

Games Programming 3

Coursework Code Explanation

Fraser McFarlane

Matriculation Number: S1434566

Computer Games (Software Development)

*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.*

Fraser McFarlane

Table of Contents

[About 4](#_Toc503146993)

[How to Play 4](#_Toc503146994)

[Known Issues 4](#_Toc503146995)

[Constants Header File 5](#_Toc503146996)

[SDL Display class 6](#_Toc503146997)

[SDLDisplay.h 6](#_Toc503146998)

[SDLDisplay.cpp 7](#_Toc503146999)

[Camera Class 9](#_Toc503147000)

[Camera.h 9](#_Toc503147001)

[Getters 10](#_Toc503147002)

[Sound Class 11](#_Toc503147003)

[Sound.h 11](#_Toc503147004)

[Sound.cpp 12](#_Toc503147005)

[Sound Manager Class 15](#_Toc503147006)

[Sound Manager Header 15](#_Toc503147007)

[Sound Manager Cpp 16](#_Toc503147008)

[Transform Class 18](#_Toc503147009)

[Transform Header 18](#_Toc503147010)

[Getter and Setter 19](#_Toc503147011)

[Enemy Class 20](#_Toc503147012)

[Enemy.h 20](#_Toc503147013)

[Enemy.cpp 21](#_Toc503147014)

[Bomb Class 22](#_Toc503147015)

[Texture Manager Class 22](#_Toc503147016)

[Texture Manager.h 22](#_Toc503147017)

[Texture Manager.cpp 23](#_Toc503147018)

[Model and Vertex Manager Class 24](#_Toc503147019)

[Model Manager.h 24](#_Toc503147020)

[Model Manager.cpp 25](#_Toc503147021)

[Light Class 27](#_Toc503147022)

[Shader Manager Class (Extension) 27](#_Toc503147023)

[Shader Manager.h 27](#_Toc503147024)

[Shader Manager.cpp 28](#_Toc503147025)

[Input Manager class (Controller Support Extension Included) 31](#_Toc503147026)

[Input Manager.h 31](#_Toc503147027)

[Input Manager.cpp 31](#_Toc503147028)

[Controller Support 31](#_Toc503147029)

[The Game Class 33](#_Toc503147030)

[theGame.h 33](#_Toc503147031)

[theGame.cpp 35](#_Toc503147032)

[Main.cpp 39](#_Toc503147033)

[References 39](#_Toc503147034)

[Bibliography 39](#_Toc503147035)

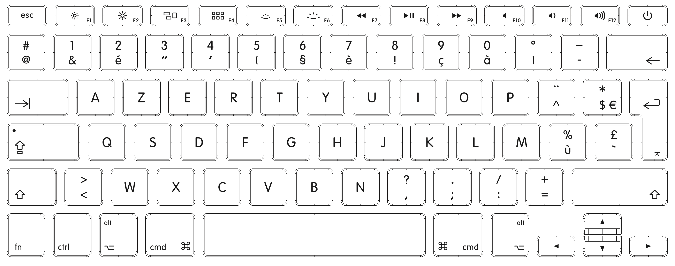
[Assets References 40](#_Toc503147036)

# About

This game challenges the player to survive as long as possible as an ever-growing horde of Aliens approach you. Protect yourself by firing bomb at the enemies. Don’t let them hit you or get behind your line of defence.

## How to Play

The game will begin, and you will be free to interact with the controls. When you are ready to start press TAB or START (Xbox 360/One Controller) to begin the waves.



**Start**: Begin Wave **Tab**: Begin Wave

**Left Analogue**: Move **WASD**: Move

**A**: Fire **SPACE**: Fire

**B**: Swap 1st/3rd person view **C**: Swap 1st/3rd person view

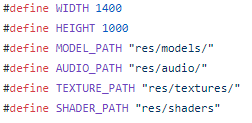
**Select**: Mute toggle **M**: Toggle Mute

## Known Issues

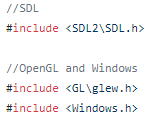
Font only appears on the first frame so as substitute, important information such as health and wave number are displayed in the console.

# Constants Header File

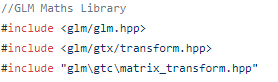
The constants header file contains includes and definitions for libraries and data that are used in many classes throughout the program. This is made to organise where all our external libraries are coming from.



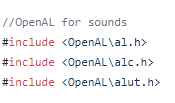
Firstly, we define integer values for the width and height of the SDL window that will be used. Then 4 strings are defined for the file paths for the resource folders for models, textures, audio and shaders. These will be used when calling our constructors for these objects.



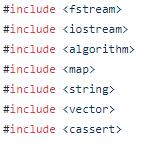
For access to our SDL functions that will be used for creating the program window and receiving input, we must gain access to the SDL2 library. This is done using #include call. As well for our functionality from Windows and OpenGL, we gain access to the Windows.h file and the Glew.h file for Windows and OpenGL respectfully.



A maths library is needed for creating vectors, transforms and matrices. Here we gain access to the appropriate GLM hpp files.

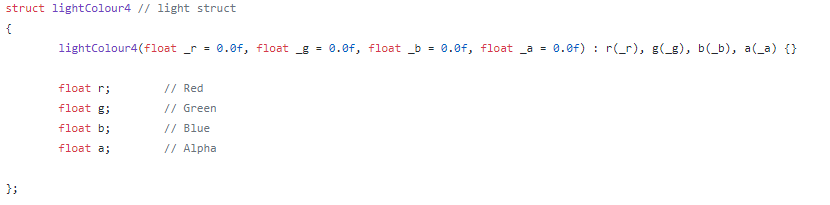


For sound we are using the OpenAL audio library. Above shows the include calls for what header files we are utilising to create our Sound and Sound Manager objects.



The last of the include files provide functionality for file streams, input streams, collections, strings, vectors and vector arrays and debugging tools. Also within this file we state that we are using the namespaces for std and glm so that we do not need to call these namespaces every time we need access to their identifiers.

This class also contains 2 structs for creating colour and light colour. The light colour struct, named lightColour4 takes in 4 float arguments for the red, green, blue and alpha values of the colour vector. Local member float variables for the colour values are set to the passed in value.



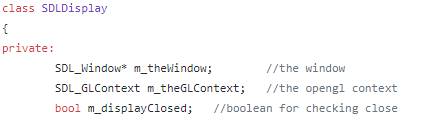
The same process is carried for the declaration and definition of the colour struct.

# SDL Display class

This class is responsible for handling the program window, opengl context, window buffer swapping, setting projection types and initialling the window display.

## SDLDisplay.h

The header file for this class contains the include for Constants.h file so that it can access the SDL, OpenGL and Windows libraries to create the window and display.



The class first declared three private variables. Fist is an object pointer of SDL\_Window names m\_theWindow. This will hold the window object of the program. The second is a SDL\_GLContext pointer that will hold the OpenGl context object. The last is a private Boolean that will hold whether the display is open or closed. This will be used to decide how to handle the destructor of the program or running of the program.



The public declarations first declares the default constructor of the class. The override constructor is declared and takes in 3 arguments, an integer for the width, another for the height and a constant string reference for the name of the window that will be displayed. The destructor of the class is then declared.



Three public methods are then declared that will set the clear colour of the display, swap the buffers for the window and return the display closed Boolean. The Clear Display method returns void and takes in three floating point variables for the red green blue and alpha values of the colour. The swap buffer method takes zero arguments and returns void. The Is Closed returns a Boolean value and takes 0 arguments.



The setOrthograohicProj method takes in integers for width and height and returns void. This method sets the projection to orthographic so that text can be printed on the screen. The resize function has the same parameters and return. This is called for setting the dimensions and projection type when the window is resized. The setMVP is used for changing the projection back to model view. This also has the same return type and parameter values.

## SDLDisplay.cpp

**SDLDisplay (int windowWidth, int windowHeight, const std::string& windowName)**

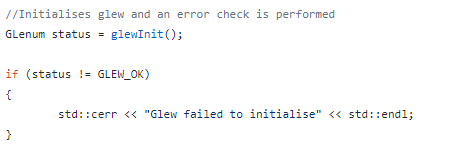
The constrictor scope first calls SDL\_Init function and passes in the flag SDL\_INIT\_EVERYTHING. This allows all the SDL features to be initialised.



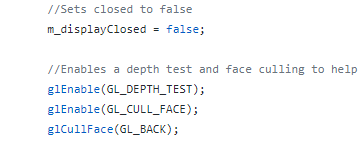
Attributes are then set for the SDL window. This includes minim sizes of bits for the colour channels, buffer channels and depth buffer sizes.



The window object is then defined by calling the SDL\_CreateWindow function which takes in parameters for the name, flags for the x and y centre of the window, the width and height and the type of window that is being initialised (An SDL OpenGL window). The opengl context object is then created from the window object which is passed through the SDL\_GL\_CreateContext method.



OpenGL is initialised by calling glewInit function from the GLEW library. This method is called, and the return value is set to a GLenum flag named status. This is used to check if glew has been initialised without errors. An if statement is called that checks if the status does not equal the enum flag GLEW\_OK. If this is true then an error is sent to the console stating that glew has failed to initialise.



The end of the constructor scope calls glEnable () for enable depth test and culling. The depth test compares depth values and updates based on the depth buffer. The culling face flag allows the window to cull polygons based on the coordinates within the window. Back facing polygons are culled by calling glCullFace and passing in the enum flag GL\_BACK.

**~SDLDisplay ()**

The destructor calls three methods, one for deleting the opengl context (SDL\_GLDeleteContext) which takes in the context object as a parameter. The second deletes the window object (SDL\_DestroyWindow) which takes in the window object. Finally, SDL\_Quit () is called which cleans up all the attributes that were initialised in the constructor.

**ClearDisplayColour (float r, float g, float b, float a)**

This method scope contains two methods. One for setting the clear colour which values have been passed into the parameters of this method, glClearColour (). This method takes in the rgba values. This sets the colour. The glClear () method is called that takes in two flags for the colour buffer and depth buffer;

GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT.

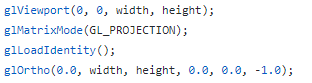
This sets the values of these attributes back to their defaults.

**Is Closed ()**

This method simply returns the m\_displayClosed bool value.

**setOrthographicProj (int width, int height)**

glDisable is called twice at the start of this scope. This method disables functionality of opengl. GL\_LIGHTING and GL\_DEPTH\_TEST are passed into these methods to display depth tests and lighting.



The viewport is then set of the opengl context by calling glViewport () and passing in 0,0 for the top left of the window, and passing in the width and height values. The matrix mode is then set to projection from glMatrixMode (GL\_PROJECTION). The current matrix is then replaced by the identity matrix by calling glLoadIdentity (). The identity matrix is then multiplied by an orthographic matrix when calling glOrtho (). This allows a render pass of 2d orthographic projection. This will allow text to be rendered on screen.

**SwapBuffers()**

The buffer is then swapped for a window by calling SDL\_GL\_SwapWindow () and passing in the window object. This allows the window to be updated for rendering.

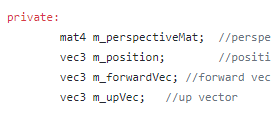
**setMVP(int width, int height)**

This method mostly reverses all the changes make in the setOrthographicProj () method. Depth tests and lighting is enables by calling glEnable () and passing in the flags values. The stencil buffer is cleared from glClearStencil (). Comparisons for the depth buffer values are carried out by calling glDepthFunc () and passing in GL\_LEQUAL which checks if the current value is less than or equal to the value stored in the buffer. The resize method is then called.

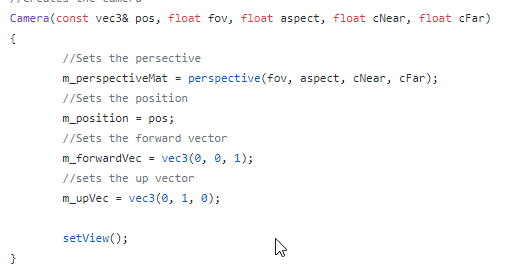
# Camera Class

The camera class is used for setting up all the vectors and matrices for creating the appropriate projection. This class includes the Constants header file to gain access to the GLM maths library so that vectors, transforms and matrices can be utilised.

## Camera.h



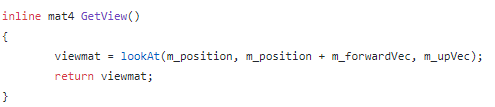
The camera class first declares the private vectors and matrices for the perspective matrix, position, forward and up vectors. The default constructor is first declared and defined within the public declaration section.



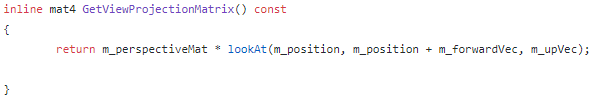
The override constructor used for initialising the camera object takes in parameters for the position, field of view, aspect ratio, and the near and far clipping plain. Within the scope of the constructor, the perspective matrix is first defined using the GLM perspective method which will create and return the matrix. This method takes in the field of view, aspect ratio and near and far clipping plane float variables. The position variable is set to the passed in argument value. The forward and up private variables are set to (0, 0, 1) and (0, 1, 0) respectively. The end of the constructor calls the setView () method.

## Getters

Methods are available from the camera class for receiving values for the private member variables. GetForwardV () and GetUpV() both returns their appropriate vectors.



The GetView () is an inline method that returns the view matrix. This sets the public viewMat variable to the return of the GLM lookAt () method. This method takes in the position vector, the position vector added by the forward vector and then finally the up vector. This creates the view matrix. This is then returned from the function.

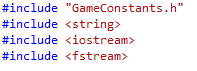


The GetViewProjectionMatrix () method returns the view projection matrix. This multiplies the perspective matrix by the view matrix and returns the product.

# Sound Class

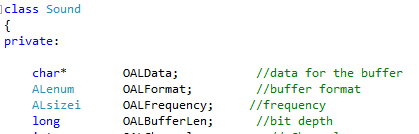
The Sound class is made for handling all sounds and audio files within the game. This includes loading sound data and files, playing and stopping audio as well as running clean-up functions. Audio is handled by the Open AL library. All code will be explained in detail within this section.

## Sound.h

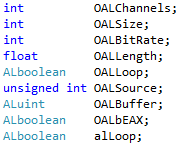


The sound header file first includes the GameConstants.h file which holds all the appropriate Windows, OpenGL, maths functions etc, include libraries. Within the GameConstants file, the Open AL library header files are accessed (alpha, alc.h and alut.h) which contain the appropriate functionality for processing audio within this project.

The sound header file also includes the string, iostream and fstream libraries that are used for string parsing, input and output stream processing and file stream processing respectively. The file also uses the ‘std’ namespace as strings will be used heavily within this class.



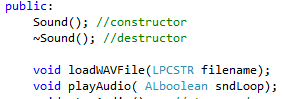
The sound class is declared here. Within the class scope, private elements are declared first. For holding the sound buffer data, a char type variable pointer named OALData is declared. For holding the format of the buffer, a ALenum type named OALFormat is made. The sound frequency will be contained within a ALsizei variable named OALFrequency. A long type variable named OALBufferLen will hold the length of the sound buffer. These variables are used extensively when loading the WAV sound files using the Open AL functions.



Further private variables are declared within this file that hold the sound channels, size, the bit rate of the audio, audio length, a Boolean for audio loops, the audio sources, Alunite type for the buffer object and a Boolean check for EAX 2.0 support.



We then have a private function for loading WAV file data. The function takes in references for the ifstream variable for the filename of the source, a string for the file name and an unsigned int for the size of the file.

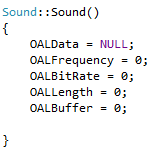


Public functions are now declared within the header file. First, the sound class constructor and destructor and declared which take zero parameter values. A function named loadWAVFile is then declared which takes in a long pointer string for the filename and returns void. A method for playing the audio is then declared named playAudio () which returns void and takes in a parameter value AL Boolean for determining audio looping.

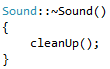


Two more methods are declared for stopping the audio and deleting all data relating to the audio, both of which return void and take 0 arguments named stop Audio () and cleanUp ().

## Sound.cpp

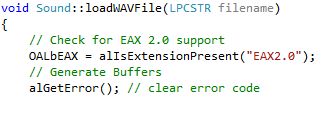


The sound constructor’s scope initialises the OALData object to null and the frequency, bitrate, length and buffer variables to 0.

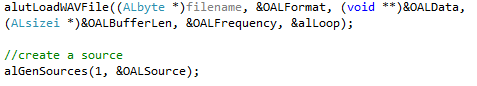


The destructor for the sound class runs the cleanUp () method that will be described further in this section.

**LoadWAVFile ()**

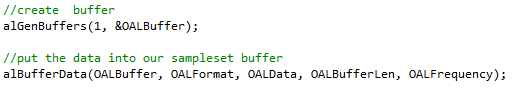


Within the scope of the loadWAVFile method, the Boolean for checking for EAX 2.0 support ‘OALbEAX’ is defined from the return value of the Open AL alIsExtensionPresent () method. This method takes in a const ALChar value for the name of the extension that is being checked. The purpose of this method is for checking that the sound context extension is available. The method will return flags for true or false depending on extension availability. The alGetError () method is called to return any current error states of the buffer, if there is any.

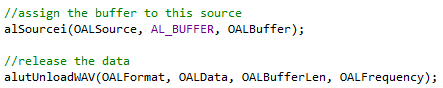


The alutLoadWavFile () is used for loading the audio files to the program. The method takes in the filename parameter as an argument which is cast as ALbyte format, the OALformat reference, the reference to OALData, the OALBufferLen used as an offset by casting at ALsizei, The OALFrequency and the alLoop.

A source for the sound file is then created using the alGenSource () method. This takes in parameters for the number of sources to be generated (1) and a pointer to where the source will be stored.

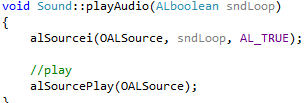


The buffer object is generated using the alGenBuffers () function, this method takes in an integer for the number of buffers that are required (1) and the object for where the buffer will be stored. The data for the buffer is then placed into our buffer by calling the alBufferData () method. This method takes in the buffer object (OALBuffer), the format type of the sounds (OALFormat), the pointer to the audio data (OALData), the size of the audio (OALBufferLen) and the frequency of the audio (OALFrequency).



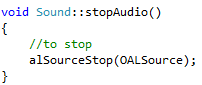
The alSourcei () method sets the buffer object to a source. The first parameter value of this method is the source which attributes are being set to the buffer, the second a flag for the name of the attribute that is being set, in this case it is a buffer (AL\_BUFFER). The third parameter takes the value that the source is being set to (OALBuffer). The data is then cleared by calling alutUnloadWAV which takes in the format, data, buffer audio size and frequency variable objects.

**PlayAudio ()**



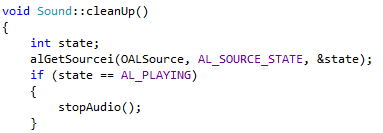
The play audio method first calls the alSourcei () method within its scope. This method takes in the source object, the name of the attribute being set (Looping attribute) and the value that this parameter attribute is being set to (true). The audio source is then played by calling alSourcePlay (). This method takes in the source object to access the audio data that is required to be played.

**StopAudio ()**

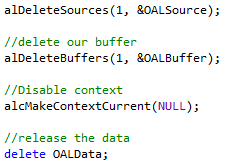


This method has a single call which is alSourceStop (). This takes in the source object as a parameter argument. This allows whatever audio that is currently being played from that passed in source to be stopped.

**CleanUp ()**

****

This method first declares a local integer variable ‘state’ which will hold the current state flag of the audio source. The state is captured by calling alGetSourcei () that takes in the source object, the attribute flag that is required (in this case AL\_SOURCE\_STATE) and then the ‘state’ variable reference that will state data will be passed into. An if statement is then made that will check to see if the state is equal to playing. If this is true, then the scope will call the StopAudio () function.



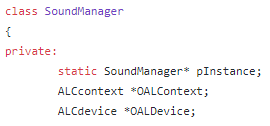
The source object and buffer data are then cleared by calling alDeleteSources and alDeleteBuffers accordingly. These two methods take in the number of sources or buffers that are required to be deleted and then the references to the buffer or source objects. The sound context is then disabled and set to null using the alcMakeContextCurrent and passing in NULL to the parameter. The sound data is finally cleared using the delete call.

# Sound Manager Class

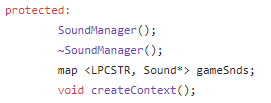
The sound manager class is used for adding sounds to the program. This also includes receiving sounds to play them when required.

## Sound Manager Header

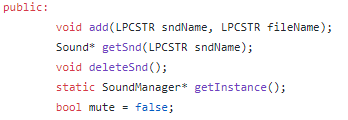
The sound manager class includes the Sound and Constants header file. So that the appropriate libraries can be accessed as well as sound object functionality.



The class first declares its private variables. This includes a static sound manager object pointer named ‘instance’. This is used to adhere to the Singleton Design Pattern that has been put in place for this class. As this class uses the OpenAL sound library a context and device object must be declared, both of which are pointers to their desired type ALCcontext and ALCdevice.



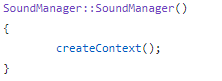
For protected members, the constructor and destructor are declared. A map container is created which uses a LPCSTR as the key and a Sound object pointer that will be the type of object that will be mapped. This is named gameSnds and will contain all the sounds within the game. A method that returns void and has zero parameters is also declared for creating the openAL context, named appropriately.



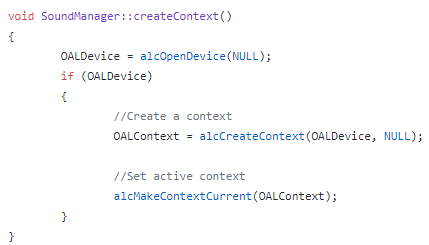
For public members, a method that will add the sounds to the map is declared. This returns void and takes in a LPCSTR for the name of the sound and another for the filename. This method is simply named ‘add’. A method that returns a Sound pointer and takes in the LPCSTR for the name of the sound is declared and is a getter for the sound object. When a sound is required to be deleted from the container, the method deleteSnd is declared that returns void and takes zero parameter values. Another getter is made that returns the instance of the Sound manager, named getInstance. Finally, a Boolean is declared and defined for muting the sounds. This is initialised as false.

## Sound Manager Cpp

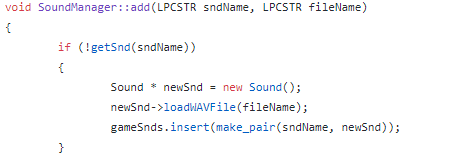
The source file first sets the instance to equal null and includes the sound manager header file.



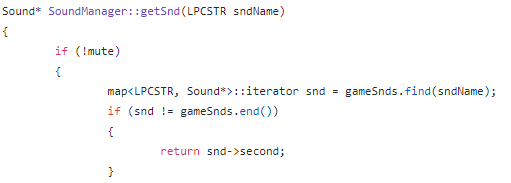
The constructor calls the create context method that will be discussed later in this section. The getInstance method of this class first checks if the instance is equal to NULL, if this true then the instance variable is set to equal a new sound manager object. Outside of the if statement scope, the instance is returned by this method.



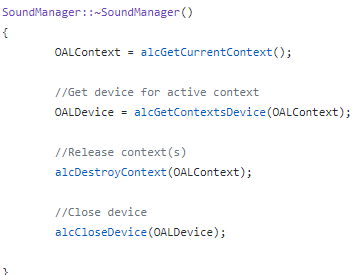
When the createContext method is called, the OALDevice variable is set to equal the return value of the alcOpenDevice(NULL) method. This method allows the device to be opened and used. An if statement is then made that checks if the OALDecive is not null. If this is true then the OALContext is set to the return of the alcCreateContext () method. This method returns a context and takes in a device object as a parameter. This context is then activated for use by passing the context object into the alcMakeContextCurrent () method.



The add method’s scope is made of an if statement that checks if the sound does not already exist by calling the getSnd method and passing in the name. If this is true then the sound can be added to the collection. This is done by first creating a new sound object. This sound is then set to equal the return of the loadWAVFile () method from the sound object class. The gameSnds collection then calls the insert method and which adds the sound to the collection, and makes the name is used as the key in the map collection.



The get sound method first checks if the mute is set to false. If false then an iterator for the gameSnds collection is made and is set to equal the return of the ‘find’ method. If so then the sound is returned.

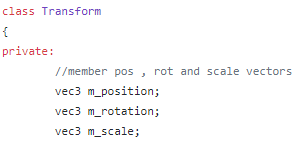


The destructor for this class sets the OALContext object to the current context by calling the alcGetCurrentContext () method. The same step is carried out for the device object by calling alcGetContextDevice () method and passing the context into this method. alcDestroyContext and alcCloseDevice is called for deleting and closing the context and device.

# Transform Class

## Transform Header

The transform header file contains both the class’ member declarations and definitions so no source file is used for this class. The class first includes the Constants header file so that it can access the functionality of the GLM maths library for the use of vectors, translations and rotations as examples.

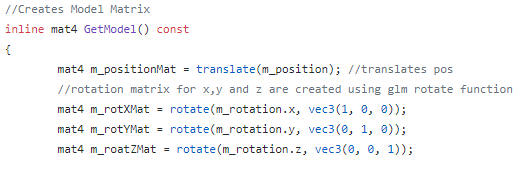


The class first declares 3 private vector 3 types from the glm namespace. This is to hold the position, rotation and scale vectors of the transform object.

For public members of the class, the constructor is first declared and defined within the scope. The constructor takes in 3 arguments. A constant vector for position, rotation and scale. The position, scale and rotation private variables are then set to these 3 passed in parameter arguments which initialises all the transform information.

**GetModel ()**

An inline method for calculating the model matrix is then declared and defined within the class. This has a return type of mat4 (GLM data type for 4x4 matrices) and takes in no parameter arguments.



Within the scope, a position matrix is first created and set to the return of the GLM translate method that takes in the member position vector 3. This will return the position matrix of the transform object. The same method is carried out for the rotation; however, the rotation must first be split up into 3 different matrices as the GLM rotate function can only work on 1 vector value at a time. So, 3 matrices are made for the rotation x, y and z. These matrices are set to the return value of the GLM rotate function which takes in the desired vector value (x,y or z) for the angular rotation, and vector 3 for the scalar.



A final rotation matrix is calculated by multiplying these separate rotation matrices together.

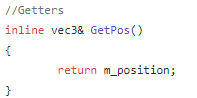


The scale matrix is then made by using the GLM scale method and passing through the private scale variable. Finally, the Model Matrix is created by multiplying the rotation, position and scale matrices together. This value is then returned from the method.

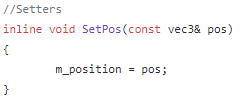


## Getter and Setter

The rest of the class declaration contains getters and setters for the private vector variables for scale, position and rotation. These methods are all similar so only the getters and setters for the position variable will be shown.



The getter for the position is an inline function that return a vector 3 reference to the position vector.



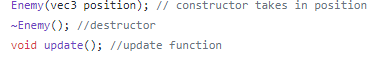
The setter is an inline method that returns void. It takes in a constant vector 3. Within the scope the position is set to the passed in argument vector.

# Enemy Class

This class is responsible for handling the enemy game objects.

## Enemy.h

The enemy class first includes the transform.h header file as the object needs to have a transform object associated with it. The class also includes the Constants header file to access the GLM maths library. All declarations in this class are public members.



The enemy object constructor is declared and takes in a single parameter value that is a vector 3, this is used for setting the transforms position value. The destructor is declared after the constructor.

The update function will be used for updating the enemies position every frame. This takes zero arguments and returns void.

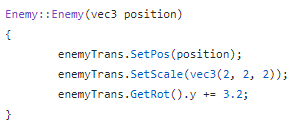


Here the enemy transform object is declared named enemyTrans. A bool value named isActive declared and set to false. This will be used for determining if the object should be rendered or not.

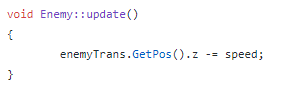


Two methods are then declared for handling collision detection between this object and another. The SphereCollision will check if a collision has occurred so it will return a Boolean value for true or false. This method takes in 2 parameter values a vector for the model position that this object is being tested against and a float variable for the mdl radius that this object is being tested against. The CalcLengthSQRD method will calculate the length of this object and the model it is being tested against squared. This takes in a vector three for the length between the two objects. This value will return a floating-point variable. After these declarations, a float variable for the enemy speed is set to 0.5.

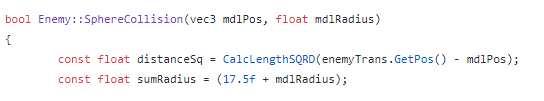
## Enemy.cpp



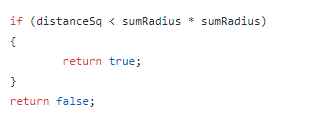
The constructor for this class first sets the enemy transform data. The position of the transform is set by calling the transforms SetPos () method. This [parameter argument for position is passed into this method. The scale is set to 2 for every scalar value as the model that is being used for the enemy was a too small. The rotation value of y is then set to the desired rotation. The destructor scope is empty for this class.



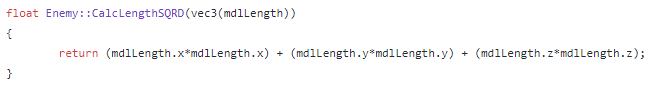
The update function will get the enemy transform and call the GetPos() method from the transform class and get access to the z value. This will then take be set to its value minus the speed value. The will allow the enemy to move towards the player position.



The sphere collision method const declared and defines 2 local float variables. The first is a constant which is hold the distance squared. This is done by calling the CalcLengthSQRD method and passing in the enemy transform position, minus the passed in position of the model is being tested upon. The return value of this method is set to the distandSq local variable. The second float variable will hold the sum of the radius of both models. For the enemy object, the radius is a hard-coded value as the technique used for model loading cannot provide the radius of the model.



To check if the collision has occurred, an ‘if’ statement is make that takes the distance value and if it is less than the sum of both radius squared. If so then the method returns true. If not then the method returns false.



The CalcLengthSQRD method returns the sum of all the length vector values squared.

# Bomb Class

The bomb class is like the enemy class, so the header and source file will not be described thoroughly in this section. The bomb class only differs from not having methods for collision detection and that that when the update method is called, the z transform position value is increased by a speed value and not decreased.

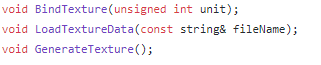
# Texture Manager Class

This class handles the creation, loading and generation of textures within this program.

## Texture Manager.h

The header file for this class includes the Constants.h file so it can access the appropriate OpenGL function from the GLEW library. The class first declares its private members, one of which is a GLuint type for holding the texture object, named m\_texture. Three integer variables are also declared for holding information about the image being loaded. These are for the width, height and number of components. Finally, an unsigned character pointer for holding the image data is declared named image Data.

The public members include a default constructor and a destructor, and an override constructor that takes in a constant string for the filename of the texture file.



Three public methods that all return void are also declared. These methods, in order of appearance above, do the following tasks. Takes in an unsigned integer variable for the unit number of the texture and checks that it is within range then binds the texture to the program. The texture is loaded using the filename string and then the texture is generated.

## Texture Manager.cpp

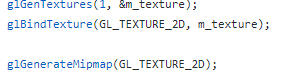
The constructor for this class first calls the LoadTextureData () method and passes in the filename argument from the constructor. After this the GenerateTexture () method is called. Following these two methods, the image loader class is used to call stbi\_image\_free () function which will clear the image data after it has all been processed and bound to the buffer.

The LoadTextureData () method takes the imageData variable and defines it as the return value of the image loading class’ stbi\_load () function.

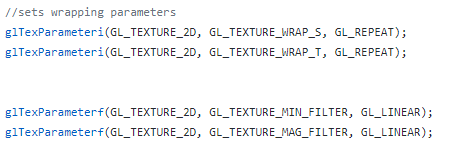


This method takes in the filename, converting it to a string and the width and height references to hold their appropriate information on the image. After this method call, the image data is checked to see if it is equal to NULL. If true then an error message is sent to the console.

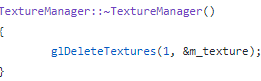
For processing the texture into opengl , the GenerateTexture () method first calls glGenTextures() which requires two arguments, the number of textures that are required to be generated from this method and a reference to the texture object that will hold the generated data.



glBindTextures () is then called that will bind the texture – m\_texture – to a certain target. In this case the target is defined by the GL\_TEXTURE\_2D target. glGenerateMipMap() is then called that will allow mipmaps to be generated to a certain target.



At this point in the scope, the texture attributes are now being set. The glTexParameteri method first sets the 2D attributes by setting the texture coordinate , s, to wrap around its target by repeating the texture. This same attribute is set for the texture coordinate, t. The texture filtering attributes are then set by allowing the minifying functionality to filter linearly. This is also set for the magnification filter.



The destructor for this class simply calls glDeleteTexture () which takes in the amount of textures and the texture object reference that are required to be destroyed.



The BindTexture() method will activate the current texture by calling glActiveTexure () taking in the unit number to specify the texture. glBindTexture () is then called again.

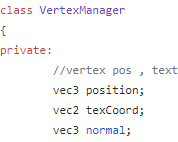
# Model and Vertex Manager Class

The header and cpp files for the Model Manager contains two classes. The Model Manager class, which is used for generating and loading models into the game, and the Vertex Manager class which is used when creating the models to hold the vertices.

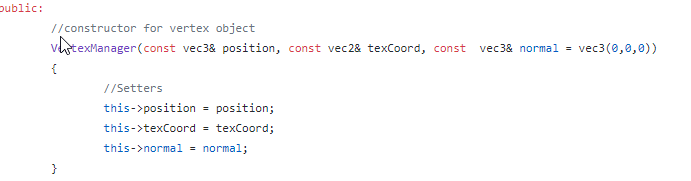
## Model Manager.h

The model manager header file first includes the constants file so that the maths library can be accessed. It also includes the obj\_loader (check references) header file that will be used for generating models into the game.

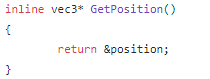
**Vertex Manager class**



The vertex manager first declared 3 private vector variables. A vector three for the position of the vertex, a vector 2 for the texture coordinates and a vector 3 for the normal of the vector.



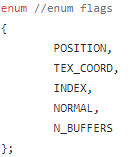
The vertex manager now declared and defines its public members. The constructor for the vertex manager object takes in a position, texture coordinate and normal vector. Within the scope, the private member vector values are set to the values of the passed in parameters.



Getters are then defined for all the private vectors. These function are inline and return pointers to the vectors.

**Model Manager Class**

The model manager members are only declared in this file, unlike the vertex manager’s which were defined. A void method named InitMesh () which takes in a single parameter object reference for an Indexed Model (Object type from obj\_loader) is declared.



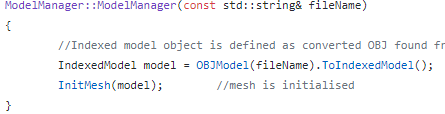
Enum flags are then declared that will be used in our Vertex Array Buffers for defining our position, texture coordinates, index and normals.



Two GLuint variables are made that will hold the vertex array object and the second will hold an array of our vertex array buffer objects. Another variable which is an unsigned integer that will hold the current draw count of the model vertices. A final private method is declared that returns void and takes in an Indexed Model object. The methods name is SendDataToGPU () and generates the model data and binds it to the GPU.

For public members of the class, a default constructor is declared as well as two overrides. One override takes in a Vertex Manager object, number of vertices integer, indices integer and the number of indices integer. The other override simply takes in a string value for the filename of the model required to be loaded. A final method is declared that returns void and takes zero arguments. This method is called DrawMesh () and its function is to render the mesh onto the screen.

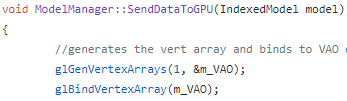
## Model Manager.cpp



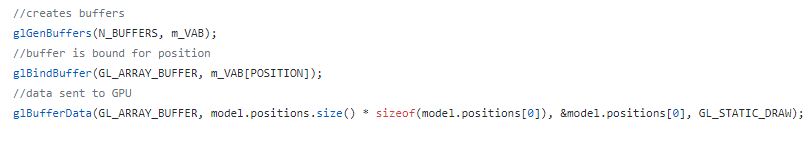
The model manager constructor that creates a model from a filename first defines an IndexedModel to equal the return of OBJModel method that takes in the filename and converts the raw OBJModel type to that of IndexedModel. The model object is then passed into InitMesh ().

The destructor calls glDeleteVertex arrays that takes in a parameter for the number of arrays and a reference to the vertex array object. This method simply deletes the vertex array object data.

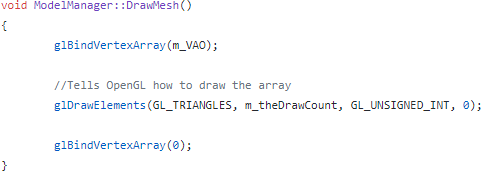
The InitMesh () method first sets the Draw Count variable to equal the size of the indexed model objects indices array size. The SendDataToGPU () method is then called, passing in the model object.



This method first creates the vertex array by calling glGenVertexArray and passing in the number of arrays that is required and a reference to the array object. The same is done for the glBindVertexArray () which binds the array object.



The buffers are created by calling glGenBuffers () this method takes in the enum of the number of buffers and the VAB array object that the buffer will be bound to. The positional buffer is then bound by calling glBindBuffer (). The appropriate enum is passed at the first parameter and the second the appropriate data from the VAB array. The buffer data is then initialised and stored by calling glBufferData (). The same steps are followed for the texture coordinates, normal and indices. The end of this method then calls glBindVertexArray () and passing in 0 to unbind the vertex array.



The draw mesh method first binds the vertex array object by calling glBindVertexArray (). The elements are drawn on to the screen by calling glDrawElements (). The parameters for this method state the type of elements that is needed, the draw count, the value type of the indices and the pointer to the location of the indices data.

# Light Class

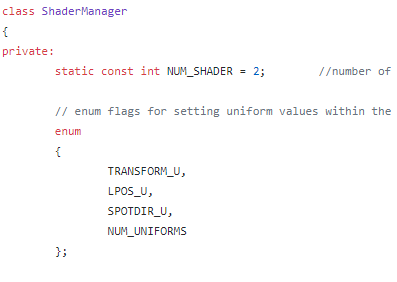
The light header class first includes the constants header file so that the light and colour structs can be accessed. GLM and OpenGL are required as vectors and GL floats are required.

The class’ scope first declares its default and override constructor. The constructor takes in a lightID, a colour structs for ambient, diffuse and specular. Vectors for position and light position. Floats for exponents, cut offs and attenuations. For private variables, the class’ private members for the values in the constructor.

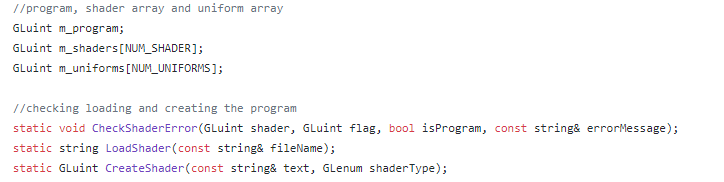
Within the light cpp file, the defaults constructor scope sets default values to the floats, light colour and vectors required to create the light. The override constructor that takes in the parameter values sets the private light information variables to the passed in values.

# Shader Manager Class (Extension)

## Shader Manager.h



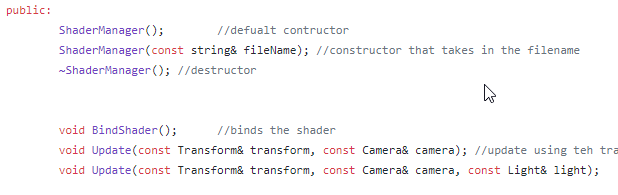
The shader manager first declares an integer for the number of shader files that are going to be used. Enum flags are also declared for shader uniforms such as the light position and direction, object transform and number of uniforms.



GLuint values for the shader program, shader and uniform array are declared with their appropriate array offset integer and enum. Two methods are made that will load the shader and create the shader. The load shader method only takes in a file name and the Create shader takes in a string of text and an enum for the shader type.



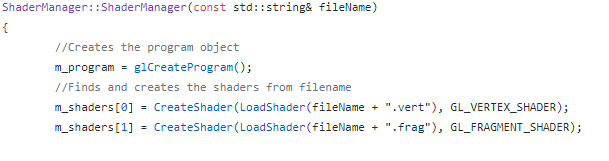
Two more methods that are void for return and argument parameters are made that set up the uniforms and attach the attribute.



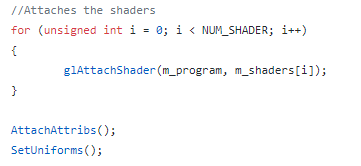
The private members include the constructors and destructor, a bind shader method and two update functions. One update method is for taking in a transform and a camera so that the view and object is updated. The override takes in an additional value for a light source.

## Shader Manager.cpp

**ShaderManager ()**



The constructor sets the program object to equal the return of the glCreateProgram () method which creates an actual shader program object. Shaders are added to the shaders array by setting the 0 and 1 positions to the return of the CreateShaders () method. This method takes the Load Shader method as a parameter. This takes in a filename and finds the vertex and fragment filetype of these names. The type of shader is defined at the end of the parameters by calling the specific enum.



The number of shaders are iterated through and the shaders are attached to the program object. The AttachAttribs and SetUniforms methods are then called.

**LoadShader ()**

The load shader method is the method that reads the files and returns the string to be compiled. The method first creates an ifstream for the file. The file is then opened and converted to a string. If the file is open, then each line is appended to an output string variable which is returned.

**Create Shader ()**

The shader is created by declaring and defining a shader GLuint object that is set to the return of the glCreateShader () method that is tasked with creating the actual shader object.



A character array is made that will hold the shader source code string. Another array is made that will hold the length of the string.

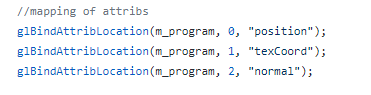


The first array index is set to the text value converted to a character. The length array index is set to the length of the text.



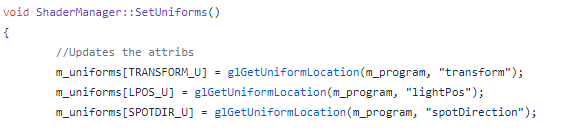
The shader source code is sent to opengl by calling glShaderSource and passing in the shader object and the two arrays. The shader is then compiled by calling glCompileShader (). After this, the shader is then returned.

**AttachAttribs ()**



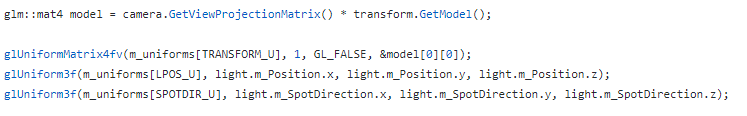
glBindAttribLocation is called so that the position, texture coordinates and normals are bind to their desired location; 0,1 and 2. The program is then linked from calling glLinkProgram () and validated from calling glValidateProgram.

**SetUniforms ()**



The shader uniform values are set by acceding the uniform array with their desired enum flag. The uniform is set to the value of the glGetUniformLocation () method which takes the program and the name of the uniform within GLSL as parameters.

**Update ()**



The model matrix is made by calling the passed in camera’s GetViewProjectionMatrix () method and multiplying this by the passed in transforms model matrix. This is accessed from the passed in transform object.

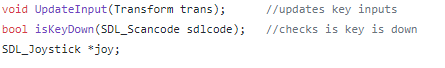
glUniformMatrix4fv and glUniform3f are called for passing in a 4x4 matric and vector 3 into the shader. glUniformMatrix4fv is called so that the MVP matrix is passed through. The second method is called to pass in the light position and the spot direction of the light.

# Input Manager class (Controller Support Extension Included)

## Input Manager.h

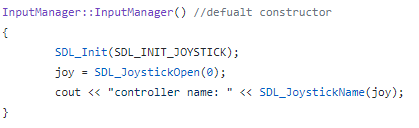


The input manager follows a singleton design patter. To do this, the input manager object is declared as a static pointer. For receiving the states when the game is running a constant pointer for Uint8 type named states will hold this information. Default constructors and destructors are declared in the public declarations. A method that returns the input manager pointer named getInstance () is declared as static.

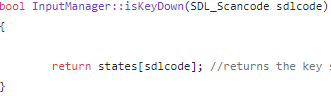


A method for updating the input named Update Input () takes in a transform object as an argument. For checking is a key is down, a method that takes in an SDL scancode and returns a Boolean value is declared as isKeyDown(). For receiving controller input, A SDL\_Joystick object pointer is created.

## Input Manager.cpp



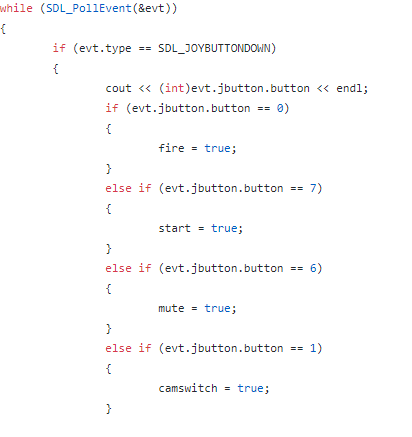
The constructor initialises the joystick capabilities by calling SDL\_Init. The joystick object is activated by calling SDL\_JoystickOpen (). The getInstance () method follows the same scope as the one described in the sound manager section.



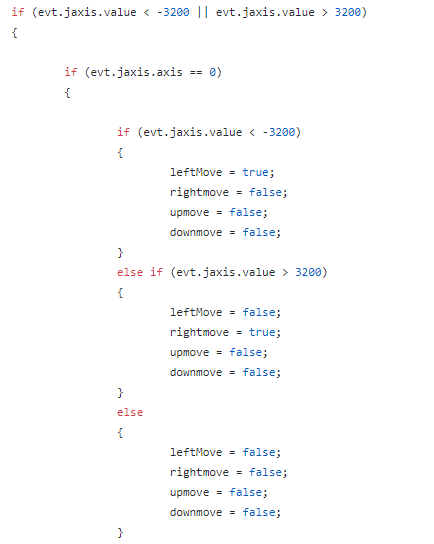
The isKeyDown method simply returns the state of the sdl code in the states array.

## Controller Support

For handling controller support, A method for checking events was made named Check Events (). The scope first decalred a new SDL\_Event named evt. The evt object is static. A while loop is made that grabs the SDL input events named SDL\_PollEvent () and evt is passed into this method. While this event returns true, the joystick is tested against.



For button presses, an if statement is made that checks if the event type that is returned is that of a button down on the joypad. If true then the button codes are tested against. If the required buttons return true then a Boolean for their associated tasks is set to true.



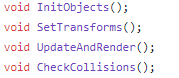
For the left analogue stick for movement. The event checks if the type is of a joystick axis motion, if so, then the follow code is executed. If the x axis is returned then the value is checked if it is less than -3200. If this is true then it is a left movement so that Boolean is set to true and the rest, false. If it is greater than positive 3200 then it is a right movement. The same steps are carried out for the y axis for moving up and down.

In the Game class. The Booleans are checked against and the movement and interactions are executed the same as if the keys were pressed.

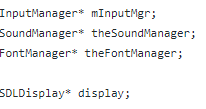
# The Game Class

## theGame.h

The game class is where all the objects come together to execute the game. The header class includes all the header files of all the classes that have been created. The game has a default constructor and destructor declared as public members. Another method named Run Game () is declared. This method will hold the code for initialising, updating and rendering game objects and elements.



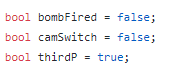
The private members include 4 void methods that take 0 arguments. These, in order, do the following tasks. Initialising all the game objects, sets the transforms accordingly, update and rendering objects and checking for collisions between objects.



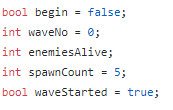
The sound, input and font manager pointer objects are declared, followed by the SDL display object.



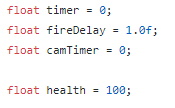
For handling objects that will be instantiated multiple times. Two vector arrays are made for Bomb and Enemy objects. Following this code objects are made for the Model robot, bomb, enemy, ship and ground. Declarations appear for all the textures and transforms for these objects as well. Three cameras are set up for camera 1, 2 and current camera.



Private members are now declared for variables that handle functionality and game mechanic data. Booleans are defined all as false and these will handle is functions have been triggered. Including firing, cam switching and the if the third person camera is active.



For handling enemy spawning and wave mechanics, a Boolean is made to se if the game has begun. An integer for holding the current wave, number of enemies alive and the spawn count of enemies. Another Boolean is made that will hold if the wave has started or not.



Here timers and delay float variables are made. These will be used for setting delays between firing bombs and switching cameras. Finally, a health variable for the player is made.

## theGame.cpp

**TheGame()**

The game object constructor calls 2 methods. The InitObjects method and the SetTransform method.

**InitObjects()**

This method simply defines the objects that are declared in the header file. The objects included here are; the sound, input and font manager, the display, all models, shaders, textures and cameras. The sound and font managers add the appropriate files to the collection as well.

**SetTransforms()**

This method sets the initial position, rotation and scale of the transforms so that their corresponding object are in the appropriate alignment and scale.

**RunGame()**

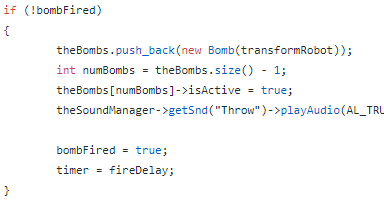
This method begins by using the sound manager to play the music of the game. An if statement is made to check if the music is not playing, if this is true then the music is set to play.



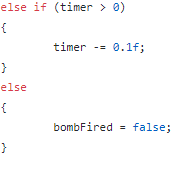
A while loop is made so that the scope is executed while the display is not closed. Inside this loop the font manager is used find and print text onto the screen by first setting the display to orthographic from the display class and then printing the text. The projection is set back to model view from the display class. The input manager is updated after this. The game class’ UpdateAndRender () and Check Collision () functions are then called. After these calls, the input manager is checked against to see if SDL buttons have been pressed. If this is true, then the buttons task will be executed. An example of this is from the WASD keys. If one of these keys returns true then the robot will be moved by a certain speed.



When the spacebar is hit, then a bomb is spawned.



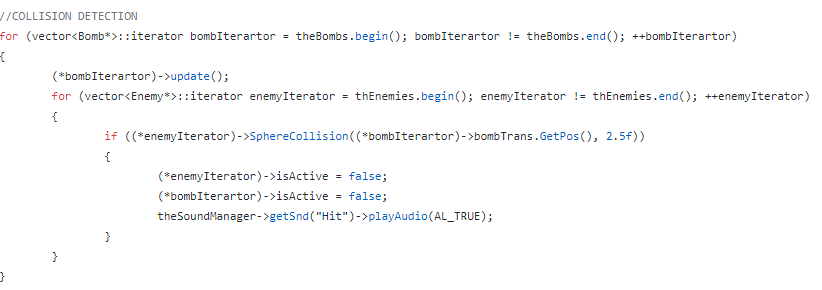
The bomb vector receives a new addition by using the push back method. A new bomb is declared and instantiated inside this methods argument. An integer that holds the number of bombs is made. This is set to the size of the bomb array vector. The bomb that has just been added is set to active and the sound manager plays the throw sound. A timer is made so that not too many bombs can be instantiated within a given time frame. A Boolean will handle whether a bomb can be thrown or not. This is then set to true.



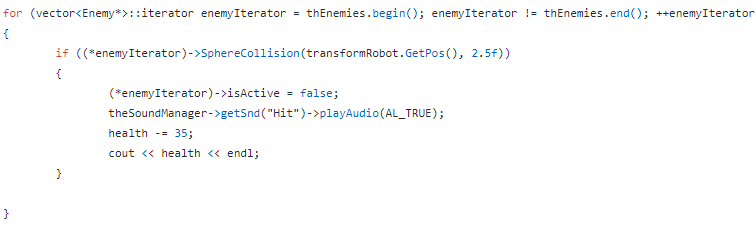
If the timer is greater than zero, then it will begin to lower by 0.1 every frame. After it hits zero then the bomb can be fired again.

When the C key is pressed then the camera must change. A Boolean is made that will detect if the camera is in third person or not when the key is pressed. The Boolean will switch to true or false whenever the c key is pressed. This is the same method the M key uses for muting the sound.

**Check Collisions ()**

****

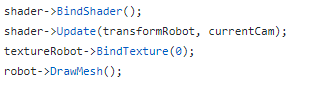
When checking for collisions, the bomb vector is iterated through. Within the iterations the current bomb is updated. A nested for iterator is made that will cycle through the enemy vector. The enemies sphere collision method is checked against passing in the bomb transform and hard coding in a radius of 2.5. If this returns true then both objects are set to inactive and the hit sound is played.



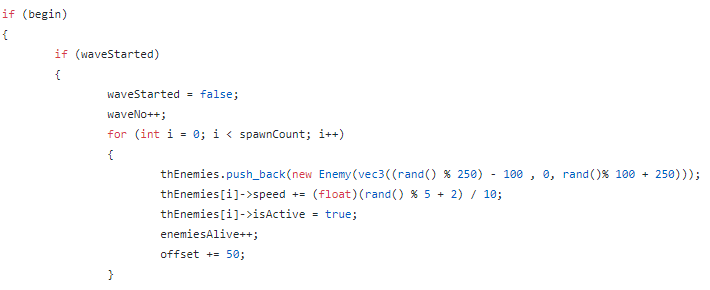
The same steps are followed for the player object. Expect when they collide the player will lose health instead of being destroyed.

**Update and Render ()**

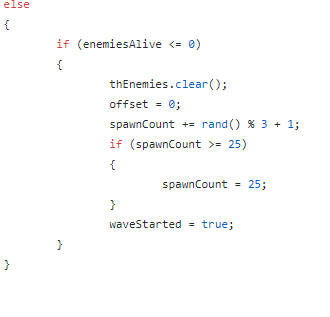
This method first defined all the camera objects. Then a check is made to see what the current camera should be based on the third person Boolean. The transforms for the robot are then set to that the x and y values match the camera while giving an offset. The second camera transform is made so that it appears in front of the robot object, so it looks like a first-person view.



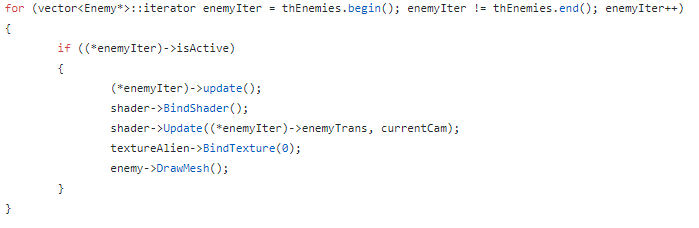
For every object, a shader is bound and updated, the texture is bound, and the model is rendered on the screen.



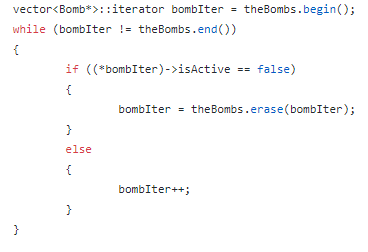
For spawning enemies, a similar method to the bombs is used. When the wave has been activated, a spawn count integer is iterated through and for each iteration, a new enemy is added to the vector with a random x and z position as well as speed. The active bool is set to true and the number of enemies alive is added by one.



If the wave is not started, then a check is made to see if the number of enemies alive is less than or equal to zero. If so then the vector is cleared, a random spawn count is made, and the Boolean is set back to true.



For updating and rendering these objects, the vectors are iterated through and the update and rendering code used for all objects is called within the scope if the active Boolean is true.



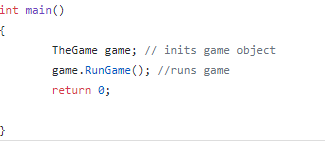
Whenever these objects are deactivated then they must be erased from the vector to free memory. Here the erase method is called so that the object is destroyed. The same is carried out for both the bomb and enemy objects.



The end of this function swaps the buffer for a window.

# Main.cpp

The main loop of the game simply creates a new Game object and calls the objects RunGame method that will allow the game to be initiated and updated accordingly.



# References

## Bibliography

Camera

<https://stackoverflow.com/questions/13706017/opengl-3rd-person-camera>

<https://forums.libsdl.org/viewtopic.php?p=16188>

<https://www.opengl.org/archives/resources/faq/technical/viewing.htm>

Audio

<https://www.openal.org/documentation/OpenAL_Programmers_Guide.pdf>

Object Loader & Image Loader

<https://github.com/BennyQBD/ModernOpenGLTutorial/blob/master/obj_loader.cpp>

<https://github.com/nothings/stb/blob/master/stb_image.h>

<https://github.com/fmcfar200/GP2CWProject/tree/Coursework-Submission>

OpenGL

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/>

Joystick

<https://wiki.libsdl.org/CategoryGameController>

<https://www.youtube.com/watch?v=XoHYvHixhBY&t=523s>

<https://www.libsdl.org/release/SDL-1.2.15/docs/html/guideinput.html>

<https://github.com/kripken/emscripten/issues/2193>

Other

<https://www.codeproject.com/Articles/14154/OpenGL-Terrain-Generation-An-Introduction>

## Assets References

Models

<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1078278>

<https://www.turbosquid.com/FullPreview/Index.cfm/ID/383539>

<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1068325>

<https://www.turbosquid.com/FullPreview/Index.cfm/ID/658720>

<https://www.turbosquid.com/FullPreview/Index.cfm/ID/798902>

Audio

<https://freesound.org/people/scorpion67890/sounds/396804/>

<https://freesound.org/people/LeMudCrab/sounds/163458/>

<https://freesound.org/people/VinnyPuX/sounds/411997/>

Textures

<http://www.3backgrounds.com/backgrounds/alien-seamless-backgrounds-124107.html>

<https://www.textures.com/download/soilbeach0087/32630>

<http://spiralgraphics.biz/packs/metal_corroded/index.htm?2>