Real-time Progressive Fractal Rendering

Corey H. Walsh
6.837, Massachusetts Institute of Technology, chwalsh@mit.edu

Problem & Motivation

Traditional ray traced rendering techniques are typically memory-bound, due to the divergent memory access pattern of most ray tracing algorithms. Many techniques exist to mitigate this, such as packeted BVH traversal, but they usually come at the expense of simplicity or speed.

Fractal geometry is interesting to me because of its near infinite complexity, with no need for in-memory scene description. While many fractal equations can be computationally expensive to evaluate, the lack of memory accesses mean that these types of algorithms are ideally suited towards the massive parallelism available in modern GPU compute architectures. Fractals are also of great personal interest because of their artistic qualities and applicability to non-photorealistic rendering.

For my project, I wanted to exploit this easy parallelism in order to interactively render fractal geometry. To ease the inevitable trade off between speed and quality, I also wanted to utilize progressive rendering such that higher quality renderings could be obtained by simply not moving the camera.

Background

I. Fractals

Fractals are mathematical sets which illustrate self similarity at a wide range of scales. While not all fractals are exactly the same at every scale, most of them exhibit a high degree of geometric complexity.

In addition to simple geometric shapes, I implemented three fractal equations in CUDA C. Fractal geometry has a very active community, with many great resources such as fractalforums.com, Inigo Quilez's site www.iquilezles.org and Mikael Christensen's blog *Syntopia* all of which proved invaluable throughout this project.

II. Distance Equations

Fractal geometry is typically represented by a single equation which gives the minimum distance between any point in space and the fractal surface, known as a distance equation ("DE"). While it might seem surprising that such an equation would exist to describe an infinitely complex geometry, many such formulae have been discovered.

What's more, distance equation open the door to some interesting effects by transforming space with standard affine transformation matrices, or even non-linear operations like repetition or twisting.

III. The Quaternion Julia Fractal

The first fractal I implemented was the Quaternion Julia fractal. The Julia fractal is created by repeatedly squaring complex quaternions of the form

$$q = r + a \cdot i + b \cdot j + c \cdot k \qquad \text{eq. 1}$$

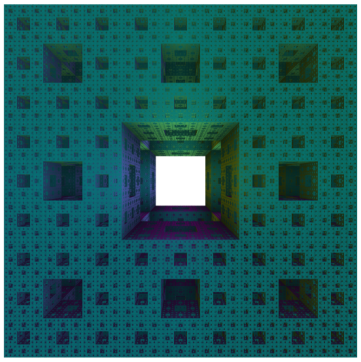
where r, a, b, and c are real, and i, j, k are imaginary. The iterated function is of the form

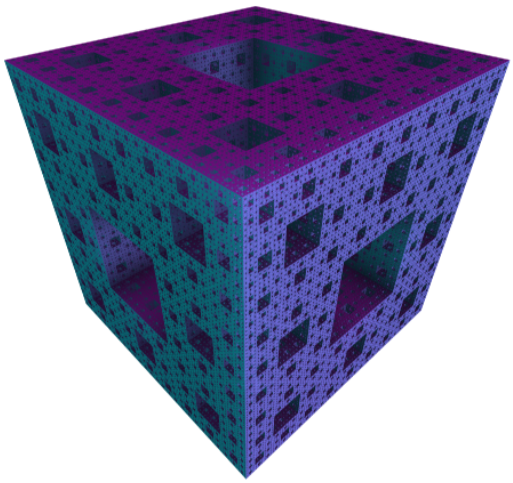
$$z_{n+1} = z_n^2 + c \qquad \text{eq. 2}$$

z_0 is the point under consideration. z and c are both quaternions of the above form. A point is considered inside the Julia if this iterated function tends to zero as n approaches infinity. After the function is iterated some number of times, we are able to obtain a lower bound distance estimate to the fractal surface using the method presented by [2]

$$d(z) = \frac{z_n}{2 * z'_n} \cdot \log(z_n) \qquad \text{eq. 3}$$

IV. The Menger Sponge

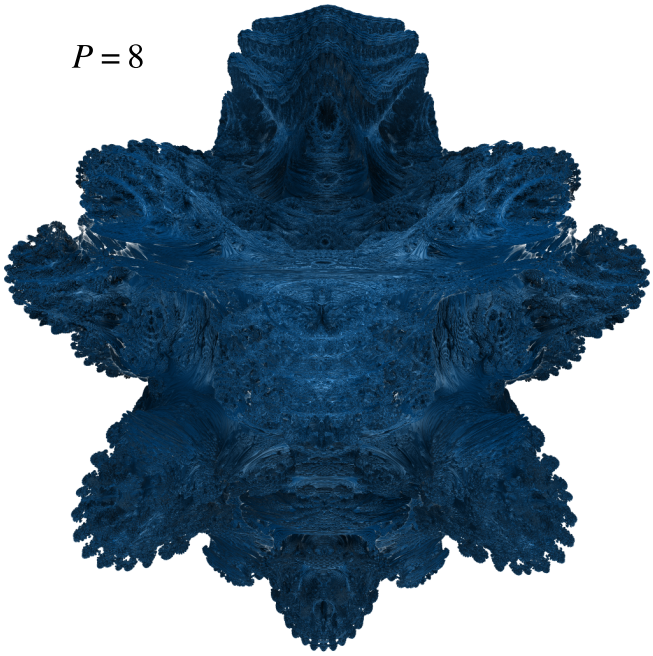




Next came the Menger sponge fractal, the 3d generalization of the Sierpinski carpet. This fractal has the interesting property of having infinite surface area, and zero volume at the limit of iteration.

A Menger Sponge can be procedurally generated by:
1. Drawing a cube
2. Dividing it into 27 equally sized sub-cubes
3. Removing the 7 central cubes - the ones that lie in the center of each face and the center of the cube
4. Recursively iterating steps 2 and 3

V. Mandelbulb Fractal



Finally, the Mandelbulb fractal is similar to the Julia fractal in that it is achieved by repeatedly iterating a function that raises a complex number to some power.

$$w_{n+1} = w_n^P + c \qquad \text{eq. 4}$$

In this case, we consider the polar coordinates of a point in space, raise the length to some power and double the angles with the x axis. This process is explained more in depth by Inigo Quilez in [3]. After the last iteration, a distance estimate is found with the same formula as eq. 2.

In practice, I found that this estimate was often an overestimate, and that stepping by that amount lead to over-stepping the surface. To mitigate this, a scaling factor of less than 1 was applied to the distance estimate before ray marching. More information on ray marching the Mandelbulb can be found in [3,4,6].

VI. Transformations

In addition to standard affine transformation matrices, distance equations allow for the easy manipulation of geometry by warping and folding space. For example, the modulo operator can be applied to each component the position of the point considered in order to create the illusion of infinite geometry.

Fractal geometry may be additively combined by simply taking the minimum of two evaluated DEs, or they may be intersected by taking the maximum.

VI. Ray Marching

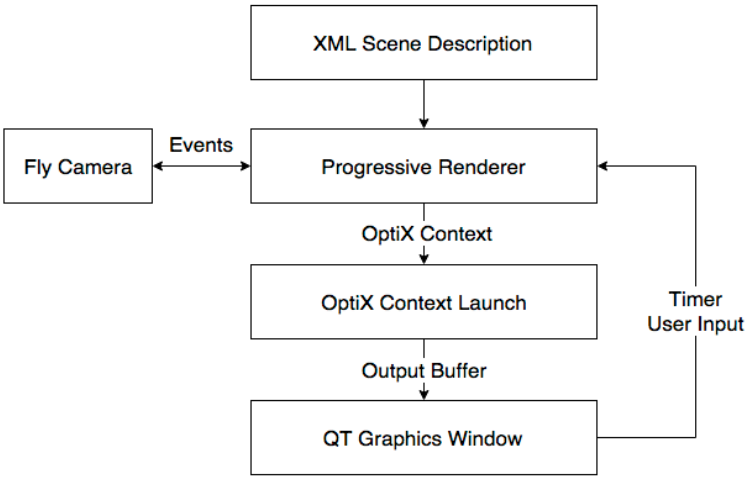
In order to render distance equations, the ray tracing algorithm must be adapted to efficiently find points of intersection. Ray marching is the most common approach of doing so. This process is extensively described in [5,7]. The process is conceptually simple, and may be accomplished via the following pseudocode:

1. Compute ray origin and direction
2. Evaluate distance equation to find d
3. If diverges or max iterations reached, break
4. If d is less than some small epsilon, report an intersection, else continue
5. Step forward length d from the ray origin in the ray direction
6. Goto 2

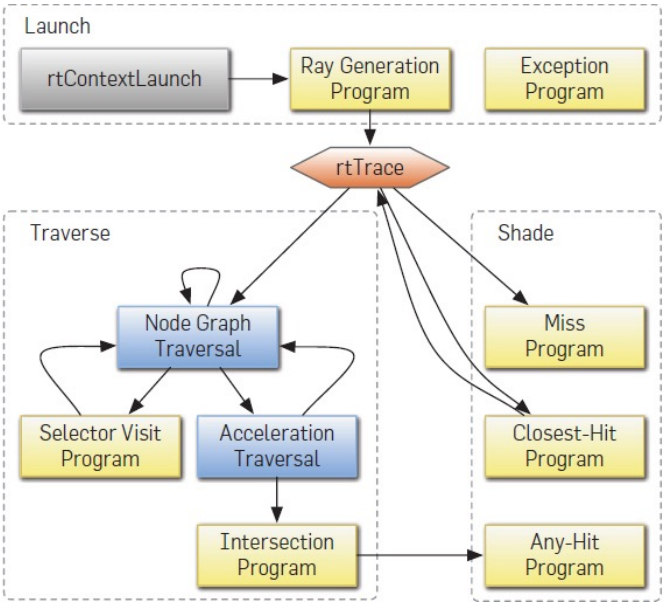
Once an intersection is reported, we can estimate the surface normal via the numerical gradient of the distance equation by sampling 6 points offset from the intersection point along the axes by some small epsilon. For signed distance equations, this approximation is found to be fairly accurate in practice.

APPROACH

I used Nvidia OptiX along with CUDA to generate a single monolithic GPU kernel that executes on the graphics device. This is accomplished by creating a number of ray generation, intersection, and shading CUDA kernels known as OptiX programs, as well as a few other programs to handle cases such as ray misses and exceptions.



HIGH LEVEL SYSTEM OVERVIEW



OPTIX RAY TRAVERSAL PIPELINE, FROM [1]

The use of OptiX allows various fractal scenes to be composed with different rendering settings, and exposes spatial control of the scene via hierarchical transformation matrices. While the acceleration data structures offered by OptiX are generally not required for ray marching, I found that some interesting effects could be achieved by applying a bounded volume hierarchy to infinitely repeating scenes. In this way, it was possible to create a nearly infinite number of geometric primitives inside of a finite cube.

I. XML Scene Parsing -> Monolithic GPU Kernel

The various kernels are combined via a custom XML scene description parser which compiles all of the programs and variables into a valid OptiX context, including any hierarchical affine transformations. This context offers a launch method, which initiates the process of ray generation, intersection, and shading (any-hit programs) in a single monolithic CUDA kernel.

III. Progressive Rendering

In order to progressively improve the visual appearance of the scene, I maintain an accumulation buffer on the device which is new samples are added to. Every 30 milliseconds, the host maps and copies an output buffer from the device to the display buffer.

A stratified sampling scheme is used to randomize each sample's location, and other ray marching parameters are slowly increased to improve clarity at the expense of render time. For example, the fractal iteration count is increased, and the maximum allowed steps is increased by up to a factor of 10. Samples were combined with with the accumulation buffer by equal importance averaging. While better techniques exist, this has the benefit of being very fast and straightforward, and in practice yields antialiased, visually pleasing images that quickly refine.

IV. Camera

Scene interactivity is facilitated with a custom fly through camera that the user can control with keyboard and mouse controls. The mouse position controls pitch and yaw, and the "**WASD**" & "**RF**" keys control the 6 direction strafing speed. The left shift and command keys on a mac are used to speed up or slow down strafe speed so that finer grain control can be achieved near to the fractal surface. A few convenience shortcuts were also added:

L toggles a camera position and orientation lock, so that progressive refinement may occur
. (period) saves the current output buffer to an image file in the host's file system
M toggles mouse control
Arrow keys control the yaw and pitch
Q, E controls camera roll

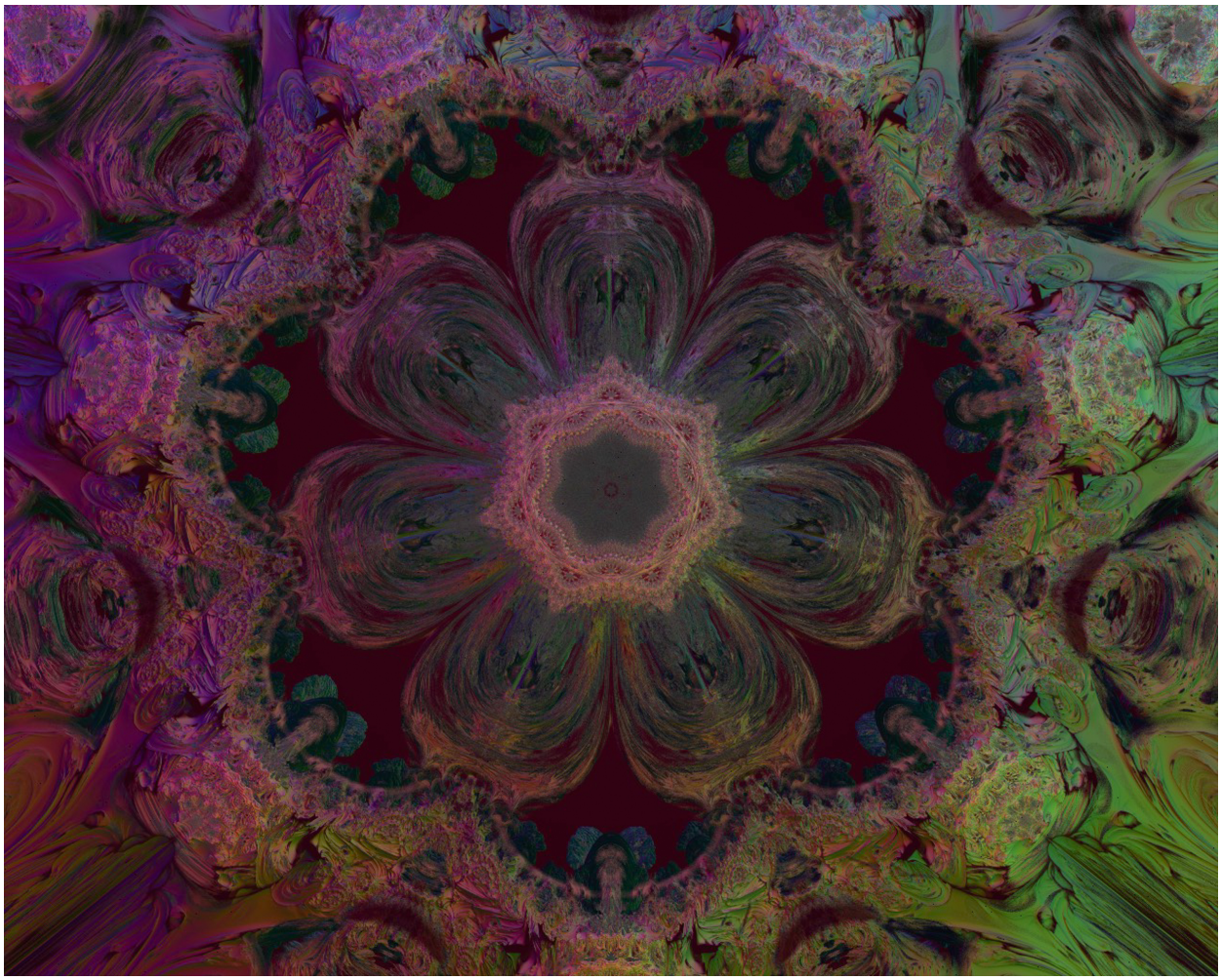
IV. Shading

A number of shading methods were added, including Phong shading, normal shading, fake ambient occlusion, fog, and distance visualization.

RESULTS

While it is not yet perfect, very nice results were obtained with my GPU rendering engine with real-time performance on my GTX 970 graphics card. The use of a custom XML scene parser allowed for easy modification of rendering and scene variables, which greatly aided in the process of development.

The performance of this algorithm is highly dependent on the type of distance equation being rendered, and the baseline rendering parameters, but for most scenes interactive performance rate was achievable with acceptable visual quality. After around 1000 additional randomized sampling rounds, the images were generally fully converged.



Videos of interactive usage are available on YouTube at the below domains, and more pictures are available in the Stellar submission.

https://www.youtube.com/watch?v=PhF4qsi0Kcs
https://www.youtube.com/watch?v=APLJTSTGxi8
https://www.youtube.com/watch?v=0dCRqrxKf7s

FUTURE WORK

In the future, I would like to incorporate additional ray tracing effects such as refraction, reflection, and hard/soft shadow. It would also be interesting to incorporate more camera effects such as depth of field and bloom.

This algorithm could also be easily adapted to generate slices necessary for 3D printing, which could result in some very interesting objects.

On the user interface side of things, I would like to expose an XML tree editor so that parameters could be manually adjusted without requiring a restart of the program. This functionality could also be extended to create sequences of scene files for use in high-quality offline animation rendering.

ACKNOWLEDGMENT

I'd like to thank Wojciech Matusik, and all of the 6.837 staff for a very informative and entertaining semester. The creation of this rendering engine would not have been possible without their instruction and guidance throughout the semester.

REFERENCES

1. http://cacm.acm.org/magazines/2013/5/163758-gpu-ray-tracing/fulltext
2. http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf
3. http://www.iquilezles.org/www/articles/mandelbulb/mandelbulb.htm
4. http://www.fractalforums.com/3d-fractal-generation/true-3d-mandlebrot-type-fractal/525
5. http://9bitscience.blogspot.com/2013/07/raymarching-distance-fields_14.html
6. http://2008.sub.blue/blog/2009/12/13/mandelbulb.html
7. http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i