DH2323 - Ray marching

Ted Klein Bergman, Sruti Bhattacharjee, Elias Alkvist Cetin

Abstract
The topic of ray marching has been known for some time and has become more relevant with time, as computer performance has gotten better. The present project aimed to develop different primitives precisely with the use of ray marching. Ray marching often uses the algorithm of signed distance functions to calculate distances, which was used in this project as well. A scene was implemented with lighting, shadows and several primitives having different materials and colors. Additionally, to achieve a twist on the primitives, distortion was added and gave rise to novel appearances of the scene. Our project can be viewed on the link above.

Introduction
Rendering is a substantial part of computer graphics and is the operation of producing scenes containing shapes, shadows and animations and giving those their appearance. It fundamentally produces 2D images projected onto a screen from 3D scenes. This process of producing 2D images from 3D scenes has different approaches and techniques with their respective advantages and disadvantages. Among one of the techniques is ray marching, which has its roots as early as in the 1980s (Perlin, K et al, 1989), but has gotten more and more relevant with time and with better computer performance.

Ray marching algorithm explained
Ray marching is a technique where a ray is generated and casted through a pixel to map an environment. It reminds a lot of ray tracing, however while in ray tracing the closest intersection on the ray is seeked and found, the ray marching algorithm rather takes a series of finite number of steps, until an object is found or the limit of steps is reached, essentially marching towards to the object step by step (Benton, 2021). It often makes use of an algorithm called signed distance functions in one ray marching derivative, sphere tracing, to improve its performance and prevent skipping solutions (Bálint et al., 2019). This means that when a ray is casted into the environment, the signed distance function calculates how far away the object is in any direction from the point, starting at the origin of the ray. This enables to move the point, the specific calculated distance, in the direction of the ray, since it is implied that the intersection will not be at that point. From that new position of the point, the algorithm continues until the intersection in the direction of the casted ray is found.

Purpose
After discovering the concepts of ray marching and the use of signed distance functions, we got eminently interested in them, leading to our aim being implementing our own project of ray marching. Much of the interest in ray marching came from the lab work in the course

involving raytracing, that is reminiscing the concept of ray marching but also differs, which we found compelling. We concluded that our interest particularly was in implementing different model shapes, such as a sphere, a box and a plane.

Related work

In our process of researching the not very great supply of the topic of ray marching and signed distance functions, we however found some projects that worked as inspiration for our own project.

The tutorial on creating a sphere with the technique of ray marching, by (Walczyk, 2021) introduced us to implementation of a sphere and additional effects such as distortion on the sphere with the help of signed distance functions. This distorted sphere sparked a greater curiosity on raymarching and what could be done.

Lighting and Phong shading with use of ambient, diffuse and specular light are things we have gained knowledge about through different courses for instance. The source of LearnOpenGL, however, gave us additional insight on how those interpolation techniques would be implemented into code. (LearnOpenGL, 2021)

Another tutorial regarding distortion caught our attention, which made use of blur and remap functions to create interesting effects on different scenes. This was a considerable source of inspiration for parts of the distortion effects together with the source of Walczyk. (The art of code, 2017)

Lastly, we found quite a substantial library of articles from Inigo Quilez, from which we got the signed distance functions for our primitives as well as the soft shadows. (Quilez, 2021)

Implementation & results

This section presents our implementation process of ray marching on different model shapes.

Due to what was found and presented in our previous work, our initial desire was to implement a scene containing some primitives based on signed distance functions, phong shading, single directional light, and a couple of displacement functions.

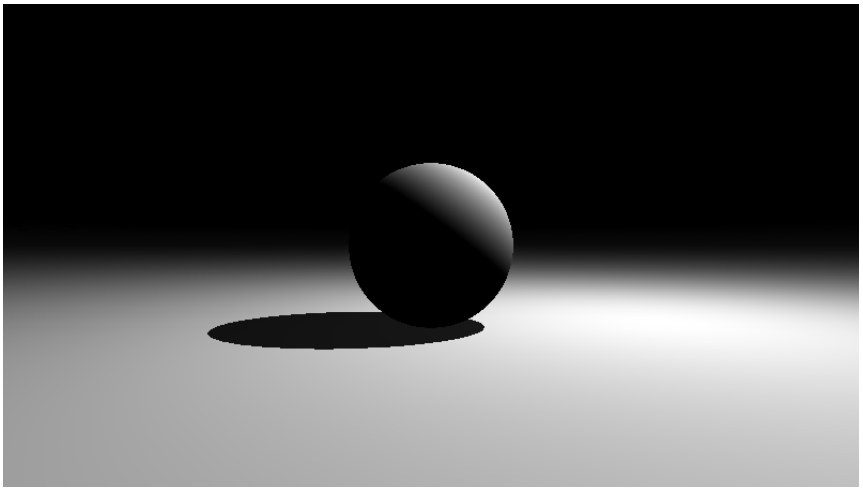
Primitives/shapes

The first primitive we added to the scene was the sphere, due to it being the simplest. All primitives are defined by a signed distance function, which is a function that returns the distance to the primitives surface. In other words, positive values define the distance to the primitive, while negative values tell us that we’re inside the primitive. This allows us to query the distance of all objects in the scene to get the shortest distance we can move along our ray without walking past any objects. Once we retrieve a value of 0, we know we’ve hit a primitive.

The signed distance function for a sphere is simply pythagoras theorem minus the sphere's radius.

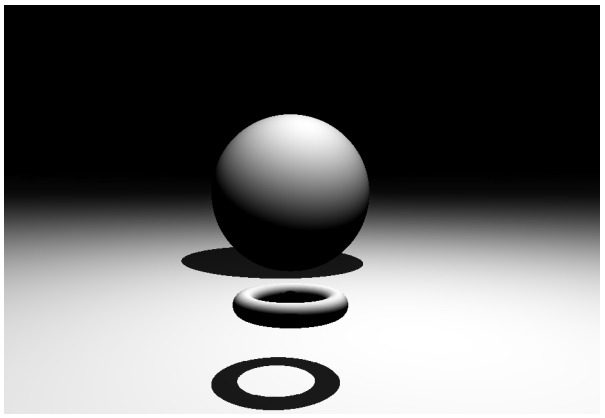
$$\sqrt{(p_x - s_x)^2 + (p_y - s_y)^2 + (p_z - s_z)^2} - r.$$

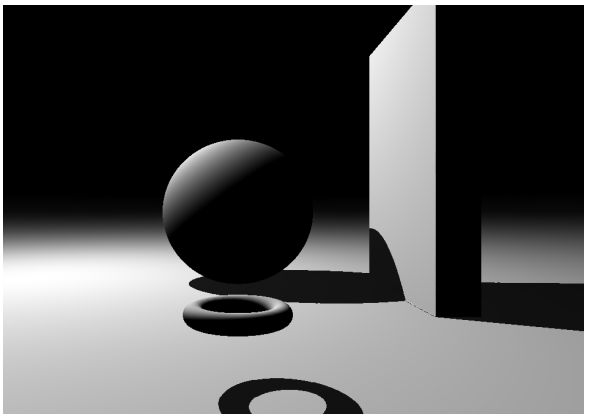
This will yield a positive value if the distance from our current point and the center of the radius is greater than the sphere's radius. If the distance is smaller, the result will be negative. For values of 0, they're equal, and we're currently on the surface of the sphere.



Next we added the torus. The torus is a bit more complicated, but builds on a similar concept as the sphere. First we'll use pythagoras to calculate the length between our current point and the center of the torus, but only in the xz-plane, subtracted by the torus' width. This will give us a circular curve, i.e. a line with no height. Then we can take the result and calculate the distance between the point and the circle in order to give it some thickness.

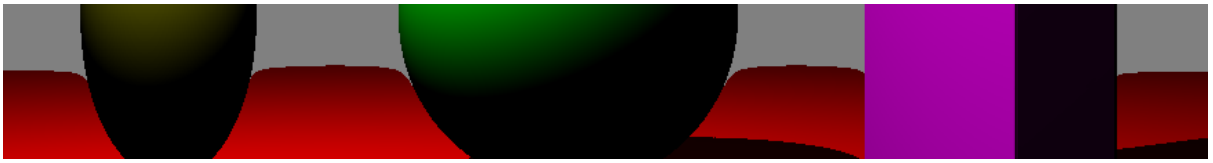
We also added a box and later an (approximated) ellipsoid.





There were a few problems with our ray marching. When a ray goes close past a primitive (without hitting it), each step will be very small. This means that we'll have to take many steps to travel a short distance. Since we have a cap on how many steps we can take, distant

points will distort since we've run out of steps and just say that we didn't hit anything. The rays between the objects will go farther and therefore hit the plane later.



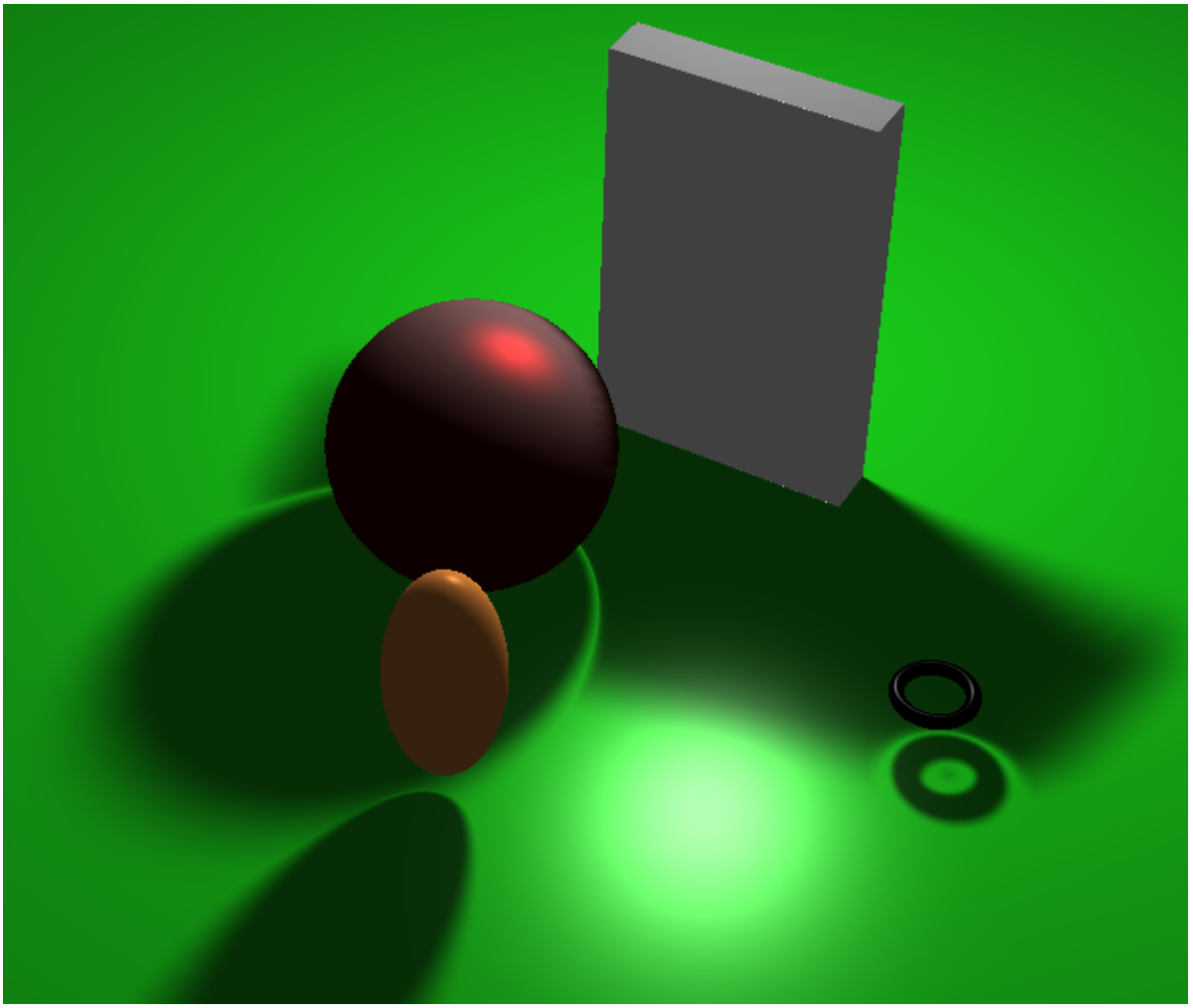
Our solution was simply to increase the maximum amount of steps the ray were allowed to take until the artifacts were less than a pixel. This, of course, made the program slower.

Lighting and shading

The shading of the project is based on Phong shading. Phong shading combines 3 basic lightning values into the final light value of our pixel, namely ambient, diffuse and specular. The ambient light is the base color value when there is no light. This is a simple approximation for global illumination for when direct light is not present. Diffuse light describes the color when the light hits the object. Depending on the angle between the surface normal and the light, it'll have different brightness. Specular light is the surface reflective properties.

For the diffuse, we multiplied it with a shadow factor to simulate shadows. We tried two different techniques, one for hard shadow and one for soft. The hard shadows were relatively straight forward. Basically, we generated a ray from the point we hit with our camera ray towards the light. If we hit the light, we didn't do anything. If we didn't hit the light, it meant that some object was in the way and should cast a shadow. Then we simply multiplied the light color with 0.1 to make it darker.

Soft shadows were a bit trickier, and came with a few artifacts. The principle is quite simple and not too far from hard shadows. Whenever we traverse our shadow ray, we made sure to check how close the ray got to hit any primitive. If the ray hit another primitive, then there was a full dark shadow. If it was far away, there would be no shadow. But if the light passed a primitive within a threshold, then it contributed to the penumbra. The closer it was, the darker it got. This worked well but gave some banding artifacts, and the intersection between the shadows was off. However, only the banding was mentioned in the article, so we believe the shadow intersection artifact is due to some bug in our code only.

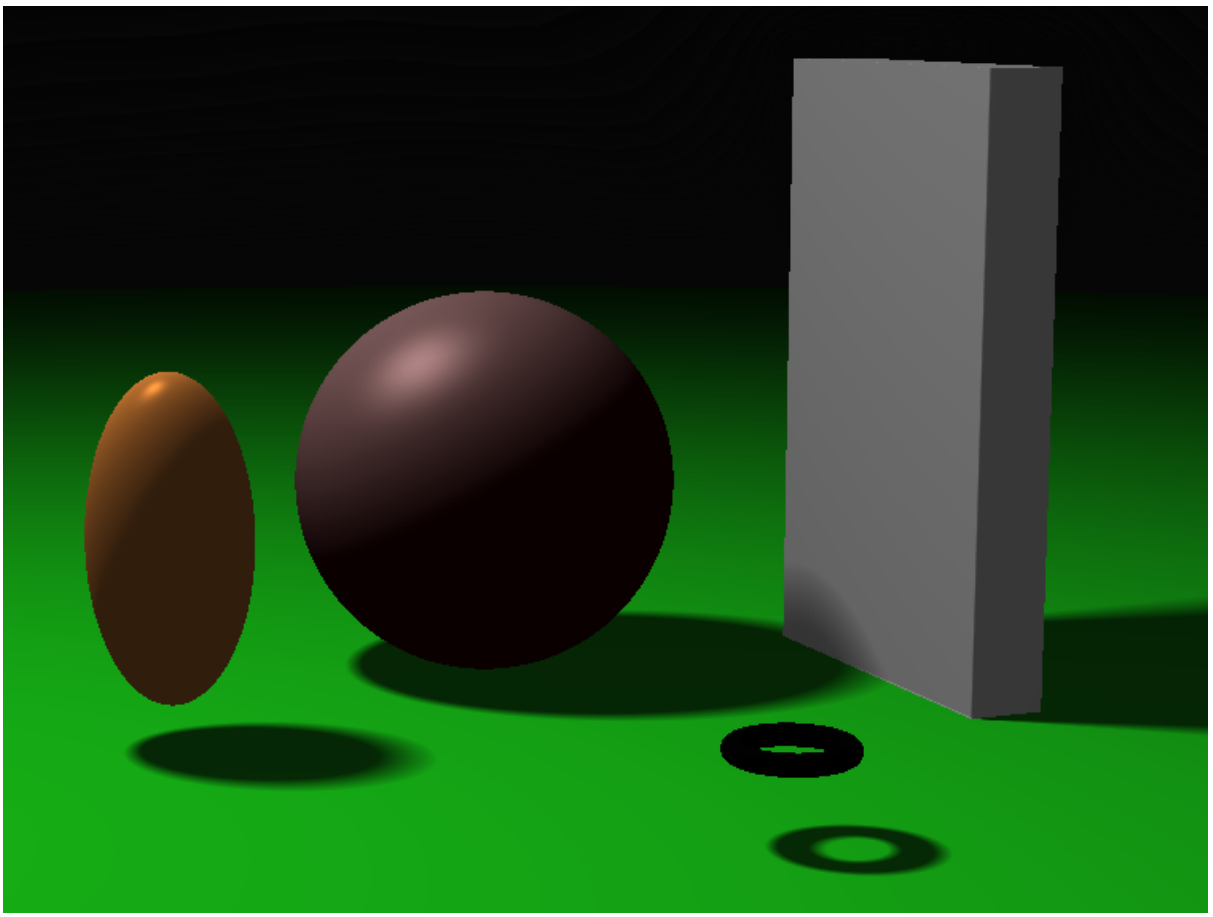


We also added a light "fog", which is just an exponential function based on the distance to each point that stays relatively close to 1 until a certain threshold and then dive towards 0. This was multiplied with the final color to make them fade out at long distances. The actual equation we used in our project was:

$e^{-0.00005 * t^3}$, where t represents the distance.

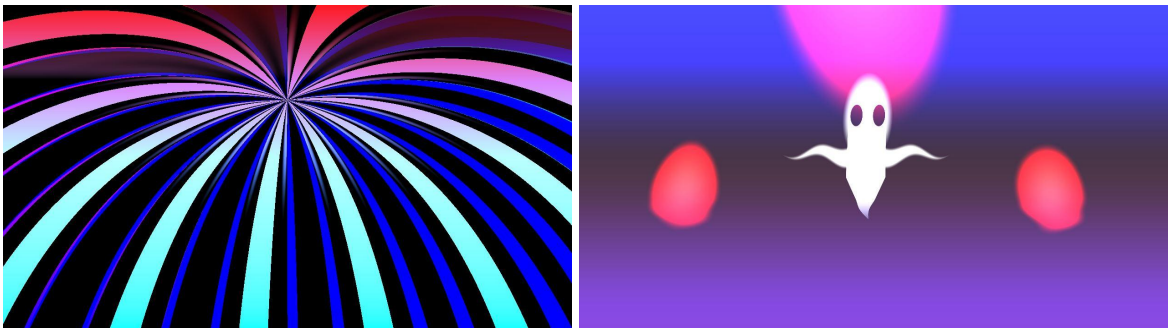
All in all, the final color was computed as

$$color = (ambient + diffuse * shadow + specular) * fog$$



Background coloring

Two background options were implemented with inspiration from two resources on shadertoy.com: https://www.shadertoy.com/view/NlSGWt , https://www.shadertoy.com/view/tdtGWr, and https://www.shadertoy.com/view/XssSRX.



Background 1 is to show a fantasy colour over the sky which changes color on its own like a firework. To create the flowering 2D shape, a combination of cosine functions was made to create the up and down combination. Then a time function was applied to make the colour changing regular and autonomous over.

Background 2 is a sky with ghost. To make the whole scenario more ghost-like we have tried to implement this. To make the ghost body a combination of geometric figures, i.e. triangles, circles were used, and to make the nubes a combination of sine functions were applied.

Distortion and displacement

The distortion functions implemented in the present project was based around sine and cosine functions which were applied to create a displaced pattern on the existing primitives in the scene. In most cases these distortion functions were applied on each dimension (x, y, z) but at times only one or two of them were considered. This, together with modifying the values in the used dimensions, gave rise to interesting outcomes of the scene presented below and also in the gallery.

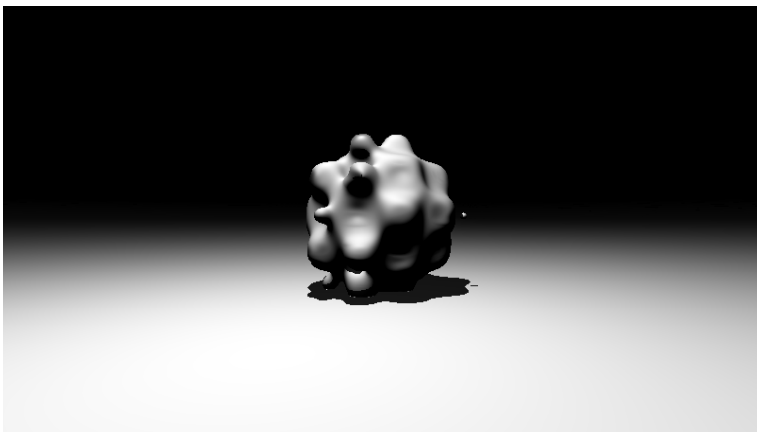
Following, the basis for the distortion function is presented:

$$float\ distortion = sin(\alpha * point.x) * cos(\alpha * point.y) * sin(\alpha * point.z) * \beta;$$

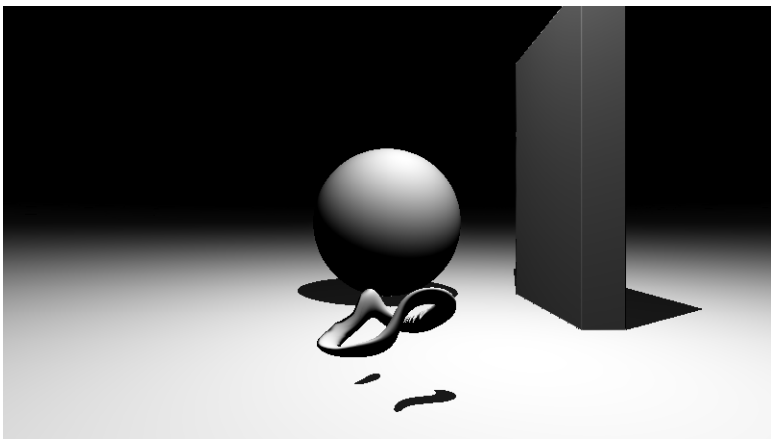
where the distortion was added to arbitrary number of dimensions (out of the three) afterwards, with the following line/s:

$$point.x = point.x + distortion;$$
$$point.y = point.y + distortion;$$
$$point.z = point.z + distortion;$$

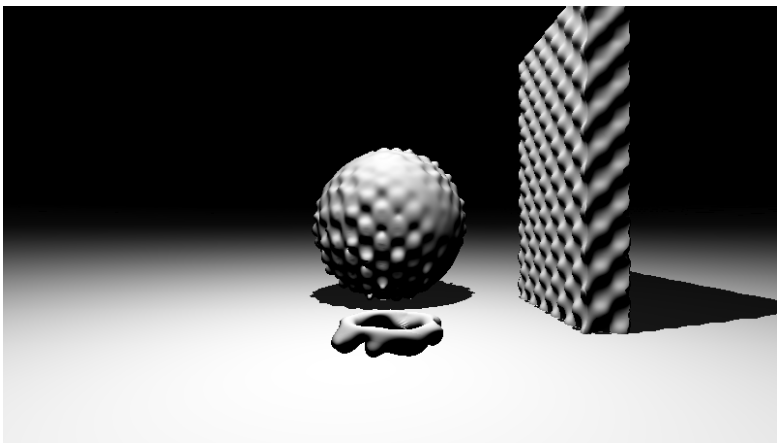
This was done to update the positions of the points. We chose the present way of doing the distortion with sine and cosine functions after finding the work by Walczyk and then taking his distortion function and slightly modifying it to adapt it to ours. Sine and cosine functions are functions that express periodic oscillation and therefore in our context, shape our primitives in a fashion that is more curvy and give rise to surfaces with spherical touches, that we strived after.



Here is the sphere distorted with values of $\alpha = 6$ and $\beta = 0.25$.

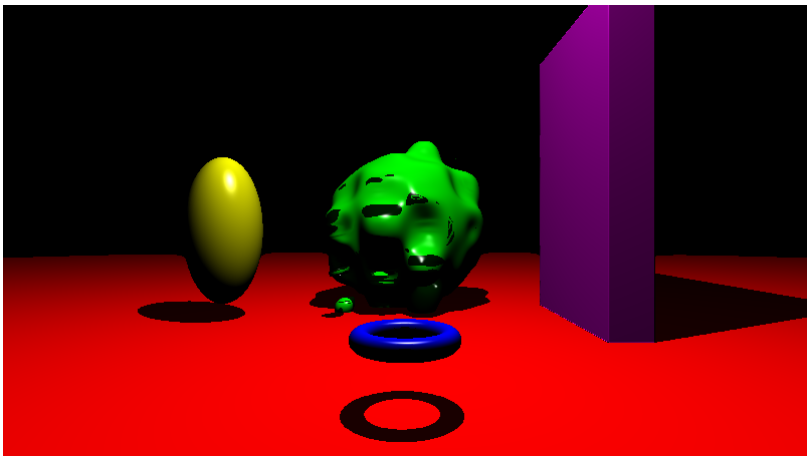


Here is the torus distorted with values of **α** = *6* and **β** = 0.25, with the box and sphere in the background.

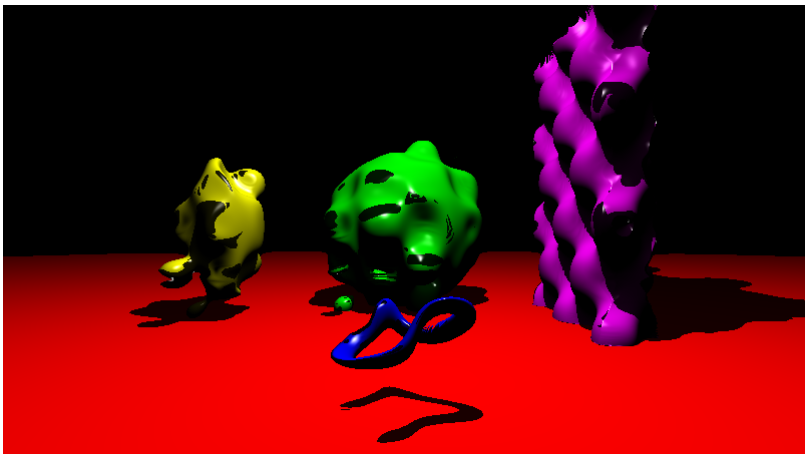


Here the sphere, torus and the box are all distorted with values of **α** = *17* and **β** = 0.05, giving the primitives another look.

Distortion after adding an ellipse, materials and color onto the primitives.



Here is the sphere distorted with values of **α** = *6* and **β** = 0.25, with the added ellips, box and torus in the scene.



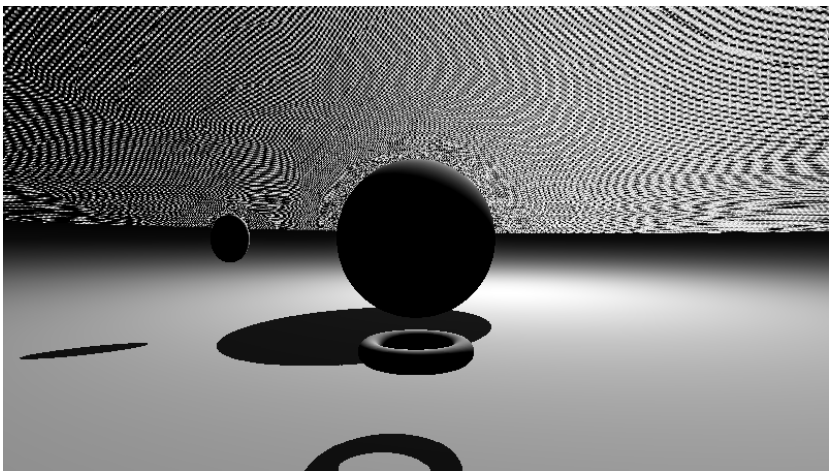
Here the sphere, torus, ellips and the box are all distorted with values of **α** = *6* and **β** = 0.25.

One more take on the distortion was done by using two remap functions that were found in the tutorial of the channel "The art of code". Those functions were taken and added into our project, one with the same values and one with modified values, see the code. The results were interesting since it once led to the primitive box being turned into a pattern in the sky, see picture below. The remap functions used looked as follows:

```
float remap01(float a, float b, float t) {
    return (t-a) / (b-a); }

float remap(float a, float b, float c, float d , float t) {
    return remap01(a, b, t)  * (d-c) + c; }
```

```
point.x = remap(-γ, γ, δ, ϵ, point.x + dist);
point.y = remap(-γ, γ, δ, ϵ, point.y + dist);
point.z = remap(-γ, γ, δ, ϵ, point.z + dist);
```



Box turned into pattern in the sky with $\alpha = 6$ and $\beta = 0.25$ and also $\gamma = 0.09$, $\delta = 0.05$ and $\epsilon = 0.15$.

One issue we discovered with our distortion function was that black spots occured on the primitives after adding distortion to them, however we didn't really manage to find out what the root of the problem was. They did not depend on the light source moving position at least, as could be seen in our project on the given link.

Discussion

Some simple improvements that could've been made are anti-aliasing and space partitioning. Anti-aliasing is as simple as just sampling the same pixel multiple times but with a slight offset. This is to reduce jagged lines and make the final render more smooth. However, doing so is very compute intensive and will reduce the performance of the shader. Space partitioning is very useful to reduce the amount of computations you have to do when querying for the closest primitive. By simply wrapping a group of primitives in a box, it becomes really easy to disregard whole groups of primitives by just checking whether the ray intersects the bounding box. Due to our simple scene with so few primitives, this would most likely not give any performance benefit at all (probably the opposite).

References

Perlin, K and Hoffert, E. M. 1989. Hypertexture. In Proceedings of the 16th annual conference on Computer graphics and interactive techniques (SIGGRAPH '89). Association for Computing Machinery, New York, NY, USA, 253–262. DOI:https://doi.org/10.1145/74333.74359

Benton, Alex. (Accessed at 20 May 2021) University of Cambridge https://www.cl.cam.ac.uk/teaching/1819/FGraphics/1.%20Ray%20Marching%20and%20Signed%20Distance%20Fields.pdf

Bálint, Csaba & Valasek, Gábor & Gergó, Lajos. (2019). Operations on Signed Distance Functions. Acta Cybernetica. 24. 10.14232/actacyb.24.1.2019.3.

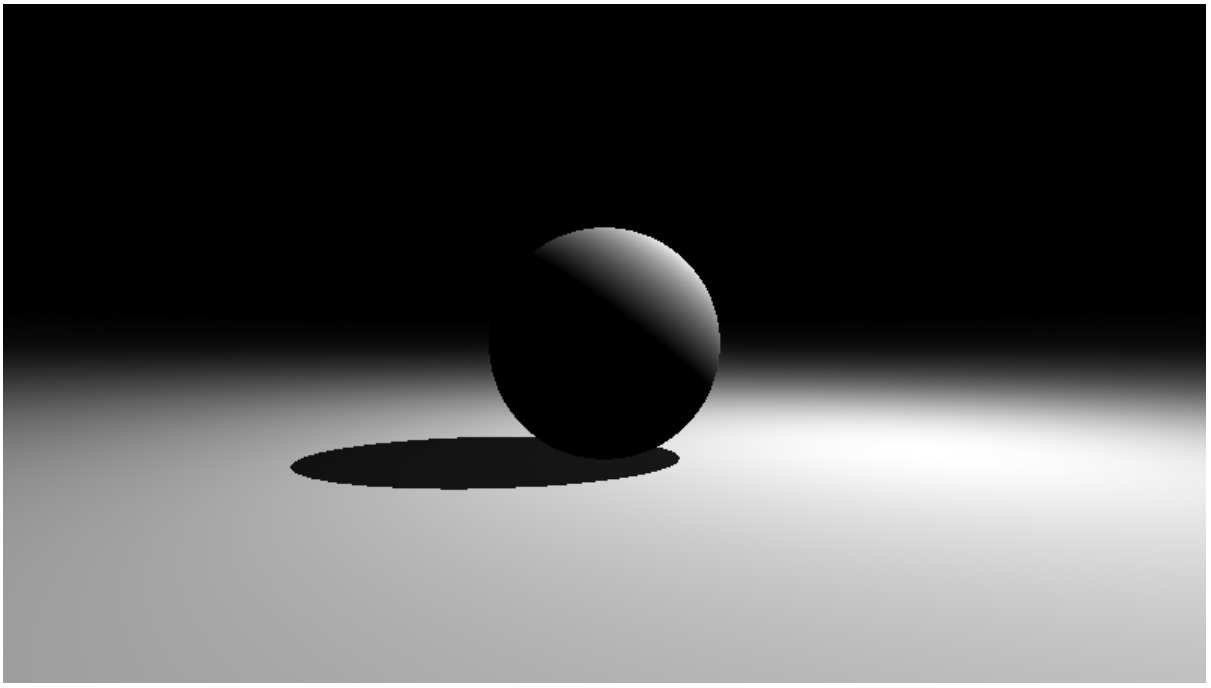
Walczyk, Michael, (Accessed at 8 June 2021)
https://michaelwalczyk.com/blog-ray-marching.html

LearnOpenGL, (Accessed at 10 June 2021)
https://learnopengl.com/Lighting/Basic-Lighting

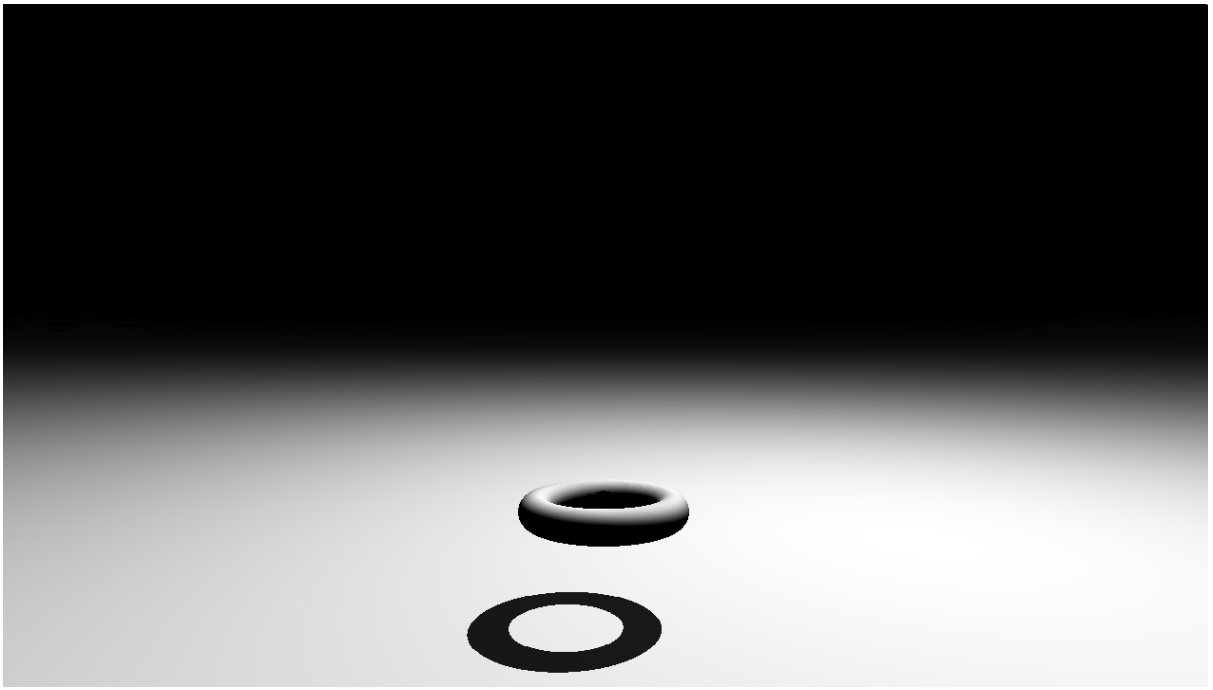
Channel the art of code, published 20 feb 2017, (Accessed at 10 June 2021)
https://www.youtube.com/watch?v=jKuXA0trQPE

Quilez, Inigo, (Accessed at 8 June 2021)
https://iquilezles.org/www/articles/rmshadows/rmshadows.htm

Gallery

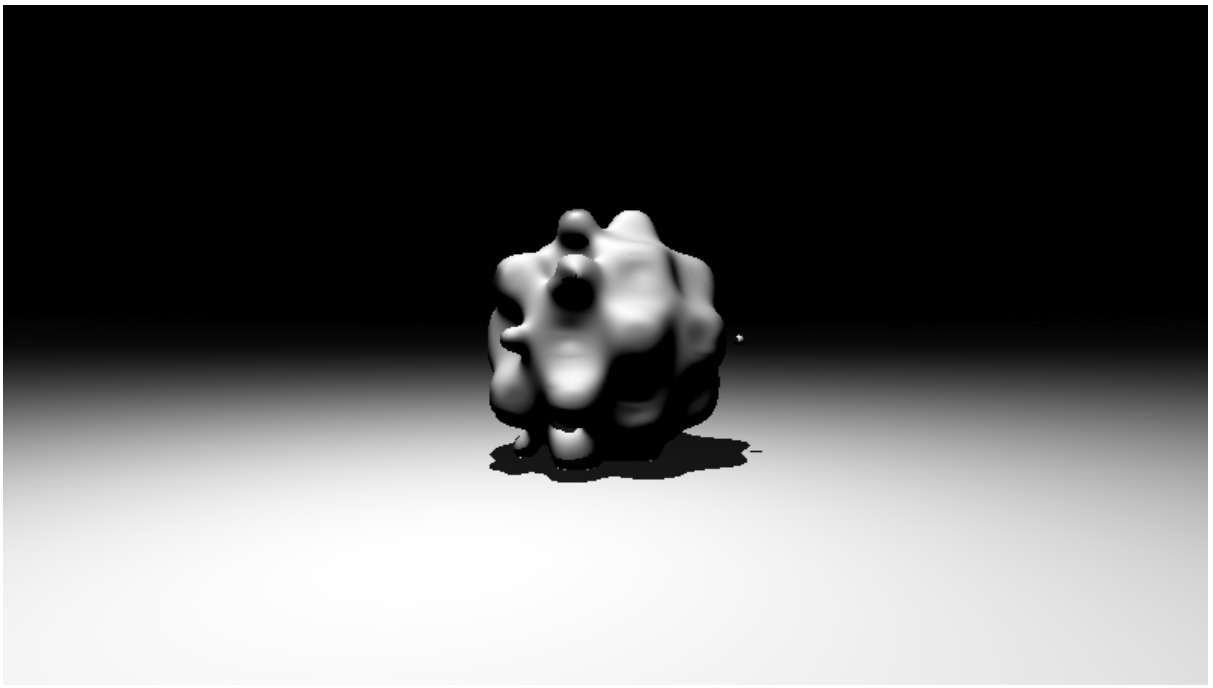


Sphere on a plane with a shadow

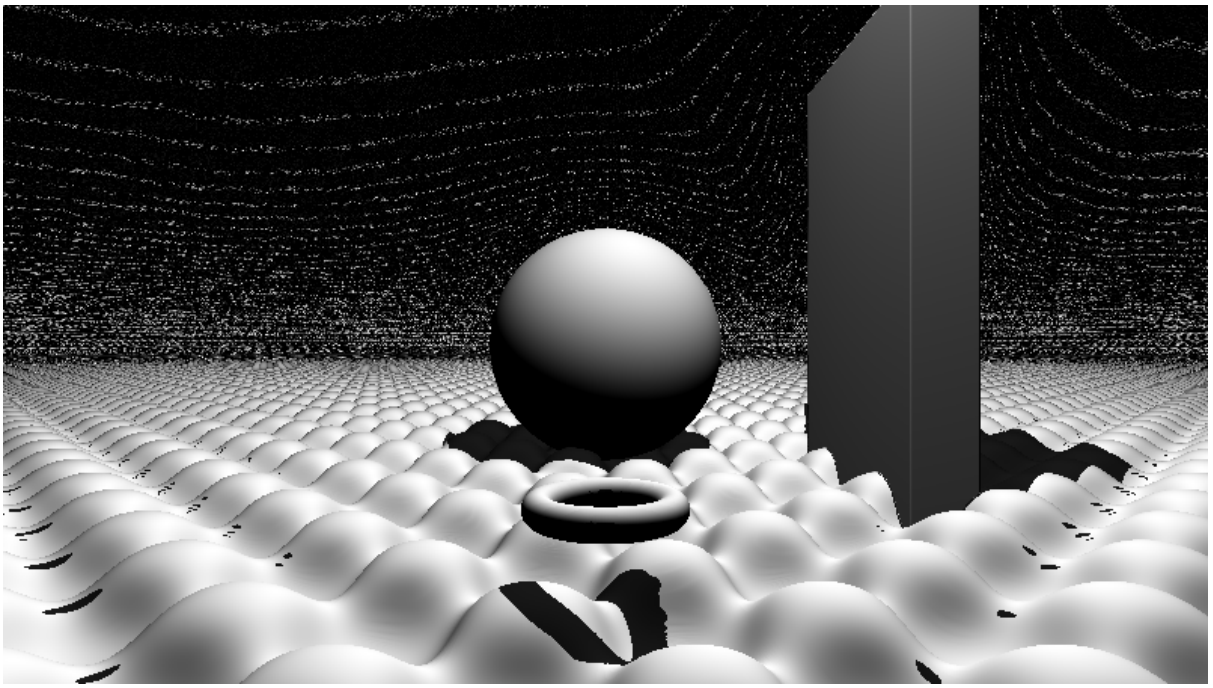


Torus on a plane with a shadow

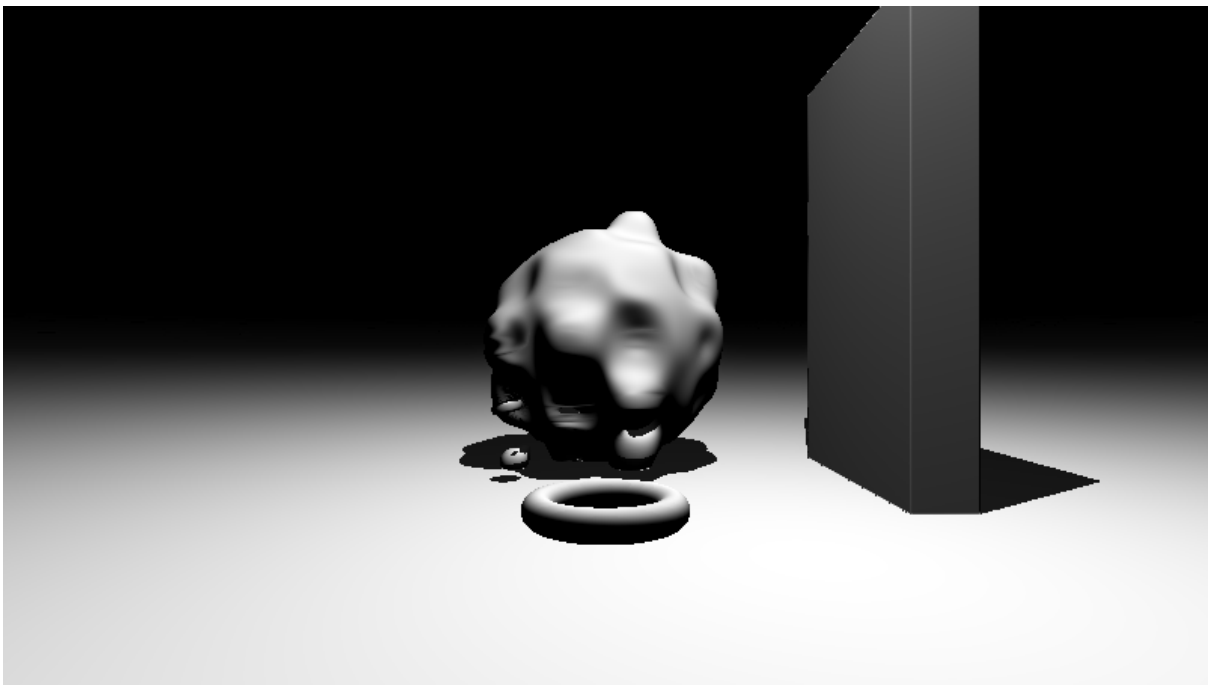
Distortion



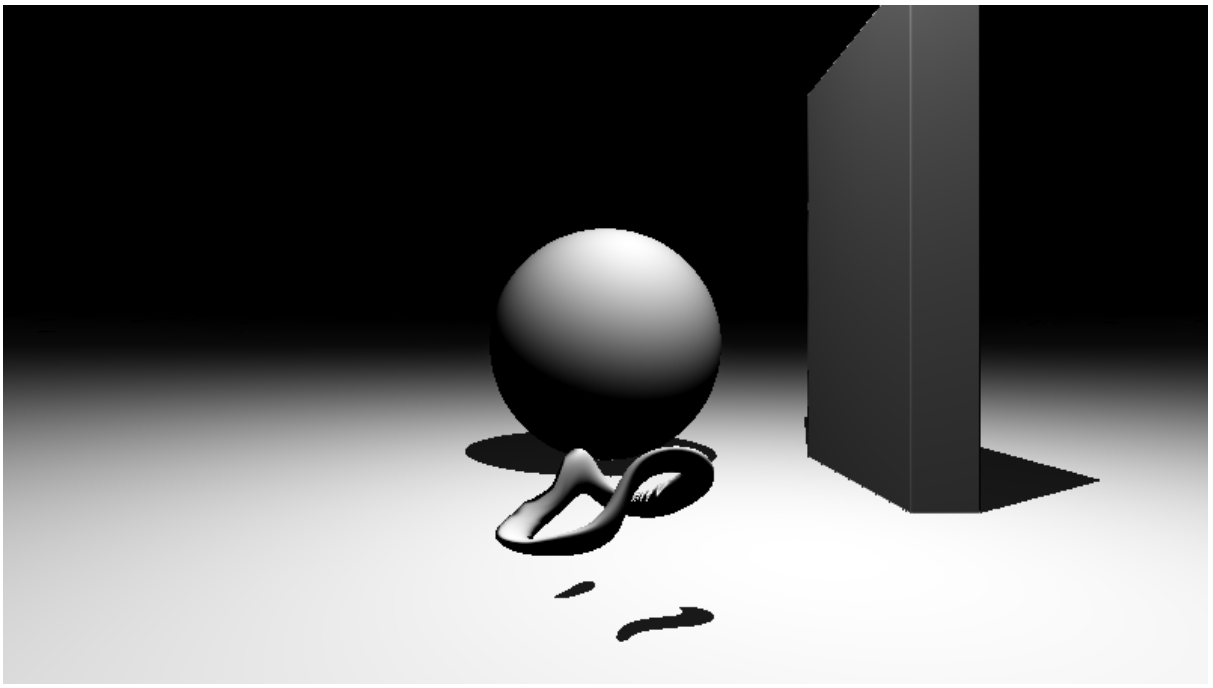
Sphere with $\alpha = 6$ and $\beta = 0.25$.



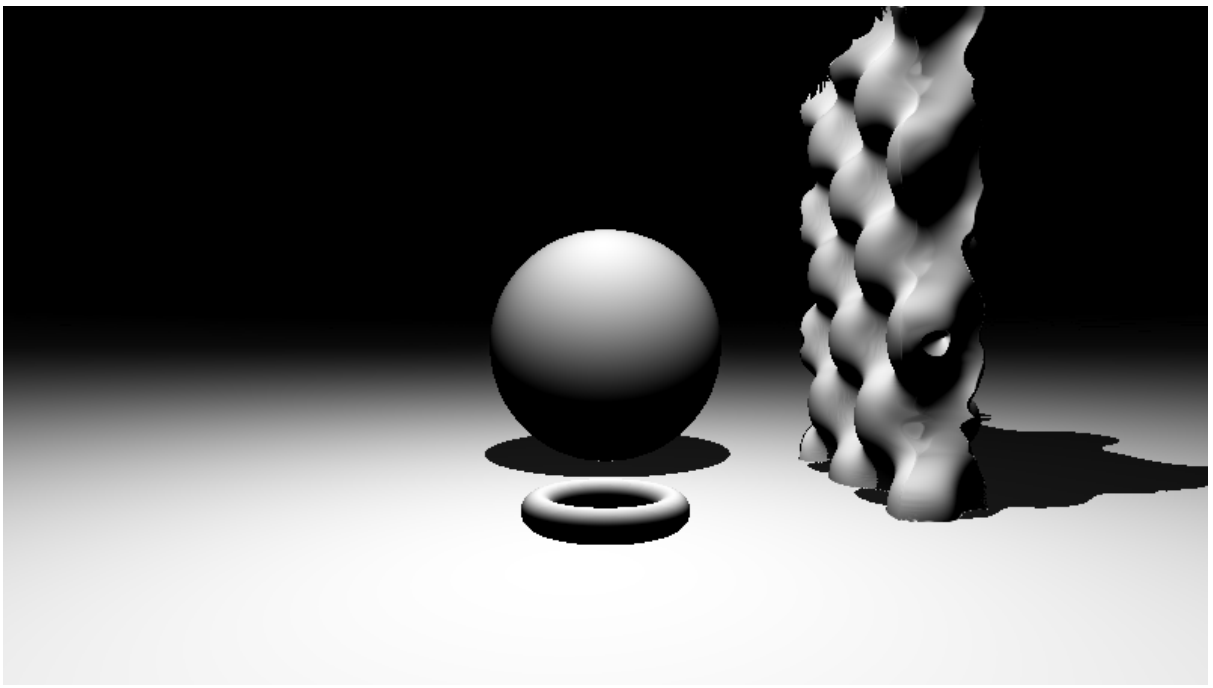
Plane with $\alpha = 6$ and $\beta = 0.25$.



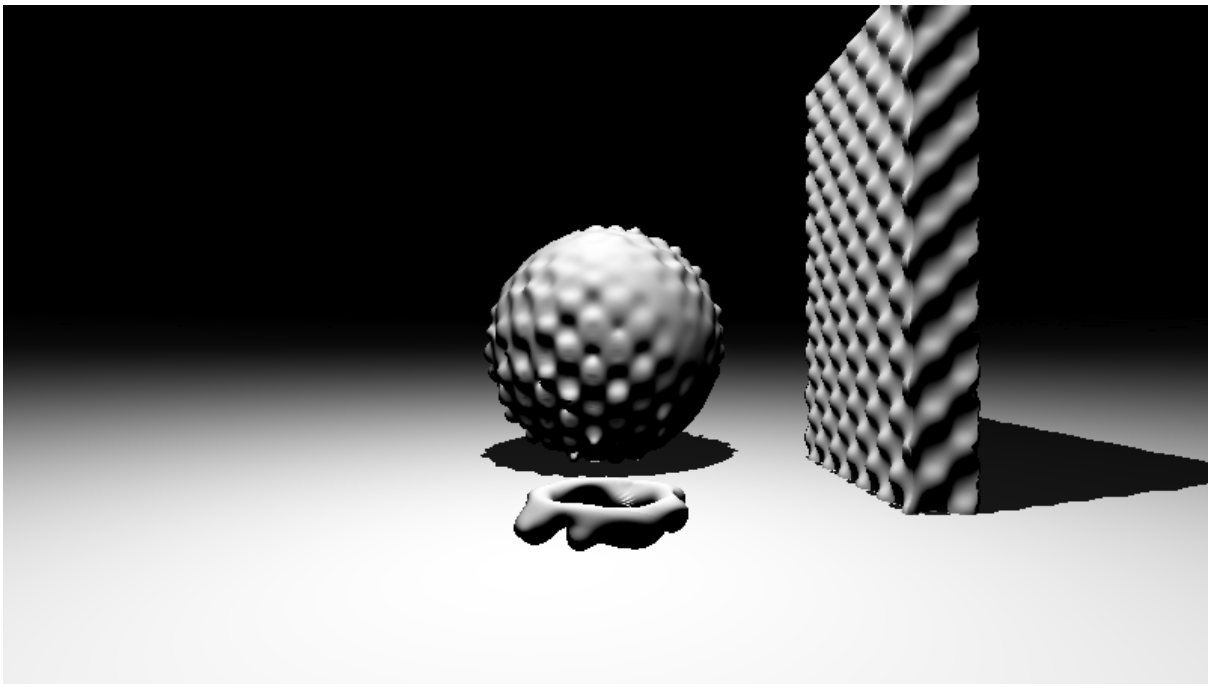
Sphere with $\alpha = 6$ and $\beta = 0.25$.



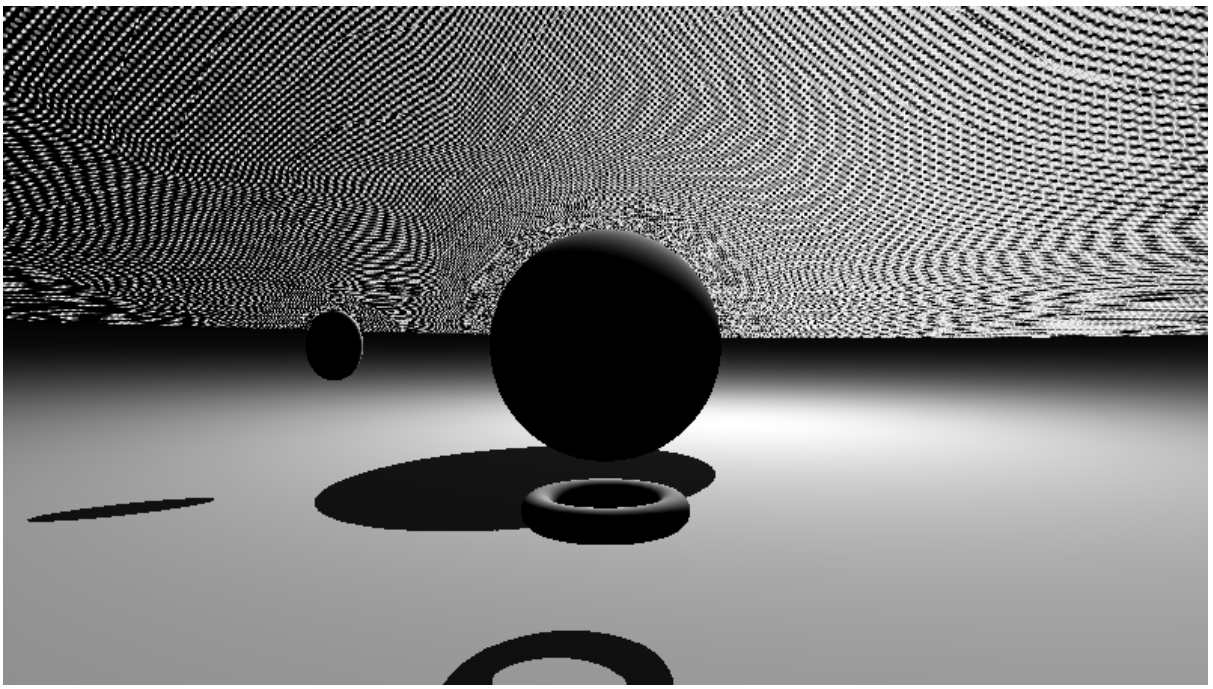
Torus with $\alpha = 6$ and $\beta = 0.25$.



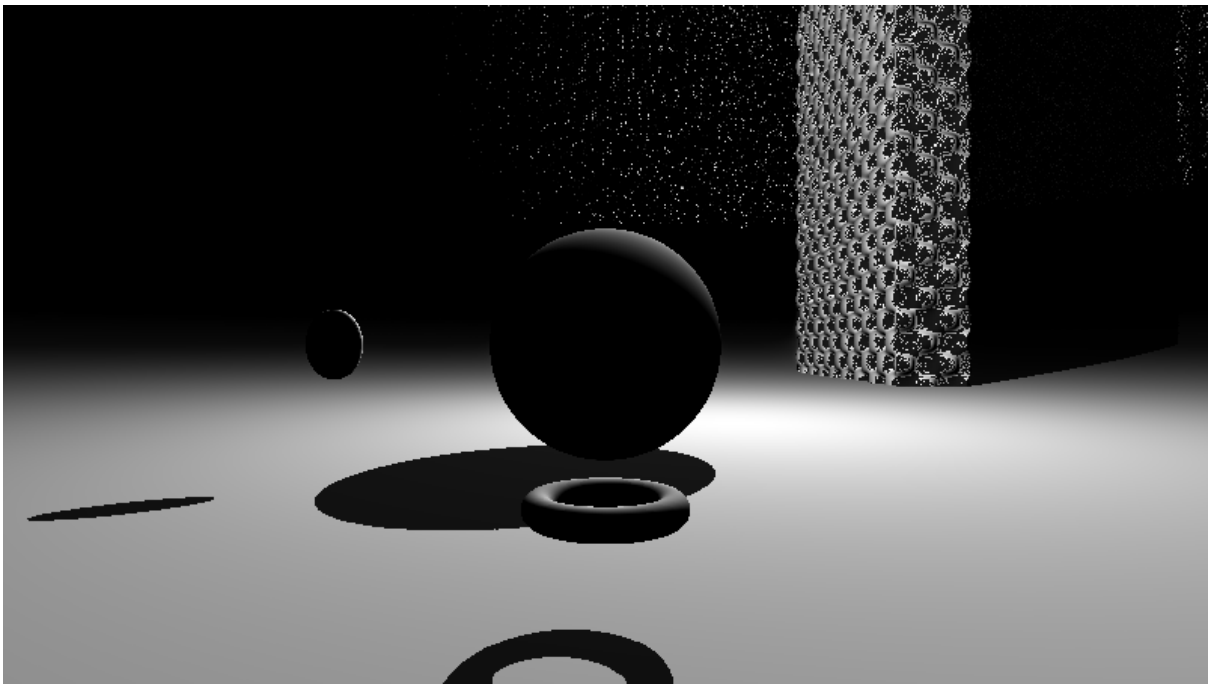
Box with **α** = *6* and **β** = 0.25.



Sphere, torus and box with **α** = *17*and **β** = 0.05.

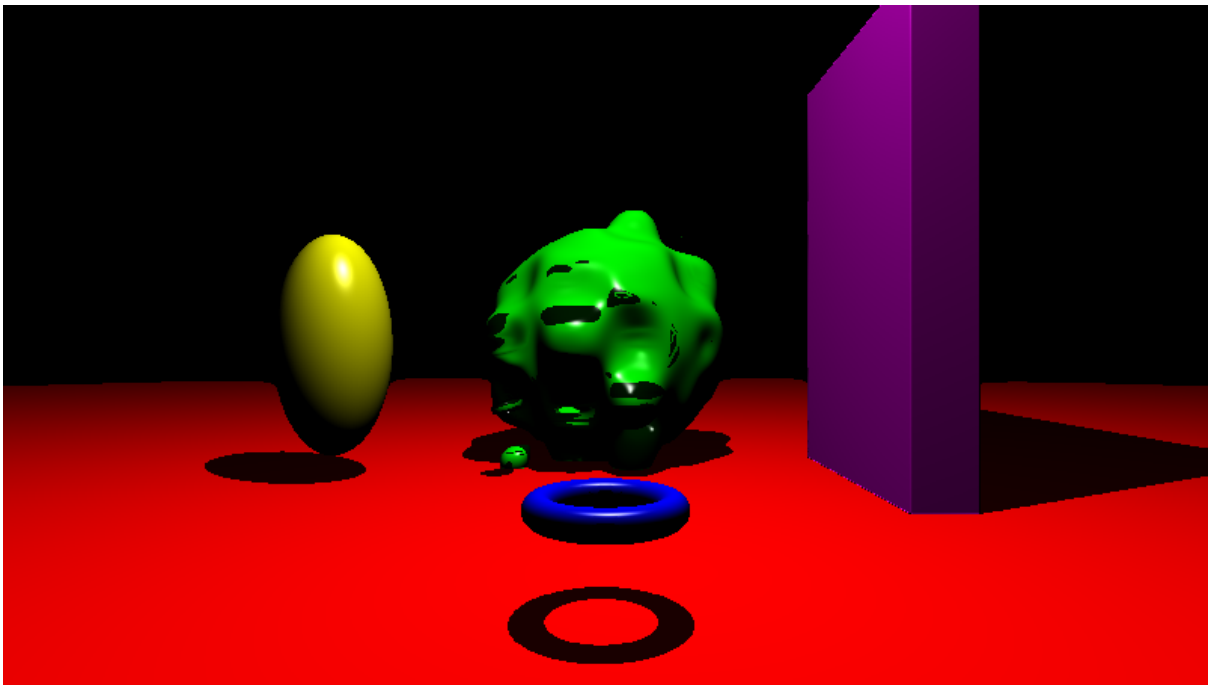


Box turned into heaven, by having $\alpha = 6$ *and* $\beta = 0.25$ *and also* $\gamma = 0.09$, $\delta = 0.05$ *and* $\epsilon = 0.15$.

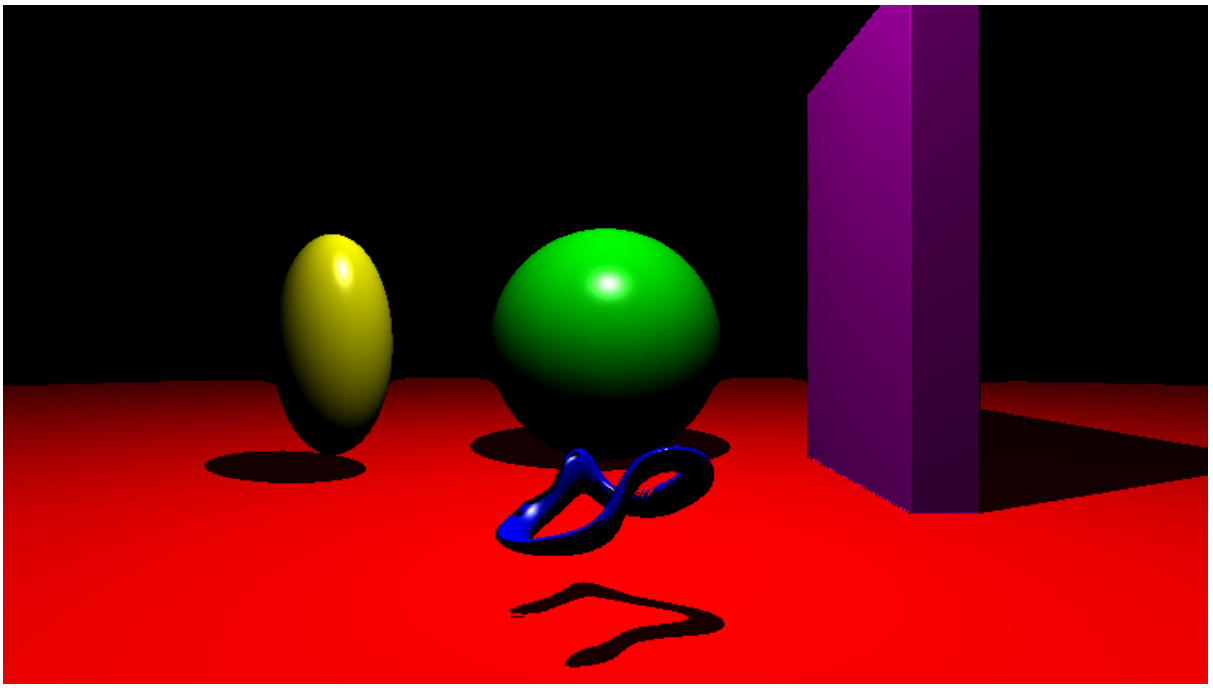


Box modified with $\alpha = 17$ *and* $\beta = 0.05$ *and also* $\gamma = 0.05$, $\delta = 0.01$ *and* $\epsilon = 0.25$.

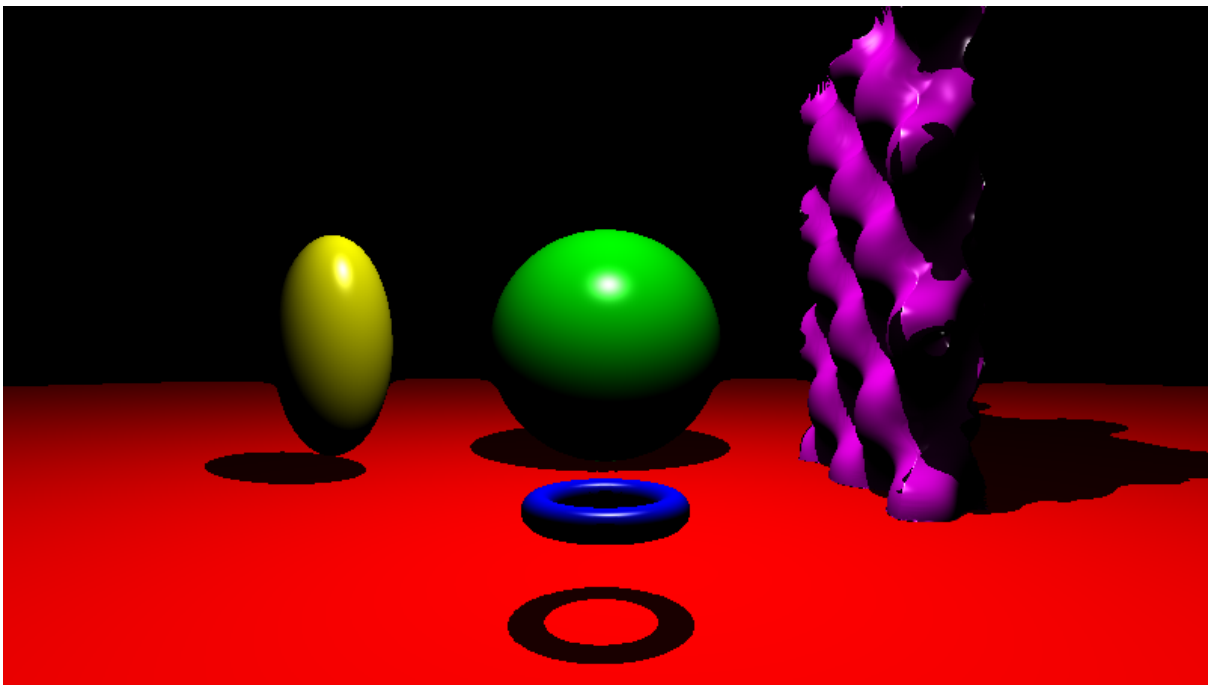
Color



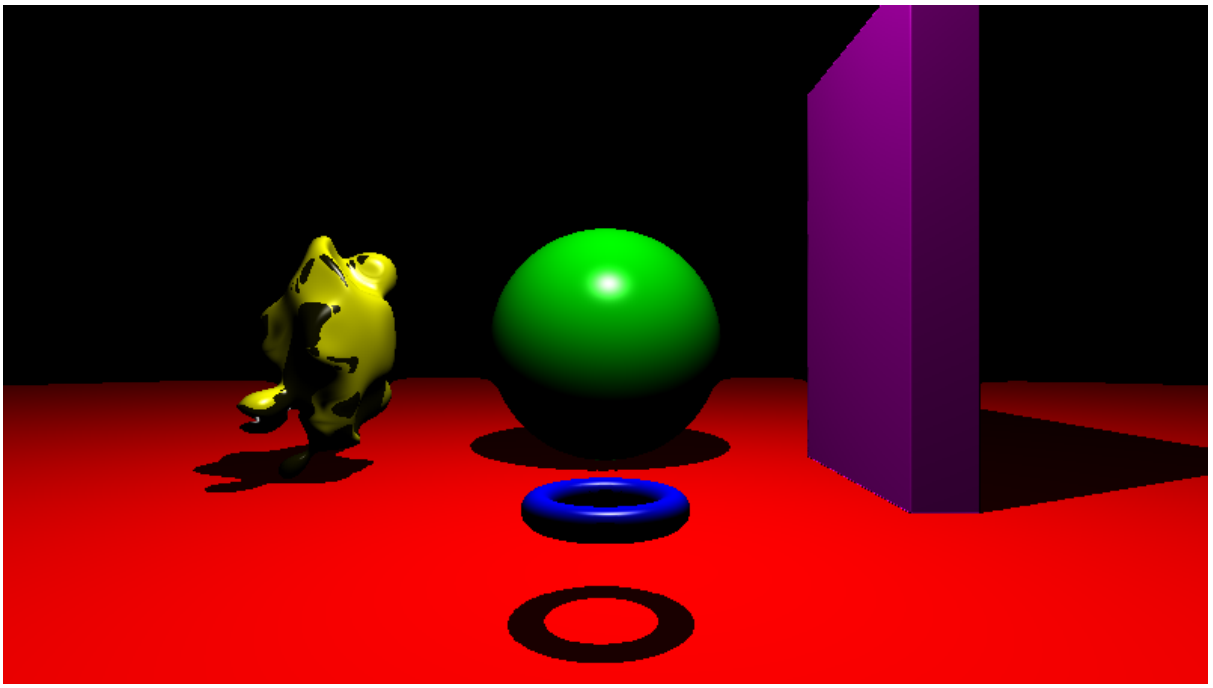
Sphere with **α** = *6* and **β** = 0.25.



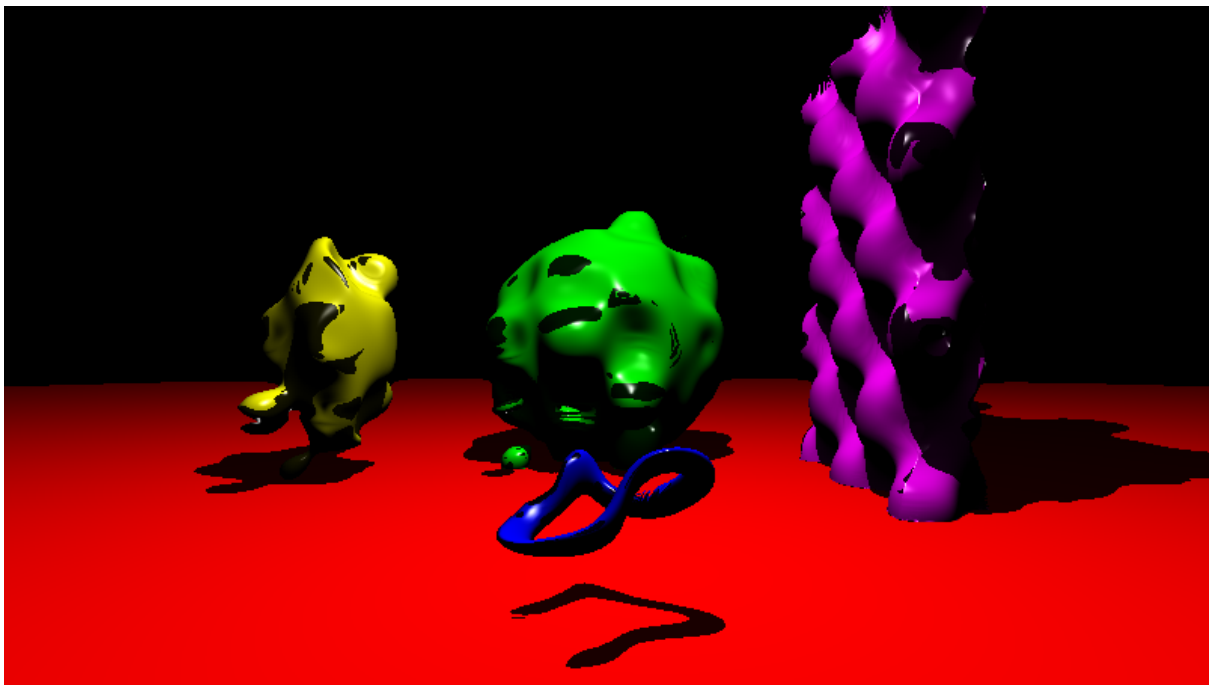
Torus with **α** = *6* and **β** = 0.25.



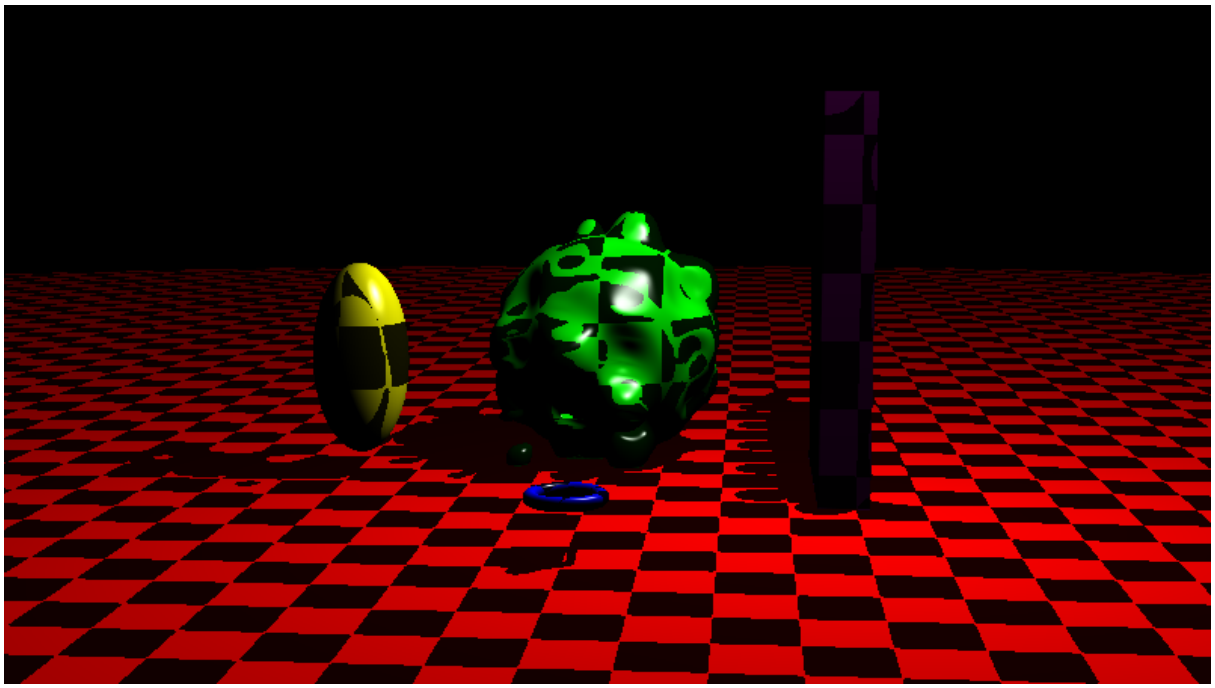
Box with **α** = *6* and **β** = 0.25.



Ellipse with **α** = *6* and **β** = 0.25.

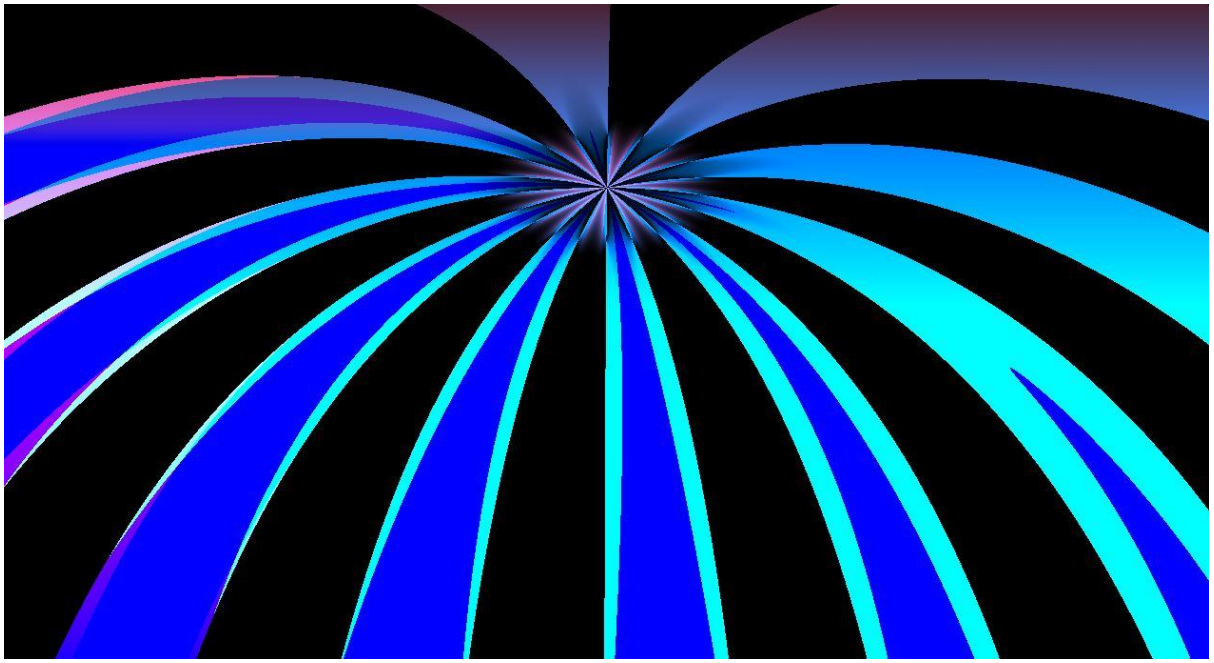


All primitives distorted with **α** = *6* and **β** = 0.25.

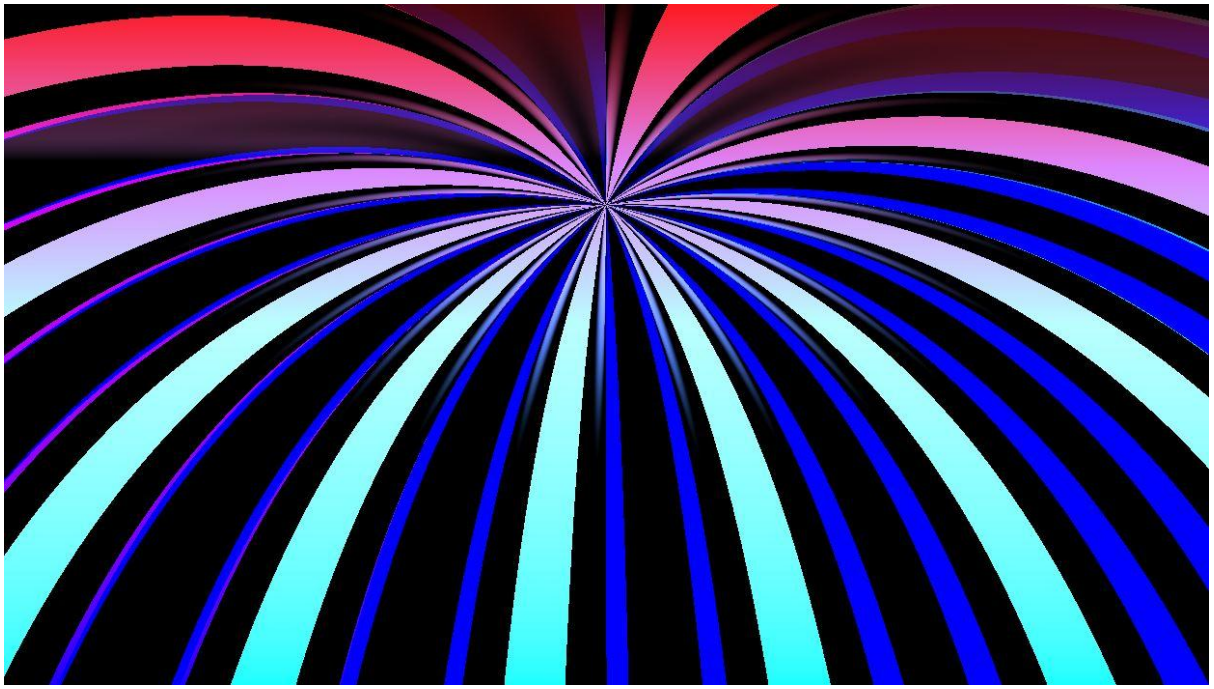


Cool picture with **α** = *6* and **β** = 0.25 and the distortion function used on the shadows in the code as well.

Background

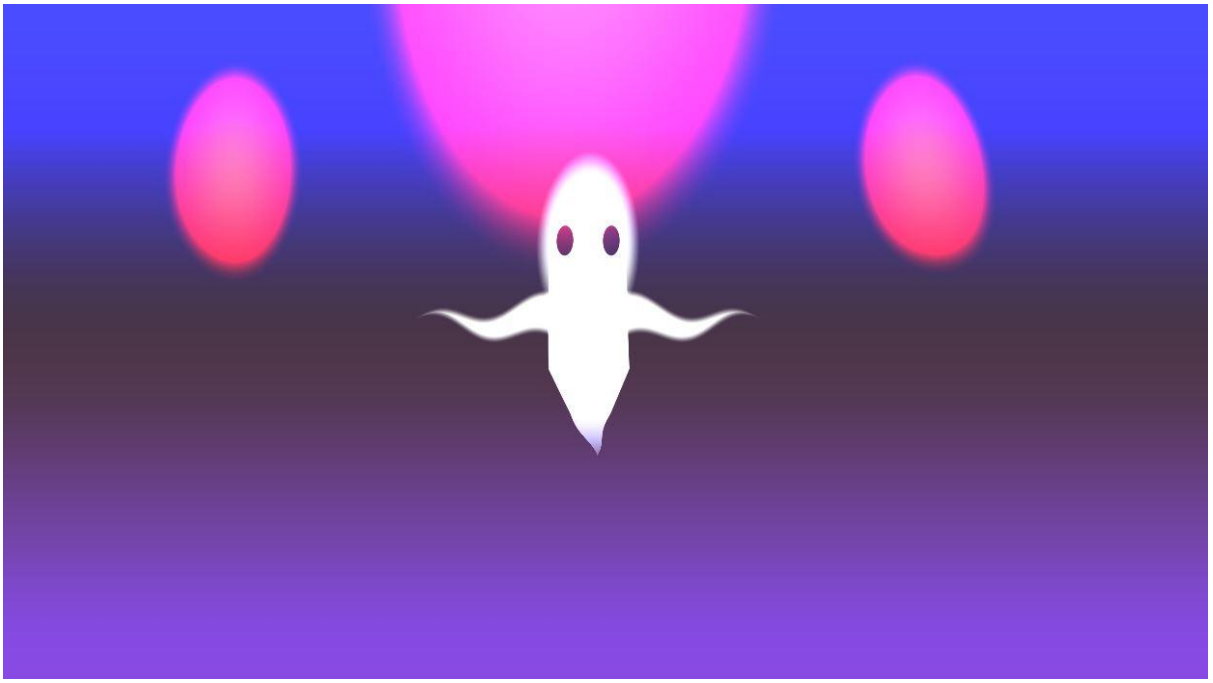


Background firework colour changing to blue

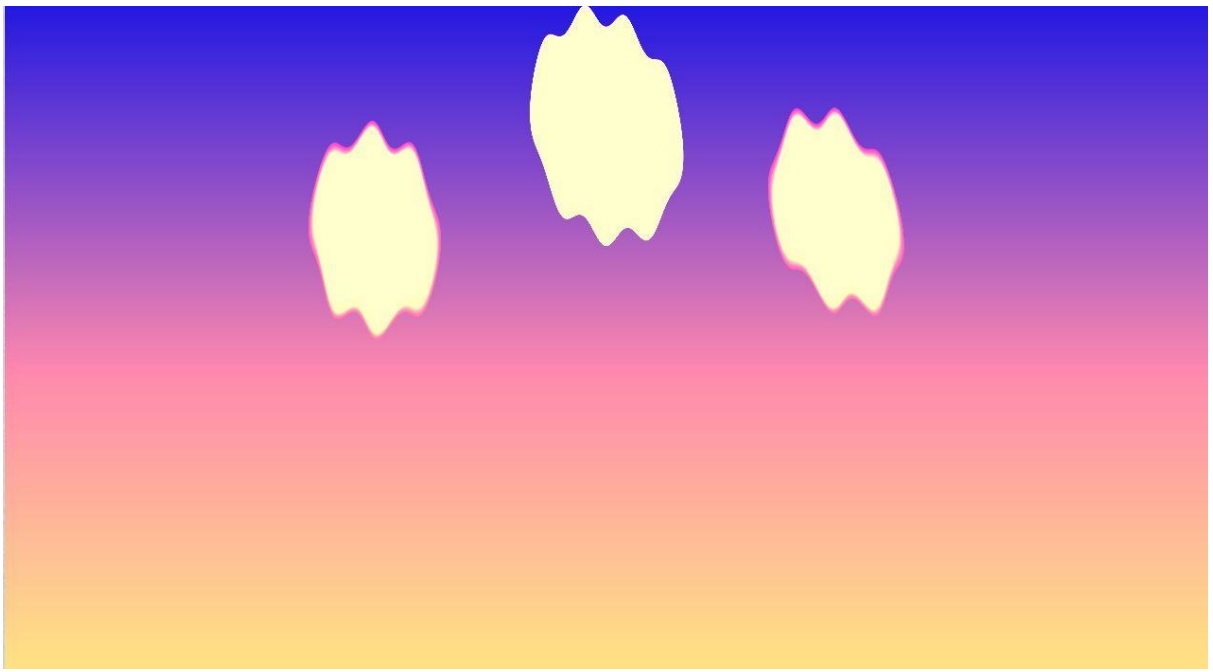


Background firework colour changing to pink

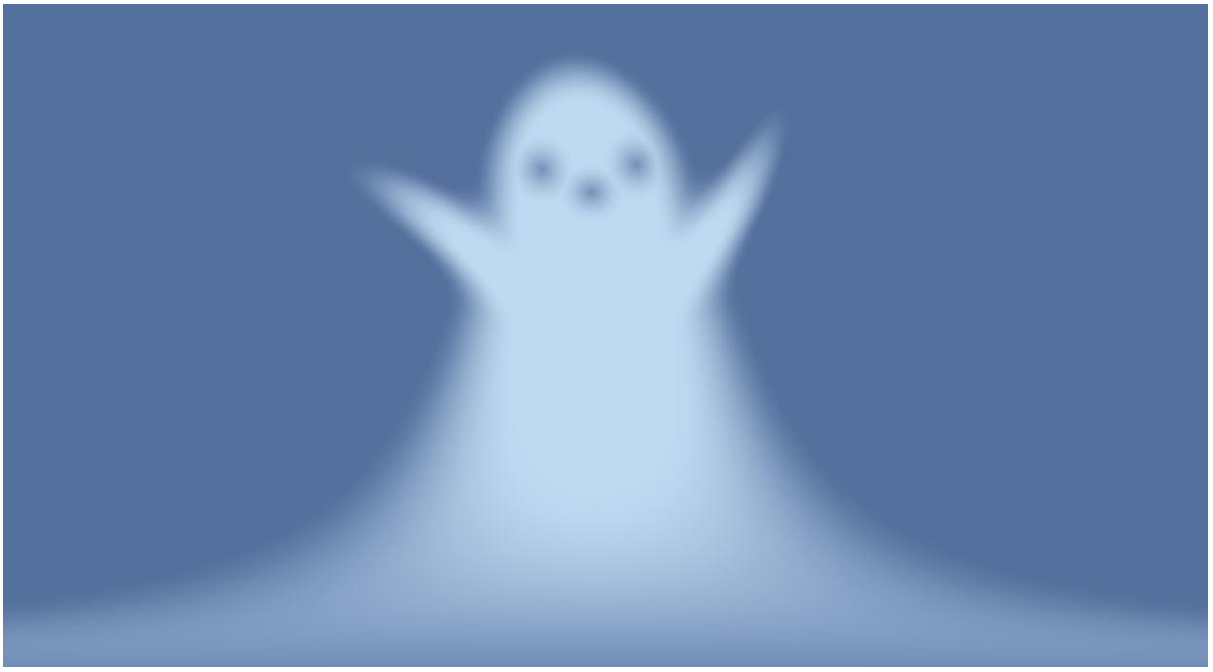
Nubes are down in the ghost sky with bigger ghost eyes



Nubes are up in the ghost sky with smaller ghost eyes



Cloud nubes distortion



ghost distortion