Disclaimer: 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.

Fotios Spinos

Name: Fotios Spinos

Matriculation Number: S1625069

Games Programming Document

**Table of Contents**

[1.0 Overview of Main Methods, Structures and Classes 4](#_Toc533123523)

[1.1 Class: main 4](#_Toc533123524)

[1.1.1 Overview 4](#_Toc533123525)

[1.1.2 Methods 4](#_Toc533123526)

[1.2 Class: Scene 4](#_Toc533123527)

[1.2.1 Overview 4](#_Toc533123528)

[1.2.2 Variables 4](#_Toc533123529)

[1.2.3 Methods 5](#_Toc533123530)

[1.3 Class: GameObject 6](#_Toc533123531)

[1.3.1 Overview 6](#_Toc533123532)

[1.3.2 Variables 6](#_Toc533123533)

[1.3.3 Methods 7](#_Toc533123534)

[1.4 Class: Display 8](#_Toc533123535)

[1.4.1 Overview 8](#_Toc533123536)

[1.4.2 Variables 8](#_Toc533123537)

[1.4.3 Methods 8](#_Toc533123538)

[1.5 Struct: Vertex 10](#_Toc533123539)

[1.5.1 Overview 10](#_Toc533123540)

[1.5.2 Variables 10](#_Toc533123541)

[1.5.3 Methods 10](#_Toc533123542)

[1.6 Struct: SphereCollider 10](#_Toc533123543)

[1.6.1 Overview 10](#_Toc533123544)

[1.6.2 Variables 10](#_Toc533123545)

[1.6.3 Methods 11](#_Toc533123546)

[1.7 Class: Mesh 11](#_Toc533123547)

[1.7.1 Overview 11](#_Toc533123548)

[1.7.2 Variables 11](#_Toc533123549)

[1.7.3 Methods 12](#_Toc533123550)

[1.8 Class: Shader 13](#_Toc533123551)

[1.8.1 Overview 13](#_Toc533123552)

[1.8.2 Variables 13](#_Toc533123553)

[1.8.3 Methods 13](#_Toc533123554)

[1.9 Class: StartScene 14](#_Toc533123555)

[1.9.1 Overview 14](#_Toc533123556)

[1.9.2 Methods 14](#_Toc533123557)

[1.10 Class: CharacterController 15](#_Toc533123558)

[1.10.1 Overview 15](#_Toc533123559)

[1.10.2 Variables 15](#_Toc533123560)

[1.10.4 Methods 15](#_Toc533123561)

[1.11 Struct: Camera 16](#_Toc533123562)

[1.11.1 Overview 16](#_Toc533123563)

[1.11.2 Variables 16](#_Toc533123564)

[1.11.3 Methods 16](#_Toc533123565)

[1.12 Class: CollisionManager 17](#_Toc533123566)

[1.12.1 Variables 17](#_Toc533123567)

[1.12.2 Methods 17](#_Toc533123568)

[1.13 Class: Transform 18](#_Toc533123569)

[1.13.1 Overview 18](#_Toc533123570)

[1.13.2 Variables 18](#_Toc533123571)

[1.13.3 Methods 19](#_Toc533123572)

[1.14 Class: Texture 20](#_Toc533123573)

[1.14.1 Overview 20](#_Toc533123574)

[1.14.2 Variables 20](#_Toc533123575)

[1.14.3 Methods 20](#_Toc533123576)

[1.15 Class: MainGame 21](#_Toc533123577)

[1.15.1 Overview 21](#_Toc533123578)

[1.15.2 Variables 21](#_Toc533123579)

[1.15.3 Methods 21](#_Toc533123580)

[2.0 Sources 23](#_Toc533123581)

[2.1 Research 23](#_Toc533123582)

[2.2 Textures / Models 23](#_Toc533123583)

[2.3 Music 24](#_Toc533123584)

[2.4 Shader 24](#_Toc533123585)

# 1.0 Overview of Main Methods, Structures and Classes

## 1.1 **Class: main**

### 1.1.1 Overview

Initiates Program.

### 1.1.2 Methods

int main(int argc, char\*\* argv)

The starting point of the program. Creates a main object instance and runs it until the game stops executing. Returns 0 to terminate the program.

## **1.2 Class: Scene**

### 1.2.1 Overview

This class is used as a blueprint for creating, updating and rendering scenes. Scene objects should inherit this class and override its virtual functions if necessary.

### 1.2.2 Variables

GameObject sceneObj[10];

Array holding the GameObjects in the scene

GameObject \*sceneObjPtr[10];

Pointer array holding the memory address of every GameObject in the scene.

CharacterController sceneChar[10];

CharacterController \*sceneCharPtr[10];

Same use, as in the mentioned variables the difference is that they refer to Character Controllers

string sceneName;

The name of the scene, used to identify the scene

string \*sceneNamePtr;

A pointer to the scene name

int objCounter;

The number of GameObjects in the scene

int charCounter;

The number of Character controllers in the scene

Camera camera;

The camera of the scene

Display \*displayPtr;

A pointer to the game display

Audio sound;

A sound instance. Represents the background sound of the scene

Audio \*sountPtr;

A pointer to the sound instance

### 1.2.3 Methods

Scene();

initializes class variables. The for loops sets each element of each pointer array to its corresponding element contained within the sceneChar and sceneObj array

~Scene();

Class destructor

void createGameObject(const std::string &meshFileName, const std::string &texFileName, const std::string &shaderFileName, Transform& transform);

The if statement if (objCounter < size(sceneObjPtr)) checks if the number of objects created do not exceed the size of the array. Every GameObject element is instantiated and stored in the heap, which provides fast access for storing and loading data. The objCounter variable is used to index the current element so that it can be initialized and positioned. The variable is increased by one so that the next time the method runs the next variable is indexed.

void createCharacterController(const std::string &meshFileName, const std::string &texFileName, const std::string &shaderFileName, Transform &transform);

The logic behind this method is the same as in createGameObject. Instead its used for character controllers.

virtual void editor();

This method defines the atributes of each game object before the execution of the game loop. This method is inspired by game Engines where the scene includes the data for each GameObject before the game is executed. The data could include the GameObjects mesh, position, texture and shader. The method creates transform instances and uses them to call the methods: createGameObject and createCharacterController.

void SetSceneName(string &sceneName);

Sets the name of the scene using a reference to a string

virtual void render();

Renders every gameObject and character controller in the scene. A for loop is used to render each type. Essentially the for loop iterates through every object in the array which is being indexed by the methods createGameObject and createCharacterController. In the scope of each for loop the render function of each object is called.

virtual void update();

Updates every game object in the scene. The logic is the same as in the previously analysed method. In the scope of each for loop the update function is called for each game object or character controller.

virtual void input();

Iterates through every characterController object and passes the keycode received by the event pulled in the mainGame class.

void initSound(string &fileName);

Initializes the sound instance. This method sets the listener by using the camera’s position and forward direction. The string soundPath adds the parameter with the path to the sounds folder which contains every sound in the game. The loadSound method sound is called to load the sound in the specified path.

void playSound();

Calls the PlaySound behaviour in the audio instance. This method is responsible for playing the audio as long as the game runs.

## **1.3 Class: GameObject**

### 1.3.1 Overview

Represents the objects in a scene. The overloaded methods of the class allow parent classes to overwrite them so that each parent has its own unique behaviours. Child classes can also use the inherited methods to set up attributes like the objects mesh, texture and position. The design goal was to inherit this class and its behaviour to create new object types applied in the scene. This was inspired by unity and how the class GameObject works.

### 1.3.2 Variables

Mesh mesh;

The mesh of the gameObject

Texture texture;

The texture applied to the mesh

Shader shader;

The shader applied to the mesh

Transform transform;

Holds the position, rotation and scale of the gameObject

Mesh\* meshptr;

Texture\* textureptr;

Shader\* shaderptr;

Transform\* transformptr;

Pointers to the variables mentioned above.

Camera\* camera;

Camera pointer used to define the camera displaying the gameObject.

string name;

The name of the gameObject

bool active;

Defines if the object should be rendered or not.

### 1.3.3 Methods

GameObject();

Declares pointer class variables and sets a default name value.

~GameObject();

Prints in the local debugger that the object has been destroyed. The method uses the name variable so that the destroyed object can be identified easily.

void initObj(const std::string& meshFileName, const std::string& TexFileName, const std::string& shaderFileName, Camera &camera);

Initializes all class variables used to set up the object so that the object can be rendered.

void initMesh(const std::string& filename);

Initializes object’s mesh

void initTexture(const std::string& fileName);

initializes object’s texture

void intitShader(const std::string& fileName);

Initializes object’s shader

void initTransform(glm::vec3 pos, glm::vec3 rot, glm::vec3 scale);

Initializes the transform component by using the 3 vectors representing the position, rotation and scale.

void initTransform(Transform &transform);

Initializes transform by using the reference passed. Functions as a setter method

void setCamera(Camera &camera);

Sets the camera, using the parameter.

void declarePtrs();

Sets the pointers using their corresponding memory taken from the value variables in the class.

void SetActive(bool active);

defines if the object should be rendered or not

void setName(string &name);

Sets the name using the parameter

virtual void update() {}

updates the object. This method is empty as it is expected that classes inheriting form GameObject have their own unique behaviours. Essentially child classes should override this method as it is virtual.

virtual void input() {}

Follows the same construction as in the update method. This method should handle input.

void render();

The method responsible for rendering the gameObject. It Binds the shader pointes, updates the shader pointer by taking into consideration the object’s position and the camera displaying it, binds the texture pointer and renders its mesh.

## **1.4 Class: Display**

### 1.4.1 Overview

The window displaying the determined scene objects.

### 1.4.2 Variables

SDL\_GLContext glContext; Global variable to hold the context

SDL\_Window\* sdlWindow; Holds pointer to out window

float screenWidth; Stores the width of the screen

float screenHeight; Stores the height of the screen

### 1.4.3 Methods

Display();

Constructor Initializes variables

~Display();

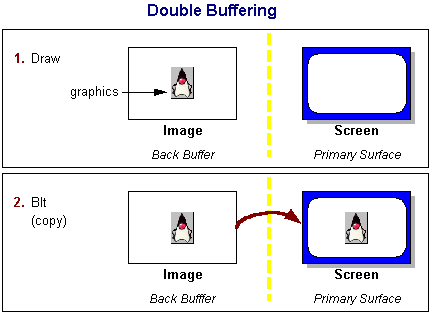
Destructor called when the object is destroyed. Clears data by deleting the SDL context, the SDL window and all the SDL initialized subsystems as the program should be terminated when the window closes.

void initDisplay();

This function initializes all the SDL libraries along with using the set attribute function which is used to “set the number of bits for each channel in the colour buffer”[[1]](#footnote-1). The method also creates a window instance and checks for errors. It then checks, if the window instance is empty, if the context in empty and if the OpenGl rendering context is being set up properly. If one of these cases is true, an error message is displayed to the local debugger window.

void swapBuffer();

Swaps buffers. The program uses two buffers. One is used to display the window and the second which is buffed so that it can be shown next (when a request for swapping them is given). The two buffers switch sides on function call. This improves the throughput and prevents bottlenecks. A bottleneck slows down the computer when a reasonable amount of data is being stored and processed in the system. This is ideal for this project as games are typically computationally expensive. Each scene could include a lot of objects which are rendered and being updated on each function call.



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

Clears the buffers involved with rendering the display. This involves: clearing the colour buffer values along with clearing the colour and depth buffer.

void returnError(std::string errorString);

Prints an error message using the parameter errorString. This method is used for error checking as its purpose is to be called, if one of the SDL initialization processes have not been executed successfully.

float getWidth();

Returns the width of the object.

float getHeight();

Returns the height of the object.

## **1.5 Struct: Vertex**

### 1.5.1 Overview

Contains information for each vertex in the imported mesh. The main difference between a struct and a class is that a struct has its default identifier to public.

### 1.5.2 Variables

glm::vec3 pos;

Holds the position of the vertex.

glm::vec2 texCoord;

Holds the texture coordinate applied to the vertex.

glm::vec3 normal;

Holds the normal to the vertex.

### 1.5.3 Methods

glm::vec3\* GetPos() { return &pos; }

Returns a reference to the position of the coordinate.

glm::vec2\* GetTexCoord() { return &texCoord; }

Returns a reference to the texture coordinate.

glm::vec3\* GetNormal() { return &normal; }

Returns a reference to the vertexes normal.

## **1.6 Struct: SphereCollider**

### 1.6.1 Overview

A sphere surrounding the object. Used to check for collisions with other GameObjects or Character Controllers.

### 1.6.2 Variables

glm::vec3 position;

Holds the position of the sphere collider.

float radius;

Holds the radius of the sphere collider

### 1.6.3 Methods

SphereCollider() {}

Sphere collider constructor called when the object is created.

SphereCollider(glm::vec3 position, float radius)

Overloaded constructor. Sets the position and radius using the parameters passed.

void setPos(glm::vec3 position)

Sets the position variable using the parameters passed.

void setRad(float radius)

Sets the radius variable using the parameter passed.

glm::vec3 getPos()

Returns the position.

float getRad()

Returns the radius.

## **1.7 Class: Mesh**

### 1.7.1 Overview

A sphere with a defined radius used to detect collisions.

### 1.7.2 Variables

SphereCollider meshSphere;

The mesh sphere instance. Represents the GameObject’s collider.

enum

{

POSITION\_VERTEXBUFFER,

TEXCOORD\_VB,

NORMAL\_VB,

INDEX\_VB,

NUM\_BUFFERS

};

An enumerator holding the buffers used.

GLuint vertexArrayObject;

Holds the vertex array of the mesh.

GLuint vertexArrayBuffers[NUM\_BUFFERS];

Array of buffers.

unsigned int drawCount;

The remaining of the vertexArrayObject which needs to be drawn.

### 1.7.3 Methods

Mesh(); Constructor, declared draw count.

~Mesh(); Destructor, clears vertex array.

glm::vec3 getSphereCollPos();

Returns the position of the meshSphere instance.

float getSphereCollRad();

Returns the radius of the meshSphere radius.

void draw();

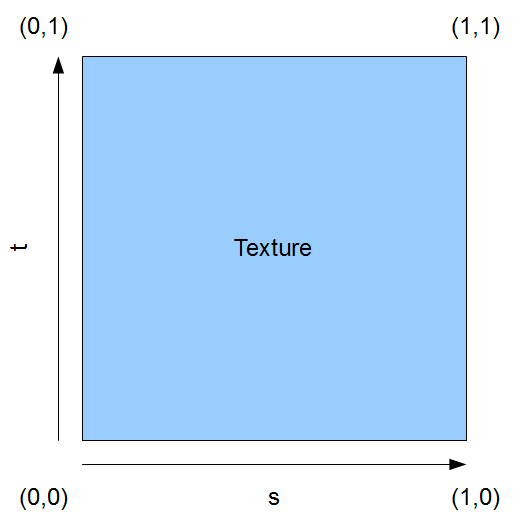
Binds vertex array, draws mesh and finally unbinds the vertex array.

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

Loads model from receives file and initializes it by calling the initModel method.

void initModel(const IndexedModel& model);

This method creates the buffers necessary so that the data received from the mesh can be passed to the GPU for rendering purposes. All the buffers created are of type: ArrayBuffer. 4 Buffers are generated in this case. The first holds the data for the mesh’s vertex position, the second holds the data for the texture coordinates. These are going to be applied to their corresponding vertex in the position buffer. For example, the first element of the buffer containing the position of the first vertex of the mesh has mapped to It, the first texture coordinate in the texture coordinate buffer.



The image above shows how points in the texture are referenced. For example, the point 0,0 of the texture OpenGl will return the bottom left point of the texture. That point will be applied to one of the vertices in the mesh.

The third buffer Contains the normal for every triangle formed by the vertices indexed

The fourth vertex contains the vertex index. Essentially it describes how vertices are connected so that the mesh can be formed.

1.8 Class: Shader

### 1.8.1 Overview

The shader is attached to GameObjects and GameObject childs. It contains rendering instructions.

### 1.8.2 Variables

static const unsigned int NUM\_SHADERS = 2; Number of shaders

enum

{

TRANSFORM\_U,

NUM\_UNIFORMS

};

GLuint program; Tracks the shader program

GLuint shaders[NUM\_SHADERS]; Array of shaders

GLuint uniforms[NUM\_UNIFORMS]; Number of uniform variables

### 1.8.3 Methods

Shader();

Class Contractor

void Bind();

Installs the program object specified by the program as part of rendering state.

void Update(const Transform& transform, const Camera& camera);

updates shader using the camera view projection received by the camera parameter.

void init(const std::string& filename);

This method initializes the shader. The method creates the shader program using the glCreateProgram() function. The method then proceeds by adding the two shaders: vertex shader and fragment shader to the shaders array. A for loop then iterates through every element in the array and attaches the shaders to the program created. The method then proceeds by associating attribute variables with the shader program. It does this for the position and texCoordinate attribute. The function call glLinkProgram(program) creates executables that will run on the GPU. After that the method checks if the shader program has been linked successfully, the program instance is valid and if the shader is valid. Then the uniform variables are being associated within the program.

std::string Shader::LoadShader(const std::string& fileName);

This method loads the shader from the file location specified by the parameter. The ifstream file represents the file which is going to be read. The method then opens the file using the.Open function. A while Loop checks if there are no errors associate with the file. While there are no errors the contents on the current line are stored in the line string. Each time the while loop iterates the next line is returned. The append method is used to add each line to the output string. When the while loop terminates the method returns the output.

void Shader::CheckShaderError(GLuint shader, GLuint flag, bool isProgram, const std::string& errorMessage);

Checks if the program is not empty. It then checks if it has been compiled successfully. If it has not, the open gl integer success is equal to 0. If the returned value is 0 an error message will be displayed in the local debugger window.

GLuint Shader::CreateShader(const std::string& text, unsigned int type);

This method creates a shader. The shader is represented by an Open GL integer. The method glCreateShader(type) creates an instance of the shader using the parameter type. If the shader is equal to zero, meaning that it is empty, an error message is printed to the screen. A pointer to an array of type Open GL character. The method sets its first element to be equal to the text passed and changed as a c type string. The method then creates an Open GL integer array and sets its first element to be equal to the size of the text string. The source code is sent to open gl. The shader code is then compiled by Open GL. The function CheckShaderError is called for error checking. The method finally returns the shader.

~Shader(); The destructor has a for loop which iterates through every shader created (end condition i < NUM\_SHADERS). In the scope of the for loop, each shader indexed is detached from the program and is being deleted. After the for loop terminates the shader program is being deleted.

## **1.9 Class: StartScene**

### 1.9.1 Overview

Inherits the scene class and overrides its editor. It uses the provided behaviours from the parent class to structure the scene differently (How many objects the scene has, what is their position, which texture each one should have applied to it, etc.)

### 1.9.2 Methods

StartScene(); constructor

~StartScene(); destructor

void editor(); This method is called instead of the default editor method located in the Scene class. StartScene inherits from this class so that it can use the functionality provided by it. As every scene should look different (have different gameObjects with different attributes) the class overrides its parent’s method as it’s a virtual function (Polymorphism).

## **1.10 Class: CharacterController**

### 1.10.1 Overview

Inherits from gameObject and overites its update and input function. Character controllers receive the input polled to move when the user enters the selected buttons defined in the classes input method.

### 1.10.2 Variables

float speed; The speed of the object on movement. The xSpeed, ySpeed and zSpeed are set to be equal to this variable when the object moves (The specified input is identified)

float xSpeed; The current speed applied to the x axis of the object

float ySpeed; The current speed applied to the y axis of the object

float zSpeed; The current speed applied to the z axis of the object

### 1.10.4 Methods

CharacterController();

The constructor initializes class members

~CharacterController();

The destructor called when the object is deleted

void update ();

Uses the object’s transform member which is inherited by the GameObject class to move the object in the x, y and z axis using the speed variables. The three variables are set to zero so that the object stops if none of the keys specified for movement are received from the polled event.

void input(SDL\_Keycode evnt);

Receives the input’s key code and sets the specified speed variables to be equal to the speed or the speeds negative value. The method uses a switch statement to set the correct variable depending on the input received. For example, if the user presses a the xSpeed representing movement to the x button will be set to the negative value of speed. (as the desirable behaviour is to move the object to the left). If d is pressed the desired behaviour is to move the object to the right and so it is set to have the positive value of speed.

## **1.11 Struct: Camera**

### 1.11.1 Overview

Represents a camera displaying objects in a scene

### 1.11.2 Variables

glm::mat4 projection;

The perspective projection matrix of the camera, taking its field of view into consideration.

glm::vec3 pos;

A vector representing position

glm::vec3 forward;

A vector representing the front direction of the camera.

glm::vec3 up;

A vector representing the top direction of the camera.

### 1.11.3 Methods

Camera()

Called when the object is created.

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

This method initializes the class variables position, forward, up, projection.

inline glm::mat4 GetViewProjection() const

Returns the view projection matrix with the points of the model so that the camera shows the objects by taking its perspective into consideration. Every object in the scene is displayed with the camera’s perspective taken into consideration.

void MoveForward(float amt)

Moves the camera forward by adding to the current position an amount multiplied by the forward direction. The amount is the magnitude of the vector while the variable forward defines the direction of movement.

void MoveRight(float amt)

Moves the Camera to the left

void RotateY(float angle)

Rotates the camera to the y Axis by using the GML::rotate function which returns the rotation matrix of the camera of the camera. The variables forward and up are redefined as the camera has been rotated. This is possible by multiplying the rotation matrix with the forward and up direction. This multiplication applies the rotation to the two direction variables.

## 1.12 Class: CollisionManager

### 1.12.1 Variables

GameObject \*gameObjects[10]; pointers to objects checked for collision

int elementCounter; The number of elements checked in the array

### 1.12.2 Methods

CollisionManager(); The constructor sets up the object by setting the objCounter variable to zero.

~CollisionManager(); Called when the object is destroyed

void setGameObject(GameObject \*gameObject);

Defines elements in the gameObjects array by using the pointer parameter

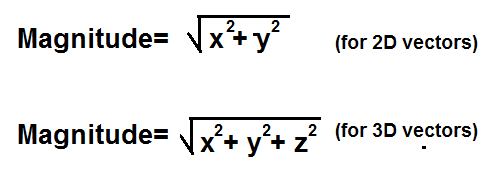
void checkForCollisions();

This method is responsible for checking if objects in the array have been collided. Essentially a nested for loop compares one of the elements with every other array element except itself. The element which is going to be compared with the rest of the GameObjects is defined by the x variable in the first for loop which is used to index the element in the array. The second loop iterates through every element in the array and checks if a collision occurs by calling the collision detection function. The method proceeds by calling the gameObjects collidedWith method which describes the necessary actions for the gameObject. It was important that a pointer to the collided gameObject is passed so that different actions can take place with different gameObjects. This behaviour is inspired by unity’s OnCollision functions in which a collision instance is passed instead.

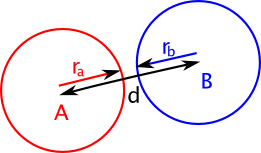
bool collisionDetection(GameObject \*gameObj1, GameObject \*gameObj2);

The method checks if the two parameters are equal, if they are then the method compares the same object and so it should not compared them, as this would always return true since the comparison will be between the same position and radius. When the if statement returns false, the following process takes place. The method stores the radius and position of each object. The GameObject in this case is suitable for simplicity reasons.

The distance is calculated by subtracting the two position and the magnitude of that distance is calculated with the Pythagoras theorem:



An if statement checks if the distance is less than the addition of the two radiuses (A radius represents the minimum distance with another collider so that a collision occurs). Essentially the statement checks if the two colliders have an intersection point which means that they have been collided.



It is worth mentioning that this method works for circles. For other shapes other calculations are required. Circles are relatively easy as every point in the circle has the same distance from its centre point. This is not true for other shapes.

## 1.13 Class: Transform

### 1.13.1 Overview

Holds positioning, scaling and rotating information along with providing a range of functions affecting this information. Meaning that it is possible to move, rotate and scale objects. Each gameObject has a Transform component which defines the mentioned attributes for the object.

### 1.13.2 Variables

glm::vec3 pos; The position stored within the transform instance

glm::vec3 rot; The rotation stored within the transform instance

glm::vec3 scale; The scale/size stored within the transform instance

### 1.13.3 Methods

Transform(const glm::vec3& pos = glm::vec3(), const glm::vec3& rot = glm::vec3(), const glm::vec3& scale = glm::vec3(1.0f, 1.0f, 1.0f)).

The constructor uses the references passed to set the values of the objects’ pos, rot and scale.

inline glm::mat4 GetModel() const

This method returns the model’s matrix which is defined by a multiplication between the model’s scale, rotation and position matrix. The matrix resulted by this multiplication is multiplied with the view projection matrix so that the model can be displayed with the perspective of the camera (displaying an object) being taken into consideration. The matrix is then multiplied with the model so that every vertex can be scaled, rotated and positioned accordingly.

glm::translate -> Uses the 3D vector parameter(in this case the class variable pos) to define the transformation matrix. A multiplication with this matrix translates the vector or matrix depending on the parameter used to call the function.

glm::scale -> Uses the 3D vector parameter(in this case the class variable scale) to define the transformation matrix. A multiplication with this matrix scales the vector or matrix depending on the parameter used to call the function.

glm::rotate -> The rotate function receives two parameters: The first is the amount of rotation required to be applied and the second is the axis in which the amount is going to be applied. That is why the method is used 3 times on for each axis(x,y,z). A multiplication with the rotation matrix rotates the vector or matrix depending on the parameter used to call the function

Using this method is convenient as the multiplication of the three matrices combines the behaviour of every matrix. The multiplication is be made in the following order: pos, rotation and scale.

inline void Scale(glm::vec3 &vector)

Adds the parameter to the class variable: scale

inline void Rotate(glm::vec3 &vector)

Adds the parameter to the class variable: rot

inline void Translate(glm::vec3 &vector)

Adds the parameter to the class variable: pos

## 1.14 Class: Texture

### 1.14.1 Overview

Represents the texture applied to gameObjects.

### 1.14.2 Variables

GLuint textureHandler; The variable representing texture data

### 1.14.3 Methods

Texture(); Called when the object is created.

void Bind(unsigned int unit);

The method checks if the unit received is included within the 32 textures the system works with. The Method then calls glActiveTexture to activate the texture. The texture is then bound: (“bind a named texture to a texturing target”[[2]](#footnote-2))

void init(const std::string& fileName);

The Method has three local variables. The three integers represent the texture’s height, width and the number of components of the image. The image is loaded using the function stbi\_load() which also defines the mentioned variables. The returned value of the method is a character pointer. The variable image data stores the returned value for future reference. An if statement checks if the image data is equal to null. If it is, an error message is displayed to the debugger: texture load failed: “The path of the texture”. The texture is then being set up by calling the function glGenTextures() which generates texture names. The function call glBindTexture binds the texture. The method then proceeds by calling glTexParameteri() This method is used to set a parameter to the texture. In this case it is used to wrap the texture outside of its width and height. glTexParameterf sets a texture parameter. The requirement of this method is that the texture must be either 2D or 1D. The code uses the function call for linear filtering for minification or magnification. glTexImage2D specifies the 2D image used as a texture. It does that by defining the following texture attributes: Target, Mipmapping Level, Pixel Format, Width, Height, Border Size, Input Format, Data Type of Texture, Image Data. stbi\_image\_free deletes the data from the CPU as they have been mode to the GPU.

~Texture();

Clears texture data

## 1.15 Class: MainGame

### 1.15.1 Overview

Represents the texture applied to gameObjects in a scene

### 1.15.2 Variables

Display \_gameDisplay;

The display (Window) instance of the application.

GameState \_gameState;

The current state of the game.

StartScene startScene;

StartScene\* startScenePtr;

The instance of the scene to be loaded.

SDL\_Event evnt;

The event polled from SDL.

SDL\_Event \*evntptr;

A pointer to the event received.

CollisionManager collisionManager;

Checks for collisions between objects in a scene.

CollisionManager \*collisionManagerPtr;

A pointer to the collision manager.

### 1.15.3 Methods

MainGame();

Initializes class variables

~MainGame();

Destructor called when the object is destroyed. It does not do anything as the program does not use the new keyword to create objects and allocate them in the heap. If that was the case the delete keyword should be removing these objects in memory.

void run();

Is responsible for initiating the required behaviours so that the game can run properly. Calls the methods initSystems and gameLoop.

void initSystems();

This method creates the game display and initiates all the SDL libraries.

void gameLoop();

This method contains a while loop which runs until the execution of the game. This method calls all the functions mentioned below. If the game state charges to exit the loop stops and the program is terminated.

void processInput();

Receives input and passes it to the running scene.

void drawGame();

Clears the display, renders the objects in the running scene, “enables or disable client-side capability”[[3]](#footnote-3), delimits the vertices of a primitive or a group of like primitives”[[4]](#footnote-4) and swaps buffers.

void updateGame();

updates the running scene

void collisionCheck()

Checks for collisions between the objects passed to the collision manager

void playSound();

Plays the sound in the running scene.

## 2.0 Sources

### 2.1 Research

* Definitions, D. and Hope, C. (2018). *What is Double Buffering?*. [online] Computerhope.com. Available at: https://www.computerhope.com/jargon/d/doublebu.htm [Accessed 1 Nov. 2018].
* Khronos.org. (2018). *glClear*. [online] Available at: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glClear.xml [Accessed 13 Nov. 2018].
* Docs.oracle.com. (2018). *Double Buffering and Page Flipping (The Java™ Tutorials > Bonus > Full-Screen Exclusive Mode API)*. [online] Available at: https://docs.oracle.com/javase/tutorial/extra/fullscreen/doublebuf.html [Accessed 10 Nov. 2018].
* Glm.g-truc.net. (2018). *GLM\_GTC\_matrix\_transform*. [online] Available at: https://glm.g-truc.net/0.9.4/api/a00151.html#ga283629a5ac7fb9037795435daf22560f [Accessed 20 Nov. 2018].
* Opengl.org. (2018). *glBindVertexArray and glBufferSubData interactions*. [online] Available at: https://www.opengl.org/discussion\_boards/showthread.php/175389-glBindVertexArray-and-glBufferSubData-interactions [Accessed 4 Dec. 2018].
* Learningaboutelectronics.com. (2018). *Magnitude Calculator*. [online] Available at: http://www.learningaboutelectronics.com/Articles/Magnitude-calculator.php [Accessed 10 Dec. 2018].
* Cite This For Me. (2018). *Save Time and Improve your Marks with CiteThisForMe, The No. 1 Citation Tool*. [online] Available at: http://www.citethisforme.com/ [Accessed 2 Dec. 2018].
* I.stack.imgur.com. (2018). [online] Available at: https://i.stack.imgur.com/KdBPf.png [Accessed 18 Nov. 2018].
* Opengl-tutorial.org. (2018). *Tutorial 3 : Matrices*. [online] Available at: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/ [Accessed 25 Nov. 2018].
* Cplusplus.com. (2018). *ifstream::ifstream - C++ Reference*. [online] Available at: http://www.cplusplus.com/reference/fstream/ifstream/ifstream/ [Accessed 4 Nov. 2018].
* Khronos.org. (2018). *glBindTexture - OpenGL 4 Reference Pages*. [online] Available at: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBindTexture.xhtml [Accessed 15 Dec. 2018].
* GitHub. (2018). *flibitijibibo/SDL2-CS*. [online] Available at: https://github.com/flibitijibibo/SDL2-CS/commits/master [Accessed 10 Dec. 2018].
* Khronos.org. (2018). *glEnableClientState*. [online] Available at: https://www.khronos.org/registry/OpenGL-Refpages/es1.1/xhtml/glEnableClientState.xml [Accessed 2 Dec. 2018].
* Khronos.org. (2018). *glBegin*. [online] Available at: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glBegin.xml [Accessed 12 Dec. 2018].

### 2.2 Textures / Models

* Turbosquid.com. (2018). *free obj model old book*. [online] Available at: https://www.turbosquid.com/FullPreview/Index.cfm/ID/896206 [Accessed 16 Dec. 2018].
* Turbosquid.com. Available at: https://www.turbosquid.com/3d-models/free-11-gun-3d-model/724710 [Accessed 16 Dec. 2018].
* Turbosquid.com. Available at: https://www.turbosquid.com/FullPreview/Index.cfm/ID/1025942 [Accessed 16 Dec. 2018].
* Blender Artists Community. (2018). *Blender Game Networking Tutorial Series*. [online] Available at: https://blenderartists.org/forum/showthread.php?111532-Blender-Game-Networking-Tutorial-Series [Accessed 18 Dec. 2018].
* The monkey model was downloaded from the university’s website. It was used in the labs for educational purposes. I do not know the model’s origin.

### 2.3 Music

The music was downloaded from the Lab file. The source is unknown.

### 2.4 Shader

The monkey shader was downloaded from the university’s website. It was used in the labs for educational purposes. I do not know the model’s origin.

1. GitHub. (2018). *flibitijibibo/SDL2-CS*. [online] Available at: https://github.com/flibitijibibo/SDL2-CS/commits/master [Accessed 20 Dec. 2018]. [↑](#footnote-ref-1)
2. Khronos.org. (2018). *glBindTexture - OpenGL 4 Reference Pages*. [online] Available at: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBindTexture.xhtml [Accessed 15 Dec. 2018]. [↑](#footnote-ref-2)
3. Khronos.org. (2018). *glEnableClientState*. [online] Available at: https://www.khronos.org/registry/OpenGL-Refpages/es1.1/xhtml/glEnableClientState.xml [Accessed 20 Dec. 2018]. [↑](#footnote-ref-3)
4. Khronos.org. (2018). *glBegin*. [online] Available at: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glBegin.xml [Accessed 20 Dec. 2018]. [↑](#footnote-ref-4)