Handmade

An Educational Game Engine

Manual

Contents

Introduction

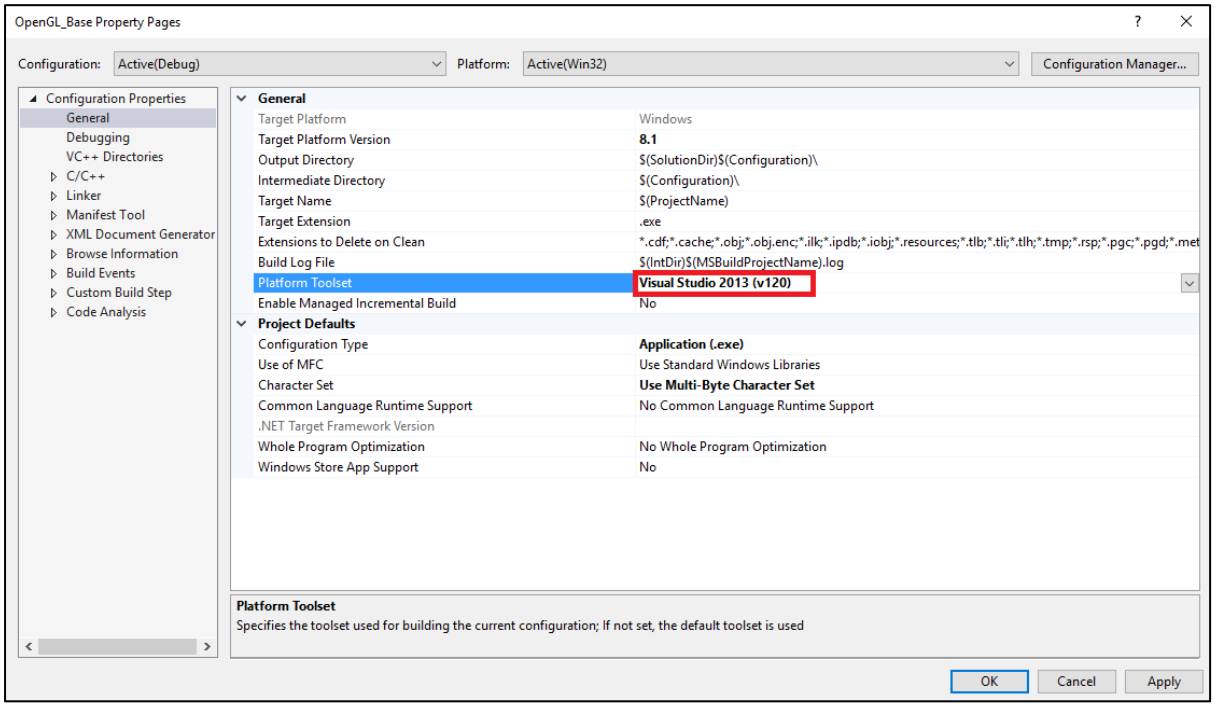
There are plenty of game engines out there today, and game developers and general all round programmers do not need to be bombarded with yet another offering. Especially since so many of the current engines are free to use, to a certain degree, and are robust enough to handle almost anything. So why make another one?
Handmade actually started out as a pet project / college assignment that evolved into something larger; almost like a few lines of code that soon became a library of tools, big enough to make a small platformer game. Of course the intention when creating this tool was never ever to compete with the likes of *Unity, Unreal, CryEngine* or the newcomer *Lumberyard*; it was really just a small assignment to build a set of tools from scratch and perhaps create some small games using those tools. It was meant to be a learning device, a means of grasping the fundamentals of *C++, OOP, maths, physics, networking, shaders, graphics programming, audio programming*, and more. In the end these exact elements are what the engine components consist of. Moreover, the engine is designed to help other fellow programmers and game developers alike learn these skills, by delving straight into the low / mid-level code the framework is built upon. *Handmade* is a pre-built small game engine that acts as a wrapper around the low level commands of *OpenGL* for rendering and shader functionality, *SDL* for input, time and game window management, *FMOD* for audio capabilities, alongside other components. But still the burning question exists - why bother at all?
The great thing about the game engines of today is that they allow the game developer to jump straight into the development of a game and within next to no time at all a pretty decent product is made, without worrying too much about what goes on in the background. No need to stress about the low-level things, because the engines only let you deal with the high-level concepts. But what if you wanted to get involved in the low-level code, what if you actually wanted to write a game from scratch, or delve into the inner workings of a matrix class or truly understand how virtual functions work? What then? Sure you could dive straight into *C++,* download a bunch of APIs like *SDL, SFML, GLFW*, use your *OpenGL* drivers and an audio tool and eventually after many days and nights of hard coding, you will have a working application capable of rendering some pretty pictures on screen and playing some great soundtracks. You can most certainly take the long way around just like I did, and it's definately the best way to learn and improve your programming skills, but you

could also meet half way and make use of a tool that prevents you from working too much in the dark; a tool that gives you a certain head start, but doesn't quite offer you all the high level game development functionality that *Unity* or *Unreal* will.
Using this engine you get to work alongside some low-level code and concepts and build a small game along the way. And because nothing is hidden from the programmer, you are exposed to everything, from the game window initialization code to the program flow from when a game state begins to where it ends. The engine has been designed and built for educational purposes, and is intended to be used by fellow programmers to get small, medium or large projects up and running rather rapidly, in order to learn the art of *C++, object oriented programming* and general *game design principles*; to serve as a tool for grasping low level programming and making games from scratch without building upon too many high level concepts. Anyone wanting to follow a certain programming path can easily do so, as the many compartments of the engine have been set up ready for use.
For example, if you wish to go straight into shader programming, by using *OpenGL's GLSL* shader language, then write a few shaders and attach, compile and run them. Networkers can use the engine to write basic network games using the underlying *SDL_Net* framework. Audio programmers can also make use of this software by working alongside the built-in audio functionality that uses *FMOD*. If you only want to learn how *OpenGL* works, you are welcome to read through all the written code and see how things work in the background - there is a little bit of everything for everyone.
The engine is by no means complete and is constantly being improved upon. It exists as a *Visual Studio* project and the source code is free to use, edit and share amongst anyone wishing to delve into raw game making. With love and absolute passion for programming, from one programmer to another, I give you *Handmade*. Made from scratch. Built by hand.

Getting Started

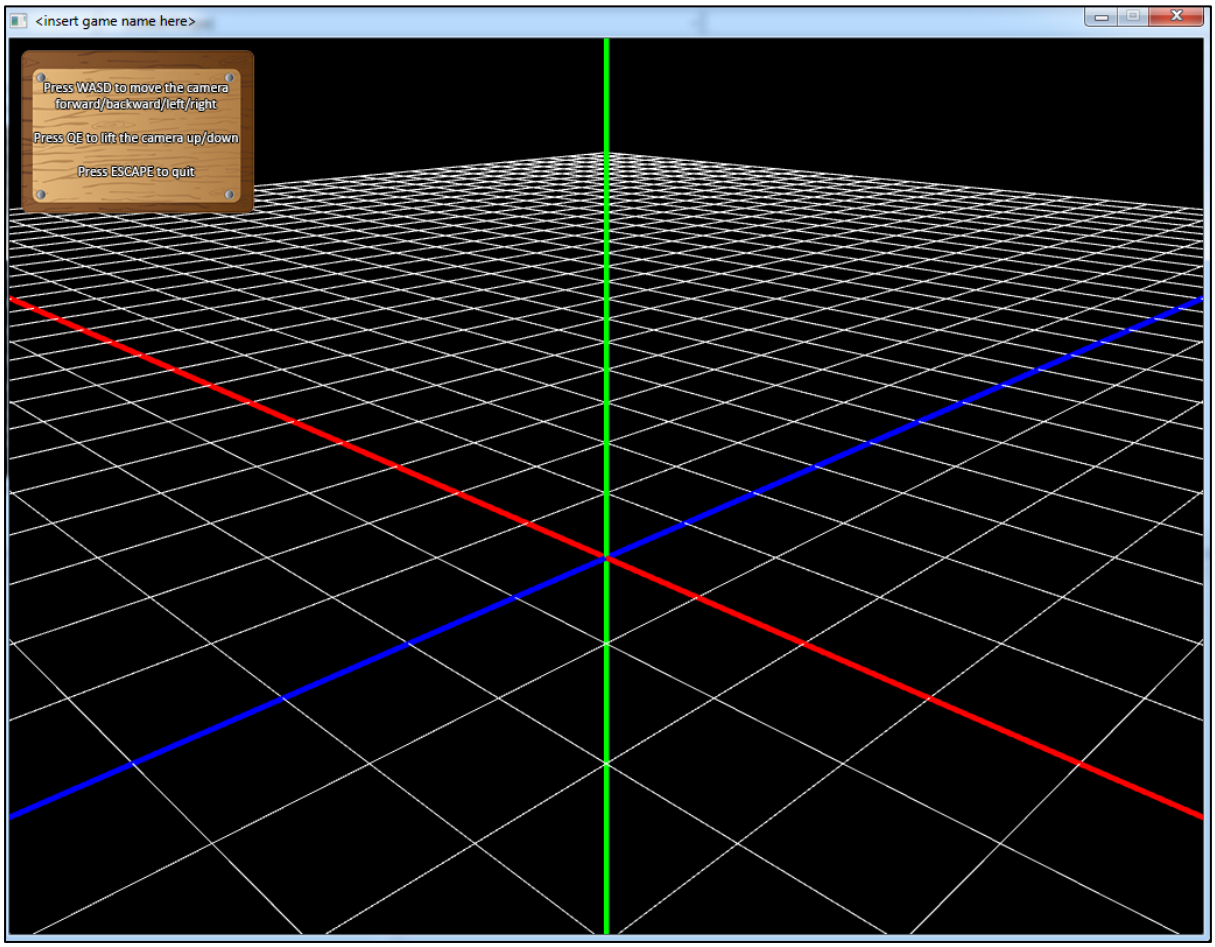
The first thing you will need is the *Visual Studio IDE* in order to open, build and run the project. You can get it free over here : https://www.visualstudio.com/

Because *OpenGL Base* was last edited in *Visual Studio 2015,* the project will by default compile and run perfectly from inside that IDE. However if you have *Visual Studio 2013* you can still run the project; the only thing you need to change is the *Platform Toolset* inside the *General Configuration Properties* of the project's *Property Pages*. Select the ***Visual Studio 2013 (v120)*** option. Now when you build and run the project, it will make sure the 2013 compiler is used.



Note : To access the Property Pages right-click the project and select Properties.

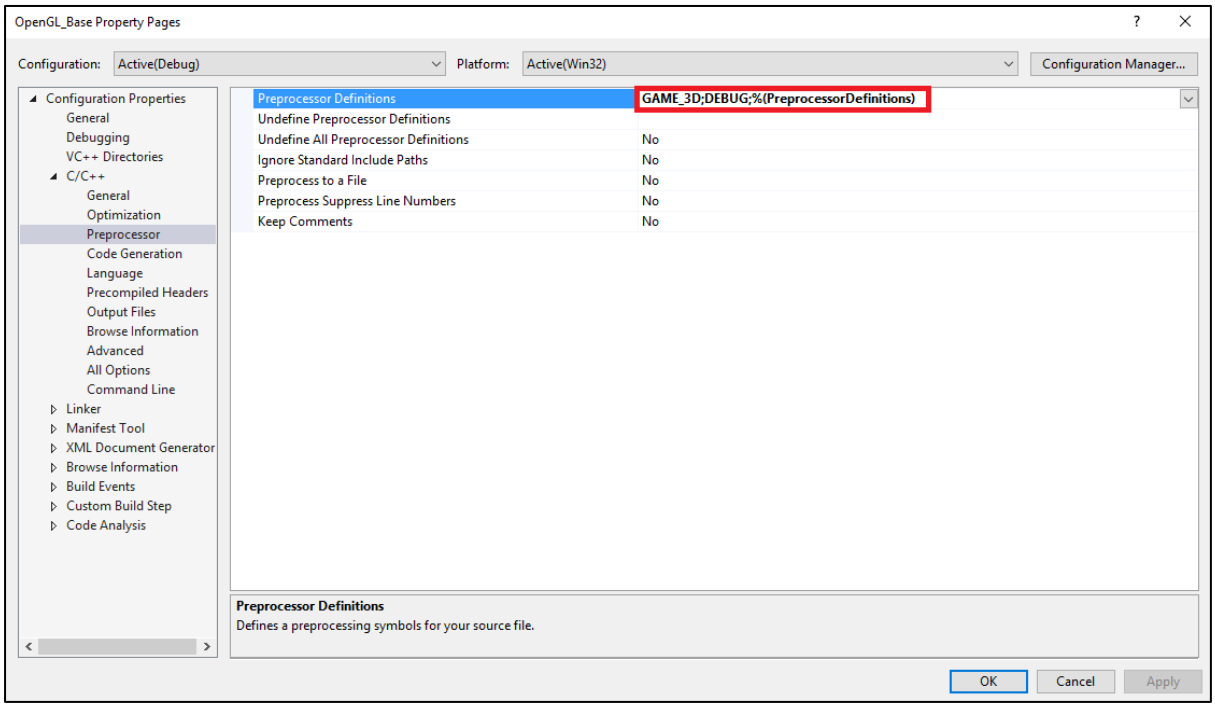
Once you have downloaded the engine, and set up your IDE, you can load up the solution file and build/run the project straight away. The engine is set up to run in both *Debug* and *Release* mode. The only difference is that in *Debug* mode a console window runs on the side displaying feedback for any errors that might occur when compiling shaders or loading resource files for instance. The very first thing you are presented with is a game window and a grid. How exciting!
Control instructions are displayed on the HUD screen in the top left corner.



Note : The grid will only draw in Debug mode, however that can be changed in the code itself, so don't panic when you don't see the grid in Release mode.

This game engine wraps the underlying *SDL, OpenGL* and *FMOD* API code so that you only have to deal with writing the game play or tech demo code. You are of course welcome to delve into the areas where the API function calls are made as they are all openly visible for all to peruse. The actual APIs are linked statically and their runtime DLL files and development headers are all shipped with the engine project. This makes the engine completely self-contained and should not present any compilation or runtime errors in this respect.

OpenGL Base runs in both 2D mode as well as 3D mode. Both modes have a grid associated with them, and to change modes, simply head over to the project's *Property Pages* and change the settings to **GAME_3D** for a perspective view or to **GAME_2D** for an orthographic view, in the *C++ Preprocessor Definitions* section.



Note : The grid represents a 2D or 3D coordinate system, just as you would see in any 3D application or game engine. The X-axis is red, the Y-axis is green and the Z-axis is blue.

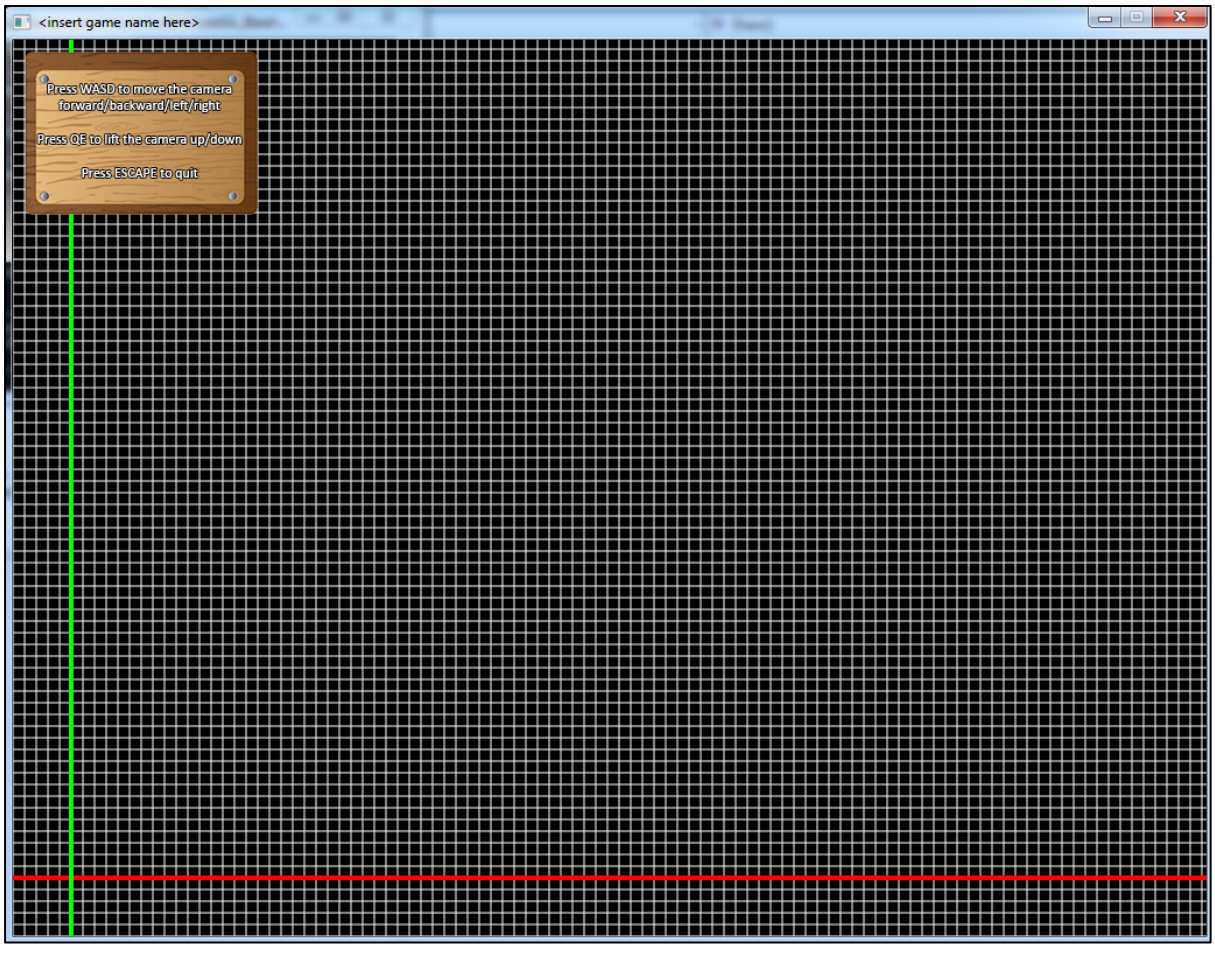
The Main Source File

Everything starts up from the *Main.cpp* file. You can set the window dimensions and name at the top of the source file.

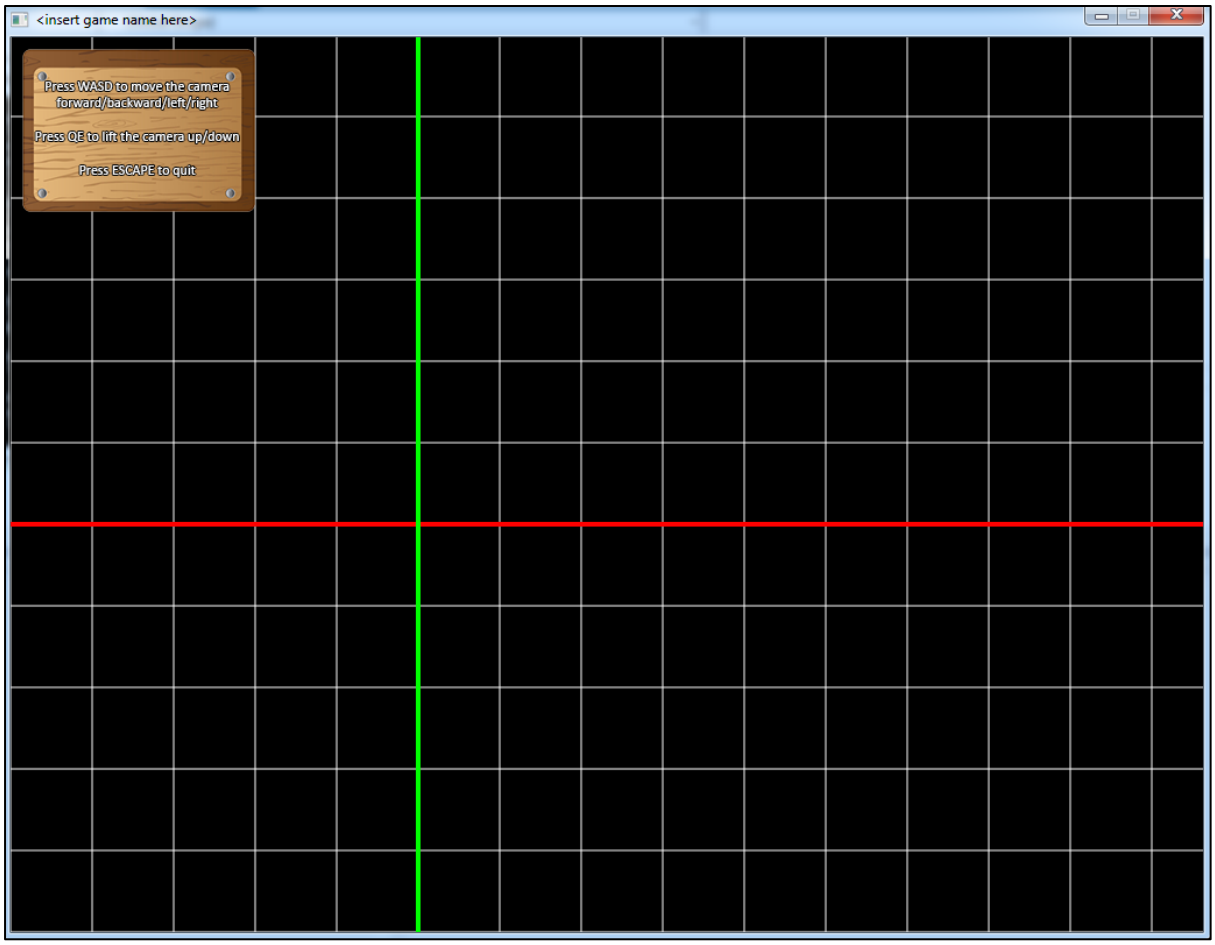
```
int screenWidth = 800;
int screenHeight = 600;

std::string gameName = "My Awesome Game";
```

The pixel per unit value is a setting for 2D applications, which determines how many pixels will represent one unit of measurement. This is because when rendering in 2D, *OpenGL* uses pixel measurement. However for us to move things in the game world, we would prefer to use units instead. Therefore our units of measurement need to be linked to some kind of pixel amount. Below are a few examples

```
int pixelsPerUnit = 10;
```



```
int pixelsPerUnit = 70;
```



Looking further down the main source file, you will find that the main class that runs the entire game engine is something called *TheGame*. Through this class, the engine can be initialized, run and shutdown. To set the game up in fullscreen mode, using the window name, width, height and pixel per unit value, one would use the following :

```
TheGame::Instance()->Initialize(gameName, screenWidth, screenHeight,
pixelsPerUnit, true);
```

After the game has been initialized, it is ready to run. That is done by calling the *Run()* function. This is nothing more than an internal loop that checks for active game states and runs those until there are no more game states to manage, at which point the game loop ends. When the game ends, it is shutdown and the application is done.

```
TheGame::Instance()->Run();
TheGame::Instance()->ShutDown();
```

There is also a main game state, *MainState*, that is created and added to the game. All game and demo specific code will be placed in the *MainState* header and source files. There is room for improvement here. For instance, currently there are plans to add support to transition between multiple states, but it hasn't been implemented properly yet. For now there is only one state, *MainState*, through which everything runs. Perhaps you could add your own functionality in here?

The Shaders

The game engine runs on two different shaders, each with their own vertex and fragment part. These shaders are compiled at runtime and processed by your graphics card. They have all been written in *OpenGL's* shading language *GLSL*.
The *Debug* shaders are intended for drawing all debug objects, which is explained a little later on. The *Main* shaders are for all the main game objects and scenes that are to be drawn in your game demo. There isn't really much difference between the two shaders, except that the *Main* shaders support texturing. The reason they were created separately was to demonstrate the use of multiple shaders and how they can be used in the overall program. Feel free to change things here as well as you see fit.
Typically, if you are going to use debug objects via the *Debug Manager*, you will need to attach and link the *Debug* shaders, and for all other rendering use the *Main* shaders.

The Main State

This is the default game state that has already been added to the game so that things can run. More game states can be added, they just have to be created manually and make sure that they derive from the *GameState* base class, just like *MainState* does. However, functionality for multiple game states has not been fully created or tested yet, therefore at present, all things run from the *MainState*. Feel free to change things here!
The first thing this state does is create and compile the *Main* vertex and fragment shaders, which will be used to render most things in the application. It also creates the debug grid and main camera.
The *MainState* will be your main hub of game management (unless you create other game states), and it is here that you will spend the most time placing your game / demo specific code and functionality. Below is a small guideline as to where you can place your code in the state so as to keep it as organized as possible :

OnEnter() – Here is the place to put all your start-up code, which can be anything like object initializations, resource loading, game object creation, or shader compilation. This function will be automatically called when the *MainState* becomes active.

Update() – Here all your game objects and regular state objects will be updated. If you have previously created and added game objects to the main game object vector, they will be updated automatically (if they are active). But you may want to manually update other things here.

Draw() – Similar to *Update(),* here all your game objects and regular state objects will be drawn. Note that inside here the 2D or 3D screen projection is set up, based on what game mode is set. Leave these at their default values for now.

OnExit() – The opposite happens here to what occurs in *OnEnter(),* ie – here all shutdown tasks are performed, like freeing objects and resources from memory, or closing down specific objects.

Note : If you create your own game states, the above guideline will still apply, as these functions are still used and called automatically. Just make sure that you virtually override the above functions in your particular game state!

The Managers

In total there are eight managers that the game engine uses to control certain sub-systems. They are all initialized and updated from within the main game class, so you don't need to worry too much about calling any functions from within the managers. Of course there will be times when you need to load resources, check delta time, or read input, and that's when these managers will come in handy. However, many of these specific procedures will be explained in upcoming sections. The main managers are described briefly below to give you an idea what they are used for.

Screen Manager : This uses the underlying *SDL* and *OpenGL* APIs to create and manage your game window and graphics context. It supports 2D orthographic and 3D perspective projection. It also stores and controls the *projection* and *modelview* matrix, the latter being the one you will use the most to transform your game objects and scenes.

Debug Manager : This is a debug tool and is only initialized and used in *Debug* mode. It allows for you to draw primitive 2D and 3D objects like vectors, cubes and spheres on screen for basic visuals and representations of game objects or scenes.

Input Manager : This manager controls all mouse and keyboard input using *SDL*, and stores values that you can used in your game / demo to allow for interactivity.

Audio Manager : All music, sound effects and voice data is stored in this manager and can be accessed at any time. It uses the underlying *FMOD* API to run and update the audio sub-system.

Texture Manager : All images, sprites and textures are loaded from file and stored here, and can be accessed globally for use later. It uses the underlying *OpenGL* API to store the images and *SDL* to load them.

Time Manager : This sub-system uses *SDL* to manage time in your game / demo. From here you can access delta time and total time elapsed since your application began running.

Shader Manager : Each vertex and fragment shader that you create is stored and managed via this manager. It can be used to attach, link, compile, detach and destroy shaders.

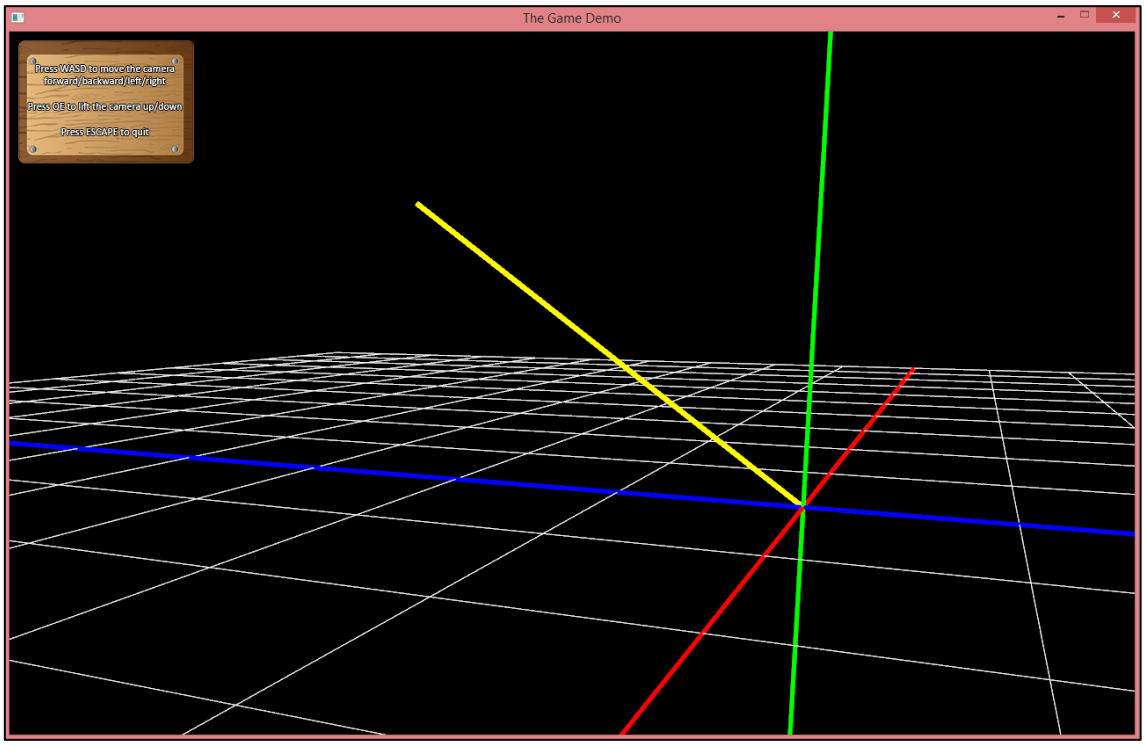
Buffer Manager : For each game object, regular object, sprite, or 3D model a VBO (vertex buffer object) is created to manage the vertex data that represents the object. All VBOs are stored and controlled via this manager.

The Debug Manager

Of all the managers mentioned above, the one you will most likely work with the most is the *Debug Manager*. This is used mainly for debug tools, and for drawing primitive objects, before going on to draw more complex 3D models or animated sprites. It offers a wide variety of tools, of which some are demonstrated below.

Vectors

```
TheDebug::Instance()->DrawVector(float x, float y, float z, float
lineWidth, Color color, int pixelsPerUnit);
```



The *x, y* and *z* arguments are the vector's components. If you're rendering in 2D, just insert a **0** for the *z* portion. The vector's origin point and how it will be rendered, is based on how the world has been transformed previously. Transforms are explained a bit more in detail later on. The *lineWidth* argument specifies how thick the line drawn should be and the *color* argument is the vector's color, which can be any color value set using the built-in *Color* object. To set it to yellow for instance, use the following argument :

```
Color::YELLOW;
```

Note : For more information on the Color class, see the Color.h header file which is part of the game engine project

The final argument is the pixel per unit value which determines how many pixels one unit of measurement will be in 2D mode. By default it has a value of **1**, but if you are rendering in 2D mode, you can simply insert its value by calling the *Screen Manager's GetPixelsPerUnit()* function, which will return the value you set up in the *Main.cpp* source file.

Below are two full examples demonstrating how to use the *DrawVector()* function in both modes :

2D mode :

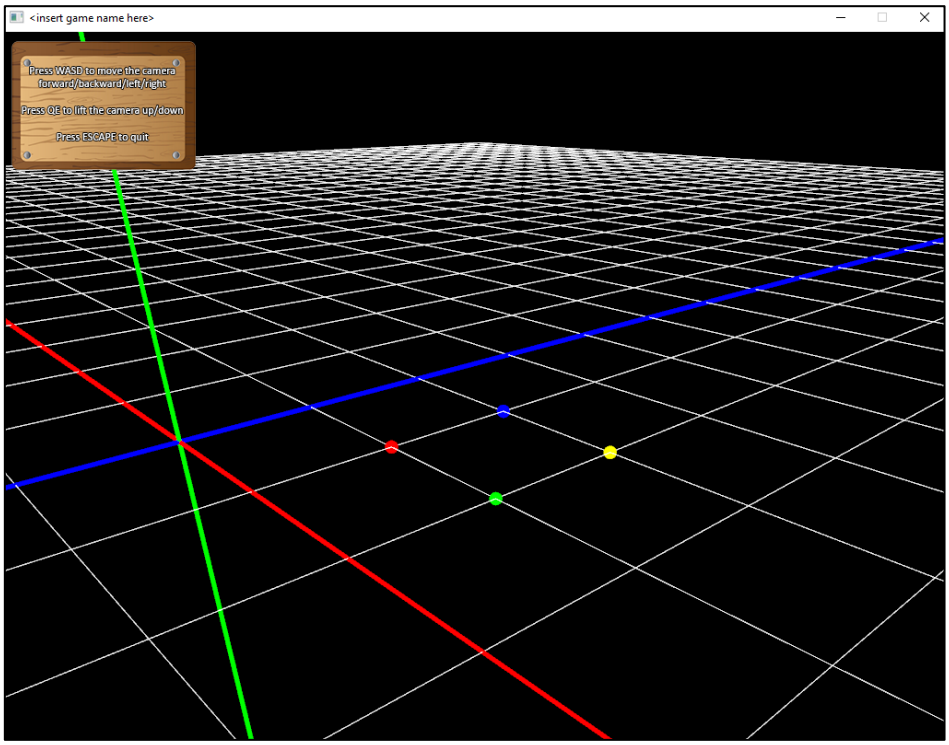
```
TheDebug::Instance()->DrawVector(2, 3, 0, 5, Color::YELLOW,
TheScreen::Instance()->GetPixelsPerUnit());
```

3D mode :

```
TheDebug::Instance()->DrawVector(1, -3, 2, 4, Color::GREEN);
```

Vertices

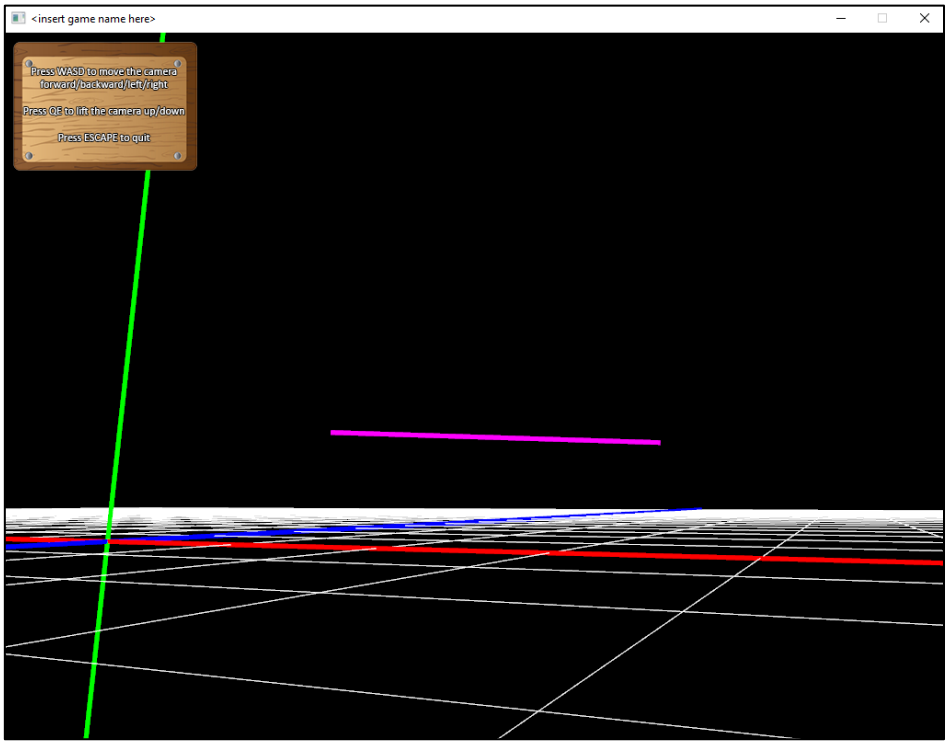
```
DrawVertex(float x, float y, float z, float pointSize, Color color, int
pixelsPerUnit);
```



The *x, y* and *z* arguments are the actual vertex points in world space that you wish to render. For 2D mode, insert a **0** for the z component. No previous transforms will affect the points drawn, as these values state exactly where in the game world you want them to be. The *pointSize* argument specifies how large you want the vertices to be. The last two arguments *color* and *pixelsPerUnit* work exactly the same as in the above mentioned *Vectors* example.

Line Segments

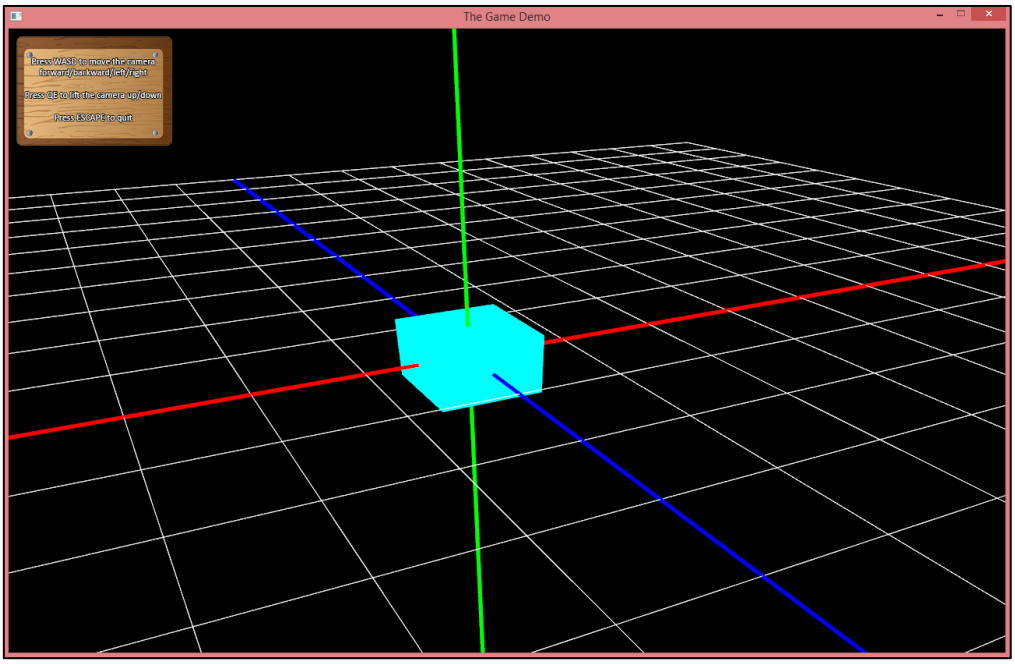
```
DrawLine(float x1, float y1, float z1, float x2, float y2, float z2, float
lineWidth, Color color, int pixelsPerUnit);
```



The first three arguments represent the starting point for the line segment, and the following three arguments represent the ending point. Again, if you are rendering in 2D mode, insert a **0** value for the two *z* components. The final three arguments work exactly the same as the ones in the above mentioned *Vectors* and *Vertices* examples.
Line segments are very similar to vectors, except that they render based on the six specific world space vertex points passed into the function. Vectors render based on how the world has been orientated, and their origin points are relative to the previous transform. The values you pass into a vector are local space coordinates.

Cubes

```
TheDebug::Instance()->DrawCube3D(width, height, depth, color);
```



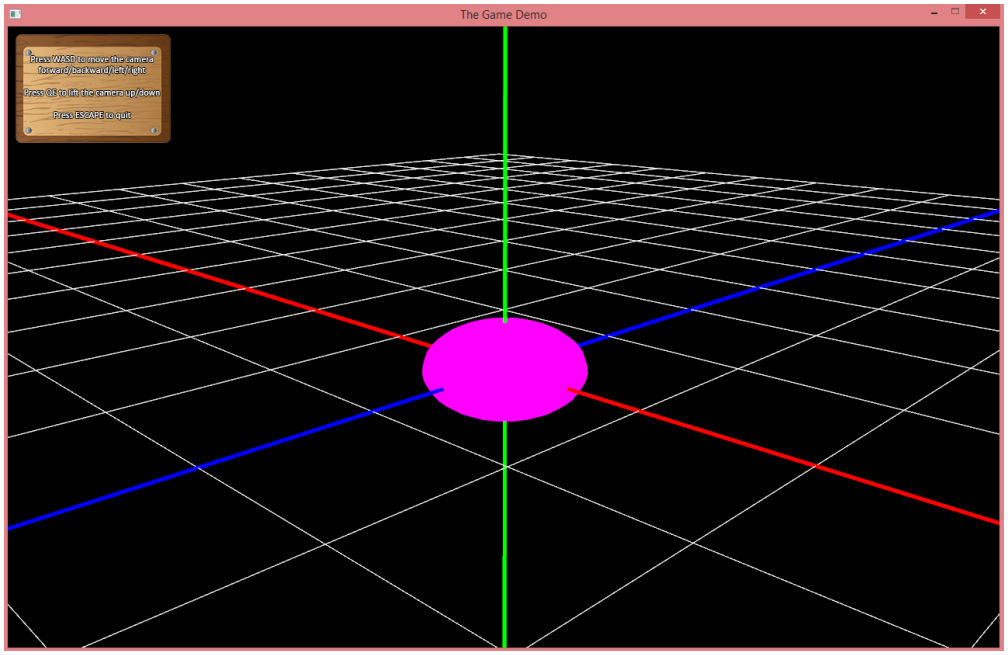
This will draw a cube in the world, based on the latest transformation, using the *width, height* and *depth* values that were passed to the function. The *color* argument is used exactly the same as the above mentioned *Vectors*, *Vertices* and *Line Segments*.
There is a 2D version for this routine as well, so if you want to render a cube in 2D mode, the function looks slightly different.

```
void DrawCube2D(float width, float height, Color color, int
pixelsPerUnit);
```

As we can see the 2D version has no depth, and the *pixelsPerUnit* argument is needed to specify pixel measurement as stated above.

Spheres

```
TheDebug::Instance()->DrawSphere3D(radius, color);
```

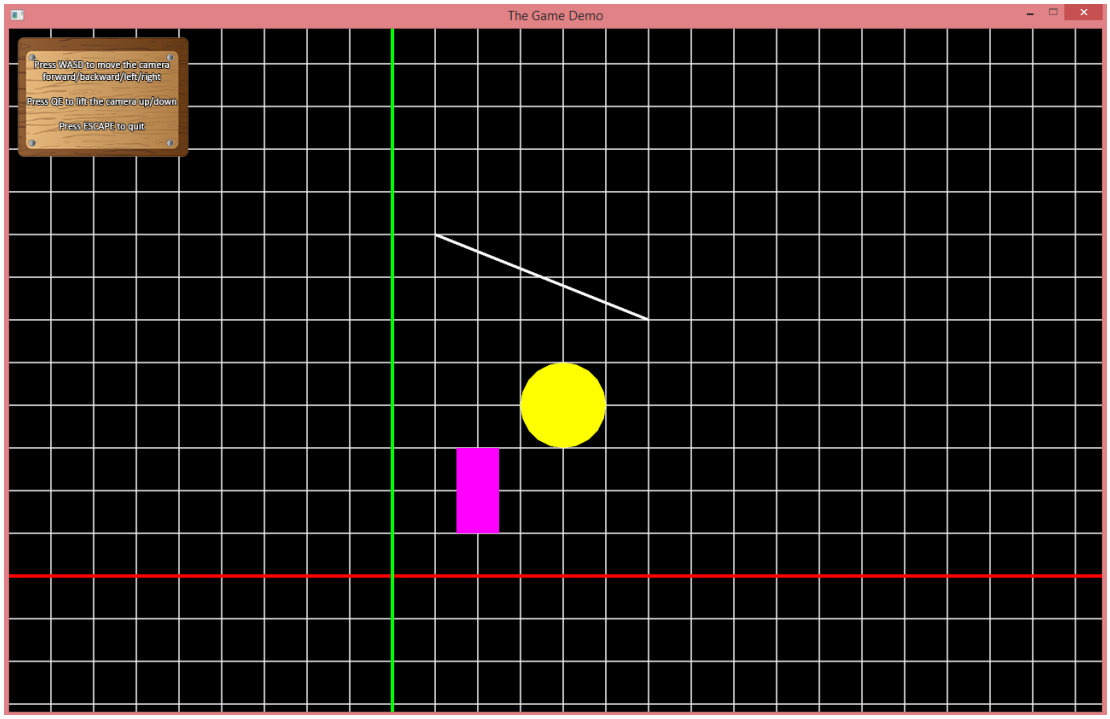


This will draw a sphere in the world, based on the latest transformation, using the *radius* value that was passed to the function. The *color* argument is used exactly the same as the above mentioned *Vectors*, *Vertices* and *Line Segments*. There is a 2D version for this routine as well, so if you want to render a sphere in 2D mode, the function looks slightly different.

```
void DrawSphere2D(float radius, Color color, int slices, int
pixelsPerUnit);
```

The 2D version has an argument called slices, which specifies how detailed the sphere should be. Imagine it to be like slices in a pizza, so the more slices you specify, the more spherical the object will appear. The *pixelsPerUnit* argument is needed to specify pixel measurement as stated above.

Below is an example of the cube, sphere and vector (or line segment) being rendered in 2D mode.



There are of course a few other features that make the *Debug Manager* very useful, but to begin with the draw tools demonstrated above are good enough to get a head start in rendering some basic shapes on screen. Take a look inside the *DebugManager.h* header and source file for more info on the class and what it can do.

Note : The debug draw function calls can be used anywhere, ie directly called from within the MainState or from within a game object's Draw() routine. The latter is of course the preferred method.

Game Objects

These are the building blocks of the game and can represent anything like a player, enemy, rock, table, bullet, etc. They are designed similarly to the way the *Unity Game Engine* designed their game objects, in that they are empty shells that will contain *components* to give them functionality.
Ideally when you create your game objects you will want them to derive from the abstract base class *GameObject*, and it should at the very least override the *Update()* and *Draw()* functions.
For instance, if you want to create a space ship game object, its header file would look like this :

```
class SpaceShip : public GameObject
{

public:

	SpaceShip();
	virtual ~SpaceShip() {}

public:

	virtual void Update();
	virtual bool Draw();

private:

	//components
	//member variables

};
```

Now all that needs to be done is implement the *Update()* and *Draw()* functions in the *SpaceShip.cpp* file, and they will be called from within the game state's *Update()* and *Draw()* routines respectively. Ideally, you will want to place all updated positional, rotational and input management data in the *Update()* function and all *modelview* transformation applications and draw calls inside the *Draw()* function. Don't panic just yet, these elements are explained and demonstrated a little later on.
To instantiate your newly designed space ship, simply create it from within the *MainState's OnEnter()* function like so :

```
m_gameObjects.push_back(new SpaceShip());
```

Alternatively, you can also create a pointer to the *SpaceShip* object in the *MainState* header file, and then instantiate the object on its own in the *OnEnter()* function :

```
//in the MainState header file
SpaceShip* m_mySpaceShip;

//in the MainState source file
m_mySpaceShip = new SpaceShip();
```

The difference is that if you create and add the game object straight into the game object vector, it will automatically update and draw in the *MainState's Update()* and *Draw()* functions. If you use the alternative method, you will need to call the *SpaceShip* object's *Update()* and *Draw()* routines manually. You may have noticed that the *MainState* already comes with three game objects already defined, namely *m_grid, m_mainCamera* and *m_HUD* which are all instantiated in the *OnEnter()* function.

Another method of instantiating the spaceships would be to create a separate vector of them, and then add each required space ship object into the vector when needed :

```
//in the MainState header file
std::vector<SpaceShip*> m_spaceShips;

//in the MainState source file
m_spaceShips.push_back(new SpaceShip());
```

The only big change would be to add a loop that will run through the space ship vector in order to update and draw them. Therefore, there would be a loop in the *MainState's Update()* function and one in the *Draw()* function.

```
//UPDATE LOOP
//loop through all space ships in vector and update them
for (auto it = m_spaceShips.begin(); it != m_spaceShips.end(); it++)
{
	if ((*it)->IsActive())
	{
		(*it)->Update();
	}
}
```

```
//DRAW LOOP
//loop through all space ships in vector and render them
for (auto it = m_spaceShips.begin(); it != m_spaceShips.end(); it++)
{
	//set up camera view before each space ship is drawn
	m_mainCamera->Draw();

	if ((*it)->IsActive() && (*it)->IsVisible())
	{
		(*it)->Draw();
	}
}
```

How you wish to create your game objects is up to you, however it is important to remember that good organization is key. Keep your game objects grouped together in specific categories and don't instantiate thousands of individual objects when they can all be added to a vector instead.

One final thing to note is cleanup! When the *MainState* exits, it will call its *OnExit()* function, which performs all kinds of memory cleanup, including the destruction of game objects. To make sure your space ship vector is being thoroughly destroyed, a loop is needed to clean up the vector :

```
//loop through all space ships in vector and remove them from memory
for (auto it = m_spaceShips.begin(); it != m_spaceShips.end(); it++)
{
	delete (*it);
}

//clear the space ship vector
m_spaceShips.clear();
```

Of course if you have any individual game object pointers, simply destroy them like so :

```
delete m_spaceShip;
m_spaceShip = 0;
```

The Grid

The *m_grid* object represents the 2D or 3D grid and coordinate system that you see when running your project the first time. It will help orient your surroundings and other game objects in relation to each other instead of working in the dark world of *OpenGL*. Looking inside the *Grid* class header and source file you will find that it simply uses the *Debug Manager's* two functions to create a grid and coordinate system.

```
TheDebug::Instance()->DrawGrid3D(size, lineWidth);
TheDebug::Instance()->DrawCoordSystem3D(size, lineWidth);
```

Feel free to adjust the *size* and *lineWidth* values to whatever you see fit. Of course a 2D version is also available, which will activate if the project is set to 2D mode, and comes with an additional argument to represent a pixels per unit size value. This value will be the same one you set up in the *Main.cpp* file, which would have been also used to initialize the game if in 2D mode.

Note : You can use the DrawCoordSystem2D() and DrawCoordSystem3D() for your regular game objects, not just for the main grid. In other words, using these functions together in the draw calls of your game objects will draw the local coordinate system of the game objects, specifying how they are rotated in world space.

The Main Camera

This is the camera that you use when moving around in your scene. Looking at the *MainCamera* header and source file you will find it uses a *Camera* component and constantly uses the *Input Manager* to determine where to move or rotate. Here is a first glimpse of how the *Input Manager* may be used to read keyboard and mouse input for your game. Feel free to make changes to the *CheckInput()* function – perhaps you would like the camera to move forward when you press **F**?
The *MainCamera* also checks for when the player presses **ESCAPE**, at which point it will signal the *MainState* to end. It also acts as the main view for the entire scene, and as you can see in the *Draw()* routine it will reset the *modelview* matrix and update and draw the internal *Camera* component. This is essential because this sets up the view each time before a game object is drawn. This is also why you see the *MainCamera's Draw()* call being executed before every other game object is drawn in *MainState's Draw()* function :

```
for (auto it = m_gameObjects.begin(); it != m_gameObjects.end(); it++)
{

	m_mainCamera->Draw();

	if ((*it)->IsActive() && (*it)->IsVisible())
	{
		(*it)->Draw();
	}

}
```

The HUD

The third object, *m_HUD*, is a GUI game object that will display the friendly 2D image in the top left corner of the screen to offer you instructions on how to move around in the scene. Feel free to also peek inside the *HUD* header and source file. Beware though, it might look a little overwhelming in here because it makes use of a *Sprite* component and the *Draw()* function looks rather wild! Sprites and other components will be detailed and demonstrated a little later down the line.
If at any point the HUD bothers you and you wish to remove it, simply comment out the line below, in the *MainState's Draw()* function, or simply remove the entire object.

```
m_HUD->Draw();
```

Components

These are the entities that will exist inside the *Game Objects* that will determine their functionality. This design and principle works similar to the way the *Unity Game Engine* uses *Game Objects* and *Components*. There are quite a few components that come pre-packed in the engine. The following sub sections will detail them all. Feel free of course to also delve into each component's header and source file to learn more about each one. They are all separately documented in each header file.

The Mathematics Components

These components feature all the mathematical functionality that is used throughout in game / demo development. Before going any further, the answer to this burning question – *"Why not use GLM instead?"* – is that these maths components were created purely for learning purposes, in order to truly understand how they function from within. There is a *Vector2D, Vector3D* and a *Vector4D* to represents vectors of various dimensions, two *Matrix* classes for transformations and a *Quaternion* class for more fluid rotations. Of course if you prefer to use the *GLM* library, you will find it here : http://glm.g-truc.net/0.9.7/index.html

The Transform Component

The class that will be used quite often is the *Transform* class. This class represents all translations, rotations and scaling. These values need to be set and then applied to the *modelview* matrix to have any effect in the game world. This is why it was previously stated that all transformations should be ***updated*** in the game object's *Update()* function, but ***applied*** to the *modelview* matrix in the *Draw()* routine before drawing anything.
Let's demonstrate this with an example using our existing *SpaceShip* class. First, add a *Transform* component to the *SpaceShip* class in its header file. This transformation will now represent our position, rotation and scale of the object. In order to have our space ship active and displayed on screen, make sure you instantiate it in the *MainState's OnEnter()* function like so :

```
m_gameObjects.push_back(new SpaceShip());
```

```
class SpaceShip : public GameObject
{

public:

	SpaceShip() {}
	virtual ~SpaceShip() {}

public:

	virtual void Update();
	virtual bool Draw();

private:

	Transform m_transform;

};
```

Translation

This is essentially the position of the object, and the translation will determine how much the object will move in terms of its x, y and z position. To translate the object, you will need to use the following command :

```
m_transform.Translate(x, y, z);
```

Now the position has been determined and set (based on the previous transformation). Remember, each transformation will accumulate, so each transform call adds on to the existing one unless the transform object is reset to the identity like so :

```
m_transform.GetMatrix() = Matrix4D::IDENTITY;
```

For the new translation to have any effect, it will need to be applied to the *modelview* matrix first. This is done like so :

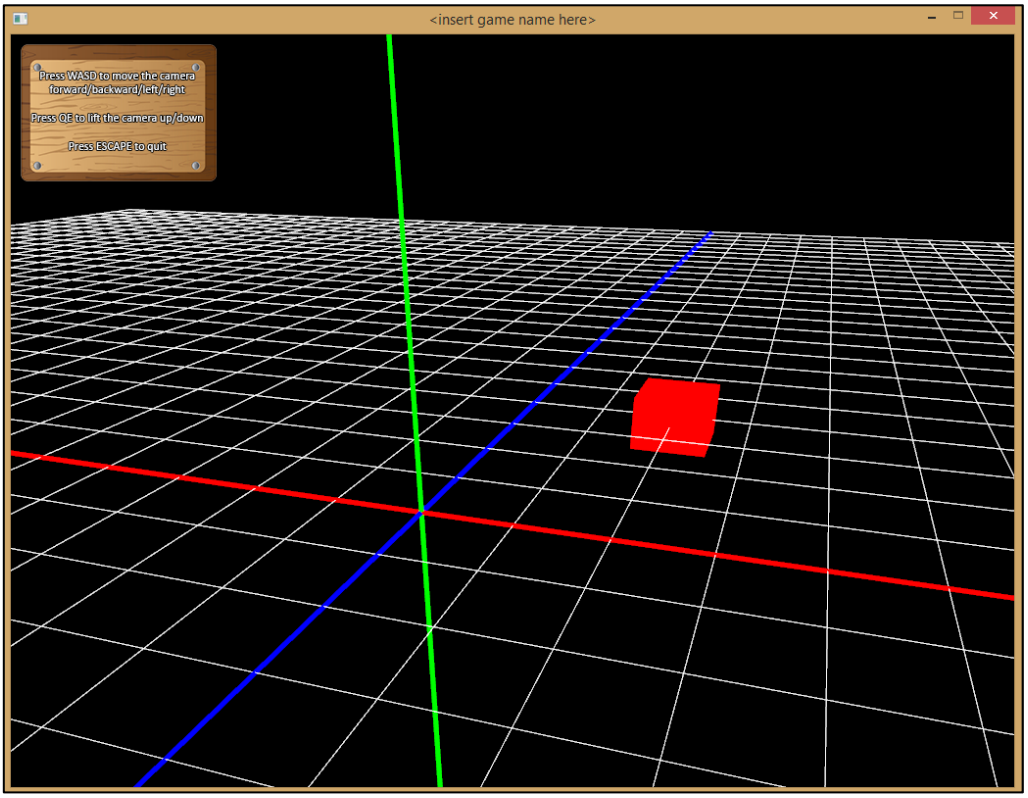
```
TheScreen::Instance()->ModelViewMatrix() * m_transform.GetMatrix();
```

As soon as the *modelview* matrix is set up with the transformation, we can draw something on screen to represent our newly transformed space ship :

```
TheDebug::Instance()->DrawCube3D(1, 1, 1, Color::RED);
```

The commands shown above should be placed in the correct functions, so let's put it all together in the *SpaceShip* class' source file :

```
void SpaceShip::Update()
{

	//reset transform component each frame
	m_transform.GetMatrix() = Matrix4D::IDENTITY;

	//position space ship at following 3D coordinates
	m_transform.Translate(2, 0, -4);

}

bool SpaceShip::Draw()
{

	//apply transformation to modelview matrix each frame
	TheScreen::Instance()->ModelViewMatrix() * m_transform.GetMatrix();

	//draw a spaceship (a very primitive looking one!)
	TheDebug::Instance()->DrawCube3D(1, 1, 1, Color::RED);

	return true;

}
```



And low and behold as you can see the space ship has moved **2** in x, nothing in y and **-4** along the z axis. As long as the space ship game object has been correctly instantiated in the *MainState* and added to the game object vector, its *Update()* and *Draw()* functions will be automatically called.

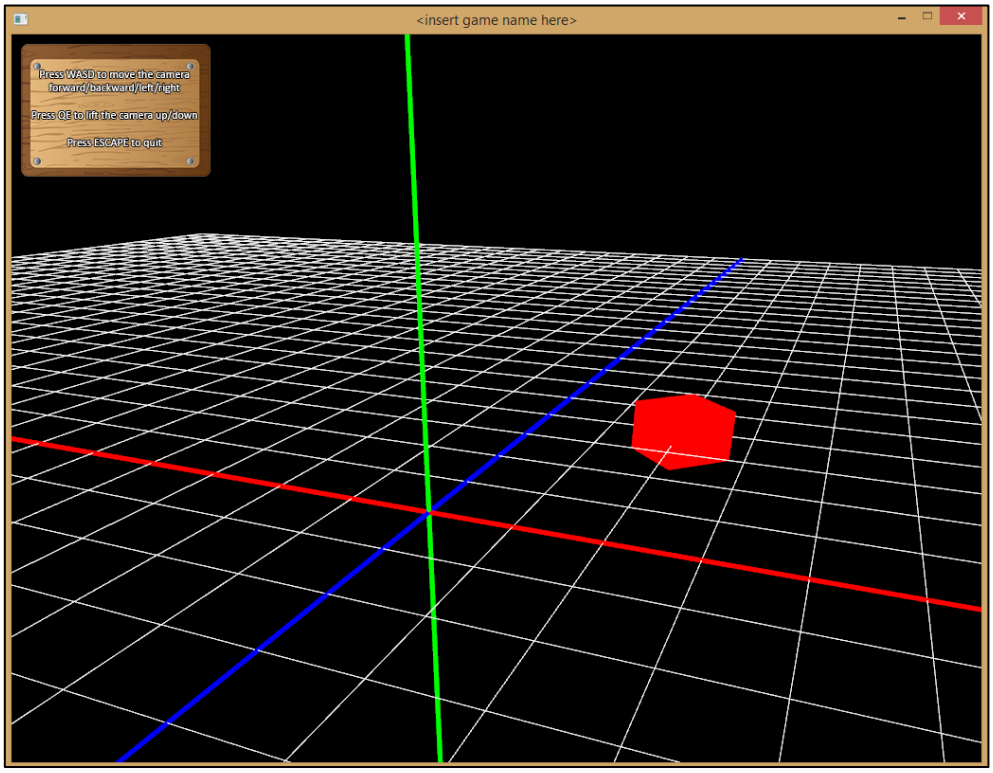
Rotation

This will rotate the object in either of the three axes, namely X, Y or Z axis. These rotations will make use of *Euler* angles, therefore rotating in a specific order will result in a specific rotation. If you want more flexible rotations, consider using *Quaternions* instead, which are demonstrated a little later on. To rotate in either of the three axes, use the following function and pass in the desired *angle* and *axis* to rotate around like so :

```
m_transform.Rotate(angle, Transform::X_AXIS);
m_transform.Rotate(angle, Transform::Y_AXIS);
m_transform.Rotate(angle, Transform::Z_AXIS);
```

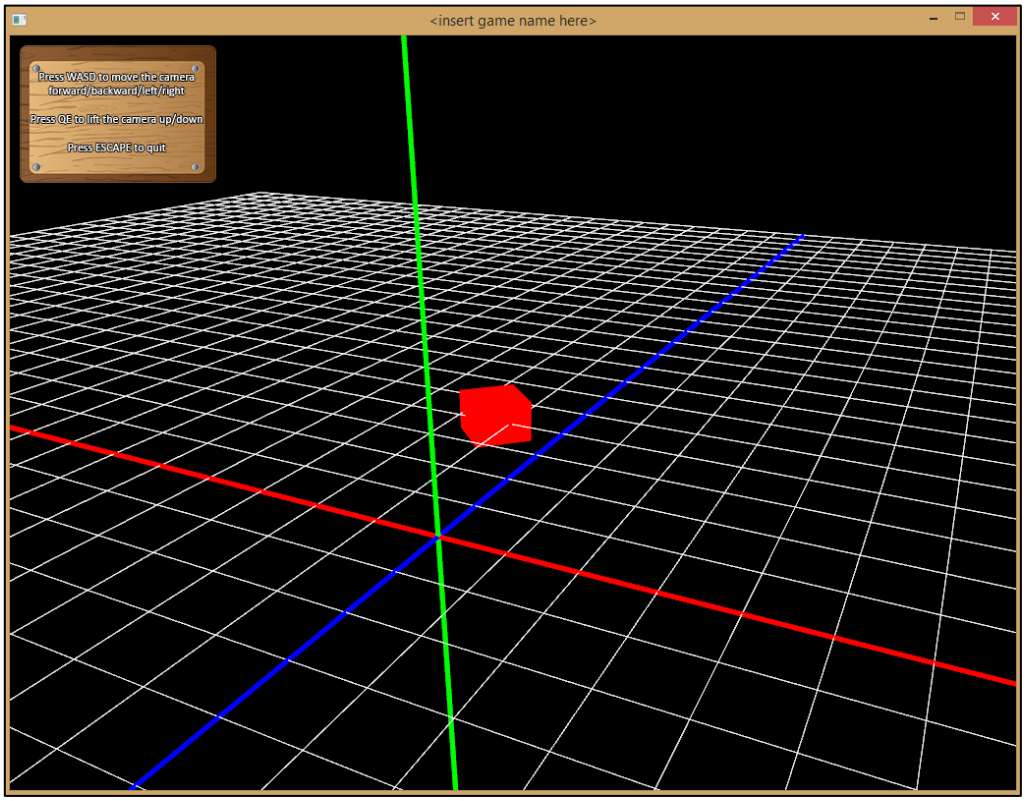
Let's now add a rotation to our space ship :

```
//position and rotate space ship
m_transform.Translate(2, 0, -4);
m_transform.Rotate(45, Transform::Y_AXIS);
```



As you can see, the rotation transformation is added to the already existing translation transformation. You will also notice that the order of transformation matters. If you switch the above statements, you will get a different result!

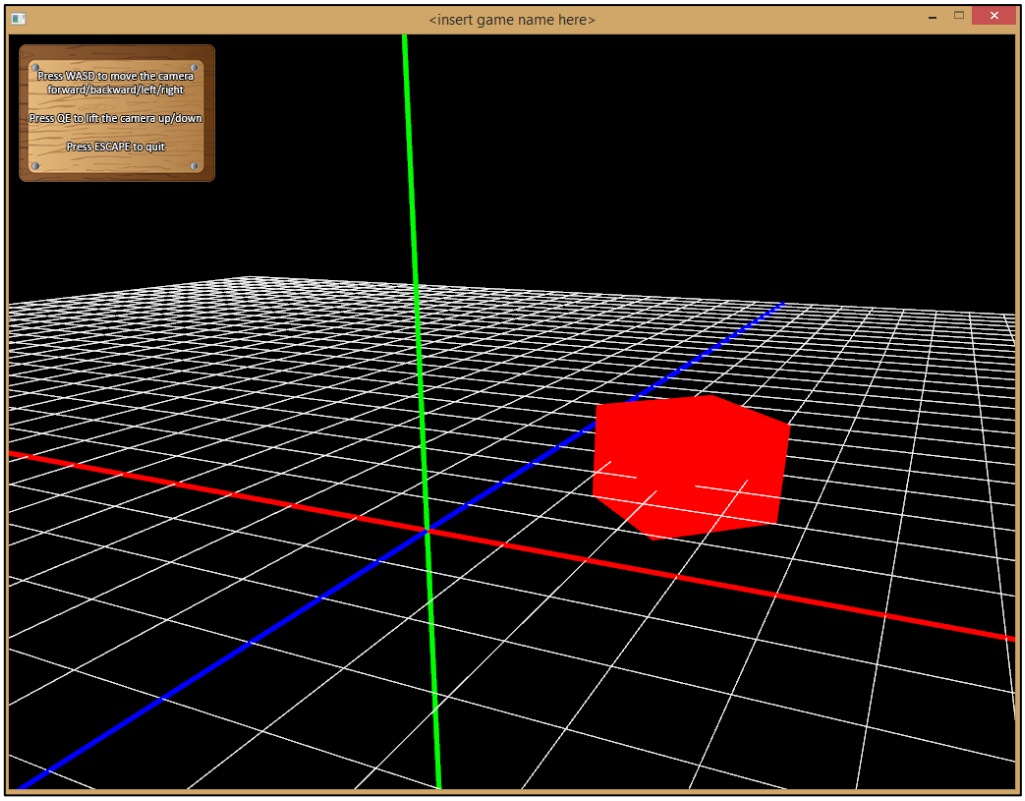
```
m_transform.Rotate(45, Transform::Y_AXIS);
m_transform.Translate(2, 0, -4);
```



Scale

This will increase or decrease the size of the object. Using a value larger than **1** will enlarge it, and a value less than **1** (but larger than **0**!) will shrink it. Of course a value of **1** keeps it the same size. Let's try this out. Update the code in the *SpaceShip's Update()* routine to look like this :

```
m_transform.Translate(2, 0, -4);
m_transform.Rotate(45, Transform::Y_AXIS);
m_transform.Scale(2, 2, 2);
```



Now as you can see, the space ship has been double in size in all of its axes, ie it's doubled in *width, height* and *depth* (x, y and z).

The *Update()* function can be further extended to update the actual position of the space ship. First, add a *Vector3D* object just under the *Transform* declaration in the *SpaceShip's* header file. This will represent the object's position :

```
Transform m_transform;
Vector3D<float> m_position;
```

Now add in the following code into the *SpaceShip's Update()* routine :

```
//reset transform component each frame
m_transform.GetMatrix() = Matrix4D::IDENTITY;

//update the position each frame
m_position.X += 0.01f;

//position space ship at following 3D coordinates
m_transform.Translate(m_position.X, m_position.Y, m_position.Z);
```

You will notice now that the space ship moves automatically each frame to the right. Let's try this with keyboard input and time management. The following demonstration will show you how to achieve this, and will familiarize you with how to use the *Input Manager* and *Time Manager*.

```
//reset transform component each frame
m_transform.GetMatrix() = Matrix4D::IDENTITY;

//store keyboard key states in a temp variable for processing below
const Uint8* keyState = TheInput::Instance()->GetKeyStates();

//move space ship forward in the Z axis each time UP key is pressed
//use delta time to control the movement
if (keyState[SDL_SCANCODE_UP])
{
      m_position.Z -= 1.0f * TheTime::Instance()->GetElapsedTimeSeconds();
}

//position space ship at following 3D coordinates
m_transform.Translate(m_position.X, m_position.Y, m_position.Z);
```

You can now see that with very little effort, a small interactive demo has been created, even if it still looks so primitive. Wait till we cover *sprites* and *3D models* a little later, things will start to look much better!

Note : For more information on how to use the Input Manager and Time Manager, and what they have to offer, feel free to look through the well documented header files of each class.

Of course the above demonstration is not only limited to translations. Rotations and scale can also be updated using similar techniques. Why not try that out yourself and see if you can rotate or scale an object using key presses?

Also remember that transformations are very much achievable in 2D as well. The only big difference is that the pixel per unit value needs to be considered when translating and scaling, and rotations only happen around the Z axis. Let's use our space ship in 2D mode now, with the pixel per unit value setting of **50** (adjusted in the *Main.cpp* file). Add a *Vector2D* object instead of a 3D one because we are in 2D mode :

```
Transform m_transform;
Vector2D<float> m_position;
```

Now change the *SpaceShip* class' *Update()* code to look like this :

```
//reset transform component each frame
m_transform.GetMatrix() = Matrix4D::IDENTITY;

//store keyboard key states in a temp variable for processing below
const Uint8* keyState = TheInput::Instance()->GetKeyStates();

//move space ship forward in the Y axis each time UP key is pressed
//use delta time to control the movement
//multiply by the set pixel scale value to consider pixel movement
if (keyState[SDL_SCANCODE_UP])
{
      m_position.Y += 1.0f * TheScreen::Instance()->GetPixelsPerUnit()
                           * TheTime::Instance()->GetElapsedTimeSeconds();
}

//position space ship at following 2D coordinates
m_transform.Translate(m_position.X, m_position.Y, 0);
```

As you can see the object moves up along the Y axis and indeed the pixel per unit value is needed to multiply by the unit of movement (**1.0**). This just makes sure we move **1** unit of **50** pixels upwards (based on time elapsed of course).

The game object's *Draw()* routine will look like this :

```
//apply transformation to modelview matrix each frame
TheScreen::Instance()->ModelViewMatrix() * m_transform.GetMatrix();

//draw a spaceship (a very primitive looking one!)
TheDebug::Instance()->DrawCube2D(1, 2, Color::RED, TheScreen::Instance()->GetPixelsPerUnit());
```

Again, the pixel per unit value is used to draw the 2D cube, meaning its size is **1x50** pixels in *width* and **2x50** pixels in *height*. When scaling, the pixel per unit value also needs to be considered, however rotations use degrees therefore they don't link with any pixel values.

Note : At this point you may still feel a little in the dark, which is totally okay. The best way to truly understand everything and get a grip on how things work is to play around with the above examples and see what results you come up with.

The Quaternion Component

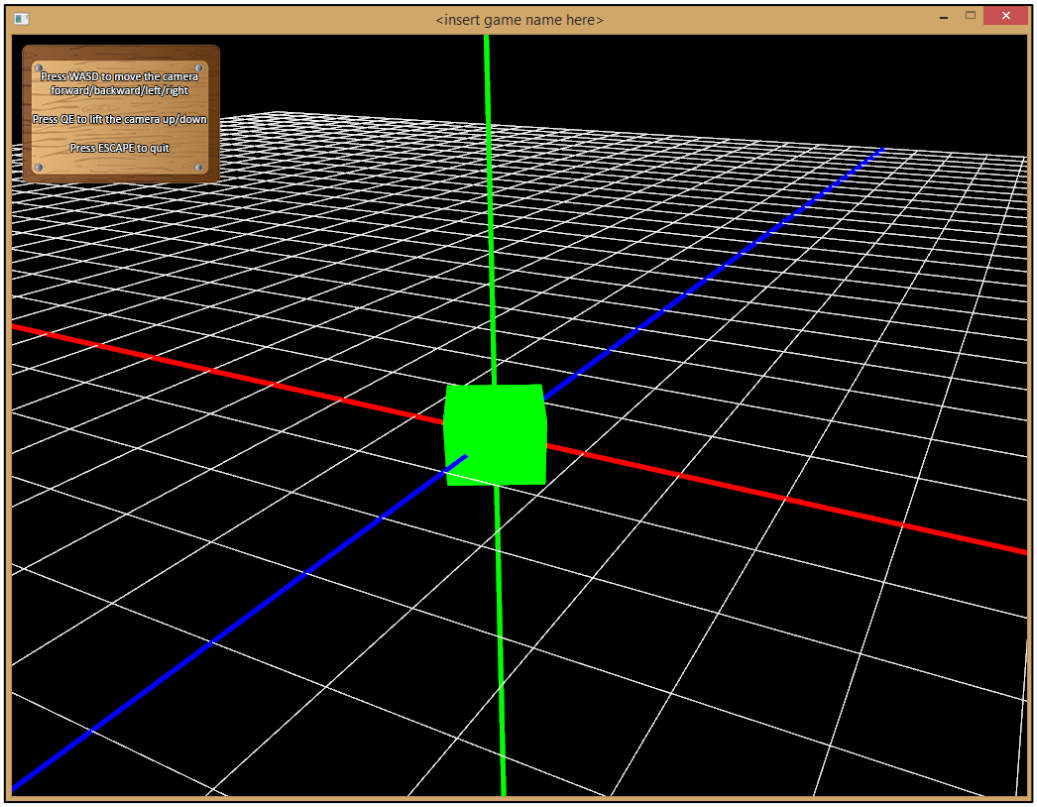
This is part of the *Mathematics* components, and can be used as an alternative to the regular *Euler* rotations used in the *Transform* component, to achieve more fluid and flexible rotations. Remember using *Euler* angles can become quite restraining because the order of rotations around the three axes X, Y and Z yields different results. The worst case scenario when using *Euler* angles is that you may even get what is called *Gimbal Lock*, which happens when one of the axes is "lost".
This is why *Quaternions* are preferred, and below we will quickly demonstrate how to use them. First declare a *Quaternion* object in the *SpaceShip's* header file :

```
Quaternion m_quaternion;
```

Now we will add the following code to the *SpaceShip's Update()* and *Draw()* routines :

```
void SpaceShip::Update()
{

	//create a rotation of 30 degrees around the Y axis
	m_quaternion.SetRotation(30, 0, 1, 0);

}

bool SpaceShip::Draw()
{

	//apply rotation to modelview matrix each frame
	TheScreen::Instance()->ModelViewMatrix() * m_quaternion.GetMatrix();

	//draw a spaceship (a very primitive looking one!)
	TheDebug::Instance()->DrawCube3D(1, 1, 1, Color::GREEN);

	return true;

}
```

Of course you can again accumulate rotations as the quaternion supports multiple rotations being made and stored. See the *Quaternion* header and source file for more info on how the quaternion can be used and what it has to offer.



Clearly using the above example, the space ship has rotated **30** degrees around the Y axis.

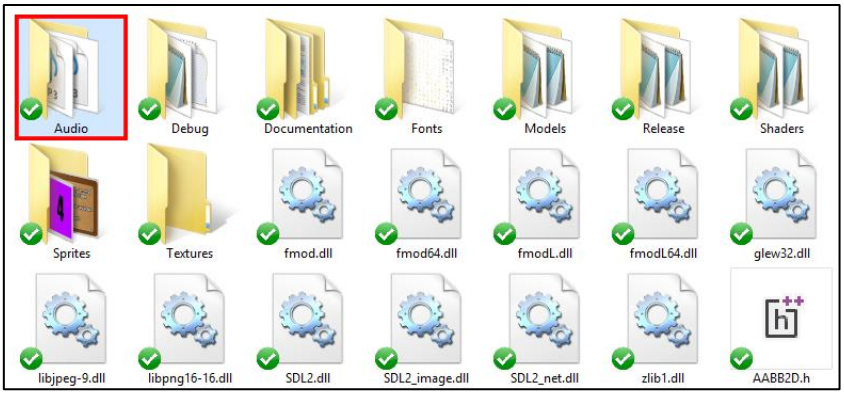
Note : Quaternions are a tricky business. Best is to first get your head around regular Euler rotations before trying out quaternions. I will create a better documentation for quaternions when I get a chance. I will also have a Quaternion Camera demo online that will help explain things a little better!

The Audio Component

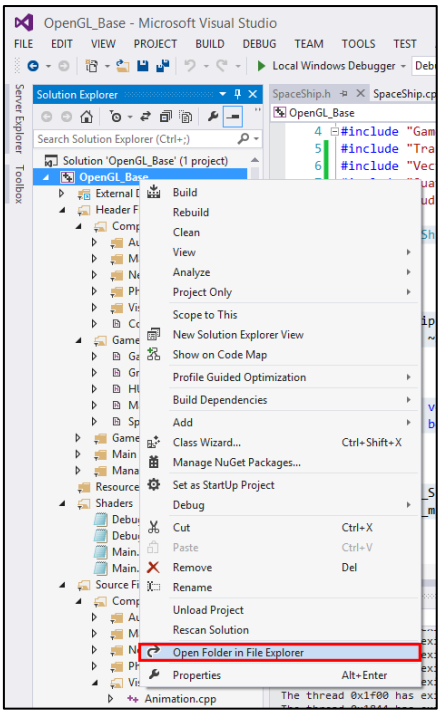
This component is great for music, sound effects and voices in your game or technical demo. It uses the underlying *FMOD* API to play any audio file and comes with a host of features to enable you to manipulate the audio properties in different ways.
To demonstrate, let's add some music and sound effects to our game scene and spaceship. To keep things simple, we will control both audio files from the *SpaceShip* class. First add two *Audio* components in the declaration section of the class, so that your class looks like this :

```
class SpaceShip : public GameObject
{

public:

	SpaceShip();
	virtual ~SpaceShip();

public:

	virtual void Update();
	virtual bool Draw();

private:

	Audio m_SFX;
	Audio m_music;

};
```

Now, let's also make sure the audio resources are added to the *Audio* folder, which you will find in the project folder, the same location where all the header and source files are located. For demonstration purposes, the files *Music.mp3* and *Laser.mp3* have been added to the Audio folder.



If you are unsure where to find the *Audio* folder, simply right click the project name and click on **Open Folder in File Explorer**. This will open the project folder and give you access to the resource folders where all the resource files should be kept.



Now, we need to load the audio files into memory, and to do this there are three ways to load audio into the *Audio Manager* :

```
TheAudio::Instance()->LoadFromFile(filename, AudioManager::MUSIC_AUDIO, tagname);
TheAudio::Instance()->LoadFromFile(filename, AudioManager::SFX_AUDIO, tagname);
TheAudio::Instance()->LoadFromFile(filename, AudioManager::VOICE_AUDIO, tagname);
```

The *Audio Manager* is able to load audio in three ways, one for each audio type. This is because each audio type is stored in a different location and also because music and voice data is streamed from file, whereas sound effects are loaded directly into memory.

The *filename* argument is the location of the file, in both our cases it will be :

```
"Audio\\Music.mp3"
"Audio\\Laser.mp3"
```

The *tagname* is the label you wish to identify your audio piece with so as to keep track of your audio. In our case it will be :

```
"MUSIC" and "LASER"
```

To link the loaded audio data with our audio component, we need to use the following command :

```
m_SFX.SetAudioData(tagname, audioType);
m_music.SetAudioData(tagname, audioType);
```

Now all that is needed is to call the *Play()* function of each audio component to play the music and sound effects. We will demonstrate this below with some working code. The music will be played once in the *SpaceShip* constructor, and remain playing in the background, while the sound effects will play only when the SPACE key is pressed. Let's demonstrate :

```
SpaceShip::SpaceShip()
{

	//load music and SFX audio from resource file
	TheAudio::Instance()->LoadFromFile("Audio\\Music.mp3", AudioManager::MUSIC_AUDIO, "MUSIC");
	TheAudio::Instance()->LoadFromFile("Audio\\Laser.mp3", AudioManager::SFX_AUDIO, "LASER");

	//assign music and sound effects data to our components
	m_SFX.SetAudioData("LASER", AudioManager::SFX_AUDIO);
	m_music.SetAudioData("MUSIC", AudioManager::MUSIC_AUDIO);

	//play the background music
	m_music.Play();

}

void SpaceShip::Update()
{

	//store keyboard key states in a temp variable for processing below
	const Uint8* keyState = TheInput::Instance()->GetKeyStates();

	//if SPACE key was pressed, play the laser sound effect!
	if (keyState[SDL_SCANCODE_SPACE])
	{
		m_SFX.Play();
	}

}
```

Remember to always clean up your act, and especially in C++, where garbage collection and shutdown tasks are the programmer's responsibility! In the *SpaceShip* class' destructor, let's add the following code to stop the music playing and remove the audio resources from memory :

```
SpaceShip::~SpaceShip()
{

	//stop playing the background music
	m_music.Stop();

	//free music and SFX audio from memory
	TheAudio::Instance()->
	UnloadFromMemory(AudioManager::MUSIC_AUDIO, AudioManager::CUSTOM_AUDIO, "MUSIC");

	TheAudio::Instance()->
	UnloadFromMemory(AudioManager::SFX_AUDIO, AudioManager::CUSTOM_AUDIO, "LASER");

}
```

Generally speaking, the destructor should always oppose the constructor, so whatever you loaded from file when the game object was created should be unloaded from memory when the game object is destroyed. The command to unload audio data looks like this :

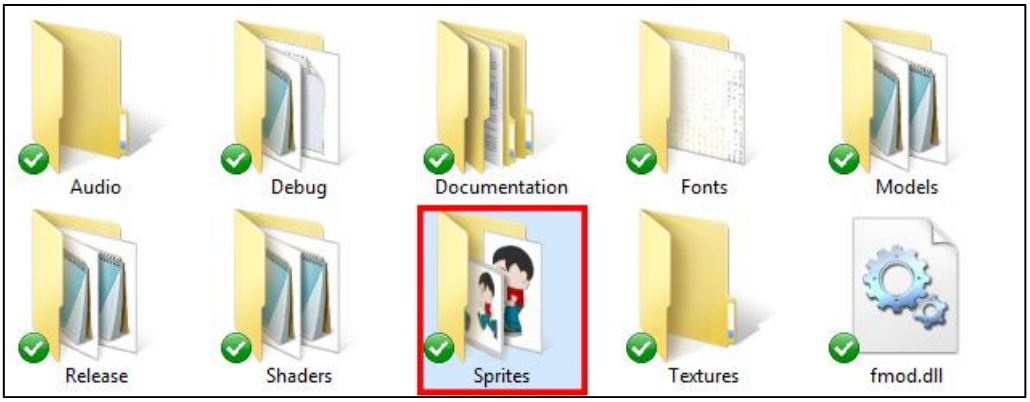
```
TheAudio::Instance()->UnloadFromMemory(audioType, removeType, tagname);
```

Again, the *audioType* can be one of three audio types, the *removeType* specifies how to remove, ie one audio piece at a time or all at once, and the *tagname* is for when you unload separate audio pieces, the name that identifies that audio data.

Note : For more information about how the audio works, take a look through the Audio and Audio Manager header and source files. They are well labelled and commented, and should be able to give you a clearer picture as to what is going on.

The Sprite Component

This component represents a 2D sprite image, which is nothing more than a regular texture that is used in two dimensional game objects and renders an image that represents that object. They are more commonly used in 2D mode, where everything is represented as a boxed shaped image and the sprite creates a pretty picture around it all.
The sprite object can consist of a singular image or an array of multiple images, also known as a *sprite sheet*. Perhaps a demonstration will offer more explanation. We will start with a singular sprite image and progress onto more advanced dynamic animated sprites from there. First, let's set our scene up in 2D mode and then make sure we have our sprite image resource files in the correct folder. Again, head over to the *Sprites* folder and add them in there.



In our case, we will add a *Player.png* file to the *Sprites* folder :

Note : The player image was taken from Google Images, and therefore I do not hold the copyright to it. I am merely using this image for demonstration purposes.

Now let's venture away from our regular *SpaceShip* class and create a new class called *Player*, and for that we will need a class declaration and its subsequent definition. The *Player* will also contain a *Sprite* component :

```
class Player : public GameObject
{

public:

	Player();
	virtual ~Player();

public:

	virtual void Update();
	virtual bool Draw();

private:

	Sprite m_sprite;

};
```

Before we begin using sprite objects, we need to create vertex buffer objects first. Now that may seem like a mouthful, and its okay to feel overwhelmed. Generally speaking, vertex buffer objects (or VBOs for short) are memory locations that contain the entire vertex, texture coordinate and color data that the sprite object consists of. Luckily we only need to create them and give them a name; *OpenGL* will handle the more advanced stuff in the background. We should create a VBO for the vertex, texture coordinate and color data separately, like so :

```
TheBuffer::Instance()->Create(BufferManager::VERTEX_BUFFER, tagname);
TheBuffer::Instance()->Create(BufferManager::COLOR_BUFFER, tagname);
TheBuffer::Instance()->Create(BufferManager::TEXTURE_BUFFER, tagname);
```

Again, just as you saw when using audio components, the *tagname* argument is the label given to identify that particular buffer.
Now it's time to load the sprite image, which we put in the *Sprites* folder earlier, into memory. So let's do that :

```
TheTexture::Instance()->LoadFromFile(filename, tagname);
```

In our case the *filename* will be

```
"Sprites\\Player.png"
```

Our sprite image is **180x300** pixels in dimension, so this information along with the texture and VBO buffer *tagnames* are used now to link our sprite component with the VBOs and texture ID :

```
m_sprite.SetSpriteDimension(width, height);
m_sprite.SetTextureID(texture tagname);
m_sprite.SetBufferID(vertex VBO tagname, color VBO tagname, texture VBO tagname);
```

All we do here is insert the *tagnames* we used to label the texture and VBOs earlier. This will associate the sprite object with the loaded texture and the three VBOs we created above.

Right, now comes the more complex part, and this involves a few steps to get the sprite object to actually render. So far we have created the VBOs that will contain the rendering information and we have loaded our sprite texture into memory, but we need to set up the main shaders (the ones set up in the *MainState*) and actually render our sprite. Let's do that next.

First we have to disable the debug shaders, because we want to use the main game shaders that have been set up in the *MainState*. To do this call the *Debug Manager's Disable()* function.

```
TheDebug::Instance()->Disable();
```

Now attach and link the main game shaders. These will be used to render the player game object :

```
TheShader::Instance()->Attach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Attach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");
TheShader::Instance()->Link();
```

Note : If you prefer to only use the main shaders, you can attach them somewhere early on and leave them attached for the duration of the game / demo. Then the above process doesn't need to be repeated each time.

Next, link the sprite component with the shader variables like so :

```
m_sprite.SetShaderAttribute("vertexIn", "colorIn", "textureIn");
```

Note : By default, the shader variable names should remain as written above. For the more advanced programmers, you are welcome to change them, but make sure you link them correctly before rendering.

Now, we send the *modelview* and *projection* matrix data to the shaders. This makes sure the shaders know how our scene is set up before rendering.

```
TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("projectionMatrix"),
           TheScreen::Instance()->ProjectionMatrix().GetMatrixArray());

TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("modelviewMatrix"),
           TheScreen::Instance()->ModelViewMatrix().GetMatrixArray());
```

Granted, the above code looks a little overwhelming. All this is doing is sending the matrix data to the shaders before rendering the sprite. The names *projectionMatrix* and *modelviewMatrix* are the shader variable names that will link with our *projection* and *modelview* matrices respectively.

Now for the best part – rendering the sprite :

```
m_sprite.Draw();
```

This will send the VBO data to the attached shaders and if everything is linked properly as stated above, our sprite should show on screen. Finally, we detach the shaders and re-enable the debug ones.

```
TheShader::Instance()->Detach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Detach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");
TheDebug::Instance()->Enable();
```

Note : Again, if you don't need the debug shaders, you are free to only use the main game shaders, or write your own.

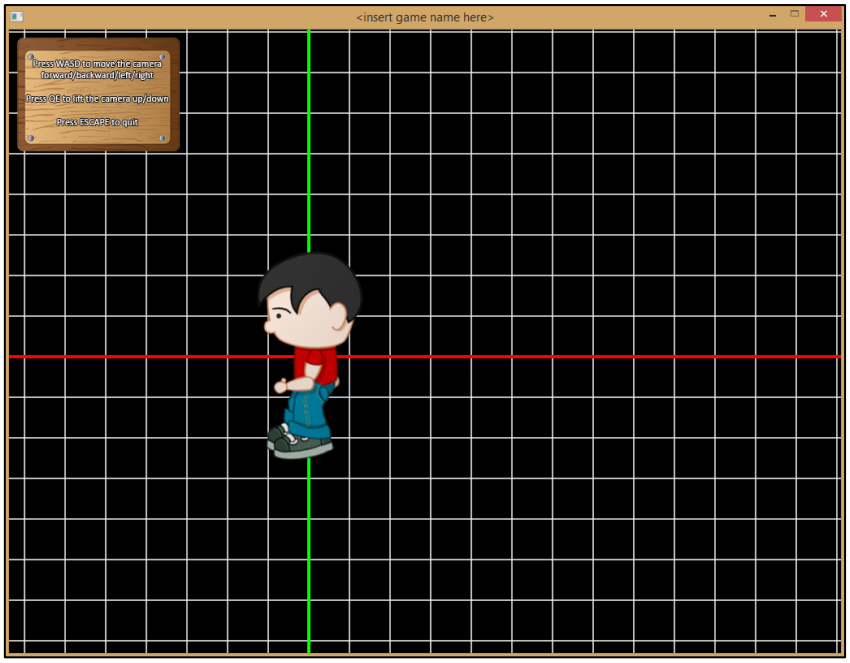
Let's see how this all looks in the *Player.cpp* source file :

Add this to the Player Constructor :

```
//create a vertex, color and texture VBO for the sprite
TheBuffer::Instance()->Create(BufferManager::VERTEX_BUFFER, "PLAYER_VERTEX_BUFFER");
TheBuffer::Instance()->Create(BufferManager::COLOR_BUFFER, "PLAYER_COLOR_BUFFER");
TheBuffer::Instance()->Create(BufferManager::TEXTURE_BUFFER, "PLAYER_TEXTURE_BUFFER");

//load player sprite image from file
TheTexture::Instance()->LoadFromFile("Sprites\\Player.png", "PLAYER_TEXTURE");

//set dimension, texture and buffer properties of sprite object
m_sprite.SetSpriteDimension(180, 300);
m_sprite.SetTextureID("PLAYER_TEXTURE");
m_sprite.SetBufferID("PLAYER_VERTEX_BUFFER", "PLAYER_COLOR_BUFFER", "PLAYER_TEXTURE_BUFFER");
```

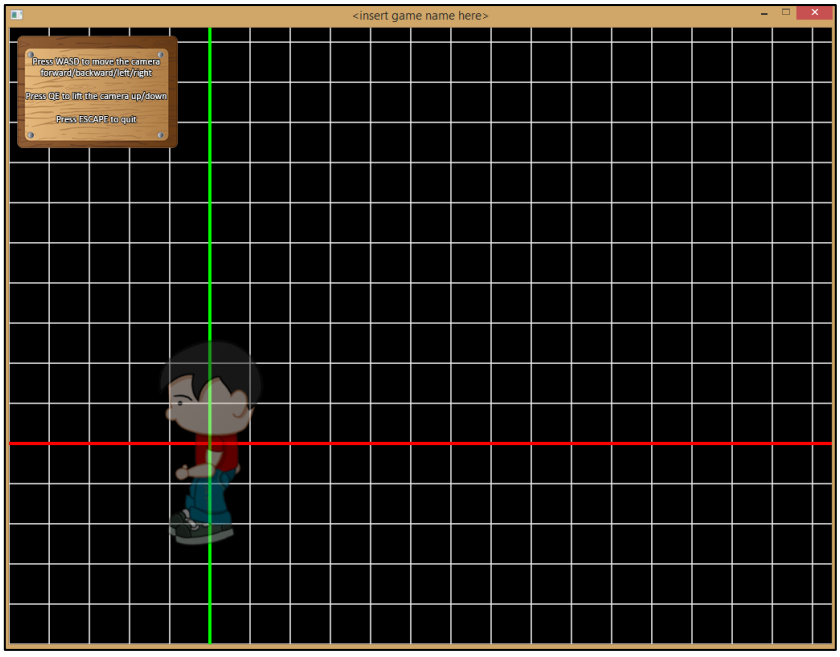
Add this to the Draw() function :

```
//temporarily disable debug shaders
TheDebug::Instance()->Disable();

//temporarily attach and link main program shaders
TheShader::Instance()->Attach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Attach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");
TheShader::Instance()->Link();

//link shader attribute variables to sprite object
m_sprite.SetShaderAttribute("vertexIn", "colorIn", "textureIn");

//send all matrix data to shaders
TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("projectionMatrix"),
           TheScreen::Instance()->ProjectionMatrix().GetMatrixArray());

TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("modelviewMatrix"),
           TheScreen::Instance()->ModelViewMatrix().GetMatrixArray());

//draw player image
m_sprite.Draw();

//detach main program shaders
TheShader::Instance()->Detach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Detach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");

//re-enable debug shaders
TheDebug::Instance()->Enable();
```

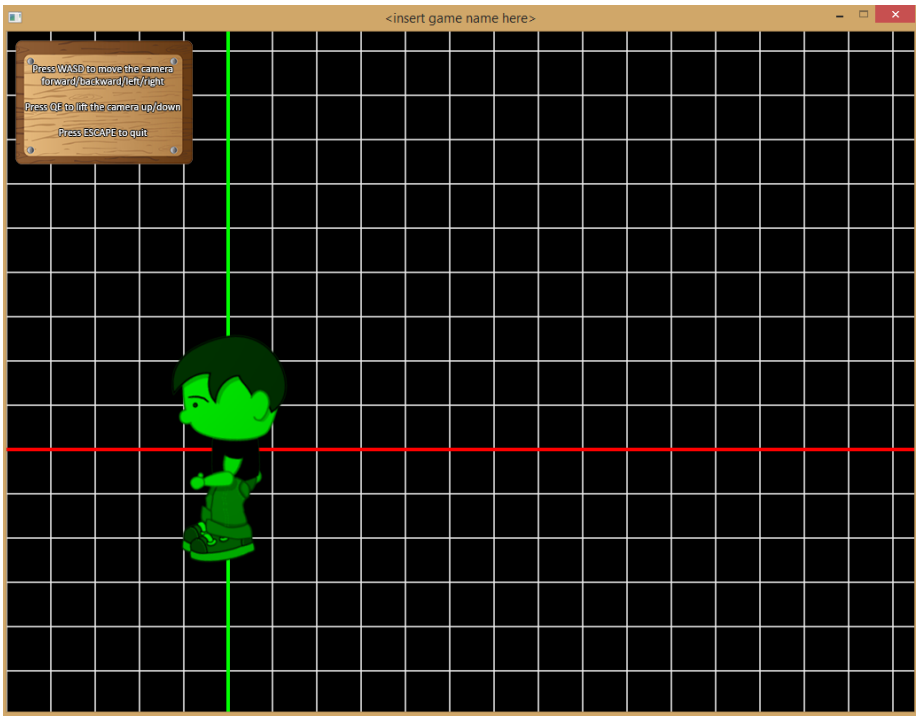


After all that we finally have our player image on the game screen. Granted it's quite a procedure to get something rendered, but that's the beauty of writing your own code to make things happen.
Go ahead and use the *Transform* component to position, rotate or even scale the sprite. You can also change the color values and alpha channel to make the player look a little more funky. Let's try that out below :

```
m_sprite.SpriteColor().A = 0.5f;
```



```
m_sprite.SpriteColor() = Color::GREEN;
```



And last but not least, make sure you clean up after yourself by destroying the VBOs and unloading the sprite image from memory. Make sure the code below goes in the *Player* class' destructor :

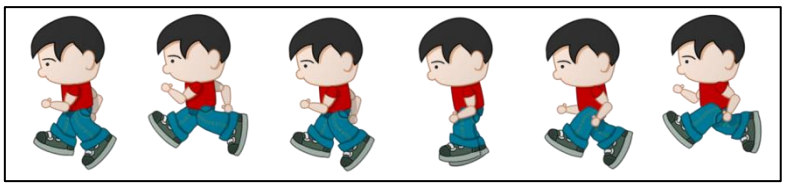
```
//destroy VBOs from memory
TheBuffer::Instance()->
Destroy(BufferManager::VERTEX_BUFFER, BufferManager::CUSTOM_BUFFER, "PLAYER_VERTEX_BUFFER");

TheBuffer::Instance()->
Destroy(BufferManager::COLOR_BUFFER, BufferManager::CUSTOM_BUFFER, "PLAYER_COLOR_BUFFER");

TheBuffer::Instance()->
Destroy(BufferManager::TEXTURE_BUFFER, BufferManager::CUSTOM_BUFFER, "PLAYER_TEXTURE_BUFFER");

//remove player sprite image from memory
TheTexture::Instance()->UnloadFromMemory(TextureManager::CUSTOM_TEXTURE, "PLAYER_TEXTURE");
```

The Animation Component

This component extends the functionality of the regular *Sprite* component in that it offers the sprite object to loop through an array of images, better known as a *sprite sheet*, and render each frame, thereby creating an animation effect. Let's see this cool feature in action.
First, we need to declare a *Animation* component in the *Player* header file :

```
Animation m_animation;
```

Then, we are going to add the following sprite sheet, called *PlayerRun.png* into the *Sprites* folder as shown in the previous example.

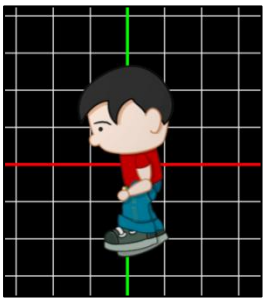
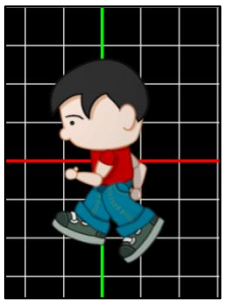
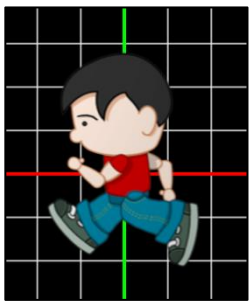


The entire sprite sheet is **1350x300** pixels in dimension, but the animation component only needs to know the dimension of each singular sprite which is **225x300**. We also need to set the animation object's texture dimension, which states how many columns and rows are in the sprite sheet. This will help it calculate how to cut out the images. In our case it's **6** columns x **1** row. And lastly, we want the speed to be set to a specific amount, like say **5**.
So in the constructor, after creating the VBOs and loading the texture as we did in the sprite example, we should add the following code :

```
m_animation.SetSpriteDimension(225, 300);
m_animation.SetTextureDimension(6, 1);
m_animation.SetAnimationVelocity(5.0f);
m_animation.SetTextureID("PLAYER_TEXTURE");
m_animation.SetBufferID("PLAYER_VERTEX_BUFFER", "PLAYER_COLOR_BUFFER", "PLAYER_TEXTURE_BUFFER");
```

Everything else we did with the *Sprite* earlier remains the same; the only difference is that we are using a *m_animation* variable instead of a *m_sprite.*

Our sprite now loops through the sprite sheet and cuts out each image individually.

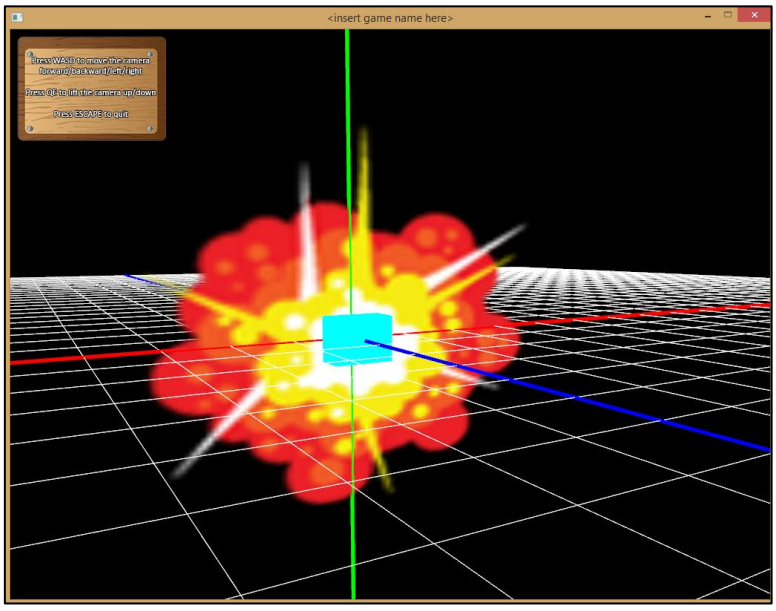




You can also set the animation to loop once only, meaning that if the last image in the sprite sheet is reached, the image stops rendering. This is good for something like an explosion. Our player however may want to loop for a while because it looks like he is running somewhere. To set the looping you will need the following command :

```
m_animation.IsAnimationLooping() = false;
```

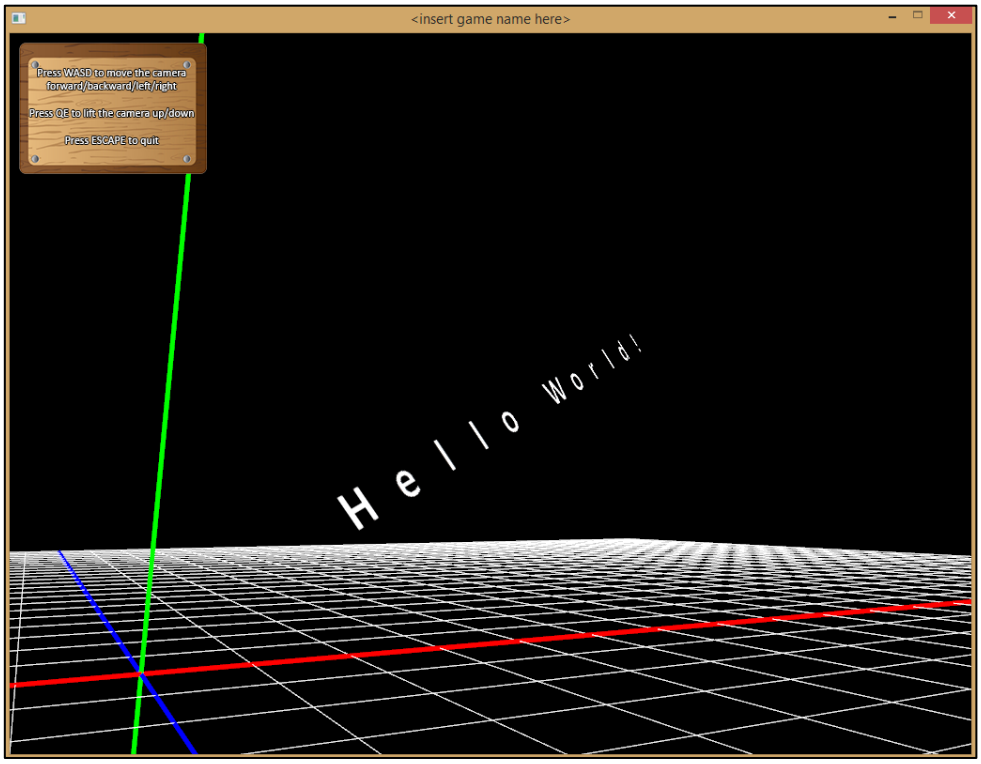
The above statement will loop once through the sprite sheet and then stop animating when the last frame is reached.

Remember, although sprites are 2D objects, they can still be used in a 3D environment. The only thing to consider is that the sprite's dimension is measured in units instead of pixels. So even if the sprite is say **100x100** pixels, you may want it to only be **1x1** units.



The Text Component

This is used for 2D and 3D text display. It can be used to display 2D text in a 2D and 3D environment. There will be more detailed documentation for this soon!

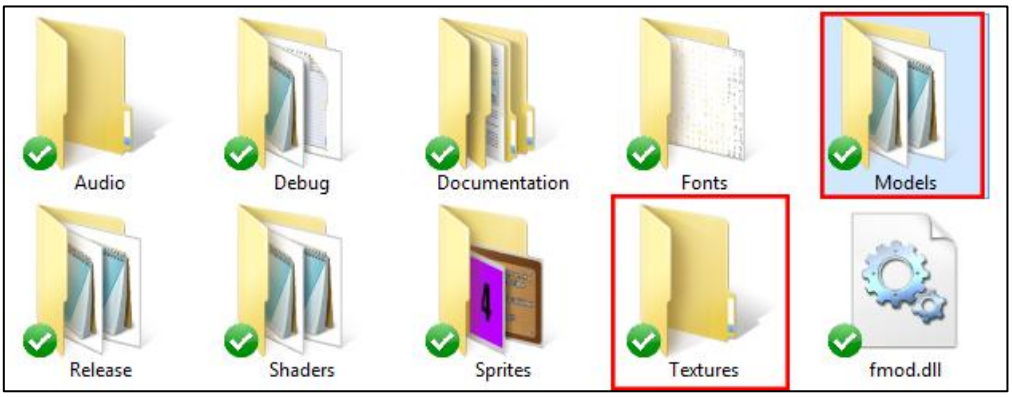


The Model Component

If you prefer to work with 3D objects, and most likely you will, then there is a component for that as well. It currently only supports *OBJ* and *MTL* files which can easily be produced via third party 3D modelling applications. Let's delve straight in and see what we can produce. We will start a fresh new class called *Camera*, which will render a, you guessed it – camera! It should contain a *Model* component as well.

```
class Camera3D : public GameObject
{

public:

	Camera3D();
	virtual ~Camera3D();

public:

	virtual void Update();
	virtual bool Draw();

private:

	Model m_model;

};
```

Now let's add the model files into the *Models* folder. We will also add the model's texture file into the *Textures* folder. In our case we have a *Camera.obj* and a *Camera.mtl* model file, and a *Camera.jpg* texture file.



Now, just as we did in the *Sprite* demo, we need to create our VBOs and load in the model's texture from memory. Place this code in the *Camera* class' constructor as well :

```
//create a vertex, color and texture VBO for the camera model
TheBuffer::Instance()->Create(BufferManager::VERTEX_BUFFER, "CAMERA_VERTEX_BUFFER");
TheBuffer::Instance()->Create(BufferManager::COLOR_BUFFER, "CAMERA_COLOR_BUFFER");
TheBuffer::Instance()->Create(BufferManager::TEXTURE_BUFFER, "CAMERA_TEXTURE_BUFFER");

//load camera model texture from file
TheTexture::Instance()->LoadFromFile("Textures\\Camera.jpg", "CAMERA_TEXTURE");
```

You will notice nothing much is different here compared to the *Sprite* demo, except maybe the texture filename and the names we give the VBOs.

Now we need to link the texture and VBOs with the model object like so :

```
m_model.SetTextureID("CAMERA_TEXTURE");
m_model.SetBufferID("CAMERA_VERTEX_BUFFER", "CAMERA_COLOR_BUFFER", "CAMERA_TEXTURE_BUFFER", "");
```

Note : Looking at the second statement, the last argument is "". This is because what should go here is a link to a VBO containing all the data for the model's normals. This has not been implemented yet, and as soon as lighting has been integrated into the engine, then this argument will become valid.

Now that the VBOs are linked to the model component, we can go ahead and load the model's vertex, color and texture coordinate data from the *OBJ* and *MTL* files we placed in the *Models* folder earlier.

```
m_model.LoadFromFile("Models\\Camera.obj", "Models\\Camera.mtl");
```

*Note : Make sure you load the model data **after** you created and linked the VBOs to the model. Otherwise you will get a runtime error.*

Note : Again all the above code should go inside the Camera class' constructor, as this is nothing more than initialization code and should only be done once upon the creation of the camera object.

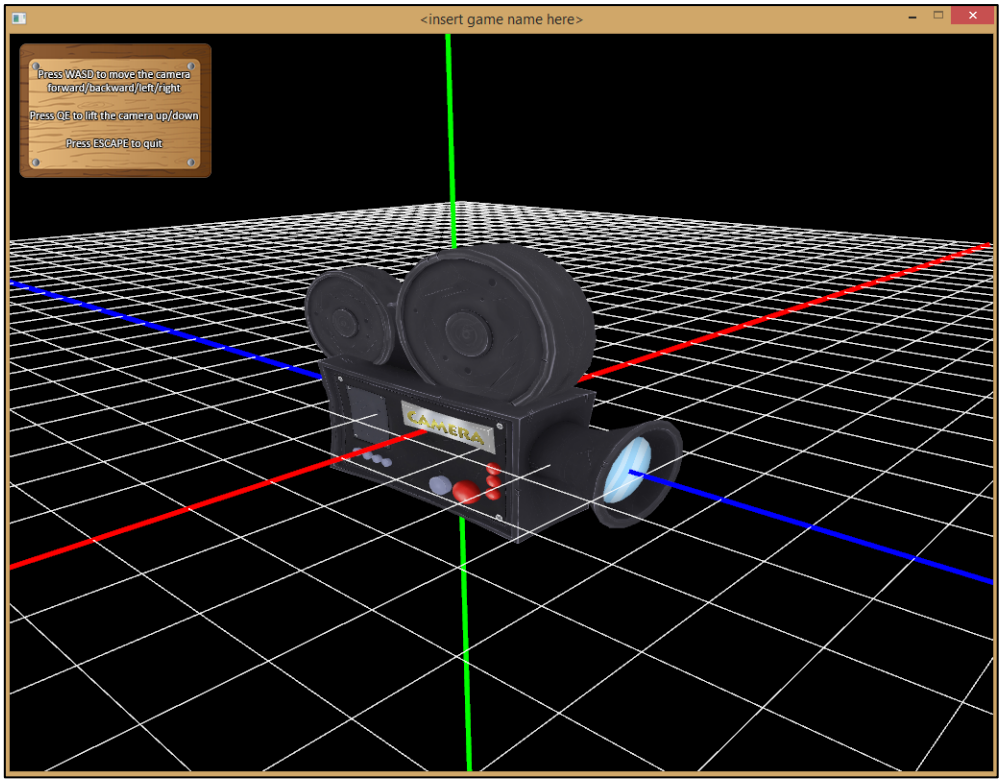
Now to actually render the model is easy, as the code is exactly the same as the draw code was for the *Sprite* example. The only significant difference is that when linking the shader variables with the model component, we use a “” value for the last argument :

```
m_model.SetShaderAttribute("vertexIn", "colorIn", "textureIn", "");
```

Again, this is because right now there is no lighting support in the shaders, so therefore no normals are being used and that’s why the last argument is left blank.

So, in all the *Camera* class’ *Draw()* routine should look like this :

```
//temporarily disable debug shaders
TheDebug::Instance()->Disable();

//temporarily attach and link main program shaders
TheShader::Instance()->Attach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Attach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");
TheShader::Instance()->Link();

//link shader attribute variables to camera model
m_model.SetShaderAttribute("vertexIn", "colorIn", "textureIn", "");

//send all matrix data to shaders
TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("projectionMatrix"),
           TheScreen::Instance()->ProjectionMatrix().GetMatrixArray());

TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("modelviewMatrix"),
           TheScreen::Instance()->ModelViewMatrix().GetMatrixArray());

//draw camera model
m_model.Draw();

//detach main program shaders
TheShader::Instance()->Detach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Detach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");

//re-enable debug shaders
TheDebug::Instance()->Enable();

return true;
```



And there we have it – a camera in our scene.

Make sure you create the camera object in the *MainState* or else you might not have anything appearing on screen :

```
m_gameObjects.push_back(new Camera3D());
```

And don't forget to clean up after yourself. Make sure the following cleanup code is placed in the *Camera* class' destructor :

```
//destroy VBOs from memory
TheBuffer::Instance()->
Destroy(BufferManager::VERTEX_BUFFER, BufferManager::CUSTOM_BUFFER, "CAMERA_VERTEX_BUFFER");

TheBuffer::Instance()->
Destroy(BufferManager::COLOR_BUFFER, BufferManager::CUSTOM_BUFFER, "CAMERA_COLOR_BUFFER");

TheBuffer::Instance()->
Destroy(BufferManager::TEXTURE_BUFFER, BufferManager::CUSTOM_BUFFER, "CAMERA_TEXTURE_BUFFER");

//remove camera model texture from memory
TheTexture::Instance()->UnloadFromMemory(TextureManager::CUSTOM_TEXTURE, "CAMERA_TEXTURE");
```

And there we have it – support for 3D models as well that can also be transformed in any way you wish. Play around with a few options if you like.

The Physics Components

If you want to add 2D and 3D collision, as well as basic physics capabilities then these are the components to use. The physics components are also meant to be placed inside your game objects, just like any other component, and allow for the objects to detect collision with each other, as well as react realistically to an array of physics capabilities.

Physics

The *Physics* class allows for your game object to behave like a *rigid body,* and to apply things like *forces, mass*, *acceleration, torque,* etc to the object. It also encompasses *velocity, angles, gravity* and other parts to enable you to make your game object respond to its environment and behave in a way so as to simulate basic physics. The class itself is not perfect, but it offers a starting point of some kind. Feel free to make changes or add new features. Alternatively if you prefer to use an external library to simulate your physics, there are some good libraries out there, including *Bullet* and *Box2D*.

Get *Bullet* here : http://bulletphysics.org/wordpress/
Get *Box2D* here : http://box2d.org/

The Collision Components

The game engine features a host of collision bounds that can be used to test if game objects are intersecting, allowing them to react to those collisions. Some of the bounds included are *bounding boxes, spheres, planes, line segments* and *oriented bounding boxes*. All collision bounds are detailed in the following sub-sections.

AABB3D

Let's begin with bounding boxes. The simplest form of these is the *Axis-Aligned Bounding Box*, and for that we have a *AABB3D* class. *AABBs* are one of the simplest forms of collision detection, however they are quite restricted in that they don't rotate, making them only useful if the game objects they are testing for collisions with are on the same aligned axis.
We will continue using our above mentioned *Camera* class, but now we will create two of them, each with a *AABB3D* component inside. Our bounding boxes will essentially surround the camera model we used before, and check for collision.

```
class Camera1 : public GameObject
{

public:

	Camera1();
	virtual ~Camera1();

public:

	virtual void Update();
	virtual bool Draw();

private:

	Model m_model;
	AABB3D m_boxBound;
};

class Camera2 : public GameObject
{

public:

	Camera2();
	virtual ~Camera2();

public:

	virtual void Update();
	virtual bool Draw();

private:

	Model m_model;
	AABB3D m_boxBound;
};
```

Inside the constructor of each class we will add some initialization code that will set the properties of the *axis-aligned bounding box*. The main property we want to set is the box's dimensions in *width*, *height* and *depth* :

```
m_bound.SetDimension(2.0f, 6.5f, 6.0f);
```

Note : Ideally, the dimension of the bounding box should be roughly the same dimension as the camera model, so that the whole object is considered when checking for intersections.

Now that our camera and its bound are set up in dimensions, we can position the bound accordingly. If the camera is static and does not move around the scene, its position can be set once only inside the constructor like so :

```
m_bound.SetPosition(1, 2, -3);
```

If the camera is dynamic and will move around the scene frequently, then it would be best to always update the bound's position from inside the game object's *Update()* function.

```
m_bound.SetPosition(pos.X, pos.Y, pos.Z);
```

Note : Looking at the above code snippet we can see that a positional x, y and a z value are used to update the position. These three values would represent the x, y and z position of the camera, based on how it is moving around in the scene. The camera's class would have to maintain these values accordingly. See the ***Transform Component*** *section of this manual for more info on how to change positions of a game object.*

Whether the camera is static or dynamic, the very next function call should be the bound's *Update()* function :

```
m_bound.SetPosition(pos.X, pos.Y, pos.Z);
m_bound.Update();
```

The *Update()* function of the bound needs to be called at least once, whether the bound is static or dynamic, so that collision calculations remain accurate.

An example snippet of what the first camera class' *Update()* routine could look like is below :

```
void Camera1::Update()
{
	//update position of camera
	//...

	//update position of bound based on camera's position
	//so that all AABB collision calculations are accurate
	m_bound.SetPosition(pos.X, pos.Y, pos.Z);
	m_bound.Update();
}
```

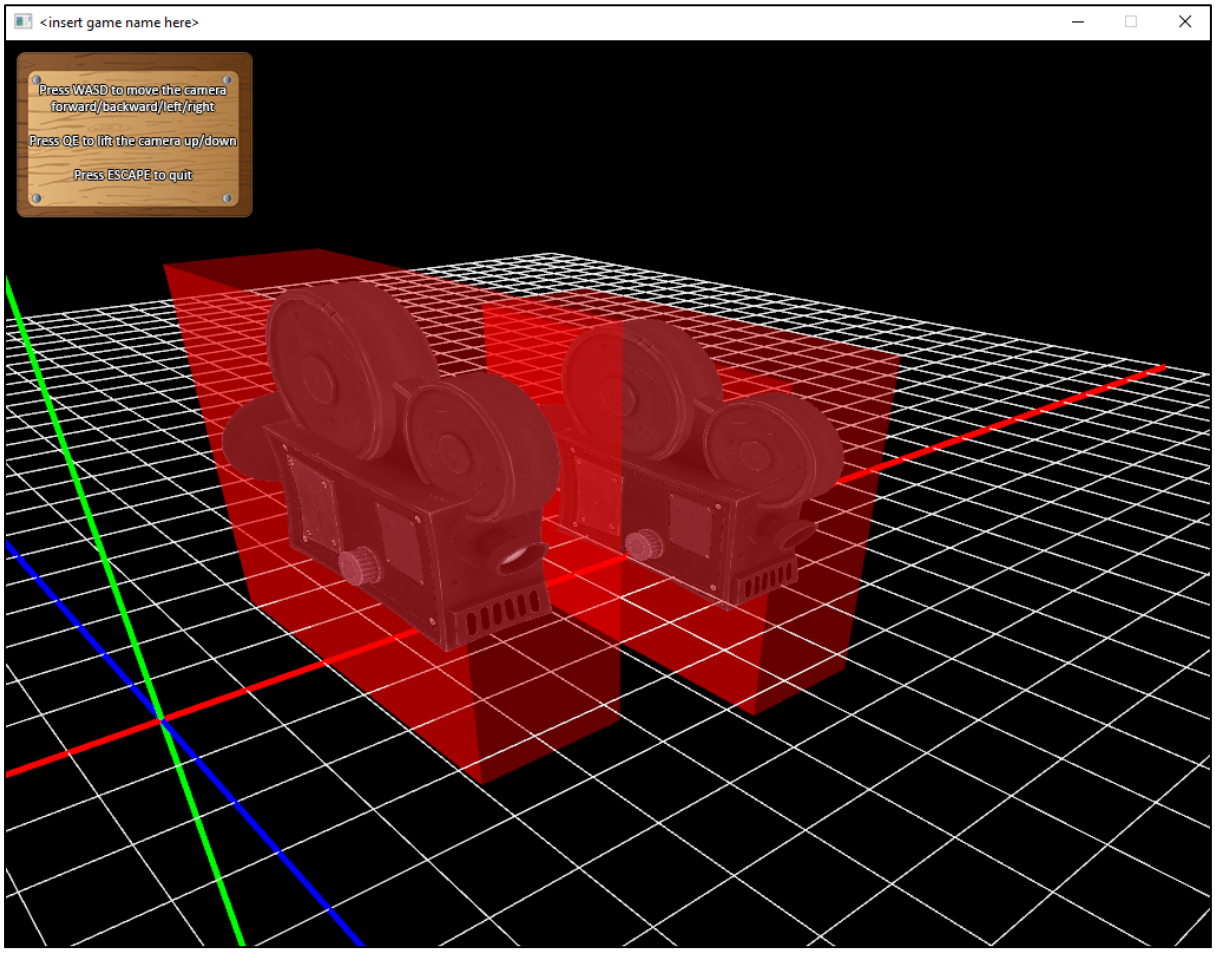
Now it's time to render everything, so in the camera class' *Draw()* function we are going to add some code that will render the *AABB*. Using the previously mentioned *Camera3D* class' exact same code, we are going to add some extras in below *(highlighted in bold)* :

```
TheDebug::Instance()->Disable();

TheShader::Instance()->Attach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Attach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");
TheShader::Instance()->Link();

m_model.SetShaderAttribute("vertexIn", "colorIn", "textureIn", "");

//update matrix
m_transform.GetMatrix() = Matrix4D::IDENTITY;

//position model (and bounding box) accordingly
m_transform.Translate(pos.X, pos.Y, pos.Z);

//apply transformation to matrix
TheScreen::Instance()->ModelViewMatrix() * m_transform.GetMatrix();

TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("projectionMatrix"),
           TheScreen::Instance()->ProjectionMatrix().GetMatrixArray());

TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("modelviewMatrix"),
           TheScreen::Instance()->ModelViewMatrix().GetMatrixArray());

m_model.Draw();

TheShader::Instance()->Detach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Detach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");

TheDebug::Instance()->Enable();
```

```
//draw bounds
m_bound.Draw();
```

Looking at the above code snippet, we can see that the bold parts are the new parts of the code. The *pos* variable represents the camera's position and this is used to translate the camera into position. This translation is then applied to the *Modelview* matrix before the camera model is drawn, after which the bounding box is drawn.

Note : The AABB is drawn after the main shaders have been detached and the Debug shaders re-linked. This is because the bounding box uses the Debug Manager's tools to render (just as do the other collision objects).



And there we have it – two camera objects rendered on screen with their bounding boxes surrounding them. Of course all of the code above should also be implemented in the *Camera2* class as well!

Note : The bounding boxes are drawn straight after the camera model is drawn, making use of the exact same transformations as the camera. Resetting the transformations before rendering the bounds would make little sense as the boxes need to reflect the game object's orientations. However with AABBs rotations are not considered. Therefore the camera may be rotated but make sure you reset the Modelview matrix before rendering the bounding box!

Of course nothing is actually happening, as long as no collisions are being tested. That's our next step, so first of all we need to add a *getter* in both camera classes' header files, like so :

```
class Camera1 : public GameObject
{

public:

	Camera1();
	virtual ~Camera1();

	AABB3D GetAABB3D()  { return m_boxBound; }

	virtual void Update();
	virtual bool Draw();

private:

	Model m_model;
	AABB3D m_boxBound;
};
```

The *getter* (in bold above) will return our *AABB3D* bounding box, which we will need to call the *AABB3D* class' *IsColliding()* function. So after adding the getter in both *Camera1* and *Camera2*, add the following code in the *MainState's Update()* routine, after all the other code :

```
if (camera1->GetAABB3D().IsColliding(camera2->GetAABB3D()))
{
	//there is collision – do something!
}
else
{
	//there is NO collision – no need to worry!
}
```

*Note : Make sure you have created two objects in the MainState somewhere of type Camera1 and Camera2. The pointers **camera1** and **camera2**, as seen in the code snippet above, are referencing these two camera objects respectively.*

After adding the above code, the two camera objects will use their *AABB3D* components to check for collision. More specifically, *camera1* will use its *AABB3D's IsColliding()* function to check for collision, by passing in *camera2's AABB3D* component. Either ***true*** or ***false*** will be returned, respectively, which can be used to react appropriately to the collisions.

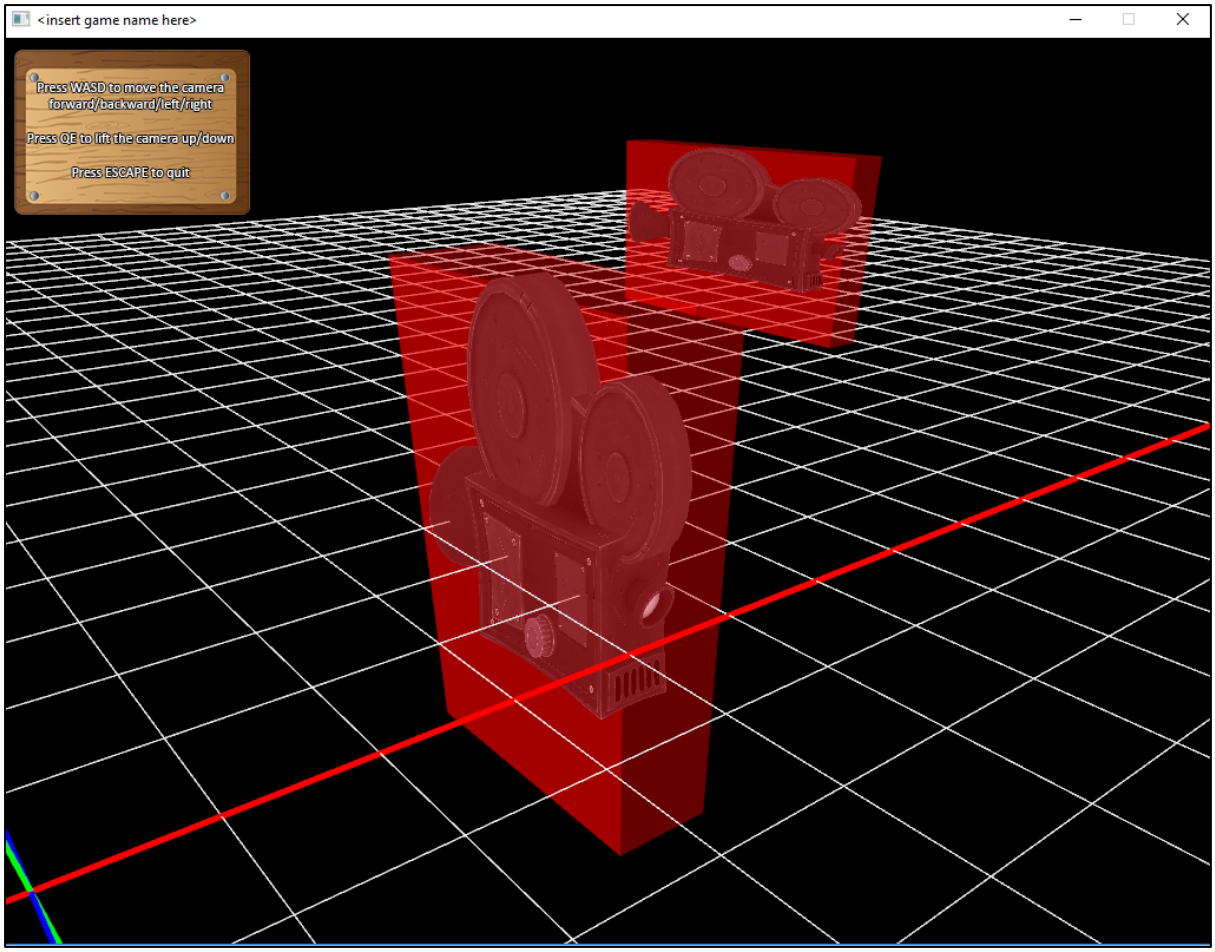
Scaling the objects to a certain size is also supported. No matter how large the objects are, the *AABB3D* bounds will adjust accordingly and calculate the intersections accordingly. So for instance, let's change the scale of the first camera to **0.1** in *x*, **0.2** in *y* and **0.3** in *z*. This can be done by adding a simple line of code inside the constructor of the *Camera1* class, like so :

```
m_bound.SetScale(0.1f, 0.2f, 0.3f);
```

This adjusts the bounding box scale value, but the game object itself still needs to be scaled too. Preferably we will want both the game object and the bounding box to synchronise. In the *Draw()* routine of the *Camera1* class, adjust the code to include scaling :

```
//update matrix
m_transform.GetMatrix() = Matrix4D::IDENTITY;

//position and scale camera (and bounding box) accordingly
m_transform.Translate(m_pos.X, m_pos.Y, m_pos.Z);
m_transform.Scale(0.1f, 0.2f, 0.3f);

//apply transformation to matrix
TheScreen::Instance()->ModelViewMatrix() * m_transform.GetMatrix();
```

Now let's do the same for the *Camera2* class, except that let's add a different scale value, say **0.5** in *x*, **1.0** in *y* and **0.5** in *z.* Afterwards, run the application again, and you should see the two bounds (and their camera game object counterparts) have been properly adjusted. All intersection calculations will adapt to the new adjustment, and the actual bounding box will also render the correct size too!



The above image shows that the bounding boxes have been scaled, and collision still works properly even with the new sizes. Of course the cameras now look a little warped and squashed, but this is just for illustration.

Note : Rotations are not supported by AABBs! This is because they need to remain in the same axis-aligned space in order to calculate properly. You may want to rotate the game object which is fine but if you render the bound straight after you render the game object it will appear rotated, so just bear that in mind.

Note : For game objects that involve rotation, consider using OBBs (Oriented Bounding Boxes) instead. These will be detailed a little later on.

Just remember that if the scale and position of the game object are dynamic and constantly changing, then the bounding box's scale and position need to be updated as well, preferably from within the game objects' *Update()* routine. Furthermore, the bound's *Update()* function needs to be called each frame so

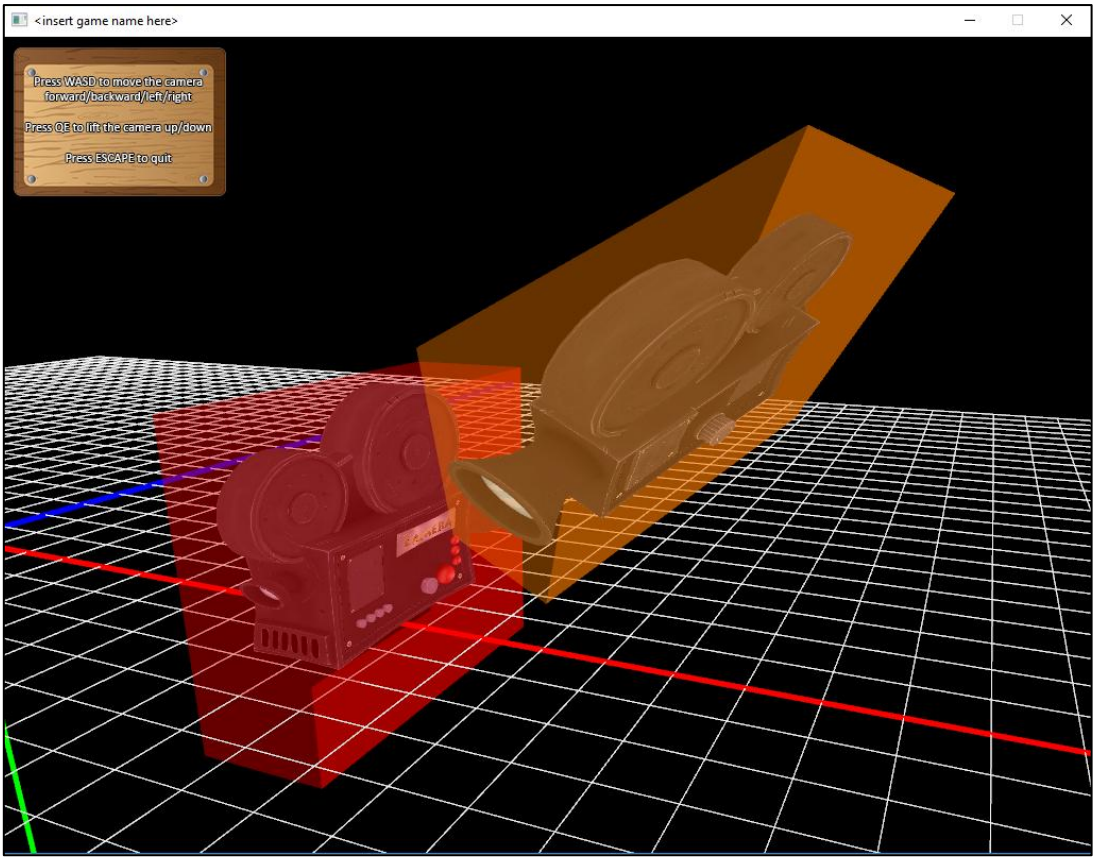
that the bound stays synchronised with the game object's properties, for example :

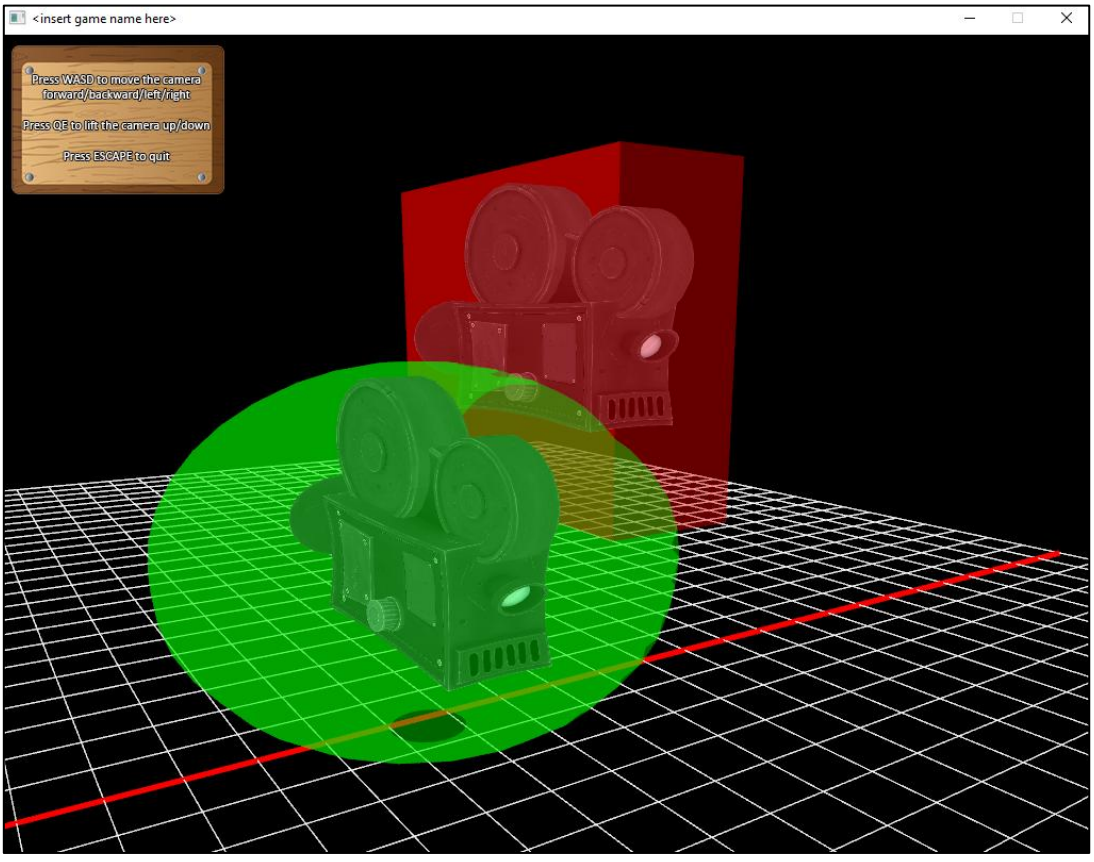
```
m_bound.SetPosition(pos.X, pos.Y, pos.Z);
m_bound.SetScale(scale.X, scale.Y, scale.Z);
m_bound.Update();
```

Note : Looking at the above code we see a pos and scale variable. These are maintained by the game object itself and sent to the bound in order for the bound to stay synchronised with the game object in terms of position and scale.

The *AABB3D* bound is not limited to checking for collisions with other AABBs only. It can also check for collisions with other *Oriented Bounding Boxes* and *Spheres*, as long as the correct bounds are set inside the second game object.

Note : Both OBBs and Spheres will be discussed in more detail over the coming sections.


<insert game name here>
Press WASD to move the camera forward/backward/left/right
Press QE to lift the camera up/down
Press ESCAPE to quit


<insert game name here>
Press WASD to move the camera forward/backward/left/right
Press QE to lift the camera up/down
Press ESCAPE to quit

AABB2D

Of course there is a 2D version to represent the *Axis-Aligned Bounding Box* as well. There isn't very much difference between this class and its 3D counterpart, except that it works mainly with objects in a two dimensional world.

Note : The AABB2D component can be used in 3D as well of course, but we will demonstrate it here as part of a 2D game object. We will make use of our Player class we used before to demonstrate sprites. All extra code will be highlighted in bold.

Note : To have the below examples work, make sure you're in 2D mode.

```
class Player : public GameObject
{

public:

	Player();
	virtual ~Player();

	AABB2D GetAABB2D()  { return m_boxBound; }

	virtual void Update();
	virtual bool Draw();

private:

	Sprite m_sprite;
	AABB2D m_boxBound;

};
```

The above code snippet uses the *Player* class but adds in a declaration for a *AABB2D* object and a *getter* to return the bound to the client code for intersection tests to actually occur.

Now, when setting the properties of the bound, and rendering it, not much will be different compared to the *AABB3D* class. Perhaps one thing to consider is when setting the dimension in the constructor. Again, ideally we want the dimension of the bounding box to be roughly the same dimension as the sprite, so that the whole sprite is considered when checking for intersections. So if we stick to our original sprite image of **180x300**, we can set both the sprite and the *AABB* bound to the same value.

```
m_sprite.SetSpriteDimension(180, 300);
m_bound.SetDimension(180, 300);
```

The position and scale of the bound work the same as the 3D version, except that they take two arguments instead of three, an *x* and a *y* value respectively.

```
m_bound.SetPosition(100, 200);
m_bound.SetScale(2.0f, 3.0f);
```

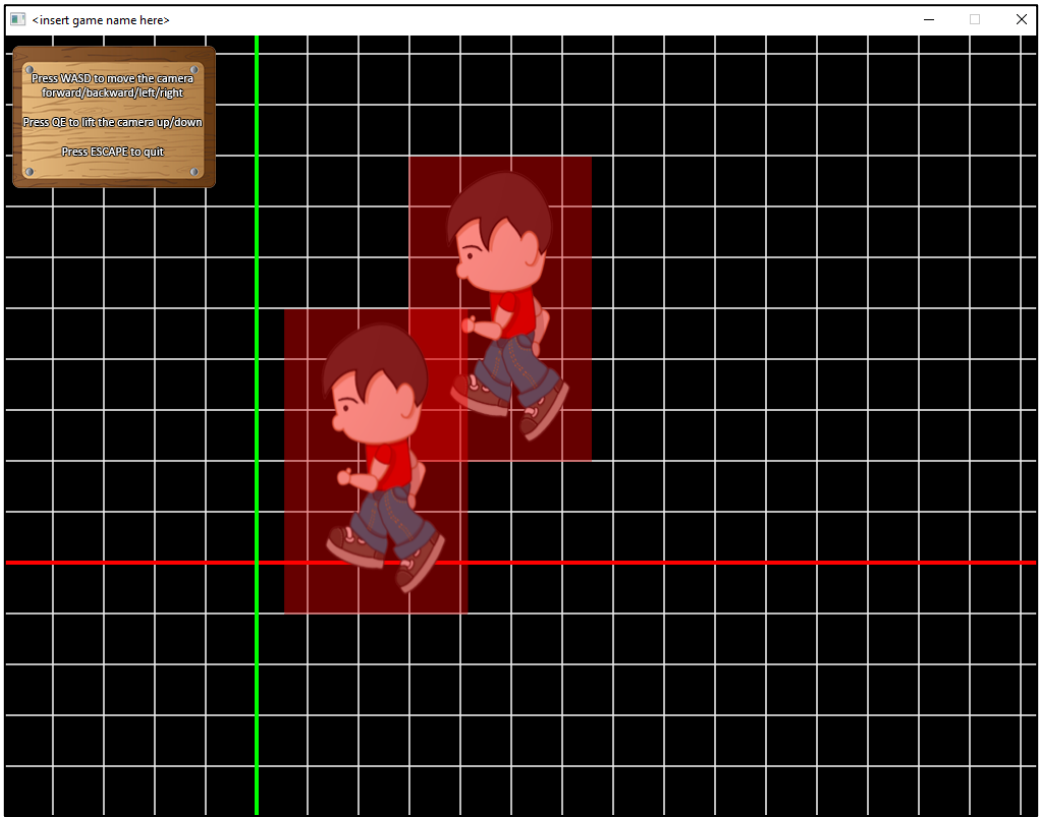
Again, based on if the sprite is static or dynamic, its properties can be set once in the constructor or each frame in the *Player* class' *Update()* function, like so :

```
m_bound.SetPosition(pos.X, pos.Y);
m_bound.SetScale(scale.X, scale.Y);
m_bound.Update();
```

Now let's add code to the *Player* class' *Draw()* function in order to position, scale and render the sprite and its bound :

```
TheDebug::Instance()->Disable();
TheShader::Instance()->Attach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Attach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");
TheShader::Instance()->Link();

m_sprite.SetShaderAttribute("vertexIn", "colorIn", "textureIn");

//update matrix
m_transform.GetMatrix() = Matrix4D::IDENTITY;

//position and scale sprite (and bounding box) accordingly
m_transform.Translate(pos.X, pos.Y);
m_transform.Translate(scale.X, scale.Y);

//apply transformation to matrix
TheScreen::Instance()->ModelViewMatrix() * m_transform.GetMatrix();

TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("projectionMatrix"),
           TheScreen::Instance()->ProjectionMatrix().GetMatrixArray());

TheShader::Instance()->
SetUniform(TheShader::Instance()->GetUniform("modelviewMatrix"),
           TheScreen::Instance()->ModelViewMatrix().GetMatrixArray());

m_sprite.Draw();

TheShader::Instance()->Detach(ShaderManager::VERTEX_SHADER, "MAIN_VERTEX_SHADER");
TheShader::Instance()->Detach(ShaderManager::FRAGMENT_SHADER, "MAIN_FRAGMENT_SHADER");
TheDebug::Instance()->Enable();

//draw bounds
m_bound.Draw();
```

Now, if all of the updated code snippets have been added to the *Player* class properly, you should be able to render our little man with a red bounding box surrounding him. Of course this makes more sense using two players to actually perform any collision. Therefore, after adding a second player, say *Player2* and duplicating the code, you should be able to see the following :



All that remains now is to perform the collision testing from inside the *MainState*, just like we did with the 3D objects, like so :

```
if (player1->GetAABB2D().IsColliding(player2->GetAABB2D()))
{
	//collision!
}

else
{
	//NO collision!
}
```

Note : The above code assumes you have created two player pointers, ***player1*** *and* ***player2*** *to reference the two player sprite objects.*

OBB3D

If you wish to use bounding boxes that include rotation, then *Oriented Bounding Boxes* may be the solution. These are similar to *AABBs* except that they have a rotational orientation so that game objects may be rotated in any way or form, and no matter how much they are rotated, their collisions calculations can still be determined.

We will once again use our *Camera* classes, just as we did in the *AABB3D* demonstration. Let us begin by adding a *OBB3D* component into our two *Camera* classes :

```
class Camera1 : public GameObject
{

public:

	Camera1();
	virtual ~Camera1();

	OBB3D GetOBB3D()  { return m_boxBound; }

	virtual void Update();
	virtual bool Draw();

private:

	Model m_model;
	OBB3D m_boxBound;
};

class Camera2 : public GameObject
{

public:

	Camera2();
	virtual ~Camera2();

	OBB3D GetOBB3D()  { return m_boxBound; }

	virtual void Update();
	virtual bool Draw();

private:

	Model m_model;
	OBB3D m_boxBound;
};
```

Note : In the two above class declarations, as you can clearly see, we have also added the two getter functions in, so that the components may be requested in the MainState for collision testing purposes.

Now for *Camera1*, we are going to set the bound's dimension and keep it the same as before. We are also going create a rotation using a quaternion, and set it to **33** degrees around the *Z* axis, which we then send to the bound to set the bound's rotation. All this code can be placed in the camera class' constructor :

```
//set dimension of bound
m_bound.SetDimension(2.0f, 6.5f, 6.0f);

//create a quaternion rotation
m_rotation.SetRotation(33, 0, 0, 1);

//set rotation of bound
m_bound.SetRotation(m_rotation);
```

We'll do something similar in *Camera2's* constructor, just that the scale will be different and the rotation will be **45** degrees around the *X* axis :

```
//set dimension of bound
m_bound.SetDimension(2.0f, 6.5f, 6.0f);

//set scale of bound
m_bound.SetScale(1.0f, 0.3f, 0.7f);

//create a quaternion rotation
m_rotation.SetRotation(45, 1, 0, 0);

//set rotation of bound
m_bound.SetRotation(m_rotation);
```

Note : The m_rotation variable belongs to the GameObject class, so as long as you derive your game objects from GameObject you will have access to this variable. By setting this variable we have now set the rotation of our actual game object and used it to set the rotation of the game object's bound.

Note : For more information about quaternions, please refer to the Quaternions section of the manual

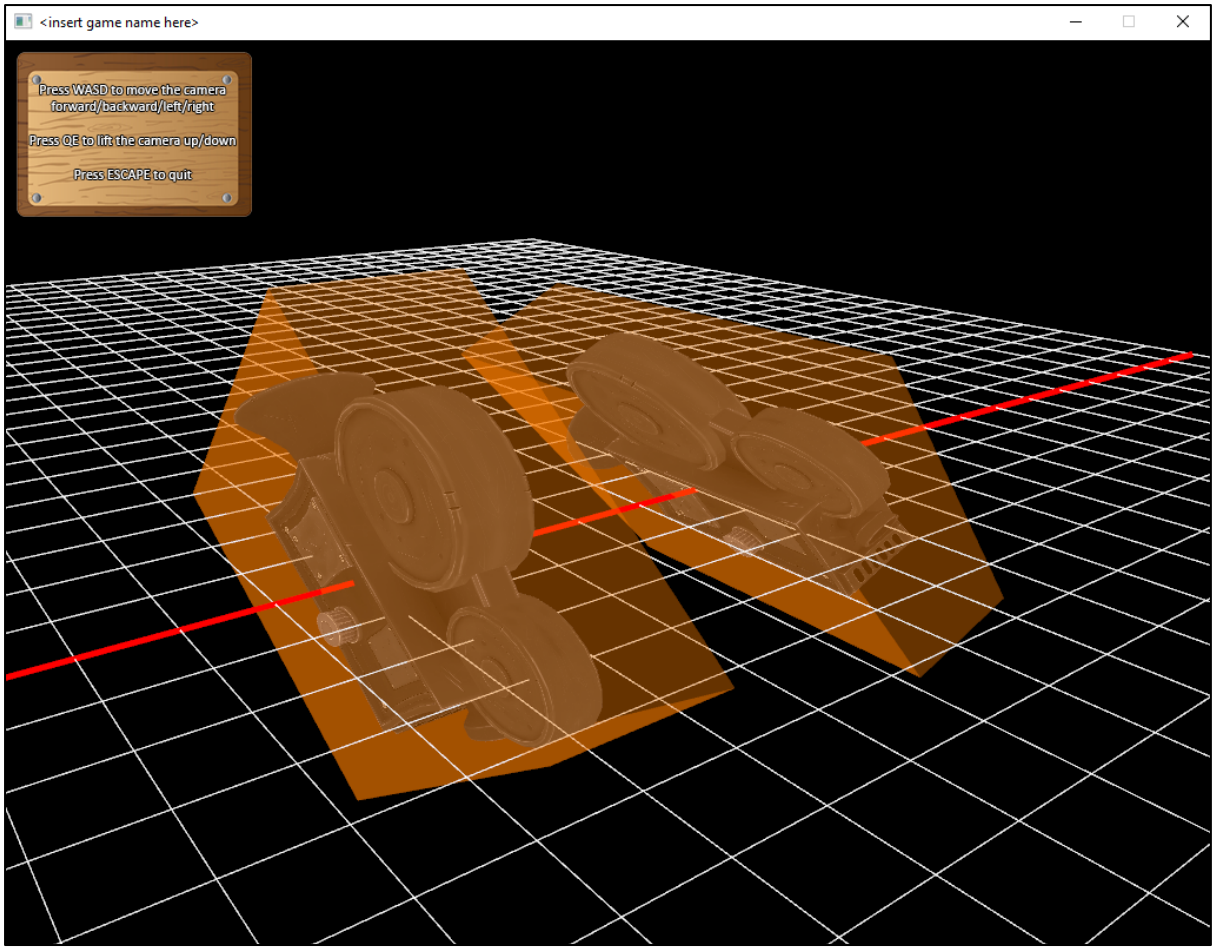
Now in both the *Update()* and *Draw()* functions of both *Camera1* and *Camera2* we want to update the bounds and render them accordingly. The code below demonstrates *Camera1's* functionality, so just duplicate it and do the same for *Camera2* :

```
void Camera1::Update()
{
	//update camera
	//...

	//update bound of camera
	m_bound.Update();
}

bool Camera1::Draw()
{

	//all other rendering code here
	//...

	//update matrix
	m_transform.GetMatrix() = Matrix4D::IDENTITY;

	//position camera (and bounding box) accordingly
	m_transform.Translate(pos.X, pos.Y, pos.Z);

	//apply transformation to matrix
	TheScreen::Instance()->ModelViewMatrix() * m_transform.GetMatrix()
	                                         * m_rotation.GetMatrix();

	//all other rendering code here
	//...

}
```

Note : For Camera2 you may want to add scale just after translation so that scale is considered before rendering.

As you can see above the *Modelview* matrix is multiplied by the *m_transform* and *m_rotation* matrices, so that all translation, scale and rotations are considered respectively.

Now, when you run the application, you should see both your cameras on screen, one rotated **33** degrees around *Z* and the other **45** degrees around *X*.



Now you can use the OBBs just like we did in the above AABB examples when it comes to collision testing. As long as the bounding boxes are updated they will calculate for intersections accurately. Again if the bounds are dynamic, then setting the position, scale and rotations would be done in the camera classes' *Update()* functions.

OBBs are not limited to checking for collisions with other OBBs only, they can also check for intersections with other AABB bounds too.

OBB2D

There is also a 2D version of the OBB class, which behaves exactly the same as its 3D counterpart, bar from a few differences here and there. One of the main differences is that when setting the rotation, instead of using a Quaternion a regular Transform object is used.

Note : More demonstrations and examples for other collision bounds will be added to the manual at a later stage. I just haven't found much time to complete them all now.

The Networking Components

There is currently basic support for networking capabilities, and the underlying *SDL_Net* API is used to connect between computers, and create a Server/Client based networking system. However the current *Server* and *Client* classes that come with the engine are still at a very basic level. More support and documentation for this will be coming soon!

Contact

And finally if everything mentioned above made no sense at all to you, or it did but you still have some difficulty understanding certain parts or aspects, then by all means contact me via one of the following channels below :

@KarstenCorner

djkarstenv@gmail.com

linkedin.com/in/karstenvermeulen

www.karstenvermeulen.com