Peter Lunardi

S1636120

Games Programming 2

*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*.

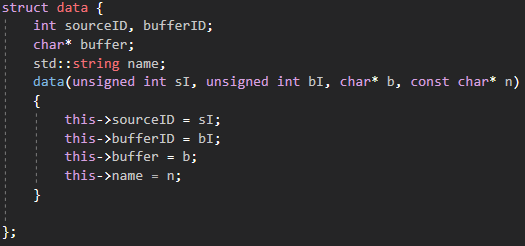
*Signature*.

Contents

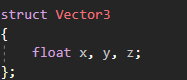
1. Audio
   1. Variables
   2. Methods
      1. loadSound()
      2. playSound()
      3. stopSound()
      4. setListener()
2. Camera
   1. Variables
   2. Methods
      1. initCamera()
      2. GetViewProjection()
      3. GetCameraPos()
3. Display
   1. Variables
   2. Methods
      1. initDisplay()
      2. swapBuffer()
      3. clearDisplay()
      4. getWidth, getHeight
4. MainGame
   1. Variables
   2. Methods
      1. run();
      2. initSystems()
      3. gameLoop()
      4. ProcessInput()
      5. CheckCollisions()
      6. drawGame()
      7. UpdatePositions()
      8. CollisionChecker()
5. Mesh
   1. Structs
      1. Vertex
      2. Sphere
   2. Variables
   3. Methods
      1. draw()
      2. initModel()
      3. loadModel()
      4. setSphereData()
      5. GetSpherePos()
      6. GetSphere()
6. Obj\_loader
7. Shader
   1. Variables
   2. Methods
      1. Bind()
      2. Update()
      3. Init()
      4. LoadShader()
      5. CheckShaderError()
      6. CreateShader()
   3. Shaders used
8. Texture
   1. Variables
   2. Methods
      1. Bind()
      2. Init()
9. Transform
   1. Variables
   2. Methods
      1. GetModel()
      2. GetPos, Rot, Scale
      3. SetPos, Rot, Scale
10. Resources used
11. **Audio**

**1.1 Variables**

Audio contains two structs, data and Vector3 shown in figures 1 and 2 respectively.



Figure

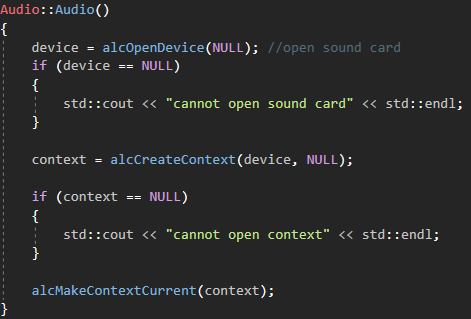


Figure

These are used to store the data of the sound file to be read. The header file also contains an ALCcontext variable “context” and an ALCdevice variable “device” which hold all of the data.

**1.2 Methods**

The constructor opens the sound card and creates a context based on this sound card, and then sets this to our current context, shown in figure 3.



Figure

The destructor simply deletes all of the openAl data by destroying the context and closing the device.

isBigEndian() checks how our computer stores data which is necessary to read sound data properly. convertToInt() converts a char array to an integer array since openAL uses integers as parameters and some sound data is stored as chars. loadWAV() reads .wav files in a way that lets us play them using openAL.

**1.2.1 loadSound()**

unsigned int Audio::loadSound(const char\* filename)

This method takes loads a file from a specified pathway and stores all of that files data in a buffer. The return type of this method is an unsigned integer since audio tracks are referenced using integers.

**1.2.2 playSound ()**

void playSound(unsigned int id, glm::vec3& pos);

This plays the sound attached to a specific integer at the vector3 position we want it to play from. This position doesn’t have to be specified since there is an overload for this method that only takes in and unsigned integer as a parameter.

**1.2.3 stopSound()**

void stopSound(unsigned int id);

Takes in an integer and stops the sound relating to that integer from playing.

**1.2.4 setListener()**

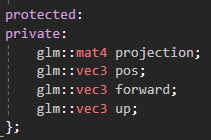
void setlistener(glm::vec3& pos, glm::vec3& camLookAt);

The listener is placed in the world and determines how sound is picked up by the user. This is set by passing in the cameras position and direction it’s looking.

1. **Camera**

**2.1 Variables**

The camera header file contains the variables shown in figure 4.



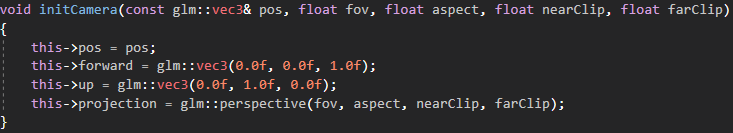
Figure

A 4x4 matrix showing the projection of the camera, a vector 3 position, and two vector3s forward and up which represent the rotation of the camera.

**2.2 Methods**

**2.2.1 initCamera()**

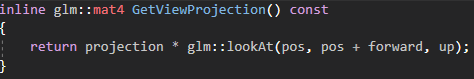
This method initialises our camera by setting the position, rotation and projection based on the values passed in as parameters. It is shown in figure 5.



Figure

**2.2.2 GetViewProjection()**

This simply returns the cameras projection by multiplying the perspective with what we want to look at, and is shown in figure 6.



Figure

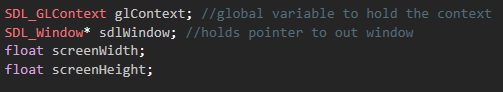
**2.2.3 GetCameraPos()**

This just returns the position of the camera.

**3.0 Display**

**3.1 Variables**

The variables in the display header file are shown in figure 7.



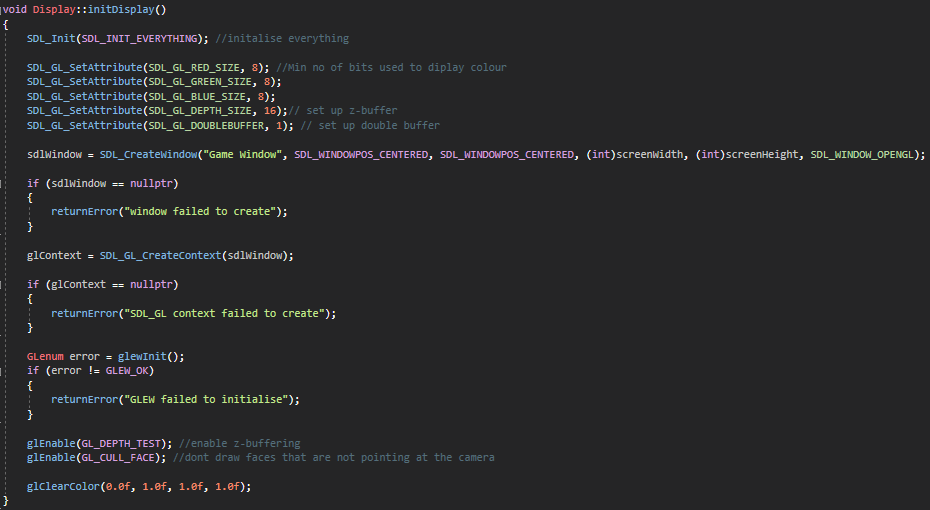
Figure

glContext holds all of our openGL context for our display. This basically lets us draw to the screen using our graphics card and openGL. sdlWindow is a pointer to the window we have created with SDL. screenWidth and screenHeight are the width and height of our window in pixels.

**3.2 Methods**

**3.2.1 initDisplay()**

This entire method is shown in figure 8.



Figure

This method uses various SDL functions to set up the display. We initialise everything, then specify how many bits we want to use to display colour, and create a window with our desired parameters.

We then create an openGL context using SDL which basically tells the operating system to give control of the display to our graphics card. We then enable z-buffering and ensure that we don’t draw anything that the camera is not facing.

**3.2.2 swapBuffer()**

This method swaps between our two buffers that we set up previously and is called after we initialise the display.

**3.2.3 clearDisplay()**

void clearDisplay(float r, float g, float b, float a);

This clears our colour and depth buffer and sets the display colour to a specified colour.

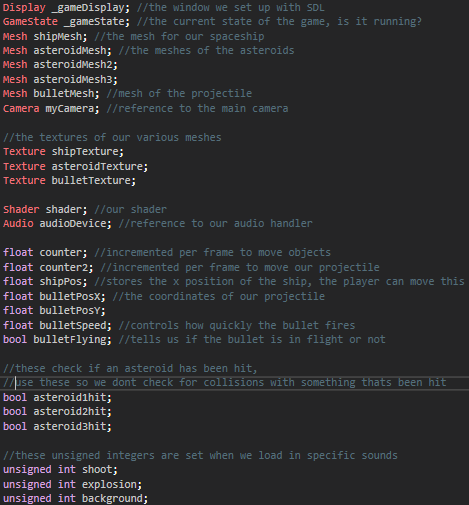
**3.2.4 getWidth, getHeight**

These functions return the width and height of the screen, useful for setting up cameras.

**4.0 MainGame**

**4.1 Variables**

MainGame contains a lot of variables which are all shown in figure 9.



Figure

**4.2 Methods**

**4.2.1 run();**

This is the main method of the game and is the only method called in int main(). It calls two other methods, initSystems() and gameLoop().

**4.2.2 initSystems()**

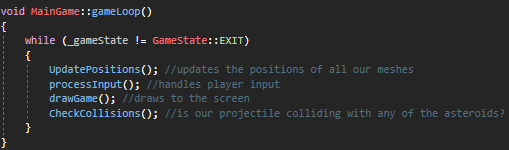
This method is run at the start of the game, and is responsible for initialising everything we need to run the game. It initialises our game display, all the models and textures we plan to use, loads all sounds, shaders and initialises the camera. We also set counter equal to 0 and set the bullet position to be equal to our ships position, as well as setting all asteroidhit bools to false. We also begin playing the background music here so it plays throughout the scene. The whole method is shown in figure 10.



Figure

**4.2.3 gameLoop**

This function checks whether our game is playing, and then runs 4 different functions every single frame. These are processInput(), CheckCollisions(), drawGame() and updatePositions(). gameLoop() is shown in figure 11.



Figure

**4.2.4 processInput()**

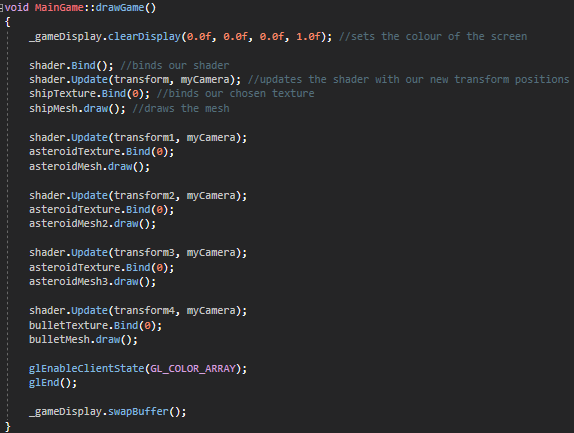
This method allows the player to alter the position of the space ship as well as fire projectiles. The rocket and projectiles positions are updated every frame based on previously defined floats. By altering these floats when certain keys are pressed the different objects can be moved. If the right key is held down, the ships x position is incremented by -1 every frame, and +1 if the left key is held. If space is pressed, we play the sound for shooting a projectile and set the Boolean “bulletFlying” to be the opposite of what it currently is. This lets us either shoot the bullet, or recall it.

**4.2.5 CheckCollisions()**

This just checks for collisions between the bullet and any asteroids. The way this is done will be discussed in 4.2.8. If a collision is detected, the asteroid is destroyed and the bullet is returned to the player. An explosion sound is played, and the asteroidhit bool for that asteroid is set to true. This means we won’t check for collisions with asteroids that have already been hit.

**4.2.6 drawGame()**

This method is where we set up what appears on the screen. We start by setting the game display to black using display.clearDisplay(). We then bind our shader using shader.Bind(). For every object we want to appear on screen we update the shader with the objects transform in relation to our camera, we set the objects texture and then draw the mesh. We then swap the buffer at the end of the function. The drawGame function is shown in figure 12.



Figure

**4.2.7 UpdatePositions()**

updatePositions is responsible for changing all of the transforms associated with each object. It sets the appropriate position, rotation and scale based on certain parameters which either change every frame or are affected by the player. It also determines how the bullet moves based on whether it is flying or not. The method is shown in figure 13.

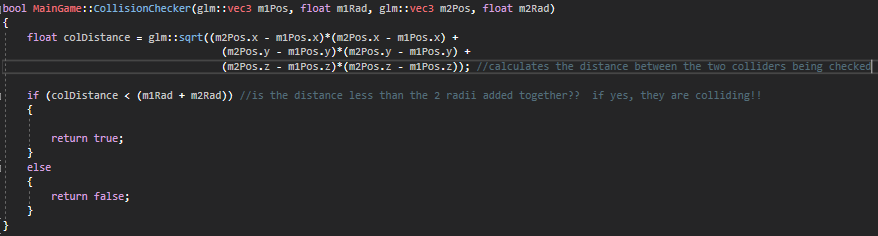


Figure

**4.2.8 CollisionChecker()**

This method contains the maths behind the collision checking. It takes in four parameters, the positions and radii of the two bounding spheres being checked. The distance between these two positions is calculated using the following formula:

This distance is then compared to the sum of the two radii. If the sum of the two radii is less than the distance calculated, then there is no collision and the method returns false. However if the distance is less than the sum of the radii then there is a collision and the method returns true. If it returns true, the appropriate code is run as described in 4.2.5. This function is shown in figure 14.

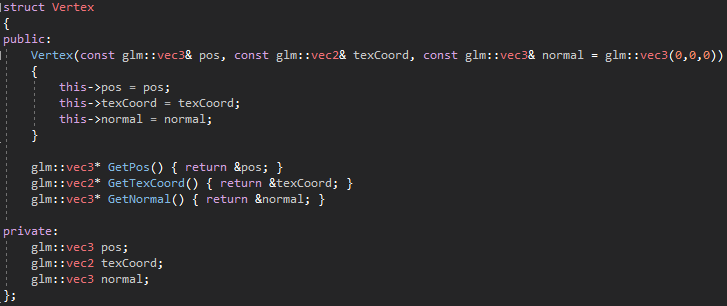


Figure

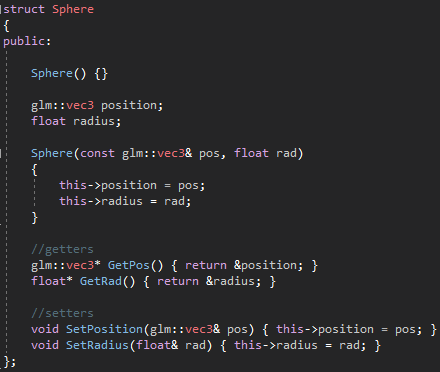
**5.0 Mesh**

**5.1 Structs**

Mesh.h contains two structs, vertex and sphere, shown in figures 15 and 16 respectively.



Figure



Figure

Vertex contains all of the information about our meshes vertexes and has methods that return positions, texture coordinates and normals of these vertices. Sphere is used to create a bounding sphere around each mesh which is used for collision detection between two meshes.

**5.2 Variables**

Mesh contains 3 variables. Two unsigned integers “vertexArrayObject” and “vertexArrayBuffers” and these contain the information for our vertices and are used when initialising our models. It also contains an unsigned int “drawCount” which lets us choose how much of our vertexArrayObject we want to draw.

**5.3 Methods**

**5.3.1 draw()**

void Mesh::draw()

{

glBindVertexArray(vertexArrayObject);

glDrawElements(GL\_TRIANGLES, drawCount, GL\_UNSIGNED\_INT, 0);

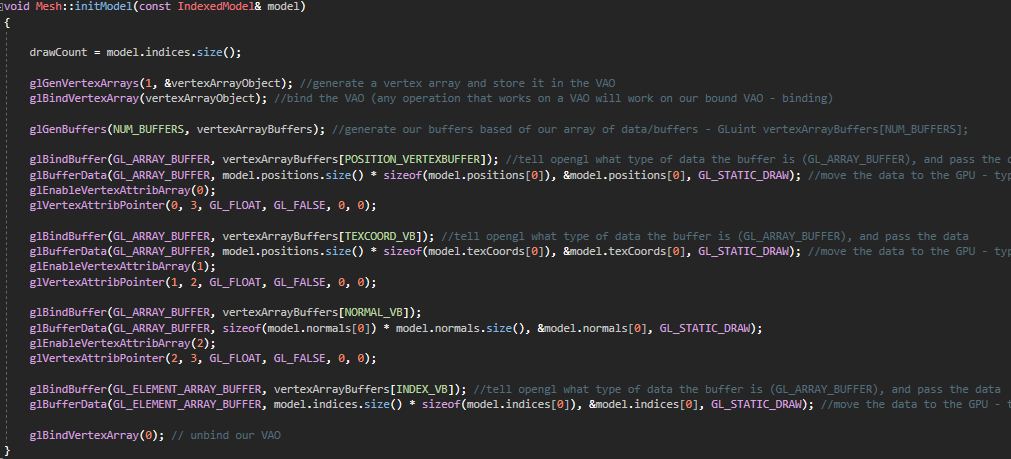
glBindVertexArray(0);

}

This binds our vertex array, then uses an openGL method drawelements to draw everything we want to draw, and then unbinds the vertex array.

**5.3.2 initModel()**

This method is used to initialise our models, and is shown in figure 17.



Figure

It generates an array of vertices and stores it in our vertexArrayObject, which is then bound. We can then generate space on our gpu to store the buffers associated with our objects positions, texture coordinates and normals. When we are finished we unbind the vertexArrayObject so it is not affected by any operation that would affect a vertex array object.

**5.3.3 loadModel()**

void Mesh::loadModel(const std::string& filename)

{

IndexedModel model = OBJModel(filename).ToIndexedModel();

initModel(model);

}

Load model takes in our file path and sets the object to be our indexed model, which is then passed into initModel() mentioned previously which initialises the model ready for use.

**5.3.4 setSphereData()**

void Mesh::setSphereData(glm::vec3 pos, float r)

{

meshSphere.SetPosition(pos);

meshSphere.SetRadius(r);

}

This simply lets us pass in a position and radius for the sphere used for collision detection. The position is that of the model, and the radius can be a suitable value for collision.

**5.3.5 GetSpherePos()**

This returns the position of the bounding sphere.

**5.3.6 GetSphere()**

This returns the radius of the bounding sphere.

**6.0 Obj\_loader**

This class provides us with all of the functions we need when loading in objects from .obj files.

**7.0 Shader**

**7.1 Variables**

Contains an unsigned int NUM\_SHADERS which is the number of shaders we are using, in our case 2. GLuint program which tracks the shader program. GLuin shaders is an array holding our shaders. GLuint uniforms which is the number of uniform variables.

**7.2 Methods**

**7.2.1 Bind()**

void Shader::Bind()

{

glUseProgram(program);

}

This sets up our GPU to use our shaders.

**7.2.2 Update()**

void Shader::Update(const Transform& transform, const Camera& camera)

{

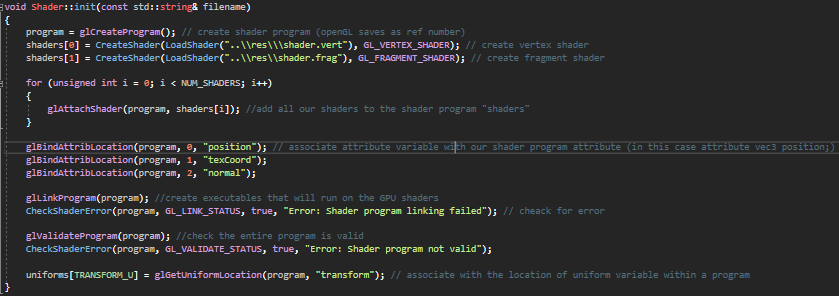
glm::mat4 mvp = camera.GetViewProjection() \* transform.GetModel();

glUniformMatrix4fv(uniforms[TRANSFORM\_U], 1, GLU\_FALSE, &mvp[0][0]);

}

This methods updates our shader with our objects position whenever we change its transform.

**7.2.3 Init()**

This method is shown in figure 18.

Figure

Here we create our shader program and our shaders. We then add our shaders to our program. Then we can associate our attribute variables with our shader programs attributes. The program can then be linked and validated.

**7.2.4 LoadShader()**

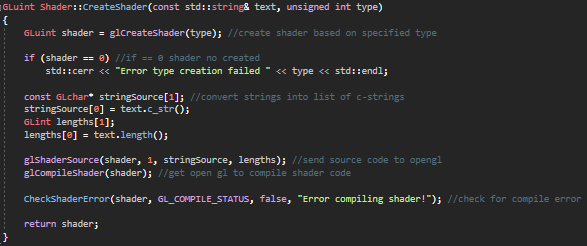
This function simply loads a text file off of the hard drive.

**7.2.5 CheckShaderError()**

This finds any errors we may encounter when binding and loading shaders and generates error messages for us to see.

**7.2.6 CreateShader()**

This method is displayed in figure 19.

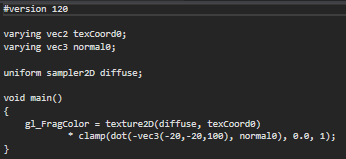


Figure

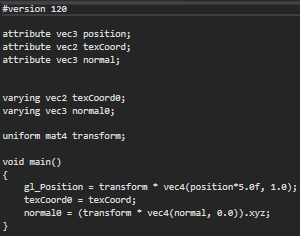
Here we use openGL method glCreateShader to create a shader based on a certain type. We then use openGL to compile our shader code, check for any errors and then return our shader.

**7.3 Shaders Used**

In our program we make use of a fragment and a vertex shader in order to simulate lighting. These are shown in figure 20 and 21 respectively.

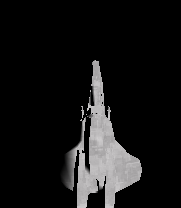


Figure



Figure

The vertex shader is responsible for translating and scaling our normal by multiplying it by the transform. The fragment shader simulates lighting by multiplying each pixel’s colour by an amount based on the cosine of the angle between a pixels normal and the negative of a specified light direction vector. This is done by taking the dot product of these two vectors. This then changes the brightness of the pixel, clamped between 0 and 1. This effect can be seen clearly by looking at the spaceship in figure 22.



Figure

**8.0 Texture**

**8.1 Variables**

Texture only contains one variable, a GLuint “textureHandler” which lets us store the data in our texture.

**8.2 Methods**

**8.2.1 Bind()**

void Texture::Bind(unsigned int unit)

{

assert(unit >= 0 && unit <= 31); /// check we are working with one of the 32 textures

glActiveTexture(GL\_TEXTURE0 + unit); //set acitve texture unit

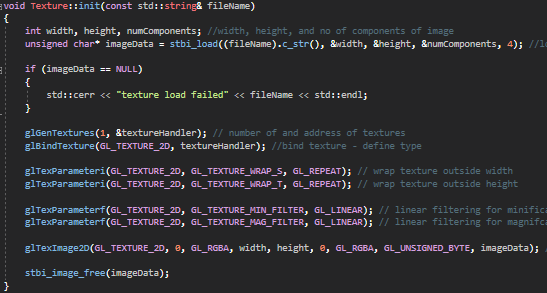
glBindTexture(GL\_TEXTURE\_2D, textureHandler); //type of and texture to bind to unit

}

This just binds the texture to our textureHandler.

**8.2.2 Init()**

This method is shown in figure 23.



Figure

This uses methods from an imported header file stb\_image.h to load in a file from a file path. We can then define the texture type, wrap the texture around width and height and handle magnification and minification. Once we have all of the data we need and stored in textureHandler we can delete the data using a method from stb\_image, stbi\_image\_free().

**9.0 Transform**

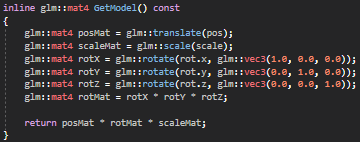
**9.1 Variables**

This class has three private vector 3 variables to hold the position, rotation and scale of an object.

**9.2 Methods**

**9.2.1 GetModel()**

This method is displayed in figure 24.



Figure

This returns a matrix4 containing the position, scale and rotation of the object. This is used when updating shaders with our objects transform.

**9.2.2 GetPos, Rot, Scale**

These three methods simply return the current value for the objects position, rotation and scale.

**9.2.3 SetPos, Rot, Scale**

These methods are used to set the position, rotation and scale of an object. They are used in the game loop.

1. **Resources used**

* The maths and code for the vertex and fragment shaders were taken from thebennybox’s tutorial series, found at the following address:

<https://www.youtube.com/watch?v=NS980twY1ZE&list=PLEETnX-uPtBXT9T-hD0Bj31DSnwio-ywh&index=12>

* Obj\_loader, stb\_image, and resource.h were all provided on GCU Learn by Bryan Young.
* 3D models were downloaded from the following address:

<https://free3d.com/3d-models/obj>

* Music and sounds were taken from the following address:

<https://freesound.org/>

* A lot of the code used and advice gained was from thebennybox’s very helpful openGL tutorial series, found at the following address:

<https://www.youtube.com/playlist?list=PLEETnX-uPtBXT9T-hD0Bj31DSnwio-ywh>

* Maths for collision detection was based on Lab 7 by Bryan Young
* Audio was implemented using techniques from Lab 8 by Bryan Young