Using Design Patterns to Improve the Run-Time Efficiency of Real-Time Fractal Generation

Benjamin M. Dotte
Computer Science Department
University of Wisconsin-Eau Claire
Eau Claire, WI 54701
dottebm@uwec.edu

Daniel C. Julson
Computer Science Department
University of Wisconsin-Eau Claire
Eau Claire, WI 54701
julsondc@uwec.edu

Abstract

In today's world of pervasive computer graphics, fractals have become an important area of research. Fractals encompass a wide range of self-repeating patterns that are often recursive and iterative in nature. The computerized generation of fractal images requires a considerable amount of processing power for even modest-sized images. Run-time efficiency is therefore a paramount concern when designing a system that is capable of generating real-time fractal images. In the past, this run-time efficiency has often been achieved by compromising design robustness.

Design patterns are an exciting area of research in object-oriented design that have been shown to offer more flexible and robust system designs, usually at the cost of performance. In this paper, we will demonstrate that design patterns can be used to improve both the robustness of the design and the run-time performance of a real-time fractal generator.

Introduction

Fractals are self-similar, rough geometric shapes that occur in a plethora of natural and synthetic environments. Since the introduction of the term "fractal" by Benoit Mandelbrot in 1975 [3], this new system of geometry has impacted such diverse fields as physical chemistry, physiology, and fluid mechanics. A classic example is the fern, which has a seemingly infinite self-repeating pattern of leaf structures.

Fractals are used in many realms of the design world, including graphic displays of mountainous landscapes, coastlines, and flowers. Their intricacy attracts digital artists who display their vivid fractal work in online galleries. Image compression through the use of fractal formulas has also emerged as a major area of study [2].

Due to the heavy computational requirements that define fractal generation, none of the previously mentioned applications could be possible without the use of a computer. Like other computationally intensive applications, fractal generators often suffer from poor design to achieve optimal performance.

With the publication of Design Patterns: Elements of Reusable Object-Oriented Software [4], design patterns have emerged, in a catalogued form, as a way to structure object-oriented code in a responsibility-driven manner. Using objects that coordinate, structure, or service the system being developed, design patterns create more modular and flexible programs. Unfortunately, increasing levels of indirection inherent in this model of programming is thought to typically result in a decrease in system performance. Through the judicious use of the State, Strategy, and a variation on the Memoize design patterns, we have found that both performance and flexibility can be attained in a real-time fractal generator written in C#.

Background - Fractal Generation

For our fractal generator, we chose to model six fractal types, split into three categories. These were chosen from personal taste and to allow for variety. Within the Mandelbrot category of fractals, we included the Mandelbrot type and the Julia type. Representing Iterative Function Systems (IFS) were the Fern and Sierpinski's Triangle fractals. Finally, we included the Gingerbreadman and Pickover Popcorn types from the Orbital category of fractals.

Mandelbrot Category

The Mandelbrot and Julia fractal types share almost exactly the same formula, as shown below [7].

Figure 1a: Mandelbrot Equation

$$Z_0 = c$$
$$Z_{n+1} = Z_n^{\ 2} + c$$

Figure 1b: Julia Equation

$$Z_0 = c$$
$$Z_{n+1} = Z_n^{\ 2} + K$$

Note that *Z, c,* and *K* are complex numbers. The Mandelbrot fractal is plotted out by running the formula listed in figure 1a for every pixel on the screen. Each pixel, in turn, represents a location within the mathematical space the fractal is being calculated over. If the equation never reaches infinity at a given location for a set number of iterations, that point is said to belong to the set. If, on the other hand, the equation does reach infinity, a color can be assigned to that location based on how many iterations were processed before it reached infinity (technically this is assumed when $Z_n^{\ 2}$ has reached 4). This value is often indexed to a gradient or palette to create colorful representations of the set.

Whereas there is only one Mandelbrot set, an infinite number of Julia sets exist based on the value of its '*K*' constant from figure 1b. An interesting relationship exists between the Mandelbrot and Julia sets wherein the singular Mandelbrot set essentially acts as a "map" for all possible Julia sets. Locations in the Mandelbrot set with the most detail provide '*K*' values to the Julia set that are also the most interesting Julia sets. Much like the formulas, the code used to generate each of these fractal types is quite similar.

Figure 2: Mandelbrot and Julia Code

```
for (yPoint = 0; yPoint < height; yPoint++) {
  // Find the mathematical location based on the current pixel location
  yLocation = yMin + (yPoint * yIncrement);
  for (xPoint = 0; xPoint < width; xPoint++) {
    xLocation = xMin + (xPoint * xIncrement);
    zReal = 0d;      // JULIA: zReal = xLocation
    zImaginary = 0d; //        zImaginary = yLocation
    distSquared = 0d;
    // Iterate until the maximum number of iterations have been reached
    // or the current location is going to infinity
    for (curItr = 0; curItr < maxItr && distSquared < 4; curItr++) {
      realOld = zReal;
      zReal = zReal*zReal - zImaginary*zImaginary + xLocation;
      zImaginary = 2*realOld*zImaginary + yLocation;
      // JULIA: zReal = zReal*zReal - zImaginary*zImaginary + juliaX
      //        zImaginary = 2*realOld*zImaginary + juliaY
      distSquared = zReal*zReal + zImaginary*zImaginary;
    }
    // Set a point here at (xPoint, yPoint)
  }
}
```

Figure 2 illustrates the code used to generate the Mandelbrot fractal, with the variations needed to create the Julia fractal shown in bolded comments.

Iterative Function Systems (IFS) Category

Iterative Function Systems were first applied to the realm of fractals by Michael Barnsley through his Collage Theorem from 1984 [1]. The Fern and Sierpinki's Triangle are among the fractal types that fall into this category that we chose to model. The formula used to calculate these fractals is as follows [1]:

Figure 3: IFS Equation

$$F_i(x) = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

The values *a-f* and the probability of each function act as input values for each type of Iterative Function System. The matrices for the Fern and Sierpinski's Triangle are as follows:

Figure 4a: Fern Matrix

```
a    {    0,   .85,    .2, -.15
b         0,   .04, -.26,   .28
c         0, -.04,   .23,   .26
d       .16,   .85,   .22,   .24
e         0,     0,     0,     0
f         0,   1.6,   1.6,   .44
prob    .01,   .85,   .07,   .07 }
```

Figure 4b: Sierpinski's Triangle Matrix

```
a    {    .5,     .5,          .5
b          0,      0,           0
c          0,      0,           0
d         .5,     .5,          .5
e          0,      1,          .5
f          0,      0,   .8660254
prob    .333,   .333,        .334 }
```

The row listed as "prob" is the probability a random number will be entered into each of the 4 systems defined for the Fern fractal or each of the 3 systems for the Sierpinski's Triangle. The code used to generate both fractal types is the same; only the input matrices differ:

Figure 5: IFS Code

```
// maxIterations here is simply the number of pixels to plot
for (curIteration = 0; curIteration < maxIterations; curIteration++) {
  randNum = rand.NextDouble();
  // Choose a row of numbers based on the given probabilities
  if (randNum < (curProb = barnsley[6,0])) {
    index = 0;
  }
  else if (randNum < (curProb += barnsley[6,1])) {
    index = 1;
  }
  else if (randNum < (curProb += barnsley[6,2])) {
    index = 2;
  }
  else {
    index = 3;
  }

  // Apply the function to the selected index
  newX = barnsley[0,index] * curX + barnsley[1,index] * curY + barnsley[4,index];
  curY = barnsley[2,index] * curX + barnsley[3,index] * curY + barnsley[5,index];
  curX = newX;

  // Find the corresponding pixel location
  xPoint = (int)(((curX - xMin)/(xMax - xMin)) * width);
  yPoint = (int)((((curY * -1) - yMin)/(yMax - yMin)) * height);
  if (xPoint > 0d && xPoint < width && yPoint > 0d && yPoint < height) {
    // Set a point here at (xPoint, yPoint)
  }
}
```

Orbital Category

Orbital fractals vary more from one another than the Mandelbrot or IFS fractal types. Essentially, each successive point is calculated by applying some formula to the previously calculated value. The Gingerbreadman fractal type is one of the most basic:

Figure 6: Gingerbreadman Equation

$$x_n = 1 - y + |x_{n-1}|$$
$$y_n = x_{n-1}$$

The pattern created by this equation resembles that of a gingerbreadman. Similarly, the Pickover Popcorn fractal type resembles a set of evenly spaced popcorn kernels. Each kernel is, itself, an orbit, around which pixels are randomly generated and fed into a formula.

Figure 7: Pickover Popcorn Equation

$$\boxed{\begin{aligned} x_{n+1} &= x_n - h \bullet \sin(y_n + \tan(3 \bullet y_n)) \\ y_{n+1} &= y_n - h \bullet \sin(x_n + \tan(3 \bullet x_n)) \end{aligned}}$$

Interestingly, while the orbits are generated within 10-pixel wide "kernels", the resulting pixels are blown up to the proportions of the entire viewing area. Thus, the orbits are actually comprised of interwoven pixels from other orbits. This creates the odd effect where zooming into a particular location appears slightly different at each zoom level from the previous zoom level. This effect is illustrated with the following images:

Figure 8a: Pickover Popcorn Position 1

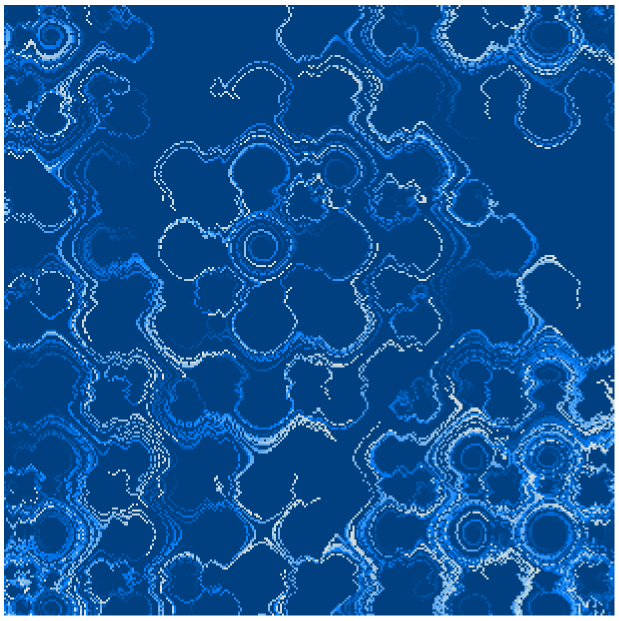


Figure 8b: Pickover Popcorn Position 2

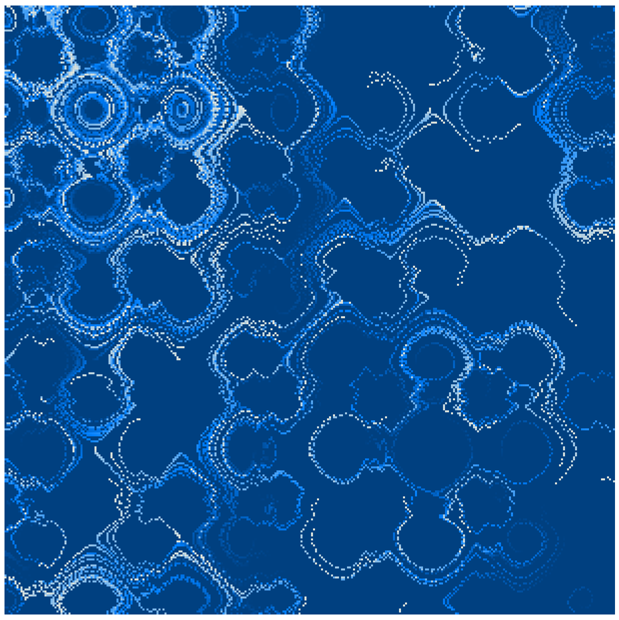


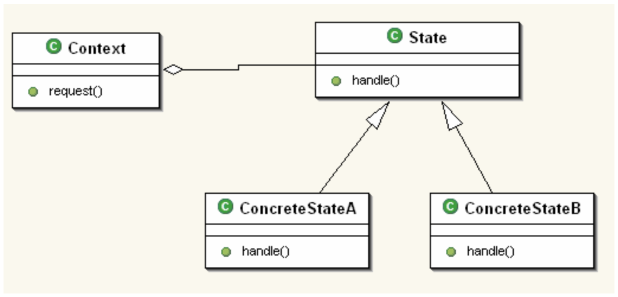
Figure 8a was zoomed into the bottom right corner to produce figure 8b; notice how the activity in the bottom right corner decreases while the activity in the center increases. This illustrates how, unlike all the other fractal types mentioned in this paper, this fractal actually changes dynamically when it is viewed in different places.

Our Design: Applicable Design Patterns

State Design Pattern

The State design pattern was designed to separate the functionality exhibited in an object that has multiple states into individual classes. When the state of the object changes, it appears to change its class and provide the desired behavior for the new state. Large, monolithic `switch` statements are commonly refactored into this pattern to improve code readability and maintainability. The structure of the classes that make up this design pattern is as follows [4]:

Figure 9: State Design Pattern

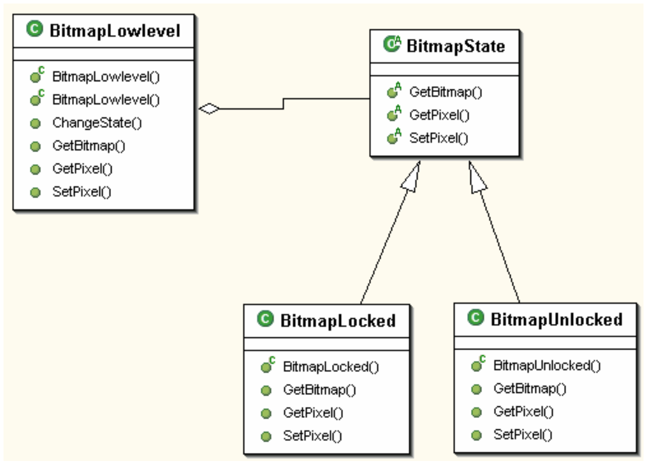


In order to display fractals in a graphical C# application window, we needed to write each individual pixel to a Bitmap object. The Bitmap class provided by C# offers two useful functions for our application, GetPixel() and SetPixel(). Every time a pixel is plotted, SetPixel() is used on the Bitmap object. The problem with these functions is that they are slow and were not intended for use on entire Bitmap objects [5].

The solution to this problem typically involves accessing the Bitmap data directly through unsafe pointer code. (Unsafe meaning, in a managed memory environment, directly accessing pointer memory.) A function is also provided for this purpose called LockBits(). In order to use LockBits(), the memory containing the Bitmap object must be locked using the Lock() function to avoid memory access issues. It must then be unlocked again using Unlock() after being used before the Bitmap can be displayed. If this locking and unlocking mechanism were used every time a pixel was set, in a display view of 500x500 pixels, 250,000 calls would be made to each function to generate one image. This would be a prohibitively expensive set of operations to perform for our real-time fractal generator.

To get around this problem, the code using a class capable of accessing a Bitmap directly would need to explicitly lock the Bitmap object before sending it GetPixel() and SetPixel() calls, and then unlock it again before the Bitmap was displayed. The problem with this lies in the fact that the client code should not have to change only because the means by which a Bitmap object is modified has changed. By using the State design pattern, the state transitions from locked to unlocked, and vice versa, can be made implicitly within the Bitmap manipulation class itself.

Figure 10: Bitmap Manipulation Classes



To use this class structure, the client code creates an instance of the BitmapLowlevel class. The BitmapLowlevel class, in turn, holds onto an instance of the BitmapState class. The GetBitmap(), GetPixel(), and SetPixel() functions correspond with the handle() function from the State design pattern class diagram. The concrete instance of BitmapState held onto by the BitmapLowlevel class changes between BitmapLocked and BitmapUnlocked depending on the operations being called by the client code.

Whenever a call to GetPixel() or SetPixel() is made, if the concrete instance of BitmapState is already BitmapLocked, the functions are simply applied immediately to the Bitmap object using unsafe pointer code; the Bitmap memory is continuously locked until a call to GetBitmap() is made. Once a call to GetBitmap() has been made, the concrete instance of BitmapState is changed to BitmapUnlocked, the Bitmap memory is unlocked, and the Bitmap object itself is returned for display.

The advantage to this approach is that the Bitmap memory need only be locked once for a series of GetPixel() or SetPixel() calls, rather than 250,000 times as in the previous example. The State design pattern makes the calls to Lock() and Unlock() implicit within the Bitmap manipulation classes.

Strategy Design Pattern

Considering the similarities between the Mandelbrot and Julia fractal types, it may make sense to create one generic algorithm using the Template Method design pattern, and subclass variant behavior. The Template Method is designed to describe the core functionality of an algorithm and allow subclasses to define functionality specific to their operation. The class structure of the Template Method is as follows [4]:

Figure 11: Template Method

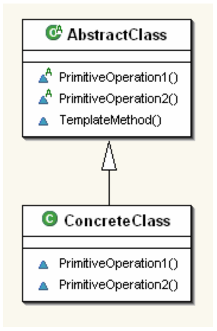
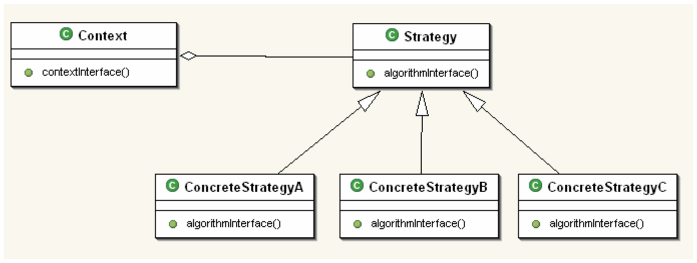


Figure 2, illustrating the operation of the Mandelbrot and Julia fractals, shows that only the values of *c* and *K* vary for each fractal type. The retrieval of these values could, then, be performed by running a subclass call for each variant equation from a superclass containing the core algorithm. The problem with this approach lies in the fact that a large quantity of additional subclass calls must be made if the variant behavior is extracted into subclass functions. At a standard zoom position and maximum iteration level on the Mandelbrot fractal, for example, the inner-most iteration loop is run 1.7 million times. The expense of all these additional calls may be too great to take advantage of the Template Method to solve this problem.

A better solution involves the use of the Strategy design pattern. Its class structure is as follows [4]:

Figure12: Strategy Design Pattern



Rather than represent the core functionality of the algorithm in a superclass, the Strategy design pattern chooses to have algorithms written out in their entirety in the subclasses. This avoids the need to make any subclass calls during the computation of the Mandelbrot and Julia fractal types.

Variation on the Memoize Design Pattern

Recall the Pickover Popcorn equation noted earlier:

Figure 13: Pickover Popcorn Equation

$$\boxed{\begin{aligned} x_{n+1} &= x_n - h \bullet \sin(y_n + \tan(3 \bullet y_n)) \\ y_{n+1} &= y_n - h \bullet \sin(x_n + \tan(3 \bullet x_n)) \end{aligned}}$$

For a display view of 500x500 pixels, 2,500 orbits would exist if spaced at 10 pixels vertically and horizontally, resulting in 250,000 sin and tan calculations each at 100 iterations per orbit. Trigonometric functions are expensive operations. We needed a way to store these values in advance so that we would not need to calculate them during the run-time operation of the program.

The Memoize design pattern describes a caching technique similar to our own approach. It stores the results of a function for each input value it receives in a table. Subsequent input values are checked against this table first to see if the value has already been calculated. If it has, the previously calculated result is returned. If it hasn't, the function is run, and the result is stored in the table [8].

Unfortunately, this design does not solve our problem because it relies on recurrent input values. Because our input values are so fine-grained and because they rarely, if ever, recur, the cached results would never be used.

As an alternative, we chose to pre-calculate a large quantity of values that fit within the desired range of input values prior to running any of the real-time code. These values were stored in a lookup table for later access. To determine the number of input values paired with results we needed to store, we relied on the fact that computer monitors are only capable of displaying a limited picture of the fractal at any given position. This effectively created a set of 500,000 "result buckets" that input values were rounded into for the sin and tan functions. We refer to the number of result buckets created (500,000 in this case) as the "granularity" of the lookup table. Specifically, we observed the point at which the Pickover Popcorn fractal looked identical using the lookup tables vs. calculating all the trigonometric functions in real-time.

The process of creating a lookup table does not change whether it is being made from the results of a sin function, a tan function, or any other kind of function. The next step was to find a way to create a generic class capable of creating a lookup table from any given function.

The answer to our problem came in the form of the "delegate" type built in to C#, which operates, to a limited extent, like higher order functions in the functional programming paradigm. Delegates function much like functors in C++. They create a pre-defined return value and set of parameters for which functions can be written and then passed as first-class objects [6]. We chose to write a class capable of accepting a Delegate-defined function, a granularity level, and a range to handle the pre-calculation of the trigonometric values:

Figure 14: Lookup Table Generator

```
// This defines the return value and parameters of the delegate
public delegate void LookupFunction(int val);
public class LookupTable {
    protected LookupFunction func;
    protected double[] table;
  public LookupTable(LookupFunction func) {
    this.func = func;
  }
  public void generateTable(int granularity, double minVal, double maxVal) {
    // Reset table to specified granularity
    table = new double[granularity]
    double curVal = minVal;
    double increment = (maxVal - minVal)/(double)granularity;

    for (int i = 0; i < granularity; i++)
    {
      // Run the function for every input value accepted in advance
      table[i] = func(curVal, maxVal);
      curVal += increment;
    }
  }
  public double[] getTable() {
    return table;
  }
}
```

The class that creates the Pickover Popcorn fractal need only pass this class the sin and tan functions, a granularity level, and the range over which the lookup tables must be generated. The generateTable() function is called initially to create the table, and again any time there is a change to the granularity level of the table or the desired input range (this does not occur in our program). When the fractal generation code needs to access these pre-generated result buckets, it simply retrieves the table itself through the getTable() function and accesses it as an integer-indexed array. This does create a limitation where the granularity level can never exceed the maximum size of an integer (32 bits in C#). Fortunately, for our problem, 500,000 buckets is more than sufficient to avoid any degradation of the image produced.

Results

Methodology

To ensure accurate test results, we chose to compare the unoptimized version of our program to the optimized version for each design pattern we discussed on three separate computers, two of which ran on Intel processors, and one of which ran on an AMD processor. Each test was run 110 times. We dropped the first 10 results and averaged the rest.

For the State design pattern, we timed the process of transferring our domain-level data structure containing the fractal information into a Bitmap object. The unoptimized version was timed transferring information via SetPixel() calls, whereas the optimized version utilized our State pattern-driven Bitmap manipulation class. Since the type of fractal being drawn does not impact the test, we simply chose to use the Mandelbrot fractal for this test. We started at the default position of the fractal and zoomed into the center 110 times, timing the process of creating the Bitmap object each time.

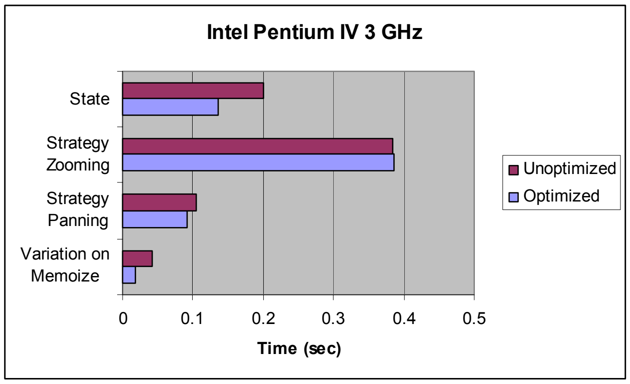
We also used the Mandelbrot fractal to time the Strategy design pattern optimization. Due to the fact that zooming into the fractal exhibited different results from panning on the top level of the fractal, we chose to include both zooming and panning in this comparison. (Panning means moving across the x and y axes, without changing the width or height of the viewing area.) Note that this is likely due to a difference in the proportion of calculations that take place between the initial starting point of the fractal, and a deeper zoom level.

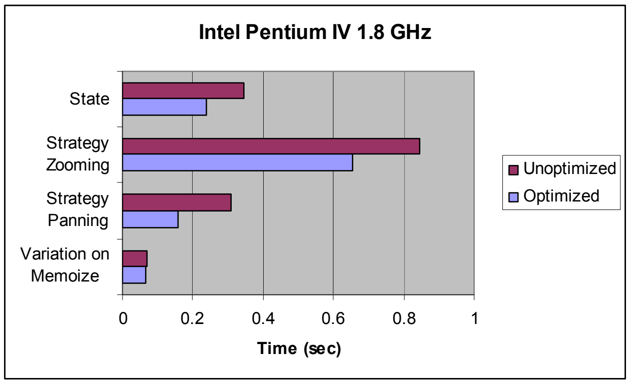
Since our variation on the Memoize pattern was applied only to the Pickover Popcorn fractal, we used this fractal to find the test results.

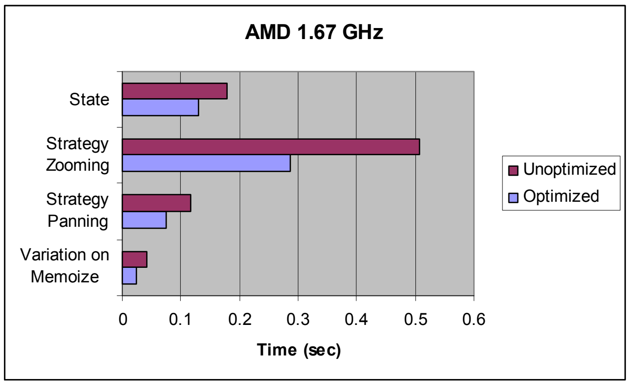
Outcome

Listed below are the test results from the three computer systems used for testing. Note that each bar represents the average amount of time it took for each of the 100 process runs to complete. Because time is the dependent variable, less is better.

Figures 15a-c: Time Trial Results







It is interesting to note that significant differences exist between the performance of an optimization on one machine versus that on another. The State design pattern seemed the most stable, on average affording a 35 – 45% increase in speed. The Strategy pattern, on the other hand, ranged from no improvement on the 3 GHz Pentium IV to almost double the performance on the AMD machine. But the greatest difference exists when using the variation on the Memoize pattern. The 1.8 GHz Pentium IV machine experienced very little improvement, while the 3 GHz Pentium IV machine showed over double the performance.

It is difficult to draw a definitive conclusion as to the reason for such inconsistent results. Note that the set of instructions run in each test case did not differ from one test run to the next. We even chose to re-run some of the tests multiple times to verify the soundness of our findings, and found the same numbers resulted each time. One theory we have come up with is that because the C# language is so new, it may not be fully taking advantage of the differences in architecture on each processor we tested. The compilation process in and of itself may also not be as refined as those created for more established languages. We are in the process of further investigating these ideas.

Analysis

Despite the inconsistencies we found in the performance of these optimizations, it is clear that none of them decreased performance in any of our tests, and in many cases, performance increased quite dramatically. The reason for this can be attributed to a single concept that ties all of the design patterns we used together; they are all utilizing some form of caching. Within the State Design Pattern section, the locked and unlocked states are in essence cached allowing for the retrieval of the proper state when needed. The Strategy Design Pattern outlays the caching of function values within the concrete classes. Finally, within the Variation of the Memoize Pattern, all of the formula values are pre-calculated and cached away in a look-up table to be used during the actual display rather than computing the calculations on each pixel during run-time.

Conclusion

In this report, we have shown that through the use of the State, Strategy, and a variation of the Memoize design patterns, both design robustness and increased performance can be achieved within a fractal generator. With the State Design Pattern, the locking and unlocking of the bitmap one time for all pixel manipulations incurred up to a 45% increase in performance. Using the Strategy Design Pattern allowed for the handling of the variant behavior to be done within the Mandelbrot and Julia subclasses. Expensive function calls were replaced in the parent class with values thus improving the performace by up to 85%. And finally, the variation of the Memoize Design Pattern permitted all of the computation time to be handled before run-time, resulting in as much as a 134% increase in performance.

References

[1] Bennetto, P. *Iterated Function Systems*. Retrieved March 13, 2004, from http://www.student.cs.uwaterloo.ca/~pbennett/fractals/.

[2] *Fractal*. Retrieved March 13, 2004, from http://en.wikipedia.org/wiki/Fractal.

[3] Fractals bring order to our world of chaos. (2000). *South China Morning Post*, 5. Retrieved March 7, 2004, from ProQuest database.

[4] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley.

[5] Gunnerson, E. (2001). *Unsafe Image Processing*. Retrieved March 13, 2004, from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp11152001.asp.

[6] Liberty, J. (2002). *Programming C# Second Edition*. Sebastopol, CA: O'Reilly & Associates, Inc.

[7] *The Mandelbrot Set*. Retrieved March 13, 2004, from http://www.students.tut.fi/~warp/Mandelbrot/.

[8] White, T. (2003). *Memoization in Java Using Dynamic Proxy Classes*. Retrieved March 13, 2004, from http://www.onjava.com/pub/a/onjava/2003/08/20/memoization.html?page=1.

Acknowledgements

We would like to acknowledge the assistance of Dr. Michael Wick in the preparation of these materials.