Cubemap Mirrors

Anton Nilsson & Morgan Nordberg

Introduction

The project goal was to create a 3D scene, where the "player" can walk around as a 3D model in a first-person-perspective; aswell as to model mirrors that reflect the models in the world. Extra goals were to gradually improve the mirror implementation with various, gradually more advanced, techniques. Yet another additional goal was to make it possible to have mutiple mirrors that can all see each other.

Background information

Any information about the kind of problem you solved that is needed to follow the rest of the report.

Cubemap texture

A cubemap consists of 6 textures that make up the 6 sides of the cube, and there is built-in support for working with cubemap textures in various contexts directly in openGL.

FrameBufferObject

A frameBufferObject (FBO) is a datastructure which allows us to render the scene to a texture, ahead of actually rendering to the screen. It has it's own texture, various buffers, and depth buffer. It's useful for capturing the scene from various persepcives before the final render.

Implementation details

Here we will go into more detail on how various techniques were implemented aswell as overall project structure.

Project structure

the project structure contains all source code in src/main.cpp, all 3D models in models/, tetures in textures/ and shader code in various shader files in the shaders/ directory. The various dependencies are contained whitin the common/ directory.

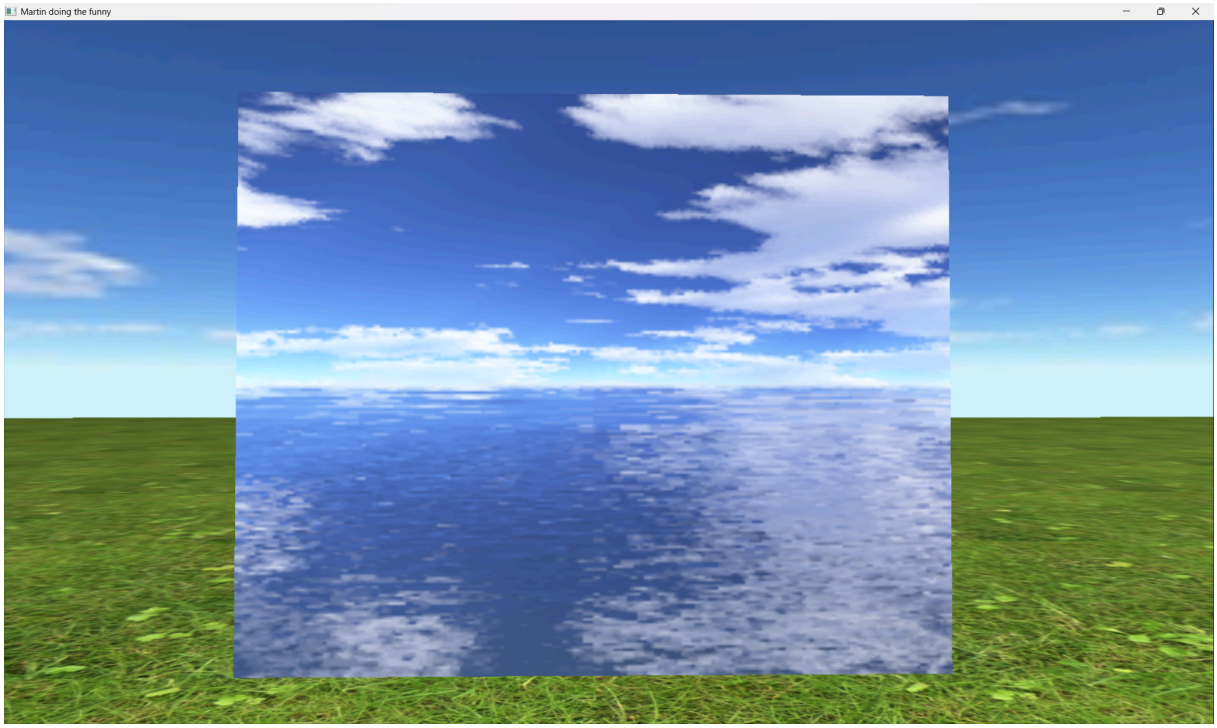
dependencies

We used no additional dependencies besides those provided for the lab assignments. We did howerver use openGL 4.6 in our shaders.

basic cubemap

In openGL there is a built-in type for dealing with cubemap textures in a shader called "samplerCube". So in the shader you pass a uniform samplerCube, and a directional vector from the observers position to the mirrors surface, and then reflect the vector w.r.t the normal vector of the surface. The resulting vector is used for sampling color from the cubemap texture.

The problem with approach is that the cubemap can't account for changes in the enviorment and thus models in the scene wont be reflected. There is also issues with scale/perspective since conceptially one can think of a cubemap as an approximation of an infinitley large cube, so in a small scene you will get very apparent distortions in perspective in the mirror. The first problem is solved by using dynamic cubemaps, and the latter problem is solved by parralax correction. Both of this techniques are covered later in the report.

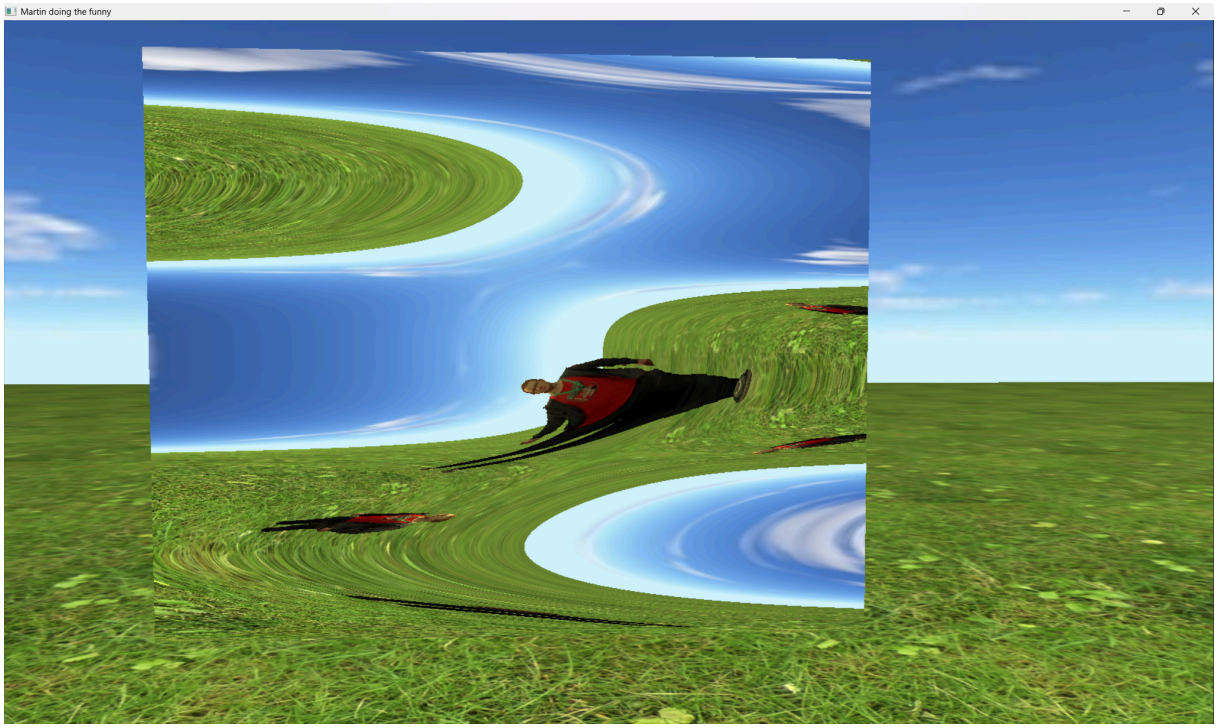


Dynamic cubemap

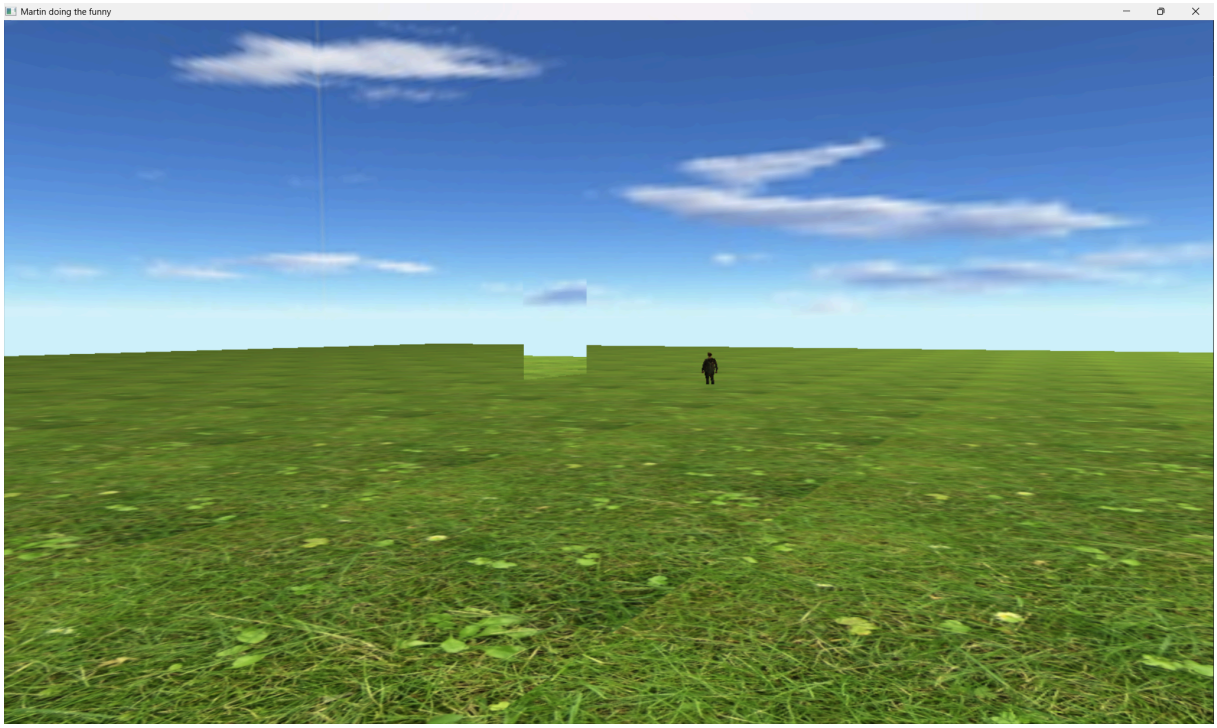
Dynamic cubemaps work by giving each mirror it's own FBO, that contains a cubemap texture, and then letting the mirror render the scene in all directions from it's own perspective and writing the result to it's associated FBO:s cubemap texture. This texture is then used for sampling when rays are reflected of the mirrors surface as desciped in the basic cubemap chapter. This does not solve for the issue of distorted perspective, but this does mean that various models and changes in the scene will be visible in the mirrors reflections.

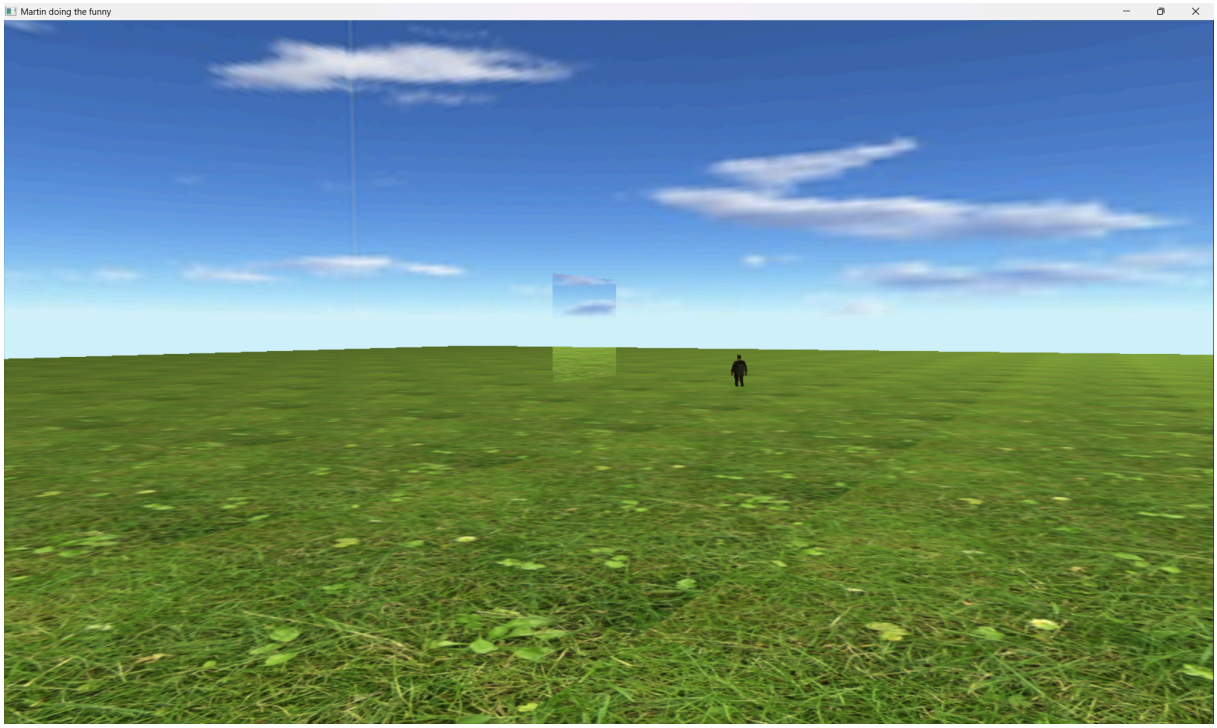
basic bumpmap

By using simple sinus functions with surface position data as input we can distort the normal vectors on the surface of the mirror to create fun affects like those found in “fun houses” at amusement parks. The limitation of this technique is that if the bumps in the mirror would create internal bounces, this will not happen since the mirror has no actual geometry besides being a flat plane as far as the reflections are concerned.



parallax correction





recursive mirrors

To solve the issue of multiple mirrors, using dynamic cubemaps, being able to see each other we used a recursive method. The idea is that each mirror has 2 FBO:s. And on alternating frames they will switch between which they read from, and write to. So one mirror will see the last frames version of the other mirror. This does cause a slight lag in the mirror reflection when reaction to movement since there will take several frames for the changes to propagate through the mirrors seeing each other.

Interesting problems

Conclusions

Overall, pretty much all of the techniques had various drawbacks, and were quite tricky to implement correctly. So it's possible that overall, for reflective mirrors/objects raytracing is probably a more approriate technique, since it has few or none of said drawbacks. Unsure what the performence cost comparison would be, but the best version of the scene we managed to achieve ran quite slowly on a decent-ish lapptop since it featured to mirrors, and that meant we had to render the scene 6 times per mirror, and then a final render from the "players" perspective.

There might be cases where these techniques are cheap and work well for flat reflective surfaces like still water, but for accurate mirrors it seems raytracing would be more worthwile.

Source-code

Open github repo: https://github.com/the-JS-hater/MartinFunnyHouse