MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA, ROMANIA

Western Town

Project Documentation

Graphic Processing

Vlad Mihali

Group: 30434

1 Contents

2 Subject specification

The main objective of the project consists in to making a photorealistic scene of 3D objects using OpenGL library. The user directly manipulates by mouse and keyboard inputs the scene of objects. The scene represents a western town.

The objectives that help in making scene are:

- Visualization of the scene: scaling, translation, rotation, camera movement
  - using keyboard and mouse
  - using animation
- Light sources (two different lights)
- Viewing solid, wireframe objects, polygonal and smooth surfaces
- Texture mapping and materials
  - Textures quality and level of detail
  - Textures mapping on objects
- Shadow computation
- Animation of object components
- Scene complexity, detailed modeling, algorithms development and implementation (objects generation, fog, sandstorm), different types of light sources (global, local), skybox.

3 Scenario

3.1 scene and objects description

Objects in the scene are all related to western world making a little western town. The town is composed of bank, hotel, sheriff building, windmill, bar and other buildings

3.2 functionalities

Camera can be control with mouse and keyboard or through animations. Camera has 4 animation from which 3 are primitive animation, that means rotate left, rotate right and move forward. The 4th animation is composed from the otter primitives resulting in moving forward, then rotate left and then move forward. This complex animation helps the user to "walk" from the bar to the hotel.

As about lighting, we have two light sources: one directional light and one point light. The directional one can be rotated, and you can see the effect taking place by looking at the shadows. The point light is represented as a light bulb in the bar and can be turned on and off.

The sign outside the bar has animation implemented and can be turned on and off.

The nanosuit can be controlled by rotating him to right and left.

The sandstorm that happens on the street also can be turned on and off.

The other functionalities are turned on and off the fog, view wireframe/solid, view the shadow map.

4 Implementation details

4.1 functions and special algorithms

In this project are some special implementation such as:

- fog for skybox;
- sandstorm;
- camera animation special implementation;
- turn off/on implantation;
- shadow PCF;

4.1.1 POSSIBLE SOLUTIONS

Fog for skybox is implemented same as other fog:

```
in vec3 textureCoordinates;
out vec4 color;

uniform samplerCube skybox;

float computeFog()
{
 float fogDensity = 1.0f;
 float fragmentDistance = length(textureCoordinates);
 float fogFactor = exp(-pow(fragmentDistance * fogDensity, 2));

 return clamp(fogFactor, 0.0f, 1.0f);
}

void main()
{
    vec4 auxColor = texture(skybox, textureCoordinates);
    float fogFactor = computeFog();
       vec4 fogColor = vec4(0.93, 0.79, 0.69, 1.0f);
       color = mix(fogColor, auxColor, fogFactor);
}
```

Sandstorm is implemented using particle structure which consist of one vec3 for position. We can bound sandstorm in a box. Sandstorm is array of particles with a size that we specify (ex. 10000). Initialize at the start of the program random bounded coordinates for particles. In render scene we translate with a specific speed the particle, if the particle rich the bound “dies” and is “born” in the opposite location. In the comments we have implanted also the rotation of the particle, but we don’t need now because the particle is a sphere.

```
void initParticles() {
       for (int i = 0; i < nr_particles; i++) {
              float x = (rand() % (int)(boundRight - boundLeft) + boundLeft) +
(float)rand()/RAND_MAX;
              float y = (rand() % (int)(boundTop - boundBottom) + boundBottom)+
(float)rand() / RAND_MAX;
              float z = (rand() % (int)(boundFront - boundBack) + boundBack)+
(float)rand() / RAND_MAX;
              /*sandParticles[i].angle = rand() % 180;
```

```
			switch (i % 3) {
			case 0:
				sandParticles[i].Rotate = glm::vec3(1.0f, 0.0f, 0.0f);
				break;
			case 1:
				sandParticles[i].Rotate = glm::vec3(0.0f, 1.0f, 0.0f);
				break;
			case 2:
				sandParticles[i].Rotate = glm::vec3(0.0f, 0.0f, 1.0f);
				break;
			}*/
			sandParticles[i].Position = glm::vec3(x,y,z);
		}
}
```

```
void renderParticles(gps::Shader shader) {

	/*myCustomShader.useShaderProgram();*/

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

		if (sandParticles[i].Position.x < boundLeft) {

			/*float x = rand() % (int)(boundRight - boundLeft) + boundLeft;

			float y = rand() % (int)(boundTop - boundBottom) + boundBottom;

			float z = rand() % (int)(boundFront - boundBack) + boundBack;

			sandParticles[i].Position = glm::vec3(x, y, z);*/

			sandParticles[i].Position.x = boundRight;

		}

		sandParticles[i].Position.x -= particleSpeed;

		//model = glm::rotate(glm::mat4(1.0f), glm::radians(sandParticles[i].angle), 
sandParticles[i].Rotate);

		model = glm::translate(glm::mat4(1.0f), sandParticles[i].Position);

		model = glm::scale(model, glm::vec3(0.1f));

		glUniformMatrix4fv(glGetUniformLocation(shader.shaderProgram, "model"), 1, 
GL_FALSE, glm::value_ptr(model));

		glUniform1i(glGetUniformLocation(shader.shaderProgram, "isReflect"), false);
```

```
                particleModel.Draw(shader);

        }

}
```

Camera animation is interesting implemented. Has 3 primitives and you can create other animation by using those.

```
void cameraAuto() {
        if (isAutoMoveForward) {
                myCamera.move(gps::MOVE_FORWARD, cameraSpeed);
        }

        if (isAutoRotateLeft) {
                myCamera.rotate(-rotateSpeed, 0.0f);
        }

        if (isAutoRotateRight) {
                myCamera.rotate(rotateSpeed, 0.0f);
        }
}
```

Template of compose animation:

If time1 < time then increment time1 and set animation Booleans else other animation

Turn off/on implementation is simple procedure. Add a Boolean variable to main which is toggle by some key pressing and is pass as uniform in GLSL.

PCF, or percentage-closer filtering is a term that hosts many different filtering functions that produce softer shadows, making them appear less blocky or hard. The idea is to sample more than once from the depth map, each time with slightly different texture coordinates. For each individual sample we check whether it is in shadow or not. All the sub-results are then combined and averaged and we get a nice soft looking shadow.

One simple implementation of PCF is to simply sample the surrounding texels of the depth map and average the results:

```
        vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
        for(int x = -1; x <= 1; ++x)
        {
                for(int y = -1; y <= 1; ++y)
                {
                        float pcfDepth = texture(shadowMap, normalizedCoords.xy + vec2(x, y)
* texelSize).r;
                        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
```

```
		}
	}
	shadow /= 9.0;
```

4.1.2 THE MOTIVATION OF THE CHOSEN APPROACH

I chose to use fog on skybox because we can add any fog we want. Also, skybox fog density can be different by the other fog density. Skybox is like a very far fog.

I chose to make camera animation “framework” to easily and fast making camera animation.

My motivation to use the implementation of sandstorm is for simplicity, because a particle system is more difficult to make.

For on and off, I chose this because we eliminate int uniforms with values 0 and 1 and we can connect to keyboard more easily.

The main reason I used PCF is to avoid pixel shadows.

4.2 graphics model

In this project, the objects are represented as polygons, some just primitive triangles.

Advantages:

-Ubiquitous in computer graphics

–Simple and straightforward

- Used as intermediate form for many other data structures

Issues:

- Approximation of curved surfaces

- Accuracy depends extremely on the number of polygons e.g. zoom, scale

- Texture mapping

- Smooth surfaces

- Shading algorithms

The objectes have:

- v - List of geometric vertices

- vt - List of texture coordinates
- vn - List of vertices normal
- f - Polygonal face element (3 or more) ex:
    - f 3/1 4/2 5/3
    - f 6/4/1 3/5/3 7/6/5

4.3 data structures

Polygonal model - Data structure

The data structure consists of four arrays:

- Object Array - an array of all surfaces of the 3D object modelled through polygons. A position of the array is a list of indices of the polygonal components.
- Surface Array - it is the set of polygons that describes the surface. A position in the array describes a polygon as a list of indices of the polygon vertices and an index of the polygon normal.
- Vertex Array - describes the information that is associated to each vertex. Three coordinates and a normal vector describe a vertex. In fact, the normal vector has three components: its projections onto the coordinate axes - Nx, Ny and Nz.
- Polynormal Array - a set of normal vectors of the polygons. Each vector has three vector components.

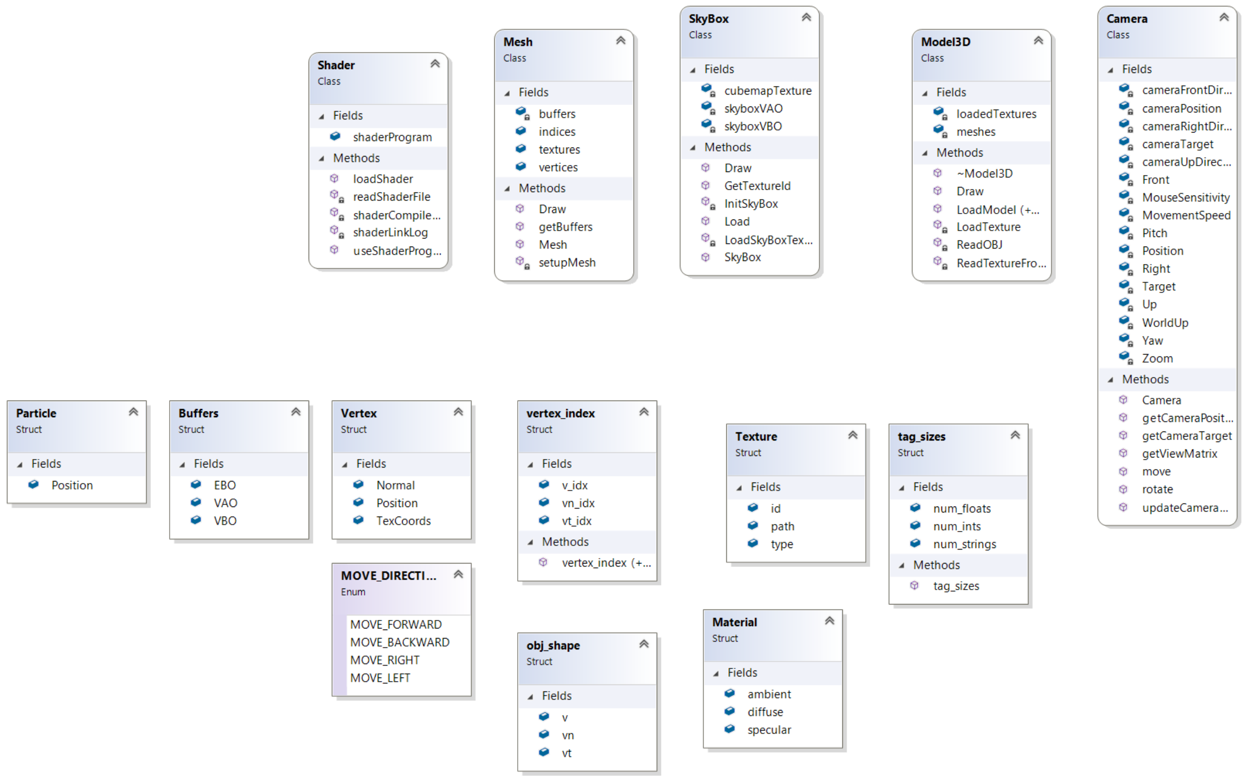
Disadvantage:

- Edges are shared between adjacent polygons. The edge is not explicitly contained in the data structure.
- An edge is processed twice.

In this project is implemented a structure for particle with one vec3 that is position, but in future can be extended to retained color, rotation, lifespan and other parameters.

```
struct Particle {
	glm::vec3 Position;
	/*float angle;
	glm::vec3 Rotate;*/
};
```

4.4 class hierarchy



5 Graphical user interface presentation / user manual

Mouse movement –Camera rotate

Mouse scroll – Camera zoom

W Key – Camera move forward

S Key – Camera move backward

A Key – Camera move left

D Key – Camera move right

1 Key – Camera rotate right animation

2 Key – Camera rotate left animation

3 Key – Camera move forward animation

4 Key – Camera animation

5 Key – Stop Camera animation

Q Key – nanosuit rotate left

E Key – nanosuit rotate right

G Key – view wireframe mode

H Key – view solid mode

J Key – directional light rotates left

L Key – directional light rotates right

Z Key – Start/Stop object animation (sign animation)

X Key – Enable/Disable fog

C Key – Turn off/on point light

V Key – Start/Stop sandstorm

M Key – view shadow map mode

6 Conclusions and further developments

In conclusion, this project is representing a photo-realistic scene of a western town with some functionalities. This are implemented from the next accomplished tasks: Visualization of the scene (animation included), light sources (directional, point), viewing solid, wireframe objects, polygonal and smooth surfaces, texture mapping and materials, shadow computation, animation of object components (the sign), scene complexity, detailed modeling, objects generation (sandstorm), fog, skybox.

The project can be extended furthermore by adding object collision (implement basic physics), ambient occlusion, motion blur, first person actions, 2D components over 3D such as life bar and/or score. Also, the project can be improved: by making shadows for point light and shadow overlap; by replacing sandstorm implementation with particle system for better performance; by extending the scene and make the interior of more buildings.

7 References

[1] https://learnopengl.com
[2] http://free3D.com
[3] www.turbosquid.com

[4] laboratory works