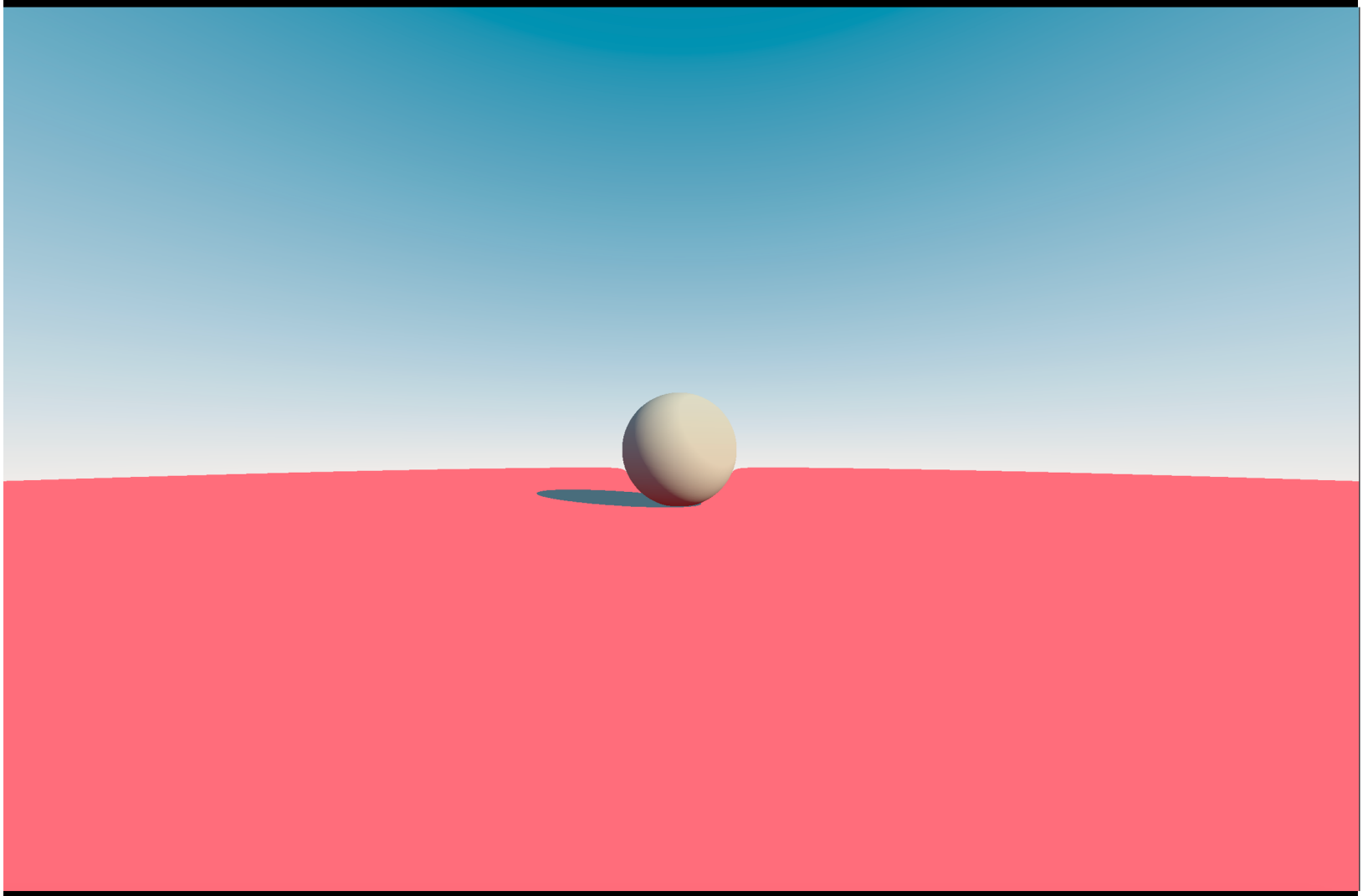
Procedural Generation using Ray Marching



Muhammad Ahmad

Computer Graphics Through Ray Marching: From Equations to Images

Muhammad Ahmad

This book is available at
https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields

This version was published on 2026-04-10



This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Muhammad Ahmad

Contents

1. Introduction 1
2. Lighting: Introduction 13
3. Lighting: Cast Shadows 14
4. Lighting: Diffuse 15
5. Lighting: Indirect 16
6. BRDF - Diffuse 17
7. BRDF - Specular 19

1. Introduction

We will start our journey by talking about an object many of us have seen; our planet has a *similar* shape to this object. If you guessed sphere, you were correct!

Spheres are all around us: from the many different kinds of balls used in sports to ball bearings in your bicycle. However, not many of us ever think about what makes a sphere a sphere? Would you be able to define an equation for the surface of a sphere? Have a go at this before moving on.

Theory: Describing a sphere

If you think about a white sphere on a black screen **without any lighting**, you are probably imagining something that looks like this:

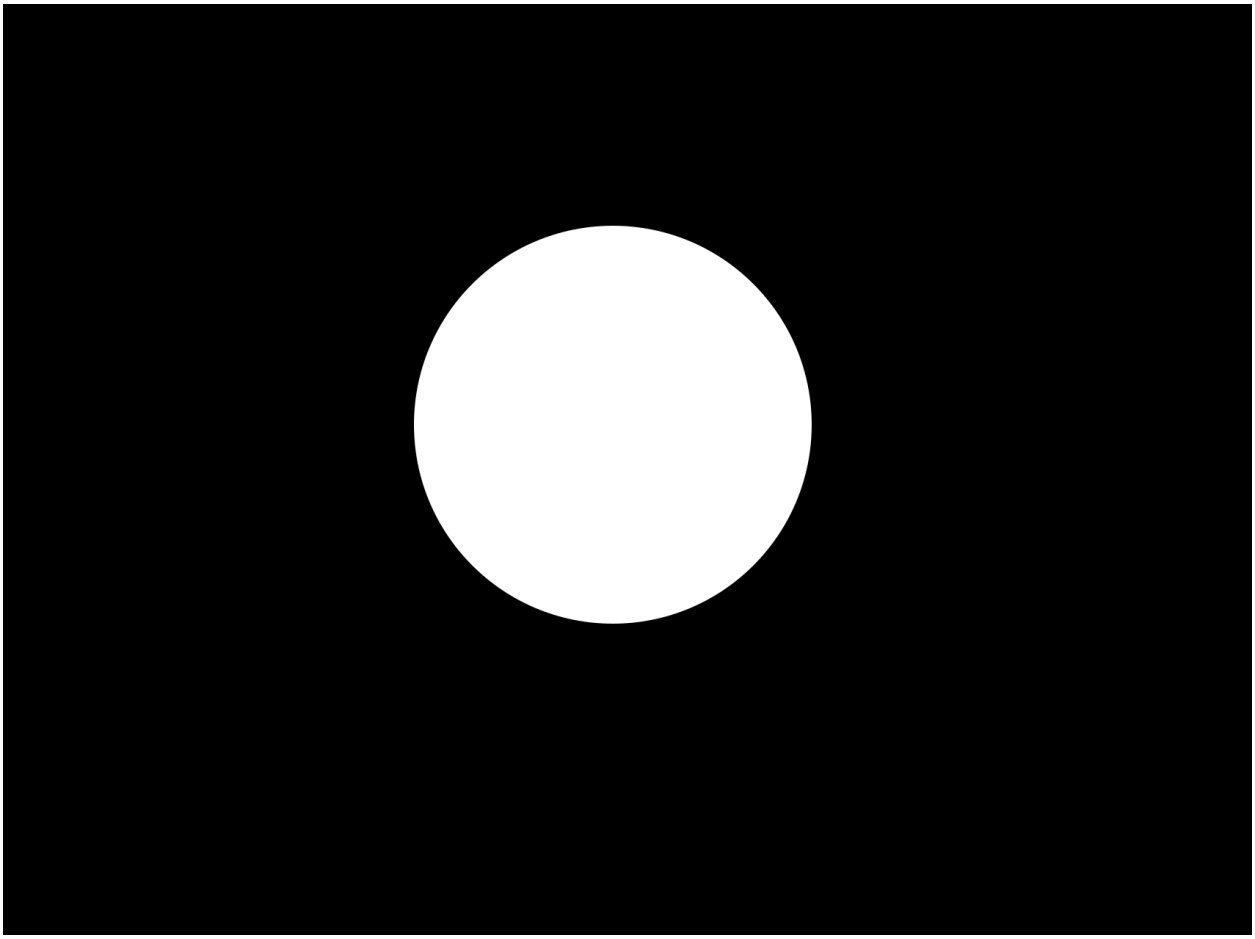


Figure 1. 1-sphere-no-lighting.png

Without any lighting, the image is no different from a white circle on a black background. We can describe the shape of a circle fully using its radius; we can do something similar for a sphere.

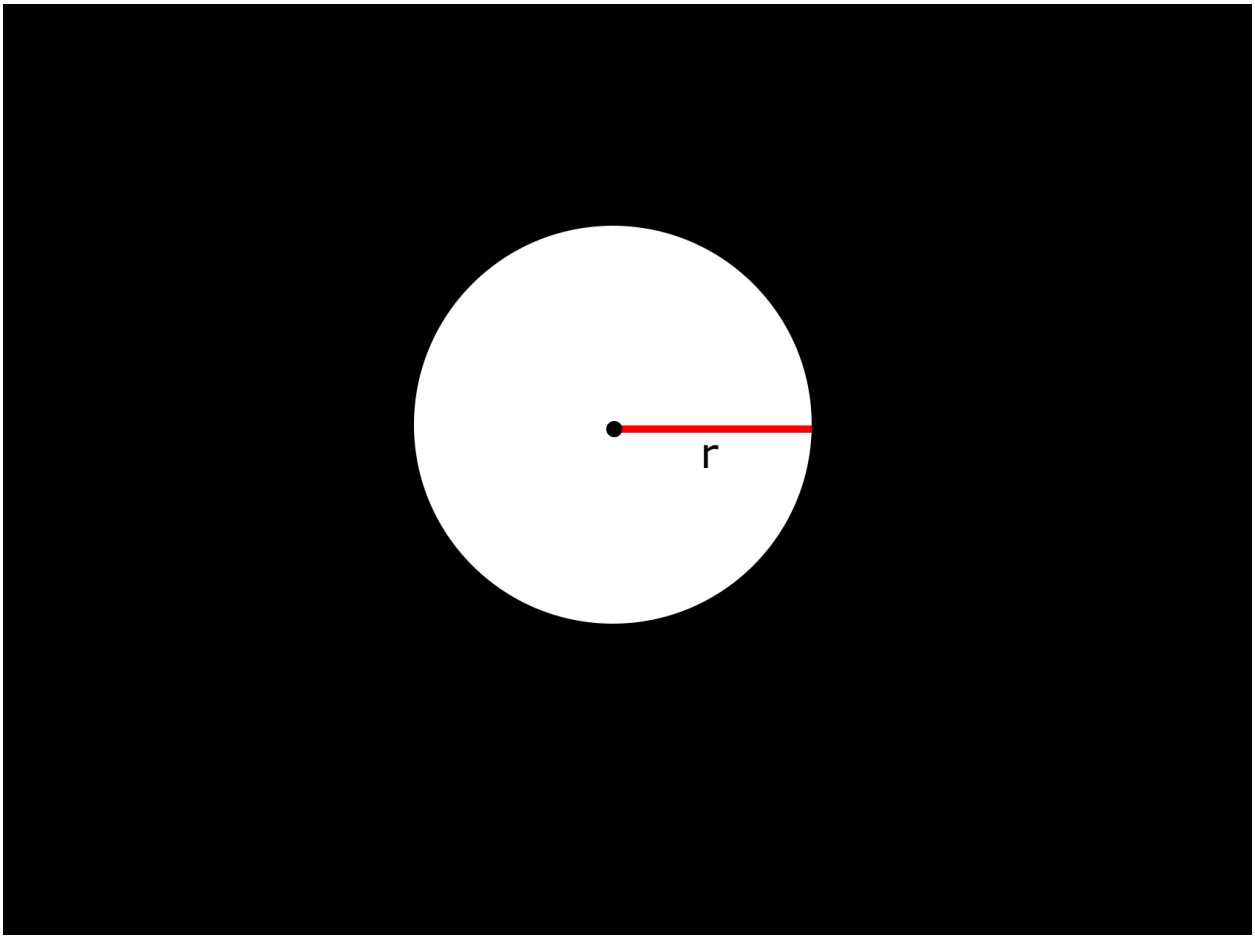


Figure 2. 2-sphere-no-lighting-radius.png

The radius is represented in the image above as the red line from the centre of the sphere (represented as the black dot) to the edge of the sphere. We can use the radius to define the points **on** a circle, using the following equation:

$$x^2 + y^2 = r^2$$

Notice that the word 'on' was in bold in the previous sentence: the equation above will give you x and y coordinates of points that are on the **hull** of a circle (i.e. its perimeter). In order to determine whether any coordinate is within a circle, we can slightly modify the previous equation:

$$x^2 + y^2 \leq r^2$$

By replacing the equal sign by a less than or equal to sign, we can use the above inequality to **test** whether any point is inside a circle or not. If we use vectors to represent the x and y coordinates, we can simplify the above equation to this:

$$\text{length}([x; y])^2 \leq r^2$$

where length() functions simply gives the length of the coordinate from the centre (by calculating $\sqrt{x^2 + y^2}$). We can apply a square root to both the left and right hand sides of the inequality, rearranging to give us the following:

$$\text{length}([x; y]) - r \leq 0$$

Fortunately, we can easily extend the equation above by using 3-dimensional vectors. The following equation gives the equivalent equation for a sphere (kind of like a circle in 3-dimensions?):

$$\text{length}([x; y; z]) - r \leq 0$$

We will use this equation to develop our first SDF (Signed Distance Function) for a sphere primitive. I am calling these primitives as we can combine these shapes to make more complex objects.

Practical: Hello Shaders!

You are free to use any 3D library and coding environment that you want, provided it supports a modern shader based rendering pipeline. For this book, we will be using GLSL (OpenGL Shading Language) along with Visual Studio Code: these should be available on all major Operating Systems. In order to install Visual Studio Code for your system, please follow the instructions at this link: Download Visual Studio Code - Mac, Linux, Windows[1]. Once you have done this, please install the following extensions that will make writing and evaluating our shaders a lot easier:

1. glsl-canvas[2]
2. WebGL GLSL Editor[3]

The first extension will allow us to quickly write our fragment shaders without having to deal with setting up the OpenGL pipeline in a programming language. The second extension tools to aid us in writing GLSL shaders.

After creating a file called `ch1.fs` in your editor, please type the code below into the file (please do not just copy-paste: learning is supposed to be an active process):

[1] https://code.visualstudio.com/download
[2] https://marketplace.visualstudio.com/items?itemName=circledev.glsl-canvas
[3] https://marketplace.visualstudio.com/items?itemName=raczzalan.webgl-glsl-editor

```
precision mediump float;

// Provides us with the screen resolution.
uniform vec2 u_resolution;

void main()
{
    // Converts pixel coordinates values to be between 0 and 1.
    vec2 uv = gl_FragCoord.xy / u_resolution.xy;

    // Colour of the background: currently set to have a red,
    // green and blue value of 0 i.e. a black colour.
    vec3 colour = vec3(0.0);

    // Sets the final colour to be output to the screen.
    gl_FragColor = vec4(colour, 1.0);
}
```

The comments in the code above should help with understanding what the code does. Essentially, we have a function called `main()` that tells OpenGL what colour each fragment should be. OpenGL provides us with some global variables like `gl_FragCoord`: this gives you the coordinate of the current pixel. The `glsl-canvas` provides us with more information by providing the uniform 2-dimensional vector called `u_resolution`, which gives us the width and height of the frame we are rendering to. We use these two values to set the `uv` variable, which transforms these coordinates into the range 0 to 1.

We then define a colour variable: this is a 3-dimensional vector where the first, second and third value represents the amount of red, green and blue in this colour (RGB values). In our code, all of these are set to `0.0`, meaning `colour` represents black. Finally, we set the output variable `gl_FragColor` (note the *American* spelling of colour) to `colour`. If you now right-click your code, click **Command Palette**..., then search and execute the command **Show glslCanvas** you should be able to see the result of the fragment shader above: you should see a black screen.

[Challenge]

Try to play with the values of the `colour` vector in the shader above. Your challenge is to:

1. Change the screen to be fully red.
2. Change the screen to be fully blue.
3. Change the screen to a more complex RGB colour (you can use an RGB colour picker online to find one you like).

[Answer]

1. This can be done by setting the red value of the colour vector to 1.0 and keep the green and blue values at zero: `vec3 colour = vec3(1.0, 0.0, 0.0);`
2. This can be done by setting the blue value of the colour vector to 1.0 and keep the red and green values at zero: `vec3 colour = vec3(0.0, 0.0, 1.0);`
3. Same as above but now using the specific values you found online. (Note: you are most likely going to find RGB values between 0 and 255. Just divide these values by 255.0 to get the correct value for the shader).

Practical: Sphere SDF and Ray Marching

Using the equation of a sphere that we formed above, we can add the following function above the `main()` function:

```
// SDFs

float sdfSphere(vec3 position, float radius)
{
    return length(position) - radius;
}
```

The equation above defines the sphere implicitly using distance. Unlike triangle meshes, this does not immediately tell us which pixel should be coloured. To render such shapes efficiently, we use a technique called ray marching, where we move along a ray in steps determined by the signed distance function.

In real life, a light source emits light rays. Light rays travel in straight lines (if you ignore the impact of refraction and diffraction) and hit an object. The object absorbs different wavelengths in the white light (which is made up of many different colours of light) and reflect the light that is not absorbed: this is essentially the colour of an object i.e. a red object absorbs all the other colours of light except red. Eventually this light will reach our eyes where it will be absorbed by the rod and cone cells in the back of our eye, causing us to 'see' the world.

The above is a perfectly valid solution for the real world, however, as mentioned before we do not have such luxuries on computers. In order to deal with this limited power, techniques such as Ray Marching change the direction of light: they shoot light from the camera (or lens) into the world and choose the colours of the pixel based on what objects this ray of light hits (this idea is similar to how humans originally believed we see the world).

We will assume that each pixel on the screen is represented by a 2D grid in the 3D world we are representing. Each ray can be represented using two quantities: the vector representing the starting point of the ray and a unit vector representing the direction of the ray. We will then incrementally increase the length of this ray (i.e. 'march' the end of the ray forward) until it hits an object. Any point on a ray can be described using the following vector equation of a line:

$$p = r_{start} + \text{step} \cdot r_{direction}$$

where p is a point on the line, r_{start} is the position where the ray starts, $r_{direction}$ is the direction of the ray and step is a variable controlling where the ray ends.

Based on all of the above, we can produce the following (**please make sure that you place this code below the SDF function but above the `main()` function**):

```
// The distance where the near and far planes are.
float nearDistance = 0.01;
float farDistance = 30.0;

// Define a struct to store information about objects the ray has hit.
struct HitInfo
{
    float dist; // Distance returned by scene at the current sample.
    float totalDist; // Total distance travelled by the ray.
    vec3 position;
    int objectId;
};

// Define a struct to store material information about different
// objects.
struct ObjectMaterial
{
    int objectId;
    vec3 objectDiffuseColour;
};

// Forward define object materials
ObjectMaterial sphereMaterial;

// Our Ray Marching function based on the discussion above.
HitInfo rayMarch(vec3 rayStart, vec3 rayDirection)
{
    // Start looking for a object hit at the near plane distance.
    float rayStep = nearDistance;
    HitInfo hitInfo;

    // Loop controlling how many times we 'march' the ray forward.
    // Here we march the ray forward a maximum of 100 times.
    for (int i = 0; i < 100; i++)
    {
        // Find the location of the end point of the ray based
```

```
        // on the equation above:
        // p = r_start + (step * r_direction)
        vec3 pointOnRay = rayStart + rayStep * rayDirection;

        // Please ignore the error saying that the function scene()
        // is undefined: we will implement this in the next code
        // snippet.
        hitInfo = scene(pointOnRay);

        // Store the position where we potentially hit an object.
        hitInfo.position = pointOnRay;
        // Store the distance returned from the scene()
        hitInfo.dist = hitInfo.dist;
        // Store distance travelled by the ray.
        hitInfo.totalDist = rayStep;

        // Stop marching the ray if we exceed the far plane or hit
        // at a distance less than the near plane.
        if (hitInfo.dist < nearDistance || rayStep > farDistance)
            break;

        // March the ray forward by increasing rayStep.
        rayStep += hitInfo.dist;
    }

    return hitInfo;
}
```

In the above code snippet, we make a reference to the `scene()` function; we will now define this function. This function essentially defines the layout of your scene i.e. allows you to define what objects are present in your 3D world, where they are etc. **Please place the `scene()` function above the `rayMarch()` function:**

```
HitInfo scene(vec3 position)
{
    // The coordinates where the centre of the sphere is. We
    // have set this to the coordinates (0.0, 0.0, 0.0).
    vec3 sphereCentre = vec3(0.0, 0.0, 0.0);

    // The radius of the sphere.
    float sphereRadius = 0.5;

    // We now use the SDF function for the sphere to determine
```

```
    // whether we have hit the sphere or not. Note the usage of
    // (position - sphereCentre) instead of sphereCentre: this
    // ensures that the sphere is translated correctly based on
    // our current position.
    float sphere = sdfSphere(position - sphereCentre, sphereRadius);

    // Store information about the object we have hit. At the moment,
    // the code below might seem unusual, however, imagine that we
    // have many different types of objects in the scene, each with
    // different material properties etc. The code below, when
    // extended will allow us to correctly determine which object
    // the ray hit.

    // This min() function will be altered later to find the
    // minimum distance of an object.
    float finalDist = min(sphere, sphere);

    HitInfo hitInfo;
    hitInfo.dist = finalDist;

    if (finalDist == sphere)
    {
        hitInfo.objectId =  sphereMaterial.objectId;
    }

    return hitInfo;
}
```

Hopefully, the above code along with the comments explains what each part of the function does. Finally, we will update the `main()` function to produce our first Ray Marched image:

```
void main()
{
    // Set object material properties.
    int maxObjectId = 0;
    sphereMaterial.objectId = maxObjectId++;
    sphereMaterial.objectDiffuseColour = vec3(1.0);

    // Screen texture UV coordinates
    vec2 uv = 2.0 * (gl_FragCoord.xy /u_resolution.xy) - 1.0;
    uv.x = (u_resolution.x / u_resolution.y) * uv.x;

    // Setup scene.
```

```
13
14      // Set the default pixel colour to black.
15      vec3 colour = vec3(0.0);
16
17      // Near plane at 4.0 on z the z-axis i.e. 4 units out of the screen
18      // (z-axis pointing out of the screen).
19      vec3 cameraOrigin = vec3(0.0, 0.0, 4.0);
20
21      // Near plane at -1.0 on z the z-axis i.e. 1 units into the screen
22      // (z-axis pointing out of the screen)
23      vec3 rayDirection = normalize(vec3(uv, -1.0));
24
25
26      // Use Ray Marching to see which object the ray that goes
27      // through this pixel has hit.
28      HitInfo closestHit = rayMarch(cameraOrigin, rayDirection);
29
30      // Make sure we have not hit an object that is closer than
31      // the near plane.
32      if (closestHit.dist < nearDistance)
33      {
34          // if the closest hit object is the sphere,
35          // set the colour of this pixel to white.
36          if (closestHit.objectId == sphereMaterial.objectId)
37          {
38              colour = sphereMaterial.objectDiffuseColour;
39          }
40      }
41
42      // Output the colour from the Fragment Shader.
43      gl_FragColor = vec4(colour, 1.0);
44  }
```

The image below shows what you should be seeing on your screen. Congratulations! You have just produced your first Ray Marched image.

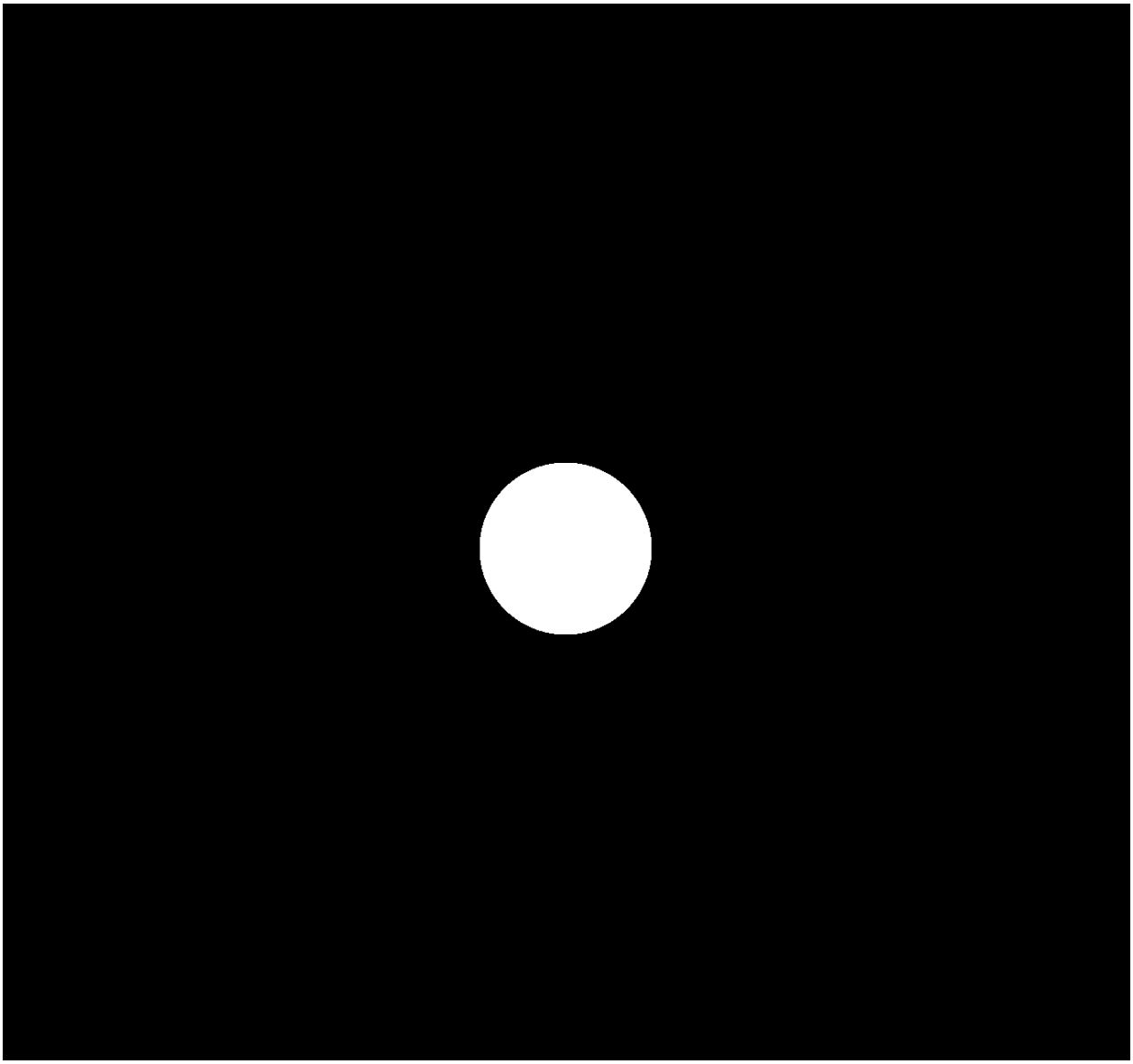


Figure 3. 3-ray-marched-white-sphere.png

[Challenge]

1. Play with the radius of the sphere: does increasing or decreasing this have the effect you expected?
2. Play with the position of the sphere: does the result correspond to what you expected?
3. Change the sphere to be a red sphere instead of a white sphere.
4. Change the background colour to be a blue colour.

[Answer]

1. This can be done by increasing or decreasing `sphereRadius` in the `scene()` function.
2. This can be done by changing `sphereCentre` in the `scene()` function.
3. You can set the sphere colour by altering the `sphereMaterial.objectDiffuseColour` variable in the `main()` function. To set it to red, you can do the following: `sphereMaterial.objectDiffuseColour = vec3(1.0, 0.0, 0.0);`
4. Change the default value of the `colour` vector to `vec3 colour = vec3(0.0, 0.0, 1.0);`. This will cause the black default colour to change to a blue colour.

2. Lighting: Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: SDF of a Plane

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: SDF of a Plane

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Colours seem too dark

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

3. Lighting: Cast Shadows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: Cast Shadow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Cast Shadow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

4. Lighting: Diffuse

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: Diffuse Lighting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Diffuse Lighting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

5. Lighting: Indirect

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: Approximation of Indirect Bounce Lighting from the Ground

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Approximation of Indirect Bounce Lighting from the Ground

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: Approximation of Indirect Lighting from the Sky

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Approximation of Indirect Lighting from the Sky

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

6. BRDF - Diffuse

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: Expanding the material properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Expanding the material properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: Changing between gamma and linear colour space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Changing between gamma and linear colour space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: Properties for BRDF

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Properties for BRDF

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: Dot product between normal and light direction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Dot product between normal and light direction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: BRDF Diffuse Calculation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: BRDF Diffuse Calculation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

7. BRDF - Specular

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Theory: Specular lobe

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.

Practical: Specular lobe

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cgproceduralgenerationusingraymarchingandsigneddistancefields.