Collégial international Sainte-Anne

MATLAB Final Project

Fractals Project Datasheet

William Tremblay

Presented to
Florence Morin

For the course
Introduction to Programming

December 16, 2019

1 Overview of the Fractal Renderer

My fractal renderer is a sophisticated Mandelbrot and Julia set renderer supported any algebraic fractal that can be derived from a polynomial with real coefficients. Example output images can be found in the `render_examples` folder. It comes with this wide array of features:

- Optimized algorithms for computing Mandelbrot and Julia sets
- Implementation of real iteration number to yield smooth coloring
- Input of the algebraic fractal as a polynomial instead of a table
- Linear and Histogram coloring modes
- Changeable colormaps
- Real-time colormap modification through offset, multiplier and power bias operators
- Dynamic zooming and panning
- Quick Julia mode
- Reset All Button
- User input verification
- Multiple view options (Mandelbrot + Julia or either)
- Image exportation

2 From Mathematics to Code

Firstly, let us quickly define the Julia set $J(P)$. If P is any polynomial on the complex plane, then $J(P)$ is the set of all complex numbers c such that the sequence $z_{n\geq 0} : z_{n+1} = P(z_n), z_0 = c$ is bounded for all n. Mathematically,

$$J(P) = \{c \in \mathbb{C} : \exists L \mid |z_n| \leq L \ \forall \ n \geq 0\}.$$

The Mandelbrot set is defined as the connectedness locus of all Julia sets $J(z^2 + c)$ as c takes on values on the complex plane. In other words, it is the set of complex numbers c such that $J(z^2 + c)$ is connected, or equivalently, that $0 \in J(z^2 + c)$. In this fractals project, we will extend the definition of the Mandelbrot set to all polynomials. Define $M(P)$, the Mandelbrot set of a complex polynomial P, as the set of points c for which 0 is in the Julia set $J(P + c)$:

$$M(P) = \{c \in \mathbb{C} : 0 \in J(P + c)\}.$$

This gives a very simple algorithm for approximating if a point c is inside a Mandelbrot or Julia set, given a number of iterations N and an escape value L. Set z_0 to 0 if we wish to compute a Mandelbrot set and c otherwise. Then compute z_n with the polynomial P for Julia sets and $P + c$ for Mandelbrot sets until either $n = N$ or $|z_n| > L$. If n reached N, the sequence has stayed below L and we consider it to be inside the set. Otherwise, the sequence has risen above L and we consider the point to be outside. This is called the *escape value* algorithm.

To make images out of this, we can take the rectangle of the complex plane that we wish to render and split it into many equally distributed points, storing them in a matrix. We then iterate over each point and run the escape value algorithm. If we store the number of escape value iterations for each point in a matrix, we can also map different iteration values to colors and create a good looking image of the set.

3 Feature Explanations

3.1 Optimized Algorithms

To optimize the Mandelbrot rendering, I utilize three techniques depending on the polynomial used:

If $P = z^2$ (normal Mandelbrot set), I first check if the point is inside the main cardioid and disk of the Mandelbrot set algebraically. This saves many expensive iterations for low magnification images, reducing computation time by 80%. I then use a different multiplication technique to iterate a point in the escape value algorithm. Normally, one would have code that does $z := z^2 + c$ (using complex variables, slow in MATLAB) or $(x, y) := (x^2 - y^2 + c_x,\ y = 2xy + c_y)$ using real variables with $z = x + yi$. This takes at least 6 multiplications every loop when checking for the magnitude. To reduce this, we can store $R = x^2, I = y^2$ and $Z = (x + y)^2$. Then we can compute the new variables as such:

$$
\begin{aligned}
x &:= R - I + c_x = x^2 - y^2 + c_x \\
y &:= Z - (R + I) + c_y = (x + y)^2 - (x^2 + y^2) + c_y = 2xy + c_y \\
R &:= x^2 \\
I &:= y^2 \\
Z &:= (x + y)^2
\end{aligned}
$$

Since we already have x^2 and y^2, we don't have to compute any multiplication for checking the magnitude. This reduces the computation time by half (3 versus 6 multiplications). In total, this makes the z^2 algorithm run about 10 times faster.

If $P = z^n$, I use the mathematical fact that the Multibrot sets ($P = z^n$) and the surface enclosed by their lemniscates (curves around iteration levels) are simply connected. This means that they contain no holes. As such, if all points around a path in the complex plane have the same iteration level, then all points inside must also have the same iteration level or higher. As such, my code manages a queue of rectangles, which originally has the entire view rectangle inside. It picks the first rectangle in the queue and calculate all points on its boundary. If they have the same iteration level, it fills all pixels inside with that iteration count. Otherwise, it splits it into four smaller rectangles and adds them to the queue, until no rectangles are left. This can be a big performance increase when many points have the same iteration value. However, this does not allow smooth coloring.

3.2 Smooth coloring

When not using the algorithm mentioned in the previous paragraph, my code will replace the iteration 'bars' from the image with continuous color change. This is achieved by using an approximation to the potential function of a Julia set which assigns a real number $\nu(z)$ to every point on the plane:

$$\nu(z) = n + 1 - \log_d \frac{\log |z_n|}{\log L}.$$

In this formula, L is the escape value, n is the iteration count before $|z_n|$ become larger than L and d is the degree of the polynomial for the set. My code stores this value instead of the iteration count and calculates the RGB matrix `colorMat` for the integer part `iPart = floor(itMat)` of the iteration using the `ind2rgb` function. Then, if the fractional part `fPart = itMat - iPart` is non zero, it will use linear interpolation of the colors: `colorMat = (1-fPart).*colorMat + fPart.*ind2rgb(iPart+1,cMap)`.

3.3 Direct Polynomial Input

Instead of using a table to input the coefficients of a polynomial, the program allows the user to directly input a univariate (one variable) polynomial inside a text box. I use the `Callback` event of the text box to call my `parsePolynomial` function. This function uses regular expressions to check the format of the polynomial and split it into monomials. It iterates over these and creates two vectors, one containing the exponents and one the coefficients. Finally, it finds the largest exponent (degree of the polynomial), creates a row vector of that length and accumulates the coefficients of each respective degree.

3.4 Linear/Histogram Color Mode

My code supports two coloring modes: Linear and histogram. The linear mode is the standard coloring mode of assigning a color of the colormap to every iteration count linearly. The histogram mode instead assigns colors based on the iteration count distribution. Thus, if there are many pixels with a certain iteration count, they will get a larger percentage of the color. This has the advantage that is does not depend on the total number of iterations. To calculate it, I use a `for` loop on every pixel of the image and store the number of pixels per iteration in a vector (`itArea`). Then I use another `for` loop to calculate the partial sums of `itArea` and store them in a vector called `itMap`. Finally, `itMap` is divided by the total sum of iteration counts, to get a value between 0 and 1. I can then multiply this value by the size of the colormap create a new colormap with these modified indices.

3.5 Changeable Colormaps

This is a fairly simple feature. I made a drop down menu of all continuous Matlab colormaps and added code to its callback to recalculate the colors of the image whenever this value is changed. The code uses `func2str` to find the associated function (e.g. `jet` or `hot`) and uses it to generate the base colormap that will be modified according to the procedures below.

3.6 Real-time Colormap Modification

The program implements three types of colormap modifications: the multiplier m (changing the frequency of the colormap), the offset o (phase shifting the colormap) and the bias b (making extreme color values stand out more). Assuming continuous colormap indices on $[0, 1)$, here is the full equation for the new calculated map:

$$f(c) = \{m \cdot c^{(b-1)/b} + o\}.$$

Here, $\{x\}$ denotes the fractional part of x. Implementing this in code required to save the current colormap as a matrix and use `ind2rgb` to apply it to the iteration data and get a RGB color matrix. This removes the need for recalculating the iterations or the colormap every time the other changes. I also had a problem with the `CallbackFcn` for sliders only firing when the user releases the slider. Since my color rendering is done real-time, I had to use `addlistener` on the `Value` property of the slider objects to update the colormap as the user is dragging the slider.

3.7 Dynamic Zooming / Panning

This was fairly hard to implement programatically. Newer versions of MATLAB support interactivity in `axes` objects, but they do not provide a callback when the view changes. As such, I had to hook the figure's `WindowButton` events to the same function which allows to track mouse press, release, and movement on the figure level. From there the code gets the current position inside the `axes` with the `CurrentPoint` property, and checks if the mouse was pressed inside. If so, it stores the last held position inside the `UserData` field of the object. This allows the program to calculate the change in mouse position and adjust the `axes`'s `XLim` and `YLim` properties accordingly. Zoom is handled similarly; the `WindowScrollWheel` event gets fired when the user scrolls, and if the mouse pointer is inside the graph window the zoom value is multiplied or divided by 1.1 depending on the scrolling direction.

3.8 Quick Julia Mode

The Quick Julia mode allows the user to quickly render a particular Julia set by clicking on a point in the associated Mandelbrot set. Programmatically, this uses the Mandelbrot `axes` object's `ButtonDownFcn` to detect mouse clicks. The variable `eventdata.Button` is then checked to be equal to 3 (right click). If this is the case, the `CurrentPoint` property of the Mandelbrot's axes is read and the Julia point is set to this value. The rendering subroutine `renderSub` is then called with arguments `(0, 1, handles)` to re-render only the Julia set to avoid unnecessary lag.

3.9 Reset All Button

This button resets all settings to their default value. To achieve this, I use the `UserData` property of GUI objects to store default values and other information. For text boxes, for example, it is a cell array where the first value is the last valid input string, the second the default input string and the third the default input value. The `Value` property is then used to store the last valid numeric value. This makes the reset code very simple: Iterate through all fields of the `handles` structure, and set their `String` or `Value` property (depending on the UI element's `Style`) to `UserData{1}`.

3.10 User Input Verification

This was a necessary feature to avoid bugs and make the code error-proof. I wanted to make an input field red if the user entered an incorrect value, and reset it to the last correct one automatically if they ignores the warning. This required creating a custom function `updateTextBox` that is run by the callback of very edit box object. The function takes the text box object, a min/max interval and an optional custom validation function handle. It checks if the user pressed enter or lost input focus by checking the `CurrentCharacter` property of the figure. If they pressed enter (VK code `13`), it will make the field red to warn them about the incorrect input. If they left the text box (lost focus), the value will be reset to the last correct one (`UserData{1}`).

3.11 View Options

The view (render) options of the program allow one to decide if he wants to render both the Mandelbrot and Julia sets or only one of them. This is achieved using two checkboxes, `Render Mandelbrot` and `Render Julia`. When any of them is changed, the `resizeAxes` function is called. This is split into three case depending on the values of the checkboxes. When both are activated, it sets the graphs to their original `Position` stored in `handles.origMandelPos` and `handles.origJuliaPos`. If onle one is active, it sets its position to the lower left corner of `handles.origMandelPos` and the top right corner of `handles.origJuliaPos` so the graph takes up all the screen.

3.12 Image Exportation

This is implemented as a simple button. When pressed, it will check which axes (Mandelbrot/Julia) are on using the value of the `Render Mandelbrot` and `Render Julia` checkboxes. For each of them, it will then use the `uiputfile` function to open a file save dialog. If the `file` output is not zero, the program will use `imwrite` with the path from `uiputfile` and the current render's `CData` property.