Name: ThankGod Ofurum

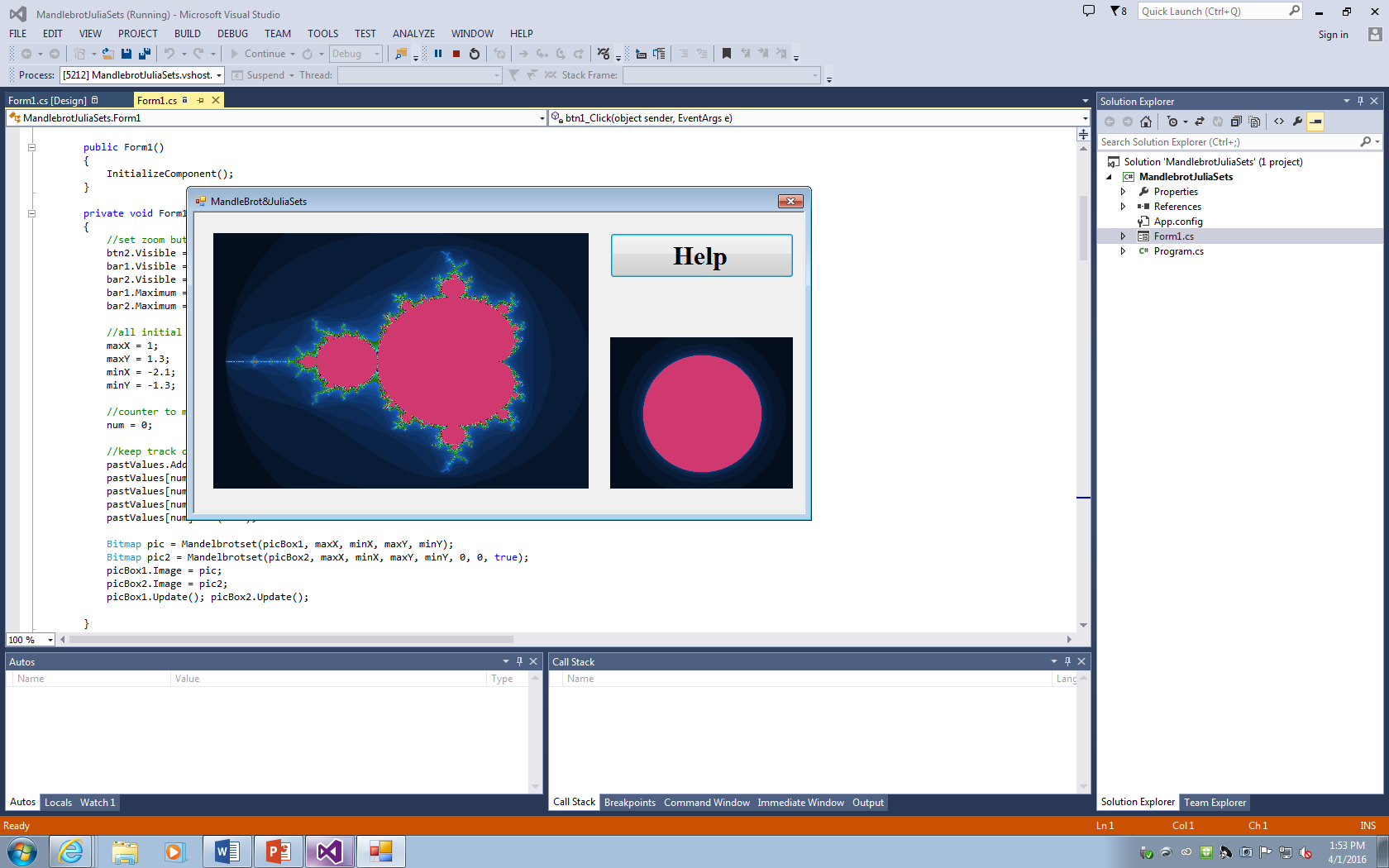
Professor: Francisco Neville

Class: CS 3350

Title: Mandelbrot & Julia sets

For this assignment we were required to create a windows form application which displayed the Mandelbrot and Julia set of the value Z square + Z, where Z refers to each point of the complex plane (1, 1.3i … -2.1, -1i in my case). In this write up I’ll explain exactly what the project is all about, the mechanics behind the construction of the Mandelbrot and Julia sets, and the method through which I implemented the project.

The Mandelbrot set is the set of all complex number, C, in which the value of Z does not exceed to the point of exponential growth, usually 2, and the Julia set is the complex number, constant C, such that the value of Z does not exceed to the point of exponential growth. For further verification, the Mandelbrot image is the result of the a composition of all possible Julia sets images, and the maximum number of iterations required before the cap Z value is reached depends on the individual designing the set. As previously stated, we were required to design this project in C# windows form app, and as for requirements, we had to implement zoom in and out functionality, pan left and right functionality and a help button, in order to assist user in understanding the dynamics of the program. We also were required to design an accompanying image of the Julia set for each point clicked on the Mandelbrot image, and below is a screenshot of my Mandelbrot and Julia application.



In order to design this application I had to understand how the Mandelbrot and Julia set were calculated and the relationship that existed between both sets. In order to do this I needed information concerning both sets. Therefore I began by making use of the half-implemented code provided generously by our professor, and upon understanding the structure behind the code, it became easier to observe its direction. So I began by initializing the Mandelbrot method in the “Form1 Load” method and provided the parameters for the method as maximum and minimum X and Y values. I then followed the code template provided almost exactly, except for the fact that I focused on painting each pixel of the image based on the count value of the iteration as it turned out colorful compared to the dull image with just black as the color for any Mandelbrot set, and I also made use of Z and Y zoom as a means of converting from the picture box coordinate plane to the complex plane. Below is the code that accomplished this task.

while ((count < Math.Pow(10, 3)) && (Zoomed == true))

{

double zx2 = Math.Pow(zx, 2); //for normalization

double zy2 = Math.Pow(zy, 2); //for normalization

if ((zx2 + zy2) > 4)

{

Zoomed = false;

}

zy = 2 \* (zx \* zy) + cy;

zx = (zx2 - zy2) + cx;

count += 1;

}

}

pic.SetPixel(x, y, Color.FromArgb((count % 128) \* 2, (count % 32) \* 7, (count % 16) \* 14));

cy += yzoom; // converted to local coordinates

}

cx += xzoom;

With this completed I had to turn my focus on how to implement zooming in and out of the windows form application. This proved to be quite tricky as in order to zoom in you’ll need to shrink the maximum and minimum X and Y values by a certain scale factor (I chose 5), and the challenging aspect of this was maintaining the relationship between the scale factor of the static picture box coordinates and that of the new complex plane. This I accomplished by simply normalizing the width and height of the picture box to the scale of the complex plane (I said simply but it was quite complicated). I then made use of this formula in scaling the point at which the picture box was double-clicked to the complex plane and factoring by the zoom value. Just below is the code that accomplished this task.

double zoomValue = 5;

double tempX = ((Convert.ToDouble(e.X) / Convert.ToDouble(picBox1.Width)) \* (maxX - minX)) + minX;

double tempY = ((Convert.ToDouble(e.Y) / Convert.ToDouble(picBox1.Height)) \* (maxY - minY)) + minY;

//zoom into specific location

maxX = (tempX + ((maxX - tempX) / zoomValue));

maxY = (tempY + ((maxY - tempY) / zoomValue));

minX = (tempX + ((minX - tempX) / zoomValue));

minY = (tempY + ((minY - tempY) / zoomValue));

With this done I had to turn my attention to the issue of zooming out, and this turned out to be less complicated than the issue of zooming in, as all I had to do was store the previous maximum and minimum values of X and Y in a list and iterate backwards in order to retrieve them. Below is the code for doing this.

//increment counter

num += 1;

//keep track of zoom values in order to revert back to previous zoom

pastValues.Add(new List<double>());

pastValues[num].Add(maxX);

pastValues[num].Add(maxY);

pastValues[num].Add(minX);

pastValues[num].Add(minY);

pastValues[num].Add(tempX);

pastValues[num].Add(tempY);

num -= 1;

double tempX = 0;

double tempY = 0;

//make sure not visible at first step

if (num <= 0) { btn2.Visible = false; bar1.Visible = false; bar2.Visible = false; }

else {tempX = pastValues[num][4]; tempY = pastValues[num][5]; }

maxX = pastValues[num][0];

maxY = pastValues[num][1];

minX = pastValues[num][2];

minY = pastValues[num][3];

The last feature left was panning horizontally and vertically, and this, though seemingly the least challenging, turned out to be the most challenging (partly due to lack of motivation and the desire to think thoroughly). This I eventually accomplished by normalizing the factor by which the bar was shifted from the image to the complex plane and adding this value to the already existing maximum and minimum values of either X or Y. This was the final, and perhaps most essential, feature which involved manipulating the image of the Mandelbrot set, and can be viewed below.

private void bar1\_Scroll(object sender, ScrollEventArgs e)

{

double zoomValue = 1;

double tempX = ((Convert.ToDouble(e.NewValue - e.OldValue) / Convert.ToDouble(picBox1.Width)) \* (initMaxX - initMinX));

changeCoordinates(tempX, 0, zoomValue);

}

private void bar2\_Scroll(object sender, ScrollEventArgs e)

{

double zoomValue = 1;

double tempY = ((Convert.ToDouble(e.NewValue - e.OldValue) / Convert.ToDouble(picBox1.Height)) \* (initMaxY - initMinY));

changeCoordinates(0, tempY, zoomValue);

}

All I had left to do after this was to implement the Julia set, and due to its similarity to the Mandelbrot set, I decided to make use of the same method in implementing the image. The two main differences being that the mouse click coordinates, XC and YC, were constant and the input parameters were the initial values of the complex plane.

Bitmap pic2 = Mandelbrotset(picBox2, initMaxX, initMinX, initMaxY, initMinY, tempX, tempY, true);

count += 1;

tempx = zx;

zx = (zx2 - zy2) + xc;

zy = 2 \* (tempx \* zy) + yc;

With all the design completed all that was left to do was to set everything in its proper position, and as observed above, that was the initial images of the Mandelbrot and Julia set. Below is the zoomed in and clicked snapshot of the application. With the greatest issue coming from the fact that it takes the application such a lengthy amount of time to accomplish its purpose.

