Dedicatória: Texto no qual o autor do trabalho oferece homenagem ou dedica o seu trabalho a alguém.

AGRADECIMENTOS

Eduardo Dias de Melo Catarina

Quero agradecer, em primeiro lugar, a Deus, pela força e coragem durante toda esta longa caminhada.

Agradeço também a todos os professores que me acompanharam durante a gradução, em especial ao Prof. Dr. Dietrich Schiel e à Profa. Iria Müller Guerrini, responsáveis pela realização deste trabalho.

Dedico esta, bem como todas as minhas demais conquistas, aos meus amados pais (José Roberto e Tininha), minhas irmãs (Gina, Adalgisa e Lívia - Que falta vocês me fazem!!!) e meus dois preciosos sobrinhos (Lucas e Pedro Henrique - Meus melhores e maiores presentes...)

E o que dizer a você Paulo? Obrigada pela paciência, pelo incentivo, pela força e principalmente pelo carinho. Valeu a pena toda distância, todo sofrimento, todas as renúncias... Valeu a pena esperar... Hoje estamos colhendo, juntos, os frutos do nosso empenho! Esta vitória é muito mais sua do que minha!!!

Rafael de Aguiar Ferreira

Quero agradecer, em primeiro lugar, a Deus, pela força e coragem durante toda esta longa caminhada.

Agradeço também a todos os professores que me acompanharam durante a gradução, em especial ao Prof. Dr. Dietrich Schiel e à Profa. Iria Müller Guerrini, responsáveis pela realização deste trabalho.

Dedico esta, bem como todas as minhas demais conquistas, aos meus amados pais (José Roberto e Tininha), minhas irmãs (Gina, Adalgisa e Lívia - Que falta vocês me fazem!!!) e meus dois preciosos sobrinhos (Lucas e Pedro Henrique - Meus melhores e maiores presentes...)

E o que dizer a você Paulo? Obrigada pela paciência, pelo incentivo, pela força e principalmente pelo carinho. Valeu a pena toda distância, todo sofrimento, todas as renúncias... Valeu a pena esperar... Hoje estamos colhendo, juntos, os frutos do nosso empenho! Esta vitória é muito mais sua do que minha!!!

Epígrafe: É um item onde o autor apresenta a citação de um texto que seja relacionado com o tema do trabalho, seguido da indicação de autoria do mesmo. (texto iniciando do meio da página alinhado à direita)

Nome do autor da epígrafe

Palavras-chave: Palavrachave1. Palavrachave2. Palavrachave3.

ABSTRACT

The state's duty is not limited to a mere judicial response, but requires the provision of effective protection that meets the constitutional principle of reasonable duration of the procedure laid down in the Constitution. However, the delay in the delivery of legal protection remains one of contemporary evils of civil procedure.

Keywords: Keyword1. Keyword2. Keyword3.

LISTA DE FIGURAS

LISTA DE TABELAS

CONTENTS

1 INTRODUCTION

Computer graphics is a field of computer science focused on the generation, manipulation, and display of digital images. Since its inception, it has transformed various industries such as entertainment, architecture, and design, enabling the creation of virtual worlds and realistic environments. As machine processing increases and researchers constantly push the boundaries of technology, increasingly complex and detailed graphics have been made possible.

With advancements in rendering techniques and algorithms, methods such as Ray Tracing and Path Tracing [REFERENCIA] have emerged to accurately simulate the behavior of light, textures, and materials in three-dimensional environments. These methods are widely used in still images and animations, though they require significant computational power. Additionally, technologies like shaders [1], which allow customization at the pixel level, have become essential in gaming and simulation environments, delivering increasingly immersive visual experiences to users.

In this context, Ray Marching stands out as an innovative technique, particularly for real-time rendering of complex surfaces and fractals. Unlike traditional Ray Tracing, Ray Marching uses an iterative approach to detect surfaces within a Signed Distance Field (SDF) [REFERENCIA]. An SDF stores the shortest distance from any point in space to the closest surface, which allows it to describe smooth and detailed shapes, without needing traditional polygonal geometry. This technique is especially useful for rendering scenes involving implicit geometry, such as fractals and abstract scenarios, and is frequently applied in virtual reality environments and visual effects for games.

Ray Marching has found several practical applications in the gaming industry, particularly for rendering complex and dynamic scenes. In the Frostbite Engine [REFERENCIA], used in titles such as Battlefield V, Ray Marching is employed to render highly realistic volumetric clouds. This approach allows the engine to

[1] https://www.khronos.org/opengl/wiki/shader

[2] Source: Schneider et al.

Figure 1: Raymarched Clouds obtained by Schneider et al.[2]

simulate depth, light scattering, and other atmospheric effects in real time, creating lifelike skies that respond dynamically to environmental lighting conditions.

1.1 SCOPE AND OBJECTIVE

In this document, we explore Ray Marching as implemented in GLSL and compare its strengths to traditional ray tracing methods. Our primary objective is to demonstrate Ray Marching's unique capabilities, particularly its use of SDFs, which allow for precise control over shapes and blending effects that are challenging to achieve with ray tracing.

We also explore the potential of Ray Marching for procedural rendering, showcasing its use in creating complex noise patterns and highly realistic volumetric effects, including cloud rendering. By examining these features, we aim to highlight Ray Marching's versatility and its advantages in generating intricate, dynamic visuals for real-time applications.

[2]Source: https://media.contentapi.ea.com/content/dam/eacom/frostbite/files/s2016-pbs-frostbite-sky-clouds-new.pdf

1.2 METHODOLOGY

We take an iterative approach to explore the Ray Marching algorithm, showcasing its capacity to render smooth, noisy, and dynamic surfaces. Real-time 3D applications often rely on graphics APIs such as OpenGL, DirectX, or Vulkan for GPU programming. To simplify development and focus on Ray Marching itself, we use Shadertoy — a tool for running fragment shaders written in GLSL (OpenGL Shading Language) without the extensive setup. Fragment shaders are GPU programs that process each pixel on the screen, taking inputs such as the pixel position and canvas resolution.

1.3 TEXT STRUCTURE

O que cada capítulo fala. (pra depois)

2 RAY TRACING VS RAY MARCHING

One of the goals of computer graphics is to synthesize images of virtual scenes simulating the behaviour of light. In order to determine the visible surfaces described in the environment and the respective color of each pixel displayed on the screen, two of the most widespread techniques are rasterization and ray tracing. Rasterization, widely used in real-time applications like video games, prioritizes speed and efficiency. In contrast, ray tracing, commonly employed in high-quality 3D animations, sacrifices speed for photorealistic accuracy by simulating the paths of individual light rays. Beyond these methods, ray marching offers a unique approach, particularly suited for rendering complex implicit surfaces and volumetric effects. In this chapter, we will focus on ray tracing and explore how it compares and contrasts with ray marching, shedding light on their respective strengths and applications.

2.1 TRAVERSING THE CANVAS

A scene consists of a camera, 3D objects, and light sources placed in the environment. The goal when using ray tracing or ray marching is to generate a 2D render of the scene from the camera's point of view. We will need to traverse a 2D grid with the dimensions of our final image - the canvas. Then, given a camera point, we iterate over each position in the canvas, shooting a ray in its direction. This process is the basis of both ray marching and ray tracing.

For didactic purposes, we will ignore details such as the field of view, and we will assume that the distance between the canvas and the camera origin is always equal to 1, and that the canvas center will be positioned at $(0, 0, 0)$.

We are then able to establish the following definitions:

- **CanvasResolution**: A 2D vector where each component represents the canvas' width and height.

- **uv** is a mapping of the canvas' coordinate system, offsetting the $(0, 0)$ coordinate to the canvas' center and ranging between -1 and 1. We later divide it by the resolution to fix the aspect ratio.

- **RayOrigin**: A 3D vector representing the viewer's position, initialized at $(0, 0, 1)$.

- **RayDirection**: A normalized 3D vector representing the ray shot from the origin towards the pixel's position.

- **Hit**: A data structure storing information about the ray's interaction with geometry, including hit position, surface normal, distance from the origin, and data for lighting calculations.

The logic for traversing a 2D canvas and computing ray interactions with objects in the scene is outlined in **Algorithm 1 - Canvas Traversal**. This process applies to both ray tracing and ray marching, with the key difference being the implementation of the `TraceRay` function, used by ray tracing, which could be exchanged for `MarchRay`, used by ray marching.

[0]Source: https://developer.nvidia.com/discover/ray-tracing

Algorithm 1 Canvas Traversal

procedure RENDER(Canvas, Scene)
  for $y \leftarrow 0$ to $CanvasResolution.y - 1$ **do**
    for $x \leftarrow 0$ to $CanvasResolution.x - 1$ **do**
      uv $\leftarrow \left(\begin{bmatrix} x & y \end{bmatrix} - 0.5 * \text{CanvasResolution}\right) / CanvasResolution.y$
      RayOrigin $\leftarrow \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$
      RayDirection $\leftarrow$ NORMALIZE($\begin{bmatrix} uv.x & uv.y & 0 \end{bmatrix} - \text{RayOrigin}$)
      Hit $\leftarrow$ TRACERAY(**RayOrigin**, **RayDirection**, Scene)
      if Hit.isHit **then**
        HitColor $\leftarrow$ GETHITCOLOR($Hit, RayDirection, Scene$)
        SETPIXELCOLOR($x, y,$ **HitColor**)
      else
        SETPIXELCOLOR($x, y,$ BACKGROUND_COLOR)
      end if
    end for
  end for
end procedure

The pseudocode explicitly uses nested loops to traverse the canvas. In the GLSL implementation that follows, the iteration logic is unnecessary because the fragment shader operates on a per-pixel basis. Each invocation of the shader corresponds to a single pixel on the canvas, and the GPU automatically handles the parallel execution of the shader for all pixels. The input `fragCoord` provides the current pixel's coordinates, `vec2` and vec3 represent 2D and 3D vectors respectively, and GLSL functions handle vector operations like normalization and dot products.

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    Scene scene = createScene();

    vec2 uv = (fragCoord - .5*iResolution.xy) / iResolution.y;
    vec3 color = vec3(0.);

    // Creating Ray
```

```
    vec3 rayOrigin = vec3(0., 0., 1.);
    vec3 rayDirection = normalize(vec3(uv.xy, 0.) - rayOrigin);
    Ray ray = Ray(rayOrigin, rayDirection);

    // Tracing Ray
    Hit hit = traceRay(ray, scene);

    if (hit.isHit) color = getHitColor(hit, ray, scene);

    // Output to screen
    fragColor = vec4(color,1.0);
}
```

Code 1: Canvas Traversal

2.2 TRACING THE RAY

In ray tracing, once a ray is cast in the direction of a pixel on the canvas, it must traverse the scene to identify the closest object intersecting the ray. This requires a function capable of calculating the intersection between the ray and the geometric objects in the scene. The closest intersection point determines what will be rendered on the screen.

Algorithm 2 TraceRay

```
procedure TraceRay(RayOrigin, RayDirection, Scene)
    ClosestHit ← RAYHIT_INFINITY
    for each object ∈ Scene do
        Hit ← RayObjectIntersection(object, RayOrigin, RayDirection)
        if Hit.distance < ClosestHit.distance then
            ClosestHit ← Hit
        end if
    end for
    return ClosestHit
end procedure
```

In traditional computer graphics, most objects on a screen are represented as a collection of triangles. This allows any geometry tesselated into triangles to be described using ray-triangle intersection calculations. However, rendering smooth shapes requires an extremely fine mesh, leading to a high polygon count. This can create a significant performance bottleneck, especially in scenes containing thousands of objects, potentially resulting in millions of triangles.

In our example, we simplify the scene by using a single sphere. Since the mathematical formula for ray-sphere intersection is well-established[1], it allows us to render a smooth and precise representation of the sphere without the need for tessellation.

The `GetHitColor` function currently returns only the object's albedo. To simulate realistic lighting behavior, additional calculations are required, which will be detailed in Chapter 3.

By rendering a scene with a black background, a large white sphere below the camera to simulate the ground, and a smaller red sphere above it, the image shown in Figure 3 is produced.

2.3 MARCHING THE RAY

If the term 'trace' implies a fast, swift motion, 'march' is used to convey taking one step at a time. The ray marching algorithm uses the `MarchRay` function instead of `TraceRay`, so rather than calculating a direct intersection between the ray and the object, it resorts to iteratively marching through space, at each step sampling the distance to all objects in the scene, until one distance turns negative, or smaller than a specified threshold, which is considered a hit. The distance-sampling function is called SDF (Signed Distance Function), as it returns positive when outside the object, negative when inside. What is handy about SDF is that they are prone to various operations, chapter 3 will go though into some of them.

Ray marching can be seen as a family of solvers, and there are variations of

[1] https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection.html

this process, on questions such as how to quantify the step size between samples. A traditional approach uses a fixed step size, in which the ray traverses space at constant steps. This has many applications, although when trying to render light in solids such as the sphere in Figure 1, the ray must be backtracked (Figure 2), so as to obtain the exact collision point. Also, there is a chance of missing a collision altogether between samples (Figure 3), especially when dealing with finely textured or detailed objects.

To prevent the march from ever skipping a surface between steps, instead of fixed, the next step size could be picked as the minimum distance between all samples to an object. The ray then acts as if it has a sensor, advising to march slower when surfaces are nearby. When a distance becomes smaller than a specified threshold, it's considered a hit. This is observed in Figure 2, at each step the circle's radius is a 2D representation of the minimum distance to a surface and thus the step size. In Algorithm 4 is the method described in code, in it the procedure GetDistance, at each step, calls within it the SDF of each object in scene, returning the minimum distance.

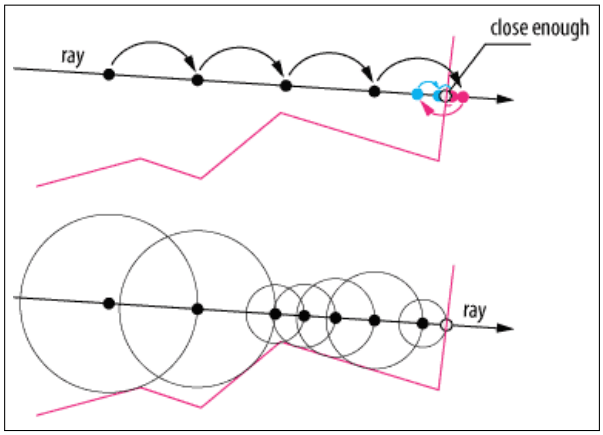


Figure 2: Above, binary search maneuver to bypass fixed step limitations, below a display of Sphere Tracing

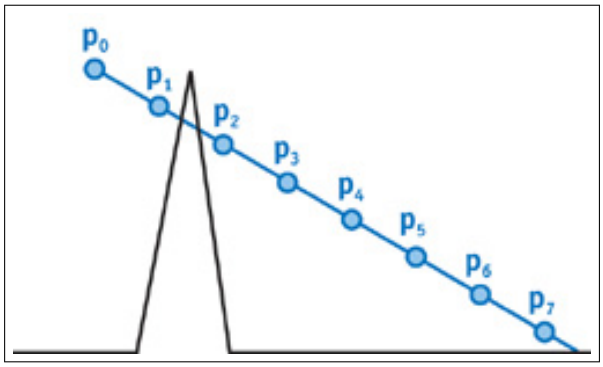


Figure 3: A problematic case of fixed step, from section 8.3 of [PF05]

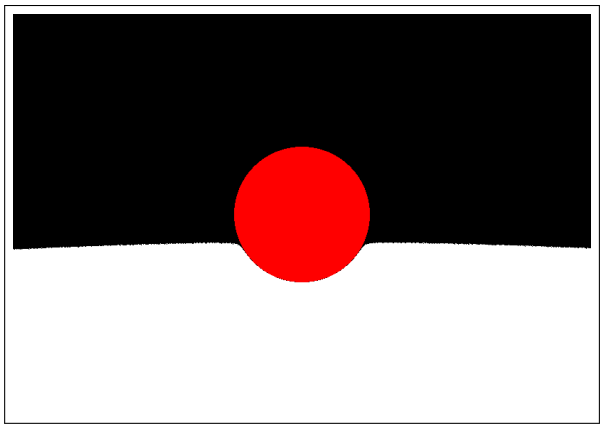


Figure 4: Unlit Raymarched Sphere

Algorithm 3 MarchRay

```
1:  procedure MarchRay(RayOrigin, RayDirection, Scene)
2:      MarchDistance ← 0
3:      Hit ← NULL_HIT
4:      IsHit ← false
5:      for step ← 0 to MAX_MARCH_STEPS − 1 do
6:          MarchPos ← RayOrigin + (MarchDistance ∗ RayDirection)
7:          Hit ← GetDistance(MarchPos, scene)
8:          if Hit.distance < SURFACE_DISTANCE then
9:              IsHit ← true
10:             break
11:         end if
12:         if MarchDistance > MAX_MARCH_DISTANCE then
13:             break
14:         end if
15:     end for
16:     return Hit
```

As to demonstrate a clear limitation of the method, Figure 4 was generated with a small 150 step limit. With close inspection, it's possible to observe that the far away ground plane starts to disappear close to the edges of the ball. This is because the ray, when very close to objects, tends to march slow, and so it takes more steps to get through their borders, and in doing so hits the step limit and fails to reach what's beyond, in the case of Figure 4 the ground. Fortunately, 150 steps is far from the optimal `MAX_MARCH_STEPS` number, a higher number can produce a featureless image with little additional computational costs, since most rays in the scene run into the ray distance limit sooner than 10 steps.

2.4 WHEN TO MARCH RATHER THAN TRACE

At first glance, when comparing both methods, ray tracing comes off as the best, most optimal one. Ray marching is prone to visual features and might take a lot of marching steps to produce the best image, especially in crowded scenes. In the landscape of real-time computing, many companies also prioritize ray tracing over ray marching. Nowadays, most of the 3d modeling done in software like Blender and Zbrush is not based on primitives such as spheres and boxes, but rather on just polygons. As such, any traditionally modeled scene, with high poly counts, textures, reflection, refraction, shadows, is better off using ray tracing, especially considering the evolution of path tracing, and NVIDIA RTX's elegant optimizations, mixing rasterization in.

Now, it doesn't mean ray marching has no competitive applications. Far from it, the flexibility of SDFs, and capability of diverse object combination and manipulation makes it particularly well-suited for rendering complex, mathematically defined shapes, such as fractals. Moreover, ray marching excels at rendering volumetric effects like clouds, fog, and smoke, which are inherently continuous and difficult to model with traditional polygonal geometry. By evaluating density functions at each step along a ray, ray marching can accurately simulate light scattering, absorption, and shadowing within these volumetric media.

Altough the demand for ray tracing specific hardware all but decreases, there is still a place for ray marching in real-time computing, even if it isn't suited for rendering high-poly solid objects. It's creative potential flourishes in communities like Shadertoy, where innovative ray-marched shaders showcase it's versatility. Beyond artistic applications, AAA games still leverage ray marching for realistic, dynamic effects like clouds characters fly through or smokes that bullets poke holes at. Additionally, in scientific fields, there are laboratories that apply ray marching on visualizations of all sorts, including 3D scalar fields, complex mathematical structures, such as Julia sets of quadratic functions and other quaternionic fractals. While its use cases may be more specialized, ray marching remains a powerful tool across both creative and technical domains, and hybrid solutions, using both marching and tracing, are widely adopted as to optimize the pros and cons of each method.

Figure 5: Chick

3 SIGNED DISTANCE FUNCTIONS TECHNIQUES

A *Signed Distance Function* (SDF) provides a way to describe geometric forms in a continuous manner by associating each point in space with its *signed distance* to a surface. The function returns positive value when the point is outside the shape and a negative value when the point is inside the shape. This framework is highly flexible, allowing the representation of shapes in 2D, 3D, and even N-dimensional spaces, since the Euclidean distance is not restricted to a particular number of dimensions.

A defining feature of Ray Marching is its use of signed distance functions to describe geometry, rather than traditional vertices, edges, and faces. This approach is precisely what makes Ray Marching a powerful technique, capable of rendering of smooth surfaces, shapes distorted by procedural noise, repetitive patterns, fractals, and dynamic shapes that evolve over time.

This chapter further describes signed distance functions for basic shapes and explores techniques for constructing complex shapes by combining and manipulating the fields to achieve the desired scene composition.

3.1 BASIC SHAPES

The general form of an SDF function includes:

- **p**, the **query point**, for which the distance from the shape positioned at the origin is computed.
- Shape-specific parameters, such as the **radius** for circles and spheres, or additional properties like **width, height, or orientation** for more complex geometries.

3.1.1 Circles and Spheres

For a circle centered at the origin in $\mathbb{R}^2$ with radius r, the SDF is given by:

$$\text{CircleSDF}(\mathbf{p}) = \|\mathbf{p}\| - r$$

A sphere in three-dimensional space follows the same principle. If the sphere is centered at the origin with radius r, the SDF becomes:

$$\text{SphereSDF}(\mathbf{p}) = \|\mathbf{p}\| - r$$

where $\mathbf{p}$ in $\mathbb{R}^3$ is now a point in three-dimensional space.

The only difference between the circle and the sphere is the dimension of the query point $\mathbf{p}$. In the case of a circle, $\mathbf{p}$ is a two-dimensional vector, whereas for a sphere, $\mathbf{p}$ is a three-dimensional vector. The formula remains structurally identical because both objects are defined by a center and a radius in their respective spaces.

```
float circleDistance(vec2 point, float radius){
    return length(point) - radius;
}

float sphereDistance(vec3 point, float radius){
    return length(point) - radius;
}
```

Code 2: Circle and Sphere SDFs

Since an SDF returns a positive distance when a point is outside the primitive and a negative distance when inside, this information can be used to assign different colors to a 2D shape rendered on the canvas.

```
vec3 plainColor (in float d){
     return vec3(1.0) - sign(d)*vec3(0.4);
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = (2.*fragCoord.xy-iResolution.xy)/min(iResolution.x,
    iResolution.y);

```

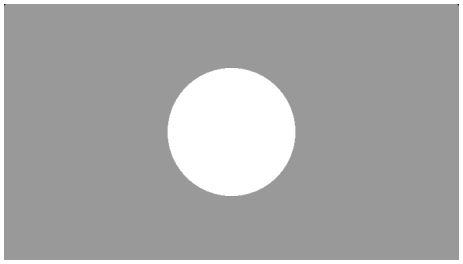


Figure 6: Plain Colored Circle

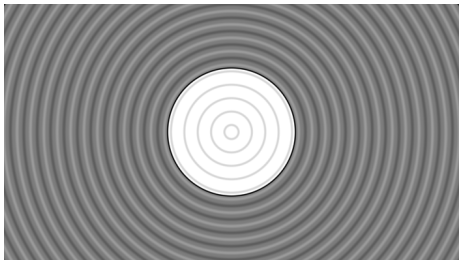


Figure 7: Custom Colored Circle

```
9     float d = circleDistance(uv, 0.5);
10
11    vec3 col = plainColor(d);
12    fragColor = vec4(col,1.0);
13 }
```

Code 3: Rendering Circles

In Figure 6, it is possible to see the color outside the circle being rendered in a darker shade than inside the circle. The implementation for such render can be seen in Code 3. For the following shapes, we will use a custom coloring function that adds isolines and highlights the border, as shown in Figure 7. This technique allows the visualization of 2D SDFs in the distance space.

3D Signed Distance Functions (SDFs) define the shapes of objects within a scene in the Ray Marching pipeline. For instance, the distance function of a sphere can represent the scene's minimum distance in the `createScene` function (see Code 1). Rays are marched accordingly, and collision information is passed to the `getHitColor` function. As previously discussed, this process closely resembles both Ray Marching and Ray Tracing techniques, including the lighting calculations used to shade the hit point. The following examples in this chapter illustrate 3D SDFs rendered with Lambertian diffuse shading and Phong specular reflection.

Since lighting models are outside the scope of this essay, readers may refer to Real-Time Rendering, 4th Edition by Akenine-Möller et al. for further details on shading techniques [AHH18].

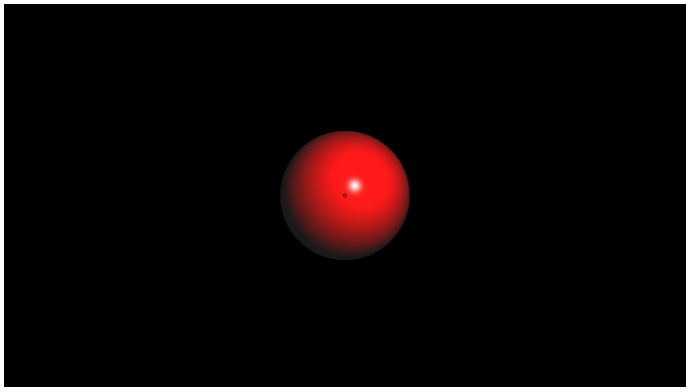


Figure 8: Raymarched Sphere SDF Visualization

3.1.2 Rectangles and Boxes

Similar to the 2D circle, the signed distance function (SDF) of a 2D box, or rectangle, can be generalized to higher dimensions.

```
float rectangleDistance( in vec2 p, in vec2 b ){
    vec2 d = abs(p)-b;
    return length(max(d,0.0)) + min(max(d.x,d.y),0.0);
}

float boxDistance( in vec3 p, in vec3 b ){
    vec3 d = abs(p)-b;
    return length(max(d,0.0)) + min(max(d.x,d.y,d.z),0.0);
}
```

Code 4: Rectangle and Box SDF

The rectangle SDF is defined at the origin $[\mathbf{0}, \mathbf{0}]$, and due to the rectangle's symmetry, it is often more convenient to work within the first cartesian quadrant. To achieve this, $\mathbf{b}$ (half each box's side) is subtracted from $\mathbf{abs}(\mathbf{p})$, yielding a point $\mathbf{d}$. As illustrated in Figure 9:

- If only $\mathbf{d.x} < \mathbf{0}$, it indicates that $\mathbf{p}$ is above the rectangle, in section 1, and the shortest distance is straight down.
- If only $\mathbf{d.y} < \mathbf{0}$, it means that $\mathbf{p}$ is next to the rectangle, in section 3, so the shortest distance is straight left.

- If both $\mathbf{d.x} < 0$ and $\mathbf{d.y} < \mathbf{0}$, the point is inside the rectangle, in section 4, thus the shortest distance is the closest of the two sides.
- Finally, if neither component is negative, the point lies diagonally outside the rectangle, in section 2, so the distance to the corner is the shortest.

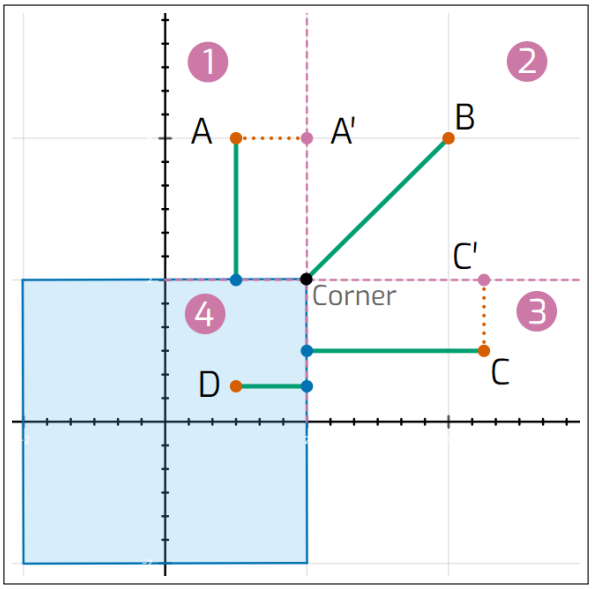


Figure 9: Graphic of Rectangle SDF calculation

It can be seen that when both $\mathbf{d.x}$ and $\mathbf{d.y}$ are limited to zero, which visually means snapping all points to section 2, the distance to the corner encases the results, although not entirely. If in section 4, inside the rectangle, $\mathbf{length}(\mathbf{max}(\mathbf{d}, \mathbf{0.0}))$ is always 0. In such cases, the SDF should return the negative distance to the nearest edge of the rectangle, adding the expression $\mathbf{min}(\mathbf{max}(\mathbf{d.x}, \mathbf{d.y}), \mathbf{0.0})$. The min() function ensures non-zero distances are only possible in section 4. With that, all sections are considered.

This process can be generalized to higher dimensions in a similar fashion. As such, the 3D box SDF can be derived by extending the approach to $\mathbb{R}^3$, where p and b are now three-dimensional vectors.

If expanded to $\mathbb{R}^4$, the SDF can help visualize forms such as the tesseract, the 4D hypercube, by fixing and scrolling through one of the four axes.

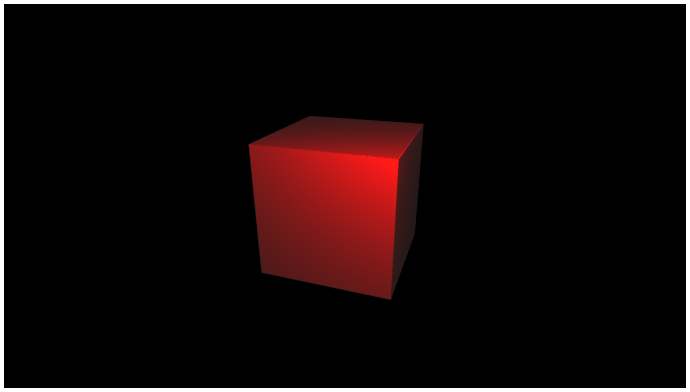


Figure 10: 3D Cube SDF Visualization

3.2 BASIC TRANSFORMATIONS

All primitive described in the previous section were centered at the origin. In order to perform a transformation such as scaling, rotation or translation, it is possible to apply the transformation at the query point **p**.

3.3 OPERATIONS

While being able to display a variety of primitives is great, both ray tracing and ray marching have such capabilities. As time goes by, what is required of modern computer graphics becomes much more than that, and in search of building more complex forms is where both methods part ways. While ray tracing depends on high polygon-count models, almost never using other types of primitives, ray marching finds more success with a variety of solutions, one such being the use of various combinations between primitives and operations as building blocks. This section describes some of the main SDF operations used in modeling based on basic primitives. Having primitives centered at the origin also facilitates many of the operations.

3.3.1 Union, Intersection and Subtraction

Operations between objects are some of the most elegant features of SDFs, and the most basic ones are the combinations: Union, Intersection and Subtraction. These three have very simple definitions and are great ways to introduce SDF operations. They take as input the SDF resulting distance of two primitive SDFs.

```
float unionOperation( float d1, float d2 )
{
    return min(d1,d2);
}
float intersectionOperation( float d1, float d2 )
{
    return max(d1,d2);
}
float subtractionOperation( float d1, float d2 )
{
    return max(-d1,d2);
}
```

Code 5: SDF Union, Intersection and Subtraction

From the perspective of a 3D raymarched vision, given **d1** and **d2**, SDF distances from objects 1 and 2:

- Taking the minimum between **d1** and **d2** is like seeing only the closest surface from both objects to camera, which is already the case for solids seen from the outside.
- Taking the maximum between **d1** and **d2** means that nothing outside the intersection of both objects is seen, since the ray march will only consider a hit if both SDF results are below the hit threshold.
- Taking the maximum between $-$**d1** and **d2** is like intersecting with the inverse of d1, which means that all $\mathbf{R}^3$ except object1. This results in the subtraction of objects 1 from 2.

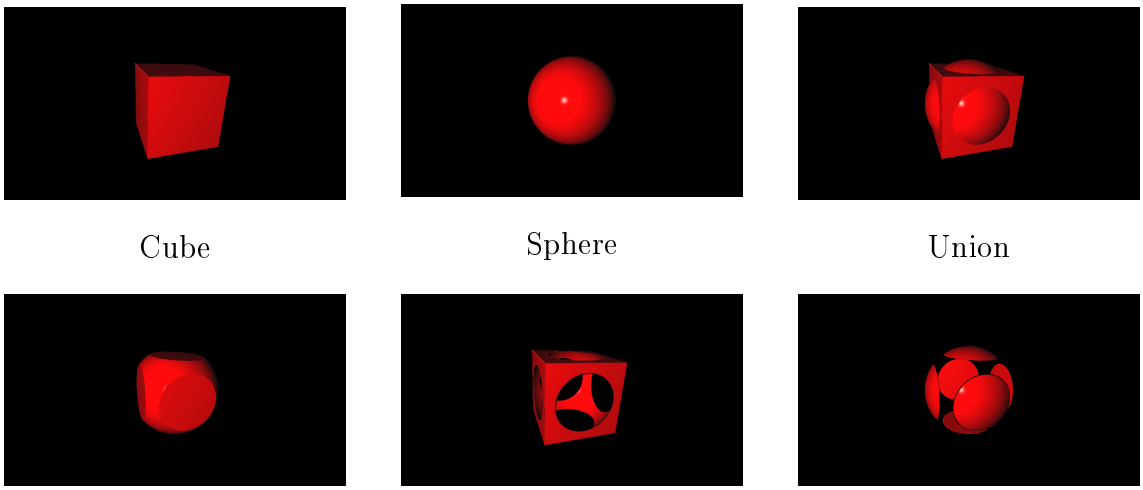


Figure 11: Raymarched Constructive Solid Geometry Operations with Cube and Sphere SDFs.

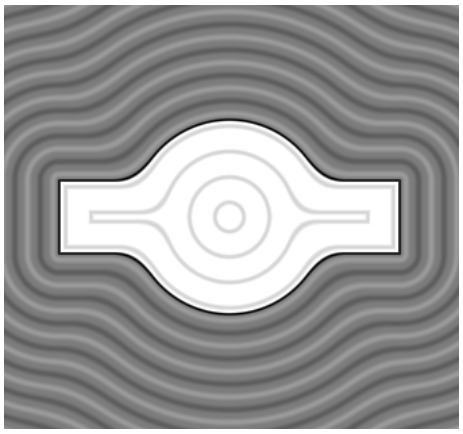


Figure 12: Smooth Union

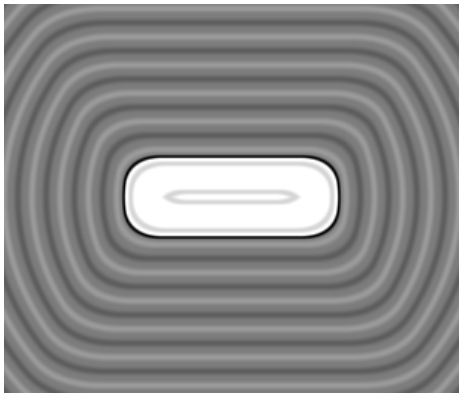


Figure 13: Smooth Intersection

3.3.2 Smooth Union, Intersection and Subtraction

A more complicated, yet visually appealing way to display the basic boolean operations is through the use of Smooth Minimum and Maximum. As the name implies, instead of binary hard edge transitions between the primitives, it offers a smooth blend. Figures 12, 13 and 14 combine the same box and circle with each smooth variant of the basic operations. This blending can be achieved through multiple types of functions, such as quadratic, cubic, exponential, sigmoid, each having it‘s pros and cons.

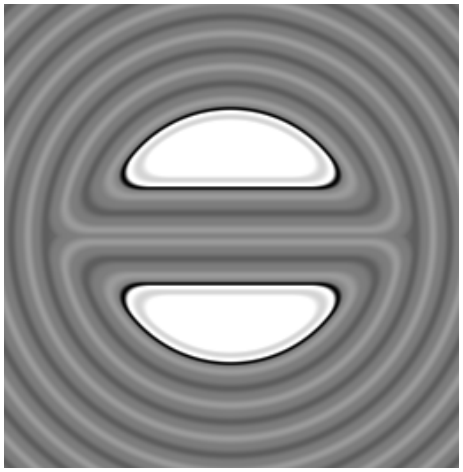


Figure 14: Smooth Subtraction

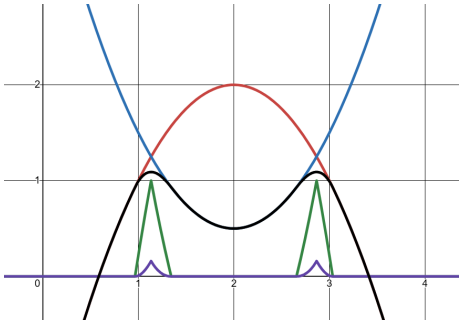


Figure 15: Smooth Minimum Representation

Because it is fast and does not overestimate distances, the quadratic polynomial is less prone to visual artifacts and is the most widely used version for ray marching, thus being the one this section focuses on. Similarly to the regular min function, smin (smooth minimum) takes as input **a** and **b**, which could be distances, or points in space, plus the threshold **k**. The blending starts where the distance between **a** and **b** is smaller than **k**.

Figure 15 will help navigate the explanation ahead, $\mathbf{a} = -(\mathbf{2}-\mathbf{x})^{\mathbf{2}}+\mathbf{2}$ is in red and $\mathbf{b} = (\mathbf{2}-\mathbf{x})^{\mathbf{2}} + \mathbf{.5}$ in blue. To find smin, all polynomial methods start by finding the $\mathbf{h}()$ function, which is 0 by default, and ramps up to 1, starting at $|\mathbf{a}-\mathbf{b}| = \mathbf{k}$ and peaking at $|\mathbf{a}-\mathbf{b}| = \mathbf{0}$:

$$h = \frac{\max(k-|a-b|, 0.0)}{k} \tag{3.1}$$

Since **k** is the distance in which the shapes start to combine, $\mathbf{h}()$ helps delimitate exactly that, as represented in green, Figure 15. Next, there needs to be a **c** such that $\mathbf{min}(\mathbf{a},\mathbf{b}) - \mathbf{c}$ produces the smoothed curve. This is no random requirement, but a simplified structure of what is the DD (Direct Difference) family of smin solutions. For polynomials, a simple solution for **c** is as follows, with $\mathbf{n} > \mathbf{1.5}$ already producing great results:

$$c = \frac{h^n k}{2n} \tag{3.2}$$

Between all values of **n**, the quadratic version ($\mathbf{n = 2}$) produces the gentler of curves, and is prioritized over the others. To better visualize what **c** means, Figure 12 represents **c** in purple. Squaring **h** makes c ease into 1, instead of following a straight ramp. Multiplying by **k** compensates any variation, as the peak of **h** decreases as **k** increases. Lastly, the divisor helps position the peak to the right place. To achieve smooth maximum, all that is needed is $\mathbf{max(a, b) + c}$.

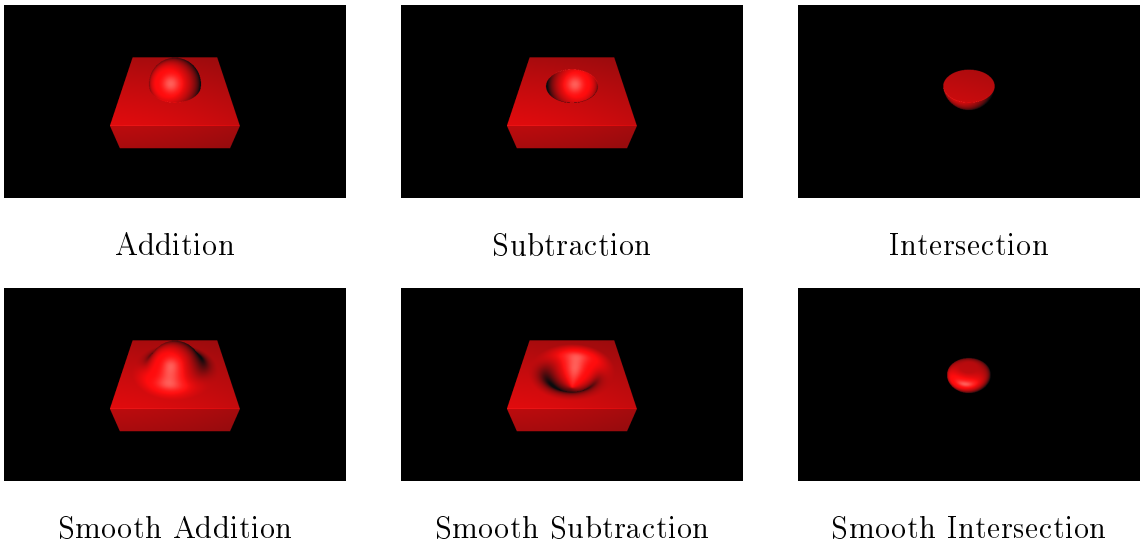


Figure 16: Raymarched Smooth Operations with Cube and Sphere SDFs.

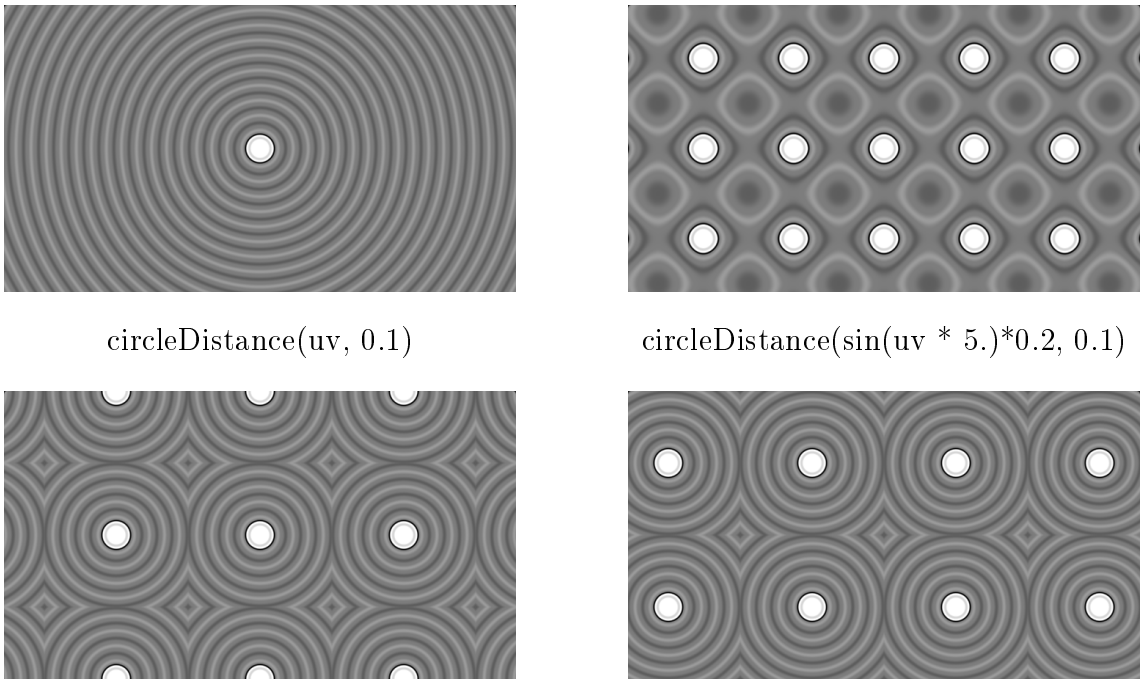
3.3.3 Domain Repetition

Since SDFs are expressed as mathematical functions, they can be transformed to repeat periodically across space. This technique, known as domain repetition, allows for the creation of infinite, repeating patterns in a scene while also optimizing performance by avoiding the need for multiple individual primitives.

There are many variations of this technique, as any domain parametrization can be made repeating, but the core idea is to use mathematical operations that will result in the desired scene.

This can be achieved through the natural repetitive pattern of trigonometric

functions such as $sin(x)$ and $cos(x)$, however, bending the domain with these functions will compromise the accurate Euclidean distances of the space the SDFs relies on, and this might generate unwished features. Functions such as $round(x)$ or $mod(x)$ (modulus operator) are more commonly used instead.



circleDistance(uv, 0.1)

circleDistance(sin(uv * 5.)*0.2, 0.1)

circleDistance(uv - round(uv), 0.1)

circleDistance(mod(uv, 1.)-0.5, 0.1)

Figure 17: Examples of Domain Repetition techniques.

Given that **uv** represents the query point in the domain, with the zero vector centered on the canvas, Figure 17 illustrates the output of the custom coloring function applied to its respective transformation. Naturally, this can also be used in 3D: Given that **p** represents the query point in the domain, the repeated domain can be represented as $\mathbf{p} - gap \cdot \text{round}\left(\frac{\mathbf{p}}{gap}\right)$, where gap is the size of the repeated domain box. Figure 18 shows an example of repeated spheres in a repeated domain.

3.3.4 Displacement

Displacement is a technique used to adjust the geometry to simulate bumps or irregularities, adding fine detail to the surface. This is often done by perturbing the distance field using patterns that look random, such as static or grain, which can

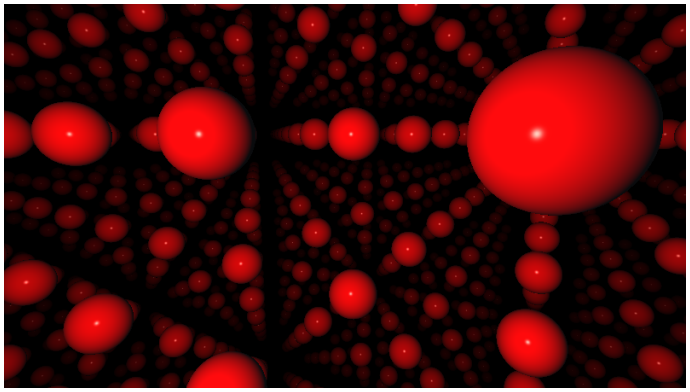


Figure 18: sphereDistance(p - 6.5*round(p/6.5), 1.0)

be generated with mathematical formulas (procedural noise) or taken from images (noise textures).

```
float displacementOperation( in sdf3d sdfShape, in vec3 p )
{
    float d1 = sdfShape(p);
    float d2 = displacement(p);
    return d1+d2;
}
```

Code 6: SDF Displacement

Although noisy functions are commonly used for displacement, any function can be applied to distort the distance field. This flexibility enables the creation of non-Euclidean effects, such as bending or twisting space. However, this comes with a caveat: the sum of an SDF and an arbitrary function is not necessarily an SDF.

A key property of 2D and 3D Signed Distance Functions is that their gradient has length 1.0 everywhere in the space. Since SDFs measure distances, the rate of change of distance must be equal to 1.0. When two SDFs are added, this property is violated — the gradient no longer has unit length, effectively "bending" space. This can cause the raymarcher to underestimate or overestimate the true distance from the surface.

This issue can be attenuated by reducing the step size and increasing the number of marching iterations, though this comes at a performance cost. In practice, small deviations in the length of the gradient rarely produce noticeable artifacts, making

displacement through function addition still useful for subtle surface displacement.

```
float distortedSphereDistance(vec3 p){
    float sphereRadius = 2.0;
    vec3 sphereCenter = vec3(0., 2., -5.);
    float sphereDist = length(p - sphereCenter) - sphereRadius;

    vec2 uv = sphereUV(point, sphereCenter); // stores spherical coordinates angles on uv
    float distortion = sin(uv.x * 64.) * cos(uv.y * 64.); // wave pattern according to the uv
    sphereDist += distortion * 0.1; // applying the distortion to the sphere distance

    return minDist;
}
```

Code 7: Wave Distortion on Sphere

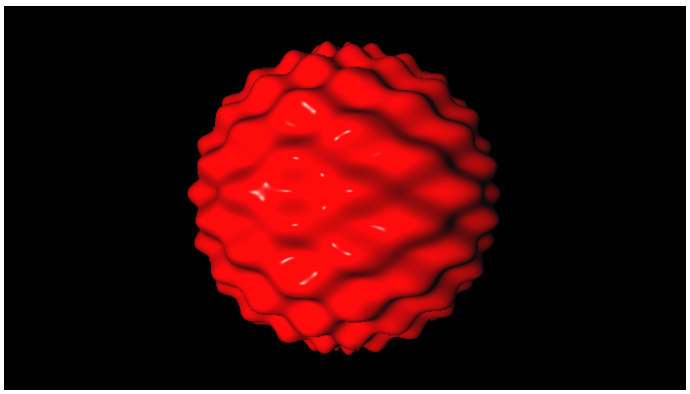


Figure 19: Distortion on Sphere from Code 7

In Code 7, the Sphere Distance is distorted according to the trigonometric functions applied on its polar and azimutal angle. The result is shown in Figure 19.

4 CLOUDS

One of the main applications of Ray Marching is in the rendering of realistic smoke, clouds, which are called volumetric objects. Clouds are mostly made of two main high-albedo types of particles, if not considering dirt or other minimal particles: air and water/ice. [Ril+04] Since clouds are not solid, when light hits them, some rays bounce, but some go through the object. Not only that, the water particles are anisotropic[Ril+04], which means that light tends to bounce differently depending on the direction from which it hits the cloud. All in all, there is a lot to adapt from the original Ray Marching concocted.

4.1 VOLUMETRIC RENDERING

Firstly, to render gaseous materials, instead of avoiding going into surfaces, Ray Marching needs to go through the clouds. For that, the SDF now represents density instead of "distance to surface". For the sake of clarity, the SDFs are inversed so that the distance to a point is as follows.

- Outside of the primitive, yields a negative density, or no density.
- Inside of the primitive, yields a positive density, and the further the point is from bounds, the higher the density is.

Secondly, to consistently sample the density along the cloud, a constant step size is better suited - instead of querying the scene's distance for the next marching step, each step will perform a constant size.

Algorithm 4 MarchRay

```
1:  procedure MarchRay(RayOrigin, RayDirection, Scene)
2:      MarchDistance ← 0
3:      Hit ← NULL_HIT
4:      IsHit ← false
5:      for step ← 0 to MAX_MARCH_STEPS − 1 do
6:          MarchPos ← RayOrigin + (MarchDistance ∗ RayDirection)
7:          Hit ← GetDistance(MarchPos, scene)
8:          if Hit.distance < SURFACE_DISTANCE then
9:              IsHit ← true
10:             break
11:         end if
12:         if MarchDistance > MAX_MARCH_DISTANCE then
13:             break
14:         end if
15:     end for
16:     return Hit
17: end procedure
```

4.2 NOISE

4.3 VOLUMETRIC SHADING

5 CONCLUSION AND FUTURE WORKS

5.1 *

References

[AHH18] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. 4th. A K Peters/CRC Press, 2018. ISBN: 9781138627000.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. ISBN: 0321335597.

[Ril+04] Kirk Riley et al. "Efficient Rendering of Atmospheric Phenomena". In: *Eurographics Workshop on Rendering*. Ed. by Alexander Keller and Henrik Wann Jensen. The Eurographics Association, 2004. ISBN: 3-905673-12-6. DOI: /10.2312/EGWR/EGSR04/375-386.

GLOSSÁRIO

Palavra	Significado da palavra
Palavra 2	Significado da palavra 2

ANEXOS

ANEXO A – Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec lacus nisl, ultricies vitae semper eu, scelerisque nec enim. Curabitur posuere tortor orci, at porta leo laoreet et. Quisque ut congue dolor. Maecenas vel sagittis diam. Praesent fermentum eleifend mi, sit amet vehicula leo pellentesque quis. Curabitur mattis luctus pulvinar. Proin auctor est nec nulla pellentesque commodo. Donec nec justo eu magna aliquet eleifend. Curabitur tristique tortor id sem dignissim, a iaculis metus interdum. Phasellus bibendum velit sit amet interdum semper. Nam vestibulum dui quis nisi consectetur, id vehicula dolor faucibus.

ANEXO B – Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec lacus nisl, ultricies vitae semper eu, scelerisque nec enim. Curabitur posuere tortor orci, at porta leo laoreet et. Quisque ut congue dolor. Maecenas vel sagittis diam. Praesent fermentum eleifend mi, sit amet vehicula leo pellentesque quis. Curabitur mattis luctus pulvinar. Proin auctor est nec nulla pellentesque commodo. Donec nec justo eu magna aliquet eleifend. Curabitur tristique tortor id sem dignissim, a iaculis metus interdum. Phasellus bibendum velit sit amet interdum semper. Nam vestibulum dui quis nisi consectetur, id vehicula dolor faucibus.