Name: Robert Klivis

Module Number: M3I622943

Module Name: Games Programming 2

Course Name: Computer Games Software Development (Year 3)

Lecturer: Bryan Young

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

Coursework

Games Programming 2

Table of Contents

[1 Overview of Main Methods, Structures and Classes 3](#_Toc533047260)

[1.1 Camera 3](#_Toc533047261)

[1.1.1 Variables 3](#_Toc533047262)

[1.1.2 PlayerCamera() 3](#_Toc533047263)

[1.1.3 initCamera() 3](#_Toc533047264)

[1.2 Game 4](#_Toc533047265)

[1.2.1 Variables 4](#_Toc533047266)

[**1.2.2** **Game()** 5](#_Toc533047267)

[**1.2.3** **run()** 5](#_Toc533047268)

[**1.2.4** **initSystems()** 5](#_Toc533047269)

[**1.2.5** **drawText()** 8](#_Toc533047270)

[**1.2.6** **playAudio()** 9](#_Toc533047271)

[**1.2.7** **gameLoop()** 11](#_Toc533047272)

[**1.2.8** **processInput()** 11](#_Toc533047273)

[**1.2.9** **Collision()** 12](#_Toc533047274)

[**1.2.10** **transformObject()** 13](#_Toc533047275)

[**1.2.11** **drawGame()** 13](#_Toc533047276)

[1.3 Fruits 14](#_Toc533047277)

[1.3.1 Variables 14](#_Toc533047278)

[**1.3.2** **TransformObject()** 14](#_Toc533047279)

[1.3.3 Texutres: 15](#_Toc533047280)

[**1.3.4** **Init()** 15](#_Toc533047281)

[**1.3.5** **Bind()** 16](#_Toc533047282)

[1.3.6 Meshes: 16](#_Toc533047283)

[**1.3.7** **init()** 16](#_Toc533047284)

[**1.3.8** **initModel()** 16](#_Toc533047285)

[**1.3.9** **loadModel()** 17](#_Toc533047286)

[**1.3.10** **draw()** 17](#_Toc533047287)

[**1.3.11** **meshSphere()** 17](#_Toc533047288)

[1.3.12 Shaders: 17](#_Toc533047289)

[**1.3.13** **init()** 17](#_Toc533047290)

[**1.3.14** **CreateShader()** 18](#_Toc533047291)

[**1.3.15** **LoadShader() & CheckShaderError()** 18](#_Toc533047292)

[1.4 Display 19](#_Toc533047293)

[1.4.1 Variables 19](#_Toc533047294)

[**1.4.2** **GameDisplay() and GameDisplay() Deconstructor** 19](#_Toc533047295)

[**1.4.3** **returnError()** 19](#_Toc533047296)

[**1.4.4** **swapBuffer()** 19](#_Toc533047297)

[**1.4.5** **clearDisplay()** 19](#_Toc533047298)

[**1.4.6** **initDisplay()** 20](#_Toc533047299)

[1.5 **Sources** 21](#_Toc533047300)

[1.5.1 Player Camera 21](#_Toc533047301)

[1.5.2 Audio 21](#_Toc533047302)

[1.5.3 Collision Detection 21](#_Toc533047303)

[1.6 Text 21](#_Toc533047304)

[1.6.1 Textures 22](#_Toc533047305)

[1.6.2 Meshes 22](#_Toc533047306)

[1.6.3 Shaders 22](#_Toc533047307)

# Overview of Main Methods, Structures and Classes

## Camera

### Variables

* A Matrix “ViewMatrix”, a camera view matrix that holds all the cameras details
* A Vector3 “Position” that holds the world position of the camera
* A Vector3 “LookAt” that holds the world position the camera is looking at
* A vector3 “Forward” that holds forward position of the camera
* A Vector3 “Up” that holds the up vector of the camera

Camera contains following methods:

### PlayerCamera()

This method creates the camera for the player.

### initCamera()

This method initialises the camera.

It includes setters for the position, forward position, up vector and projection matrix:

this->pos = pos;

this->forward = glm::vec3(0.0f, 0.0f, 1.0f);

this->up = glm::vec3(0.0f, 1.0f, 0.0f);

this->projection = glm::perspective(fov, aspect, nearClip, farClip);

## Game

### Variables

Game contains the following variables:

* A mesh “object1” which initialises mesh for the first object
* A mesh “object2” which initialises mesh for the second object
* A mesh “object3” which initialises mesh of the third object
* A texture “texture” which initialises first texture for the first object
* A texture “texture2” which initialises second texture for the second object
* A texture “texture3” which initialises third texture for the third object
* A shader “shader” which is used to initialise the shader
* A collision detector “collision” which is used to initialise collision detection
* An audio device “audioDevice” which initialises audio device for use
* An object collision detection “mesh” which is used to recognise collision between meshes
* A Boolean “start” which is used to recognise when the game is launched
* A Boolean “win” which is used to recognise when the player has won the game
* A float “d” which sets the distance value between objects for collision detection
* A float “radius” is radius that is used for the set bounding sphere
* A float “move” is used to set movement counter, that is later is used to transform objects position or rotation.
* An unsigned integer “BackgroundMusic” is used to set background music
* An unsigned integer “SplashSound” is used to set splash sound whenever any fruit is destroyed
* A Boolean “Collision” which is used to detect that a collision has happened
* A Boolean “fruit1” is used to detect whether first fruit is active or not
* A Boolean “fruit2” is used to detect whether the second fruit is active or not
* A Boolean “fruit3” is used to detect whether third fruit is active or not
* A Boolean “fruit4” is used to detect whether fourth fruit is active or not
* A Boolean “fruit5” is used to detect whether the fifth fruit is active or not
* A model “banana” is an object used for the “object1” that is used for one of the fruits
* A model “Apple” is also an object that is used for one of the fruits but as “object2”
* A model “Orange” is used as an “object3” for one of the fruits in the program
* A texture “BananaTexture” is a texture that is used for the “banana” model
* A texture “AppleTexture” is a texture that is used for the “Appple” model
* A texture “OrangeTexture” is a texture that is used for the “Orange” model
* A shader “shader” is initialised as a shader
* A camera “myCamera” is a player camera that is initialised from the player point of view
* A string “text” is a text that is used to display the name of the game
* A string “text2” is a text that is used to display the controls of the game
* A string “text3” is a text that is used to display ways for the player to start the game
* A string “text4” is a text that is used to display how the player can exit the game after the player has won

The game class contains the following methods:

### **Game()**

This method sets the game state to play and it also creates a new game display.

It also initialises object meshes, textures as well as audio.

### **run()**

This method is used to initialise the systems and activate the game loop.

### **initSystems()**

This method is used to initialise the game display.

It is also used to load different models that are needed.

This is done by loading a model from an object in set location as can be seen by the following codes:

object1.loadModel("..\\res\\banana.obj");

object2.loadModel("..\\res\\Apple.obj");

object3.loadModel("..\\res\\Orange.obj");

System initialisation method is also used to load different textures that can be used for the objects:

texture.init("..\\res\\BananaTexture.jpg");

texture2.init("..\\res\\AppleTexture.jpg");

texture3.init("..\\res\\OrangeTexture.jpg");

It is also used to load necessary shaders:

shader.init("..\res\\shader");

Or also to initialise the player camera and set the x, y , z positions based on the width and height of the game display:

myCamera.initCamera(glm::vec3(0, -5, -45), 70.35f, (float)gameDisplay.getWidth()/gameDisplay.getHeight(), 0.1f, 100.0f);

To initialise the camera, initCamera() method is used which uses the following code:

void initCamera(const glm::vec3& pos, float fov, float aspect, float nearClip, float farClip) //initlise the camera

{

this->pos = pos;

this->forward = glm::vec3(0.0f, 0.0f, 1.0f);

this->up = glm::vec3(0.0f, 1.0f, 0.0f);

this->projection = glm::perspective(fov, aspect, nearClip, farClip);

}

From this code a 4x4 projection matrix for the camera can be created based on the field of view, the aspect ratio and the far and near clipping planes using projection part of the code.

Then 4x4 “look at” view matrix based on Vector3s which specify the eye position, the position that the player is looking at and the up vector is created. This projection matrix is then multiplied by the look at matrix to create a view projection matrix which is displayed in the code below:

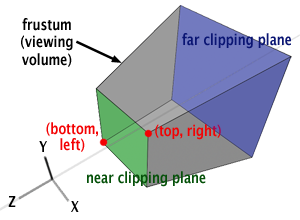
inline glm::mat4 GetViewProjection() const

{

return projection \* glm::lookAt(pos, pos + forward, up);

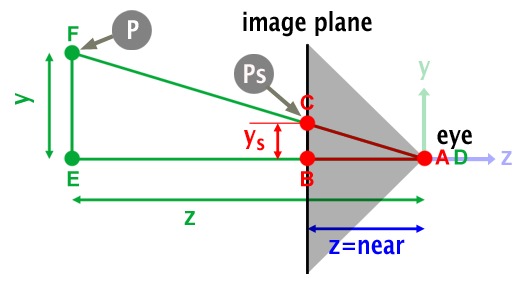
}

As for the camera in general, perspective projection is often used for camera in 3D games. In perspective projection, a 3D point in eye coordinates is mapped to a cube. This diagram shows an overview of this:



The frustum of viewing volume of a camera is defined by the camera’s field of view, the near and far clipping planes and the image aspect ratio.

In OpenGL, points are projected on the front face of the near clipping plane. Line is drawn from the camera’s origin to the point P that it is wanted to project, and the intersection of this line with the image plane indicates the position of the projected point Ps. Which is displayed in image below:



Then view projection matrix is multiplied by the transformation matrix to create a model view projection matrix and sent to shader by using glUniformMatrix4fv. Matrices could be sent to the shader individually to do the calculations inside the shader, but it is more efficient to carry out the calculations prior to sending the data to the shader.

This is also display in the code below:

void ObjectShader::Update(const ObjectTransform& transform, const PlayerCamera& camera)

{

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

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

}

It is also important to note that the model, view and projection matrices are three separate matrices. Model maps from an object’s local coordinate space into the world space, view from world space to camera space, projection from camera to screen. If all three matrices are composed into one matrix then it can be moved straight from coordinate space to the screen. This matrix is called the MVP matrix.

Also, this initSystems() method has been used to set necessary Booleans to false and setting the movement counter value.

It is also used to load audio files from the “wav” files and playing the background music so that it is being played on launch of the game which can be seen by the code below:

BackgroundMusic = audioDevice.loadSound("..\\res\\BackgroundMusic.wav");

SplashSound = audioDevice.loadSound("..\\res\\SplashSound.wav");

audioDevice.playSound(BackgroundMusic);

### **drawText()**

This method is used to draw the text that can be initialised in draw method afterwards.

Method used to draw the necessary text was sourced from the internet.

Following is the code that is included in drawText() method to render the text as necessary:

glMatrixMode(GL\_PROJECTION);

double \*matrix = new double[16];

glGetDoublev(GL\_PROJECTION\_MATRIX, matrix);

glLoadIdentity();

glOrtho(0, 800, 0, 600, -5, 5);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity;

glPushMatrix();

glLoadIdentity();

glRasterPos2i(x, y);

for (int i = 0; i < length; i++) {

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_24, (int)text[i]);

}

glPopMatrix();

glMatrixMode(GL\_PROJECTION);

glMatrixMode(GL\_MODELVIEW);

It is also important to note that the method used was not the most efficient that can be used.

From the code used following is the part that can be edited to change the font and size of the text that is being rendered:

for (int i = 0; i < length; i++) {

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_24, (int)text[i]);

}

It renders a bitmap character using OpenGL which in this case is a text with set size and font.

### **playAudio()**

This method can be used to set the audio so that it can be played at necessary time and from direction that is needed.

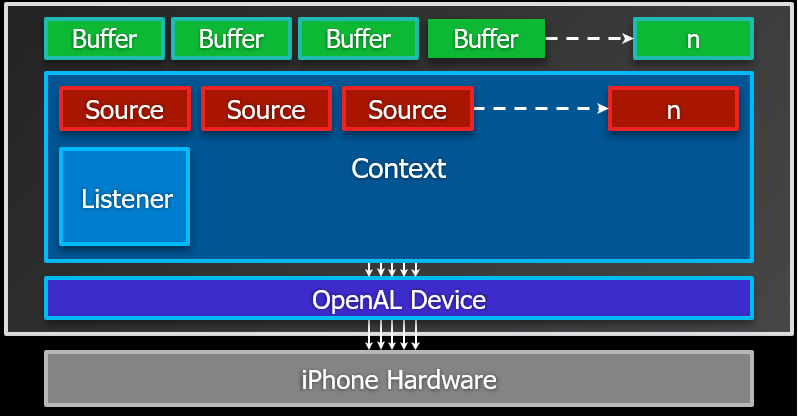
For this project OpenAL is being used. It is API for 3D audio mixing which is design to be used for games and cross-platform.

It models audio in 3D space, as heard by a single listener.

It is also designed to be a compliment to OpenGL as it mimics OpenGL conventions and uses the same coordinate system.

It can be implemented by using Core Audio’s 3D MixerAU.

Architecture of OpenAL is display in the following picture:

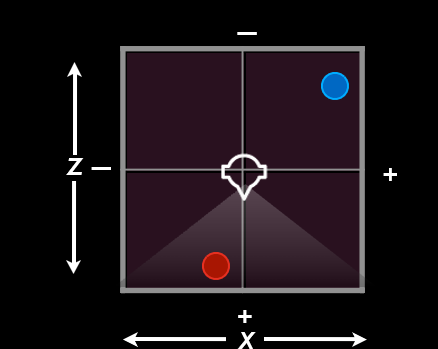


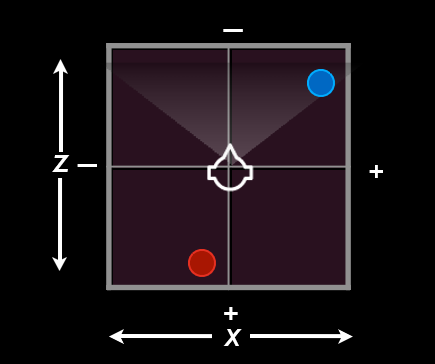
As can be told by architecture Listener is an important part of OpenAL as it represents the user’s experience in the 3D environment.

Its orientation is described by two vectors which are AT or direction the Listener is facing and UP which is direction pointing up from the top of the Listener’s head

Cartesian coordinate system plays an important for audio in OpenGL when using OpenAL.

Below are picture displaying cartesian coordinates when z is set to +1 so that the Listener is facing in the Positive Z direction or when z is set to -1 so that the Listener is facing in the Negative Z direction.





This is a code that can be used for basic setup of OpenAL which is also being used for this project:

oalDevice = alcopenDevice(NULL);

oalContext = alcCreateContext(oalDevice, NULL);

alcMakeContextCurrent(oalContext);

It is used to open an OpenAL device, create a new OpenAL context and the Listener, and set new context to be the current OpenAL context.

There is also more code that can be used to create buffers and sources for example:

alGenBuffers(1, &oalBuffer);

alBufferDataStatic(oalBuffer, AL\_FORMAT\_MONO16,

audioData, audioDataSize, 44100);

alGenSources(1, &oalSource);

alSourcei(oalSource, AL\_BUFFER, oalBuffer);

This code creates an OpenAL buffer to hold the audio data, fills the OpenAL buffer with data, creates and OpenAL source object and attaches the OpenAL buffer to the OpenAL source.

### **gameLoop()**

The method is used to set the loop for the gameplay. It makes sure that the program processes the inputs and draws the game while the game state is not set to exit so that the game is displayed while it is being played and not exited.

### **processInput()**

This method is mainly used for SDL events like key presses and setting the game states.

For the keypresses it uses SDL\_KEYDOWN method for checking if “return” key has been pressed, if it has been pressed then it makes sure that “start” Boolean is set to false and if it is then both “fruit1” and “start” Booleans are set to true which means that the game is started and first fruit is loaded:

if (evnt.key.keysym.sym == SDLK\_RETURN) {

if (start == false)

{

fruit1 = true;

start = true;

}

}

It also checks that if “escape” key is pressed then the program is exited.

As for SDL\_KEYUP it makes sure that the space button is pressed, also if first fruit is active and Boolean “fruit1” is true then it becomes inactive and “fruit1” Boolean is set to false and second fruit is loaded by setting the Boolean “fruit2” to true and “SplashSound” is being played:

if (evnt.key.keysym.sym == SDLK\_SPACE) {

if (fruit1 == true)

{

fruit1 = false;

fruit2 = true;

audioDevice.playSound(SplashSound);

}

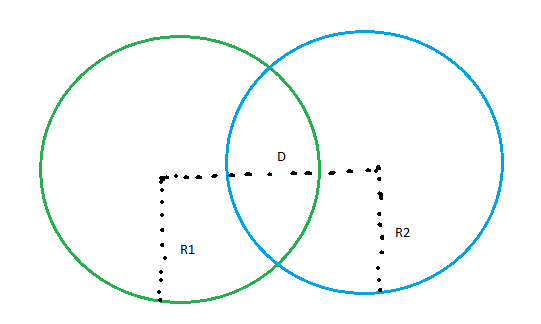
This is done for the rest of the fruits as well.

### **Collision()**

The collision method is used to detect when the collision happens between objects. It is done by calculation the distance between objects based on both of the objects x, y and z axis positions.

A function is used to find out whether the distance is equal or less than a radius and if it is then Boolean “collision” is set to true to say that the collision has indeed happened.

Following picture displays how this collision works and is detected based on these variables:



It can be seen that the distance is calculated between centres of both bounding spheres and both of the radiuses can be used to do so.

### **transformObject()**

This method is used to transform either position, rotation or scale of the objects that are used in the draw method. For this particular method, if function is used to detect which of the fruits is displayed by using Boolean for each fruit. Depending on which fruit is being displayed at the time, set position, rotation or scale is set and transformed as necessary.

### **drawGame()**

Draw method is used to too draw objects, shaders, text as well as setting the background colour and radius for the bounding sphere.

To set the background following code was used and correct values were researched online to set the colour that is necessary for the game:

gameDisplay.clearDisplay(0.0f, 0.1f, 0.0f, 0.0f);

This code sets the background colour to dark green, so it fits with the team of the game.

In this method set text is also loaded before the has been started by making sure that the “start” Boolean is false.

Text is loaded by setting the position for it, updating the shader and setting up the string for the text and making sure that it is being drawn by the right size and font:

transform.SetPos(glm::vec3(15.0, 20.0, 20.0));

shader.Update(transform, myCamera);

std::string text;

text = "Fruit Splash! - The Game";

drawText(text.data(), text.size(), 0, 0);

This code was used to draw the name of the game in the first menu screen.

Also, this method is used to draw necessary fruit when one of the fruit Booleans is set to true. If it has been set to true then, object is being transformed the way it is necessary by using transformObject() method, texture is binded, shader is updated and object is drawn depending on which fruit it is, different objects and textures are being used.

Following is an example for code used for the first fruit:

if (fruit1 == true) {

transformObject();

texture.Bind(0);

shader.Update(transform, myCamera);

object1.draw();

object1.meshSphere;

transform.GetPos();

}

This method is also used to bind shaders, set movement counter and swapping the buffers:

shader.Bind();

move = move + 0.03f;

gameDisplay.swapBuffer();

## Fruits

### Variables

* A Vector3 “position” that holds the world position of the fruit
* A vector3 “direction” that hold the direction of the fruit
* A Boolean “fruit1”, “fruit2”, “fruit3”, fruit4”, “fruit5” that lets the program know if the fruit is active in the world
* Object models for each fruit
* Textures for each fruit model

Fruit contains the following methods:

### **TransformObject()**

This method takes in objects position, rotation and scale based on x, y and z axis.

For the position it sets the position based on axis and moves it downwards on y axis in different set speeds.

As for rotation, it changes z rotation to make the object look more three dimensional.

Scale is set to a static value throughout the gameplay based on the object.

Following is an example of a code being used to move the object as well as change its rotation:

transform.SetPos(glm::vec3(50.0, 20.0 \* cosf(move) \* 0.5, 20.0));

transform.SetRot(glm::vec3(0.0, 0.5, move \* 1.0));

transform.SetScale(glm::vec3(0.5, 0.5, 0.5));

Here it can be seen that position is set and then changed on y axis by using the set move float and using cosf option and multiplying it by that.

Also, rotation is set and then its z rotation is changed by using set move float as well.

### Texutres:

Textures are used to load textures for each model that is being used.

### **Init()**

This method makes sure that textures are initialised so that they can be used.

It sets integers for width, height and number of components.

It also loads the image used for the set texture and stores the necessary data.

As well as makes sure that appropriate error is display if texture fails to load.

It is also used to bind textures and wrap texture outside the set width and height and uses linear filtering for both minification and magnification:

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

It also sets variables for the texture:

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA, width, height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, imageData);

### **Bind()**

This method binds the textures.

It makes sure that one of the 32 textures are being used.

It sets active texture unit and sets type and texture that has to bound to the unit:

glActiveTexture(GL\_TEXTURE0 + unit);

glBindTexture(GL\_TEXTURE\_2D, textureHandler);

### Meshes:

Meshes are used for loading models from set objects.

### **init()**

This initialising method is used to push back positions, texture coordinates and indices for models:

for (unsigned int i = 0; i < numVertices; i++)

{

model.positions.push\_back(\*vertices[i].GetPos());

model.texCoords.push\_back(\*vertices[i].GetTexCoord());

}

for (unsigned int i = 0; i < numVertices; i++)

model.indices.push\_back(indices[i]);

### **initModel()**

Model initialising method is used to initialise models.

It is done by generating vertex array and storing it into the vertex array object and binding the vertex array object.

This method also generates the buffer based on array data and buffers:

glGenBuffers(NUM\_BUFFERS, vertexArrayBuffers);

It makes sure that OpenGL knows that the type of data that buffer is using:

glBindBuffer(GL\_ARRAY\_BUFFER, vertexArrayBuffers[POSITION\_VERTEXBUFFER]);

It also moves the necessary data to the graphics card:

glBufferData(GL\_ARRAY\_BUFFER, model.positions.size() \* sizeof(model.positions[0]), &model.positions[0], GL\_STATIC\_DRAW);

### **loadModel()**

This method sets indexed model to be loaded object model from the set filename and it also makes sure that this model is initialised:

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

initModel(model);

### **draw()**

This draw method binds vertex array object and draws elements:

glBindVertexArray(vertexArrayObject);

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

### **meshSphere()**

This method includes setters for both position and radius of the collision sphere:

this->position = position;

this->radius = radius;

### Shaders:

These shader methods are used to render shaders so that they can be used to load objects in appropriate way for the project.

### **init()**

This initialising method is used for creating shader program as well as creating vertex and fragment shader.

It is done by loading these shaders from set shaders previously resources or provided and included from resource folder in the project:

shaders[0] = CreateShader(LoadShader("..\\res\\\shader.vert"), GL\_VERTEX\_SHADER);

shaders[1] = CreateShader(LoadShader("..\\res\\shader.frag"), GL\_FRAGMENT\_SHADER);

This method also makes sure that set shaders are added to the shader program.

It also makes sure to bind attributed location based on positions and texture coordinates of the shaders.

It also creates executables that run on the graphics shader.

If there are any issues with loading or linking shaders, then necessary errors are displayed.

Initialising of shader method also makes sure that location of uniform is associated with the program.

### **CreateShader()**

This method creates set shaders:

GLuint shader = glCreateShader(type);

It also makes sure that there are no issues with creating the shader and if there are then appropriate error is displayed.

This method also makes sure that necessary source code is sent, and it compiles shader code within OpenGL:

glShaderSource(shader, 1, stringSource, lengths);

glCompileShader(shader);

Then shader is returned.

### **LoadShader() & CheckShaderError()**

Shader loading method is used to load shaders from a file.

If there are any issues with loading shaders, then error is displayed in the console with the name of the set shader.

As for shader error checking method, it checks for any error and issues with loading and rendering the set shaders.

## Display

### Variables

Display contains following methods:

### **GameDisplay() and GameDisplay() Deconstructor**

This method is used to create the game screen with set height and width values:

screenWidth = 1024.0f;

screenHeight = 768.0f;

Deconstructor for this method makes sure that the game screen is closed after the program has been exited:

SDL\_DestroyWindow(sdlWindow);

SDL\_Quit();

It also includes getters for both width and height of the game display.

### **returnError()**

This method displays an error if for any reason game display cannot be opened and it lets the user know to press any key to quit the console screen if that is the case.

### **swapBuffer()**

This method makes sure that the buffers are being swapped:

SDL\_GL\_SwapWindow(sdlWindow);

### **clearDisplay()**

This method is used to set clear colour and depth buffer.

This method can be later used to set the background colour of the OpenGL screen.

### **initDisplay()**

This method makes sure that everything is initialise in terms of SDL screen.

It also sets attributes for it and then creates the SDL window with set variables like name of the window, position, width and height:

sdlWindow = SDL\_CreateWindow("Fruit Splash", SDL\_WINDOWPOS\_CENTERED, SDL\_WINDOWPOS\_CENTERED, (int)screenWidth, (int)screenHeight, SDL\_WINDOW\_OPENGL);

It also makes sure that if SDL window is not opened or if there are any issues if GLEW or GL context is not created then appropriate errors are displayed.

This method also includes GL\_CULL\_FACE to make sure that faces that are not facing the camera are not rendered so that too much memory is not being used when it is not necessary.

## **Sources**

The following sources were used throughout the creation of the game:

### Player Camera

The player camera was created using Lab 4, that was created by Bryan Young.

Images used to display frustum and image planes were sourced from Tutorial 3 answers, created by Bryan Young.

### Audio

The audio was created by using Lab 8, that was created by Bryan Young.

Images used to display OpenAL structure and Cartesian coordinate systems were sourced from Week 8 Lecture from Bryan Young.

### Collision Detection

The collision detection was done based on Lab 7 created by Bryan Young.

Also, image used to display the collision detection was used from example from Bryan Young.

## Text

To draw the text for menu screen and ending screen following YouTube link was used:

<https://www.youtube.com/watch?v=elE__Nouv54>

Also, following link was researched to find more information for creating the necessary text:

<https://rbellek.wordpress.com/>

### Textures

To create the code for the texture Lab 3 from Bryan Young was used.

Code used to initialise these textures was based on Lab 4, also created by Bryan Young.

### Meshes

To create the code used for the meshes Lab 4 that was created by Bryan Young was used.

For the code that was used to load these models Lab 4 was also used.

### Shaders

To create the necessary shader code, Lab 3 was used as well as other resources created by Bryan Young were used.

To create code to load these shaders Lab 4 was also used.