A FRIENDLY INTRODUCTION TO RAY MARCHING

ZACH STRONG

Date: November 9, 2024.

Contents

1. Introduction

Imagine a door. It doesn't have to be very fancy. All that's important is that its shape is clear. Very likely, that shape is going to be a rectangle (ignoring its three-dimensional existence for a moment and just focusing on its face—the side with the doorknob). Now, imagine that same door opening. Since the door is merely rotating on its hinge and not being deformed in any way, its shape shouldn't change—it should remain rectangular. Indeed, when the human eye sees an open door, it interprets the object as having the same rectangular shape as a closed door.

This may seem like a totally uninteresting fact. It seems blatantly obvious that moving an object through space should preserve its shape. For instance, moving a cup of tea from one part of an apartment to another should leave the appearance of the cup completely unchanged[1]. The only thing that would appear to change about the cup is its size. If the cup was moved farther away, it would look smaller, while if it was moved closer, it would look bigger. This can easily be understood as the cup taking up more or less of one's vision depending on its distance from the eye. The cup isn't *actually* changing size, of course. It's just how humans perceive distance.

However, this "tea cup changing size" idea raises an important point about the "opening door" idea. If objects can appear to grow/shrink depending on their distance from some observer, then shouldn't the edge of the open door closest to an observer look larger than the opposite edge? If this is true, then an open door should look less like a rectangle and more like a trapezoid, with the shorter far edge and the taller close edge connected together by the top and bottom edges. Though, as stated above, the human eye sees the open and closed doors as being the same shape: rectangular. It seems the human eye is able to look past the "tea cup" observation when viewing open doors and see the shape for what it is: a rectangle.

This ability of the human eye is what allows 3D environments to be drawn on 2D mediums, such as paintings or computer screens. If objects are drawn in just the right way so that they emulate the "tea cup" observation—which we'll now refer to as depth—then the eye will view the image not as a 2D surface, but a 3D environment. The question now is how we'd go about capturing the proper depth of objects in a scene in order to "activate" this ability of the eye. Anyone who's attempted to draw 3D environments before will know that it isn't as easy as simply making objects bigger or smaller; they have to grow/shrink in a particular way.

[1]...so long as none of it was spilled.

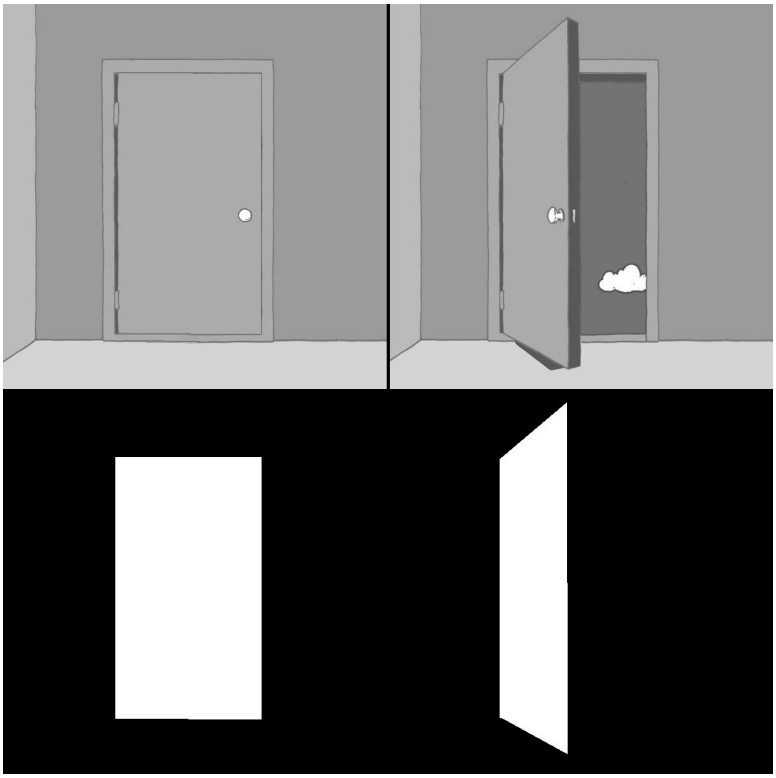


FIGURE 1. Open and closed doors, along with their pure "shapes". Though the human eye sees both doors as rectangular, the "actual" shape of the open door is trapezoidal. Image taken/modified from Blue "Gene" Tyranny's album *Out of the Blue* (1978).

How exactly should objects grow/shrink in an image to convey their depth in a scene? It's an innocent enough question, but its answer requires opening the door to some rather intricate areas of math, mainly projective and analytic geometry. Our aim with this project isn't to fully understand all the subtleties and complexities that go into answering this question. Rather, we'll focus on one of the many techniques employed to draw three-dimensional scenes: ray tracing/marching. We'll cover the basic motivation behind why ray tracing/marching is a reasonable way to approach this question, then we'll dive into how we'd go about implementing it into computer code (specifically through MATLAB). As well, we'll discuss some of the many applications of ray tracing/marching, showcasing the power (and beauty!) of this method.

2. MOTIVATION

Back in the time of the Renaissance, people were keenly interested in understanding how to properly represent three-dimensional scenes onto two-dimensional canvases. Up until then, people's artwork looked markedly flat, with no real sense that the

FIGURE 2. *The Annunciation* by Simone Martini is a great example of the type of "flat" artwork created before the Renaissance.

image depicted a realistic scene. Objects that were supposed to be farther away in an image were rarely drawn to scale, and the environments in which the objects were placed were wholly separated from reality. This type of artwork certainly had its appeal, but for Renaissance painters who strived to capture the "mathematical law" of the natural world (Kline [1955] 80), it was a far-cry from what they wanted to achieve.

So, when Renaissance painters began their quest for realism, they needed to devise a way to capture the depth of a scene onto a flat surface. This is not a trivial task, as it took Renaissance artists over a hundred years to come up with a solution (Kline [1955] 80). However, once one hears the solution, it seems perfectly reasonable, and maybe even obvious.

Think of how we as humans[2] see a scene. In order to see *anything*, we need light to hit our eyes. In order to see a particular object, we need light to bounce off of that object and *then* hit our eyes. We use the angle at which the light hits our eyes to deduce where an object is placed in a scene. We interpret an object as being

[2]...assuming the creature reading this actually is human...

separate from the environment based on how certain parts of the light are absorbed by the object's material, leaving our eye with different "pieces" of light that make it look different from its surroundings[3].

One way we can visualise this process is by imagining beams of light as thin strings that travel in straight lines from a light source. When they bump into an object, they bounce off of it and travel in some other straight-path direction. If the string happens to bump into our eyes, we see the last object with which it collided. This collection of "light strings" that hit our eye (and none of the strings that don't hit our eye) is known as a *projection* of a scene (Kline [1955] 80).

Now, with this visual established, imagine looking out a window at some scene. In order to see anything from the scene, our projection must cross through the glass of the window before it reaches our eye. The brilliant idea of the Renaissance painters was to imagine the window as their canvas (Kline [1955] 80). Since the strings of light in our projection are all aimed at our eye (since otherwise they wouldn't be in the projection), it doesn't matter where exactly the strings of light "start". Whether a string starts from the sun, the last object it touched, or even the window we're looking through, so long as its trajectory stays the same, then nothing will change. So, if a group of light strings bounce off an object and cross through the window at a particular area, and a painter paints that object in exactly the same spot on the window (using the correct colours and textures and all that), then shouldn't the painted object's light hit our eye in exactly the same way as the real object? If done correctly, the answer is yes.

Thus, by imagining the light strings in a projection as crossing through some plane (like a window), Renaissance painters were able to capture the depth of a scene and all the objects within it. The plane through which a projection crosses is known as a *section* of a scene (Kline [1955] 80).

In fact, Renaissance painters often did more than just imagine light strings. They took the idea quite literally, using *actual* pieces of string and an *actual* viewing window to connect the light strings from an object to the section (see figure 3). This process worked well for scenes that Renaissance painters could construct in front of them, but what about scenes that they couldn't? What if a painter had the urge to paint a scene they could only imagine? In that case, this technique wouldn't be of any help; light strings couldn't be attached to objects in their mind.

[3]Thankfully, this isn't a physics project, and so we don't need an exact definition for what we mean by light "pieces". What's important is that, because of the different ways materials interact with light, different objects look different from each other.

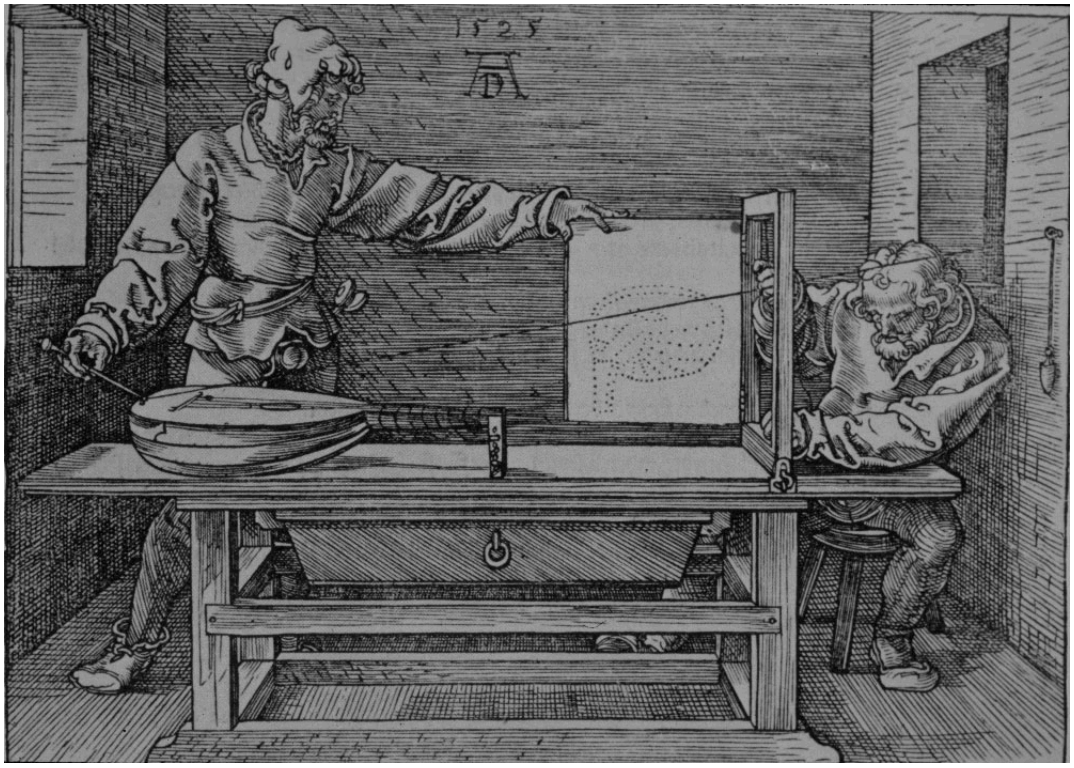


FIGURE 3. *The Designer of the Lute* by Albrecht Dürer, demonstrating the process of using physical light strings to accurately capture a section of a scene.

If the ideas of sections and projections were to be applicable beyond simply copying predetermined scenes, the techniques had to be understood from a conceptual point of view. Renaissance artists had to understand exactly *how* the light strings created a realistic image. Once they understood that, then attaching physical strings to things would be unnecessary—all they'd need to do is apply their reasoning to any scene they wished and determine what image their light strings *would've* produced[4].

The quest to understand these concepts led to the development of projective geometry (Kline [1955] 80-81), a vast area of mathematics which is way too large to even pretend to cover in this project. For us, though, all we need to know is that the tools of projective geometry *were* created, and painters *were* able to paint scenes that only existed in their mind thanks to these tools.

At this point, the reader may wonder why we're so interested in the techniques of painters from the 1600s. This is a valid question. It turns out that the problems Renaissance artists faced when creating paintings of imagined scenes are the exact problems computer scientists face when attempting to render three-dimensional

[4]It should be noted here that the term "light string" is by no means a standardised term. We use it only for ease of understanding, and because a formal term is not yet needed. When we switch our discussion to ray tracing/marching, more standard terminology will be used.

scenes on a computer. Though, in some sense, the computer scientists have it worse. Not only do they need to find a way to realistically display a three-dimensional scene onto a two-dimensional canvas (i.e. the monitor), but they must also devise a way to represent that scene in a way the computer can understand. With the Renaissance painters, there was no need to worry about how a scene was stored in the brain. Once it was imagined, it was there. With a computer, however, there's an obvious need to translate a scene from brain to binary before it can be displayed.

In any case, the tools of projective geometry are just as useful to the discipline of computer graphics as they are to Renaissance painters, so much so that the techniques the painters used are largely the same as those that contemporary digital programs use to render three-dimensional scenes. In other words, Renaissance painters basically used ray tracing to create their artwork, the same technique modern computers use to create animated movies, render medical images, and more[5]. In section 4, we'll discuss this similarity more explicitly. Before that, there are a few important concepts we need to establish before we can delve into the actual algorithms.

3. Setup

3.1. **Storing Objects.** How do we store scenes in a computer? It isn't as simple as storing a *photograph* of a scene, which is just a collection of coloured pixels in an array. A photograph doesn't store any data regarding the actual objects in a scene; everything gets flattened. If we tried to use an image of a scene as our storage medium, we wouldn't be able to view the scene from any angle other than the one at which the image was taken. For things like video games where we want the player to be able to roam around the game world as they please, this is an issue.

A more robust solution is to use tools from analytic geometry, which is essentially the study of geometric things as purely algebraic objects (Weisstein [2024]). We may remember from our high-school math courses that a circle in the xy-plane can be represented by the set of all solutions (x, y) to the equation $(x-a)^2+(y-b)^2 = r^2$, where (a, b) is the center point of the circle and r is the circle's radius. Using this representation, a circle becomes nothing more than a set of three numbers: a, b, and r. This is very easy for a computer to store. Representing a sphere in the xyz-space is no more complicated. For a sphere centred at the point (a, b, c) with radius r, the

[5]Section 5 gives some fun examples of where ray tracing/marching is used!

set of all solutions (x, y, z) to the equation

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2$$

will give us that sphere. So, to store this sphere in a computer, we just need to keep track of four numbers: a, b, c, and r.

Other shapes can be stored similarly by making use of equations whose solutions represent the object of interest. The equations themselves get a bit more complicated, but the idea remains the same. For example, given a non-rotated rectangular prism centred at the origin with side lengths r_x, r_y, and r_z, the equation whose set of solutions (x, y, z) forms the prism is given by

$$\max(|x| - r_x,\, 0)^2 + \max(|y| - r_y,\, 0)^2 + \max(|z| - r_z,\, 0)^2 = 0.$$

Some readers may find such equations slightly dissatisfying as we make use of the maximum function and the absolute value function, which feel a tad "artificial" compared to the algebraic operations we used to define the sphere. However, if we note that

$$\max(x,\, 0) = \frac{|x| + x}{2}$$

and

$$|x| = \sqrt{x^2},$$

then we can rewrite the box equation using only algebraic operations. Almost all equations for geometric objects can be reduced down to purely algebraic operations if we so desire. However, for the purpose of succinctness and computational efficiency, it's much better to leverage these "artificial" functions for defining our objects.

What about more complex objects? The majority of three-dimensional environments we want to render are composed of much more complicated shapes than boxes and spheres. If we were to try and derive equations to define them, we'd very likely end up with an absolute mess! Thankfully, there's a much easier solution, one that should be fairly familiar to any readers with a background in calculus.

Recall the process of Riemann integration. To integrate some function over an interval is to find the signed area beneath the curve of its plot. Depending on the function chosen, it could be really difficult, if not impossible, to find an exact formula to calculate the area. So, instead of trying to calculate it exactly, we *approximate* the area using a collection of rectangles (see figure 4).

The size of our rectangles clearly affects the quality of our approximation; the skinnier we make them, the closer our rectangles cover the curve, and the closer their

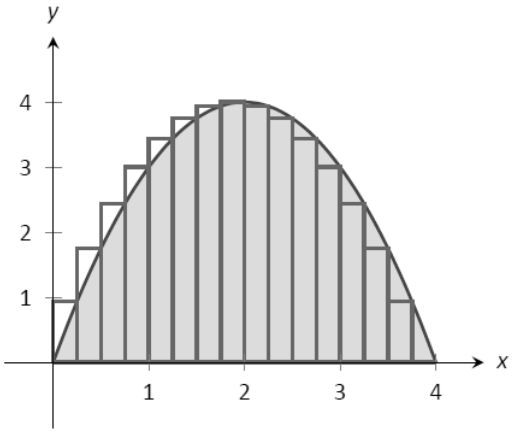


FIGURE 4. Using a bunch of rectangles, we can approximate the area under a given curve. Plot taken from figure 5.3.8 of et al. [2024]. The content is licensed under CC BY-NC 3.0 (https://creativecommons.org/licenses/by-nc/3.0/). Desaturated.

area will get to the true area of the curve. This exact principle can be used to approximate objects in three-dimensional space.

Say we have a model of a video game character we'd like to store in a computer. Rather than create some monstrous equation to define the character, we'll *approximate* the shape of our character using basic shapes that we know how to store. Typically, the shape chosen for this purpose is the humble triangle. Other shapes like quadrilaterals are also sometimes used, but for our example we'll stick to triangles. If we piece together a collection of flat triangles in just the right way, we can approximate our model (see figure 5).

Note that only the surface of a model needs to be approximated. The player will never need to see *inside* of a model, and so to approximate its shape, it's sufficient to use only flat shapes to cover the model, rather than try to fill it using polyhedra.

Much like the case with Riemann integration, the smaller we make our triangles, the better our model approximation will be, and the closer our model will look to the intended character. Though, as with anything involving accuracy on a computer, there's a trade-off between model accuracy and performance. Increasing the number of triangles we use will certainly make our models look more accurate, but it'll also put a bigger strain on the computer as there will be a greater number of shapes on-screen to manage. Thus, 3D modellers must find a fine balance between realism and performance.

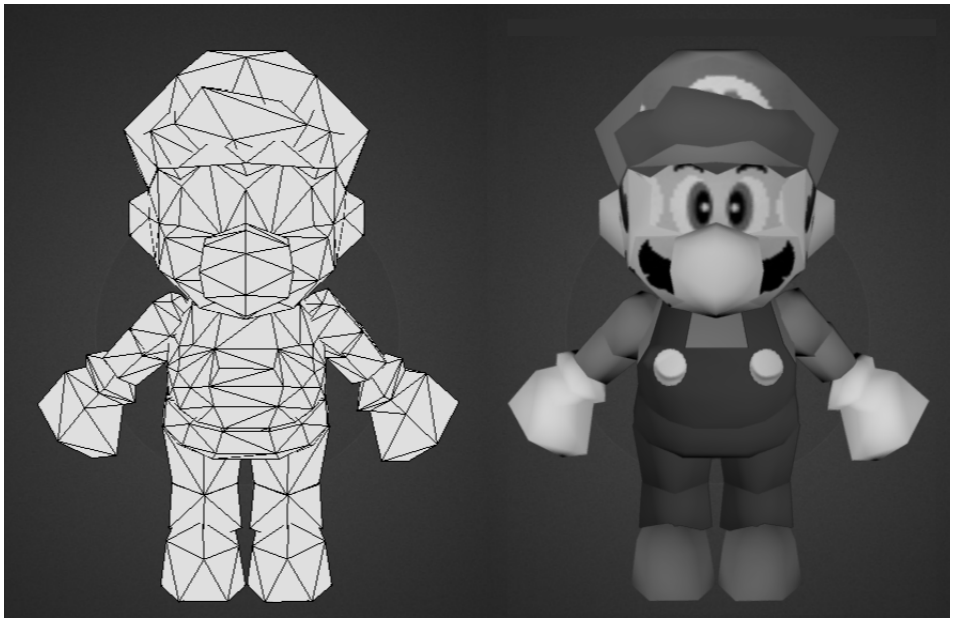


FIGURE 5. The player model from *Super Mario 64* (right), alongside its polygonal representation (left). Image taken from Tulsiani [2019].

This balance between accuracy and performance is very similar to the balance between numerical accuracy and computational effort involved in computing a Riemann sum. The more rectangles we use, the better the approximation, though using more rectangles means performing more computation. And, just like how we created techniques like the trapezoid rule and Simpson's rule for improving Riemann integration, so too are there optimisations for making polygonal approximations of models more computationally efficient without sacrificing accuracy (see, for instance, Ruin [2022]). This is a topic worthy of its own project, so we'll simply acknowledge it and continue without further thought.

3.2. **Interacting with Objects.** Now, with a method for representing (approximately) any object we want within a computer, the last thing we need to discuss before we can actually begin rendering scenes is how to make use of these representations. In particular, how do we ensure that the depth of a scene is properly captured? For instance, if we have a sphere sitting at $z = 5$, and another sphere of the same size sitting at $z = 10$, then viewing this scene from the origin should make the sphere at $z = 10$ seem smaller than the sphere at $z = 5$, just as we discussed in section 1.

The thing which will "encode" depth for us is the *signed distance function*, or SDF (Walczyk). As the name suggests, an SDF is a function which computes the distance from a point to some given object. The function is *signed* so that the function can detect whether a point is inside or outside of an object. If the sign is positive, then

the point is outside. Otherwise, it's inside. For this project, we won't deal with any cases where points are inside of objects, so this detail is mentioned only for completeness.

Having mathematical representations of our objects makes creating SDFs for them fairly straightforward. Let's consider again the humble sphere. An arbitrary sphere will be centred at some arbitrary point, say $\vec{a} = (a_1, a_2, a_3)$, and it'll have some given radius, say r. How do we compute the distance between this sphere and a given point $\vec{x} = (x_1, x_2, x_3)$? For any point to be touching the sphere, it must be *exactly* r units away from the centre point $\vec{a}$. In fact, this is exactly what the sphere's defining equation says:

$$\begin{aligned} &(x_1 - a_1)^2 + (x_2 - a_2)^2 + (x_3 - a_3)^2 = r^2 \\ \iff &\sqrt{(x_1 - a_1)^2 + (x_2 - a_2)^2 + (x_3 - a_3)^2} = r. \end{aligned} \tag{1}$$

The left-hand side of equation 1 is calculating the distance between $\vec{a}$ and $\vec{x}$ (via the 3D Pythagorean Theorem). If that distance is equal to the radius, then the point $\vec{x}$ must be on the sphere by definition. This is equivalent to saying

$$\sqrt{(x_1 - a_1)^2 + (x_2 - a_2)^2 + (x_3 - a_3)^2} - r = 0. \tag{2}$$

Now, the left-hand side of equation 2 must be zero for $\vec{x}$ to be on the sphere. Notice that if the distance between $\vec{a}$ and $\vec{x}$ is greater than r, the left-hand side will be positive, while the left-hand side will be negative if the distance is less than r. This is exactly what we want for our SDF. Thus,

$$\text{SDF}_{\text{sphere}}(\vec{x}, \vec{a}, r) = \sqrt{(x_1 - a_1)^2 + (x_2 - a_2)^2 + (x_3 - a_3)^2} - r.$$

Using this function, we can quantitatively measure which objects are farther away in our scenes, allowing us to incorporate depth into our renders (Walczyk).

For the earlier example with the two spheres at $z = 5$ and $z = 10$, this SDF is a bit overkill for determining which sphere is farther away. Though, for complex scenes where objects aren't placed at "nice" coordinate points, the SDF provides a simple way to determine distance from an observation point (e.g. where a video game's player is standing). As well, for objects which aren't as simple as spheres, determining distance is much more difficult, and so the SDF will handle all the complexity involved in calculating distances.

Consider two cubes sitting in a scene, both having their center points an equal distance away from an observation point (see figure 6). One cube has one of its faces angled directly towards the observation point, while another cube has one of its *vertices* angled directly towards the observation point. If we were to try and

FIGURE 6. Two equally-sized cubes, each with center points an equal distance away from an observation point. Which one is closer?

naively measure which cube was closer using something simple (like their center points), we'd determine that both were the same distance away. However, the cube with its vertex pointing towards the observation point is clearly closer—its vertex reaches closer than the flat face of the other cube.

In cases like these, the benefits of having an SDF are plain to see. We may then wonder how such an SDF would be implemented. For the case of a sphere, we didn't need to worry about things like orientation since a sphere looks identical from all angles. A cube, on the other hand, can look *very* different depending on the angle at which it's viewed, and so an SDF for a cube would have to take that into account. However, if we look back at our defining equation for a box, we notice that it required the box to have no rotation, and it required the box to be centred at the origin. It seems, then, that we'll have to derive a much more complicated expression for representing general boxes!

While we certainly *could* derive a more complicated representing equation for a box, it turns out that, using some clever manipulations, it's not required (Quilez [a]).

Say we have a box centred at the point $\vec{a} = (a_x, a_y, a_z)$ with side lengths along the axes given by $\vec{r} = (r_x, r_y, r_z)$ and rotations around the axes given by $\vec{\theta} = (\theta_x, \theta_y, \theta_z)$. To compute the distance between this box and a point $\vec{p} = (p_x, p_y, p_z)$, we first notice that translating the entire scene by some fixed amount won't change the distance between the box and the point. In particular, we could translate the entire scene so that $\vec{a}$ gets brought to the origin. With this translation, we'd now have our box at the origin, and our point $\vec{p}$ would become $\vec{p} - \vec{a}$.

So, without loss of generality, we can assume that $\vec{a} = \vec{0}$. If it isn't, we can just translate our scene so that $\vec{a}$ is brought to the origin. In a similar vein, we notice that we can *rotate* the entire scene without affecting the distance between our point and the box. In particular, we can rotate the scene so that our box is aligned perfectly

with the axes as we assumed in our defining equation for the box. Using rotation matrices, our point $\vec{p}$ would then become

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-\theta_x) & -\sin(-\theta_x) \\ 0 & \sin(-\theta_x) & \cos(-\theta_x) \end{bmatrix} \begin{bmatrix} \cos(-\theta_y) & 0 & \sin(-\theta_y) \\ 0 & 1 & 0 \\ -\sin(-\theta_y) & 0 & \cos(-\theta_y) \end{bmatrix} \begin{bmatrix} \cos(-\theta_z) & -\sin(-\theta_z) & 0 \\ \sin(-\theta_z) & \cos(-\theta_z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \vec{p}^{\mathsf{T}},$$

where $\vec{p}^{\mathsf{T}}$ represents the point $\vec{p}$ treated as a column vector (so the matrix multiplication makes sense)[6].

So, again without loss of generality, we can assume that no rotation is applied to the box (i.e. $\vec{\theta} = \vec{0}$). If there was rotation applied, we could simply rotate the entire scene so that the box is perfectly aligned with the coordinate axes. With these two simplifications, we notice that our box is now of the form described by our earlier equation:

$$\begin{aligned} &\max(|p_x| - r_x,\, 0)^2 + \max(|p_y| - r_y,\, 0)^2 + \max(|p_z| - r_z,\, 0)^2 = 0 \\ \iff &\sqrt{\max(|p_x| - r_x,\, 0)^2 + \max(|p_y| - r_y,\, 0)^2 + \max(|p_z| - r_z,\, 0)^2} = 0. \end{aligned} \tag{3}$$

Much like equation 1, the left-hand side of equation 3 is calculating the distance between the point $\vec{p}$ and the surface of the box. However, unlike the case with the sphere, there's a slight technicality we should address. A true SDF for our box will give distance values when the point is inside the box. Because of how our defining equation works, the LHS will always give a value of 0 when the point is inside the box. So, we can't just use the LHS as our SDF in this case.

Thankfully, modifying the LHS of equation 3 to give these "missing" distance values isn't too difficult. When inside the box, the distance to the box's surface is given by the smallest difference between the magnitude (absolute value) of any of the point's coordinates and the corresponding box length for that coordinate (e.g. r_x for the x-coordinate, r_y for the y-coordinate, or r_z for the z-coordinate). As an expression, this is given by

$$\min(||p_x| - r_x|,\, ||p_y| - r_y|,\, ||p_z| - r_z|).$$

[6]Note that the actual matrices used for "undoing" the rotation of the box depend on the way in which rotations are handled by any given program. The expression provided here assumes that rotations are applied in the order of pitch (around x-axis), yaw (around y-axis), and roll (around z-axis). The specific details of this expression aren't so important for our purposes. What *is* important is knowing that the rotation can be undone in the first place.

So, by taking the maximum of this expression and the LHS of equation 3, we obtain our desired SDF[7]:

$$\mathrm{SDF}_{\mathrm{box}}(\vec{p}, \vec{r}) = \max\left(\sqrt{\sum_{i \in \{x,y,z\}} \max(|p_i| - r_i,\, 0)^2},\ \min_{i \in \{x,y,z\}} ||p_i| - r_i|\right).$$

Symbolically, this is a rather scary function, but all it's doing is calculating the signed distance between a point $\vec{p}$ and the surface of a box with side lengths given by $\vec{r}$. Perhaps now it's clear why we've chosen not to list the SDFs for other basic shapes.

In any case, the exact expression we ended up obtaining for our box isn't all that important for our purposes. The important takeaway from this digression is that, given some arbitrary shape we'd like to place in our environment, we can always transform and rotate the entire scene to place that object at the origin with no rotation. Thus, for any shape, if suffices to find an SDF for a "simplified" version of that object.

In fact, in some cases, it suffices to find an SDF for particular *parts* of an object, rather than the entire object. Consider again our box example. A box is rather symmetric: it has at least three planes of symmetry that split the box up into eight "corner sections". These eight corner sections happen to all be reflections and rotations of one another. Thus, one way we could simplify our box SDF is by assuming our given point lies closest to one of the eight specific corners of the box and deriving an expression for the signed distance to that corner. Via translations, we could then extend this SDF to cover the other seven corners.

4. Implementation

4.1. **Ray Marching Intro.** With the concept of SDFs established, we can now explain the algorithm used to render a scene. The algorithm we'll discuss first is known as *ray marching*, a close relative of *ray tracing*. Both algorithms rely on the same idea of following "light strings", or light rays, and determining where they cross through some viewing window. The points through which they cross will be the places where our image will be drawn. The difference between the methods lies

[7]Note that this expression isn't quite the *true* SDF for a box, since points inside the box will be measured as having a positive distance rather than a negative. In code, this can easily be fixed by using a conditional statement to swap the distance to negative whenever we're inside the box (which we can detect by checking whether the radical expression evaluates to zero). Here, we leave the expression as-is so as not to bog down the project with more unnecessary notation—there's already enough of that!

solely in the computations performed for following the rays of light as they bounce around in the scene. Typically, ray marching is the simpler of the two algorithms, and in many cases, it performs much better than ray tracing (as we'll see in the following sections), so we'll focus the majority of our attention on it. Section 4.4 will briefly note the differences between the two algorithms in more detail.

The premise of ray marching is this: at some observation point from which we want to render our scene, we shoot light rays out through an imaginary window/frame into the scene, keeping track of the objects they hit. If a ray hits an object, then the place through which the ray crosses the window/frame is coloured depending on what object it hit and other properties we impose on the scene (e.g. reflectivity, lighting, etc.) (Walczyk). Below, we'll go into more detail as to why the algorithm works the way it does, but this is the basic idea.

So long as the light rays we shoot out behave like they would in reality (so, they follow straight paths, reflect off of reflective surfaces, etc.), then the resulting scene should be realistically depicted in the window we colour. Why? Well, this algorithm is basically just mirroring how our eyeballs perceive a scene in real life, just in reverse. Rather than light finding its way into our pupils, the light *starts* in our pupils and works backwards onto the objects in the scene. However, this reversal doesn't make a difference as long as the light rays shot from the observation point eventually find their way back to a light source. Below, we'll discuss methods for handling lighting, but really, this is a minor detail. The heart of ray marching is simply following the light rays and seeing what objects, if any, they hit.

One of the major advantages of ray marching compared to other types of scene rendering is that we don't need to explicitly specify how a scene is to be converted to a two-dimensional image on a screen. We just follow light rays and colour the imaginary window according to what objects they hit. Almost like magic, the scene will be properly flattened onto the window without ever having to explicitly tell the computer how to convert the three-dimensional space into two dimensions. The downside is that following light rays around a scene takes a lot of time and computation, even for simple scenes (as we'll see in the next section).

4.2. **Ray Marching Implementation.** To start, we need to define the window (or the *section*, using the terminology of the Renaissance artists) through which to view our scene. This translates to us defining the width and height of our image, as well as specifying how far from the observation point/viewer the window is placed. Why must we define the distance? Imagine standing in front of a window and slowly walking backwards. The amount of "outside" we see through the window decreases as we walk backwards, and so specifying the window's distance is, in fact,

an important thing to do.[8] In MATLAB, we can specify a width, height, and view window distance as follows:

```
imageDimensions = [480 640]; %y x
image = zeros(imageDimensions(1), imageDimensions(2), 3);
distToFrame = 5;
```

Next, we need to implement our SDFs from section 3.2 into code. There are a number of ways to do this, though for this implementation, we'll define a MATLAB function that takes as input a point and the properties of an object, and spits out as output the signed distance from the point to the object. For example, our SDF for the sphere could look like

```
function [dist] = dist_to_sphere(sphereVect, sphereRadius, posVect)
        dist = sum((sphereVect - posVect).^2)^(1/2) - sphereRadius;
end
```

Along with our SDFs, we also need a way to store our actual objects as data. Otherwise, the SDFs will have nothing to operate on! Again, there are several ways to do this. Our examples for this project will be relatively small, so we'll store our object data in simple structs and chuck all these structs into a list. Here's an example of a single sphere object specified in our program, centred at $(0, 0, 8)$ with a radius of 1:

```
objects = {struct("objectType", "sphere", ...
                  "center", [0 0 8], ...
                  "radius", 1, ...
                  "sdf", ...
                  @(this,p) dist_to_sphere(this.center,this.radius,p))};
```

Of course, the specifics of these code snippets aren't all that important. They're provided here solely as an example of how the math behind the algorithm might be translated into computer code. Appendix A includes the MATLAB code used to implement ray marching for this project, should we want a more complete reference.

Now, with our data specified and functions defined, let's render this scene. As with the Renaissance painters in section 1, we imagine how all the different light rays in our scene will behave when interacting with our objects. However, instead of tracing the light rays from the source of light to our objects, and then to our viewing window,

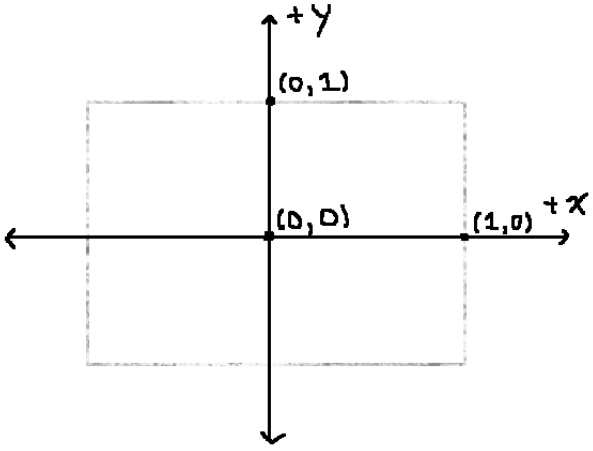
[8]The window's distance from the viewer is related to the concept of *field of view*—how much we can see of our scene at any given time. Many first-person video games have an option to adjust the field of view during gameplay. In essence, options such as these are simply adjusting the view window within the game's code (Walczyk).

we'll go in the opposite direction: tracing our light rays from the viewing window to the objects, then to the light source.

There are a few reasons for this switch. Firstly, if we shoot a bunch of light rays out of some light source, there's no guarantee that they'll eventually make their way through our viewing window. Worse, there's no guarantee they'll even hit an object! This would waste a lot of computation on stuff that wouldn't result in any sort of change to our image. So, by starting at the viewing window instead, we can ensure that all the light rays we compute will meaningfully impact our final image.

Secondly, even if all the light rays from a light source were to eventually cross through our viewing window, how can we guarantee that every part of our window is hit? For example, if all the light rays end up crossing through our viewing window on the left, what do we draw on the right? We can't immediately conclude that the right side of the image is completely dark—what if we just need to shoot out more light rays? From any given light source, there are an infinite number of directions at which a light ray can shoot out (think of all the places we could stick a pin into the surface of a sphere), so we'll never truly be sure whether an untouched part of our viewing window truly is dark, or whether we simply "missed" a direction to shoot a light ray out from a light source. We can avoid this problem by shooting light rays out from the viewing window. This ensures that every part of our viewing window is crossed by a light ray, meaning we'll know for sure whether that part of the final image needs to be drawn or not.

To ensure we shoot enough light rays through our window, we'll create one light ray for every pixel in our final image. This requires using the dimensions and the distance to the window we defined earlier. Naively, we may assign a coordinate system over our viewing window like so:



Then, if (w, h) represents the width and height, respectively, of the viewing window, and z is the distance to the window we defined, then we could easily define our light rays using the following expression:

$$\vec{p}_{i,j} = \left(-1 + i\frac{2}{w},\ -1 + j\frac{2}{h},\ z\right), \quad 0 \leq i < w,\ 0 \leq j < h. \tag{4}$$

Our rays would range from one end of the viewing window to the other in step sizes that would account for all the pixels in our final image, and the rays would all cross the viewing window by construction of our coordinate system. Note that we've encoded the light rays as vectors. The coordinates of the vector represent the "tip" of the light ray—that is, if our light rays are imagined as taut strings shooting into our scene, then the vector coordinate represents where the end of the string currently is, or how far the light ray has "reached out" into the scene. The direction of the light ray is implicitly given by the directed vector between the viewer and the tip of the light ray.

The setup explained above has a few flaws, the main one being that it doesn't take into account the aspect ratio of the viewing window. For square viewing windows (when $w = h$), this approach works fine. If the window is not square (as is the case with most images), we run into a problem with scaling. Notice that the distance between the left and right edge of our viewing window under our imposed coordinate system is 2, while the distance between the top and bottom edge is also 2. If the width and height of the viewing window differ, then moving a point a pixel in one direction will change the coordinate of the point a different amount then if we moved the point a pixel in the other direction.

To see why this is a problem, imagine trying to render a sphere using a non-square viewing window. Looking at a sphere head-on, it should appear like a circle on our viewing window. However, because our coordinate system doesn't take into account the aspect ratio of the viewing window, the sphere will be "squished" along one axis. The radius of the sphere will appear shorter along one axis since it'll take fewer pixels to span the length of the radius. Thus, our sphere will render as an ellipse.

Thankfully, changing our code to take into account the viewing window's aspect ratio isn't too difficult. We simply need to perform a transformation on our imposed coordinate space so that a pixel in one direction is "worth the same" as a pixel in another direction.

For example, suppose that our viewing window is longer than it is tall—most computer monitors are like this. Let (w, h) again be the width and height of our viewing

window, and z be the distance to the window. We'll set the length of one pixel to be $\hat{s} = \frac{2}{h}$. This means the distance between the top and bottom edge of the viewing window will be 2, like before. However, because the window is longer than it is tall, the distance between the left and right edge should be greater than 2. We can accomplish this by modifying expression 4 to start our light rays further left in the scene:

$$\vec{p}_{i,j} = \left(-\frac{w}{h} + i\hat{s},\ -1 + j\hat{s},\ z\right), \quad 0 \leq i < w,\ 0 \leq j < h. \tag{5}$$

One final adjustment can be made to expression 5 to better balance the image. While it wouldn't be all that noticeable for reasonably-sized images, our light rays are technically off-center by half a pixel. We start at $\left(-\frac{w}{h}, -1, z\right)$ and end at $\left(\frac{w-2}{h}, \frac{h-2}{h}, z\right)$. Our light rays don't end at the positive end of the viewing window; they stop just short. For large windows, this difference won't really be noticeable, but if we wanted to ensure our light rays were truly centred, we could shift both starting coordinates by one half:

$$\vec{p}_{i,j} = \left(-\frac{w}{h} + \left(i + \frac{1}{2}\right)\hat{s},\ -1 + \left(j + \frac{1}{2}\right)\hat{s},\ z\right), \quad 0 \leq i < w,\ 0 \leq j < h.$$

Finally, we have an expression for creating one light ray for every pixel in our image, ensuring that every spot on our viewing window has light to colour it. In MATLAB, we could implement this as follows:

```
rowShift = -imageDimensions(2)/imageDimensions(1); %x/y
colShift = -1;
pixelInc = 2/imageDimensions(1); %The "width" of a single pixel

for col = 1:imageDimensions(1)
    for row = 1:imageDimensions(2)
        %Create directed ray through pixel
        rowPlace = (row-0.5)*pixelInc;
        colPlace = (col-0.5)*pixelInc;
        pixelRay = [rowShift+rowPlace colShift+colPlace distToFrame];
        unitRay  = pixelRay ./ sqrt(sum(pixelRay.^2));

        %Insert other code here...

        end
end
```

Note that we're calculating a second vector `unitRay`. This vector is a unit-length version of our light ray, `pixelRay`, which will be used in later snippets of code.

Now that we have our light rays, what do we do with them? Well, we need to follow the rays out through the window and into the scene, but how do we do this? How do we know how far to shoot our rays out so that we don't miss any objects in the scene? One technique would be to step the tip of each light ray along its direction by some small amount until we either hit an object or shoot off to infinity. This, however, would be computationally expensive, as each ray would likely have to be stepped hundreds or even thousands of times before it reaches an object in the scene. Worse, because the step size would be fixed, it's possible that the tip of a light ray would "step over" a small object in the scene, passing directly through it without detecting that it's there (see figure 7)!

This is where our SDFs come in handy. Our SDFs, by definition, tell us how far a point is from a given object. If we use our SDFs to calculate the distances between the tip of a light ray and all the objects in a scene, we can then calculate the *minimum* distance between the light ray's tip and the scene. If we were to step the tip of our light ray along its direction by this minimum distance, then we're *guaranteed* not to miss any objects. Otherwise, if we did miss an object with this step size, then we clearly stepped more than the minimum distance to the scene as a smaller step size would've taken us to one of the objects. So, by repeatedly "marching" the tip of our light ray along its direction by the minimum distance to the scene, we can ensure we don't miss any objects without having to take a bunch of painfully small steps. This is how this rendering algorithm received the name "ray marching".

Visually, we can imagine this process as surrounding our light ray with a small bubble. This bubble grows and grows until it just touches some object in the scene. Then, the light ray extends forward until it reaches the edge of this bubble. The bubble pops, and a new bubble forms, starting the process over again. In this way, light rays which are far away from any objects will take bigger steps (as the

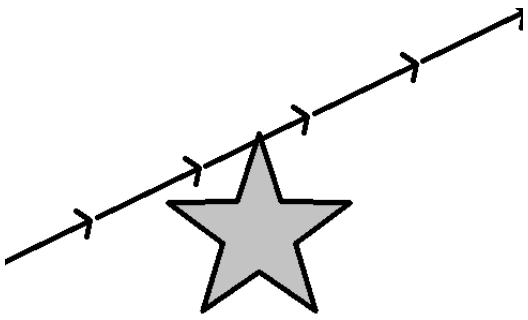


FIGURE 7. If we march forward our light rays by a fixed amount, it's possible that the tip of the light ray will "step over" an object, missing it.

bubble will be able to grow quite large before it touches anything), while rays close to objects will take small, careful steps (as the bubble won't grow very big before touching an object). Figure 8 below shows a visual depiction of this process.

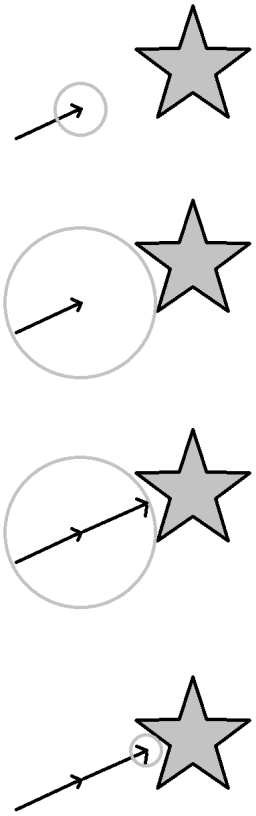


FIGURE 8. A bubble extends outwards until it touches the scene. Then, the ray "marches" to the edge of the bubble, and the process starts again. We repeat until the distance between an object and the ray is within some threshold, at which point we say the two are touching. This prevents light rays from "stepping over" objects.

Interestingly, a light ray's perception of the scene around it depends entirely on the outputs of our SDFs. The light rays don't know nor care about what the objects actually look like. All they care about is how close they are to said objects. This "decoupling" of objects from their SDFs allows us to do some very interesting things that would otherwise take a lot of extra effort via other rendering methods. We'll discuss this in more detail in section 4.3.

In any case, so long as we have all our objects and SDFs stored in our code somewhere, then determining the minimum distance between the tip of our light ray and the scene is relatively straightforward to implement. We simply loop over all objects in our scene and calculate the light ray's distance to them using the SDF. Then, we take the minimum. In code, we may write a function to calculate this minimum distance for us:

```
function [d] = signed_dist(point, objects)
    d = Inf;

    %Loop over all given objects
    for i = 1:size(objects, 2)
        %Getting distance to this particular object
        tempd = objects{i}.sdf(objects{i}, point);

        if tempd < d
            d = tempd;
        end
    end
end
```

Using this function, we can then loop over all our light rays as we showed above and calculate whether their tips are touching the scene or not. If they are, then we'll colour the place where those rays cross the viewing window white. Otherwise, we'll march the rays forward by the minimum amount and repeat. If a ray goes too far in the z direction, we'll assume it won't hit anything and move on to the next light ray. In code, this could look like

```
while pixelRay(3) < 100
        %100 is the farthest distance a light ray can travel
        % in the z direction before we assume it doesn't hit our scene

        abs(distToScene) = signed_dist(pixelRay, objects);

        %If the ray is close enough to an object, draw it!
        if distToScene <= 10e-4
                %Draw the light ray's pixel white
                image(col:col, row:row, :) = [1 1 1];
                break;
        else %Otherwise, march by minimum distance
                pixelRay = pixelRay + unitRay*distToScene;
        end
end
```

The last bit of code we need to write is just to write our `image` array out to an actual image so we can see what our ray marching algorithm produced:

```
imwrite(image, "test.png");
imshow("test.png");
```

This is enough code to produce our first ray marched image! The scene we've defined is quite simple: a sphere of radius one is sitting eight units in front of the viewer. The resulting image is exactly this:

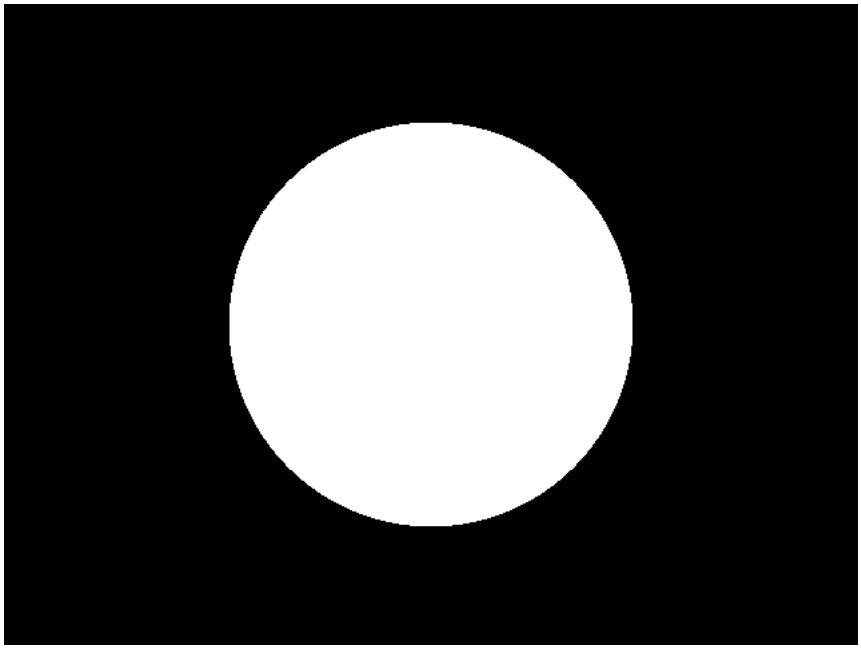


FIGURE 9. A sphere rendered using ray marching.

In practice, we'd likely never use a language like MATLAB to implement ray marching. A single sphere is about the most basic scene we can produce via ray marching, and yet the code provided here takes several seconds to produce the final image! If we were attempting to program a video game using this method, where the screen needs to be updated at least thirty times every second, this would be far too slow! An environment more apt for graphics processing, such as WebGL, would perform much better, as these environments can leverage the power of the computer's GPU, running thousands of calculations *in parallel* as opposed to sequentially like we're doing here.

While the specific structure and syntax of our code may change if we implement ray marching in another environment, the logic and algorithms would remain exactly the same. Everything we've written in MATLAB can be almost directly translated into any other language, and all the tools and intuition we've built up throughout this article will remain equally valid.

4.3. **Ray Marching Extensions.** Rendering a three-dimensional scene involves a lot more than merely drawing objects. For starters, the scene we rendered in figure 9 has no concept of environmental lighting. The only "light source" in our scene is the viewer, and it acts as an unrealistically-bright source of light. Anything our light

rays hit, they illuminate perfectly. In real life, this very rarely happens. Imagine pointing a flashlight directly onto a ball. The part of the ball facing the beam will likely be very bright, while parts near the edge will be darker since less light is pointed there. As well, objects which are farther away from the flashlight should be darker than objects that are closer since more of the light beam will scatter into the environment the longer it travels. None of this is accounted for in our implementation. No matter how far away an object is, so long as our light beams hit the object, it'll be displayed at 100% brightness.

There are several different ways lighting can be implemented within a ray marcher. There are also several different *types* of lighting that can be added: ambient, diffuse, specular... the list goes on. The topic is large enough for its own project, so for our purposes, we'll focus on one particular type of light: point diffuse lighting.

Diffuse lighting, put simply, is a way to emulate the impact direction has on how an object is illuminated (de Vries). Consider the screenshot of *Super Mario 64* gameplay in figure 10. Notice that the brick wall behind Mario is illuminated differently depending on whether it's to the left or right of the wall's corner. This is because the light from the stage's imagined sun is hitting these two sections of the wall at different angles, and so they're brightened differently, just as they would be in reality. A wall facing towards the sun is going to be brighter than a wall facing askew. This is diffuse lighting in action. Without it, both wall sections in the screenshot would be brightened the same, and the illusion of depth for the corner would be lessened.

To implement diffuse lighting into our code, we need to find a way to calculate in which direction an object's surface is facing (de Vries). A relatively straightforward way to do this is by making use of normal vectors. A normal vector is a vector that points perpendicularly (orthogonally) outwards from a point on some surface. If we remember some multivariate calculus, we can find the normal vector of a surface defined by an equation/function (e.g. an SDF) by finding the *gradient* of the function at the point of interest. The (normalized) gradient of a multivariate function at a point tells us in what direction from the given point the function is increasing the fastest. For an SDF, the function is increasing the fastest when the point is moving perpendicularly away from the surface of its object, since this is the quickest way to build distance from the surface, and the function is, by definition, measuring this distance. So, if we'd like to find the normal vector of a surface at some point, we simply need to calculate the gradient of the surface's SDF at that point and create a unit vector that points in the gradient's direction.

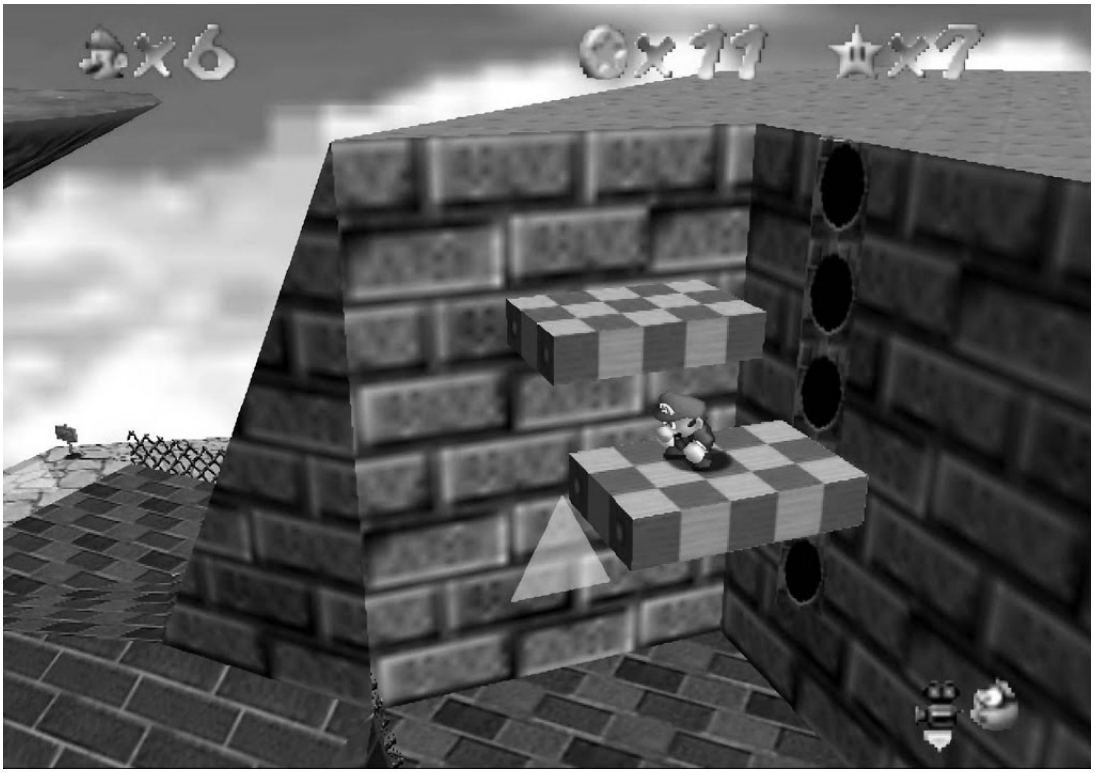


FIGURE 10. A corner of Whomp's Fortress from *Super Mario 64*. Image taken from "Super Mario 64: Whomp's Fortress (Chip Off Whomp's Block) [1080 HD]" by @GamerJGB via YouTube.

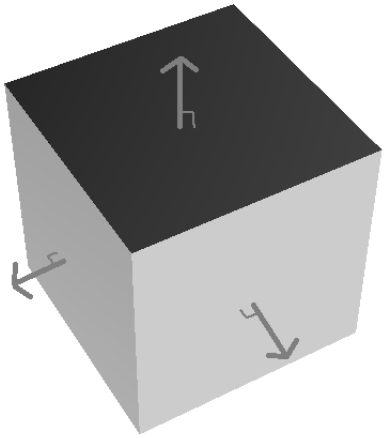


FIGURE 11. A box with corresponding normal vectors drawn for each visible face. The normal vectors protrude out perpendicularly from the surfaces.

In general, computing exact expressions for the gradients of SDFs can be difficult. Thankfully, gradients can be numerically estimated rather easily. When we numerically estimate derivatives, we take two points *close* to the point of interest and calculate the slope of the resulting secant line through those points evaluated by the function. Since a function's gradient is just a list of the function's derivatives in each

orthogonal direction (i.e. along the x-, y-, and z-axes), we can use the exact same technique to numerically estimate the gradient of our SDFs: for each orthogonal direction, take points *close* to our point on the surface of an object and calculate how the value of the SDF changes between those points (Walczyk).[9] If we store these values in a vector, we'll have a very good estimate of our gradient. If we normalise the gradient, we'll get a very good estimate to the normal vector to the surface of our object!

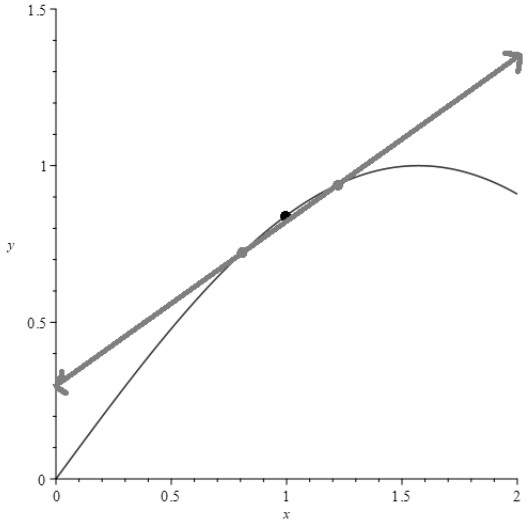


FIGURE 12. To estimate the derivative at the black point, we take points *close* to the black point and calculate the slope of the secant line running through them. Estimating gradients uses the exact same principle, just in multiple orthogonal directions.

The following code demonstrates how we may go about computing a normal vector in our example ray marcher:

```
NORMALSTEP = 10e-8;
XVECT = [NORMALSTEP 0 0];
YVECT = [0 NORMALSTEP 0];
ZVECT = [0 0 NORMALSTEP];
normalX = signed_dist(pixelRay + XVECT, objects) - ...
                  signed_dist(pixelRay - XVECT, objects);
normalY = signed_dist(pixelRay + YVECT, objects) - ...
                  signed_dist(pixelRay - YVECT, objects);
normalZ = signed_dist(pixelRay + ZVECT, objects) - ...
                  signed_dist(pixelRay - ZVECT, objects);
normalRay = [normalX, normalY, normalZ];
unitNormalRay = normalRay ./ sqrt(sum(normalRay.^2));
```

[9]The term "close" here is intentionally left vague. How close we take these points depends on how accurate our data types are, and how accurate we want our normal vector to be.

This code would be computed once our light ray touches a surface. Here, the variable `NORMALSTEP` is defining how close the estimate points are to the point on the surface for which we want to find a normal vector. The vectors `XVECT`, `YVECT`, and `ZVECT` are controlling which orthogonal direction we use to calculate the derivative of the SDFs.

So, when one of our light rays touches the surface of an object, we can use the above method to get a normal vector to the surface. Now what? If we want to calculate whether the surface of an object is pointing towards a light source, we should probably define a light source within our scene, separate from the light rays we shoot out from the viewing point. We can do this similarly to how we defined our scene's objects: using a list of light sources:

```
lights = {struct("lightType", "point", "center", [-5 -10 3])};
```

With a light source defined, we can now use a helpful operation from linear algebra to see how directly a surface is facing the light: the dot product. Given two unit vectors, the dot product of those vectors will return the cosine of the angle between them. If the vectors are facing in the same direction, the dot product will be 1. As they point further and further away, the dot product will decrease. Once the vectors are facing in opposite directions, the dot product will be -1. Thus, the dot product gives us an easy way to measure how closely a surface's normal vector points towards a light source.

More specifically, we'll take the dot product between the unit normal vector on a surface, and a normalised vector pointing from the surface to the light source. The dot product of these vectors will give us an indication as to how closely the surface is pointing to a light source. Importantly, if the dot product is less than zero (meaning the vectors are at least over right angles apart), then that means the surface is pointing *away* from the light, and so it shouldn't be illuminated. If the dot product is greater than zero, then we'll use that value to scale how brightly that surface is coloured in our final image. The code for obtaining the dot product might look something like the following:

```
dot = -1;
for i = 1:size(lights, 2)
        if lights{i}.lightType == "point"
                toLight = lights{i}.center - pixelRay;
                unitToLight = toLight ./ sqrt(sum(toLight.^2));
                dot = max(dot, sum(unitNormalRay .* unitToLight));
        end
end
```

Then, using the value of `dot`, we can update the code we used to colour the pixels of our image to incorporate point diffuse lighting:

```
image(col:col, row:row, :) = max([1 1 1]*dot, [0 0 0]);
```

The value of `dot`, then, scales how bright a shade of white we use to colour a point on a surface. With our simple diffuse lighting implemented, rendering our sphere from before produces the following image:

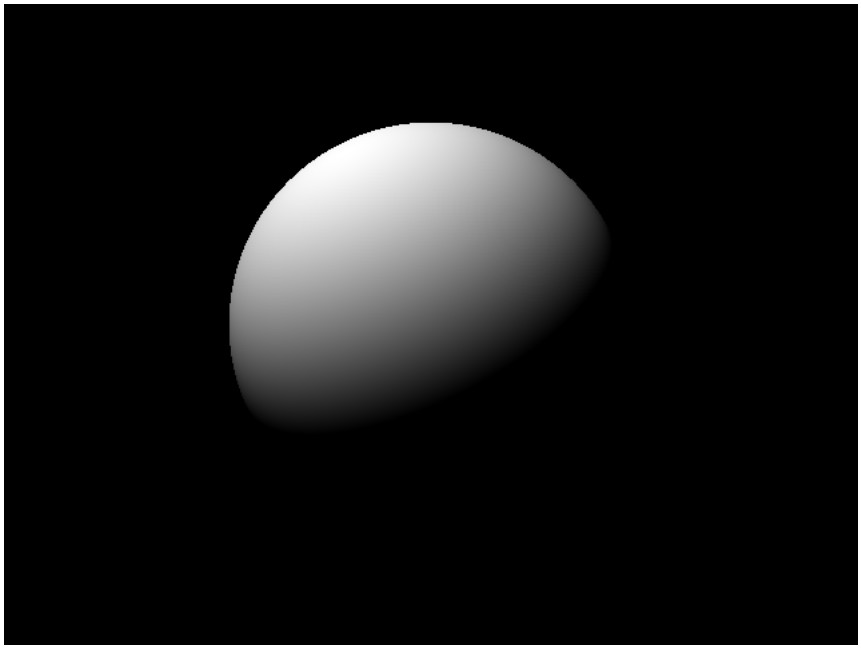


FIGURE 13. The same sphere from figure 9, rendered using diffuse lighting.

Because we have no ambient lighting, the parts of the sphere not facing the light are entirely enshrouded in darkness, as we'd expect. A simple way to implement ambient lighting is to change the second argument of the `max` function we use above to colour each pixel in the image. Rather than setting it to black (i.e. `[0 0 0]`), we could set it to a shade of grey (i.e. `[0.5 0.5 0.5]`) so that *all* parts of an object receive some light.

Before we talk about other ways to extend ray marching, it's worth briefly mentioning another type of lighting that is much easier to implement for ray marching than for other rendering algorithms: ambient occlusion. Put simply, ambient occlusion is a lighting effect that handles the proper lighting of corners within a scene (Garcia [2019a]). Typically, the corners of a room are darker than the walls themselves since, due to the geometry of the corner, less light rays are able to squeeze their way into the corner. With ray marching, we can use our SDFs to calculate how "cramped" a corner is by calculating how quickly the SDF increases as we move outwards from

the corner (Garcia [2019a]). If the SDF increases slowly, then the corner is very sharp, meaning it should receive very little light. If the SDF increases rapidly, then the corner is very wide, meaning it'll receive about as much light as the walls around it. This calculation can be done once we detect that a light ray is touching a surface, affecting how bright the resulting surface will be.

In general, lighting is one of the primary ways in which we add realism to a scene. After all, every environment in the real world requires some amount of light in order for us to see it. Another way to add realism is to use more complex objects in the scene. It isn't very often that we see mathematically perfect spheres and boxes floating around in space. Often, the objects we see in reality are imperfect, slanted, jagged, etc. How can we implement these sorts of shapes into our ray marcher?

This is where the "decoupling" between an object and its SDF comes in handy. As light rays are marching through a scene, they don't directly interact with any of the underlying objects. Their view of the scene is completely determined by the SDFs. If an SDF says a ray is close to an object, then it is. If we were to slightly modify an SDF to describe a slightly different shape, then the light rays would "see" that slightly modified shape (Walczyk). Importantly, this doesn't require changing anything about the original object. By modifying an SDF, we're essentially applying some sort of distortion to the object while leaving the underlying object itself intact. We can think of it like applying a filter over an object, a filter that can change the shape of an object. Or, perhaps more accurately, we can think of modifying SDFs as giving our light rays some sort of hallucinogenic substance that changes their perception of the world around them to be much more funky.

One easy distortion we can do to a sphere is making its surface ribbed. This can be done by adding an extra value to the output of the sphere's SDF that depends on the height of the input point. So, our SDF would look something like

$$\text{SDF}_{\text{sphere}}(\vec{x}, \vec{a}, r) = \sqrt{(x_1 - a_1)^2 + (x_2 - a_2)^2 + (x_3 - a_3)^2} - r + c_1 \sin(c_2 x_2),$$

where c_1 and c_2 are scaling constants that affect the amplitude and the frequency of the ribbing, respectfully. Using $c_1 = \frac{1}{16}$ and $c_2 = 30$, our sphere from before renders with a wavy surface (see figure 14).

Notice that, despite the sphere being distorted, the lighting is correct; points on the surface that are facing away from the light are still dark, while points facing towards the light are illuminated. This is a direct consequence of the decoupling between the underlying object and its SDF. Our lighting logic doesn't care what the object is supposed to be, it only cares about how the SDF describes it. So, if the SDF says

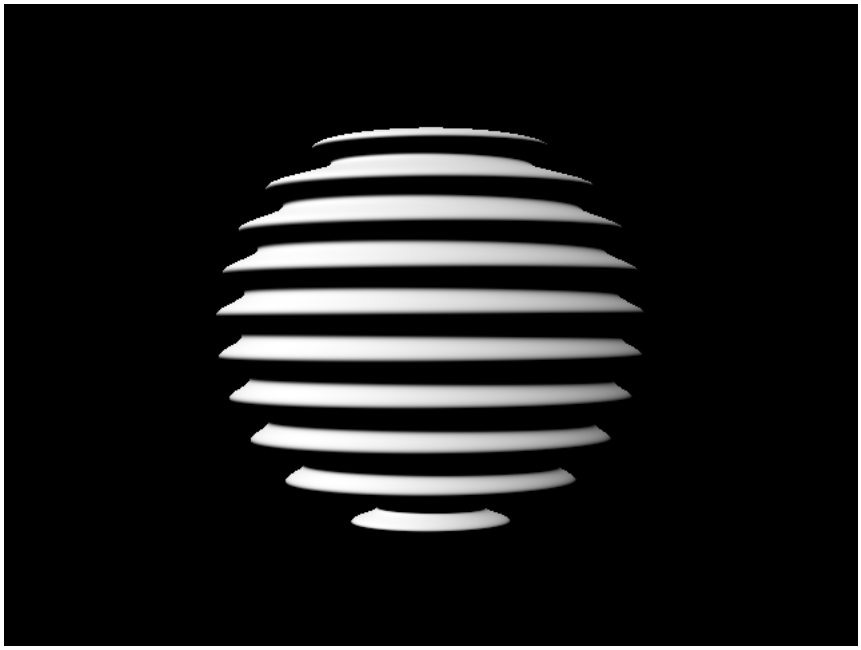


FIGURE 14. The same sphere from figure 13, with added "ribbing".

that the object has ribs, then everything in the code will behave as if the sphere has ribs.

Now, this lighting isn't 100% realistic since we haven't implemented any logic to check whether surfaces are blocking other surfaces from a light source, but with the lighting logic we *have* implemented, the ribbed sphere looks exactly as it should.

We can easily extend this SDF distortion technique to produce more realistic-looking objects. Adding a small random element to the SDF's output using something like Perlin noise can make an object look more "lumpy" and imperfect. Adding slight offsets to certain parts of an object can give the appearance of a "chipped" surface. With just a few slight modifications to an SDF, we can completely change the appearance of an object in some controlled way. This is yet another benefit to using ray marching over ray tracing. In ray tracing, these sorts of changes would have to be incorporated into an object's "definition" rather than an SDF, and this is much harder to do.

Another type of "distortion" we can do is combining objects. Technically, the scenes that we've been constructing up to now have been the unions of many different SDFs: given two or more objects, the resulting scene is a combination of all those objects. However, taking a union is not the only way to combine SDFs. For example, we could take the intersection of SDFs rather than the union (Quilez [a]).

What would this look like? Well, to take a union of SDFs (like we've been doing up to now), we compute the minimum distance that all the SDFs return and use this as our "distance to the scene". This makes intuitive sense. If one of our SDFs returns a small number, that means our light ray is close to that particular object, and so the ray is close to the scene. For an intersection of SDFs, a light ray shouldn't care whether it's close to *one* of the objects. A light ray would have to be close to *all* the objects if it's close to their intersection. So, to take an intersection, we'd take the *maximum* value our SDFs return and use that as our distance to the scene (Quilez [a]). Then, *all* our SDFs would have to be small in order for the light ray to be close to the scene, and so the light ray would interpret the scene as the intersection of our objects rather than the union.

This technique can be used to make "cuts" on an object. For example, consider the scene in figure 15. It shows an elongated box cutting through a sphere. If we were to take the intersection of these two SDFs, the outer parts of the sphere and box would be removed, leaving us with a sphere that's "sliced" along the edges of the box as in figure 16.

Other combinations we can take of SDFs include subtractions, XORs (exclusive or), smooth interpolations between SDFs, etc.[10] All of these provide alternative ways to create more complex shapes. Rather than trying to derive weird SDFs for sliced or combined objects, we can simply combine our SDFs using various operations!

[10]For some examples of these other operations, see (Garcia [2019a]) and (Quilez [a]).

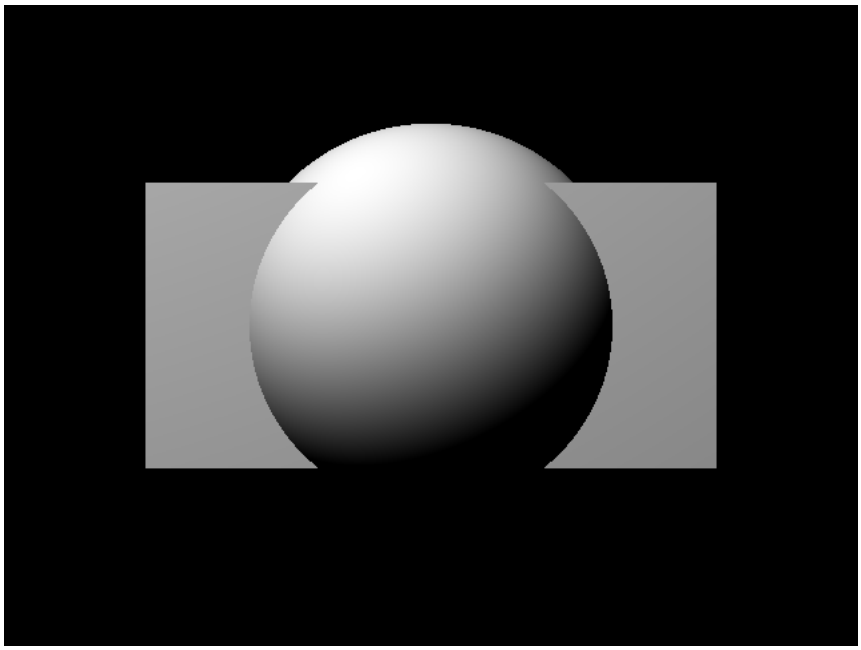


FIGURE 15. A box cutting through a sphere.

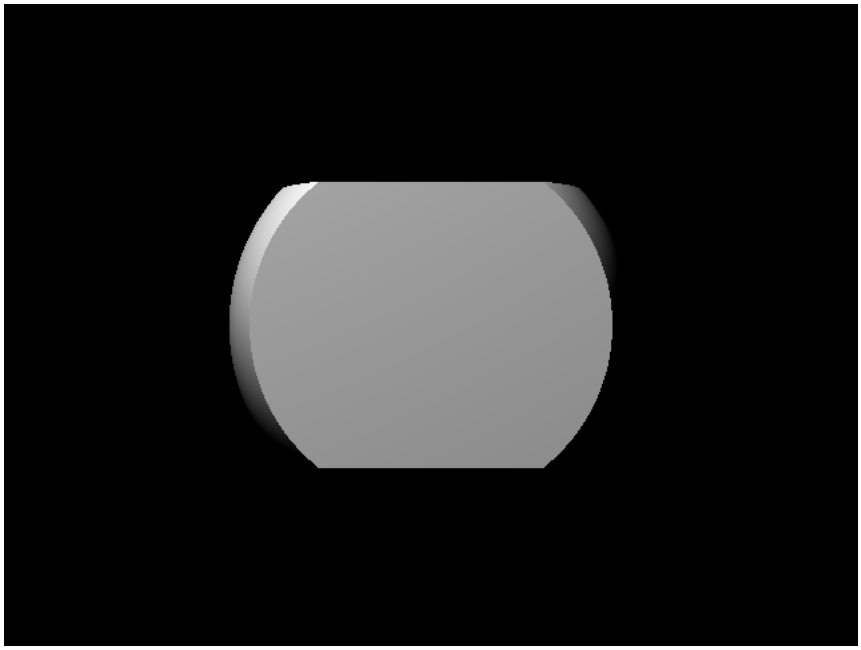


FIGURE 16. The same scene as in figure 15, except now the intersection of the SDFs is used rather than the union.

4.4. **Ray Tracing.** Up to now, we've focused our attention on understanding the ray marching algorithm. How does ray tracing differ? The two methods of rendering scenes are markedly similar: they both rely on describing objects mathematically, they both shoot light rays out into a scene, and they're both capable of producing incredibly detailed renders. The key difference between them is exactly how the light rays are shot out into the scene. While ray marching uses SDFs to "march" its rays into a scene, stopping them when the functions declare that the ray is touching an object, ray tracing *directly* calculates where a ray intersects with the scene (Walczyk). It does away with SDFs and instead works with the defining equations of the objects in the scene.

This approach is both an advantage and a disadvantage. The process of ray marching can be incredibly slow for complex scenes; thousands of SDFs may need to be calculated for each ray as it marches into an environment. Ray tracing, on the other hand, only needs a single batch of calculations for each ray rather than having to iterate over a "marching loop" countless times. The downside is that this batch of calculations can get much, much more complicated. We know from our math classes that finding where functions intersect is a nontrivial task, even for simple functions like polynomials. So, imagine trying to find intersection points in a three-dimensional space for complicated functions describing geometrical shapes. Things get ugly very quickly.

As well, because ray marching works directly with the defining equations for objects and not their SDFs, it isn't as easy to distort space as it was for ray marching, and so certain visual effects are harder to achieve with ray tracing. However, because ray tracing finds *exact* intersection points between rays and objects instead of points where an SDF is close to zero, ray tracing can often lead to more realistic-looking images, where reflections are more precise and shadows are more defined (Garcia [2019b]).

For completeness' sake, let's walk through a basic example of ray tracing a sphere to compare the process to ray marching.

As with our ray marcher, we'll start with a sphere of radius one centred at the point $(0, 0, 8)$. The equation whose solution points define this sphere is given by

$$x^2 + y^2 + (z - 8)^2 = 1.$$

Our viewing point will be at $(0, 0, 0)$. Like before, we'll shoot out light rays from our viewing point into the scene through some imaginary window. If a ray hits an object, we'll colour where the ray intersected our viewing window. Once we've shot out enough light rays to cover the viewing window, we'll use the window as our final image.

Here's where things begin to differ from ray marching. Since we want to compute the *exact* intersection between our sphere and our light rays, we should describe our light rays as *equations* rather than points in space. That way, we can solve for the intersection point in the usual way. One way to encode a ray (a line) in three-dimensional space is to find two vectors that are orthogonal to the line (and orthogonal to themselves), and solve for all the vectors that, when taking a dot product with the two vectors, return zero. Because all our rays are assumed to start at the origin (the viewing point), we don't need to worry about cases where the line doesn't cross the origin, which is a nice simplification. Let $\vec{\ell}$ be a vector giving the direction of a light ray, and let $\vec{a} = (a_x, a_y, a_z)$ and $\vec{b} = (b_x, b_y, b_z)$ be vectors orthogonal to $\vec{\ell}$ as well as themselves. Then the solutions to the system of equations

$$a_x x + a_y y + a_z z = 0$$
$$b_x x + b_y y + b_z z = 0$$

give us the ray we want.[11]

[11]There are numerous ways to encode a line in three-dimensional space. This way was chosen for this example as it gives equations similar to the ones we've dealt with up to this point.

The only challenge here is finding the vectors $\vec{a}$ and $\vec{b}$ given a light ray $\vec{\ell}$. The easiest way to do this is to slightly change the way we create our light rays from how we did previously. Before, we created our light rays by imposing a coordinate grid over our viewing window and then setting the tips of the light rays to the coordinates on the viewing window that corresponded to each pixel in our resulting image. In this way, we ensured that every pixel of our final image had a light ray to colour it. We'd still like to colour each pixel here, but if we use our previous method for doing so, then finding two orthogonal vectors to each light ray that are orthogonal to each other becomes a massive pain. Instead of directly using the coordinates of pixels on our viewing window, it turns out to be much easier to describe our light rays in terms of angles resulting from these coordinates.

Consider the unit vector with its tip at $(0, 0, 1)$ (so, the unit vector along the z-axis). We'll denote it as $\hat{z}$. By construction, the unit vectors with tips at $(1, 0, 0)$ (the vector $\hat{x}$) and $(0, 1, 0)$ (the vector $\hat{y}$) are not only orthogonal to $\hat{z}$, but are also orthogonal to each other. Thus, if we can find a way to rotate $\hat{z}$ so that it points in the direction of a light ray, then applying that same rotation to $\hat{x}$ and $\hat{y}$ will give us our two orthogonal vectors to define the ray algebraically.

Using our scheme from before, the x and y coordinates of a generic light ray's tip passing through the viewing window are given by

$$\ell_x = -\frac{w}{h} + \left(i + \frac{1}{2}\right)\hat{s}, \quad 0 \leq i < w,$$
$$\ell_y = -1 + \left(j + \frac{1}{2}\right)\hat{s}, \quad 0 \leq j < h,$$

where (w, h) gives the width and height of the final image (where the width is assumed to be at least as large as the height), and where $\hat{s} = \frac{2}{h}$. If we start at the origin, what angles would we need to tip $\hat{z}$ by in order to have it point to these coordinates on the viewing window? Let θ_x be the angle needed to rotate around the x-axis (which would tip $\hat{z}$ up and down), and let θ_y be the angle needed to rotate around the y-axis (which would tip $\hat{z}$ left and right). Using some trigonometry, we find that

$$\theta_x = \arctan\left(\frac{\ell_y}{Z}\right),$$
$$\theta_y = \arctan\left(\frac{\ell_x}{Z}\right),$$

where Z is the distance from the origin to the viewing window. So, using these angles and throwing them in some rotation matrices, we have that

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-\theta_x) & -\sin(-\theta_x) \\ 0 & \sin(-\theta_x) & \cos(-\theta_x) \end{bmatrix} \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ 0 & 1 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) \end{bmatrix}$$

gives us the rotation we need to apply to $\hat{z}$ to point it at a given pixel coordinate on the viewing window.[12] So, $A\hat{z}$ gives us a unit vector pointing in the direction of our light ray, and $A\hat{x}$ and $A\hat{y}$ are two orthogonal unit vectors that are also orthogonal to each other.

Now that we have our ray and our sphere encoded as algebraic objects, all that's left is to find the intersection points between the two. This can be done by solving the following system of equations:

$$\begin{aligned} a_x x + a_y y + a_z z &= 0, \\ b_x x + b_y y + b_z z &= 0, \\ x^2 + y^2 + (z-8)^2 &= 1, \end{aligned}$$

where (a_x, a_y, a_z) and (b_x, b_y, b_z) are the orthogonal vectors to our light ray. The easiest way to solve this is likely to isolate for x and y using the first two equations, then plug them into the third equation and solve the resulting quadratic for z, using that value to then get x and y. The expressions we'd get would be messy, but since it'd be the code solving them and not us, it isn't too big of an issue.

Note that, since we'd be solving a quadratic for z, we'll (in general) get *two* intersection points between the ray and the sphere: the point where the ray enters the sphere and the point where it exits. If we were going to extend the ray tracer to include reflections, shadows, etc., we'd need to add an extra check in our code to differentiate between these two points. This could be done by, for example, calculating the distance between the origin and the two intersection points and seeing which point is closer (and is thus the first place the ray hit on the sphere).

We can imagine that, had the object we were rendering been a box or something even more complicated, this process of solving for the intersection points would be rather difficult. It may even be impossible to solve for intersections using an exact formula if the resulting expressions are complex enough (e.g. if we had to solve a degree five or higher polynomial instead of a quadratic). In these cases, we'd

[12]Once again, the specifics of this rotation matrix may vary depending on the implementation and how exactly rotations are represented. The important part is that such a rotation matrix does exist.

resort to using something like Newton's method or fixed point iteration to solve the resulting equations. This is the crux of what makes ray tracing slightly more difficult than ray marching: while we don't need to repeatedly march a ray into a scene to find an intersection, we do need to solve some rather unwieldy expressions. Otherwise, the implementation of the two methods is very similar.

5. Applications

There are many, many applications for ray marching/ray tracing. Any time some three-dimensional environment needs to be rendered on a computer screen, chances are that ray marching/ray tracing can be used. However, seeing as there are literally thousands upon thousands of computer programs out there, we can't possibly list every single example of where these techniques can be used. Thus, we'll settle for a few notable use cases.

5.1. **Medical Imaging.** The medical field takes a lot of scans of the human body. MRI scans, x-ray scans, ultrasound scans, and retinal imaging are all examples. Sometimes, it's useful for a doctor or practitioner to be able to visualise these scans in a way that allows them to view them from different angles, under different lighting/colouring, etc. This sort of technology could make diagnosis easier for cases where visual symptoms are the main "tell" for whatever they're looking for.

So long as enough data is available, then ray tracing/ray marching can be used here to create accurate renders of whatever scan has been taken (see figures 17 and 18). More specifically, a modification of these techniques called *volumetric ray casting* is typically used, where the *volumes* of objects are used for renders, not just the surfaces as we did for our ray marcher (Adobe). However, the main techniques for rendering the scene remain the same.

5.2. **CGI Film/Animation.** Another obvious usage of ray marching/tracing is in 3D animation and film. Since ray marching/tracing offers more realistic lighting and rendering than other rendering techniques such as rasterization (Adobe), it's typically the method chosen by animation studios who want to create ultra-detailed scenes and objects for their projects. Though, ray marching/tracing is also suitable for smaller-scale projects, too (see figure 19). A super-powerful, million dollar computer isn't necessary for using these techniques![13]

[13]Though, it would definitely speed up the process, as creating the images for this project has proven...

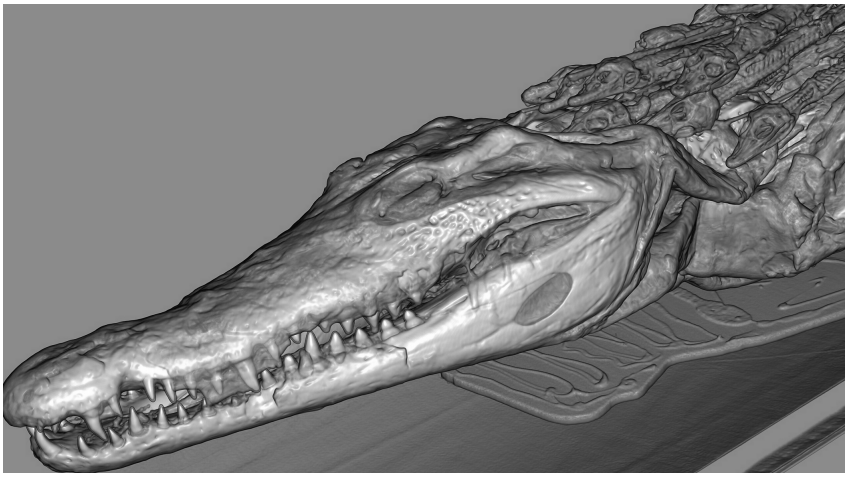


FIGURE 17. From Wikimedia Commons (public domain). A CT scan of a crocodile mummy with juveniles on its back, rendered using the High Definition Volume Rendering® engine (Fovia, Inc).

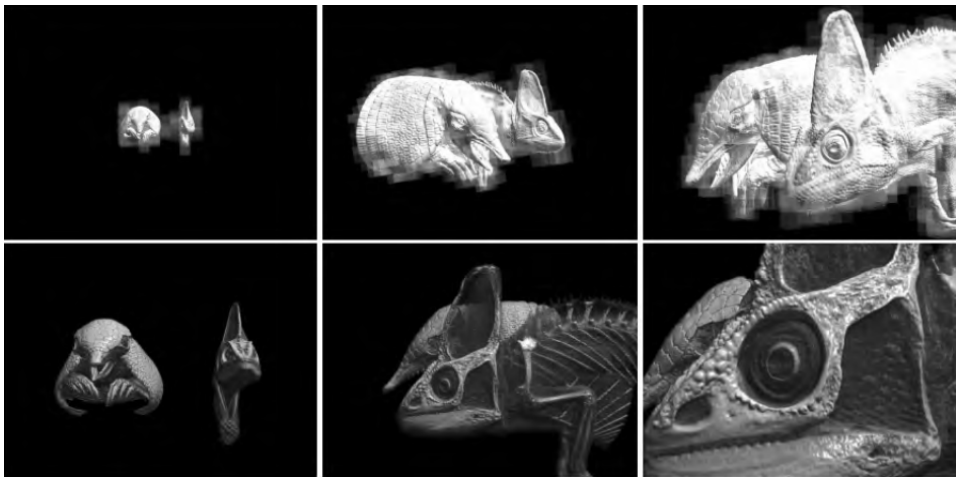


FIGURE 18. Figure 2.28 from Paulin et al. [2011], showing a visualisation of medical data by a volumetric ray caster.

5.3. **Video Games.** Perhaps the most common application of ray marching/ray tracing in recent years is in video games. Because ray marching/ray tracing offers more realistic lighting than other rendering techniques, and because it offers so many benefits over other methods (such as those benefits listed in section 4.3), it's been a popular choice for game developers who want to push the limits of what's graphically possible in gaming (Adobe).

Ray marching can also push the limits of what *objects* are possible within gaming. Inserting complex objects into a game world can be difficult for a variety of reasons. Rendering the object may be difficult (e.g. finding a suitable SDF may be cumbersome). As well, adding collision detection (which allows game objects to interact and collide with the object) may be computationally expensive. Imagine, as an example,

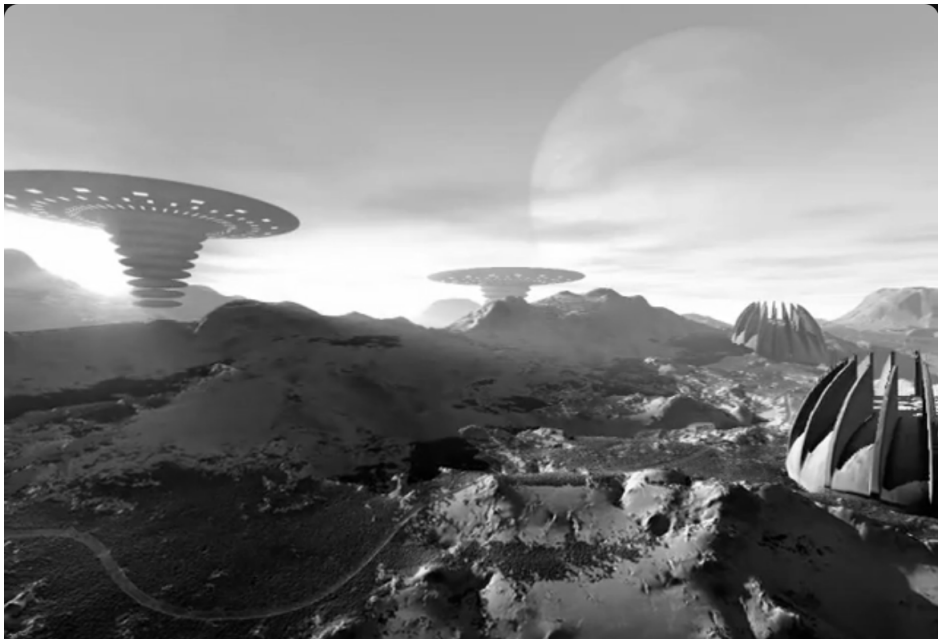


FIGURE 19. An example of the types of scenes possible for a regular person to whip up on a computer using ray marching. The animation this frame is taken from consists of only 4096 bytes of data! This frame comes from the animation "Ixaleno" (2008) by Inigo Quilez (https://youtu.be/XAWPCmVC5jA). The render was good enough to win first prize at the demoscene art show Breakpoint 2008, hosted in Bingen, Germany. See Quilez [b].

a bird flying through the branches of a tree. If we wanted to write code so that the bird would be able to land on any of these branches, we'd have to check *every single branch* for collisions between the branch and the bird's talons! Not only that, but we'd have to do this for *every frame* of the game (so, likely thirty times a second)! That's a lot of computation, especially for large trees with hundreds of intertwining branches! With the right setup, however, ray marching allows such complexities to be implemented.

A perfect example of ray marching being used for this purpose is in the game *Marble Marcher* (CodeParade [2019]). The premise of the game is simple enough: manoeuvre a marble through a series of obstacles towards a goal post. What makes this game special is its choice of obstacles. The environments on which the marble rolls are three-dimensional, dynamically changing fractals, rendered in real-time using ray marching. Even more impressive, the game's marble can actually roll around on this surface. The fractals are not only rendered, but are a physical object you can interact with in the game!

Ray marching is the key to making this game work. Because fractal objects are self-similar, the SDFs that define these objects will also, inevitably, possess some

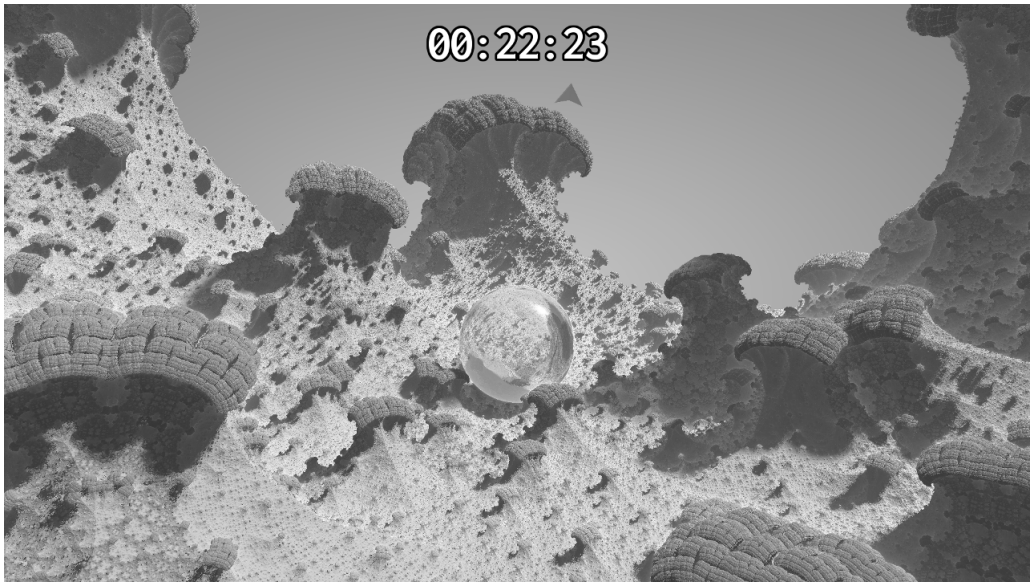


FIGURE 20. An example of the types of terrain encountered in *Marble Marcher*. Image taken from CodeParade [2019].

amount of self-similarity. So, rather than having to perform unique computations for *every* point on a fractal object, computations can be performed on a small subset of the object, then extrapolated to construct the entire environment!

We can demonstrate this principle in our own ray-marching code. Say we wanted to render an infinite grid of spheres, rather than the single sphere we've used up until now. Rather than try and program an infinite number of spheres into our code, we can make a slight modification to the coordinate space to achieve the same effect. Instead of letting our rays wander around the entire space, we'll confine them to a small box around the sphere. When a ray tries to exit this box, we'll "wrap" it around back into the box, sort of like the old video game Asteroids (where if you leave the screen on one edge, you'll reappear on the opposite edge). With this slight change, our basic sphere from before is suddenly duplicated off to infinity (see figure 21).

This same principle is used in *Marble Marcher*, just with more sophisticated techniques. *Marble Marcher* and its seamless integration of complex environments into its world is yet another example of the power of ray marching. Something which seems impossible to implement—infinitely-detailed fractal objects, complete with real-time rendering and collision detection—is made possible through ray marching. Even more impressive, ray marching allows games like *Marble Marcher* to run on any old computer, speaking to the efficiency of the method.

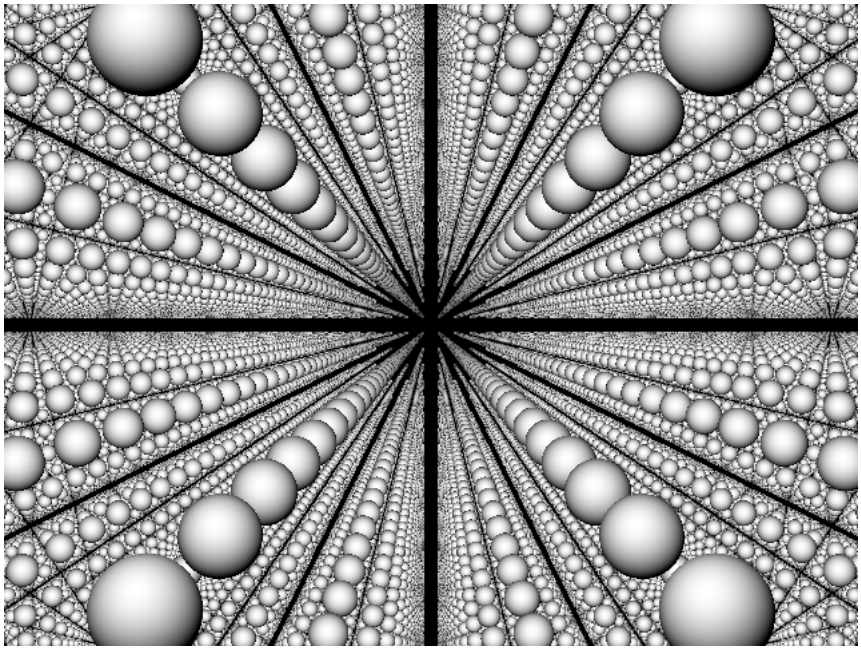


FIGURE 21. The same sphere as in figure 13, just with a modular reduction applied to the coordinates of our light rays. Note that the lighting isn't quite right in this render: the same lighting effect is applied to all spheres, even though they're in different points in space, and so the light should be hitting them in different places. This is simply a result of the way in which the modular reduction was applied for this example. The lighting can be corrected with a few slight modifications.

6. Conclusion

Hopefully, after following the implementation of our basic ray marcher, and after seeing some of the method's many, many applications, we can appreciate how a simple technique borrowed from Renaissance painters gives rise to some truly stunning imagery. By encoding objects in a scene as algebraic equations, and applying some formulas for computing distances between these objects, ray marching (and ray tracing) allow us to construct almost any scene we can imagine. Things like lighting, object distortion, and object replication are easily added to this basic framework, allowing more realistic images to be produced with far less effort than other rendering techniques would require. Harnessing this power has allowed us to make animated films, video games, detailed medical imagery, and, as is the case with centuries-old artists, realistic-looking paintings.

Three-dimensional rendering is an art form that'll only continue to grow in popularity, sophistication, and accessibility in the coming years. Advances in graphics cards, artificial intelligence, and mathematical theory are constantly pushing the

boundaries of what's possible. The things we can render on home computers nowadays were unimaginable only a few short years ago.[14] It's likely that this trend will continue: the things possible now with high-end PCs could potentially become the baseline for all computers in a few years. Though, we can be sure that ray marching/ray tracing will be involved, as no other method captures the properties of light quite so accurately, and no other method can be abstracted to mathematical models quite as elegantly.

Appendix A. MATLAB Code

dist_to_sphere.m:

```
function [dist] = dist_to_sphere(sphere, posVect, modular)
%DIST_TO_SPHERE Calculates the distance to a given sphere's surface.
%   Calculates the distance from posVect to the surface of the sphere
%   defined by sphereVect (the center) and sphereRadius (the radius). The
%   function throws an exception if the arguments given are of the
%   incorrect type (either 3D vectors for sphereVect and posVect or a
%   positive scalar for sphereRadius).

    %For demonstrating how an infinite grid of objects can be created
    if nargin < 3
        modular = false;
    end

    %Extracting properties from sphere
    sphereVect   = sphere.center;
    sphereRadius = sphere.radius;

    %Making sure parameters are correct
    if (size(sphereVect) ~= 3)
        throw(MException("Vector:DimensionError", "Sphere's center " + ...
            "is of incorrect dimension (should be 3)."));
    end
    if (size(posVect) ~= 3)
        throw(MException("Vector:DimensionError", "Given position " + ...
            "is of incorrect dimension (should be 3)."));
    end
    if (sphereRadius < 0)
        throw(MException("Scalar:RangeError", "Radius %s must be " + ...
            "positive.", sphereRadius));
    end
```

[14] Something like *Marble Marcher* would only be a dream to computer scientists of the 1990s!

```
    if modular == true
        posVect = mod(posVect, 10) - 5;
        posVect(3) = posVect(3) + 5;
    end

    dist = sum((sphereVect - posVect).^2)^(1/2) - sphereRadius;

    %Apply any relevant distortions
    for i = sphere.distortions
        if i == "ribbed"
            dist = dist + 2^(-4)*sin(30*posVect(2));
        end
    end
end
```

dist_to_box.m:

```
function [dist] = dist_to_box(boxCenter, boxSize, invBoxRot, posVect)
%DIST_TO_BOX Calculates the distance to a given box's surface.
%   Calculates the distance from posVect to the surface of the box
%   defined by boxCenter (the center), boxSize (the dimensions), and
%   invBoxRot (the rotation). The function throws an exception if the
%   arguments given are of the incorrect type.

    %Making sure parameters are correct
    if (size(boxCenter) ~= 3)
        throw(MException("Vector:DimensionError", "Box's center " + ...
            "is of incorrect dimension (should be 3)."));
    end
    if (size(boxSize) ~= 3)
        throw(MException("Vector:DimensionError", "Box's size " + ...
            "is of incorrect dimension (should be 3)."));
    end
    if (size(invBoxRot, 1) ~= 3 || size(invBoxRot, 2) ~= 3)
        throw(MException("Matrix:DimensionError", "Box's rotation " + ...
            "is of incorrect dimension (should be 3x3)."));
    end
    if (size(posVect) ~= 3)
        throw(MException("Vector:DimensionError", "Given position " + ...
            "is of incorrect dimension (should be 3)."));
    end
```

```
    %First, translate box to origin
    tempPos = posVect - boxCenter;

    %Now, we undo the box's rotation
    tempPos = (invBoxRot*tempPos')';

    %Calculate distances for points both inside the box and outside
    insideDist = min(abs(abs(tempPos) - boxSize));
    outsideDist = sqrt(sum(max(abs(tempPos)-boxSize, 0).^2));

    %Adding the sign to the output
    if outsideDist == 0
        dist = -insideDist;
    else
        dist = outsideDist;
    end
end
```

dist_to_cone.m:

```
function [dist] = dist_to_cone(coneBase, ...
    coneHeight, coneRadius, invConeRot, posVect)
%DIST_TO_CONE Calculates the distance to a given cone's surface.
%   Calculates the distance from posVect to the surface of the cone
%   defined by coneBase (the center of the base circle), coneHeight
%   (the height of the cone), coneRadius (the radius of the base) and
%   coneRot (the rotation). The function throws an exception if the
%   arguments given are of the incorrect type.

    %For deriving the SDF
    %https://www.desmos.com/calculator/ea4wwcfs4n

    %There are definitely more efficient ways to do this, but I wanted to
    % try and derive this myself.

    %Making sure parameters are correct
    if (size(coneBase) ~= 3)
        throw(MException("Vector:DimensionError", "Cone's base pos " + ...
            "is of incorrect dimension (should be 3)."));
    end
    if (size(invConeRot, 1) ~= 3 || size(invConeRot, 2) ~= 3)
        throw(MException("Matrix:DimensionError", "Cone's rotation " + ...
            "is of incorrect dimension (should be 3x3)."));
```

```
end
if (size(posVect) ~= 3)
    throw(MException('Vector:DimensionError', 'Given position ' + ...
        'is of incorrect dimension (should be 3).'));
end

%Flip cone rightside up
posVect(2) = -posVect(2);

%First, translate cone to origin
tempPos = posVect - coneBase;

%Next, we undo the box's rotation
tempPos = (invConeRot*tempPos')';

%Then, rotate the scene so that the point is on the positive x-axis
angToX = atan(abs(tempPos(3))/abs(tempPos(1)));
if (tempPos(1) < 0) && (tempPos(3) < 0)
    angToX = pi + angToX;
elseif (tempPos(1) < 0) && (tempPos(3) > 0)
    angToX = pi - angToX;
elseif (tempPos(1) > 0) && (tempPos(3) < 0)
    angToX = 2*pi - angToX;
end
invRot = [cos(angToX)   0  sin(angToX);
          0               1  0;
          -sin(angToX)  0  cos(angToX)];
tempPos = (invRot*tempPos')';

%tempPos should now be aligned with the x-axis now.
%Calculate distance to cone.

%If point is above cone tip
if tempPos(2) > coneHeight
    dist = sqrt(tempPos(1)^2 + (tempPos(2)-coneHeight)^2);

%If point is directly below cone
elseif (tempPos(2) < 0) && (tempPos(1) < coneRadius)
    dist = abs(tempPos(2));

%If point is below cone, but not directly below
elseif tempPos(2) < 0
```

```
        dist = sqrt(tempPos(2)^2 + (tempPos(1)-coneRadius)^2);

    %The general case
    else
        x_i = coneHeight + (coneRadius/coneHeight)*tempPos(1) - tempPos(2);
        x_i = x_i*coneHeight*coneRadius/(coneHeight^2 + coneRadius^2);
        y_i = (coneRadius/coneHeight)*(x_i - tempPos(1)) + tempPos(2);
        dist = sqrt((tempPos(1)-x_i)^2 + (tempPos(2)-y_i)^2);
    end
end
```

signed_dist.m:

```
function [d] = signed_dist(point, objects, mode)
%SIGNED_DIST Given a set of objects, computes a point's minimum
% signed distance to those objects.
%   d is the distance to the objects given from the point given.
%   objs is a set of objects in a scene. Each object is assumed to be a
%   struct with a field named "objectType" that tells the function what
%   kind of object it is. Based on this, the function will use the correct
%   function to compute the distance to it.
%   The function then computes the minimum of all these distances and
%   returns it.

    %Union is the default mode for the function
    %Thanks to Scott, Peter Mortensen, and simon from
    % stackoverflow.com/questions/795823.
    if nargin < 3
        mode = "union";
    end

    if mode == "union"
        d = Inf;
    elseif mode == "intersection"
        d = -Inf;
    end

    %Loop over all given objects
    for i = 1:size(objects, 2)
        %Getting distance to this particular object
        tempd = objects{i}.sdf(objects{i}, point);

        if tempd < d && mode == "union"
```

```
            d = tempd;
        elseif tempd > d && mode == "intersection"
            d = tempd;
        end
    end
end
```

step_vector.m:

```
function [newVector] = step_vector(v,unitV,stepSize)
%STEP_VECTOR Returns a new vector who stepped some number of units in the
%given vector's direction from the given vector's position.
%   Given a vector v and a stepSize, we return a new vector
%   newVector = v + stepSize*e,
%   where e is a unit vector in the same direction as v.
%   The function throws an error if v is the zero vector, or if the
%   given vector is a column vector.
    vectSize = size(v);

    %If we're given a column vector instead of a row vector
    if vectSize(1) > 1 && vectSize(2) ~= 1
        throw(MException("Vector:DimensionError", "Given vector " + ...
            "cannot be a column vector."));
    end

    %Now, let's actually compute the new vector
    newVector = v + unitV*stepSize;
end
```

scrap.m:

```
%Some basic ray marching code
%Heavily inspired by Michael Walczyk's tutorial at
% https://michaelwalczyk.com/blog-ray-marching.html.

%Inverted colours
INVERTEDCOLOURS = false;

%For calculating normal vectors
NORMALSTEP = 10e-8;
XVECT = [NORMALSTEP 0 0];
YVECT = [0 NORMALSTEP 0];
ZVECT = [0 0 NORMALSTEP];
```

```
%Colours
%BLACK = [0 0 0];
WHITE = [1 1 1];

%                       y    x
imageDimensions = [480 640];
image = zeros(imageDimensions(1), imageDimensions(2), 3);

if INVERTEDCOLOURS
    image = image + 1; %Making the scene white
end

%How far the imaginary screen is in our scene
distToFrame = 5;

%How far a ray should be able to go before we stop marching it
drawDistance = 1000;

%How close a ray needs to get to be considered touching an object
touchDist = 10e-4;

%All the objects in our scene
%Each sphere is represented as a coord for its centre and a radius
%Thanks to Dan, Amro, radarhead from stackoverflow.com/questions/18189227
% for helping solve the "this" problem
objects = {struct("objectType", "sphere", ...
                  "center", [-2 0 16], ...
                  "radius", 1, ...
                  "distortions", [], ...
                  "sdf", ...
                  @(t, p) dist_to_sphere(t, p))...
           struct("objectType", "box", ...
                  "center", [0 0 20], ...
                  "size", [1 1 1], ...
                  "rotation", [1 1 1/3], ...
                  "invRotMat", zeros(3, 3), ...
                  "sdf", [])...
           struct("objectType", "cone", ...
                 "center", [3, -0.5, 20], ...
                 "height", 4.5, ...
                 "radius", 1.4, ...
```

```
                    "rotation", [0.0 1.1 0.8], ...
                    "invRotMat", zeros(3, 3), ...
                    "sdf", [])};
%{
objects = {struct("objectType", "sphere", ...
                    "center", [0 0 12], ...
                    "radius", 1.5, ...
                    "distortions", [], ...
                    "sdf", ...
                    @(t, p) dist_to_sphere(t, p))...
           struct("objectType", "box", ...
                  "center", [0 0 12], ...
                  "size", [2 1 0.8], ...
                  "rotation", [0 0 0], ...
                  "invRotMat", zeros(3, 3), ...
                  "sdf", [])};
%}
%{
objects = {struct("objectType", "sphere", ...
                    "center", [0 0 8], ...
                    "radius", 1, ...
                    "distortions", {"ribbed"}, ...
                    "sdf", ...
                    @(t, p) dist_to_sphere(t, p))};
%}

%The light sources in our scene
lights = {struct("lightType", "point", ...
                 "center", [-5 -10 3])};

%How much things are lit up if they aren't directly hit by a light
ambientLight = 0.0;

%Do any necessary precomputation on our objects
for i = 1:size(objects, 2)
    if objects{i}.objectType ~= "sphere"
        %Precomputing inverse rotation matrices
        % then binding it to SDF
        objRot = objects{i}.rotation;
        zInvRot = [cos(-objRot(3)) -sin(-objRot(3)) 0;
                   sin(-objRot(3)) cos(-objRot(3))  0;
                   0               0                1];
```

```
        yInvRot = [cos(-objRot(2))   0  sin(-objRot(2));
                   0                 1  0;
                   -sin(-objRot(2))  0  cos(-objRot(2))];
        xInvRot = [1  0                  0;
                   0  cos(-objRot(1))  -sin(-objRot(1));
                   0  sin(-objRot(1))   cos(-objRot(1))];
        objects{i}.invRotMat = zInvRot*yInvRot*xInvRot;

        %Bind SDF to inverse rotation matrix
        if objects{i}.objectType == "box"
            objects{i}.sdf = ...
            @(t, p) dist_to_box(t.center, t.size, t.invRotMat, p);
        elseif objects{i}.objectType == "cone"
            objects{i}.sdf = ...
            @(t, p) dist_to_cone(t.center,t.height,t.radius,t.invRotMat,p);
        end
    end
end

%Getting relative scaling for the edges of the virtual screen
%Basically, how much farther one direction do we have to go over another
%when creating pixelRays?
percentDiff = max(imageDimensions)/min(imageDimensions);

%Is the image wider on the x or y direction?
if (imageDimensions(1) > imageDimensions(2))
    biggerDimension = "col";
    rowShift = -1;
    colShift = -percentDiff;
else
    biggerDimension = "row";
    rowShift = -percentDiff;
    colShift = -1;
end

pixelInc = 2/min(imageDimensions);

%Now, iterate over all pixels in our image,
% shoot out a ray through that pixel and see where it goes.
for col = 1:imageDimensions(1)
    for row = 1:imageDimensions(2)
        %Create directed ray through pixel
```

```
rowPlace = (row-0.5)*pixelInc;
colPlace = (col-0.5)*pixelInc;

%Do we need to worry about the zero vector here?
pixelRay = [rowShift+rowPlace colShift+colPlace distToFrame];
unitRay = pixelRay ./ sqrt(sum(pixelRay.^2));

%Keep marching until ray goes past our draw distance range,
% or it hits something.
while pixelRay(3) < drawDistance
    distToScene = signed_dist(pixelRay, objects);

    %If the ray is close enough, draw!
    if abs(distToScene) <= touchDist
        %Now, let's calculate some basic lighting

        %Get normal vector
        normalX = signed_dist(pixelRay + XVECT, objects) - ...
                  signed_dist(pixelRay - XVECT, objects);
        normalY = signed_dist(pixelRay + YVECT, objects) - ...
                  signed_dist(pixelRay - YVECT, objects);
        normalZ = signed_dist(pixelRay + ZVECT, objects) - ...
                  signed_dist(pixelRay - ZVECT, objects);
        normalRay = [normalX, normalY, normalZ];
        unitNormalRay = normalRay ./ sqrt(sum(normalRay.^2));

        %Find maximum dot product between unitNormalRay and lights
        %In order to do global shading, I'd have to run a second
        % loop through the ray marching algorithm here, just with
        % the normal vector
        dot = -1;
        for i = 1:size(lights, 2)
            if lights{i}.lightType == "point"
                toLight = lights{i}.center - pixelRay;
                unitToLight = toLight ./ sqrt(sum(toLight.^2));
                dot = max(dot, sum(unitNormalRay .* unitToLight));
            end
        end

        %Using dot calculated above, colour the sphere based
        % on the intensity of light hitting it.
```

```
                if ~INVERTEDCOLOURS
                    image(col:col,row:row,:)=max(WHITE*dot, ambientLight);
                else
                    image(col:col, row:row, :) = min(1-ambientLight, ...
                                                     1-WHITE*dot);
                end
                break;
            else %Otherwise, march
                pixelRay = step_vector(pixelRay, unitRay, distToScene);
            end
        end
    end
end

%Write out image data to file
imwrite(image, "test.png");

%Show the image
imshow("test.png");
```

References

Adobe. What is ray casting? URL https://www.adobe.com/products/substance3d/discover/what-is-ray-casting.html. [Online; accessed 28-October-2024].

CodeParade. Marble marcher, 2019. URL https://codeparade.itch.io/marblemarcher. [Online; accessed 02-November-2024].

Joey de Vries. Basic lighting. URL https://learnopengl.com/Lighting/Basic-Lighting. [Online; accessed 15-October-2024].

Gregory Hartman et al. *Calculus 3e (Apex)*. Virginia Military Institute, 3rd edition, 2024. URL https://math.libretexts.org/Bookshelves/Calculus/Calculus_3e_(Apex)/05%3A_Integration/5.03%3A_Riemann_Sums. [Online; accessed 05-October-2024].

Maxime Garcia. An introduction to raymarching, 2019a. URL https://typhomnt.github.io/teaching/ray_tracing/raymarching_intro/. [Online; accessed 15-October-2024].

Maxime Garcia. Ray tracing resources, 2019b. URL https://typhomnt.github.io/teaching/ray_tracing/raytracing_practs/. [Online; accessed 18-October-2024].

Morris Kline. Projective geometry. *Scientific American*, 192(1):80–87, 1955. ISSN 00368733, 19467087. URL http://www.jstor.org/stable/24943732.

Mathias Paulin, Rapporteur Toulouse, Wimmer, Jean-Michel Dischler, Carsten Dachsbacher, Miguel Sainz, Francois Sillion, and Fabrice Neyret. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, Jean Kuntzmann Laboratory, 07 2011. [Online; accessed 17-October-2024].

Inigo Quilez. distance functions, a. URL https://iquilezles.org/articles/distfunctions/. [Online; accessed 16-October-2024].

Inigo Quilez. demoscene, b. URL https://iquilezles.org/demoscene/. [Online; accessed 28-October-2024].

Upbeat Ruin. Texture mapping rule. Reddit, 2022. URL https://www.reddit.com/r/196/comments/11z3i1y/texture_mapping_rule/. [Online; accessed 05-October-2024].

Rohan Tulsiani. The making of mario in 3d. Illumin Magazine, 2019. URL https://illumin.usc.edu/the-making-of-mario-in-3d/. [Online; accessed 04-October-2024].

Michael Walczyk. Ray marching. URL https://michaelwalczyk.com/blog-ray-marching.html. [Online; accessed 12-October-2024].

Eric W. Weisstein. Analytic geometry. Wolfram MathWorld, 2024. URL https://mathworld.wolfram.com/AnalyticGeometry.html. [Online; accessed 30-September-2024].