Final project report

02561 - Computer Graphics

Jakub Janaszkiewicz
s191925

December 15, 2019

Contents

1 Introduction

The Computer Graphics course at DTU serves as a great introduction to the world of rendering 3D scenes using an approach based on a polygon transforming pipeline. The purpose of this project is to explore an alternative rendering technique called *ray marching*, which is beyond the scope of the course. In order to introduce the reader to this method, we create an interactive 3D scene which aims to demonstrate a selection of unique features this method possesses. We also present how some of the well-known concepts, such as lighting and shadowing, can be implemented with this rendering technique, and compare these algorithms with their respective implementations in polygon-based setups.

2 Method

The essential difference which separates ray marching from more conventional approaches to computer graphics is that we are not dealing with polygons anymore. A lot of the things that we are used to in a typical 3D toolkit - like geometry, lights, or cameras - are nonexistent. This poses a question: if we don't have points, lines, triangles, and meshes, how are we supposed to render anything? The answer comes from the name of the family of rendering techniques this method belongs to: pixel-based techniques. At a high level, for each pixel on the display, we will shoot out an imaginary ray from a virtual camera that is looking at our world. If this ray collides with an object, we will set this pixel's value to its colour at the intersection point, and if not - the background colour. In order to achieve that, we need two things: a way to define the positions and shapes of the objects in absence of polygons, and a method for finding ray intersections with those objects.

2.1 Signed distance functions

In order to define an object's shape and position in a ray marching setting, we need to find a mathematical function that describes distance in 3D space to the surface of that object. This function needs to return negative values for points that lie inside of it, hence the name "signed distance function". These functions are usually easy to implement for geometrical primitives. We can show a simple example by defining one for a sphere with a radius r centred at a point $s = (s_x, s_y, s_z)$. An SDF for such an object is $SDF(p) = ||p - s||_2 - r$, where $p = (p_x, p_y, p_z)$ is the point we sample the distance at. We can see that the initial condition is satisfied - for points inside the sphere, the distance to the centre will be smaller than the radius, and thus, the result will come out as negative.

2.2 Constructive solid geometry

Having defined SDFs for objects we want to place in the scene, we need a way to combine them into one global "world SDF", which, again, for any given point in space, will return the distance to the nearest surface. Why will explain why this function is vital to the rendering

process in section 2.3. For now, we can define three binary operators which we can use to combine shapes:

1. Union - in order to join two shapes together, creating their sum, we take the minimum of their distance functions. This follows from the definition of the SDF - a distance to the nearest object is the minimum of distances to both objects.

2. Intersection - to find the intersection, we take the maximum of the SDFs. While less intuitive than the union, we can see why this is the case - such a function would only return zero if the point in question lies on a surface of an object (zero), that also happens to lie inside the other object (negative value). This ensures we only consider surfaces where the two objects cross. A maximum of two SDFs will return a negative value of both values are negative, indicating volume shared by two shapes.

3. Subtraction - an SDF for a subtraction of object B from object A is

$$f_{A-B}(p) = max(f_A(p), -f_B(p))$$

Just like numerical subtraction, this operation is not commutative. To build an intuition as to why this formula is correct, we can draw an analogy to the one before. Negating the second SDF makes this an intersection of the first shape with the inverse of the second shape, removing the part it has in common with the first object from its volume.

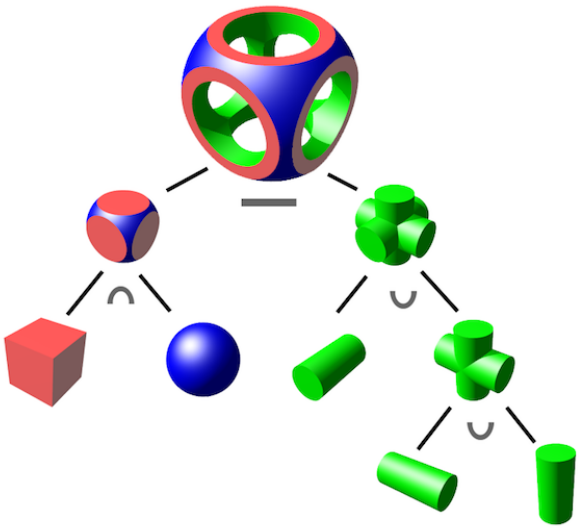

Figure 1: Example construction of a complex shape from 3D primitives

Source: An article on constructive solid geometry on Wikipedia

We call this concept of creating a complex surface or object by using Boolean operators to combine simpler objects "constructive solid geometry". An example of this method in action is shown on Figure 1. In this case, the red cube first has its corners rounded by

taking an intersection with the blue sphere. Then, we subtract a cross we obtained from consecutive unions of axis-aligned cylinders. The resulting shape is shown on the top, with the surface colours corresponding to the simpler shapes used in its creation.

Because of the continuous nature of signed distance functions, it is possible to create soft versions of the aforementioned operators. Such functions would, for instance, blend objects together instead of strictly merging their geometry, with the use of smooth approximations to the minimum (or maximum) function. Because we want the objects we render in our scene to have sharp edges however, we will not be using smooth operators in this project.

2.3 The ray marching algorithm

Equipped with the knowledge about CSG, let's assume we have defined the world SDF for the scene we want to render. Let us also assume that we know our virtual camera's position and orientation in 3D space. We can imagine the camera's display as a grid of pixels floating in front of the camera at a fixed distance. We shall assign a ray to each of them, the origin of which is going to be the camera's position and the direction will point towards the pixel in question.

In order to find an intersection of the ray with our 3D scene, we *march* it by taking steps of a fixed length along the direction of the ray and evaluating the world SDF at each step. If the sampled value is lower than some threshold, that is, the distance to some object is small enough (or we landed inside of an object), we consider this a collision. Otherwise, we march until an upper limit of iterations is reached or the total distance travelled by the ray exceeds some threshold value. Upon collision, a function that implements this algorithm would return the point of intersection with a hit surface. The returned value comes with a margin of error dependent on the "closeness" margin. Exceed the total iteration count or travelling too far from the origin indicates that there was no solid surface for the ray to collide with along its path. In this case, the function would return the position of the last valid step.

This algorithm, albeit conceptually simple, has two potential problems. Firstly, the pessimistic complexity is constant in the maximum number of steps taken along each ray. This limits the distance at which objects can be rendered to a constant value. Secondly, we risk overshooting a surface if its thin enough in the dimension parallel to the ray. In this case, the ray would continue to travel behind an object. With this sufficient thinness, one step could be taken right before hitting the object, but after adding the step size the next one would land behind it, causing the ray to "tunnel through", not detecting the collision at all. To combat this, we use an improved version of the algorithm, called *sphere tracing*. Instead of a fixed value, we sample the world SDF at a given point and use the result as the step size. This achieves two things - one, we will never overshoot our targets because we can never make a step that's longer than the distance to the nearest surface, and two, in regions further away from any object we can cover more distance in one step. Figure 2 illustrates this process, showing the consecutive points on the path of a ray starting at the point p_0. We can see that using this method, the spheres' radii converge exponentially

quickly when approaching a wall. Point p_4 in this example is already close enough to the surface to consider this a collision.

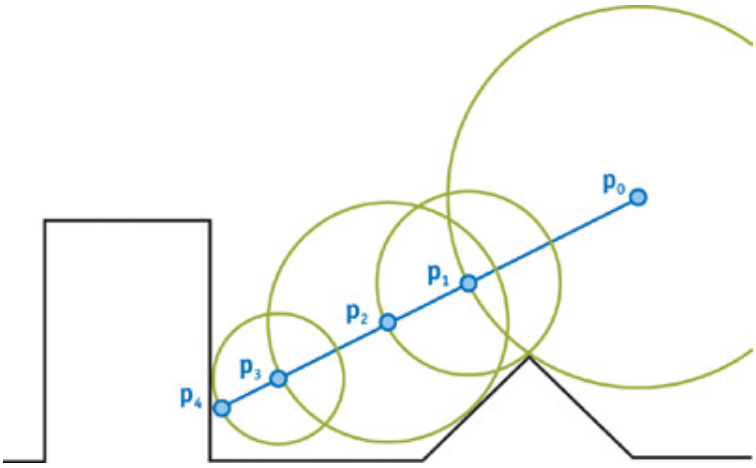

Figure 2: Sphere tracing

Source: "GPU Gems 2", chapter 8

2.4 Lighting

We now turn our attention to recreating the basic phenomena native to the polygon-based approach to computer graphics, starting with lighting. The only difference in implementing a full Phong reflection model in the case of ray marching lies in the way we obtain surface normals for shaded objects. In a classical setting, we would have passed them in a buffer to the shader in which we perform lighting. When ray marching however, since the scene is defined as a mathematical function, we can instead get the surface normal at any point in space using numerical approximation. After a ray collides with an object in the scene, we can sample the world SDF at a few points around the point of collision. Specifically, we do that three times, each time applying an epsilon offset in the direction of one of the main axes. This allows us to approximate the SDF gradient at the point of collision, which near the surface of an object points in the direction of the normal vector. Having calculated the normal, the rest of the reflection model's implementation is no different from the usual.

2.5 Shadowing

There are a number of techniques used to create shadows in a polygon-based setting, each having its advantages and disadvantages. Applying global shadowing, however, is not trivial. Approaches like shadow mapping usually require additional computational resources, which introduce significant overheads and complexity to the pipeline. In the light of this, an undeniable advantage to using the ray marching algorithm is that it allows us to apply global shadowing in a conceptually simple way. Given the position of the light source, we

can simply create a ray originating at the point on the surface we want to shade. We can get its position from the initial, collision-detecting ray march. We then set its direction to point at the light source and march this ray until we either collide with something (which means the origin is obscured from the light) or the distance travelled by the ray exceeds distance to the light (we didn't collide with anything on the way - the point is not in shadow). Figure 3 shows how the rays are relatively positioned when we include shadowing. Albeit simple to understand and to implement, the downside to this method is that it requires us to now march two rays per pixel - the first one from the camera to the object, the second - from the object to the light source. This is, in the worst case scenario, doubling the amount of computation required.

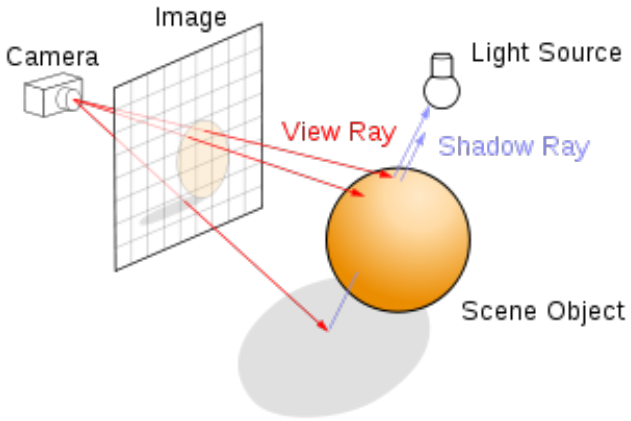

Figure 3: Tracing shadows in the ray marching setup

Source: An article on ray tracing on Wikipedia

2.6 Reflections

Simulating the light reflecting off the objects' surfaces is usually achieved in a polygon-based pipeline using texture mapping. One example solution to this problem is environmental mapping. Using this method, the appearance of a reflective surface is approximated by means of a precomputed omnidirectional texture image. This image-based technique is restricted in a way that the only thing which can appear as a reflection is the static environment. Nowadays, those who strive for photo-realism usually turn to ray tracing and other modern techniques that can simulate light bouncing around the scene ray by ray. Ray marching is one of such settings in which simulating reflections is trivial. At a point of collision with an object, we simply march another ray in the direction of the view ray reflected off the surface normal. Provided that the second ray collides with something, we can use the returned information and blend the colours of both objects. We can continue to bounce rays off of objects as many times as we want, to capture more levels of reflection. This, however, comes with a computational cost of consecutive ray marches, and can tax the graphics card severely if too many iterations are used.

2.7 Volumetric effects

We can apply volumetric effects, such as fog or glow, without any costly calculations. While marching along the ray, we can sample the density field we are interested in at each step along the ray and sum up the results. The value we get will be an approximation to the mean density along the ray's path. In the scene rendered for this project, we make it so that the space around some of the solid surfaces glows with a soft pink colour. To implement this effect, we first define the glow radius, which is the maximum distance from the surface at which the glow is still present. Then, at each step of the march, we check whether we are inside of this radius (the SDF gives us the distance to the nearest surface) and if so, we add the glow density at this point to an accumulator. In the end, the colour obtained for each ray that passes close enough to a glowing surface (special case - collides with it) has a proportion of pink blended in. Because of the nature of volumetric phenomena, the ray marching algorithm's traversal of space is a perfect tool for implementing them. In fact, in a polygon-based setting, we would most likely implement volumetric effects using a simple ray march in the fragment shader that traverses a 3D texture that represents the density field. Such simplified ray marching algorithms usually take steps of constant length, for a total distance limited by some z-buffer.

2.8 Distance attenuation

Distance-based effects are also available to us without any additional computation. During the march, we already collect the total distance travelled for each ray, to allow for early termination if the ray travels too far from its origin. We can use this value to, for instance, reduce the lighting intensity based on the distance to the camera. This is especially useful for limiting the range at which the camera can see, since every ray is ultimately bounded by its maximum travel distance. If we render objects that are further away as darker, we can work around the rendering distance limits of the technique by obscuring the artefacts that appear at the horizon.

3 Implementation

In this section, we will describe the implementation of specific parts of the interactive scene created for the project. First, we will shortly describe how we use parts of the classical graphics pipeline. Then, we will go into detail, explaining different modifications and optimisations that have been applied to the algorithms to enable smooth, real-time rendering on devices which do not boast a dedicated graphical unit.

3.1 Used technologies

The underlying technological stack will most likely sound familiar to any computer graphics enthusiast. Starting from the front-end, we display our scene using a `canvas` HTML object embedded into a web page. The core functionalities for rendering images this way are provided by the WebGL (Web Graphics Library) API for JavaScript â the main scripting

language for the Web. This API allows us to create interactive 2D and 3D graphics within any compatible web browser without the use of plug-ins. Because rendering complex 3D scenes is a computationally heavy task, WebGL uses hardware acceleration on the GPU to allow for parallel processing. As of 2019, WebGL 1.0 is a standard implemented in any modern browser, including mobile devices.

To render a scene, we process its geometry on two devices, and hence, two levels of abstraction. First â on the processor, using the aforementioned JavaScript API, we configure our environment and send the geometry data to the graphics processing unit. The GPU then runs programs that process this information to arrive at a 2D image. We call these programs *shaders*, and in the case of web based computer graphics, write them in the OpenGL Shading Language (GLSL).

3.2 Classical rendering pipeline

In classical polygon-based rendering pipelines, the journey from a 3D scene to displaying it as a 2D image on the screen can be somewhat of a convoluted one. In order to render a scene composing of many objects, we need to first load their geometry to the memory. We use the JavaScript WebGL API to load vertex coordinates and other vertex-specific data to array buffers, which are then passed to the vertex shader. This shader runs once for every vertex and is used to transform its position to a 2D coordinate at which it will appear on the screen. Vertex shaders can manipulate properties such as position, colour and texture coordinates, but cannot create new vertices. Coordinate change is usually achieved by applying multiplications with transformation matrices. The next stage is the rasterizing, a process which produces fragments that are going to be displayed on the screen and passes them to the fragment shader. This one runs for every fragment and is tasked with assigning a colour to at most one output pixel, usually based on interpolated position and surface normal values passed from the vertex shader in form of *varying* variables. These shaders are commonly used to apply lighting, to do bump mapping, shadows, specular highlights, translucency and other phenomena.

3.3 Ray marching pipeline

The ray marching algorithm is entirely implemented on the graphics hardware, specifically in the fragment shader. We can use the fact that the shader knows the position of each fragment passed to it, and can treat it as the pixel's position on the camera's display (and thus, the actual display). We can use the classical pipeline to create our pixel-based setup. With ray marching, the only polygon vertices we pass to the vertex shader are the four corners of the 2×2 quad located at $z = -1$. With default projection, this quad fills the display entirely. The interpolated vertex coordinates passed to the fragment shader match the corners of the virtual display. Assuming that the camera position is at the origin of the coordinate system, these are the only input we need to define our rays and render the scene.

3.4 The rendered scene

Having equipped the reader with the basics on the rendering technique, we can now proceed to explaining how the scene was written. From the technical point of view, we will focus briefly on each part of the code and then elaborate on the construction of the fractal solids which are rendered to the screen.

3.4.1 Scene definition

As explained in Section 3.3, all code relevant to the algorithm is located in the fragment shader. The JavaScript part only creates the display quad and passes it along with the aspect ratio and time offset (used to perform animation) to the vertex shader. This shader adjusts the quad's height to correct for the aspect ratio and passes on the quad corners' positions to the fragment shader.

Before we start explaining the code, let's define the scene we want to render. It's composed of three glowing Menger sponges placed in a sequence along the X axis. There is a light source located above the sponges, which swings along the X axis to demonstrate dynamic shadowing on the plane below. The camera rotates around the scene in the XZ plane, looking down at the fractals and the ground plane.

3.4.2 Main function

We start from the entry point to the shader, the `main` function. Inside, we create a transformation matrix which we will use to alter the positions of the rays leaving the camera eye. By default, the camera can be though of as positioned at point (0, 0, 0), the origin. We translate this position by an offset in the positive Z direction and then rotate the resulting vector around the Y axis using the supplied time value. We then use the same transformation matrix on the rays' directions. The transformation matrices are embedded into the shader's code, but its also possible to pass camera position and orientation to the shader from outside using uniform variables. This way we could achieve a greater level of interactivity, for instance allowing the viewer to control the camera.

3.4.3 Lighting effects

The ray position and direction are passed to the `getLight` function, which performs the ray marching process and returns the colour value for the given pixel. First, it performs the march itself, altered to return the material information of the object it detects collision with. Taking into account the position of the light source, we use this material parameters to shade the pixel according to the Phong reflection model. If the proper setting was enabled, the lighting terms use exponential attenuation to reduce their levels based on distance to the light source (diffuse and specular terms), or to the viewer (the ambient term). This is also where the shadowing, if applicable, is performed. In this scene, we implemented soft shadows - something that cannot be easily obtained in polygon-based rendering. The idea is

to record the closest the ray that marches towards the light source has ever been to a surface. If it ultimately doesn't hit anything along the way, but came very close to something, we assume the ray's origin lies in penumbra. This improvement to hard shadows comes with no extra overhead, since we already evaluate the world SDFs during the march of a ray towards the light source.

3.4.4 The marching procedure

The `rayMarch` function implements the ray marching algorithm, while also sampling the volumetric glow density along the way. We only apply glow to the floating fractals and not to the ground plane, so that the shadows are clearly visible on the ground. To distinguish between surfaces we collide with, we use additional information stored in the vector returned by the world SDF, in this case, the material colour. As mentioned above, the returned value is then used in the lighting process as an addition to the three components specified by the Phong model.

3.4.5 World SDF

Finally, the `distanceField` function implements the scene SDF. Before we calculate any distances, however, we take the sampling point's position and set its x coordinate to be modulo a constant that defines the spacing between consecutive fractals. This modulo operation is only performed for points that are in a certain distance along the X axis away from the origin. This results in an effect called "domain repetition" â the world's objects' geometry will repeat along the axes we specify in the `vecMod` function for as many times as they fit in the defined bounding box dimensions. A special case for this technique is to take an unbounded modulo for all three dimensions. This leads to the scene's geometry repeating itself in every dimension, creating an infinite world.

The creation of the Menger sponge fractal is an iterative process. We start with an SDF for a cube. Then, we define SDFs for three axis-aligned cuboids that intersect in the middle of that cube. Each goes through the centres of a pair of the cube's opposite sides. If we then take the union of these cuboids (minimum of the SDFs), we obtain a three-dimensional cross-like shape. We can then subtract this shape from the original cube, creating holes in it, which results in the Menger sponge pattern. We then proceed iteratively, repeating this process at smaller scales to increase the level of detail. In our case, we limited the iteration count to a small value, to make a trade-off between fractal resolution and performance.

We apply some optimisations to this construction. We don't use the actual cuboid SDF, but instead take advantage of the fact that, because our geometry is axis-aligned, we can only look at two dimensions at a time. For instance, a function

$$f(p) = min(|p_x| - 1, |p_y| - 1)$$

is an SDF for a cuboid extending infinitely in the Z direction, with the other two dimensions being contained in the range $[-1, 1]$. We use those simplified 2D SDFs to carve out the sides of the fractal.

4 Results

Since the purpose of this project was to create a scene that demonstrates unique features of the ray marching rendering technique, it's equipped with a range of inputs that a viewer can use to control different variables used in the process. In particular, they allow to:

1. Set coefficients for the ambient and diffuse light
2. Specify the intensity and shininess of specular highlights
3. Control the intensity and radius of volumetric glow
4. Toggle distance attenuation and modify its coefficient
5. Switch shadow tracing on and off

The numerical values and ranges for the parameters are not shown on purpose. We encourage the viewer to play with the settings and experiment with different values to see how the result changes in response.

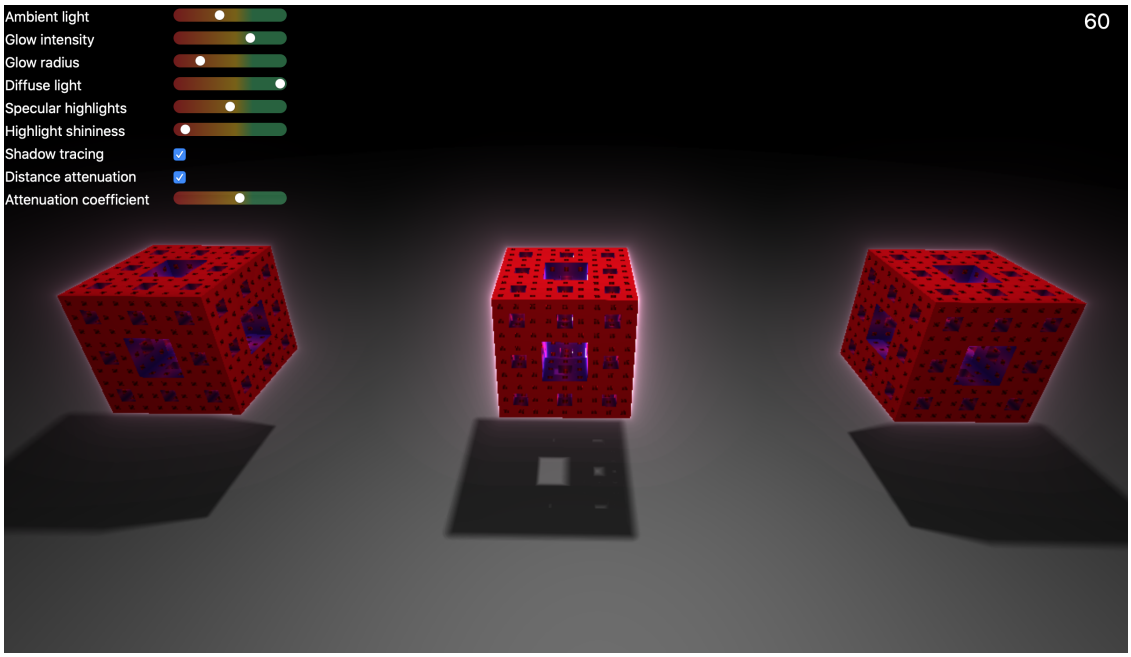

Figure 4: A screenshot of the scene

A frame from the scene with a default selection of the aforementioned parameters is shown in Figure 4. One can also notice a frame rate counter in the upper right corner. The different constants used in the rendering were chosen carefully, as to allow for a frame rate of 60 frames per second on integrated graphics chips. This means any fairly modern laptop without a dedicated GPU, as well as mobile phones. The aspect ratio is chosen automatically based on the viewer's device.

The default setting for the distance attenuation coefficient allows us to hide the fact that the range at which we can render objects is limited. Since the SDF for the ground plane is defined as by the formula:

$$f(p) = p_y - y_{plane}$$

the plane spans an infinite area. If we disable the distance attenuation, we will see that the plane looks circular instead. The curvature of the horizon is caused by the fact that the maximum length of a path travelled by a ray, and thus the upper bound for the rendering distance, defines a sphere around the camera beyond which nothing can be displayed.

5 Discussion

The scene created for the project intends to palpably demonstrate the different rendering capabilities of the ray marching technique. In this section, we discuss the limitations of the solution we arrived at and propose improvements and possible future work that can be done on the scene.

5.1 Limitations

Even though the ray marching algorithm allows us to implement a range of graphical phenomena in a conceptually simple way, it does come with some downsides. One of them is of course the performance - since everything is processed on a pixel-by-pixel basis, the animation can get choppy on high resolution displays. Also, simulating shadows or reflections requires additional ray marches per pixel, in the worst case lowering the performance by half.

The method we use to render volumetric glow effects could also see some improvement. Because the intensity of the glow depends on the sum of the densities along the ray, we can see artefacts arising from the discrete nature of the ray marching process. This can be observed when we set the glow radius to a high value - rays that pass near the edge of the glowing region will only take a couple of steps through it, allowing the relative difference in intensity between neighbouring rays to be clearly visible.

5.2 Possible improvements

The scene was implemented with many optimisations put in place to achieve low hardware requirements. Therefore, the premise behind most of the decisions we made was not to compromise performance. Improving computational efficiency of the code would moderate the extent of this trade-off. For instance, a relatively low maximum distance for the rays could be increased, allowing for a larger world to be rendered.

The scene contains three fractals and a plane - arguably not a very diverse selection of objects. Even though the choice of the Menger sponge was dictated by its ability to demonstrate the complexity of surfaces we can arrive at while using only simple mathematical transformations, we could take this idea a step further and make use of a wider range of tools constructive solid geometry gives us. Soft blending, rounding or interpolation between SDFs to name a few techniques not covered in this project.

For now, the material properties of the fractals are determined by the iteration count during their creation. The outer surfaces are red, and every level of detail introduces more blue to the mix. The ground plane is plain white, and the same goes for the colour of the light. A possible improvement to the scene could be to implement texturing in the shader. A way to distinguish which object does the surface belong to could be to have SDFs pass identifiers through the CSG operators. However, finding the texture coordinate for a given point in 3D space may prove challenging.