Fractal Marching

Made by: Kaushik Vishwanath, Michael Swan, Mozhdeh Rouhsedaghat, Prashant Gupta, and Siddhant Shenoy

https://fractalmarchers.github.io/fm3/

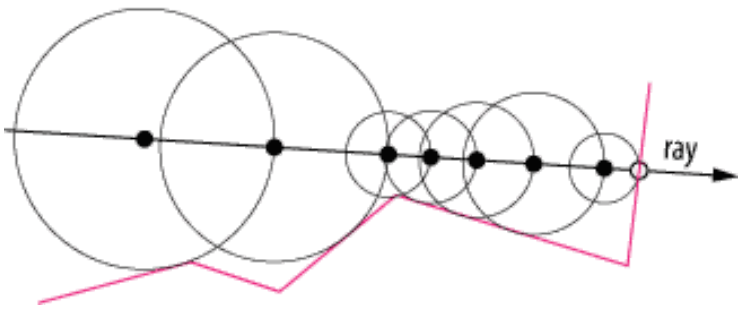
https://youtu.be/PimAbb3aBlk

Introduction

The purpose of this project is to use ray marching to render procedurally generated fractals in real time. Ray marching is a rendering technique in which we shoot imaginary rays from a camera that is looking at our scene. We march each of these rays in its direction until it intersects with an object in the scene or exceeds a number of permitted steps. Once the ray intersects with an object, we know what has to be rendered at that point in our scene. To calculate whether the ray intersects with an object in the scene, we make use of the Signed Distance Function (SDF). Ray marching is similar to ray tracing, where we calculate the intersection of the ray with objects in the scene. In ray tracing, the scene is defined in terms of explicit geometry of each object, whereas in ray marching, we make use of the Signed Distance Functions. The real power of ray marching lies in its ability to render shapes where this exact point of intersection is not as obvious.

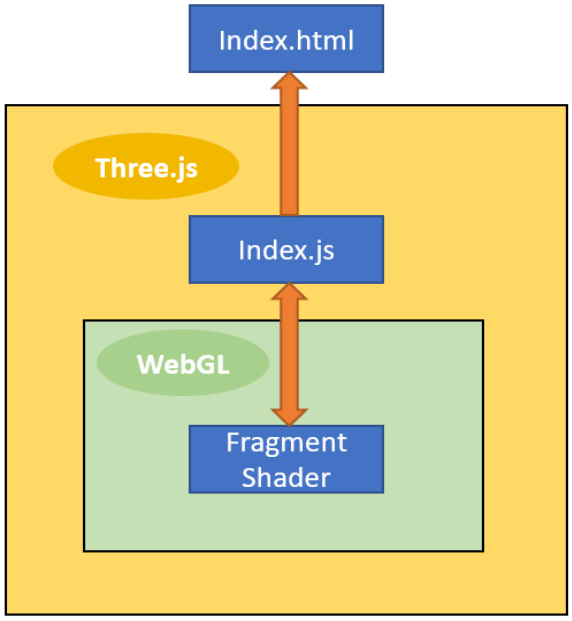
Signed distance functions (SDF), when passed the coordinates of a point in space, returns the shortest distance between that point and some surface in the scene. The sign of the return value indicates whether the point is inside the surface or outside (hence signed distance function).

We can march the ray forward by a very small value every time until it intersects with an object. However, this can be time-consuming and can be optimized with the help of SDFs. As we know, the SDF returns the signed distance of the ray to each object in the scene, we know that the nearest object to the ray in the scene will be at a distance greater than or equal to the minimum value returned by the SDFs. We can use this information and march the ray forward by that distance. This is called Sphere tracing (refer to the figure below) and this optimizes our Ray marching technique.



Design & Architecture

The design of this project was centered around rendering a user-configurable ray marched fractal in real time. To that end we looked at a number of OpenGL options and settled on what we were most comfortable with. Since each member of the team wanted to do their own implementation of ray marching, our final project actually includes two separate versions: one ThreeJS version (refer to the figure below) which combines almost all of our work, and another Unity version which focuses on building up a very interactive fractal.



Platform:

ThreeJS:

A javascript (js) library and API used to create and display animated 3D graphics on the web using WebGL. ThreeJs supports the use of shaders and we can pass variables from js to shader using uniforms. We can add multiple shader files just by concatenating GLSL code strings. Since we are using javascript, we also add some simple UI to the screen which allows us to modify uniform inputs.

Unity:

Ray marching in Unity can be implemented by writing a custom shader file. The shader runs directly on the GPU, thus making it easier to run the ray marching algorithm. Shader programs are written in HLSL language, by embedding snippets in the shader text, inside the Pass command. We create a custom material for the shader and set the hyperparameters of ray marching using a C# script, that can be controlled by the player at runtime.

GLSL:

GLSL is a C-style primary shading language for several 3D graphics APIs like OpenGL and WebGL. GLSL scripts are executed directly on the GPU instead of the CPU hence providing massive speedups for 3D real-time graphics. Modern games achieve photorealism by performing computations parallely using GLSL. Other applications of GLSL include the creation of VR, AR, Parallel Computing and many other applications. Hence, we decided to use this language which is a highly sought after skill for graphics designers.

Implementation

Lighting:

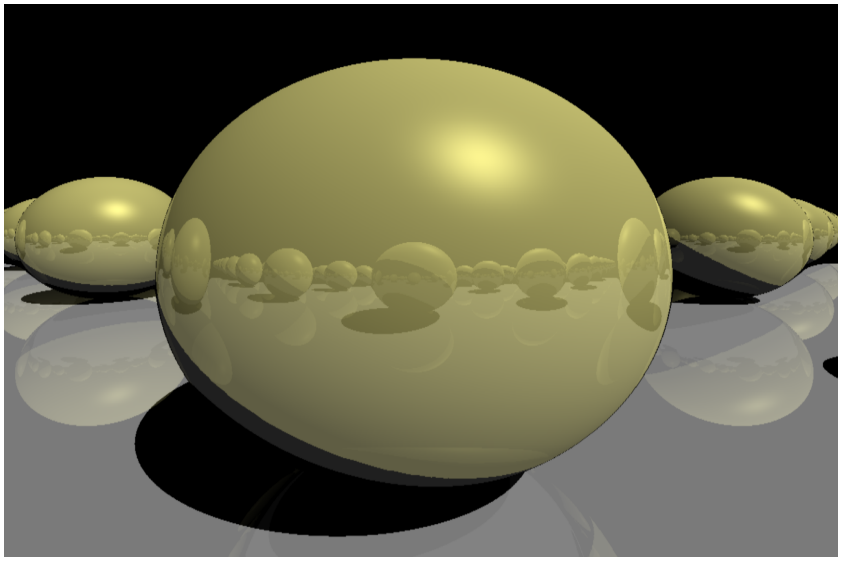
To obtain the normal at each intersection point, we compute the gradient of the SDF at the point as the vector with minimal distance from an object is orthogonal to it. Then using the normal vector at the intersection point, the vector pointing to the light, and the vector pointing to the camera, the diffuse and the specular lights are computed and added to the ambient light. The result is the color at the intersection point.

Reflection:

For implementing the reflection, after computing the color at the current point, the direction of the reflection vector is computed (as explained in the lectures) and a ray is marched from the current point in the reflection direction . If the reflection ray hits anything, the color at the new intersection point is computed, multiplied by the reflection coefficient and added to the computed color at the main point.
The reflection depth shows how many times the reflected ray will bounce between objects. The final color of the main point is the weighted sum of its initial color and the color of the points the reflected ray hits.

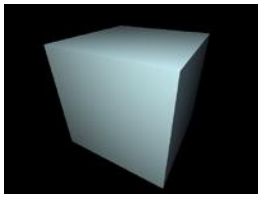
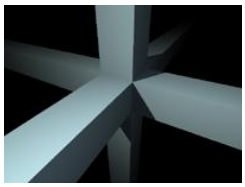
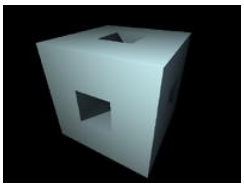
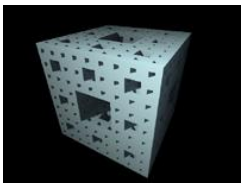
Shadow:

When computing the lighting at a point, we march a ray from the current point toward the light source. If it hits any object, it means the light is blocked and can not reach the point so we set the color of the point to black.



Collision:

Basically Ray marching is nothing but a distance estimator. We can use it to detect collisions. So in our scene, just like our camera, we took one object (say sphere) as our current position and started marching towards the other. If the distance between them is less than some small value, we change the direction of the sphere. Using this we can create our own Physics system and introduce gravity etc. Though just estimating collision won't be enough, as we also need to figure out the angle at which our sphere would have to move after collision, which can be solved using the

point of collision and its normals. Other factors includes mass, velocity, so on and so forth.

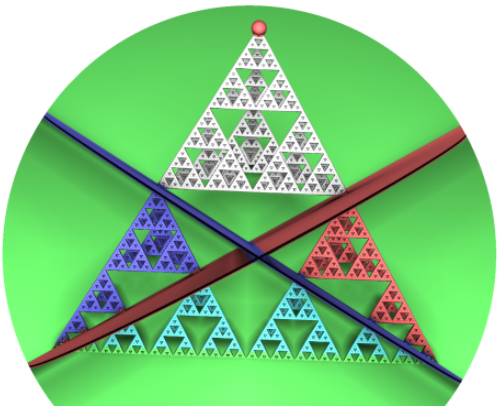
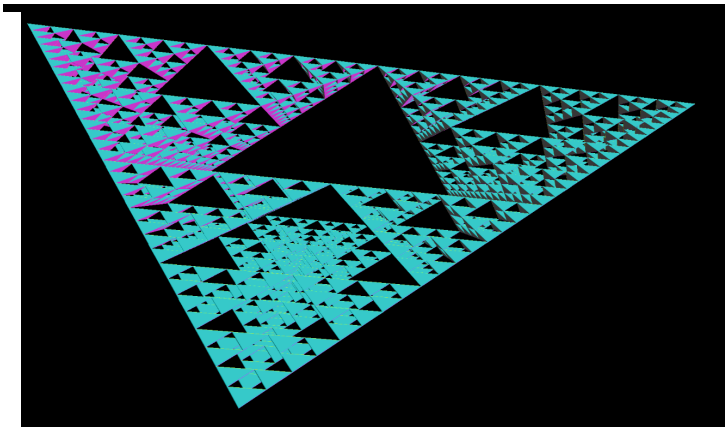
Fractals

Menger Sponge:

The Menger sponge (also known as the Sierpinski cube) is a fractal curve. It is a three-dimensional generalization of the one-dimensional Cantor set and two-dimensional Sierpinski carpet. The construction of a Menger sponge can be described as follows:

1. Begin with a cube(Fig 1).
2. Take the cross structure (Fig 2) and delete it from the cube, you will get Fig 3.
3. Now taking modulus of the cross makes it infinite. Divide its size by 3 and repeat.

Sierpiński tetrahedron:

The Sierpiński tetrahedron is a 3D version of the Sierpiński triangle. The basic algorithm is similar to the Menger Sponge, in that it is all about dividing your whole into increasingly smaller parts, except in this case you use a tetrahedron. The subtraction approach outlined above could be applied to the tetrahedron, but we decided to go with a space folding approach which is outlined in the Syntopia blog, written by Hvidtfeldt. Essentially what is happening is we select symmetric axes about which to reflect about and mirror our tetrahedrons about those axes. Do this for enough iterations and you have your Sierpinski tetrahedron.

Miscellaneous

Noise is seen everywhere in nature so we wanted to add some of that chaos to our scene. **Fractional Brownian Motion (fBM)** does a good job of replicating “natural” noise or texture. When we look at objects like rocks, grass, etc. we see that they are composed of statistically different yet self-similar structures. The fractional part is what adds self-similarity while the brownian motion adds the randomness.

For the fBM technique, 3 things are important governing factors:

1. **Noise Function**: We used value noise instead of the more popularly known Perlin noise which computes gradients. Value noise divides the 3D space into cubes whose points are then assigned random values. Interpolation between these 8 values then gives the noise pattern. See Fig. 3 for value noise.
2. **Self-Similarity**: Self-similar means how soon does the pattern repeat itself? This indicates the memory of the noise pattern and is determined by the *Hurst exponent (H)*. We used a value of 1 which leads to lesser chaos by using lower frequency initial signals commonly used in CG.
3. **Level-of-detail (LOD)**: This is determined by the number of octaves computed. It is the number of times we exponentially decrease the amplitude and geometrically increase the frequency of the initial noise function (wave).

$$fBM(p) = n_1 + n_2 + n_3 + \ldots$$

$$n_1 = (2^0)^{-H} * noise(2^0 * p) \quad \text{For no\_octaves} = 1$$

Add

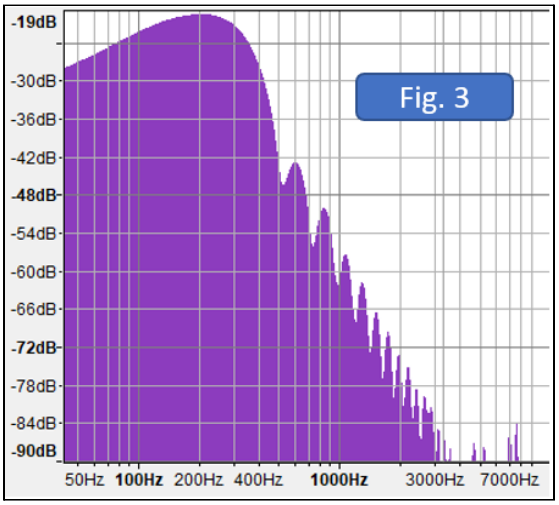
$$n_2 = (2^1)^{-H} * noise(2^1 * p) \quad \text{For no\_octaves} = 2$$

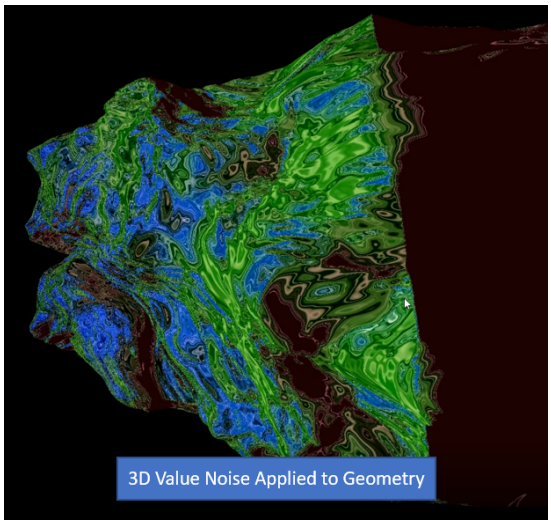
Add

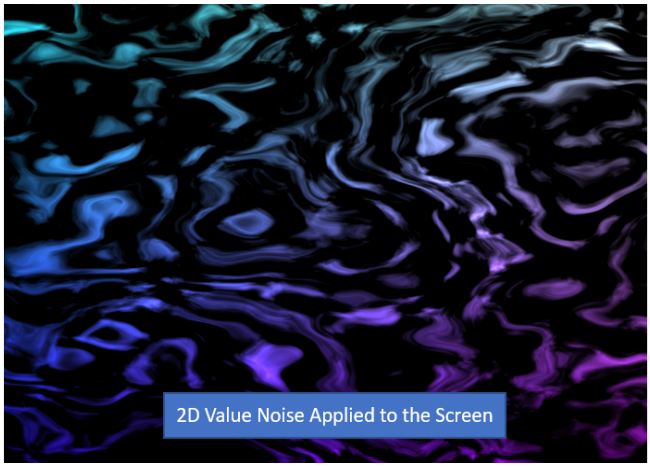
$$n_3 = (2^2)^{-H} * noise(2^2 * p) \quad \text{For no\_octaves} = 3$$

Add

...







Note that the simple **fBM** returns a scalar value for a given 3D point. To add more chaos, we do this twice to perform first domain warping (f(p) = fBM(p + fBM(p))). This is then added to the **SDF** (using the 3D location as **p** in the formula above) and to the **lighting** (using the ambient, diffuse and specular components as **p**).

Color change overtime is accomplished by using the shader playback time (iTime). We change r, g, and b values of the object color by multiplying them with different coefficients obtained from the playback time. Each coefficient is between 0 and 1 and changes with a different rate as time passes so different colors are generated. In order to have a smooth transition between colors, each coefficient repeatedly increases from 0 to 1 and then decreases from 1 to 0.

Conclusion

Ray marching is a great technique to leverage the power of GPU-based parallel processing while at the same time handling complex CG effects such as lighting, reflections and shape-mergers. It provides a different way of thinking about geometry using mathematical functions which has some interesting side-effects. That said, ray marching can get computationally inefficient quite quickly, so one must always be careful that their signed distance functions do not get too large.

The traditional graphics pipeline has its advantages such as handling a large number of objects in the scene. Ray marching becomes incrementally slower since we need to process each pixel for each object present in the scene to find the closest counterpart. Not to mention supporting tasks such as reflections and noise pushing the limits of the GPU further. This project was a great experience, and perhaps future GPU improvements will enable really interesting systems based on ray marching.

References

https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm
https://www.iquilezles.org/www/articles/menger/menger.htm
https://www.iquilezles.org/www/articles/normalsSDF/normalsSDF.htm
RayMarching For Beginners - by Art Of Code
Fractional Brownian Motion - by Inigo Quilez
Value v/s Gradient Noise - by Inigo Quilez
N-Dimensional Noise Computation - by Inigo Quilez
Warping - by Inigo Quilez
Noise - Book of Shaders - by Patricio Gonzalez Vivo & Jen Lowe