# Games Programming 2

# Coursework Report

## Jack Oswald- S1312968

## Fraser McFarlane - S1434566

## Steven Smith - S1315451

## David Muir - S1428976

## Github Repository:

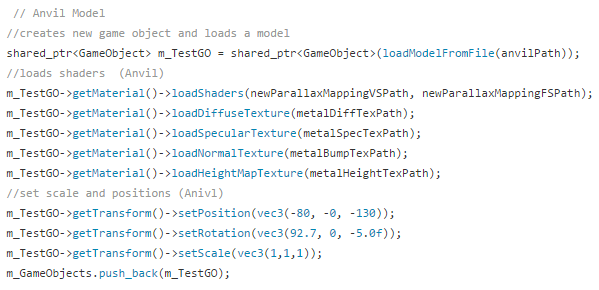
## <https://github.com/fmcfar200/GP2CWProject/>

### Software Architecture

### Required Elements

#### Objects

All of our game objects are added to a vector when they are created. This vector ‘m\_GameObjects’ hold all game objects. When we first initialize our game objects we load a model to that object by using the ‘loadModelFromFile’ method from the model loading class. When the game object is created, a material and transform component is attached to it. For each object that is created a shader and all the correct textures should be loaded onto the object. This is all done through the material component. To apply these shaders and textures we used the ‘getMaterial()->’ call on the game object then call the correct method we want to use like loadShader() or loadDiffuseTexture(). The object's position scale and rotation now need to be set. This is done by accessing the transform component that is attached when the object is created just like the material component.

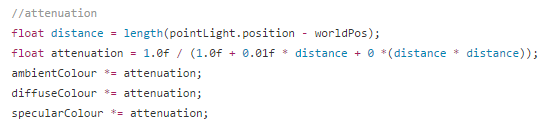


This step is repeated for each game object that is required to create the scene. The transform matrices are updated every frame. This includes the Translation, Rotation , Scale and Model matrix.

#### Lighting & Graphical Effects

For our scene. We created a point light that is generated through our normal and parallax shaders. The Fragment shaders hold a Point Light structure that holds the vec3 for position, the vec4 for ambient colour, specular colour and diffuse colour. It also holds 3 float values named constant, linear and quadratic. These values deal with the attenuation of the point light. In this shader we must modify the way the light direction is calculated. Previously we had a Directional light applied to the shader. As the direction light only shines in one direction, the code has to be changed so that it shines in multiple direction just like a lamp would do. So the code for light direction now takes in the point light position and subtracts it from the fragment world position.

Attenuation needed to be applied to the shaders. This was done by creating a distance float variable that calculates the length between the point light position and to the fragment position. The actual attenuation value is now calculated using the distance, constant, quadratic and linear. This attenuation is now applied to each colour variable by multiplying the attenuation by their value.



One problem that was faced with this is that we were having trouble with accessing the the attenuation variables from outside the shader. We had a lot of tweaking to do when it came to this part due to making the slightest change in some of the variables resulted in the light being non existent or only slightly visible and really dim. This led to creating the attenuation variable by just hard coding the constant linear and quadratic values into the equation.

#### Cameras

The debug view is controlled by keypresses. When pressing the relevant keys in W,A,S and D, the camera will be updated accordingly. To correctly accommodate for the different size vectors in the different camera variables, the result is normalized. The cross product is then used when moving left or right. To correctly balance the velocity of the camera between, a deltaTime variable is used. The function “SDL\_GetTicks()” is used to return a value which represents the number of milliseconds since the SDL library was initialized. This return value allows us to correctly measure the time between the current frame and the last frame, and balance out the speed of the camera from this.

To handle and control the mouse movement, a function called “SDL\_GetGlobalMouseState()” is used first. This function is used to get the current state of the mouse in regards to the desktop. The output of this function is an x position and y position variable. These variables are then used to calculate an offset since the last frame. This offset value for x and y then used to calculate the camera pitch and yaw values. The pitch value represents the angle that depicts how much we are looking up or down by. The yaw values represents the magnitude in which we are looking to the left or right. From these values, the last step is to calculate the direction vector. Once this vector is calculated, it is then mapped onto the front camera.

When the user presses the Q key the camera will switch between being in either FPS or Debug mode. While the camera is in FPS mode it will be unable to pass through the walls that make up the scene whereas the debug camera will be able to fly through the walls to gain an outside view of the scene. This is achieved through the use of boolean values and if statements in the Camera component. Each time that the player wants to move the camera checks which mode it is currently in. If the Camera is in Debug mode then the camera will carry out the movement without any further check. The camera is FPS mode ,however, will check if the camera’s current position is within the set boundaries of the scene (Within the walls). When the Camera gets too close to exiting the scene the Camera will be stopped from moving any further.

Optional Elements

#### Shadows

Several techniques exist for creating shadows, for this project the method looked at was shadow mapping. Shadow mapping works by testing whether a pixel is visible or not from a light source by comparing the pixel to a depth image of the light source’s view. To create the shadow map the scene is rendered from the light source’s perspective. The depth buffer is then extracted and saved from this rendering as a texture. The scene is then drawn from the main camera view whilst applying the shadow map. This results in a scene with dynamic shadows. This was attempted to be implemented into the game but was never completed. The attempted code remains in the project.

#### Post-Processing

To apply a post processing effect on the whole screen, we can render directly to a single texture through a framebuffer object. The framebuffer created contains a framebuffer object, a depth/render buffer and a colour buffer which stored in a texture. Once all these object are linked and the frame buffer is working correctly, the scene is rendered with the newly created framebuffer. A quad is then drawn that will span the entire screen and it will use the framebuffers colour buffer as its texture. Once a scene is rendered to a texture, the texture data can easily be manipulated to give of a multitude of effects such as blur, grayscale and edge detection.

This project couldn’t get a working version of a post processing effect. Although the code is included in the final version, there is currently no post processing effect being displayed. Creating the framebuffer and all its contents seems to work fine, the problem seems to be when displaying this information back to the screen. The specific known issue has not been identified though. The code remains in the project even though not completed.

### References

#### External Models Used:

<http://www.turbosquid.com/3d-models/free-damaged-wall-3d-model/747378>