

O M N I  
E N G I N E

# OMNI T3D Engine Programmers Manual

Winterleaf Entertainment L.L.C.

# OMNI ENGINE

## Forward

I set out over five years ago to build a game with a couple of friends. We had no idea how difficult the endeavor would be and had I known then what I know today, I might not have taken on the challenge. It seems that no matter what engine you use, you always find a short-coming in it. Some are more expensive than others, some are more refined than others, and it always seems the more refined the engine the less access to source code you have.

Our first attempts at building our game focused mainly on C++ programming. We quickly found out that you should not be doing game logic in C++. It lacks the necessary flexibility to prototype quickly. We also discovered during this process that each engine had a custom language. Some custom languages were better than others, but overall, they required learning a language that was unique to the engine and would not translate easily if you wished to change engine technology mid development.

After the development of our game flopped, we sat back and re-evaluated where we had made wrong turns and bad decisions. As a programmer I was focused on the language and structure, whereas our artists were focused on eye candy and pipe-lines. After doing some serious soul searching, and many hours talking to the other team members we decided to take on a challenge so big it seemed impossible.

I made a promise back then to my team members. The promise was simple, "I would not quit until everyone else did." The funny thing is that the other team members made the same promise. So now we had this team that was committed to solving the problems and no one wanted to be the person to go back on their word. It became an all or nothing situation.

After many prototypes and attempts at building a better engine Winterleaf Entertainment L.L.C. released the first version of an attempt to build a better engine. It was more of a bolt on to an existing engine than its own engine. The bolt on (DotNetTorque) improved performance of the T3D engine by an easily 20-25 percent and allowed developers to start writing the scripts which ran their games in C# versus the stock proprietary language for the engine called TorqueScript. Currently there are over 20 games being developed currently using this modification for T3D.

After taking a couple months off, (Actually I wrote a product called OneWorld, but that's a different story) I came back to the engine and took a fresh look at what I was trying to accomplish. I used every static code analyzer and performance monitor known to man to benchmark the source code. After a couple weeks of research I realized that not only that I could do a better job, but the changes I was proposing would simplify the programming pipeline.

To prevent any sins from DotNetTorque carrying over to the new product I started from scratch and OMNI was born.

After demonstrating the power of Omni to other engine programmers we realized that the new short coming in the engine wasn't the C# framework but the old open source engine we were was using.

# OMNI ENGINE

Several very talented engine programmers decided to join my effort by re-writing many parts of the original C++ engine code.

They started by tackling the obvious things like native 64 bit support, OpenGL, shader systems and the like. Unlike other efforts like ours, we started with a Road Map which documented where we wanted the engine to go. So even though our developers tended to work on things they liked, they were restricted to only items on the Road Map. This prevented massive scope creep by keeping all of us corralled.

The development of the engine slowly became a show-off competition where each of us would like to rub in each other's noses what we had pulled off. The competition has grown quite fierce and today it continues to grow.

Finally, the fruit of our labor paid off when we hit the last milestone in our development Road Map and Omni was born. Omni is a 3D engine design for all types of games and simulations. The core of the engine is written in C++ and all scripting is performed via C#. Omni represents the collaborative efforts of many programmers focusing on common goal of building an engine that is friendly to developers and artist alike without sacrificing performance.

# O M N I E N G I N E

## Table of Contents

Forward .....	2
Frequently Asked Questions .....	8
What is the Omni T3D Engine? .....	8
Why should I buy your engine when I can get T3D free? .....	8
How different is Omni from T3D? .....	8
Can I build a MMO/MMOFPS/MMORTS/MMOONG with Omni? .....	8
Requirements.....	10
Chapter 1 Internal workings. ....	11
Introduction .....	11
Basics.....	11
Model-view-controller (MVC) .....	11
Object Creation .....	13
Object Destruction .....	14
Callbacks to Proxy Objects .....	15
Low Level MIT T3D Engine Function Calls.....	15
Chapter 2 Installation.....	16
Omni Engine.....	16
Omni Engine Tools .....	16
Chapter 3 Creating your first Game.....	20
Project Manager .....	20
Generated Solution Files.....	27
Omni Framework Solution .....	27
<Project Name>.sln Solution.....	34
Omni Live Scripts Solution .....	37
Static Code Generator - Update the Omni Framework Code .....	38
Static Code Generator (Visual Studio Extension).....	39
Step 1, Check out the source code (if applicable).....	39
Step 2, Open the “Omni Framework Solution” .....	39
Step 3, Open the Static Code Generator Extension .....	39

# O M N I E N G I N E

Step 4, Select the Omni C++ DLL project .....	40
Step 5, Select the C++ DLL Project .....	41
Step 6, Select the Winterleaf.Engine project.....	41
Step 7, Select the C# Game Logic Project .....	41
Step 8, Review the configuration.....	42
Step 9, Click “Generate” .....	43
Step 10, Recompile .....	44
Static Code Generator (Stand-Alone) .....	45
Step 1, Check out the source code (if applicable).....	45
Step 2, Start the Static Code Generator (Stand-Alone) .....	45
Step 3, Click “Select Omni T3D Solution File” .....	46
Step 4, Select the “<Project Name>.sln” file.....	47
Step 5, Click “Select Omni Framework Solution” .....	48
Step 6, Select the C++ DLL Project .....	48
Step 7, Select the Winterleaf.Engine project.....	48
Step 8, Select the C# Game Logic Project .....	48
Step 9, Click “Execute” .....	49
Step 10, Wait for the Static Code Generator to finish. ....	50
Step 11, Add new files to the “C# Game Logic Project” .....	50
Step 12, Recompile .....	52
Running the Game .....	54
Chapter 4 Customizing Winterleaf.Game Executable Name .....	55
Chapter 5 Customizing Winterleaf.Demo.Full .....	61
Chapter 5 Object Models Overview.....	68
Introduction .....	68
Reasoning.....	69
Chapter 6 Creating Views (Omni T3D Objects) .....	70
Introduction .....	70
There are three rules that apply to all Creator Types:.....	70
Rule 1: You cannot assign properties after you call the “Create()” function.....	70

# O M N I E N G I N E

Rule 2: Any Creator based object being assigned as a property, the property must be prefixed with a “#” .	71
Rule 3: All Creator based objects assigned as properties must be the last properties assigned. ....	71
ObjectCreator Class: Create Instanced based Views .....	71
DatablockCreator Class: Create Datablock based Views .....	72
SingletonCreator Class: Create Singleton based Views .....	73
Creating Objects in TorqueScript .....	74
Chapter 7 Extending the User.Models.Extendable.....	75
Member Functions (Overriding T3D engine Callbacks) .....	75
“ConsoleInteraction” Decoration.....	76
Member Variables.....	77
Traditional Member Variable .....	77
C# Member Variable .....	78
Hybrid Member Variable .....	78
Static Member Functions (Global functions) .....	78
Function: OnFunctionNotFoundCallTorqueScript.....	79
Chapter 8 Custom Models .....	81
Chapter 9 Building Gui’s and converting them to C#.....	84
Using the GuiParser .....	84
Advanced Gui Creation .....	87
Old Style .....	87
New Style .....	87
Chapter 10 Global Functions.....	90
Chapter 11 Run-Time C# Programming (LiveScripts!) .....	91
Chapter 12 File Dialogs .....	94
Chapter 13 Threading with the Omni Framework.....	96
Chapter 14 Debugging C# and C++ .....	100
Appendix .....	110
Appendix 1 - Static Code Generator Configuration Options.....	110
Configuring C++ Class plInvoke Serializations.....	110

# O M N I E N G I N E

Configuring C++ Class/Enum Map to C# Class/Enum .....	110
Configuring C++ Class/Function Ignores .....	111
Configuring C++ Constants.....	111
Configuring C++ Return Type Casting Overrides.....	111
Configuring C++ SimObject Based Classes .....	112
Configuring C++ Source Files To Ignore For Enumeration Parsing.....	113
Configuring C++ Source Files To Ignore On Interrogation .....	114
Appendix 2 - Special Omni C# Syntax.....	115
Appendix 3 - Built in conversion functions for Casting.....	117
Appendix 4 - Casting Game Objects.....	118
Appendix 5 - Overriding functions and such.....	120
Appendix 6 - Creating Objects (ObjectCreator/SingletonCreator/DatablockCreator) .....	121
Gui's and nested Objects .....	122
Appendix 7 - uGlobal, sGlobal, iGlobal, bGlobal, fGlobal, dGlobal, fGlobal .....	123
Appendix 8 - Where did "ClassNameSpace" and "superclass" go? .....	125

# O M N I E N G I N E

## Frequently Asked Questions

### What is the Omni T3D Engine?

The Omni T3D Engine is a derived engine based off of the MIT Open source T3D project from Garagegames.com.

### Why should I use your engine when I can get T3D free?

This is true, you can download the T3D engine for free from Github and build your game. But there are many differences between Omni and stock T3D.

- C# integration
- Improved Mathematics
- Optimized engine code
- Add more here.

Most importantly you get support from people who work with the engine daily and are constantly pushing the boundaries of the engine's capabilities.

### How different is Omni from T3D?

Omni is a branch version of T3D. Just like RedHat and Ubuntu are branches of Linux, Omni is a branch of T3D. Omni can run a TorqueScript game, and run it quite fine but of course it's designed to run optimally in C#.

A knowledge of the MIT T3D project won't hurt you when it comes to Omni. Mechanics are mechanics and for the most part they are the same. The biggest difference is in the syntax and object oriented design of the script code structure.

There are some differences in the way you do things between Omni and MIT T3D, some of the more noted ones include threading, file dialog boxes, and just in time script compilation. Another big difference between Omni and MIT T3D is in inheritance. MIT T3D's scripting interface only allows a very limited form of inheritance of objects. Omni on the other hand allows programmers to inherit to any depth and create rich object oriented design using C# inheritance.

### Can I build a MMO/MMOFPS/MMORTS/MMOIMG with Omni?

Can you? Yes you can. One of the biggest problems with the MIT T3D code base is that simple things like database interaction has to be added to the engine per build. Want to add "Some Library for communication", you need to find the C++ code base and shoehorn it into the T3D C++ engine.

With Omni, adding database support is as simple as including a reference to the C# project. If you can find a Microsoft.Net DLL to do what you want you can roll it into Omni with ease. You can extend Omni to do things that would just take months in MIT T3D.



# OMNI ENGINE

Imagine building a simulator that shows the layout of your house, and if you click a door you want to unlock the door remotely. Doing this in MIT T3D would be painful at best, but with Omni, you just include the Microsoft.Net DLL and go on your merry way.

Omni enables the MIT T3D SDK to leverage Microsoft.Net to simplify programming and prototyping.

# O M N I E N G I N E

## Requirements

1. Visual Studio 2010 or 2013
  - a. The Professional version is preferred. Visual Studio 2013 can be purchased without the MSDN support for about \$300 dollars here.  
([http://www.microsoftstore.com/store/msusa/en\\_US/list/ThemeID.33363200/categoryID.62687600?WT.mc\\_id=vssitebuy\\_2013](http://www.microsoftstore.com/store/msusa/en_US/list/ThemeID.33363200/categoryID.62687600?WT.mc_id=vssitebuy_2013))
  - b. If \$300 dollars is out of your price range, visit <http://www.microsoft.com/bizspark/> , signing up only takes a few minutes and you gain access to all of Microsoft's tools free. You do need to renew it each year by clicking a button on their website.
2. Microsoft Dot Net version 4.0 or greater
3. I recommend a decent video card and computer, or the builds will take forever. I currently use an old Intel Core 2 Quad with 8 Gigs of ram to build the engine and it seems adequate.

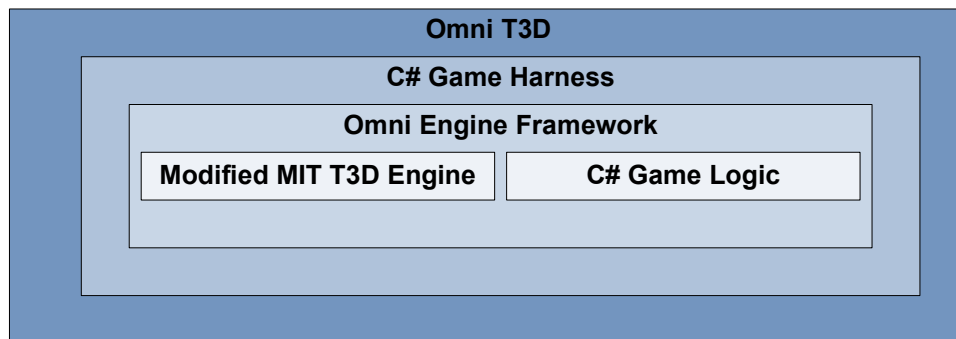
## Chapter 1 Internal workings.

### Introduction

Most approaches done by various engines are similar, they use a series of Platform Invocation Services (P/Invoke) defined in the C++ and C# to facilitate communication between the languages. Omni is no different in that regard it uses an extensive list of P/Invokes between the C++ and C#. The feature that sets Omni above the rest is the Framework which rests on top of the P/Invoke foundation.

### Basics

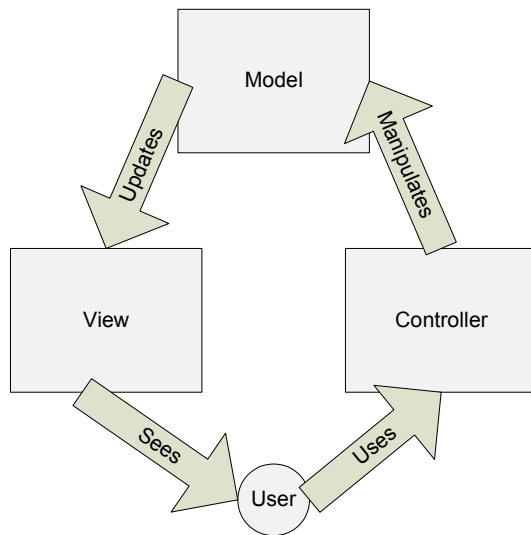
The Omni T3D engine is comprised of 4 major components. Each component serves a distinct purpose in separating the tasks that need to be done for the game to operate correctly. These four layers are the C# Game Harness, Omni Framework, Omni Branch of MIT T3D, and the C# Game Logic. This was done for many reasons but one of the more major reasons was to achieve a Model-View-Controller software design. The below diagram displays the container order of the four major parts.



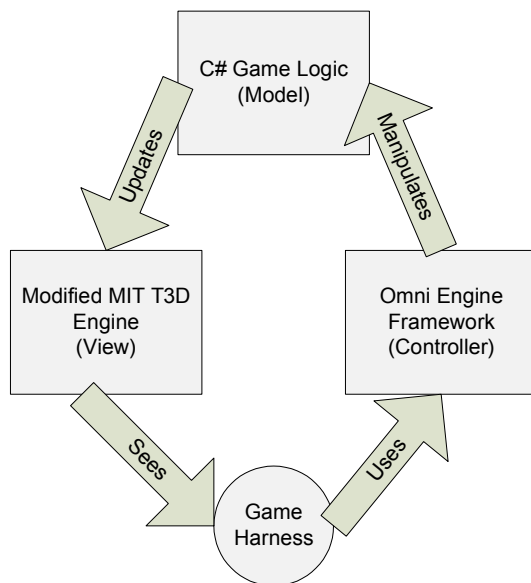
**Model-view-controller (MVC)** is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. The central component, the model, consists of application data, business rules, logic and functions. A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the controller, accepts input and converts it to commands for the model or view. —[en.wikipedia.org](https://en.wikipedia.org)

# OMNI ENGINE

According to this concept a typical design model follows the following schematic.



Applying this concept to the Omni T3D Engine we end up with:



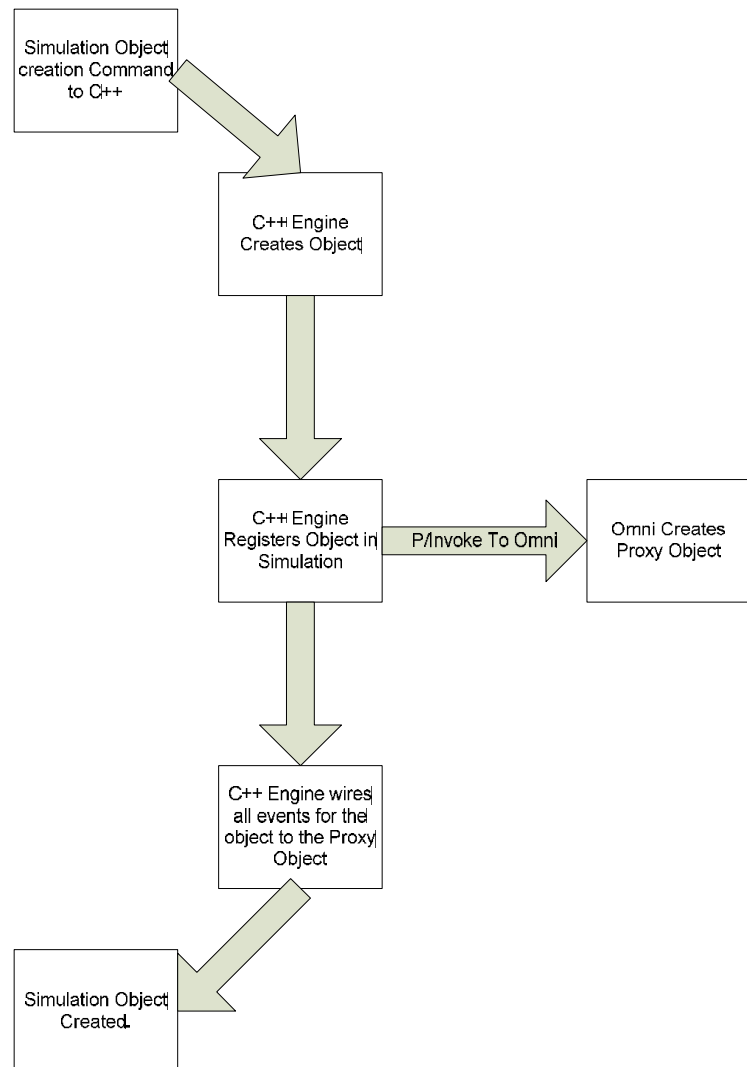
# O M N I E N G I N E

## Object Creation

Omni uses a concept of creating Model objects for every object inside the C++ engine. These Model objects are state-full and can be passed around in the C# like any other type of C# object. They are created automatically when the C++ engine creates a new object and they are cleaned up and destroyed when the C++ engine destroys them.

In the diagram to the right you can see the process that happens inside the engine when you create a new Simulation Object in the C++.

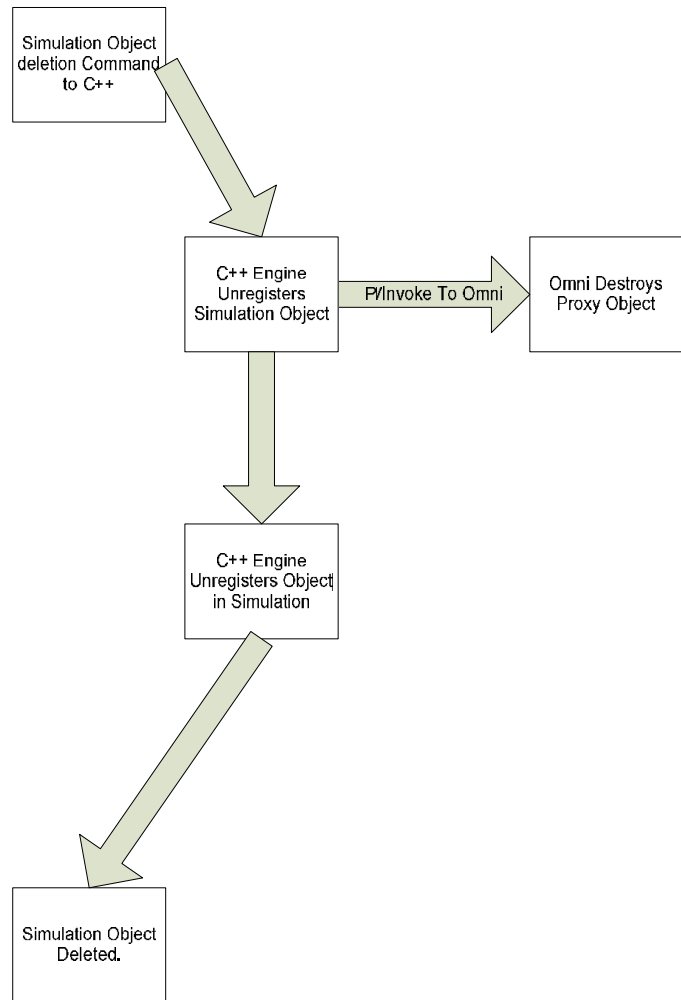
When the C++ gets a request to create a new simulation object, the engine will make a callback to the C# Omni Framework informing it to create a new Model for it.



# OMNI ENGINE

## Object Destruction

Object destruction works in a similar fashion to creation. The difference being that the C++ Simulation engine will inform the OMNI Framework to destroy the Model object attached to the Simulation object. After the proxy object is destroyed, the C++ Simulation engine will dispose of the object.



# O M N I E N G I N E

## Callbacks to Proxy Objects

Every simulation object in the game has events. These events can be simple events like “OnAdd” to more complex events related to collisions. When a Simulation Object event occurs, the Model object is looked up in the C# and the member function which matches the event name is called. This is also true if a TorqueScript running inside the engine makes a function call. All function calls are first passed to the C#, if no function exists in the C# and the proxy object is not configured to block TorqueScript, it will then look up the function in any TorqueScripts that might exist and execute it.

## Low Level MIT T3D Engine Function Calls

Unfortunately, sometimes you will still need to call or execute some code which the Omni Framework doesn’t handle or expose. These situations are quite rare, but they do happen. It is because of this several low level functions have been added to the framework. These functions let you evaluate TorqueScript via C# without having to save them to a TorqueScript file. These fringe cases will be discussed later in the document.

## Chapter 2 Installation

### Omni Engine

Run the Omni Engine Install, this program will install the source code for the Engine plus dependencies needed to compile.

**Add screen shots and process.**

### Omni Engine Tools

To build the Omni Framework you will need to run a program called the Omni Static Code Generator as either a plug in to Visual Studio (2010/2012/2013 Standard edition+) or as a standalone program. This program is included in the installation file.

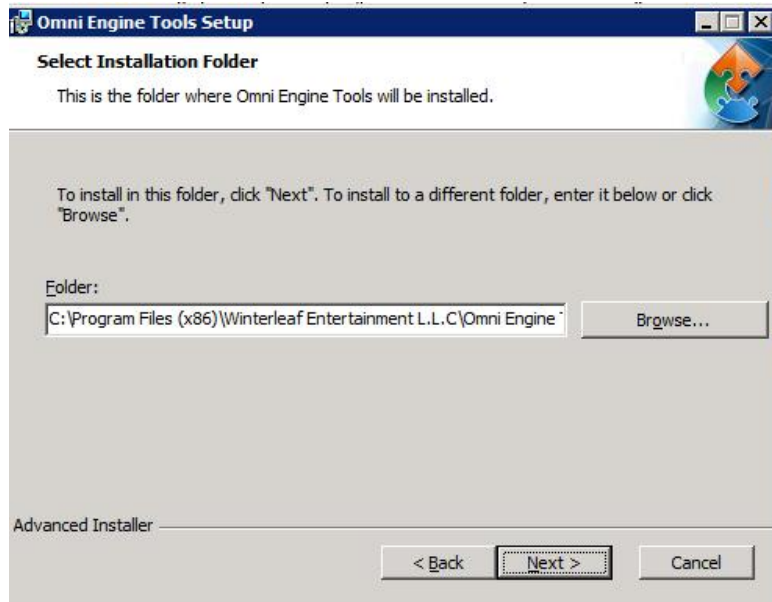
To install the tools run the “Omni Engine Tools Setup.exe”



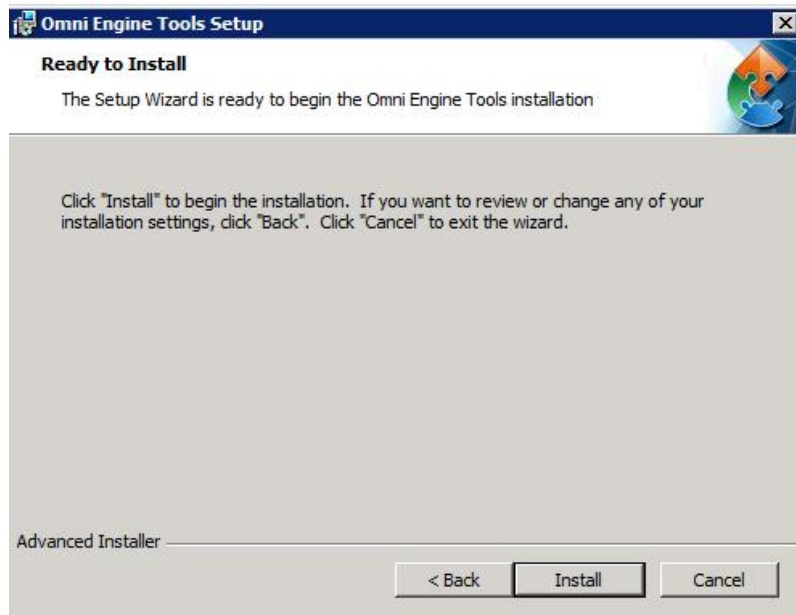
Click “Next”.



# OMNI ENGINE

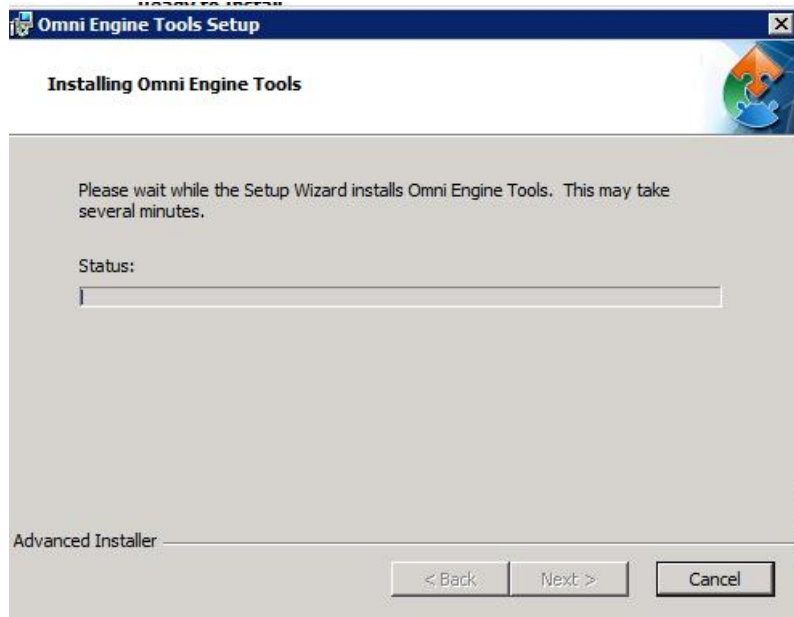


Change the default Location if you want to install it somewhere else.

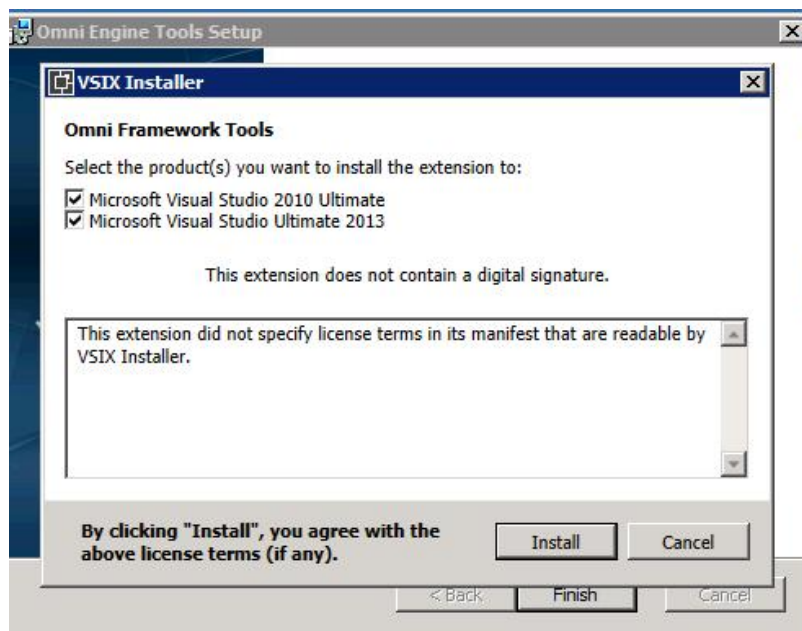


Click "Install".

# O M N I E N G I N E

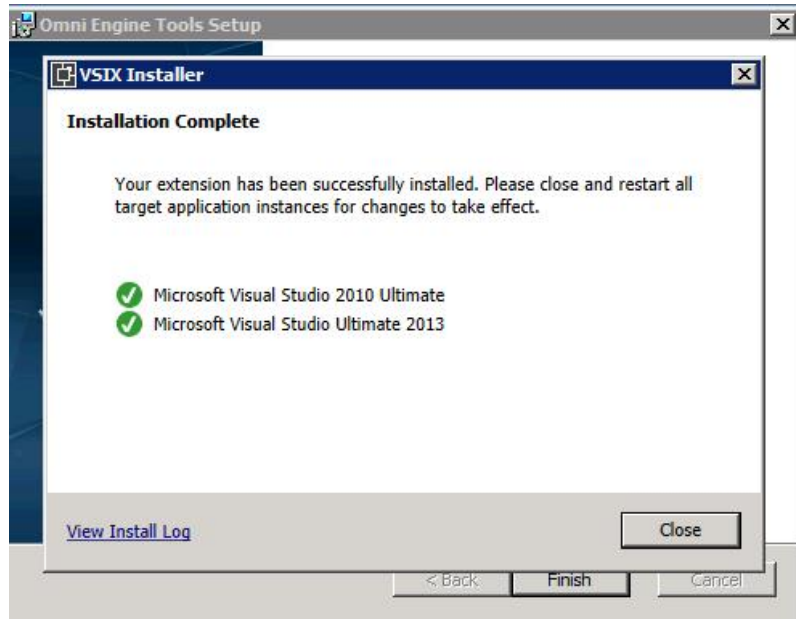


After the install finishes, the Visual Studio Extension installer will run. If you have a compatible version of Visual studio it will show a screen like.



Click "Install" to install the Visual Studio Extension.

# OMNI ENGINE



Click "Close" to finish the install.



Click "Finish"

# O M N I E N G I N E

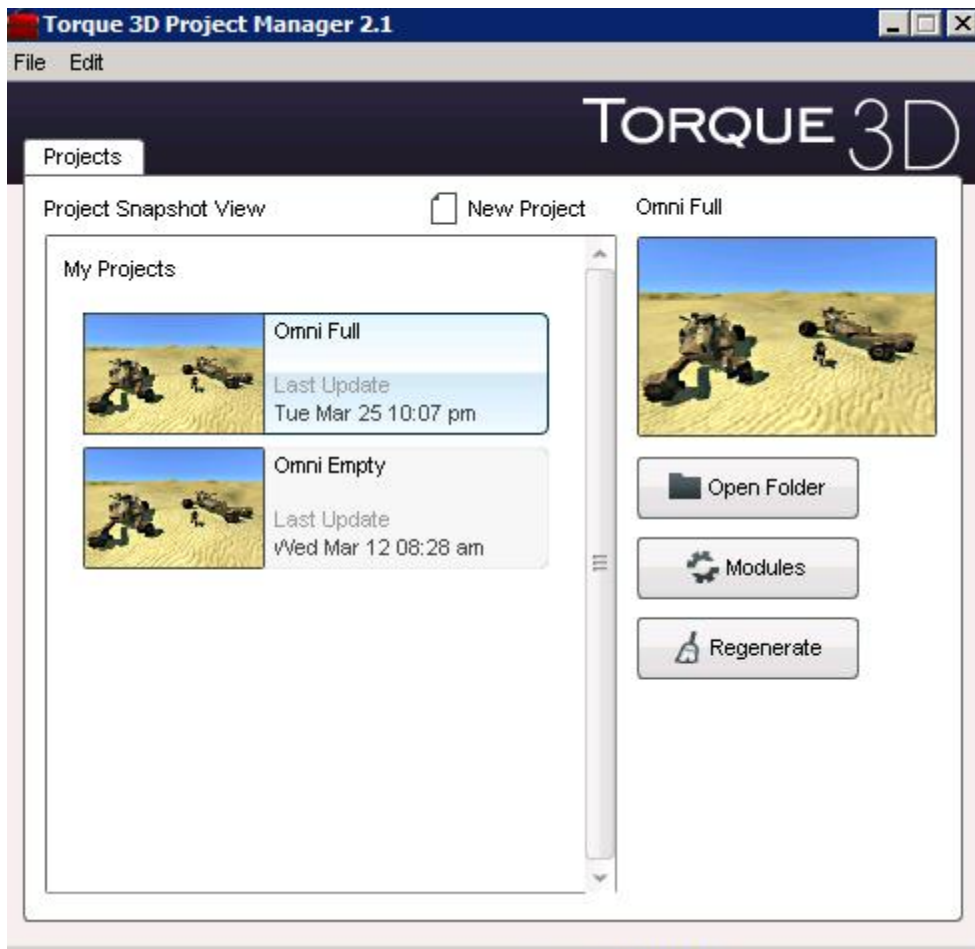
## Chapter 3 Creating your first Game.

After installing the software and SDK, the first process is to make your first game.

In the Omni SDK folder you will see a program called “Project Manager.exe”. This program is a toolbox that lets you create new games using the Omni Engine.

### Project Manager

Double click “Project Manager.exe”

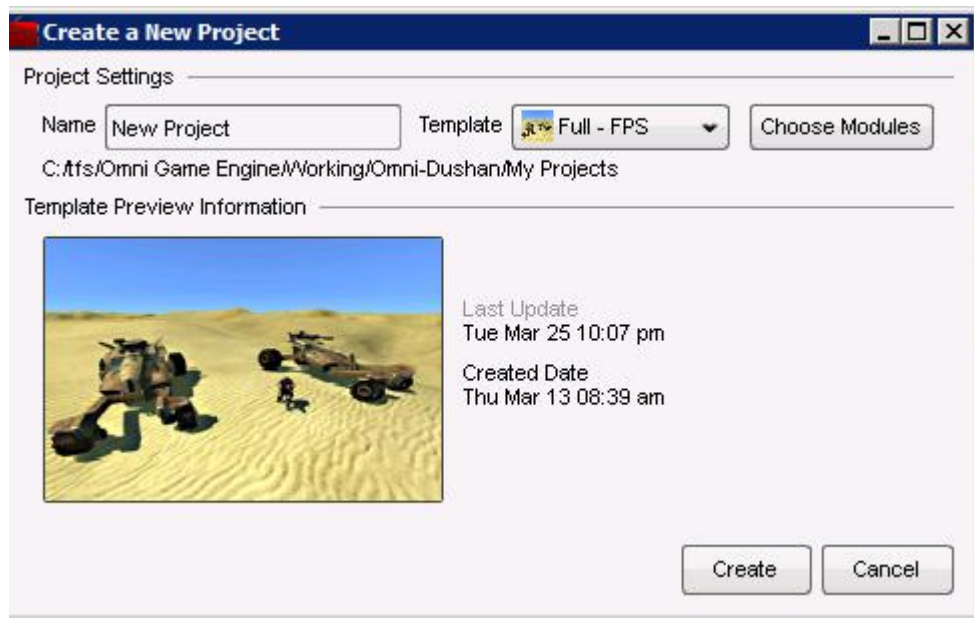


In the “My Projects” scrollable area you will see any projects you have already created. If this is your first time running the program, this area will be empty.

# OMNI ENGINE

New Project

To create your first project click “New Project”

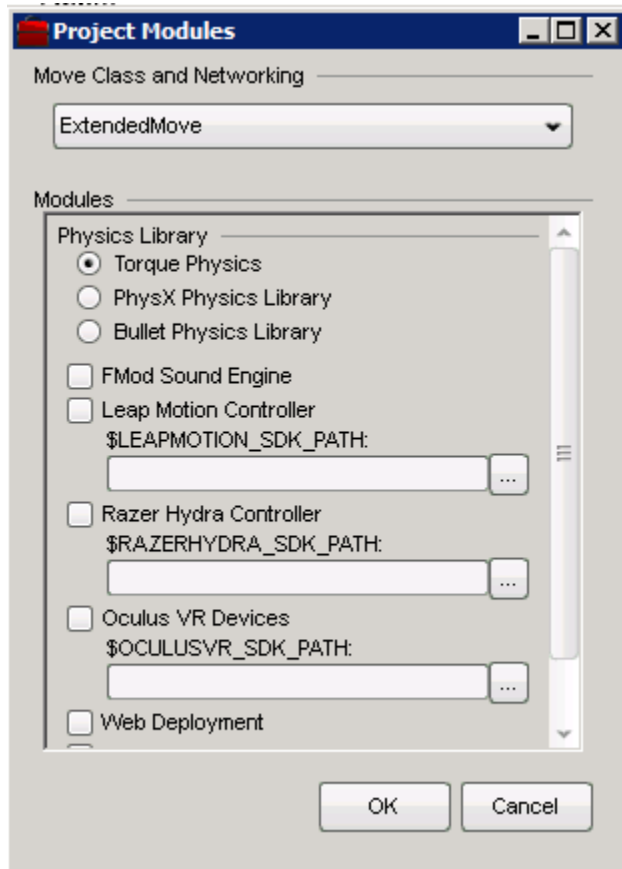


Enter a name for your new project in the “Name” text area.

Currently there are only four “Templates” available.

- C#-Full
  - This template contains all of the logic necessary to build a simple First Person shooter in C#.
- C#-Empty
  - This template is a bare-bones implementation of the engine in C#.
- TS-Full
  - This template contains all of the logic necessary to build a simple First Person shooter in TorqueScript.
- TS-Empty
  - This template is a bare-bones implementation of the engine in TorqueScript.

Click “Choose Modules”



It is always recommended to use the “Extended Move” class and networking. This is required to support the latest VR headgear.

You have two choices for Physics

- Torque Physics
  - Simple Physics internal implementation
- Bullet
  - Advanced physics via the “Bullet” Library.

If you wish to support FMod, click the option.

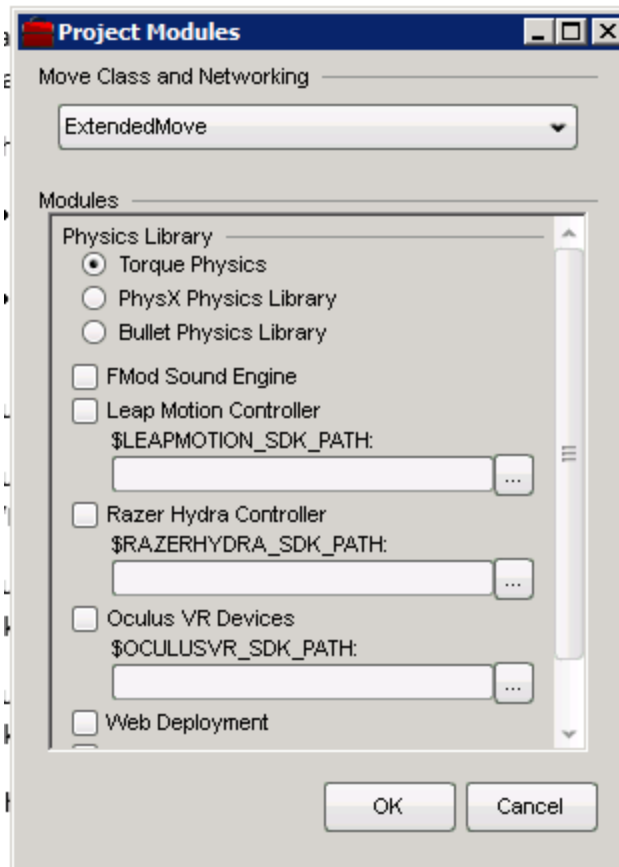
If you are planning on using “Leap” and you have the SDK installed, check the box and select the path to the “Leap” SDK.

If you are planning on using the “Razer Hydra Controller” and you have the SDK installed, check the checkbox and select the path.

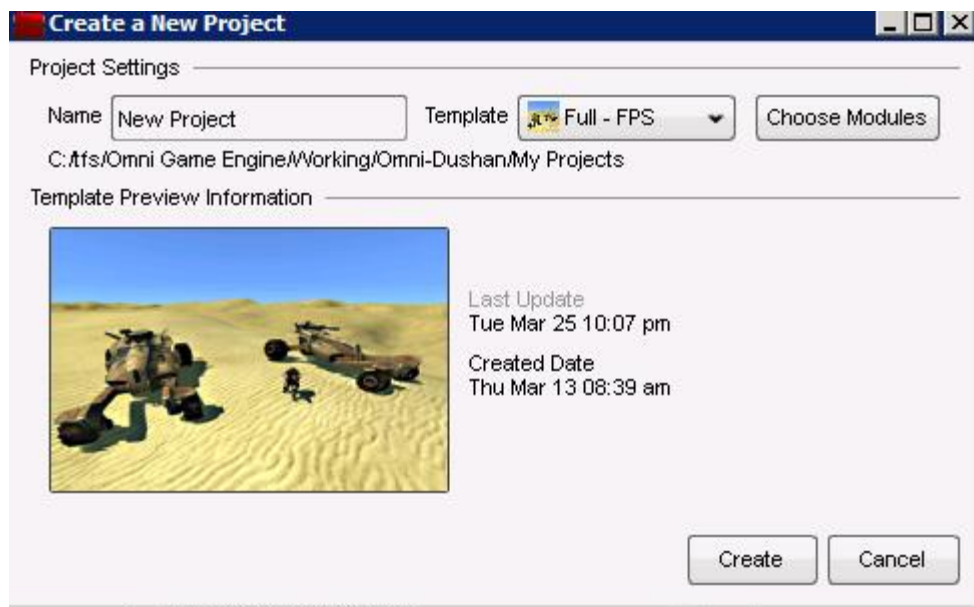
If you are planning on using the “Oculus VR Devices” and you have the SDK installed, check the checkbox and select the path to the sdk.

# OMNI ENGINE

For this example, let's just use the Torque Physics and the extended move class.

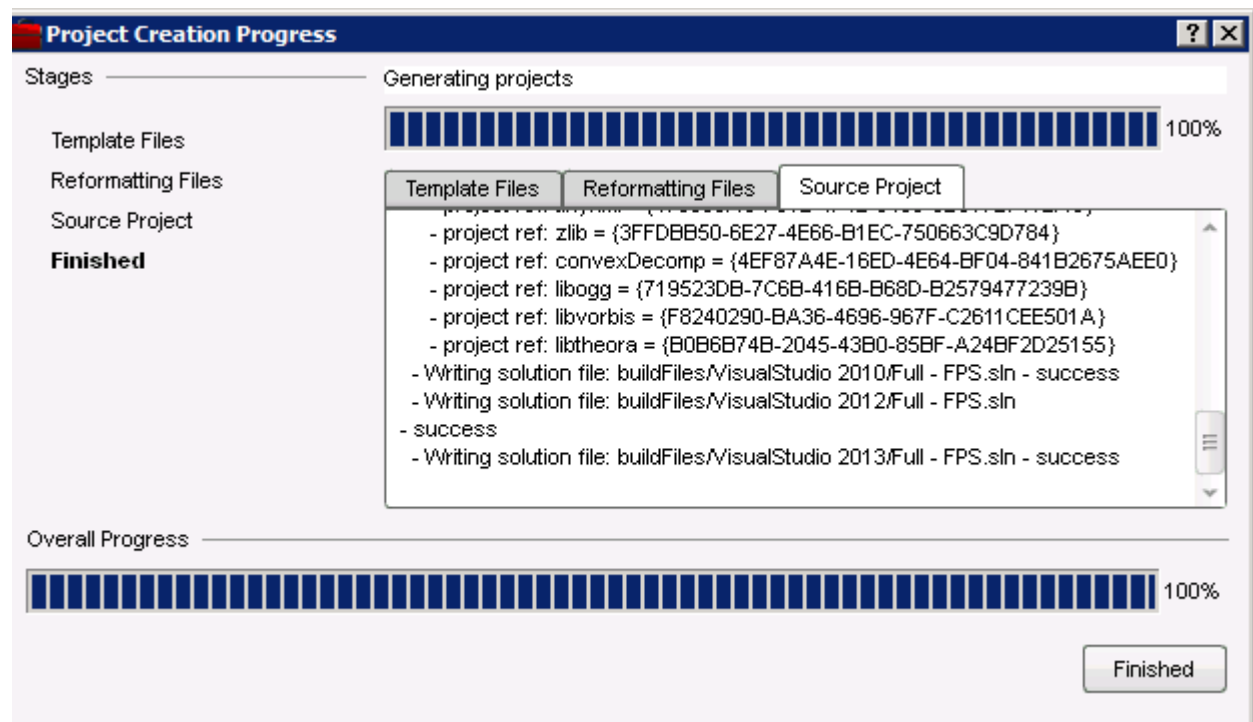
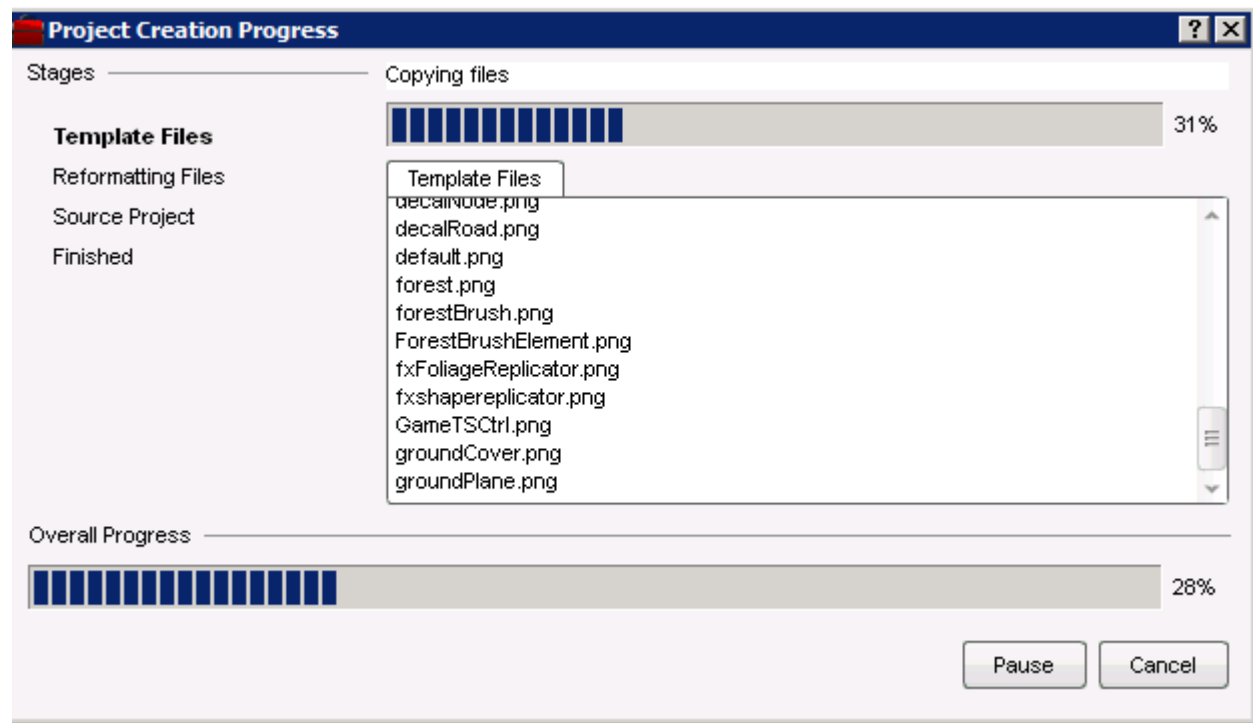


Click "OK"



# O M N I E N G I N E

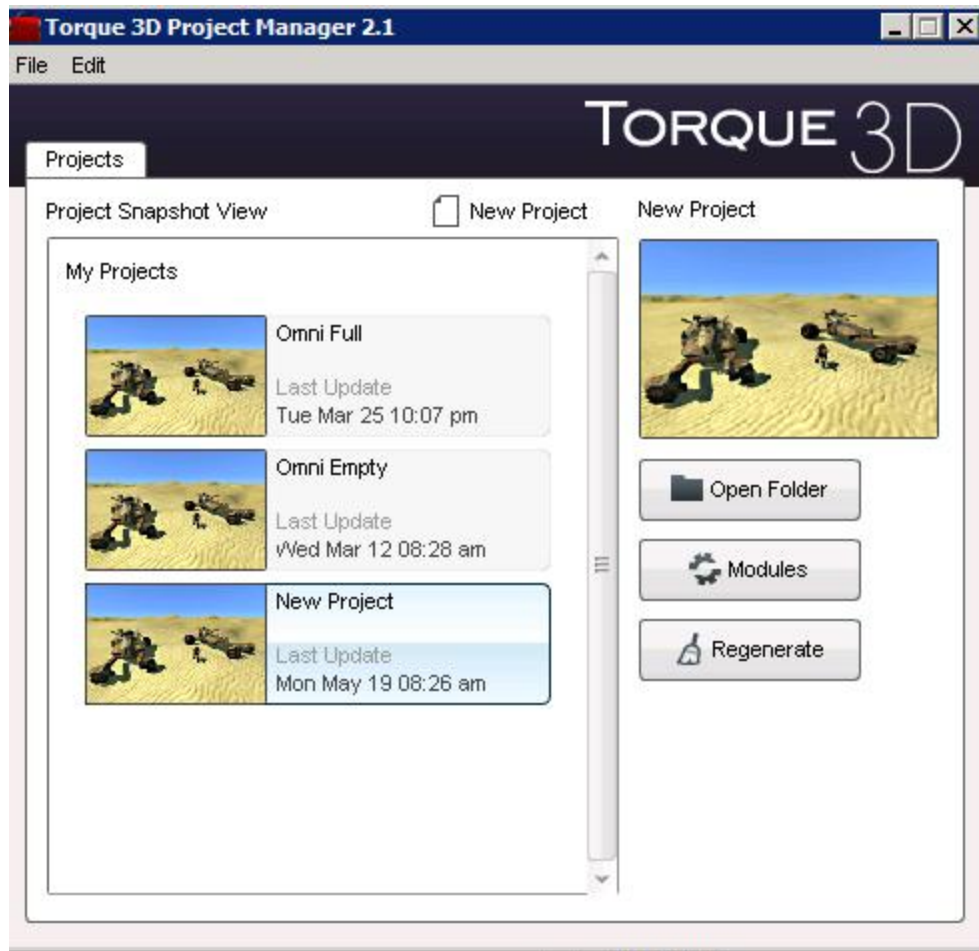
Click "Create"



Click "Finish"


















# OMNI ENGINE

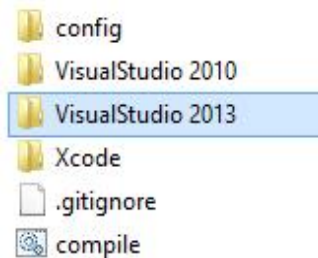


Click "Open Folder"

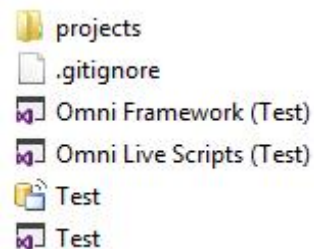
# O M N I E N G I N E

 buildFiles	10/9/2014 2:21 AM	File folder	
 game	10/9/2014 2:22 AM	File folder	
 source	10/9/2014 2:21 AM	File folder	
 Winterleaf.Demo.Full	10/9/2014 2:22 AM	File folder	
 Winterleaf.Engine.Omni	10/9/2014 2:22 AM	File folder	
 Winterleaf.Game	10/9/2014 2:22 AM	File folder	
 cleanShaders	10/8/2014 11:36 PM	Windows Batch File	1 KB
 cleanShaders.command	10/8/2014 11:36 PM	COMMAND File	1 KB
 DeleteCachedDTs	10/8/2014 11:36 PM	Windows Batch File	1 KB
 DeleteCachedDTs.command	10/8/2014 11:36 PM	COMMAND File	1 KB
 DeleteDSOs	10/8/2014 11:36 PM	Windows Batch File	1 KB
 DeleteDSOs.command	10/8/2014 11:36 PM	COMMAND File	1 KB
 DeletePrefs	10/8/2014 11:36 PM	Windows Batch File	1 KB
 DeletePrefs.command	10/8/2014 11:36 PM	COMMAND File	1 KB
 thumb	10/8/2014 11:36 PM	PNG image	51 KB

Open the folder “Build Files”



Open the “Visual Studio 2010” folder



# O M N I E N G I N E

## Generated Solution Files

There are three solution files in this folder. Each solution file is used for a different purpose.

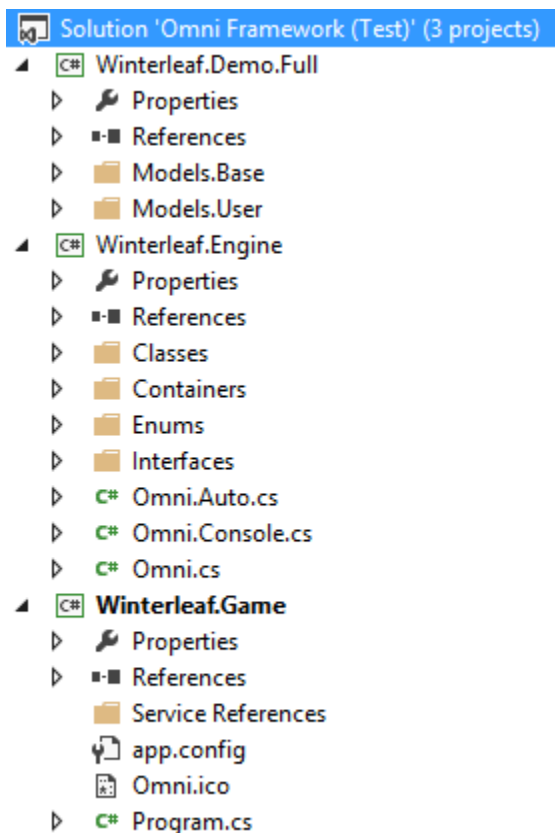
<Project Name>.sln – This is the C++ solution file.

Omni Framework.sln – Csharp game logic solution file

Omni Live Scripts.sln – Csharp runtime game logic solution file.

## Omni Framework Solution

The Omni Framework solution is comprised of three project files. Each project files fulfills a specific task. Of the three projects you will find most if not all of your work will be done in the “Model” class which is where the game logic resided. All of the projects in this solution are C#.



The three projects in this solution are:

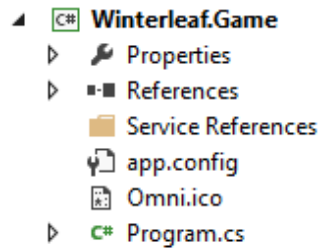
- Winterleaf.Demo.Full - This project contains the business logic to your game.

# OMNI ENGINE

- Winterleaf.Engine – This project is the Omni Framework project used to link C++ and C#.
- Winterleaf.Game – This project is the harness that generates the .exe that people will play the game with.

## *Winterleaf.Game Project*

This project file is the executable game project file.



It should contain an icon for your windows and a program file. The icon is used in any simulation window created.

The other file is the C# file which actually starts everything up. A sample file:

# OMNI ENGINE

```

3  #region
14 [assembly: SecurityRules(SecurityRuleSet.Level1)]
15 namespace Winterleaf.Game
16 {
17     internal static class Program
18     {
19         /// <summary>
20         ///     The main entry point for the application.
21         /// </summary>
22         private static Omni omni;
23
24         [STAThread]
25         private static void Main()
26         {
27             Application.EnableVisualStyles();
28             Application.SetCompatibleTextRenderingDefault(false);
29             try
30             {
31                 //Application.Run(new main_window());
32                 omni = new Omni(Process.GetCurrentProcess().Handle);
33                 //Initialize Torque, pass a handle to this form into T3D so it knows where to render the screen to.
34                 //If you don't do this, you can't pass the mouse and key strokes, w/out the mouse and keystrokes
35                 //being redirected the application will hang intermittently.
36
37                 #if DEBUG
38                     omni.Initialize(MessageBox.Show("Dedicated", "Dedicated", MessageBoxButtons.YesNo) == DialogResult.Yes ?
39                         new[] { "-dedicated", "-mission", @"levels/Empty_Terrain.mis" } :
40                         new[] { "" }, "Test_DEBUG", "WinterLeaf.Demo.Full.dll", "WinterLeaf.Demo.Full", "csScripts");
41                 #else
42                     ...
43                 #endif
44
45                 omni.Debugging = false;
46                 omni.ScriptExtensions_Allow = true;
47                 omni.ScriptExtensions_HandleExceptions = true;
48
49                 omni.DebuggingShowScriptCalls = false;
50                 omni.WindowIcon = new Icon("Omni.ico");
51                 while (omni.IsRunning)
52                     Thread.Sleep(1000);
53                 omni = null;
54             }
55             catch (Exception err)
56             {
57                 MessageBox.Show("An Error has occurred in the application. " + err.Message);
58             }
59             Application.Exit();
60         }
61     }
62 }

```

To make everything work you must create an instance of the Omni Framework. This is done on line 32 of the code.

Lines 37 through 44 are used to initialize the engine. I wrapped the code with the `#if DEBUG` so that it will switch the DLL's used depending on whether you are using a Debug or Release build. The parameters required to initialize the Omni Framework are:

- Commands to be passed to the C++ DLL.
- The name of the C++ DLL
- The name of the DLL for the C# business logic for your game
- The Namespace for your C# DLL

# OMNI ENGINE

- And optionally you can specify a directory for the engine to monitor where you can put C# scripts and edit them while the engine is running. (More on this later.)

After the Omni Framework is initialized you have several options. There are several features available inside the Omni framework that will assist you in developing your game. They are:

- Debugging – If this flag is set to true, then all interaction between the C# and C++ will be displayed in the console inside the game. This is useful when you are trying to figure out why a callback is not working.
- ScriptExtensions\_Allow – This flag turns on or off the runtime compilation functionality of the Omni Framework. If this is turned on then any C# scripts in the monitored folder (Last Parameter of the Initialization function.) will be compiled any time they changed while the engine is running.
- ScriptExtensions\_HandleExceptions – If this flag is enabled than any errors in the C# scripts will be caught by a wrapping Try/Catch block and the error will be displayed in the console. If this is turned off, any error in the scripts will cause a crash of the engine.  
Try/Catch blocks consume memory and CPU time since the .Net CLR must take a snap shot of all the variables and flags prior to executing the code. Because of this, production C# should handle all possible errors without the use of Try/Catches. It is highly recommended that this feature is turned off in production games.
- DebuggingShowScriptCalls – This flag controls whether or not C++ engine callbacks will be printed in the console. Once again, very useful for debugging.
- WindowIcon – This is the Icon file which will be displayed in the window while your game is running. Currently it uses the Icon image which is flagged as “Large”.

Finally, after it is initialized we want to enter a loop and continue to loop until the Omni Framework is no longer running.

Note: Currently it is not possible to create two instances of the Omni Framework. You can with minor changes to the framework reuse an instance.

The Omni Framework can also run in a WPF window, but to stay compliant with mono our example only shows Window Forms.

# OMNI ENGINE

## *Winterleaf.Engine Project (Omni Framework Engine)*

This project contains all of the code necessary for communications between the C++ and C#. Under normal circumstances you should not need to edit the files in this project. The only time the files are updated is if you need to run the Omni Static Code Generator.

There are several folders inside this project, they are:

- Classes
- Containers
- Enums
- Interfaces

## Classes

The class's folder contains the C# files used by the Omni Framework.

- [Dialogs.cs](#) – Used to show file dialogs.
- [arrayObject.cs](#) – Used for providing C# syntactical access to properties of Simulation objects which are defined as Arrays inside the engine.
- [ConsoleInteraction.cs](#) – This class provides the C# decoration logic which the Omni framework looks for to determine which functions to expose to the C++ console.
- [csFactory.cs](#) – This class is used to provide run-time compilation of C# scripts.
- [CustomClassDef.cs](#) – Internal class used by the Omni Framework to manage the mapping of C# functions and provide a cached reflection of the functions.
- [CustomQueue.cs](#) – A simple Queue implementation in C# which is designed for performance.
- [IndexingResult.cs](#) – Internal class used by the engine to return the found member and global functions generated when a run-time script is compiled and exposed in the engine.
- [LibraryManager.cs](#) – Used to determine if the code is executing on Mono/C# and Window/Linux. It will use the appropriate method to set up the P/Invokes based on this information.
- [MyExtensions.cs](#) – This class provides C# Extensions for object conversion. An example would be casting a Point3F to a string and vice-versa.
- [MyReflections.cs](#) – C# doesn't always know how to cast one type to another. This class handles all of the details of casting for the engine. C++ also has some casting that isn't available to C#, an example of this would be casting a Boolean to a integer, the string "True"/"False" to an integer or a Boolean, etc.
- [ObjectCreator.cs](#) – This class provides a method for creating Simulation objects.
- [pInvokes.cs](#) – This class is used to manage all of the p/Invokes used by the Omni Framework and expose them to developers using the Omni Framework. It should not be edited and it is automatically updated every time the Omni Static Code generator is used.

# OMNI ENGINE

- [ProxyObject.cs](#) - In the C++ the lowest Simulation object in the derivation tree is SimObject. In the Omni framework the lowest object is ProxyObject. It provides some base member functionality which all SimObject's expose.
- [SafeNativeMethods.cs](#) – Every p/Invoke is declared in this file. This file is updated by the Omni Static Code generator.
- [SingletonCreator.cs](#) – Much like the object creator, but used to create Singleton objects.
- [TypeConverterGeneric.cs](#) – This template class provides type conversion helpers used by the Omni Framework to cast object types from strings.
- [xmlOverrideData.cs](#) – This class is used by the csFactory.

## Containers

This folder contains the C# implementation of data classes implemented in the C++.

- [AngAxisF](#) – Contains an Angle, Axis X, Axis Y, Axis Z as floats
- [Box3F](#) – Contains the points in a box as floats.
- [ColorF](#) – Contains 3 floats used to determine color.
- [ColorI](#) – Contains 3 integers use to determine color.
- [EaseF](#) – No Clue
- [MatrixF](#) – Contains 3 sets of X, Y, Z coordinates of a matrix as floats.
- [MatrixPositions](#) – Contains W, X, Y, Z as floats.
- [Point2F](#) – Contains X, Y as floats.
- [Point2I](#) – Contains X, Y as integers.
- [Point3F](#) – Contains X, Y, Z as floats
- [Point3I](#) – Contains X, Y, Z as integers
- [Point4F](#) – Contains X, Y, Z, W as floats
- [Polyhedron](#) – Contains 4 Point3Fs
- [RectF](#) – Contains 2 X, Y positions as float
- [RectI](#) – Contains 2 X, Y positions as integers
- [RectSpacingI](#) – Contains Bottom, Left, Right, and Top as integers
- [TransformF](#) – Contains Orientation (X, Y, Z), Position (X, Y, Z) and Angle as floats.
- [TypeCubemap](#) – Used to handle this type defined in the C++
- [TypeImageFileName](#) – Used to handle this type defined in the C++
- [TypeMaterialName](#) - Used to handle this type defined in the C++
- [TypeName](#) – Used to handle this type defined in the C++
- [TypePrefabFilename](#) – Used to handle this type defined in the C++
- [TypeSFXAmbienceName](#) - Used to handle this type defined in the C++
- [TypeSFXDescriptionName](#) - Used to handle this type defined in the C++
- [TypeSFXEnviromentName](#) - Used to handle this type defined in the C++
- [TypeSFXParameterName](#) - Used to handle this type defined in the C++
- [TypeSFXSourceName](#) - Used to handle this type defined in the C++



# OMNI ENGINE

- [TypeSFXStateName](#) - Used to handle this type defined in the C++
- [TypeSFXTractName](#) - Used to handle this type defined in the C++
- [TypeShapeFilename](#) - Used to handle this type defined in the C++
- [TypeStringBase](#) - Used to handle this type defined in the C++
- [TypeTerrainMaterialName](#) - Used to handle this type defined in the C++
- [Vector](#) – The C++ engine supports three types of vectors. (Integer, Float, and Boolean) This class simplifies accessing them via C#.

## Enumerations

This folder contains the enumerations exposed in the C++ console in C#.

- [domUpAxisType](#) – Enumeration for the TSStatic maskbits.
- [GuiGraphType](#) – Graphic enumeration.
- [SceneObjectTypes](#) – All object type masks in the engine.
- [T3D\\_Enums](#) – This file is generated by the Omni Static Code Generator. It exposes all of the C++ console types in a format that is friendly with C#.

## Interfaces

This folder holds all class interfaces used in the Omni Framework.

- [iEnum](#) – An enumeration which all custom enumerations support for automation inside the Omni Framework.

## Remaining files in the Winterleaf.Engine

- [Omni.Auto.cs](#) – This file is autogenerated using the Omni Static Code Generator. It contains the wrappers for all the P/Invokes inside the engine.
- [Omni.Console.cs](#) – This file contains custom implementations for getting/setting values inside the engine. This code is pretty static and is not regenerated when the Omni Static Code Generator is run.
- [Omni.cs](#) – The heart of the Omni Framework, this file is the coordinator for all objects and management of the Omni C++ DLL. It contains the logic for passing data back and forth between the C++ and C#.

## *Winterleaf.Demo.Full or Winterleaf.Demo.Empty*

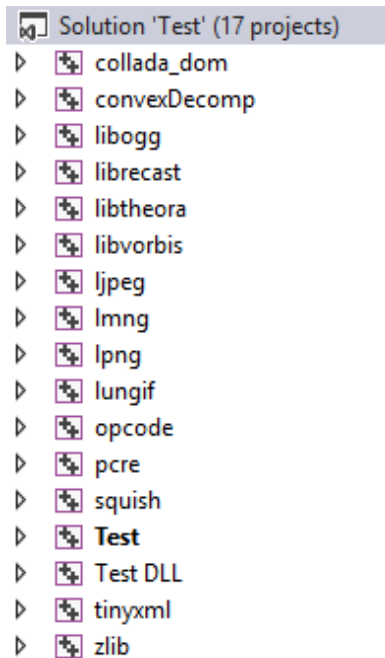
This project contains the unique business logic that runs the game. It is completely object oriented. There is a base file and folder structure that is required for the project to work with the Omni Framework.

# OMNI ENGINE

## <Project Name>.sln Solution

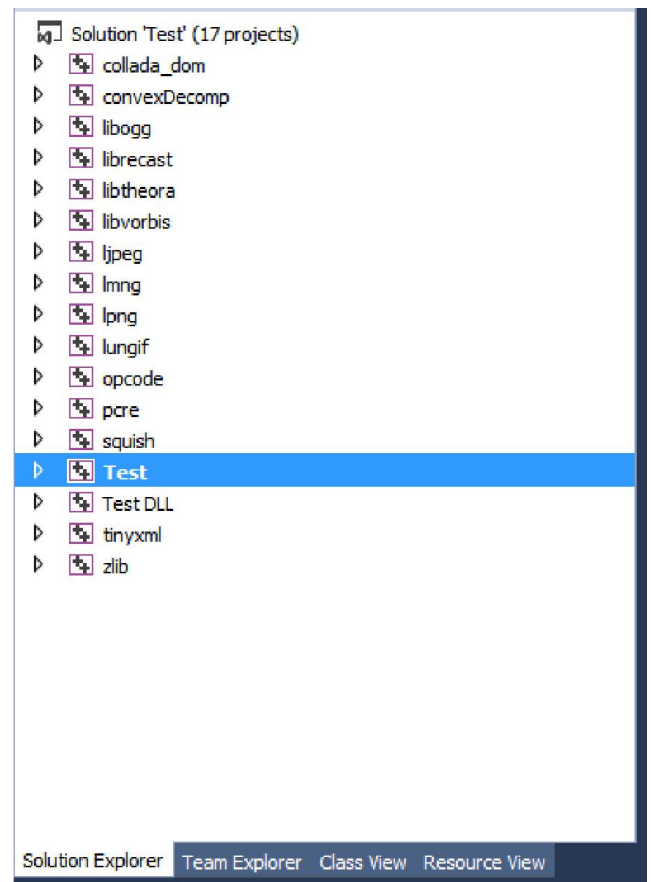
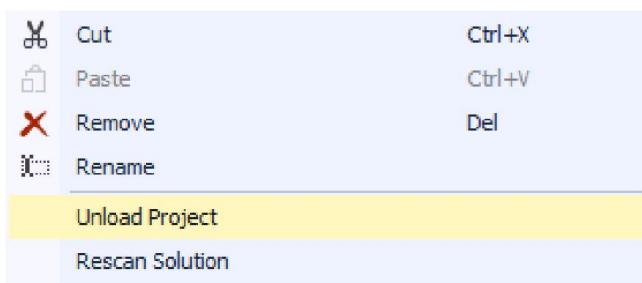
This solution is Winterleaf Entertainment's branch of the MIT T3D engine SDK. Full documentation on this solution can be found at <http://www.garagegames.com>. This solution contains all of the C++ source code used in the rendering engine which the OMNI Framework encapsulates.

After you create your own project and open this solution you will see a project called "<Project Name> DLL" This C++ project is the branched C++ MIT T3D Engine code.



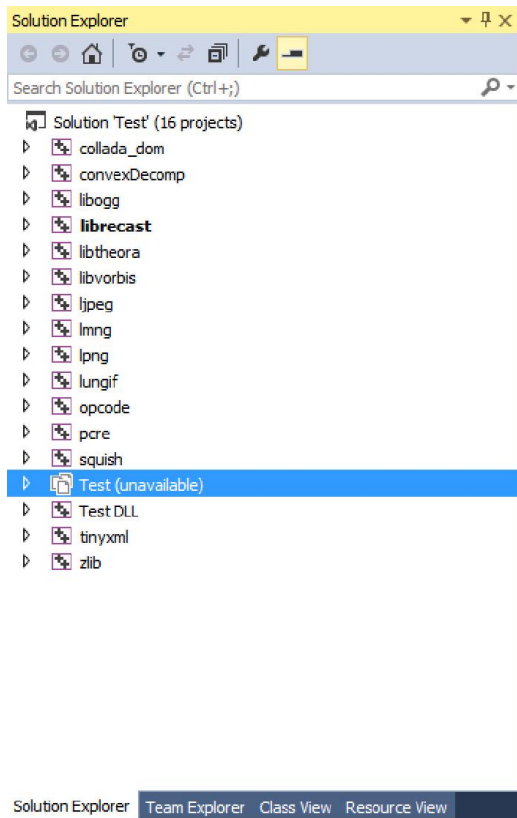
If you are planning to use C#, you can unload or turn off building the standard T3D exe project since it won't be used.

To do this, right click on the "Test" project and click "Unload".

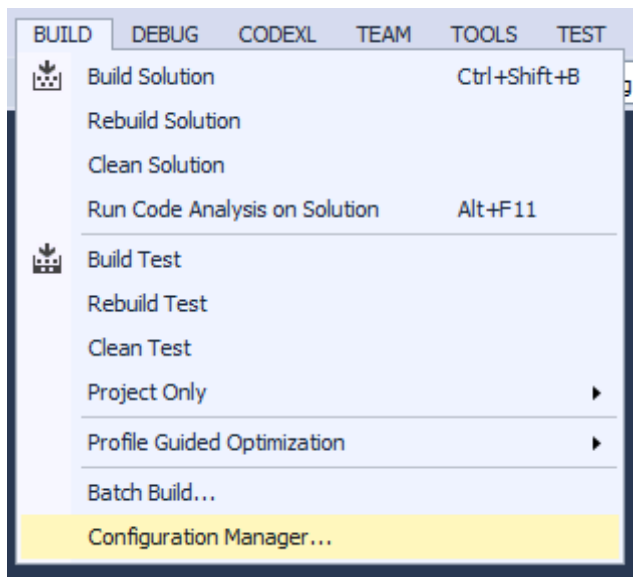


# OMNI ENGINE

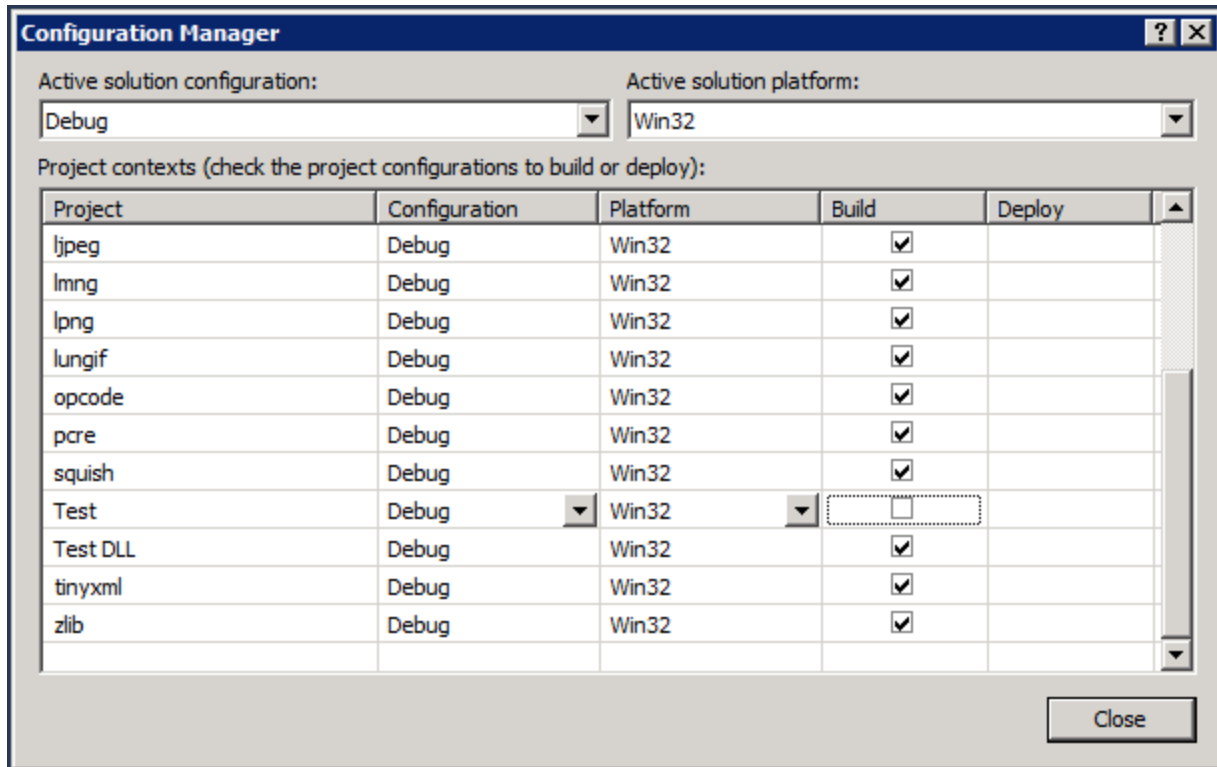
After clicking “Unload” the project should look like the picture below.



If you do not wish to unload the project, you can alternatively turn off the Build of the C++ Executable via the Build Configuration.



# O M N I E N G I N E



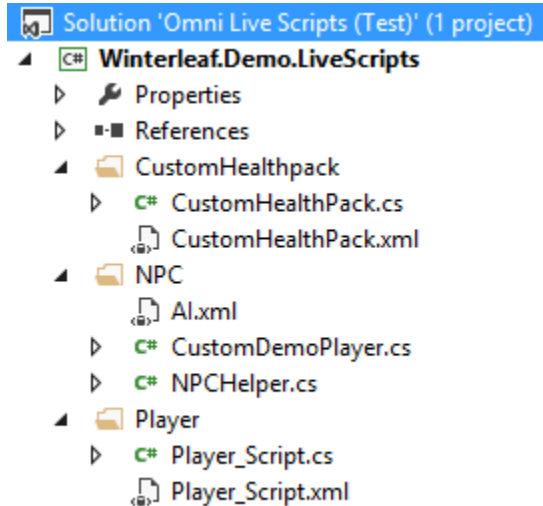
Uncheck the checkbox next to the C++ executable project.

# O M N I E N G I N E

## Omni Live Scripts Solution

The only project in this solution is “Winterleaf.Demo.LiveScripts”. The purpose of this solution is to simplify and provide intellesense for writing code which can be re-compiled while the engine is running so that you do not have to stop and restart the engine every time you wish to fix something.

There are three examples of how live scripts work in this project.



The three examples are:

- Extending the Player Object (coPlayer folder)
- Creating a custom “HealthPack” proxy object based off of the stock “HealthPack” (CustomHealthpack folder)
- A custom “NPC” proxy object based off the base “NPC” object. (NPC Folder)

The files are not compiled into a DLL and included in the engine at run-time, but instead they are read by the engine at run-time and are compiled into in-memory types. These types can then be used just like any other object except they can be recompiled in memory at any time by just re-saving the source code or XML.

## Static Code Generator - Update the Omni Framework Code

This section will discuss how you update the Omni Framework C# and C++ code. This process needs to be repeated every time you modify the C++ MIT T3D project.

For more advanced usages of the Omni T3D Engine you will find that you have to modify the C++ MIT T3D code base. This could be for many reasons including adding a new SimObjects, console functions, etc. Sooner or later at some point you will need to dig into the C++ code to customize it for your particular game.

When this happens, if you do not re-run the Static Code Generator, your C# proxy objects will become out of sync with the C++ objects. This program parses the entire MIT T3D engine source code and extracts information about the objects, variables, intellesence, enumerations, etc.

The Static Code Anaylzer WILL NOT overwrite your custom code, the code has been structured in a format so that it can be regenerated at any time without modifying/erasing custom logic you added to the C# Game Logic Project (Controller).

This program is provided in two formats, one is a standalone application and the other is a Visual Studio extension. Both of which are installed during Chapter 2.

**Important: If you are using the “Free” versions of Microsoft Visual Studio, you will need to use the stand-alone version of the Static Code Analyzer.**

## Static Code Generator (Visual Studio Extension)

This section explains how to use the Visual Studio Extension provided by Winterleaf Entertainment to run the Static Code Generator Visual Studio Extension.

### Step 1, Check out the source code (if applicable)

If you are using a source control plugin such as Microsoft Team Foundation Server or any other Source Control solution which makes the files read-only, you must check out the entire **“engine\source”** folder. The Static Code Analyzer will not only need to read these files but also append code to the end of them.

If you are using the stand-alone version of the Static Code Analyzer you will also need to check out the following folders in the game folder:

- **“My Projects/<Game Name>/Winterleaf.Demo.Full”**
- **“My Projects/<Game Name>/Winterleaf.Engine.Omni”**

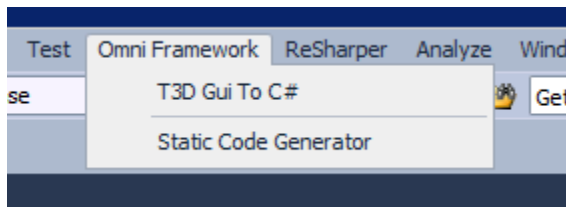
### Step 2, Open the “Omni Framework Solution”

Go to the **“c:\Omni\My Projects\<Game Name>\buildFiles\VisualStudio 2010”** folder

Open the solution **“Omni Framework.sln”**

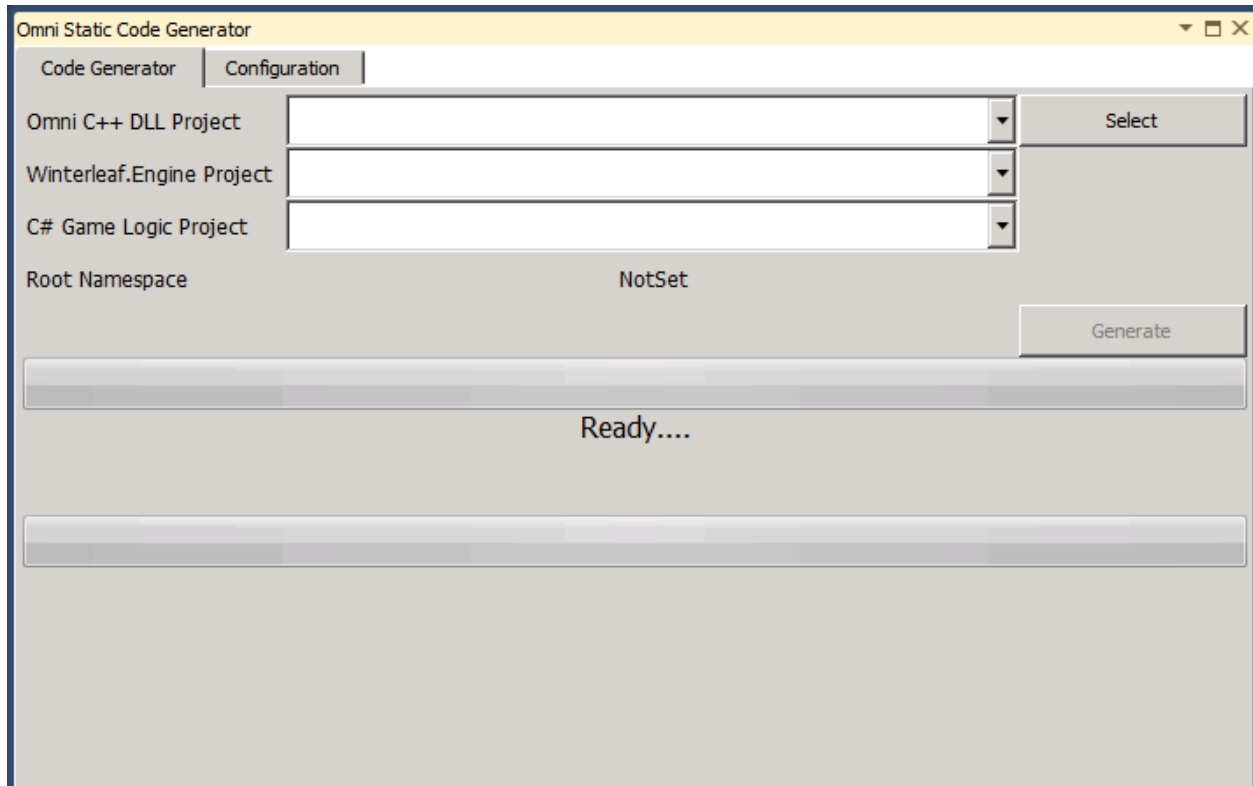
### Step 3, Open the Static Code Generator Extension

You will find the button on your Visual Studio toolbar right after the test option.



After clicking the button you will see the following tool window.

# OMNI ENGINE



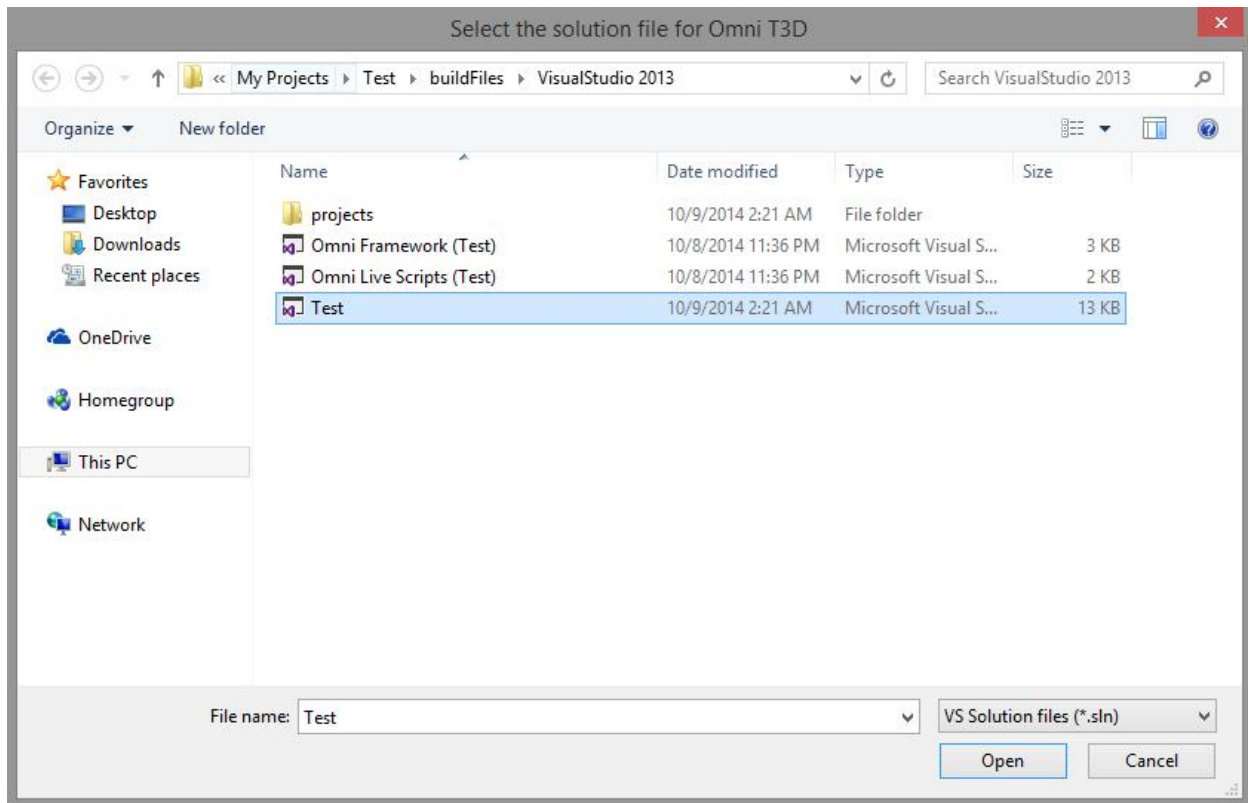
## Step 4, Select the Omni C++ DLL project

The first thing you must do is open the MIT T3D SDK solution file.

To do this, click "Select", and then pick the <Game Name> solution file.



# OMNI ENGINE



Click “Open”.

## Step 5, Select the C++ DLL Project

In the “Omni C++ DLL Project” dropdown select the project that is named “<Project Name> DLL”.



## Step 6, Select the Winterleaf.Engine project.

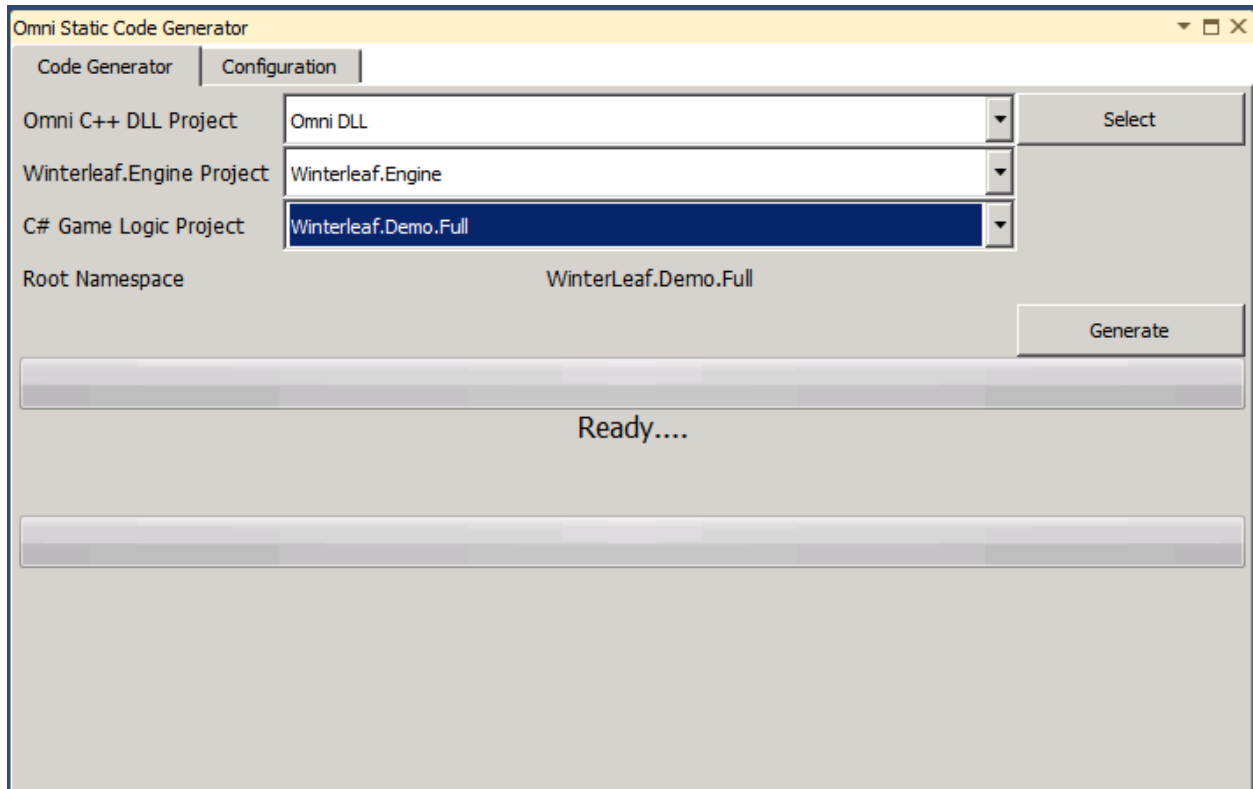


## Step 7, Select the C# Game Logic Project



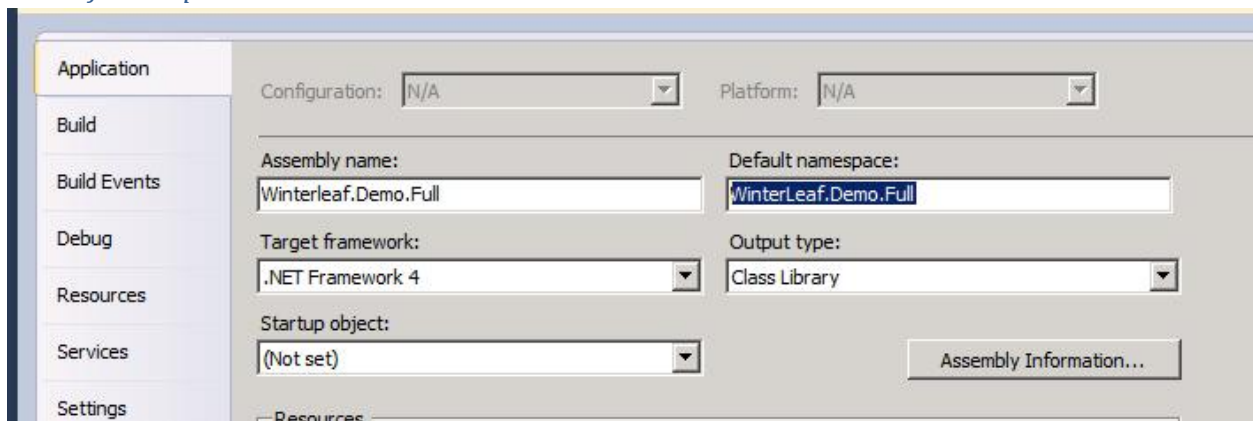
# OMNI ENGINE

## Step 8, Review the configuration



It is important, that the “Root Namespace” is correct. An incorrect “Root Namespace” can cause the game to appear to not work since the Omni Framework uses a great deal of reflections to achieve its functionality. So if you create your own C# Game Logic project, make sure the “Root Namespace” in the project properties match the “Root Namespace” in your C# code.

### *C# Project Properties*



# OMNI ENGINE

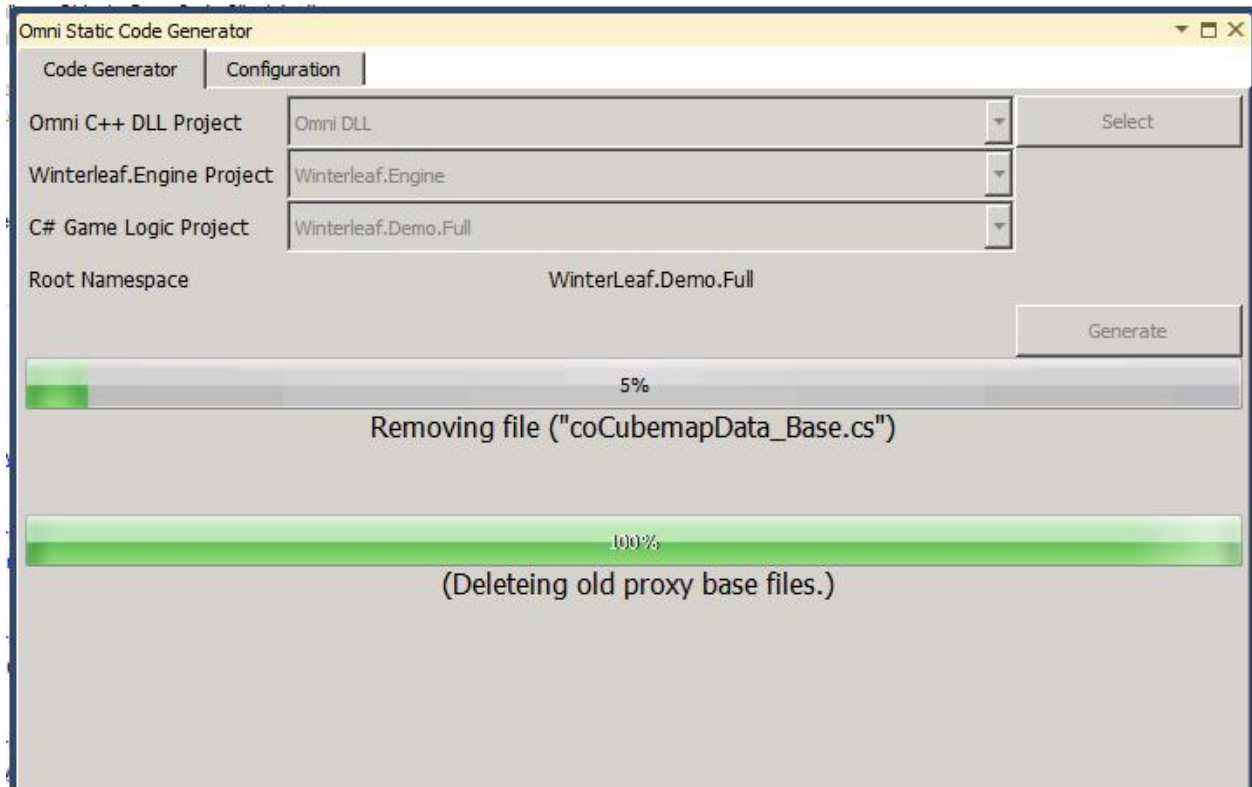
## Sample Code File

```

1  #region
2
3  using System.ComponentModel;
4  using System.IO;
5  using System.Linq;
6  using System.Threading;
7  using System.Windows.Forms;
8  using WinterLeaf.Demo.Full.userObjects.GameCode.Client;
9  using WinterLeaf.Demo.Full.userObjects.GameCode.Client.Audio;
10 using WinterLeaf.Demo.Full.userObjects.ProxyObjects;
11 using WinterLeaf.Engine;
12 using WinterLeaf.Engine.Classes;
13 using WinterLeaf.Engine.Containers;
14
15 #endregion
16
17 namespace WinterLeaf.Demo.Full.userObjects.GameCode
18 {
19     /// <summary>
20     /// This is a required file, replaces main.cs in the root directory
21     /// </summary>
22     ///
--

```

## Step 9, Click "Generate"



# O M N I E N G I N E

The first thing the Static Code Generator will do is remove all of the old generated code from the project to prevent artifacts carrying over from major C++ changes. Once this is complete, the Static Code Generator will read the C++ source code and generate all of the necessary C# files.

If the code generator experiences an error, a message will be displayed on screen and the detail of the error will be in the job log text file which pops up after the Static Code Generator completes.

## Step 10, Recompile

You will need to recompile the following solutions:

- <Project Name>.sln
- Omni Framework.sln

# OMNI ENGINE

## Static Code Generator (Stand-Alone)

This section mainly applies to programmers who are using the “Free” version of Microsoft Visual Studio, or just prefer to run stand-alone applications.

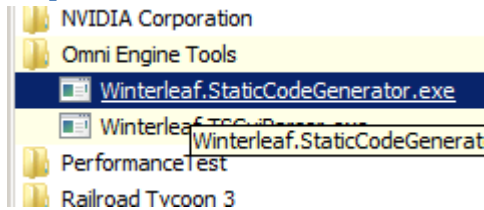
### Step 1, Check out the source code (if applicable)

If you are using a source control plugin such as Microsoft Team Foundation Server or any other Source Control solution which makes the files read-only, you must check out the entire **“engine\source”** folder. The Static Code Analyzer will not only need to read these files but also append code to the end of them.

If you are using the stand-alone version of the Static Code Analyzer you will also need to check out the following folders in the game folder:

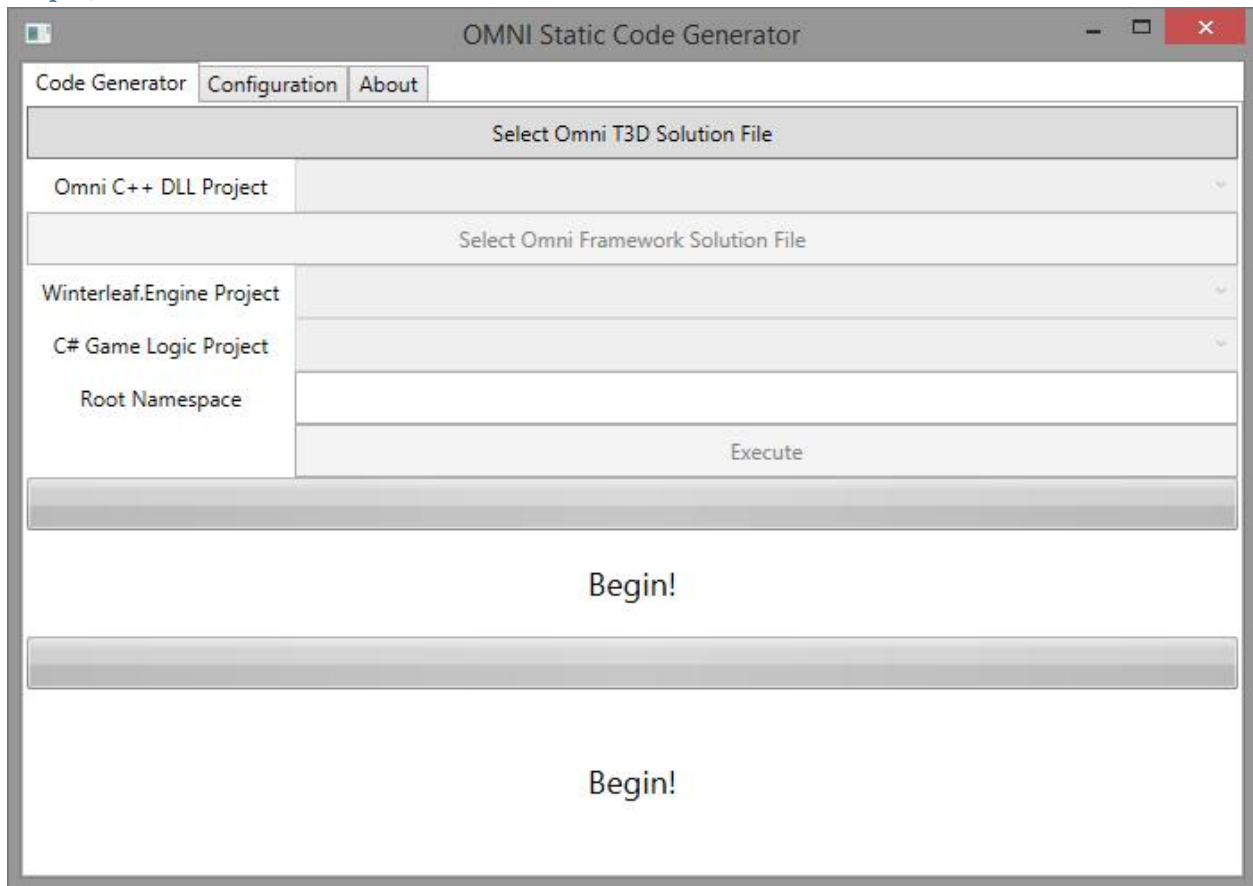
- **“My Projects/<Game Name>/Winterleaf.Demo.Full”**
- **“My Projects/<Game Name>/Winterleaf.Engine.Omni”**

### Step 2, Start the Static Code Generator (Stand-Alone)



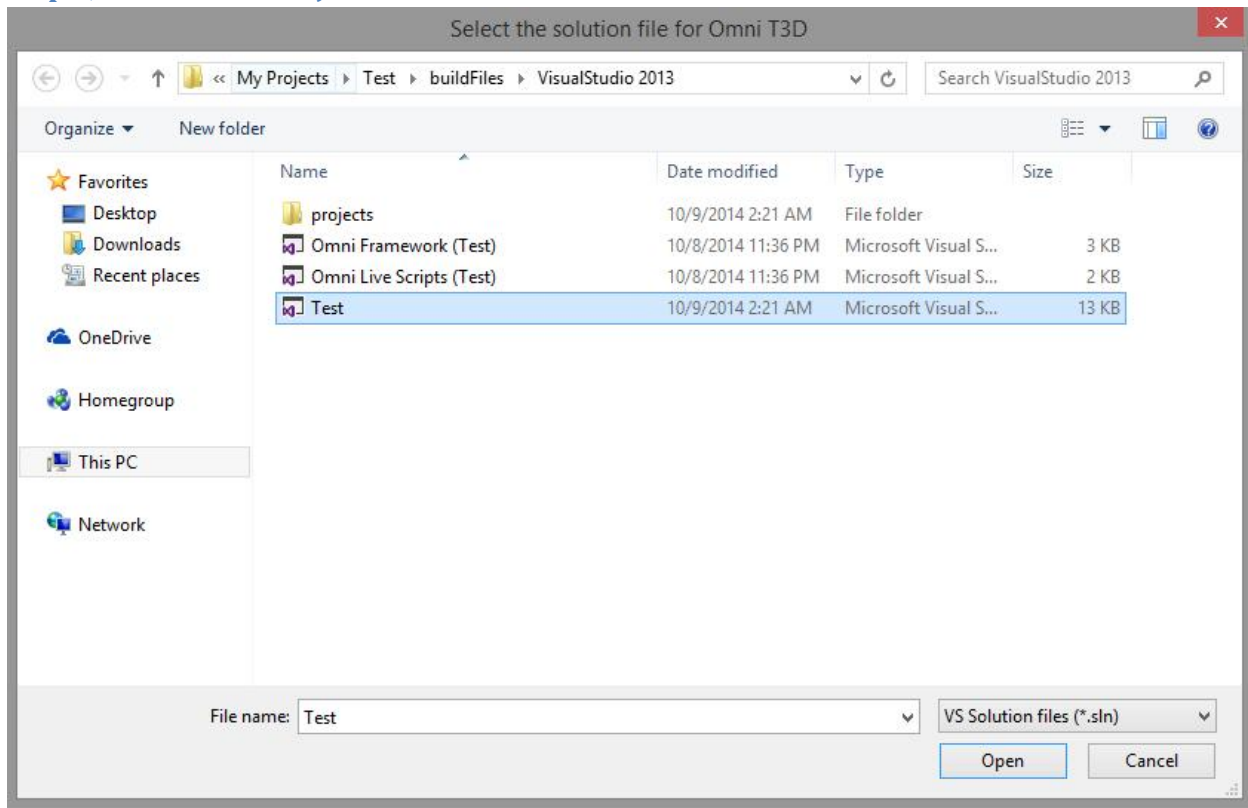
# OMNI ENGINE

## Step 3, Click “Select Omni T3D Solution File”



# OMNI ENGINE

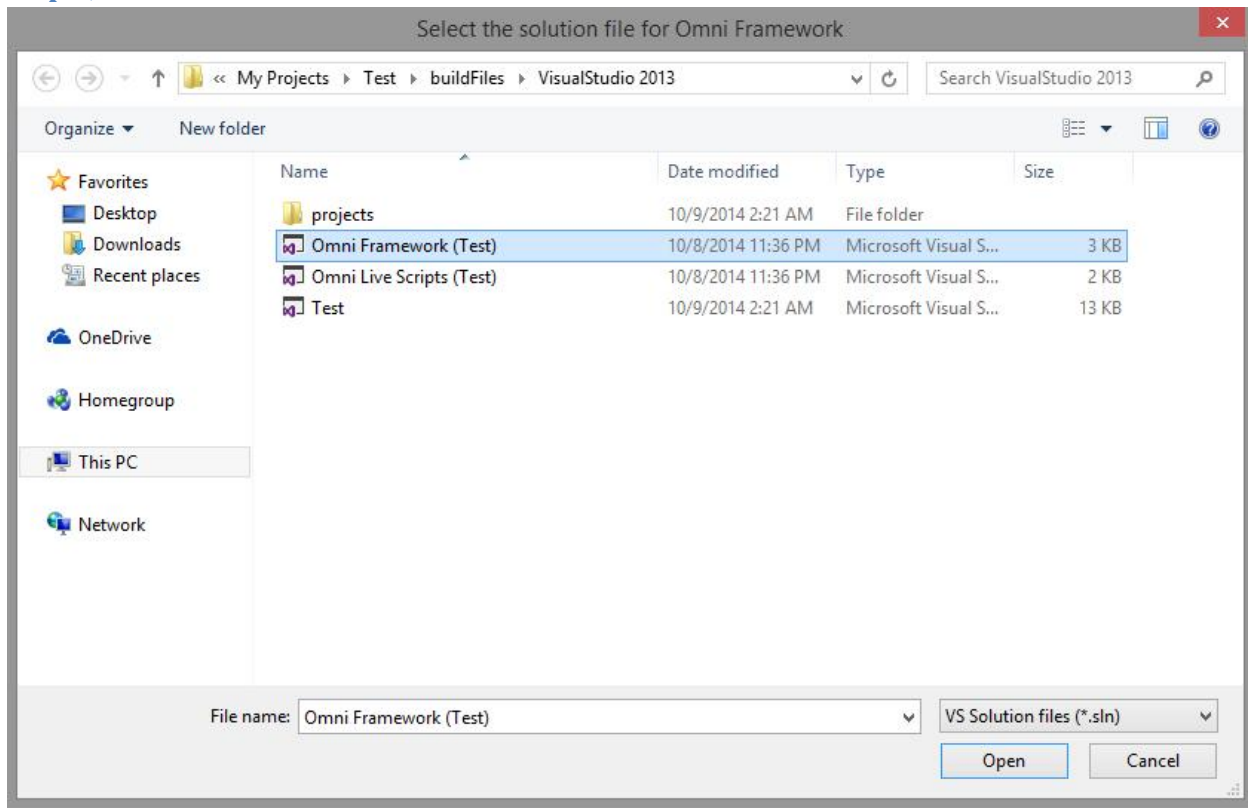
Step 4, Select the "<Project Name>.sln" file.



Click "Open"

# O M N I E N G I N E

## Step 5, Click “Select Omni Framework Solution”

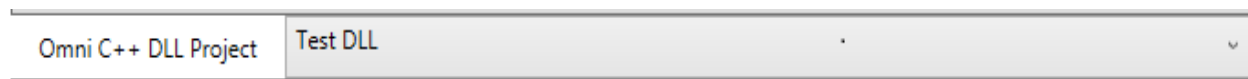


Select the “Omni Framework.sln” file.

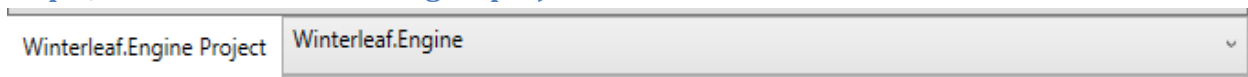
Click “Open”.

## Step 6, Select the C++ DLL Project

In the “Omni C++ DLL Project” dropdown select the project that is named “<Project Name> DLL”.



## Step 7, Select the Winterleaf.Engine project.



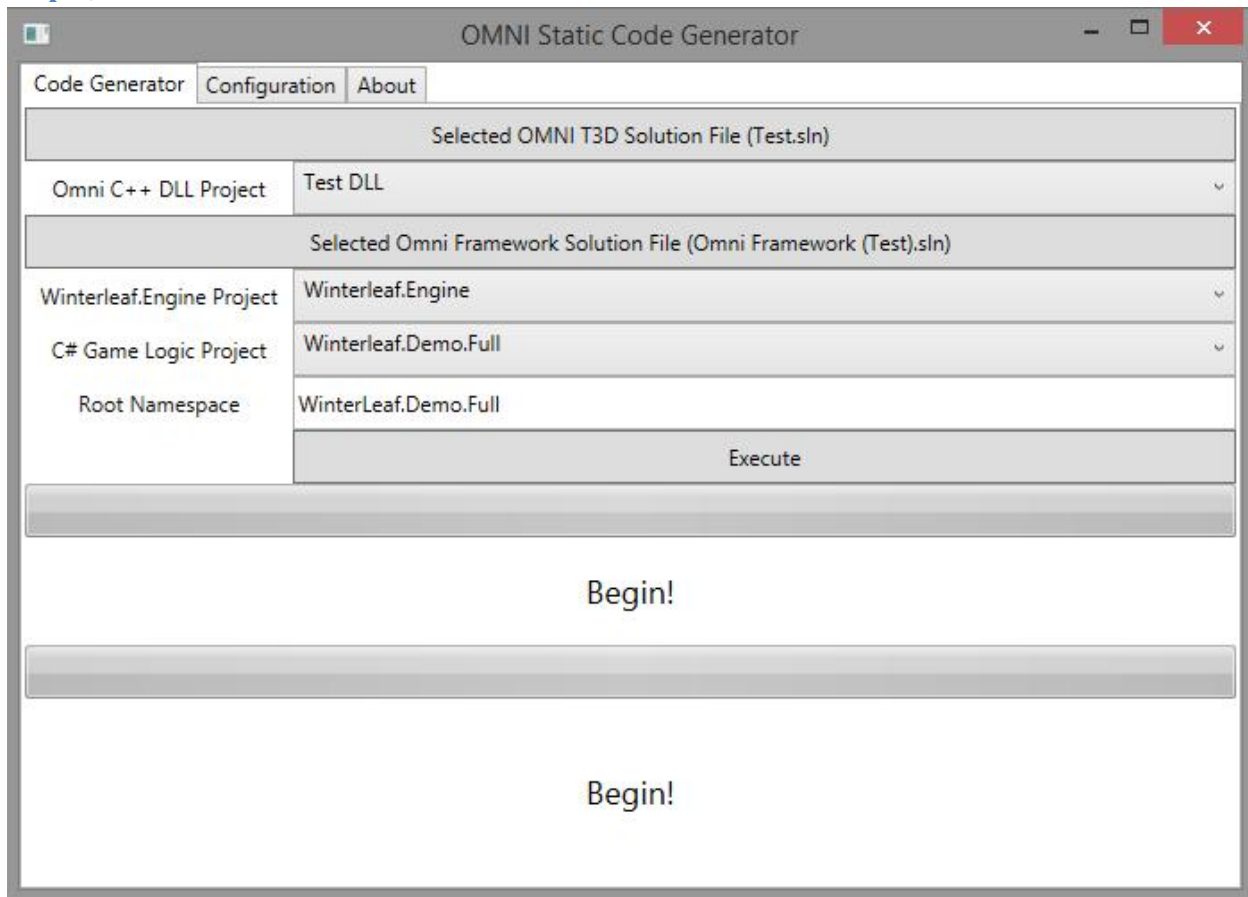
## Step 8, Select the C# Game Logic Project





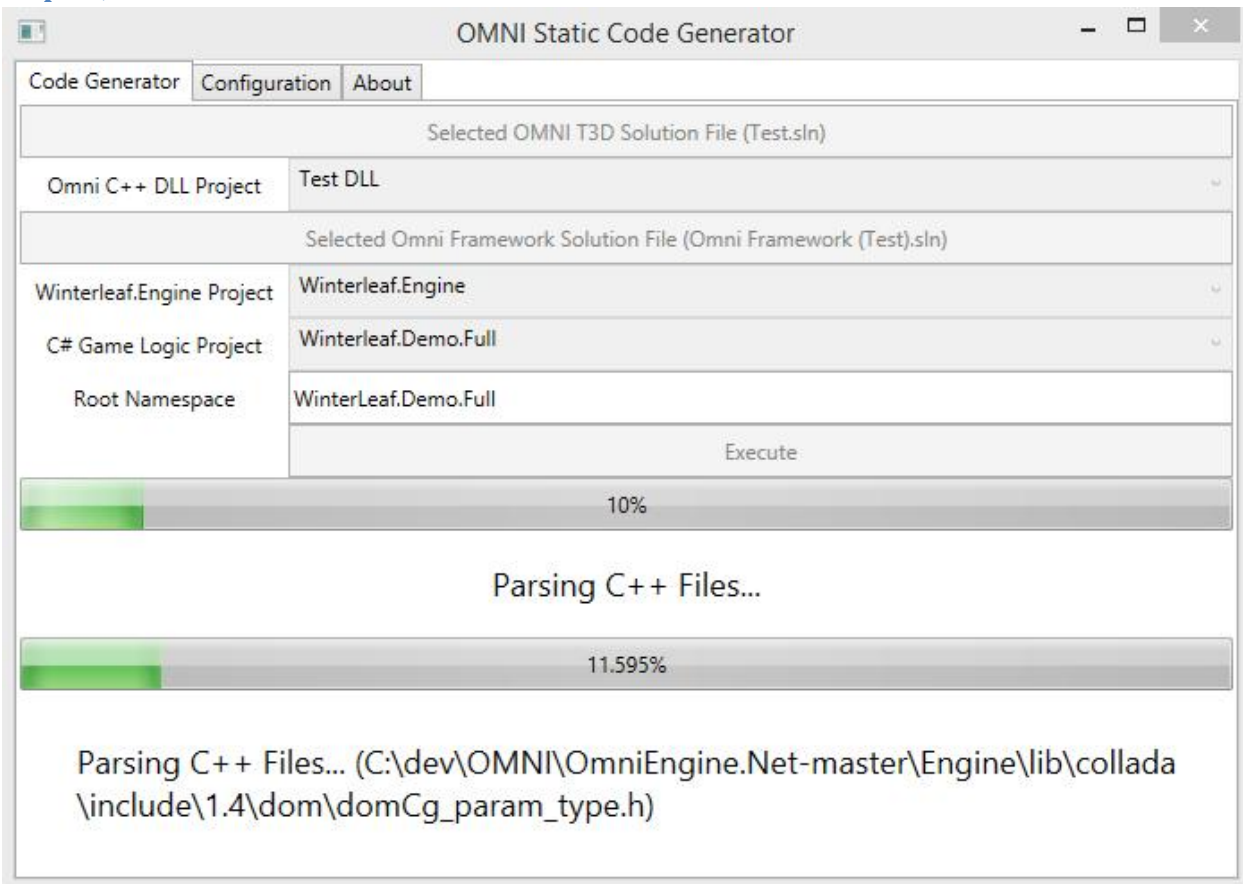
# OMNI ENGINE

## Step 9, Click “Execute”



# OMNI ENGINE

## Step 10, Wait for the Static Code Generator to finish.

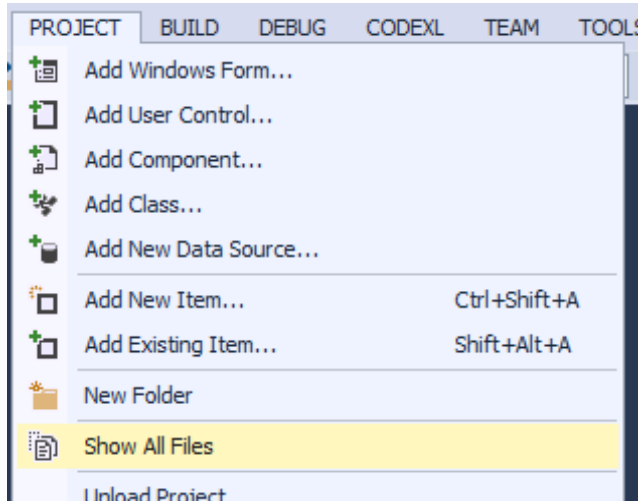


## Step 11, Add new files to the “C# Game Logic Project”

The stand-alone application does not have the ability to modify the project files to add new classes to it. Because of this you must open the project up and manually add any new class files which might have been generated.

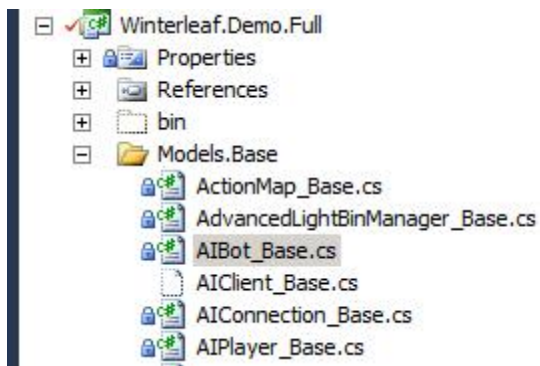
To do this, open the “Winterleaf.Demo.Full” project and have it show all files.

# OMNI ENGINE



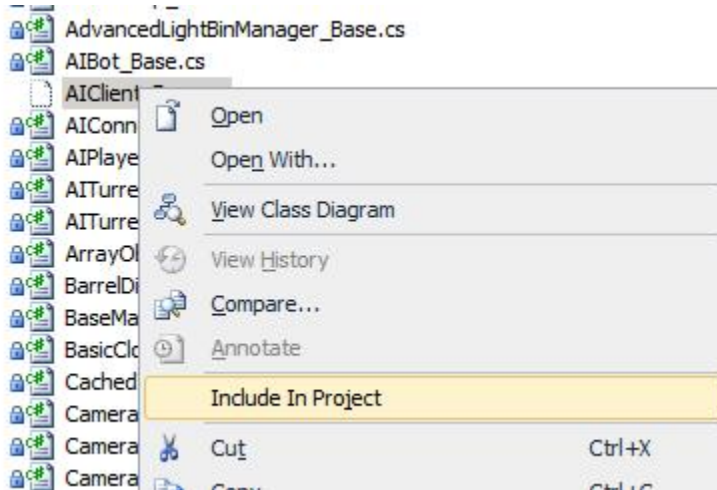
New files generated by the Static Code Generator will appear grayed out.

**Note:** If you haven't added any new SimObject C++ types to the engine, then there won't be any greyed out files.



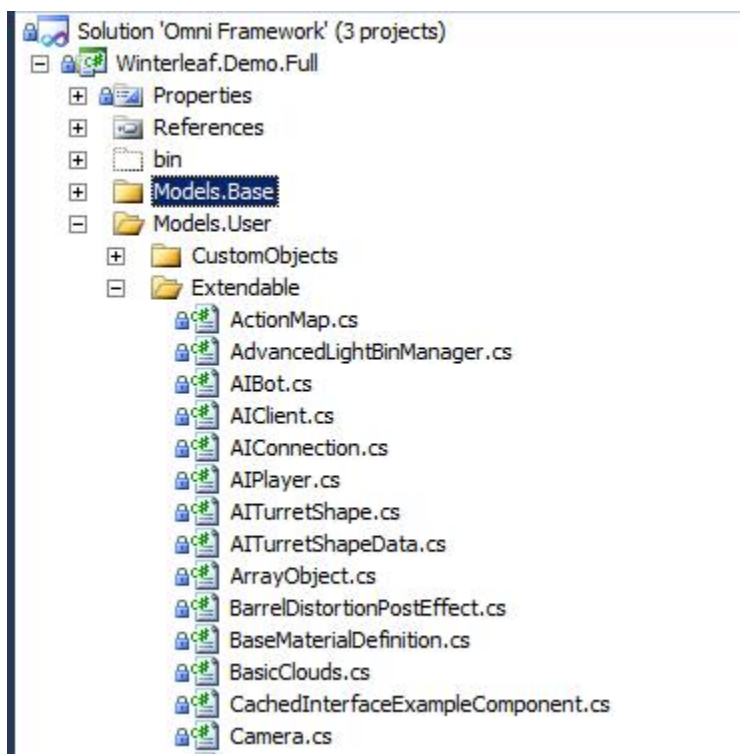
Right click the file and click "Include in Project"

# OMNI ENGINE



Repeat this process for all the grayed out files in the “Models.Base” folder.

Next go to the “Models.User/Extendable” folder and repeat the process.



## Step 12, Recompile

You will need to recompile the following solutions:

- <Project Name>.sln
- Omni Framework.sln

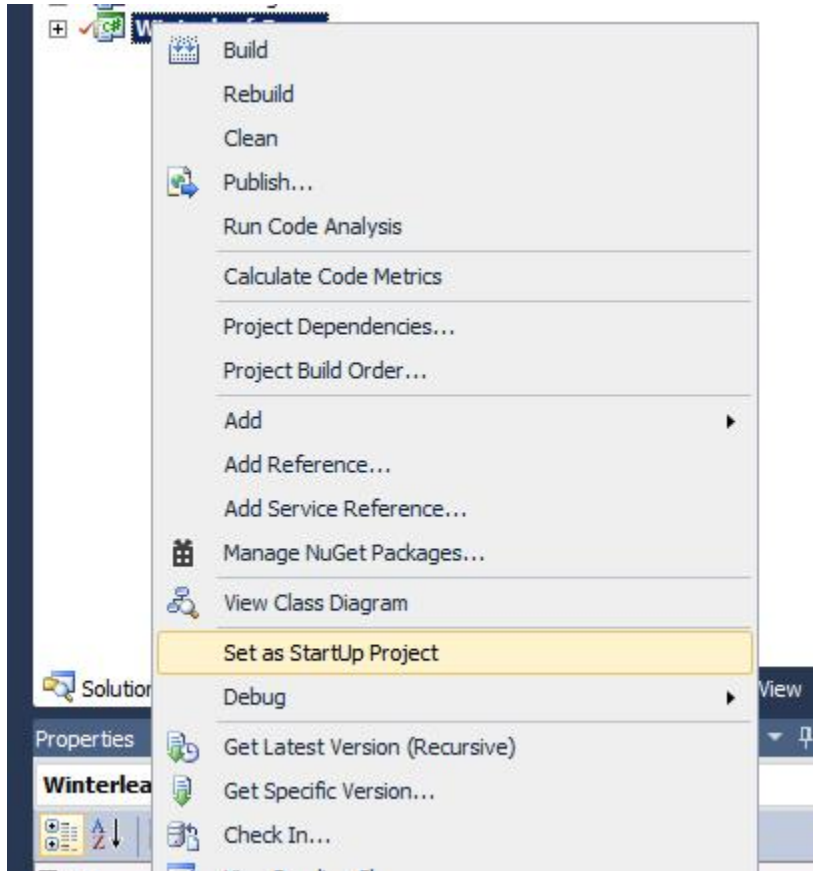
# OMNI ENGINE

For more information including how to extend the Static Code Generator to handle custom code please see Appendix 1.

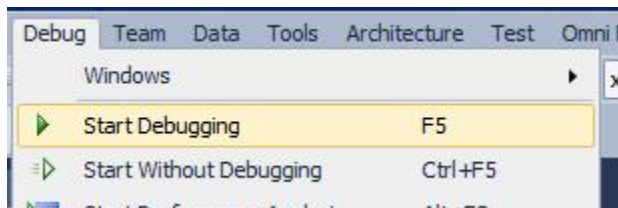
# OMNI ENGINE

## Running the Game

Now that the files are up to date you are able to run the game. Open up the “Omni Framework.sln” solution and right click the “Winterleaf.Game” project. In the menu, pick “Set as Startup Project”.



Finally, click “Debug”->”Start Debugging”

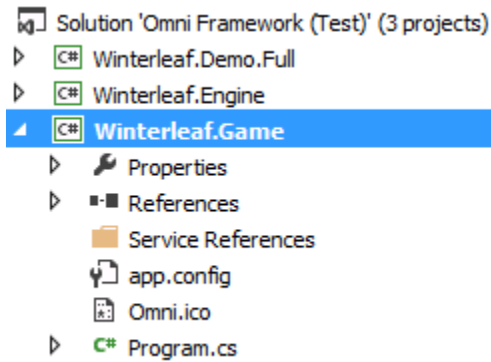


Congratulations, you have created your first project!

## Chapter 4 Customizing Winterleaf.Game Executable Name

At this point we assume you have created a new project and it has compiled. Now begins the time of customizing the Game Executable to your game information.

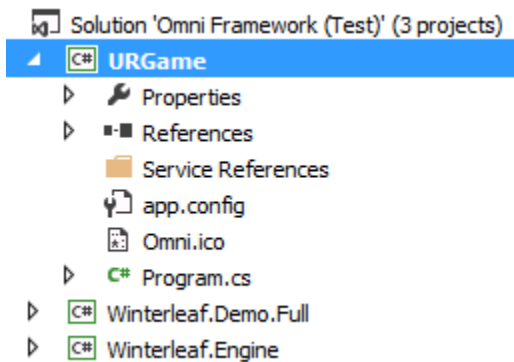
So open the “Omni Framework” solution.



Right click on the “Winterleaf.Game” project and select “Rename”

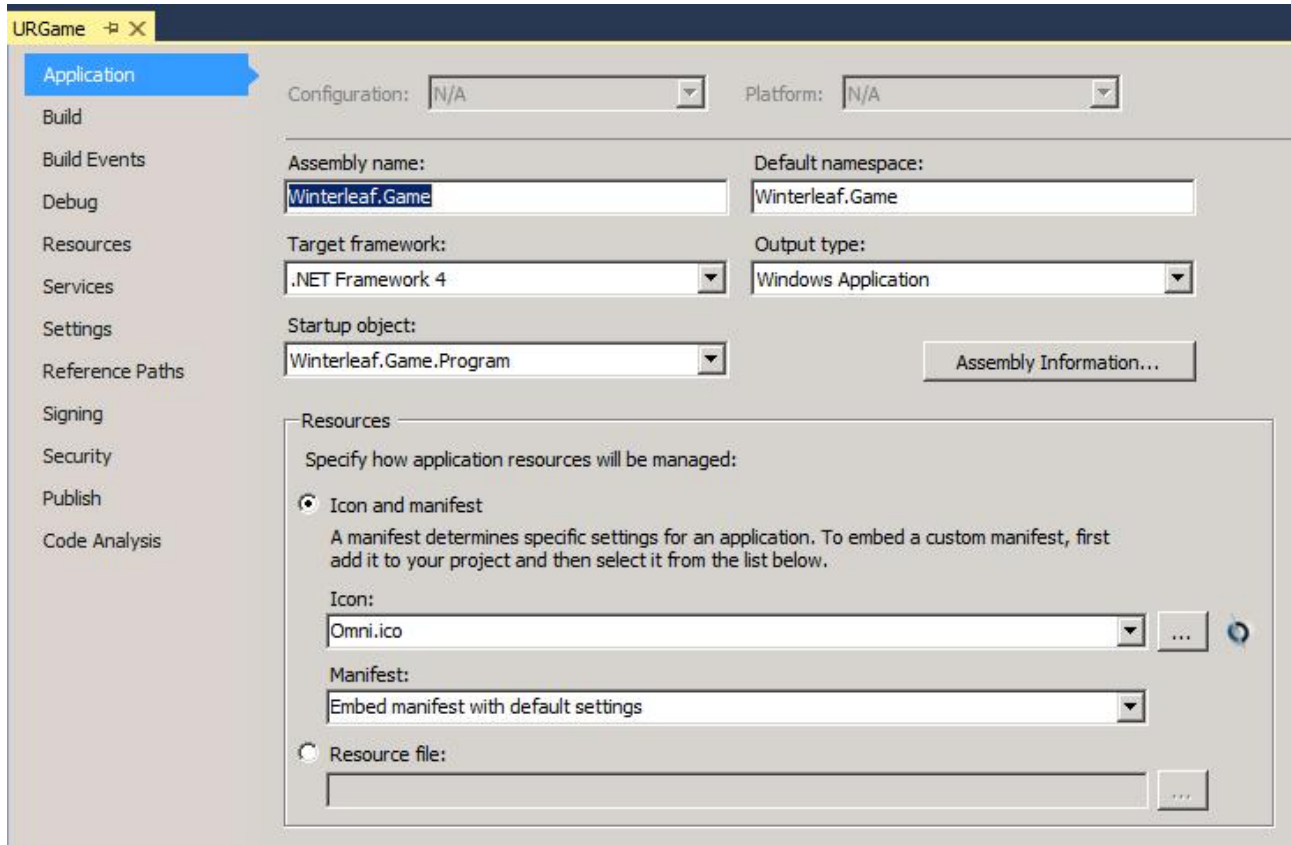


For this example, we will call our new executable “URGame”.



Now, we must configure the Properties for this project to reflect the renaming of the project. So once again right click on the “URGame” project and select “Properties”.

# OMNI ENGINE



As you can see, the Assembly name is still “Winterleaf.Game”, we need to change this to “URGame”. By changing this property we are changing what the compiler will name the executable.

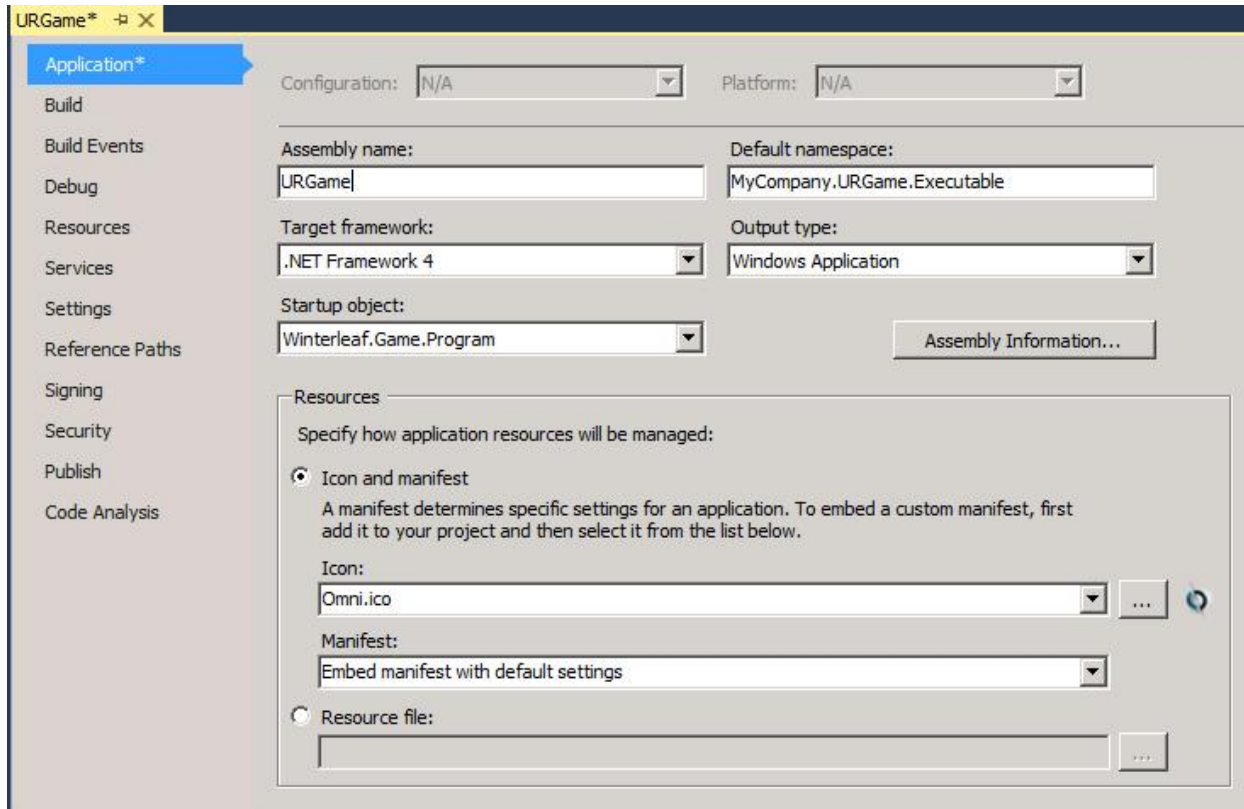
I would also highly recommend changing the Default Namespace for the project as well. Something to keep in mind, it is best to use the same base namespace name for both the Business Layer and Game Executable. So in this example, we will make the default namespace for the “URGame” executable be “MyCompany.URGame.Executable”.

You will also notice that we can select the Icon for the game here as well.

When we finish, it should look like the image on the next page.



# OMNI ENGINE



Now that we have updated the Project's properties, we need to update the source code in the project to reflect our namespace change.

Open the source code to "Program.cs" and highlight "Winterleaf.Game" like the image below.

# OMNI ENGINE

```

1 // Copyright (C) 2012 Winterleaf Entertainment L,L,C.
2
3 #region
14 [assembly: SecurityRules(SecurityRuleSet.Level1)]
15 namespace Winterleaf.Game
16 {
17     0 references | 0 authors | 0 changes
18     internal static class Program
19     {
20         /// <summary>
21         ///     The main entry point for the application.
22         /// </summary>
23         private static Omni omni;
24
25         [STAThread]
26         0 references | 0 authors | 0 changes
27         private static void Main()
28         {
29             Application.EnableVisualStyles();
30             Application.SetCompatibleTextRenderingDefault(false);
31             try
32             {
33                 //Application.Run(new main_window());
34                 omni = new Omni(Process.GetCurrentProcess().Handle);
35                 //Initialize Torque, pass a handle to this form into T3D so it knows where to render the screen to.
36                 //If you don't do this, you can't pass the mouse and key strokes, w/out the mouse and keystrokes
37                 //being redirected the application will hang intermittently.
38
39                 #if DEBUG
40                     omni.Initialize(MessageBox.Show("Dedicated", "Dedicated", MessageBoxButtons.YesNo) == DialogResult.Yes ?
41                         new[] { "-dedicated", "-mission", @"levels/Empty_Terrain.mis" } :
42                         new[] { "" }, "Test_DEBUG", "WinterLeaf.Demo.Full.dll", "WinterLeaf.Demo.Full", "csScripts");
43                 #else
44                     ...
45             }
46             #endif
47         }
48     }
49 }

```

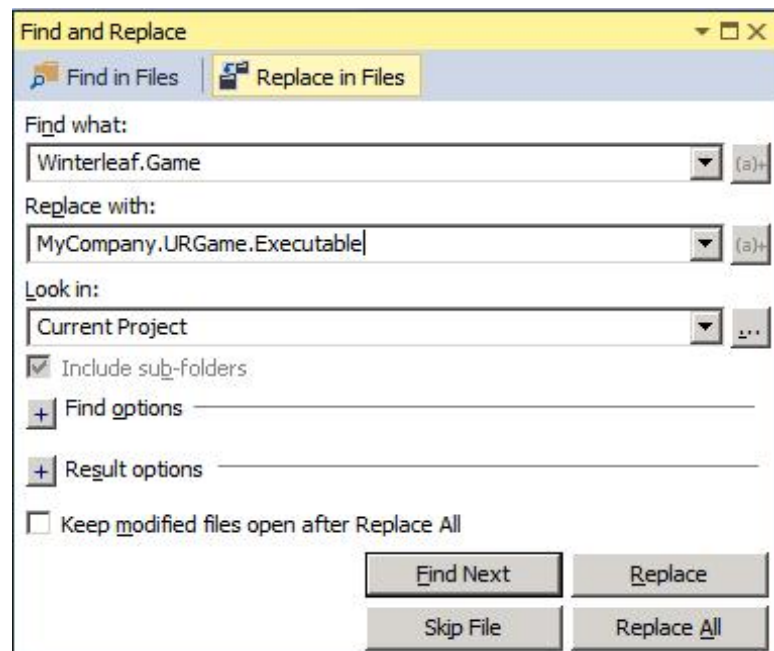
“Winterleaf.Game” is the old namespace for the program, we need to update this with our new namespace “MyCompany.URGame.Executable”.

So to do this, press <ctrl><alt><f>.

“Winterleaf.Game” should be selected for “Find What” and type into “Replace with” our new namespace of “MyCompany.URGame.Executable”.

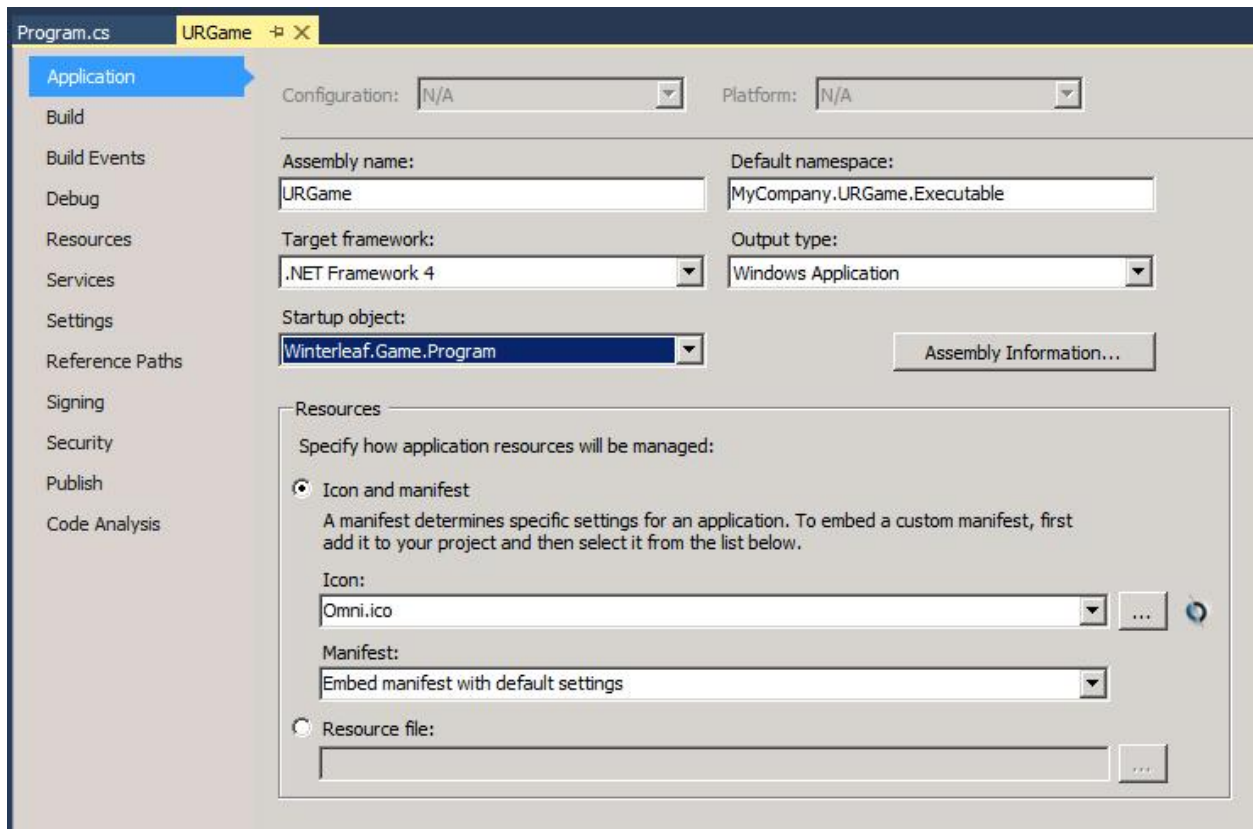
IMPORTANT: Make sure the “Look in” is set to “Current Project”. We aren’t ready to replace the namespaces in the other projects yet.

Press “Replace ALL”.



# OMNI ENGINE

Now that we have replaced the namespace we need to update the Projects Properties one more time. The Project Properties should look like the image below.



We need to change the “Startup Object” from “Winterleaf.Game.Program” to our new namespace, “MyCompany.URGame.Executable.Program”.

After changing the drop down it should look like the image below.

# OMNI ENGINE

Application\*

Build

Build Events

Debug

Resources

Services

Settings

Reference Paths

Signing

Security

Publish

Code Analysis

Configuration: N/A Platform: N/A

Assembly name: URGame Default namespace: MyCompany.URGame.Executable

Target framework: .NET Framework 4 Output type: Windows Application

Startup object: MyCompany.URGame.Executable.Program

Assembly Information...

Resources

Specify how application resources will be managed:

☒ Icon and manifest

A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.

Icon: Omni.ico

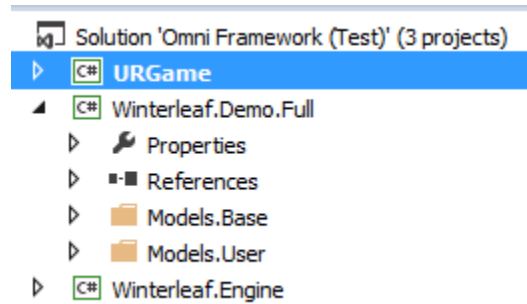
Manifest: Embed manifest with default settings

☐ Resource file:

Congratulations! You have successfully customize the Game Executable to your game needs.

## Chapter 5 Customizing Winterleaf.Demo.Full

It is assumed at this point you have already customized your Game Executable project and you are now eyeing the “Winterleaf.Demo.Full” project. There is no reason you can’t change the name to be more reflective of your own game, this chapter will address how to customize it to your project.



So starting with the Solution Explorer, if you are following the examples, it should look like the image below.

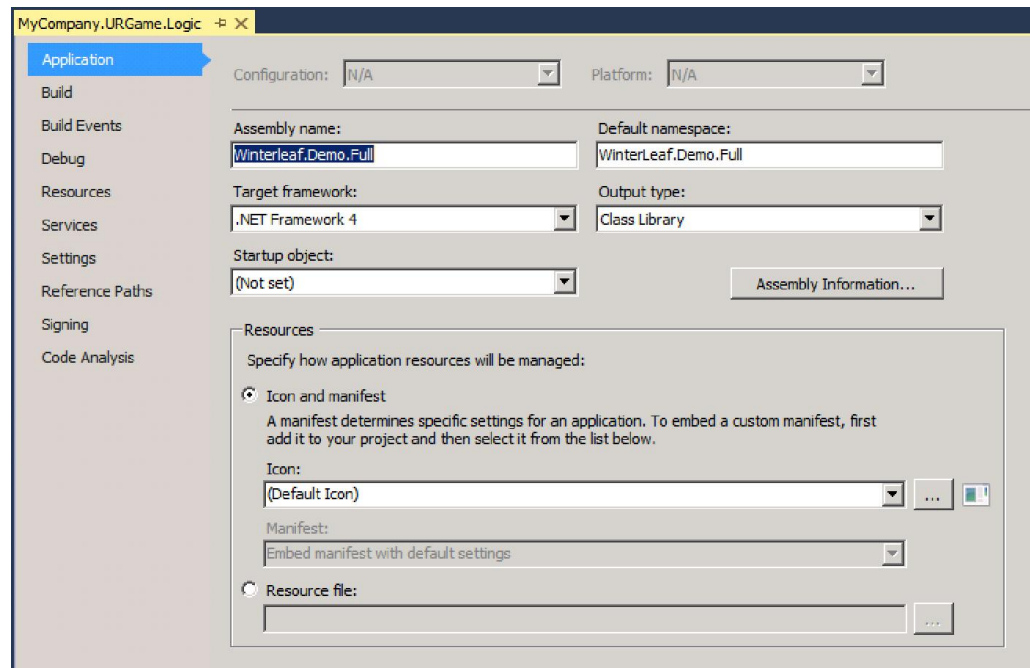
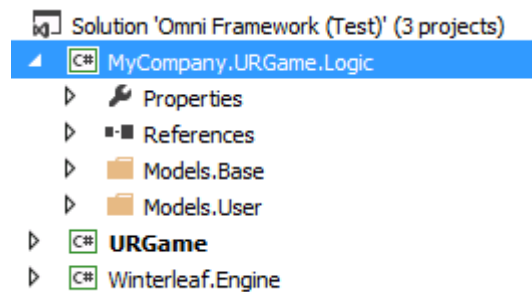
Just like in the previous chapter, the first step is to rename “Winterleaf.Demo.Full” to something more fitting for your game. Our first step is to right click on the “Winterleaf.Demo.Full” project and select “Rename”.

We will rename it to “MyCompany.URGame.Logic”.

After renaming the project our solution explorer should look like the image to the right.

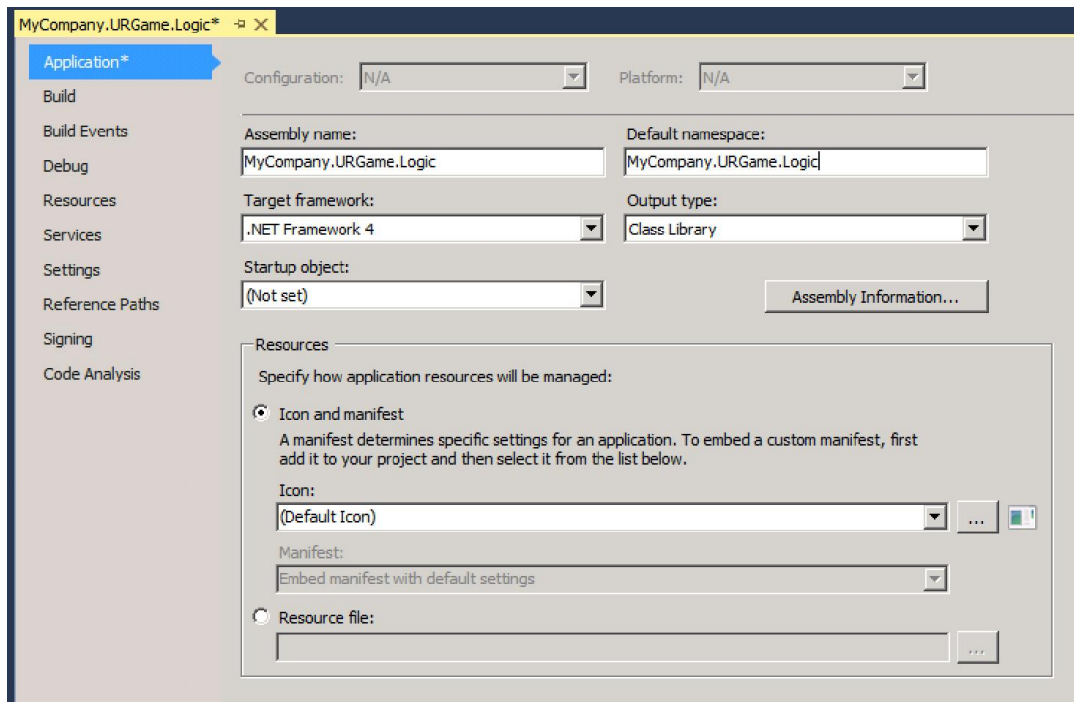
After renaming the project we now need to update the projects namespace. This process is identical to the process in the previous chapter.

So, right click on “MyCompany.URGame.Logic” and go to properties.



# OMNI ENGINE

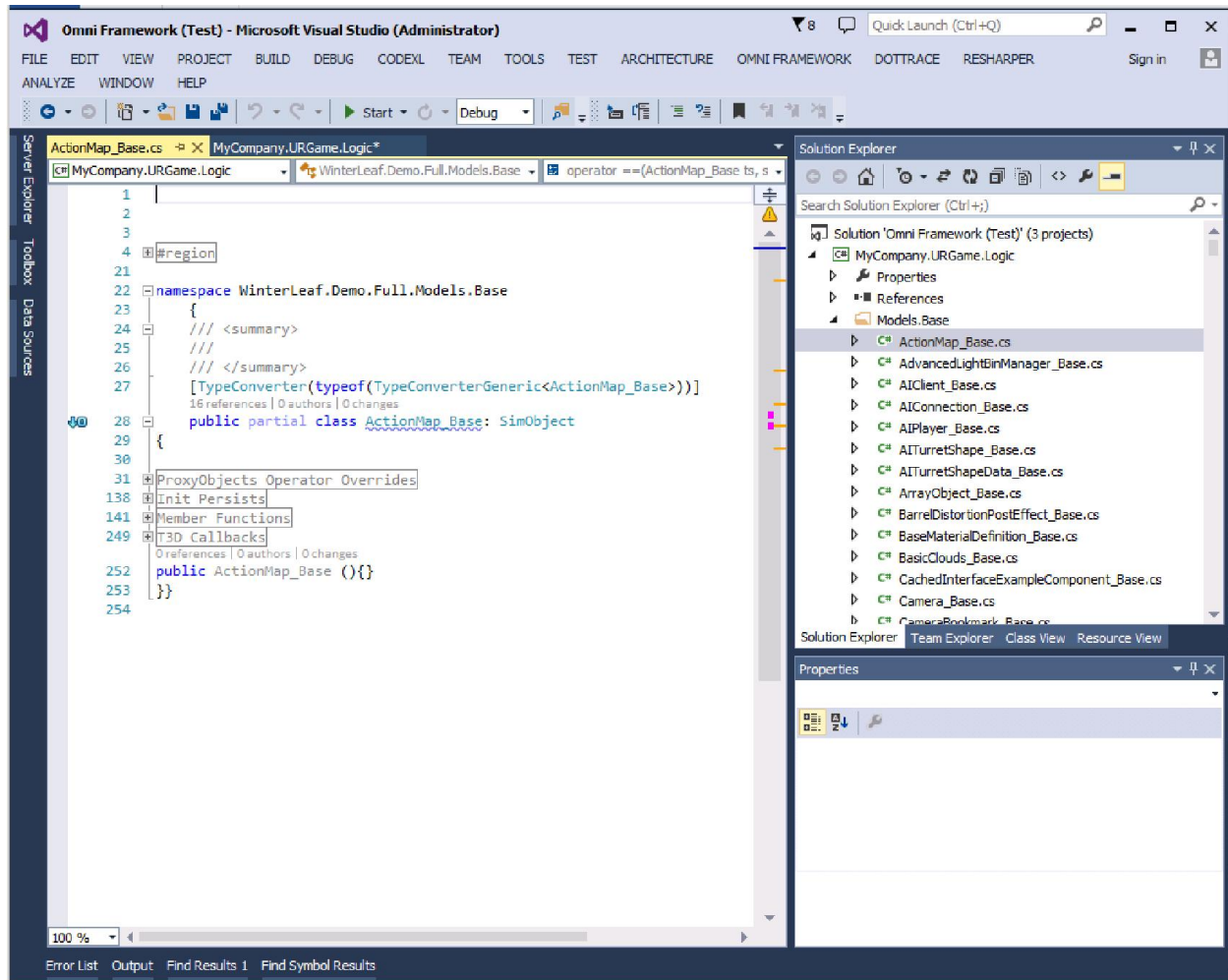
First we need to change the assembly name and Default namespace to “MyCompany.URGame.Logic”. Once you change it, it should look like the image below.



Now that we have updated the project properties, we need to update all the C# to use our new Namespace.

# OMNI ENGINE

To do this we need to open up one of the C# files inside our project, in the image below I choose “ActionMap\_Base.CS”.



Highlight “Winterleaf.Demo.Full.” and press <Ctrl><Alt>F.

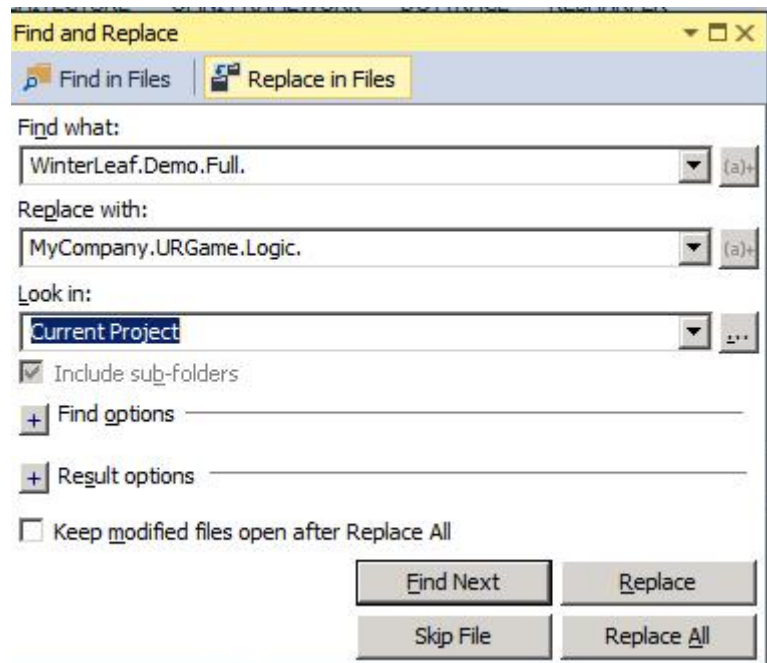


# OMNI ENGINE

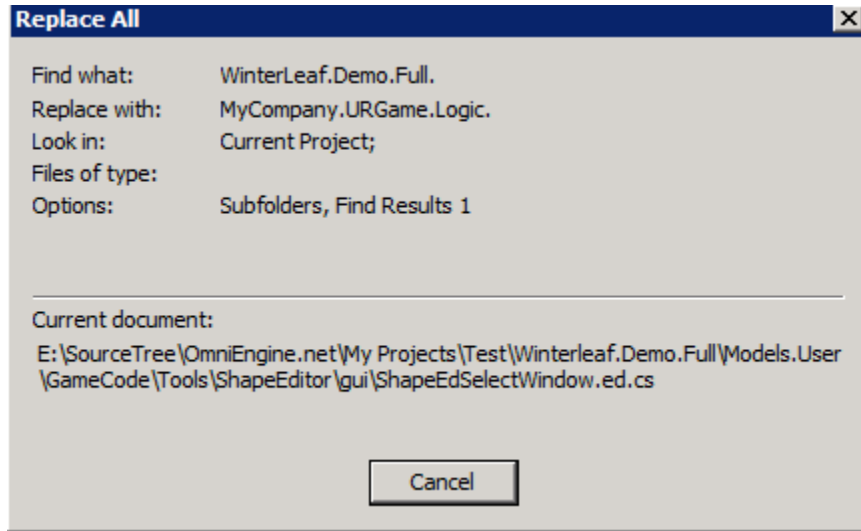
The find and Replace window should look like the image to the right. Make sure when you type your new namespace you end it with a period. So we are replacing “Winterleaf.Demo.Full.” with “MyCompany.URGame.Logic.”.

Make sure the “Look in” is set to “Current Project”.

Press “Replace ALL” when all fields are filled in.



It will then begin replacing the namespace in all the files, this can take some time, so go get a fresh cup of coffee.



After it completes we just need to update the Game Executable with the new information. So open up the “Program.cs” file in the “URGame” project.



# O M N I E N G I N E

```

14 [assembly: SecurityRules(SecurityRuleSet.Level1)]
15 namespace MyCompany.URGame.Executable
16 {
17     0 references | 0 authors | 0 changes
18     internal static class Program
19     {
20         /// <summary>
21         ///     The main entry point for the application.
22         /// </summary>
23         private static Omni omni;
24
25         [STAThread]
26         0 references | 0 authors | 0 changes
27         private static void Main()
28         {
29             Application.EnableVisualStyles();
30             Application.SetCompatibleTextRenderingDefault(false);
31             try
32             {
33                 //Application.Run(new main_window());
34                 omni = new Omni(Process.GetCurrentProcess().Handle);
35                 //Initialize Torque, pass a handle to this form into T3D so it knows where to render the screen to.
36                 //If you don't do this, you can't pass the mouse and key strokes, w/out the mouse and keystrokes
37                 //being redirected the application will hang intermittently.
38
39                 #if DEBUG
40                     omni.Initialize(MessageBox.Show("Dedicated", "Dedicated", MessageBoxButtons.YesNo) == DialogResult.Yes ?
41                         new[] { "-dedicated", "-mission", @"levels/Empty_Terrain.mis" } :
42                         new[] { "", "Test_DEBUG", "WinterLeaf.Demo.Full.dll", "WinterLeaf.Demo.Full", "csScripts" });
43                 #else
44                     omni.Initialize(MessageBox.Show("Dedicated", "Dedicated", MessageBoxButtons.YesNo) == DialogResult.Yes ?
45                         new[] { "-dedicated", "-mission", @"levels/Empty_Terrain.mis" } :
46                         new[] { "", "Test", "WinterLeaf.Demo.Full.dll", "WinterLeaf.Demo.Full", "csScripts" });
47                 #endif
48
49                 omni.Debugging = false;
50                 omni.ScriptExtensions_Allow = true;
51                 omni.ScriptExtensions_HandleExceptions = true;
52
53                 omni.DebuggingShowScriptCalls = false;
54                 omni.WindowIcon = new Icon("Omni.ico");
55                 while (omni.IsRunning)
56                 {
57                     Thread.Sleep(1000);
58                     omni = null;
59                 }
60                 catch (Exception err)
61                 {
62                     MessageBox.Show("An Error has occurred in the application. " + err.Message);
63                 }
64                 Application.Exit();
65             }
66         }
67     }
68 }

```

Specifically, we are looking at the code on lines 37 to 44. These lines configure the OMNI Framework initialization values.

```

36 #if DEBUG
37     omni.Initialize(MessageBox.Show("Dedicated", "Dedicated", MessageBoxButtons.YesNo) == DialogResult.Yes ?
38         new[] { "-dedicated", "-mission", @"levels/Empty_Terrain.mis" } :
39         new[] { "", "Test_DEBUG", "WinterLeaf.Demo.Full.dll", "WinterLeaf.Demo.Full", "csScripts" });
40 #else
41     omni.Initialize(MessageBox.Show("Dedicated", "Dedicated", MessageBoxButtons.YesNo) == DialogResult.Yes ?
42         new[] { "-dedicated", "-mission", @"levels/Empty_Terrain.mis" } :
43         new[] { "", "Test", "WinterLeaf.Demo.Full.dll", "WinterLeaf.Demo.Full", "csScripts" });
44 #endif

```













Looking at the code above, you can see that the old DLL name being used was “Winterleaf.Demo.Full.dll” and the old namespace was “Winterleaf.Demo.Full”. We need to update these values to reflect the new names we assigned.

# O M N I E N G I N E

So replace “Winterleaf.Demo.Full.dll” with the new name you assigned the DLL. In this example it is “MyCompany.URGame.Logic.dll” and then change the Namespace to “MyCompany.URGame.Logic”. Make sure you change it on both lines 39 and 43.

































Go to your game directory and delete all the exe and dll’s, we will need to regenerate them. Your folder should be similar to the image below.

---

 art	10/9/2014 10:55 AM	File folder	
 bin	10/9/2014 10:55 AM	File folder	
 core	10/9/2014 10:55 AM	File folder	
 csScripts	10/9/2014 10:55 AM	File folder	
 DBCaches	10/9/2014 10:55 AM	File folder	
 levels	10/9/2014 10:55 AM	File folder	
 scripts	10/9/2014 10:55 AM	File folder	
 shaders	10/9/2014 10:55 AM	File folder	
 tools	10/9/2014 10:55 AM	File folder	
 web	10/9/2014 10:55 AM	File folder	
 CS-Full.torsion	10/6/2014 7:54 AM	Torsion Project File	1 KB
 Omni.ico	10/6/2014 7:54 AM	Icon	1,606 KB

Now, rebuild both solutions. After rebuilding your game folder should look like the image below.

# O M N I E N G I N E

 art	10/9/2014 10:55 AM	File folder	
 bin	10/9/2014 10:55 AM	File folder	
 core	10/9/2014 10:55 AM	File folder	
 csScripts	10/9/2014 10:55 AM	File folder	
 DBCaches	10/9/2014 10:55 AM	File folder	
 levels	10/9/2014 10:55 AM	File folder	
 scripts	10/9/2014 10:55 AM	File folder	
 shaders	10/9/2014 10:55 AM	File folder	
 tools	10/9/2014 10:55 AM	File folder	
 web	10/9/2014 10:55 AM	File folder	
 console.log	10/9/2014 9:44 PM	Text Document	17 KB
 CS-Full.torsion	10/6/2014 7:54 AM	Torsion Project File	1 KB
 MyCompany.URGame.Logic.dll	10/9/2014 9:22 PM	Application extension	4,722 KB
 MyCompany.URGame.Logic.pdb	10/9/2014 9:22 PM	Program Debug Dat...	12,002 KB
 Omni.ico	10/6/2014 7:55 AM	Icon	1,606 KB
 prefs.banlist.cs	10/9/2014 9:44 PM	Visual C# Source file	0 KB
 prefs.client.cs	10/9/2014 9:44 PM	Visual C# Source file	5 KB
 prefs.server.cs	10/9/2014 9:44 PM	Visual C# Source file	1 KB
 Test_DEBUG.dll	10/9/2014 9:37 PM	Application extension	31,617 KB
 Test_DEBUG.exe	10/9/2014 9:37 PM	Application	507 KB
 Test_DEBUG.exp	10/9/2014 9:37 PM	Exports Library File	847 KB
 Test_DEBUG.lib	10/9/2014 9:37 PM	Object File Library	1,385 KB
 URGame.exe	10/9/2014 9:23 PM	Application	1,614 KB
 URGame.exe.config	10/6/2014 7:55 AM	XML Configuration File	1 KB
 URGame.pdb	10/9/2014 9:23 PM	Program Debug Dat...	18 KB
 URGame.vshost.exe	10/9/2014 9:44 PM	Application	23 KB
 URGame.vshost.exe.config	10/6/2014 7:55 AM	XML Configuration File	1 KB
 WinterLeaf.Engine.Omni.dll	10/9/2014 9:12 PM	Application extension	1,976 KB
 WinterLeaf.Engine.Omni.pdb	10/9/2014 9:12 PM	Program Debug Dat...	3,196 KB
 Winterleaf.Game.exe.config	10/6/2014 7:55 AM	XML Configuration File	1 KB
 Winterleaf.Game.vshost.exe.config	10/6/2014 7:55 AM	XML Configuration File	1 KB
 Winterleaf.Game.vshost.exe.manifest	3/17/2010 11:39 PM	MANIFEST File	1 KB

Run the URGame.exe to fire up your customized game!

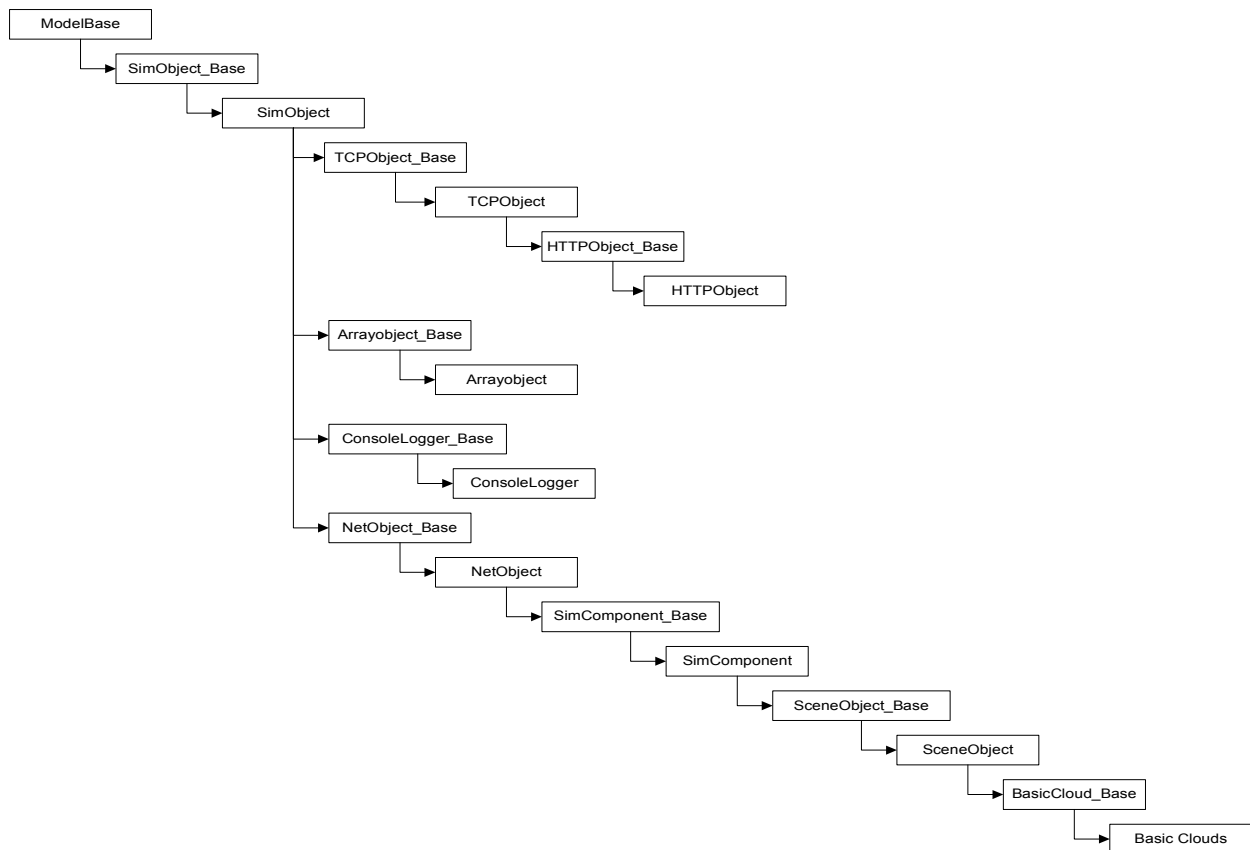
## Chapter 5 Object Models Overview

### Introduction

All Objects Models in the OMNI Framework are derived from the BaseModel class found in the namespace “WinterLeaf.Engine.Classes”. The BaseModel provides a foundation for all objects derived from it. Its functionality is limited to subset of functions found on SimObject.

### General Inheritance Design

The inheritance structure was specifically designed to be used in conjunction with a code generator. A sample of the inheritance tree is:



The above doesn't represent all the classes in the engine, but instead demonstrates how the inheritance is modeled.

## Reasoning

All of the base functionality exposed by the C++ object is incorporated into the “\_Base” class. This would include member functions and member variables. These would be combined with the inherited member functions and member variables of the parent classes. In the event that in the C++ a member function was overridden and redefined, you would have to use the C# syntax of “base.somefunction()” to call the hidden function.

When you want to add custom code to a Base Model, you simple create a partial class of the Extendable class. This design puts the custom C# code in a separate class file from the “\_Base” and “Extendable” class. That way when the code generator is re-executed, none of the custom code added to the class is overridden.

The Static Code Generator will automatically empty the folders “Models.Base” and “Model.User/Extendable” every time it is run. Because of this it is not recommended to save your class extensions to the Extendable classes in the “Model.User/Extendable” folder in the project.

In the example code provided in the “WinterLeaf.Demo.Full” project you will see that all partial class extensions for the “Extendable” objects are stored in various places inside the “Model.User/GameCode” folder. Where it is saved is not important, the only thing that is important is that the partial class namespace matches the namespace of the partial class defined in “Model.User/Extendable” class.

## Chapter 6 Creating Views (Omni T3D Objects)

### Introduction

Due to the different types of Views (Omni T3D Objects) available in the game there are different ways to create each of the View (Omni T3D Object) and in some cases multiple ways to create that View(Omni T3D Object).

An important thing to remember is that everything in the Omni T3D Engine is an object: players, vehicles, items, etc. It is the purpose of the objects that determine what sub-type of object they are.

To begin with, there are three different types of View types. This can get confusing and more information about the different types of objects can be found at <http://docs.garagegames.com/torque-3d/reference/syntaxGameObjects.html>.

The View Types are:

- Instance based View (Omni T3D Instanced Objects)
  - Create() function returns the uint handle to the new object.
- Datablock
  - Create() function returns the name of the new datablock.
- Singleton
  - Create() function returns the name of the new singleton.

### There are three rules that apply to all Creator Types:

**Rule 1: You cannot assign properties after you call the “Create()” function.**

Once the “Create()” function is called you cannot assign more properties.



# OMNI ENGINE

**Rule 2: Any Creator based object being assigned as a property, the property must be prefixed with a “#”.**

```

169  ObjectCreator oc_Newobject5 = new ObjectCreator("GuiScrollCtrl", "EWCreatorPane");
170  oc_Newobject5["canSaveDynamicFields"] = "0";
171  oc_Newobject5["isContainer"] = "1";
172  oc_Newobject5["Profile"] = "ToolsGuiScrollProfile";
173  oc_Newobject5["HorizSizing"] = "width";
174  oc_Newobject5["VertSizing"] = "height";
175  oc_Newobject5["position"] = "0 0";
176  oc_Newobject5["Extent"] = "288 202";
177  oc_Newobject5["MinExtent"] = "8 8";
178  oc_Newobject5["canSave"] = "1";
179  oc_Newobject5["Visible"] = "0";
180  oc_Newobject5["hovertime"] = "1000";
181  oc_Newobject5["willFirstRespond"] = "1";
182  oc_Newobject5["hScrollBar"] = "dynamic";
183  oc_Newobject5["vScrollBar"] = "dynamic";
184  oc_Newobject5["lockHorizScroll"] = new ObjectCreator.StringNoQuote("true");
185  oc_Newobject5["lockVertScroll"] = "false";
186  oc_Newobject5["constantThumbHeight"] = "0";
187  oc_Newobject5["childMargin"] = "0 0";
188
189  + GuiTreeViewCtrl (Creator)  oc_Newobject4
214
215  oc_Newobject5["#Newobject4"] = oc_Newobject4;
216
217  #endregion
218
219  oc_Newobject13["#Newobject5"] = oc_Newobject5;
220

```

As you can see in the above code snippet, the assignment of an Object Creator is assigned to a property prefixed with “#”. The name of the property must be unique.

**Rule 3: All Creator based objects assigned as properties must be the last properties assigned.**

As you can see in the above code snippet, the assignment of an Object Creator must be last in the list of properties.

## ObjectCreator Class: Create Instanced based Views

Most of the Views you create using the Omni T3D Framework will be of this type. This includes players, game items, vehicles, etc.

In the simplest form, a View can be created with one line of code:

```

57  |  GuiCanvas canvas = new ObjectCreator("GuiCanvas", "Canvas", typeof(canvas)).Create();

```

This single line of code would create a new “GuiCanvas” View called “Canvas” and bind it to the “canvas” C# model.

The exact syntax for using the ObjectCreator is:

# O M N I E N G I N E

```
21 public ObjectCreator(string className, string instanceName = "", Type model = null)
```

- “ClassName” is the name of the Omni T3D Object.
- “InstanceName” is a friendly name you want to assign to the object.
- “model” is the c# class type which an instance will be created for when the object is created.

You can also specify properties along with specifying the Class Name, Instance Name and Model.

```
26 ObjectCreator oc = new ObjectCreator("GFXSamplerStateData", "SamplerClampLinear");
27 oc["textureColorOp"] = "GFXTOPModule";
28 oc["addressModeU"] = "GFXAddressClamp";
29 oc["addressModeV"] = "GFXAddressClamp";
30 oc["addressModeW"] = "GFXAddressClamp";
31 oc["magFilter"] = "GFXTextureFilterLinear";
32 oc["minFilter"] = "GFXTextureFilterLinear";
33 oc["mipFilter"] = "GFXTextureFilterLinear";
34 oc.Create();
35
```

The above code shows how to add initial values to properties on the object. The values provided for the properties can be simple strings or complex objects.

```
449 oc = new ObjectCreator("GuiButtonCtrl");
450 oc["HorizSizing"] = "width";
451 oc["profile"] = "ToolsGuiButtonProfile";
452 oc["extent"] = this["fileButtonExtent"];
453 oc["position"] = (curXPos + columnOffset) + " " + curYPos;
454 oc["modal"] = "1";
455 oc["command"] = new ObjectCreator.StringNoQuote(this + ".getFileName(" + index + ");");
456
457 this["controls[" + numControls + "]"] = oc.Create().AsString();
---
```

Any object which can be serialized to a string via “toString()” can be assigned as a value to a property. Also, you can specify friendly names of other objects as the value to a property. You can even go so far as to nest Creator based objects inside of other Creators as seen on line 455.

The key here is that after you assign the properties to the View you call the Create() function to create the View. The Create() function will then return to the caller the ID of the View in the world which you can use as a handle.

You will find that Gui’s use extensively nested ObjectCreators. This is because we want to create a TextBox inside a panel that is inside a Scrollable area which is inside a window. The important thing to remember is that the order of everything is very important.

## DatablockCreator Class: Create Datablock based Views

When creating Instance based Views in the Omni T3D engine you will want to assign static values. These static values are saved in Datablocks. Datablocks are only transmitted once to the client when the client connects to the server. Use of Datablocks reduce the network traffic to all clients.



# OMNI ENGINE

The exact syntax for using the ObjectCreator is:

```
23 public DatablockCreator(string className, string instanceName, Type model = null)
```

- “ClassName” is the name of the Omni T3D Object.
- “InstanceName” is a friendly name you want to assign to the object.
- “model” is the c# class type which an instance will be created for when the object is created.

You can also specify properties along with specifying the Class Name, Instance Name and Model.

```
4852 DatablockCreator object1 = new DatablockCreator("ParticleEmitterNodeData", "TestEmitterNodeData");  
4853 object1["timeMultiple"] = "1";  
4854 object1.Create();
```

The above code shows how to add initial values to properties on the object. The values provided for the properties can be simple strings or complex objects.

Any object which can be serialized to a string via “toString()” can be assigned as a value to a property. Also, you can specify friendly names of other objects as the value to a property. You can even go so far as to nest Creator based objects inside of other Creators as seen on line 455.

The key here is that after you assign the properties to the View you call the Create() function to create the View. The Create() function will then return to the caller the ID of the View in the world which you can use as a handle.

## SingletonCreator Class: Create Singleton based Views

If you need a global script object with only a single instance, you can use the singleton keyword. Singletons, in TorqueScript, are mostly used for unique shaders, materials, and other client-side only objects.

For example, SSAO (screen space ambient occlusion) is a post-processing effect. The game will only ever need a single instance of the shader, but it needs to be globally accessible on the client. The declaration of the SSAO shader in C# can be shown below:

```
139 ts = new SingletonCreator("ShaderData", "SSAOShader");  
140 ts["DXVertexShaderFile"] = "shaders/common/postFx/postFxV.hlsf";  
141 ts["DXPixelShaderFile"] = "shaders/common/postFx/ssao/SSAO_P.hlsf";  
142 ts["pixVersion"] = 3.0;  
143 ts.Create();  
...
```

The constructor is:

```
24 public SingletonCreator(string className, string singletonName = "", Type model = null)
```

- “ClassName” is the name of the Omni T3D Object.
- “SingletonName” is a friendly name you want to assign to the object.

# OMNI ENGINE

- “model” is the c# class type which an instance will be created for when the object is created.

You can also specify properties along with specifying the Class Name, Instance Name and Model.

The above code shows how to add initial values to properties on the object. The values provided for the properties can be simple strings or complex objects.

Any object which can be serialized to a string via “toString()” can be assigned as a value to a property. Also, you can specify friendly names of other objects as the value to a property. You can even go so far as to nest Creator based objects inside of other Creators as seen on line 455.

The key here is that after you assign the properties to the View you call the Create() function to create the View. The Create() function will then return to the caller the ID of the View in the world which you can use as a handle.

## Creating Objects in TorqueScript

Sometimes some code will make more sense to be left in TorqueScript than to convert to C# during development. Datablocks are the biggest offender since artists may want to swap out animations, file references and such while they are working on assets. Forcing them to have Visual Studio and to recompile the code every time they want to change something could be cumbersome.

Because of this, you can hook a Model up to any object via TorqueScript with one simple property.

```
27 datablock CameraData(Observer)
28 {
29     mode = "Observer";
30     WLE_OVERRIDE_PROXY_CLASSTYPE = "WinterLeaf.Demo.Full.Models.User.GameCode.Server.Camera.Observer";
31 };
```

This is done by using the property “WLE\_Override\_Proxy\_ClassType”.

This property states the full namespace to the Model this object will use when it is created. Obviously for code management this is not the recommended way doing it, but it does provide a work-around when you have a situation where a TorqueScript/C# combination environment going on.

If you do not specify the WLE\_Override\_Proxy\_ClassType Model in the properties, the default Model will be used when the Omni T3D View is created. So in the case above, if the Observer Model had not been specified, it would have used the Model “CameraData”.

## Chapter 7 Extending the User.Models.Extendable

This section will explain how to take the base code and extend them. To create your game you will need to override and create new functions and member variables in the Extendable Models. This would include adding generic logic for collisions, object addition/removal, mounting, etc.

A couple of things to remember!

- Callbacks from the T3D engine are already exposed in the base class, so to add logic for one of them you just need to override the function.
- If you plan to allow a custom function to be re-implemented in an inherited class, the function must be defined as “virtual”.
- Functions that are not callbacks from the T3D engine that you wish to be callable from the T3D console must be marked with the “ConsoleInteraction” decoration.
- Extended Models are always defined as a “**public partial class**”.

### Member Functions (Overriding T3D engine Callbacks)

This is just the process of overriding the intended public function to the class. In the class file located at Model.User/GameCode/Server/Player/PlayerData.cs file you will see the following code.

```

1  #region
13
14  namespace WinterLeaf.Demo.Full.Models.User.Extendable
15  {
16      // <summary>...
19      public partial class PlayerData
20      {
21          public override bool OnFunctionNotFoundCallTorqueScript()
22          {
23          }
24
25          public override void onAdd(GameBase obj)
26          {
27              Player player = obj.getId();
28              player["mountVehicle"] = true.AsString();
29              player.setRechargeRate(this["rechargeRate"].AsFloat());
30              player.setRepairRate(0);
31          }
32
33
34          public override void onRemove(GameBase obj)
35          {
36          }
37
38
39          public override void onMount(GameBase obj, SceneObject mountObj, int node)
40          {
41          }
42
43          public override void onUnmount(GameBase obj, SceneObject mountObj, int node)
44          {
45          }
46
47          public override void doDismount(Player obj)
48          {
49          }
50
51
52          public override void onCollision(ShapeBase obj, SceneObject col, Point3F vec, float len)
53          {
54          }
55
56          public override void onImpact(ShapeBase obj, SceneObject collObj, Point3F vec, float len)
57          {
58          }
59
60
61          public override void damage(ShapeBase obj, Point3F position, GameBase sourceobject, float damage,
62              string damagetype)
63          {
64          }
65
66          public override void onDamage(ShapeBase obj, float delta)
67          {
68          }
69
70          public override void onDisabled(ShapeBase obj, string lastState)
71          {
72          }
73
74
75      }
76  }

```

# OMNI ENGINE

Take notice that the class Namespace is not reflective of where the file is. This is because you must store your partial class extensions to “Extendable” classes in a different folder than the “Extendable” location.

Next you will notice that each function starts with “public override...” Each of the functions defined in this partial class are defined in either the “\_Base” of this model or a parent model all the way to the “ModelBase” model. On line 28 you can see where custom code for the “onAdd” callback has been added to the class. Whenever a View (T3D Object) has an “onAdd” event, the event will be rerouted to this Model class and the “onAdd” member function will be called. This is the same for all of the overridden member functions.

## “ConsoleInteraction” Decoration

There are times when you will want to add a new member function to a Model and have it callable via the console inside of the T3D engine. Usually these are times when you are using the old style of Gui interface event wiring. Other times, you might want to expose a function to the console for debugging purposes. No matter what the reason, any function can be exposed to the T3D console using the “[ConsoleInteraction]” decoration placed before it.

```

24     {
25         return false;
26     }
27
28     //todo this does not need to be exposed to T3D in the end.
29
30     [ConsoleInteraction(true)]
31     public virtual void damage(GameBase sourceobject, Point3F position, float damage, string damagetype)
32     {
33         if (!isObject())
34             return;
35         ShapeBaseData datablock = getDataBlock();
36         datablock.damage(this, position, sourceobject, damage, damagetype);
37     }
38
39     //todo this does not need to be exposed to T3D in the end.
40     [ConsoleInteraction(true)]
41     public virtual void setDamageDT(float damageAmount, string damageType)
42     {
43         // This function is used to apply damage over time. The damage is applied
44         // at a fixed rate (50 ms). Damage could be applied over time using the
45         // built in ShapBase C++ repair functions (using a neg. repair), but this
46         // has the advantage of going through the normal script channels.
47
48         if (getDamageState() != "Dead")
49         {
50             damage(0, new Point3F("0 0 0"), damageAmount, damageType);
51             this["damageSchedule"] = schedule("50", "setDamageDt", damageAmount.AsString(), damageType).AsString();
52         }
53         else
54         {
55             this["damageSchedule"] = string.Empty;
56         }
57     }
58
59     //todo this does not need to be exposed to T3D in the end.
60     [ConsoleInteraction(true)]
61     public virtual void setDamageDT(float damageAmount, string damageType)

```

In the example code above, you can see how the “damage” and “setDamageDT” member functions are being exposed to the T3D console. (You can bring up the T3D console when the game is running by pressing the ` key.

## Member Variables

When defining variables in your code you have two options. You can go the traditional route and use the View[“variable name”] method or you can define a property in the cSharp. You can also combine the two to create programming shortcuts.

### Traditional Member Variable

```
this["damageSchedule"] = schedule("50", "setDamageDt", damageAmount.AsString(), damageType).AsString();
```

What this is doing is creating a T3D member variable for object “this” called “damageSchedule”. This variable is accessible from both C#, TorqueScript, and the console since it lives in the Omni T3D Engine.

# OMNI ENGINE

Omni T3D will manage the variable and clean up and the memory space the variable uses is reserved in the un-managed memory area of the application.

## C# Member Variable

A C# member variable would be defined in the C# like the following.

```
22 |  
23 | public string Databaseld { get; set; }  
24 |
```

And

```
23 | private string _Databaseld = string.Empty;  
24 | public string Databaseld  
25 | {  
26 |     get { return _Databaseld; }  
27 |     set { _Databaseld = value; }  
28 | }
```

Both are the same except one uses a backer variable and the other doesn't. The C# member variable is only stored in the C# and is not exposed to the Omni T3D engine. It is persisted for as long as the object exists, so it is state-full. It cannot be accessed via TorqueScript or the console. The memory is stored in the managed memory area of the application.

## Hybrid Member Variable

```
24 | public string Databaseld  
25 | {  
26 |     get { return this["Databaseld"]; }  
27 |     set { this["Databaseld"] = value; }  
28 | }  
29 |
```

In this case we are using the "Traditional Memory Variable" memory storage and access with the ease of the "C# Member Variable" format. This simplifies the c# coding and reduces the changes for errors while still exposing the member variable to the console.

### NOTE:

**All of the member variables defined in the "InitPersist" C++ code block are exposed to the C# using the Hybrid Member Variable format automatically casted to the appropriate mapped C# class type derived from the C++ code.**

## Static Member Functions (Global functions)

Any function that is defined as static and has a ConsoleInteraction decoration is assumed to be a global function inside the Omni T3D engine regardless of where it is defined. Because of this, all static functions must have a unique name and the unique name is not case-sensitive.

# O M N I E N G I N E

```

13 [TypeConverter(typeof (TypeConverterGeneric<chatHud>))]
14 public class chatHud : GuiMessageVectorCtrl
15 {
16
17
18 public override bool OnFunctionNotFoundCallTorqueScript()...
22
23 public static void initialize()...
41
42 public void AddLine(string text)...
92
93 public void PageUp()...
129
130 public void PageDown()...
169
170 // Helper function to play a sound file if the message indicates. ...
172 public static int PlayMessageSound(string message, string voice, string pitch)...
227
228 public static void OnChatMessage(string message, string voice, string pitch)...
248
249 [ConsoleInteraction(true)]
250 public static void OnServerMessage(string message)...
259
260 [ConsoleInteraction(true)]
261 public static void PageUpMessageHud()...
265
266 [ConsoleInteraction(true)]
267 public static void PageDownMessageHud()...
271
272 [ConsoleInteraction(true)]
273 public static void CycleMessageHudSize()...
277
278 ProxyObjects Operator Overrides
387 }
388

```

In the above code on line 228 you will see there is a static member function “OnChatMessage”. Even though this static function lives inside the chatHud Model the function is a global function. This allows you to group T3D Global console functions in the Models that they pertain to or just create a static c# class that has generic global Omni T3D function calls in it.

## Function: OnFunctionNotFoundCallTorqueScript

All C# Models have the “OnFunctionNotFoundCallTorqueScript” either via inheritance from the base model or overridden in the class definition. By default this function returns true.

This function is called whenever the T3D engine calls to the Omni Framework and the Omni Framework is not able to find the function defined in the C#. If this function returns true, then the Omni Framework will pass control back to the C++ engine and have it check if the function exists in a TorqueScript file. If the function exists in a TorqueScript function, it will call the function and continue processing.

# OMNI ENGINE

Calling TorqueScript is very expensive, so to improve performance we provided this method as a way to shortcut the engine. When you have finished converting a class from TorqueScript to C# it is always recommended to override this function and have it return false.



## Chapter 8 Custom Models

A custom model is a model which is derived from an “Extendable” model where the goal is to override or extend functionality thus creating a completely new Model based on an “Extendable” one.

An example of a Custom Model is the DemoPlayer class located at User.Models/GameCode/Server/AI.

```

1  #region
14
15 namespace WinterLeaf.Demo.Full.Models.User.GameCode.Server.AI
16 {
17     /// <summary> ...
20     [TypeConverter(typeof (TypeConverterGeneric<DemoPlayer>))]
21     public class DemoPlayer : AIPlayer
22     {
23         private static readonly Random r = new Random();
24
25         public override bool OnFunctionNotFoundCallTorqueScript()...
29
30         [ConsoleInteraction]
31         public virtual void Think()...
40
41         public virtual void MoveToNode(uint index)...
49
50         public virtual void MoveToNextNode()...
60
61         public virtual bool CheckForEnemy()...
131
132         public override void onMoveStuck(AIPlayer obj)...
143
144         public override void onReachDestination(AIPlayer obj)...
152
153         public override void onTargetEnterLOS(AIPlayer obj)...
157
158         public override void onTargetExitLOS(AIPlayer obj)...
162
163         public override void damage(GameBase sourceobject, Point3F position, float damage, string damagetype)...
203
204         public virtual void FollowPath(SimSet path, int node)...
228
229         AutoGen Operator Overrides
346     }
347 }

```

This class derives from AIPlayer which is a base model. In this case we are redefining how any View (T3D AIPlayer Object Instance) assigned to this model will behave by telling the Omni Framework not to use the AIPlayer class but instead the DemoPlayer class.

```
[TypeConverter(typeof (TypeConverterGeneric<DemoPlayer>))] ]
```

# OMNI ENGINE

The above code is needed for casting and conversion. This decorator automatically creates a TypeConverter for your custom object.

Also needed for proper operation is the code defined in the “AutoGen Operator Overrides”

```

70  #region AutoGen Operator Overrides
71
72  /// <summary> ...
78  public static bool operator ==(Observer ts, string simobjectid) ...
84
85  /// <summary> ...
89  public override int GetHashCode() ...
93
94  /// <summary> ...
99  public override bool Equals(object obj) ...
103
104  /// <summary> ...
110  public static bool operator !=(Observer ts, string simobjectid) ...
116
117
118  /// <summary> ...
123  public static implicit operator string(Observer ts) ...
129
130  /// <summary> ...
135  public static implicit operator Observer(string ts) ...
140
141  /// <summary> ...
146  public static implicit operator int(Observer ts) ...
153
154  /// <summary> ...
159  public static implicit operator Observer(int simobjectid) ...
163
164  /// <summary> ...
169  public static implicit operator uint(Observer ts) ...
176
177  /// <summary> ...
181  public static implicit operator Observer(uint simobjectid) ...
185
186  #endregion

```

Every custom Model you create must have these functions implemented so that the Omni Framework can properly cast your model to a string and other Model types. For the most part, you can just copy the code from an existing class and paste it in your new class and just change the type to the type of your new Model.

# OMNI ENGINE

There are times when you would want to create class(s) which derive from DemoPlayer. Assuming that DemoPlayer has all the base movement and combat code in it, we would create a new class which inherits from DemoPlayer but changes how it picks targets, (Let us call it TargetDemoPlayer). So when creating an AIPlayer View we would have 3 options for Models.

- AIPlayer – The default model assigned if none is specified.
- DemoPlayer
- TargetDemoPlayer

All three models could be casted down,

- TargetDemoPlayer could be casted down to a DemoPlayer and AIPlayer and lower.
- DemoPlayer could be casted down to an AIPlayer and lower.
- AIPlayer could be casted to anything lower.

This functionality gives us the ability to re-use an Omni T3D View Object, but assign different behavior sets by swapping out the C# Models.

So in a sense, applying this to the player, you could create a base player class, and then derive it to create a WarriorPlayer, WizardPlayer, TheifPlayer, etc. Then when you create the View in the Omni T3D engine, you would just assign the model that the player choose at character creation.

Another example:

In this example let's say you had a C++ class called "Ball". "Ball" has an "onCollision" event.

So after creating our Model we could spawn an Omni T3D Instance View object of "Ball" with our Model class as the back end for the C++ object.

So now we have a "Ball" C++ View object instance looking at our "Ball" C# Model.

We could then build a new Model object "BouncingBall" by inheriting our "Ball" Model. Change the "onCollision" to give it some upward velocity. And then spawn the object again with the typeof being "BouncingBall". Since we aren't changing the C++ code, we can reuse the core C++ View object code over and over again, just assigning a new Model object to it to change its behavior.

This design lets you re-use objects defined in the C++ by changing the Model logic behind them. You could end up having 5 balls in your simulation all using a different Model class as there back end.

For information on how to create objects and switch out the Models please see Chapter 5.

## Chapter 9 Building Gui's and converting them to C#

The Omni Framework offers many ways to build Gui's inside the game. As a developer you have the choice to build the Gui's using TorqueScript or C#. The C# model is based upon the Windows Form model.

To build Gui's using the TorqueScript model please see MIT T3D documentation at <http://docs.garagegames.com/torque-3d/official/index.html?content/documentation/Setup/Overview.html>.

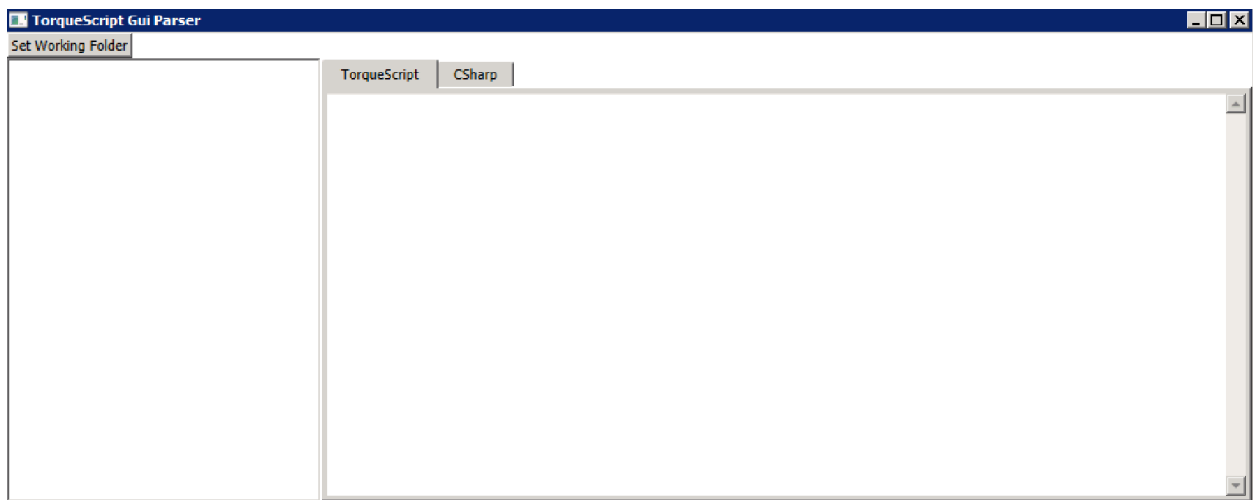
### Using the GuiParser

The GuiParser is a utility which is able to read a TorqueScript Gui file and convert it to C#. Since it is a parser it is not always 100% correct. Most common problems include the keywords like: NL, TAB and SPC. So always check the generated code for possible errors.

**Note: The GuiParser can also be used on Datablocks and Singletons.**

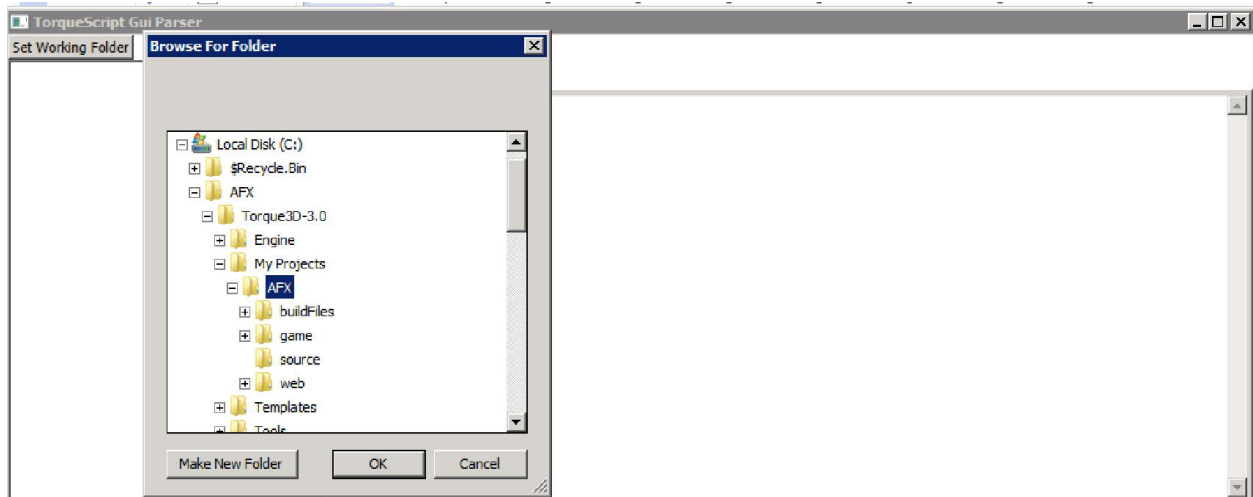
**Note: The GuiParser will not parse TorqueScript syntax like functions.**

When you start the program, a window will appear.

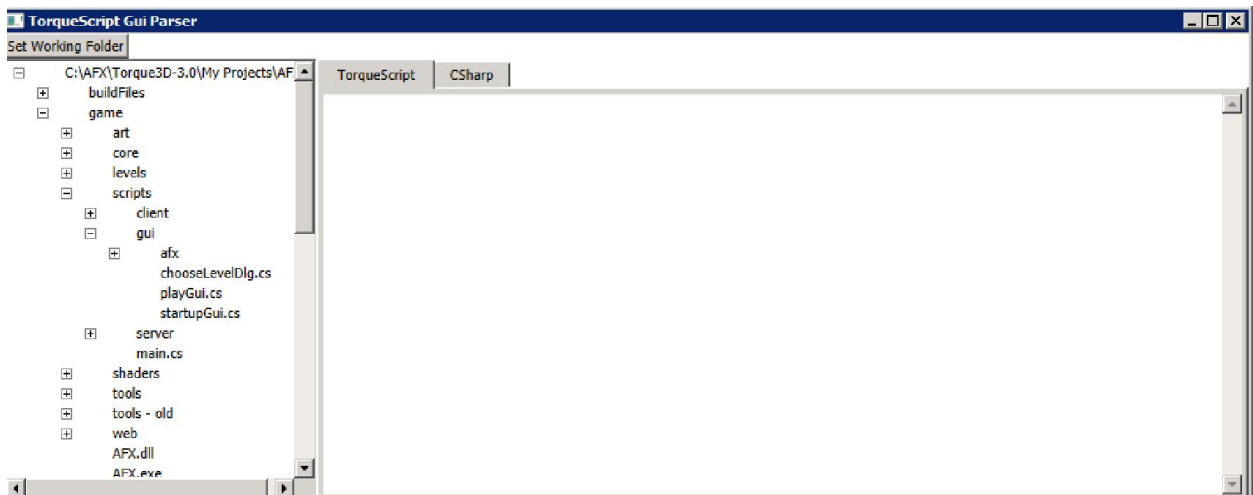


First set your working folder by clicking "Set Working Folder."

# OMNI ENGINE

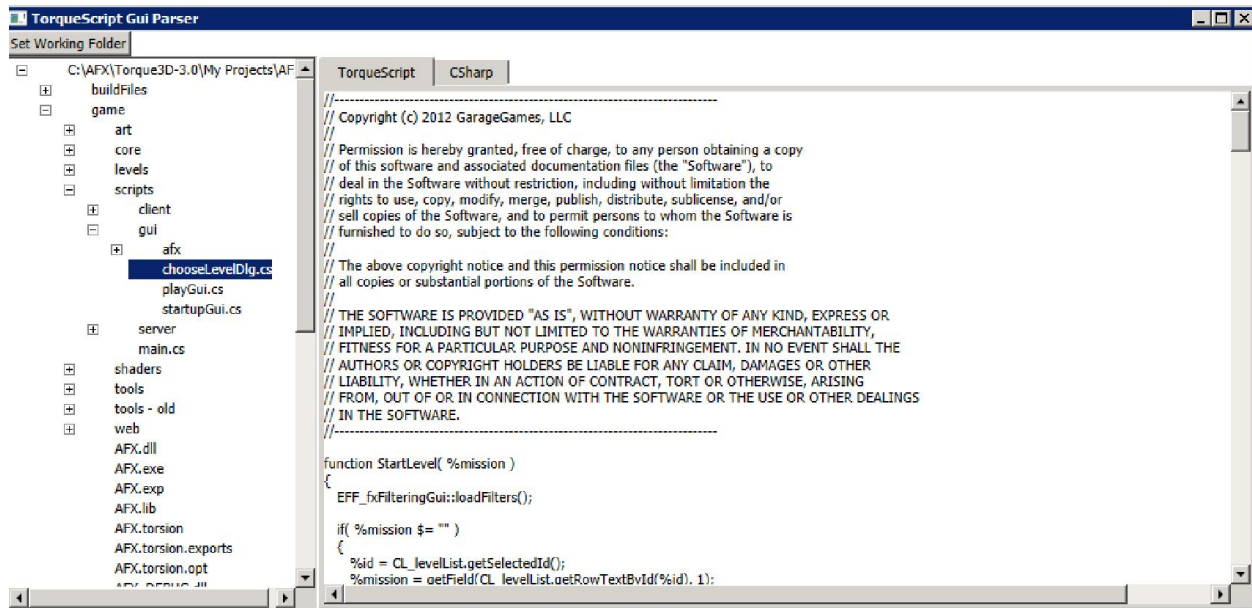


Once you set the working folder you will see a file tree on the left window.

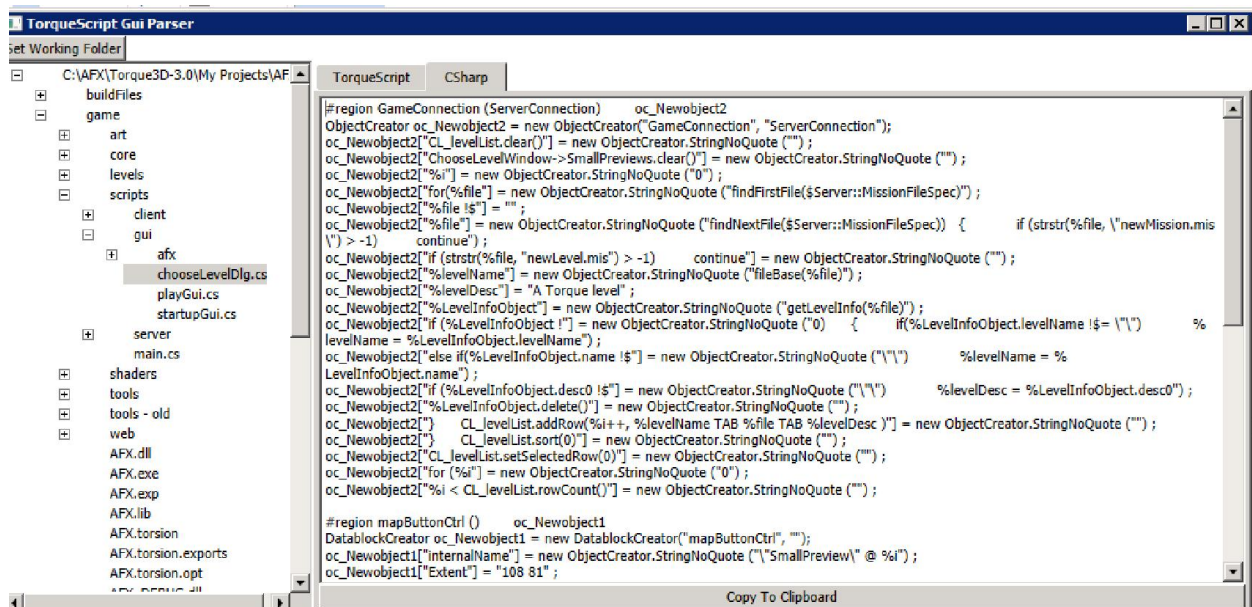


After navigating to where the Gui is saved in the game folder, click the file you wish to convert.

# OMNI ENGINE



It will show you the TorqueScript source to the file on the “TorqueScript” tab and the C# code on the “CSharp” tab.



Just copy the generated code into a C# class as a static function and add a call to it to load the screen.

If you need to change your Gui, you can either edit the C# directly or edit the screen using the T3D built in screen editor, save the TorqueScript, open it in the parser and reconvert it putting the generated C# code back into your function.

## Advanced Gui Creation

The GuiParser takes ninety percent of the work out of converting TorqueScript Gui files to C#. But once the Gui is converted you still need to wire up the events.

The Omni Framework provides two methods to wire up events inside of Gui's. You can either use the old style of putting the command to fire inside the click event, or wire the event up using the C# Model framework.

### Old Style

An example of putting the command to fire inside a click event would be:

**TorqueScript syntax:**

```
closeCommand = "CallMyFunction(MyVar) ;";
```

### C# Syntax

```
oc_Newobject11["closeCommand"] = "CallMyFunction(SomeVar);";
```

Where oc\_Newobject11 is some object creator and the CloseCommand is a member property on the T3D View object.

### New Style

First we would modify our code and add the C# Model name we wish to attach to the gui component.

```
ObjectCreator oc_Newobject11 = new ObjectCreator("GuiWindowCtrl", "ChooseLevelWindow",  
    typeof(ChooseLevelWindow));  
oc_Newobject11["canSaveDynamicFields"] = "0";  
oc_Newobject11["Enabled"] = "1";
```

The above code sample is creating a new "GuiWindowCtrl" which is named "ChooseLevelWindow" and it is assigning the C# Model "ChooseLevelWindow" to it.

After assigning the C# Model to our T3D View, we then can create the view like such.

# O M N I E N G I N E

```

[TypeConverter(typeof (TypeConverterGeneric<chooseLevelDlg>))]
public class chooseLevelDlg : UIControl
{
    public override bool OnFunctionNotFoundCallTorqueScript()...

    [ConsoleInteraction(true)]
    public static void StartLevel(string mission, string hostingType)...

    public override void onWake()...

    public void addMissionFile(string file)...

    public override void onSleep()...

    public string getLevelInfo(string missionFile)...

    public string getLevelInfo1(string missionFile)...

    public static void Initialize()...

    AutoGen Operator Overrides
}

```

As you can see we are then able to override the events needed to handle things like mouse clicks, etc. This provides a clear and readable binding between the Gui objects and the code behind them.

Below is an example of a C# button Model.



# O M N I E N G I N E

```
[TypeConverter(typeof (TypeConverterGeneric<chooseLevelDlgGoBtn>))]  
public class chooseLevelDlgGoBtn : GuiButtonCtrl  
{  
    public override bool OnFunctionNotFoundCallTorqueScript()  
    {  
        return false;  
    }  
  
    // Do this onMouseUp not via Command which occurs onMouseDown so we do  
    // not have a lingering mouseUp event lingering in the ether.  
    public override void onMouseUp()  
    {  
        // So we can't fire the button when loading is in progress.  
        if ("ServerGroup".isObject())  
            return;  
  
        if (((chooseLevelDlg) "ChooseLevelDlg")["launchInEditor"].AsBool())  
        {  
            Util.activatePackage("BootEditor");  
            ((chooseLevelDlg) "ChooseLevelDlg")["launchInEditor"] = false.AsString();  
            chooseLevelDlg.StartLevel("", "SinglePlayer");  
        }  
        else  
        {  
            chooseLevelDlg.StartLevel("", "");  
        }  
    }  
}  
  
+ AutoGen Operator Overrides  
}
```

Ultimately the decision of which design paradigm to follow is up to the programmer, but it is highly recommended to use the Windows Form paradigm.

## Chapter 10 Global Functions

Due to the difference between TorqueScript and C#, it is not possible to define Global functions in some C# file outside of a class file. Because of this, the decision was made that all global functions would be declared inside a class file and the defining attribute that made them global was flagging the function as “static” in C#.

We moved all global functions that dangled all over the FPS Demo game into the class files they mostly applied to in an effort to organize them in a manageable way. So instead of having one static class with a long list of static functions, we moved the static functions into their respective classes.

There were cases when for C# naming convention purposes we ended up with same name used twice in two classes. Because of this we added the ability to take a static function and alias its name to something different.

This is done in the ConsoleInteraction decoration.

```
121 | [ConsoleInteraction (true, "SomeAlias")]  
122 | public static void ToggleConsole(bool make)
```

The above code would rename the static function “ToggleConsole” to “SomeAlias” when exposing it to the Omni T3D console. In the event we had more than one “ToggleConsole” function, we could alias them as “SomeAlias1”, “SomeAlias2”, “SomeAlias3”, etc.

This way if we had used the same static function name in more than one class, we could have it exposed to the console as something different for each class but structurally in C# the functions would be called the same.

For more information on Global Function implementation please see Chapter 6.

## Chapter 11 Run-Time C# Programming (LiveScripts!)

LiveScripts provides a mechanism to load C# Models and functions into the game without compiling them into the source code. This feature allows you to modify the running C# game scripts without restarting the game. Especially useful when debugging code during development or allowing end-users of your game to modify game play.

All LiveScripts must be in the Game/csScripts folder. When the game starts, it will read this folder and compile all of the C# code into an in-memory DLL. If the DLL compiles without errors, it is then loaded into the Omni Framework and it can be used just like any other class. Any time you change the C# or XML configuration files, all objects that are defined in the modified C# files or XML Files is automatically deleted from memory and passed for garbage collection.

**NOTE:** If compilation errors occur when you save the C# script, you will need to re-save the XML file to have the C# script recompile.

There are two types of LiveScripts:

1. **Classes** – Used when you wish to add new C# Model classes to your game.
2. **MemberConsoleFunctionOverride** – Used when you wish to replace or add a Console Function to a C# Model Class.

All LiveScripts require an XML file for configuration. An Example XML file is:



```
<WLEOverrideConfig Type="Classes">
  <Files>
    <FileName>CustomHealthpack\CustomHealthPack.cs</FileName>
  </Files>
  <ReferencedAssemblies>
    <Assembly>System.dll</Assembly>
    <Assembly>Microsoft.CSharp.dll</Assembly>
    <Assembly>System.Core.dll</Assembly>
    <Assembly>System.Data.dll</Assembly>
    <Assembly>System.Data.DataSetExtensions.dll</Assembly>
    <Assembly>System.Xml.dll</Assembly>
    <Assembly>System.Xml.Linq.dll</Assembly>
    <Assembly>Winterleaf.Demo.Full.dll</Assembly>
    <Assembly>WinterLeaf.Engine.Omni.dll</Assembly>
  </ReferencedAssemblies>
</WLEOverrideConfig>
```

LiveScript will compile all of the <Files> into a DLL and register them inside the Omni Framework. Remember to add the referenced assemblies under <ReferencedAssemblies> tag, as shown in the picture above.

# OMNI ENGINE

## Steps you need to take for Classes:

- Define the `<WLEOverrideConfig/OverrideConfig>` Type to **Classes**.

```
<WLEOverrideConfig Type="Classes">
```

- Add the new files under the `<Files>` tag.

```
<Files>  
  <FileName>CustomHealthpack\CustomHealthPack.cs</FileName>  
</Files>
```

- Add the referenced libraries/dlls inside the `<ReferencedAssemblies>` tag.

```
<ReferencedAssemblies>  
  <Assembly>System.dll</Assembly>  
  <Assembly>Microsoft.CSharp.dll</Assembly>  
  <Assembly>System.Core.dll</Assembly>  
  <Assembly>System.Data.dll</Assembly>  
  <Assembly>System.Data.DataSetExtensions.dll</Assembly>  
  <Assembly>System.Xml.dll</Assembly>  
  <Assembly>System.Xml.Linq.dll</Assembly>  
  <Assembly>Winterleaf.Demo.Full.dll</Assembly>  
  <Assembly>WinterLeaf.Engine.Omni.dll</Assembly>  
</ReferencedAssemblies>
```

For **MemberConsoleFunctionOverride**, there are some extra steps to be taken:

- Define the `<WLEOverrideConfig/OverrideConfig>` Type to **MemberConsoleFunctionOverride**

```
<OverrideConfig Type = "MemberConsoleFunctionOverride">
```

- Define the `<NameSpace>` to the class you wish to add/override methods to.

```
<NameSpace>WinterLeaf.Demo.Full.Models.User.Extendable.Player</NameSpace>
```

In this case, **WinterLeaf.Demo.Full.Models.User.Extendable.Player** class is being overridden.

# OMNI ENGINE

- The referenced libraries and the files to be included are referenced as in the case of **Classes**.

```
<Files>
  <FileName>Player\Player_Script.cs</FileName>
</Files>
<ClassName>Player_Script</ClassName>
<ReferencedAssemblies>
  <Assembly>System.dll</Assembly>
  <Assembly>Microsoft.CSharp.dll</Assembly>
  <Assembly>System.Core.dll</Assembly>
  <Assembly>System.Data.dll</Assembly>
  <Assembly>System.Data.DataSetExtensions.dll</Assembly>
  <Assembly>System.Xml.dll</Assembly>
  <Assembly>System.Xml.Linq.dll</Assembly>
  <Assembly>Winterleaf.Demo.Full.dll</Assembly>
  <Assembly>WinterLeaf.Engine.Omni.dll</Assembly>
</ReferencedAssemblies>
```

- Add the functions to be overridden/added under **<FunctionOverrides>**.

```
<FunctionOverrides>
  <Function>playPain</Function>
  <Function>playDeathCry</Function>
  <function>DoSomethingcool</function>
</FunctionOverrides>
```

TADA! With these simple steps, you'll able to create a **livescript**, which can be edited without exiting your game/compiling into the game.

## Chapter 12 File Dialogs

OMNI uses a new way of using the file dialogs, as opposed to T3D. The file dialog in OMNI is used in combination with C# form file dialogs, so you can define a ton of different properties as per your need. The file dialogs support callbacks with the use of delegates. The callback function accepts the DialogResult as the parameter, and will be called after the dialog execution is complete. By default, it doesn't include a callback.

This is a simple example of a **file dialog**.

*Figure 1: File Dialog with delegate*

```
[ConsoleInteraction]
public static void Test_FileDialogWithDelegate()
{
    OpenFileDialog fd = new OpenFileDialog();
    DialogResult dr = Engine.Classes.Helpers.Dialogs.FileDialog(ref fd, Test_FileDialogWithDelegate_finished);
}
```

*Figure 2: File Dialog without delegate*

```
public static void Test_FileDialogWithoutDelegate()
{
    OpenFileDialog fd = new OpenFileDialog();
    DialogResult dr = Engine.Classes.Helpers.Dialogs.FileDialog(ref fd);
    MessageBox.Show(dr.ToString());
}
```

**NOTE:** Make sure you call Dialogs.GetForwardSlashFile to get the filename on request complete. This ensures you get the correct filename for use.

There are two type of file dialogs in OMNI.

1. **OpenFileDialog:**

For OpenFileDialog, you need to create an object of **System.Windows.Forms.OpenFileDialog** class and pass its reference to the static OpenFileDialog constructor.

If you need to use a callback for the dialog, you can pass the delegate as shown in Figure 1 above. Below is an example of OpenFileDialog:

```
var ofd = new System.Windows.Forms.OpenFileDialog();
ofd.FileName = currentFile;
ofd.Filter = filespec;
ofd.CheckFileExists = true;
ofd.Multiselect = false;
//FileDialog fd = ofd;

if (omni.Util.filePath(currentFile) != "")
    ofd.InitialDirectory = omni.Util.filePath(currentFile);

DialogResult dr = Engine.Classes.Helpers.Dialogs.OpenFileDialog(ref ofd);

if (dr == DialogResult.OK)
{
    var fileName = Dialogs.GetForwardSlashFile(ofd.FileName);
    omni.Util.eval(callback + "(" + fileName + ")");
    omni.sGlobal["$Tools::FileDialogs::LastFilePath"] = omni.Util.filePath(fileName);
}
```

## 2. SaveFileDialog:

For SaveFileDialog, you have to create an object of **System.Windows.Forms.SaveFileDialog** and pass its reference to the static SaveFileDialog constructor. For the callback, you can use it the same way as OpenFileDialog.

This is a simple example of SaveFileDialog.

```
var sfd = new System.Windows.Forms.SaveFileDialog
{
    Filter = omni.sGlobal["$GUI::FileSpec"],
    InitialDirectory = omni.Util.makeFullPath(defaultPath, ""),
    FileName = defaultFileName,
    OverwritePrompt = true,
};

string filename = "";

DialogResult dr = Dialogs.SaveFileDialog(ref sfd);

if (dr == DialogResult.OK)
{
    filename = Dialogs.GetForwardSlashFile(sfd.FileName);
    GuiEditor["LastPath"] = omni.Util.filePath(filename);
    //filename = dlg["FileName"];
    if (omni.Util.fileExt(filename) != ".gui")
        filename = filename + ".gui";
}
```

## Chapter 13 Threading with the Omni Framework

Interacting with T3D from an asynchronous thread can be tricky. Thankfully we have added a mechanism to the OMNI Framework to assist you.

First, let the reader be warned, Threading is an advanced topic and should be avoided unless absolutely necessary.

An example of threading is in the Model.User/GameCode/AI/AI.cs file. This class demonstrates how to do the threading properly.

This example's purpose is to rid the logic from using the "Schedule" function inside of T3D. Instead we are keeping track of all AI's in a collection and tracking their time when to activate outside of T3D.

The whole logic starts in the function "CreateAI", this console function will create and spawn the AI monsters.

```

74 public static void createAI(int count)
75 {
76     if (count == 0)
77         return;
78
79     if (lastcount > 0)
80     {
81         omni.console.error("Mobs already spawned");
82         return;
83     }
84     using (BackgroundWorker bwr_AIThought = new BackgroundWorker())
85     {
86         bwr_AIThought.DoWork += bwr_AIThought_DoWork;
87         bwr_AIThought.RunWorkerAsync();
88     }
89
90     lastcount = count;
91     int team = 0;
92     for (int i = 0; i < count; i++)
93     {
94         team++;
95         if (team == 3)
96             team = 1;
97
98         ObjectCreator ocf = new ObjectCreator("ScriptObject", "Mob" + i.AsString());
99         ocf["player"] = "";
100         ocf["aiteam"] = team.AsString();
101         ScriptObject MobRoot = ocf.Create();
102         ((SimSet)"rootgroup").pushToBack(MobRoot);
103         BackgroundWorker spawnwait = new BackgroundWorker();
104         spawnwait.DoWork += spawnwait_DoWork;
105         spawnwait.RunWorkerAsync(new threadparam(i * 10, MobRoot));
106     }
107 }

```



## OMNI ENGINE

So, when you type into the console “CreateAI(10);” the function does some basic checks to make sure you haven’t spawned monsters already. This is done in lines 76 to 83. If you have spawned monsters already, the function quietly exits out.

Otherwise, the code will create a new BackgroundWorker (C# threading object) which will monitor a Queue of intervals that tell the C# when to process the thinking process for each AI.

```
84         using (BackgroundWorker bwr_AIThought = new BackgroundWorker())
85         {
86             bwr_AIThought.DoWork += bwr_AIThought_DoWork;
87             bwr_AIThought.RunWorkerAsync();
88         }
```

The syntax is pretty simple, line 84 we create the new BackgroundWorker object “bwr\_AIThought”. We assign the name of the function to call when the thread is started on line 86.

```
86             bwr_AIThought.DoWork += bwr_AIThought_DoWork;
```

Then we kick the thread off asynchronously on line 87.

```
87             bwr_AIThought.RunWorkerAsync();
```

We will get into what the “bwr\_AIThought\_DoWork” function does farther down, but let’s look at the remaining code.

```
89         lastcount = count;
90         int team = 0;
91         for (int i = 0; i < count; i++)
92         {
93             team++;
94             if (team == 3)
95                 team = 1;
96
97             ObjectCreator ocf = new ObjectCreator("ScriptObject", "Mob" + i.AsString());
98             ocf["player"] = "";
99             ocf["ateam"] = team.AsString();
100             ScriptObject MobRoot = ocf.Create();
101             ((SimSet) "rootgroup").pushToBack(MobRoot);
102             BackgroundWorker spawnwait = new BackgroundWorker();
103             spawnwait.DoWork += spawnwait_DoWork;
104             spawnwait.RunWorkerAsync(new threadparam(i * 10, MobRoot));
105         }
```

We are looping from 0 to “count” times, where count is the number of AI’s you wish to spawn. We create a new ScriptObject to manage our AI instance and assign the team number to it. After creating the ScriptObject on line 101, we then push it into the “rootgroup”.

This doesn’t create the AI, it creates an AI Manager, so now we need to kick off creating the AI. We could have used a “Schedule” call here.

# OMNI ENGINE

On line 103, we create a new BackgroundWork object this time telling it to call the function “spawnwait\_DoWork”, and we are passing a parameter to the function when we invoke the thread on line 105.

Looking at the code in “spawnwait\_DoWork” we see:

```
1 reference | Vincent Gee, 9 hours ago | 1 change
132 public static void spawnwait_DoWork(object sender, DoWorkEventArgs e)
133 {
134     threadparam tp = (threadparam)e.Argument;
135     Thread.Sleep(tp.Delay);
136     lock (Omni.self.tick)
137         spawnAI(tp.MobRoot);
138 }
```

The background worker only allows us to pass one parameter to a thread, so if you need to send multiple parameters to a thread function, it must be inside a class. In this case we created a “threadparam” class to hold our values to be passed to this function.

The passed parameter object will always be in EventArgs.Argument object. To use it, we need to cast it back to our class type as seen on line 134.

So, in this code example this thread will read the arguments and sleep the thread for the delay specified, after the duration is over, the thread will lock the T3D process and inject a call to the function “spawnAI”. The syntax for locking the T3D process is

```
136 lock (Omni.self.tick)
```

After locking the thread, we can then call the function “spawnAI” safely without worrying about corrupting the T3D memory stacks.

Now, going back to the function “bwr\_AIthough\_DoWork”.

```
144 public static void bwr_AIthough_DoWork(object sender, DoWorkEventArgs e)
145 {
146     while (lastcount > 0 && Omni.self.IsRunning)
147     {
148         List<AIInterval> t =
149             m_thoughtqueue.ToArray().Where(item => item.Intervaltime < DateTime.Now).Take(20).ToList();
150         lock (Omni.self.tick)
151             foreach (AIInterval item in t)
152             {
153                 m_thoughtqueue.Remove(item);
154                 if (item.Player_id.AsString().isObject())
155                     ((DemoPlayer)item.Player_id).Think();
156             }
157         Thread.Sleep(100);
158     }
159 }
```

This function is in a continuous loop until the lastcount equal zero or the game is shut down. Using lambda we query our queue of AI’s that need to think where their interval time is less than the current system time and we only take the top 20 objects from the queue. This is to protect against say having

# OMNI ENGINE

100 AI's that need to think and processing them all at once which would cause a hitch in the game because it would pause the engine while it is processing there "Think" function.

So on line 150, we lock the "Tick" and start a loop through each of our fetched AIIntervals. The program removes the interval from the main queue and then checks to make sure that the player\_ID is an object. This is the actual ID for the bot. If the bot exists, i.e. hasn't been killed, we then call the "Think" function on that bot. Once all 20 bots have been processed the whole thread pauses 100 ms to give the simulation time to catch up before it repeats the process.

In a situation where you create 500 AI's, the AI's will begin to act very strange because even though you told the AI to think every 500 ms, it might not actually think for 600, 700 or 1000 ms due to load. Why would you do this?

Well if you were running 400 AI and using the T3D schedule routine to process there thought, there is a possibility that all 400 AI would process there thinking in order, thus stopping the game while it processed all 400. In this threading example we are limiting the engine to only processing 20 AI moves every 100 ms. So yes the AI can get backlogged, but the will never cause your game to crash or slow to a standstill.

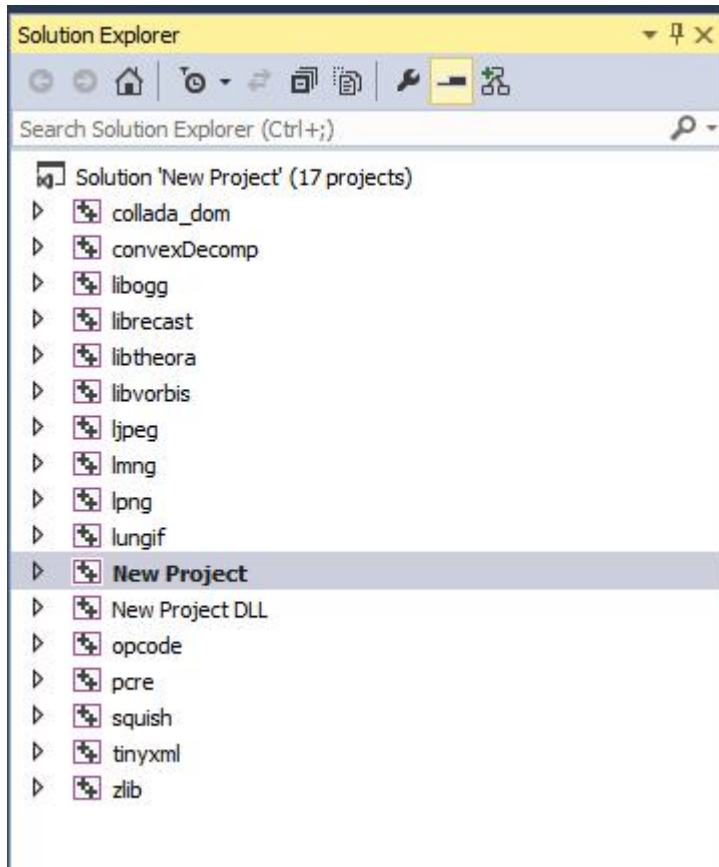
The only hard rule in threading is that you must call "Lock(Omni.self.tick)" before interacting with the T3D dll from outside a callback from the engine.

## Chapter 14 Debugging C# and C++

Debugging is always a challenge, how can you debug the C++ and the C# at the same time. In this example I created a new project using the “C#-Full” template.

**NOTE: This only works with Visual Studio 2010/2013 Professional and higher. The express versions of Visual Studio do not allow mixed solutions.**

First thing to do is open up the “New Project.sln” inside of Visual Studio.



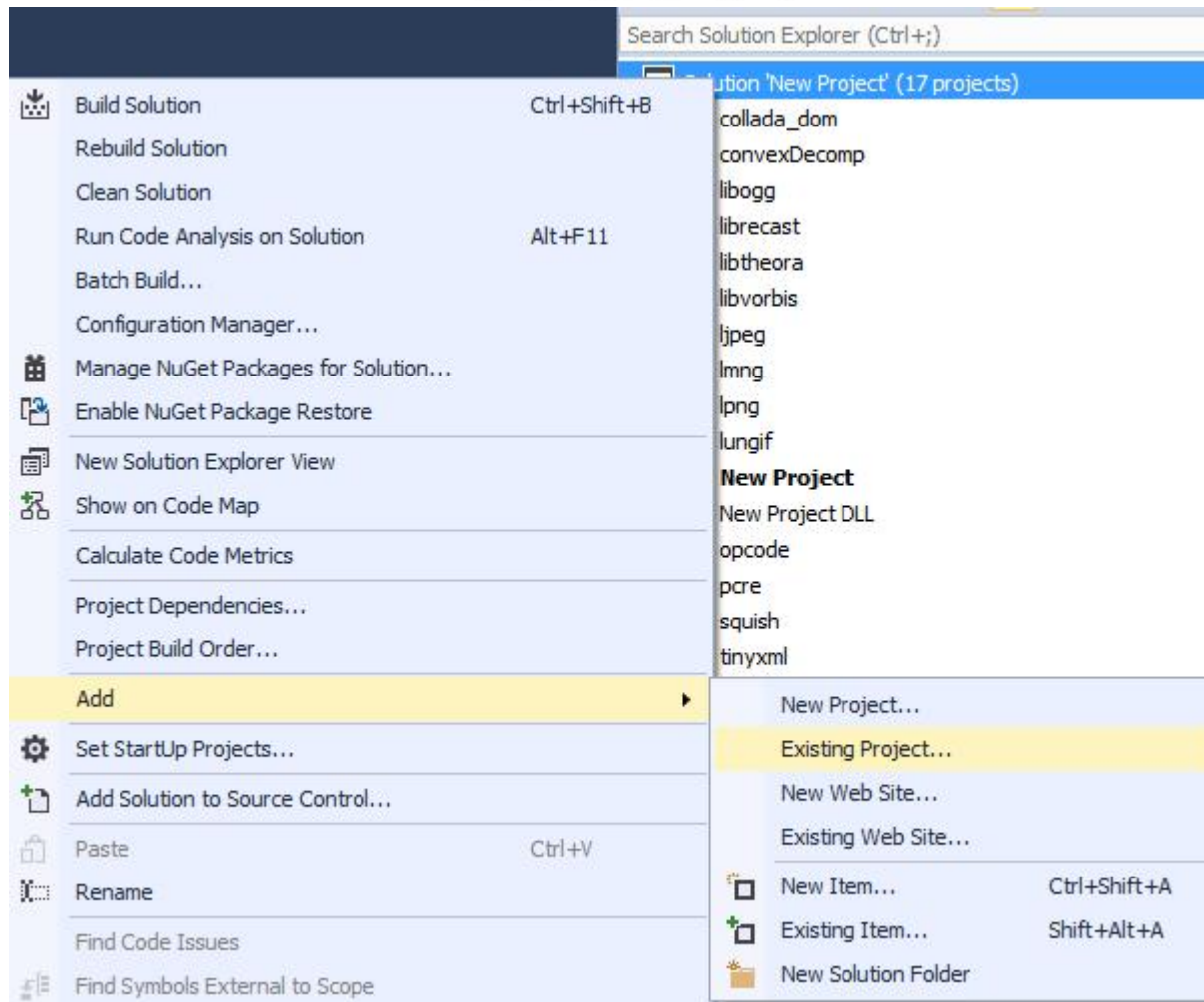
This looks like a normal T3D solution with all the C++ projects in it. Where are the C# projects?

Well since I had to set OMNI up for people who use the Express versions of Visual Studio I split the solutions into two separate solutions. All the C++ is in the <Project Name>.sln and all of the C# is in a solution called Omni Framework (<Project Name>).

The key to debugging is getting all the projects into one solution, so the first step is to add the C# projects to the C++ solution.

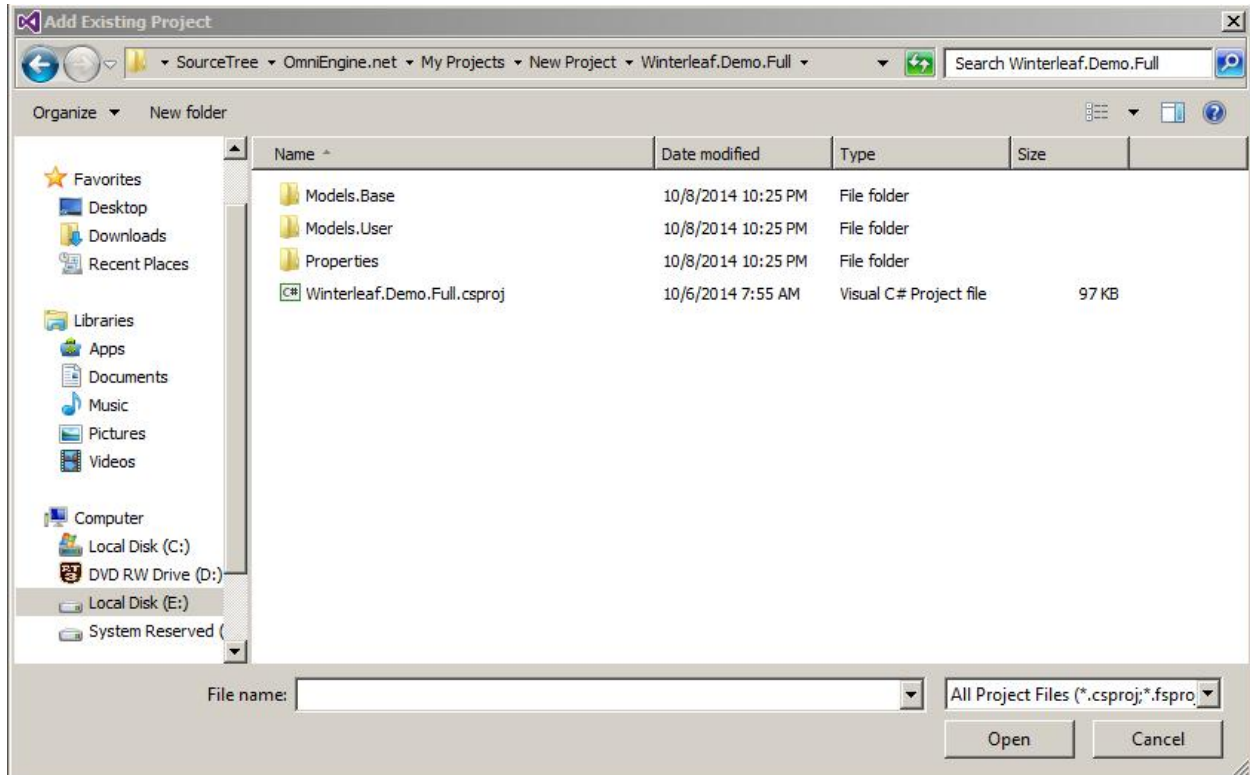
So click on the solution, right click -> add->Existing Project.

# OMNI ENGINE



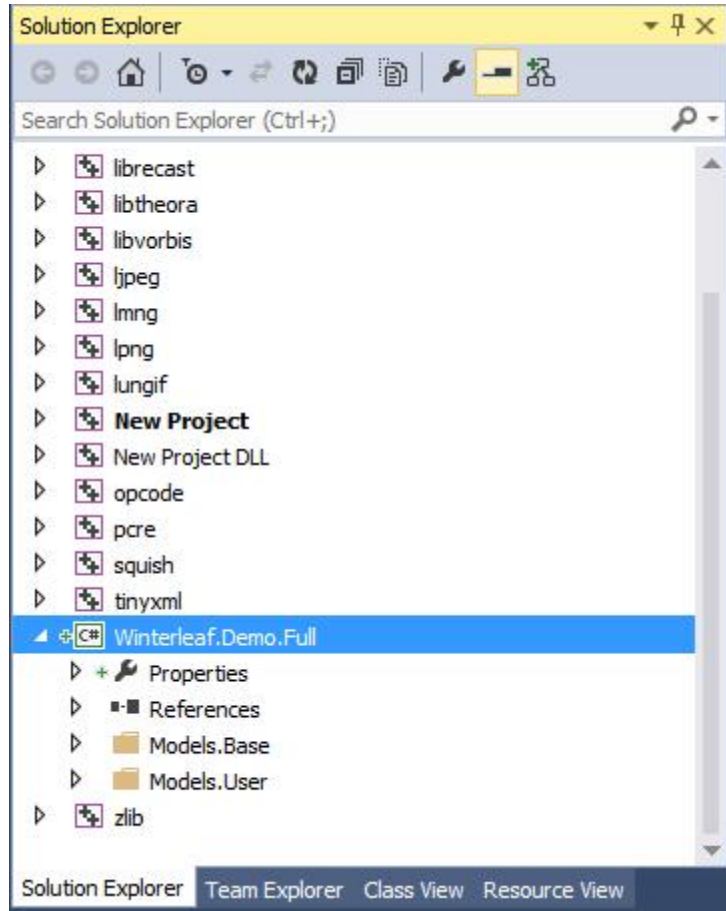
Navigate to your Project folder and into the “Winterleaf.Demo.Full” project folder.

# OMNI ENGINE



The solution should now look like the picture below.

# OMNI ENGINE



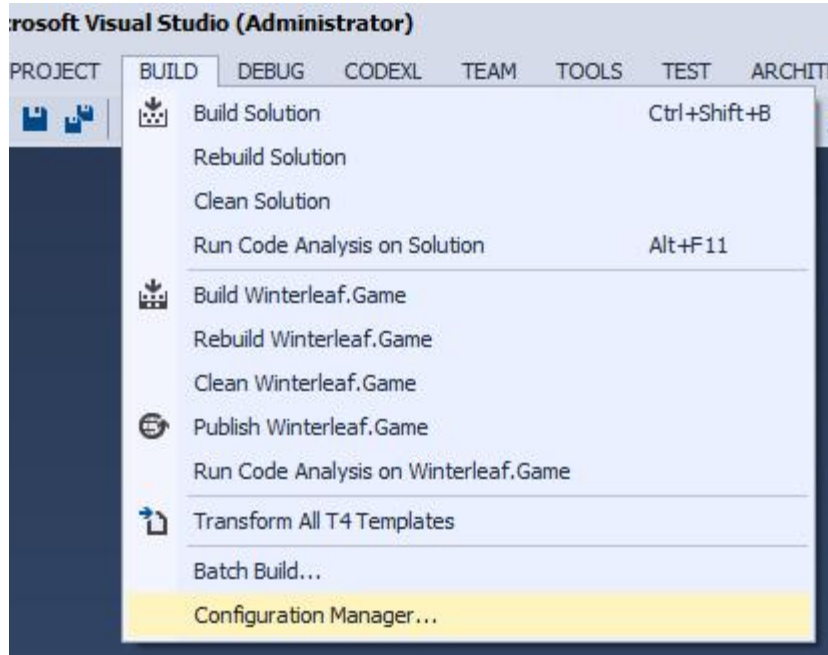
We repeat the process for the “Winterleaf.Engine.Omni” and “Winterleaf.Game” projects. After adding those two remaining projects to our C++ solution our solution should look like the picture below.

The screenshot shows the Visual Studio Solution Explorer. At the top, the title bar reads 'Solution Explorer'. Below it is a toolbar with icons for navigation and development. A search bar contains the text 'Search Solution Explorer (Ctrl+;)'. The main area displays a tree view of the solution 'New Project' (20 projects). The projects listed are: collada\_dom, convexDecomp, libogg, libreicast, libtheora, libvorbis, ljpeg, lpng, lungif, New Project, New Project DLL, opcode, pcre, squish, tinyxml, Winterleaf.Demo.Full, Winterleaf.Engine, Winterleaf.Game (selected), and zlib. The 'Winterleaf.Game' project is highlighted with a blue background.

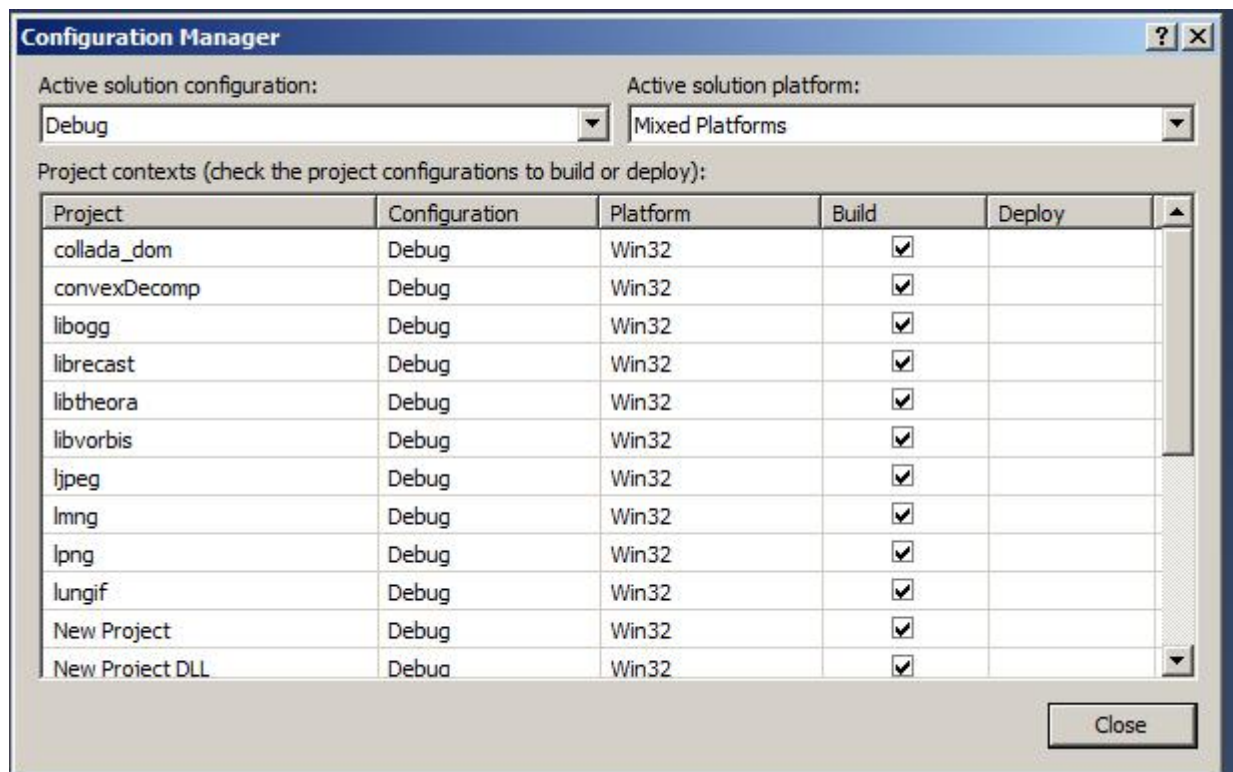
Now the next step is to configure the Build. So go to Build->Configuration Manager



# OMNI ENGINE



It should look something like this.



We need to change the “Active Solution Platform” to either “Win32” or “X64”, really your choice on whether to build the 32 or 64 bit executable. In my example I will choose the “X64” configuration.

# O M N I E N G I N E

Make sure that the C# projects are set to the same Build configuration as the C++ projects. If building a 64 bit configuration it should look like the following.

Winterleaf.Demo.Full	Debug	x64	<input checked="" type="checkbox"/>	
Winterleaf.Engine	Debug	x64	<input checked="" type="checkbox"/>	
Winterleaf.Game	Debug	x64	<input checked="" type="checkbox"/>	

A 32 bit configuration would look like:

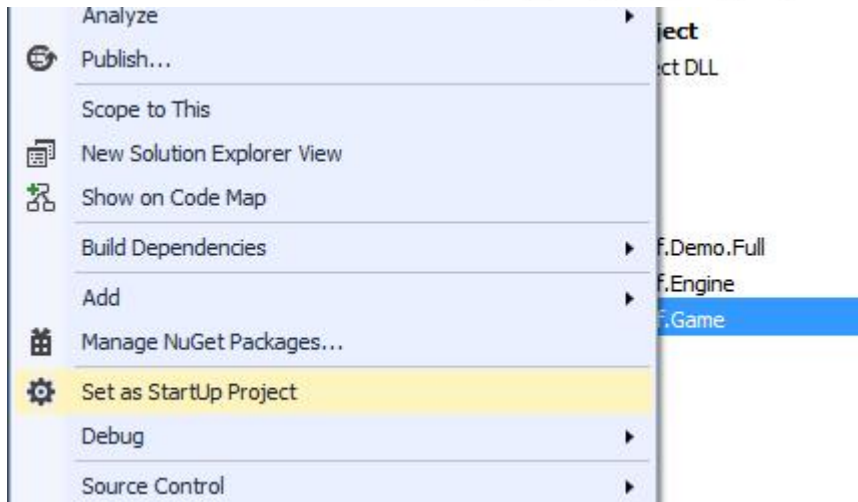
tinyxml	Debug	Win32	<input checked="" type="checkbox"/>	
Winterleaf.Demo.Full	Debug	x86	<input checked="" type="checkbox"/>	
Winterleaf.Engine	Debug	x86	<input checked="" type="checkbox"/>	
Winterleaf.Game	Debug	x86	<input checked="" type="checkbox"/>	

Since we are using OMNI we do not need to build the stock C++ executable anymore so we can uncheck that from the build.

librecast	Debug	x64	<input checked="" type="checkbox"/>	
libtheora	Debug	x64	<input checked="" type="checkbox"/>	
libvorbis	Debug	x64	<input checked="" type="checkbox"/>	
ljpeg	Debug	x64	<input checked="" type="checkbox"/>	
lmng	Debug	x64	<input checked="" type="checkbox"/>	
lpng	Debug	x64	<input checked="" type="checkbox"/>	
lungif	Debug	x64	<input checked="" type="checkbox"/>	
New Project	Debug	x64	<input type="checkbox"/>	
New Project DLL	Debug	x64	<input checked="" type="checkbox"/>	
opcode	Debug	x64	<input checked="" type="checkbox"/>	
pcre	Debug	x64	<input checked="" type="checkbox"/>	
squish	Debug	x64	<input checked="" type="checkbox"/>	

Now that we have the projects configured, we need to switch the startup application to our C# project. This is done by right clicking on "Winterleaf.Game"->Set as Startup Project.

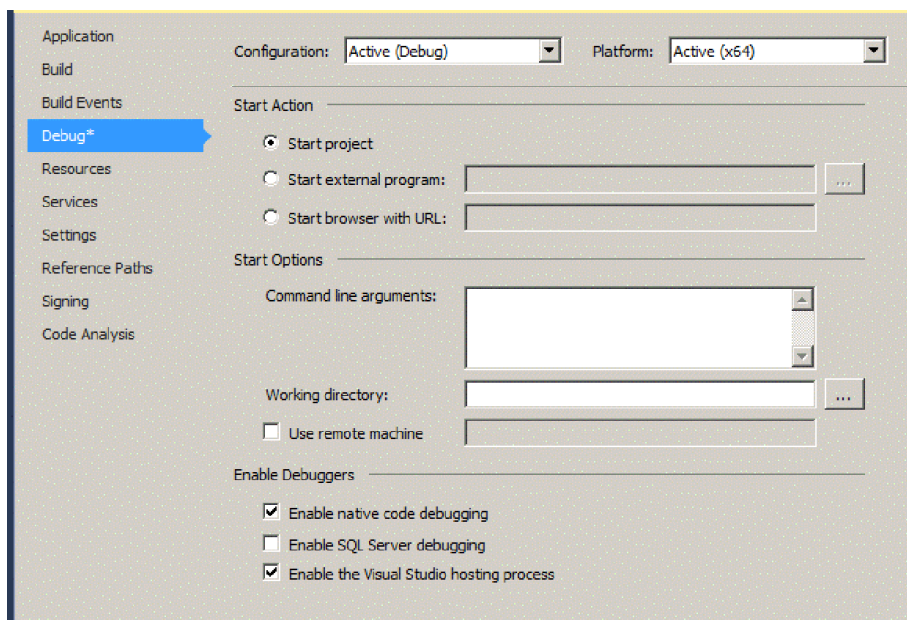
# O M N I E N G I N E



This tells Visual Studio that the project “Winterleaf.Game” is the project it should start when you fire up the debugger.

Next, we need to allow the debugger to be able to debug both the C++ and C# projects in the same session. To do this, right click on the “Winterleaf.Demo.Full” project and click properties.

Go to the “Debug” tab, and click the “Enable native code debugging” check box.



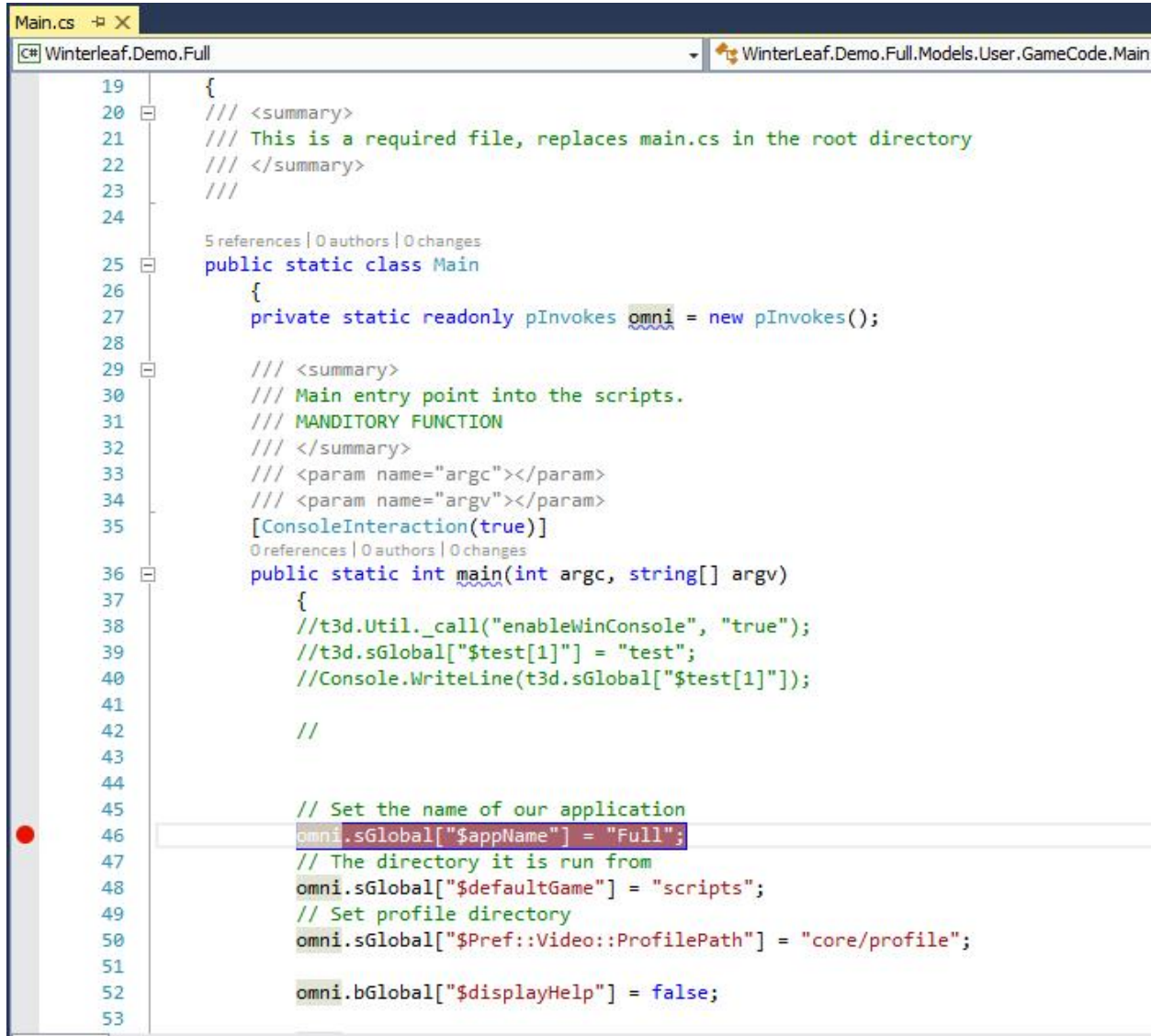
You will need to do this on each of the C# projects. This will allow you to set break points in both the C++ and the C# as well as step debug from the C# into the C++ and back out again while debugging.

Before building our new solution, make sure you run the Static Code Analyzer (Chapter 3).

# OMNI ENGINE

After running the Static Code Analyzer you are then able to build the Solution.

After it is built, open the Winterleaf.Demo.Full project and navigate down to Model.User/GameCode/Main.cs. Place a breakpoint on line 46.

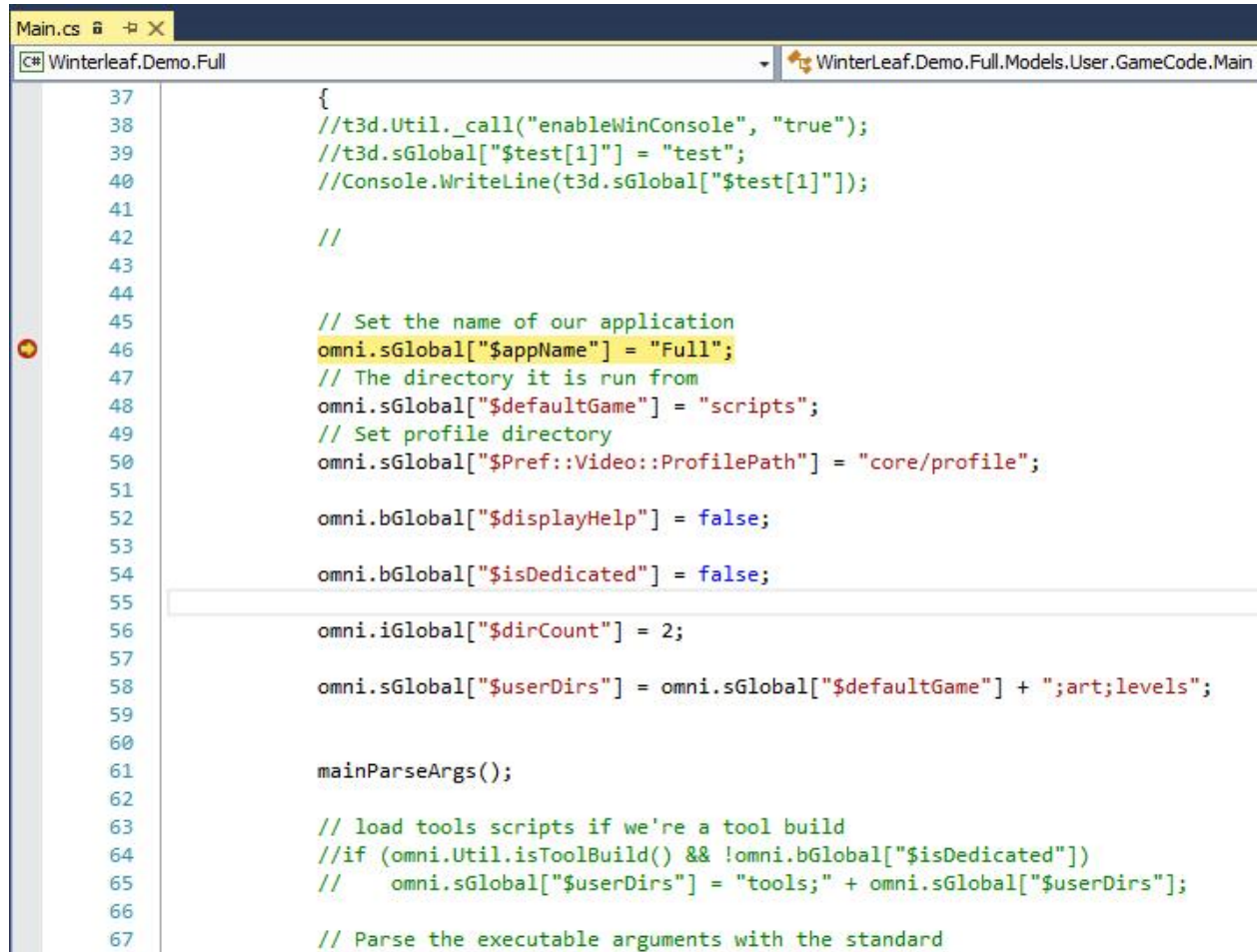


```
19 {
20     /// <summary>
21     /// This is a required file, replaces main.cs in the root directory
22     /// </summary>
23     ///
24
25     5 references | 0 authors | 0 changes
26     public static class Main
27     {
28         private static readonly pInvokes omni = new pInvokes();
29
30         /// <summary>
31         /// Main entry point into the scripts.
32         /// MANDATORY FUNCTION
33         /// </summary>
34         /// <param name="argc"></param>
35         /// <param name="argv"></param>
36         [ConsoleInteraction(true)]
37         0 references | 0 authors | 0 changes
38         public static int main(int argc, string[] argv)
39         {
40             //t3d.Util._call("enableWinConsole", "true");
41             //t3d.sGlobal["$test[1]"] = "test";
42             //Console.WriteLine(t3d.sGlobal["$test[1]"]);
43
44             //
45
46             // Set the name of our application
47             omni.sGlobal["$appName"] = "Full";
48             // The directory it is run from
49             omni.sGlobal["$defaultGame"] = "scripts";
50             // Set profile directory
51             omni.sGlobal["$Pref::Video::ProfilePath"] = "core/profile";
52
53             omni.bGlobal["$displayHelp"] = false;
54         }
55     }
56 }
```

After placing the Break Point, start the game up in debug mode.

After starting the game, you should see a yellow highlight where your break point is

# OMNI ENGINE



```
37 {
38     //t3d.Util._call("enableWinConsole", "true");
39     //t3d.sGlobal["$test[1]") = "test";
40     //Console.WriteLine(t3d.sGlobal["$test[1]"]);
41
42     //
43
44
45     // Set the name of our application
46     omni.sGlobal["$appName"] = "Full";
47     // The directory it is run from
48     omni.sGlobal["$defaultGame"] = "scripts";
49     // Set profile directory
50     omni.sGlobal["$Pref::Video::ProfilePath"] = "core/profile";
51
52     omni.bGlobal["$displayHelp"] = false;
53
54     omni.bGlobal["$isDedicated"] = false;
55
56     omni.iGlobal["$dirCount"] = 2;
57
58     omni.sGlobal["$userDirs"] = omni.sGlobal["$defaultGame"] + ";art;levels";
59
60
61     mainParseArgs();
62
63     // load tools scripts if we're a tool build
64     //if (omni.Util.isToolBuild() && !omni.bGlobal["$isDedicated"])
65     //    omni.sGlobal["$userDirs"] = "tools;" + omni.sGlobal["$userDirs"];
66
67     // Parse the executable arguments with the standard
```

Enjoy Debugging!



## Appendix

### Appendix 1 - Static Code Generator Configuration Options

The Static Code Generator is an application which takes care of setting up the linkage between C# and C++. It achieves this by reading all of the C++ source code and generating Extern's in the C++ and delegates in the C#. Without this application, the Omni Framework would not be possible due to how the C++ code can change over time.

Let us talk about configuring this monster!

#### Configuring C++ Class pInvoke Serializations

This configuration file is responsible for parsing the different types of C++ structures between C++ and C#.

So for example,

In the C++ source code we have a parameter type of "VectorF". To be able to pass a VectorF as a parameter or accept it as a return type from a C++ console function we need to provide how to translate it. So let's look at the configuration for VectorF.

```
'''
#ObjectType#=VectorF
#DeserializeString#={0} {1} = {0}();\r\nscanf(x_{1},\"%f %f %f\", &{1}.x, &{1}.y, &{1}.z);\r\n
///DeserializeString Generates---->Point3F size = Point3F(); sscanf(x__size,\"%f %f %f\", &size.x,&size.y,&size.z);
#SerializeString#=dSprintf(retval,1024,\"%f %f %f\",wle_returnObject.x,wle_returnObject.y,wle_returnObject.z);\r\nreturn;\r\n
```

The DeserializeString tells the Static Code generator how to convert a string to the C++ object.

So in when the extern is generated in the C++, it changes the parameter type to a const char \*, and inside the function it injects the deserializeString into it replacing the parameters with the variable name.

When a console method returns a VectorF, it replaces the return text with the serializeString to build the return value. So once again it replaces the placeholders with the variable name in the C++ code.

So, if you add a new object type to the engine which isn't SimObject based and you have a need to pass it as a parameter or return type in a console function you will need to define the object here.

#### Configuring C++ Class/Enum Map to C# Class/Enum

This configuration file is the second part to configuring C++ Class pInvoke Serializations. Once you have defined the serialization you need to tell the Static Code Analyzer what C# object maps to the C++ object.

So continuing our example of VectorF...

```
TransformF TransformF Class;
VectorF Point3F Class;
ColorI ColorI Class;
```

Here you can see we are mapping the Point3F C# class to the C++ VectorF class inside the Static Code Generator.

## Configuring C++ Class/Function Ignores

There are times when you want the Static Code Analyzer to ignore certain functions and or whole classes. They might be Console functions that you just don't need or you can't get to work correctly inside of the Static Code Generator.

## Configuring C++ Constants

This configuration file is used to define the C# equivalent to the C++ constant.

```
ColorF::ZERO      ##BEGIN##    new ColorF(0.0f, 0.0f, 0.0f,0.0f)##END##
ColorF::ONE       ##BEGIN##    new ColorF(1.0f, 1.0f, 1.0f,1.0f)##END##
ColorF::WHITE     ##BEGIN##    new ColorF(1.0f, 1.0f, 1.0f,1.0f)##END##
ColorF::BLACK     ##BEGIN##    new ColorF(1.0f, 1.0f, 1.0f,1.0f)##END##
ColorF::RED       ##BEGIN##    new ColorF(1.0f, 1.0f, 1.0f,1.0f)##END##
ColorF::GREEN     ##BEGIN##    new ColorF(1.0f, 1.0f, 1.0f,1.0f)##END##
ColorF::BLUE      ##BEGIN##    new ColorF(1.0f, 1.0f, 1.0f,1.0f)##END##

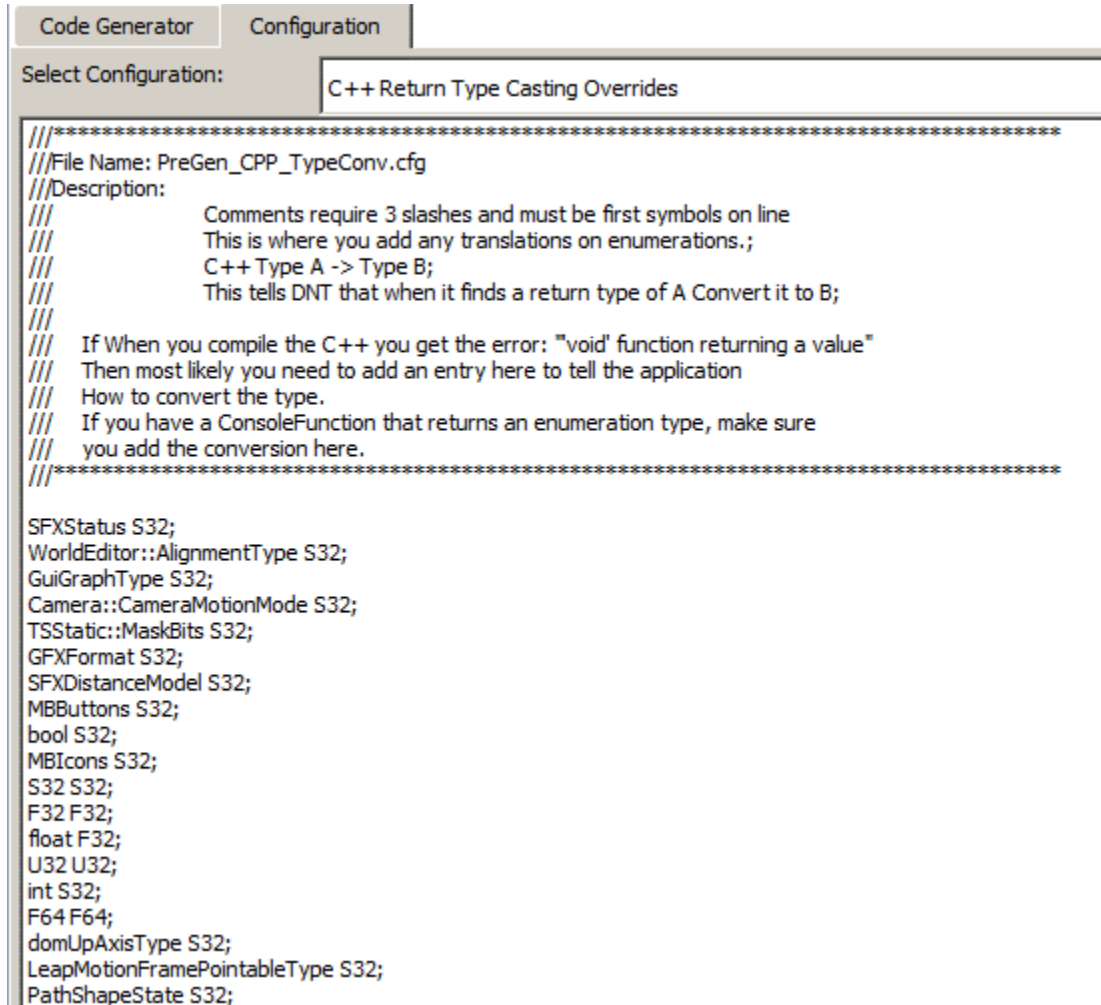
Point2I::Zero     ##BEGIN##    new Point2I(0,0)                ##END##
Point2I::One      ##BEGIN##    new Point2I(1,1)                ##END##
Point2I::Min      ##BEGIN##    new Point2I(int.MinValue,int.MinValue)    ##END##
Point2I::Max      ##BEGIN##    new Point2I(int.MaxValue,int.MaxValue)    ##END##
```

In the C++ code, you might have default parameters set up for the Console Function. The Static Code Generator will attempt to convert them to C#. Usually, if the constant is not defined in this file, the result will be code generated in error.

## Configuring C++ Return Type Casting Overrides

We need to provide a mapping between C++ and C# base object types. This is because the type conversion between C++ and C# are not always clear, and also since you might use a #Define in the C++ to redefine a base type. Usually this file is reserved for defining enumerations and types that are redefined with #Defines.

# O M N I E N G I N E

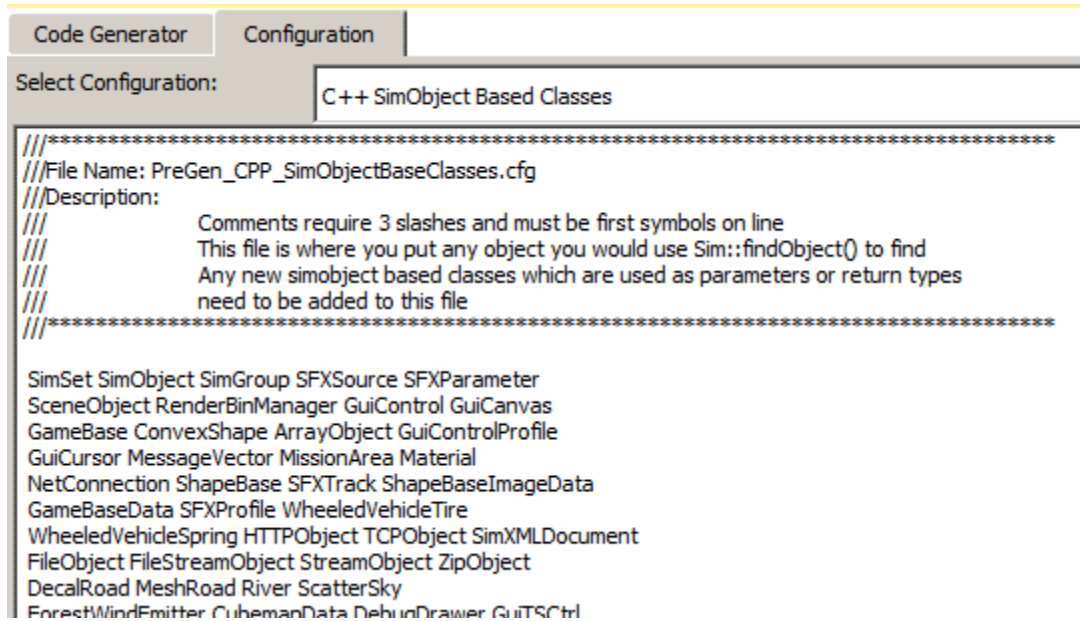


## Configuring C++ SimObject Based Classes

Whenever you add a new object type which is derived from the SimObject base class you need to add it to this file. Usually, the Static Code Generator will spew out an error stating that "The Class XXXX is not defined, if it is a SimObject based class please add it to the C++ Simobject Based Classes."



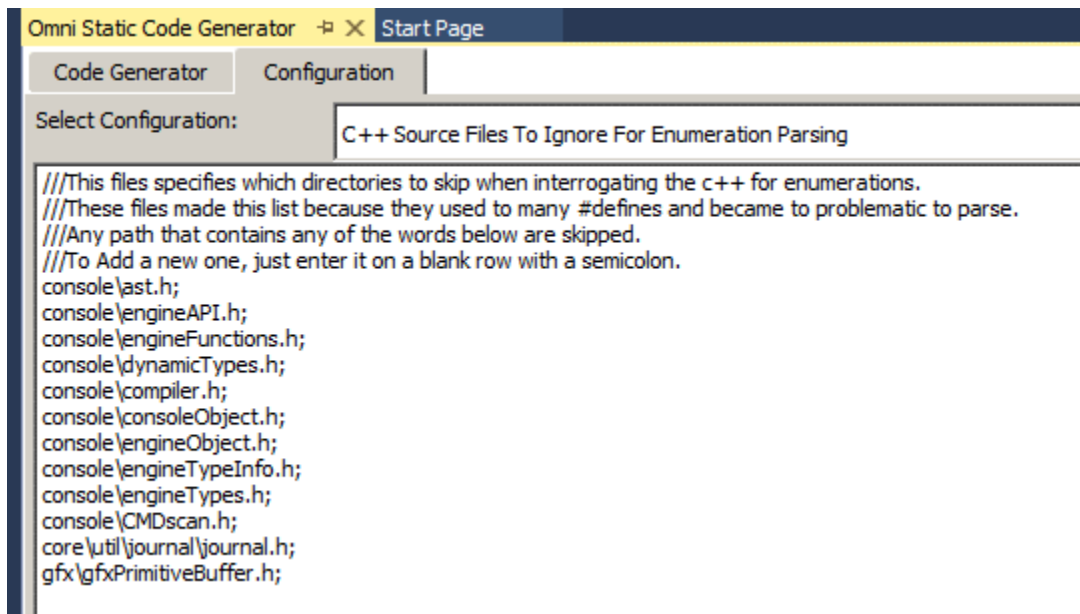
# O M N I E N G I N E



This configuration file is nothing more than a catch all, so you don't have to define a serialize/de-serialize for every SimObject based type.

## Configuring C++ Source Files To Ignore For Enumeration Parsing

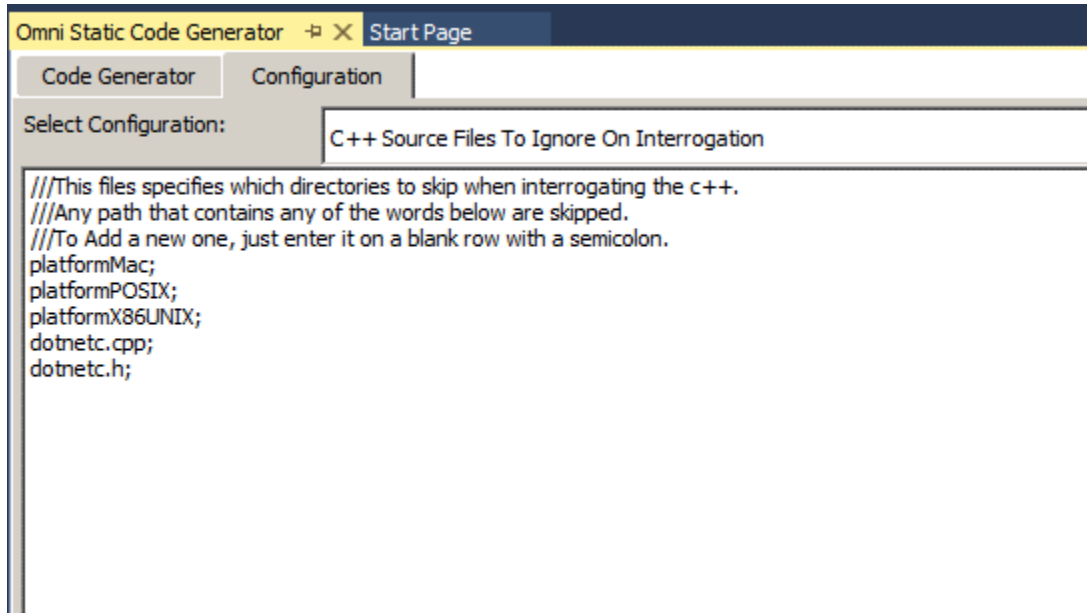
There are just some C++ classes where the enumerations contained in the C++ code serve no purpose to the interopt between C++ and C#. By putting the name of the file in this configuration file, the Static Code Analyzer will not parse those files for enumerations.



# O M N I E N G I N E

## Configuring C++ Source Files To Ignore On Interrogation

This configuration file specifies the files to completely ignore when parsing the source code. Any file listed in this configuration will not be processed. It is basically an exclusion list for the Static Code Generator.



## Appendix 2 - Special Omni C# Syntax

All the decorations used in DNT have been replaced with a single C# Decoration. The new decoration is “ConsoleInteraction”. The “ConsoleInteraction” decoration tells the Omni Framework to expose the function as a callable function from the C++ console. The decoration is not necessary for functions which are overridden base functions, in those cases it is implied.

There are two usages available for this decoration. Under normal circumstance the default decoration, “[ConsoleInteraction()]” will suffice. This flags the member function for use in the console.

If this decoration is used on a static function then that function is exposed to the console as a global function versus a member function.

An example of the difference is:

```
[ConsoleInteraction(true)]
public static void OnServerMessage(string message)...
```

Exposes the function OnServerMessage as a global function in the console. From the game console you could type “onServerMessage(“blah”);” and the engine would call the function. This is different than member functions which would have a signature like:

```
[ConsoleInteraction(true)]
public void Respawn()...
```

This function would only be exposed as a member function to an object. To invoke this object you would need to know the name or id of the object followed by a period and the function. To call this function you would have to type something like “MyObject.Respawn();”.

In regards to Global functions exposed to the console. There are times when in C# you are following a naming convention and you will end up with two static functions called the same thing. For example you have an “Initialize()” function that you want to expose to the console. Unfortunately, “Initialize” could be used as a name for a static function in more than one class. Thus causing a problem since Global functions must be unique.

In this case you would have to pass an alias to the ConsoleInteraction decoration like such:

# OMNI ENGINE

```
[ConsoleInteraction(true, "ConvexEditor_initialize")]  
public static void initialize()...
```

This tells the Omni Framework to Alias the global static function “Initialize” as “ConvexEditor\_Initialize”.

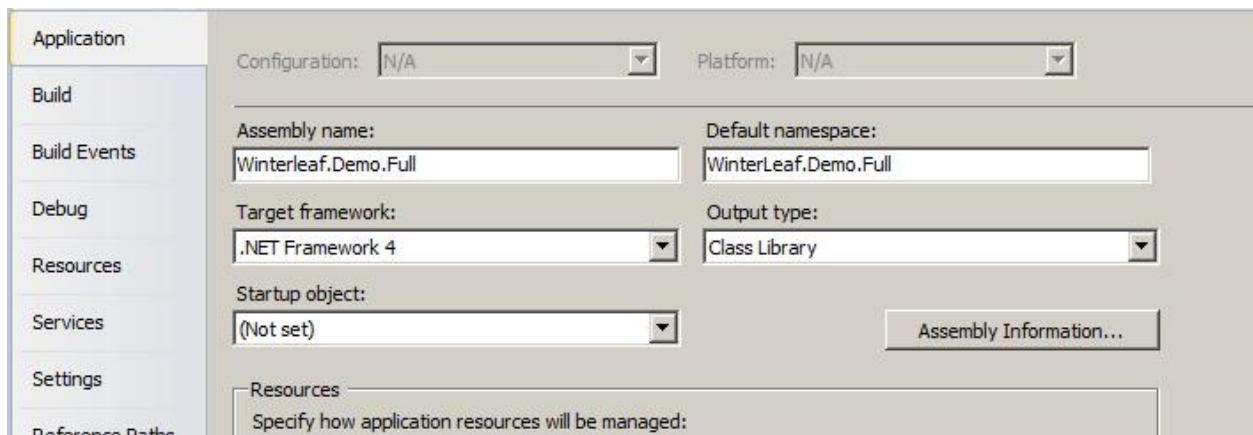
## NOTES:

Exposed Static Functions must be unique across all classes.

Functions are not case sensitive, so for the framework “OnNewItem” is the same as “onnewitem”. So even though C# function naming is type sensitive, having two functions with the same spelling will throw an exception that the function is already defined.

Namespaces are also very important inside the Omni Framework. You can use any namespace for you custom code. But you cannot mix namespace base names in your code. In the example provided in the FPS demo, you will see that the base namespace is “WinterLeaf.Demo.Full”. So pick a namespace and stay with it.

Also, once you have a namespace set it in the project properties for the C# library.



### Appendix 3 - Built in conversion functions for Casting.

Omni has built in conversion functions for converting strings into various other types. These conversion functions are written as C# Extensions and are defined in the Engine project.

When casting from a number to a string type use the “AsString()” function versus the C# stock function “ToString()”. “AsString()” handles precision and conversions for Booleans properly. Also when casting from strings to number types use the appropriate “As<Type>()” function. These functions were written to limit the expose to TryParse and Try/Catch blocks for performance reasons.

## Appendix 4 - Casting Game Objects

Time to time you will need to cast complex objects. C# allows you to cast a derived object to a base object with no problem. This is the base functionality of C#, for example:

```
namespace TestApp
{
    public class ClassA
    {
        public int a;
    }

    public class ClassB : ClassA
    {
        public int b;
    }

    public class ClassC : ClassB
    {
        public int c;
    }

    public class test
    {
        public void CastMeOk()
        {
            ClassC c = new ClassC();
            ClassA a = c; //Implicit cast, this is allowed
        }

        public void CastMeBad()
        {
            ClassA a = new ClassA();
            ClassC c = a; //Invalid cast, cannot upward cast an object.
        }
    }
}
```

This is a base foundation of C#. You cannot cast a base class to a derived class without writing something that does the dirty work for you.

This leaves a problem with programming with events like collisions. Think of the situation when we have a “HealthPatch”, a “Player” and a “AIPlayer”. Now, many objects will collide with the “HealthPatch”, but we only want to do something if it is of a type of “Player” or “AIPlayer”. How do we handle it because the signature for the inherited collision function is:

```
public override void onCollision(ShapeBase obj, SceneObject collObj, Point3F vec, float len)
```

This means that the function is passing a “ShapeBase” object and we can’t cast it up in C# to a player or AI.

This is overcome by the Proxy Object Cache. When a simulation object is created in the C++, a proxy object of the same type is also created for it in the C#. BUT, Omni saves the proxy C# class instance in a collection of “ModelBase” which is our base proxy object class definition type.

# OMNI ENGINE

So this means that all proxy objects are actually of our base class type and of our derived type at the same time. Because of this we are always casting objects down (Which is allowed) even though it may seem we are casting objects up.

This example of code is from the HealthPatch proxy object.

```
public override void onCollision(ShapeBase obj, SceneObject collObj, Point3F vec, float len)
{
    if (!((collObj.GetType() == typeof (Player)) || (collObj.GetType() == typeof (AIPlayer))))
        return;
    Player player = (Player) collObj;
    if (player.getDamageLevel() <= 0.00 || player.getState() == "Dead")
        return;
    .
    .
    .
}
```

So even though the parameter coming in is of a lower defined class type, we can cast it up to a Player object, since in reality it could be a Player, AIPlayer, or any other scene object type instance of an object.

The concept works like below.

```
public void CastMeOk()
{
    ClassA a = new ClassC();
    ClassC c = (ClassC)a; //Implicit cast, this is allowed
}
```

So with basic type checking, we can use objects that are defined farther down in the inheritance model at any point from there instance to the base object type of “ModelBase”.

# O M N I E N G I N E

## Appendix 5 - Overriding functions and such

The Omni Framework uses function overriding quite heavily. This is the process of deriving a base object and inheriting it and potentially replacing functionality of an inherited function.

More info is available at <http://msdn.microsoft.com/en-us/library/ms173149.aspx>.



## Appendix 6 - Creating Objects (ObjectCreator/SingletonCreator/DatablockCreator)

At some point you will need to create in game objects. This is done using the “ObjectCreator” helper class in the Omni Framework. The ObjectCreator class simplifies object creation.

An example used in the Winterleaf.Demo.Full project:

```
public static UInt32 Spawn(string name, TransformF spawnpoint)
{
    ObjectCreator npcScriptObject = new ObjectCreator("AIPlayer", "", typeof (DemoPlayer));
    npcScriptObject["dataBlock"] = "DemoPlayer";
    npcScriptObject["path"] = "";
    coAIPlayer npc = npcScriptObject.Create();
    .
    .
    .
    return npc;
}
```

The ObjectCreator uses a constructor that expects three parameters. They are:

1. Name of the C++ Simulation Object Class
2. Name Alias for the new instance
3. Optional, the Proxy Class to bind the C++ object to.

In this case we are creating an “AIPlayer” C++ object, we are not assigning a name to it, so it will be only available by SimObjectID and we are telling it to bind the C++ object instance to an instance of our “DemoPlayer” class. Had we omitted the last parameter it would have used the stock proxy class of “AIPlayer”.

The ObjectCreator exposes a Dictionary key/value pair that lets you send in initializing parameters. These parameters will be set when the C++ engine creates the instance of the object you requested.

In the above code we are setting a property “dataBlock” equal to “DemoPlayer”. This datablock is defined in the folder “Game/Art/Datablocks/AIPlayer.cs”.

The second parameter we are setting is the “Path” parameter. This is just a path defined in the mission that this AI will follow.

To create the object we just call the member function “Create” which will pass your creation request to the C++ and return the ID of the object back if it was successful or zero if it failed.

The ObjectCreator collection value type is variant. This means, any C# class type which provides a “toString()” function can be assigned as a property of the initialization of the object.

This bit of code demonstrates that the value fields truly can be anything serializable to a string.

```
ObjectCreator tch = new ObjectCreator(this["projectileType"]);
```

# OMNI ENGINE

```
tch["dataBlock"] = this["projectile"];
tch["initialVelocity"] = muzzleVelocity;
tch["initialPosition"] = obj.getMuzzlePoint(slot);
tch["sourceObject"] = obj;
tch["sourceSlot"] = slot;
tch["client"] = obj["client"];
tch["sourceClass"] = obj.getClassName();

Projectile projectile = tch.Create();
```

So in general, when creating objects you can assign `Point3F`'s, integers, floats, or even your own class's as values as long as they implement and override the `"ToString()"` function.

## Gui's and nested Objects

There are times when you will need to nest object creations. This usually occurs usually in the creation of Gui's. To accommodate this you can assign `ObjectCreator` instances as values to another `ObjectCreator`. Since the `ObjectCreator`, `SingletonCreator`, and `Datablock creator` all provide a `"ToString()"` function they can be used as parameters to themselves. When assigning an `ObjectCreator` as a property to another `ObjectCreator`, the property must be marked with a `'#'` sign. This tells the creator that it is a child creator property.

```
ObjectCreator oc_Newobject2 = new ObjectCreator("GuiBitmapBorderCtrl", "");
oc_Newobject2["canSave"] = "1";
oc_Newobject2["canSaveDynamicFields"] = "0";
oc_Newobject109["#Newobject2"] = oc_Newobject2;
```

## Appendix 7 - uGlobal, sGlobal, iGlobal, bGlobal, fGlobal, dGlobal, fGlobal

These properties provide access to the global console variables defined inside of the C++ engine. Since all variables inside the C++ are of type string, these functions provide convenient casting to commonly used types in C#.

- uGlobal – Unsigned Integer
- sGlobal – String
- iGlobal – Integer
- bGlobal – Boolean
- fGlobal - Float
- dGlobal – Decimal
- fGlobal – Float

Global console global variables are denoted with a “\$”.

# OMNI ENGINE

## Appendix 8 – OMG I have a ton of “WARNING:: COULD NOT RESOLVE... in my console”

A sample console would look like the image below.

```
Executing core/art/skies/blank/materials.cs.
UDP initialized on port 0
WARNING:: COULD NOT RESOLVE 'BlankGui' to an object.
GuiCanvas::setContent - Invalid control specified')
WARNING: (art/gui/omni) texture size = 439x474
WARNING: (art/gui/background_g) texture size = 800x600
% - Initializing Tools
Missing file: tools/gui/guiDialogs.ed.cs!
WARNING:: COULD NOT RESOLVE 'materialEd_previewMaterial' to an object.
WARNING:: COULD NOT RESOLVE 'notDirtyMaterial' to an object.
WARNING:: COULD NOT RESOLVE 'materialEd_cubemapEd_cubeMapPreview' to an object.
WARNING:: COULD NOT RESOLVE 'matEdCubeMapPreviewMat' to an object.
WARNING:: COULD NOT RESOLVE 'materialEd_justAlphaMaterial' to an object.
WARNING:: COULD NOT RESOLVE 'materialEd_justAlphaShader' to an object.
WARNING:: COULD NOT RESOLVE 'EditorInspectorBaseDatablockFieldPopup' to an object.
WARNING:: COULD NOT RESOLVE 'EditorInspectorBaseFieldPopup' to an object.
WARNING:: COULD NOT RESOLVE 'EditorInspectorBaseFileFieldPopup' to an object.
WARNING:: COULD NOT RESOLVE 'EditorInspectorBaseShapeFieldPopup' to an object.
WARNING:: COULD NOT RESOLVE 'EditorInspectorBaseProfileFieldPopup' to an object.
% - Initializing Tools Base
WARNING:: COULD NOT RESOLVE 'EditorClassesCleanup' to an object.
WARNING:: COULD NOT RESOLVE 'EditorClassesCleanup' to an object.
% - Initializing Base Editor
% - Initializing World Editor
WARNING: (tools/worldEditor/images/LockedHandle) texture size = 10x12
% - Initializing Sketch Tool
% Initializing Datablock Editor
% - Initializing Debugger
% - Initializing Decal Editor
WARNING:: COULD NOT RESOLVE 'DecalPreviewWindow' to an object.
WARNING:: COULD NOT RESOLVE 'DecalEditorLibraryProperties' to an object.
WARNING:: COULD NOT RESOLVE 'DecalEditorTemplateProperties' to an object.
WARNING:: COULD NOT RESOLVE 'RetargetDecalButton' to an object.
WARNING:: COULD NOT RESOLVE 'SaveDecalsButton' to an object.
WARNING:: COULD NOT RESOLVE 'NewDecalButton' to an object.
WARNING:: COULD NOT RESOLVE 'DeleteDecalButton' to an object.
WARNING:: COULD NOT RESOLVE 'DecalPreviewWindow' to an object.
WARNING:: COULD NOT RESOLVE 'DecalEditorLibraryProperties' to an object.
WARNING:: COULD NOT RESOLVE 'DecalEditorTemplateProperties' to an object.
WARNING:: COULD NOT RESOLVE 'RetargetDecalButton' to an object.
WARNING:: COULD NOT RESOLVE 'SaveDecalsButton' to an object.
WARNING:: COULD NOT RESOLVE 'NewDecalButton' to an object.
WARNING:: COULD NOT RESOLVE 'DeleteDecalButton' to an object.
WARNING: (tools/editorclasses/gui/images/rollout) texture size = 39x50
% - Initializing Forest Editor
WARNING:: COULD NOT RESOLVE 'ForestEditorInspector' to an object.
WARNING:: COULD NOT RESOLVE 'ForestEditBrushTree' to an object.
WARNING:: COULD NOT RESOLVE 'ForestEditMeshTree' to an object.
WARNING:: COULD NOT RESOLVE 'ForestEditorInspector' to an object.
WARNING:: COULD NOT RESOLVE 'ForestEditMeshTree' to an object.
% - Initializing In-game GUI Editor
% - Initializing Material Editor
% - Initializing Mesh Road Editor
% - Initializing Mission Area Editor
% - Initializing Particle Editor
WARNING:: COULD NOT RESOLVE 'ParticleEditor' to an object.
WARNING:: COULD NOT RESOLVE 'ParticleEditor' to an object.
WARNING:: COULD NOT RESOLVE 'ParticleEditor' to an object.
```

This is normal in debug mode and remember it's just a warning. Basically there is a script trying to reference an object by name and that object doesn't exist inside of T3D yet. If the messages continue after the initial startup of the game then you might want to research further to make sure you didn't misspell the object.

## Appendix 9 - Where did “ClassNameSpace” and “superclass” go?

They still exist but they are no longer needed. In T3D inheritance was limited to 2 layers. Instead of using ClassNameSpace and SuperClass, one should derive the classes in the appropriate order. So the superclass would be the base, which the ClassNamespaces would inherit from, which your class in turn then inherits from.

Unlike TorqueScript, you have unlimited inheritance in OMNI due to C#. There is nothing preventing you from taking a base object and inheriting it over and over again to define subtle changes in behavior.

A note of warning, do not mix TorqueScript syntax w/ C#. The console does not handle switching and gets confused when both are used. If you are using Omni, it is highly recommended that you dispose of all your “ClassNameSpace” and “Superclass” structures and convert them to inheritance.