Submission 1: Thalassophobia

Andrei Ioan Georgescu - 0927863

Tobias Watzinger – 1325884

Requirements

Gameplay

The second level(After the door) is just jumping on different platforms and getting another key; you also have to jump over pillars before if you don't cheat. Thus, there is 3D gameplay.

Effects

Complex Objects

We are loading a Zombie that walks around, which we see as a sufficient complex object.

Animated Objects

We created a different mesh from the hand of the original zombie. Then we used hierarchical animation to have the arm have the same motion (forwards and backwards) as our zombie with an added rotation.

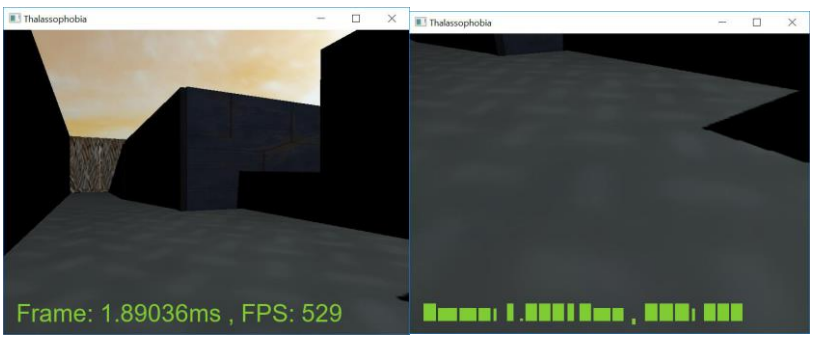
View-Frustum Culling

A bit wonky, works most of the time.

Experimenting with OpenGL

- Buffer-Objects: FBO (Frame Buffer Object) or UBO (Uniform Buffer Object)
- Blending: Use hardware blending (glBlend*) somewhere in your code. Make sure it can be toggled on/off
- Mip Mapping (on/off)
- Textur-Sampling-Quality (Bi/Trilinear Filtering)

Blending: Available, used with FreeType.



One can only see text when glBlend is activated(Standard, F9 turn off). If turned off, you only see the according quads.

MipMapping & Texture sampling: Mipmaps are generated.

- F4 - Textur-Sampling-Quality: Nearest Neighbor/Bilinear
- F5 - Mip Mapping-Quality: Off/Nearest Neighbor/Linear

Mipmapping and TextureSampling is now working completely.

Frame Time, Wireframe and Help is working.

Features of the game

The horror experience got removed due to it being too dark and us wanting to evade possible problems. Thus there is no roof anymore, but at least we still have our zombie! It turned into a puzzle-platformer. Find the keys to exit the zombie-room and jump to the cube!

First-Person Camera

(Mostly from Submission1)

To implement this, we basically followed this tutorial: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-6-keyboard-and-mouse/.

Thus, we had a camera where we could easily fly through the room. Since we are supposed to have an FPS-like game where jumping is important, flying around wouldn't make it. Therefore, we added a movement-vector that basically is the direction vector without the y-component.

```
glm::vec3 movement(cos(angle_vertical) * sin(angle_horizontal), 0, cos(angle_vertical) * cos(angle_horizontal));
```

Following that, the only needed change to walk and stay on that plane was using the movement vector instead of the direction vector when the up or down button is pressed.

```
if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS){
            player_position += movement * time_delta * speed;
      }
```

Following this, we needed the option to jump in the game. Our implemented option is based on this forum post: http://www.gamedev.net/topic/324744-jumping/#entry3094973. We simply had to adjust it in some ways. The jumping itself was enabled by using the sinus of the velocity/30, where 30 is the start position. The velocity is slowly falling in value through gravity until it gets <= -30, ending the jump.

```
glm::vec3 jump(0, sin(velocity/30), 0);
```

To enable the jumping itself, every frame the "keystate" of the Space key is being saved. If in the current frame the space key is pressed, but was saved as released in the frame before, it means the player wants to jump. The jumping itself and following operations are only done if there is no current jumping happening.

```
if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS){
            if (keystate == GLFW_RELEASE){
                  if (jumping == 0){
                        jumping = 1;
```

Thus, when the jumping itself is enabled, the vector gets added to the player position:

```
player_position += jump * time_delta * speed;
```

Leftclicking

The player can now pick up items. We achieve this by sending out a ray and try to intersect; The first object to get hit was clicked, if the intersect-distance was short enough. We are sending out a constant ray and checking for the zombie. If the intersect-distance is short enough, the zombies modelmatrix basically came too near to the ray-origin(player) and it's a "collision". The checks happen in RayObbIntersection.cpp and it's called in main.cpp during the computation of user inputs. The calls of the function basically are the same every time, the only difference being the model matrix of the object we give with it for intersecting.

Shadow Mapping + Omni-Directional

See "Which Effects are implemented"

Environment Mapping

See "Which Effects are implemented"

GPU particle system

See "Which Effects are implemented"

CollisionDetection (AABB)

See "Other Features"

How and which objects were illuminated (description of light sources) or textured.

Our scenes are illuminated by a single white light, which uses a perspective projection matrix instead of an orthogonal one. This can be seen by the shadow of the monster in the first level changing its shape based on its position to the light. The light also incorporates an attenuation variable, which controls just how strong the light should be. This value can be changed by the user in order to make the scene brighter or darker, based on preference and of course, need.

We also have a PointLight after the door near three keys which are illuminated by it. Its in in the middle and is supposed to light up two keys, leaving one out intentionally (no shadow) to make clear that it's the one to be picked up.

What additional libraries (e.g. for collision, object-loader, sound, …) were used, including references (URL) (see restrictions)?

We used FreeImage for loading the textures. (http://freeimage.sourceforge.net/)
FreeType is being used for the text. https://www.freetype.org/download.html (Version 2.8)
GLFW+GLM for handling of keys, window & math-operations. http://glew.sourceforge.net/
http://glm.g-truc.net/0.9.8/index.html

Which Effects are implemented

Shadow Mapping + Omni-Directional

Normal Shadow Mapping was implemented via the classical way, with two passes over the scene. During the first, a depth map is generated by rendering the entire scene from the light's point of view. The generated texture contains the distances from the light's position to objects in our scene. To do this we used both orthogonal and perspective projection matrixes to render our scene. We chose to use perspective projection matrixes because it is more similar to an actual, real-life light source than an orthogonal one. To store this depth map we used a framebuffer object.

Thus we can determine whether an object is in shadow or not during the second pass, where we render the scene proper. By calculating a point as seen from the light's point of view, and sampling the respective location in the depth map, we can determine whether or not a point is seen by the light source or not. If it can not be seen, it is in shadow, and as such we color it black. Normal rendering is handled with a normal Blinn-Phong shader. Initial results were promising, but presented artifacts such as shadow acne and Peter Panning. To solve these issues we used a shadow bias value and we enabled Front Face Culling. Blocky shadows also made us use PCF to render smoother ones. To render Shadow Maps we used a variety of tutorials such as the ever useful https://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping and http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/.

Point Shadows were a bit more complicated, as we had to substitute the depth map for a depth cube. This requires the scene to be rendered from the cube's point of view 6 times instead of just one. To do this in one single render pass, we used a geometry shader. The geometry shader takes a triangle as an input and generates 6 more triangles, for a grand total of 18 vertices. Triangles are calculated by transforming each world space vertex to light space. The second pass works very similar to normal shadow mapping, instead using a cube to sample distances and determining whether a point is in shadow or not. To do this we used the very detailed tutorial at https://learnopengl.com/#!Advanced-Lighting/Shadows/Point-Shadows, which was a big help, especially during the debugging stage. We learned that we could render the depth map directly to the screen and see if it was generated properly, and that helped us understand the entire process better and solve any issues.

F11 turns on/off PCF-Sampling.

Environment Mapping

There is a cube in "Level 2" that has to be reached to win the game. The Cube has environment mapping.

Environment Mapping was implemented according to two tutorials, https://learnopengl.com/#!Advanced-OpenGL/Cubemaps and http://antongerdelan.net/opengl/cubemaps.html. We only implemented reflection, which we did by calculating a reflection vector around the surface normal based on the view direction of the camera. We use this Reflection Vector to sample the point in the cubemap and render it on the face the camera is looking at, thus giving the impression of a mirror. Initial results were rather confusing, as we did not realize the upside-down image rendered was in fact a correct reflection of our skybox.

GPU particle system

Our game also implements Particle Effects on the GPU side with the use of Transform Feedback. Tutorials from http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html and

http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=26 helped us on our way. This method involves using two Transform Feedback buffers to draw into and read from, respectively. The idea of any particle system is to render a large number of objects, each with their own properties, such as position, direction, color etc. Taking one primitive and generating more is exactly the type of job the geometry shader is used for. Our particles are defined by four attributes: its type (0 for launcher, 1 for particle or shell), position, velocity and lifetime (or age). The vertex shader initially takes these four attributes and sends them to the geometry shader. There, the attributes are collected and ready to be sent further down the pipeline after some checks. If the primitive in question is of type 0, i.e. a launcher, it is automatically sent further. The geometry shader also generates a set number of new primitives (default is 10), assigning each one a new position, random direction (also calculated in the geometry shader), and age. If the primitive is not a launcher, then its age is checked against its given (unifrom) lifetime. If the particle has exceeded its lifetime, then it is discared. If not, it is sent further.

The actual rendering of the particles is done via a technique called billboarding. Billboards are very similar to the billboards one sees when driving on the highway, in that they are quads that always face the looker. By using a geometry shader, we take a list of points and generate triangle strips from them, which we then use to build quads. To have the quads face the user, we calculate the cross product between the vector from the point to the camera and the up vector. The result is the “right” vector, a vector that points towards the direction in which we need to grow our quad.

While the theory behind the implementation is easy to grasp, problems arose when trying to actually write the code. One error that wasted quite a lot of time was a vague access violation error, that we eventually managed to track back to a buggy Intel driver. Apparently, Intel drivers for openGL are notoriously lackluster, and cause problems when trying to process our Transformation Feedbacks. We both did our coding on our laptops, which, like most modern laptops, have an integrated graphics card, as well as a discrete one. Forcing Visual Studio to use the discrete graphics card seems to be the fix for this, but our problems continued with many other errors, which we want and need to iron out.

NOTE

Now working!!!!

How you've implemented those Effects (Links/References to papers, books or other resources where the effect is described and a description of your extensions to it)

See Effect-Implementation

Other special Features in your Game

When you pick up a key(Except one of the 3 at the end, which we put in in a hurry), we write “Key” in the top right corner. Thanks, FreeType! We followed this tutorial: https://learnopengl.com/#!In-Practice/Text-Rendering

CollisionDetection (AABB)

Very good working AABB-Collision Detection. Its implementation is mostly based off different online-resources in forums. When loading the objects and translating it, we also save the exact same values in the object to have them stored. These are later used in the Collision Detection, where a playerbox(defined as 0.2f length each) is used for the player-position as cube.

Our checkCollisions method checks the positions of the player-argument and goes through all our designated objects. Not exactly efficient, but hey. To check for this collisions, we are calling:

```
bool CheckCollisionPlayer(glm::vec3 player, std::unique_ptr<VertexBufferObject>& pBox)
```

In there we create a playerbox in a relative simple way. Remember, the playerbox value is 0.2f;

```
//Arbitrary playerbox
	min.x = player.x - playerbox;
```

The corresponding Bounding-Box is then translated by the saved translate values to get the correct position. CollisionValues is an artifact we simply didn't remove and its values are just zero.

```
float lowerleftx = pBox->lowerLeftFront.x + pBox->collisionValues.x + pBox-
>translateValues.x;
```

We then simply return the check of overlapping boundaries: `return(max.x > lowerleftx && …)`

An older problem was that we did not check if a player landed on a surface while jumping. We now solved this by checking for all our "landable" surfaces while the player is jumping. If the player collides with one, the jumping marker is set to allow jumping again and every movement is stopped. Velocity is set to -30.f to make the player fall as soon as leaves the "collidable" floor/platform.

For clicking on objects and getting hit by the zombie we are not using Collison Detection but RayOBBIntersection. We send out a ray and if it hits the objects box, depending on the range, we accept and do our stuff. See "Leftclicking" under "First-Person Camera".

We are using a skybox in our scene.

We have our own ObjectLoader for OBJ-Files.

What Tools have you used to create the Models (Maya, 3DS MAX, …)

We used Blender to create the objects. Every single object needed to have its UVs unwrapped in edit mode. We added the texture for the most parts directly and in the resulting MTL we changed the texture path. When exporting as OBJ we always had to check the following boxes: Apply Modifiers, Include Edges, Write Normals, Include UVs, Triangulate Faces(Activate!), Objects as OBJ Objects.

For complex interaction sequences (which could already be something like opening a door in the game for example) please also include a step-by-step instruction on how to get through the game.

To instantly get to level 2, press "F10". It opens up the big door immediately without needing to do the puzzle.

The following works as long as you don't get hit by the zombie:

Level1: At the end of the zombie corridor to the left, there is a key right to the alien-poster. Leftclick on the key to get it and have the text "key" in the top right corner. Next search the other alien-poster which is next to the zombie-corridor. Click on the textured metal-bar-door while having the key to have it rotate. After it has finished rotating you can take a key that's stuck in the wall. Leftclick on the key to get it once again. The "key" text disappeared when you clicked on the door, but reappeared when you clicked on the key. Now you can open the big door, but you will notice that there are two

pillars in the way. You need to jump over them(spacebar) and with a leftclick on the door it starts moving. Now you have reached Level2!

Level2: Before jumping on the platforms, turn right and get the key without a shadow. Then simply jump on the platforms (Spacebar) until you reach the cube. Jump into the cube for a “Game Won”-Text! Look into the cube though, the coordinates aren’t perfected.