Bachelor project

Environment Mapping

A practical investigation of common environment mapping techniques.

August Heining Holst

Knud Henriksen

Handed in: January 15, 2022

Contents

1 Abstract

This report will take the reader through four different environment mapping algorithms, showing the underlying mathematics and implementing them in C++ using OpenGL as the graphics API. The prorgram will be supplied along with the report, so the user can try and run it for themselves.
I also intend to show how they perform in comparison to each other, however I did not reach anything conclusive in regards to the running times.

2 Introduction

In the field of computer graphics, reflective or refractive surfaces are often rendered using environment mapping, due to being less computationally expensive than other reflection rendering approaches. Environment mapping is interchangeable with reflection mapping.
An environment map is a texture of the scene surrounding a given point, the given point is often the centre of the reflective or refractive object.

3 Theory

3.1 Texture mapping

3.2 Reflection vectors

A lot of rendering techniques, as well as all of the following environment mapping techniques work by computing a reflected vector off a surface. The reflected vector is intuitively the vector mirrored from the viewer's position, as seen in
The reflected vector is computed by flipping the incoming view vector around the surface's normal.

$$\mathbf{R} = \mathbf{V} - 2(\mathbf{V} \cdot \mathbf{N})\mathbf{N}$$

The OpenGL API has a built-in function, that takes the $\mathbf{V}$ and $\mathbf{N}$ as arguments and return $\mathbf{R}$[Incb].

3.3 Environment mapping techniques

The different techniques may(unsure) only differ in the way they scale and bias the reflected vector, before using its $\mathbf{R}_x$ and $\mathbf{R}_y$ coordinates to index into the corresponding texture map.

3.3.1 Spherical maps

This type of environment mapping approximates the reflective surface by emulating the environment as a sphere enclosing the model. To obtain the desired effect, the image texture has to depict a mirror ball with the environment orthographically projected onto it. Thus a single image covers the entire environment,

except for the spot right behind the object. Such an image gets more distorted as it approaches that spot.

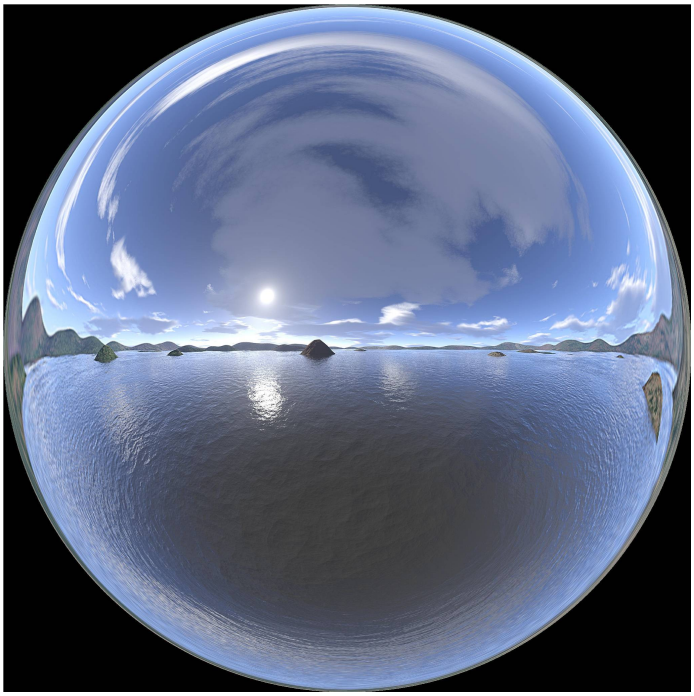

Figure 1: Example of a sphere map texture map

Indexing the sphere map is done with (r_x, r_y), where they're scaled and biased:

$$(\frac{r_x}{2 \cdot \sqrt{\mathbf{R} \cdot \mathbf{R}}}, \frac{r_y}{2 \cdot \sqrt{\mathbf{R} \cdot \mathbf{R}}}), \quad \text{where } \mathbf{R} = \begin{pmatrix} r_x \\ r_y \\ r_z + 1 \end{pmatrix}$$

The derivation of this formula is due to the normal vector's z-coordinate being on the unit sphere. As well as mapping of spherical unit vectors from their range (-1,-1) to (1, 1) onto the sphere map's range $(0, 0)$ to $(1, 1)$[Pau99].

3.3.2 Equirectangular maps

Equirectangular mapping is similar to spherical maps, by emulating the environment as a sphere and using a single image. The image here has to contain an equirectangular projection of the environment, meaning that the circles of longitude and latitude are mapped to vertical and horizontal lines of constant spacing. Equirectangular projections have historically been used to map the earth (from a globe to a 2-dimensional map), as an aside the more common projection for maps is the mercator projection.

Figure 2: Example of an equirectangular texture map

Before indexing the equirectangular map, the y compononent is extracted from $\mathbf{R}$, afterwards we project $\mathbf{R}$ onto the xz-plane. The projected reflection vector is normalized.

$$\mathbf{R}' = \begin{pmatrix} r_x \\ 0 \\ r_z \end{pmatrix}$$

Indexing the equirectangular map, uses a point consisting of a scaled and biased (r'_x, r'_y), which then essentially form the altitude and azimuth angle.

3.3.3 Cubemaps

In cube mapping, the environment is projected onto a cube, such that each face stores a square texture on each of the cardinal axes, in both directions for each. Thus requiring 6 individual textures.
The cubemap is indexed using all three coordinates of the reflection vector (r_x, r_y, r_z) without any modifications. The texture lookup is then following the reflection vector, from the centre of the cube and acquiring the desired fragment color. See figure 3.

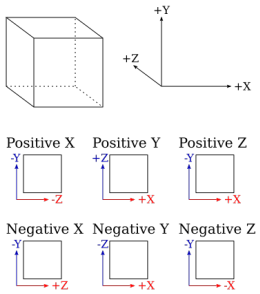

Figure 3: How each face in the cubemap is oriented relative to the defined cube, and how a direction's co-planar axes serves as texture coordinates[Inca].

This technique should be the most intensive of the bunch. Intuitively it uses the most the memory, but it also leads to worse spatial locality for sampling look-ups.

3.3.4 Dual paraboloid maps

In dual paraboloid mapping, a point is enclosed by two parabolic hemispheres. These hemispheres' normals are used calculate the reflection vector, which then is used to sample one of the two hemisphere's textures. The hemispheres cover the front and back of a point, so the two images should show the reflected

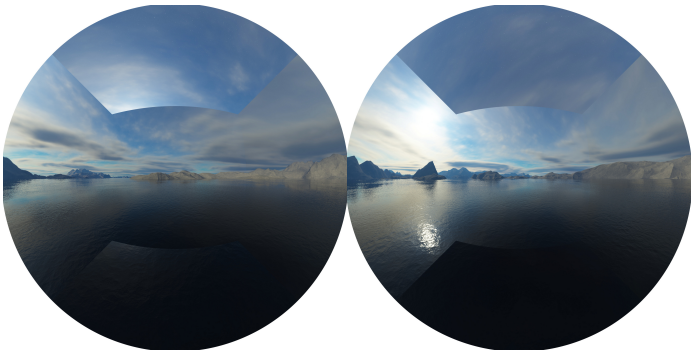

Figure 4: Example of dual paraboloid texture map

environment projected onto an elliptic paraboloid.
The paraboloid to be used has the following definition (see Figure. 5 for plot):

$$\frac{1}{2} - \frac{1}{2}(x^2 + y^2) \quad \text{for } x^2 + y^2 \leq 1$$

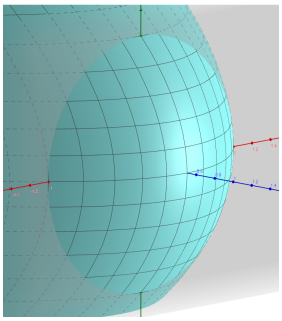

Figure 5: A plot of one hemisphere. The X-axis is red, Y-axis green and Z-axis blue.

Thus a point on the hemisphere's surface can be defined as:

$$P = (x, y, f(x, y))$$

3.4 Skyboxes

Since we are texturing objects to emulate environmental reflections, it would be nice to render the surrounding environment too. This can be done with a cubemap.

3.5 Environment mapping for refractions

4 Implementation

My implementation is written in C++ 11, using OpenGL 3.3 as the API to send 3D vector drawing instructions to the GPU. Meaning that the environment mapping technqiues where written in GLSL.

4.1 Project structure

4.1.1 lib

Subdirectory containing common libraries used in conjunction with OpenGL.

- `glad`: Loads OpenGL functions. I used the webservice to generate the header file[Her].
- `glfw`: Framework for window, context and input management.
- `glm`: OpenGL mathematics library matching the specifications of GLSL.
- `assimp`: Asset importer. Used to load `.obj` files.

4.1.2 headers

This subdirectory contains class and function declarations, as well as definitions. Four of these are a slight modified versions of code seen on `learnopengl.com`.

- `shader.h`: Class for instantiating and using shaders written in GLSL[Vrid].
- `camera.h`: Class for generating a view and projection matrix in the abstraction of a camera. Allows for user input too[Vria].
- `mesh.h`: Class for holding vertex data and drawing the supplied faces[Vrib].
- `model.h`: Class for holding multiple meshes[Vric].
- `stb_image.h`: Functions for loading common image file formats[Bar].

4.1.3 src

Only contains `main.cpp`. Its primary purpose is naturally being the program's entry point. It should be noted I have commented out line 159, since it messes with controlling the camera using the keyboard.

```
glfwSwapInterval(0)
```

This call will disable vertical sync, allowing for the frames per second to go uncapped, instead of matching your monitor's refresh rate.

4.1.4 res

Contains two subdirectories `textures` and `models`. One containing the textures I used, the other one containing `.obj` files of models to be rendered. The primitive models were created by me using Blender, I downloaded the Utah Teapot and Stanford Dragon and scaled them down in Blender.
I couldn't find any dual paraboloid textures for free use in a proper resolution, so I generated one from the cubemap using this program I found `https://github.com/usnistgov/cubemap-stitch`, however I did not seem to work correctly.

4.1.5 shaders

4.2 Building and running the program

I used CMake to generate the Makefiles, I expect the reader to be familiar with CMake if they intend on building the project. Using a command line interface, the instructions are:

1. `cd build`
2. `cmake ..`
3. `make`
4. `./bisc/bisc`

The camera uses the WASD keys, mouse and mouse wheel for controls. The program allows for the user to change model using the J and K keys, as well as the change the shader using H and L.

5 Results

5.1 Renders

Figure 6. to figure 13. Contains renders I generated using my implementation of the 4 techniques. The textures used are the ones shown in the theory section and have sizes:

- Sphere map: 2048x2048 pixels
- Equirectangular map: 4096x2048 pixels
- Cube map: 6 images of 2048x2048 pixels
- Dual paraboloid map: 4096x2048 pixels

Figure 6: Utah Teapot textured using a sphere map

Figure 7: Stanford Dragon textured using a sphere map

Figure 8: Utah Teapot textured using an equirectangular map

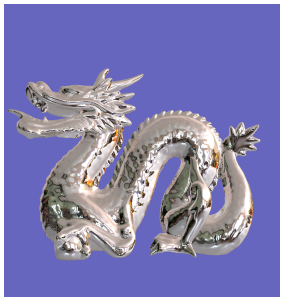

Figure 9: Stanford Dragon textured using an equirectangular map

Figure 10: Utah Teapot textured using a cubemap

Figure 11: Stanford Dragon textured using a cubemap

5.2 Performance

To measure the performance of the different environment mapping techniques, I used C++'s `chrono` library to measure the time it takes to run the code:

```
models[current].Draw(shaders[current_shader]);
```

This code ends up running the `Draw` function in `Mesh.h`, which is:

Figure 12: Utah Teapot textured using a dual paraboloid map.

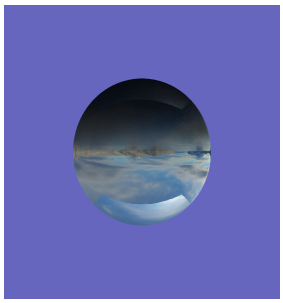

Figure 13: Sphere consisting of 10454400 vertices, textured using a dual paraboloid map

```
glBindVertexArray(VAO);
glDrawElemnts(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
glActiveTexture(TEXTURE0);
```

Where the vertex array object holds the vertices, belonging to one of the model's meshes.
I jotted down the performance in a table (see table 1.). The entries in table is the average time it takes to draw a textured model (in seconds), out of a 1000 frames, for models of increasing vertex count (noted in the top row). These

	960	63488	116160	10454400
Sphere map	$3.1621 \cdot 10^{-}5$	$3.2395 \cdot 10^{-}5$	$3.1953 \cdot 10^{-}5$	$3.0416 \cdot 10^{-}5$
Equirectangular map	$3.2262 \cdot 10^{-}5$	$3.2900 \cdot 10^{-}5$	$3.2785 \cdot 10^{-}5$	$3.1803 \cdot 10^{-}5$
Cubemap	$3.2743 \cdot 10^{-}5$	$3.2545 \cdot 10^{-}5$	$3.3859 \cdot 10^{-}5$	$3.1533 \cdot 10^{-}5$
Dual paraboloid map	$3.2322 \cdot 10^{-}5$	$3.3507 \cdot 10^{-}5$	$3.3822 \cdot 10^{-}5$	$3.2667 \cdot 10^{-}5$

Table 1: Performance/benchmarks for 4 environment mapping shaders.

tests were run on a machine running Linux and Mesa 21.3.3 with the following specifications:

- `CPU: Intel i5-8250U (8) @ 3.400GHz`
- `GPU: Intel UHD Graphics 620`

I didn't run a proper test noting the frames per second. Although on my machine it swung wildly between approximately 400fps to 1200fps.

6 Discussion

It should be clear from the screenshots that the shader using dual paraboloid mapping does not work quite as intended. As mentioned earlier the texture is a

bit off, but mostly I had issues with multiplying the vertices with the camera's projection matrix. I think it might due to it having the property of being view-independent, but I can't quite wrap my head around what can be done to solve it.

As for the comparison of performance, I did not get any consistent results in accordance with the theory.
This may be due to how I structured the program, with loading and caching the different shaders and models, so the program could run as a demo. Although I can't make much sense of the increasing vertex count not slowing it down, as draw call only processes the selected shader and model.
Although when looking at the code for the different fragment shaders, they don't differ much in execution, in the sense that they only consist of arithmetic operations (no loops nor branching). So it would be expected that their running times are similar, however I would have expected cubemapping to be marginally slower than the other methods. One could also argue that the method for estimating running times using the `chrono` library and a simple fps counter doesn't suffice in this case. Instead I should have maybe looked into an OpenGL profiling tool or inspect instructions and how they're pipelined.

7 Conclusion

Overall I think I succeeded in implementing the different environement mapping techniques, except for the implementation of dual paraboloid maps. Hopefully the reader also gained more insight into the workings of the algorithms.

I think my investigation of running times failed quite a bit, however I was impressed by the frames per second I could achieve on my machine and how fast the algorithms are.
If one where to take this project further, one could either start investigating dynamic reflections by looking into Screen Space Reflections or Ray Tracing. Another direction would be to take these projections and use them Shadow mapping.

References

[Pau99] Rui Bastos Paul Simmon. *Exploiting Sphere Mapping*. Tech. rep. http://www.cs.unc.edu/techreports/99-028.pdf. Computer Science department at The University of North Carolina at Chapel Hill, Dec. 1999.

[Bar] Sean Barrett. *stb_image*. URL: `https://github.com/nothings/stb/blob/master/stb_image.h`.

[Her] David Herberth. *Glad*. URL: `https://glad.dav1d.de/`.

[Inca] Khronos Group Inc. *Cubemap Texture - OpenGL wiki*. URL: `https://www.khronos.org/opengl/wiki/Cubemap_Texture`.

[Incb] Khronos Group Inc. *reflect - OpenGL 4 Reference Pages*. URL: `https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/reflect.xhtml`.

[Vria] Joey de Vries. *Camera - LearnOpenGL*. URL: `https://learnopengl.com/Getting-started/Camera`.

[Vrib] Joey de Vries. *Mesh - LearnOpenGL*. URL: `https://learnopengl.com/Model-Loading/Mesh`.

[Vric] Joey de Vries. *Model - LearnOpenGL*. URL: `https://learnopengl.com/Model-Loading/Model`.

[Vrid] Joey de Vries. *Shaders - LearnOpenGL*. URL: `https://learnopengl.com/Getting-started/Shaders`.