3D Scene in OpenGL

Last F.C. Inter Conquer

Giacomo Benso

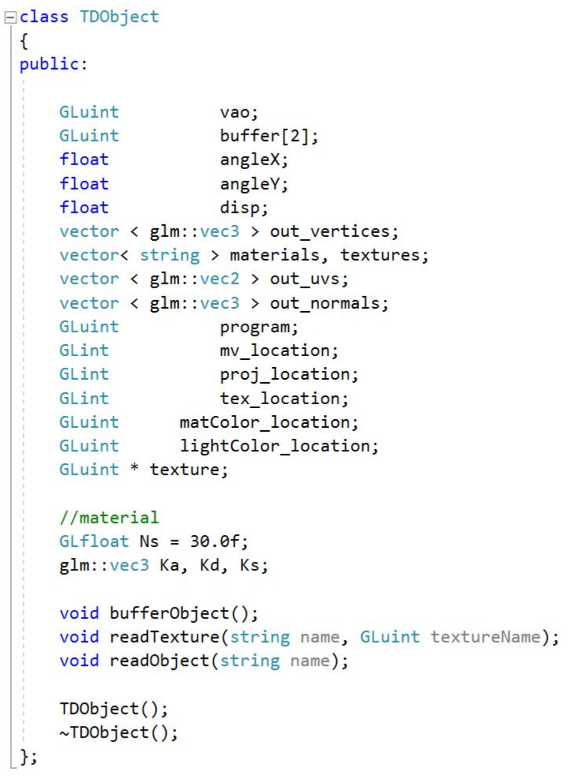
Introduction

A 3D scene with fps camera, animations, lights and materials created in OpenGL. The scene represents a floating island in a far galaxy which has been conquered by the F.C Internazionale in 2010.

Model and instancing

The objects in my scene have all been created and/or modified in blender. Some of the models (e.g. the flag) have been downloaded from the web and modified, others have been entirely developed by myself (Turbosquid.com, 2014). The object used in the scene are a floating island, a tree, a cloud, an F.C Inter flag, a castle and a sphere for the sun.

The Models in my implementation are encapsulated in the TDObject class. This class header file contains all the necessaire variables for each model such as the vector containing the vertices (mesh), the one for the textures, the values for the material (Ka,Kd,Ks,Ns), etc. (see Fig. 1).



```
class TDObject
{
public:

    GLuint          vao;
    GLuint          buffer[2];
    float           angleX;
    float           angleY;
    float           disp;
    vector < glm::vec3 > out_vertices;
    vector< string > materials, textures;
    vector < glm::vec2 > out_uvs;
    vector < glm::vec3 > out_normals;
    GLuint          program;
    GLint           mv_location;
    GLint           proj_location;
    GLint           tex_location;
    GLuint      matColor_location;
    GLuint      lightColor_location;
    GLuint * texture;

    //material
    GLfloat Ns = 30.0f;
    glm::vec3 Ka, Kd, Ks;

    void bufferObject();
    void readTexture(string name, GLuint textureName);
    void readObject(string name);

    TDObject();
    ~TDObject();
};
```

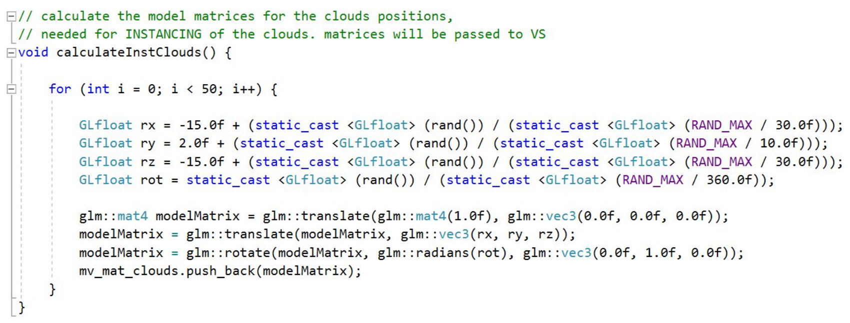
Figure 1: TDObject Class

The cpp file for the TDObject class contains the implementations of the functions for creating the buffers (for vertices, UV coordinates and Normals), for loading the file from .obj format and to read the texture image from .png format.

The models, as mentioned above, are imported as .obj files. The “readObject” function takes as parameter the string containing the location of the .obj file and reads the file line by line placing the vertices, triangle faces, locations for texture and materials, UV coordinates and normal coordinates in the right variables of the class. The TDObject class includes also a “program” variable which is used to store information about the shader for the object. The shader for each object is compiled before entering the render loop.

6 different models are present in my demo-scene, and two of them are instantiated multiple times using intancing.

The clouds are a “copy” of the same model which is drawn multiple times in different (random) positions passing a vector containing all the model matrices to the vertex shader. The model matrices are calculated using the rand() function to create random coordinates and a random angle which are then used to translate and rotate the model matrix (Fig. 2).



```
// calculate the model matrices for the clouds positions,
// needed for INSTANCING of the clouds. matrices will be passed to VS
void calculateInstClouds() {

    for (int i = 0; i < 50; i++) {

        GLfloat rx = -15.0f + (static_cast <GLfloat> (rand()) / (static_cast <GLfloat> (RAND_MAX / 30.0f)));
        GLfloat ry = 2.0f + (static_cast <GLfloat> (rand()) / (static_cast <GLfloat> (RAND_MAX / 10.0f)));
        GLfloat rz = -15.0f + (static_cast <GLfloat> (rand()) / (static_cast <GLfloat> (RAND_MAX / 30.0f)));
        GLfloat rot = static_cast <GLfloat> (rand()) / (static_cast <GLfloat> (RAND_MAX / 360.0f));

        glm::mat4 modelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 0.0f));
        modelMatrix = glm::translate(modelMatrix, glm::vec3(rx, ry, rz));
        modelMatrix = glm::rotate(modelMatrix, glm::radians(rot), glm::vec3(0.0f, 1.0f, 0.0f));
        mv_mat_clouds.push_back(modelMatrix);
    }
}
```

Figure 2: Random clouds

To retrieve the model matrices for each cloud in the vertex shader from the vector I use the gl_InstanceID variable which returns the instance of the current object. the matrix is then normally multiplied by the view_matrix, project_matrix and position as I also do for all the other objects (See AP 2 in the Appendix).
To enable the drawing of multiple instances through the shader I used the function “glDrawArraysInstanced” which allow to specify the number of instances to be drawn.

Also the tree have been instantiated in the same way of the clouds, the only difference is that the clouds have been instantiated 50 times and with semi-random coordinates and rotations while the trees only 15 times using defined positions and rotations.

Light and Material

Different lights and materials have been implemented in my demo scene.

The Light model used in my demo scene is the Phong model. This model reflects the real world light model and is based on 3 different types of light: ambient, diffuse and specular (the formula for the phong light model is shown in Fig. 3).

$$I_p = k_a i_a + \sum_{lights} k_d i_d (\hat{L} \cdot \hat{N}) + k_s i_s (\hat{R} \cdot \hat{V})^{shininess}$$

Figure 3: Phong Light Model (Padilla S., 2018)

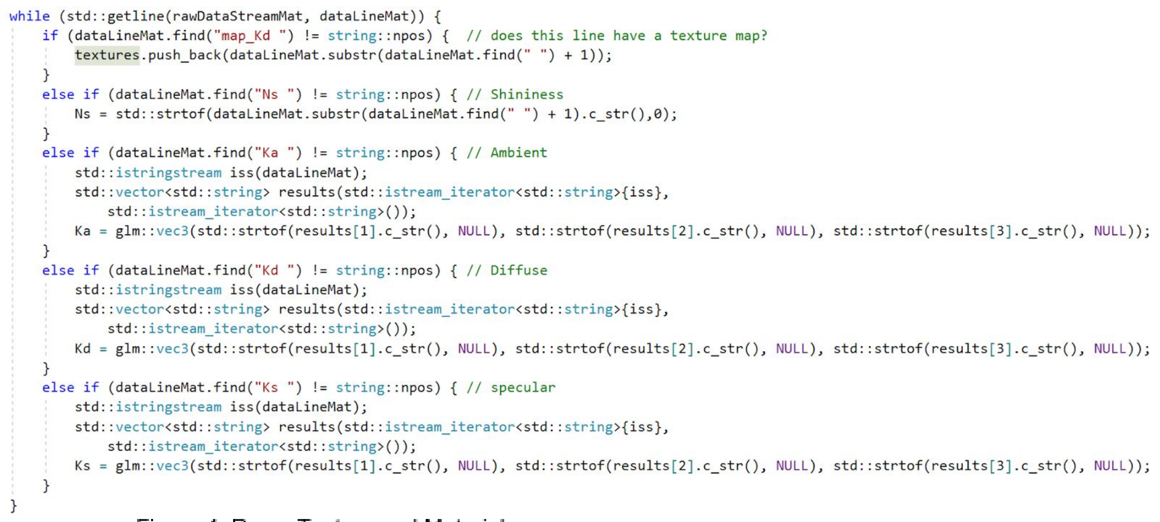
There are two light sources in my demo scene, The Sun and a torch.
The sun is the main source of light for the scene and it rotates all around the floating island, the torch initially starts “turned off”, but it can be turned on (See section on animations and interactions below).

Each of the lights in the scene has its own vector defining the color for both the diffuse and specular component. the ambient component is only one and it’s set to the usual value (0.25, 0.25, 0.25).

Each of the two lights is calculated separately in the fragment shader. The two lights are then added together to modify the color of the objects in the world depending on both lights if the torch is on, otherwise only the light of the sun is taken into account (see AP 2 in the Appendix).

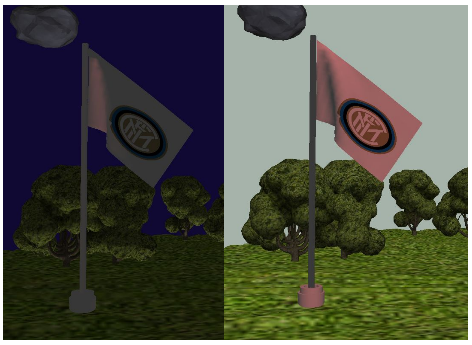
The torch light is what it can be called a flashlight, or a spotlight positioned at the camera position. A spotlight shoots all the light rays in a direction instead of dispersing them in the scene. To achieve this I had to define a position, a direction and a cut off angle, which are passed to the fragment shader (See AP 2 in the appendix). To identify the area of impact of my flashlight, in the fragment shader I compute the dot product between the “lightDir” (calculated in the beginning in my fragment shader) and the spotlight direction. Doing this I obtain the angle Theta which I then compare with the cutoff angle to determine where the limit for the spotlight is. To make the “cut” less neat I also calculate the angle Epsilon passing an outer cut off to the shader and when “clamping” the light that is also taken into consideration.

Another factor that contributes to the color of an object in my scene is the material. The material colours are defined in the .mtl file and I parse their values for each object (Ka (ambient), Kd (diffuse), Ks (specular) and Ns (shininess)). These values are parsed and saved in the object as well as the texture which is also defined in the .mtl file (Fig. 4).



```
while (std::getline(rawDataStreamMat, dataLineMat)) {
    if (dataLineMat.find("map_Kd ") != string::npos) {  // does this line have a texture map?
        textures.push_back(dataLineMat.substr(dataLineMat.find(" ") + 1));
    }
    else if (dataLineMat.find("Ns ") != string::npos) { // Shininess
        Ns = std::strtof(dataLineMat.substr(dataLineMat.find(" ") + 1).c_str(),0);
    }
    else if (dataLineMat.find("Ka ") != string::npos) { // Ambient
        std::istringstream iss(dataLineMat);
        std::vector<std::string> results(std::istream_iterator<std::string>{iss},
            std::istream_iterator<std::string>());
        Ka = glm::vec3(std::strtof(results[1].c_str(), NULL), std::strtof(results[2].c_str(), NULL), std::strtof(results[3].c_str(), NULL));
    }
    else if (dataLineMat.find("Kd ") != string::npos) { // Diffuse
        std::istringstream iss(dataLineMat);
        std::vector<std::string> results(std::istream_iterator<std::string>{iss},
            std::istream_iterator<std::string>());
        Kd = glm::vec3(std::strtof(results[1].c_str(), NULL), std::strtof(results[2].c_str(), NULL), std::strtof(results[3].c_str(), NULL));
    }
    else if (dataLineMat.find("Ks ") != string::npos) { // specular
        std::istringstream iss(dataLineMat);
        std::vector<std::string> results(std::istream_iterator<std::string>{iss},
            std::istream_iterator<std::string>());
        Ks = glm::vec3(std::strtof(results[1].c_str(), NULL), std::strtof(results[2].c_str(), NULL), std::strtof(results[3].c_str(), NULL));
    }
}
```

Figure 4: Parse Texture and Material



In the fragment shader the material values for each object are multiplied by the light to give a unique look to each object, an example is the flag which has the vector corresponding to the diffuse component set to reflect the red light more than the other colours (Fig. 5 shows how the flag turns red when the sunlight reaches its surface).

Figure 5: Flag reflection

The texture colour for the each fragment is also multiplied by the material colour to make the texture have an effect on the objects. To calculate how the texture is positioned on the object the technique used is UV mapping, this technique maps the coordinate of the texture (in my case a .png image) to the UV map of the object created in blender.

The lighting model in my scene also takes into account the attenuation component, which makes the light affecting an object more when the light source is closer to the object (See AP 2 in the Appendix). To define the level of attenuation the fragment distance to the light source is calculated in the shader ant then the light is multiplied by this value.

Animation, Interaction and Camera System

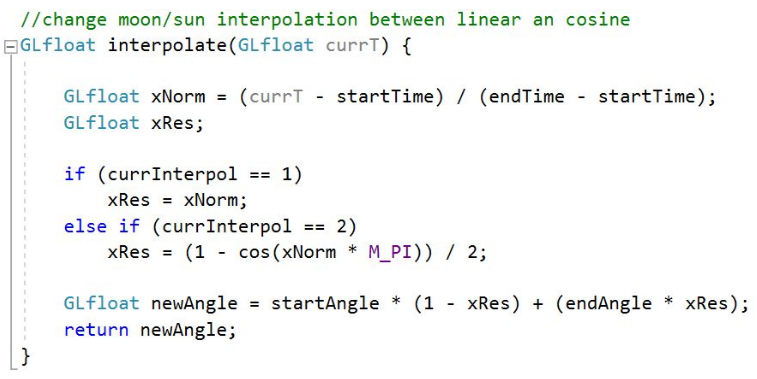
In my scene I implemented three different major animation and some user interaction designed to influence the camera system, lights and animations. All the animations have been developed using interpolation (linear and cosine). My interpolations use a start and an end time for the animation and the current time to calculate the position, angle or colours depending on the latter. For both linear and cosine interpolation the first thing to do is to calculate the normal value (top-left formula in Fig. 6) then for linear interpolation the formula left-bottom formula in pic is applied. For the cosine interpolation we need one more step, shown on the top right of pic before applying the bottom right formula in pic.

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} \qquad x_{cos} = \frac{1 - \cos(x_{norm} * \pi)}{2}$$

$$k_{new} = \mathbf{k_1} * (\mathbf{1} - \mathbf{k_{norm}}) + \mathbf{k_2} * \mathbf{k_{norm}} \qquad k_{new} = \mathbf{k_1} * (\mathbf{1} - \mathbf{k_{cos}}) + \mathbf{k_2} * \mathbf{k_{cos}}$$

Figure 6: Interpolation formulas (Padilla S., 2018)

Animation of the sun (light source):



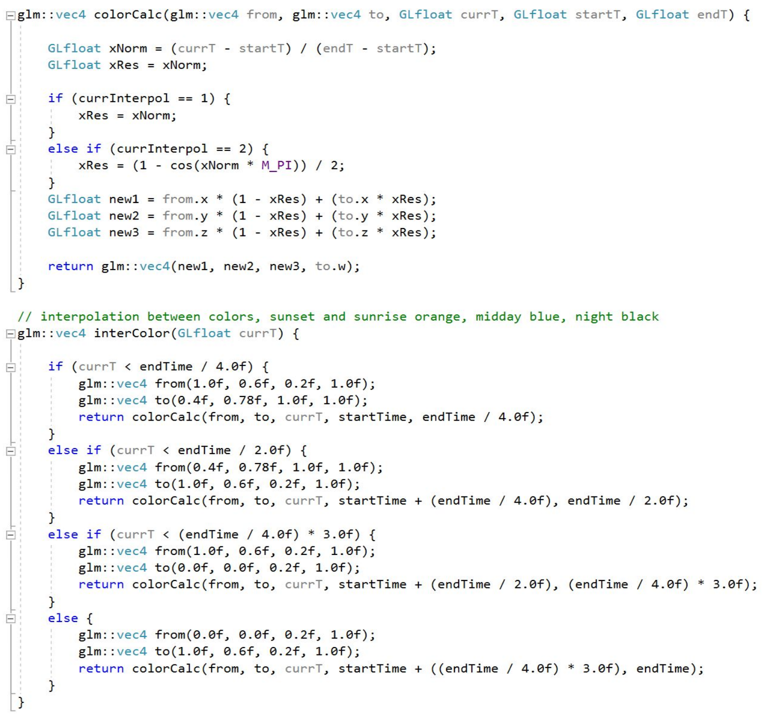
```
//change moon/sun interpolation between linear an cosine
GLfloat interpolate(GLfloat currT) {

    GLfloat xNorm = (currT - startTime) / (endTime - startTime);
    GLfloat xRes;

    if (currInterpol == 1)
        xRes = xNorm;
    else if (currInterpol == 2)
        xRes = (1 - cos(xNorm * M_PI)) / 2;

    GLfloat newAngle = startAngle * (1 - xRes) + (endAngle * xRes);
    return newAngle;
}
```

Figure 7: Sun animation

The sun moves all around my floating island to give a day and night effect to the scene. Changing the position of the sun changes the angle in which the light is reflected and thus the color of the items (darker or lighter).

This animation has been implemented using linear interpolation. A start and an end angle for the animation are given (0-360 degrees); depending on the current time (passed to the render function) the current angle is calculated using linear interpolation as shown in Figure 7 and used to calculate the position of the sun.

Animation of the sky:



```
glm::vec4 colorCalc(glm::vec4 from, glm::vec4 to, GLfloat currT, GLfloat startT, GLfloat endT) {

    GLfloat xNorm = (currT - startT) / (endT - startT);
    GLfloat xRes = xNorm;

    if (currInterpol == 1) {
        xRes = xNorm;
    }
    else if (currInterpol == 2) {
        xRes = (1 - cos(xNorm * M_PI)) / 2;
    }
    GLfloat new1 = from.x * (1 - xRes) + (to.x * xRes);
    GLfloat new2 = from.y * (1 - xRes) + (to.y * xRes);
    GLfloat new3 = from.z * (1 - xRes) + (to.z * xRes);

    return glm::vec4(new1, new2, new3, to.w);
}

// interpolation between colors, sunset and sunrise orange, midday blue, night black
glm::vec4 interColor(GLfloat currT) {

    if (currT < endTime / 4.0f) {
        glm::vec4 from(1.0f, 0.6f, 0.2f, 1.0f);
        glm::vec4 to(0.4f, 0.78f, 1.0f, 1.0f);
        return colorCalc(from, to, currT, startTime, endTime / 4.0f);
    }
    else if (currT < endTime / 2.0f) {
        glm::vec4 from(0.4f, 0.78f, 1.0f, 1.0f);
        glm::vec4 to(1.0f, 0.6f, 0.2f, 1.0f);
        return colorCalc(from, to, currT, startTime + (endTime / 4.0f), endTime / 2.0f);
    }
    else if (currT < (endTime / 4.0f) * 3.0f) {
        glm::vec4 from(1.0f, 0.6f, 0.2f, 1.0f);
        glm::vec4 to(0.0f, 0.0f, 0.2f, 1.0f);
        return colorCalc(from, to, currT, startTime + (endTime / 2.0f), (endTime / 4.0f) * 3.0f);
    }
    else {
        glm::vec4 from(0.0f, 0.0f, 0.2f, 1.0f);
        glm::vec4 to(1.0f, 0.6f, 0.2f, 1.0f);
        return colorCalc(from, to, currT, startTime + ((endTime / 4.0f) * 3.0f), endTime);
    }
}
```

Figure 8: Color interpolation

The background color is animated interpolating different colours depending on the current time (similarly to the sun animation described above). the colours are interpolating between orange and light blue for the first quarter of the animation (sunrise to midday), then from light blue to orange for the second quarter (midday to sunset), then from orange to dark blue for the third quarter (sunset to midnight) and ultimately from dark blue to orange (midnight to sunrise). The implementation is shown in Figure 8.

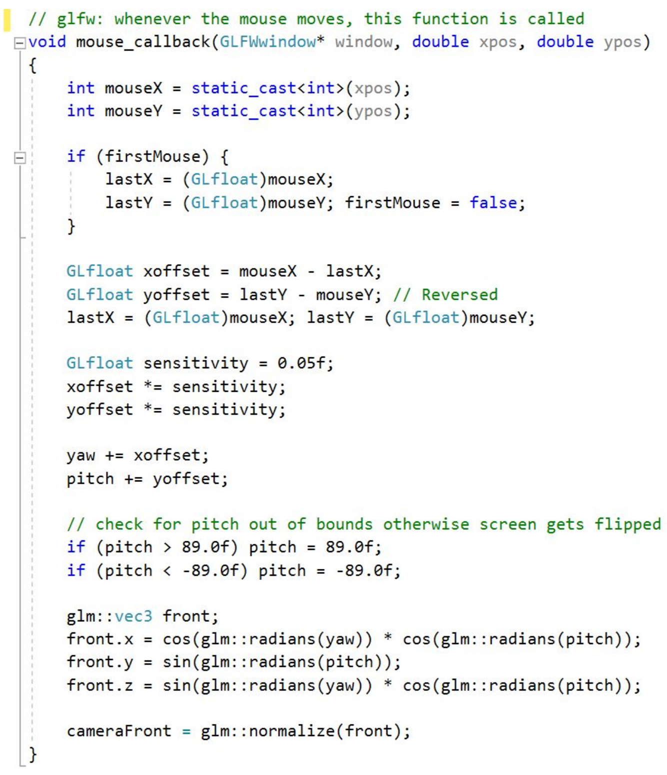
Interaction with the sun and the sky:
When pressing the buttons “1” or “2” the interpolation for both the sun translation and the sky color is interchanged between linear and cosine. When using the linear interpolation the sun moves always at the same speed and the colour of the sky changes gradually too. When the cosine interpolation is used the sun speed follows the cosine function and the colour interchange goes side by side with the sun movement (See Fig. 7 and 8).

Interaction with the torch (second light source) :

- **On and off**: The torch can be turned on and off pressing the keys “K” and “L”. Those buttons set the value of the variable “OnOff” to either 0 or 1. This variable is then passed to the vertex shader and multiplied by the diffuse and specular component of the light source so that when the value is 0 the light does not affect the colour of the fragments and when it’s 1 the light is taken into consideration normally (See AP 2 in the Appendix).
- **Colour**: The colour of the torch can also be changed. Pressing “I”, “O”, or “P” the light colour can be set to red, blue or “disco”. To set the light to either blue or red the “rgb” values in the corresponding two vec3 (diffuse and specular components of the light) are simply changed to the corresponding colours. for the “disco” colour I use linear

interpolation to interpolate colours and give a fading effect to the light (Fig. 8). The result of the interpolation (a vec3) is used as both diffuse and specular component for the torch light. The interpolation is divided into two parts, one going from colour 1 to colour 2 and the second doing the opposite, in this way we ensure no abrupt change between colours happen.

Camera System:
For my scene I decided to use an fps camera. This type of camera allows the exploration of the whole demo scene in all the possible spots and from all possible angles.
To define my camera I had first of all to define its position in world space, the direction it's looking at, a vector pointing to the right and a vector pointing upwards from the camera.
I then had to define the lookat matrix which gets multiplied by the model matrix to change the perspective in which the objects are seen in the scene.



```
// glfw: whenever the mouse moves, this function is called
void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    int mouseX = static_cast<int>(xpos);
    int mouseY = static_cast<int>(ypos);

    if (firstMouse) {
        lastX = (GLfloat)mouseX;
        lastY = (GLfloat)mouseY; firstMouse = false;
    }

    GLfloat xoffset = mouseX - lastX;
    GLfloat yoffset = lastY - mouseY; // Reversed
    lastX = (GLfloat)mouseX; lastY = (GLfloat)mouseY;

    GLfloat sensitivity = 0.05f;
    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw += xoffset;
    pitch += yoffset;

    // check for pitch out of bounds otherwise screen gets flipped
    if (pitch > 89.0f) pitch = 89.0f;
    if (pitch < -89.0f) pitch = -89.0f;

    glm::vec3 front;
    front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    front.y = sin(glm::radians(pitch));
    front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));

    cameraFront = glm::normalize(front);
}
```

Figure 9: Camera movement

For an fps camera I needed to define the keyboard interaction for the movement and the mouse interaction to enable the “lookaround”.
Pressing one of the keys “W”, “A”, “S” or “D” the cameraPosition variable is changed, this variable is then used in the LookAt matrix to define the position of the camera (Fig. 9).
Moving the mouse changes the direction vector depending on the position of the mouse in the previous loop cycle (as this is checked in the loop), the two angles of movement (horizontal and vertical) generated by the mouse movement are used to calculate the vector.

References

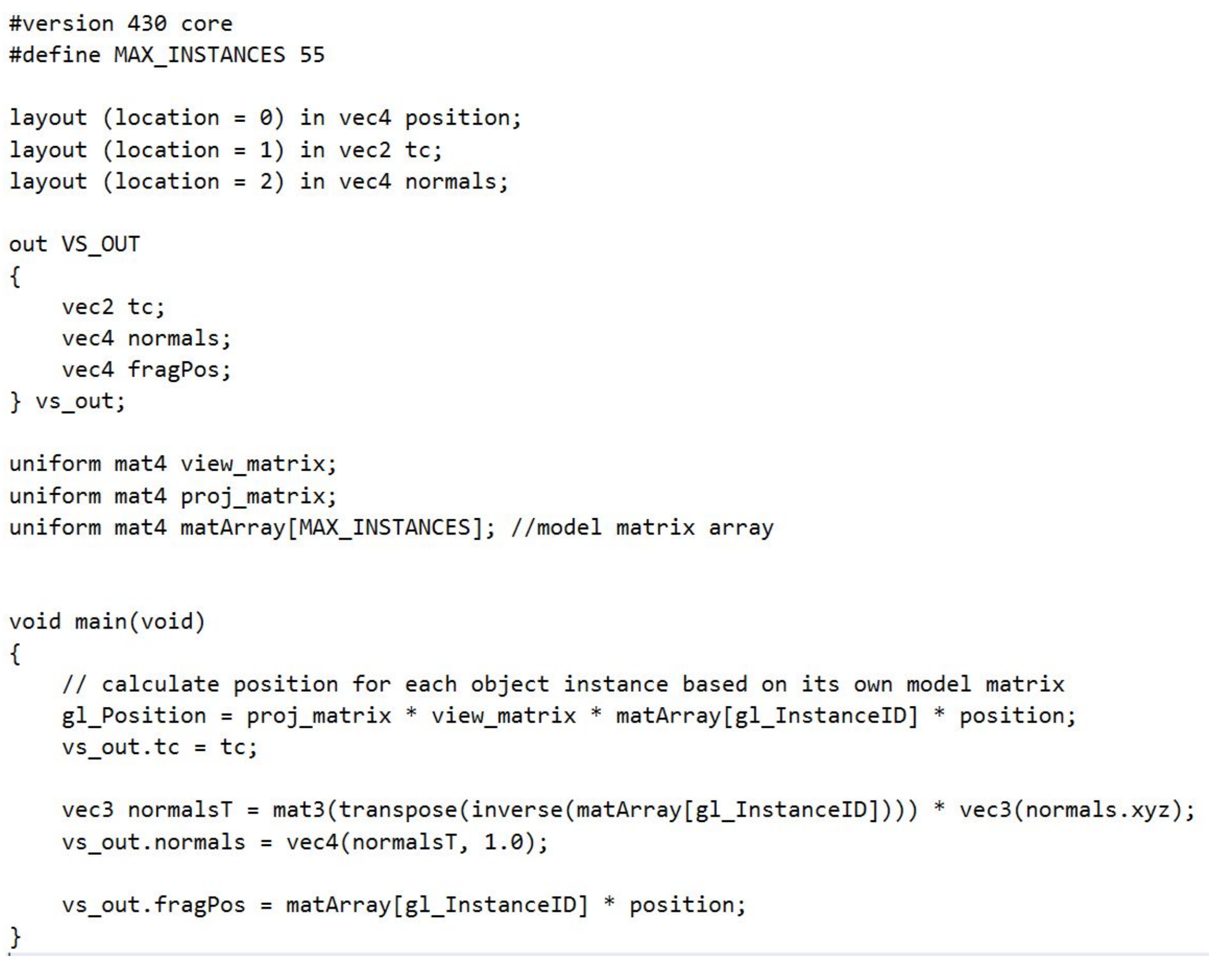
Turbosquid.com. (2014). *3D Models for Download | TurboSquid*. [online] Available at: https://www.turbosquid.com/Search/3D-Models [Accessed 28 Nov. 2018].

Padilla Stefano. (2018). *Lights and Framebuffer. F20GA, 3D Graphics and Animations.* [Lecture Notes] Heriot-Watt University [Accessed 28 Nov. 2018].

Padilla Stefano. (2018). *Animations 1. F20GA, 3D Graphics and Animations.* [Lecture Notes] Heriot-Watt University [Accessed 28 Nov. 2018].

Appendix

AP 1: Vertex shader for instanced models



```
#version 430 core
#define MAX_INSTANCES 55

layout (location = 0) in vec4 position;
layout (location = 1) in vec2 tc;
layout (location = 2) in vec4 normals;

out VS_OUT
{
    vec2 tc;
    vec4 normals;
    vec4 fragPos;
} vs_out;

uniform mat4 view_matrix;
uniform mat4 proj_matrix;
uniform mat4 matArray[MAX_INSTANCES]; //model matrix array


void main(void)
{
    // calculate position for each object instance based on its own model matrix
    gl_Position = proj_matrix * view_matrix * matArray[gl_InstanceID] * position;
    vs_out.tc = tc;

    vec3 normalsT = mat3(transpose(inverse(matArray[gl_InstanceID]))) * vec3(normals.xyz);
    vs_out.normals = vec4(normalsT, 1.0);

    vs_out.fragPos = matArray[gl_InstanceID] * position;
}
```

AP 2: Fragment Shader for models with multiple lights

```
uniform vec3 la;        // Ambient light colour
uniform vec4 ka;        // Ambient color material
uniform vec3 ld;        // diffuse colour sun
uniform vec3 ldT;           // diffuse colour torch
uniform vec4 kd;            // Diffuse color material
uniform vec3 ls;            // specular colour sun
uniform vec3 lsT;           // specular colour torch
uniform vec4 ks;            // specular color material
uniform float shininess;        // shininess material

uniform float lightConstant;
uniform float lightLinear;
uniform float lightQuadratic;
uniform vec4 lightSpotDirection;
uniform float lightSpotCutOff;
uniform float lightSpotOuterCutOff;
uniform float onoff; //torch on or off

void main(void){
    vec4 lightDir = normalize(lightPositionT - fs_in.fragPos); //torch
    vec4 lightDir2 = normalize(lightPosition - fs_in.fragPos); //moon

    // Ambient (used only once)
    vec3 ambient = la * texture(tex, fs_in.tc).rgb;

    // Diffuse torch
    float diff = max(dot(normalize(fs_in.normals), lightDir), 0.0);
    vec3 diffuse = kd.rgb * ldT * diff * texture(tex, fs_in.tc).rgb;

    //diffuse moon
    float diff2 = max(dot(normalize(fs_in.normals), lightDir2), 0.0);
    vec3 diffuse2 =  ld * kd.rgb * diff2 * texture(tex, fs_in.tc).rgb;

    // Specular torch
    vec4 viewDir = normalize(viewPosition - fs_in.fragPos);
    vec4 reflectDir = reflect(-lightDir, normalize(fs_in.normals));
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
    vec3 specular = ks.rgb * lsT * spec;

    // Specular moon
    vec4 viewDir2 = normalize(viewPosition - fs_in.fragPos);
    vec4 reflectDir2 = reflect(-lightDir2, normalize(fs_in.normals));
    float spec2 = pow(max(dot(viewDir2, reflectDir2), 0.0), shininess);
    vec3 specular2 = ls * ks.rgb * spec2;

    // Attenuation torch
    float distance    = length(lightPositionT - fs_in.fragPos);
    float attenuation = 1.0f / (lightConstant + lightLinear * distance + lightQuadratic * (distance * distance));

    // Attenuation moon
    float distance2    = length(lightPosition - fs_in.fragPos);
    float attenuation2 = 1.0f / (lightConstant + lightLinear * distance2 + lightQuadratic * (distance2 * distance2));

    // Spot light torch
    float theta = dot(lightDir, normalize(-lightSpotDirection));
    float epsilon = (lightSpotCutOff - lightSpotOuterCutOff);
    float intensity = clamp((theta - lightSpotOuterCutOff) / epsilon, 0.0, 1.0);

    ambient  *= attenuation2;

    diffuse  *= attenuation * intensity * onoff; //check torch is on or off
    specular *= attenuation * intensity * onoff;
    diffuse2  *= attenuation2;
    specular2 *= attenuation2;

    // Final Color adding lights
    color = vec4(ambient + diffuse + diffuse2 + specular + specular2, 1.0);
}
```