Ray Tracing

Project for the Mathematical Modelling course

Rok Cej, David Ocepek

University of Ljubljana, Faculty of Computer and Information Science

8.6.2018

1. Introduction

Ray tracing is a method for rendering 3D images. It works by tracing light rays from the camera through space and simulating their collisions with virtual objects. That includes finding out whether the given collision point is illuminated and then calculating the color of that point based on the properties of the object's material – diffusion, reflection and transparency (which includes refraction of light rays). This method is capable of producing highly realistic images, however, it is not suitable for real-time rendering as it is quite slow. We are using C++ to implement this method.

2. Creating the light rays

First off, we need to model a virtual camera that will serve as a light ray source. We want to send a light ray through each pixel on the screen, so we can get the color of each pixel. To do that, we have to map each pixel onto an imaginary canvas in front of the camera. The canvas is 1 unit of distance away from the camera and its size is determined by a parameter called field of view (FOV). The span of the canvas in each direction is therefore obtained by:

$$span = 2 * \tan(\pi * \frac{FOV}{360})$$

We also want our screen to be bound by the canvas, so that regardless of the aspect ratio of our screen (the ratio between the width and the height), all pixels will be mapped within the canvas. For that, we need to define a ratio:

$$ratio = span/\max(width, height)$$

If we imagine the canvas as an x and y coordinate plane, then the coordinates of the center of a pixel in the row row and in the column col are:

$$x = ratio * \big((col + 0.5) - width * 0.5\big)$$

$$y = ratio * (-(row + 0.5) + height * 0.5)$$

To fully model the camera we need 3 unit vectors: v_{dir}, v_x, v_y and a position vector v_{pos}. v_{dir} is the direction in which the camera is facing, v_x is the direction of the x axis of the canvas and v_y is the direction of the y axis of the canvas. Once we have all that, the direction of a light ray through the given pixel is:

$$r_{dir} = normalize(x * v_x + y * v_y + v_{dir})$$

The origin of the light ray r_{origin} is the same as the position of the camera v_{pos}:

$$r_{origin} = v_{pos}$$

3. Creating objects

Our objects are represented by functions of the form $f(x, y, z) = 0$. These functions take a point in 3D space as a parameter. If the point lies on the surface of that object the return value is 0, otherwise the return value can be either a positive or a negative number. The sign of the function value tells us whether the point is inside or outside of the object (or simply on which side of the object the point is located in case of infinite objects such as planes). That means if we have 2 points with different function value signs, the surface of the object must lie somewhere on the line between those 2 points.

Each object also has a material, which determines the color of the object and how the object responds to light. Each material has 4 attributes – color, reflection, transparency and the refractive index. Color tells us what the material looks like when lit by a light source, reflection

tells us how much light bounces off the surface of the object, transparency tells us how much light goes through the object and the refractive index tells us how light passes through the object.

4. Tracing the light rays

4.1. Moving the ray through space

Once we have the origin and the direction of a light ray, we move the ray through space by small steps, checking function value signs for every object in each step. The point in each step is acquired by:

$$point = r_{origin} + step * stepSize * r_{dir}$$

Once a function value sign for a certain object changes, we know that the collision of the light ray with the object must be somewhere between the current and the last step. To find the collision point, we need to find the zero of the function:

$$g(t) = f\left(r_{origin} + t * r_{dir}\right) = 0$$

4.2. Finding the collision point

We can get a good approximation of the collision point using Newton's method. We use the middle point between the current and the previous step as the initial guess:

$$t_0 = (step - 0.5) * stepSize$$

Then we can iteratively use the equation:

$$t_{n+1} = t_n - g(t_n)/g'(t_n)$$

until we've reached a predetermined level of accuracy or surpassed a predetermined maximum number of iterations (which is used in the rare case that the iteration doesn't converge).

4.3. Simulating the collision

Now that we have the collision point, we have to take a look at the properties of the object's material, based on which we have to create up to 3 rays – the shadow ray, the reflection ray and the refraction ray.

4.3.1. Shadow ray

We need the shadow ray to find out if and how the collision point is illuminated, that's why a shadow ray is created for each light in the scene. The direction of a shadow ray is calculated by taking the difference between the position of the given light and the current point, then normalizing it:

$$shadow_{dir} = normalize(light_{position} - point)$$

Similarly to the light ray, we track the shadow ray towards the light source by small steps, checking for collisions with objects in each step. If the shadow ray collides with an object, that means the current point lies in the shadow of the given light source. If the object is transparent, we can simply weaken the strength of the light source:

$$intensity = intensity * transparency$$

If the shadow ray reaches the light source, it means that light source is illuminating the current point. We also have to take into account the angle at which the shadow ray hits the surface of our object. The smaller the angle φ between the shadow ray and the normal of the object in that point is, the more illuminated the point will be. The cosine of the angle can be calculated using the dot product of both vectors (they're both unit vectors so we don't have to normalize them):

$$\cos(\varphi) = shadow_{dir} \cdot normal$$

The final illumination is then calculated by:

$$illumination = \cos(\varphi) * brightness * intensity$$

where brightness is a property of each light source. Once we have the illumination, we can calculate the visible color of the object in the current point by:

$$visibleColor = color * illumination$$

where $color$ is a property of the current object's material.

4.3.2. Reflection ray

If the current object is reflective, we need to create a reflection ray. The direction of the reflection ray is calculated by:

$$reflection_{dir} = r_{dir} - 2 * normal * (r_{dir} \cdot normal)$$

A light ray is then recursively simulated in the direction of the reflection ray. Once the reflection ray color is returned, the visible reflection color is calculated by:

$$visibleReflectionColor = reflection * reflectionColor$$

where $reflectionColor$ is the color returned by the recursive call and $reflection$ is a property of the current object's material.

4.3.3. Refraction ray

If the current object is transparent, we need to create a refraction ray. For that we need the law of refraction, which states:

$$\frac{\sin(\varphi_1)}{\sin(\varphi_2)} = \frac{n_2}{n_1}$$

where φ_1 is the angle of incidence (the angle between the light ray and the normal), φ_2 is the angle of refraction (the angle between the refraction ray and the normal), n_1 is the refractive index of the material where the light ray is coming from and n_2 is the refractive index of the material where the light ray is traveling to. It is assumed, that the "air" in this renderer has a refractive index $n = 1$.

First, we need to calculate the cosine and sine of the angle of incidence:

$$\cos(\varphi_1) = -r_{dir} \cdot normal$$

$$\sin(\varphi_1) = \sqrt{1 - \cos(\varphi_1)^2}$$

Depending on the angle of incidence, two things can happen.

4.3.3.1. Total internal reflection

If $\sin(\varphi_1) > \frac{n_2}{n_1}$, then we have total internal reflection, which means that the light ray doesn't travel through the object as a refraction ray, but it actually bounces like a reflection ray, which we have already described. In this case, we can simply reuse the reflection ray direction.

$$refraction_{dir} = reflection_{dir}$$

4.3.3.2. Refraction

If $\sin(\varphi_1) \leq \frac{n_2}{n_1}$, then we have regular refraction. We can calculate the cosine and sine of the angle of refraction by:

$$\sin(\varphi_2) = \sin(\varphi_1) * \frac{n_2}{n_1}$$

$$\cos(\varphi_2) = \sqrt{1 - \sin(\varphi_2)^2}$$

Once we have those, the refraction ray direction is calculated by:

$$refraction_{dir} = \left(r_{dir} + normal * \cos(\varphi_1)\right) * \frac{n_1}{n_2} - normal * \cos(\varphi_2)$$

Just like with reflection, a light ray is recursively simulated in the direction of the refraction ray. Once the refraction ray color is returned, the visible refraction color can be calculated by:

$$visibleRefractionColor = transparency * refractionColor$$

where $refractionColor$ is the color returned by the recursive call and $transparency$ is a property of the current object's material.

4.3.4. Putting it all togeter

To mix all three types of rays together, we used the following equation:

$$pointColor = (visibleColor * (1 - reflection) + visibleReflectionColor) * (1 - transparency) + visibleRefractionColor$$

which is then used to directly determine the color of the given pixel at the lowest level of recursion.

5. Multisampling

If we try to render an image with what we have, this is the result:

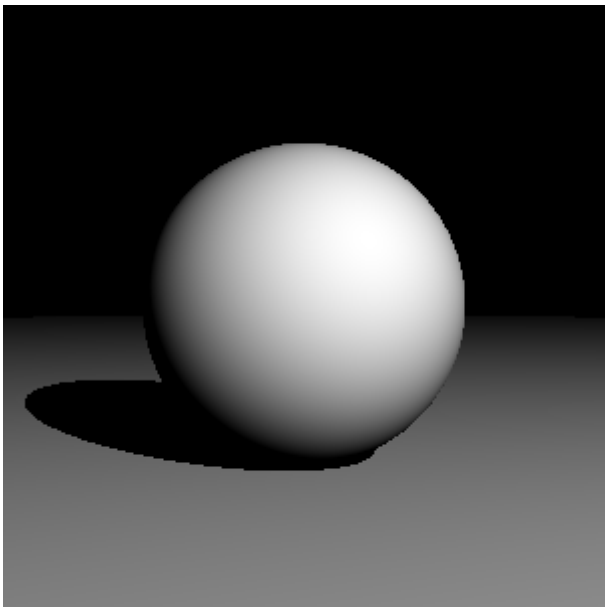


Figure 1: 1 sample per pixel

The edges look really rough, because we're only sending one light ray per pixel. If we want our images to look smoother, we can resort to multisampling – sending multiple light rays per each pixel, and then calculating the average color. If we increase it to 16 samples per pixel, we can immediately see the difference:

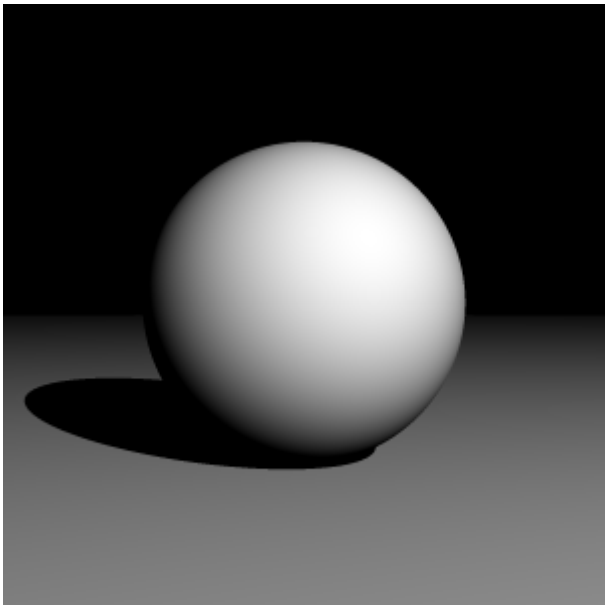


Figure 2: 16 (4x4) samples per pixel

Why stop at 16? Let's try 256 samples per pixel:

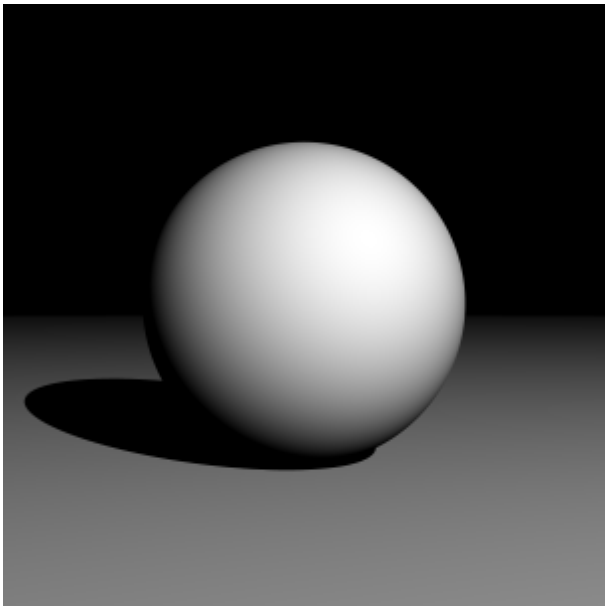


Figure 3: 256 (16x16) samples per pixel

As you can see, it doesn't make much of a difference anymore. While the difference between 1 and 16 samples is colossal, we can't see any difference between 16 and 256. That's why it's usually best to stick with 4, 9 or 16 samples. After all, multisampling massively increases render time. If you have n times more samples, that means it will take n times longer to render the image.

6. Multithreading

As was mentioned before, the ray tracing method is slow, that's why all optimizations are extremely important. One such optimization is multithreading. It's extremely suitable for ray tracing, because pixels are independent from one another, therefore rendering can be parallelized. It works by splitting the program into multiple threads, each responsible for

rendering only a part of the whole screen. In this case, we split the screen into groups of rows and created a thread for each group that renders it. After all threads are done and synchronized, the image is saved. This was achieved by using the built-in C++ multithreading support in the <thread> header. It is recommended that the number of threads is set to the number of cores on your CPU. On an i7 4770k CPU parallelizing the program resulted in approximately 4-5x faster render times.

7. Examples

7.1. Basic objects

This program can render any function of the form $f(x, y, z) = 0$ that has all 3 partial derivatives. Here are some examples:

7.1.1. Plane

$$(x - x_0)a + (y - y_0)b + (z - z_0)c = 0$$

Figure 4: A 3D plane

7.1.2. Sphere

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0$$

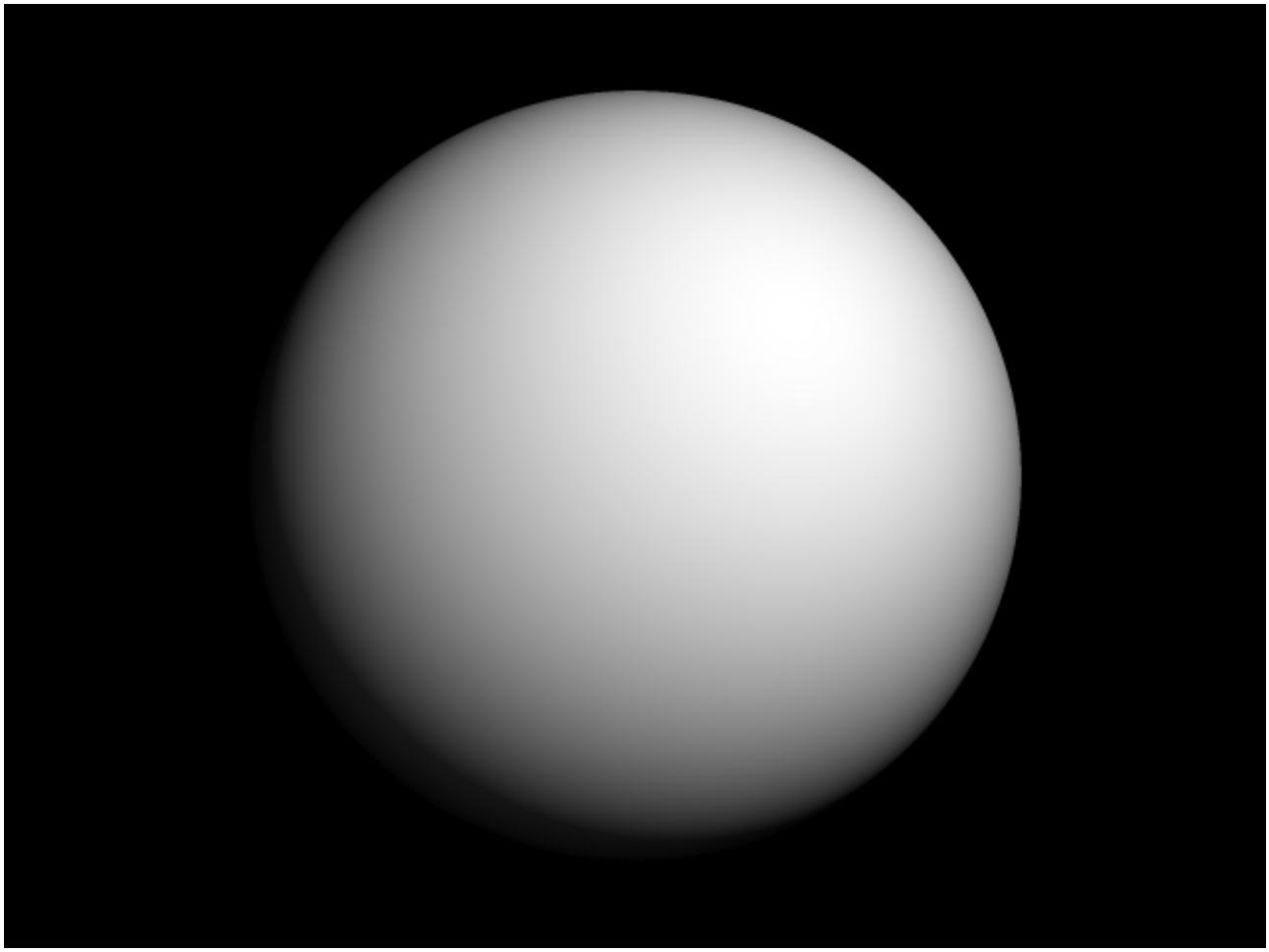


Figure 5: A 3D sphere

7.1.3. Ellipsoid

$$\frac{(x-x_0)^2}{a^2}+\frac{(y-y_0)^2}{b^2}+\frac{(z-z_0)^2}{c^2}-1=0$$

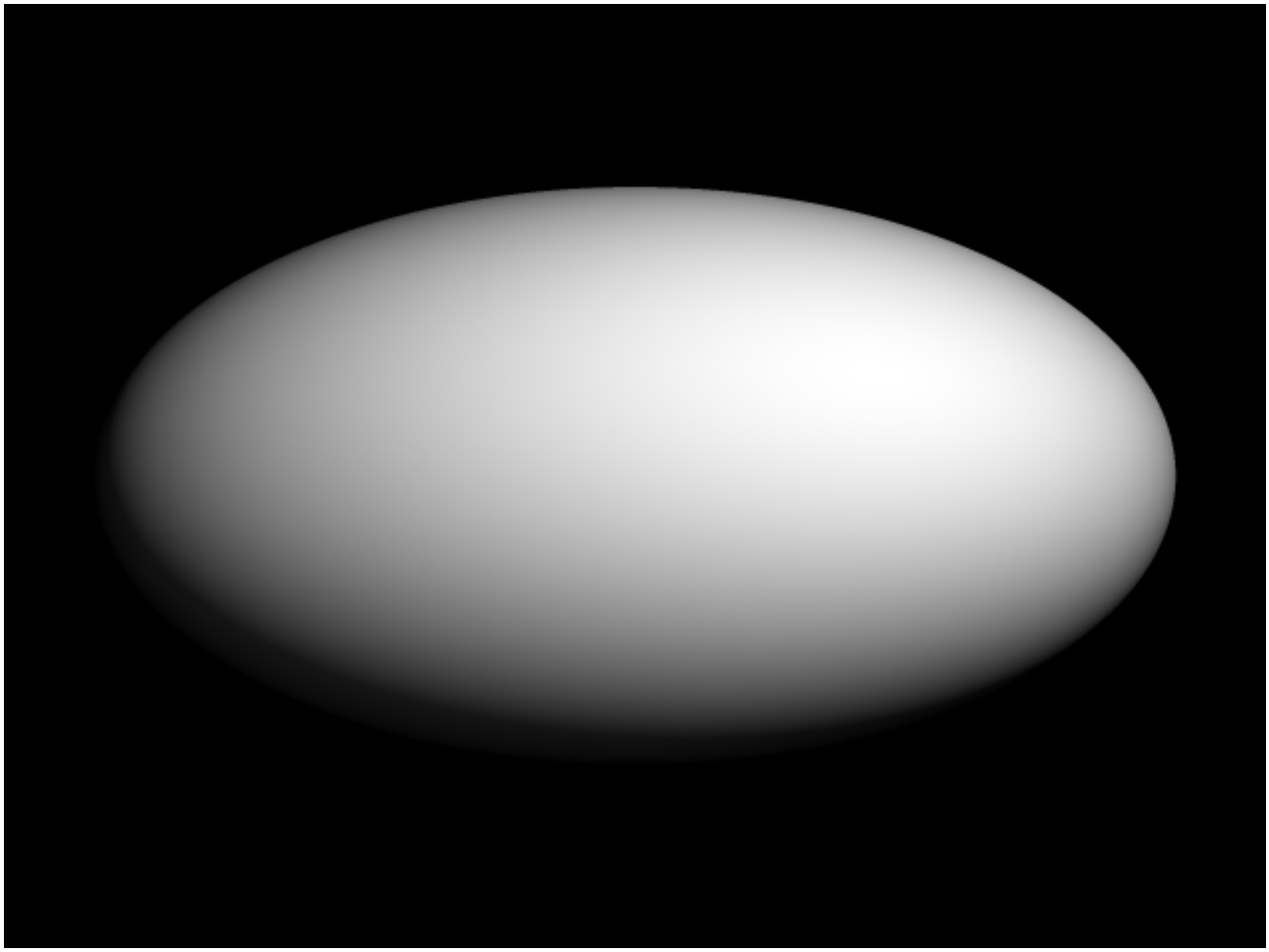


Figure 6: A 3D ellipsoid

7.1.4. Torus

$$(\sqrt{(x-x_0)^2+(y-y_0)^2}-R)^2+(z-z_0)^2-r^2=0$$

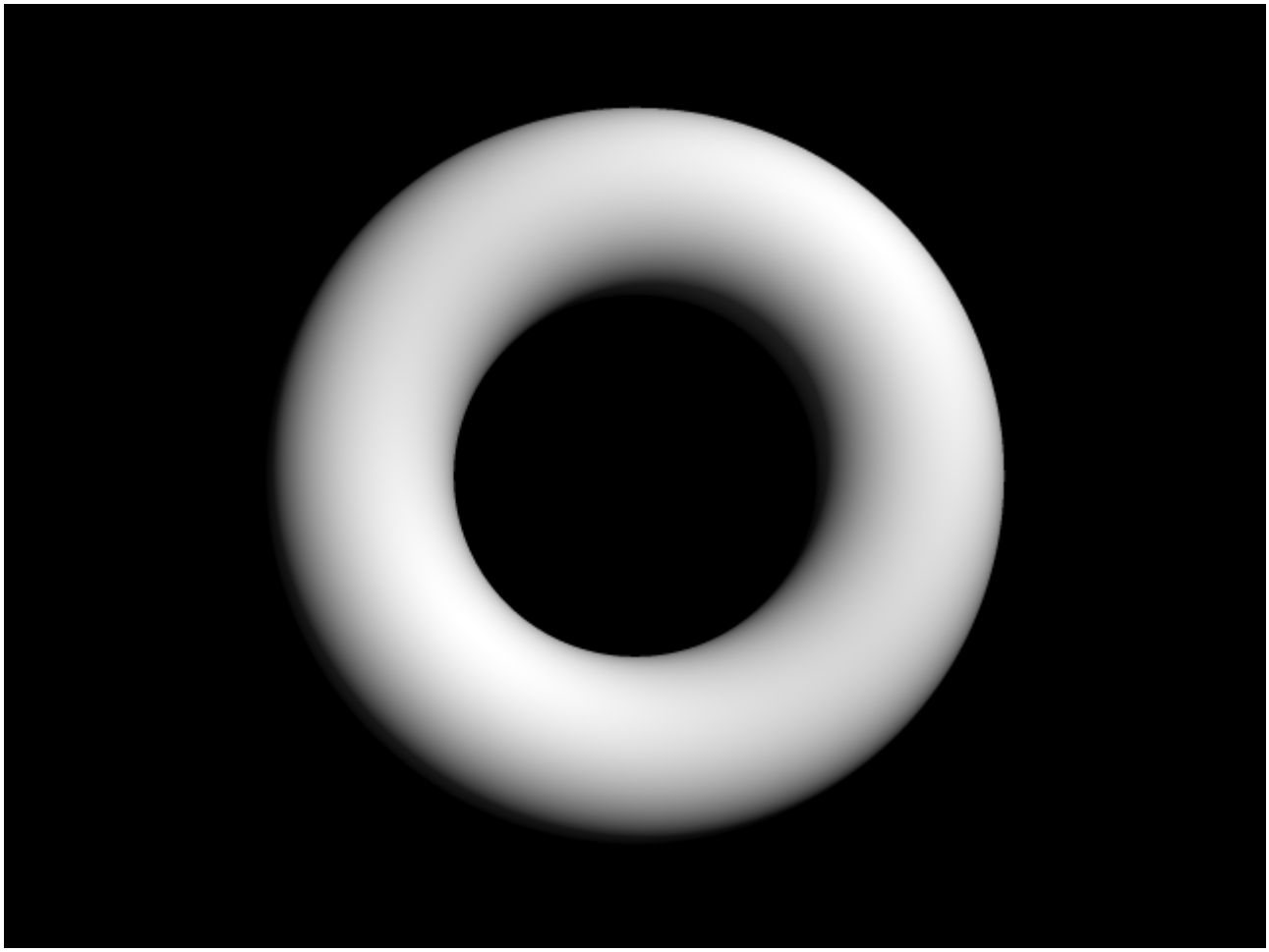


Figure 7: A 3D torus

7.1.5. Elliptic paraboloid

$$z - z_0 = \frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2}$$

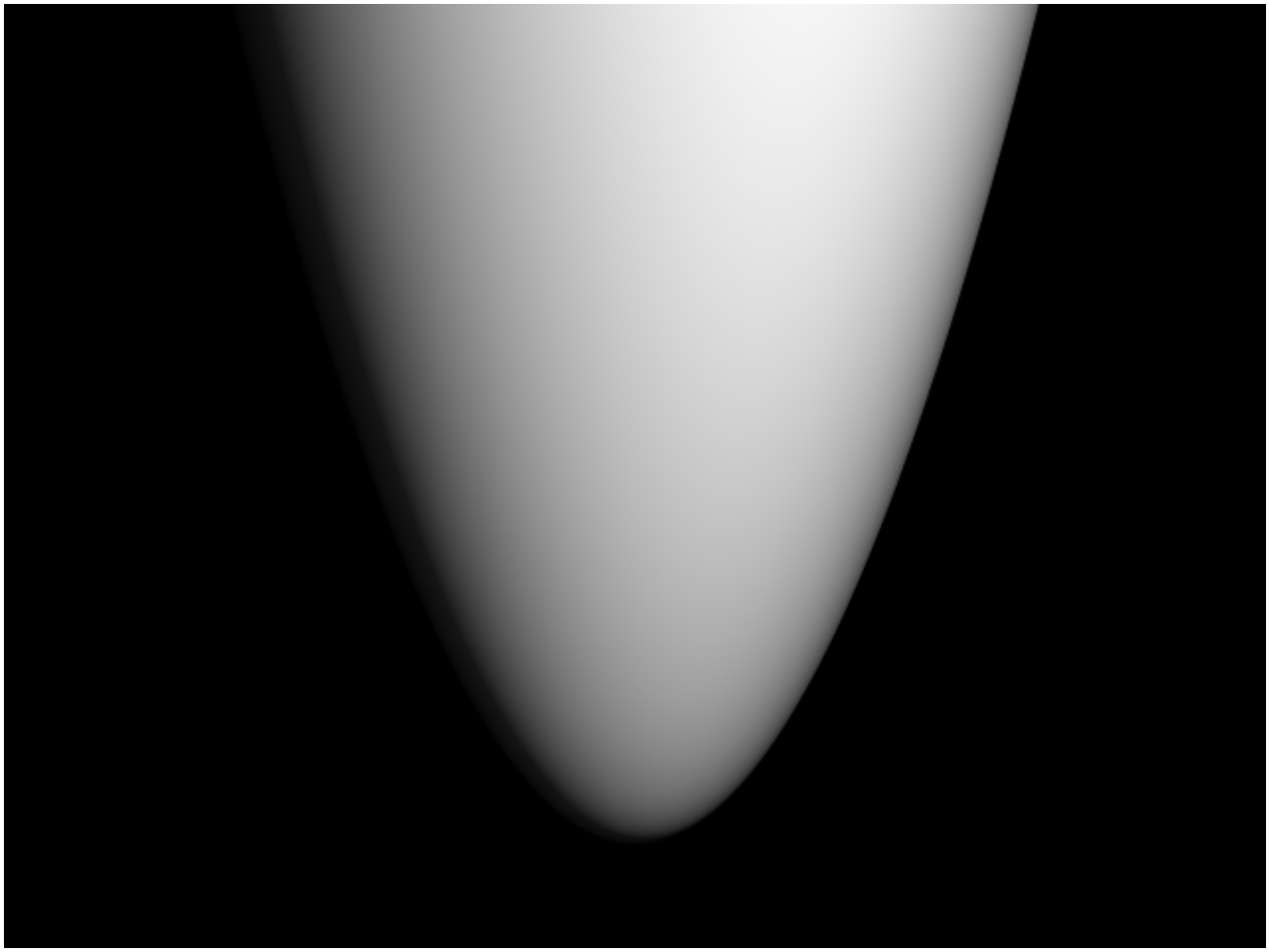


Figure 8: A 3D elliptic paraboloid

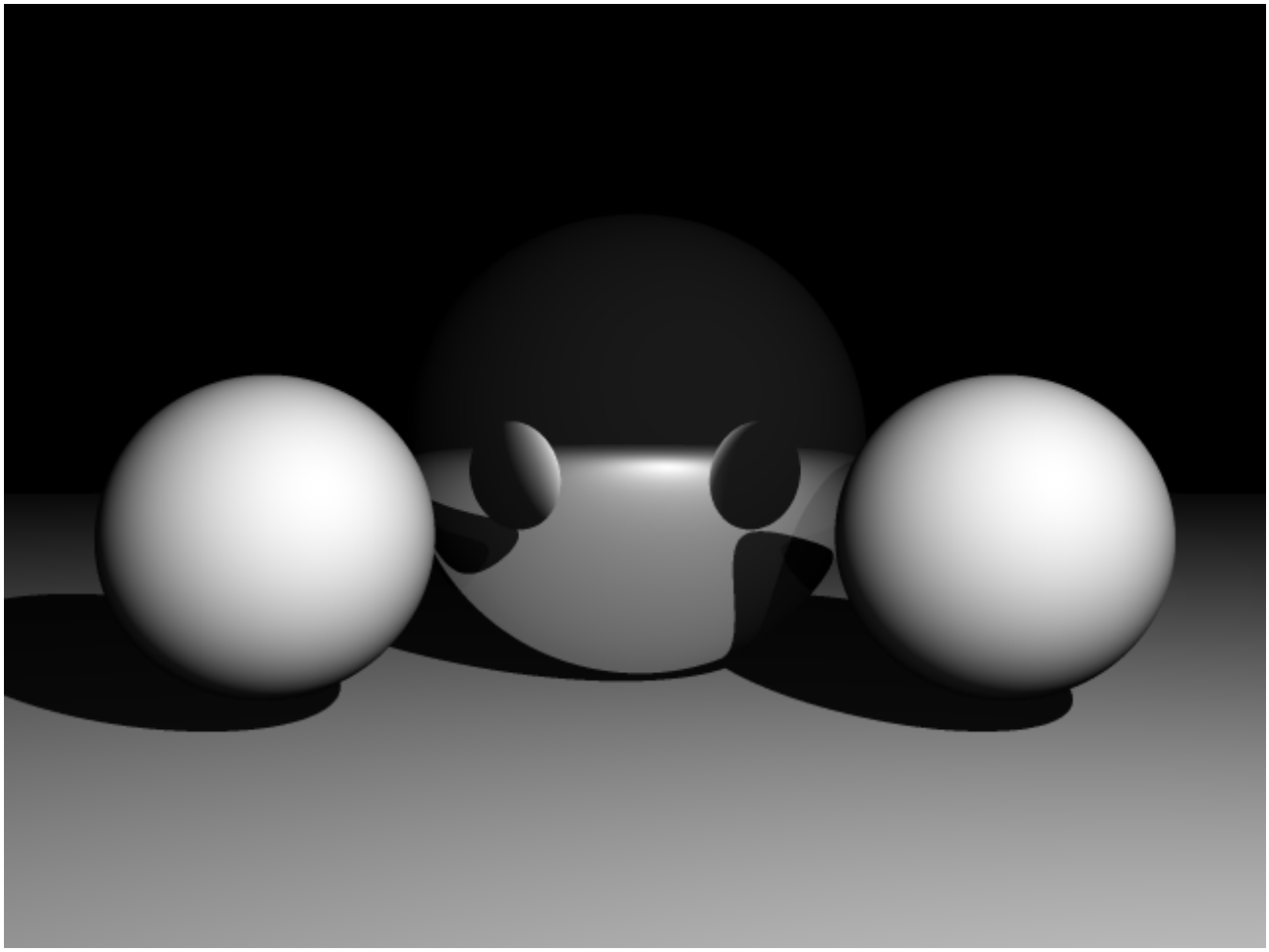
7.1.6. Heart

$$(x^2 + \frac{9}{4}z^2 + y^2 - 1)^3 - x^2y^3 - \frac{9}{80}z^2y^3 = 0$$

Figure 9: A 3D heart

7.2. Reflection

Here's a quick example of what reflection looks like in this program. The sphere in the middle is reflective, and you can see the reflections of the other 2 spheres in it:



7.3. Refraction

These examples show how refraction works and how the refractive index n affects it:

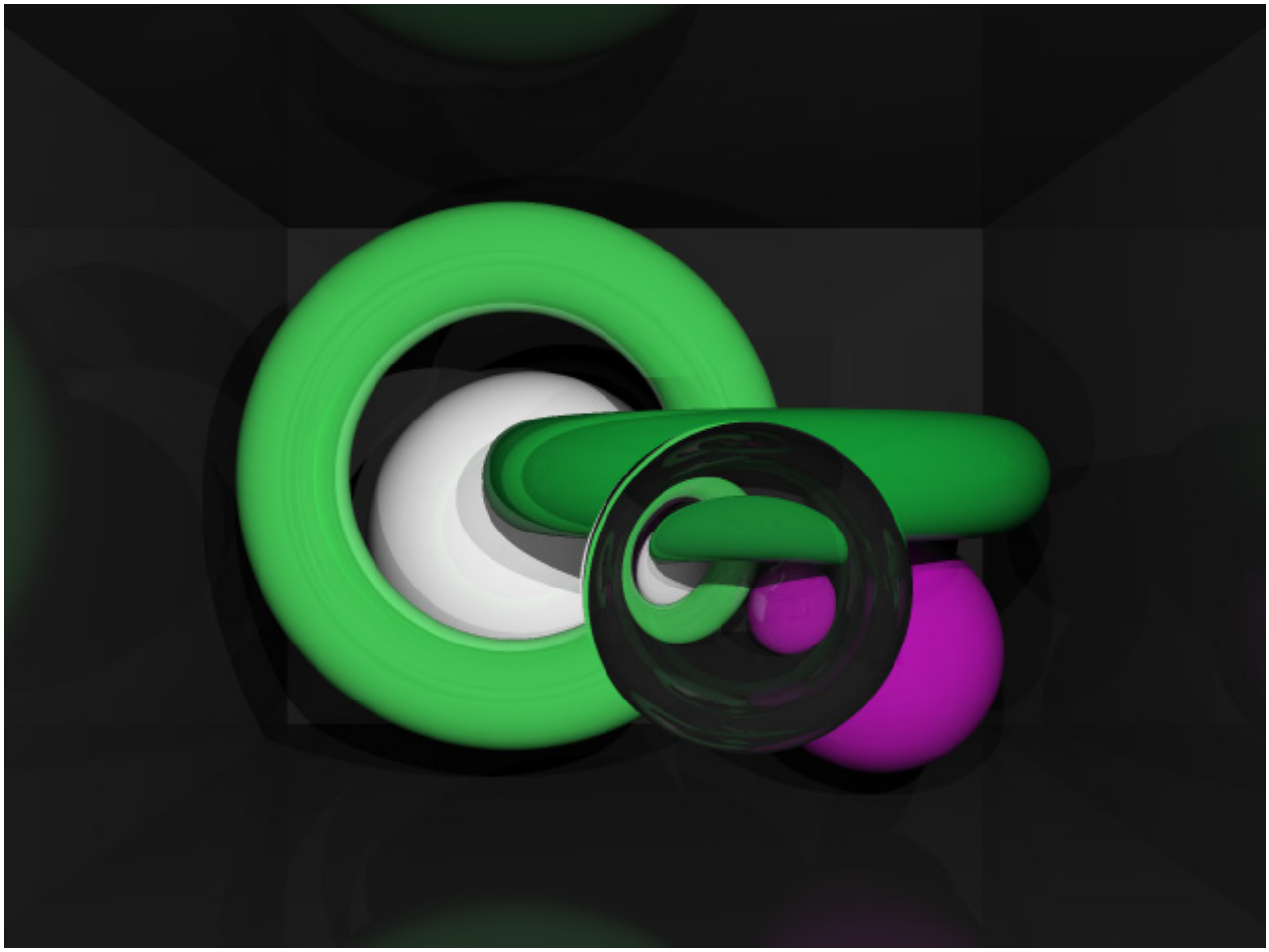


Figure 10: n = 0.8

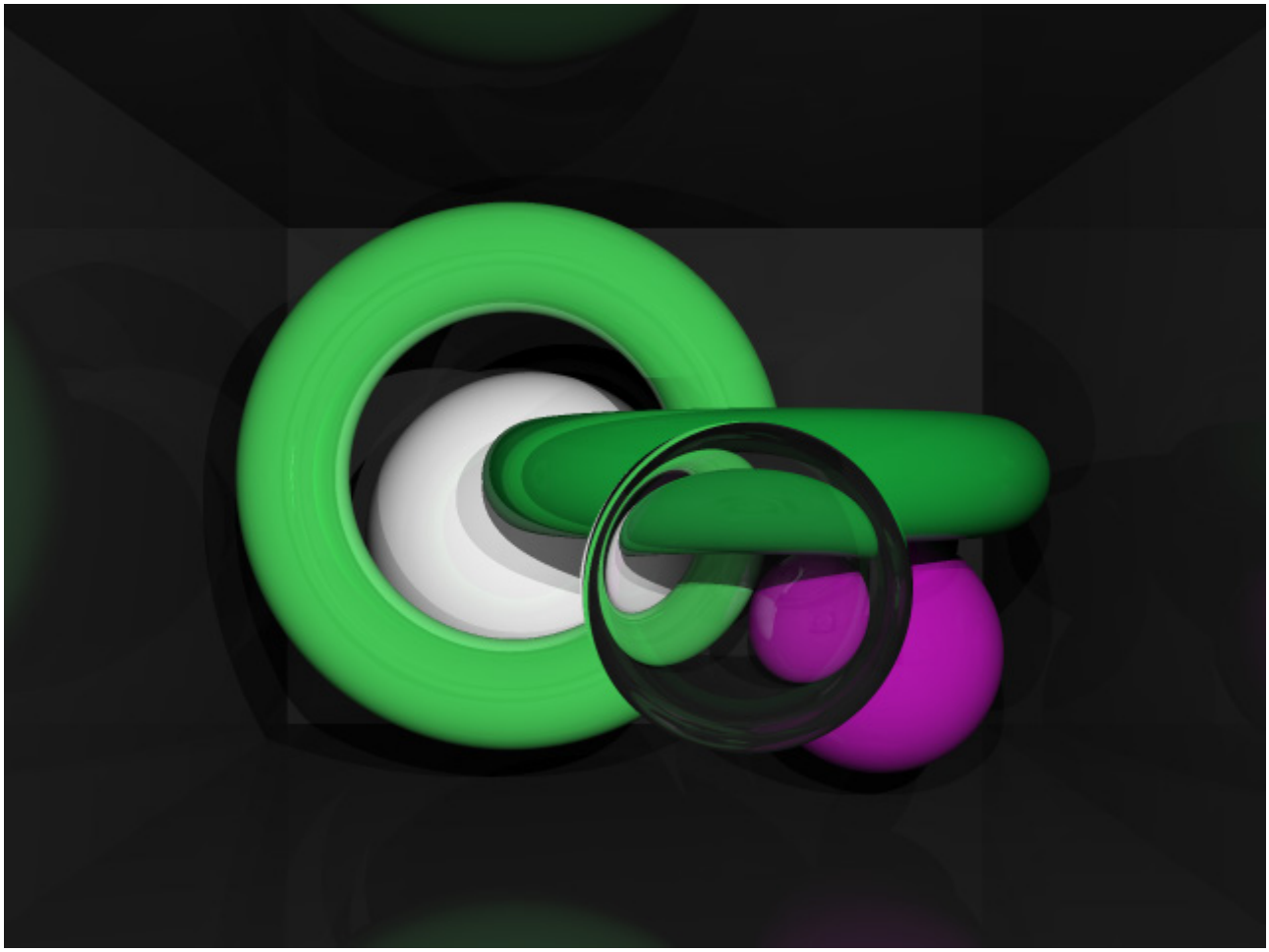


Figure 11: n = 0.9

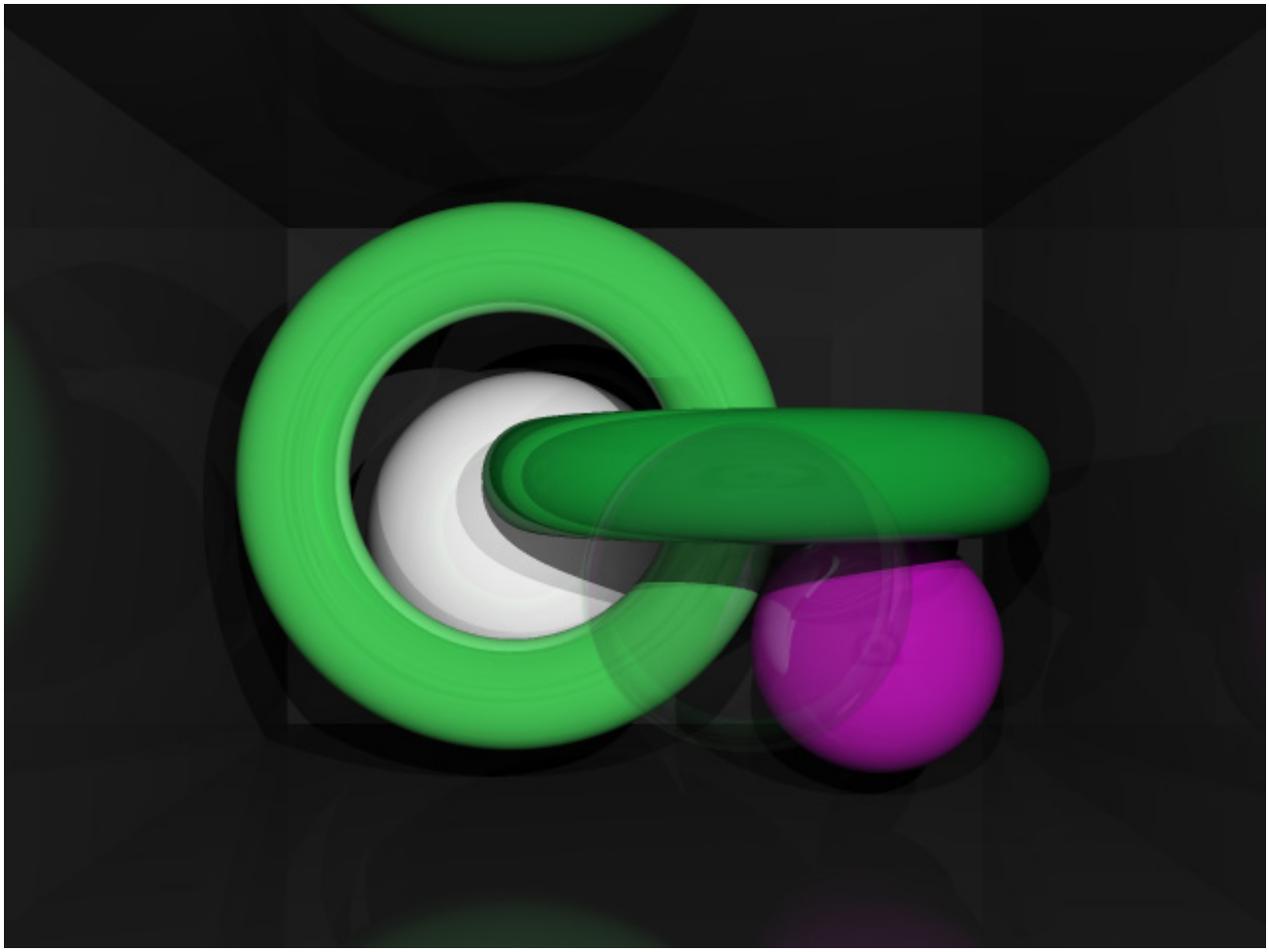


Figure 12: n = 1.0

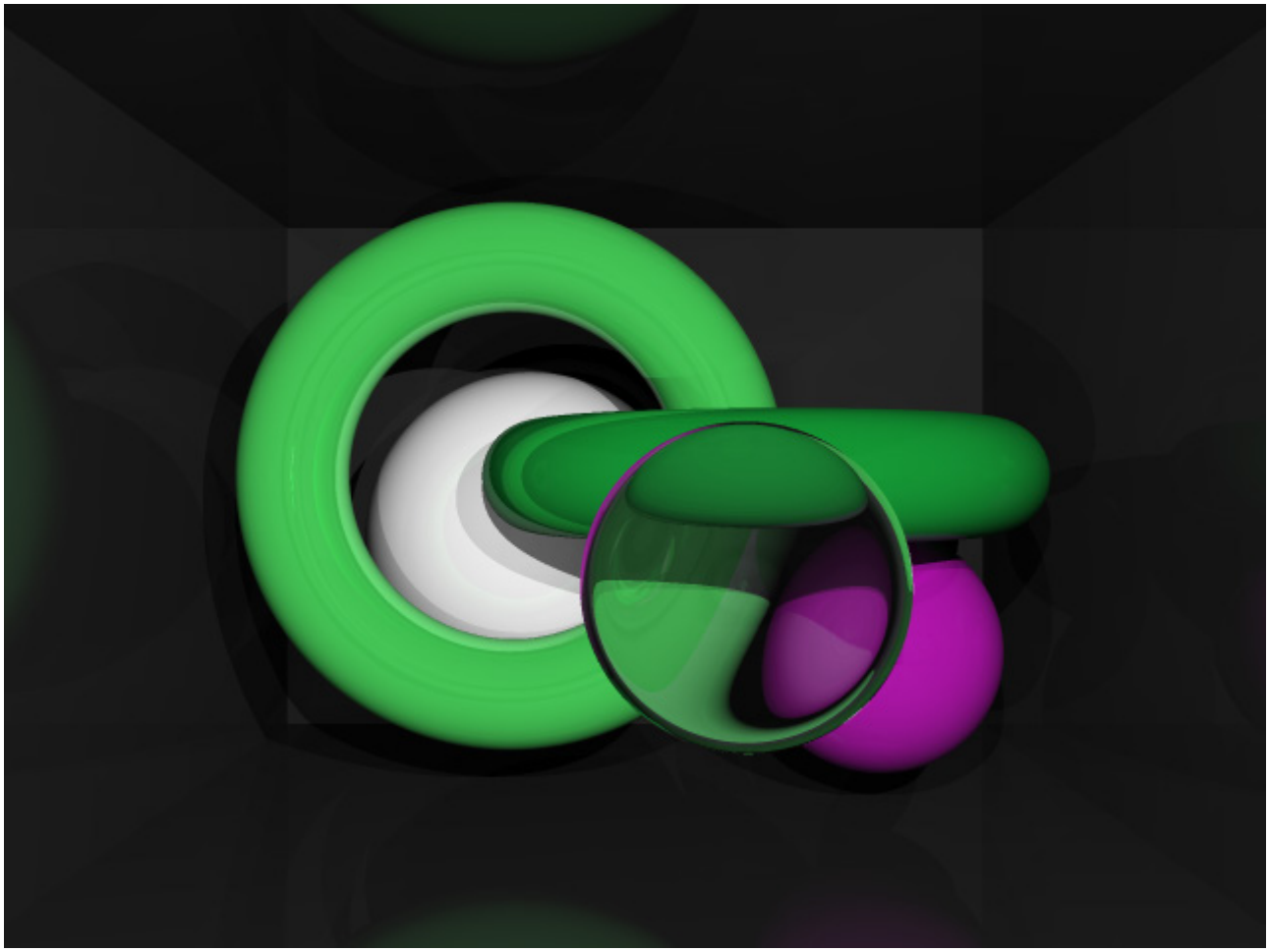


Figure 13: n = 1.1

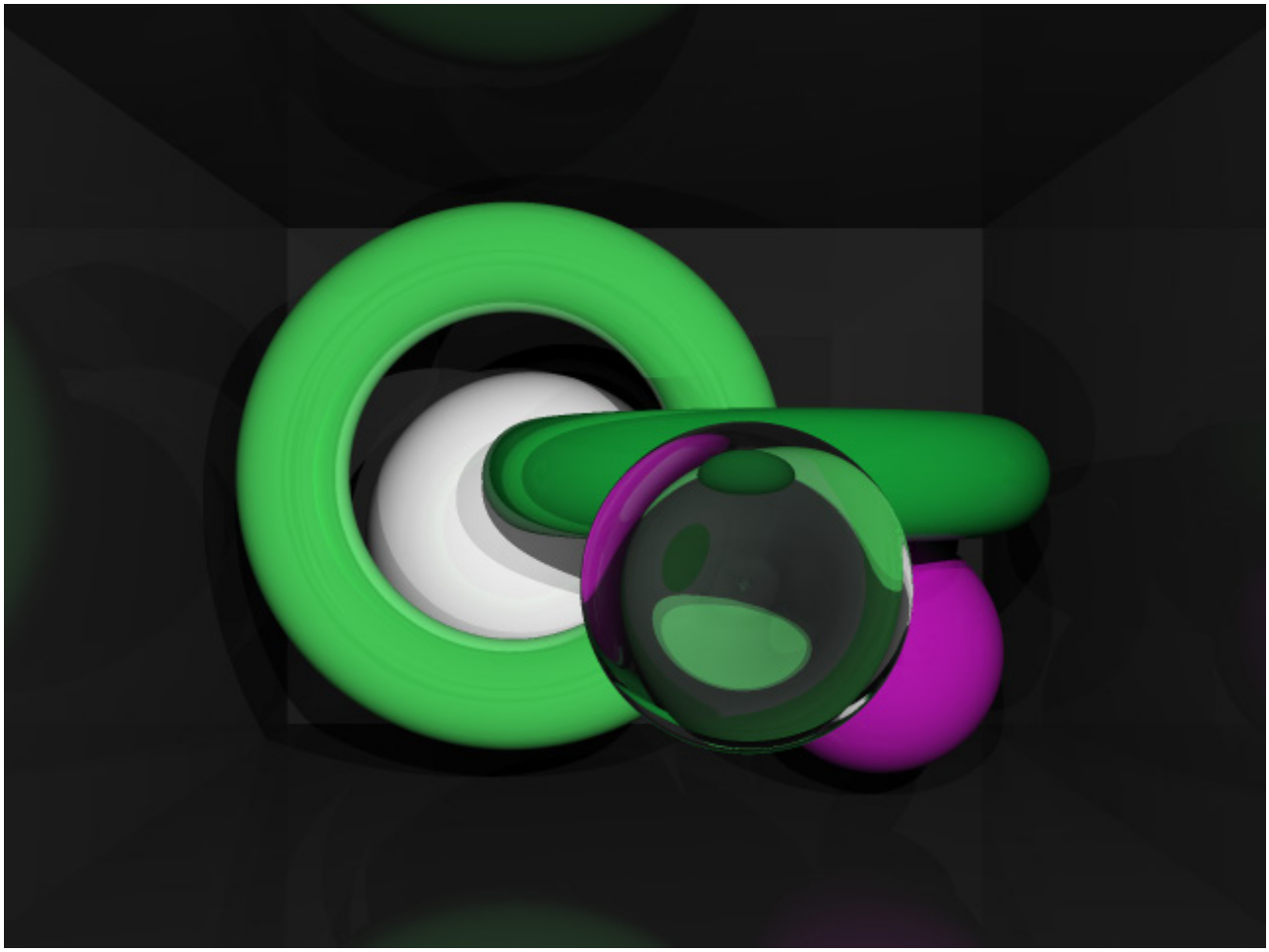


Figure 14: n = 1.2

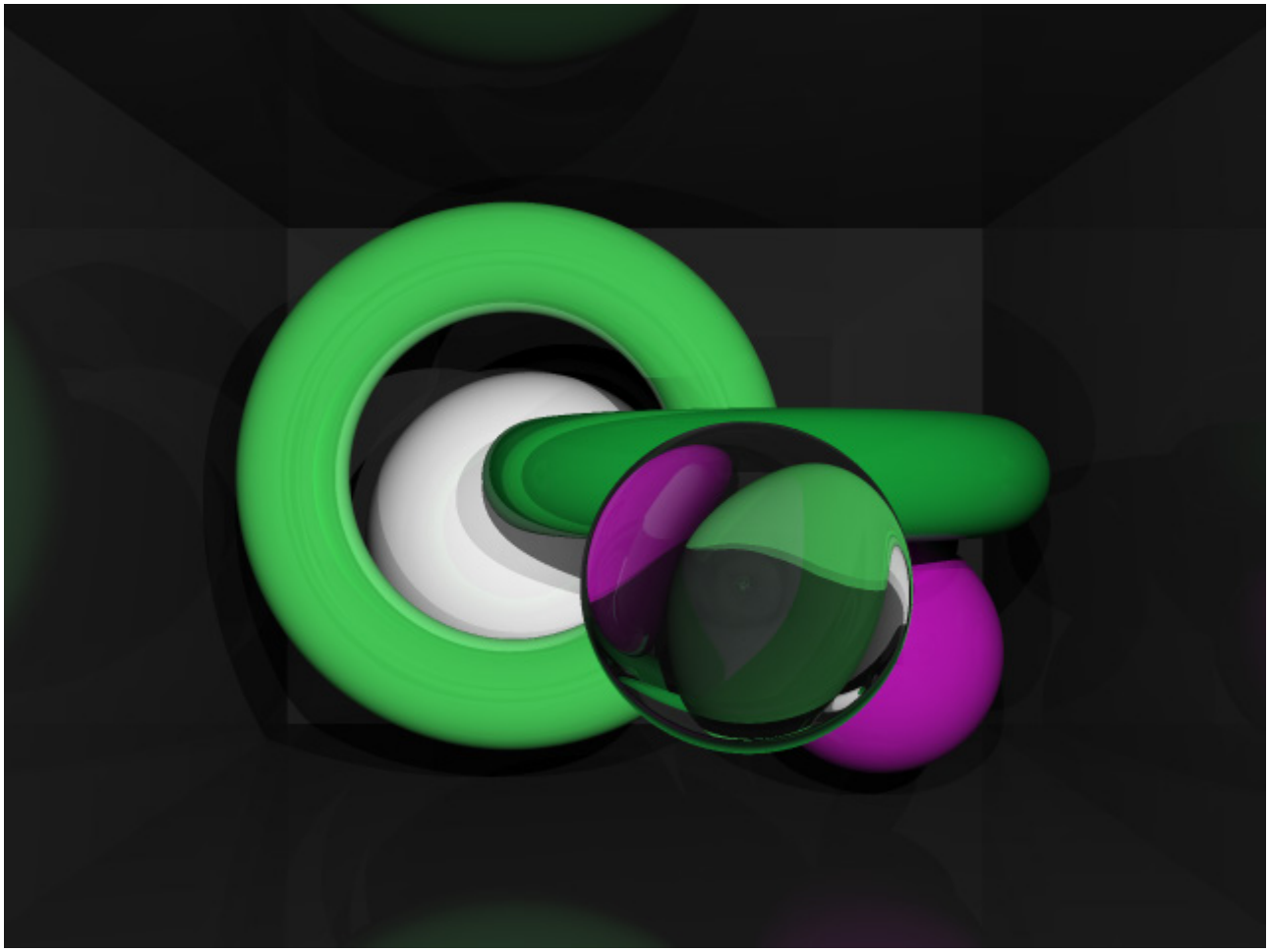


Figure 15: n = 1.3

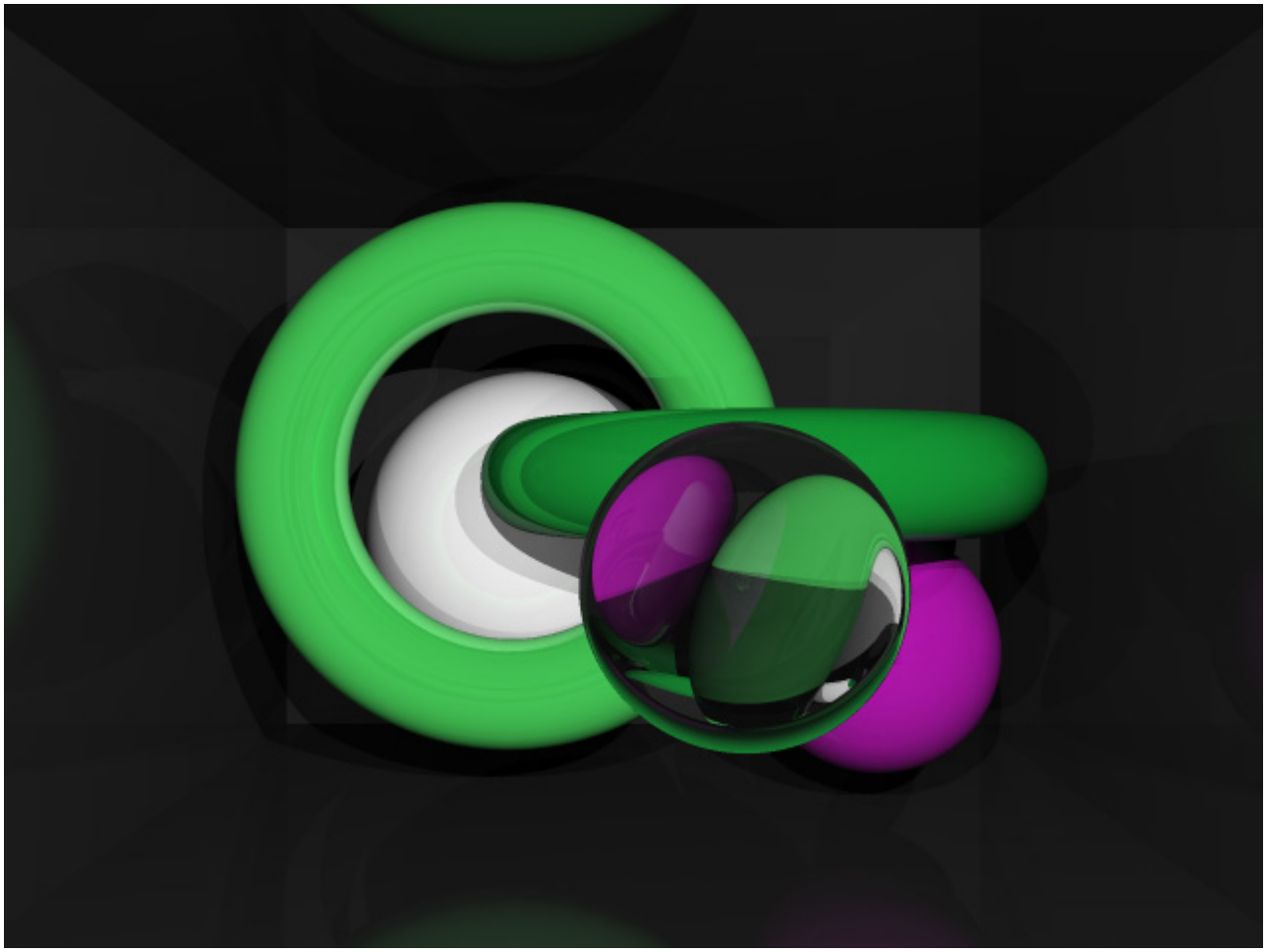


Figure 16: n = 1.4

7.4. Other renders

Here are some other examples of scenes with multiple objects:

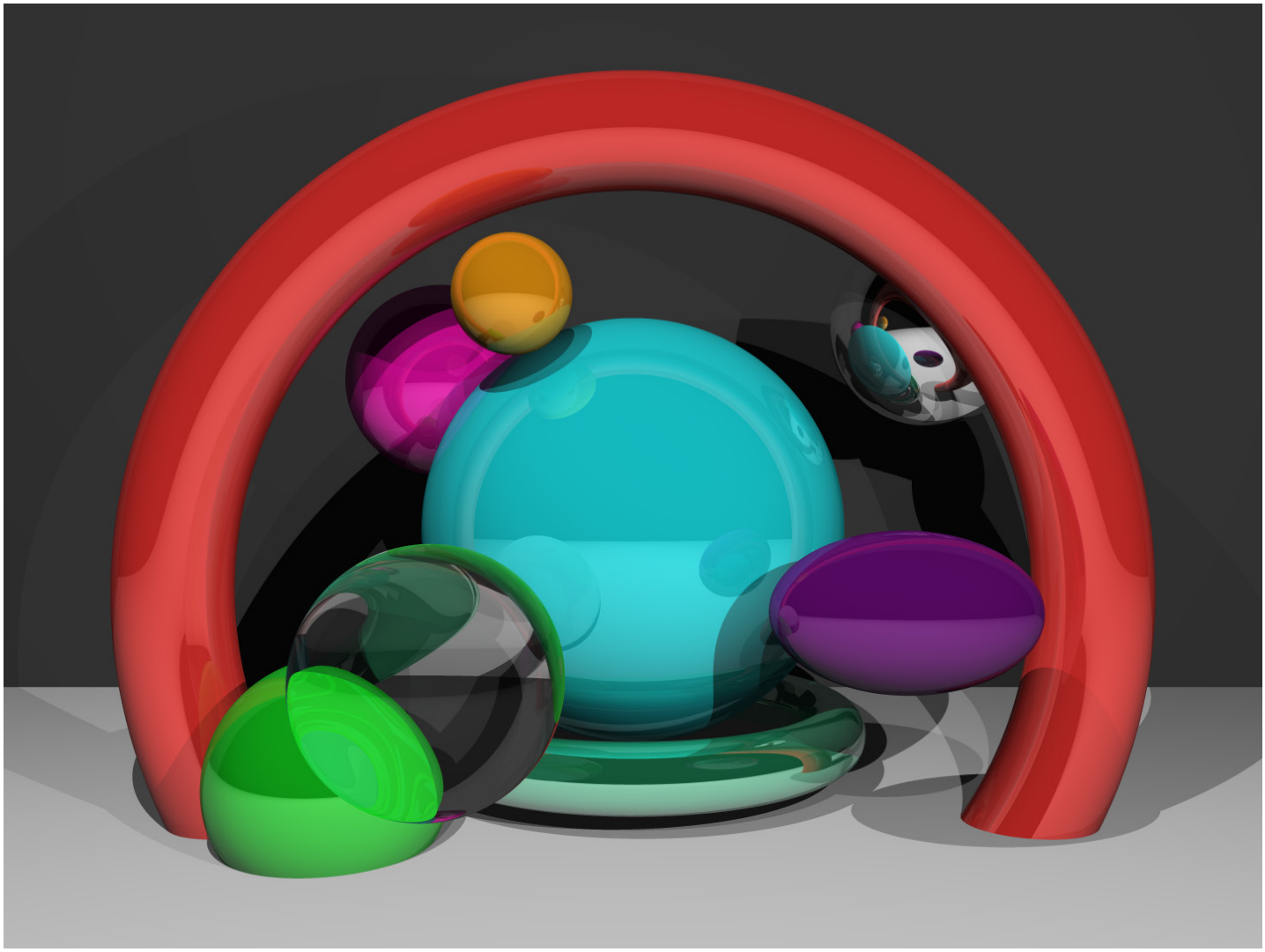


Figure 17: Scene 1

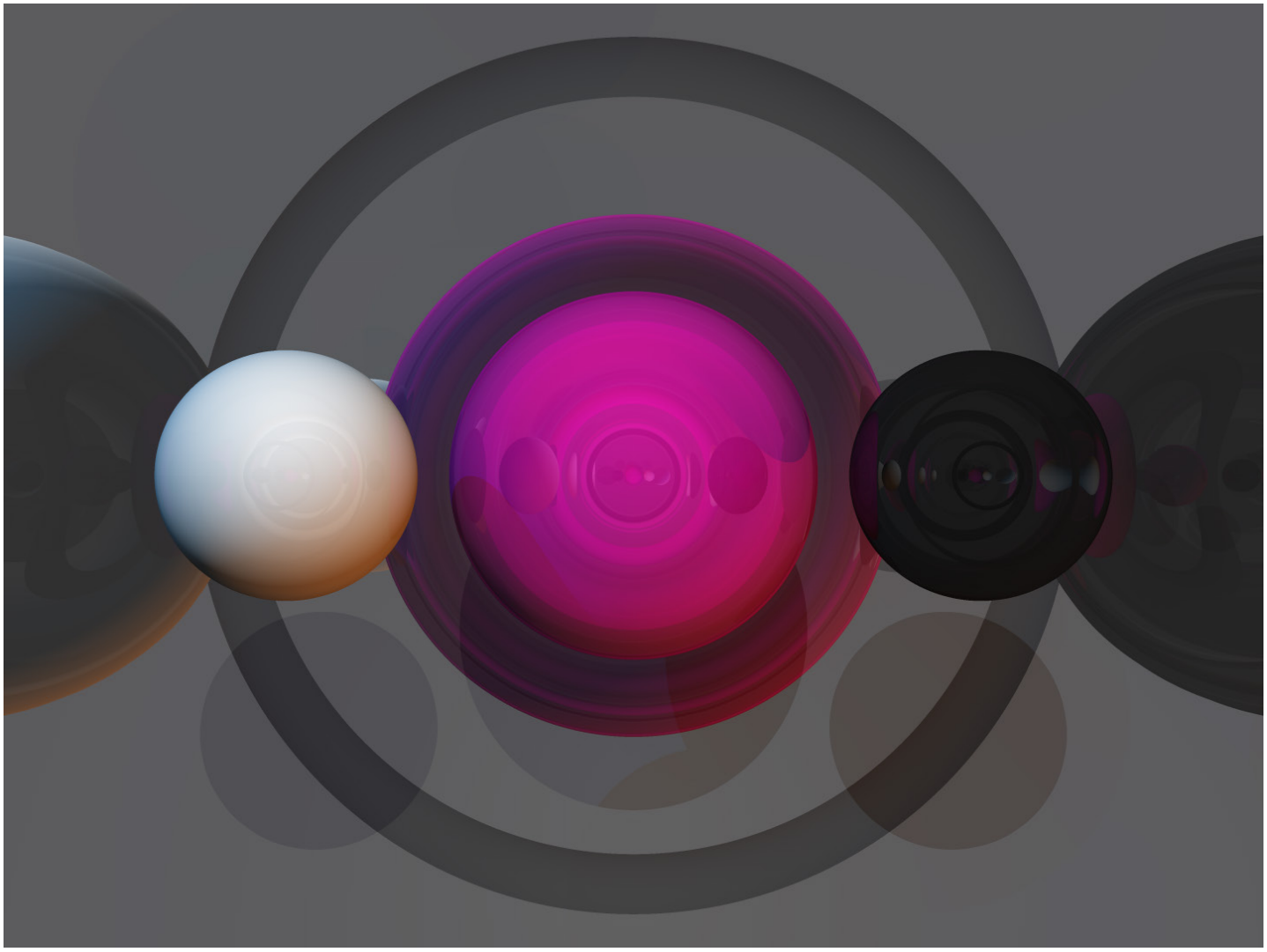


Figure 18: Scene 2

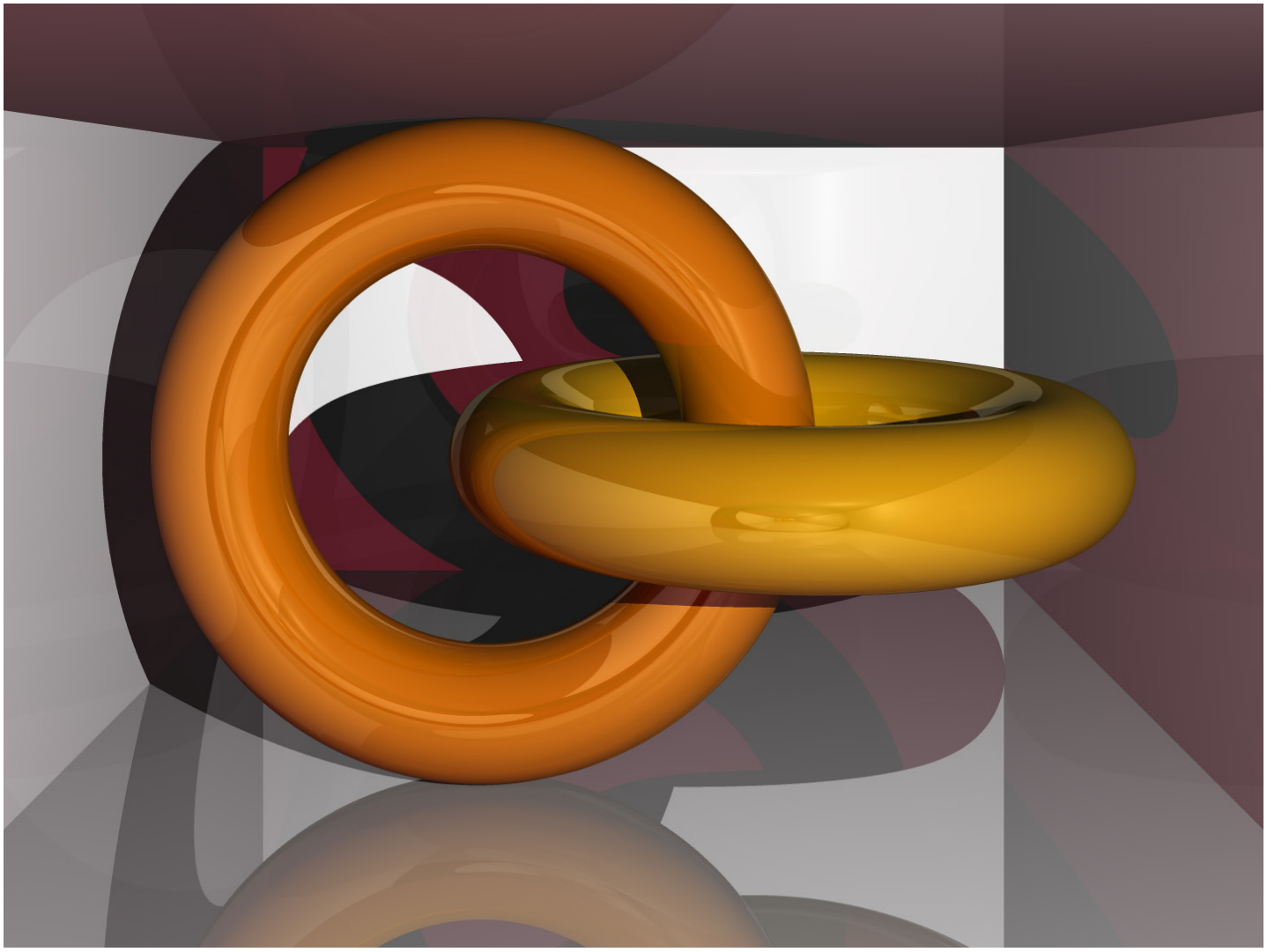


Figure 19: Scene 3

8. Project work division

8.1. Rok Cej

- Wrote this report
- Programmed the ray tracer
- Rendered all images

8.2. David Ocepek

- Wrote functions and partial derivatives for Heart, Liquid and Vibration objects
- Helped Rok with planning and programming the ray tracer
- Researched ray tracing and light mixing