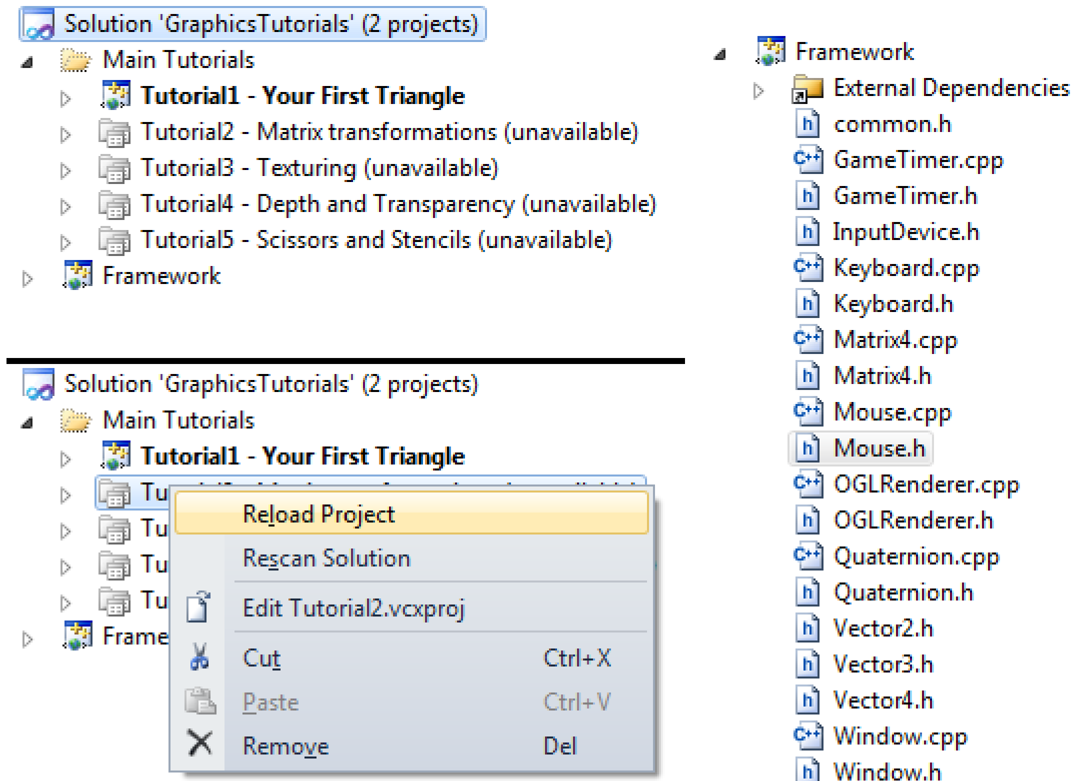
# Introduction to the Graphics Module Framework

## Introduction

Unlike the C++ module, which required nothing beyond what you typed in, the graphics module examples rely on lots of extra files - shaders, textures, and mesh files. You also need to perform quite a lot of vector and matrix manipulation, which if coded incorrectly could cause very undesirable rendering results! To this end, for this module you're going to be provided with a small 'framework', made up of a Visual Studio solution, and some data files. Every week, you'll get a new framework download, which if you drop into your module folder, will add the new functionality and tutorial folders required for that week.

Upon opening the solution file for the first week's tutorials, you should be greeted with something like the following:

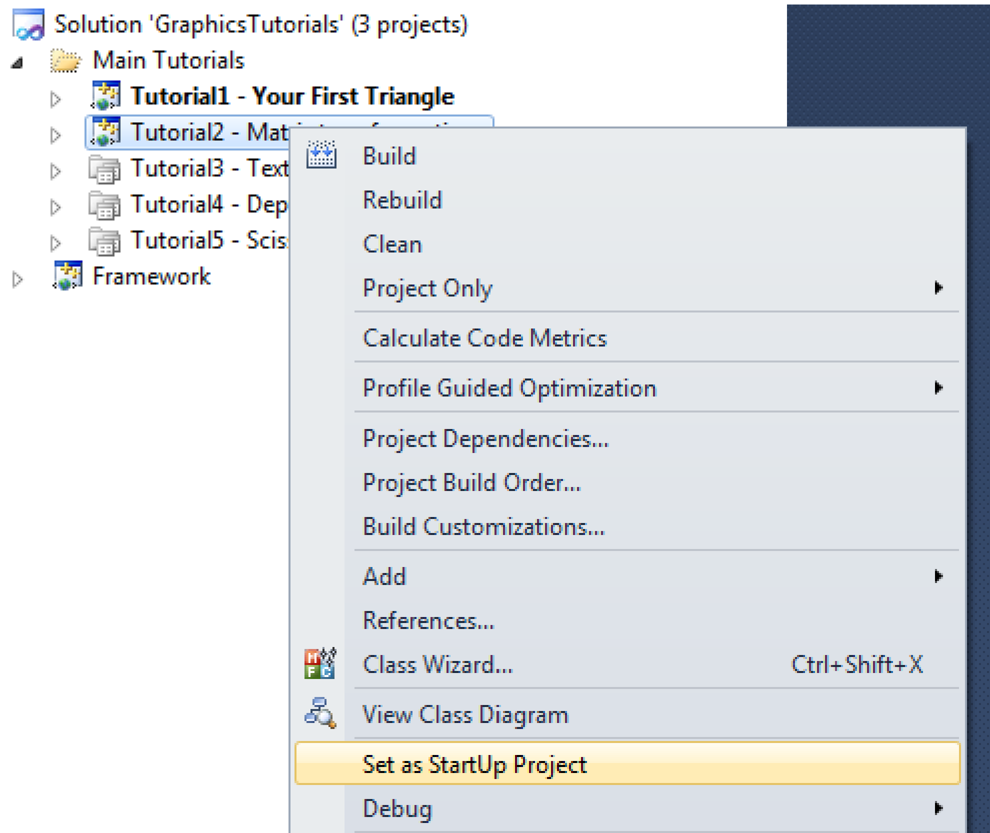



*Top Left: The contents of the solution file Bottom Left: Loading in an unloaded project Right: The files provided by the Framework static library.*

You should have 6 individual projects - 5 called *Tutorial* and numbered from 1 to 5, and one called *Framework*. The tutorial projects don't have much in them - as their names imply, you're going to be filling these with code as you go along.

The *Framework* project is a little bit different, though. When compiled, the Framework project won't create an executable, but a **static library**. This is a big chunk of executable code, that can be **linked** into a project to add extra functionality and classes. Each of the tutorials relies on this static library to help create an OpenGL context, and to provide math functionality such as *Vector* and *Matrix* classes.

Each tutorial has its own project folder, but to start off with, only the project for Tutorial 1 is active. To activate further tutorials, you must right click on the project, and select *Reload Project* - this'll load in the project, ready for compiling and running. Then, to select which tutorial to execute when pressing F5, right click the desired project, and click on *Set as StartUp Project*.




*Setting which executable to run*

## Maths Classes

Provided for you are some simple classes that encapsulate the mathematical constructs you'll need in order to manipulate the graphics data you want to draw on screen. Each of them **supports operator overloading** to make their manipulation easier, including the **stream** operator, which will allow you to print their values to the console. Common to all of these classes is that their member variables are **public**, and so can be freely manipulated - this is to keep the code generated from these classes optimised - it's more trouble than its worth to try and hide data in purely numeric classes such as these.

Although each of these classes provides everything you need for this module, they're not perfect - constructs like vectors and matrices are ideal for optimising using SIMD extensions of modern CPUs, which these classes do not do. As you experiment beyond the tutorial codes, you'll probably also find there are operator overloads and some other operations missing - try adding them in!

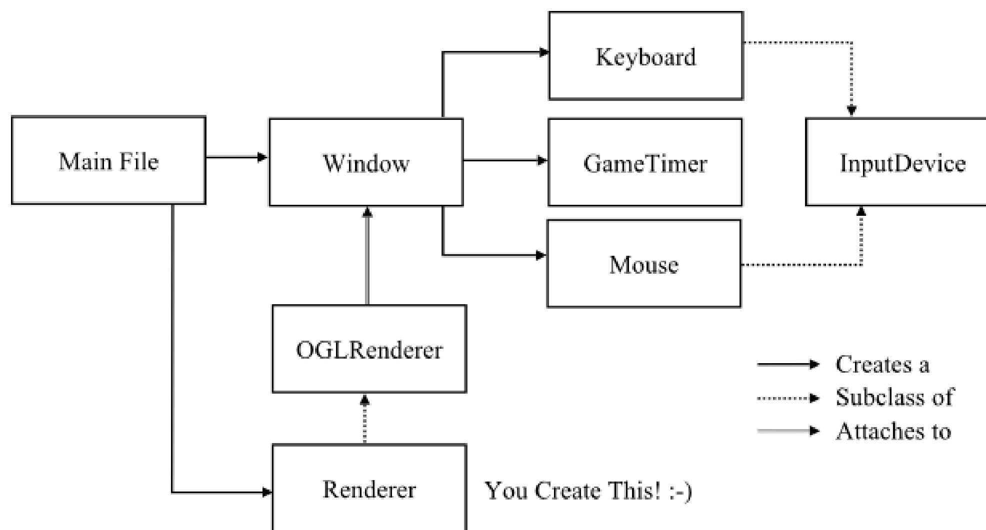
The **Matrix4** class provides you with the capability of manipulating 4 by 4 square matrices - in the second full tutorial you'll see how these are used to position and rotate the objects you wish to render on screen.

The **Vector2**, **Vector3**, and **Vector4** classes deal with algebraic *vectors*, stored as floats. These are useful for storing things like texture coordinates, positions in space, and colours - as well as providing linear algebra functions such as the *dot product*, the *cross product*, and *normalisation* functionality.

Finally, the **Quaternion** class adds the ability to manipulate quaternions to your projects. Quaternions are like 4 component vectors, but the values are combined in such a way as to encode an axis and a rotation.

## Class Hierarchy

Along with the math classes, the framework also provides classes to enable you to create a Window, read from the Keyboard and Mouse, and calculate how much time has elapsed. In each tutorial, you will a *main.cpp* file, within which you will have your **main** function. From here, a Window class is initialised - this encapsulates all of the code required to get the Windows operating system to create a **Window** suitable for rendering into. The creation of the Window class will create a **Keyboard** and **Mouse** - these inherit from a common **InputDevice** class, and allow you to check which keys and buttons have been pressed down, and see how much the mouse has moved. You'll also find you have access to a **GameTimer**, which allows you to see time how many milliseconds have passed since the window was created. Finally, in each tutorial you'll make a **Renderer** class, which includes an example of how to implement the new functionality you learn about in each tutorial. The Renderer class inherits from the **OGLRenderer** class, which creates the actual OpenGL context you need to render, as well as provide some other simple rendering-related functions.




## Data files

Shaders, Textures, and vertex data, are kept in the *../Shaders/*, *../Textures/*, and *../Meshes/* folders of the framework, respectively. Although you'll be writing all of the shaders for this module yourself, you do get some textures, and some geometry, stored in the OBJ and MD5Mesh formats.

## Additional Libraries

### Simple OpenGL Image Library

http://www.lonesock.net/soil.html

In this tutorial series, you'll build up C++ *classes* that will handle the loading of vertex and shader data, as doing so will teach you a lot about the graphics pipeline. Textures though, are pretty much just a big chunk of data, generally 24 or 32 bits per pixel, depending on whether there's an alpha channel or not. On top of that, some graphics file formats are *compressed* - TGA files are compressed in an easily understood format known as *Run Length Encoding*, but something like a JPEG file uses a much more complex compression format. We could cover texture loading in this tutorial series, but you'd not actually learn a lot about graphics rendering in doing so. Instead, we're going to use a small library called **SOIL** - the *Simple Opengl Image Library.* This small library supports loading in (and

saving out!) a variety of common texture formats, taking away much of the pain of getting texture data into OpenGL. SOIL lives in the *../SOIL/* folder of the framework download, and is added to the framework solution via an include, and a pragma to link the SOIL static library.

```
1 #include "SOIL.h"
2 #pragma comment(lib, "SOIL.lib")
```

OGLRenderer.h

## GL Extension Wranger

http://glew.sourceforge.net/

Since OpenGL was originally created, its functionality has been increased greatly. This has primarily been through the use of *extensions* - the mechanism that allows an OpenGL implementation to support more features than the standard core of OpenGL provides. Over time, many of these extensions have become part of the OpenGL core functionality - for example, Vertex Buffers started as an extension, and are required to fully implement the OpenGL 3 spec. Unfortunately, these exstensions are not available by default on the Windows operating system due to the way the OpenGL API is exposed. Fortunately, Windows does provide a mechanism for getting a *function pointer* to a specific extension, via the **wgl** (often pronounced 'Wiggle') functionality. So, we could get a function pointer to the *SwapInterval* extension like so:

```
1 PFNWGLSWAPINTERVALEXTPROC            SwapInterval;
2 SwapInterval = (PFNWGLSWAPINTERVALEXTPROC)
3                  wglGetProcAddress("wglSwapIntervalEXT");
```

WGL example

Doesn't look very nice, does it? Now imagine trying to do that for 10s of extensions! Fortunately, there's a library that'll automatically take care of all of the nasty extension business, called the *GL Extension Wrangler*, or **GLEW** for short. The files for GLEW are included in the framework download, in the *../GLEW/* folder, and automatically added to the framework solution via an include and a pragma.

## AMD gDEBugger

http://developer.amd.com/tools/gDEBugger/Pages/default.aspx

AMD's gDEBugger plugin for Visual Studio 2010 allows you to debug the OpenGL graphics pipeline, and inspect what is currently loaded into graphics memory - handy for tracking down weird errors! gDEBugger allows the use of breakpoints, but unlike when debugging your C++ code, you cannot set graphical breakpoints just anywhere. What you *can* do though, is set gDEBugger to break when a specific OpenGL command is added to the command queue. To aid in debugging, the framework defines a **macro**, *GL_BREAKPOINT*, that wraps around a little-used OpenGL command, allowing you to set graphical breakpoints like so:

```
1    triangle->Draw();
2    glUseProgram(0);
3
4    GL_BREAKPOINT //Application will break before SwapBuffers is called
5    SwapBuffers();
```

gDEBugger breakpoint example

# Using the Window Class

The *Window* class encapsulates all of the code required to create a Microsoft Windows window, suitable for rendering into. Multiple *Windows* can be opened in a single application, each with their own *Renderer*. The first Window initialised will create a *Mouse*, *Keyboard* - each application has only one Mouse and Keyboard instance. Each Window also has its own *GameTimer* class instance.

## Functions

**Window(string title, int sizeX, int sizeY, bool fullScreen):** The constructor takes in 4 parameters - the first is a **string**, denoting the identifying text that will be displayed at the top of the window. The next two parameters are the width and height of the window - note that this includes the window surround and top bar, so the actual rendering area will be slightly smaller than this. The final parameter is a **bool**, and determines whether the window will be fullscreen or not. If so, it will attempt to create a fullscreen window - in such cases, *sizeX* and *sizeY* should match a resolution supported by the display device, or the application is likely to fail.

**UpdateWindow():** In order to update the enhanced input device functionality and check internal windows messages, each *Window* should have *UpdateWindow* called once per frame. *UpdateWindow* will return **true** if the close window widget has been pressed, or **false** otherwise.

**SetRenderer(OGLRenderer &r):**For a *Window* to output anything useful, it must have an *OGLRenderer*, or a subclass of it, attached. *SetRenderer* will attach the given *OGLRenderer*, and if necessary, resize the output of the *OGLRenderer* to match the Window dimensions.

**HasInitialised():** When a *Window* is constructed, it is possible for its basic initialisation and OS integration to fail, leaving it unable to display. *HasInitialised*, if it returns **false**, denotes that this initialisation has failed, and the program should exit.

**LockMouseToWindow(bool lock):** If *LockMouseToWindow* is called with a boolean argument of **true**, the mouse pointer (whether visible or not) will be locked to the window area - useful if you intend to have 1st person camera rotation in a windowed application. A value of **false** will free the mouse pointer.

**ShowOSPointer(bool show):**An argument of **false** sent to this function will 'hide' the windows pointer when it is within the current *Window* dimensions. A value of **true** will show the mouse pointer.

**GetScreenSize():** Returns a *Vector2*, containing the current *Window* dimensions.

**GetKeyboard():** Returns a pointer to the **static** *Keyboard* object.

**GetMouse():**Returns a pointer to the **static** *Mouse* object.

**GetTimer():**Returns a pointer to the *GameTimer* object of this *Window*.

# Using the GameTimer Class

The *GameTimer* class encapsulates the high precision timer of the operating system. Multiple *GameTimer* classes can be instantiated to keep track of multiple times.

## Functions

**GetMS():** Returns a **float** value containing the number of milliseconds that have passed since the *GameTimer* was instantiated.

**GetTimedMS():** Returns a **float** value containing the number of milliseconds since the last call to *GetTimedMS*. The function is automatically triggered on class instantiation, making it always safe to call to get a correct value.

# Using the Mouse Class

The *Mouse* class encapsulates all of the Windows mouse functionality provided via the RAW API. It can handle mice of up to 5 buttons, and supports mouse wheel movement. This class also features various game specific features, like adjustable sensitivity, double click detection, and multiple frame mouse button holds.

## Functions

**ButtonDown(MouseButton Button):** Takes a single argument, of the *MouseButton* **enumerator** type, and returns whether the mouse button was pressed in the previous update.

**ButtonHeld(MouseButtons button):** Takes a single argument, of the *MouseButton* **enumerator** type, and returns whether the mouse button was pressed in multiple consecutive frames.

**DoubleClicked(MouseButtons button):** Takes a single argument, of the *MouseButton* **enumerator** type, and returns whether the mouse button was double clicked in the previous frame.

**GetRelativePosition():** Returns a *Vector2*, containing how much the mouse has moved since the last update.

**GetAbsolutePosition():** Returns a *Vector2*, containing the mouse pointer position, in screen coordinates.

**SetDoubleClickLimit(float msec):**Sets how much time (in milliseconds) can pass between mouse clicks while still counting as a double click. Higher values will create 'false' double clicks from fast 'single clicks', while values too low will make it difficult to trigger a double click.

**WheelMoved():** This function returns **true** if the mousewheel has been moved in the last update, **false** otherwise.

**GetWheelMovement():** Returns a positive **integer** if the mousewheel has been moved upwards, or a negative **integer** if it has been moved downwards.

**SetMouseSensitivity(float amount):** The values returned by *GetRelativePosition* are multiplied by this **float** value, scaling mouse movement per frame. So lower values will equate to more mouse movement required to rotate the camera, and vice versa.

## Enums

The *Mouse* class relies upon an **enumerator** to equate integers to physical mouse buttons, called **MouseButton**. It can have the following values: *MOUSE_LEFT, MOUSE_RIGHT, MOUSE_MIDDLE, MOUSE_FOUR, MOUSE_FIVE.*

# Using the Keyboard Class

The Keyboard class encapsulates all of the Windows keyboard functionality provided via the RAW API. It's pretty simple, but can detect keys held across multiple updates.

## Functions

**KeyDown(KeyboardKeys key):** Takes a single argument, of the *KeyboardKeys* **enumerator** type, and returns whether the key was pressed in the previous update.

**KeyHeld(KeyboardKeys key):** Takes a single argument, of the *KeyboardKeys* **enumerator** type, and returns whether the key was pressed in multiple consecutive frames.

**KeyTriggered(KeyboardKeys key):** Takes a single argument, of the *KeyboardKeys* **enumerator** type, and returns whether the key was pressed in the latest update, but was **not** held prior to that. Useful for detecting the first frame in which a key has been pressed for events that should be triggered only once per press, like throwing a grenade etc.

### Enums

The *Keyboard* class relies upon an **enumerator** to equate integers to physical keyboard keys, called **KeyboardKeys**. It can have the following values: *KEYBOARD_0 - KEYBOARD_9, KEYBOARD_A - KEYBOARD_Z, KEYBOARD_NUMPAD0 - KEYBOARD_NUMPAD9, KEYBOARD_F1 - KEYBOARD_F10, KEYBOARD_LSHIFT, KEYBOARD_RSHIFT, KEYBOARD_LCONTROL, KEYBOARD_RCONTROL, KEYBOARD_SPACE.*

## Credits

The following files are from id Software's Doom3 title -
*../Meshes/HellKnight.md5Mesh*
*../Meshes/idle2.md5Anim*
*../Textures/gob_h.tga*
*../Textures/gob2_h.tga*
*../Textures/hellknight.tga*
*../Textures/hellknight_local.tga*
*../Textures/tongue.tga*
*../Textures/tongue_local.tga*

The 'rusted' Cube Map textures in the *../Meshes/* folder come courtesy of *http://www.katsbits.com*