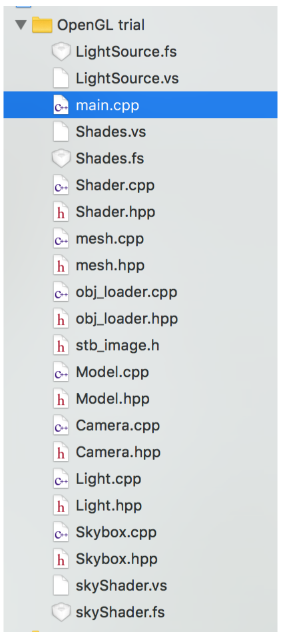
Graphics Pipeline and Basic Game Engine

Rashid Hafez

December 18, 2018

1 Introduction

The aim of this task was to understand the underlying foundations of 3D graphics, using the OpenGL library to setup a very basic engine which supports model loading, lights and animation.



OpenGL trial
LightSource.fs
LightSource.vs
main.cpp
Shades.vs
Shades.fs
Shader.cpp
Shader.hpp
mesh.cpp
mesh.hpp
obj_loader.cpp
obj_loader.hpp
stb_image.h
Model.cpp
Model.hpp
Camera.cpp
Camera.hpp
Light.cpp
Light.hpp
Skybox.cpp
Skybox.hpp
skyShader.vs
skyShader.fs

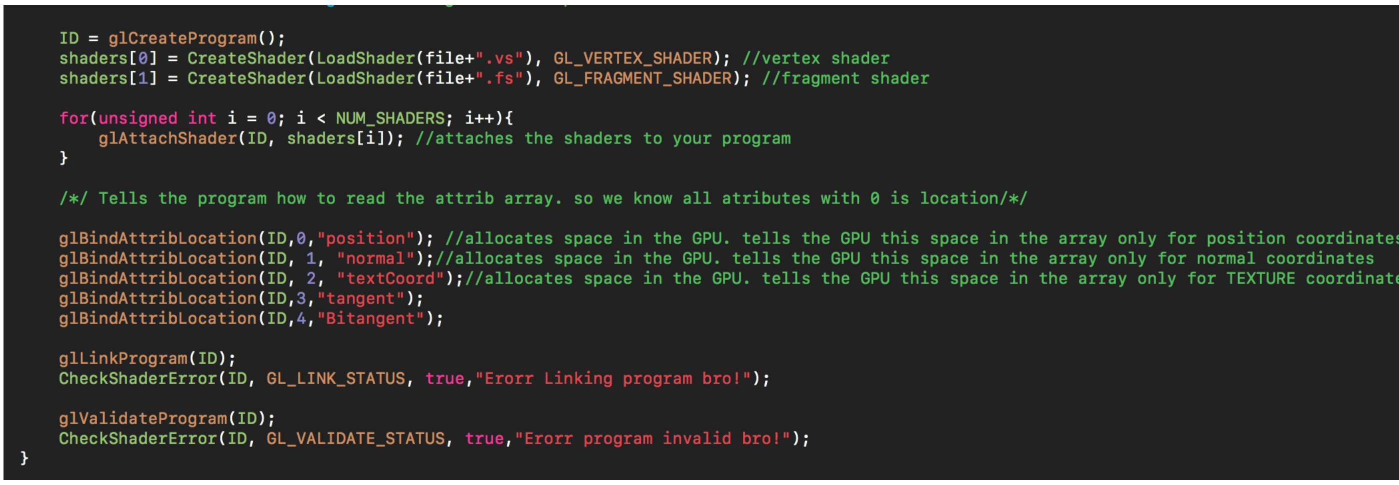
Figure 1: All the classes in the project

2 Class Structure/Abstraction

The program is broken down into multiple classes (refer to figure 1). The main class initializes and calls all the other objects, this is also where one would load their models in, given a specific path. This program was created to imitate a very basic fundamental game engine, therefore some aspects must be automated when creating models.

The following function automates the MVP matrix allocation.

```
void setCamera(GLuint ID, glm::vec3 translation, float angle, bool trans){

    glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)width /
        (float)height, 0.1f, 100.0f);

    GLuint proj = glGetUniformLocation(ID, "projection"); //projection IS UNIFORM
        VARIABLE IN SHADER
    glUniformMatrix4fv(proj, 1, GL_FALSE, &projection[0][0]);

    glm::mat4 view = camera.GetViewMatrix();
    GLuint viewI = glGetUniformLocation(ID, "view"); //view IS UNIFORM VARIABLE IN
        SHADER
    glUniformMatrix4fv(viewI, 1, GL_FALSE, &view[0][0]);
```



```
    ID = glCreateProgram();
    shaders[0] = CreateShader(LoadShader(file+".vs"), GL_VERTEX_SHADER); //vertex shader
    shaders[1] = CreateShader(LoadShader(file+".fs"), GL_FRAGMENT_SHADER); //fragment shader

    for(unsigned int i = 0; i < NUM_SHADERS; i++){
        glAttachShader(ID, shaders[i]); //attaches the shaders to your program
    }

    /*/ Tells the program how to read the attrib array. so we know all atributes with 0 is location/*/

    glBindAttribLocation(ID,0,"position"); //allocates space in the GPU. tells the GPU this space in the array only for position coordinates
    glBindAttribLocation(ID, 1, "normal");//allocates space in the GPU. tells the GPU this space in the array only for normal coordinates
    glBindAttribLocation(ID, 2, "textCoord");//allocates space in the GPU. tells the GPU this space in the array only for TEXTURE coordinat
    glBindAttribLocation(ID,3,"tangent");
    glBindAttribLocation(ID,4,"Bitangent");

    glLinkProgram(ID);
    CheckShaderError(ID, GL_LINK_STATUS, true,"Erorr Linking program bro!");

    glValidateProgram(ID);
    CheckShaderError(ID, GL_VALIDATE_STATUS, true,"Erorr program invalid bro!");
}
```

Figure 2: Shader Class file.

```
    glm::mat4 model(1.0f);

    if(trans){
        model = glm::translate(model, translation);
        model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 1.0f, 1.0f));
    }
    else{
        model = glm::mat4(1.0f);
    }

    GLuint modelI = glGetUniformLocation(ID, "model"); //model IS UNIFORM VARIABLE IN
        SHADER
    glUniformMatrix4fv(modelI, 1, GL_FALSE, &model[0][0]);
}
```

3 Shaders

Shaders are very important, this is one of the classes and functionalities of this engine. The constructor takes in a string file path of where both shaders are. In order for the shader loader to work, you must save both fragment and vertex shader as the same name. In this class we can bind the vertex attributes to tell the shader which location refers to which vertex attribute.

I used 3 different sets of pairs of shaders. 1 pair for the model loading of the human models, 1 pair for the light objects, so that it does not interfere with the normal models and 1 pair for the background/skybox.

The vertex shader sends the fragment position to the fragment shader. When we want to calculate light functions, we send the normal to the fragment shader to calculate the normal, but we want to remove the transfromation properties so we use the following command:

“vec3 normalT = mat3(transpose(inverse(model))) * vec3(normal.xyz);”

When we want to calculate the position of each fragment we use this function: $gl_position = projection * view * model * FragPos; //usedtobe * vec4(position, 1.0)$;

This will draw the position of the fragment according to the MVP matrices set in the main program, since they are UNIFORM they can be read in the shader aswell.

The main shader class contains 2 structs for initializing 2 different lights (see figure 3)

Then two different functions to calculate how light is rendered from the light source to the object.

For direction light, we use the function called “TheSun” takes in the parameters of the struct light.

When calculating directional light we need to calculate how the light hits the object, we calculate the light FROM the source TO the object. Therefore we use this function: $vec4lightDir = normalize(-sun.lightPos)$; //normalize returns a vector with the same direction as its parameter, v, but with length 1.” This function returns the vector of the light direction going towards the object.

For point lights we want the vector to calculate light FROM the object TO the light source. We use this function:

$vec4lightDir = normalize(plight.position - fragPos)$;

For my skybox/cubemap texture:

```
void main()
{
    TexCoords1 = position; //setting the texture as position!
    vec4 pos = projection * view * vec4(position,1.0);
    gl_Position = pos.xyww; //Z COORDINATE ISN'T USED FOR BACKGROUND IMAGES. W is the
        homogenous coordinate
}
```

The skybox doesnt need perspective, therefore we just set the Z coordinate to the homogenous coordinate, for a static background.

3.1 Animation

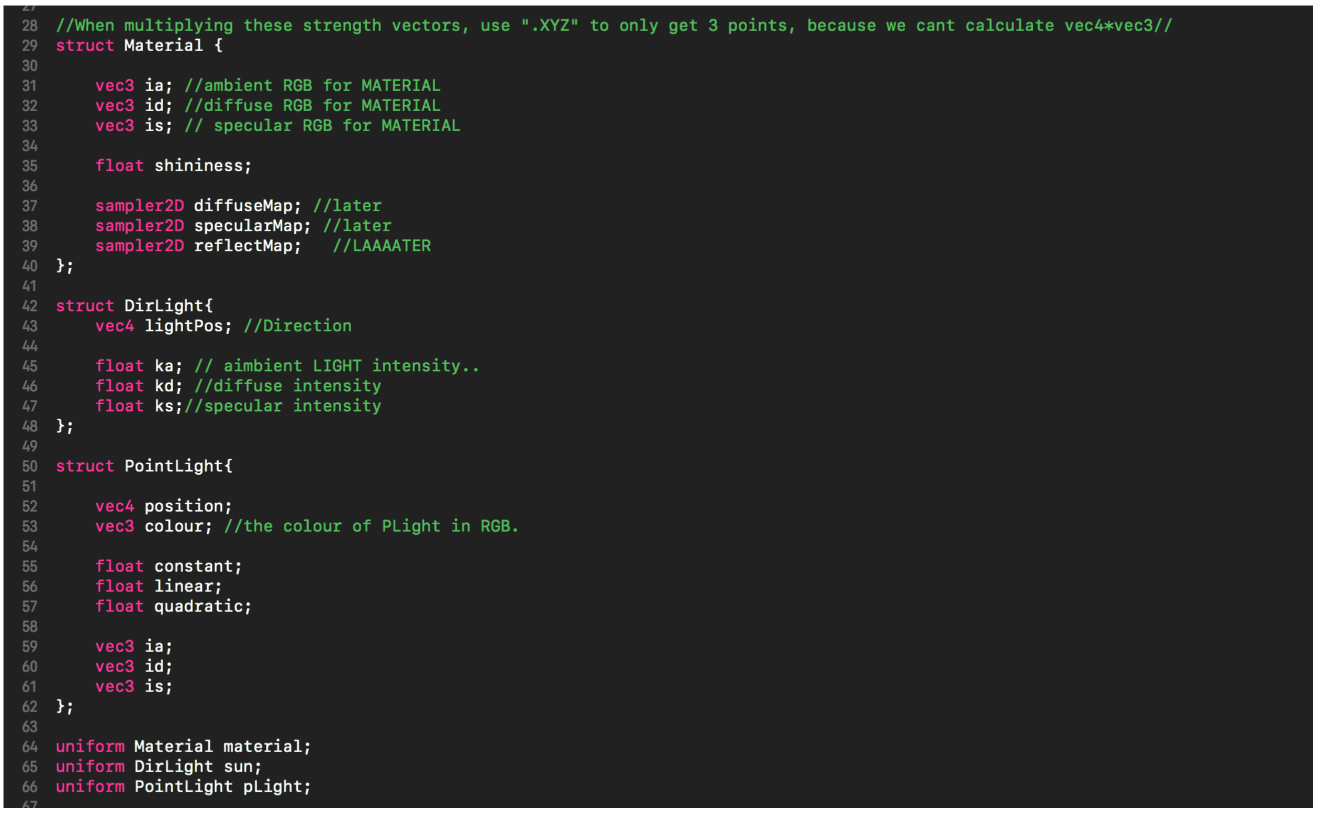
The following code shows how to use keys from the keyboard to influence vectors. This function was added by myself to effect only the point light colour. We change the DIFFUSE vector of the point light, since it is a "vec3 " we can associate XYZ with RGB and change these according to what button the user presses.

```
  if(glfwGetKey(window, GLFW_KEY_X) == GLFW_PRESS) {
      if(glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS) pLightColour.r -= 0.1f;
      if(glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS) pLightColour.g -= 0.1f;
      if(glfwGetKey(window, GLFW_KEY_B) == GLFW_PRESS) pLightColour.b -= 0.1f;
   }
   else if (glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS){
      if (glfwGetKey(window, GLFW_KEY_X) != GLFW_PRESS) pLightColour.r += 0.1f;
   }
   else if (glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS){
      if (glfwGetKey(window, GLFW_KEY_X) != GLFW_PRESS) pLightColour.g += 0.1f;
   }
   else if (glfwGetKey(window, GLFW_KEY_B) == GLFW_PRESS){
      if (glfwGetKey(window, GLFW_KEY_X) != GLFW_PRESS) pLightColour.b += 0.1f;
   }
}
```

Animation of moving light source:

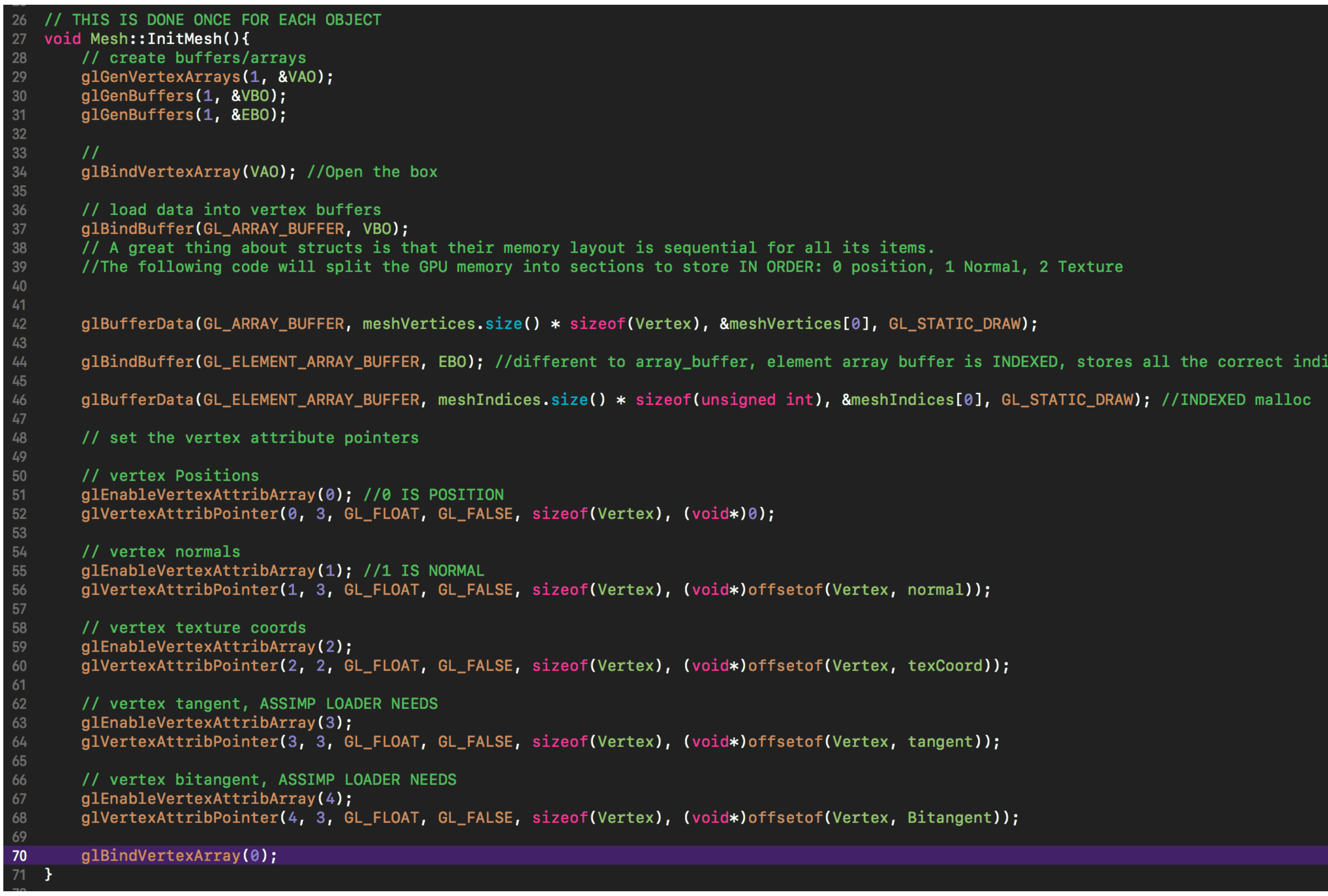
```
      float pX = 10.0f * sin(deltaTime);
      float pY = 15.0f +cos(deltaTime)+10.0f/tan(deltaTime)*sin(deltaTime); //10.0f
          is the offset
      float pZ = 10.0f * cos(deltaTime);
      pLightPos = glm::vec3(pX,pY,pZ);
```

pLightPos is then updated in the Position light struct, the vector position is assigned the values of the pLightPos vector. The SIN AND COS AND TAN give the object its smooth rotational values and angles. The above code snippet must be executed in the loop.



```
//When multiplying these strength vectors, use ".XYZ" to only get 3 points, because we cant calculate vec4*vec3//
struct Material {

    vec3 ia; //ambient RGB for MATERIAL
    vec3 id; //diffuse RGB for MATERIAL
    vec3 is; // specular RGB for MATERIAL

    float shininess;

    sampler2D diffuseMap; //later
    sampler2D specularMap; //later
    sampler2D reflectMap;   //LAAAATER
};

struct DirLight{
    vec4 lightPos; //Direction

    float ka; // aimbient LIGHT intensity..
    float kd; //diffuse intensity
    float ks;//specular intensity
};

struct PointLight{

    vec4 position;
    vec3 colour; //the colour of PLight in RGB.

    float constant;
    float linear;
    float quadratic;

    vec3 ia;
    vec3 id;
    vec3 is;
};

uniform Material material;
uniform DirLight sun;
uniform PointLight pLight;
```

Figure 3: Snippet demonstrating structs inside shader.



```cpp
// THIS IS DONE ONCE FOR EACH OBJECT
void Mesh::InitMesh(){
    // create buffers/arrays
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    //
    glBindVertexArray(VAO); //Open the box

    // load data into vertex buffers
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // A great thing about structs is that their memory layout is sequential for all its items.
    //The following code will split the GPU memory into sections to store IN ORDER: 0 position, 1 Normal, 2 Texture


    glBufferData(GL_ARRAY_BUFFER, meshVertices.size() * sizeof(Vertex), &meshVertices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO); //different to array_buffer, element array buffer is INDEXED, stores all the correct indi

    glBufferData(GL_ELEMENT_ARRAY_BUFFER, meshIndices.size() * sizeof(unsigned int), &meshIndices[0], GL_STATIC_DRAW); //INDEXED malloc

    // set the vertex attribute pointers

    // vertex Positions
    glEnableVertexAttribArray(0); //0 IS POSITION
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);

    // vertex normals
    glEnableVertexAttribArray(1); //1 IS NORMAL
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, normal));

    // vertex texture coords
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, texCoord));

    // vertex tangent, ASSIMP LOADER NEEDS
    glEnableVertexAttribArray(3);
    glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, tangent));

    // vertex bitangent, ASSIMP LOADER NEEDS
    glEnableVertexAttribArray(4);
    glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Bitangent));

    glBindVertexArray(0);
}
```

Figure 4: Snippet showing allocation of buffers and vertex array objects.

4 Model Loading

Model loading is constructed of multiple elements. Firstly, I used an open source external add on for blender which can automate human model creation (MANUELBASTIONILAB(2018)). Then I added and created the clothes objects myself.

Secondly in the program code, I used an open source external library (ASSIMP 2018), to read object files and store the object in a data structure scene, consisting of tree nodes for each model and each object of the model.

5 References

1. Model MANUELBASTIONILAB 2018: http://www.manuelbastioni.com/manuellab.php

2. Model Loader ASSIMP 2018: http://www.assimp.org/