CS500 – Final Project: Deterministic 3D Fractals

PROBLEM DESCRIPTION:

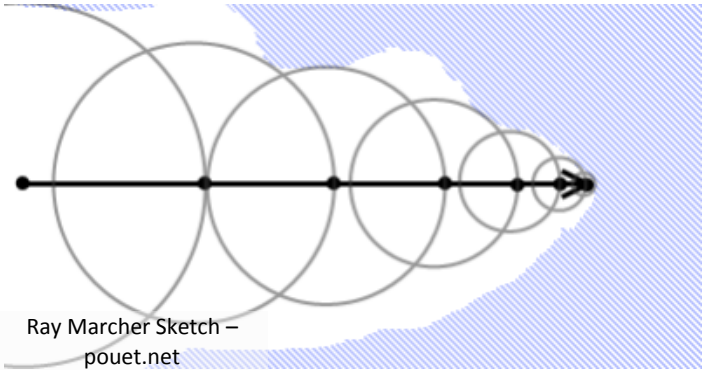
The objective of the project is to build a 3D visualizer of fractals. For that, it is needed a way for coloring them as well as a way for getting the structure information of them. The paper explains how to get Julia sets, which is a specific family of the 3D fractals.

PAPER'S SOLUTION:

The paper 'Ray Tracing Deterministic 3-D fractals' by John C. Hart, Daniel J. Sandin and Louis H. Kauffman, gives a solution that makes use of ray marching.

Ray marching

Ray marching is a technique that makes small steps from the ray origin in the given direction and checks if each iteration point is on the geometry or not. For speeding up, unbounding sphere volumes can be used. These are spheres that do not enclose any kind of geometry. So, by computing them in each iteration, the ray can advance the given amount of distance. The radius of these spheres will be computed with distance estimators. Ray marching can be really slow with the scenes used during the course, as it needs to make the intersection checks more than once per ray thrown. However, if complex geometries like fractals are being drawn, this comes in handy, as it only needs to estimate the distance to them.



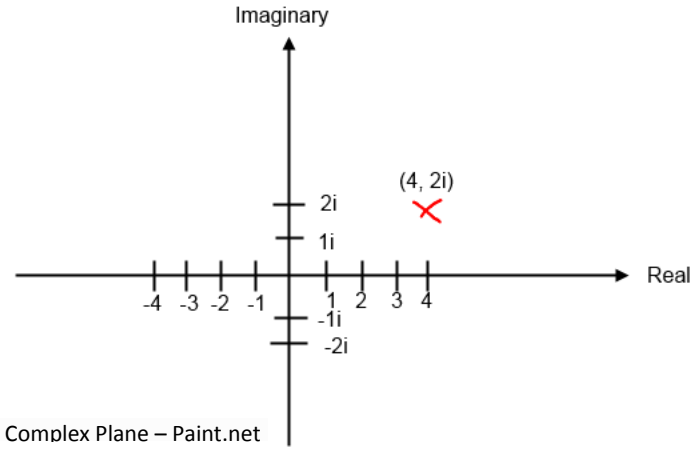
Ray Marcher Sketch – pouet.net

Once a collision happens, the normal needs to be computed (mostly for lighting computations). For getting the normal of the surface, the paper suggests two ways. On the one hand, it suggests to approximate the normal with a cross product with two vectors of the surface. On the other hand, it suggests to compute the gradient by taking near surface points with a defined epsilon. The second one is a better option, as it does not need the z-buffer, which is something not needed in ray tracing.

$$N_x = de(z_{x+\varepsilon,y,z}) - de(z_{x-\varepsilon,y,z})$$
$$N_y = de(z_{x,y+\varepsilon,z}) - de(z_{x,y-\varepsilon,z})$$
$$N_z = de(z_{x,y,z+\varepsilon}) - de(z_{x,y,z-\varepsilon})$$

Complex Numbers

The paper explains how to draw Julia sets using ray marching, but for that, first a simple understanding on complex numbers is needed. A complex number is composed on 2 components: a real part and an imaginary one. This number can be represented on the complex plane as we represent a 2D vector:



Complex Plane – Paint.net

The mathematical operations are also different. The addition is defined as the addition of each components separately (as in the vector spaces). However, the multiplications are a little bit tricky. Multiplications are similar to the ones between polynomials, but with a particularity: If we multiply two imaginary numbers, then the result will be minus one. Here we have an example:

$$(2+i)\cdot(1+3i) = 2+6i+i+3(ii) = 2+7i-3 = -1+7i$$

Quaternions

However, for drawing 3D fractals, another type of complex numbers are used, as the presented ones can only be represented in 2D. For that, quaternions come to the rescue. Quaternions are complex numbers composed by a real and 3 imaginary elements. Note that as the rendering will be in 3D and quaternions are in 4D, only a slice of it will be rendered.

The operations between quaternions are the same than for the complex numbers, but with extra rules for the multiplications. Here are all the rules to take into account:

$$i^2 = j^2 = k^2 = -1;$$

$$ij = k;\ jk = i;\ ki = j;\ ji = -k;\ kj = -i;\ ik = -j;$$

Julia Sets

Julia sets are a group of fractals obtained by iterating complex numbers (in our case quaternions) in a holomorphic function. Holomorphic functions are functions that take complex numbers are return complex numbers, but with the particularity that are differentiable in all the domain. This detail is important, as the distance estimator to these fractals will need the derivative of the holomorphic function. The function used on the paper is the classical quadratic formula:

$$f(z) = z^2 + c$$

Distance Estimator

The paper defines a lower bound for the quadratic family of fractals in the following way:

$$de(z) = 0.5 \cdot \ln(|f^n(z)|) \cdot \frac{|f^n(z)|}{|f'^n(z)|}$$

In fact, this is the distance estimator that needs to be used on the algorithm. For that, the n^{th} iteration of the quadratic formula seen before and its derivative are needed. The derivative is obtained in the following way:

$$f(z) = z^2 + c$$

By the chain rule: $$f'^n(z) = 2 \cdot f^{n-1}(z) f'^{n-1}(z)$$

MY SOLUTION:

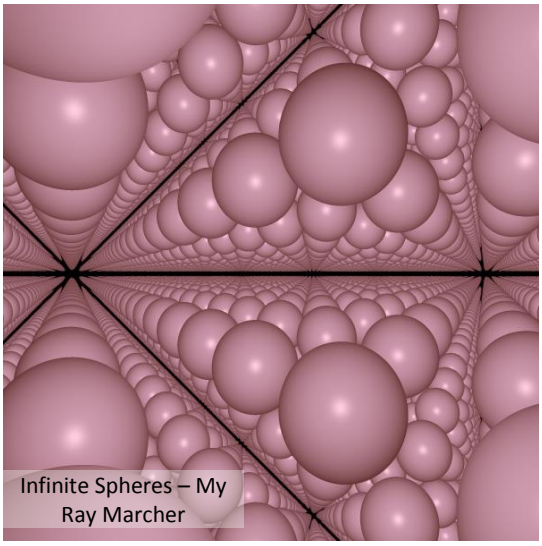
First, I started implementing the ray marcher. It is a simple class that takes a ray, a function pointer to a distance estimator and makes user defined amount of iterations. If more iterations are made, the more accurate the result will be. When the distance of the unbounding volume is smaller than the user defined epsilon, then it collided. In this case the normal of the surface will be computed using the gradient. Pseudo code:

```
float RayMarch(rayOrigin, rayDir, normal)
{
    float traveledDistance = 0.0;

    for(iterations from 0 to maxIterations)
    {
        vec3 point = rayOrigin + traveledDistance * rayDir;

        float distaceEstimated = DistanceEstimator(point);

        traveledDistance += distaceEstimated;

        if (distaceEstimated < epsilonDistanceEstimator)
        {
            vec3 xDiffDir{ epsilon, 0.0, 0.0 };
            vec3 yDiffDir{ 0.0, epsilon, 0.0 };
            vec3 zDiffDir{ 0.0, 0.0, epsilon };

            normal.x = mCurrDistanceEstimator(pointToCheck + xDiffDir) - distaceEstimated;
            normal.y = mCurrDistanceEstimator(pointToCheck + yDiffDir) - distaceEstimated;
            normal.z = mCurrDistanceEstimator(pointToCheck + zDiffDir) - distaceEstimated;

            normal = glm::normalize(normal);

            return traveledDistance;
        }
    }

    return NO_INTERSECTION;
}
```

Take the farthest point possible and perform the distance estimator for it

If collided, compute the gradient

There is a difference with the paper in the way the gradient is computed, though. Instead of computing the gradient between the points around in both directions, it only takes them in one direction and use the current computed one. The result is almost the same and some computations costs are saved.

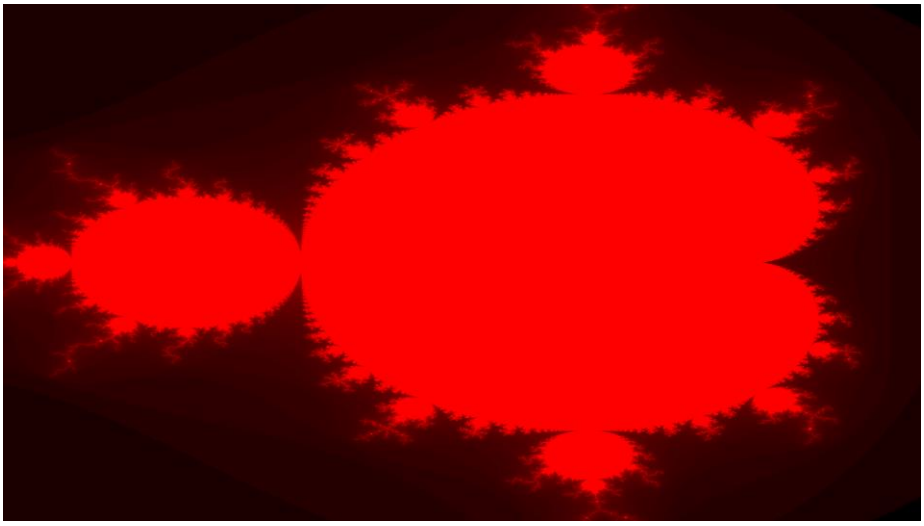


Infinite Spheres – My Ray Marcher

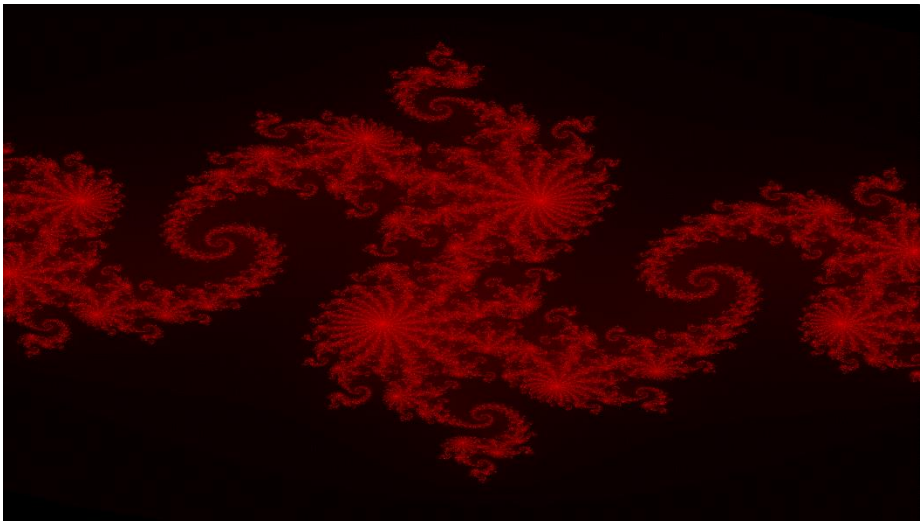
For testing it, I implemented a simple infinite sphere distance estimator, giving already interesting results. With a simple modulus on the intersection against a sphere, suddenly, we get infinite many spheres, without almost extra computations. To render this in the traditional ray tracing, would be really slow.

After ensuring that the ray marcher was working, I implemented some 2D fractals on shadertoy. I made it to understand better how fractals work.

Here are attached my results:



Mandelbrot set
drawn with shadertoy



Julia Set (c = -0.8 + 1.156i)
drawn with shadertoy

To obtain these results, the program will iterate through all the pixels and for each of them, it will perform the quadratic formula explained on the paper, using the x as the real component and the y as the imaginary. In this way, if during this iterations it ends bounded, it will be drawn in red, if not it will be shown in black. I found interesting how the only difference between the Mandelbrot and Julia sets is the c on the quadratic formula. In Mandelbrot sets, the c will be equal to z while in the Julia sets, c will be equal for all the points. In fact, the Julia and Mandelbrot sets are very closely related.

In 3D however, the images are obtained applying the distance estimator explained above. For that, the n^{th} iteration and its derivative needs to be computed using a for loop. In this for loop, each iteration will be computed until the maximum iterations are done or the point is farther from the origin than the escape radius. These points are not interesting, as they will tend to go to infinite (in other words, they are not part of the fractal). After the loop the distance estimator formula is computed. Pseudo code:

```
float DEJuliaSet(vec3 point, avec4 c)
{
    //Note the kth element will be 0 at the start
    vec4 z{ point, 0.0f };
    float dzLength = 1.0f;
    float zLength = 1.0f;

    for (i from 0 to maxIterations)
    {
        vec4 z2;
        z2.x = z.x * z.x - z.y * z.y - z.z * z.z - z.w * z.w;
        z2.y = 2.0 * (z.x * z.y);
        z2.z = 2.0 * (z.x * z.z);
        z2.w = 2.0 * (z.x * z.w);
```

Compute the square of a quaternion using the rules explained above

```
        z = z2 + c;
```

Compute the next iteration point

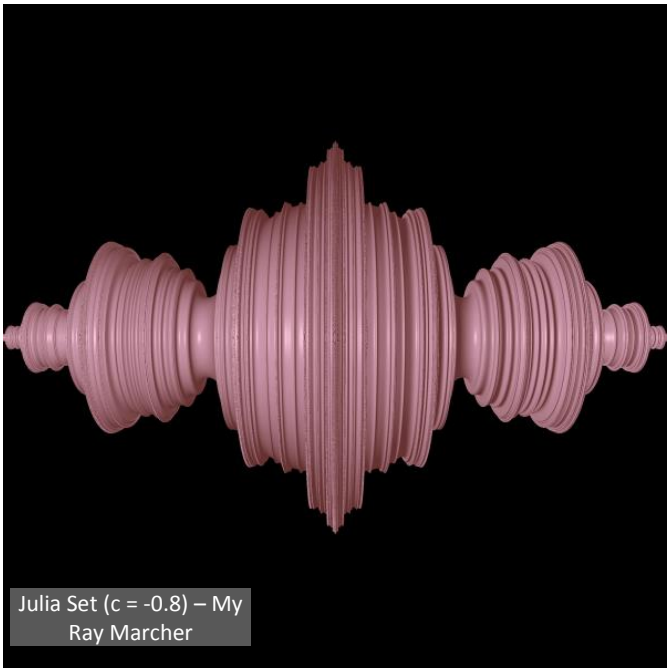
```
        zLength = length(z);

        dzLength = 2.0 * zLength * dzLength;
```

Compute the next derivative

```
        if (zLength > 2.0)
            break;
    }

    return  0.5 * ln(zLength) * zLength / dzLength;
}
```

If it is bigger than the escape radius, exit

Finally, apply the lower bound distance estimator to the complex planes



Julia Set (c = -0.8) – My Ray Marcher

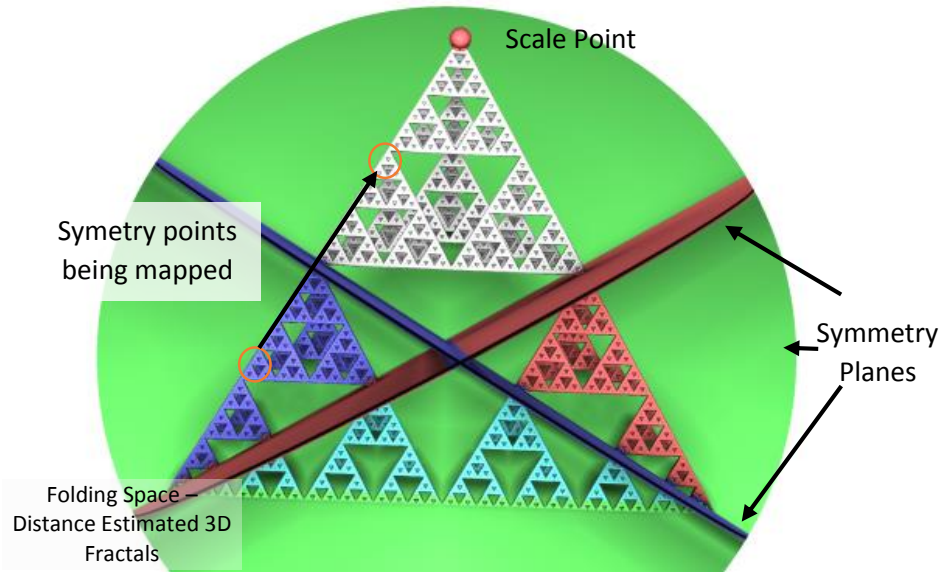
This Julia set is drawn using the above code. The coloring is implemented using the default Phong model. Also, it uses super sampling to get smoother results (as it tends to be a little nosier).

Basically, with this, the project is done. The ray marcher is able of drawing Julia sets and coloring them. However, I wanted to experiment with the project and learn more cool staff. In the next apart, it is explained what extra things are implemented on the ray marcher.

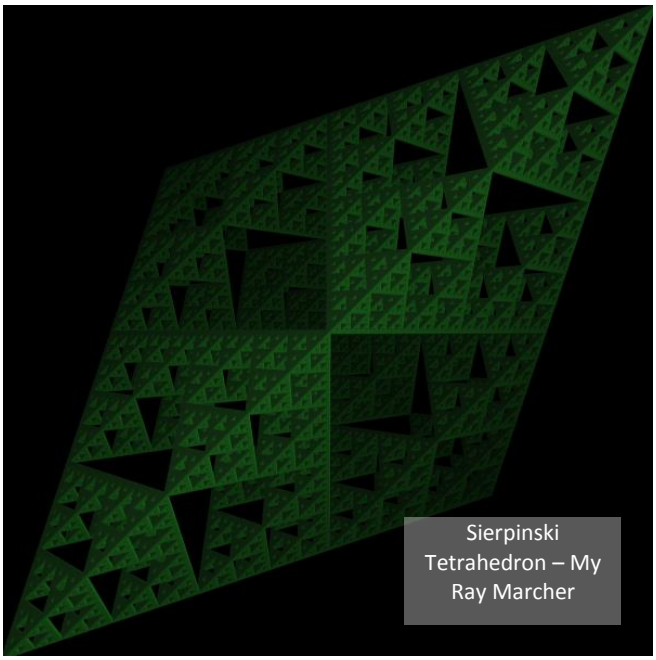
EXTRA THINGS:

Sierpinski Tetrahedron

First, let's talk about the Sierpinski Tetrahedron. This is a typical fractal that shows tetrahedrons inside of other tetrahedrons infinitely. As it is explained in the fantastic blog 'Distance Estimated 3D Fractals' by Mikael Hvidtfeld Christensen, it can be approached through a distance estimator that uses folding operations. These operations will take the mirror points using the symmetry planes below in order to pass all the points to the same octant and then scale from a specified point.



Folding Space – Distance Estimated 3D Fractals

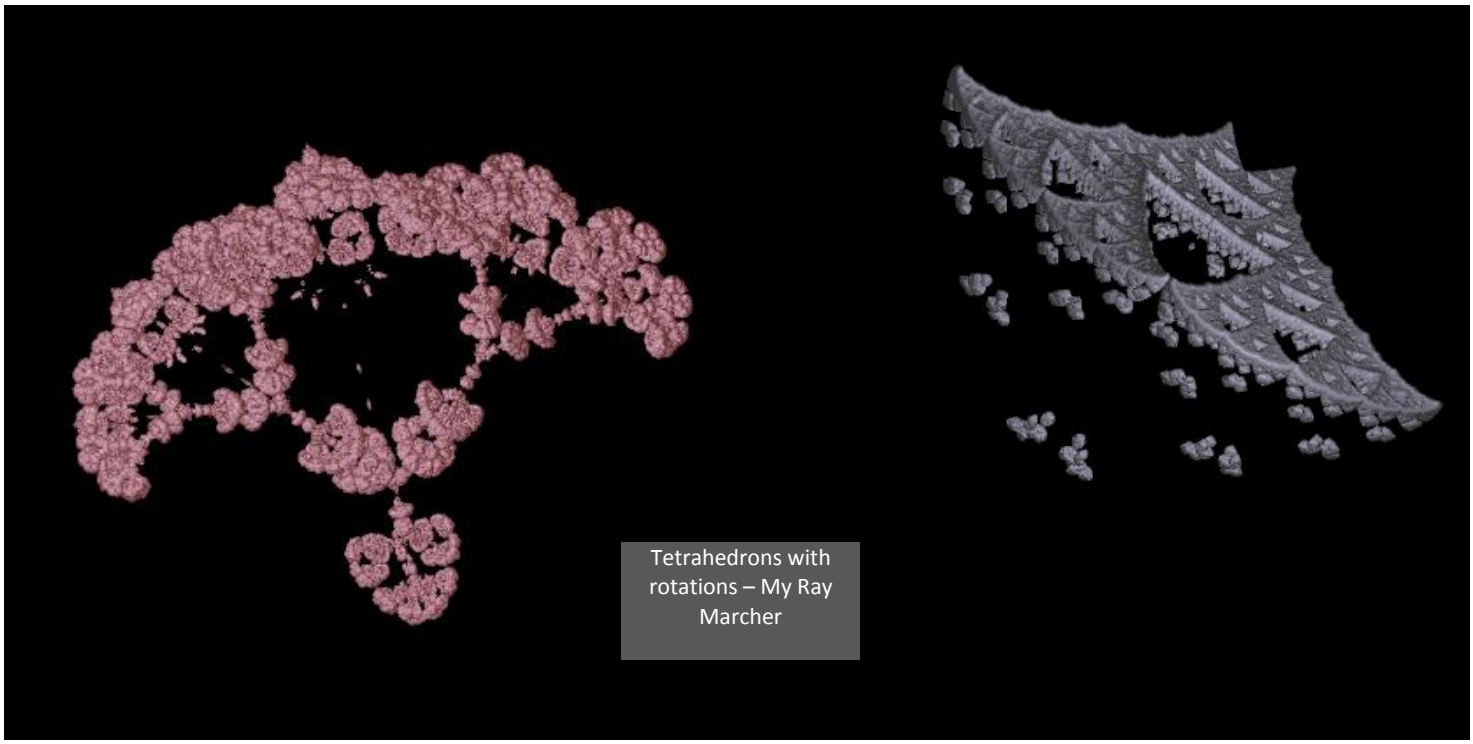


Sierpinski Tetrahedron – My Ray Marcher

On the left is the result of the ray marcher. I have problems drawing it though. I needed to readjust the iterations and the escape epsilon a lot of times until it gave me these result.

However, we can go farther with this kind of fractals. As knighty explains in a post in fractalforums.com, we can get interesting results only by changing the folding operation, by adding rotations or even changing the scaling point. So, I decided to add rotations and the adjustable scale point onto the ray marcher.

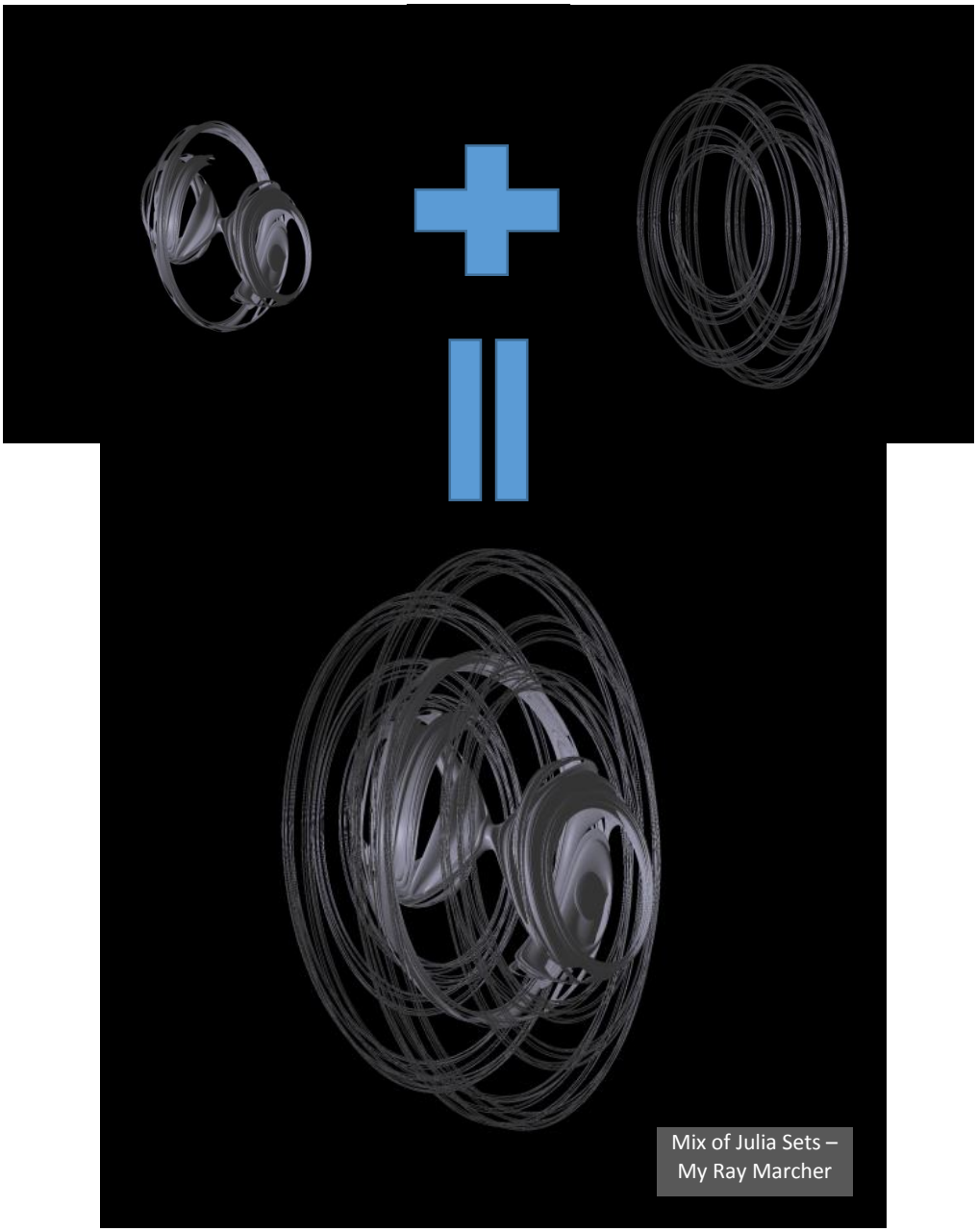
The following are some of the results I get using rotations:



Tetrahedrons with rotations – My Ray Marcher

Mixing Julia Sets

Apart from this, I also make a small experiment of mixing two Julia sets. This is made by taking the minimum distance between two different sets. Although, the results are not that interesting, this is really easy to implement. Here a result:



Mix of Julia Sets – My Ray Marcher

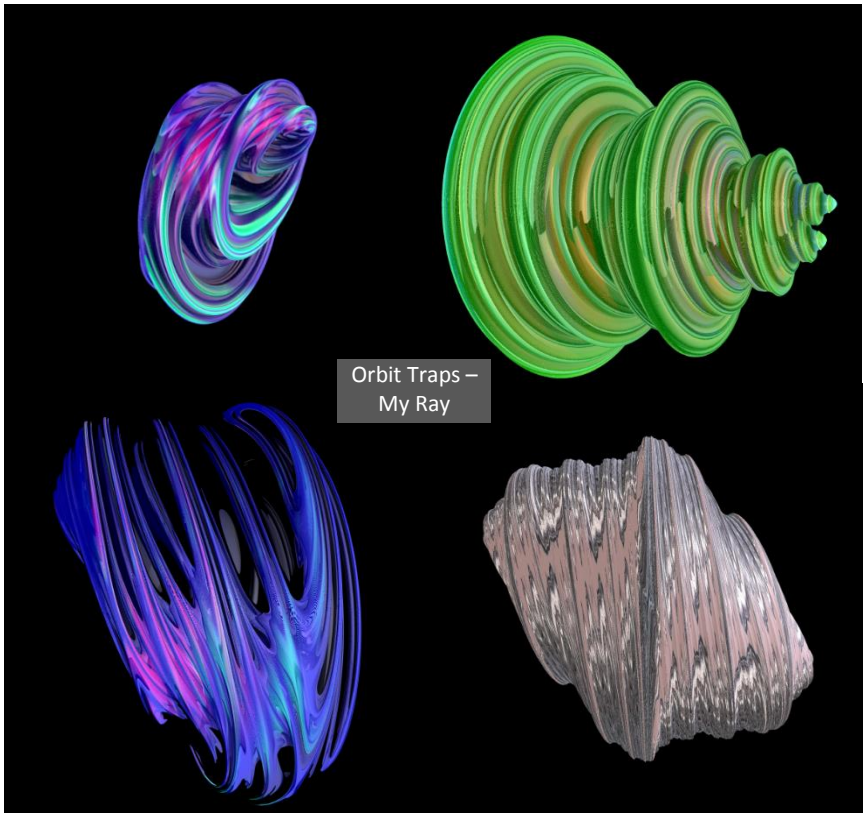
Orbit Traps

The last thing I implemented on the ray marcher are orbit traps. I did not go very in depth on them, though. In the article "Geometric Orbit Traps" by Inigo Quilez it is explain that the orbit traps are just the minimum distance between a geometric structure (usually points, lines or planes) and the points computed on each iteration. For example, we could store in the red component the distance from the point to the X axis, on the green to the Y axis and on the blue to the origin. Or even use them to access a texture. In this way, the fractals will look more interesting. Here is a possible orbit trap pseudo code:

```
float DEJuliaSet(vec3 point, vec4 c)
{
    for (i from 0 to 10)
    {
        //...
        //DE computations
        //...

        if (zLength > 2.0f)
            break;

        OrbitTrap.x = min(OrbitTrap.x, glm::abs(z.x));
        OrbitTrap.y = min(OrbitTrap.y, glm::abs(z.y));
        OrbitTrap.z = min(OrbitTrap.z, zLength / 2.0f);
    }

    return  0.5f * glm::log(zLength) * zLength / dzLength;
}
```

Per each iteration, close the closest distance in each case

And here we have some interesting results:



Orbit Traps – My Ray

HOW TO USE

The program is works in a similar way to the last project. There is a scene file where the camera, lights, surface properties and attenuation can be changed. They are formatted on the following way:

- LIGHT: <position >, <diffuse color>, <radius>
- CAMERA: <position>, <u vector>, <v vector>, <eye length>
- SP: <ambient color>, <diffuse color>, <specular reflection>, <specular exponent>
- ATTENUATION: <attenuation color>

There is also a configuration file, where a lot of things can be changed regarding the ray marcher:

- Scene: Select the scene file to use
- Window: 1 if window activated, 0 if not
- WindowSize: Select the window size (if window activated
- Shadow: 1 if hard shadows activated, 0 if not
- AntialisingSideSamples: The side of the super sampling. For example, if 2, then 2x2 samplings.
- RayMarchingIterations: How much iterations the ray marcher will do on each ray thrown.
- EpsilonOfDistanceEstimator: Epsilon for the ray marcher
- DistanceEstimator: We can select a distance estimator from the followings:
    0. Tetrahedron
    1. Tetrahedron with rotations
    2. Infinite spheres
    3. Julia sets
    4. Mix of Julia sets
- ActivateOT: 1 if render with orbit traps, 0 if not
- OTType: Diferent types of orbit types from 0 to 4. The 4^{th} is the one drawn with textures. The texture can be changed by overwriting 'Texture.png' image.
- RotationBF: Rotation before folding is performed (in Euler angles)
- RotationAF: Rotation after folding is performed (in Euler angles)
- Scalepoint: Point from which to scale tetrahedron
- FractalC: C constant for the fractal <real part>, <imaginary part>
- FractalC2: C constant for the 2^{nd} fractal on the mix <real part>, <imaginary part>

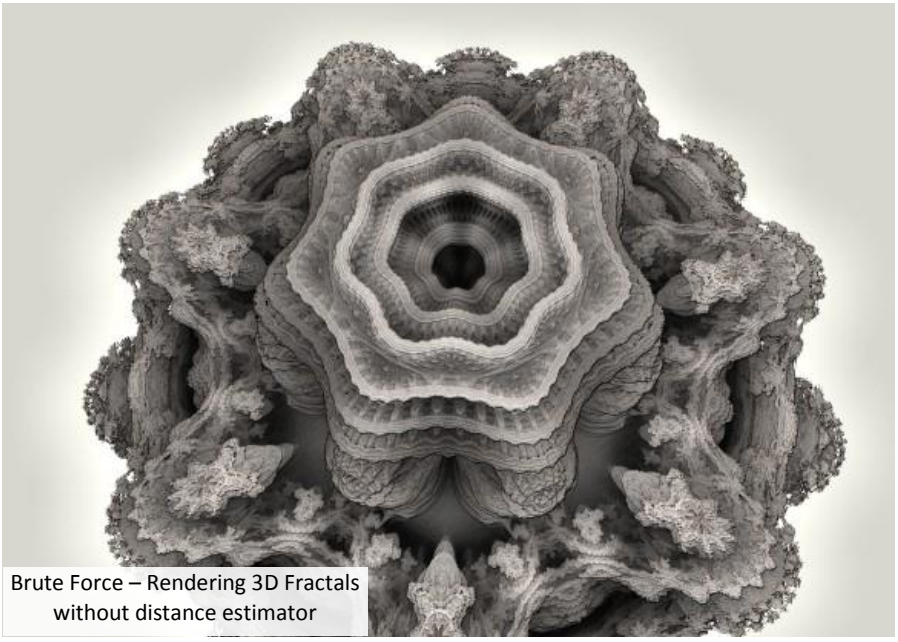
(On the BIN folder I attached some interesting configurations. Copy them to Configuration.txt to use them)

OTHER IMPLEMENTATIONS

Using distance estimators for rendering Julia sets is by far the most popular way of doing it, as it is a fast enough way and gives good results. Moreover, as it uses ray marching, it can be inside of a bigger project done by using this rendering method. However, to use distance estimators, first they need to be computed, which in some cases is not that trivial.

So, in this cases, a brute force approach could be worth it. Referring to the blog post by Mikael Hvidtfeld Christensen 'Rendering 3D Fractals without a distance estimator' there is a way of drawing them without these distance estimators, but for that a way of knowing if the point is inside of the geometry is needed. In this post, Mikael explains an algorithm that consists on taking random points on the given ray direction. If this point is inside of the geometry, then it will only take nearer points, as the nearest point from the ray origin is the one to be obtained.

Of course, this algorithm is much slower, but as said before, it could have its uses with some complex systems. What I do not understand, though, is why to take random points instead of implementing the raw ray marcher. It is not easier if small steps are taken in the given ray direction instead of random numbers? It is true that the result can differ a lot depending in the step size and that if what is wanted is to have a general method, it could be problematic to select the correct size. But, still it gives quite good results.



Brute Force – Rendering 3D Fractals without distance estimator

PROBLEMS ENCOUNTERED

The biggest problem I had on the project was with the math. It is very depth math and at the beginning I did not have any understanding on fractals. That is why making first 2D fractals was essential, as I understood the basics of the topic.

In addition, I found it hard to test. In most cases I was only getting black screens and debugging it was a nightmare. There are many iterations and to go one by one was not an easy tasks.

However, apart from this problems, I did not have big issues to get the final results. There is a lot of information on the internet that helps a lot.

CONCLUSION

In conclusion, the project was an interesting journey. I am glad that I select this topic, as I learned a lot. Using distance estimators for rendering 3D fractals is a very big subject and yet I have a lot to learn, but as the first approach to it, I think that I get interesting results.

Distance estimators are quite fast, easy to use and portable, as they can be exchanged with a different estimators without needing to change the ray marcher. Another good reason of using ray marching is that Phong lighting is still supported, as well as reflections. This comes in handy, as changing from traditional ray tracing to ray marching has been easy. The biggest problem is developing and understanding these estimators, as they can have heavy math. But, they totally worth it.

Finally, I want to mention orbit traps. I think that they are the optimal way of rendering fractals, not only because they look prettier, but because they give more information from the geometry itself, making it easier to understand how they work. They also are very easy to implement.

BIBLIOGRAPHY

Hart, J. C., Sandin, D. J. and Kauffman, L. H. (1989). Ray Tracing Deterministic 3-D Fractals. *Computer Graphics, 23*(3), 289-296.

Dang, Y., Kauffman, L. H., Sandin, D. (2002). *Hypercomplex Iterations, Distance Estimation and Higher Dimensional Fractals.* Singapore: World Scientific Publishing Co Pte Ltd.

Christensen, M. H. (2012) *Distance Estimation*. Retrieved March 21, 2020, from http://blog.hvidtfeldts.net/index.php/category/distance-estimation/

Quilez, I. (2002) *Animated orbit traps.* Retrieved April 7, 2020, from https://www.iquilezles.org/www/articles/ftrapsgeometric/ftrapsgeometric.htm

Quilez, I. (2013) *Julia - Quaternion.* Retrieved March 30, 2020, https://www.shadertoy.com/view/MsfGRr

CodeParade (2018) *How to Make 3D Fractals.* Retrieved March 29, 2020, https://www.youtube.com/watch?v=svLzmFuSBhk&t=316s

The Art of Code (2018) *The Mandelbrot Fractal Explained!.* Retrieved March 28, 2020, https://www.youtube.com/watch?v=6IWXkV82oyY&t=9s