Rendering Fractals in 3D Geometry

Innovations Project Report

Jonathan Peel

Rendering Fractals in 3D Geometry

Abstract

An investigation into whether the volume of a piece of 3D geometry can be represented with the use of fractal patterns or 'fractal flames' with an end objective completing an offline (not real time) standalone renderer capable of performing this task. The final product will make use of information acquired through detailed research into how fractal patterns can be generated in 3D and how geometric volumes can be represented within a computer.

Contents

1. Introduction

Over recent years, there has been a large amount of interest in producing 'fractal art' – "[Art which] is created by calculating fractal objects and representing the calculation results as still images, animations, music, or other media" (Fractal Art - Wikipedia, 2007). As computing power has progressed, so has the amount of experimentation into these fractal objects. Faster calculations have lead to far more complex and intricate designs and simplified software. Software such as Apophysis (2007) has opened up this relatively new medium to many potential artists who lacked the previously necessary programming and mathematical skills necessary in producing this art form.

I became fascinated with fractal art fairly recently after discovering several images that made use of Scott Drave's 'Fractal Flame' algorithm (Draves, 2003). Soon after, I delved a little further into what the possibilities of fractals are and what people had managed to produce with them, stumbling upon a large collection of deviantART (2007) submissions, many of which making full use of the abilities of Apophysis.

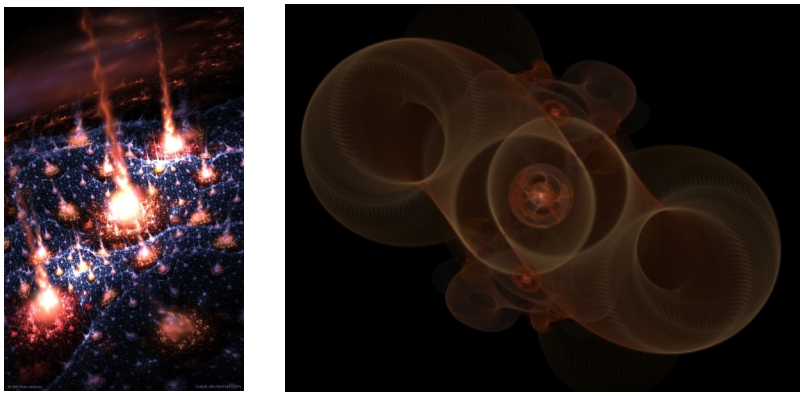


*Fig 1. Two images rendered in Apophysis. 'Cataclysm' by *Zueuk and 'Power Core' by EminentEnd, both submissions on deviantART (2007)*

However, as beautiful as these images are, they, within fractal art at least, all seemed to share a common theme of using the final constructed fractal object as the aesthetic piece of art. Fractal objects are not really used as a display tool, but as an entity on its own which is to be admired. That said, many tools, outside the context of fractal art, make good use of the algorithms. Fractals are very closely related to proceduralism (Ebert, 2003, pg 436) through which procedural modelling, texturing and shading of 3D geometry is possible. For example, the modelling of clouds or a mountainous terrain in a 3D system could be done with the use of Fractional Brownian Motion (fBm), both illustrated in Procedural Texturing and Modeling (Ebert, 2003). These fractional procedures though, are often less concerned with the aesthetic beauty of the fractal algorithms they use but more with a specific task at hand. What I wanted to investigate therefore, was whether it would be possible to take the procedures behind the beautiful fractal objects seen in fractal art and utilise them to display something much more constrained; whether they could be used to represent the volume of an object in 3D whilst still retaining their unique and pleasing aesthetic.

With this aim, I went about researching for and piecing together the final product of this project; a piece of standalone software capable of performing off line (not real time) rendering of fractal patterns inside imported geometry dependent on a set of given variables.

The software and the creation process behind it is best understood by breaking this project down into the different obstacles I had to overcome in the order I encountered them:

- Storing and displaying volumes
- The creation of fractal patterns
- Defining geometry, 'inside' and 'outside'
- Functionality and usability

As will be shown, whilst rendering fractals in geometry was paramount to the project, other areas of research also became relevant and challenging.

2. Storing and Displaying Volumes

Since what I had set out to do was in essence a volumetric renderer, I required a method of being able to display the inner volume of a piece of geometry. One of the initial pieces of advice I received when starting this project was to look into the use of voxels.

A voxel is essentially a three dimensional pixel (volumetric pixel); a small cuboid in space of predefined width, height and depth which can be assigned a colour value or just an identity of being 'inside' or 'outside' a geometric shape. These voxels are collectively stored inside a voxel map – a volumetric pixel map of a 3D environment.

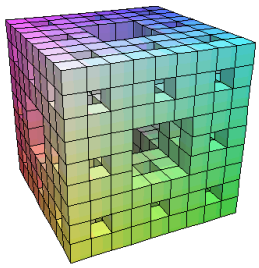


Fig 2. Voxels are simply a collection of small cubes in space. (Piekniewski)

Voxels are put to good use in several real time games (for example, Sid Meier's Alpha Centauri, Crysis and Command & Conquer: Red Alert 2 (Voxel – Wikipedia, 2008)) for a variety of purposes and also in dynamic systems such as Maya's fluids (Maya Help, 2008). It's unfortunate that at present, much of their capabilities are over-shadowed by polygons given that most, if not all hardware acceleration is oriented toward polygon engines (Silverman, 2000). Although the marching cubes (Lorenson and Cline, 1987) algorithm does offer a method of converting voxels back into polygon data but this can be a slow and meaningless process should you not require polygon information for any reason other than display.

Voxels offer a very dynamic method of storing volumetric data, meaning it's very easy to alter the structure of an object at any time. Ken Silverman, author of the Voxlap engine gave this example in an interview:

[With voxel maps,] unlike polygons, it's very easy to modify the map, for example, you can shoot a rocket at a wall, and you can explode a sphere (or any shape) out of it and have it part of the permanent board map.

Ken Silverman interview by 'Cyborg' (2000)

The Voxlap engine itself is an example of a fast real-time method of rendering a voxel map and is available from Ken's website (Silverman, 2000).

With regards to this project, a voxel map could be useful. If I could create voxels inside a given piece of geometry, I would then, in theory, be able to colour any point in that volume as I wanted it, creating patterns inside it – exactly the final result I was after.

However in practice, it is not this simple. In order to define a voxel map of the inside of a geometric volume, I would need to know exactly where in the inside of a shape starts and where it ends. Also, it would lead to complications in working out where a stored voxel was in relation to others as there would be no uniform number of voxels per row or column inside an irregular piece of geometry.

Instead then, I chose to create a voxel map inside a shapes bounding box (the box specified by taking the maximum and minimum points of a mesh's geometry). In this uniform space I generated voxel map of a given resolution which encompassed the given geometry. In this space, it was far easier to move between adjacent voxels in all directions, making plotting a pattern in space far easier. Any point specified in Euclidian space could be transferred easily to voxel space (integer values of width, height and depth in the voxel map of the voxel which contains the Euclidian point) through simple multiplication:

*VoxelIndex.w = (EucPoint.x / VS.x) * VR.x*
*VoxelIndex.h = (EucPoint.y / VS.y) * VR.y*
*VoxelIndex.d = (EucPoint.z / VS.z) * VR.z*

Where:

- The voxel map indexing system starts from the front bottom left
- *VS* is the size of the voxel map in Euclidean space (the size of the bounding box around the geometry) in *x*, *y* and *z*
- *VR* is the resolution of the voxel map in *x*, *y* and *z*

Of course, this solution didn't have any concept of whether a voxel was actually inside the geometry it was given or not; a problem that is addressed later in this report.

One of the problems of using a voxel map is that it requires an incredibly large amount of storage in memory. My initial experiments with a voxel map with a resolution of 500x500x500 voxels were taking up around about 1Gb in RAM whilst running. Given that I had asked this system to store 125,000,000 voxels, each containing 4 numeric RGBA values, it is understandable. However, if I chose to lower the resolution of the voxel map it would greatly reduce the final quality of any render in my system.

To work around this, I looked into fast and efficient forms of memory management for this scale of data and analysed how much information I actually needed to store. Looking at the structure of a voxel map, it became clearer what was unnecessary storage:

- A voxel did not have to store a colour value unless it was actually used
- A voxel could re-use the same colour value of another voxel should they be the same colour
- The only thing unique to a voxel is it's position in a voxel map

In my program, I moved to making a voxel storable if and only if its colour had been explicitly set. One of the obvious forms of data structure for holding this information when it had been set was that of a binary tree. In such a tree, when using a voxels position as the sorting device, fast storage and access of voxel data became possible. To further improve the efficiency of this system, I went on to research and implement a balanced AVL tree after reading over code made available by Brad Appleton (Appleton, 1989).

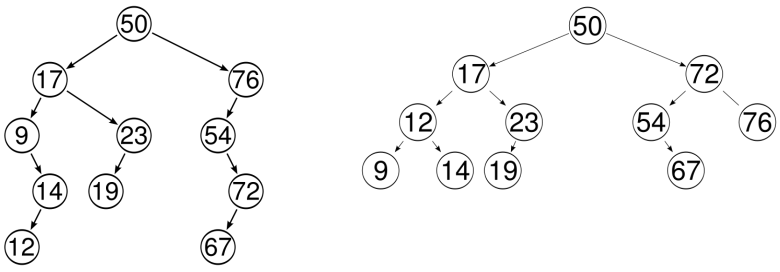


Fig 3. Left, an unbalanced binary tree and right, an AVL tree. (AVL Tree – Wikipedia, 2008)

With a feasible storage mechanism for a voxel map, I essentially had a blank three dimensional canvas around any given target geometry in which I could paint a pattern, such as a fractal. This though, was only half of the voxel map development. Having stored this information, I then needed a method of extracting the colours held within it and projecting them onto a two dimensional canvas which the viewer could see.

One such process is known as ray marching (Computer Graphics Lab of the Unversity of Waterloo, 1998) – for every pixel on the final 2D image plane, a ray (or several rays) is cast from the virtual camera through the point on the 2D plane and into 3D space. At set intervals along this ray, the current colour value of the voxel map (if inside one) is collected and accumulated. If the accumulated colour value has reached an opaque opacity or the ray has marched down a pre set number of iterations, the final colour value is returned.

For this, I had to devise my own virtual camera and method of projection. The reason behind this, rather than using existing solutions such as DirectX or openGL is that such systems are very orientated towards the use of polygons. When trying to avoid the use of polygons, their use becomes more loss than gain. Thankfully, the creation of my virtual world was largely managed with direct implementations of the maths described in Peter Comninos' Mathematical and Computer Programming Techniques for Computer Graphics (2005), which describes the necessary functions in detail, and some simpler illustrative help from CodeGuru (Joe Farrell, 2005).

In my ray marching system, the step between each ray sample is set to be a fraction of the width of a voxel (by default ½), meaning a useful number of samples is always obtained when marching through a voxel map. This does assume however, that there is only one voxel map present in the scene. At each step along a ray, this same fraction of a voxel is added to the final colour. By decreasing this fraction to a smaller number, it is possible to vary the detail gained from marching through a voxel map at a cost to the final render time. In addition to this, I also added a couple of supposedly time saving devices to what is a fairly slow process.

Rather than trace a ray for every pixel in the final image, I first project the co-ordinates of the voxel map bounding box to the screen. By taking the maximum and minimum bounding pixels, I reduced the amount of rays to be marched along significantly when rendering a voxel map that does not escape the screen edges.

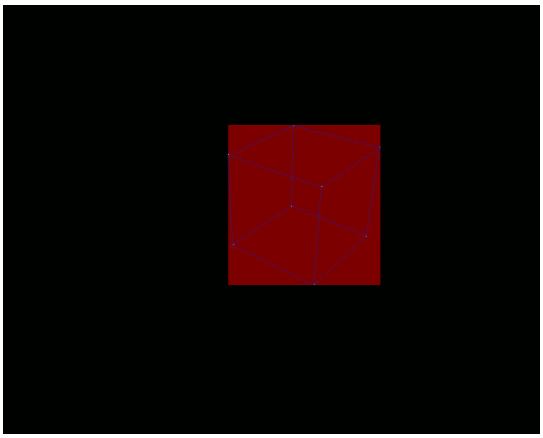


Fig 4. Illustrating how many pixels are traced given a target voxel map

Secondly, by making use of a ray tracer I wrote (details of which are found later on), I first ray trace rays through the image plane rather than ray march. A ray that collides with the bounding box of a voxel map will return two points in space – one entrance and one exit. By marching between these two points, I avoid having to march arbitrarily on until a maximum number of useless steps down a ray are performed. One of the criticisms of this is that doing this ray tracing may take longer than ray marching but given the relatively short processing time this takes, I have left it in the final implementation of my renderer.

With such voxel system in place and able to store and display its information, I started looking into how to fill it with the fractal patterns I wanted.

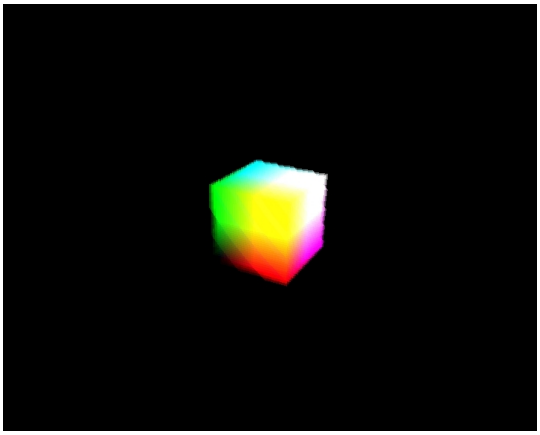


Fig 5. A 10^3 voxel cube in a 30^3 voxel map

3. The Creation of Fractal Patterns

3.1. Introduction to Fractals

'Fractal' was a term originally defined by Benoît Mandelbrot as "a rough or fragmented geometric shape that can be subdivided into parts, each of which is (at least approximately) a reduced-size copy of the whole" in his book The Fractal Geometry of Nature (1982). Mandelbrot himself was known as 'The Father of Fractals' (Yale Bulletin, 2003) and is most renowned for his creation of the Mandelbrot Set. This set, is one of two very common sets within fractal art generation. The second, the Julia was derived by Gaston Julia.

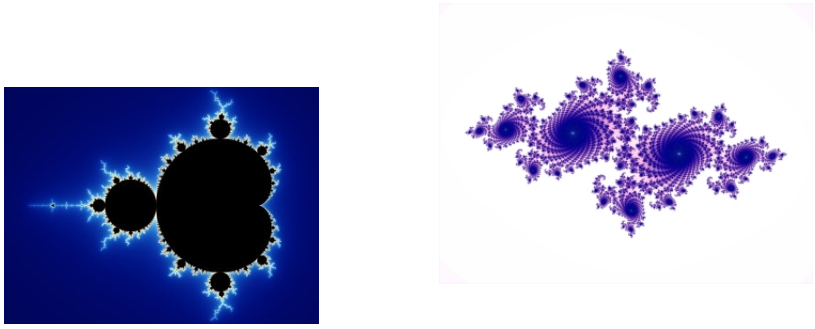


Fig 6. Mandelbrot and Julia sets. (Wikimedia Commons, 2008)

As suggested by his the book titles, Mandelbrot also studied the use of fractal algorithm's in regards to quite a diverse range of things such as it's appearance in nature (Mandelbrot, 1982) and in financial market systems (Mandelbrot, 2005). Given this broad range of application, it's possible to see how the term 'fractal' can be applied to a quite variety of sub categories. Some of the mathematical functions that fall into a fractal classification include - L-Systems, flame fractals, fractal terrain, Iterated Function Systems (IFS), attractors or the iteration of complex polynomials. For most of my research, I stayed away from the creation of L-Systems and fractal terrain for reasons stated in my introduction and chose to focus on the other listed forms of fractal.

Fractal function ‘sets’ tend to fall into one or more of these categories (dependent on the implementation).

IFS’ or Iterated Function Systems are often used to create this style of recursive fractals given that the resulting fractal object will always be self similar at an ever decreasing or increasing scale. Two prevalent examples of these would be, in 2D, the Sierpinski Triangle (Sierpiński, 1915) and in 3D, the Menger Sponge (Menger, 1926). The recursive nature of this definition of fractals makes defining them in Euclidian space (i.e. the space in which any target geometry will be defined) more difficult. At the very least, limiting the depth of recursion would be necessary.

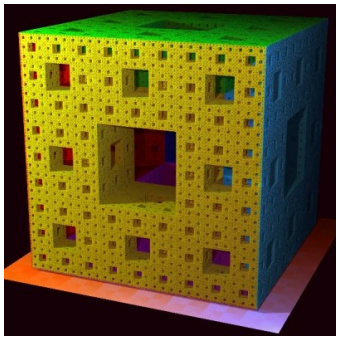


Fig 7. The Menger Sponge (Wikimeda Commons, 2008)

Some of the fractal function sets that got me interested in this area of research originally were based on this IFS structure. The fractal flame algorithm by Scott Draves (2003) created some incredibly beautiful imagery and forms the basis for much of the fractal art created today. Since it’s creation in 1992, fractal flames have been implemented as part of larger packages such as the open source image editor, The GIMP (Draves - website, 2005). Draves’ work as a VJ (Video Jockey) meant much of his work was geared to be beautiful imagery over beautiful mathematics. This lead to a visual style of ghostly lines unlike those seen in previous fractal renders using other function sets. It was this style of imagery that had first inspired me and that I wanted to use in my software, but using it did raise several problem.

3.2. What I was Looking For

For my software, I was looking for something specific from my fractal research. My primary aim was to be able to render geometry with Drave’s fractal flame style patterns filling its volume. This meant, whilst trying to emulate the style of fractal flames, I also needed to fill some practical criteria:

- The pattern must be three dimensional in order to allow clipping to 3D geometry.
- If using a recursive system, there must be a pre-defined limit to the depth of recursion.
- The pattern must allow the constructive use of colour (i.e. not randomly coloured)

Initially, I envisioned my program making use of some or all of the fractal flame function sets. However, after reading up on how they are created, I discovered it would not be quite so straightforward. Originally my perception of these flames is that they were constructed in 3D. However, all of the functions used are strictly two dimensional but manage to give the illusion of depth. This meant unless I attempted to redesign some of these functions to work in 3D, I’d be unable to convincingly clip them to three dimensional geometry. This was also true of other function sets like Mandelbrot and Julia. It seemed that without creating my own IFS functions to approximate the look and style of fractal flames, I was unlikely to be able to use these simple functions. Through my research around fractals, however, I found examples of other methods of generating patterns, of which some display fractal qualities.

3.3. Attractors

Attractors do not always conform to the definition of a fractal object, but they do share a lot in common with them. Attractors are a function set in which a dynamic system evolves over a long enough time (Attractor – Wikipedia, 2008). Some of these attractors, such as Lorenz (Lorenz, 1963) which have a fractal structure are described as 'strange attractors' which describe a chaotic dynamic trajectory in the attractor. Such attractors are strongly linked with research into chaos theory (non-linear dynamic systems). For the scope of this project however, I will be focusing on their use in creating visuals.

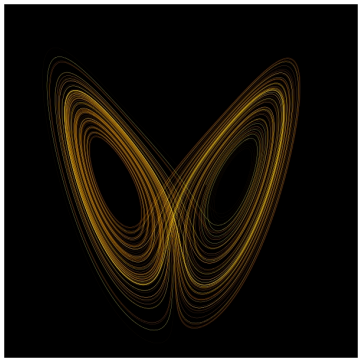


Fig 8. A lorenz attractor (Wikimedia, 2008)

In essence, an attractor is a mathematical function which when iterated repeatedly *may* yield a set of two or three dimensional points in an orbit around a point in a pattern not too dissimilar in style to a fractal flame. These functions are computed in the form:

$$Px_{n+1} = f(Px_n)$$
$$Py_{n+1} = f(Py_n)$$
$$Pz_{n+1} = f(Pz_n)$$

Where the attractor function *f* makes use of a number of preset parameters. Small changes in these parameters can make huge changes to the final outcome of the attractor. In my software, I've attempted to put this to good effect; allowing an attractor to be animated by altering it's parameters over time. However, in some cases the next successive point in an attractor may break its orbit and all successive points disappear towards infinity. This makes using them fairly experimental and often reliant on trial and error.

Despite this, I recognised the potential of these attractors in what I wanted to do, particularly if I chose to implement the 'safer' function sets – those unlikely to allow a point to escape the orbit of an attractor.

There are quite a number of different attractor functions in both 2D and 3D form. My main resource for the implementation of these different attractors was an online manual of the software Chaoscope (2007). This software provides a platform for experimenting with a variety of different attractors and their parameters, enabling users to find a multitude of viable solutions (where the orbit is maintained) and a means to render them as a high quality image.

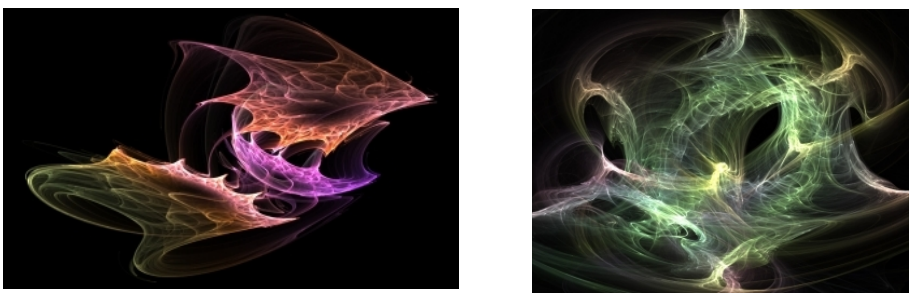


Fig 9. Example attractors rendered using Chaosope (2007)

Initially in my program, I chose to use some of the simpler attractors – those that were easiest to implement and easiest to alter whilst still producing viable solutions. Two of the most useful in testing and development were those derived by Lorenz – the Lorenz and Lorenz84 attractors. Both of these attractors offered very simple and light equations that were easily recognisable when correct given the level of documentation on them available.

One of the other noteworthy additions to the attractors available in my program was the Pickover attractor, as derived by Clifford Pickover and illustrated in Computers, Pattern, Chaos and Beauty (1991). This attractor uses trigonometric functions, trapping any output point in sinusoids (the scope of a cos and sin waves, from -1 to 1). This means this particular attractor does not allow points to escape orbit (although it will still to collapse to only a few recurring points on some parameter combinations). This makes this attractor much more flexible to experiment with when animating or searching for new parameter combinations.

Fig 10. A Pickover Attractor (Pickover, 1991 pg 164)

Appendix B contains a full list of the other available attractors and their mathematical functions.

When implementing my pattern creator in my program, I realised there was another hurdle to overcome. Often one of these attractors would be much greater in size than the original geometry and therefore extend beyond the borders of the voxel map around it. In order to scale down the points of an attractor, I first had to iterate once over the points in the attractor and map the minimum and maximum bounding points; after that, I could accurately scale down the attractor on a second iteration and whilst doing so, plot colour points from it inside my voxel map space. This unfortunately meant I was looping over my attractor twice the amount of the given iterations. In hindsight, I recognise I may well have been able to use fewer sample points on the first instance to get a rough idea of a scaling ratio, rather than a precise one. One of the side effects of this, though, was that when rendering multiple frames of an animated attractor, the scaling ratio and translation of the attractor would differ resulting in jumping and flicking between frames. For this reason, scaling in this manner has become a user choice rather than an assumption. In order to help with the scaling process and allow smooth animation, my application instead now reports back the ideal scale and translation to fit a pattern inside the designated voxel map when rendering a frame. A user can then choose whether to use this information by using the 'patternTransform' command (Appendix A).

A similar process to this was again used in applying colours.

3.4. Applying Colour

With a good knowledge of methods on how to generate points in a pattern, I also had to address the problem of what colour to set this point in my voxel map. Paul Bourke of the University of Western Australia outlines three plausible methods for doing this (Bourke, 1999). An attractor can either be shaded by time, by curvature or by speed.

Shading by time simply uses changes the colour of a point dependent on how far through the iteration of the attractor this point is. The result of which, given the repetative looping nature of attractors is a seemingly random noise effect of colour.

Alternatively, by taking the cross product of two vectors (one to the point previous to the current and one to the next) an attractor can be successfully coloured by curvature. However, this requires much more information an assumes that both the next and previous points are always known. The only method of using the next point in the attractor would be to either pre-compute all points all and store them (requiring a large amount of memory) or to re-compute the next point on each iteration as well as the current point (a very wasteful and time consuming processing task).

The final and most plausible solution is colouring by speed. By taking the distance between the last point and the current point being set, it's possible to add colour dependent on the current velocity. In such a system as this, the minimum and maximum speeds of an attractor must either be pre-defined by the user or pre-calculated. Given that I was already repeating through an attractor twice to gain scaling ratio, I chose to pre-compute the colour in this step. This allows the user to define two (RGBA) colours for minimum and maximum speed which would always be fully interpolated between dependent on velocity.

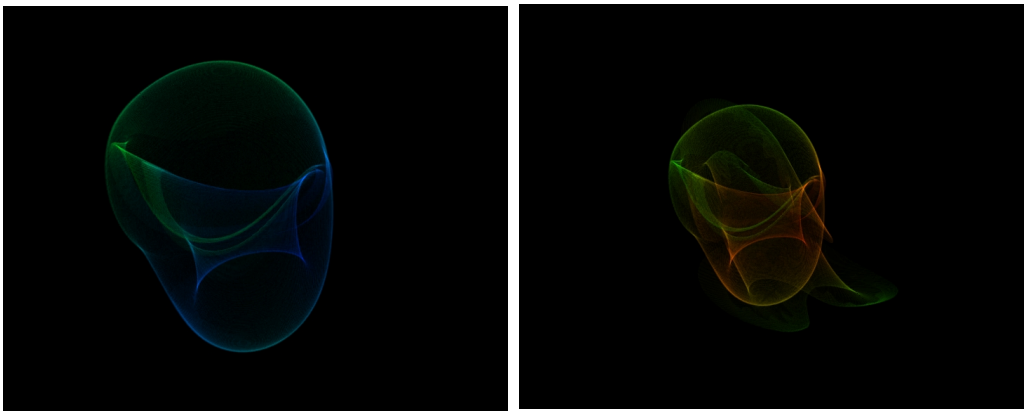


Fig 11. and 12. My coloured attractors. Left, Lorenz84 and right, Lorenz84 with Pickover

One of the possible changes I would make to this project is caused by this interpolation. Given that I chose to store colours in RGBA format, linear interpolation causes colours to get darker while not at one extreme or the other. Had I chosen to use the HSV (Hue, Saturation and Value) model of colour representation, I could easily interpolate between colours without this loss of vibrancy and, in so doing, produce a clearer final portrayal of the pattern.

As a final task in colouring my attractors, I required a method of ensuring that I would always extract the broadest possible colour range out of my final image. By setting all values from an attractor at a very small opacity (0.0001%), I could be reasonably sure no one voxel would ever be 'maxed out' (at RGBA values beyond 255, 255, 255, 1.0) and unable to add any more colour to itself. When retrieving these colours to the final image, the maximum and minimum colour values could be found and scaled to ensure the broadest range of colour and shade. This would, however, make darkest point in the pattern near black and the lightest point a very saturated colour. To aid in this, I chose to add clamping values for the minimum and maximum possible saturation (of 0.0 and 1.0 by default). These values could be changed to shrink the range in which the final image was multiplied to. Whilst this method doesn't yield the most striking result, it does lend itself to be easily manipulatable in image post processing.

3.5. Possible Expansions

With these methods of pattern creation, I was able to approximate the appearance of a fractal-flame within a 3D voxel map. With this area of research though, I felt I could have taken it much further and involved more methods of pattern creation in my system.

One of the features Draves' functions offered was the possibility of blending between the different functions to produce much broader range of visual results. In some of my earlier, and unfortunately lost, experiments I had attempted to do something similar with attractors but this only seemed to result in clouds of points rather than any visible pattern. Perhaps, though, my mistake was in ruling out Draves' equations on the basis they were two dimensional. Having seen how simple some of the attractors functions were, and how some, such as Pickover rather neglected the *z* value in their functions (Appendix B), it may have been quite possible to adapt Draves' equations to 3D. In so doing I may have made blending a possibility as well as ruled out the problem of collapsing attractors.

Other possibilities for generating patterns were seen in the quaternion Julia set and work by Julien Sprott. The quaternion Julia set takes the original 2D Julia set to a 4D one. This 4th dimension gives yet another range of possible results for this fractal pattern, as outlined by Bourke (2001). Julien Sprott shared an absolute wealth of possible attractors in many combinations in his book 'Strange Attractors: Creating Patterns in Chaos' (2000). Sprott's attractors use a polynomial form of equation allowing him to easily blend and multiply these functions together – as shown in the book. This creates an incredibly complex and broad range of possible attractors in both 2D and 3D, many of which would work well in my system.

Fig 13. Variants of the quaternion Julia set (Bourke, 2001)

Lastly, since I was pursuing a visual style more than specific fractal methods. I would've liked the opportunity to branch out and create my own patterns. One such possibility would be simple interpolating between the loaded geometry to produce a curve or line, along which I would be able to use whichever IFS functions I chose to define myself or recycle from other sources. By using the geometry itself as a parameter, I may well have created further interesting results.

4. Defining Geometry, 'Inside' and 'Outside'

4.1. Simple Geometry

In order to be able to clip the patterns I was able to generate to 3D geometry, I required a way of loading and storing this polygon information. By basing my method of storage on that found in Maya's API (2008), I was able to easily define what a geometric shape would be inside my system; vertices of an object are stored in the order they are defined, faces of an object reference the vertices by the index in number of which they were stored. This also lead to me using a similar exporting/importing technique to the Wavefront .obj file format (obj - Wikipedia) but ignoring most of the functionality (since I was not concerned with a surface but volume, UV's and normal information was all surplus).

With this information, I was able to load geometric data, compute the necessary bounding box volume around it and populate that bounding volume with a voxel map. To ensure the successful loading of the entire object, I also mapped the vertices into the voxel space which encompassed it – highlighting the appropriate vertex voxels red. This functionality is still available, but only within the source code.

4.2 Inside and Outside

Unfortunately, when dealing with polygons there is no easily recognisable way of differentiating between the inside or outside of an object. This problem is directly related to the 'point in polygon' problem illustrated as long ago as 1974 by Ivan Sutherland (1974), though in this instance, we are using each polygon face in the geometry rather than an edge of a single polygon.

The most common form of solving the point in polygon problem is to trace a ray in any arbitrary direction and count the number of intersections with, in this case, the polygon surface. If there are an odd number of intersections with the faces of the polygonal mesh, the point lies inside the volume. This is by far the simplest form of inside/outside test and hence the one I chose to implement. Looking at other possibilities for solving the same problem, most seem to encounter difficulties with efficiency. For example, one idea I had was to take the normal of the nearest polygon face and determine whether the point lay on the positive or negative side of this plane. This sounded reasonable but actually trying to compute the nearest polygon to the point is potentially very intensive. Instead, I stuck with this simpler method with a mind to improve its efficiency at a later date. One of the major refinements I had hoped to use was to plot the central positions of each face into a an octree (as illustrated by Octree – Wikipedia, 2008) and therefore, when tracing a point, would be able to arbitrarily dismiss any faces that do not lie directly along a vector in the X direction.

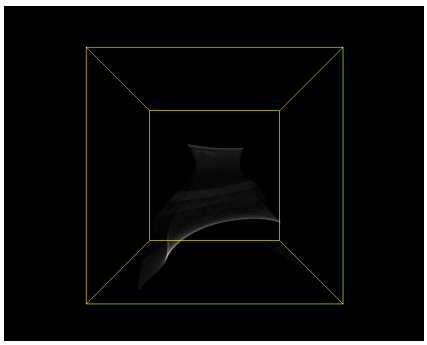
In order to do this method of testing, I required a ray tracer to go with my polygonal mesh. This ray tracing process is broken down into the following steps:

- A ray is defined as having a start point and a direction
- For every triangle in every polygon in the mesh, a plane equation is derived in the form $ax + by + cz + d = 0$
- If the ray and plane are not parallel, a point of intersection between the ray is computed using the above equation.
- The triangle vertices are flattened to two dimensions (UV space) by ruling out one axis coordinate
- The number of intersections with an edge of this triangle and a ray, defined by the point of intersection and the direction of the U axis, is calculated. If an odd number is returned, the ray has hit this triangle.

The above algorithm is clearly outlined and its implementation described in a Siggraph tutorial at (Owen, 1999). Using this, I was able to write a basic and vaguely efficient ray tracer that sat with my mesh definition, able to trace a given ray against itself and return the required information (see source files, JMesh.h).

Using this ray tracer, I was able to determine whether or not a voxel colour should be set dependent on if it was inside or outside of a mesh shape by tracing a point from the center of the target voxel across in the direction of the X axis. Since my rudimentary ray tracer checks both directions on the ray, I could be sure an accurate number of intersections would be returned regardless of the X coordinate of the point in space. However, this method always assumes it is working with a closed volume. This is one of the limitations of my implementation, but one I accept considering the complexities of trying to define the inside or outside of an unclosed volume. For efficiency’s sake, I also built an inside/outside flag into my voxels to avoid pointlessly checking a voxel repeatedly to see if it was inside the same volume.

This functionality, combined with that outlined earlier, gave me all of the essential parts of a functional, pattern clipping renderer.



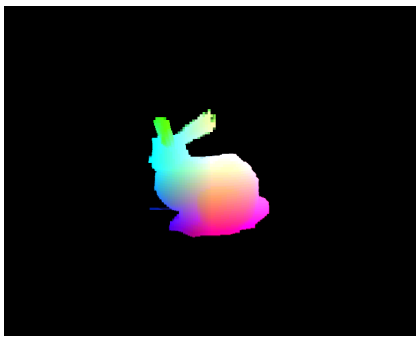


Fig 14. Lorenz84 clipped to a cone and Fig 15. Voxel clipping of the Stanford Bunny (www.stanford.edu)

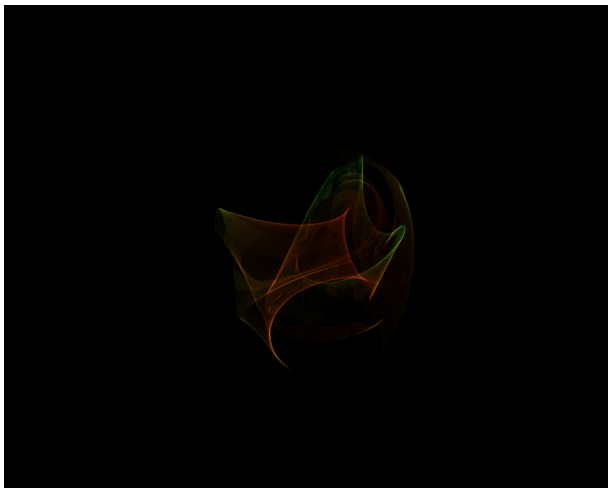


Fig 16. Clipping to a cube

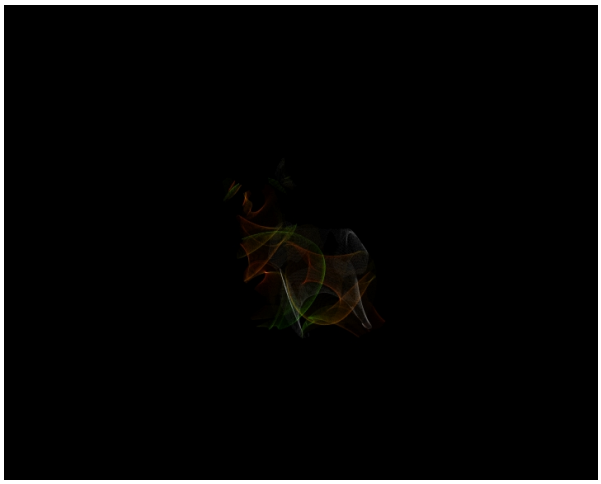


Fig 17. Clipping to the Stanford Bunny

5. Functionality and Usability

5.1. Writing Images

With most of the core functionality of my program in place, I needed to make it perform like a true renderer. In development, I had been relying heavily on wxWidgets (wxWidgets, 1992), a multi-platform user interface tool which allowed me to ensure my program worked on both Linux and Windows based machines. Whilst testing, my program had simply been drawing the final pixel data on the screen. To turn this into a renderer capable of outputting final images, I simply had to redirect this pixel data into a recognised file format. Within wxWidgets are the capabilities to write out images in various file formats. After employing these capabilities, to not make the original windowed program obsolete, I altered the display to show only the last frame successfully rendered.

5.2. Making it Usable

In order to easily change what my program rendered I needed a method by which I could set a great deal of variables quickly and easily. Rather than attempt to construct a full user interface for my program, I chose to simply use a command line given file as my input. Whilst this gave me the functionality I needed, it meant I was free to spend more time pursuing the main aim of this project.

To do this, I created my own simple ASCII based file format, a '.fs' (fractal settings) file which my program was able to recognise and read certain commands and values. In order to quickly construct these files, I wrote a separate Mel script for use in Maya which exports any selected geometry and a few pre-sets (Appendix D). This script was very rudimentary but managed to perform the tasks for testing and creating different outputs in my renderer. With further development, I had hoped to create a small user interface for this script in Maya and present to make my program far easier to understand and manipulate. Currently, this script doesn't properly capture Maya's camera settings.

A full breakdown of the commands recognised in the .fs file format is found in Appendix A.

5.3. Threading

When I undertook this project, I had envisioned being able to render out animations of its final results. Whilst this was not part of what I had originally outlined in my brief, I felt that the impact of an animated solution would outweigh that of developing further patterns or optimisation within a static one. My initial renders were being completed in times of 1-3 minutes after the full process of pattern generation, clipping, ray marching and image writing. By learning how to thread my application, I was able to render multiple images simultaneously. Through use of the wxWidgets threading abilities, I was outputting as many images as there were cores on a computer in the same time it was originally taking to do one.

One of the side effects of threading in this way, however, is that passing information between rendered frames becomes problematic. Information which is worked out on a per-frame basis could, in theory, be successfully passed to successive frames for re-use. Sharing such information would have greatly enhanced the performance of my renderer.

I should also point out a recent over-sight of mine when running multiple threads in my current implementation; Threads may conflict when trying to re-use pattern data unless a separate pattern is specified on every single frame.

6. Failures, Successes and Refinement

As a finished piece of software, I have managed to produce the effect I set out to at the start of the project. I do indeed have a renderer that takes a piece of geometry and clips a fractal-flame style pattern inside of it. Yet within this project, I can see a lot of room for refinement and modification to achieve a better result, some of which are mentioned beforehand. As well as this simple improvements such as the way in which a user is able interact with my renderer or easy speed optimisations would've gone a long way.

As a project, one of my regrets is delving straight into the development of a voxel renderer as the viable solution. Whilst I've managed to utilise voxels and get a respectable result from them, it's possible to argue their position in the project to begin with. Whilst I used voxels to store and recall image data in a 3D format, this was rather obsolete given the method in which I create patterns. Surely, if my patterns are created as points in Euclidean 3D space, I would be able to simply map a colour point at this value directly to the corresponding pixel on the screen. Such a system would greatly improve the quality and 'smoothness' of a final render of these volumetric patterns; Not only is marching through a voxel map a fairly intensive process, but detail is likely to be missed and passed over, particularly when using larger ray step sizes.
Of course, had I removed the voxel map completely, I would be left with an extremely large number of points from which to ray trace against a polygonal mesh to find out if it was inside or outside. In my tests, I was plotting 5,000,000 points per attractor. To ray trace this many times would be incredibly time consuming.

To that end, I would argue voxels do have their place in a system such as this by using them to define inside and outside positions of a mesh. In so doing, I would be able to ray trace once per voxel, as the current implementation does, but use that only as a guide to whether a colour value should be projected to the image plane. Each subsequent pattern point that fell in the same voxel could then be quickly labeled as inside or outside the mesh volume. Also, if such information was then stored in a BSP octree, rather than an AVL tree, I would have a fast and efficient method of checking a point against a mesh. Similarly, I could also clip this against the viewing frustum to further reduce the amount of needed ray tracing.

This criticism would be different had my system made full use of the voxel map data. With the information stored in a voxel map, it would be perfectly feasible to start passing the voxel map between frames in which either the pattern data or mesh data remained constant. It would always be easier to pass this data than re-compute it on another frame (something that would be particularly apt if attempting to build a real time system rather than an offline renderer). At one point I had considered implementing a version of this but found it became incredibly complicated when it came to threading; more often than not threads would need to access the same voxel map and organising and cloning these maps in memory would've taken more development time than I had remaining.

Whilst I'm glad I managed to use voxels successfully and to a good result, having now learnt more about their capabilities, I feel I did not use them for their most efficient purposes and that pursuing this system was detrimental to other areas of research, particularly the development of additional patterns and user interface.

Another minor downside to my system is its inability to process a polygon of more than 4 vertices. I did this largely because of my ray tracer and it's method of selecting triangles within a face to trace against. If a polygon had a greater number than 4 vertices, there was a good chance my simple ray tracer would ignore a hole in the center of the polygon. Another reason was that it made reading in a polygon of an arbitary number of vertices more difficult. By explicitly saying a polygon must be of 3 or 4 vertices, I side stepped the problem. So as a general note, for efficient render ring, a triangulated mesh is best.

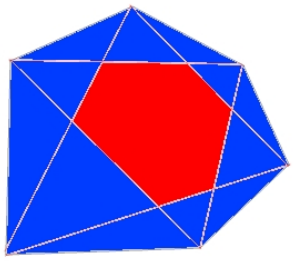


Fig 18. In a 6 sided polygon, my system will only define the blue triangles as part of the face.

A slightly more annoying trait in my renderer is that, since my virtual camera uses an aim point, it requires a default up vector. Without the addition of a way of altering this up vector from the default [0, 1, 0] a camera looking directly up or down breaks down and returns a black image. Similarly, whilst my camera settings allow near and far clipping planes, these are not strictly honoured by the voxel rendering method. Only if the entire voxel map is past one of these borders will it be removed from rendering. To take more time to implement this feature properly would have been simple and beneficial.

Some post processing of the final image could also aid the final outcome of this renderer; particularly gamma correction for displaying on CRT monitors, as addressed by Draves in his paper 'The Fractal Flame Algorithm' (Draves, 2003 pg 10), or motion blur to smooth out the movement of a fractal over time are both viable additions.

These points and refinements aside, I am very pleased with the results from this project. I have managed to successfully produce and clip fractal-flame like patterns in a three dimensional environment with decent results. Visually, I expected to be able to see a representation of the geometry a little clearer than I have currently been able to manage. However, I believe this is largely to do with a lack of efficient test of my application in order to retrieve the best results out of it. Had I had better time management skills, I may have been able to give further examples that I am certain this software is capable of.

I believe this project forms the basis for a lot of experimentation I wish to attempt later; Experimentation that will encorporate the valueable lessons learned from how this project could have been improved and the degree to which this method of rendering could potentially be taken.

A list of all included source files and their purpose in my program can be found in Appendix C

7. Bibliography:

1. Apophysis Program. (2007). Apophysis. Available: http://www.apophysis.org/. Accessed March 2008
2. Appleton (1989). AVL Trees: Tutorial and C++ Implementation. Available: http://www.cmcrossroads.com/bradapp/ftp/src/libs/C++/AvlTrees.html . Accessed March 2008
3. Bourke, Paul (1999). Colouring Attractors. Available: http://local.wasp.uwa.edu.au/~pbourke/fractals/colourattract/ . Accessed March 2008
4. Bourke, Paul (2001). Quaternion Julia Fractals. Available: http://local.wasp.uwa.edu.au/~pbourke/fractals/quatjulia/ . Accessed March 2008
5. Chaoscope (2007). 3D Strange Attractor Rendering Software. Available: http://www.choascope.org/ . Accessed March 2008
6. Comninos, Peter (2005) Mathematical and Computer Programming Techniques for Computer Graphics. : Springer
7. Computer Graphics Lab of the Unversity of Waterloo (1998) Ray Marching. Available: http://www.cgl.uwaterloo.ca/Projects/rendering/Talks/implicit/html/sld010.htm . Accessed March 2008
8. 'Cyborg' (2000) Interview with Ken Silverman. Available: http://archive.dukertcm.com/knowledge-base/reviews-interviews/ken-silverman-voxlap.html. Accessed March 2008
9. Draves, Scott (2003). The Fractal Flame Algorithm
10. Draves – Website (2005). Cosmic Recursive Fractal Flames. Available: http://flam3.com/index.cgi?&menu=children . Accessed March 2008
11. DeviantArt, (2007) Fractal Art. Available: http://browse.deviantart.com/digitalart/fractals/. Accessed December 2007.
12. Ebert, David S (2003). Texturing and Modeling: A Procedural Approach. : Morgan Kaufmann.
13. Farrel, Joe (2005) Deriving Projection Matrices. Available: http://www.codeguru.com/cpp/misc/misc/math/article.php/c10123__3/ . Accessed March 2008
14. Lorenson, William and Cline, Harvey (1987) Marching Cubes: A high resolution 3D surface construction algorithm. In: Computer Graphics, Vol. 21, Nr. 4
15. Lorenz, E. N. (1963). Deterministic nonperiodic flow. J. Atmos. Sci. 20: pg 130-141
16. Mandelbrot, Benoît (1982). The Fractal Geometry of Nature. W.H. Freeman and Company
17. Mandelbrot, Benoît (2005). The (Mis)behaviour of Markets: A Fractal View of Risk, Ruin and Reward. Profile Books Ltd
18. Maya Help (2008). Creating a 3D fluid Container· Available: http://download.autodesk.com/us/maya/2008help/wwhelp/wwhimpl/common/html/wwhelp.htm?context=Dynamics&file=Emit_fluid_properties_into_grids.html. Accessed March 2008
19. Maya Help (2008). MfnMesh · Available: http://download.autodesk.com/us/maya/2008help/API/MFnMesh.html. Accessed March 2008
20. Menger, Karl (1926) General Spaces and Cartesian Spaces. Communications to the Amsterdam Academy of Sciences. English translation reprinted Gerald Edgar (editor) (1993) Classics on Fractals. Addison-Wesley
21. Owen, G. Scott (1999) Ray Tracing. Available: http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm. Accessed March 2008
22. Pickover, Clifford (1991). Computers, Pattern, Chaos and Beauty. St Martins Pr
23. Filip Piękniewski (unknown) Fractals in Maple. Available: http://www-users.mat.uni.torun.pl/~philip/Maple_fractals/. Accessed March 2008
24. Sierpiński, W. (1915) Sur une courbe dont tout point est un point de ramification, C. R. Acad. Sci. Paris pg 302-305
25. Silverman, Ken (2000). Voxlap. Available: http://www.advsys.net/ken/voxlap.htm. Accessed March 2008
26. Sprott, Julien (2000). Strange Attractors: Creating Patterns in Chaos.
27. Sutherland, Ivan et al (1974) A Characterization of Ten Hidden-Surface Algorithms. ACM Computing Surveys vol. 6 no. 1.

28. Attractor – Wikipedia (2008). Attractor. Avaialble: http://en.wikipedia.org/wiki/Attractor . Accessed March 2008

29. AVL Tree – Wikipedia (2008). Avl Tree. Available: http://en.wikipedia.org/wiki/AVL_tree. Accessed March 2008

30. Fractal Art - Wikipedia. (2007). Fractal Art. Available: http://en.wikipedia.org/wiki/Fractal_art. Accessed February 2008

31. obj – Wikipedia. Obj. Available: http://en.wikipedia.org/wiki/Obj. Accessed March 2008

32. Octree – Wikipedia. Octree. Available: http://en.wikipedia.org/wiki/Octree. Accessed March 2008

33. Voxel - Wikipedia (2008). *Voxel.* Available: http://en.wikipedia.org/wiki/Voxel. Accessed Februrary 2008

34. Wikimedia Commons (2008). Wikimedia Commons. Available: http://commons.wikimedia.org/. Accessed March 2008

35. wxWidgets (1992) Cross-Platform GUI Library. Available: http://www.wxwidgets.org. Accessed March 2008

36. Yale Bulletin (2003). Father of fractals' wins Japan Prize. Available: http://www.yale.edu/opa/v31.n15/story6.html . Accessed March 2008

8. Appendix

8.1. Appendix A – FS File Format Commands

An example – example.fs is included

Command	Purpose
Global	
outDir <string>	Sets the out directory for output files
filename <string>	Sets the file prefix for output files
/frameStart	Starts a new frame.
/frameEnd	Ends this frame description
Frame (for use within /frameStart and /frameEnd)	
voxmap <int> <int> <int>	Use a voxel map of this resolution
transform <float>x16	Transform the voxel map by the given matrix (used to translate to the objects transform co-ordinates)
cameraPostion <int>x6	Set the virtual camera position where the first 3 numbers give the camera translation and the last 3 give a point to aim the camera toward
cameraAspect <float> <float>	Set the camera aspect ratio
setCameraFOV <float>	Set the camera Field of View angle
setCameraClip <float near> <float far>	Set the camera clipping planes
setClamp <float low> <float hi>	Clamp the minimum and maximum opacity levels in the final image from 0.0-1.0
resolution <int width> <int height>	Set the output image size
/objectStart	Start a new object description
/objectEnd	End object description
/patternStart	Start a new pattern description
/patternEnd	End a new pattern description
Object (for use within /objectStart and /objectEnd)	
v <float x> <float y> <float z>	Specify new vertex at x,y,z
f <int> <int> <int> <optional int>	Specify a new face in the shape with 3 or 4 vertex's where the integers are indexes to the vertices (vertices are stored in the order they are entered)
Pattern (for use within /patternStart and /patternEnd)	
wipe	Scraps any pattern data copied from previous frames
patternRescale <int>	Scale this pattern set to the size of the voxel map. 1 = on, 0 = off
patternTransform <float> x 16	Transform this pattern set by the given matrix. Performed after rescale.
setSpeedColour <float> x 8	Where the 8 numbers represent 2 RGBA colours in order, the first representing the slowest speed colour, the second the fastest speed colour
lorenz <int n> <optional float> x 4	Specify a Lorenz attractor iterated over n number of times. 4 optional parameters to differ from default settings. If one parameter is set, all must be. This is true for all attractors.
lorenz84 <int n> <optional float> x 5	Specify a Lorenz84 attractor. 5 optional parameters to differ from default settings.
pickover <int n> <optional float> x 4	Specify a Pickover attractor. 4 optional parameters to differ from default settings.
polyA <int n> <optional float> x 3	Specify a polynomial type A attractor. 3 optional parameters to differ from default settings.
polyB <int n> <optional float> x 6	Specify a polynomial type B attractor . Current disabled.
polyC <int n> <optional float> x 18	Specify a polynomial type C. 18 optional parameters to differ from default settings.

8.2. Appendix B – Included Attractors and Equations

Lorenz

$$\dot{x} = x + ad(y - x)$$
$$\dot{y} = y + d(bx - y - zx)$$
$$\dot{z} = z + d(xy - cz)$$

Default Parameters: a = 28, b = 46.92, c = 4, h = 0.01

Lorenz 84

$$\dot{x} = x + d(-ax - y^2 - z^2 + af)$$
$$\dot{y} = y + d(-y + xy - bxz + g)$$
$$\dot{z} = z + d(-z + bxy + xz)$$

Default Parameters: a = 1.111, b = 1.479, f = 4.494, g = 0.44, h = 0.135

Pickover

$$\dot{x} = \sin ay - z\cos bx$$
$$\dot{y} = z\sin cx - \cos dy$$
$$\dot{z} = \sin x$$

Default Parameters: a = -0.759494, b = 2.449367, c = 1.253165, d = 1.5

Polynomial Type A

$$\dot{x} = p_0 + y - zy$$
$$\dot{y} = p_1 + z - xz$$
$$\dot{z} = p_2 + x - yx$$

Default Parameters: p_0 = 1.586, p_1 = 1.124, p_2 = 0.281

Polynomial Type B

$$\dot{x} = p_0 + y - z(p_1 + y)$$
$$\dot{y} = p_2 + z - x(p_3 + z)$$
$$\dot{z} = p_4 + x - y(p_5 + x)$$

Default Parameters: p_0= -0.237, p_1 = 0.453, p_2 = 0.486, p_3= 0.616, p_4 = 0.141, p_5 = 0.483

Polynomial Type C

$$\dot{x} = p_0 + x(p_1 + p_2x + p_3y) + y(p_4 + p_5y)$$
$$\dot{y} = p_6 + y(p_7 + p_8y + p_9z) + z(p_{10} + p_{11}z)$$
$$\dot{z} = p_{12} + z(p_{13} + p_{14}z + p_{15}x) + x(p_{16} + p_{17}x)$$

Default parameters in patternDescriptor.h

Equations courtesy of Chaoscope.

8.3. Appendix C – Overview of Source Code Files and Functions

mainApp.h
Defines a wxWidgets application object of a wxWidgets application
main.cpp
Creates and initialises the application.

myFrame.h
Defines the main frame of the application interface
myFrame.cpp
Initializes the frame and manages the rendering process. This creates and starts the render threads and writes images to files when they are found to be completed.

renderThread.h / renderThread.cpp
Defines and initalises the rendering threads and performs the computation of a frame image.

myCanvas.h / myCanvas.cpp
Defines a wxWidgets frame that sits within the main frame of the application. It is tasked to display the last completed image. This is updated by myFrame.

renderSettings.h / renderSettings.cpp
Defines a storage class for reading in a .fs file. This is created in main.cpp and attached to the input file. The input file is processed resulting in an initialised render settings object with an array of ready to render frames.

renderFrame.h / renderFrame.cpp
Defines a storage class for all render data necessary for this frame – The voxel map to be populated, the patterns (in the form of a pattern descriptor) to be used in it, geometry data, camera data, image width and height, the output file name and the output directory.

voxmap.h / voxmap.cpp
Defines a class for storing and manipulating voxel data in a voxel map. Geometry data is passed to this from the parenting renderFrame.

patternDescriptor.h
Defines a class that holds a main pattern description class and derived classes for all available pattern types. These descriptions hold individual pattern data for each pattern to be rendered. These also hold the default pattern parameters when they are not specified by the user.

pattern.h / pattern.cpp
Defines a class that performs the drawing of patterns in a voxel map. When the final image is requested from a renderFrame, the renderFrame attatches a pattern object to the voxel map it holds, then feeds the pattern object the patternDescriptors it holds. These are drawn and clipped against the voxel maps held geometry.

voxTree.h / voxTree.cpp
Defines a storage class for an AVL tree of voxels.

Data Types

JVoxel.h
Defines a Voxel storage class. A pointer to these is held in a voxTree node. The voxmap handles the creation and destruction of voxels. Holds: R, G, B, A and boolean insideMesh/outSideMesh data.

JCanvasPixel.h
Defines a storage class for colour detail for the final drawn image in the form: Window x, window y, R, G, B A.

JCamera.h / JCamera.cpp
Defines a camera object. These perform the necessary projections to a 2D plane.

JMesh.h / JMesh.cpp
Defines a mesh object capable of holding the necessary mesh data as well as tracing a given ray against itself.

JFace.h
Defines a face storage class that is used by JMesh to store face data. Holds the integer index values corresponding to the array of vertices held in a JMesh object.

JVector.h / Jvector.cpp
Defines a vector data storage and manipulation object. Based on Rob Batemans libMath version*.

Jmatrix.h / Jmatrix.cpp
Defines a matrix storage and manipulation object. Also based on Rob Bateman's libMath version*.

JIntArray.h
Defines a class that stores an array of integers. Used by JMesh on initialisation.

JVectorArray.h
Defines a class that stores an array of JVectors. Used by JMesh on initialisation.

**LibMath available from http://www.robthebloke.org/*

8.4. Appendix D – writeFS Mel Script

Please see writeFS.mel
Suitable for easy use with testScene.ma