TDT4230 - Graphics and Visualization

Final Project

Thomas Aven

April 8, 2019

Introduction

I've been interested in the world of ray marching for a while, in particular the fantastic work of Inigo Quilez[1]. His blog explains an enormous amount of techniques to create unbelievable procedural scenes. Most of these shaders are hosted at Shadertoy.com, which is an online tool for sharing shaders with WebGL. Code from other shaders I have been inspired by are all credited in comments in the source code.

My primary goal for this project thus became to create interesting fragment shaders, primarily using `gl_FragCoord`. For this report I will go through the process step by step, from the baby steps required to render a simple sphere, to the final leaps that render a realistic looking scene. Limitations of the techniques, as well as problems encountered will be pointed out along the way.

Building the project

The project uses CMake (Gloom) for the building process, as is required for the assignment. I've also taken the liberty of adding three libraries to use sound: OpenAL Soft, freealut and FFTW. OpenAL and freealut are added as submodules with Git, while FFTW is added as an external project in CMake. I suggest pulling the repository from GitHub[2], as this has been tested to work cleanly from scratch.

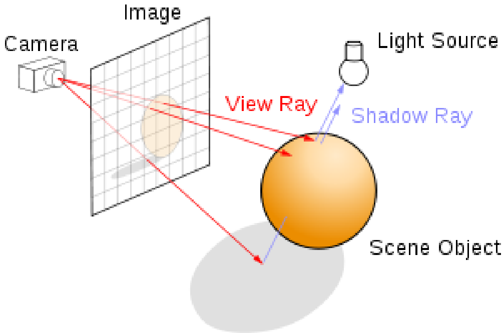
Ray Marching

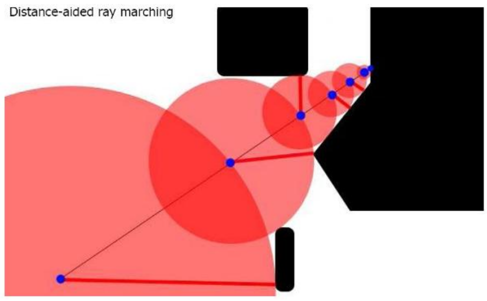
Now – what is this ray marching thing? In the world of *ray casting*, it is common to be familiar with *ray tracing* to compute the intersections of a light ray with surfaces. *Ray marching* may be used within such a ray tracing method, as it is a specific algorithm for this purpose. Using ray marching in combination with something that's called *signed distance functions* can make extraordinary scenes from infinitesimal binary executables, as all that's required are the underlying mathematical formulas.

A signed distance function, let's say a sphere centered at the origin, $f(x, y, z) = \sqrt{x^2 + y^2 + z^2} - 1$, can be used to determine whether a point is inside or outside an object, as well as the distance to the object if it is outside. This is in contrast to more well known ray tracing implementations that have to check for intersections with quite a lot of primitives.

So how does this work in practice? In `simple.frag`, there is a function, `sd_sphere`, that takes a point and a radius as arguments, and returns the distance to the sphere's perimeter. This is becoming interesting: below are two illustrations I have found to explain the further steps the best.

[1] `https://iquilezles.org/www/index.htm`
[2] `https://github.com/thomaav/graphics`





The left[3] image shows how rays are traced from a camera. The right[4] image illustrates how the iterative steps are taken by the ray marching algorithm according to the distance to the object closest to the current point.

Humble beginnings

Let's put our new knowledge to a test – the execution of the ray marching algorithm is found in any of the `trace_<object>` functions. A simple rendering of a sphere with some phong shading is shown in this image. Spheres are just the beginning; we have SDFs for a wide range of shapes[5].

Setting the stage

So we have a method of rendering a single sphere. How do we go about turning this into a complex scene? The first trick we will pull out of our sleeve is intersections and unions. If we compute the distance to more than one object, and then do `max` (intersect) or `min` (union) between them, we can have multiple objects in our scene. The technique can be seen in action here. A problem I encountered at this stage was using GLSL effictively. I practically gave up on optimization, but most of all I was missing the ability to pass around function pointers for distance functions (which could perhaps be done with `switch` statements anyway, as I'm already murdering the performance with conditionals).

Shadows

An advantage of signed distance functions is that they provide us with global information. Given a point on a surface in a scene, we can fairly easily explore our surroundings – we just have to recalculate the SDF with new points. For shadowing, we simply follow what's called a *shadow ray* from the surface point towards the position of a given light. If it

[3] http://hugi.scene.org/online/hugi37/sphere_tracing.jpg
[4] http://jamie-wong.com/images/16-07-11/raytrace.png
[5] https://iquilezles.org/www/articles/distfunctions/distfunctions.htm

intersects some other object on the way, the light will not contribute to the illumination. We can also put areas that are *almost* within the shadow under penumbra by checking how close we are to intersecting objects on the way. Illustration. This perhaps the most pressing advantage of ray tracing in general: effects such as shadows and reflections are natural results of the algorithm.

Ambient occlusion

So the shadowing in the previous section looks quite good, but the ambient lighting looks a little flat. We can get fake, fast ambient occlusion in a fairly simple manner: evaluate the distance function at a few points around the actual point we are shading. By comparing the results of the scene SDF at these points to the original point, we gain information about the proximity of other surfaces around us, and with this information we can make an educated guess on the occlusion of the surface we are shading. A limitation of this method is that it's a crude approximation, and may give results that seem *off* (e.g. floor occluding *a lot* of the light hitting the bottom of a wall).

Reflection and refraction

Planes can easily be represented as SDFs with a single height value, and wave-like displacements can be added with a simple sine, as can be seen here. Adding reflection is no harder than adding shadows – we simply march again from points of intersection in a reflected direction, and mix the reflection color with the reflective surface color (example). We also add a fresnel effect such that steeper angles give weaker reflections. At this point I started noticing how optimizing ray marching could give numerical instability, especially when estimating the normals of a sinc wave for lighting purposes. This is a weakness with ray marching, as we have to estimate the normal, as opposed to it being passed into the rendering pipeline.

Another important effect to add when working with water is refraction. Water is transparent, so we should be able to see the sphere when it's underwater. Refraction is similar to reflection in that we do another ray march, but this time we first bend the ray according to the refractive index of water, giving this effect.

Realistic waves

So we might be tempted to say that the effects above make a pretty cool shader, but we can do much better: time for a noise texture and fractal Brownian motion. Explanations of these methods are slightly too complicated to fit into four pages, but the implementation contains comments on the workings, as well as links to further readings. The effect of adding this noise is moving water that looks to be flowing in the pseudorandom motion water does in reality.

Realistically colored realistic waves

Our waves still look like plastic, much due to a weakness with the specular shading from the phong lighting, and the fact that the water still has intrinsic color. Now, let's set the default color of water to to resemble the darkness below, and make sure we only color the water by the color of the reflected sky. If we also lay a sheet of rain on the screen according to the noise texture, as well as spreading some splashes on the water surface in a random manner. We are starting to get something that looks like real water. At this point I was starting to notice one of the major disadvantages of ray marching: the performance. Rendering on my laptop, which has an integrated graphics card, required me to lower the resolution to 512x256.

Further incremental improvements

Now we add some clouds to the sky, by simply sampling our noise texture again, such that we can see the horizon in the distance. Then we add some lightning so the scene lights up at random intervals. Then we make the sphere into something that looks like a planet with lava by sampling another texture suited for this purpose (however, it is still procedurally generated). We're getting somewhere.

Sound and a Fast Fourier Transform

The CPU is mostly idling between the rendering of frames, but we can do something about this. Usage of a Fast Fourier Transform is very common in shaders. For this project I used FFTW to do an STFT over a .wav file of music (stolen from YouTube[6]), and set the sphere in the scene to visualize the lower frequencies of the song ($<$ 30Hz). This creates an effect of the sphere expanding on the onset of bass notes, especially the kick drum. When expanding the sphere we also see a problem with wrapping a square texture around a sphere – the poles stretch a lot.

A finishing touch

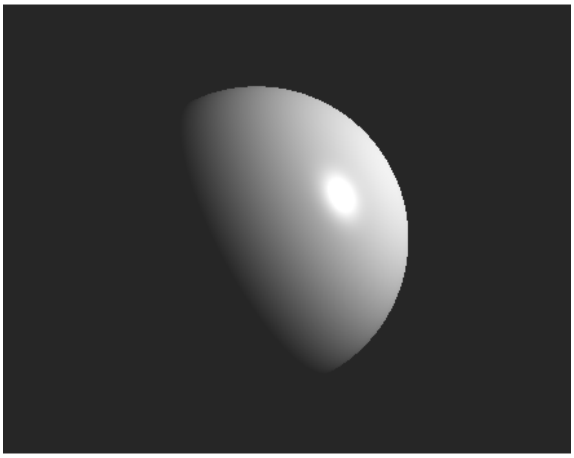
To finish the scene, I decided to combine some SDFs to create a periscope that would float across the scene. This is done by combining two cylinders with an elongated torus to create the pipes and window. They are combined with a smooth union. The pipes are made reflective, which looks fairly good, but a more matte, rusty surface might make it look less out of place. By doing this modelling by hand with SDFs, I got to feel how cumbersome this process is. There is a reason we have modelling tools, but I still have an immense amount of respect for the demo sceners that create these models procedurally. The final scene can be seen in this screenshot, or in a video that I've uploaded to YouTube[7]. The periscope is visible from about 8 seconds into the video.

[6] `https://www.youtube.com/watch?v=WeIIrFhrePE`
[7] `https://www.youtube.com/watch?v=hDzagq61y1U`

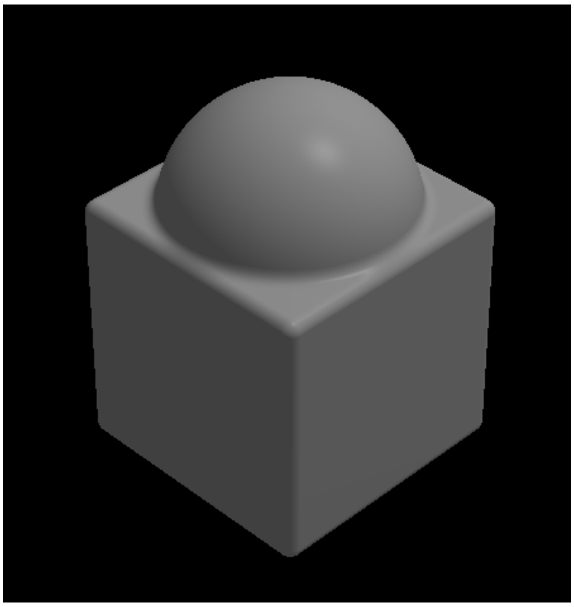
YouTube really did a number on the quality, so the full quality version is available[8] (recommended version – try with VLC or Chrome, the new Firefox wouldn't play the file).

[8] http://folk.ntnu.no/thomaav/graphics/shader.mp4

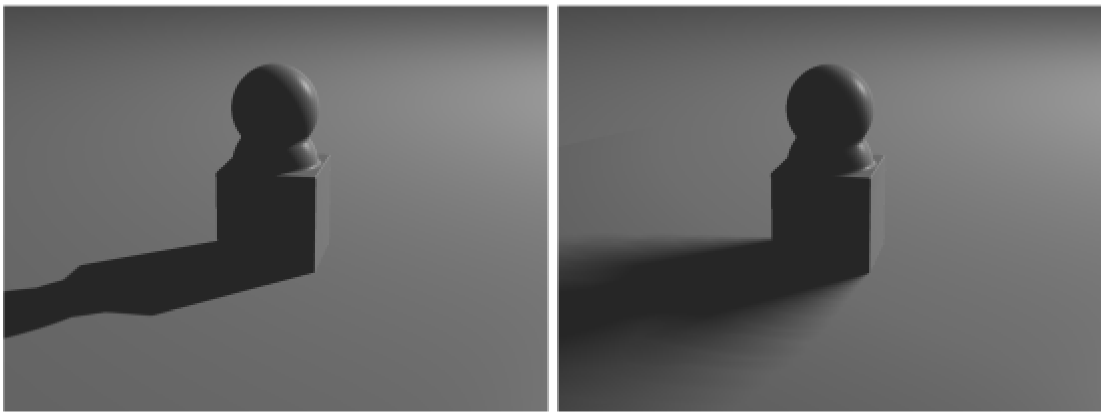
Appendix A - Images



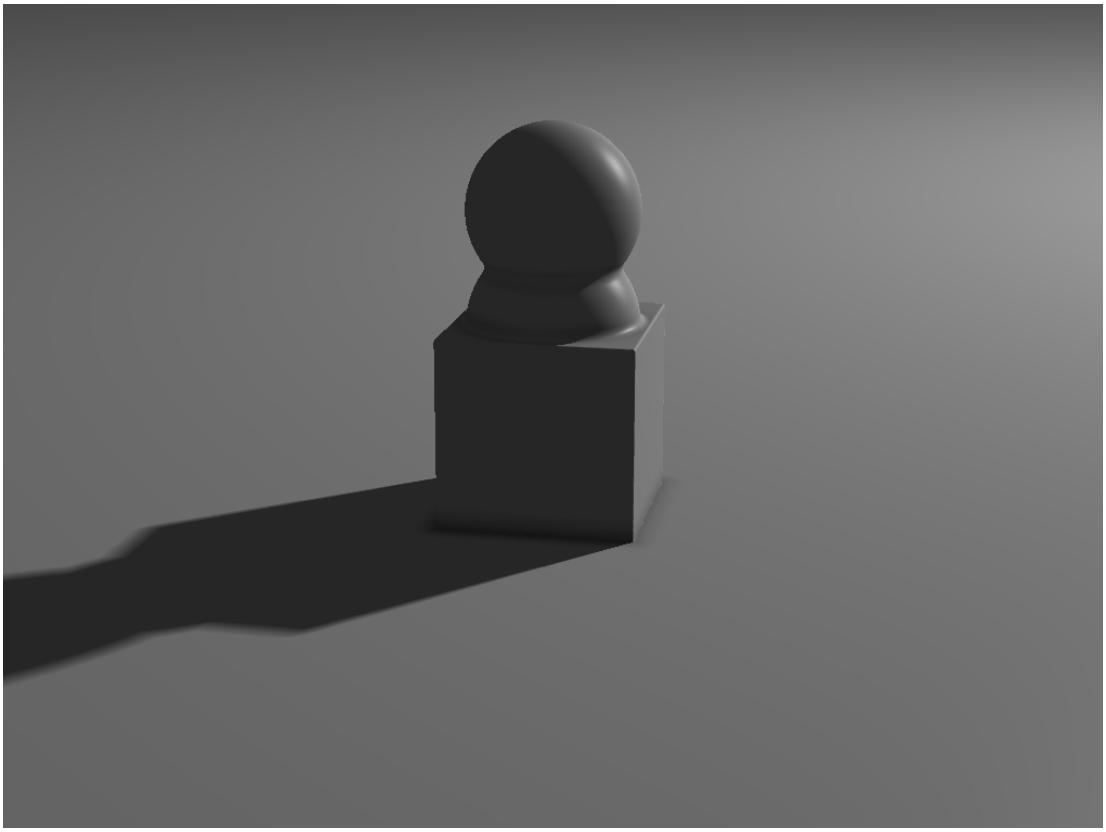
A simple ray marched sphere. Back to section.



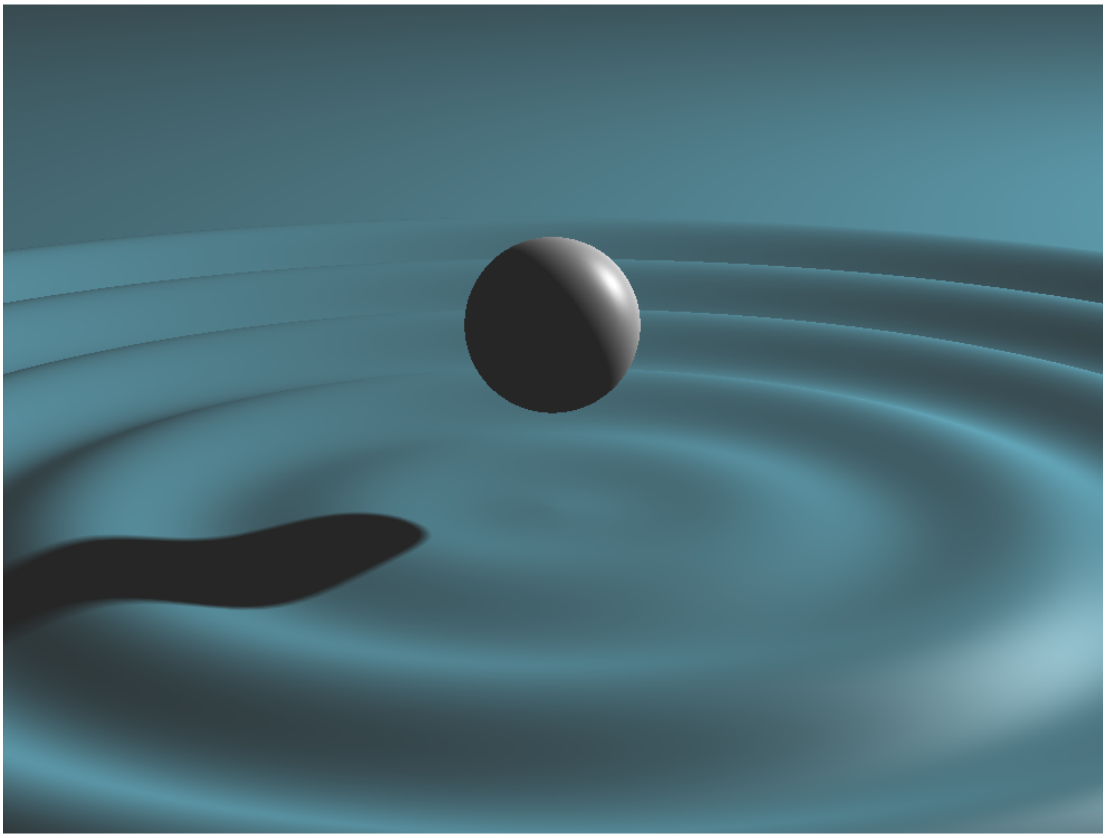
The union between a sphere and a cube. Back to section.



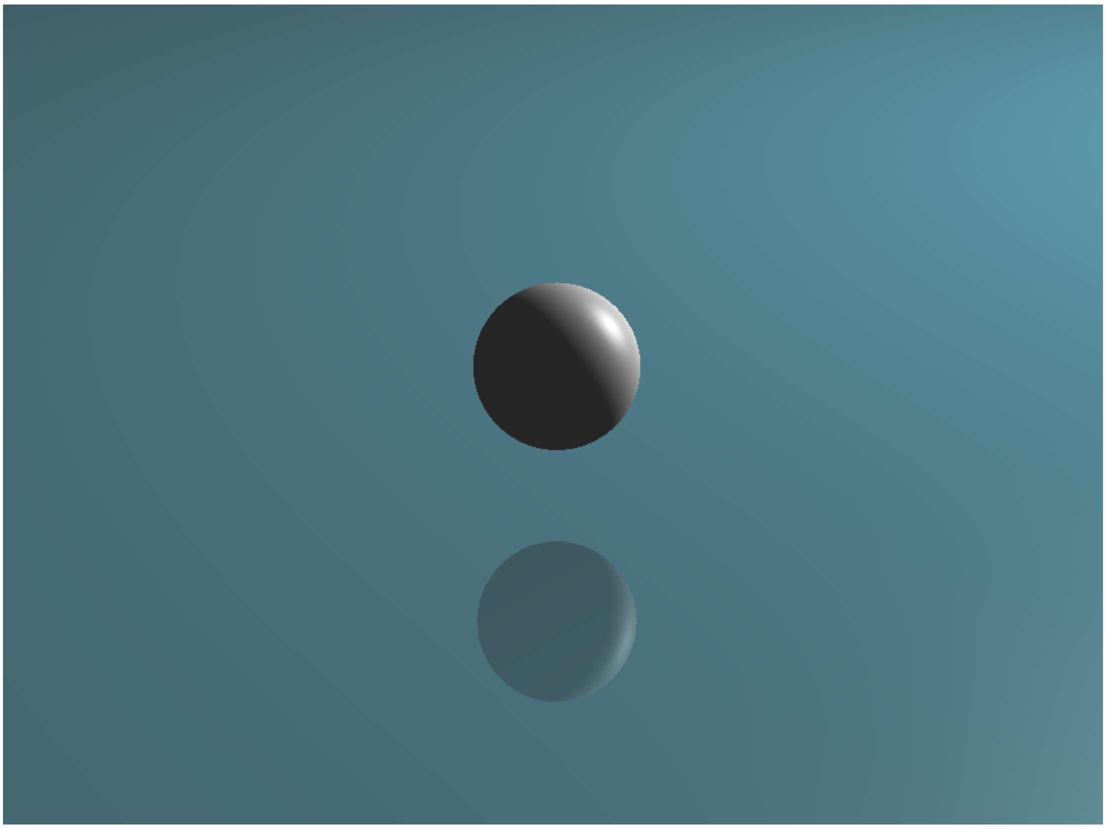
Penumbra shadowing in action. The left image has a `k`-value of only 2, while the right image has a value of 128. Back to section.



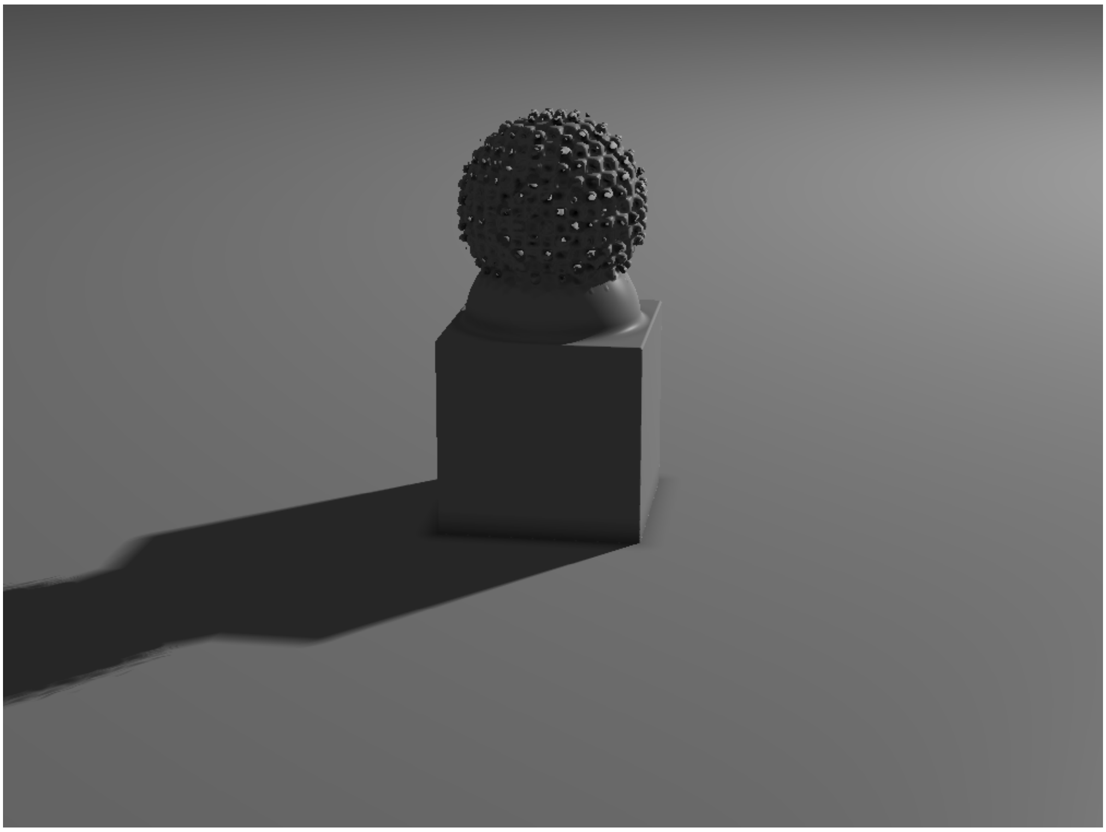
Ambient occlusion. Notice how some edges of the box are occluded by the floor. Back to section.



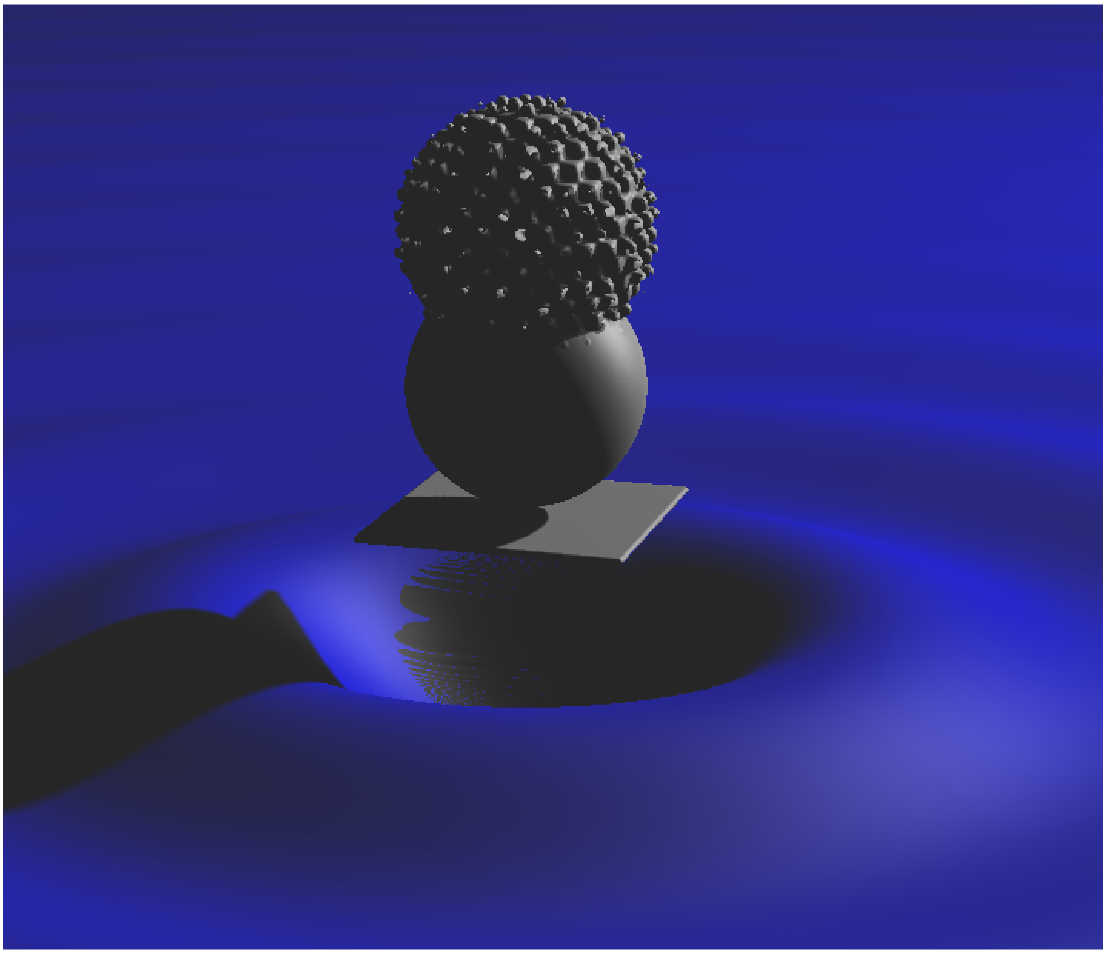
Very simple water shader in action, a gif can be found here: `http://folk.ntnu.no/thomaav/graphics/simplewater.gif`. Back to section.



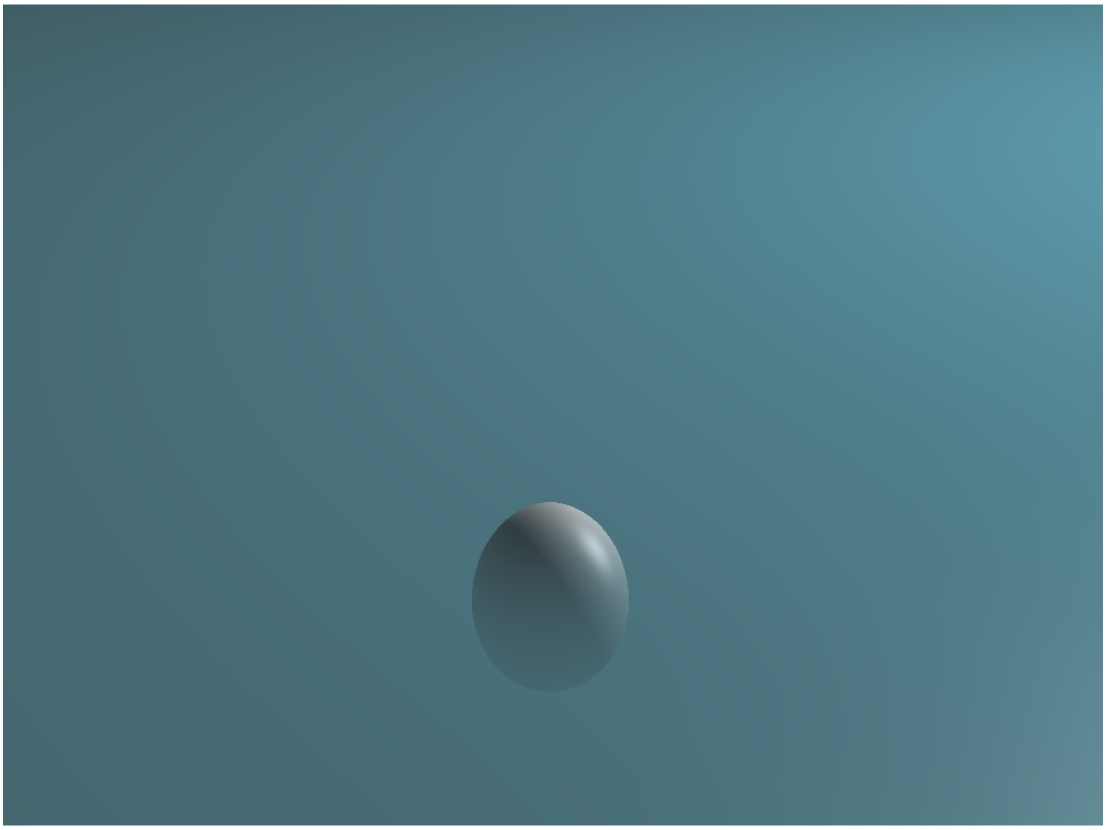
Reflection on the water surface, gif found at: `http://folk.ntnu.no/thomaav/graphics/reflection.gif`. Back to section.



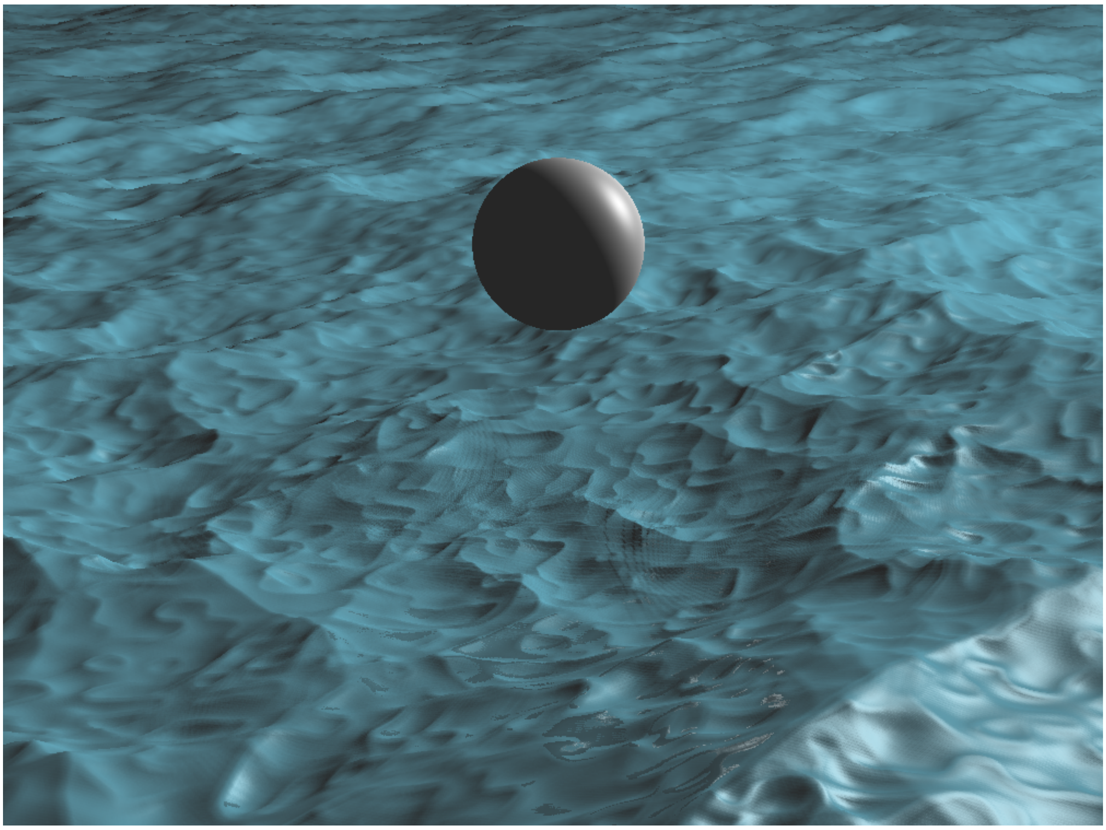
An example of the numerical precision failing when estimating shadowing on a sphere with displacement. Back to section.



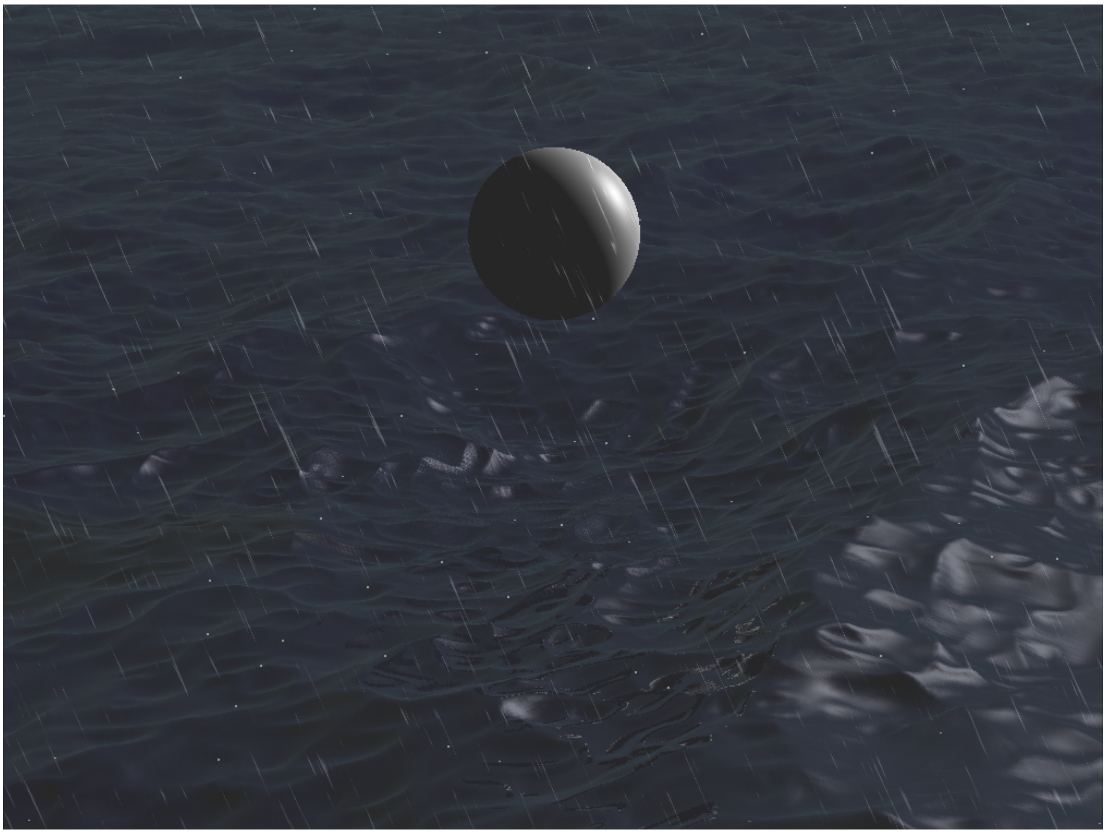
Another image of the numerical precision failing when estimating shadowing on a sinc wave. Back to section.



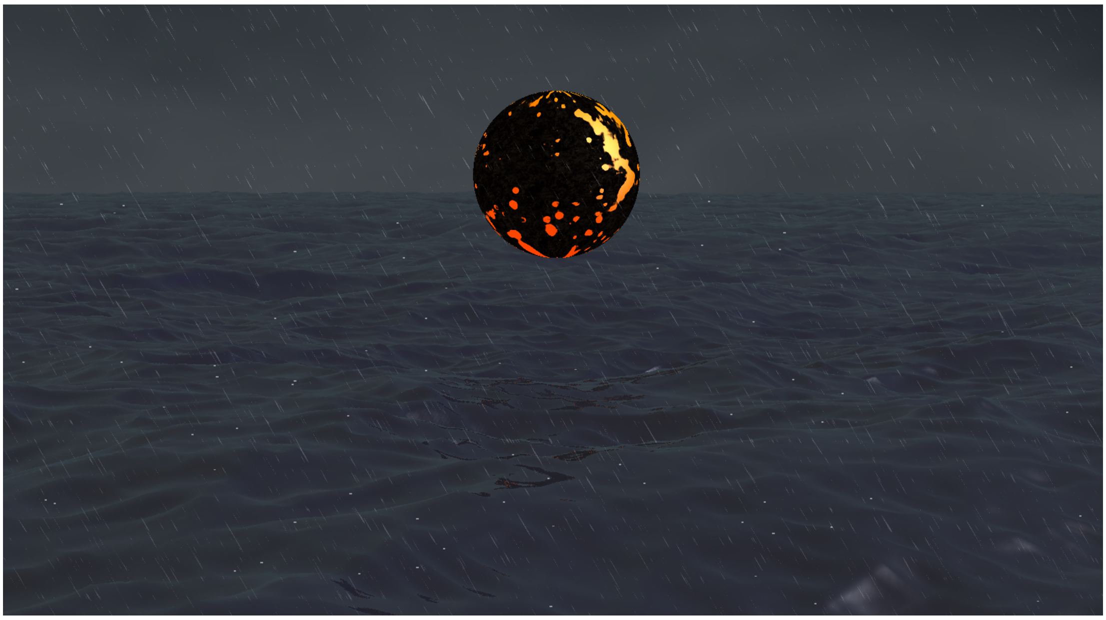
Refractive water surface. http://folk.ntnu.no/thomaav/graphics/refraction.gif.
Back to section.



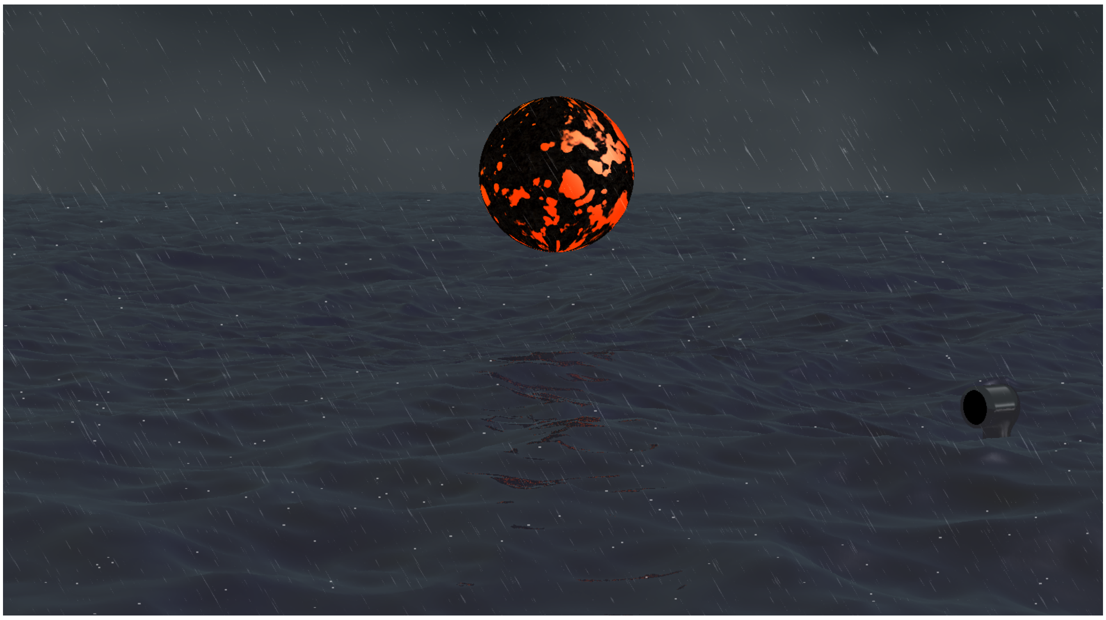
Water surface that is displaced with fBm. http://folk.ntnu.no/thomaav/graphics/noise.gif. Back to section.



More realistic coloring of the water. `http://folk.ntnu.no/thomaav/graphics/okwater.gif`. Back to section.



Further improvements on the scene. Includes procedurally texturing the sphere and adding clouds and lightning. Back to section.



The final scene – with the periscope visible in the lower right. The video is found at `https://www.youtube.com/watch?v=hDzagq61y1U` or `http://folk.ntnu.no/thomaav/graphics/shader.mp4`. Back to section.