Games Programming 2

Computer Games: Software Development

Ritchie Maclean

S1828592

Table of Contents

[Main 1](#_Toc29554308)

[MainGame 2](#_Toc29554309)

[MainGame Constructor 2](#_Toc29554310)

[run 2](#_Toc29554311)

[initSystems 2](#_Toc29554312)

[gameLoop 2](#_Toc29554313)

[Collision 2](#_Toc29554314)

[drawGame 2](#_Toc29554315)

[Display 3](#_Toc29554316)

[Display Constructor 3](#_Toc29554317)

[Display Deconstructer 3](#_Toc29554318)

[Clear Function 3](#_Toc29554319)

[IsClosed Function 3](#_Toc29554320)

[Update Function 3](#_Toc29554321)

[Mesh 4](#_Toc29554322)

[Mesh Constructor 4](#_Toc29554323)

[Mesh Deconstructer 4](#_Toc29554324)

[Init Mesh 4](#_Toc29554325)

[Draw 5](#_Toc29554326)

[updateColData 5](#_Toc29554327)

[Shader 6](#_Toc29554328)

[Shader Constructor 6](#_Toc29554329)

[Shader Destructor 6](#_Toc29554330)

[Bind 7](#_Toc29554331)

[Texture 7](#_Toc29554332)

[Texture Constructor 7](#_Toc29554333)

[Texture Deconstructor 7](#_Toc29554334)

[Bind 8](#_Toc29554335)

[Camera 8](#_Toc29554336)

[Transform 8](#_Toc29554337)

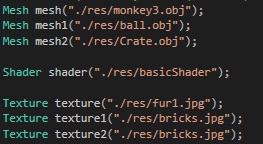
*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

*Ritchie Alasdair MacLean*

# Main

Firstly we create the display by stating the width height and name of the display.

We then define the MainGame.



Here we call the constructors for our different meshes, textures and shader by stating then file path. We also define the Audio, called the camera constructor, set multiple transforms can call mainGame.run();

Two floats are set that are called counter and counter1.

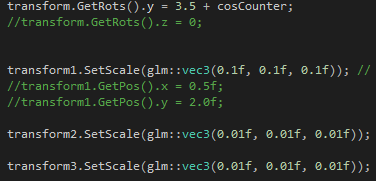
Next we have a while loop that will only run while display.isClosed is false.

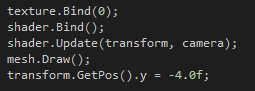
Firstly we call the display constructor and set the RGB & alpha values.

Three floats are then set.

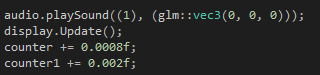


Next in the code we set the various different transforms that we want to set.





For the above we bind the texture that we want to use, bind the shader, update the model view projection matrix and the draw our mesh. The last line sets the y position for the mesh. This is then repeated for all of the meshes that we are using.



For the last little bit we call the playSound method and set the position that the sound is played from. We called the Update method and we swap the simply window with the buffer window. The last two lines just increment our counters by a set amount.

# MainGame

## MainGame Constructor

Set the GameStae to play and set the audio decive

## run

Call initSystems & call the gameLoop functions.

## initSystems

Set backGroundMusic to be the audio file that we want to load

## gameLoop

Call the draw game function

## Collision

Set distance by using Pythagoras



Then check to see is the distance is less the two combined radius, is so there is a collision if not no collision.

## drawGame

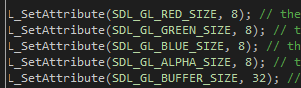
Play the background music audio and set the position it is played at(the source).

# Display

## Display Constructor

The first function is the display constructor which takes in three parameters. An int for the height, an int for the width and a constant string for the name of the display. The first line initialises all of the SDL.

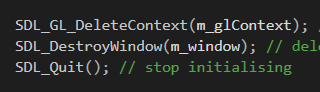
The next block of code sets the number of bits that can potentially be used for the colour



Here we have set all three colours and the alpha you use up to 8 bits each. Then we state that the buffer size should be 32 bits. The total number of bits used for the colours and alpha.

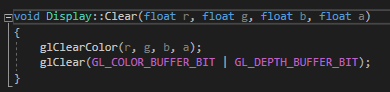
Create space for the Depth Buffer

## Display Deconstructer



Here we delete everything in the reverse order hat it as created. So we delete the context then we delete the window and finally stop the initialising and quit.

## Clear Function



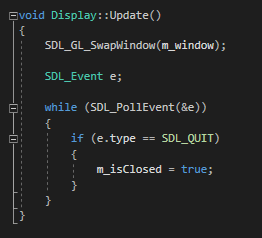
In this function we take in four float parameters. These are used of setting the RGB & alpha values when we run glClearColor();. The last line of code is going to clear the colour buffer so the window is filled with the colour that we set in glClearColor(); The GL\_DEPTH\_BUFFER\_BIT makes it so that we only draw the pixels that are in front. If there is a pixel behind another pixel, it won’t get drawn.

## IsClosed Function

When this is called it will return weather m\_isclosed is true or false as it is store as a bool.

## Update Function

On the first line we are going to swap between the display window and the buffering window.

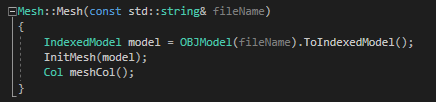


Store an operating system event as “e”. Now we will look for an operating system event and if the system event is SDL\_QUIT then we set m\_isClosed to true.

# Mesh

## Mesh Constructor

This constructor takes in one parameter that is a constant string.



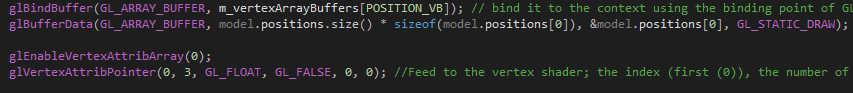
Set model to be the 3D object that we want to load into the game. This will be defined by the file path when the function is called. We will then called the InitMesh function and pass model as the parameter.

Lastly we call the Col struct meshCol();

## Mesh Deconstructer

Here we delete the vertex array object

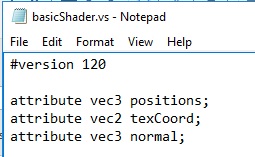
## Init Mesh



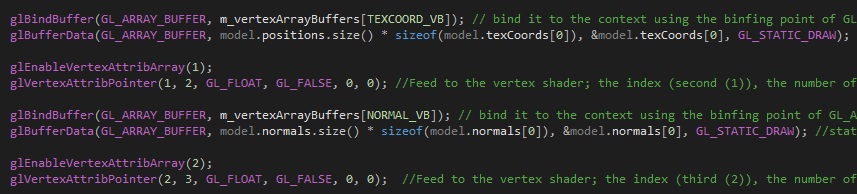
First or all with the above we bind the vertex array buffer that is POSITION\_VB. Anything that would affect a buffer will now affect this buffer[POSITION\_VB].

The next line we tell the GPU how much data that we want to store, we tell the GPU the memory address of the data and then we tell it how to store the data with GL\_STATIC\_DRAW.

With glEnableVertexAttribArray(0) we are setting to the first attribute in the vertex shader with is a vec3 stored as positions



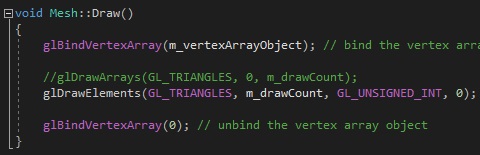
In the last line of code in the first image we feed back to the vertex shader the attribute in this case is 0, we state the number of components it has which in this case is 3 we’re working with a vector 3, we state the data type as a float with GL\_FLOAT. The last three parts of this is set to GL\_FALSE, 0, 0) as we’re not using them.



This process is then repeated for the other two attributes and vertex arrays TEXCOORD\_VB / texCoord & NORMAL\_VB/normal

Lastly we unbind the vertex array with glbindVertexArray(0).

## Draw

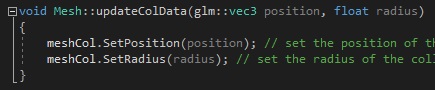


For the draw function we firstly bind the vertex array object.

glDrawElements states the primitives that we’re going to render, the total number of elements to render and the data type (in this case unsigned in).

Lastly we unbind the vertex array.

## updateColData



For the updateColData function we set the colliders position and radius.

# Shader

## Shader Constructor

We create a new shader program called m\_program by calling glCreateProgram();

The nex line we creat a new shader called m\_shader[0] and using the method at the top of the file we load in the shader file, and define the shader type as GL\_VERTEX\_SHADER.

We do the same for m\_shader[1] but we define the shader type as GL\_FRAGMENT\_SHADER

To add the shaders to our main program by using a for loop.



This will attach the first shader file to m\_program, then increment to the next shader file and add that to m\_program. Then when “i” is greater than the number of shaders the for loop with stop running.



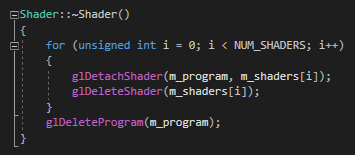
The above tells openGL what part of the data to read in as what variable in the shader program. We bind “positions” to attribute 0 , “texCoords” is bound to attribute 1 and “normal” is bound to attribute 2.

The next line of code links all of the shaders to m\_program and the line after that will check to see if there are any errors in the linking.

Similarly in the next two lines we validate m\_program and check to see if it is valid or not.

## Shader Destructor

Here we have another for loop which does the detached the shaders from m\_program just like the constructor, but the opposite.



The last line of code deletes the shader program

## Bind

Here we bind m\_program with glUseProgram

# Texture

## Texture Constructor

This constructor takes in one parameter which is a constant string.

The first link defines three integers as width, height and numComponets.

Next we create an unsigned char pointer to store the imageData which we call image date. This will hold the name of the image file, the memory address of the width (the x axis), memory address of the height (the y axis), memory address of numComponets and the number of required components.



The above code deals with the wrapping of the texture file around the object. We state that it’s a 2D texture, that we’re wrapping around the x axis “WRAP\_S” and next we state that is the next is no big enough to cover everything along the x axis we repeat the texture along the x axis. We do the same again for the next line but “WRAP\_T” deals with the y axis instead.



The above code will interpolate and extrapolate the texture to fit what it is being applied to. With MIN\_FILTER it will scale down to fit what it is being applied to. The MAG\_FILTER will do the opposite and will and scale up the image if what it’s being applied to is larger than the texture.



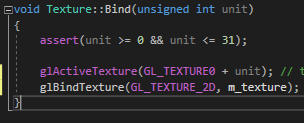
Next we send the texture to the GPU. We state that it’s a 2D texture, the default level, next we chose how to store the pixels. We use GL\_RGBA to store the red, green, blue and alpha components, store the width which we’ve set, store the height which we’ve set, the next component is the boarder which we’re not using, next state the format which we’ve set to send to the GPU as GL\_RGBA, for the next parameter we define how the pixel data is stored which we’ve set as a GL\_UNSIGNED\_BYTE, the last one is the pointer that has all of the data of the pixels which is imageData.

The last line of code in the function just deletes the texture data.

## Texture Deconstructor

Here we delete the texture

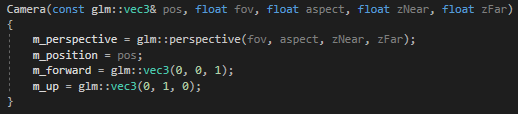
## Bind



The above code will set the active texture that we’re working with, with glActiveTexture. GL\_TEXTURE0 only goes up to 31 which is why we have the “assert” line which will throw up an error if we go out of those bounds. Lastly we will bind the 2D texture that is stored under m\_texture.

# Camera

The camera will only have a header file.



There is a public camera constructor will take in five parameters as follows; a vector 3 for the position, a float for the field of view, a float for the aspect, a float the z axis near and a float for the z axis far.

We will define m\_perspective (which is a matrix 4 variable) as glm::perspective and take in the field of view, the aspect, the z axis near and the z axis far. This will give us the prespective that we see

M\_position (with is a vector 3 variable) is define as the position parameter that we take in the constructor

M\_forward will give us the direction that is in front of us. This will be hard coded as a vector 3 of (0, 0, 1) so that it’s 1 on the Z axis

The same is then done for M\_up, a vector three of (0, 1, 0) where it is just 1 on the Y axis.

# Transform

For he transform we just have a header file which contains the following.

We have a constructor that takes in three vector threes references which are defined as pos for position, rot for rotation and scale for the scale. This give us our model matrix.

For inline glm::mat4 GetModel() const we are going to set the position matrix, rotation matrix and the scale matrix. The rotation matrix is set out a little bit differently and this is to prevent gimbal lock from happening.

We create getters and setters for the different vector 3s.

