|  |
| --- |
|  |

|  |
| --- |
| Handmade  An Educational Game Engine |
| Manual |
|  |

*Contents*

Introduction.........................................................................................................3

Getting Started.....................................................................................................5

The Main Source File............................................................................................8

The Shaders........................................................................................................11

The Main State...................................................................................................12

The Managers....................................................................................................13

The Debug Manager..........................................................................................15

Game Objects.....................................................................................................22

Components.......................................................................................................27

Contact...............................................................................................................74

*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 definitely 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*. Designed to teach. Made from scratch. Built by hand.

*Engine Design*

**Components**

**Audio**

**Audio**

This audio class is designed to encapsulate all the components needed to play audio files on the sound card. It contains all the necessary member variables that control the audio *panning*, *volume*, *frequency*, etc. The three main FMOD variables are there to get a handle on the audio *channel*, *channel group* and the actual audio *data*. The latter pointer references the data by requesting it from the *Audio Manager*.

The audio data can be *played*, *paused*, *stopped*, *looped*, and within the data, an arbitrary position of the audio can be found and moved to. This class acts as a *component* and should exist within a *game object*.

*There are big changes planned within this class and the Audio Manager class.*

**Core**

**Texture**

This class encapsulates a *texture manager* of sorts, without using the dreaded *Singleton* design pattern! It will load images from disk and store them as *OpenGL* integer IDs in a map, with a string reference to that texture. The images are loaded using *SDL*'s image loading function and generated through *OpenGL* to produce an ID that is needed to use that image.

Currently the images supported are **BMP**, **PNG**, **JPG**, **GIF**, **TGA**. The main routine, *Load()*, will take in a *filename* and load that file, process the raw image data, generate a *OpenGL* image in VRAM and store the *OpenGL ID* in a map. The map that stores all the texture ID is on the heap and statically referenced so that there is only one single ‘catalogue’ of texture IDs.

Textures are meant to be loaded once only and stored in the given map. If that same texture is required by another object, instead of loading it again, we simply use *SetTexture()* to set an existing texture to the object. This models around the *Flyweight* pattern a little bit. When textures are loaded or set, the *m\_ID* variable is assigned the ID of that texture.

The *Bind()* and *Unbind()* routines will bind the current texture ID with the shaders so that when rendering the objects the bound texture is used. There are also two further overloaded *Bind()* functions, which allow for a texture ID to be searched for in the map ‘catalogue’ and bound, if found. The other *Bind()* function will allow multiple textures to be bound to *OpenGL*’s *texture units*.

The *SetWrapping()* function allows the UV wrapping modes to be set for a particular texture.

Textures can also be unloaded from memory individually or in bulk.

To use this class, an object in the client code will need to create a **Texture** instance within itself, treating it like a *component*. Ideally, *game objects* will contain *Texture* objects and use them accordingly.

Lastly, a static function has been implemented to display the total amount of texture IDs currently in VRAM, together with their ID name. This function can hence be accessed anywhere in the code for debug purposes.

*Big changes are still planned for this class in due course!*

**Material**

This class is another core component and represents a material that can be applied to any *game object* or *3D model* in the game world. All of its *ambient*, *diffuse* and *specular* properties can be set accordingly, as well as the shininess value, and these are passed to the fragment shader to be processed by the lighting calculations. The *SendToShader()* function will do the heavy lifting of passing the raw material data down the graphics pipeline.

**Light**

This class encapsulates a light component in the game world. It comprises of all lighting properties that will affect the lighting calculations in the fragment shader. The light itself can also be drawn and this is for debug purposes to see where the light is in the scene.

In particular, the *Create()*, *Update()*, and *Draw()* functions are for DEBUG only and are there to create and render an actual ‘light bulb’ in the scene. Internally, these functions will create, update and render a **Primitive** vertex object on screen.

The *SendToShader()* function passes all the light properties to the fragment shader to be processed to light up the scene accordingly.

The *ambient*, *diffuse*, *specular* and *attenuation* properties can all be set accordingly. Furthermore, for each light object created, a static variable will keep track of the total amount of lights so that when passing data to the shader, the correct light ID is used and so that the shader also knows how many lights to loop through.

*Some changes are still planned for this class. Support for directional and spot lights has been added, but not properly implemented yet!*

**Camera**

This class encapsulates FPS camera movement using basic matrix transformations to represent the rotations. The camera can move in the basic six directions. It can be set to be in ‘free flow’ mode or not, which will greatly affect these movements. In other words, the camera will move forward, backward, up and down differently in ‘free flow’ mode because the X rotations or ‘tilting’ is taken into consideration. The left and right movements are always relative no matter what. Non-free flow mode means the camera is always restricted to the Y-axis or plane.

The camera has a *view matrix*, which will represent the camera view, and a *projection matrix* to store the 2D *orthographic* and 3D *perspective* projections. Both of these matrices are used in the MVP multiplication within the vertex shader. The view matrix is transformed in the *Update()* routine, and no need to manually transform it.

The other variables define all the basic properties of the camera, such as *position*, *speed*, *sensitivity*, etc. The free flow variable allows setting the free flow mode on or off.

There are two main matrix variables, one to hold the *X rotations* and one to hold the *Y rotations* of the camera. If the camera is free flowing, the total rotations are used, ie X and Y combined, making sure the camera moves forwards / backwards / up / down, taking X rotations into account. Alternatively, for a non-free flow camera, only the Y rotations are used, meaning that no matter how much the camera is rotated in X, the forwards / backwards / up / down movements remains on the global horizontal Y plane.

The **m\_moveDirection** variable determines which direction the camera moves in and the **m\_lookAt** determines where the camera is looking. Both these properties are not always going to be the same. There is also a **m\_threshold** variable which determines the min and max allowance value that the camera can tilt on its X axis, so that it doesn't look upwards or downwards beyond its limits.

The *SetLookAt()* function sets the initial direction the camera is looking in. By default, this is always down the negative Z-axis, but for multiple cameras, each one might initially be looking in different directions, and these initial directions will be transformed accordingly.

The main functionality of the camera happens in the *Update()* routine, which will orientate and position the camera in the game world, using the correct camera rotations, and applies the camera orientation and positioning to the internal *view matrix*, to correctly set up the view.

The movement functions all set the movement vectors for the camera which are later transformed by the matrix transformations so that the camera actually moves in the right direction based on where it's looking. The two rotation functions will rotate the X and Y axis of the camera, making sure that the camera's X-axis is local and its Y-axis is global. Using this functionality, the six movements will always work correctly and accordingly.

The *Camera* class supports *orthographic* and *perspective* views. In 2D mode, the depth buffer is disabled and the camera only looks down the negative Z-axis. In 3D mode, the camera moves in 3D space using 3D movement vectors and can adjust to rotation as well. More calculations occur in *Update()* when the camera is in *3D perspective view*.

The *SendToShader()* function passes the *view matrix* and the *camera position* to the shaders for use there. The projection matrix is passed to the shaders via the individual setters, because they only need to be sent when the projection changes and not each frame, when the camera moves or rotates.

The camera can also have its *field of view* changed so as to apply the zooming in and out functionality.

The *Camera* class is intended to be used as a *component* within a *game object* class. For example, you could create a **MainCamera** or **FPSCam** class, with this class declared within.

**Buffer**

This class is designed to contain the *VAO*, *vertex*, *color*, *normal* and *texture* *VBO*s, and *EBO* IDs all in one place. Ideally every *game object* or any generic object in the client code can use this class as a *component* to store the buffer IDs. It is through these IDs that the *OpenGL* buffers in VRAM are used to store all vertex, color, etc data for each object, 3D model, etc in the scene.

Buffers are meant to be created once only and stored in the given map. If that same buffer is required by another object, instead of creating it again, we simply use *SetBuffers()* to set an existing buffer to the object. This models around the *Flyweight* pattern. When buffers are created or set, the *m\_ID* variable is assigned the IDs of that buffer.

There are functions to create the buffers, bind them together and fill them with data. Buffers may also be appended so that existing VBOs have data added on to existing data within, instead of re-fill and replace.

Lastly, a static function called *Output()* has been implemented to display the total amount of buffer IDs currently in VRAM, together with their ID name. This function can hence be accessed anywhere in the code for debug purposes.

**Visual**

**Component**

This base class is designed to be the base class to a variety of visual components such as geometric shapes, 3D models or 2D and 3D text. It contains all the common functionality such as *translation*, *rotation* and *scale*, so that all child objects, when created, can be transformed accordingly. It also contains a Buffer object, so that all child objects can store and manage their own vertices, colors, textures, etc.

Ideally, all visual component objects can be created alone-standing, or as a *component within a game object*. If on their own, they can be transformed via the transformation functions within this base class. If used within a game object, that game object’s class transformations can be combined with those of the component class.

The class has two matrices, one for all common *translation*, *rotation* and *scale* transformations. The other matrix is for storing a *transposed 3x3 matrix* used for lighting. All components can be lit or textured, or just rendered in plain colors. The geometric components are especially useful for drawing simple shapes on screen, helpful for mathematical visualizations.

The class has no *Update()* routine, as this should be done in the game object class that will contain this class. The *Destroy()* function is not overridden because there is only one specific thing it does which can be done here in the base class. Overall, this class and all child classes do not make use of *polymorphism* as it is not necessary. Each *Draw()* and *Create()* function is different in every child class.

The *SetBuffers()* function is called when you wish to share existing buffers. For example, if you create a **Sphere** component and then a second one, if they are identical in shape, size and color, you can simply re-use an existing buffer. This models around the *Flyweight* pattern.

The *SendToShader()* function will pass both matrices and flags to the shaders so that they can be processed when rendering a particular component. This needs to be called from within every game object that uses the component, or, if alone-standing, called before drawing the component.

**Model**

This class represents a 3D model that can be loaded from an **OBJ** file and then rendered on screen. Any 3rd party 3D modelling application can be used to produce an **OBJ** file, which will be read in by the class and all values stored accordingly. The OBJ file should store all the raw model vertex, texture coordinate and normal data, which will be read in, filtered and stored in the class' properties variables.

The setter functions will set the specific properties of the model. The *SetBuffer()* function is able to link the model's internal buffer with the correct vertex, color and texture VBO.

All the raw model data loading occurs in the *Load()* routine, where each line of text in the **OBJ** file is read and filtered. Based on what tokens precede the lines of text, the function will convert and store all the values accordingly. The raw data after being sorted is then stored in the internal buffer's data vectors before being sent to the respective VBOs to be filled.

The model class is intended to be used within a game object class or can be used on its own. If the game object also has a texture or material, then those classes are to be separately declared. No need to add the **Texture** and **Material** class within the **Model** class – the **Model** class only pertains to the mesh of the object.

*Current support for MTL files is temporarily halted and this model class only loads in the OBJ files. There are many improvements still being done to this class.*

**Text**

This class encapsulates text objects and is a component. It will represent a single character or a string of text to be used to display ASCII formatted text. These **Text** objects use *Bitmap Font Builder* ‘font sprite sheets’ that are specifically laid out so that the ASCII values of the letters and numbers in them correspond to the texture index values of each of the Text object's string characters. This means that a string of text can easily be ‘cut out’ of the sprite sheet by using the text's ASCII values. Each character in the string text can be spaced out as well.

There is a local version of *Create()* because the Text class will generate its own set of VBOs, by first creating some empty ones. This is because we never know at the start how large the buffers should be as text is very dynamic. There will be multiple sprites for each word, ie one sprite for every letter in the text string. This routine does most of the workload in that it fills all the VBOs with the vertices, colors and textures based on the current text string. This is ideally only called once to prevent cost, but if the text is constantly changing (like a score) then it needs to be called each time. The Create() function will also now create the exact amount of quads needed for the text so as not to waste buffer space.

If you wish to change the text, you need to re-create the text object. If you change the color or dimension, you will merely loop through the existing VBOs and re-set them. Changing the texture or font style is handled externally using a separate **Texture** class. Ideally, the Text and Texture classes should be in a game object of sorts.

There is a *m\_spacing* variable that sets the space between the individual letters. This cannot be changed after text is created. When changing the dimension of the text, the spacing value is taken into account.

Whereas the **Quad** class only creates a single quad object that is textured and colored, the Text class creates multiple quads. It could have been designed such that the Text class contains a *vector of Quad* *objects*, but then you have an issue of too many different VBOs and buffers used for each single letter in the string text.

Unlike the **Quad** class, you don't need a *SetTextureDimension()* routine because you won’t ever need to scale/tile textures for the fonts.

*Plans to replace this method of rendering text with that of using the FreeFont library is underway and will be implemented in a future update of the engine.*

**Geometric**

**Circle**

This class encapsulates a simple geometric circle in both 2D and 3D. It uses sin and cos to generate vertex points around a middle point. The circle can be broken down into slices so that the more slices the more circular the circle appears.

The *Create()* function will create the circle for the first time. It will create all the VBOs needed to store the vertices and colors for the circle. It will then break the circle up using the slices provided and generate the outer vertices using sin/cos. This is all done using a default radius of 1 and a white color.

Only the radius and color can be altered after creating the sphere. If the user wants a different looking sphere, then they will need to call *Create()* again.

The *SetRadius()* function will change the radius of the sphere and will re-loop through the entire buffer and reload it with new vertices with the radius being calculated into the equation this time.

The *SetColor()* function will change the RGBA values of the sphere and re-loop through the entire color VBO and fill it with new colors.

If this class is used in 2D mode, make sure to use screen coordinates instead of units. This is because we measure in pixels and not units.

**Cuboid**

This class encapsulates a geometric cube or cuboid. More specifically, it's called a cuboid because its dimensions will likely change, making the width, height and depth all unequal. The cuboid will share all 8 of its vertices to create 6 faces each time.

The *Create()* function will create the cuboid for the first time. It will generate the positional vertices such that they span out from the centre of the cube.

The *SetTextureDimension()* routine will allow the texture to stretch such that one texture can be mapped multiple times around one face. The default is 1x1 but you could make it repeat, like what you would do to a wall. This function will re-fill the texture VBO accordingly.

The *SetDimension()* routine will change the width, height and depth and re-fill the vertex VBO and the *SetColor()* function will change the RBGA elements of the cube and re-fill the color VBO.

The sizing of the cuboid’s dimensions could have been done with a simple scale matrix, but we don’t want to set the scale matrix each time we set the dimensions. Instead, the individual vertices are stored and a separate scale can be done externally.

**Primitive**

This class encapsulates a raw primitive geometric shape, such as a *line*, *vector* or *vertex*. Because of their simplicity, all three types have been wrapped into one single class. This class is used for both 2D and 3D modes.

The big difference between the *DrawVector()* and *DrawLineSegment()* functions is that the former will render a displacement from a (0, 0) origin, based on client transformations. The latter will draw a line segment from the given starting point to the given ending point. These points in space are also dependant on client transformations.

If this primitive class is used in 2D mode, make sure to use screen coordinates instead of units. This is because we measure in pixels and not units.

The *Create()* routine will create the buffer that will store the vertex and color data for the set primitives. The buffer will only be created but not filled with anything. The separate draw functions will fill the respective buffers and render the primitives. This is because the *Create()* routine cannot really do much except set the buffer up. The draw functions will determine what to fill the buffers with. Also, because these shapes are so simple, we don't need functions to set the dimension of lines and vertices.

There is also no way to set the colors of the primitives using separate color setters - this is all done via the draw calls. The same applies for the vertex and line width sizes.

We also don't want a generic *Draw()* call full of if-statements as to what to draw. That would also mean more setters to set up the buffers before rendering, which we don't have.

Alpha values are not changed - this is really not required for vertices and simple lines

The *DrawVertex()* function will draw a simple vertex point. It requires the XYZ coordinate, the color and a vertex point size. The vertex is rendered based on previous transformations.

The *DrawVector()* routine will render a vector from the origin to the destination point. The origin will depend on the latest transformation. It requires the target vertex point, the color and the line thickness.

The *DrawLineSegment()* function will draw a line between two set points. It requires the source and destination vertex points, the color and the line thickness. The line is drawn based on previous transformations.

**Quad**

This class encapsulates a quadrilateral shape and can be used in 2D and 3D modes. It supports static textures such that a single texture may take up the entire quad. It also supports animation, so that different texture 'cels' can be 'cut-out' and sampled to the quad so as to create an animation effect.

The private *GenerateUV()* function will sample the new UV coords based on the texture cel. This is called multiple times in *Draw()* if the quad is set to be animated, or just once if texture cel is already set for static textures.

This class can be used to create ‘sprites’ and will use a texture as the sprite image. This texture should ideally be pre-loaded and managed using a **Texture** component. The image can be a single image, or a ‘sprite sheet’ of multiple images, which is useful for animations. The main aim of this class is to be able to assign single images or a specific sprite sheet cel image to be used.

The setter functions will set the specific properties of the sprite. The *SetDimension()* routine will set the width and height of the quad object.

The *SetTextureDimension()* routine will allow the texture to stretch such that one texture can be mapped multiple times around one face. The default is 1x1 but you could make it repeat, like what you would do to a wall. This function will re-fill the texture VBO accordingly.

The *SetColor()* function will change the RBGA elements of the quad and re-fill the color VBO.

There are some animation setters that allow for specific animation properties to be set accordingly. The *SetAnimationCel()* sets the specific column and width of the cel image that needs to be ‘cut out’. The *SetAnimationDimension()* sets the size of the sprite sheet. So for animations, a sprite sheet could consist of 5x5 images per texture.

The *IsAnimationLooping()* function returns a bool reference variable, which makes this routine a getter/setter. This makes setting the flag in the client code a lot easier. It pertains to whether the animation runs continuously, such as a walk cycle. The *IsAnimationDead()* function can be called from the client code to determine if an animation is complete so that the game object that contains it may be destroyed. This is needed so that the game object is only destroyed once its animation components have finished rendering. For example, an asteroid game object first needs to render its explosion animation before its removed from memory.

The *Create()* routine will create and bind all the necessary VBO buffers and the Draw() function does a bit of extra work, checking whether there is animation or not. If not, the default UV coordinates (full image) will be used to texture the quad. If indeed animated, then the *Draw()* function has to perform some calculations as to what sprite cel to cut out of the texture and apply those new UV coordinates to the quad. Hence the need for the *GenerateUV()* routine. It will also check if an animation will loop or not. Sometimes an animation will loop endlessly (like a player's walk cycle) or just loop once (like an explosion). When the animation is set to loop only once, the first texture cell in the animation has to come around again before the animation is finally set to end. This is because if we were to end it at the last cell, the last cell wouldn't be drawn, so instead we let the entire sprite sheet draw before killing the animation.

If this class is used in 2D mode, make sure to use screen coordinates instead of units. This is because we measure in pixels and not units.

**Sphere**

This class encapsulates a geometrical sphere. It uses the parametric equation for a sphere and procedurally generates a particular number of vertices based on that. It will create the sphere using a certain number of segments (the longitude of the sphere) and for each segment there are slices (the latitude of the sphere). The VBO is filled with these vertices and the index buffer then uses these vertices to generate triangles to fill out the surface of the sphere.

The *Create()* routine will create the sphere for the first time. It will create all the VBOs and EBOs needed to generate the actual triangles for the edges. When creating the sphere, the user needs to pass in the segment and slice amount so that when drawing the sphere, it will always look the same. Only the radius and color can be altered after creating the sphere. If the user wants a different looking sphere, then they will need to call 'Create()' again.

The *SetRadius()* function will change the radius of the sphere and will re-loop through the entire buffer and reload it with new vertices with the radius being calculated into the equation this time. Beware of calling this function each frame if you have a sphere with many segments and slices – you might slow down your render pipeline! Consider using matrix transformations instead.

The *SetColor()* function will change the RGBA values of the sphere and re-loop through the entire color VBO and fill it with new colors.

The following site was used as a reference:

<http://mathworld.wolfram.com/Sphere.html>

**Debug**

**Axes2D**

This class encapsulates the 2 axes of a Cartesian Coordinate system. It represents the X and Y axis. The color cannot be changed - X is always RED and Y is always GREEN.

The *Create()* routine will create the coordinate system for the first time. After creating the coordinate system, you cannot change its size. You will have to re-create it again for the size to change. The function also takes in a **pixelsPerUnit** value which is used to create the axes. When the axes size is given and the axes are created, each axis will be sized by that unit size. But in 2D mode we work with pixels, therefore 1 unit is seen as 1 pixel. Hence, the PPU advances the axis with pixels taken into account. If not, each axis would advance would be 1 unit in size, ie 1 pixel and you wouldn't see it.

The *SetLineWidth()* function sets the thickness of the XY lines.

**Axes3D**

This is the 3D version of the above class and is treated separately so as to remove the use of too many different **Buffer** classes declared within, as well as too many if-else statements being used.

**Grid2D**

This class encapsulates a 2D grid, which is especially good for debugging purposes. It does not render the coordinate system - that has to be created separately.

The *Create()* routine will create the grid for the first time. After creating the grid you cannot change its size. You will have to re-create the grid for the size to change. The function also takes in a **pixelsPerUnit** value which is used to create the grid. When the grid size is given and the grid is created, each line in the grid advances by 1 unit. But in 2D mode we work with pixels, therefore 1 unit is seen as 1 pixel. Hence, the PPU advances the lines with pixels taken into account. If not, each line would advance by 1 unit in the loop, ie 1 pixel.

The *SetLineWidth()* function sets the thickness of the grid lines and *SetColor()* will sets the individual RGB elements of the line colors.

**Grid3D**

This is the 3D version of the above class and is treated separately so as to remove the use of too many different **Buffer** classes declared within, as well as too many if-else statements being used.

**Game States**

**GameState**

This abstract base class encapsulates a game state that will exist within the game. A game state can be anything like a play state, pause state, menu state, etc and will contain all the game specific code specific to that state. Every game state needs to be instantiated as a sub-class of this abstract base class.

Ideally, the instantiated game states will be created and controlled inside the Game class, which serves as the game state machine.

The main flag variable **m\_isActive** will keep track if the game state is currently active or not. As long as its active, its overridden *Update()* and *Draw()* routines will constantly be called until the state is set as not active, at which point the state will close down. As soon as the state is flagged as not alive, it will be removed by the state machine.

The *OnEnter()* routine is called when the state is activated and this function will load all resources from file, instantiate the game objects for that state and do all start up tasks for the active state. The opposing *OnExit()* function will do the opposite when the state is deactivated, ie - call all shutdown tasks, remove all game objects from memory and free all resources from memory for the given state.

**StartState**

This game state monitors the beginning state of the game. Things like menu's, start-up screens, splash screens, etc are intended to be used here. This particular state consists of two **SplashScreen** objects, which will draw two splash screen images before the Main State begins. It will wait for one splash screen image to fade in and out before the next one becomes active.

The *OnEnter()* function will create and compile the main game's vertex and fragment shaders as well. These shaders are then only destroyed in the End State's *OnExit()* function so that they remain active throughout the program.

**MainState**

This game state controls the main state of the game. It consists of a main camera for the main view, and a debug coordinate system object for orientation in the game world. Other game objects can be added as needed. The HUD, coordinate system and main camera are treated as separate objects. The coordinate system is only drawn in debug mode, since it isn’t required in release mode.

There are actually two cameras, one for 2D mode and one for 3D mode, based on what the game is set to. A separate **GUICamera** object has been added to be used when rendering GUI objects such as a HUD or UI of some kind. All 2D GUI objects are drawn after all other 3D objects because GUI objects are best drawn last.

The *Update()* function will update the main camera and all the objects in the scene and the *Draw()* routine draws all the objects in the scene. Because the coordinate system is rendered first, and because it sets the matrix to the identity, all objects thereafter are rendered at the identity, unless otherwise transformed.

The camera’s *Draw()* function will set up the appropriate view matrix and make sure the correct projection is achieved for all objects in the scene.

**EndState**

This game state monitors the final state of the game. Things like credits, end screens, final messages, etc are intended to be used here. This particular state consists of only one single **SplashScreen** object, which will render the final credits message splash screen before the application ends for good. The *OnExit()* routine will destroy the main game's vertex and fragment shaders that were created and compiled in the Start State's *OnEnter()* function.

**Main**

**Singleton**

This is a Singleton template class. It will instantiate another class of any type and allow access to it via the Instance() member function. The constructor, copy constructor and assignment operator all need to be private so that this class cannot be individually instantiated.

**Main**

This is the main starting point for the game demo. The main game state is created here and added to the main game manager, from where it is run. Change the screen width and height accordingly, and set the pixel per unit value to whichever value you find most comfortable when in 2D mode. Feel free to create and add more game states.

**AABB**

This class encapsulates a 2D axis-aligned bounding box that determines collisions that may occur with other axis-aligned bounding boxes. Even though the bound represents a 2D object, all member variables are floats because that provides more accuracy when calculating the bounds, and will provide better collision precision even if it's just a minute amount.

The *IsColliding()* function checks two AABB boxes for intersection, and for this the routine uses a minimum and maximum X and Y value within a formula to calculate for collision.

Separate floats are passed into the function instead of Vector2D objects, otherwise the routine would have to be templatized.

The *Update()* routine calculates the min, max and half dimension values of the bounding box which are then used to calculate for collision. The position and dimension of the bound are used to calculate these min and max values, and they exist in world space. These values are purely for collision detection and not used for rendering.

The intension of this class is to provide support for mouse cursor collision for 2D objects used within the game, such as GUI.

**Debug**

This class offers debugging resources to help make life a little easier when working with OpenGL. It contains routines to help display OpenGL errors in a more detailed manner.

It presents a log system that will allow messages, warnings and errors to be displayed on the console for better feedback and debugging. It uses some low-level Windows code to get a handle on the console terminal window and display messages in different colors. The error codes in the enum follow the color codes for the Windows console window.

**Game**

This class controls the main game. It is a Singleton so that it can be accessed from anywhere in the main code, and because only one game can exist at any one time. At its core, it consists of a deque of game states, that can be added and removed at any time from within the main client code. The main game loop that controls the entire game and its game states runs from within the *Run()* function.

The *AddState()* routine will add a temporary game state to the FRONT of the deque. This is ideal for states such as a pause or instruction state. The previous state remains in the deque and as soon as the temporary state is complete, it is removed again. The *ChangeState()* function will add a state to the BACK of the deque and remove the previous state from the FRONT. This is ideal for when game states transition from one to another.

The active state will have its *OnEnter()*, *Update()*, *Draw()* and *OnExit()* routines called accordingly from within this class. As soon as the deque contains no more states, the game will end.

This class also keeps track of all time passed. It records elapsed time which is great for frame-independent games, where physics or graphics updates rely on time instead of frame rate. The class also keeps track of total time passed since the game first started. The *Run()* function calculates the elapsed time for each frame.

The *Initialize()* and *ShutDown()* functions start up and close down all the other managers respectively.

**Managers**

**AudioManager**

This class encapsulates an audio manager which will load music, sound effects and voice audio files from disk and store them as FMOD sound objects in a map, with a string reference to that audio object. Currently the audio files supported are **MP3**, **WAV**, **OGG**, **FLAC**, <TBA>. To access any particular audio inside the map the string reference is needed to sift through the map and find the audio. Audio objects can also be unloaded from memory individually or in bulk. This class is a Singleton. The main member variable is **m\_audioSystem**, which is a handle on the entire FMOD audio sub-system and is used to actually play any audio.

Two enum types have been created. The **RemoveType** enum is used when removing audio objects from the map. Either one single specific FMOD audio object can be removed, or the entire map of audio objects can be cleared. The **AudioType** enum is there for letting the manager class know what type of audio it will dealing with.

To use this class, a Audio object needs to be created inside the game client code. Then that audio object will be able to link to a specific FMOD audio object from within the Audio Manager. The *GetAudioData()* function is used to request the FMOD sound pointer and link the audio data with the audio object. The other getter routine *GetAudioSystem()* will return the FMOD audio system pointer which is used inside the Audio class' Play() function to play audio.

The *LoadFromFile()* routine will load in the audio data from a file and store the audio in the correct audio map. Both the music and voice audio is loaded into memory in a streamed way, so as to save on memory because these files can become plentiful and large. SFX files are loaded in directly into memory and kept there. Streamed audio is CPU intensive!

An *Output()* routine is there for debug purposes only and will print to the console how many audio objects are currently stored in each of the three supported maps.

*A major revamp coming to this class soon!*

**InputManager**

This is a Singleton Manager class that controls all the input from the mouse and keyboard (More controls later!!) It uses a set of SDL library tools to manage the input and stores the mouse and keyboard values in various property variables. The class as a whole can determine what keys have been pressed and what state the mouse buttons are in, and these states can be queried from external code at any time.

Three enumerated types have been created to be used when storing the button states of the mouse buttons, the mouse cursor state, and the SDL mouse cursor flag values, whose values all correspond with SDL's numeric values for those specific mouse cursors.

The class variables store all the details of the mouse and keyboard and the **m\_isXClicked** and **m\_isKeyPressed** variables store whether the X in the top right corner of the game window has been clicked or whether a keyboard key has been pressed or not respectively. The **m\_isKeyPressed** variable is handy if you want to quickly check if a random key has been pressed or released as opposed to checking each key's state. For that there is a separate variable called **m\_keyStates**. The **m\_keyStates** variable, which is a pointer to an array of Uint8 values, is filled each time a key is pressed and will store a kind of binary snapshot of the keyboard, stating which keys are pressed, and which are not. Each individual element in the array represents a key on the keyboard and can be queried in the client code as to whether that key is pressed or not.

Three button state variables store the pressed state of the three mouse buttons. This is perfect for mouse functions, but for keyboard keys there are too many keys to individually set their state so for that there is a separate **keystate** array variable.

For each class property variable there is a getter function for when mouse and keyboard states are queried in the client code. There are two *IsMouseColliding()* functions, both using the current mouse coordinates to create either a AABB box or Sphere around that to check if the mouse collides with the passed bound. This is handy for checking if the mouse cursor is hovering over buttons, spherical things or any other game objects. The *IsMouseColliding()* could have been passed a generic Bound object, but then dynamic casting is needed to set the dimension each time and that is expensive!

Templatizing could also work but the two functions are not exactly identical. Basic function overriding seems to be the best solution here for now. (More TBA)

There are two mouse cursor setter functions to set the type and state of the mouse cursor. Using these functions, a specific Windows OS mouse cursor can be created and the cursor may be enabled, disabled, shown or hidden

The *Update()* function is the core of the Input Manager class. It will process all SDL events that build up on the event queue and will store particular keyboard and mouse property values in the relevant variables. These values can then be used at any time in the client code. This routine will need to be called once per frame in the client code.

**ScreenManager**

This class encapsulates screen management, and is designed to set up all screen and window related matters. It is there to set up all *SDL*, *OpenGL* and *GLEW* components, screen projections, viewports, and keep track of the projection matrix, which has a matrix class declared for that. The frame buffer is also cleared and swapped via this class. The **m\_width** and **m\_height** member variables are separate ints because it looks clearer to identify them separately. The **m\_pixelsPerUnit** is there to determine how many pixels will represent one unit of measurement in 2D applications, and this value is stored here and requested by the client code when needed later on. There is a **m\_windowHandle** variable that is a pointer to a Windows OS specific window and can be used in conjunction with the Win32 API to create Windows specific applications. This class is a Singleton.

The *Initialize()* routine probably does the most work in the class. It is there to set up the *SDL* subsystem, and create an *OpenGL* context that the programmer requests. The context might not always be what the programmer wants, but it will set up the next best version instead. This all depends on what the graphics card limitations are and what versions are supported. A context of 3.2 or higher can be set to compatibility or core profile mode, meaning that deprecated functionality can or cannot be accessed respectively. By default if no specific context is set, a compatibility profile supporting the latest version of *OpenGL* will be setup. Compatibility mode only works for versions 3.2 and upwards. On some graphics cards setting a context of 3.0 or less will automatically default to the latest version with compatibility mode on. Mine for instance defaults to 4.5 in compatibility mode.

The function will also create the game screen window and then go ahead and initialize the *GLEW* subsystem, which will manage all extensions for *OpenGL* core access beyond 1.1. This is for Windows OS only, because Windows only allows developer access to the *OpenGL* core up until version 1.1.

Finally, the function will gather the **HWND** window handle from the *SDL* created window and store it so that it may be used if requested in the client code. This is useful when integrating Windows specific functionality like Windows dialog or message boxes, because the **HWND** handle will always reference the main *SDL* parent window.

The function *DisplayExtensions()* will display all extensions available to the programmer based on their graphics card hardware. The *DisplayGraphicsProfile()* function helps with displaying *OpenGL* graphics and shader capabilities. It will show the programmer what graphics card type, manufacturer, *OpenGL* version and shader version is installed and supported.

The *Update()* and *Draw()* routines will clear and swap the frame buffer respectively, and the *ShutDown()* function closes down the *SDL* and *OpenGL* subsystems.

**ShaderManager**

This is a Singleton Manager class that controls the main shader program and all of its shader objects. It can create and destroy multiple vertex, fragment and geometry shaders and manage their components accordingly. Shaders can be attached and detached at any time to allow for multiple shaders to be used in the client code. The manager class has an ID for the main shader program and three maps of IDs, each for the vertex, fragment and geometry shaders.

The getters and setters allow access to the different shader variables. There is a getter that allows us to get the ID of a specific vertex attribute.

There are also various functions that allow us to pass uniform or vertex attribute data to the shaders. Simply pass the functions the name of the shader variable and the data to pass and the manager will pass that data through.

The main functions in the class allow for the main shader program to be created and initialised as well as allowing various shaders to be created, compiled, linked, attached, destroyed etc. For each main function there is a corresponding opposing routine, ie *Initialize()* and its counterpart *ShutDown()*. However, there is no *Unlink()* routine for the *Link()* function. To use different shaders with the current shader program, simply detach the shaders and attach the new shaders before linking them. The shaders need to be attached before they are linked with the main program.

An *Output()* routine is there for debug purposes only and will print to the console how many shader IDs are currently stored in the various shader ID maps.

**Objects**

**SplashScreen**

This game object represents and draws a 2D splash screen which is used in the Start State to display the game engine logo and the API logos used in the engine. It is also used in the End State to display any last minute contact info. The main component is the Sprite, which is used to represent the 2D image and draw it on screen. The fade variable monitors whether the image is fading in or out, which is used to determine when to deactivate the image, ie as soon as it has faded out again. The constructor takes a string parameter to load in the correct splash image to be used.

**MainCamera3D**

This game object represents the main viewing camera and contains a Camera component that will represent the camera's orientation and position in the game world. The Update() function will constantly check what keys are pressed and determine how to move the camera based on the mouse motion. The Draw() function will pass the view matrix to the vertex shader This should be done before all game objects are drawn on screen, meaning that this camera's Draw() routine should be called before all other game objects are drawn.

**MainCamera2D**

This game object represents the main viewing camera and contains a Camera component that will represent the camera's orientation and position in the game world. The Update() function will constantly check what keys are pressed and determine how to move the camera based on the mouse motion. The Draw() function will pass the view matrix to the vertex shader This should be done before all game objects are drawn on screen, meaning that this camera's Draw() routine should be called before all other game objects are drawn.

**HUD**

This game object represents and draws a 2D heads-up display on screen for the viewer to have on-screen information at hand. The main component is the Sprite, which is used to represent the 2D image and draw it on screen.

**GUICamera**

This game object represents the 2D camera that is needed to render 2D objects on screen, such as the HUD or text. It works as a secondary camera to the main camera and should be called before each 2D object is rendered. The Draw() function will simply reset the camera component's view matrix to the identity, which is all we need as a view before rendering our 2D objects.

This camera should not be confused with the main camera's 2D setting, which is for orthographic view in games, and is treated separately to this one.

*2D mode only - this is a static camera intended for UI/GUI/HUD interfaces, no movement!*

**CoordSystem**

This object will create, manage and render a grid system and axes and is intended to be used for debug purposes only!

*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