Raytracing 4D fractals, visualizing the four dimensional properties of the Julia set

Torkel Ödegaard
Institute of Computer Science
Mälardalen University
Vitmårargatan 5c
722 26 Västerås, Sweden
tod00001@student.mdh.se

Joakim Wennergren
Institute of Computer Science
Mälardalen University
Odensvigatan 7
723 42 Västerås, Sweden
jwn98021@student.mdh.se

ABSTRACT

By using rules defined by quaternion algebra the Julia set can be extended to four dimensions. Techniques for visualizing the four dimensional Julia set as a three dimensional object has previously been explored, however those techniques usually ignores the fourth dimension. In this paper we describe our attempt to extend the already established technique of ray tracing Julia sets to fully incorporate its four dimensional properties. We also discuss an optimization algorithm that drastically increases the amount of details in images and shortens rendering time.

We have tried to describe the methods we use in detail in order to help people interested in fractals or computer graphics to reproduce our results.

1. INTRODUCTION

The world we live in has only three space dimensions. But what if there were more? How would the world be like if we had four dimensions? Our minds are not well equipped to answer such questions, we are used to live in a three dimensional room. We can imagine how a house might look on the other side when we approach it, but how would a simple object like a cube look like if it existed in four dimensions? Many mathematicians and physicists are used to think in this abstract way, and many have tried to visualize it.

With the introduction of fractal geometry, that appeared in the 1980s, mathematics has presented some very interesting but challenging objects to computer graphics. The word "fractal" was created by B. Mandelbrot to describe a family of shapes he discovered when he was studying iterative functions at IBM. The fractal shapes are very special in that their surfaces and borders are not smooth like Euclidean spheres or cones. A fractal can be defined as a geometric pattern that is repeated at every scale and so cannot be represented by classical geometry. Mandelbrot investigated many kinds of fractals, both created by random functions and others created deterministically by iterating the same function an infinite number of times (in theory).

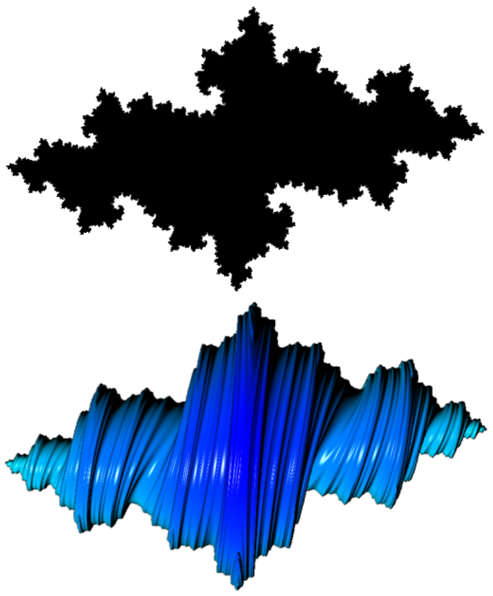

Figure 1: A standard 2D Julia with the corresponding quaternion Julia below ($\mu = -0.7 - 0.2i$).

In this paper we discuss how to visualize a non random fractal called the quaternion Julia set. The Julia set can be described as the set of numbers that never increases to infinity when iterated over the function:

$$f(z) = z^2 + \mu \qquad (1)$$

Where the variables z and μ are usually complex numbers, but the formula can also be applied to quaternions. By choosing different value of the constant μ, different sets will be generated. The most common Julia set, the set over complex numbers, can be plotted on a 2D diagram setting the real part along the x axis and the imaginary part along the y axis. The shape that appears is an image that is well known (Figure 1), mainly from the work by B. Mandelbrot [1]. By using quaternions instead of complex numbers, you get a set of numbers that can be plotted in four dimensions. A. Norton was the first to implement this [3], he created images by "dropping" one dimension, and only using the three dimensions that were left. He created a method to display this set by calculating a number of points on the surface of the set, and then displaying them on a screen. Finding this surface is rather computer intensive though, and a method to only find the parts of the surface that will be visible on the screen was developed by Hart, Sandin and Kaufmann [5].

They adopted the already known method of ray tracing to render the Julia set. By shooting rays at the set from a given "virtual camera" they managed to create images much faster than Norton. However, they too dropped one dimension and used 3D rendering techniques. We have tried to create a virtual camera that exists totally in 4D-space, so it can be freely moved and rotated in any direction in four dimensions. This way we hope to capture more of the Julia set's four dimensional structure.

The scientific value of these images can be questioned, but they are fascinating to watch. However the development of visualization techniques has greatly aided the research on the dynamics of iterative functions.

This paper will start with describing the basis for creating the fractals, namely the quaternions. We will then describe how normal ray tracing of the Julia is done, and then discuss how we extended this into four dimensions. We finish with the optimization technique we developed and some results that show how it improved quality and rendering time.

1.1 Related Work

One of the latest published papers on this subject is Terry W. Gintz paper [2], which describes different types of four dimensional fractal shapes. He has developed a commercial program that is capable of ray tracing these shapes as well as animate them.

There is a free 3D fractal rendering program that is called Quat[1], developed by Dirk Meyer. Quat is program that ray traces quaternion Julia sets in a similar method that we describe in this paper, however with only a 3D camera and no animation capabilities.

2. QUATERNIONS

Quaternions, as first described by Hamilton in [7], is an extension of the two-tuple complex numbers into four-tuple. They have the form:

$$q = q_1 + q_2 i + q_3 j + q_4 k \tag{2}$$

Where i,j and k are all imaginary units, with the special properties

$$\begin{aligned} i^2 &= j^2 = k^2 = -1 \\ ij &= -ji = k \\ ki &= -ik = j \\ jk &= -kj = i \end{aligned} \tag{3}$$

[1]The program can be downloaded from: `http://www.physcip.uni-stuttgart.de/phy11733/index_e.html`

Important to note here is that quaternion multiplication is not commutative, thus it matter in which order you perform the multiplication. Given these set of multiplication rules, the Julia formula $q' = q^2 + \mu$ can be calculated in the following way.

$$\begin{aligned} q'_1 &= q_1^2 - q_2^2 - q_3^2 - q_4^2 + \mu_1 \\ q'_2 &= 2q_1q_2 + \mu_2 \\ q'_3 &= 2q_1q_3 + \mu_3 \\ q'_4 &= 2q_1q_4 + \mu_4 \end{aligned} \tag{4}$$

2.1 Julia quaternions

The rendered images that are discussed in this paper are derived from the function

$$z_{n+1} = f(z_n) = z_n^2 + \mu$$

where z is the iterated variable and μ the constant. The result of n iterations is denoted as $f^n(z)$ and it should not be confused with raising the result of $f(z)$ to the nth power. The resulting value of $f^n(z)$ determines if the initial point z_0 is a apart of the Julia set J:

Definition 2.1.

$$J = \{z : \lim_{n\to\infty} f^n(z) \nrightarrow \infty\} \tag{5}$$

By choosing different constants μ different Julia sets are created. We will from here on refer to n as the *max iteration depth* of the Julia set.

3. RAY TRACING

Ray tracing is a very realistic method of rendering geometric shapes. Graphical effects like shadows, reflection and refraction can easily be implemented. However the technique is very slow and near impossible to perform in realtime with today's computers, at least with a desired degree of quality.

Given a camera position you can calculate a virtual projection plane through which you cast rays. In normal ray tracing with Euclidian shapes you solve an equation for each ray and object. What you get after solving the equation is the intersection point where a ray enters the surface of the object. At this intersection point you can calculate a surface normal which is used for lighting and reflection. More detail on normal ray tracing can be found in [8].

When ray tracing a Julia fractal there is no finite method that can determine the exact surface-point where an arbitrary ray intersects the set. The reason for this is because the surface of the Julia set is fractal, meaning you can always find finer details by examining closer. Since we only iterate a finite number of times, we only create an approximation of the surface.

To find the intersection point one has to *scan* the $4D$ space testing if each point along the ray is apart of the Julia set. This process is extremely time consuming since each point's classification is based on a large number of function iterations. The scanning can be done at different resolutions (from here on called *z-resolution*), the higher resolution the closer to the actual surface one will find, but it will also take more computing time.

The surface of the Julia set is very different depending on the constant μ and the max iteration depth. The higher number of iterations we perform the closer to the actual set we will get, and the more detail the surface will have (see Figure 8).

3.1 The Depth Buffer

After an intersection point has been found the distance from the camera to the point is stored in a depth buffer, also known as a *Z-Buffer*. This is not a necessary step when ray tracing Euclidian shapes, however we need the Z-Buffer to approximate surface normals(see section Shading).

4. THE 4D CAMERA

To be able to render the Julia set from different angles and positions we need a defined way of determining directions of the rays. The general solution for this is to create a virtual camera. Our virtual camera is described as two four dimensional vectors, one describing the point of origin, and one describing the facing direction. We will refer to the forth element in these vectors as the w element.

When creating rays to cast from the camera a projection plane representing the image to be produced is placed in front of the camera. The distance from the camera to the projection plane determines how wide the field of view will be. Placing the plane near the camera will result in a "fish-eye" effect, placing it very far away will remove all sense of perspective and the image will appear flat and without depth. It is important to note that the plane itself plays no other role than to calculate the directions of the rays. Wether the plane is in front or behind or even intersecting the objects that are to be rendered does not matter.

Normal methods that are used in 3D ray tracing, which handle 3D vectors, can not be applied directly when handling 4D vectors. In 3D ray tracing an "up-vector" is often chosen to make all images appear oriented in a natural way. This vector is used to orient the plane so the upper border of the image is always facing "upwards". You then perform a cross product between the up-vector and the camera direction to calculate the equation of the plane. The cross product between two vectors result in a new vector perpendicular to both start vectors, however we cannot use the standard 3D cross product. Since we do not want to "drop" a dimension, we had to define a new cross product. Our "4D-cross product" need three vectors, and return a fourth that is perpendicular to all three. Trying to imagine four vectors, all perpendicular to each other is not easy, but mathematically it is possible:

$$4Dcross : \mathbb{R}^4 \times \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R}^4$$

Where $q = 4Dcross(a, b, c)$ gives:

$$\begin{aligned}
q_1 &= a_2(b_4c_3 - b_3c_4) + b_2(a_3c_4 - a_4c_3) + c_2(a_4b_3 - a_3b_4) \\
q_2 &= a_1(b_3c_4 - b_4c_3) + b_1(a_4c_3 - a_3c_4) + c_1(a_3b_4 - a_4b_3) \\
q_3 &= a_1(b_4c_2 - b_2c_4) + b_1(a_2c_4 - a_4c_2) + c_1(a_4b_2 - a_2b_4) \\
q_4 &= a_1(b_2c_3 - b_3c_2) + b_1(a_3c_2 - a_2c_3) + c_1(a_2b_3 - a_3b_2)
\end{aligned}$$

It can be proved that the above function is correct by calculating the scalar product between the new vector and each of a, b, c, and the result will always be zero, thus the new vector is perpendicular to all of them.

To use this $4Dcross$ function we need one more vector, not only the camera direction and the "up-vector". This third vector will be a vector to which the plane of the image will be perpendicular to, a *limbo* vector. If this *limbo* vector is set to $(0, 0, 0, 1)$ and the w element in the camera direction is zero we effectively "drop" the fourth dimension and we will see the same 3D Julia set as Norton.

By choosing another *limbo* direction, or point the camera in a direction with a $w \neq 0$, the rays will travel in the fourth dimension as we traverse them. This means that the further away from the camera the Julia set is hit, the more displaced it will be in the fourth dimension. One analogy would be if we saw the fourth dimension as time, and the Julia set as changing with time. Then the surface near the camera would have had less time to change than surfaces further away, as if the rays of light were travelling very slowly.

The rays cast by the camera in normal ray tracing are "real" rays, that is they have one starting point (the camera) and then extends to infinity in some direction. Since we test our rays by stepping along them we can not let them go on into infinity since that would take infinite time. What we do is have a maximum distance we travel. This means in effect that we have a *farplane* beyond which we can not see. This distance must be chosen so that no ray stops inside the Julia set, since this would make the Julia seem cut off. Since the Julia usually do not start immediately at the camera, we have also implemented a *nearplane* so that we start our testing a bit in front of the camera. By choosing good *near-* and *farplane* we make sure only to test points that have a chance of being in the Julia set.

One way to choose these planes is to first render a low-resolution image of the set and record where the closest and furthest hits were made. We can then set the *near-* and *farplane* to these values, with a small extra margin for error.

5. SHADING

Shading is an extremely important process in creating a realistic and aestheticaly pleasing image. When shading an image each pixel is lit depending on its surface normal. The angle between the direction of the light and the surface normal determines the pixel's brightness b:

$$b = normal \cdot light_{direction} \tag{6}$$

Where $\cdot$ is the standard scalar product. The color of the surface is then multiplied with the brightness factor b. To perform this calculation we need surface normals for each pixel, and the direction of the light.

There are more realistic methods than that shown in eq.(6), for example algorithms where the distance to the light source also effects the brightness.

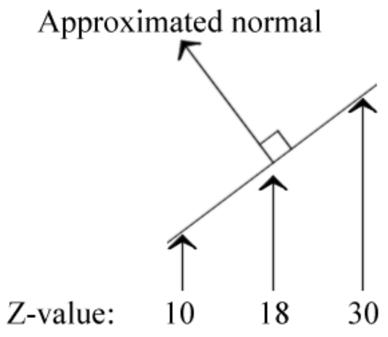

Figure 2: The normals are calculated by looking at the heights of the neighbouring pixels in the Z-Buffer.

5.1 Normal calculation

When ray tracing Euclidean shapes a surface normal can be exactly calculated to a desired degree of accuracy. The slope at any point on the surface of the Julia set is undefined, closer examination of a points surroundings, for example scanned at higher iteration and z-resolution, would reveal a different environment. Therefore the normal can not be exactly calculated. However there are two methods of calculating an approximation of the surface normal. These two methods are discussed in [5], one using a Z-Buffer and one calculating a gradient. We have chosen to use the Z-Buffer approach.

5.1.1 *Using the Z-Buffer neighbors*

Given the Z-Buffer values, that were calculated in ray traversal process, an approximation of a point's normal can be calculated out of the height of its neighbors (see Figure 2). Our method, that is shown below, uses the the mean difference in hight of a point's horizontal and vertical neighbors.

$$N_x = (Z_{x-1,y} - Z_{x+1,y})/2 \tag{7}$$

$$N_y = (Z_{x,y-1} - Z_{x,y+1})/2 \tag{8}$$

$$N_z = \epsilon \tag{9}$$

Where ϵ is the mean distance between two pixels in the Z-Buffer. We have tested several methods for creating normals, using larger sets of surrounding pixels to get a better approximation. We tried using pixels only left and below, using neighbors in all four directions, using all eight surrounding pixels, using two pixels in each direction and a combination of the last two mentioned. We also tried giving the surrounding pixels different weight in the calculation. Using large sets of pixels might result in a better approximation of the normal, but only assuming all surrounding pixels are part of the same, continuous surface. If they instead are on a different level, either outside the Julia set, or on a lower "shelf" of the set, the normal will lean towards the edge. Therefore, fractals with many ledges and free floating parts (typically fractals created with a high iteration depth) will not benefit from increasing the number of surrounding sampling pixels.

The method discussed in [5] only uses the point to the right and below for the normal calculation.

The problem with these methods is that they require the Z-Buffer to be stored during all rendering steps. In traditional ray tracing the normal can be calculate directly for each point and does therefore not require a Z-Buffer. The gradient method discussed in [5] does not require a Z-Buffer and is therefore a more memory efficient method, but somewhat slower.

5.2 Specular highlight

Specular highlight is a popular technique in ray tracing. It is used to make objects look shiny. Unlike diffuse illumination, where light scatters equally when it hits a surface, specular illumination goes in a particular direction. When a light source reflects of a shiny surface it creates a highlight on the object where the angle between the reflected light ray and the view vector is small. More information about this technique can be found in [6].

6. ANIMATION

The ray traced images that are produced by the method we have described can be extraordinary in their own right, however when animating the Julia set over different values of the constant μ truly fascinating scenes can be produced. For example seeing a sphere being morphed and twisted into a convoluted Julia fractal. Moving the camera position during the animation is also interesting. If we are only moving in the first three dimensions, we get a "flyby" of the Julia set, and can see it from different angles. By moving the camera's position and target vectors along their w axis, we can render the Julia at different four dimensional angles. With such camera animations we hope to capture some of the properties that are normally not seen.

7. OPTIMIZING

The Z-Buffer generation is extremely time consuming due to the high number of points that needs to be tested. Having a high z-resolution is important for capturing all surface properties, but a high z-resolution means you will test a high number of points. Trying to limit this number can reduce the rendering time considerably. The goal is to reduce the number of point's that are tested outside the surface.

Bounding volumes are a way of speeding up this process, limiting the amount of point's that needs to be tested. A bounding volume, usually a sphere or a ellipsoid, encase the object such that if a ray does not intersect with the bounding volume it will not intersect the object inside it. However this technique has not proven successful when applied to deterministic fractals such as the Julia set.
Unbounding volumes, which are described by Hart, Sandin and Kauffman in [5], are defined as volumes that do *not* contain any part of the object. Using an approximative distance function to create unbounding volumes the ray traversal process can be speeded up considerably. Hart, Sandin and Kauffman used the following distance approximation:

$$d(z) = \frac{|f^n(z)|}{|f'^n(z)|} \log f^n(z) \tag{10}$$

When trying to find how far away the surface is from a point in space you calculate an approximate of the distance

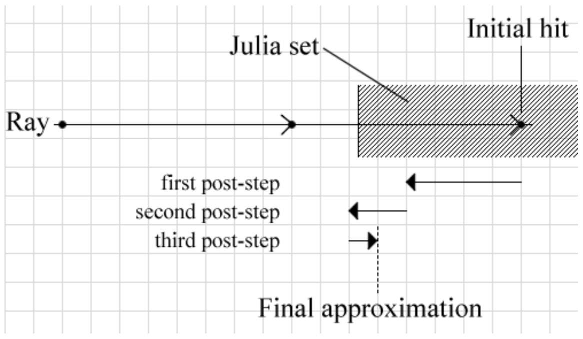

Figure 3: Post-stepping process after three iterations, resulting in a better approximation of the surface than the initial hit.

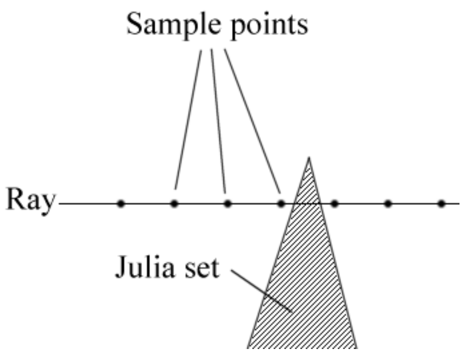

Figure 4: The Julia set is overstepped when traversing the light ray.

with eq.(10), which always gives a lower value than the actual distance. Then you step this distance forward, and try again until the approximate distance is below some chosen boundary value.

7.1 Post-stepping

To speed up the ray traversal process we developed a method that we call post-stepping. The algorithm is very simple but can be quite effective depending on a few parameters.

The ray is traversed as usual from the eye into the scene in incremental steps. The length of this step depends on the currently chosen z-resolution. When the ray has reached a point that lies within the Julia set (a hit) the post-stepping process starts. The post-stepping is a iterative process that finds a more accurate position of the surface. In each step we move half the previous distance, either forward or backward depending on wether we are still in the set or not (see Figure 3). This can be stated as an inductive function, as defined below.

The distance to jump in step n is defined as:

$$d_n = d_{n-1}/2$$

The position p of the approximated surface after n iterations:

$$p_n = \begin{cases} p_{n-1} \in J: & p_{n-1} - d_n \\ p_{n-1} \notin J: & p_{n-1} + d_n \end{cases} \tag{11}$$

Where d and p are four dimensional vectors and J is the Julia set.

The post-stepping process will increase the accuracy of the surface point approximation. This will result in more smooth and detailed images, even at low z-resolutions. The effective z-resolution is doubled in each post-step iteration. When using a post-step iteration depth of n the comparable z-resolution is:

$$Resolution'_z = Resolution_z * 2^n \tag{12}$$

However the method has some drawbacks. When the method is used with a low z-resolution the high step-distance may cause the surface to be overstepped and thereby missed. A ray with a high jump distance could easily overstep edges and thin strands of the set, which results in jagged edges and missing surface areas (see Figure 4). This is especially noticeable when used with a high max iteration depth, where the resulting Julia set is very convoluted.

To overcome this it is imperative to choose the right z-resolution for the specific Julia your are rendering. A thumb rule one can derive from the nature of the Julia set is that with a higher iteration depth a higher z-resolution is needed.

8. ADDITIONAL QUATERNION CLASSES

By changing the algebraic rules of the quaternion other classes of four-tuple numbers emerge. As seen in [2] these will generate very different shapes. We experimented with several different sets of rules for the imaginary parts in the quaternions. The shapes that we saw after these changes were very different from the standard quaternion Julia set.

One of these sets of rules is the *hypercomplex numbers*, as described by W. Gintz in [2]. It uses a set of commutative rules:

$$\begin{aligned} i^2 &= j^2 = -1 \\ k^2 &= 1 \\ ij &= ji = k \\ ki &= ik = -j \\ jk &= kj = -i \end{aligned} \tag{13}$$

The resulting Julia sets under these rules tent to appear "square" and not as rounded as the quaternion Julias. Another sets of rules also presented by Gintz is something he calls *Complexified Quaternions (CQuats)*. The CQuats uses the following rules:

$$\begin{aligned} i^2 &= -1 \\ k^2 &= j^2 = 1 \\ ij &= ji = -k \\ ki &= ik = -j \\ jk &= kj = i \end{aligned} \tag{14}$$

The CQuat Julia sets are also with square angles and has often square holes. Images of Julia sets using the sets of rules above can be seen in Gintz paper [2].

We initially tried to make a commutative version of the the quaternion rules ourselves, and used the naive approach:

$$\begin{aligned} i^2 &= j^2 = k^2 = -1 \\ ij &= ji = k \\ ki &= ik = j \\ jk &= kj = i \end{aligned} \quad (15)$$

This set of rules created highly irregular shapes, as can be seen in Figure 7.

9. RESULTS

We rendered a large number of images in an attempt to evaluate our post-stepping algorithm. The images were rendered on a 1.6GHz Pentium 4 PC with 512Mb RAM running Windows XP.

As can be seen in Figure 5 the post-stepping procedure increases the detail considerably. Both images were rendered with a z-resolution of 250. The left image without post-stepping has defects caused by inaccurate normals, due to too large steps in the ray traversal. In the right image all such artifacts were removed, and this with only ten post-steps. The time to render the images were 17.9s and 19.1s respectively, so with little time increase we get much more detail. The post-stepped image gets a practical z-resolution of $250 * 2^{10} = 256000$, meaning that to get an image with the same detail level without post-stepping the z-resolution had to be a thousand times higher!

When trying to capture the four dimensional properties of the quaternion Julia we rendered images from different points in 4D-space. In Figure 6 we can see the same Julia set from different 4D-positions. As can bee seen the set seems to grow as we approach the $w = 0$ position.

Something similar can be seen when animating the camera's target vector. When we interpolated the point we look at between $w = -1.5$ and $w = 1.5$ the Julia set was seen morphing into existence and then morphing out. You can think of this as if the camera is first directed to a point beside the Julia set, then moves over the set, ending on the other side. Since the camera's motion is purely in the fourth dimension the Julia set seems to appear from nowhere, growing to a maximum and then shrinking into oblivion again.

10. CONCLUSION AND SUMMARY

We have shown how a four dimensional quaternion Julia set can be rendered using the already established ray tracing method. Our 4D camera system allowed us to visualize the Julia set from different four dimensional angles, which we have not seen done or discussed before. We have also described a new optimization technique, that we call post-stepping, that can either be used to enhance surface detection or speed up rendering time.

The reason we call post-stepping an optimization technique is that it can be used to shorten rendering time. When post-stepping is enabled, an decent looking image can be generated with a low z-resolution. If the same image is to be rendered without post-stepping a much higher z-resolution would be needed, which would require a lot more time.

The problem the post-stepping algorithm faces is that it does not help the initial surface detection. When rendering hollow Julia sets or sets with high iteration depths a high z-resolution is still needed so the closest surface is not overstepped.

The four dimensional properties of the Julia set are present in every picture of the quaternion Julia set, as a projection. Previously, however, all rendered images of the Julia set has been from three dimensional angles. When we animated our camera to move in four dimensions we witnessed some interesting behavior in the set. It is very hard to describe these animations, they are in a way similar to the animations where the constant μ is changed, in that it is morphing motion from one shape to another. However all that is changed during our animations are the camera's target and position vectors.

We believe that animations greatly help in the understanding of four dimensional perception. When rendering movies instead of images, we can see the Julia set as it changes when looked upon from different angles, and by studying the geometric changes in these animations it is easier to pinpoint some of the set's four dimensional properties.

10.1 Future Work

There are further ways of optimizing the process we have described. When we have time we are going to try to combine our post-stepping algorithm with the unbounding volume method shown in [5]. That should speed up the ray traversal process even more. As in general ray tracing our method can be easily parallelized. The program we developed has some support to distribute the work over network clients by using a work-pool, but this is not completed yet because of time constraints.

11. REFERENCES

[1] B. B. Mandelbrot, *The Fractal Geometry of Nature*, W. H. Freeman Press, 1982.

[2] Terry W. Gintz, *Artists statement CQUATS–a non-distributive quad algebra for 3D renderings of Mandelbrot and Julia sets*, Computers & Graphics, Vol. 26, pp. 367–370, 2002.

[3] Alan Norton, *Generation and Display o f Geometric Fractals in 3-D*, Computers & Graphics, Vol. 16(nr 3), pp. 61–67, 1982.

[4] Yan Ke, *A Jurney into the fourth dimension*, IEEE Visualization, pp. 219–229, 1990.

[5] John C. Hart, Daniel J. Sandin, Louis H. Kauffman, *Ray Tracing Deterministic 3-D Fractals*, Computer Graphics, Vol 23(number 3), pp. 289–296, 1989.

[6] Bishop, Gary, Weimer, D.M. *Fast Phong Shading*, Siggraph, vol 20(number 4), pp. 103–106, 1989.

[7] William R Hamilton, *Elements of Quaternions, 3rd Edition*, Vol 1-2, Chelsea Publishing Company, New York, 1969.

[8] Yagel R., Cohen, D., Kaufman, A.,*Discrete ray tracing*, Computer Graphics and Applications, IEEE , Volume: 12 Issue: 5, pp. 19–28, 1992.

APPENDIX

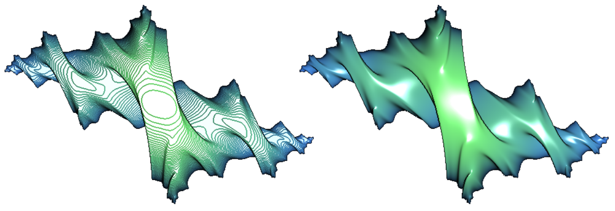

Figure 5: Julia fractal rendered with the constant $\mu = -0.803762 - 0.40615i$, a max iteration depth of 8 and a z-resolution of 250. The left image was rendered without post-stepping while the right used 10 post-steps.

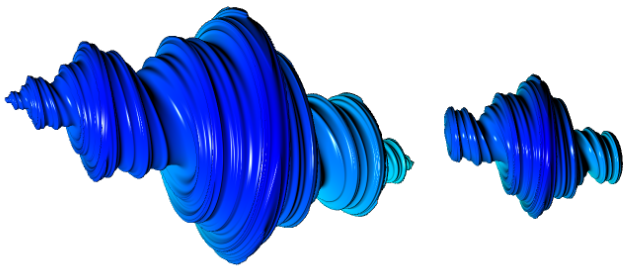

Figure 6: The julia set with ($\mu = -0.7323 - 0.2179i$) where the camera is in different w positions. The image to the left has the camera is at $(-2, 1, 5, 0)$ while the right image is shot from $(-2, 1, 5, 8)$. The right image is noticeably smaller, since the camera is further away in the fourth dimension.

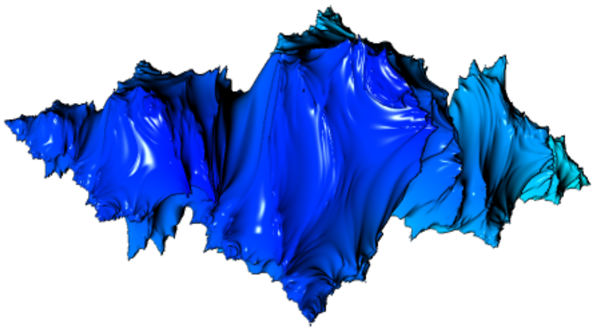

Figure 7: A Julia set created with the commutative rules shown in eq.(15). $(\mu = -0.7323 - 0.2179i + 0.1j)$

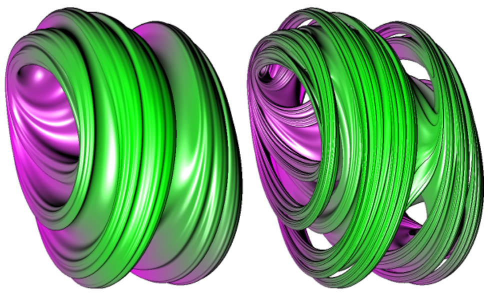

Figure 8: Two images showing the same Julia set but rendered with different iteration depth. The left stops at 8 and the right goes to 12 steps, giving the right image higher detail. $(\mu = 0.4 + 0.5i)$

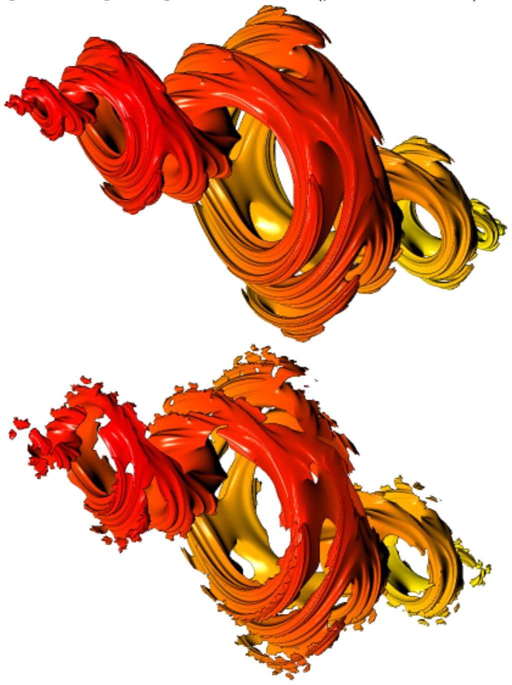

Figure 9: Two images showing how a low z-resolution can cause rays to over-step the Julia set. In the upmost image the z-resolution is set to 2000 and in the lower to 35. Note that because of the post-stepping the surface itself is smooth in both images. $(\mu = -0.8 + 0.4i)$