# GPU Fractals!

Pete Corey

California Polytechnic State University

Computer Science Department

Fall 2010

## *Abstract*

This project implemented a real-time, GPU-based Mandelbrot and Julia set fractal viewer. The fractal calculations were carried out entirely in GLSL fragment shaders. A standard iteration count coloring algorithm was implemented along with a triangle inequality based coloring algorithm. Coloring of the fractals was done through the use of color look-up textures provided by the user. The viewer can achieve zoom scales of approximately $1x10^{-6}$ before visible artifacts produced by lack of floating point precision can be seen.

# Introduction

The Mandelbrot set and the Julia sets are complex forms of fractal geometry that have incredibly interesting and beautiful visualizations. The Mandelbrot set is created by defining a sequence of complex numbers, denoted *z*. Each sequence is calculated recursively, such that:

$$z_{n+1} = z_n^2 + c \qquad z_0 = 0 + 0\mathrm{i}$$

Where $z_{n+1}$ represents the next value in the sequence, and c represents a point in the complex plane being tested for acceptance into the Mandelbrot set. If the sequence *z* diverges to infinity, point *c* is not a part of the Mandelbrot set. The point *c* is part of the Mandelbrot set if the sequence *z* converges.

Julia sets are calculated in a similar way to the Mandelbrot set. Instead of defining $z_0=0$ and *c* equal to the point being tested for acceptance, Julia sets are calculated by defining $z_0$ to be the point being tested for acceptance, and *c* is defined as a constant. Each value of *c* produces a unique Julia set. Acceptance into a Julia set is still determined by whether the sequence of *z* converges or diverges.

Testing for the convergence or divergence of an infinite set is a difficult task that by definition requires an infinite number of calculations to determine. Fortunately, due to a convenient mathematical property of the Mandelbrot and Julia sets, divergence of a sequence, *z*, can be determined relatively easily. Each sequence of complex numbers, *z*, can be viewed as an orbit around a point on the complex plane (*0+0i* for the Mandelbrot set, and *c* for Julia sets). If the radius of the orbit exceeds *2* at any point in the sequence, the sequence is proven to diverge to infinity. Unfortunately, testing for convergence is a more difficult task. Suppose that after calculating fifty values in the sequence *z*, the sequence has not diverged. It is impossible to determine if $z_{51}$ will reach an orbital radius beyond *2*. Because of this, the sequence of *z* is usually truncated to some large value of *N*. By truncating the sequence, the convergence test becomes a finite algorithm with an order of complexity of *O(n)*.

The nature of the Mandelbrot and Julia sets lend themselves perfectly to being solved on a modern graphics processing unit. The sets are comprised of a large number of points that define the "shape" of the set. Each point must individually be tested for acceptance in the set. The process for testing acceptance is the repeated calculation of a relatively computationally simple recursive relationship. Each point in the set is completely independent of any other point in the set and can be tested for acceptance with no external information about the set or its neighbors. Because of these factors, a GPU based implementation of a Mandelbrot and Julia set viewer would produce incredibly efficient and interactive real-time visualizations of the Mandelbrot and Julia sets.

## Previous Work

Due to the incredibly large number of mathematical operations required to render a single image of the Mandelbrot set, the interesting fractal geometry of the set was largely unknown until the advent of computers. It wasn't until March 1st, 1980 that Benoit Mandelbrot created and witnessed the first ever rendering of the Mandelbrot set. Since that day, countless hours of research and computer processing cycles have been poured into researching these unbelievable mathematical objects [5].

Traditionally, Mandelbrot and Julia set fractal viewers have been implemented using the CPU as its primary means of computation. Programs such as Fractint and Ultra Fractal were developed in the mid 1990s. Fractint, created in 1995, allowed users to create scripts which described the fractal formula and coloring algorithms to be used. The script was then interpreted by the program, and the resulting image of the described fractal was then procedurally rendered on the CPU [3]. Ultra Fractal, created in 1997 and still being actively developed today, built upon the interface developed by Fractint. Like Fractint, Ultra Fractal allows the user to manually write scripts to describe fractal formulas and coloring algorithms. However, Ultra Fractal has evolved into a massive software suite designed to allow the user to customize every imaginable aspect of the resulting fractal rendering through a vast

series of menus and control panels [6]. However, the overall work flow remains the same as that of Fractint. Parameters of the rendering, such as zoom depth, position and constant values, much be specified before the image is rendered. Any modifications to these values require that the rendering be entirely recalculated.

The process of rendering a Mandelbrot or Julia set on the CPU is a lengthly one. For a reasonable value of N, such as 1000 (each sequence, *z*, is calculated to a maximum of 1000 iterations), a full screen rendering of a fractal set where each pixel represents a single point being tested for acceptance is approximately an $O(n^3)$ algorithm. Additionally, when the zoom depth becomes large enough, standard floating point precision or double precision is not enough to accurately represent the set. To overcome this problem, larger, more complex data structures must be used to compensate for the high degree of precision required. Introducing these additional factors greatly increases the time required to calculate and render the Mandelbrot and Julia sets using a single CPU.

Fortunately, the limitations imposed by single core CPUs are almost entirely removed by using a Graphics Processing Unit (GPU) instead of a CPU. While single core CPUs only have one processor on which to do a series of calculations, one after the other, a modern GPU can perform a huge number of computations in parallel. This means that certain programs can be executed on a GPU in a fraction of the time it would take to execute the same program on a comparable CPU. With the recent boom in popularity of Graphics Processing Units, many developers have begun to notice how well the Mandelbrot and Julia sets lend themselves to computation on these highly parallel pieces of hardware. Because the calculation of the *z* sequence requires very few inputs (zoom depth, a point and a constant) and is completely independent of its neighbors and produces a single output (a color), it is perfectly suited for a GPU based implementation. Several GPU based Mandelbrot and Julia set viewers exist, such as Eric Bainville's OpenCL implementation [1]. However, many of these implementations exists as simple proofs of concept. The were built with minimal regard for usability and functionality.

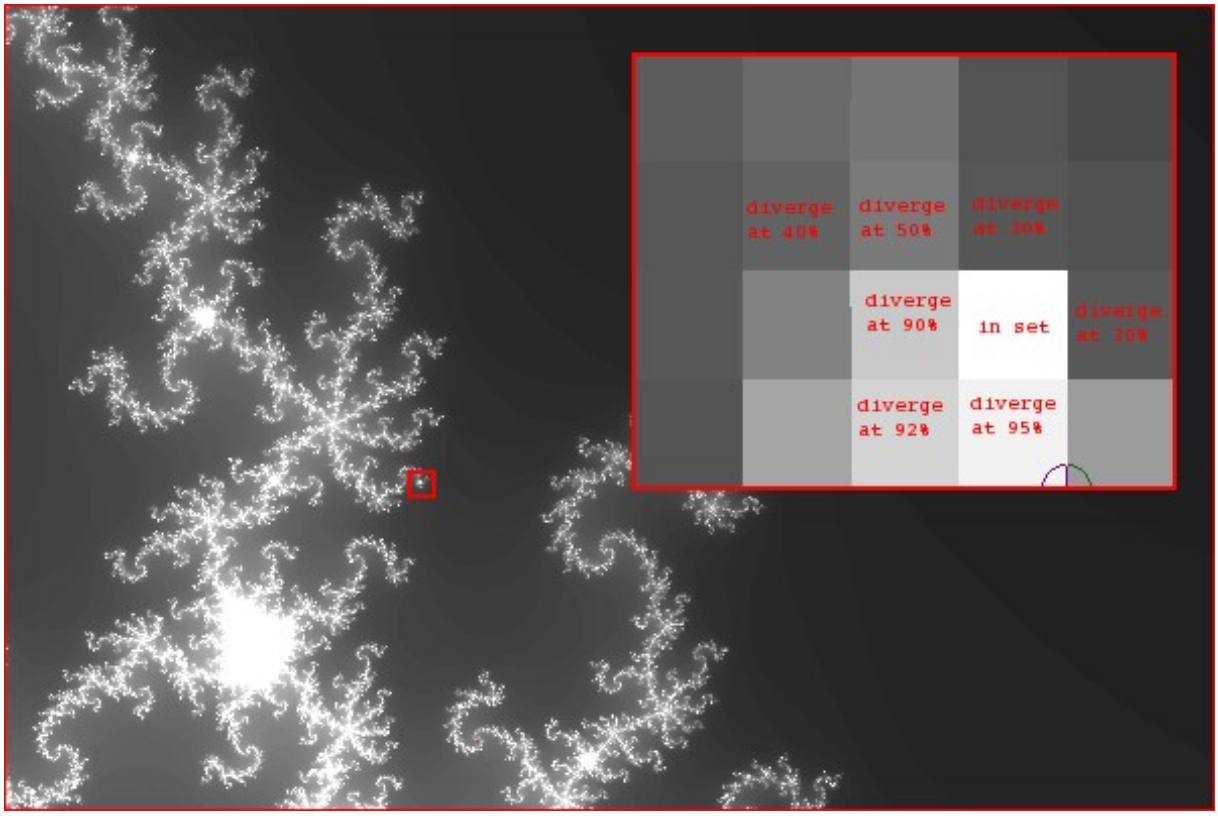
Additionally, because most GPUs only support floating point precision, many of these GPU based implementations either break down at high zoom depths, or switch to a CPU based approach once they reach a certain depth.

# Algorithm

## *General Overview*

From a high level perspective, the algorithm used to generate images of Mandelbrot and Julia fractals is very simple. Imagine the computer screen as a graph of the complex plane. The graph is oriented in such a way that the origin of the graph is at the center of the screen, the imaginary axis is the screen's *Y* (vertical) axis, and the real axis of the graph is the screen's *X* (horizontal) axis. For simplicity's sake, assume that the screen bounds the graph between 1 and -1 on the imaginary axis and 1 and -2 on the real axis. Each pixel on the screen can now be viewed as a discrete point on the complex plane. For example, the top left pixel on the screen represents the complex number $-2+1i$, and the bottom right pixel on the screen represents $1-1i$.

It quickly becomes apparent that each of these discrete points on the complex plane can be tested for acceptance into either the Mandelbrot or Julia sets. By calculating acceptance, we can determine whether the point/pixel is included in the set, or not included. If the point is not included, we can approximate its rate of divergence to infinity by determining how many iterations of $z$ were calculated before the series diverged. A simple coloring algorithm could then color the graph accordingly. In the example image below, white pixel are complex points within the Mandelbrot set, while non-white pixels are points found the be outside of the set. The gray scale color is determined by the rate of divergence. A black pixel means the series diverged instantly and a white-ish pixel means that a large number of iterations of $z$ were calculated before a divergence was detected:




## *Shader Overview*

This implementation of the Mandelbrot and Julia set viewer was creating using C/C++, OpenGL and GLSL shaders. One fragment shaders exists for each combination of fractal type (Mandelbrot or Julia) and coloring algorithm (Iteration count and Triangle Inequality). The OpenGL rendering loop in the program is incredibly simple; it simply draws a *GL_QUAD* from (-4,2) to (2,-2). This rectangle will be the "canvas" on which the fractals are drawn. The vast majority of the work is done in the vertex and fragment shaders. The four vertecies of this rectangle are passed to a simple vertex shader that is used for all fractal renderings. This shader simply performs the correct vertex transformations and most importantly populates a *varying* variable called *fragPos* with the position of each vertex. When this variable is passed to the subsequent fragment shaders, its values are interpolated across the rectangle and the values in the fragment shaders are used as the "points" being tested for acceptance by the fractal set.

***default.frag:***

```
varying vec3 fragPos;

void main()
{
    gl_Position = ftransform();
    fragPos.xyz=gl_Vertex.xyz;
}
```

The real intricacies of the GPU based implementation are found in each of the fragment shaders. Each shaders is used to render each combination of fractal type and coloring algorithm. With two different fractal types and two different color algorithms, this means there are four separate shaders. Each of these fragment shaders carries out a basic calculation of each *z* set. This calculation requires the repeated recalculation of $z_i$ where *i* ranges from 0 to N (denoted *maxIter* in the shaders):

***mandelbrot.frag:***

```
void main()
{
    vec2 z=vec2(0,0);
    vec2 c=zoomTarget.xy+fragPos.xy*zoomFactor;
    bool inSet=true;

    int maxIter=2000;
    int escapeVal=0;

    for (int i=0;(i<maxIter)&&(inSet);i++) {
        z=complexMult(z,z)+c;
        if (complexAbs(z)>=2.0) {
            inSet=false;
            escapeVal=i;
        }
    }
    if (inSet) {
        gl_FragColor=vec4(1,1,1,1);
    }
    else {
        vec4 color = texture2D(colorTex,vec2(0,(float)escapeVal/maxIter));
        gl_FragColor=color;
    }
}
```

The code found in the ***mandelbrot.frag*** sample simply carries out the Mandelbrot equation with $z_0$ and $c$ set to their corresponding Mandelbrot values. If the point is found to be within the Mandelbrot set, it is colored white. Otherwise, the resulting fragment color is set to the value of a coloring texture with a lookup value of the number of calculated iterations before divergence, divided by the maximum number of iterations (*escapeVal/maxIter*). The basic Julia set calculations are nearly identical to that of the Mandelbrot set, except that the initial values of $z_0$ and $c$ are different:

***julia.frag***

```
...

vec2 z=zoomTarget.xy+fragPos.xy*zoomFactor;
vec2 c=cInit; //where cInit is some constant specified by the user

...
```

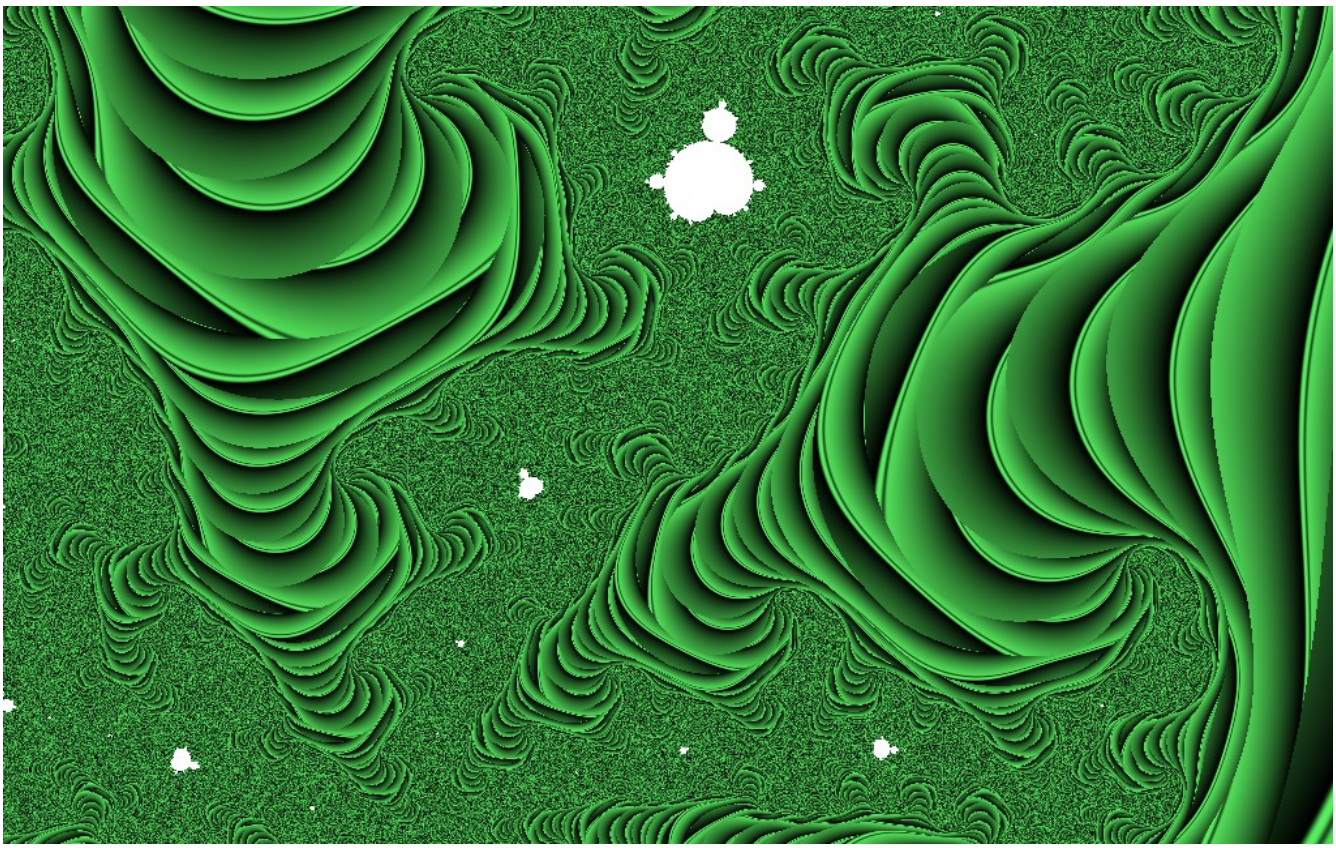
## *Iteration Count & Coloring Textures*

The primary coloring algorithm implemented by this application is a simple iteration count coloring algorithm. Iteration count algorithms simply count the number of iterations of $z$ which have been calculated. If the sequence is determined to be divergent (the radius of the orbit exceeds the escape value), the number of prior iterations is stored. This stored number is then divided by the total possible number of iterations, N. This division results in a real between 0 and 1, with 0 representing immediate divergence and 1 representing the slowest representable divergence. Instead of using this value directly in the fragment color computation, the value is used as a lookup into a color texture. This allows the user to very easily define the entire spectrum of colors available to the program. Additionally, different scales of coloring, such as logarithmic coloring, can be easily approximated by appropriately coloring the the color texture. The use of the color texture and the bailout coefficient give the user a great deal of flexibility when it comes to artistically coloring their fractal renderings. This technique is applied to both the iteration count coloring technique and the triangle inequality

algorithm described below.

## *Triangle Inequality Coloring*

The coloring algorithm referred to as the "Triangle Inequality" algorithm in the application is a implementation of an average coloring algorithm described by Jussi Harkonen in his master's thesis. The implementation was based on triangle inequality coloring algorithm found in Ultra Fractal and created by Damien M. Jones. The algorithm operates in a two-fold way. By averaging properties (usually the absolute value) of the sequence, $z$, as it is iterated out to N, a branching tree-like structure emerges. Unfortunately, a simple average results in large unsightly bands between iteration levels [4]. These bands can be seen in the screen shot below:



The solution to this banding problems is to smoothly interpolate between different iteration

levels. By smoothly transitioning between iteration levels, a smooth tree-like structure emerges. Screen shots of the results of this coloring algorithm can be found in the next section.

### *Precision*

In its current state, the algorithm is limited to 32bit floating point precision. This is largely do to the fact that most GPUs only natively support up to 32bit floating point numbers, with a small number of newly emerging GPUs supporting double floating point precision. With floating point precision, the application can reach zoom scale factors of approximately $1x10^{-6}$ before visual distortions in the renderings become apparent. Fixed point implementations are intended as future work, but a fixed point implementation would only benefit from a relatively small increase in the total possible zoom scale. Reaching a much larger zoom depth would require a drastic rework of the entire algorithm.

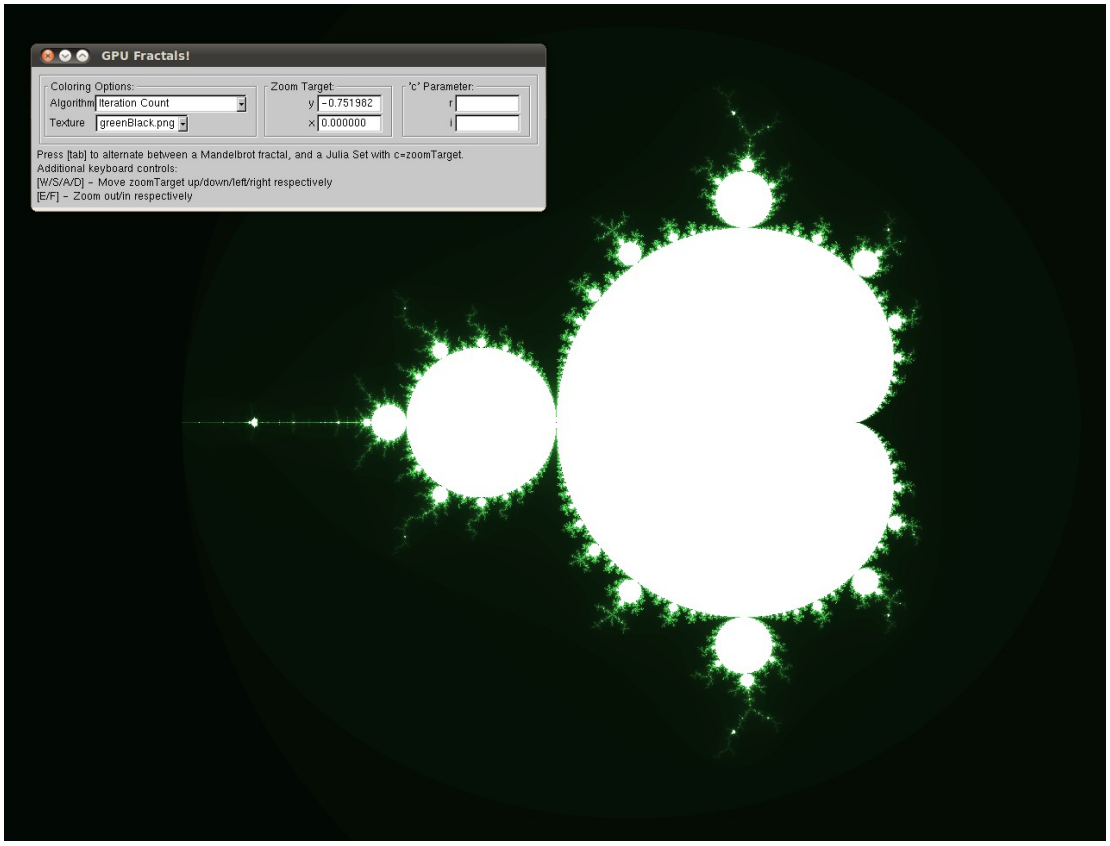
## Results

This fractal viewer was build and tested entirely on a Intel Core 2 Due 3.2 Ghz CPU with 4GB RAM and an 1024 MB Nvidia GTX 285 GPU. When running at a resolution of 1280x1024, the program responds in real-time. Unfortunately, because the program only re-renders on user input, no frame per second calculations were done. With the restrictions on floating point accuracy imposed by the GPU, the render displays visible artifacts at a zoom factor of $1x10^{-6}$ and beyond.

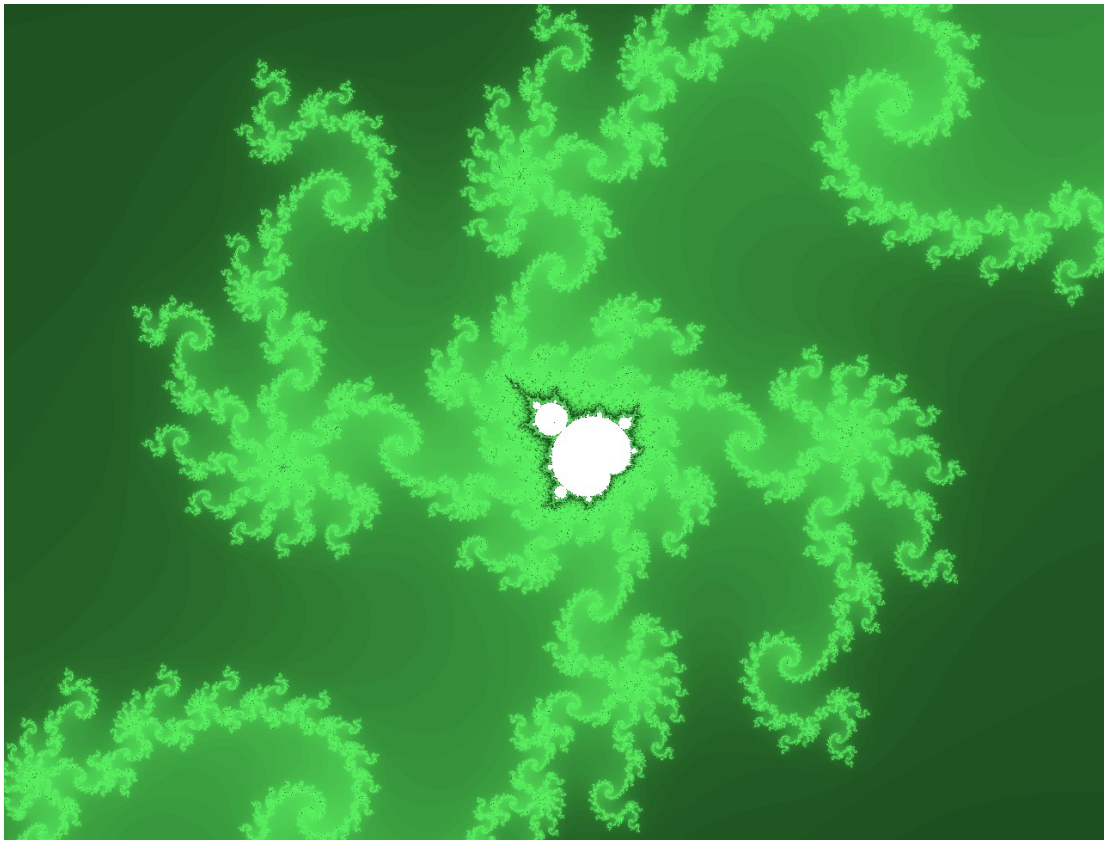
Additionally, due to an interesting aspect of the algorithm, sections of the fractal that are composed of a large number of pixels within the set will be renderer much slower than sections containing mostly diverging pixels. This is because the algorithm does not calculate the total number of iterations of $z$ for divergent points. Instead, only a fraction of the total number of iterations are calculated before a divergence is detected. Divergent point bail out of the loop faster than convergent

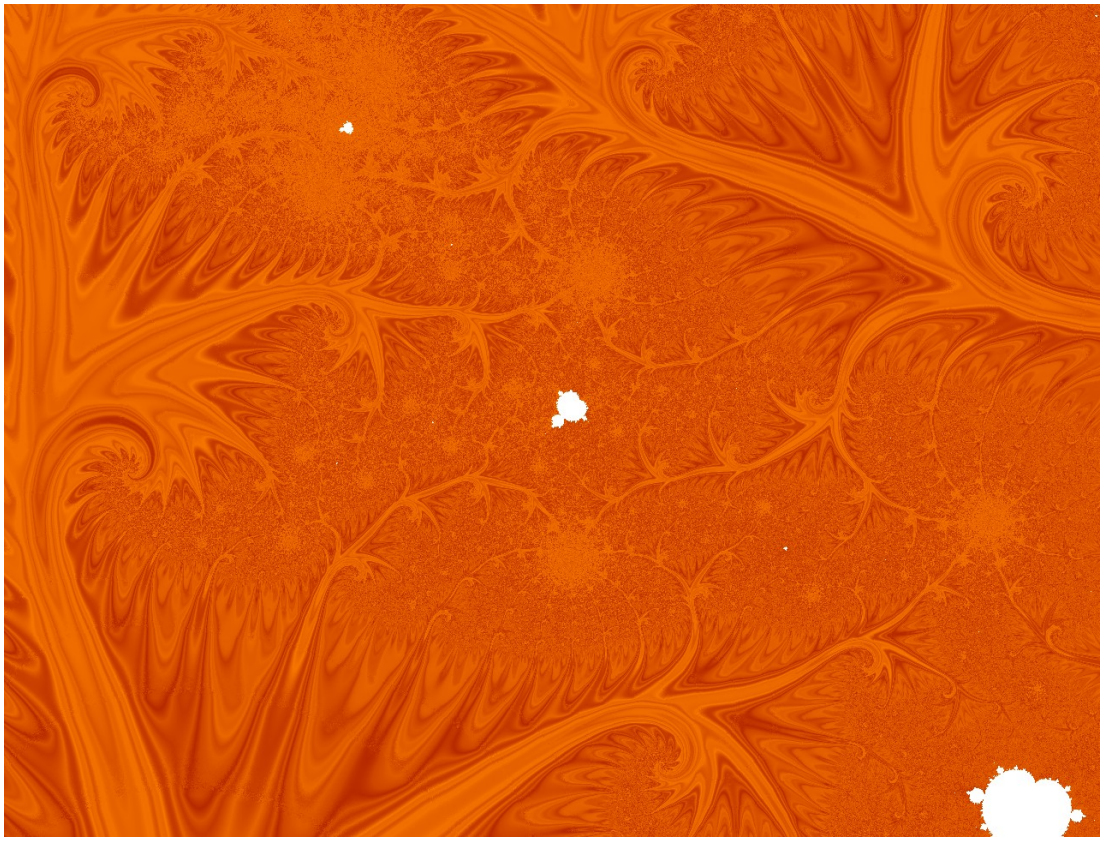
points, and are renderer more quickly.

The next several pages show screen shots of application's user interface, and the Mandelbrot and Julia set renderings created by the program.

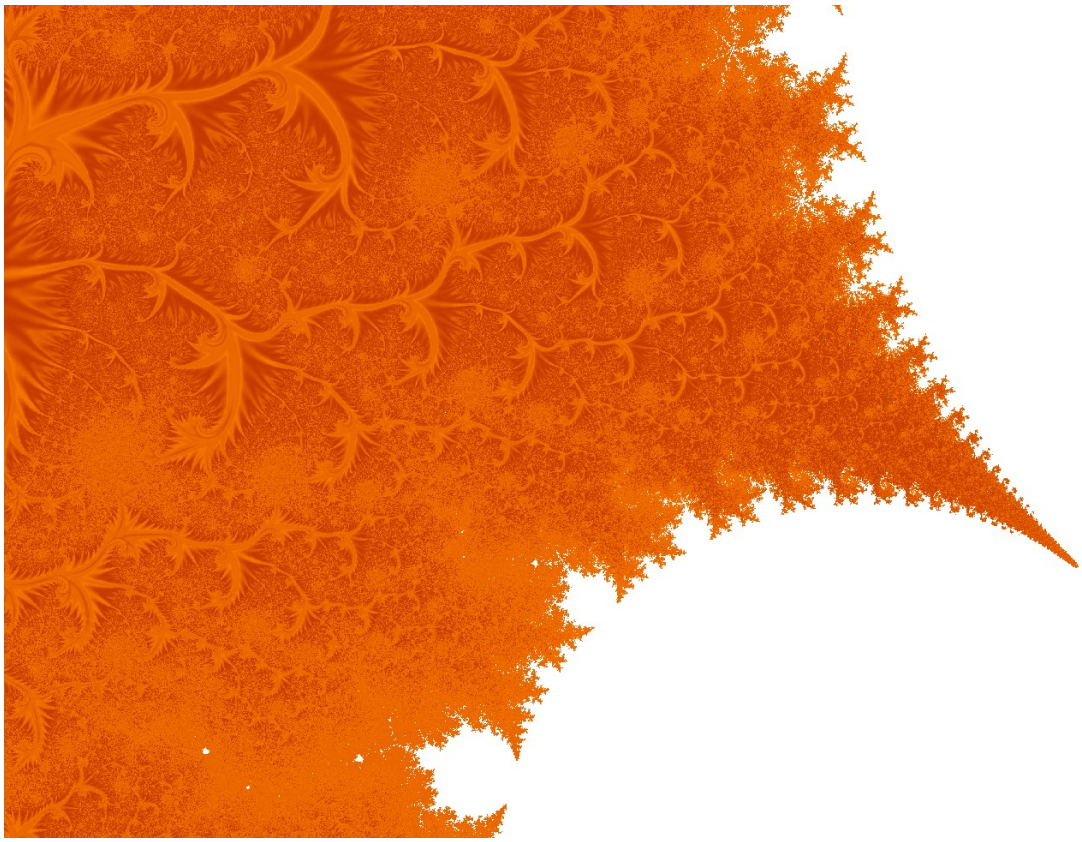


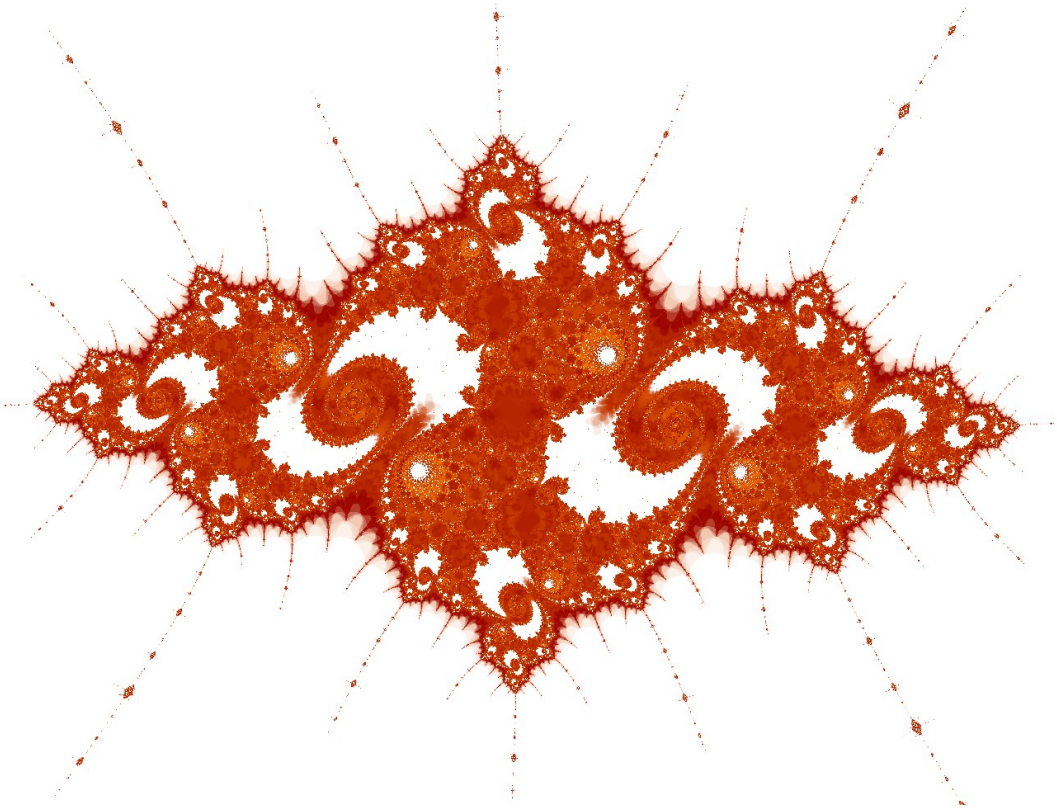

*Simple Mandelbrot sets w/ Iteration Count coloring algorithm.*



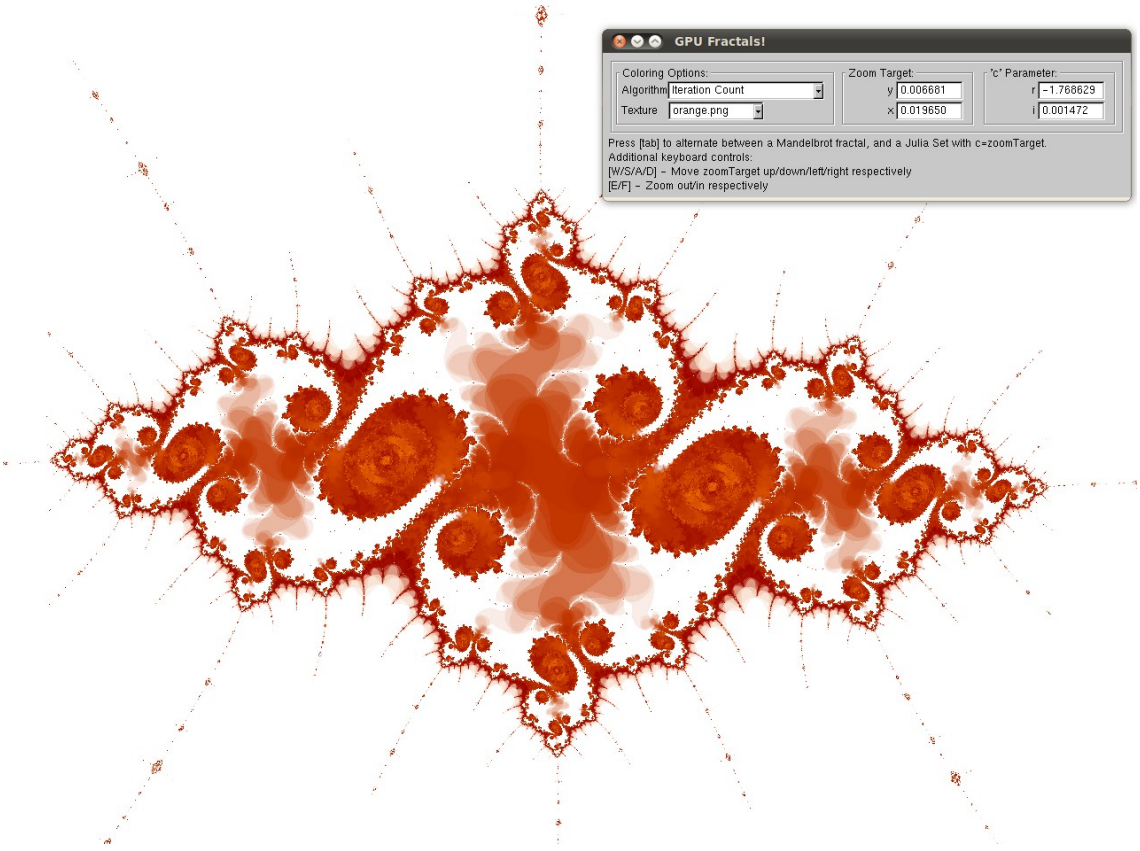


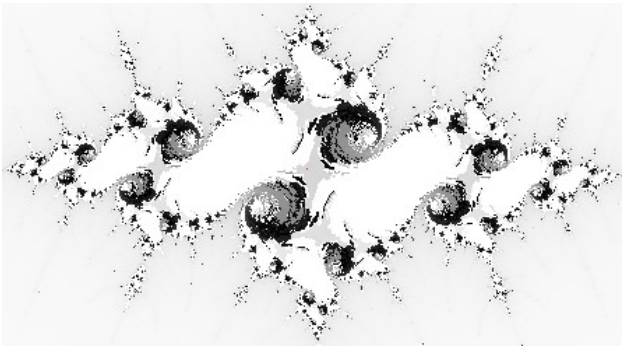
*Mandelbrot with Triangle Inequality coloring algorithm.*





*Above: Julia set w/ Iteration Count. Below: Careful selection of C lets user exploit the color texturing.*





*The artifacts left by the GPU's low precision don't always look bad!*

## Future Work & Final Thoughts

The ease in which the calculations of the Mandelbrot and Julia sets lend themselves to a GPU based approach cannot be emphasized enough. The highly parallel and independent nature of GPU based computations creates a perfect platform to efficiently calculate and draw these beautiful pieces of fractal geometry. Currently, the most formidable obstacle facing computation on GPUs is the lack of native extended floating point support. Because of this, the primary future work on this project should be dedicated to creating a system (either fixed point or otherwise) that would support further floating point accuracy and thereby increasing the possible zoom depths reachable by the renderer.

Many more avenues of work exist as well. Additional parameters of the rendering should be presented to the user. For example, the user should be able to control the color and coloring algorithm of the interior of the fractal sets as well as the exterior. Additionally, many more coloring algorithms could be implemented to give the user more artistic options. Countless artistic additions could be made to accentuate the fractal geometry of the rendering such as color cyclings (a continuous shifting of the color texture), and user defined animations. Hopefully, the power of the GPU will lead more people to discover interesting aspects of the amazing Mandelbrot and Julia sets.

## Works Cited


[1] Bainville, Eric. “CPU/GPU Multiprecision Mandelbrot Set.” *Bealto*. <http://www.bealto.com/mp-mandelbrot_intro.html> Web. 20 Oct. 2010.

[2]Beacham, Bradley. “An Introduction to the Fractint Formula Parser”. Revision 1. Feb 1995. <http://spanky.triumf.ca/www/fractint/frm-tut/frm-tutor.html> Web. 1 Nov. 2010.

[3] Giffin, Noel. “Fractint”. <http://spanky.triumf.ca/www/fractint/fractint.html> Web. 1 Nov. 2010.

[4] Harkonen, Jussi. “On Smooth Fractal Coloring Techniques.” *Violet Industries*. 2007. <http://www.violetindustries.com/gallery.php?cat=techniques> Pdf. 1 Aug. 2010.

[5] R.P. Taylor & J.C. Sprott. "Biophilic Fractals and the Visual Journey of Organic Screen-savers." *Nonlinear Dynamics, Psychology, and Life Sciences, Vol. 12, No. 1*. Society for Chaos Theory in Psychology & Life Sciences. Retrieved 1 November 2010.

[6] Slijkerman, Frederik. “Features.” *Ultra Fractal*. <http://www.ultrafractal.com/features.html> Web. 15 Sep. 2010.

# Appendix

## *README*

***GPU Fractals!***

```
By Pete Corey (pcorey@calpoly.edu)
California Polytechnic State University (Cal Poly)
December 2010

Requirements:
  Linux
  GPU with OpenGL 2.0 support.
  GLUT, GLUI, GLEW, DevIL libraries.

Control Notes:
  'WSAD'/Mouse Drag: translates around the fractal.
  'F'/[Scroll Wheel Up]: Zooms into the fractal. The red dot (sometimes
       difficult to see) marks the point the camera will zoom in on. This is
       always in the center of the screen.
  'E'/[Scroll Wheel Down]: Zooms out of the fractal.
  [TAB]: Switches between the basic iteration count coloring algorithm and a
       triangle inequality average coloring algorithm.  The triangle inequality
       algorithm was based on an algorithm found in UltraFractal.
  'P': Print zoom factor to console as a floating point number.
  'Q': Quit the program.

  Note: translating and scaling are both scaled proportionally according to how
        far the user is zoomed into the fractal. This provides smooth motion.


Menu Notes:
  Values found in the user interface window are updated in real time and can be
  changed to any value.

  The files found in the "Textures" dropdown menu are populated from all files
  found in the ./colors/ directory. Users can add images to this directory to
  be used as coloring textures (only the first column of pixels will be used).

If you'd like to do any future work on this project, please contact me at
   pcorey@calpoly.edu AND petecorey@gmail.com
```