Cubemap Mirrors

Anton Nilsson antni004@student.liu.se
Morgan Nordberg morno368@student.liu.se

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; as well as to model mirrors that reflect the models in the world. Additional goals were to gradually improve the mirror implementation using more and more advanced techniques. Yet another goal was to make it possible to have multiple mirrors that can all see each other.

Background Information

Some basic datastructures needed for the overall discussion are explained here.

Cubemap Texture

A cubemap consists of 6 textures that make up the 6 sides of the cube. Built-in support for working with cubemap textures exists directly in OpenGL and GLSL. The cubemap texture can for example be sampled using a 3D vector.

FrameBufferObject

A FrameBufferObject (FBO) is an object in OpenGL which allows us to render the scene to a texture, instead of rendering it to the screen. It has its own texture and various buffers, such as a depth buffer. In our case it's useful for capturing the scene from various perspectives and positions before rendering to the screen.

Implementation Details

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

Project Structure

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

Dependencies

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

Pre-rendered Cubemap

In OpenGL there is a built-in type for dealing with cubemap textures in a shader called "samplerCube". In the shader you pass a uniform samplerCube which is sampled using a vector. This vector is found by reflecting the directional vector from the observer to the mirror's surface w.r.t the normal vector of the surface. The result of using a pre-rendered cubemap can be seen in Figure 1.

The problem with approach is that the cubemap can't account for changes in the enviorment and thus models in the scene won't be reflected. There is also issues with scale/perspective because the sample vector is only a good approximation if the enviorment is an infinitley large cube. In a small scene this will cause very apparent distortions in perspective in the mirror. The first problem is solved by using dynamic cubemaps, and the latter problem is solved by parallax correction. Both of these techniques are covered in later sections of the report.

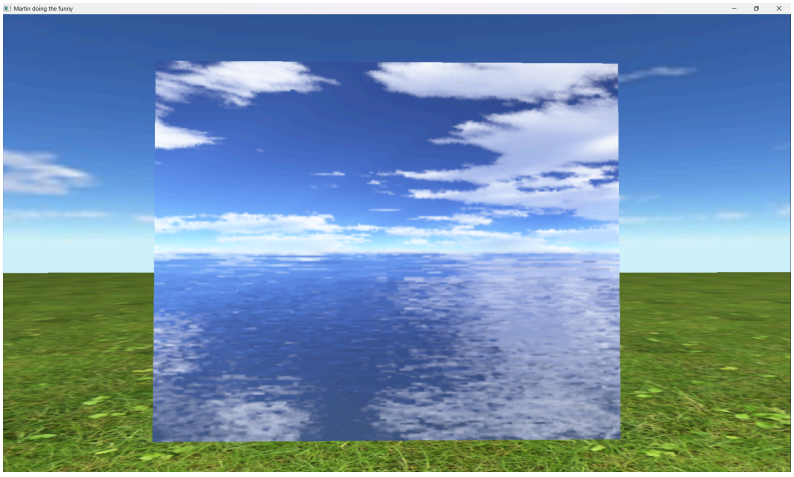

Figure 1: Pre-rendered cubemap.

Dynamic Cubemap

Dynamic cubemaps work by giving each mirror it's own FBO, that contains a cubemap texture. The scene can then be rendered from the mirrors position, in the positive and negative x, y and z directions. The result of these renders is saved in the cubemap texture of the mirror's associated FBO. This texture is then used for sampling when rays are reflected on the mirrors surface as described in the pre-rendered cubemap section. This makes the various models and changes in the scene visible in the mirrors reflections. Figure 2 shows the scene being reflected in the mirror using a dynamic cubemap.

Figure 2: Dynamic cubemap.

Basic Bumpmap

Using the sinus function on the surface position can distort the normal vectors to create fun affects like those found in "fun houses" at amusement parks. This effect is shown in Figure 3. A problem with this technique is that bumps in a real mirror could cause the mirror to reflect itself. Because of limitations with how the dynamic cubemaps are created, this effect can't be replicated. The mirror will also never be able to reflect itself since the actual mirror model is flat.

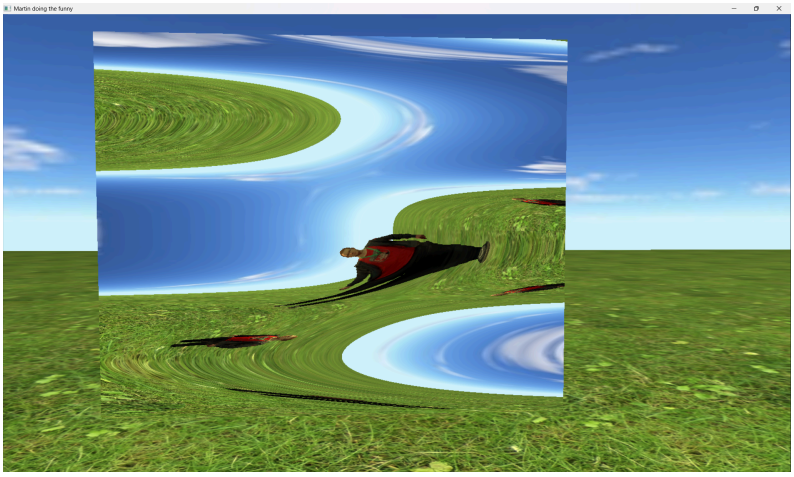

Figure 3: Normal vectors distorted.

Parallax Correction

As said using cubemaps for reflections comes with the drawback that the reflection can become very distorted as they are only approximations. This distortion is a parallax issue which is a result of sampling using the reflection vector. The cause is that the position of the cubemap is not the same as the position of the reflection point.

The solution to this parallax issue is using parallax correction to get the vector which samples the cubemap. The difference this makes on the reflections is showcased in Figure 4. The idea is to define an axis aligned bounding box with dimensions slightly larger than the size of the room/scene. The sampling vector can then be found by finding the intersection point between reflection vector from the reflection point and the AABB.

Define $p_0 = (x_0, y_0, z_0)$ as the reflection point, $p = (x, y, z)$ as the intersection point with the AABB and $d = (x_d, y_d, z_d)$ as the reflection direction/vector. Also define $b_{\min} = (x_{\min}, y_{\min}, z_{\min})$ as the corner of AABB with the smallest x, y and z components and also define $b_{\max} = (x_{\max}, y_{\max}, z_{\max})$ as the corner with the largest components. The idea to find the intersection point with the AABB is to handle each axis separately. Calculating if the ray $p_0 + dt$ intersects with the $x_{\min}$ or $x_{\max}$ plane and the distance to that plane can be calculated as follows:

$$
\begin{aligned}
x_0 + x_d t_{x_{\min}} = x_{\min} \Rightarrow t_{x_{\min}} &= \frac{x_{\min} - x_r}{x_d} \\
t_{x_{\max}} &= \frac{x_{\max} - x_r}{x_d} \\
t_x &= \max\left\{t_{x_{\min}}, t_{x_{\max}}\right\}
\end{aligned}
$$

The scalars $t_{x_{\min}}$ and $t_{x_{\max}}$ are the scalars of d used to get $x_{\min}$ and $x_{\max}$, respectivly. The positive of the two scalars will be the scalar t_x giving the intersection point of the AABB in the x-axis, that is the scalar with the same x direction as x_d. Similarly this can be done to find t_y and t_z. The intersection point with the AABB is then given by the minimum of these three (since it will give the first plane intersection) and solving for p:

$$
\begin{aligned}
t &= \min\left\{t_x, t_y, t_z\right\} \\
p &= p_0 + dt
\end{aligned}
$$

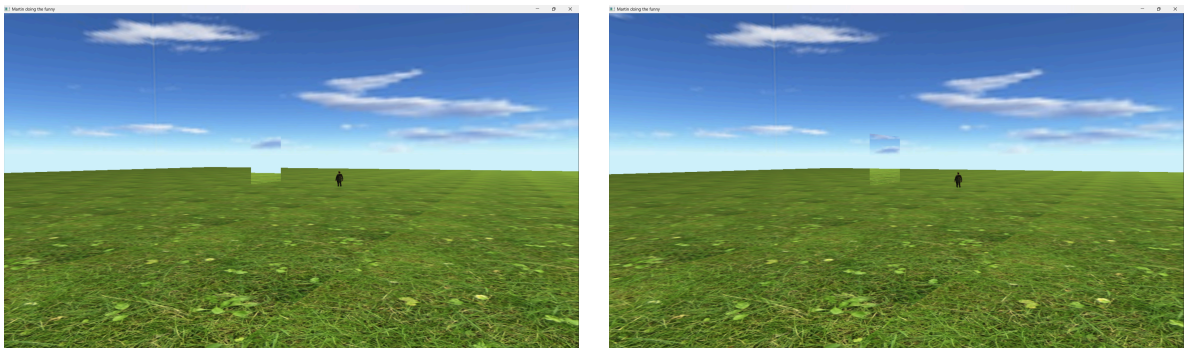

Figure 4: Parallax corrected cubemap on the left; normal cubemap on the right.

Recursive Mirrors

If there is multiple mirrors in a scene they won't be able to reflect eachother correctly when using only one FBO for each mirror. Since one mirror will render its cubemap texture before the other there will be visible artifacts. One idea to solve this is to use two FBO:s per mirror, which are switched between each frame. This will solve the problem by using the cubemap texture from the previous frame when rendering the cubemap texture for the current frame. It will cause each recursive reflection step to lag one frame behind, but after a while the changes will propagate through the reflections and look correct.

These recursive mirrors also makes the issue of not parallax correcting the sample vector apparent. The other mirror will appear larger than it should, which causes an unwated effect. When the sample vector is corrected the resulting reflection looks more like expected as seen in Figure 5.

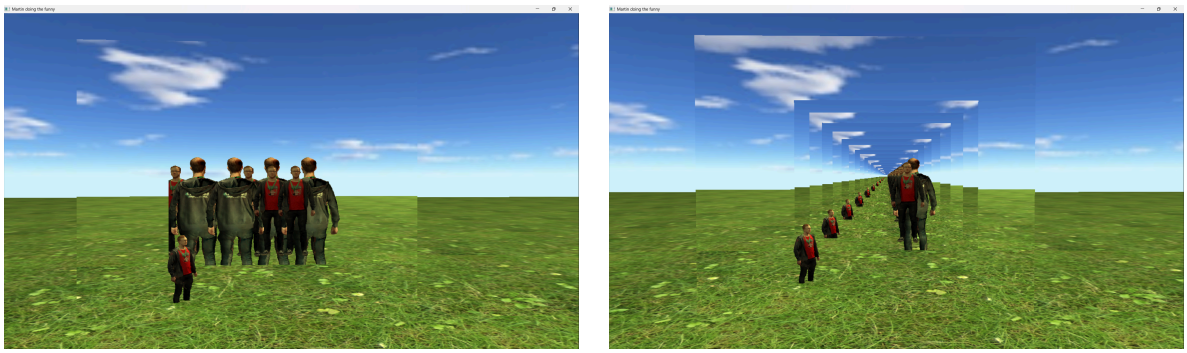

Figure 5: Comparison of recursive mirror with and whitout parallax correction

Problems

Most of the problems encountered such as the cubemap parallax error has been explained before. Outside of these intrinsic issues with the solutions we chose the most common problems encountered were us misunderstanding how OpenGL works. One such is example is that we forgot to change the framebuffer we used before trying to render to the texture.

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.

It also generally doesn't work too well for multiple reflective surfaces reflecting each other recursivley, atleast not in our implementation. So for scenes with multiple reflective surfaces raytracing is probably prefreable.

Source Code

The code for the project is available in an open github repo: https://github.com/the-JS-hater/MartinFunnyHouse