Graphics Programming Final Project Report: Procedural Scene Generation with Ray Marching

Jaerin Lee*

June 17, 2020

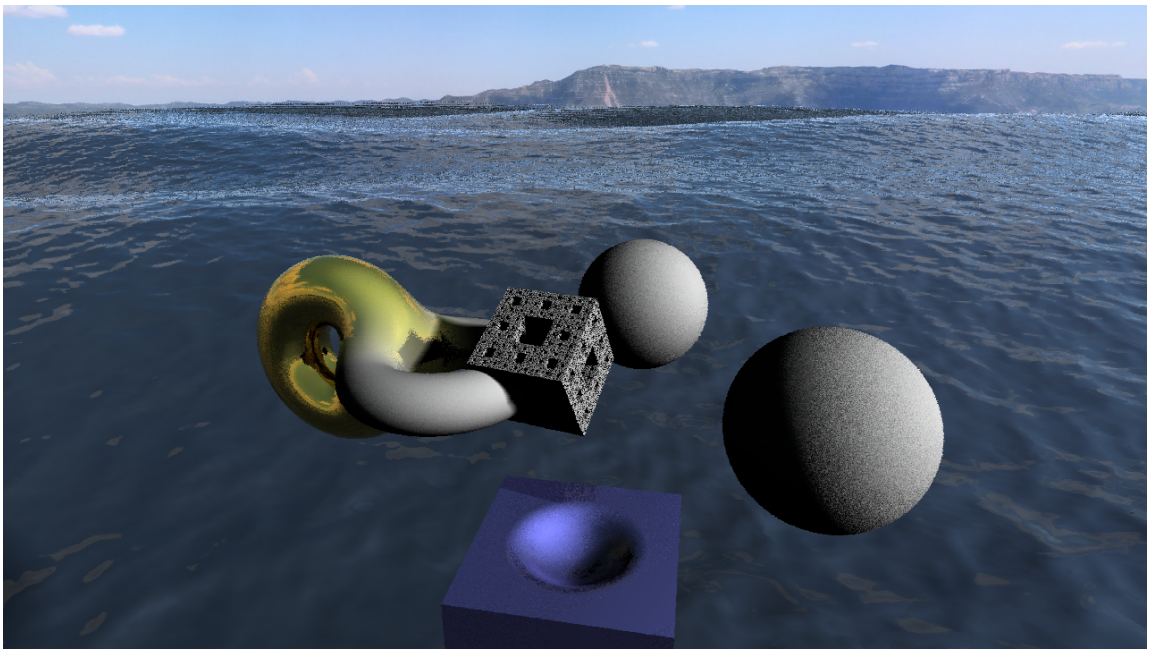


Figure 1: **Project result**. This scene contains deformed and merged smooth geometric primitives, a complex 3D fractal object, and a procedurally generated ocean waves using 2D continuous noise. The scene is rendered in HD resolution (1280×720). All objects are rendered in *real-time* with a *single* ray marching shader.

1 Project Goal

The goal of this project is to implement a ray marching shader and demonstrate its capability especially on real-time rendering of a procedurally generated scene. Ray marching, or sometimes sphere tracing, is a rendering technique that estimates the color of each pixels by casting a ray onto

*Student ID: 2019-20239, Department of ECE, Seoul National University (ironjr@snu.ac.kr).

the scene. Unlike ray tracing, which uses ray-scene intersection test, in ray marching, each ray is iteratively *marched* based on the output of a minimum distance estimator (DE) function defined by the geometries in the scene. By modifying the distance estimation function, we can render highly complex and repetitive scenes.

2 How to Run

The project is compiled and demonstrated in Linux machine running Ubuntu 18.04. The code is based on the Homework #5. There is no additional libraries used in the project. For OpenGL, I used GLFW library version 3.3. FreeImage library is used to capture the screenshots. The compilation procedure is as follows:

1. Download TIFF library from its official git repository[1] and compile it with `gcc`.
2. Download FreeImage library from its official website[2] and compile it.
3. Configure CMake using `ccmake` if needed. Compile the main project in directory `src`. Run the project with the generated executable `main` in directory `build`.

In the demo, you can navigate yourself with mouse and keyboard **W** (up), **A** (left), **S** (right), **D** (down), **left control** (below), and **space** (above). **Left shift** increases the movement speed ($\times 3$). Azimuth and altitude of the sun (directional light) is controlled with four **arrow keys** in the keyboard. The demonstration scene has *ten* different modes. You can switch between the scenes with the ten number digits (**0-9**) in the keyboard. The default scene at the startup is **Scene 0**. Details of each scene is explained in Section 3. Finally, there are *three* additional options you can toggle. Keyboard **I** turns on/off shadows. The shadows are shed by the only directional light in the scene. I implemented *soft shadow* using the edge detection powered by my ray marching engine. Details of the shadow is provided in Section 3.2. Keyboard **O** turns on/off the solar disc. Keyboard **P** switches between *realistic scene rendering* and *edge glow rendering*. Edge glow rendering is a direct visualization of detected edges/planes using my ray tracing engine.

3 Implementation Details

3.1 Main Demonstration

In this section, the details of *ten* different scenes of the demo are explained. You can switch between the scenes with the number specified in the list below.

0. If you run the demo, the first screen you see is only a rendered cubemap.
1. Basic geometric primitives including a plane, a cube, two spheres, and two tori with different materials are rendered. The parameters determining the shapes and the sizes of these objects are hard-coded in the shader `shader_ray_march.frag`. The objects are rendered with a ray marcher with exact *signed distance function* (SDF) in the shader. The normals are calculated from the numerical gradients of these SDFs. You can turn on/off the shadows using keyboard **I**.

[1] https://gitlab.com/libtiff/libtiff
[2] https://freeimage.sourceforge.io/

2. Now, the geometric objects are transformed, deformed, and merged. Smooth merging is implemented by a soft minimum of two SDFs. Smooth subtraction of a sphere from a cube is implemented by inverting the SDF of a sphere and taking an intersection using a soft maximum between two SDFs. Please note that the two tori in the scene has different materials with different *scattering mechanism*. A gold torus is using a non-ideal specular scattering to bounce the ray, while a white Lambertian torus is using a Lambertian scattering. The merging process of these tori is rendered very smoothly.

3. If you take a modular operation on the SDF of an object, you can render infinite amount of it. The scene containing infinite *dielectric refractive* spheres are rendered in real-time. A single RTX 2080 Ti GPU is used in the experiment.

4. These objects are then translated by a sine of its position. Continuous waves of these infinite glass spheres are generated.

5. The repetition of this kind can be extended to 3D. Infinite number of Lambertian spheres are rendered in 3D space and a purple torus is traversing through them.

6. The power of ray marching can be best demonstrated with rendering fractals. Here, I implemented a Menger sponge of level 7. With the shadows on (keyboard **I**), you can see this complex cube is rendered with a soft shadow in *real-time*. I recommend turning on the glow rendering option (keyboard **P**). The pixel value in this mode depends on the number of iterations of each ray before it converges (collides) or diverges to the open space.

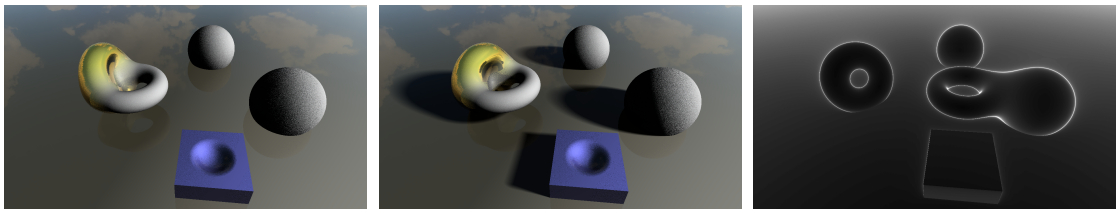
7. By applying modular operation on the fractal SDF, we can create arbitrarily many fractals in a single scene without computational overhead. This scene is rendered with *edge glow rendering mode*.

8. Complex ocean wave is generated as a height map and rendered with my ray marching renderer. I used the technique called procedural scene generation to generate the scene. The larger waves are hard-coded sinusoidal waves with multiple modes. Smaller waves are generated using 2D continuous noise. The continuity of the noise function and its complex structure embracing the hierarchy of scales can be obtained by using fractional Brownian motion (fBm) on a 2D noise generator. I used 2D wave noise for the base of fBm. It is a modified version of 2D simplex noise. Solar disc can be rendered by pressing keyboard **O**. Height map-ray intersection is obtained using iterative march and height checking. Finding the normal vector of an arbitrary point on a height map is trivial. Light reflection on water surface is modeled with Phong lighting model using high albedo. Comparing Figure 4i with Figure 4j, we can see that the surface of the water properly reflects the added clouds.

9. Likewise, we can generate 3D structures with fBm on 3D noise function. Cloud is generated as a density map using an fBm on 3D gradient noise. For each ray that intersects with a density map, multiplicative attenuation factor is calculated as the travelled distance, weighted by the density of the cloud. I applied exponential decay of the light based on this weighted distance. Finally for each marching step inside the cloud, a secondary ray is cast toward the light source (the sun). From this, the secondary weighted travelling distance is obtained. This value is used for calculating the illumination of the point passed by the primary ray.

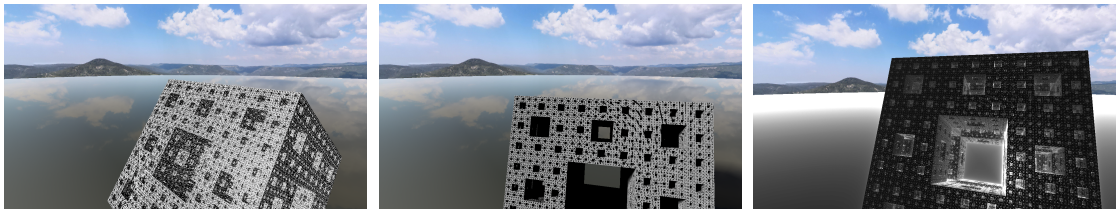
3.2 Edge Detection and Soft Shadowing

In this section we briefly discuss the effectiveness of ray marching on obtaining edge information of the objects. As an iterative algorithm, ray marching shader keeps track of the closest point of a ray point from the objects in the scene. If the ray misses an object by a small gap, the shortest distance, or the adaptive marching step decreases. We can use the number of steps taken before the convergence/divergence of a ray to detect the edges of an object, since this number *increases* as the ray misses an object by a smaller gap. This can be directly used for coloring the pixel, which we call *edge glow rendering* in this report. Moreover, the information on the closest distance can be used to determine the occlusion of a point. We can approximate the shadow attenuation using this metric to simulate soft shadows *without* casting multiple shadow rays. Figure 2 and 3 shows both application shown in the demo. Note that fractal objects are rendered nicely (and fast, too) using the edge glow rendering technique.



(a) Without modification. (b) With soft shadows. (c) Edge glow rendering.

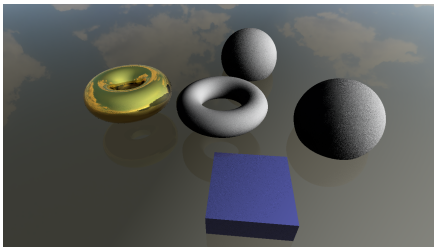
Figure 2: **Scene 2 with various rendering options**. Shadows of the geometries seem natural and smooth using soft shadow techniques. Edge glow rendering shows that ray marching can capture edge information *without* additional computations.



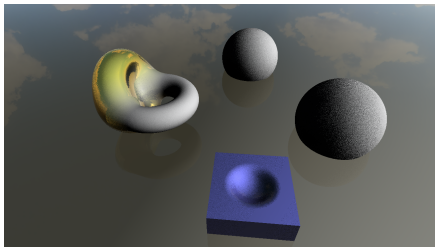
(a) Without modification. (b) With soft shadows. (c) Edge glow rendering.

Figure 3: **Scene 6 with various rendering options**. Adding shadows improves visual quality of the rendering of a fractal. This increase the system overhead by only a barely noticeable amount. The scene can be rendered in real-time. Edge glow rendering enhances geometric details of a fractal.

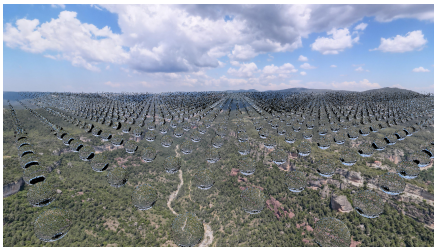
(a) Scene 0.



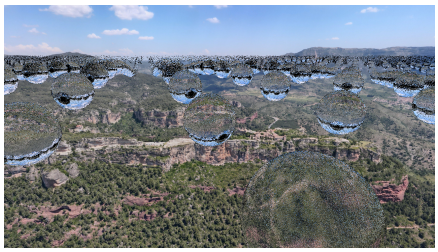
(b) Scene 1.



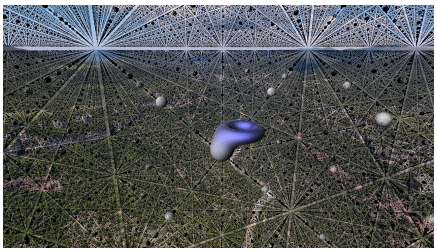
(c) Scene 2.



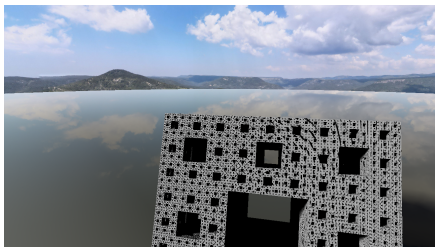
(d) Scene 3.



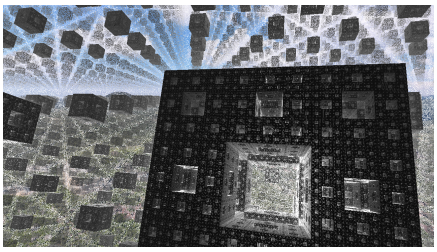
(e) Scene 4.



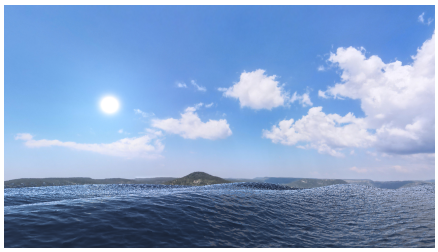
(f) Scene 5.



(g) Scene 6.



(h) Scene 7.



(i) Scene 8.

(j) Scene 9.

Figure 4: **Screenshots of each demo scene**.