TDT4230 Final Project

Tore Bergebakken

Concept

The project's goal was a scene with a heap of CRT monitors showing various visuals rendered in real-time, as well as reflecting the surrounding scene in a somewhat realistic manner.

A personal note: I stumbled down the rabbit hole of this rendering technique a month or so before the project started via some intriguing videos [1], [2], and decided I would delve into this in the project in some way. This is why the project ended up being an exploration of ray marching in addition to a more conventionally rendered scene.

Implementation

The project was written in Rust with the `glow-rs` wrapper for OpenGL. The scene graph was implemented with node references as indexes into a list of nodes, inspired by an article that described a simple way to do tree structures in Rust without dealing with the language's pecularities [3]. Lighting and material properties was handled similarly to the last assignment. The final scene uses models from *PolyHaven*, one specific goose model by *Atlas on Sketchfab* and a simple CRT model (with artistic liberties) made for this project.

Animated Camera

The camera was intended to revolve around the center of the scene, showing the content of the screens and reflections appearing on the screens as the viewing angles turned steep. This was achieved using a look-at matrix with vectors defined like so:

$$
\begin{aligned}
v_{\text{ground}} &= [r \cdot \cos\theta, 0, r \cdot \sin\theta], \quad v_{\text{eye}} = [r \cdot \cos\theta, h, r \cdot \sin\theta] \\
v_{\text{center}} &= \text{lowermost point of the crt structure} \\
v_{\text{up}} &= (v_{\text{center}} - v_{\text{eye}}) \times ((v_{\text{center}} - v_{\text{eye}}) \times (v_{\text{ground}} - v_{\text{eye}})),
\end{aligned}
$$

where r is the radius and h is the height of the orbit, and θ is the current angle of rotation.

For inspecting visuals on the screens more closely, button presses were tied to animations that zoomed in and made the camera stay in front of a chosen screen. The transition was achieved using linear and spherical interpolation (GLM's `mix` and `slerp` functions) of position and orientation vectors respectively, inspired by a StackOverflow answer [4]. The animations flowed smoothly, but it was possible to clip through the heap of CRTs during the transition.

Reflections

For real-time reflections, two different approaches were tested. Both involved rendering the scene from the perspective of the reflective objects. Regardless of the approach used for rendering the reflected environment, the reflection was added to the color of the reflective surface weighted by an approximation of the Fresnel effect [5], see fig. 1 for a visual example.

$$\max(0, \min(1, \text{bias} + \text{scale} \cdot (1 + I \cdot N)^{\text{power}}))$$

The first approach was to render the scene from the perspective of the reflective object to *one* two-dimensional texture. When rendering, the texture coordinates of the objects were combined with the reflection vector to approximate the look of a reflection, using the following homemade mapping that basically shifts the texture coordinates by a factor of a transformed reflection vector:

$$r_{uv} = (0.25 \cdot (t_{uv} - 0.5) + 0.75 \cdot 0.5 \cdot (TBN^{-1} \cdot r_{x,y})) \cdot 0.5 + 0.5,$$

where t_{uv} is the texture coordinates, TBN is a matrix that transforms from tangent to world space, r is the reflection vector and r_{uv} is the reflection texture coordinates. This mapping resulted in an odd bending of the reflections, and did not look particularily realistic, see fig. 2.

The second approach was to use cubemaps — collections of six textures representing the faces of a cube. In the case of reflections, the textures are based on what can be seen of the surrounding area from a reflective object. This was implemented using look-at matrices with up and center vectors tailored to each of the six faces. See fig. 3. (*entering personal mode*) I struggled with rotating and indexing the cubemaps properly and ended up with an inadequate implementation. My attempts at rotating the viewpoint that the cubemaps were rendered from did not garner much success, and rotating the reflection vectors used to index the cubemaps did not give meaningful results either. It did not help that I was fundamentally unsure of how the reflections of this particular scene would look in the real world.

Implementing both approches was more troublesome than expected. Creating the transformation matrices for the perspectives of the reflective objects turned out to be a minefield of near-solutions that flipped some part of the scene around and sometimes required reversing the winding order to have the result make any sense at all. The transformation for the planar reflections was reworked to use the total rotation along each axis instead of extracting rotation from the model matrices. This turned out to be easier to deal with. In the final stretch of the project, the reflections were tested in a more complex environment (compare to fig. 4 and 5) and several issues were discovered and fixed — the most egregious was the lack of depth testing because depth buffers had not been bound in the creation of the off-screen framebuffers, see fig. 6. Another issue was that the reflections were rendered upside down, which was not clear from the test scene. In the final result, the reflections based on cubemaps

did not handle rotations between the four cardinal directions well. Additionally, since the reflections needed a moderately large canvas for details to carry over properly, and adding interesting animations to the surrounding environment was not prioritized, they were rendered once at the start instead of for each frame. Lastly, the reflections were not intended to contribute *that* much to the scene, given how they only become visible at steep angles and do not obscure the visuals on the screens when viewed from the front.

Visualizations

The contents of the screens were rendered one by one in a first pass to a texture, before being merged with the 'ordinary' scene in a post-processing step. This initial rendering involved drawing each screen plane with the specific fragment shader that rendered its contents, to the same framebuffer successively. This allowed the visuals to be displayed at arbitrary resolutions depending on how the camera was positioned, see fig. 7 and 8. Rendering only the pixels that are needed would be beneficial for reflections as well, but testing yet another approach to reflections was not prioritized.

Ray Marching

Most of the screens were made to show some ray-marched scene. Ray marching is performed by stepping along a ray with step distance equal to an estimate of how far away anything in the scene is, until the distance drops below some threshold and the ray must have hit something. This technique can be modified to achieve various effects.

To find the closest distance to anything in the scene, ray marching may use signed distance functions, hereafter called SDFs. SDFs give exact or estimated distances to a surface, as well as the distance to a surface from *inside* the surface, with the latter represented as negative values. As an example, consider the SDF $d = |p - c| - r$, where d is the distance from p to the sphere with center c and radius r. Here, any point inside the sphere will have $d < 0$ since $|p - c| < r$, and any point outside the sphere's surface will have $d > 0$ since $|p - c| > r$.

SDFs can be combined using simple operations like `min` and `max` (corresponding to boolean union and intersection, respectively) to create more complex shapes [6], see fig. 9 for a smoothened example. Additionally, one can increase the complexity of a surface by adding some other function to its SDF, as shown in fig. 10, or by applying other transformations like scaling or twisting, typically by transforming the input to the SDF. Some of the shaders in this project use Inigo Iquelez' smooth combination operators [7], and some SDFs were taken from the same person's library [8].

On the less accurate side of SDFs, it is possible to use functions from 2D coordinates to scalars as 3D SDFs by letting the distance from a point to the y value given by the function be the distance estimate

to the whole landscape. Two of the shaders use this approach, with terrain generation from a tutorial on generating landscapes in this way [9], see fig. 13. This sort of SDF has blatant inaccuracies that can be alleviated by stepping *less than* the estimated distance to the scene along the ray while marching (compare fig. 11 to 12), which might lead to reduced performance. *Note that the two landscape shaders may not render properly on Nvidia GPUs. I do not know why.*

Effects

Glow was added to some of the shaders by finding the number of steps taken to reach the back of the scene and mixing in a color based on a multiple of this step count [1], [10]. This created a ringing effect outside the intense glow that I personally liked the look of. However, adding this effect to a scene with a periodic and somewhat inaccurate SDF showed a weakness of this technique, where the glow would be strong in parts of the scene that were empty, see fig. 8.

Since ray marching casts rays through a scene, shadows can be rendered by casting a ray to each light source from the point hit by the intial ray, and checking if these rays hit anything on the way. Shadows can be softened, as shown in fig. 14, by using the minimal distance found on the way to determine how dark the occluded point should be. The shaders with shadows in this project use a soft shadow approch proposed by Inigo Quilez [11]. Some of the shaders displays the light source as a glowing orb by sampling the distance to the light source at fixed point along a ray, see fig. 15 and fig. 16.

Conclusion

Doing reflections in the ways I attempted in this project does not seem *ideal* for live reflections. Rendering textures of decent resolution with full detail for even a few reflective objects might be more taxing as rendering the scene itself. I chose to do it this way because it seemed like a simple way to add reflections, but alas, I was wrong.

Compared to the reflections, displaying the output of different fragment shaders on meshes in a scene was rather easy. The project ended up having a wide variety of shaders demonstrating different effects accomplished with ray marching.

References

[1] C. Parade, “How to make 3D fractals.” https://youtu.be/svLzmFuSBhk (accessed Apr. 15, 2022).

[2] M. Steinrucken, “Ray marching for dummies!” https://youtu.be/PGtv-dBi2wE (accessed Apr. 14, 2022).

[3] B. Lovy, “No more tears, no more knots: Arena-allocated trees in rust.” https://dev.to/deciduously/no-more-tears-no-more-knots-arena-allocated-trees-in-rust-44k6 (accessed Apr. 15, 2022).

[4] A. M. Coleman, “Answer to ’how to implement a smooth transition between two different camera view in opengl’.” https://stackoverflow.com/a/27192680 (accessed Apr. 17, 2022).

[5] R. Fernando and M. Kilgard, “The cg tutorial,” Addison-Wesley, 2003. Accessed: Apr. 15, 2022. [Online]. Available: https://developer.download.nvidia.com/CgTutorial/cg_tutorial_chapter07.html

[6] J. Wong, “Ray marching and signed distance functions.” http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/ (accessed Apr. 17, 2022).

[7] I. Quilez, “Smooth minimum.” https://www.iquilezles.org/www/articles/smin/smin.htm (accessed Apr. 15, 2022).

[8] I. Quilez, “Distance functions.” https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm (accessed Apr. 15, 2022).

[9] I. Quilez, “Painting a landscape with maths.” https://youtu.be/BFld4EBO2RE (accessed Apr. 17, 2022).

[10] M. H. Christensen, “Distance estimated 3D fractals (II): Lighting and coloring.” http://blog.hvidtfeldts.net/index.php/2011/08/distance-estimated-3d-fractals-ii-lighting-and-coloring/ (accessed Apr. 17, 2022).

[11] I. Quilez, “Soft shadows in raymarched SDFs.” https://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm (accessed Apr. 15, 2022).

Appendix

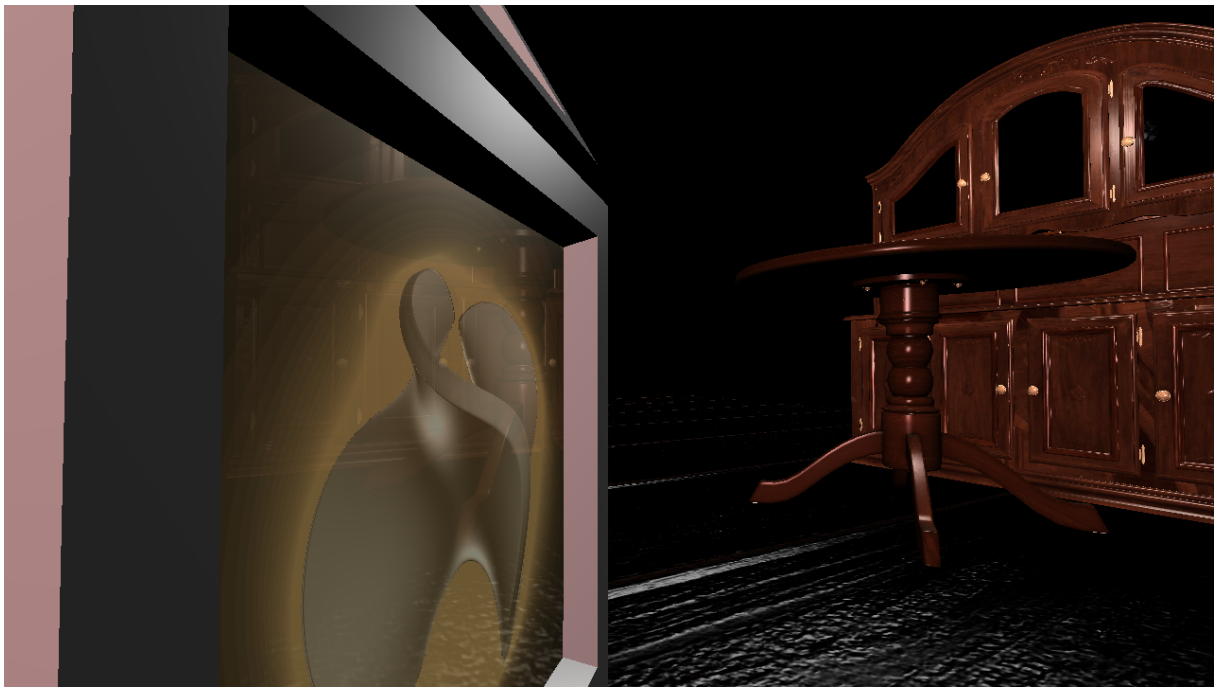

Figure 1: Fresnel effect kicking in at a steep viewing angle

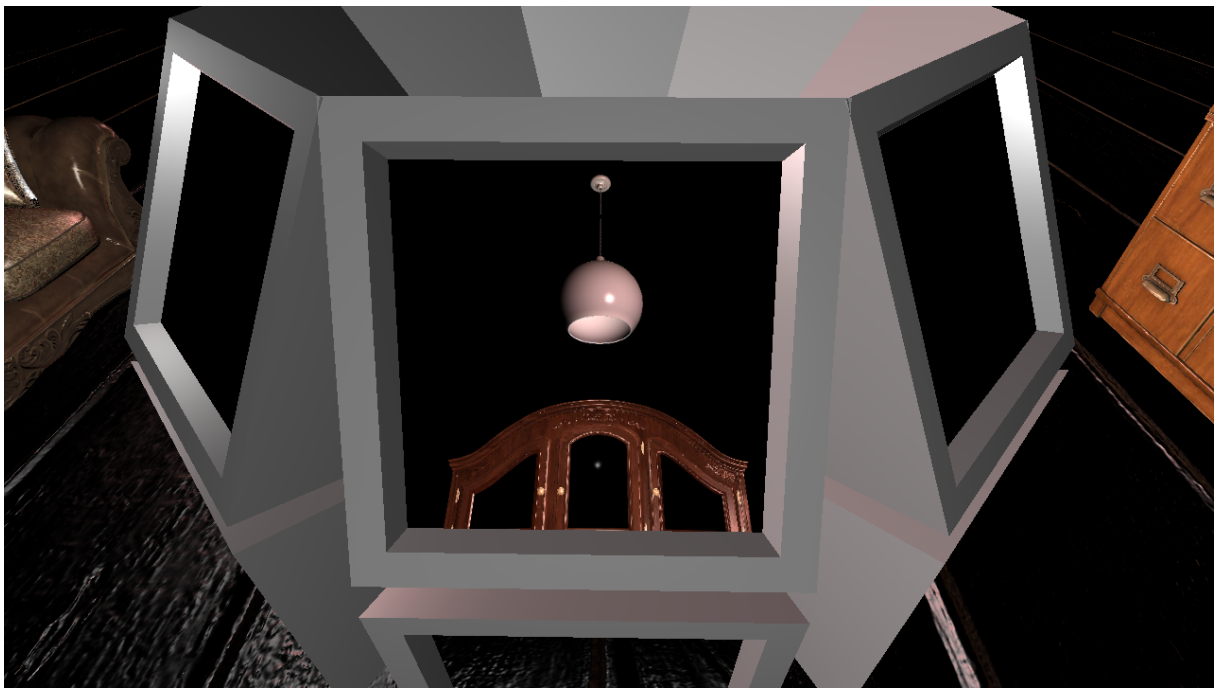

Figure 2: Planar reflection displayed in full w/o Fresnel

Figure 3: Cubemap reflection displayed in full w/o Fresnel

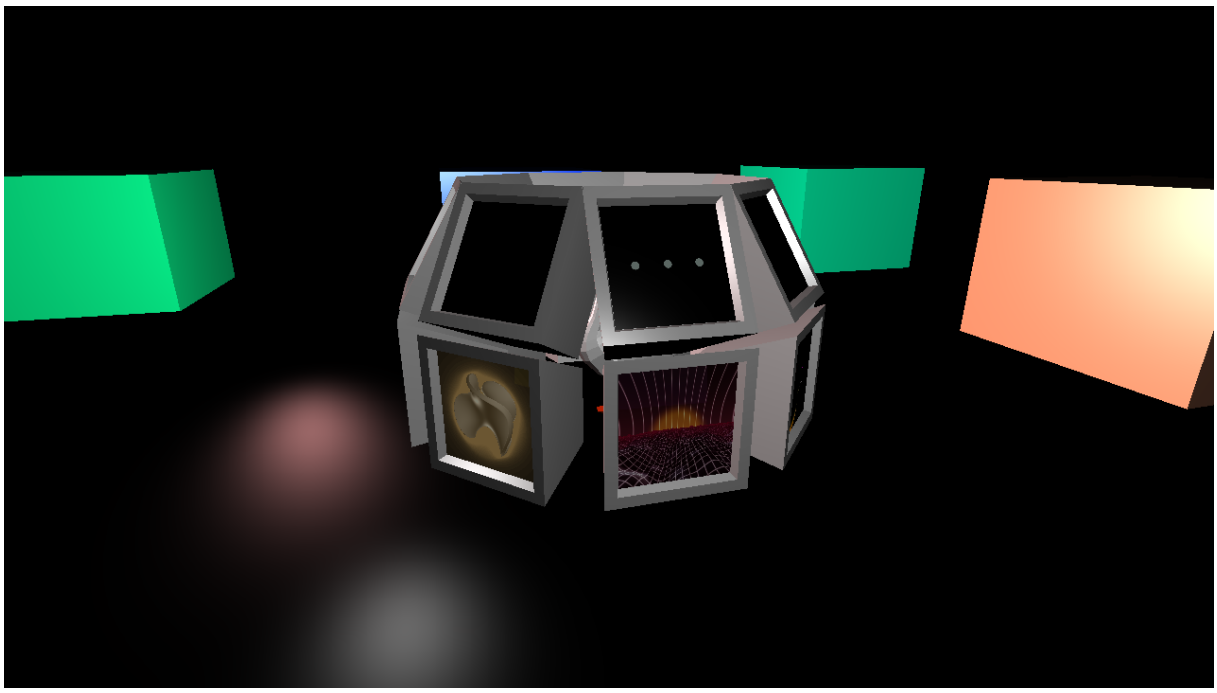

Figure 4: Test scene with differently colored cubes

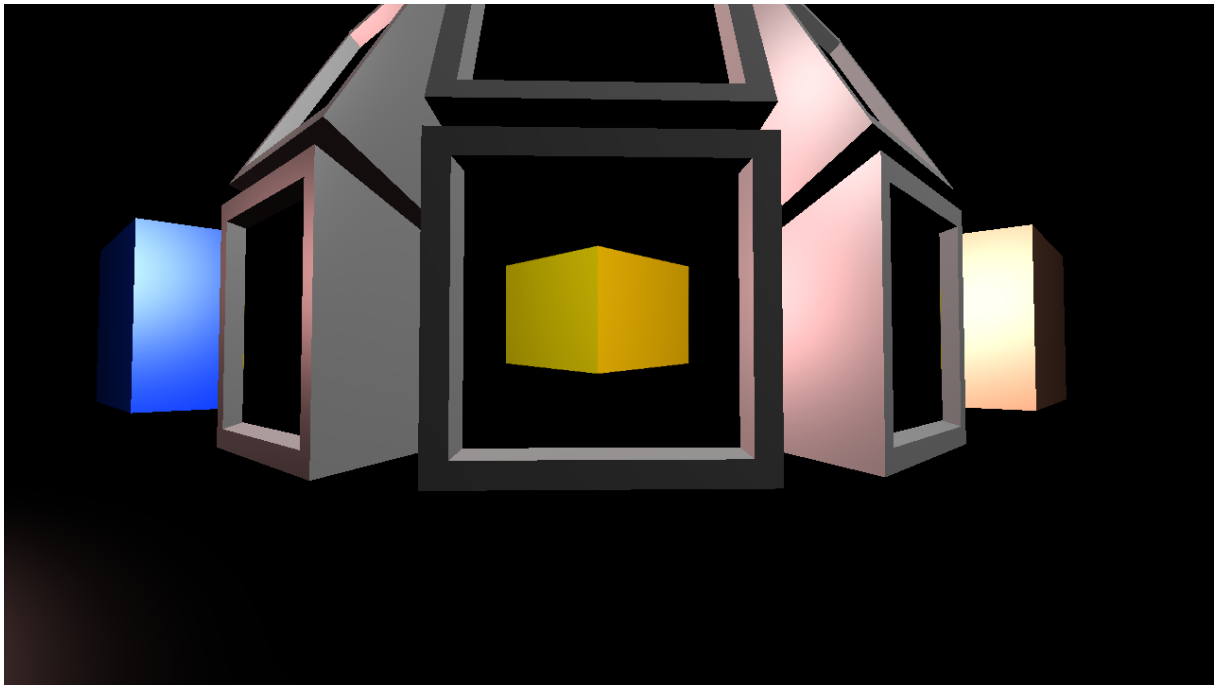

Figure 5: You see why it was a little hard to tell if this reflected cube was upside down?

Figure 6: Lack of depth testing, kept for posterity

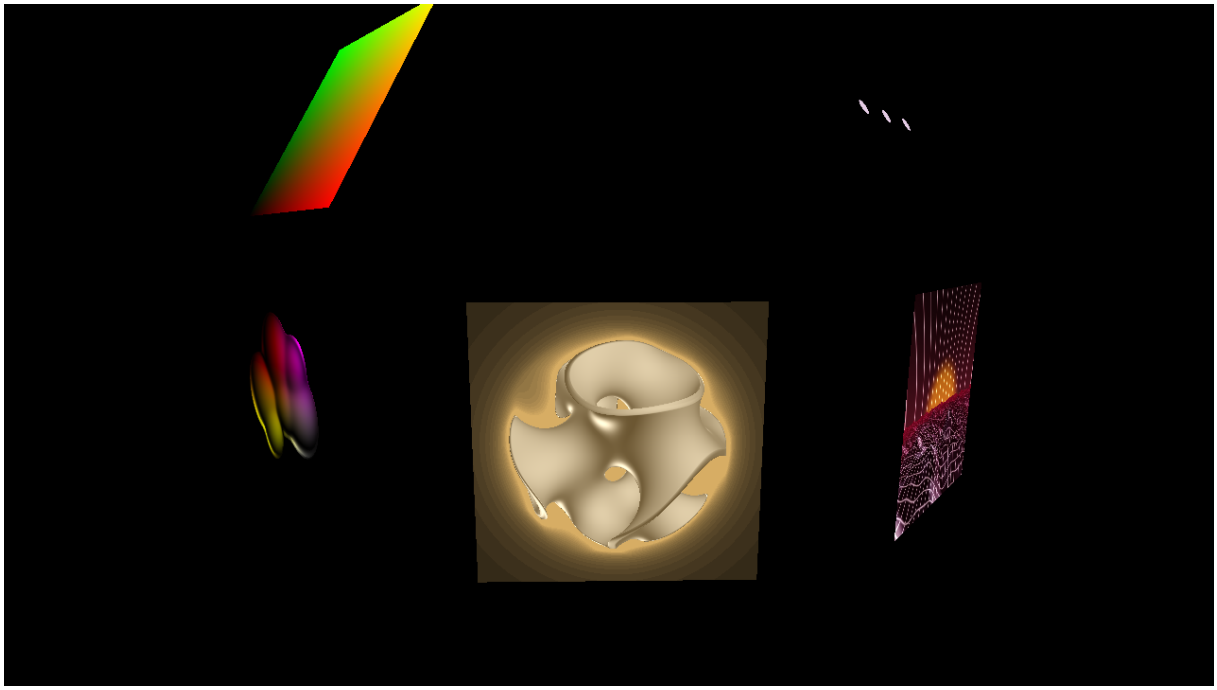

Figure 7: Several shaders rendered to a single texture based on the geometry they are displayed on

Figure 8: Moving closer to one screen increases the resolution its shader is rendered at

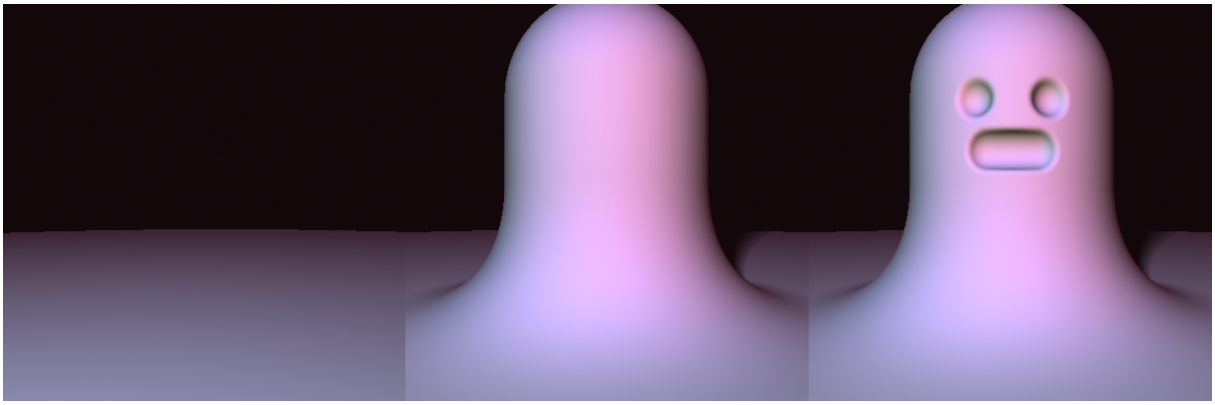

Figure 9: Intersecting a plane and a large capsule, then carving out a face with a difference operation

Figure 10: Sine waves added to a sphere SDF

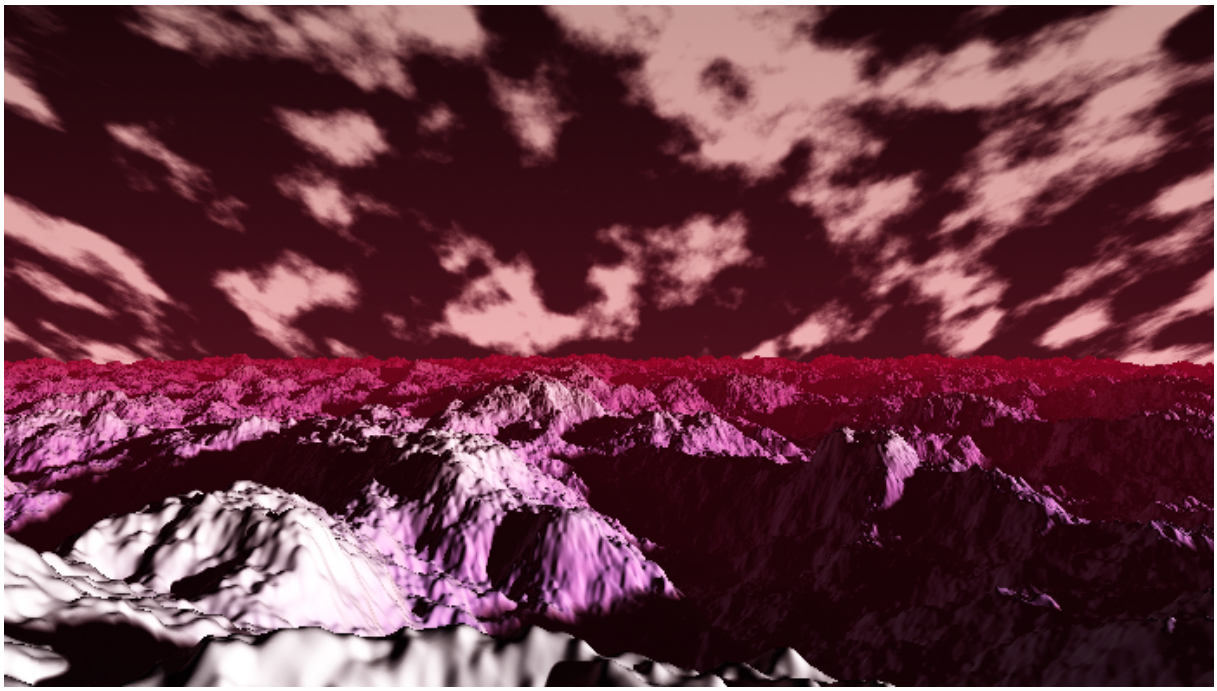

Figure 11: Value noise-based landscape

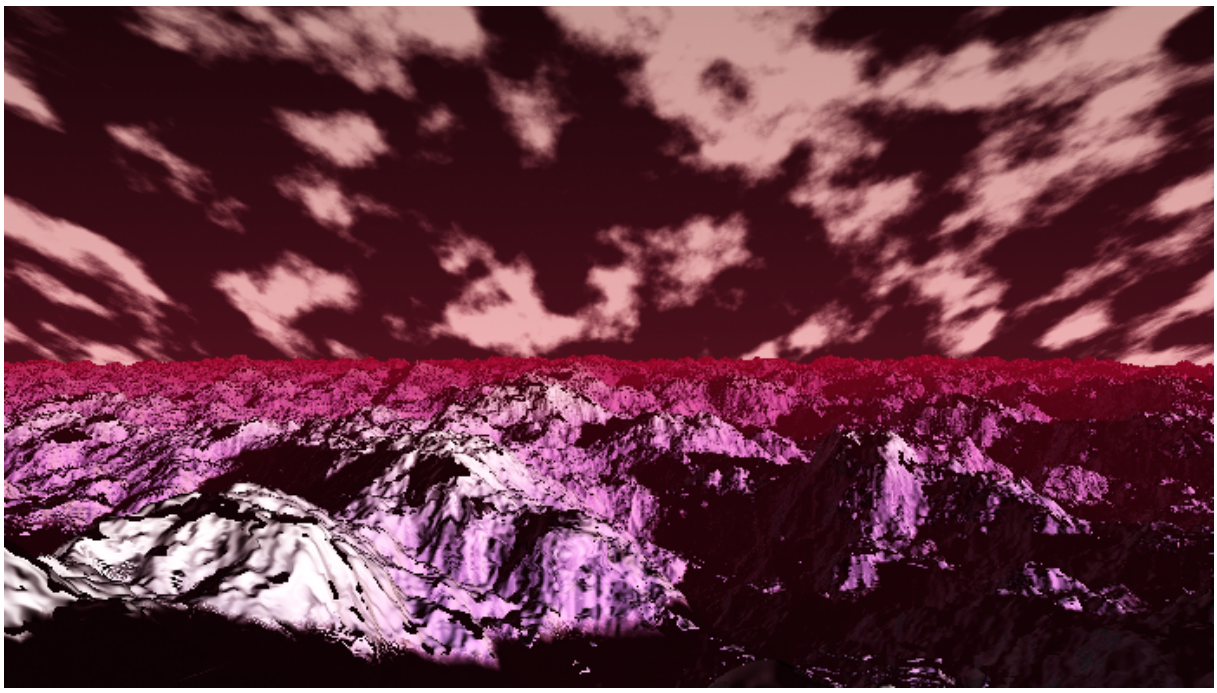

Figure 12: The same landscape with severe overstepping

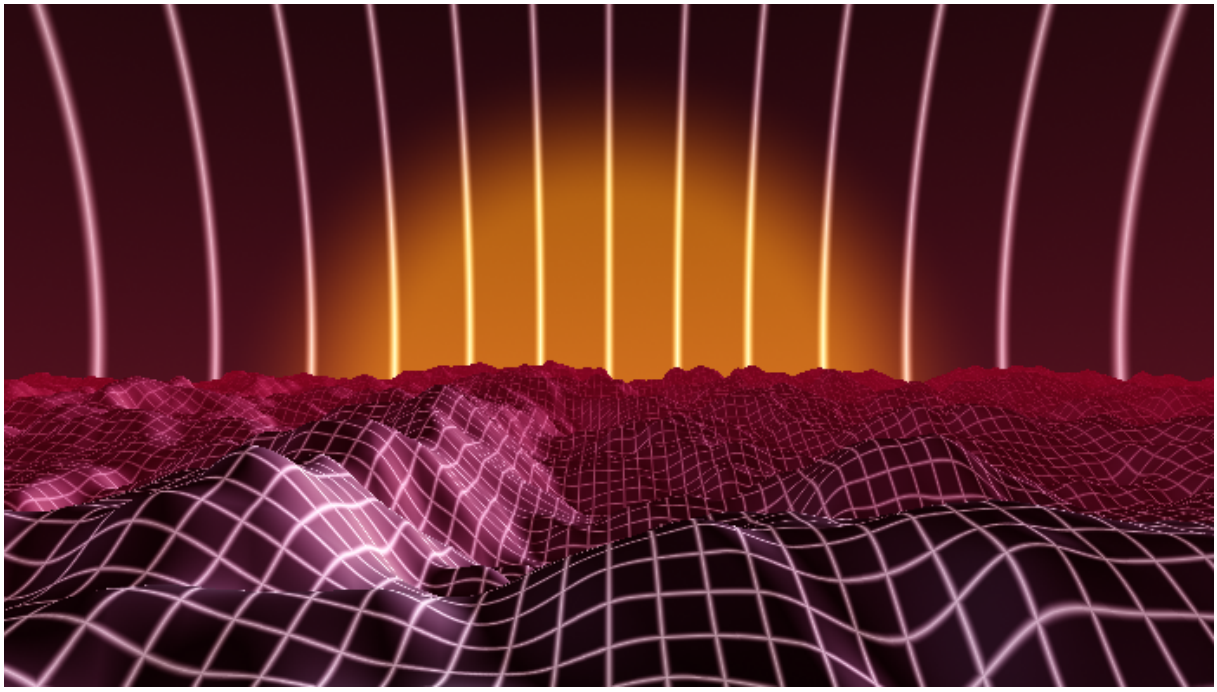

Figure 13: Landscape in retro- or whatever-it-is-wave style

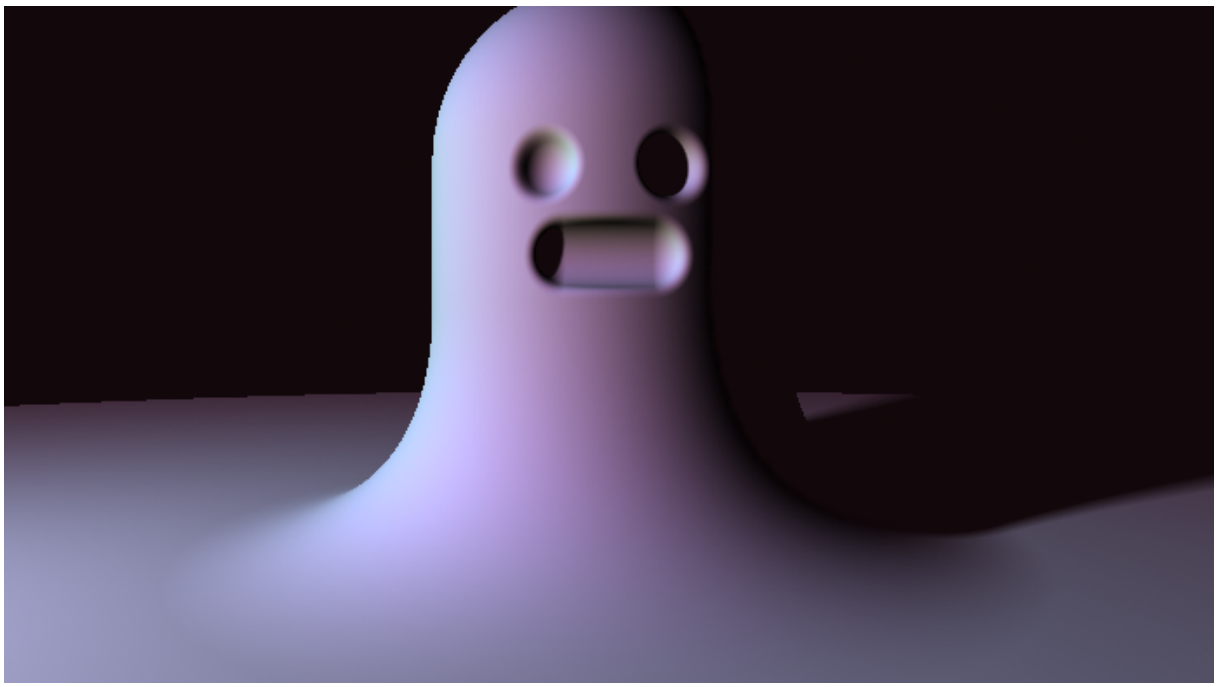

Figure 14: Partially shadowed statue

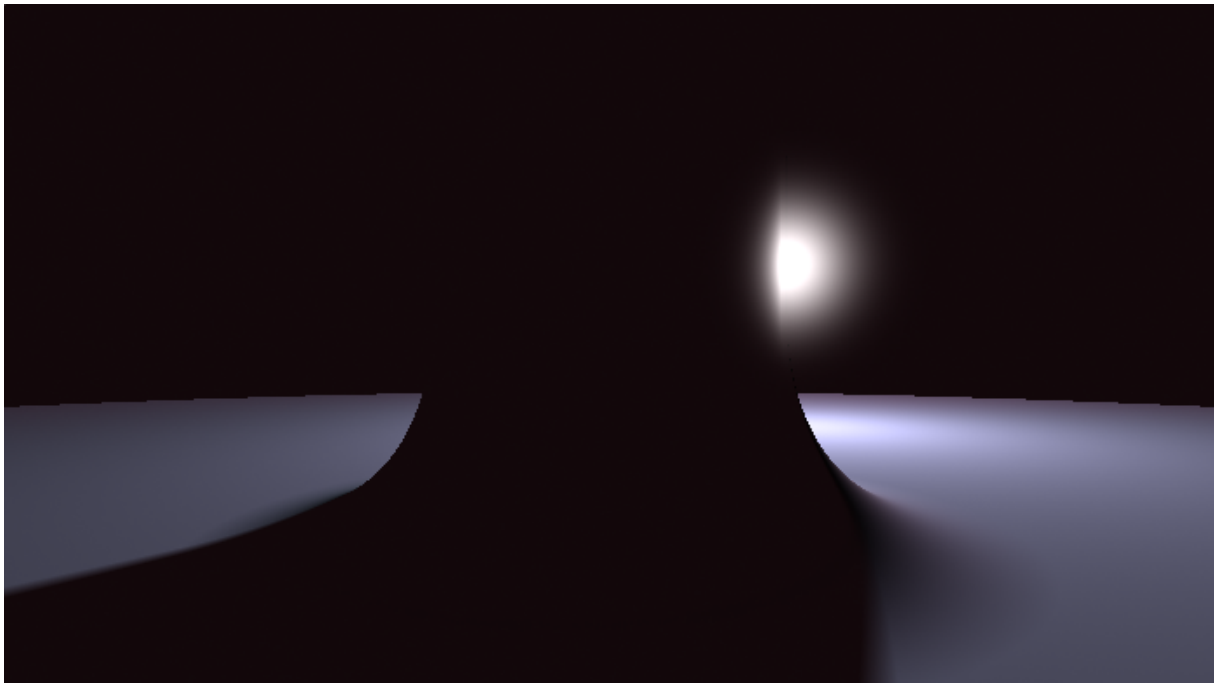

Figure 15: Light source peeking out beside shadowed statue

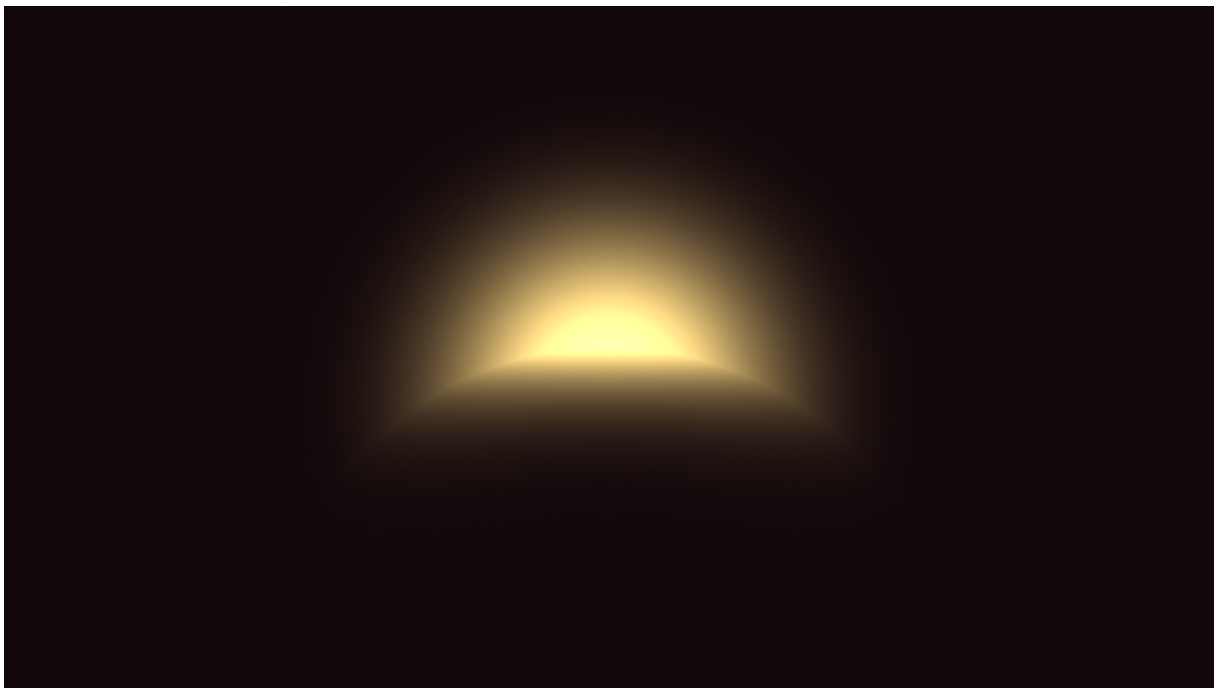

Figure 16: Sun rising or setting over some planet