMod.set(1.0f, 1.0f, 1.0f, 1.0f);
}

MyExampleObject::~MyExampleObject() {

}

void MyExampleObject::initPersistFields() {
    addGroup( "Rendering" );
   addField( "colorMod", TypeColorF, Offset( mColorMod, MyExampleObject ),  

        "The color modulation of the beam instance." );
   addField( "startPt", TypePoint3F, Offset( mStartPos, MyExampleObject ),  

        "The starting point of the beam instance." );
   addField( "endPt", TypePoint3F, Offset( mEndPos, MyExampleObject ),  

        "The ending point of the beam instance." );
    endGroup( "Rendering" );

    Parent::initPersistFields();
}

void MyExampleObject::inspectPostApply() {
    Parent::inspectPostApply();
    setMaskBits(UpdateMask);
}

bool MyExampleObject::onAdd() {
    if(!Parent::onAdd()) {
        return false;
```

```
        }
    resetWorldBox();
    addToScene();
    return true;
}

void MyExampleObject::onRemove() {
    removeFromScene();
    Parent::onRemove();
}

void MyExampleObject::setTransform(const MatrixF & mat) {
    Parent::setTransform(mat);
    setMaskBits(TransformMask);
}

U32 MyExampleObject::packUpdate(NetConnection *conn, U32 mask, BitStream
*stream) {
    //First, allow the parent classes the write access.
    U32 retMask = Parent::packUpdate(conn, mask, stream);
    //TransformMask
    if(stream->writeFlag(mask & TransformMask)) {
        mathWrite(*stream, getTransform());
        mathWrite(*stream, getScale());
    }
    if(stream->writeFlag(mask & UpdateMask)) {
        stream->writeFloat(mColorMod.red, 8);
        stream->writeFloat(mColorMod.green, 8);
        stream->writeFloat(mColorMod.blue, 8);
        stream->writeFloat(mColorMod.alpha, 8);
        mathWrite(*stream, mStartPos);
        mathWrite(*stream, mEndPos);
    }
    return retMask;
}

void MyExampleObject::unpackUpdate(NetConnection *conn, BitStream *stream) {
    //Have the parent read method complete first
    Parent::unpackUpdate(conn, stream);
    //TransformMask
    if (stream->readFlag()) {
        mathRead(*stream, &mObjToWorld);
        mathRead(*stream, &mObjScale);
        setTransform( mObjToWorld );
    }
    //UpdateMask
    if (stream->readFlag()) {
        mColorMod.red = stream->readFloat(8);
        mColorMod.green = stream->readFloat(8);
        mColorMod.blue = stream->readFloat(8);
        mColorMod.alpha = stream->readFloat(8);
        mathRead(*stream, &mStartPos);
        mathRead(*stream, &mEndPos);
    }
}

void MyExampleObject::prepRenderImage(SceneRenderState* state) {
```

```
ObjectRenderInst *ri = state->getRenderPass()-
>allocInst<ObjectRenderInst>();
ri->renderDelegate.bind(this, &MyExampleObject::render);
ri->type = RenderPassManager::RIT_ObjectTranslucent;
state->getRenderPass()->addInst( ri );
}

void MyExampleObject::render(ObjectRenderInst *ri, SceneRenderState *state,
BaseMatInstance* overrideMat) {
    if (overrideMat) {
        return;
    }
    if (mSBlock.isNull()) {
        GFXStateBlockDesc desc;
        desc.setBlend( true, GFXBlendSrcAlpha, GFXBlendOne );
        desc.setCullMode(GFXCullNone);
        desc.zWriteEnable = false;
        desc.samplersDefined = true;
        desc.samplers[0].magFilter = GFXTexureFilterLinear;
        desc.samplers[0].minFilter = GFXTexureFilterLinear;
        desc.samplers[0].addressModeU = GFXAddressWrap;
        desc.samplers[0].addressModeV = GFXAddressWrap;
        mSBlock = GFX->createStateBlock(desc);
    }
    //Constants
    S32 beamStartRadius = 10;
    S32 beamEndRadius = 2;
    //
    GFX->setStateBlock(mSBlock);
    PrimBuild::color4f(mColorMod.red, mColorMod.green, mColorMod.blue,
mColorMod.alpha);
    PrimBuild::begin(GFXTriangleStrip, 4);
    //Create the cross vectors
    Point3F crossA, crossB, sPt1, sPt2, ePt1, ePt2;
    Point3F beamDir = mEndPos - mStartPos;
    Point3F clientView = state->getCameraPosition();
    Point3F clientVec = clientView - mStartPos;
    clientVec.normalize();
    crossA = mCross(clientVec, beamDir);
    crossB = mCross(beamDir, clientVec);
    crossA.normalize();
    crossB.normalize();
    sPt1 = (crossA * beamStartRadius) + mStartPos;
    sPt2 = (crossB * beamStartRadius) + mStartPos;
    ePt1 = (crossA * beamEndRadius) + mEndPos;
    ePt2 = (crossB * beamEndRadius) + mEndPos;
    //Draw the beams
    PrimBuild::texCoord2f(0, 0);
    PrimBuild::vertex3f(sPt1.x, sPt1.y, sPt1.z);
    PrimBuild::texCoord2f(1, 0);
    PrimBuild::vertex3f(sPt2.x, sPt2.y, sPt2.z);
    PrimBuild::texCoord2f(1, 1);
    PrimBuild::vertex3f(ePt2.x, ePt2.y, ePt2.z);
    PrimBuild::texCoord2f(0, 1);
    PrimBuild::vertex3f(ePt1.x, ePt1.y, ePt1.z);
    PrimBuild::end();
}
```

As you can see the bulk of the heavy lifting for this particular rendering method is done directly in the rendering call itself. This is a very simple mathematical operation where we know the starting and ending points of the beam, as well as the client's camera position and vector. From there, we can use a few vector subtractions to get a direction of rendering and apply the vector cross product between this direction and the client's viewing vector to properly draw the beam instance.

Again, instead of using one prep method, we use the **prepRenderImage** as a forwarding call method to the customly written **render** call. This method itself establishes the internally stored state block and then performs the mathematical calculations. The **PrimBuild** class is the lowest level of the GFX rendering system in Torque, allowing you to directly draw primitives by defining verticies and then having the engine put the pieces together, per say.

Hopefully this brief introduction of the engine's rendering systems will help you to establish your own object as necessary. Again it's important to refer to the documentation and online resources when you need help learning how to use these tools for your own project.

Rendering: Quick Conclusion Points

So, let's quickly recap the rendering module in the Torque engine so you know the quick to-go points when you actually get down to this step.

- Most objects store multiple rendering related fields on them in order to save calls to repeat methods when not necessary.
- Most forms of rendering require a "preparation" method where internal object instances are assigned and readied for the actual rendering calls, the obvious exception to this case is when you are using the direct calls to the primitive builder.
- All object instances that need to be rendered must go through the render pass manager via **void prepRenderImage()**.
- It's extremely important to remember that when you perform rendering that the server must retain important object properties as well as the client to ensure the rendering sequence is done properly, and in synch with all of the clients at the correct time. By making the preparation calls on both the server and the client (mainly for shape rendering), you'll ensure that everything looks as it needs to.

Now that we've gotten all of the important concepts of rendering out of the way, let's move on to our next discussion topic as it pertains to custom object instances you create in the engine.

Collision in C++

Our next stop for our new object instance is a visit with the engine's physics system, mainly to discuss the topic of object to object collision in the C++ side of the object.

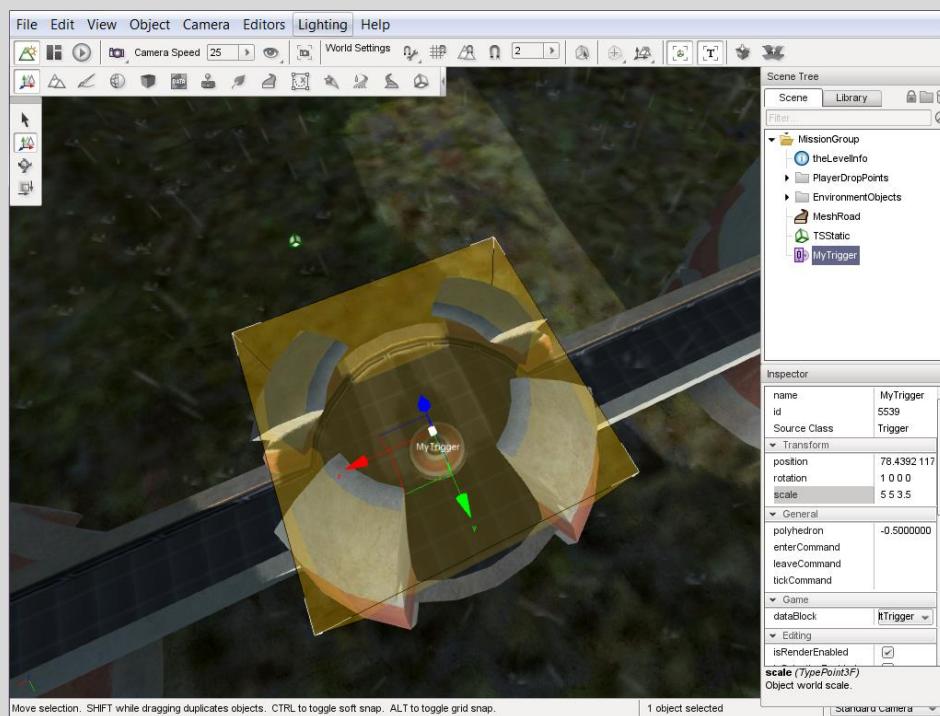
Unlike the lengthy section we just talked about, getting an object to have collision support in the engine is a really easy task. The tough stuff, as you can probably imagine is done through the heavy lifting behind the scenes in higher level objects. This does provide the advantage of making implementation extremely simplistic, but it does remove some higher level features, such as having two

objects communicate with each other during the collision event to determine properties of the other object.

So let's get starting by talking about how the engine actually handles object to object collision, and how you can make sure your object works as intended.

Bounding Boxes

This is a topic that was briefly mentioned a long while ago in Chapters 8 and 11, and now we're going to expand upon it here. Each object instance in the world that has a collidable aspect to it has to also have a **bounding box**. For a reference, a bounding box is simply a box (obvious, right?) that surrounds the collidable points of the object. More advanced structures can be defined within the box through collision functions that you may add to your object instance, but the point of the manner is simply an object that has a bounding box, by definition will also have collision support.



While this particular object instance doesn't offer collisions as you would imagine it, you can see the bounding box for this object in the editor (it's the white corners around each of the cube's edges). This particular case (trigger) is hard-coded to not actually collide with objects, however the bounding box is used in entry and exit calculations.

The property of interest as it pertains to our individual objects we create is called **mObjBox**. This is defined by the SceneObject class instance and is then used by any game instances that require collision itself. Since it's defined on the SceneObject, you don't necessarily need to redefine the instance on the header, just to make sure that it exists within the object instance itself.

Let's go back to our mesh rendering example now. You don't need to adjust the header file at all, but let's make one change to the source code on the object to add support for this field. And you just need to change one function:

```
bool MyExampleObject::onAdd() {
    if(!Parent::onAdd()) {
        return false;
    }
    mObjBox.set(Point3F(-0.5f, -0.5f, -0.5f), Point3F(0.5f, 0.5f, 0.5f));
    resetWorldBox();
    addToScene();
    return true;
}
```

If you remember from Chapter 9, I briefly mentioned how object instances would use up the lowest section it could for a space, which means a 1x1x1 box would actually exist with its origin at the point (0, 0, 0) and simply have an extent of 1x1x1, so the lowest position would be at (-0.5, -0.5, -0.5) and the highest point would be at (0.5, 0.5, 0.5). The scale property within the SceneObject class is "smart" and adjusts this parameter as it is also changed, so we don't need to make any updates beyond that point.

But let's say you're using the shape rendering case now, and you need to place a bounding box around your shape instance. That would seem to be a little more challenging, right? Well, actually it's just as easy because the resource module has a way to calculate this for you. Start by making the same adjustment to the **onAdd** method. Providing an initial bounding box of 1x1x1 is usually a good place to start, from there we use the **createShape** method we added to handle the rest, and that too is a one line adjustment:

```
void MyExampleObject::createShape() {
    if(mShapeFile.isEmpty()) {
        return;
    }
    if (mShapeInstance && mShapeFile.equal(mShape.getPath().getFullPath(),
String::NoCase)) {
        //We don't need to reload the same file if we already have one.
        return;
    }
    if(mShapeInstance) {
        SAFE_DELETE(mShapeInstance);
    }
    mShape = NULL;
    mShape = ResourceManager::get().load(mShapeFile);
    if(!mShape) {
        //Did loading fail?
        Con::errorf("MyExampleObject::createShape() - Unable to load shape:
%s", mShapeFile.c_str());
        return;
    }
    if(isClientObject() && !mShape->preloadMaterialList(mShape.getPath()) &&
NetConnection::filesWereDownloaded()) {
        //Load the materials, if possible...
        mShape = NULL;
    }
}
```

```
        return;
    }
    //Finish up...
mObjBox = mShape->bounds;
resetWorldBox();
setRenderTransform(mObjToWorld);
mShapeInstance = new TSShapeInstance(mShape, isClientObject());
}
```

So, you'll notice here that the **mShape** instance that we've just loaded in, can actually calculate its own bounding box for you. Isn't that just nice of them to add that feature for you?

Advanced Collision Notes

But, if that doesn't cover the whole list of things you'll need for your object's collision, then you'll need to dig a little deeper into the engine itself. For instance, when I was writing my Force Field pack for the engine, I had to not only write a collision code for the force field itself, but I also had to modify the behavior of collision on the objects I was interested in scanning.

The key thing to note about collision events in Torque is that it occurs on both objects that are triggering the event, therefore if one object is colliding with another object, then by definition the other object is colliding with our object.

This is a very important thing to note if you plan on making objects that for example allow certain other objects to pass through without triggering an event. The other way to achieve this kind of behavior is to remove collision altogether from the object, and to use a Physical Zone with special properties to mimic collision.

The important concept of advanced collision is to understand what methods are at your disposal and what they do. Here's a short collection of these methods for you to do some research with if you need some more advanced functioning:

- `bool Object::buildPolyList(PolyListContext context, AbstractPolyList* polyList, const Box3F&, const SphereF&):` This function replaces the standard bounding box with a polygon instance to handle collisions, the collision area is defined by **polyList**, and the object context is defined by **context**.
- `void Object::buildConvex(const Box3F& box, Convex* convex):` This function defines a convex instance that covers the bounding area. This is a testing function to see if objects are "near" a collision point with our object.
- `void Object::notifyCollision():` This function is triggered when a collision event occurs and the scripting system needs a notification event.
- `bool Object::castRay(const Point3F& start, const Point3F& end, RayInfo* info):` Perform a raycast test and fetch collision info from that raycast test.

And that covers the important key functions. Remember, if you plan on using any of these functions, you should first consult the engine documentation on these functions, or refer to other

samples present within the engine on how to properly use these tools to obtain the desired result for your custom object instance.

Processing Time Events

Our final stop for this chapter will teach you how to process time advances inside the engine for your own custom objects. For stationary objects that are static in nature like our example object, this section will have no importance. But when you're ready to step it up to the next level to learn how to make dynamic objects that move around the world or do special events as time progresses, you'll need to learn how to process time events inside the engine.

This is actually not as hard as it sounds, there are three methods you'll need to learn how to use, and they come from the **ProcessObject** class, which as you can see from our inheritance diagram goes into SceneObject. The three methods you need to learn are:

- void Object::processTick(const Move *m): This method is called once every 32ms in the engine and is called both the server and the client. This method should be used when you need to guarantee things that must match value wise on both the client and the server at the same time.
- void Object::advanceTime(F32 dt): This function is called every time the engine performs a "tick" cycle on both the server and the client. While this function does run faster than the **processTick** implementation, it's recommended that you keep functioning outside of this function unless it deals with the object's animation sequences.
- void Object::interpolateTick(F32 dt): This function is also called every tick, however this function only runs on the client end. This mainly is used to perform rendering related jobs in order to ensure smooth looking transitions on the client where processTick would otherwise be unbeneficial.

For most time related applications as it pertains to object programming in the engine, you'll usually be fine using the **processTick** method, and either forking the call off to another more time accurate function, or simply using **advanceTime** with some property checks to ensure that synchronization does not fall off between server and client.

The **interpolateTick** method, you can usually avoid unless you're making game related objects that are "high" in terms of priority that need to move more realistically where the server network unreliability would cause issues otherwise.

Since we're not actually making a more advanced object instance, we're going to skip the examples for this particular function set. Feel free to read through the other object headers and source code to get an idea of how these functions work together and with the other functions we have been working with through this chapter.

This, like every other topic to this point, takes a lot of practice to gain mastery over, but with patience and numerous attempts at it, you'll find that it's no more difficult than what we've been doing over the course of this chapter.

Final Remarks

And congratulations! You've successfully completed the key chapters of this guide. It's been a nice long journey to show you the engine inside and out while providing some samples on how to expand your toolset to create some interesting new features in the engine, but now that you have all of this down you are free to set your own path in the future of the Torque Game Engine!

I hope that after reading these first seventeen chapters of the guide that you now have a good understanding of the engine's editors, scripting system, and the basics and understanding of C++'s role in the engine itself. I can't wait to see what kind of cool new objects and features come out of your imagination through the tools you now have access to.

Before I wrap things up with the **full** guide however, I do have a few more nifty things to teach you about, so let's check those out!

Chapter 18: Some Final Goodies

So, you've finally made it through all of the "tough stuff" as it pertains to learning how to code inside the engine, and by means of the scripting provided by the engine. Now, we're going to make one last programming related stop for this guide, and talk about some additional tools that are at your disposal when it comes to adding new features or mechanics to the engine. With these nifty little features, you'll be able to further exploit the engine's networking system to benefit your additions to the engine and to ease the potential work load involved with it otherwise.

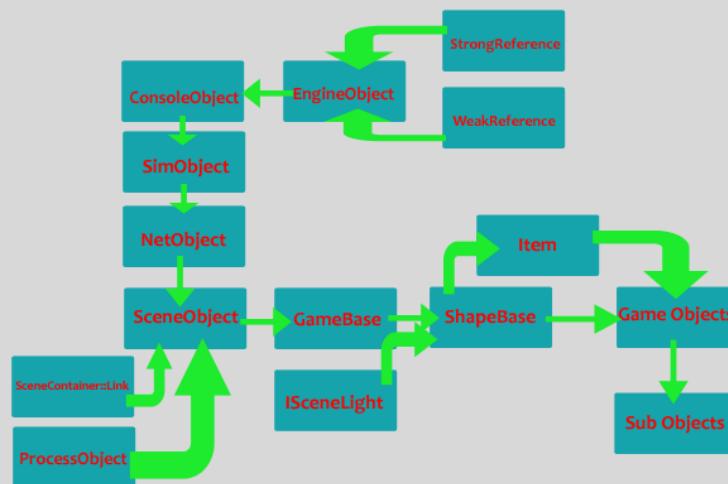
With all of that in mind, let's take a peek at some of those neat features at your disposal when it comes to building some new features.

NetEvents, sending data between server & client without objects

In the prior chapter, I talked about how to network your custom objects so they could communicate between the server and the clients on the server. If you remember however, all the way back in Chapter 12, we had another tool at our disposal. The client and server commands (**serverCmd**, **clientCmd**) which allowed us to send messages between individual clients and the server instance itself.

Now, there may come a point in time where you may want this single-client communication feature enabled on the C++ end of things. Where an object instance would originally be needed with your current knowledge, a new tool called the **NetEvent** could easily replace this.

A **NetEvent** as it pertains to the engine, is a special object class that inherits from the **ConsoleObject** class, a la:



That defines only the network related object functions as required by the engine itself, without itself being considered an object instance. This basically means, your event class instance that will derive directly from **NetEvent** can run only by ensuring two macro definitions are called, and a few network related functions are overridden.

To make this concept easy to learn and to use, we'll jump right into the examples and demonstrate how to properly use this feature to the greatest benefit you'll need.

NetEvent: Basic Example

```
#include "platform/platform.h"
#include "console/simBase.h"
#include "sim/netConnection.h"
#include "core/stream/bitStream.h"
#include "T3D/gameBase/gameConnection.h"
#include "sim/netObject.h"
#include "console/engineAPI.h"

class MyBasicNetEvent : public NetEvent {
    typedef NetEvent Parent;
    char *myMsg;

public:
    MyBasicNetEvent(char *in = NULL);
    ~MyBasicNetEvent();

    virtual void pack(NetConnection *, BitStream *);
    virtual void unpack(NetConnection *, BitStream *);
    virtual void write(NetConnection *, BitStream *);
    virtual void process(NetConnection *);

    DECLARE_CONOBJECT(MyBasicNetEvent);
};

IMPLEMENT_CO_NETEVENT_V1(MyBasicNetEvent);

MyBasicNetEvent::MyBasicNetEvent(char *in) {
    myMsg = NULL;
    if(in) {
        myMsg = dStrdup(in);
    }
}

MyBasicNetEvent::~MyBasicNetEvent() {
    dFree(myMsg);
}

void MyBasicNetEvent::pack(NetConnection *conn, BitStream *stream) {
    stream->writeString(myMsg);
}

void MyBasicNetEvent::unpack(NetConnection *conn, BitStream *stream) {
    char inBuffer[256];
    stream->readString(inBuffer);
    myMsg = dStrdup(inBuffer);
}

void MyBasicNetEvent::write(NetConnection *conn, BitStream *stream) {
    pack(conn, stream);
}

void MyBasicNetEvent::process(NetConnection *conn) {
    Con::printf("MyBasicNetEvent: Processing Message, Have: %s", myMsg);
```

```
}

//Console Methods
DefineEngineFunction(sendToServer, void, (const char *msg),, "NetEvent Demo")
{
    MyBasicNetEvent *myEvent = new MyBasicNetEvent(msg);
    GameConnection::getConnectionToServer() ->postNetEvent(myEvent);
}

DefineEngineFunction(sendToClient, void, (S32 clientID, const char *msg),,
"NetEvent Demo") {
    //Find the client...
    NetConnection *con = (NetConnection *)Sim::findObject(clientID);
    if(con == NULL) {
        Con::errorf("sendToClient(): No such client...");
        return;
    }
    MyBasicNetEvent *myEvent = new MyBasicNetEvent(msg);
    con->postNetEvent(myEvent);
}
```

Aside from going back to a single source file for the first time in a very long time, let's actually look into this example to see how things work. I shouldn't need to explain what all of the inheritance stuff is, or the class related information as that should be stuff you know inside and out by now, so let's jump right into the important bits.

There are four methods that you need to override when you create a NetEvent instance, here's what they are and what functioning they should perform:

- void NetEvent::pack(NetConnection *, BitStream *): This should ring some bells from our previous chapter's networking section as the overall behavior is exactly the same here. The pack function is called when either the server or the client (remember, NetEvents can go both ways) is getting ready to send the information to the destination. This function is responsible for packing the information into a BitStream.
- void NetEvent::unpack(NetConnection *, BitStream *): This again should be familiar. This command is called when the NetEvents arrive on the other end of the wire. This is where you should perform some quick data tests to ensure things are where they belong, and in the correct sequence (if sending multiple events). This function should also populate the data fields that were sent originally, especially if you plan on using them post receive.
- void NetEvent::write(NetConnection *, BitStream *): The write function is new this time around however. This function mainly deals with the Torque demo recording system, which is used for replays. In about 99% of cases, this function should be identical to the pack command, which is why most of the time you can get away with simply calling pack again. For more specialized cases, you may need to alter this code for replay purposes.
- void NetEvent::process(NetConnection *): This function is called once the event is fully received on the other end of the wire. This is where you should actually handle post-receive and post-clean logic on your specific object instances. There are some cases where this method may not

be useful for you, but in most events this function can be used once the event is deemed to be arrived at its destination.

So, now that you know what these four functions do, go back and re-read the code again to see how this implementation follows the standards mentioned above in the functions. You'll see that our particular object will send a basic message across the wire. The message is first written to a pointer variable when the object is created, and then packed (or written) across the wire. We then unpack into a character buffer (Remember pointer scope rules, Chapter 14), which is placed into our buffer. Once we're all done unpacking the bitstream information, we then run the process command which sends a message to the console of the destination.

As for actually sending the event, the **NetConnection** class defines a custom method called **postNetEvent**, which is responsible for accepting a NetEvent instance. The above example shows you how a client would for example post an event to the server, or how the server could post to a single client instance. This effectively mirrors the behavior of the **clientCmd** and **serverCmd** definitions back in Chapter 12.

NetEvent Macros

So now let's take a quick peek at those macro definitions that are needed by the NetEvent instance. So, the first one should be familiar to you, it defines the object instance to be used as a console object, which is pretty much standard for any class you plan on having internally used in the engine (Except some little tricky ones we'll get to in a bit). But what about that second macro, **IMPLEMENT_CO_NETEVENT_V1**. This is a special macro that allows the class to be treated as a two-way network event. By two-way, this means that the network event is allowed to transmit to both the client and the server instances.

There are quite a few options out there for your use, so let's take a look at these options now with a nice and easy to use table:

Macro	Misc Notes	Usage
IMPLEMENT_CO_NETEVENT_V1	Provide a class name	Creates a two-way net event that allows transfers between server and client
IMPLEMENT_CO_CLIENTEVENT_V1	Provide a class name	Creates a singular event that can transmit from the server to the client, but not to the server
IMPLEMENT_CO_SERVEREVENT_V1	Provide a class name	Creates a singular event that can transmit from the client to the server, but not to the client
IMPLEMENT_CO_NETEVENT	Provide a class name, and a group mask (NetClassGroupGameMask or NetClassGroupCommunityMask)	Creates a two-way net event with an additional mask layer that can transmit both ways between server and client

IMPLEMENT_CO_CLIENTEVENT	Provide a class name, and a group mask (NetClassGroupGameMask or NetClassGroupCommunityMask)	Creates a singular event with an additional mask layer that can transmit from the server to the client, but not the other way
IMPLEMENT_CO_SERVEREVENT	Provide a class name, and a group mask (NetClassGroupGameMask or NetClassGroupCommunityMask)	Creates a singular event with an additional mask layer that can transmit from the client to the server, but not the other way

So just look carefully at your options for the macro selection and then you can pick the one you'll need for your particular event and run with it. Just use the above example to see how to transmit between each of the two directions and you'll be fine.

If you are however, trying to create an event instance that loops through all of the clients on the server end, you'll need to run a loop through the **NetConnection *NetConnection::getConnectionList()** method and treat the instance as a linked list, where the next instance is retrieved by **NetConnection *NetConnection::getNext()**.

Ghosting in C++

Now that you know how to use the Network Event system in the engine, let's go ahead and revisit the resulting topic from that, which was the engine's ghosting system that we briefly introduced way back in Chapter 12.

So a quick recap of the topic from before, the engine's Ghosting system is used in conjunction with the engine's Scoping system. A ghost is basically a client's copy of a server object stored on a separate client list instance, and is used for interpolation purposes to ensure the gameplay remains smooth, even in an instance where the network connection may not be the most stable. This list is used in conjunction with a client ID system that can be established to resolve both server objects and client objects on the other end of the spectrum.

Now, we need to be careful when using this system, especially when you try to maintain high precision calculations on ghost instances, because in order to be a legitimate object on the server or client end, they first must be **resolvable** as it pertains to your particular instance. This means that the instance must first be networked to the other end and have received at least one transmission from the object where it lies within the scope of the instance.

There really isn't a great "easy" example that can be provided to you to show you how to work this system, so instead I'm just going to provide the list of functions you'll need to actually get this to work and provide a general set of guidelines to follow when programming using the ghosting system.

Functions for Ghosting

There are three functions you'll need to learn about, and learn how to use when it comes to the engine's object ghosting system. Let's introduce them and show you how to use each of these tools. All

of these functions exist on the **NetConnection** class instance, so you'll have access to them through either a standard NetConnection instance or the GameConnection class.

The first function is called **getGhostIndex**. This function is used to fetch the table ID of the ghost object on your end, provided the networked object instance as a parameter. This function returns an integer and accepts a **NetObject *** as a parameter. Here's an example of this function:

```
#include "platform/platform.h"
#include "console/console.h"
#include "console/consoleTypes.h"
#include "console/engineAPI.h"
#include "sim/netConnection.h"
#include "sim/netObject.h"

S32 fetchGhostInstance(NetConnection *clientConn, NetObject *serverObject) {
    S32 cGID = clientConn->getGhostIndex(serverObject);
    if(cGID == -1) {
        Con::errorf("fetchGhostInstance(): Cannot resolve client ghost
index.");
    }
    return cGID;
}
```

You can use this function to parse objects of any type into a ghost index. If the object cannot be ghosted, or is not resolved on the targeted client instance, it will return a -1 to the function. You can use the **dynamic_cast<NetObject *>(obj)** style to convert another object type to the **NetObject** type, so long as it exists on the inheritance tree diagram below the **NetObject** class instance.

So what exactly do we do with this specific ID number? You can then use one of the other two functions to resolve the local object instance.

If you're resolving a ghost on the client end, you use the **resolveGhost** function and provide it with the obtained ghost ID number. The function will return NULL if the object instance cannot be resolved. This function returns a **NetObject *** and accepts an integer as a parameter. Here's an example of the **resolveGhost** function in action:

```
#include "platform/platform.h"
#include "console/console.h"
#include "console/consoleTypes.h"
#include "console/engineAPI.h"
#include "sim/netConnection.h"
#include "sim/netObject.h"
#include "scene/sceneObject.h"

SceneObject *resolveGhostObject(S32 ghostID) {
    NetObject *tObj = NULL;
    tObj = NetConnection::getConnectionToServer()->resolveGhost(ghostID);
    if(tObj == NULL) {
        Con::errorf("resolveGhostObject(): Unable to resolve ghost
object, returning NULL");
        return NULL;
    }
}
```

```
    return dynamic_cast<SceneObject *>(tObj);  
}
```

You'll see that in order to get this variant to work, you'll need to obtain the connection to the server to resolve the server object that's stored on our local end with the obtained ghost ID. We use the **SceneObject** class instance, because that's the lowest level object we can safely type-cast to before we hit the **NetObject** level. You could just as easily type-cast to whatever object you need, so long as it's below the NetObject level.

The final ghosting function is used when you're trying to resolve a client object on the server. Since both the server and the client have their own lists for storing ghost objects, you need to use a separate function to resolve a client ghost. This function is the **resolveObjectFromGhostIndex** function on the server. The syntax is exactly the same; you just need to change the target of the pointer:

```
#include "platform/platform.h"  
#include "console/console.h"  
#include "console/consoleTypes.h"  
#include "console/engineAPI.h"  
#include "sim/netConnection.h"  
#include "sim/netObject.h"  
#include "scene/sceneObject.h"  
  
SceneObject *resolveClientGhostObject(NetConnection *tCl, S32 ghostID) {  
    NetObject *tObj = NULL;  
    tObj = tCl->resolveObjectFromGhostIndex(ghostID);  
    if(tObj == NULL) {  
        Con::errorf("resolveClientGhostObject(): Unable to resolve ghost  
object, returning NULL");  
        return NULL;  
    }  
    return dynamic_cast<SceneObject *>(tObj);  
}
```

And that covers the functions for ghosting in the engine. Now, let's provide a few quick guidelines to help you ensure the best results from your ghosting tools.

Ghosting Guidelines

So, now that you know the tools for using the ghosting system inside the engine, let's talk about some really quick reminders that will help you to ensure that things run as intended.

- Objects must be resolved on either the target instance before they can be safely ghosted, and used on the target instance. You can use a NetEvent to ensure the object is resolved by checking the object ID with the **getGhostIndex** function.
- Objects are not resolved in the ghost table until they are properly scoped through the networking system on the other instance.
- While the ghosting system does provide a perfect replica of the object instance, it does not give you realtime updates on the object, or authority over the object. The server instance still holds the “true” object list and therefore has direct authority over the objects inside of it.

- You should not use ghosting of the object to handle high performance calculations using the object. If you need realtime updates of a particular object, use a two-way NetEvent to directly fetch this information from the server or client as needed.
- Remember, **BOTH** server and client objects can be ghosted to the other instance for use. You need to make sure you're handling the index values properly and not mix up the two tables when working with both instances.

And that should just about cover all you need to know about the engine's ghosting system. Just be sure to follow these standard guidelines when working with ghost objects, and you'll be up and running with this system in no time at all!

Getting an external C++ Library into the Engine

And now we're going to talk about something that's even more fun to do with access to the engine's source code. And this is the ability to take an external C++ library, and plug it into the engine for usage.

To this point in time, your work has only been with some basic C++ stuff, and a few topics of more advanced C++. So, what exactly is a library and why should we at all be interested in using one?

What is a Library?

In C++, when you choose to compile your code instance, there are actually more options available than just a simple executable file instance. You can create what is called a Dynamic Linked Library (.DLL) which stored a huge library of your function instances to be used by another program, for instance, or you can even create a static library (.lib), which contains referenced functions that can be compiled into another program instance.

When it comes to making a library in C++, the developer will usually choose to either allow their code to be released under a standard limited restriction license, and release the code base under a static librarart format (.lib), which allows the code to be inserted into the compile process without needing external referenced, or a more restricted form of a license where a .DLL file will be created and it will need to exist alongside the executable in order for the program instance to run.

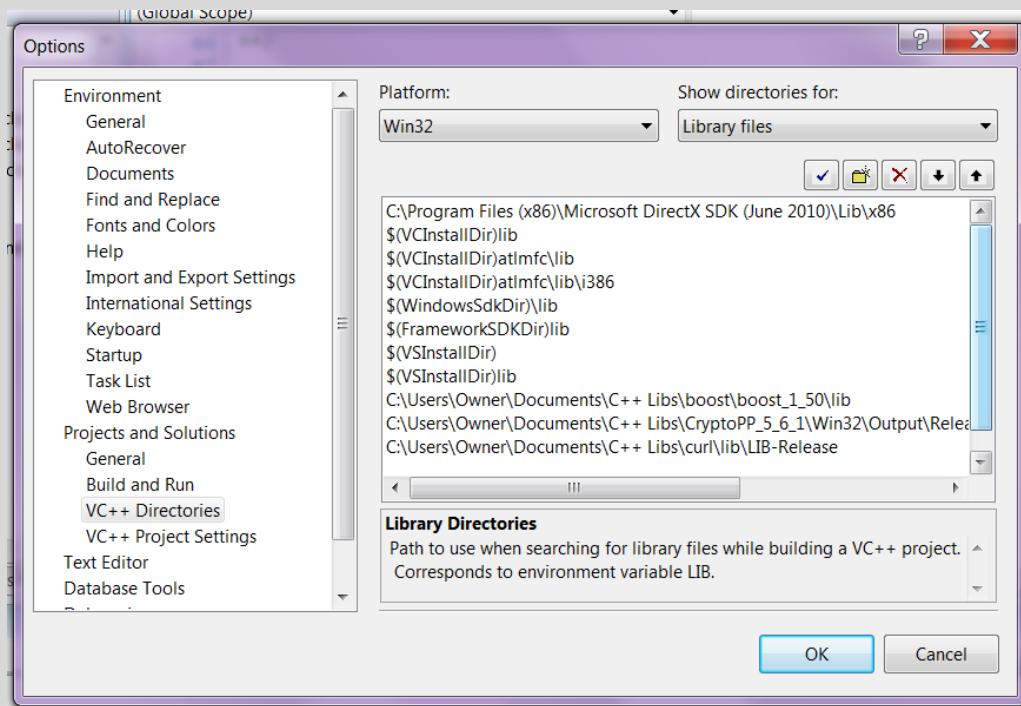
Libraries usually provide you with access to new header files for compilation purposes, and to give pointer entries to their functions and classes that exist within the bounds of the library itself. When you add a library to your solution, you'll be able to use the **#include** statement to point to a library reference instead of a header reference, this is that old:

#include <x.h>

Notation we used a long while ago in Chapters 13, 14 and 15. In order to actually get things to work though, you'll either need to include proper preprocessor statements in your code, or you'll need to set up visual studio to recognize the format. Let's show you how to do this now.

Using a Preprocessor to include a Library

In order to have our preprocessor directive detect the library of interest for our program, we first need to add the library path to be recognized by visual studio itself. You can do this by opening up the options menu for MSVS (Tools -> Options) and navigating the the directories area:



Once you open up this menu, you can add a new read area for your library files to be scanned in from. It's also standard practice at this point to add the relevant directory of the library folder where the include files are also located.

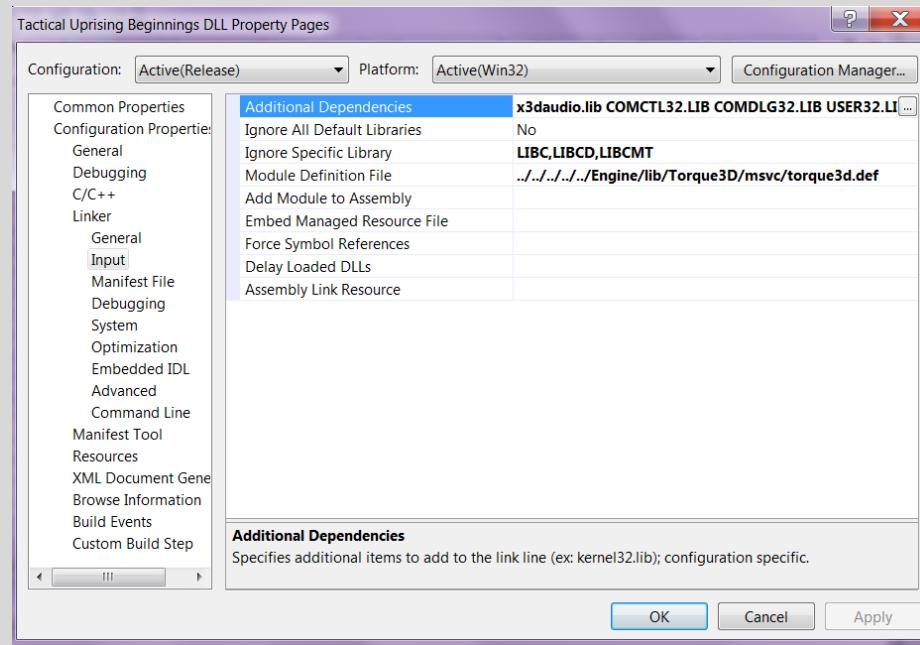
After you install the directories you can hit the OK option and then navigate to the file where you'll want the library to be accessible from, and inset the following preprocessor directive into your code at the top of the file:

```
#pragma comment ( lib, "x" )
```

You replace the x with the name of the library file, minus the .lib extension. This will tell the compiler to include the library and its relevant code at that location. At this point in time, if you've also installed the location of the header files, you can now use the reference syntax to load in your headers from the library of interest and get to work.

Configuring MSVS to include a Library

The other way you can add a library to the engine is to configure Microsoft Visual Studio itself to include the library during the compilation process. You can do this by accessing the project properties page and navigating to the linker input section:



From this page, you can use the **Additional Dependencies** section to add libraries that need to be included to your entire project. You need to make sure that you watch carefully the Configuration and Platform dropdown boxes above, because if you change from release to debug, you'll need to configure the other setting as well.

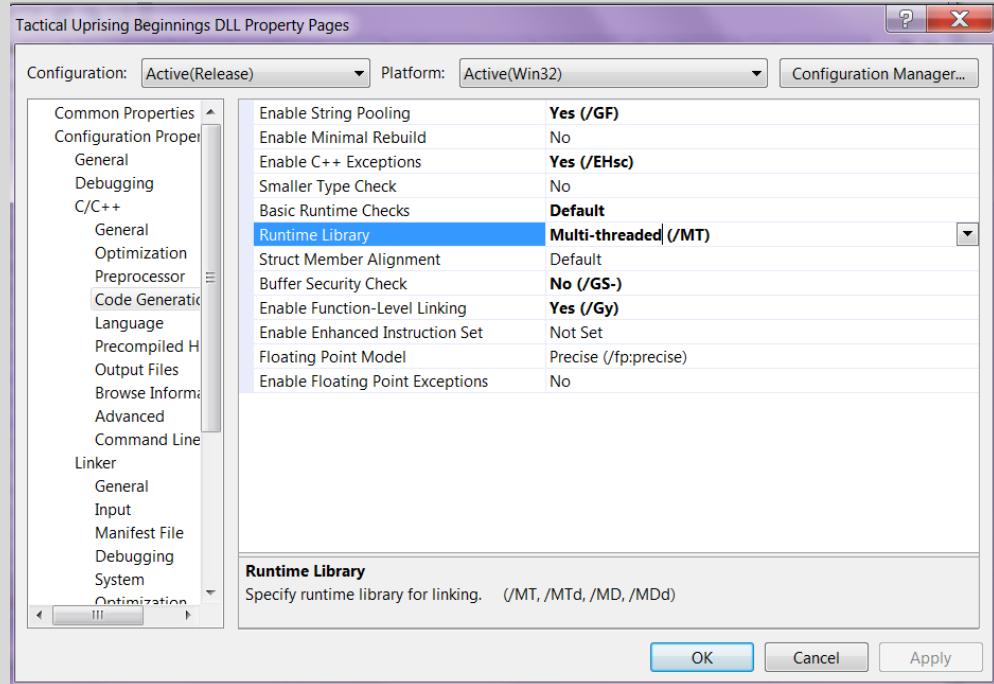
Resolving Linker Errors

More times than none, you'll insert a new library into the engine and create linker errors due to references or duplicates. Most of these duplicates, especially with core mechanics libraries will come from the standard C++ header files (LIBC, LIBCD, LIBCMT, etc). You can resolve these errors by navigating back to the linker input page and ignoring those specific default libraries.

If however, the error arises due to a duplicate reference issue or some other error, you'll need to investigate the error message itself. It will usually contain a brief mention of the function responsible for the linking error to give you a good place to start looking.

The best route of attack for fixing these errors is to get a list of responsible functions, and then check the entire engine to see if Torque already has a duplicate. In the event of a duplicate, you'll either need to gain access to the library's source code to change and recompile the library itself, or you'll need to adjust the Torque function to have a new name (obviously make sure all other internal definitions match the torque name at that point).

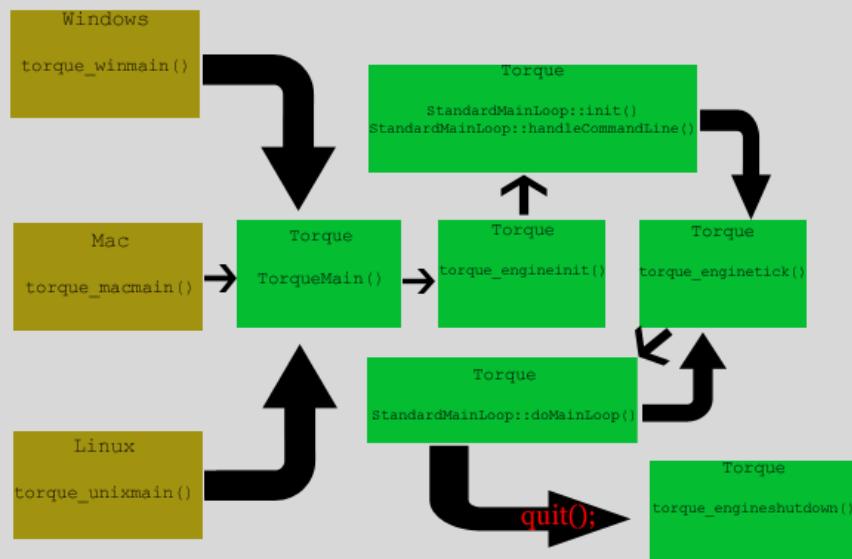
Finally, if you do get a linker error related to the Windows functions, or WinMain, that means that both the library and the engine are trying to work with alternating startup configurations, you'll need to adjust the runtime library setting to a compatible version (Usually Multithreaded, or Multithreaded DLL will work):



For any other linker related errors you may receive by including a library into the engine, resort to a search of the linker error code and try the applicable remedy to the solution. The GarageGames community is also a great place for help with linker errors, so don't be afraid to hop onto there to post a request for help if you need it.

Entry Point Revisit

So, back in Chapter 15 I talked a bit about the Torque 3D engine's entry point for the C++ code, and provided you with this diagram:



This diagram is actually extremely relevant when it comes to working with libraries that you may want to insert into the engine with code that runs alongside the engine itself.

If this is the case, you may want to insert a static module into the engine to run alongside the other engine modules. The ones that are most known to you already are the Console module (**Con::**) and the Math module (**Math::**). You can just as easily create your own static module that runs library code alongside the standard engine code at your cue. Here's the general format of a static module in the engine:

myModule.h

```
#ifndef MODULE_HEADER
#define MODULE_HEADER

#ifndef _CONSOLE_H_
#include "console/console.h"
#include "console/consoleInternal.h"
#endif

class MyModule {
public:
    MyModule();
    ~MyModule();

    static void create();
    static void destroy();
};

extern MyModule *myMod;

#endif //MODULE_HEADER
```

The standard format for a module is to include the relevant header files in the header (including any libraries). The module is initialized using the create function and destroyed with the destroy function. The reference to the module is maintained by the external myMod pointer variable. Now let's look at the source:

myModule.cpp

```
#include "PGD/myModule.h"

#include "platform/platform.h"
#include "console/console.h"
#include "console/consoleInternal.h"
#include "console/engineAPI.h"

MyModule *myMod = NULL;

MyModule::MyModule() {

}

MyModule::~MyModule() {
```

```
}
```



```
void MyModule::create() {
    if(myMod == NULL) {
        myMod = new MyModule();
    }
}

void MyModule::destroy() {
    if(myMod != NULL) {
        delete myMod;
        mymod = NULL;
    }
}
```

From this code here, you'll see how the module itself is simply an initialized pointer instance, so be sure to initialize all of your variables in the constructor, and free them in the destructor. To actually get the module to run alongside the engine, we'll modify one of Torque's engine loop methods mentioned above, specifically the **StandardMainLoop::init()** method, which is located in the file **app/mainLoop.cpp**:

```
#include "PGD/myModule.h"

void StandardMainLoop::init() {
    #ifdef TORQUE_DEBUG
    gStartupTimer = PlatformTimer::create();
    #endif

    .

    .

    Con::init();
    Platform::initConsole();
    NetStringTable::create();

    MyModule::create();
    .
    .
}
```

You'll want to make sure to include the header file we just made with our module in the main loop file, then scroll down to the method itself and find the short block where the Console, Platform, and Network String Table are initialized. Once these three modules are up and running, you can safely insert code into the main loop that uses torque's standard functions.

Also, if your module defines memory instances that are held throughout the engine cycle, be sure to add the destroy statement in a similar position to **StandardMainLoop::shutdown()**.

General Guidelines for Engine Modifications

The final bit of information I'm going to give you for this chapter mainly deal with the inevitable future you'll have at your hands with this engine, which is actually going inside and messing with how things work.

We've already been doing this to an extent by adding custom objects, making modules, and overall just doing what you're supposed to do as a developer, making the game you want to make.

So, this last bit of advice I have for you will make sure you're doing things to the book and as they're supposed to run.

- Make sure that when you're inside the engine, that you actually use the engine classes in your code. There is no reason to bring in the ultra massive Standard Template Library to handle simple jobs like Strings, Vectors, Etc. Torque defines internal implementations, use them.
- I cannot stress enough, how important it is to actually read, and re-read the engine's documentation. More times than none, you'll have a problem that can be very easily solved by reading the internal implementation of the item in question in the documentation.
- Consult help sooner rather than later if you need it. Sometimes programming introduces very complex problems that involve C++ wizardry or nothing short of programming black magic to get something to work as intended. There are hundreds of thousands of expert programmers out there just waiting to help if you need it. Don't be afraid to use websites like Stack Overflow, or cplusplus.com to help solve your problems.
- Always start by using an object that closely resembles the solution intended. If you start with something in the engine that's already working then you can knock out some of the hard code right away.
- Document EVERYTHING. And keep your code clean and neat. It will help you find problems and bugs so much more quickly when you know exactly where to find a bit of code.
- Use the tools at your disposal. You can very easily solve a naming dispute with a Namespace for instance, you don't need to grind out hundreds of unneeded hours to get something easy fixed when one or two lines of code will fix it.

There are so many great pieces of advice that could be offered to you at this point in time. Now that you have a strong understanding of how things can be done inside the engine, you should jump right into things. Try, and try again. You'll never actually make something work until you actually make it to get started with, so don't be afraid to experiment. The worst you can ever do is create an access violation on your computer, which will likely be something you'll come to love as a programmer.

It always seems like every programmer always finds something to be their "Bane of Existence". Semicolons are always a fun thing, and when you start writing more complex and advanced pieces of code, you'll come across other nuisances, but with more and more practice, you'll have no problem overcoming them.

Final Remarks

And there you have it! I hope this final chapter on the C++ side of things helped you learn some really cool and powerful tools to expand your knowledgebase with. Just a reminder, to always consult the online references and readmes for anything you may be adding to the engine and the documentation for the engine itself whenever you plan on making large scale changes inside the engine.

So, you've now completed all of the tutorial related topics of the guide, let's finish things up with some useful tools that will help you outside of the engine.

Chapter 19: Packaging for a Release

And here we are, the very last chapter in The Ultimate Guide to Torque 3D. It feels like it's been quite a long run for me to talk about all of the important things I wanted to show you, but now let's get down to that final piece of information.

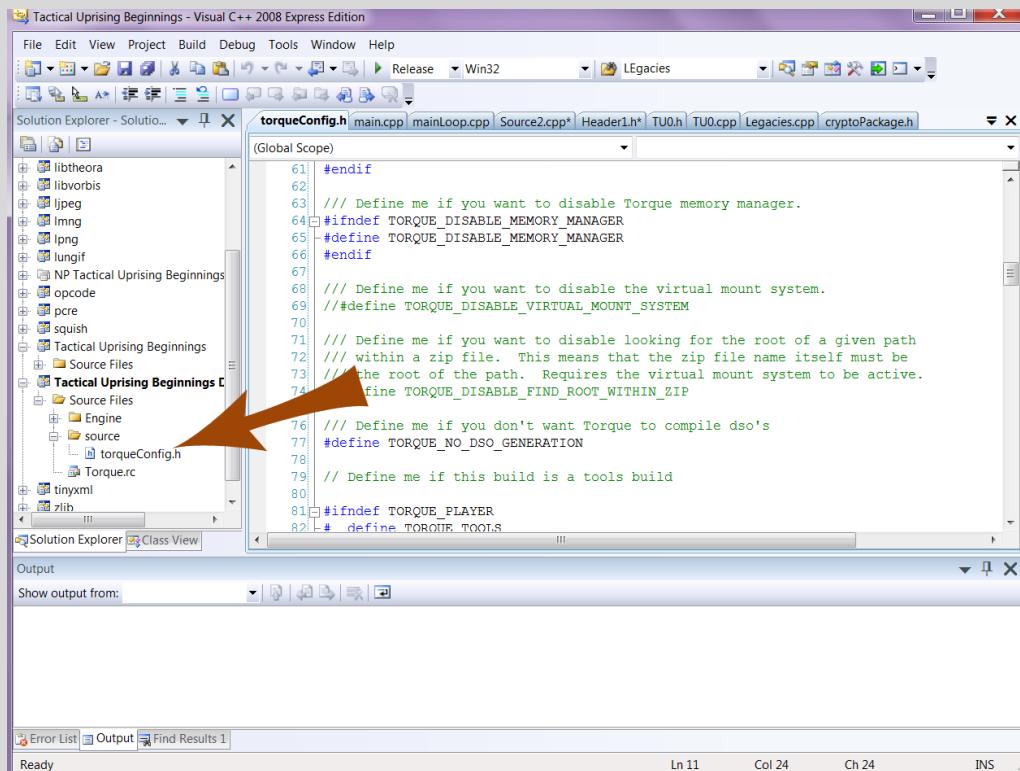
What happens when you finally get a project to a state that you're happy with, and are ready to release it to the public?

Well, there are quite a few tools out there that can help you do this, but we'll cover the important bits of information here to help you along!

CAUTION: BACK UP YOUR DEVELOPER'S COPY OF ALL OF YOUR FILES BEFORE PROCEEDING WITH THE STEPS IN THIS CHAPTER, OTHERWISE YOU WILL NOT BE ABLE TO ALTER YOUR PROJECT ANY FURTHER.

Configuring the Shipping Build

The first step in a completed build is to make sure that the engine is properly configured as a **shipping** build of the game. To set up this configuration, you'll need to open the engine's configuration file and make the proper settings there:



Once you open up this file, you'll want to add two defines after the header guard preprocessors, and they are as follows:

```
#define TORQUE_PLAYER
```

```
#define TORQUE_SHIPPING
```

The first define tells the engine this is a build for a player to use, and not a developer's version of the game instance. This strips out a bunch of the internal developer toolsets contained within the world and gui editors, and it removes a good amount of the debugging guards on these tools to cut back on the performance needed by the overall executable file.

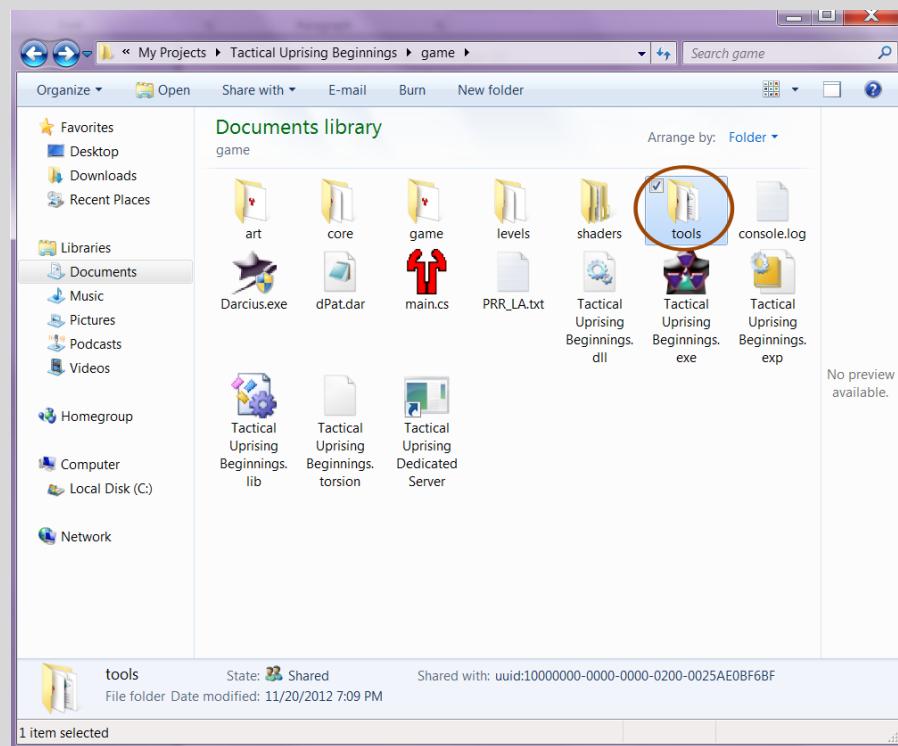
The second define is the actual flag to tell the engine it's a shipping build. This define eliminates a large number of the internal debugging tools as well as some of the TorqueScript debugging functions that would otherwise be memory intensive in nature.

Once you've added these lines, you'll want to save all of your files and perform a Clean & Rebuild on the entire solution. This will remove all of the compiled instances including these debug guards. Next, you'll want to verify that MSVS is set to the **Release** mode of the compiler, and then build a fresh copy of the executable for your package to contain. This process will take anywhere between 10 and 30 minutes depending on the computer.

When it's done building, you're finished up with the first step of the packaging process for the engine. Now, let's move back to the TS side of things and clean that up.

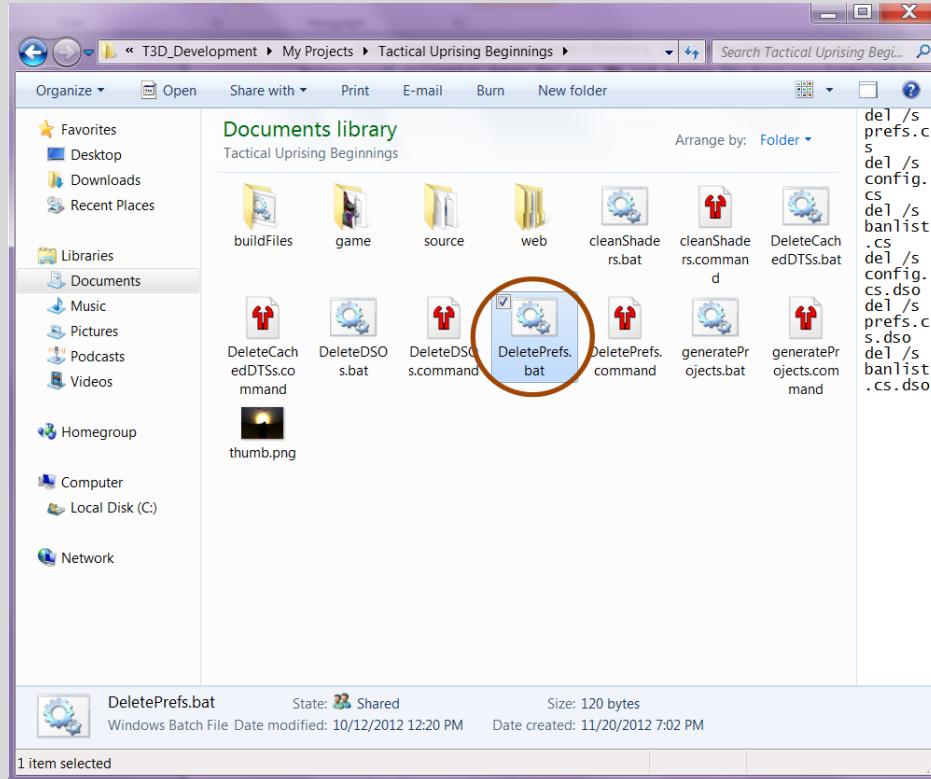
Compiling TorqueScript

The next step to the packaging process is to clean up the TorqueScript side of things. The first step to this, is to remove the toolset from the engine to free up a boatload of performance. This is very easy:



Just go back to your game's main directory, click on the **tools** folder, and delete it. At this point in time, you'll also want to delete the **.exp**, **.lib**, and **.torsion** files from your folder and the **console.log** file as well. If you want to add any readme, licensing, or extra files to your game folder, this is the time to do this operation.

The next thing we need to do is remove any client or server preference files that were added to the folder during any test runs you may have performed with the engine. Thankfully Torque provides you with a simple batch file to perform this.



You'll want to run the **DeletePrefs.bat** file and allow this process to complete. Once you've completed that step, you're going to want to compile all of your TorqueScript files into the released form of **.DSO**, unless you plan on making your game have modification support. To do this, you'll want to highlight your newly created executable file, and create a shortcut to run it.

Instead of running this however, modify the properties of the executable and add the following command to the end of the **Target** line: **-compileall**

When you run this executable, the game window will appear and then quickly disappear. This will compile all of your TorqueScript files (**.cs**) into TorqueScript Compiled Assets (**.DSO**).

To ease the step of actually deleting all of these files, I've included my own batch file in the guide's files folder named **CleanCS.bat**. Drop this in the same folder as the **DeletePrefs.bat** file and then run this batch file to clean up all of the **.cs** files in your folder.

Once you've finished this step, you're all done with the cleaning of your source code assets. You're almost done, there's just one more thing left to do!

Removing Source Assets

The last important step is to remove any source assets that are uncompiled from the engine. Whenever you purchase outside art files for instance, the creators will usually want you to make sure their assets are protected from outside access.

If this isn't the case for your project, then you can ignore this step completely and move onto the final steps section below, but if you do have art assets that need to be secured, then you'll want to have the engine convert them into .DTS files for protection.

Since the remainder of the engine is already compiled, all you'll need to do is jump into your game itself and load up all of your mission files. This will ensure that each art asset is compiled into a .DTS format, this is also the time to perform final release checks from the inside of your game, to test for anything that may be broken or not performing as intended, and to remedy the problem in your developer version.

Once you've compiled all of your files, you'll want to run the **DeletePrefs.bat** file again as it will generate a preferences file on your computer for loading up the engine.

Finally, inside the same folder as the included **CleanCS.bat** you'll see another file named **AssetCleaner.bat**, go ahead and move that to the outside directory and run it. This will make one final sweep of any of the files you may have missed earlier and wipe them from your **game** folder.

At this point in time, you now have a **Finished Build** of your game that is ready for shipping/deployment. So let's talk about actually getting it out there now!

Final Steps

The last step in the deployment process is all on you. This is where you find an installation program, or to simply zip up your **game** folder (Be sure to change the name to something relevant for your game) and ship it out.

From that point forward, the focus shifts from development to support, where your job will be to support the future of your project, and provide updates and feature addons to your game. For most people, this will mean to ship a patcher tool alongside the game itself (since Torque doesn't have a built in patching system), and to provide standard updates until you're ready to move onto the next big idea you have.

Anyways, this is where my journey ends with you. It's been a fantastic ride to show you the cool features this engine has, and how to set up the engine to work as you need it to. I'm sure you'll still have many unanswered questions that this guide didn't cover for your particular topic, but remember there are so many people out there who have been through the similar point in a project who have come across the same question, so ask away!

I can't wait to see what kind of cool games come out as a result of the things you've learned through this guide. I'll still be out there watching and learning some things as I go along too, who knows. Maybe there's a Ultimate Guide to Torque 3D Volume 2 lurking out there that one day you'll be able to pick up.

But for the time being, this is where I get to say thanks for reading the guide, and farewell!

Chapter 20: Conclusion, Final Remarks

So congratulations! You made it! We've covered quite a lot with this guide, and believe me, actually writing this lengthy piece of information for you was equally challenging from my perspective, but it was an enjoyable experience none the less. I hope you've pulled out as much knowledge from this to help your own project move in the right direction as was the time and effort from the many that helped me put this together for you.

From this guide, I hope you now have an insight into the game's editors, the scripting language behind the engine, the C++ code that runs the engine, and how it all comes together to form a powerful and awesome piece of technology to help bring your own visions of games to life.

Thanks!

I personally want to thank the following people for this resource and all of the wonderful input they brought to the table to help me bring this to you:

- Robert MacGregor: Pretty much the current voice of C++ wisdom in my own projects, and he's been one of my partners in crime with the Torque engine for many years now.
- Richard Ranft: The numerous assists and catches of my many bugs over the years and pretty much my eyes in the right direction for the instruction manual of the engine's documentation.
- Daniel Buckmaster: This awesome guy brought us the Recast system which was discussed in great detail in chapter 11, and he's been fixing my C++ problems for many years now. He pretty much knows this entire engine inside and out, and is an active member of the steering committee for the engine's future.
- Steve Acaster: Steve's been around Torque for years, and has brought us countless resources to bring some amazingly awesome things to the engine, and he always seems to have the answer right when I need it!
- The GarageGames Community: I cannot say how many times I've had a very difficult problem saved and solved by the wonderful members of the GarageGames community. They're there because they love this engine, and they want to help you learn and make it work for your project, you guys have my thanks for pretty much everything!

And that's all... or is it?

So we've reached the end of yet another journey, only this one encompassed many long months of hard work, and head-scratching for words, and yet, I still feel there's much more to talk about for this wonderful engine. There's quite a few topics that I didn't get to bring in here due to their length but they still have a very important role in the engine. Perhaps there will be a Volume II in the near future, or perhaps there won't be, but I think we've all learned quite a bit from this journey through the engine, and that is just how much there actually still is to learn!

Thank You!!!

Appendix I: Torque-Script Quick Reference

The following is a quick reference you can use when working with the TorqueScript scripting language for the Torque Game Engine series.

Here is a repeat of the operator list as seen in Chapter 6:

Arithmetic Operators

Operator Name	Operator Symbol	Example	Short Explanation
Multiplication	*	%a * %b	Multiply two numbers together
Division	/	%a / %b	Divide one number from another
Modulation	%	%a % %b	Fetch the remainder of the division between %a and %b
Addition	+	%a + %b	Add two numbers together
Subtraction	-	%a - %b	Subtract one number from another
Increment	++	%a++	Add one to the number
Decrement	--	%a--	Subtract one from the number
Totalizer	+=	%a += %b	Add a number to the value of a variable
Deduction	-=	%a -= %b	Subtract a number from the value of a variable
Multiplier	*=	%a *= %b	Multiply a number to the value of a variable
Divider	/=	%a /= %b	Divide a number from the value of a variable
Modulo Assign	%=	%a %= %b	Modulate a number from the value of a variable

Assignment Operator

Operator Name	Operator Symbol	Example	Short Explanation
Assignment	=	%a = %b	Set the value of a variable

Bitwise Operators

Operator Name	Operator Symbol	Example	Short Explanation	Binary Example
Left Shift	<<	%a << %b	Shift the bits of %a, %b positions to the left	00001000 << 1 = 00010000
Right Shift	>>	%a >> %b	Shift the bits of %a, %b positions to the right	10000000 >> 1 = 01000000
Bitwise And	&	%a & %b	Set the flag of values where they match with the number 1 to 1, and where they don't match to 0	10011001 & 00001001 = 00001001
Bitwise Or		%a %b	Set the flag of values where either are 1 to 1. If they are both 0, the	10000100 00111000 = 10111100

			number remains 0.	
Bitwise XOR (Exclusive Or)	\wedge	$\%a \wedge \%b$	Sets non-matching values to 1 and matching values to 0.	10000100 \wedge 00111000 = 10111100
Bitwise Complement	\sim	$\sim \%a$	Flip the value of all bits in the string	$\sim 00010001 =$ 11101110
Bitwise Assignment	X= (Where X is $<<$, $>>$, $\&$, $ $, \wedge , or \sim)	$\%a X= \%b$	Apply the bitwise operation specified to $\%b$ and add the result to $\%a$.	10010000 $>> 2$ will set $\%a = 00100100$

Comparison (Logical) Operators

Operator Name	Operator Symbol	Example	Short Explanation
Equal To	$==$	$\%a == \%b$	Test if the two values are equal
Not Equal To	$!=$	$\%a != \%b$	Test if the two values do not match
String Equal To	$\$=$	$\%a \$= \%b$	Return true if two strings match
String Not Equal To	$!\$=$	$\%a !\$= \%b$	Return true if two strings don't match
Greater Than	$>$	$\%a > \%b$	Return true if $\%a$ is greater than $\%b$
Greater Than or Equal To	$>=$	$\%a >= \%b$	Return true if $\%a$ is greater than or equal to $\%b$
Less Than	$<$	$\%a < \%b$	Return true if $\%a$ is less than $\%b$
Less Than or Equal To	$<=$	$\%a <= \%b$	Return true if $\%a$ is less than or equal to $\%b$
Logical And	$\&\&$	$\%a \&\& \%b$	Return true if $\%a$ and $\%b$ both evaluate to true (non-zero)
Logical Or	$\ $	$\%a \ \%b$	Return true if $\%a$ or $\%b$ evaluate to true (non-zero)
Logical Not	!	$!\%a$	Return true if $\%a$ evaluates to false (zero)

Miscellaneous Operators

Operator Name	Operator Symbol	Example	Short Explanation
Group, Parenthesis	()	$(\%a == \%b) (\%a != \%b)$	Compound statements in the order that they need to be executed. Required syntax statement for control blocks & function definitions
Block	{ }	If($\%a == 1$) { } else { }	Defines code block sections, required for function. Optional for loops and conditionals, but enforces one line statements
Array	[]	$\%a[0] = 1;$	Defines array elements (Chapter 7)
Listing, Variable Separator	,	getWord($\%a$, 2)	Separates variable parameters, list parameters
Access	.	$\%a.position$	Accesses an object's definition or field (Chapter 8)
Conditional	? :	$\%a == 1 ? \text{true} : \text{false}$	Defines a conditional statement

Namespace	::	parent::onAdd(%this, %obj)	Access a function within the specified namespace / object
Single Line Comment	//	//Hello!	Single Line comment, not executed by compiler. Used to document
Block Comment	/* */	/* This area is ignored */	Used to define large areas of code or comments to be ignored by the compiler
String Concatenation	@	%a @ " " @ %b	Add a string to the final result
Object Inheritance	:	new ScriptObject(A : B) {	Inherit the properties of B into A
String Space	SPC	%a SPC %b	Insert a space character at the specified point in a closed string
String Tab	TAB	%a TAB %b	Insert a horizontal tab character at the specified point in a closed string
String Newline	NL	%a NL %b	Insert a newline break character at the specified point in a closed string

The following is a list of all of the keywords that are present in the TorqueScript scripting language:

break	case	continue	datablock
default	else	else if	false
for	function	if	new
or	package	parent	return
singleton	switch	switch\$	true
return			

The following is a list of the datablock types and a description of them:

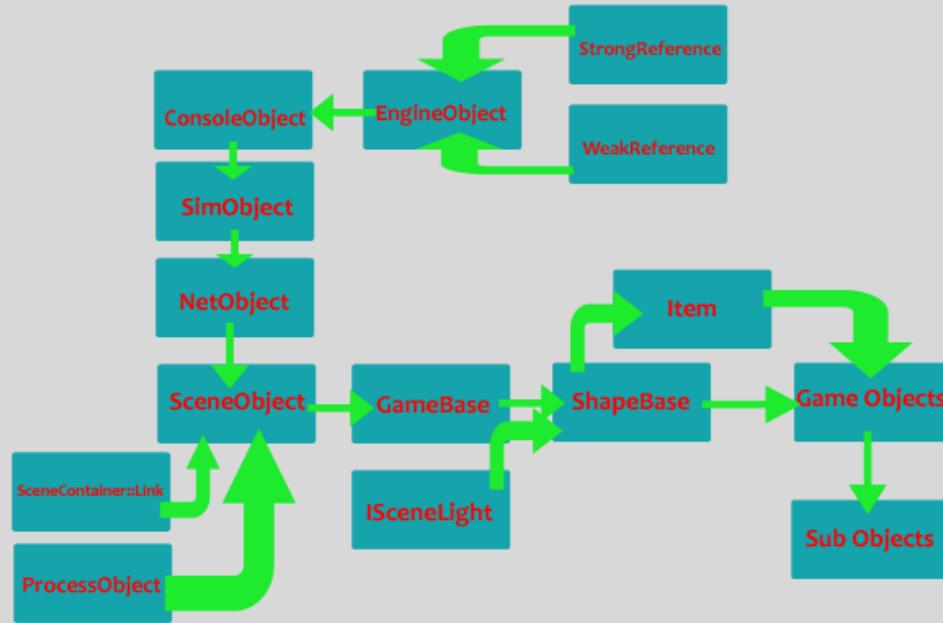
- AI{TurretShapeData}: Used by AI controlled turret instances
- CameraData: Used by camera instances
- DebrisData: Used by debris, which are individually spawned objects
- DecalData: Used to define decal instances
- ExplosionData: Used to define explosions, which are either directly spawned by projectiles, or by you
- FlyingVehicleData: Used to define properties and settings for a flying vehicle object
- ForestItemData: Used to define objects and their settings to be used by the forest editor
- GameBaseData: Generic datablock inherited by most visual objects seen in game
- HoverVehicleData: Used to define properties and settings for a hovering vehicle object
- ItemData: Used to define properties and settings of items
- LightAnimData: Used to define light instances that change with time
- LightDescription: Used to define overviewed properties applies to individual light datablocks

- LightFlareData: Used to define lights with camera flaring effects, such as a lens flare
- LightningData: Used to define lightning properties and settings for storm objects
- MissionMarkerData: Used to define properties of objects such as waypoints, and markers
- ParticleData: Used to define individual properties and elements of a single particle effect
- ParticleEmitterData: Used to define properties and settings of a rendered particle effect
- ParticleEmitterNodeData: Used to define emitter properties for a spawned particle instance
- PathCameraData: Used to define properties and settings for a camera instance that follows a path
- PhysicsDebrisData: Used when the physics module is enabled, debris instance with physics
- PhysicsShapeData: Used when the physics module is enabled, shape instance with physics
- PlayerData: Used to define properties and settings for a player object
- PrecipitationData: Used to define properties and settings for precipitation
- ProjectileData: Used to define properties and settings for projectile instances
- ProximityMineData: Used to define properties and settings for a proximity mine object
- ReflectorDesc: Used to define properties and settings for objects with light reflection properties
- RigidShapeData: Generic datablock used for shape instances with rigid collision properties
- SFXAmbience: Sound block for ambient world effects
- SFXDescription: Used to define properties of sound for SFX blocks
- SFXEnvironment: Sound block for environmental world effects
- SFXPlayList: Used to define a list of sounds to randomly select when this sound block is called
- SFXProfile: Generic sound block with properties and settings
- SFXState: Used to define the overall properties of the SFX sound when enabled
- ShapeBaseData: Very generic level object datablock, can be used to create almost any object type
- ShapeBaseImageData: Very generic level image datablock, used to define images that are mounted on another object instance
- SplashData: Used to create a water splash effect instance
- StaticShapeData: Very generic level object datablock, used to create non-moving object instances
- TriggerData: Used to define properties of a map trigger instance
- TSForestItemData: Used to define individual forest objects to be spawned by the forest editor
- TurretShapeData: Used to define properties of a turret object
- VehicleData: Generic datablock inherited by all vehicle type datablocks
- WheeledVehicleData: Used to define properties and settings of a wheeled vehicle object
- WheeledVehicleSpring: Used to define a spring instance for a wheeled vehicle, like a buggy
- WheeledVehicleTire: Used to define the properties of the tires on a wheeled vehicle

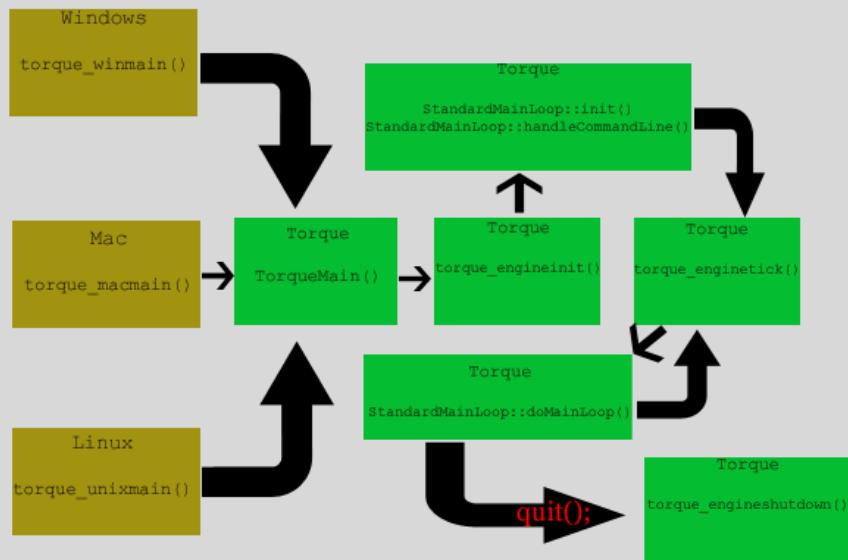
Appendix II: Quick C++ Reference For Torque

The following is a quick reference of C++ information and its relevance to the Torque 3D game engine.

The Following is the standard object inheritance tree for Torque 3D's object classes:



The Following is the entry point function call list for Torque 3D:



The following is a list of common bit sizes used by data types in C++ and the maximum value of these bit sizes:

Bit Amount	Maximum Value
8 Bits	256
16 Bits	65536
32 Bits	4294967296
64 Bits	18446744073709551616

The following is a list of standard C++ data types and their sizes:

Variable Name	C++ Keyword	Bit Size	Description
Character	char	8	A single ASCII character or a terminated character sequence
Wide Character	wchar_t	16	The largest of character sequence sizes available.
Short Integer	short	16	The smallest of integer sequence sizes available.
Integer	int	16 - 32	Standard integer value, size of the variable is ranged based on value
Long Integer	long	32	Standard integer value, forced to 32 bits
64-Bit Integer	long long	64	The largest of integer sequence sizes available.
Floating Point Number	float	32	The lowest of decimal precision available for C++.
Double Floating Point Number	double	32 - 64	Higher precision available compared to <i>float</i> but it is ranged based on the value in the variable.
Long Double Floating Point	long double	64	Offers the highest precision for decimal point numbers in C++.
Boolean	bool	8	A boolean flag, holds true or false
Void Type	void	0	An empty data type (mainly used for return flags on functions)

The following is a list of all of the commonly used C/C++ standard library header files and a description of what these headers contain:

Name	Header File	Short Description
Basic C++ Utilities		
C-Standard Lib	<cstdlib>, <stdlib.h>	General purpose utilities & functions
Signal Library	<csignal>	Signal tools & macros
Set & Jump Library	<csetjmp>	Macro tools for jumping to execution context.
Standard Arguments	<cstdarg>	Handling of variable length argument lists
Runtime Info	<typeinfo>	Runtime type information utilities & functions
Bitset Class	<bitset>	std::bitset class
Function Objects	<functional>	Object class for working with standard algorithms

Utility Library	<utility>	Functions & utilities to work with the standard library.
Time Classes	<ctime>, <time.h>	Defines standard time classes
Standard Definitions	<cstddef>	Defines type definitions for NULL, size_t, and others
C++ Memory Utilities		
Low Level Memory	<new>	Provides tools and functions for advanced low-level memory operations
High Level Memory	<memory>	Provides tools and functions for advanced high-level memory operations
C++ Numerical Data Limits		
Limits of Integral Data	<climits>	Limits of basic numerical types (int)
Limits of Floating Data	<cfloat>	Limits of basic numerical types (float, double)
Numerical Limits	<limits>	Standardized limits method of <climits> and <cfloat>
C++ Error & Exception Handling		
Exception Utilities	<exception>	Standard C++ exception types & handles
Exception Objects	<stdexcept>	Exception objects & classes
Assertion	<cassert>	Conditional macros to test the condition and assert an error otherwise
Error Number	<cerrno>	Macro definitions containing the latest error number
C++ Character & String Tools		
Character Examination Class	<cctype>	Functions and utilities to determine the contents of a character
Wide Char Examination	<cwctype>	Functions and utilities to determine the contents of a wide character
String Handling	<cstring>	Functions and utilities for string data
Wide Character Tools	<cwchar>	Functions and utilities for wide character and multi byte types
String	<string>	Defines std::string and std::basic_string types
Container Classes		
Vector	<vector>	Defines std::vector class
Deque	<deque>	Defines std::deque class
List	<list>	Defines std::list class
Set & Multiset	<set>	Defines std::set and std::multiset

classes		
Map & Multimap	<map>	Defines std::map and std::multimap classes
Stack	<stack>	Defines std::stack class
Queue & Priority Queue	<queue>	Defines std::queue and std::priority_queue classes
Container Functions	<algorithm>	Algorithms & utilities that operate on the container classes
Iteration Tools	<iterator>	Defines the iteration class object for container classes
C++ Numerical Tools & Functions		
Common Math Lib	<cmath>	Defines basic mathematical functions
Complex Numbers	<complex>	Defines the complex number type
Value Array	<valarray>	Defines a class for representing arrays of multiple values
Numerical Operations For Containers	<numeric>	Numeric operations for values that are stored in container classes
C++ Tools for Input & Output		
Pre-Define IOS Classes	<iostream>	Forward declarations for all IO classes (You shouldn't need this ever)
IOS Base	<iostream>	Define the base class for all input and output operations
Input Stream	<istream>	Define the std::istream base class for all input operations
Output Stream	<ostream>	Defines the std::ostream base class for all output operations
Input/Output Stream	<iostream>	Defines a multi-function class to handle input & output operations
File Stream	<fstream>	Defines the std::fstream class to handle file manipulation
String Streams	<sstream>	Defines the numerous string stream classes for advanced string type manipulations
IO Formatting	<iomanip>	Tools and functions for manipulating input and output of the stream classes
Stream Buffer	<streambuf>	Defines the stream buffer class
C-Style Input/Output	<cstdio>	Defines C-Style classes used for input and output
Miscellaneous Tools & Operations		
Localization	<locale>	Localization utilities & functions

C-Style Localization	<clocale>	Defines C-Style classes used for localization functioning.
-----------------------------	-----------	--

The following is a list of Torque Engine Data-Types as defined by platform.h:

C++ Data Type	Torque Shortcut
char	S8
unsigned char	U8
short	S16
unsigned short	U16
int	S32
unsigned int	U32
float	F32
double	F64
char	UTF8
unsigned short	UTF16
unsigned int	UTF32

Appendix III: Binary Numbers & Operations

The following table is a representation of the 32-bit binary numbering system, read from bottom right to top left.

2147483648	1073741824	536870912	268435456	134217728	67108864	33554432	16777216
8388608	4194304	2097152	1048576	524288	262144	131072	65536
32768	16384	8192	4096	2048	1024	512	256
128	64	32	16	8	4	2	1

Here's a repeat of the Bitwise Operators Table found in Chapter 6 for quick reference:

Operator Name	Operator Symbol	Example	Short Explanation	Binary Examples
Left Shift	<<	%a << %b	Shift the bits of %a, %b positions to the left, increase a number by a power of 2.	00001000 << 1 = 00010000 10000000 << 1 = 00000001
Right Shift	>>	%a >> %b	Shift the bits of %a, %b positions to the right, decrease a number by a power of 2.	10000000 >> 1 = 01000000 00000001 >> 1 = 10000000
Bitwise And	&	%a & %b	Set the flag of values where they match with the number 1 to 1, and where they don't match to 0	10011001 & 00001001 = 00001001
Bitwise Or		%a %b	Set the flag of values where either are 1 to 1. If they are both 0, the number remains 0.	10000100 00111000 = 10111100
Bitwise XOR (Exclusive Or)	^	%a ^ %b	Sets non-matching values to 1 and matching values to 0.	10000100 ^ 00111000 = 10111100
Bitwise Complement	~	~%a	Flip the value of all bits in the string	~00010001 = 11101110
Bitwise Assignment	X= (Where X is <<, >>, &, , ^, or ~)	%a X= %b	Apply the bitwise operation specified to %b and add the result to %a.	10010000 >>= 2 will set %a = 00100100

Appendix IV: Online Resources

The following web sites are great online resources for both Torque 3D and for C++ learning references.

- <http://www.garagegames.com> – Should be an obvious first choice for anything Torque related, forums & documentation for the engine.
- <http://forums.torque3d.org/> - Community ran Torque forums. Lots of good resources and tutorials to be found on this website. This also is where active discussions on feature additions are held.
- <https://github.com/GarageGames/Torque3D> - The official engine repository. The current development version of the engine can be located here, if there is an engine bug you're encountering, there may be a fix for it here as well.
- <http://en.cppreference.com/w/> - The most up-to-date C/C++ reference on the web, you'll find C++ documentation and implementation pages there. If you ever have a question about how something is supposed to work in C++, start there.
- <http://www.cplusplus.com/> - A wonderful community of programmers who established this website containing references and documentation for C++. There are also a great selection of tutorials and the forums to visit to help you solve C++ problems, just be sure to put in some effort before posting there though, they want to see that you've at least tried something.
- <http://stackoverflow.com/> - If you ever have any kind of programming related question that you just can't seem to solve, this is a great place to ask for help. The people there are skilled in all forms of programming languages (Even TorqueScript!), so if you need help with something, ask away.