**Adam S Hosie**

**Student No: S1624519**

**Computer Games (Software Development)**

**Module: Games Programming 2**

**Coursework Documentation**

*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

*Adam Hosie*

**Contents**

**1 Audio**

1a Audio.h - Audio Header File (pg 3)

1b Audio.cpp - Audio Source File (pg 4-5)

**2 Camera**

2a Camera.h initCamera Method (pg 5)

2b Camera.h GetViewProjection Method (pg 5-6)

2c Camera.h Methods for Camera Control (pg 6)

**3 Display**

Display.h

3a Display.h Public Methods (pg 6)

3b Display.h Private Methods (pg 7)

Display.cpp

3c Display.cpp Constructor and Destructor Methods(pg 7)

3d Display.cpp initDisplay Method(pg 7)

3e Display.cpp Remaining Methods (pg 8)

**4 Main Game**

MainGame.h

4a MainGame.h Methods (pg 8-9)

4b MainGame.h Calling Classes (pg 9)

4c MainGame.h Variables (pg 9)

MainGame.cpp

4d MainGame.cpp Initial Methods (pg 9)

4e MainGame.cpp Initialise Systems Method (pg 10)

4f MainGame.cpp Check Collision Method (pg 10)

4g MainGame.cpp Draw Game Method (pg 10-11)

4h MainGame.cpp Update Model Transform (pg 11)

4i MainGame.cpp Process Input Method – Player Controls (pg 11)

4j MainGame.cpp Process Input Method – Spawn Controls (pg 12)

4k MainGame.cpp Game Loop (pg 12)

**5 Mesh**

Mesh.h

5a Mesh.h Structs (pg 12)

5b Mesh.h Mesh Methods (pg 12-13)

Mesh.cpp

5c Mesh.cpp initModel Method (pg 13)

5d Mesh.cpp Remaining Methods (pg 13)

**6 Shader and Lighting**

Shader.h

6a Shader.h Methods (pg 14)

Shader.cpp

6b Shader.cpp Methods (pg 14)

Lighting Shaders

6c Vertex Shader (pg 14)

6d Fragment Shader (pg 15)

**7 Texture**

7a Texture.h Methods (pg 15)

7b Texture.cpp Calls (pg 15-16)

**8 Transform**

8a Transform.h Methods (pg 16)

8a Transform.h Getters and Setters (pg 16)

**9 References**

Code Description and Implementation

The aim of this report is to provide an explanation of the code that is used to generate the game and show how it has been implemented.

The game itself is a simple concept where the player can press any number from 1-7 on the keyboard and can spawn in a new animal. The audio changes with each animal and the player can fly around the animal using the WASD, Space and LAlt to see different parts of the animal. However, if the player flies too close to the animal it will disappear until the fly away again.

**1 Audio**

The Audio header and source files are used to load audio files into the game and control playback of these audio files.

*1a Audio.h - Audio Header File*

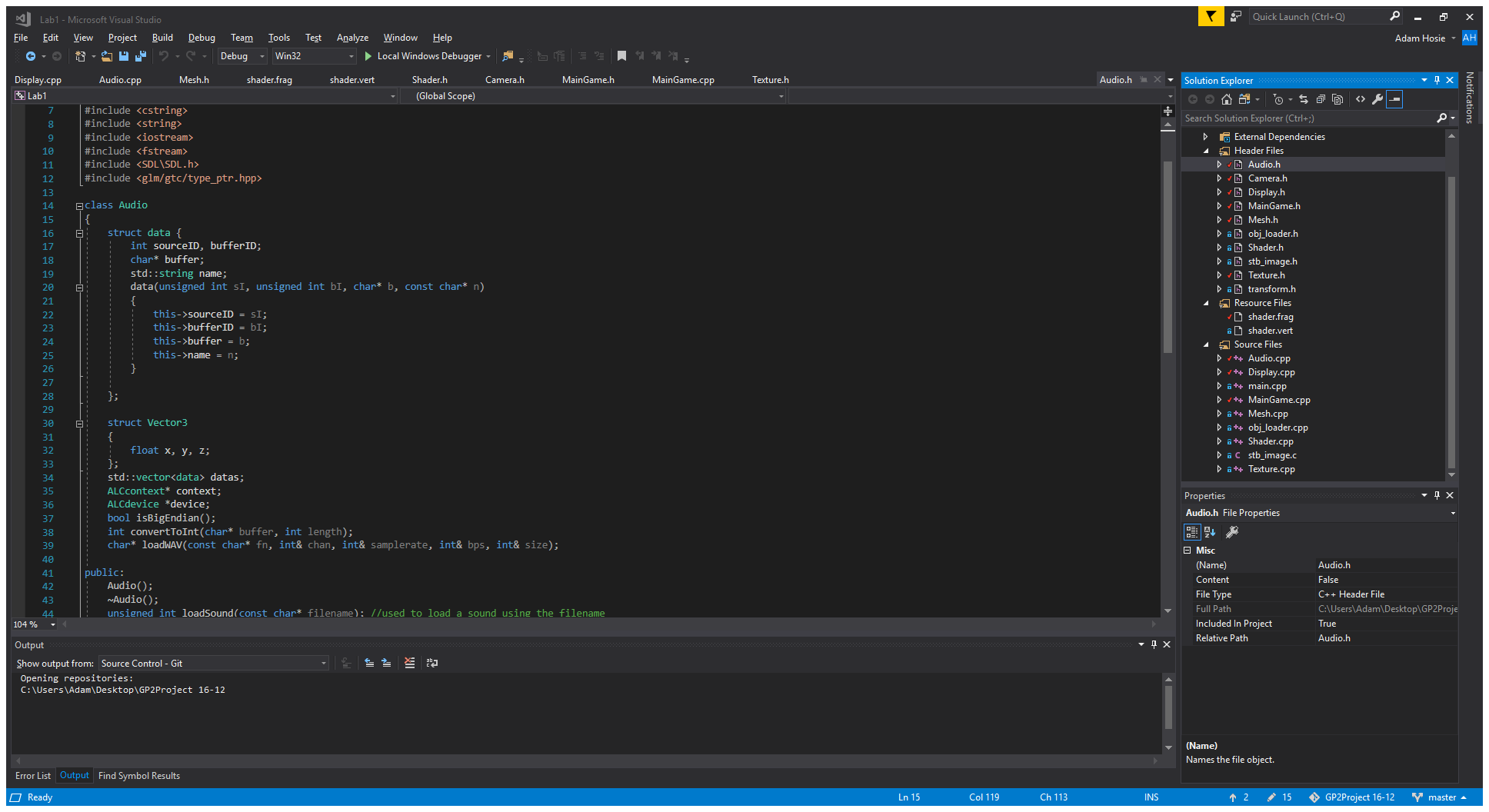


Figure 1 – Audio Data Struct

Figure 1 shows the Audio Data struct, this is used to store all the information related to audio data, which is necessary for reading the audio file.

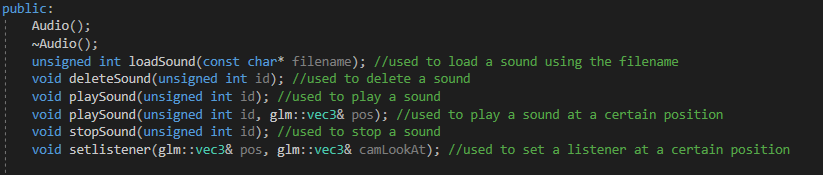


Figure 2 – Audio Methods

Figure 2 shows the methods that can be called from the Audio header file. These have all been fully commented to explain what they do. They are essentially used to load audio and control playback of that audio.

*1b Audio.cpp - Audio Source File*

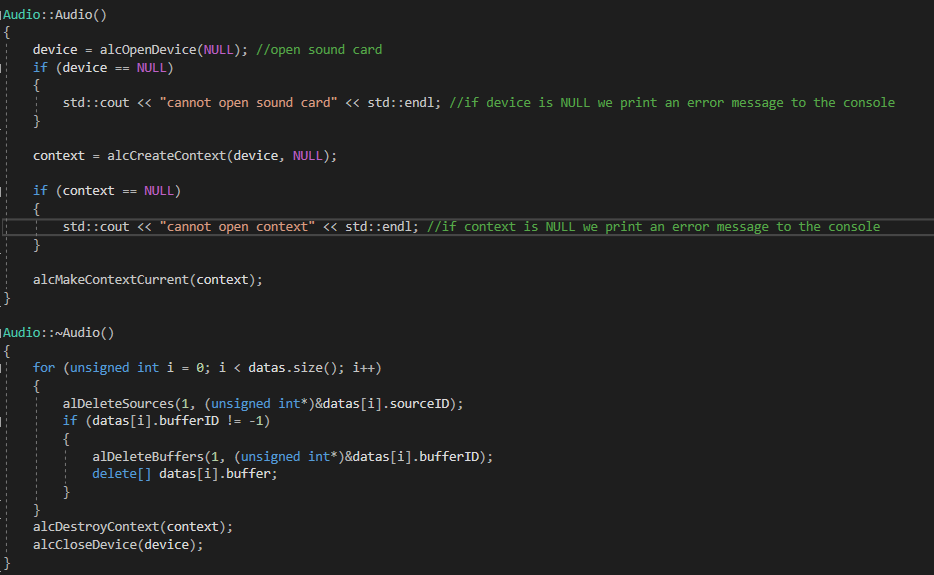


Figure 3 – Audio Constructor and Destructor

Figure 3 shows the Audio constructor and destructor methods. The constructor method is used to open the sound card and create a context for it. It will send an error message to the console if this isn’t possible.

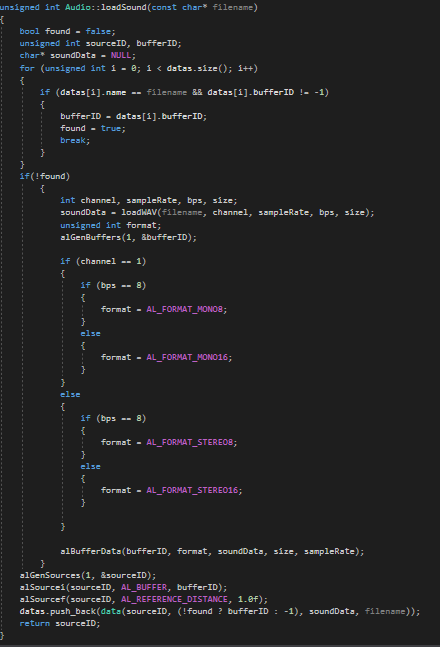
Figure 4 shows the method used to load sound. This method is called in MainGame.cpp whenever we wish to load sounds. This used the loadWavFile method found in Audio.h to load in the file, assign soundData to an AL buffer and then assign it a channel and a format.

Figure 4 – Method used to load sound

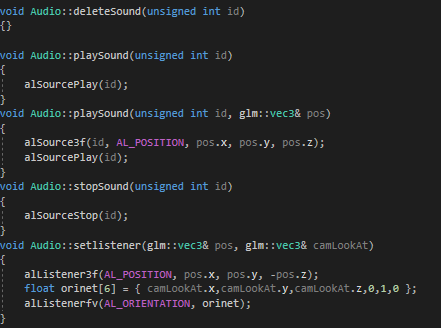


Figure 5 - Methods for Sound Playback

Figure 5 shows the methods used for sound playback. This includes the playSound and stopSound method which are used in MainGame.cpp to start and stop audio.

**2 Camera**

The Camera header file is used to initialise the camera and to allow for control over the cameras movements. The implementation of camera controls was originally important however the methods to do this are no longer being called in MainGame.cpp in the final build. This is because the camera position is now controlled relative to the player position so the inputs for this have been changed. The methods are still able to be used if necessary but don’t have any actual implementation currently.

*2a Camera.h initCamera Method*

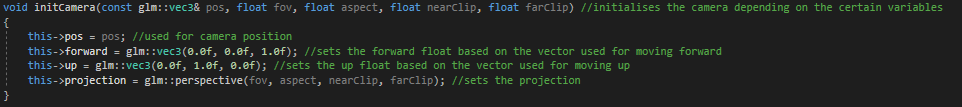


Figure 6 - Method used to initialise the camera

Figure 6 shows the method used to initialise the camera. It has been fully commented. This shows the way we assign information used to initialise the camera. We need to assign the camera position, the up and forward directions and the matrix we use for projection. This matrix sets the field of view, the aspect ratio and the clipping planes.

*2b Camera.h GetViewProjection Method*



Figure 7 - Method used to Get View Projection matrix

Figure 7 shows the method we use to get the view projection. This method is called in Shader.cpp where we use it to get the Model View Projection matrix by multiplying the above matrix with the GetModel matrix, the method for this is found in the transform.h file.

*2c Camera.h Methods for Camera Control*

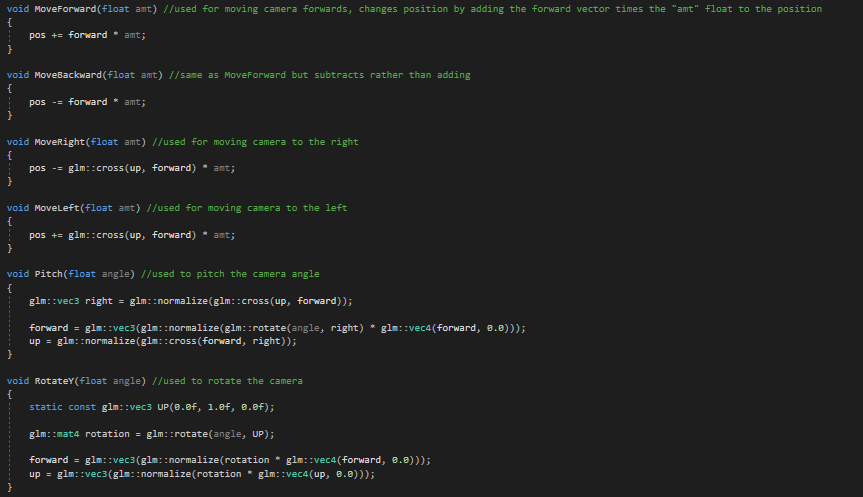


Figure 8 - Method used to control camera movement

Figure 8 shows the six methods used to control camera movement. These are fully commented and are just simple equations used to change camera position. These would normally be called in MainGame.cpp using key inputs but as mentioned, this was removed.

**3 Display**

**Display.h**

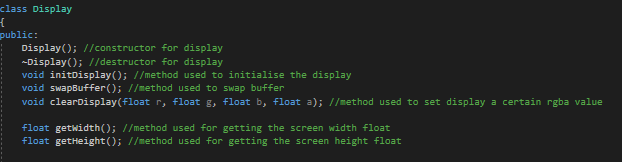
*3a Display.h Public Methods*

Figure 9 – Public Display Methods

Figure 9 shows the methods we use for rendering the display. Firstly we have the constructor and destructor methods. The next method is used to initialise the display, the following swaps the buffer and the clearDisplay() method can be used to set the display to certain rgba value. The next two methods set the width and height.

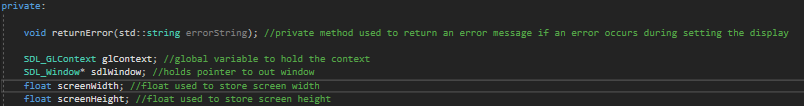
*3b Display.h Private Methods*

Figure 10 - Private Display Methods

Figure 10 shows the private Display.h methods. The first is returnError which is used to return an error if one occurs while rendering the display. The remaining content has been commented.

**Display.cpp**

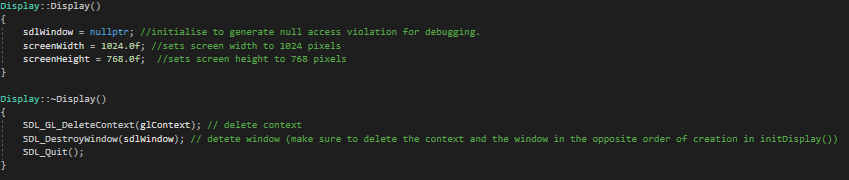
*3c Display.cpp Constructor and Destructor Methods*

Figure 11 - Constructor and Destructor

Figure 11 shows the constructor and destructor methods as they are called in Display.cpp. These have been commented to explain their purposes.

*3d Display.cpp initDisplay Method*

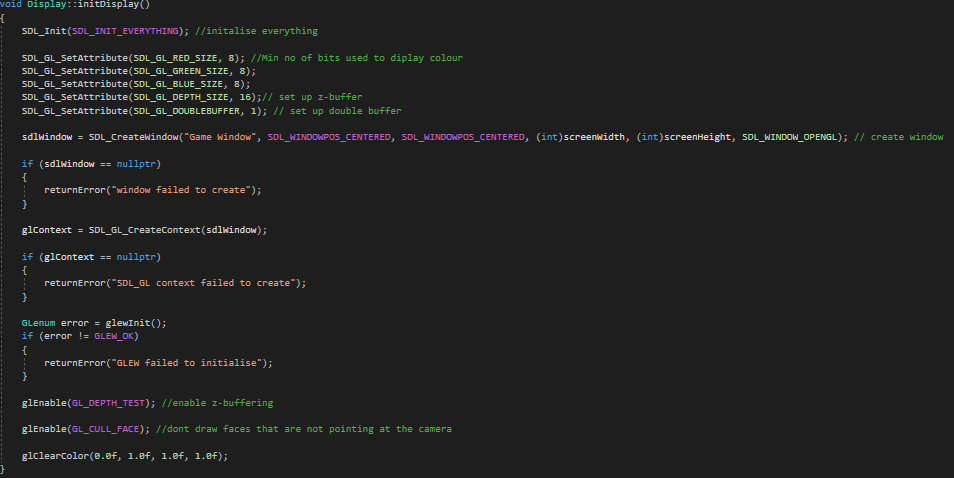
**

Figure 12 - Method used to initialise display

Figure 12 shows the method we use to initialise the display, this uses SDL to do so. This method also has implementation of error prevention, wherein it writes to the console the likely reason for an error, if one occurs during initialisation.

We also enable z-buffering and tell the program to not draw faces that aren’t pointing at the camera in this method. This is to improve performance.

*3e Display.cpp Remaining Methods*

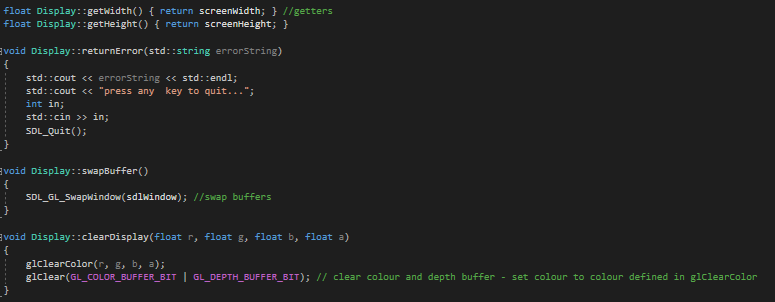


Figure 13 - Remaining methods used in Display.cpp

Figure 13 shows the remaining methods we call in Display.cpp. We firstly have two getters for the screen width and height. We then have the method which is used to return error messages. The next method swaps buffers and the following clears the display the rgba value we desire, this is called in MainGame.cpp.

**4 Main Game**

**MainGame.h**

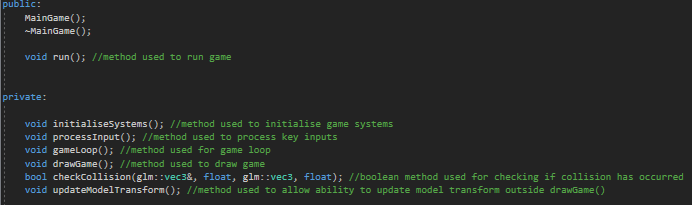
*4a MainGame.h Methods*

Figure 14 - Methods used in Main Game

Figure 14 shows the methods that we use in MainGame.cpp. Firstly we have the constructor and destructor method and the run method which we use to run the game. This run method is called in main.cpp and it runs the program itself. The remaining methods are private, the first of which is initialiseSystems which is used to initialise the game systems. Then we have the processInput method which is used to control the inputs the player makes on the keyboard. The gameLoop and drawGame methods can be explained further in the MainGame.cpp section. We also have the updateModelTransform method which we use so we don’t have to continually update the models position in the drawGame method. The checkCollision method is a Boolean method which we use to do as it suggests and check if a collision has occurred between two models based on their collision spheres.

*4b MainGame.h Calling Classes*

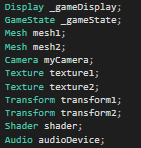
** Figure 15 shows how we call we other classes for use in the MainGame.cpp method, in order to do this we also must include the header files for these classes at the top of MainGame.h. We have to call transform, mesh and texture twice since we use two models and we need to assign these for both.

Figure 15 - Used to call other classes

*4c MainGame.h Variables*

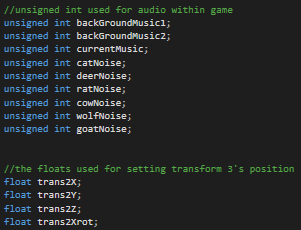
**

Figure 16 - Variables used in MainGame.h

Figure 16 shows the variables we use in the MainGame.cpp file. The unsigned int’s are used for the audio implementation and the floats are used to set the mesh 2 (the spaceship) and camera position.

**MainGame.cpp**

*4d MainGame.cpp Initial Methods*

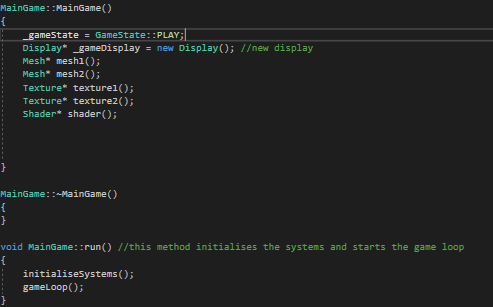
** Figure 17 shows the methods we use initially in MainGame.cpp. Firstly we have the Constructor and Destructor method. The destructor method isn’t used but it has to be included since it is declared in MainGame.h, however the run method is important as this initialises the systems and begins the game loop, as has been commented.

Figure 17 - Initial Methods used in MainGame.cpp

*4e MainGame.cpp Initialise Systems Method*

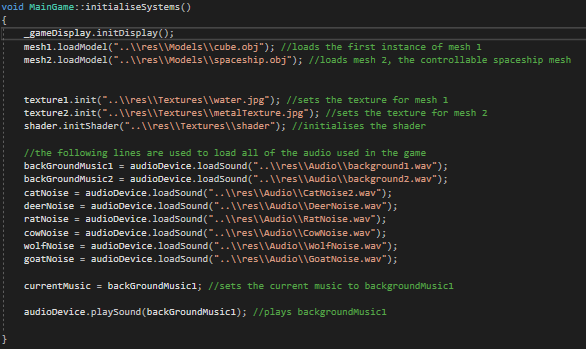
**

Figure 18 - Initialise Systems Method

Figure 18 shows the initialise systems method. This is where we load in all the audio files, texture objects and the model meshes. We also assign current music to backGroundMusic1, this is used so that we know which music is currently being played, in order to prevent overlap of music as we change it when we load models in. We also begin playing the music at the end of this method.

*4f MainGame.cpp Check Collision Method*

The check collision method is a Boolean method which carries out an equation to check if a collision has occurred between the two models. In the case that a collision does occur, we return true otherwise we return false.

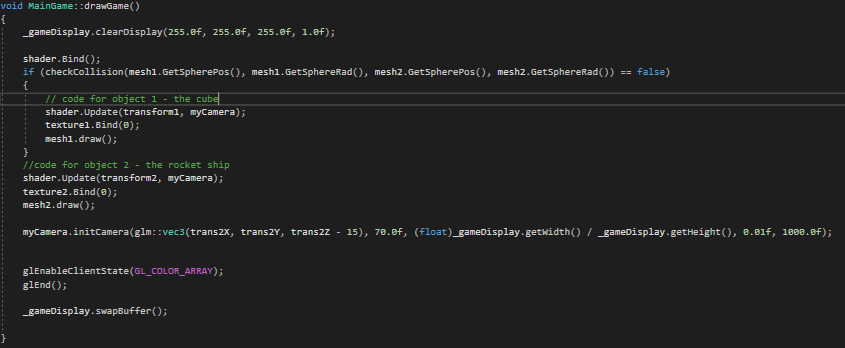
*4g MainGame.cpp Draw Game Method*

Figure 19 - Draw Game Method

Figure 19 shows the draw game method. This method is important as it facilitates the placement, shader implementation, transformation and texturing of the objects we see on the screen. It also initialises the camera, and as we can see the camera uses the same floats in the position vector that we use for the mesh 2. This allows for us to have a camera that essentially follows the player. The code used to generate mesh 1 is contained within an IF loop, this IF loop checks that a collision has not occurred, this is done so that if the player object flies into the second model it will disappear. This is done in order to show that the collision detection works.

*4h MainGame.cpp Update Model Transform*

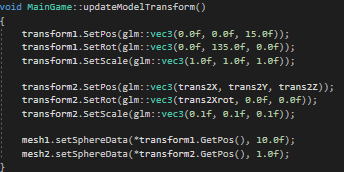
**

Figure 20 - Update Model Transform Method

Figure 20 shows the update model transform method we use in MainGame.cpp. This method has been created so we don’t need to set the models transform continually as we draw game. We see the use of the trans2 floats which are used to allow for control over the model’s position. The data for the collision sphere is also set in this method.

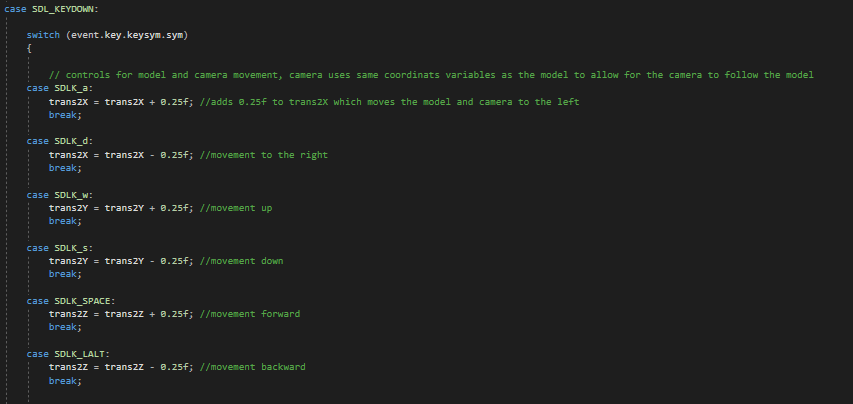
*4i MainGame.cpp Process Input Method – Player Controls*

Figure 21 - Player Controls

Figure 21 shows the implementation of the player controls within the Process Input method. This makes use of case statements. These controls are implemented to allow for movement of the player and have the camera follow them. This section has been fully commented.

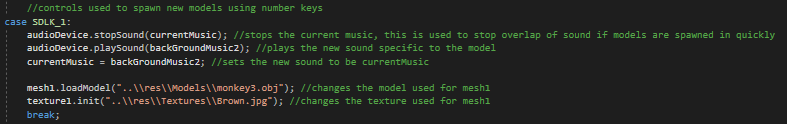
*4j MainGame.cpp Process Input Method – Spawn Controls*

Figure 22 - Spawn Controls

Figure 22 shows just one of the cases for spawning new models. There are 7 in the final game. This code has been fully commented. Firstly we stop the current music from playing so we don’t get any overlap, then we begin playing the new music. Then we set the current audio playing to the music we’ve just started playing. Then the mesh 1 model is changed to the desired model and the texture is changed accordingly. For some of these cases we also change the radius size of the setSphereData method to a different size to make it more fitting for the model size.

*4k MainGame.cpp Game Loop*

The game loop method calls the Process Input method, the update model transform method and the draw game method while the game is running. This creates the game loop which we call in run. This becomes a loop as we use a while statement, so while the game state is not set to exit i.e. running we loop through the previously mentioned methods.

**5 Mesh**

**Mesh.h**

*5a Mesh.h Structs*

Mesh.h makes use of two structs, one for the sphere and one for the vertex. These are generally used to set information regarding them such as position and radius for the sphere. The texture coordinates, position and normals are set for the vertex.

*5b Mesh.h Mesh Methods*

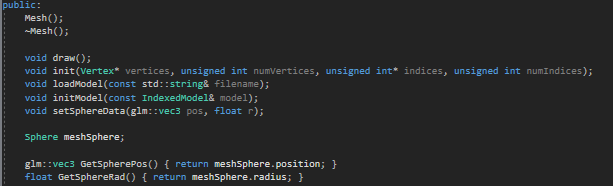
**

Figure 23 - Mesh.h Methods]

Figure 23 shows the methods we have written in Mesh.h, the first two of which are the constructor and destructor methods. We then have the draw method which we use in MainGame.cpp to draw our two models. We also have the method for loading and initialising the model in here, which we use in order to set up the models. This section also contains the data we use for setting the mesh sphere we use for the sphere collider. We have two getters used for getting the sphere position and radius which we then put into the setSphereData method, this is used to fully create the mesh sphere.

**Mesh.cpp**

*5c Mesh.cpp initModel Method*

The initModel method within Mesh.cpp has been fully commented to explain it’s purpose. This method essentially generates a vertex array and stores it in the Vertex Array Object before binding it to such. We then generate our buffers and tell openGL what type of data the buffer is and pass this data, several times. Once we have completed this buffer process we unbind the Vertex Array Object.

*5d Mesh.cpp Remaining Methods*

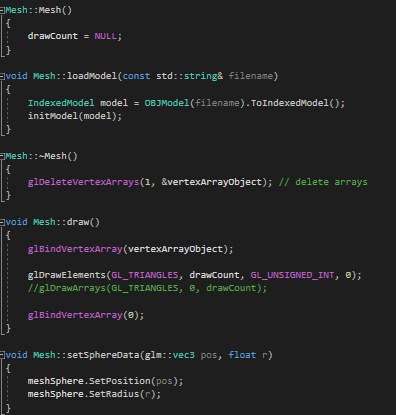
**

Figure 24 - Remaining Methods in Mesh.cpp

Figure 24 shows the remaining methods we have in Mesh.cpp, the first of these is the constructor method where we set drawCount equal to NULL. We then have the method set for loading the model, this then calls the previously mentioned initModel method. Following this is the destructor method where we delete arrays. The important draw method follows this, we call this in MainGame.cpp. As mentioned before, the setSphereData method is then written, this is is used to set the mesh sphere position and radius.

**6 Shader and Lighting**

**Shader.h**

*6a Shader.h Methods*

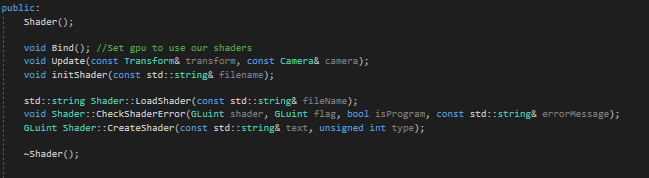
****

Figure 25 - Shader Methods

Figure 25 shows the methods we have in shader. These are used to initialise, bind and update the shaders. We also have methods in places to load and create the shader, as well as return an error in the case that one occurs.

**Shader.cpp**

*6b Shader.cpp Methods*

All of the methods shown in Figure 25 are called in Shader.cpp, these have all been fully commented in order to explain them further.

**Lighting Shaders**

The shaders used creates a lighting effect on the models within the game.

*6c Vertex Shader*

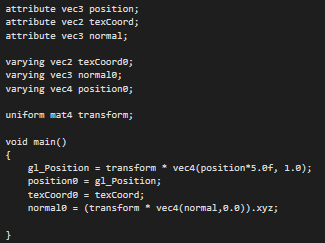
**Figure 26 shows the vertex shader used within the game. We have 3 vectors that we use in the main method and 3 varying vectors that are carried through to the fragment shader. We also use the transform matrix. The main method firstly sets the gl\_Position to the transform matrix multiplied by the vector 4 of the vector 3 position multiplied by 5, with a 1.0 at the end. We then set the varying vector of position0 equal to gl\_Position. The varying vector of texCoord0 is then set to texCoord and the normal0 is also set.

Figure 26 - Vertex Shader

*6d Fragment Shader*

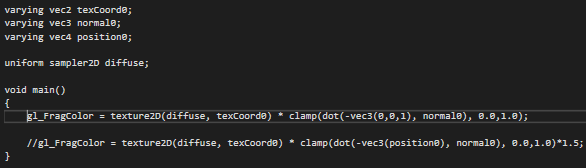


Figure 27 - Fragment Shader

Figure 27 shows the fragment shader used in the game. This takes in the three varying vectors created in the vertex shader and also makes use of a uniform sampler 2D for diffuse. The part of this shader that creates the lighting effect is the gl\_FragColor equation which, without this the images don’t show any detail and appear 2D rather than 3D objects with interesting shadows. Figure 28 shows a side by side of the game with and without the lighting effect. The effect could be made better but it works and shows a clear difference. The line that has been commented out creates a lighting effect that works relative to the players position, however the result didn’t look as good.

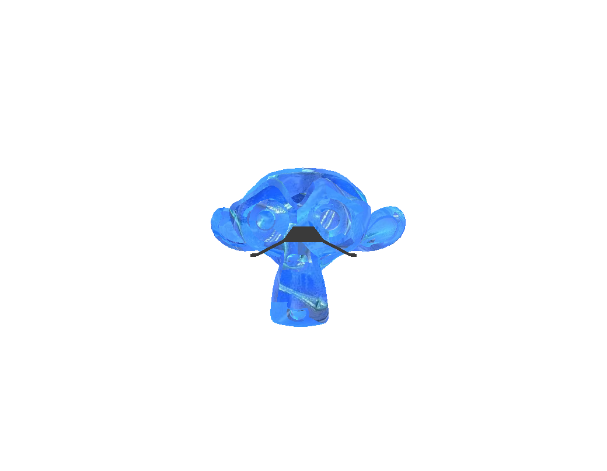


Figure 28 - Difference between shaders

**7 Texture**

*7a Texture.h Methods*

Texture.h only contains 4 methods, these are the constructor and destructor, a bind method and an initialisation method. The use of these is self explanatory, the bind method binds textures to models while the initialisation method prepares them for this binding.

*7b Texture.cpp Calls*

The code used in Texture.cpp has been fully commented to explain it’s purpose. Firstly the init method is called, we create variables for the width, height and number of components in the image we intend on loading. An unsigned char is then created for loading and storing the image. An if statement is then used to check if the image data is null, if it is then we return an error message to the console. Following this we use some openGL code to handle the texture. After this method we call the destructor which can be called to delete the textures. The bind method is then written after this which is used in MainGame.cpp to bind the textures to the models.

**8 Transform**

*8a Transform.h Methods*

We have two methods in Transform.h, the first of these contains the information for the position, rotation and scale. The following method is used to create the matrix we use in the vertex shader, this is named GetModel. We also have a method which has been commented out that can be used to get the Model View Projection matrix, this is commented out as it is not used at any point.

*8a Transform.h Getters and Setters*

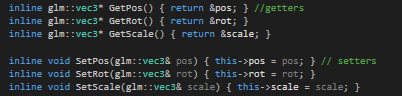
**

Figure 29 - Transform.h Getters and Setters

Figure 29 shows the getters and setters we create in Transform.h, these are used in the Update Model Transform method in MainGame.cpp to set the scale, position and rotation of the models within the game.

**9 References**

Background Music 1: Provided in Lab sessions

Background Music 2: <https://freesound.org/people/Sirkoto51/sounds/370175/>

Wolf Noise: https://freesound.org/people/killyourpepe/sounds/395192/

Cow Noise: https://freesound.org/people/Benboncan/sounds/58277/

Goat Noise: https://freesound.org/people/reinsamba/sounds/57794/

Rat Noise: https://freesound.org/people/AntumDeluge/sounds/188043/

Deer Noise: https://freesound.org/people/dobroide/sounds/104580/

Monkey Model: Provided in Lab sessions

Spaceship Model: https://free3d.com/3d-model/low-poly-spaceship-37605.html

Animal Models: https://free3d.com/user/snippysnappets