

Coursework

Game Programming 2

|  |  |
| --- | --- |
| **Module Name:** | Games Programming 2 |
| **Module Code:** | M3I625656-19-A |
| **Module Leader:** | Bryan Young |
|  |  |
| **Student ID:** | S1716629 |
| **Student Name:** | Daniel Dutton |

*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*.

Contents

[Main Game Class 2](#_Toc29516341)

[User Movement 2](#_Toc29516342)

[Collision & Respawn 2](#_Toc29516343)

[General Code 3](#_Toc29516344)

[Main 3](#_Toc29516345)

[Game Display Class 4](#_Toc29516346)

[General Code 4](#_Toc29516347)

[Game Audio Class 5](#_Toc29516348)

[General Code 5](#_Toc29516349)

[Game Mesh Class 6](#_Toc29516350)

[Collision 6](#_Toc29516351)

[General Code 6](#_Toc29516352)

[Game Texture Class 7](#_Toc29516353)

[General Code 7](#_Toc29516354)

[Game Shader Class 8](#_Toc29516355)

[General Code 8](#_Toc29516356)

[Camera 9](#_Toc29516357)

[Game Camera 9](#_Toc29516358)

[Game Camera Transform 9](#_Toc29516359)

[Other 10](#_Toc29516360)

[General 10](#_Toc29516361)

[New Parts 10](#_Toc29516362)

[Attempted Code 10](#_Toc29516363)

# Main Game Class

## User Movement

The game allows the user to control the 3D model using a keyboard. This is done in the “Main\_Game\_Class” in the process input function. The function detects what state the game is in and changes it through cases, for movement it detects the pressing of a key and if that key is one involved in moving the user’s 3D model it changes the game state to one of the states associated with movement. If this wasn’t the case or the button was released, then the game would default to the PLAY state which allows everything else to continue except the movement of the user’s model. The only minor issue this can have is the fact only one state is active at one time so the user’s model can either turn or move or neither, although this doesn’t affect much else.

The movement states are split into four; FORWARD, BACK, RIGHT & LEFT. Each of these get called on in the update function. The FORWARD state, when active, moves the user’s model forward based on the players current X/Y coordinates based on the players speed which is set to have a maximum to prevent constant acceleration. The same code is applied to the BACK state, however, to ensure the model reverses the speed values are set to negative. The LEFT state is simpler as when active it rotates the model on the Z axis by a counter, as the game is mostly top-down. The right does the same, but the counter goes down instead of up. The update function also constantly updates the models sphere data to ensure detection for when the asteroid object collides with it.

## Collision & Respawn

Collision is a Boolean function called in the game loop, a function in the “Main\_Game\_Class” that ensures all the necessary functions are running when the game has not been exited yet. The collisions are called twice in this game loop, one for a collision between the player model and the obstacle model and one for collision between the planet model and the obstacle model. It also uses a specific sphere radius for each of these models which is set in the update function so the sphere moves with the model.

The collision function uses the positions of two models and the radius for each of these to create a distance equation for a new float variable called “distance”, this variable is then compared to the sum of the two radiuses. If the distance is larger than the sum then the function returns a false and nothing happens, however if it is smaller then it gets the listener attached to the camera and plays some audio for the collision from the first model. Once this sound has been played the function check if the first model is the player model or the planet model, once this is found that model is reset with the obstacle model using the respawn function.

The respawn function uses 3 Boolean variables to decide which of the 3 models are to be reset. If the obstacle Boolean is true then the function first sets a random direction integer variable that will decide if the obstacle is traveling on the Y or Z axis, this function is also accessed in the update function so the obstacle moves appropriately. If it is set to 1 then the Y position is set to the bottom of the screen and the z position is set to 0 otherwise it sets the z position close to the camera and the y position is set randomly on the screen. Once this has been complete the x position is randomly set to a position on screen. If the player Boolean is true then the function sets the player model to its origin position and resets the rotation of it. Similarly, if the planet Boolean is true then the planet model is set to the planet models origin position and the orbit is also reset.

## General Code

The header files start of by getting all the necessary information from the other classes for later use with “#include” and then it creates the 6 games states that exist in the “Main\_Game\_Class” source file. The class “Main\_Game\_Class” is then created and split into two sections.

The first being the Public section, where the constructor that is used to initialise the variables of the class in the source file, the destructor that destructs variables in the source file and the run function that runs initialisation of the systems and the game loop, are created in addition to the transform variable that allow the models to move in the scene.

And the second being the Private section, where all the other main functions are created. This includes: the “initSystems” function that is used to initialise various things like the display, the camera position, sound files, models files, textures and shaders as well as the starting position of the models; the “update” function that constantly updates the game throughout; the “playAudio” function that is used to play audio; the “drawGame” function that draws all the necessary things to the screen; and the “gameloop”, “collision” and “respawn” functions that were explained earlier. The main variables are also created in this section specifically the variables that use other classes like “\_gameDisplay” that uses the display class, “myCamera” that uses the game camera class, various variables like “player”, “texture” & “shader” that use the mesh, texture and shader classes respectively to create a model, and the “device” class that uses the audio class. Then there are some simpler variables like “sfx” and “bkgMusic” that is used to assign appropriate audio to the correct function and various floats that control in game models positions, rotations, scale and speed in addition to one integer that assists in the direction one model heads in called “randDir”.

## Main

While not exactly part of the “Main\_Game\_Class”, “main” is a .cpp file that uses the class. It starts off using the main function, a function that is predefined in C++ and is the entry point into the program. It uses arguments when called to call the SDL main. The first thing the function does is create a variable called “mainGame” that is made using “Main\_Game\_Class”, it then starts off the program by using “mainGame.run”. This starts off the entire program and the rest is taken care of. The function then returns 0.

# Game Display Class

## General Code

The header files start of by getting all the necessary information from the appropriate locations, like “SDL/SDL.h” and “GL\glew.h”, using “#include”. It then tells the file to use the standard namespace before it creates the class. First the class creates a public section that contains functions that can also be used elsewhere in the program. These functions are the constructor; that initialises variable in the class, the destructor; that destructs the variables and deletes all the necessary parts before quitting in the source file, the “initDisplay” function; that is used to initialise the games display, it’s window and some parts of the graphic library that it uses which is all called in the “Main\_Game\_Class” file, the “swapBuffers” function; that swaps the buffers and is called in the “Main\_Game\_Class”, the “clearDisplay” function; that clears the display and is called in the “Main\_Game\_Class”, and finally the “getHeight” and “getWidth” functions that get the size of the screen and are also called in the “Main\_Game\_Class”.

The class then creates a Private section that holds a function called “returnError” that is used to return a message to the user should anything go awry. It also holds a few variables such as glContext that holds the context for the display, the “sdlWindow” that holds the pointer for the display window and the integer variables for the screen’s width and height.

# Game Audio Class

## General Code

The header files start of by getting all the necessary information from a large amount of locations using “#include”, these include things like the audio library and the SDL. Once this has been complete a class is created, this class first creates two structs. The first struct, the data struct, contains information about the source of the sound file and information about the buffer. It also holds a name variable for the files name and then sets these variables appropriately. The second struct, the Vector3 struct, just creates 3 floats called x, y & z.

Once these structs are done a standard vector is created that points to the data struct and context and a device are created for the audio library. The class then creates a Boolean function that checks whether a number is a big-endian or not, a function that converts numbers into integers appropriately based on if they are a big-endian or not, and a function used to load a WAV file and return it as sound data by converting parts to integers and uses the filename, channel, sample rate, bps and size as input.

The class then creates a public section containing multiple functions. First are the constructor that initialises the variables like the context and device and destructor that destructs variables, deleting data, context and the audio library device in the source file. Next is the “loadSound” function that is called in the “Main\_Game\_Class”, this function takes in a filename, loads the .WAV file using the “loadWAV” function to create sound data, sorts out this data and returns the source ID variable to be used in “Main\_Game\_Class” whenever that specific sound is played. Next up are multiple functions that are used to control the sounds like the ”playSound” function that take in the id of the sound, and if input the location it is coming from, and plays them, the “stopSound” function that stops a specific sound and finally the “setListener” function that sets up the listener for the audio.

# Game Mesh Class

## Collision

The game mesh class is where the detection for the collisions is set up. This uses a struct in the header file that to store the appropriate data for the sphere, the struct itself contains a function called sphere that sets the position of the sphere appropriately and a function called “SetPos” and “SetRadius” that uses and sets two variable created in the private part of the struct to hold the position and radius of the sphere. The sphere data is gathered using the function made in the Mesh\_Game\_Class called “updateSphereData”, this gets the appropriate transform and radius from the “Main\_Game\_Class” to create a sphere for each model so collisions can occur. The collisions themselves uses the getters created in the mesh class to check for collisions.

## General Code

The header files start of by getting all the necessary information from the appropriate file locations using “#include”, it then creates the its first class. The first class is the Vertex Class that is used to hold simple mesh data and can be used to make simple shapes, like a triangle, in the “Main\_Game\_Class” if needed. It contains variables to hold the position, texture Coord and normal of a vertex and functions that get the variables and sets them to the appropriate values when needed. Once this class has been done the file creates the sphere struct used for collisions.

Finally, the file creates the main class called “Game\_Mesh\_Class”. This class first creates the constructor to initialise variables and a destructor to destruct the variables and deletes the vertex arrays. Once that is done the draw function is created to draw the models on screen in the “drawGame” function in the “Main\_Game\_Class”. Next the “init” function is created, this function is used in the “Main\_Game\_Class” to initialise the models by setting their “pos”, “texCoord” and “normals”. Once that is done the function proceeds to use the “initModel” function, that is also created in the public section of the header, to create and bind the vertex arrays and appropriate buffers appropriately. The header also creates the “loadModel” function that takes in the filename for a model in the “Main\_Game\_Class” and loads it into the game. Next the “updateSphereData”, “getSpherePos” & “getSphereRadius” are created to use for the sphere struct and collisions. Next the private section is created, this starts with an enumerator that holds the information for various buffers used in the source file, specifically the “initModel” function. Next a meshSphere variable is created using the sphere class for the “updateSphereData” function. Next the “vertexArrayObject” (VAO) and buffer arrays are created to use in the mesh source and the “drawCount” that is created to define how much of the VAO we want to draw.

# Game Texture Class

## General Code

The header files start of by getting all the necessary information using “#include”, it then creates the class “Game\_Texture\_Class”. The class first creates the constructor that can be used to initialise variables. Next a Bind function is created that is used in the source to check if an appropriate texture is used, activate the texture and bind it to the appropriately. An “init” function that takes in a file name is created next, this function uses the file, the appropriate width, height and number of components to load and store image data. The texture is generated, then it binds the texture, wrapped and filtered before it is defined with glTexImage2D using the image data and other appropriate variables. Finally, the image date is freed. The header then creates the destructor that would destruct the variables and deletes the textures. The header finally then creates the private section that holds the texture handler integer variable that handles the textures.

# Game Shader Class

## General Code

The header file start of by getting all the necessary information from other files using “#include” and then the class is created. The class itself starts off making a public section which contains multiple functions and variables. The first is the constructor that can be used to initialise variables. Next is the Bind function where the GPU is set to use our shaders by installing the program object as part of the rendering state. Next the update function is created, this function takes in information about the camera and the transform of the camera. Using this the camera creates the model view projection (MVP) matrix by multiplying the VP by the models(M). This is then used to specify the value of a uniform variable for a model. Next the “init” variable is created, this variable takes in a filename and uses it to create the frag and vert shaders after creating the shader program. It then adds all the shaders to the program and binds attributes to the shaders. An executable is then created, and the program is validated before the uniform’s variable adds its location to an array. The “LoadShader” function is then created taking in a filename, opening said file and putting it to an output, this output is then returned to whatever called the function. The “CheckShaderError” function is created next and is used to check for errors when creating or initialising the shaders, if one is detected it returns an error message and if not, it doesn’t. The header file then creates the “CrateShader” function that takes in specific text and an integer representing the type it is. The shader is first created using the type and checked if it exists or is set to 0 (doesn’t exist). It then creates “stringSource” and “length” in the source file as lists to hold information about the text. This is then used to send the source code to OpenGL and then the shader is compiled. An error is then checked before the shader is returned to the place the function was called. This section ends with the creation of the destructor that destructs the variables and detached and deletes the shaders followed by deleting the program.

The private section is then created, this starts of by creating an integer to hold the number of shaders. It then proceeds to create an enumerator to hold the uniforms. Finally it creates 3 variables, the program variable used to track the shader program, the shaders variable that is used as an array to hold shaders using the number of shaders integer to decide how big it is, and finally the uniforms variable that is an array that holds the number of uniform variables and is set to be the size of the number of uniforms.

# Camera

## Game Camera

This is a header file that is used with the “Main\_Game\_Class” files and the game camera transform header, it starts by getting the appropriate information using “#include”. This file starts of creating a struct called “Game\_Camera” and within that creates a public section. This public section starts with a constructor that can be used to initialise variables, it the creates the “init\_Game\_Camera” function that asks for multiple variables when called; a vector3 called “pos” that is used to set the position of the camera in the function and several floats called “fov”, “aspect”, “nearClip” and “farClip” that are used to create the projection matrix. The function also sets up the camera’s forward direction and up direction that can be used to move the camera if needed. Next the struct creates an inline function that creates the view projection matrix and returns it to where it’s called from. After this the struct creates a “getPos” function to return the position of the camera to where this is called from in the “Main\_Game\_Class”. Finally, the public section creates tow functions that move the camera in different directions, “MoveForward” that moves on the camera forward and backwards and “MoveSide” that moves the camera left and right.

Finally, the struct creates the private section that creates the variables that hold information about the camera; the projection variable for getting the projection matrix, the “pos” variable for the cameras position, the “forward” variable to store how far forward the camera moved and the “up” variable that stores how far up the camera is.

## Game Camera Transform

This is a header file that is used with the “Main\_Game\_Class” files, it starts by getting the appropriate information using “#include”. This file starts of creating a struct called “Camera\_Transform” and within that creates a public section. This public section starts with a constructor that takes in 3 vector3 variables; “pos” a position variable, “rot” a rotation variable and “scale” a variable holding the scale of an object. These variables are used to set the structs own “pos”, “rot” and “scale”. Next an inline function is created called “GetModel”, this function gets all the information about a model and then returns the multiplication of the position, rotation and scale of the model. Finally, a bunch of getters and setters are created to get the position, rotation and scale and set the structs own values equal to them. Theses values are stored as vector3 variables in the private section of the struct.

# Other

## General

The program uses much of the lab code provided as a basis, this can be specifically seen in the “obj\_loader” class that has been used as is, this can also be said for the “stb\_image” class that was used as provided in class. Additionally, the shaders used in the program were also the ones provided in labs and not edited due to worries of breaking them.

## New Parts

Various textures, sounds and models were taken from the internet to be used and although effort was taken to use CC0 items the items links will be listed in case of any issue: the explosion sound and music was found in these places respectively - //freesound.org/people/timgormly/sounds/162792/ , //freesound.org/people/Clacksberg/sounds/491288/ .  
The textures of earth, the rock and the metal from these places - https://www.deviantart.com/nazo-ryuu/art/Earth-Like-Texture-Small-133967438 , https://www.filterforge.com/filters/3215-v8.html , https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Ftse1.mm.bing.net%2Fth%3Fid%3DOIP.XBA1rvQXLBQyH2vLDB1tJAHaEK%26pid%3DApi&f=1 .  
And the ship, rock and planet models were found here - https://free3d.com/3d-model/model-rocket-very-thin-v1--686111.html , https://free3d.com/3d-model/low-poly-rock-4631.html , https://free3d.com/3d-model/earth-v1--590680.html .

## Attempted Code

Effort was put into implementing a skybox, however due to difficulties this was not successful. Despite this however the code remains in the program, commented out, for viewing. In addition to this a cube was created in blender and two new shaders where created in C++ using one of the lecture slides provides, although due to the failure to implement they don’t do anything at this time and were also left in for viewing. The attempt consisted of creating the cube and adding the texture and shader as well as updating it in the “drawGame” function in the Main\_Game\_Class and creating a new function in the texture class called “skyboxInit” that worked similarly to “init” but would hopefully texture the way that was wanted, inside the cube.