PENUMBRA
TSBK07 Project Report

Andreas Sahlin

1 INTRODUCTION

In this project I tasked myself with creating a 3D, 1st person, portal-jumping, enemy-fighting game in a mysterious castle world. Textures were to be made in Gimp (normal maps included), Simple polygon meshes for walls and floors etc. Outside section that required the legendary red skybox that inspired this entire project were also to be included. Multiple light sources, plant billboards were to decorate the scene. The main attractions were planned to be portals, that could be used to traverse as well, while fighting your way out of the hell dimension you were in.

To accomplish this, a number of specific implementation points were set up and that could be either "will do" or "might do".

1.1 Will do

- 3D-terrain from heightmap
- Skybox
- Simple inside environments - floors, walls, ceilings, doors
- 1st person perspective and controls
- Collision between player and world
  - Player collision with sphere against walls, and enemies as spheres too
- Lights in the scene - placed with diffuse and specular on different objects
  - 10 - 20 lights per level
- Portals - to move between rooms, risk-free
- Simple system for enemies and attack/health
- Billboards for plants
- Optimisation: Split world into cells based on x,y,z axes, only draw current/adjacent cells

1.2 Might do

- Shadows: objects that occlude light and cast shadows
- Simple level system: fade to black, switch current level with the next
  - Save data in a `.txt` or `.json` file
- Particle effects:
  - fire, smoke, portal effects
- Portals - creating non-euclidean worlds
- Optimisation: Frustum culling
- Sound effects: footsteps, enemies, attacks, ambient/music
- UI-elements: health, enemy health etc.

2 IMPLEMENTATION

A multitude of tools were used during the implementation of the game. The game was developed in C++ using the VS Code editor for game coding, shader coding and obj mesh editing. For the custom made textures like the brick texture, Gimp was used to make them from scratch, including normal map generation. The meshes and textures used for objects come from Zsky on Patreon [5].

The structure of the game code consists of the main file `penumbra.cpp` where the game is ran from, it also contains most global variables, lists and the main render loop, consisting of the heightmap phase, the picking phase and the recursive portal render phase. The implementation of portals is based on Thomas Rinsma's method [4]. The rest of the code is written in an object-oriented fashion and all contain a main class of the same name as the file. These files are:

- `actor.cpp`
- `billboard.cpp`
- `camera.cpp`
- `clickable.cpp`
- `gameObject.cpp`
- `heightmap.cpp`
- `light.cpp`
- `player.cpp`
- `portal.cpp`
- `skybox.cpp`
- `terrain.cpp`
- `texture.cpp`

Each corresponding `.h` file are in a `/headers` folder, all vertex, fragment, and the single geometry shader files are in a `/shaders` folder, and all game assets such as textures and meshes are in an `/assets` folder.

The `actor` class is meant to be an interface to be implemented by enemies and the player. It contains basic common info such as health and a method to call to apply damage and a method to heal the actor.

The `billboard` class is used to represent many randomly placed billboards in the world, it contains a method for generating billboard positions and a method for drawing them using the `glDrawArrays` call. This billboard implementation uses the geometry shader's ability to create new geometry during a draw call and is based on Etay Meiri's method [3] for billboards using the geometry shader.

The `camera` struct is a very simple struct currently only holding the player's view and projection matrices.

The `clickable` class is also meant as a catch-all class representing all clickable objects in the picking phase, it contains a reference to a function that is called when the `onCLick` method is called. The `clickable` class itself inherits from the `gameObject` class, as clickable objects can be seen as a game object with extra functionality. The picking implementation is based on Etay Meiri's method [2] for picking, only simplified to not allow per-primitive picking. This is due to how model drawing calls are currently set up.

The `gameObject` class is the main class that represents objects in the game, and more specialised classes are supposed to inherit from such as the `terrain` and `clickable` classes. This class contains the main drawing procedure which loads textures to the object's shader and uses the `DrawModel` call, modified for normal map support.

The `light` class is used to represent light sources in the game, represents both positional and directional light. It holds values for ambient, diffuse, and specular light.

The `player` class represents the player, contains methods for player controls and movement, and contains an instance of the `camera` struct to represent the player's vision.

The `portal` class represents the portals in the game, and contain a transformation matrix that is used to draw the portal in the right place, while the separated position vector and rotation matrix is used for calculations to get the correct portal view and to clip the projection plane to avoid the bug of the virtual camera being blocked. The methods for calculating this are also contained in the class.

The `skybox` class is purely for representing the skybox and is very simple. It does the same things as the `gameObject` class but pared down, since a skybox has no need for multiple textures and needs the depth test and culling disabled while drawing.

The `terrain` class derives from the `gameObject` class and takes in a heightmap instead of a mesh and has a method for making a model object out of the heightmap as well as calculating tangents and bitangents for proper normal mapping.

The `texture` class represents a texture in several components, and must contain a diffuse map (the texture itself) and optionally a normal map, and specular map. The `texture` class also has a method to call to properly load the texture to the shader being passed as a parameter, it will pass all the optional maps as well if they exist.

3 PROBLEMS AND BUGS

There were a lot of problems that occurred during development, some still unsolved to this very day. Many minor bugs were fixed and are not noteworthy but the following were or are some of the biggest bugs encountered during development.

3.1 Reading pixels from FBO

One particularly pernicious bug encountered during the development of Penumbra was the interaction between an FBO and the `glReadPixels` function. Etay Meiri's [2] implementation makes use of an FBO and the `glReadPixels` function, yet shows no sign of bugs. What causes this bug is therefore unknown. What is known is that several days were wasted trying to correct it before consulting with the course professor led to the current implementation reading pixels straight from the screenbuffer instead of an FBO. This current implementation was achieved in a mere couple of hours. Since the cause of the bug was never discovered and no fix for the FBO implemented, the bug is not considered to be solved. Solving a bug is like fixing a hole in the road. What the current solution does is essentially equivalent to avoiding the problem by swerving off the road around the hole.

3.2 Heightmap navigation bugs

The way the heightmap method works causes issues when faced with sharp changes in elevation, such as when facing a wall. If the player zig-zags when stuck against a wall or other elevation difference unexpected things may happen, such as the player climbing said wall and permanently changing the player camera's elevation

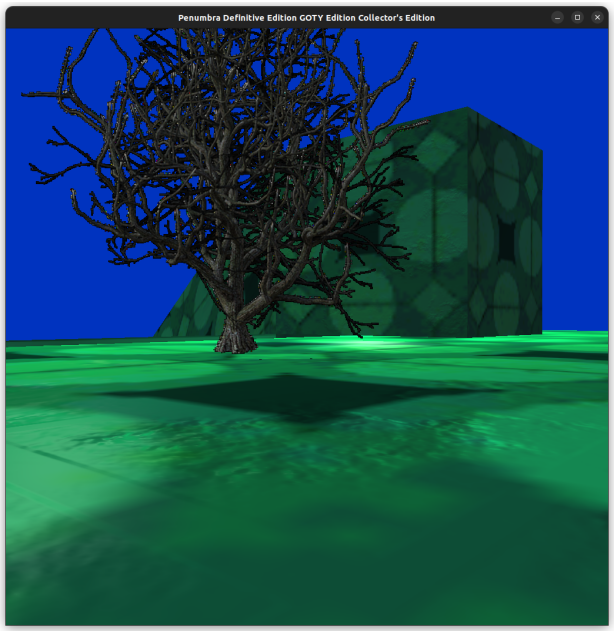

Figure 1: Issues with heightmap method can cause dwarfism.

off the heightmap causing the player to appear giant. The reverse is also possible where the player camera shrinks towards the ground and gives the appearance of a tiny player as seen in Figure 1. The cause of this is not yet fully known, but is most likely related to the step size used during calculations between where the player currently is on the heightmap and where the player's forward vector is pointing.

Another possibility is that the zig-zagging motion itself tricks the current implementation to think that the sharp elevation difference is smaller than the threshold set to stop the player from almost teleporting over walls.

3.3 Skybox overriding portals

One issue that still plagues the portal implementation is the skybox bug. Due to the fact that the skybox needs to have back face culling and depth testing turned off to be rendered correctly, it interferes with the current portal implementation as seen in Figure 2. If the skybox is enabled, the view inside the portals is rendered correctly, but covered up by the lower portion of the skybox. The full problem and thus cause of the bug is not known, as the nature of the recursive stencil stack obfuscates things. One theory to what the cause is that it could be that the same cause as for the next portal bug.

3.4 Possible portal bug

It is a hard to spot bug, but from some angles, some portals seem to still suffer from the blocked virtual camera bug supposedly fixed by the clipping of the projection matrix. This suggests that the implementation isn't correct, which is extremely likely. The bug stems from how the portal system works. Since a virtual camera view is created and is placed behind the portal on the other side of the portal the player is looking into, if an object is in between the

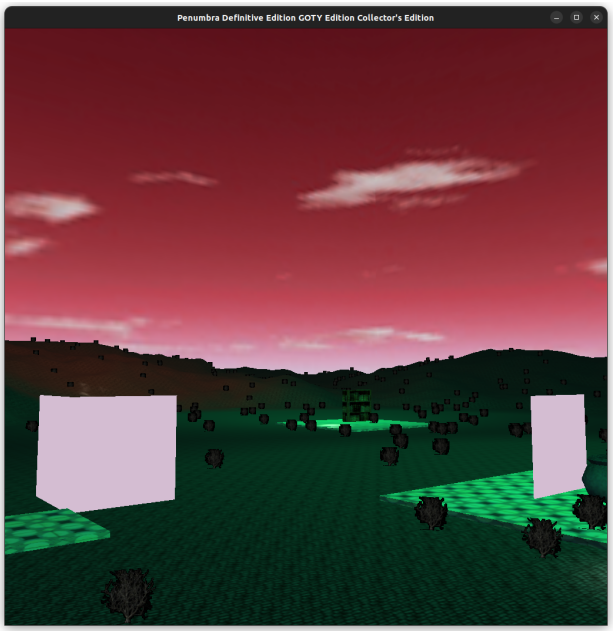

Figure 2: Portal-rendering issue when skybox is enabled.

back of the portal and the virtual camera view, it obstructs the view of the portal. Suspect number one for causing this bug is using the wrong value for the `dist` variable in the clipping method.

4 CONCLUSIONS

All in all, the project is both a failure and a success. On one hand it is a failure because a playable game was not achieved. On the other hand it was a success because a lot was still achieved and implemented and a lot of new lessons were learned.

With all the experience I have now, there are a lot of things I would like to change when it comes to the implementation. The main structure of the game and how classes are set up was decided at the start, and mostly made up as the development went along. It is hard to know what will be a good class structure before you have anything implemented, and now I see that my structure isn't the best. For example, the current state of the project ended up having many different classes that all have draw methods implemented in different ways. I initially wanted the `gameObject` class to be the only class with a draw function for every need, but this quickly turned out to not be realistic. Instead, a `Drawable` interface should be implemented that each class can implement that dictates that each class provide their own specific drawing method implementation. This would allow for more diverse and specified classes that all can still reside in a single list to be looped through for drawing by making a `vector<Drawable>` variable.

Something else regrettable is that the tangent and bitangent calculations are currently stuck inside the terrain class when they should be separated out and made available to all classes that represent objects that might want sensible normal maps applied. There might be more such unlucky functions needlessly stuck in classes when they should be more general. Overall major refactoring is something I want to do to the code base, which will undoubtedly produce new classes and interfaces more in line with how the actor and player classes are set up to work. Currently not even the actor class is a true interface, but this would also be improved in a code refactoring.

There are also some general implementation that could just be improved in general, such as player movement. The heightmap method was conceived and implemented mainly due to time constraints and not because I ever considered it the superior choice. Bugs like the one discussed in section 3.2 have shown that the method is too limited for what Penumbra is meant to be. Instead, time to properly study and implement collision detection and physics based motion would be a great improvement to the game as is. Simplified bounding boxes is all that is needed to provide a vastly more realistic game experience.

The second benefit to swapping out the heightmap method for collision detection-based player movement that it allows for fully functioning portals that the player can walk through. The current method provides no help for smooth teleportation between portals that would look seamless in-game. With collision detection, knowing where the contact point is between player and portal allows for seamless teleportation, with the player coming out at the other side on correct relation to the other portal.

I have also read about further improvements that could be made to the portal system, such as allowing objects to pass through and get teleported seamlessly. As explained by Ali Kömürcü [1], objects can be seamlessly transported through portals by using the geometry shader to cut faces in half and make new primitives on each side. A lot of points in the specification were not achieved and would greatly improve the game if implemented into the game, such as a level system, enemies, and a story too come to think of it. These will have to be added in later as Penumbra goes from being a course project to a passion project.

To conclude, this experience has been extremely fun and rewarding, and I'm still proud of what was achieved, considering how many outside factors took away time from this project, and Penumbra will only continue to improve and get better.

REFERENCES

[1] Ali Kömürcü. 2023. *Portals with OpenGL.* Medium. https://medium.com/@alikomurcu/portals-with-opengl-d74da6241dd4
[2] Etay Meiri. 2022. *Selecting 3D Objects With The Mouse Using OpenGL // Intermediate OpenGL Series.* Youtube. https://youtu.be/71G-PVpaVk8?si=BxJkP5OUH2t0-ZCA
[3] Etay Meiri. 2023. *Billboarding With The Geometry Shader // Intermediate OpenGL Series.* Youtube. https://youtu.be/nvCEpMUxAHE?si=G1hnMjIM2xJ1ufOv
[4] Thomas Rinsma. 2013. *Rendering recursive portals with OpenGL.* https://th0mas.nl/2013/05/19/rendering-recursive-portals-with-opengl/
[5] Zsky. 2021. *creating 3D Lowpoly Assets and Tutorials.* Patreon. https://www.patreon.com/Zsky