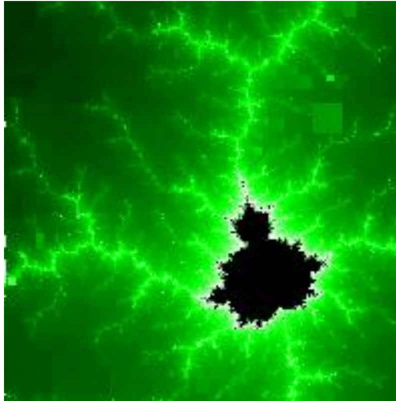
Interactive Auto-Refining Visualization of the Mandelbrot Set

Ben Ball

Introduction to Visualization

Rensselaer Polytechnic Institute

Computer Science



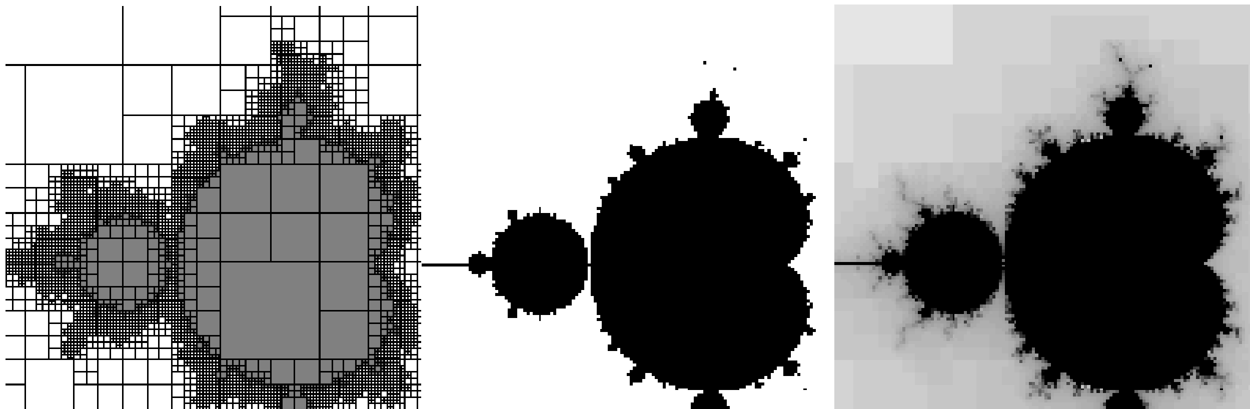
The Mandelbrot set is the set of all complex numbers which satisfy a specific condition. The condition is that the series $Z_{n+1} = {Z_n}^2 + c$ with $Z_0 = 0$ and c being the number itself, is bounded. When the set is plotted on a grid with one axis representing the real component and the other representing the imaginary component, the shape formed by numbers which are included in the set is a fractal. A fractal is a shape with finite area, but infinite perimeter. If you zoom in anywhere on the edge of the Mandelbrot set, you will find more and more detail, no matter how closely you look.

To calculate if a value is in the set, one simply iterates the equation given above. If the series ever exceeds 2, it will continue to grow infinitely, and therefore the value is not part of the set. However it is impossible to determine that a point is definitely in the set. Some points close to the set boundary may take thousands of iterations before they are determined to be outside the set. When multiplied by the one million or more pixels that need to be calculated to fill a modern computer screen, it can often take minutes to draw the set, especially at high zoom levels. My goal in this project was to attempt to reduce the calculation time to the point where it is possible to explore the set interactively.

The goal of interacting with the Mandelbrot set is twofold. The first purpose is an informal education in the nature of a fractal. While one may know for example that a fractal is bounded by a line which is continuous everywhere but straight nowhere, it is very difficult to visualize this. By freely exploring the shape it is possible to gain a better understanding of what a fractal is. The second purpose is to provide a medium for artistic expression. While the set itself is defined, the choice of what section to render and what colors to use provides a nearly infinite array of possibilities. This project makes it possible to discover, view, and share these images.

From a technical viewpoint, the most interesting piece of the program is the algorithm I use to calculate the boundaries of the set, and draw the image. The basic idea behind the algorithm is to only calculate the fine detail where it is actually needed. It starts off with a grid of 4 squares, and begins calculating the series for the point in the upper left corner of each square. If the series for one square diverges (it isn't in the set) but the series for an adjacent square hasn't diverged after a greater number of iterations, it determines that there is something interesting happening in those squares (they are close to the boundary). In this case the algorithm splits each of those squares into 4 smaller squares, and continues to calculate the series for any squares where it hasn't diverged. To conserve memory and computing power, only squares which are visible on the screen are computed, and squares are only split down to a specific size based on the zoom level. The state of the entire set is kept for the duration of the program, allowing very fast recalculation as the viewpoint changes. This however also leads to a very large memory footprint, which can make the program grow sluggish after prolonged use.

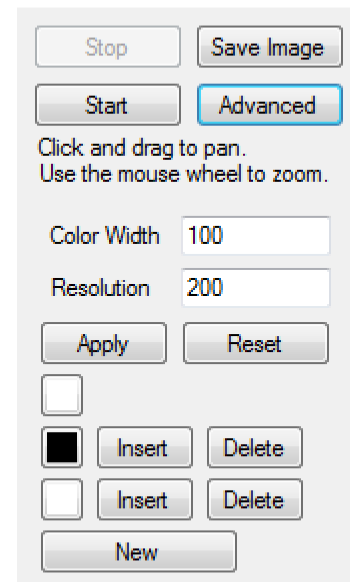
From an artistic perspective the procedure for determining a squares color is most intriguing. The number of iterations a square takes before diverging is an approximation of its distance from the set. Values range from 0 anywhere outside a circle of radius 2, up to infinity for points that are inside the set. The program maps these values to repeating bands of color. The bands are defined by a set of user selected colors, with the values in-between determined by a linear blend from one color to the next. The width of the color bands is determined by the user. In an attempt to keep the visual width of the bands the same at all zoom levels, I take the square root of the number of iterations, before converting it to a color.



When I first created the program, I visualized the set in a way that emphasized the approximation method used to calculate it (left image above). This resulted in images completely different from those created by other programs, and was also a useful tool for debugging my algorithm. The square borders cluttered the image however, taking the emphasis off of the set itself. I moved to a more traditional depiction of the set with black and white (center image above). Up to this point, any square which was not part of the set was left as white. When I started coloring the squares based on how many iterations they took to diverge, the image (left above) began to look very similar to many standard tools. The blocky pattern which reflects the underlying algorithm never completely disappears, however. While it would theoretically be possible to interpolate values between the calculated points

and make a smooth image, this would significantly increase the time required to draw the set, which is already the most time consuming part of the program.

The interface for the program is as simple as I could make it while giving the user all the options they need. They can start and stop the calculation, and save the image to a file. The viewpoint can be moved across the image by clicking and dragging, and the image can be zoomed in or out with the scroll wheel on the mouse. The other "advanced settings" allow the manipulation of the image colors and the size of square at which calculation stops.



When I received feedback from users of the program, they didn't always agree. For example, the zoom operation is performed slowly over about a second, to provide a sense of continuous flow instead of abrupt change. One user commented that they loved how this made the image appear almost three dimensional, like they were moving forward into the image as they zoomed in. Another user however became frustrated with what they interpreted as slow performance. While it might seem appropriate to add a new option for the user to select in every such case, I decided that too many options could have a worse effect on the program than limited freedom.

If I continue to work on this project in the future, the emphasis will most likely be on performance. While the program would be reasonably simple to parallelize, doing so would probably have very little effect. The majority of the computation time is spent on drawing the actual image to the screen, which can only be done by one thread. The second most limited resource is memory to store the set data in. If I remove the minimum square size limitation, the program will consume all of my systems 6gb of memory in under a minute. While I have no easy solution to either of these problems, a significant performance increase could probably be seen through small optimizations. While there are also several features I have thought of or that were requested by my users (like a more intuitive control for adjusting the colors) I think the biggest gain for the program as a whole would be to draw the image with a higher frame rate, smoothing the users interaction with the Mandelbrot set.