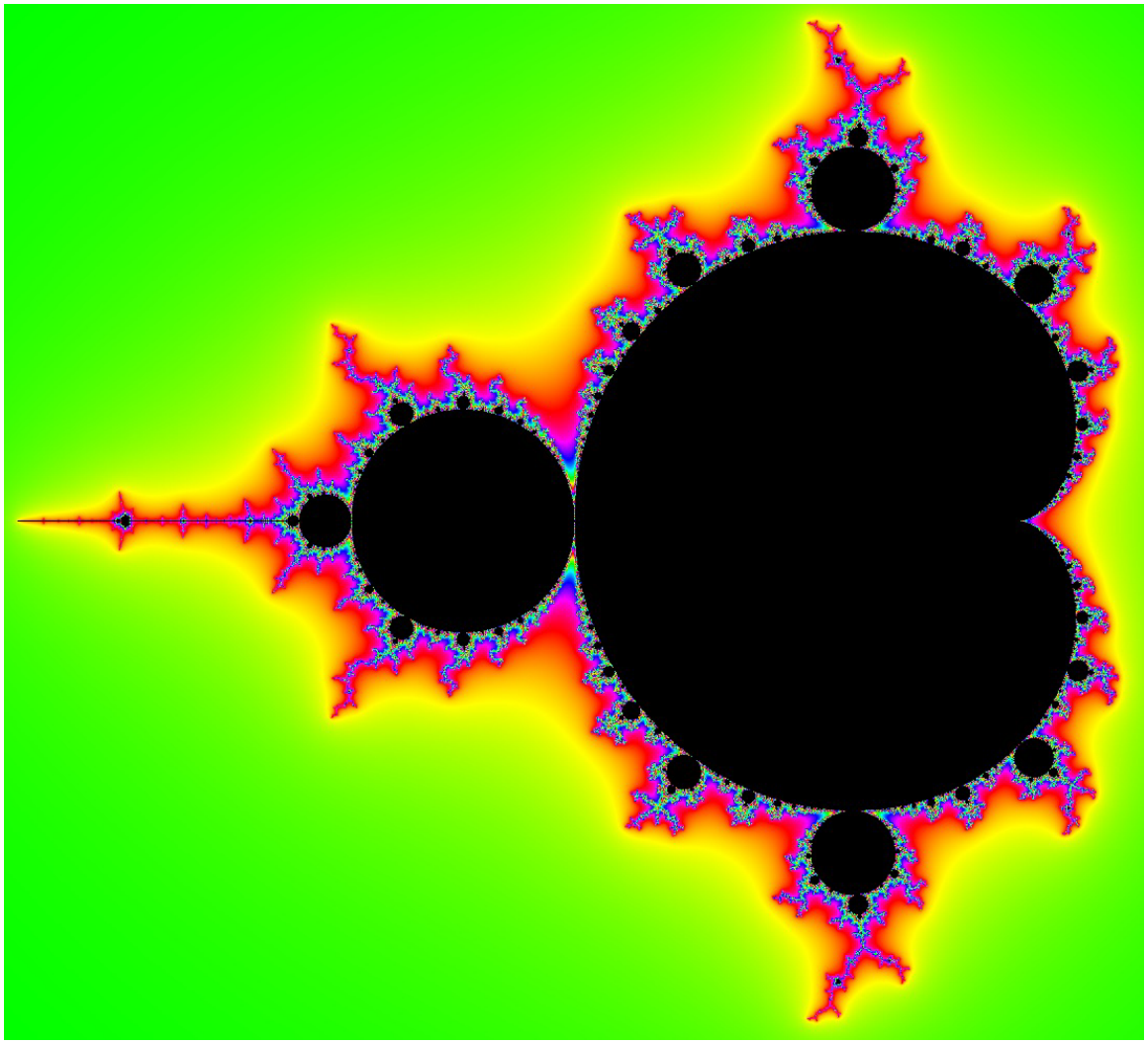
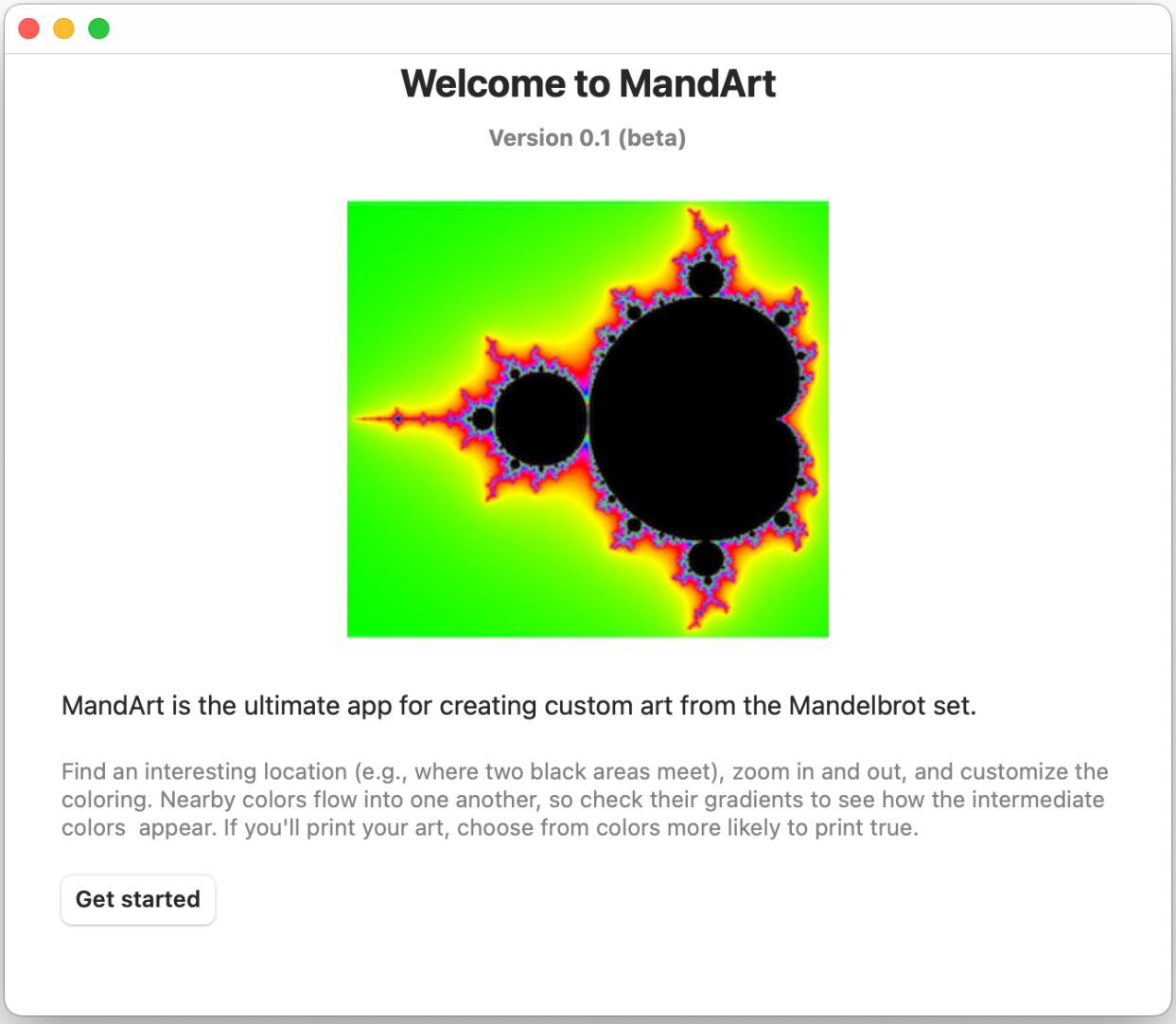
Computing the Mandelbrot set

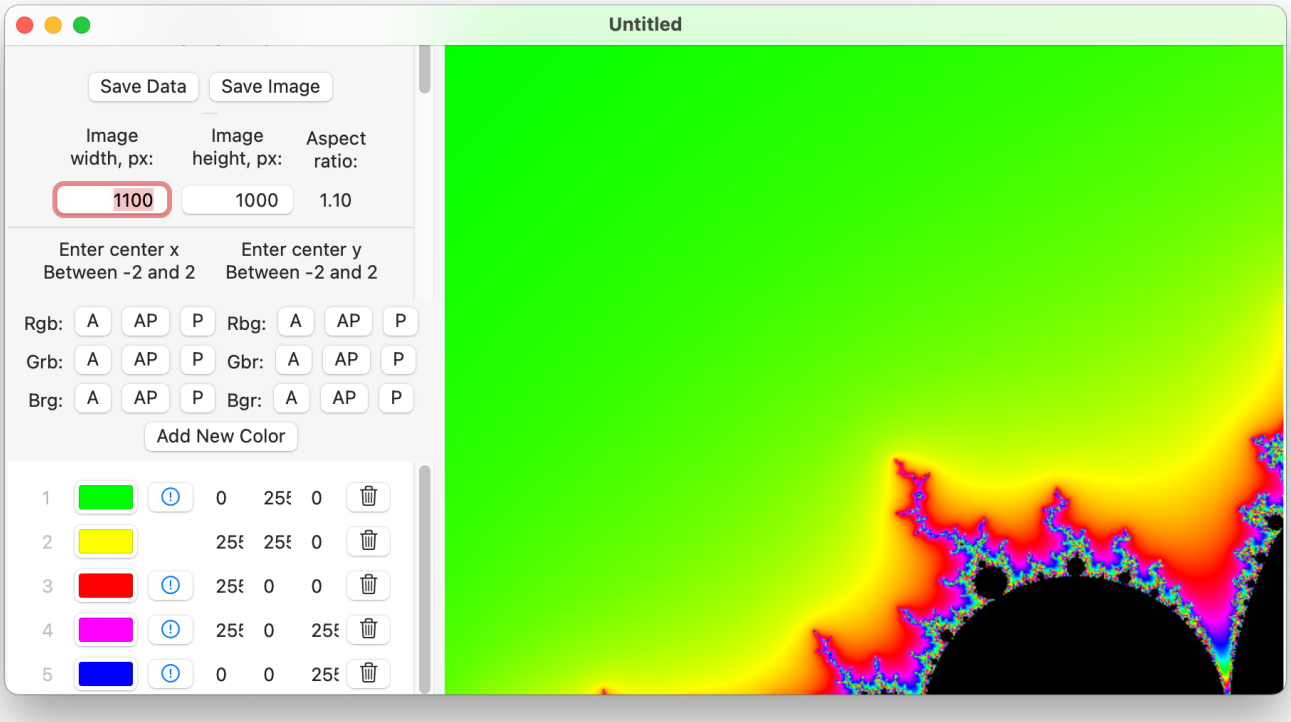
This is an art app, not a math app. It was specifically developed to generate pictures good enough to be printed, framed, and hung. You can completely ignore the math behind it, but it would help to have an idea of what's going on. So, we'll put all of the math at the end. This idea had to wait for modern computers to become available since the process requires a very great number of calculations. The app is written in the Swift computer language and uses SwiftUI for the interface.



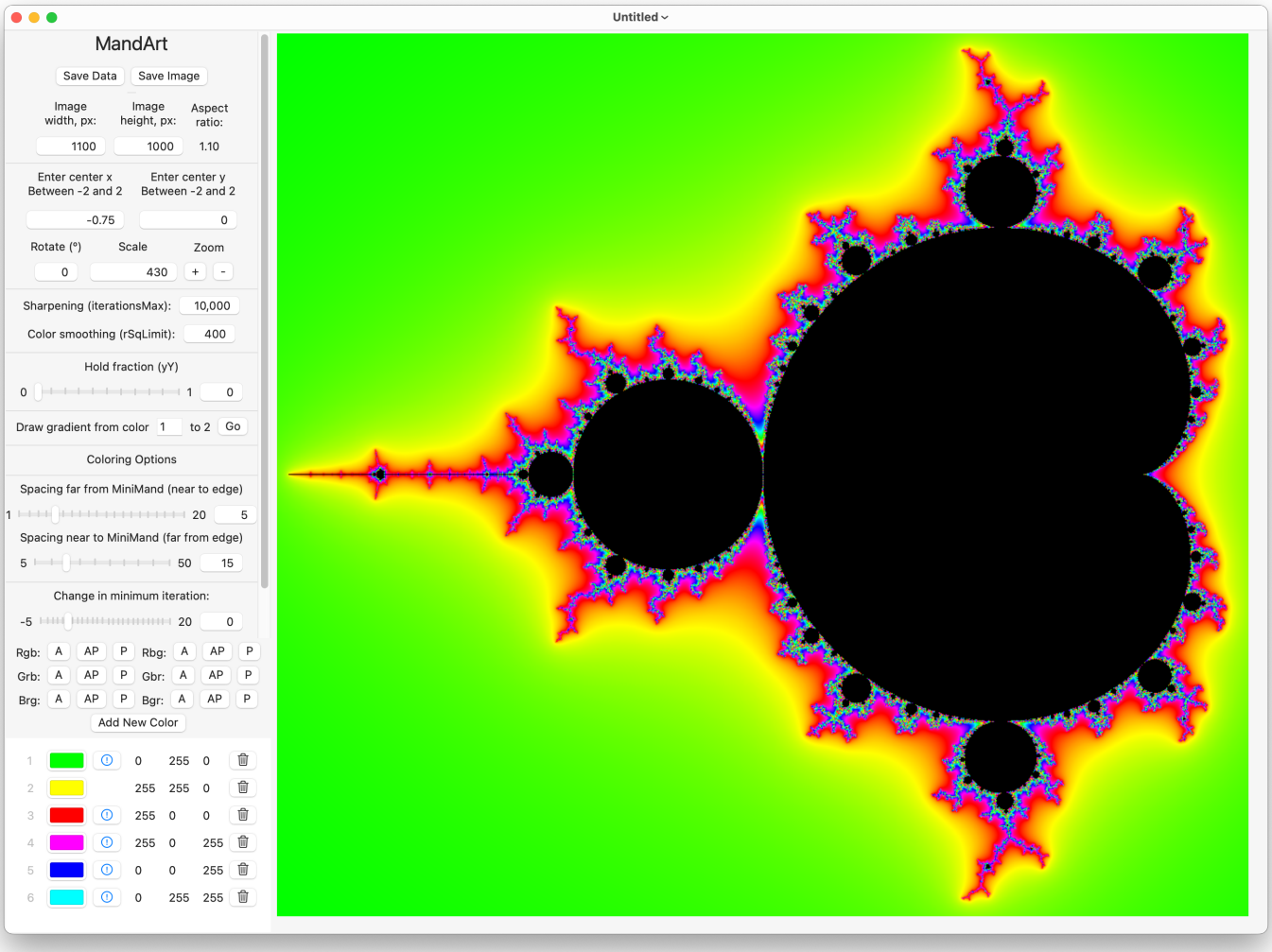
The above picture shows us the region of the x-y plane where the action will take place. The black area is the major part of the Mandelbrot set, but around it is an infinite number of smaller, distorted versions of that part, which we can't see unless we zoom in. Since we will keep referring to these objects, we'll call the black area in the picture the **Mandelbrot**. The other versions of it we'll call **Mini-Mands**.



This picture shows the opening window in the app. Click on the button to start.



This shows the window after opening. The window isn't big enough to show the whole picture, so resize it by drawing the lower-right corner until it shows the whole picture.



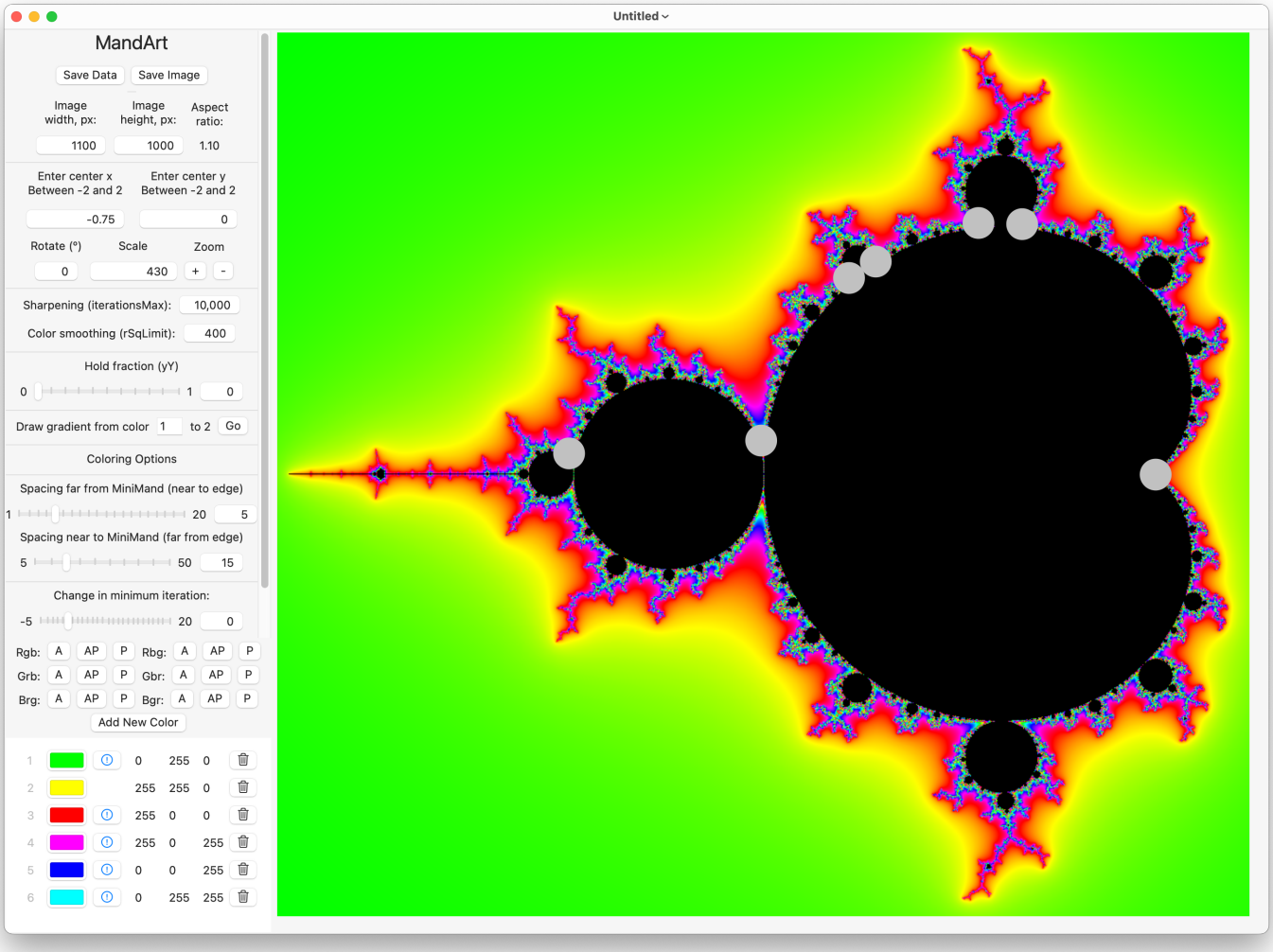
We'll use this picture to explain centering, dragging, and zooming. If you click anywhere in the right-hand picture, the program will present a new image, centered on that location. If you press and hold the mouse button, you can drag the image. It will take a few seconds to recalculate and show it in the new location. You can zoom in or out by a factor of two by hitting the "+" or "-" button. You can also change the magnification by changing the **scale** value.

When you zoom in, it is like looking through a microscope. You can also think of it as expanding the image. If you zoom in as far as the 15 significant places of a Swift variable will allow, it is like moving a sheet of paper the size of your screen around in an area the size of the solar system. You should have no trouble finding a picture that no one has ever seen. In fact, if you don't record that location, you'll probably never be able to find your way back.

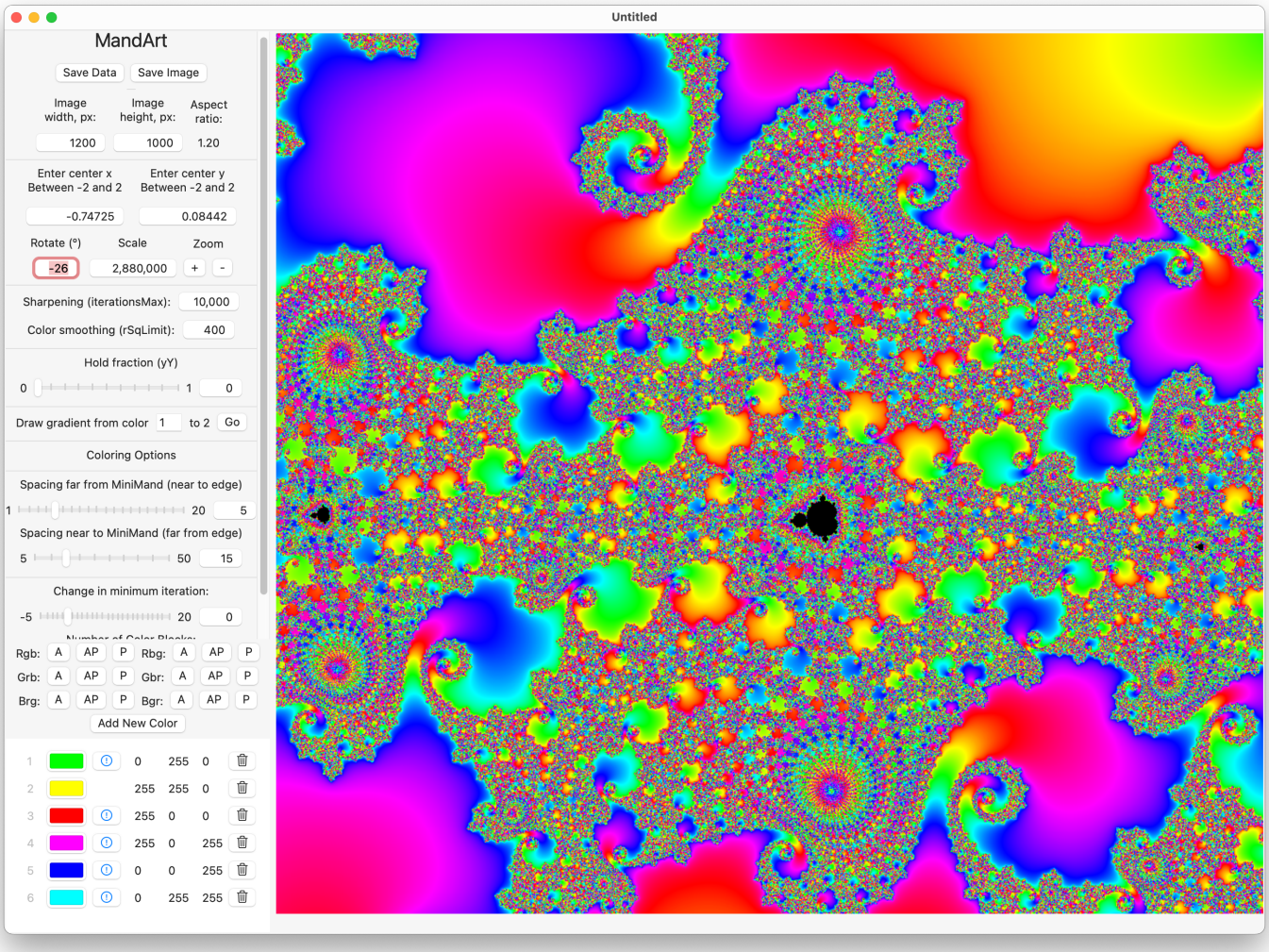
A number of variables are listed in the green area. You can change any of those values and hit **tab**, **return**, or **enter** to use those new values. Sometime, you may have to hit return or enter twice. The screen goes blank when you start to enter any values, otherwise the program will update the image when each digit is entered. This is a problem that SwiftUI will solve, hopefully. This lets you fine-tune the image or input a set of values that you found in another reference.

The next set of variables relates to coloring the image. Default values were chosen so that the initial image would show up colored. In order to reduce the number of colors that need to be entered, the program will cycle through the defined colors as many times as necessary. The region between each pair of numbers we call a block of colors. The program will make a smooth gradient between each pair of colors. To emphasize the few colors that we define over

the colors the program puts into the gradient, a slider is provided to define the fraction of a block of colors that uses the defined color before starting the gradient. If a value near 1 is chosen, the blocks of colors will show up as solid bands.



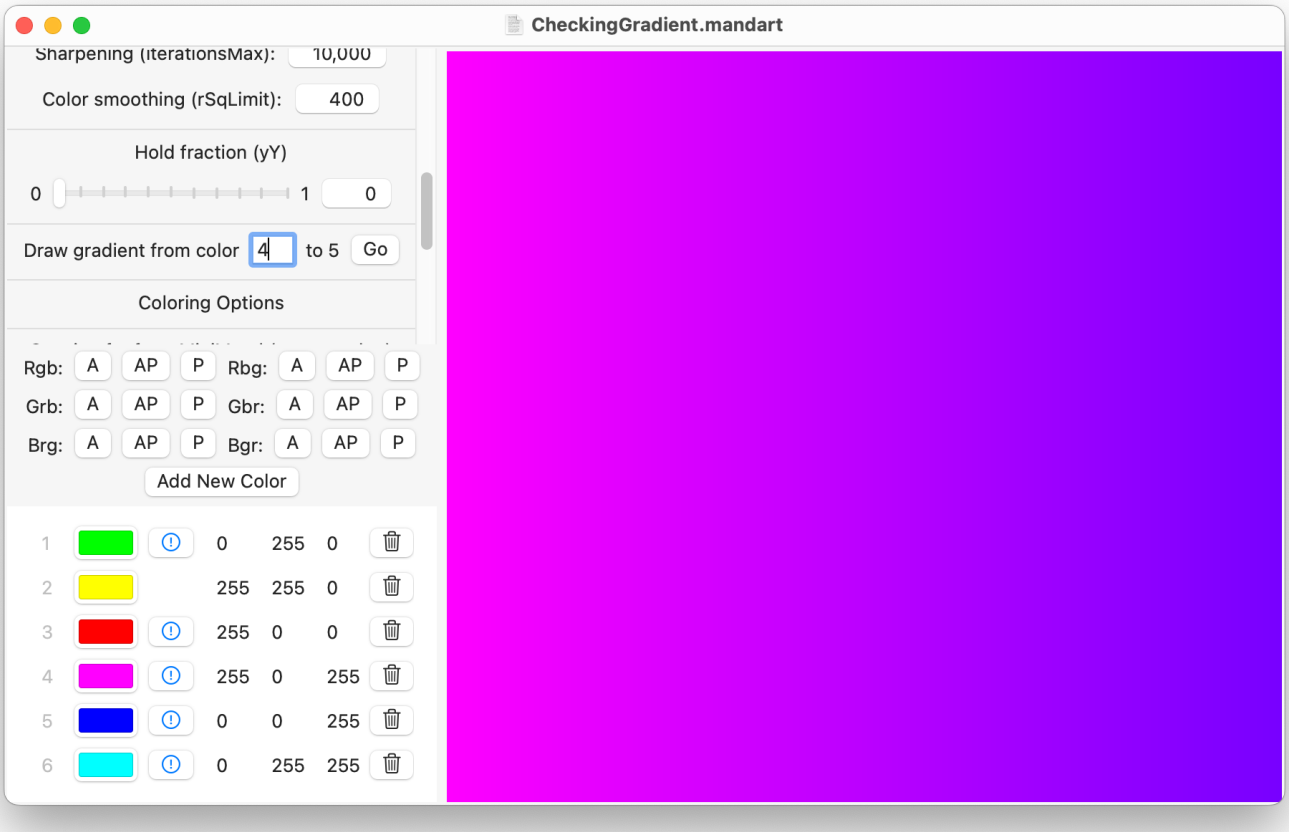
This shows typical areas where you might look for interesting images. They seem to be in areas that are near where two black areas come together.



Here we show an image that has been moved to a new center, rotated, and zoomed in by a factor of about 2,880,000/430 = 6,698. There are some obvious paths that seem to flow from the edge of the image into the interior. These paths can only end on the Mandelbot figure or on one of the Mini-Mands. However, the Mini-Mand may be so small that the we won't ever be able to see it. If we keep zooming in, eventually the 15-digit significant places of a number in Swift will just start to show big blobs of color. The locations with the fewest iterations occur outside of the image, so the program ignores them. If we want to move the initial color in the image into the image, we can enter a value for **Change in minimum iteration**. If we enter a negative value, we'll get a white area in the image, indicating where that minimum value is.

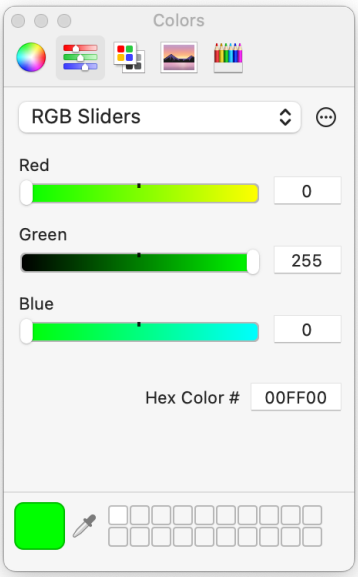
We typically use a large value for the maximum number of iterations and a small number for the number of blocks of color, so we use an exponential relation between the two. The spacing between colors near the edges of the image, which is far from the Mini-Mand destination, is set by one variable and the spacing near the Mini-Mand destination is set by another. Only trial and error and artistic preference will determine the best values. Likewise for the number of defined colors and the number of blocks of color.

If we define only a few colors, the gradients joining a pair of colors may go through regions of color that don't look good. To see what a gradient between two defined colors looks like, input a number for the left-hand number. The program will select the right-hand number. Hit "Go."

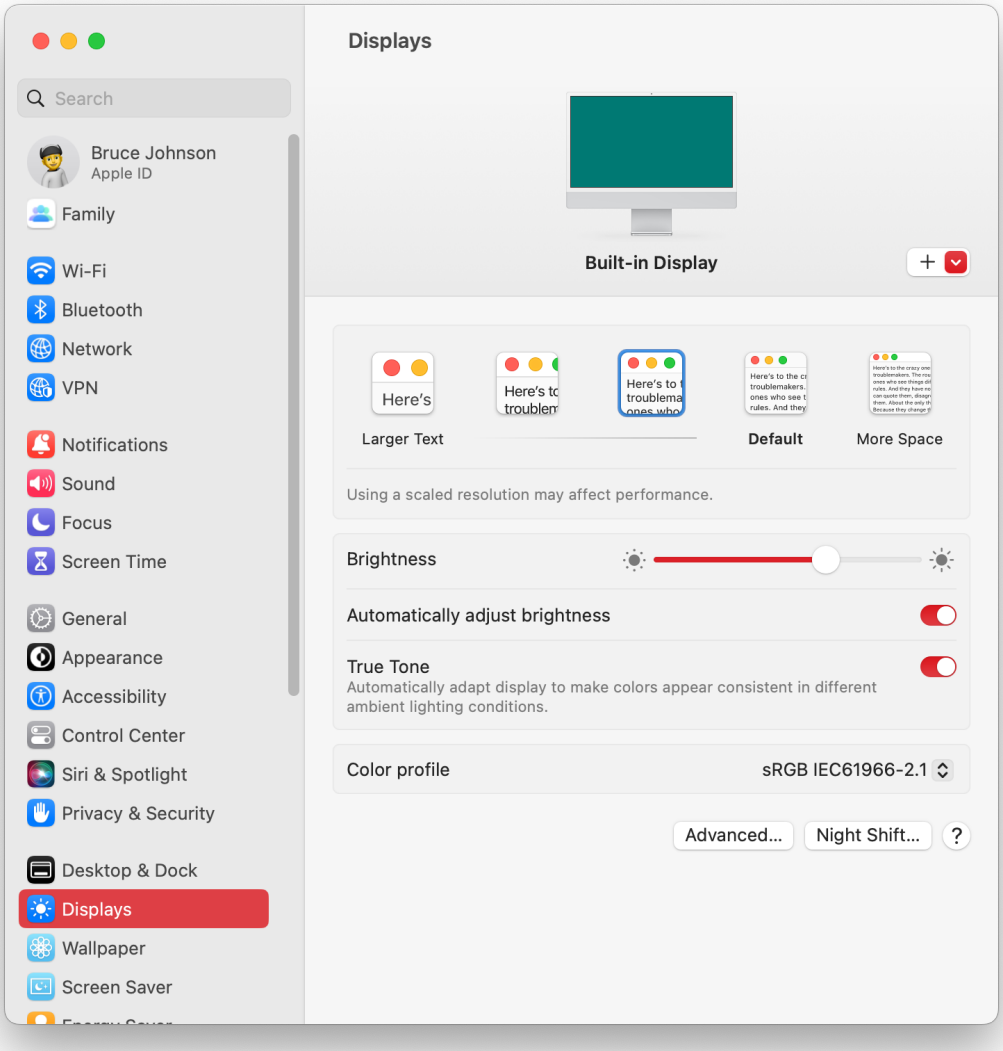


After that, just enter a different number to see that gradient. If the gradient shows a region of color you don't like, you can change one or both colors or you can add an intermediate color. We'll come back to finding an intermediate color after we discuss coloring.

The colors can be defined by inputting the (RGB) values or by clicking on a color to get the Color Picker. The Color Picker has the usual Color Wheel, Color Sliders, Color Palettes, Image Palettes, and Pencils. It also has an **eyedropper** tool. This tool can select any color visible, in the app or not. Click the eyedropper, and then click on any color on your screen to select that color.

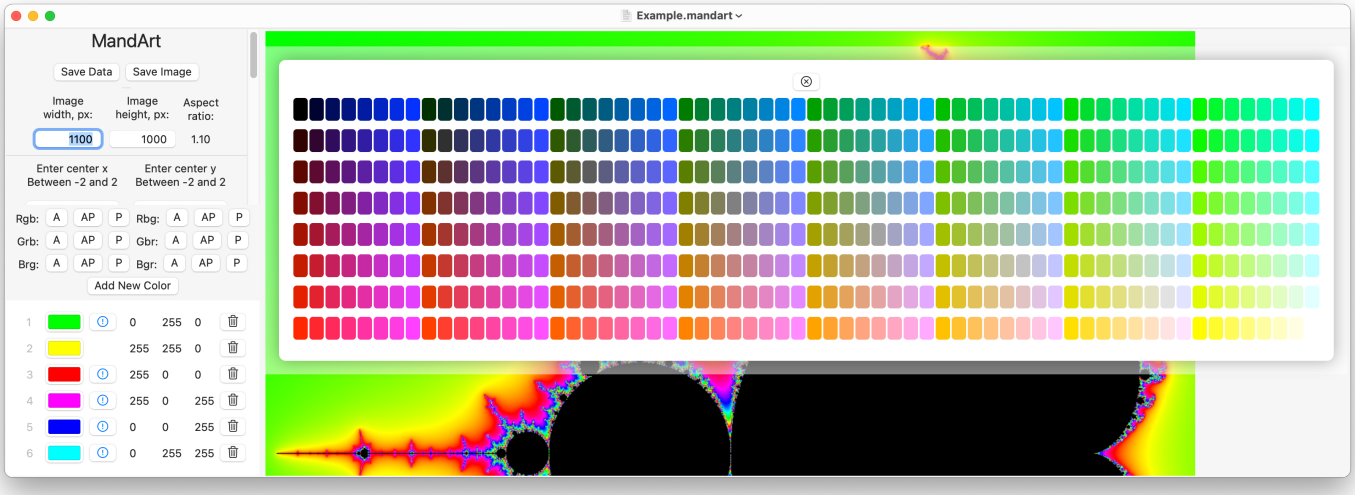


To get the eyedropper to work properly, you may need to adjust your Color Profile. Open System Settings / Displays / Color Profile and, on the drop-down, select the sRGB option as shown below.

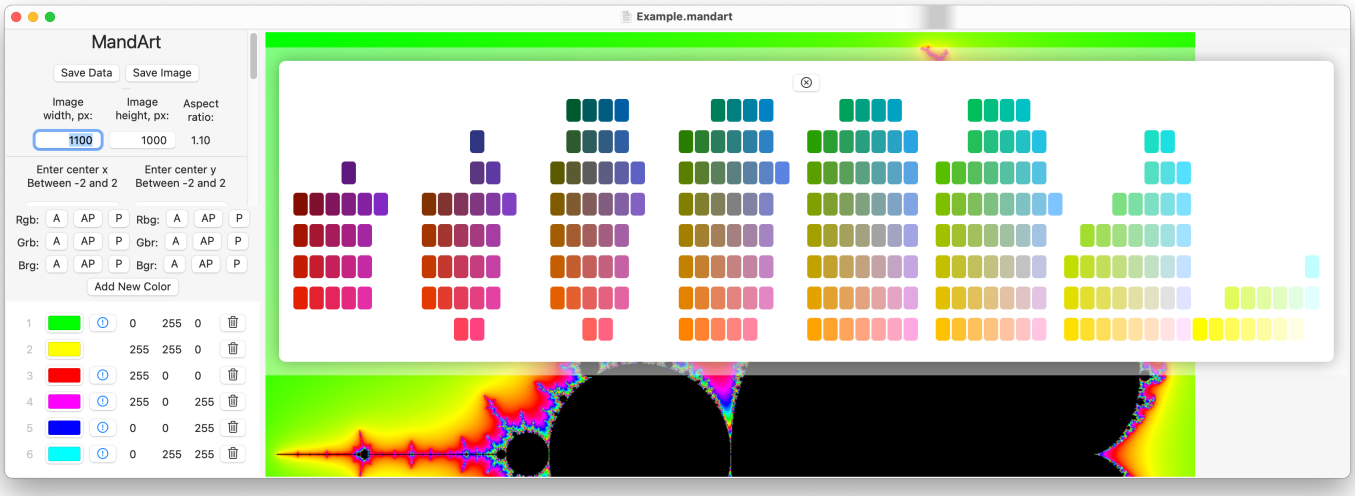


A selection of 512 colors that look good on the screen is available. These colors only use R, G, and B values of 0, 36, 73, 109, 146, 182, 219, and 255, but it is difficult to distinguish between adjacent colors, so they may be adequate. As mentioned, this app was designed to make a frameable picture. Unfortunately, printers can't reproduce the range of colors available on the screen. So, a set of 292 colors that should print well is also available.

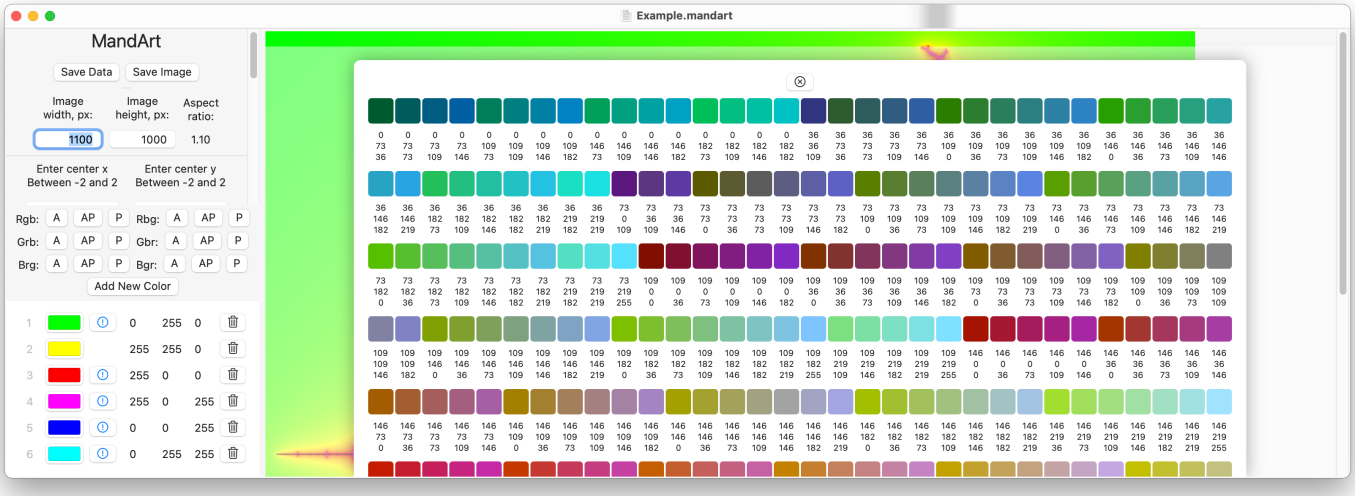
The button **A** will display all 512 colors.



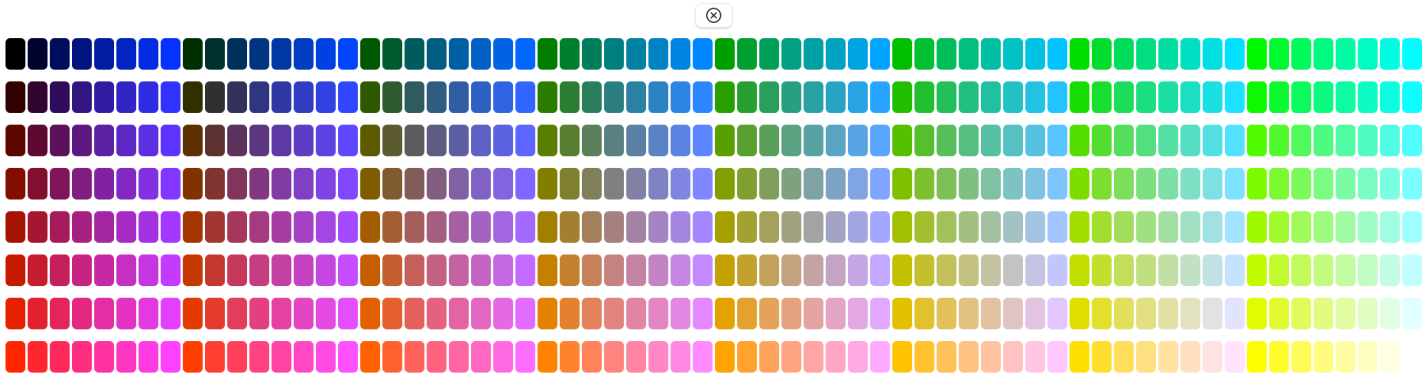
The button **AP** displays the 292 colors in the format of the screen colors, so it is obvious that many of the best colors are missing.

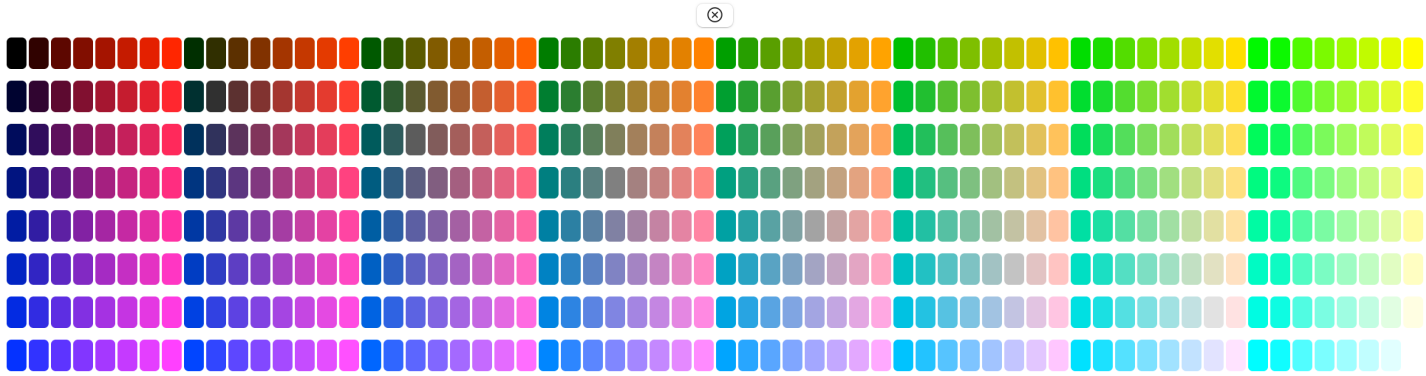


The button **P** will show the 292 colors, but not in the format of the screen colors. They also show the RGB values.

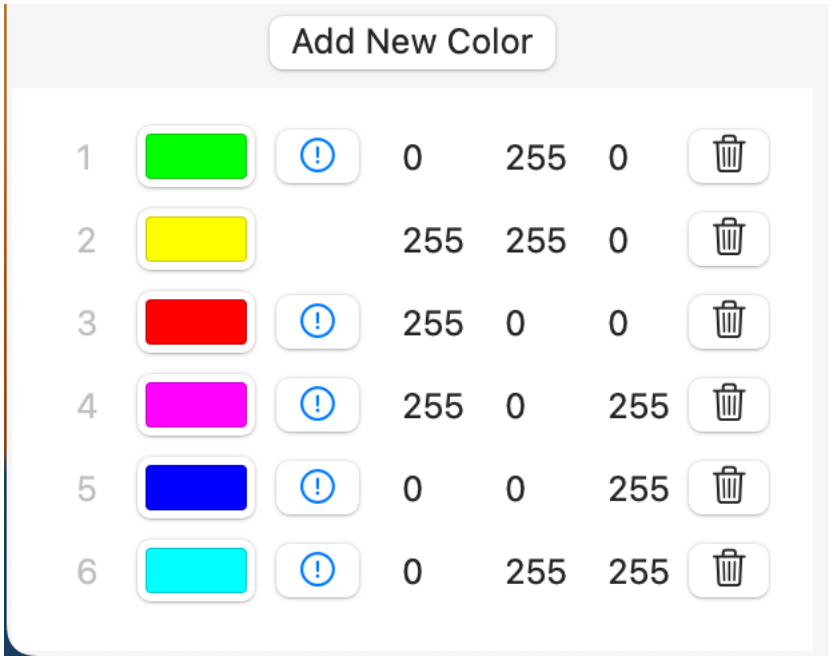


The **A**, **AP**, and **P** colors can be shown in six different orders (Rgb, Rbg, Grb, Gbr, Brg, and Bgr), but the upper-left is always black and the lower-right is always white. The different orders just make it easier to see the color groupings. The following pictures show the **A** colors in Rgb and Bgr orders.





The pictures of the colors can be dismissed by clicking on the little **X** button at the top.



The input colors can be reordered by dragging the color number or one of the three color components up or down.

An input color can be deleted by clicking on the Trash icon.

An input color can be redefined by clicking on its colored button and using the Color Picker or by changing the color components.

The exclamation point in a circle icon warns you that the color may not print well. It can be ignored or the color can be adjusted.

Getting back to finding an intermediate color in a gradient: We'll try an example. Make the first color for the gradient lime (0, 255, 0) and the second color magenta (255, 0, 255) and hit the **Go** button to draw a gradient between them. The resulting picture shows a gray area in the middle. Gray doesn't usually look good in a picture, so you may want to change it.

To fix a poor intermediate color (e.g., gray).

1. Click the **Add New Color** button.
2. Click in its displayed colored area.
3. Use the Color Picker Eyedropper to select a color near the middle of the gray area of the gradient.
4. Verify the new color displays the unfortunate intermediate color.
5. Select the new color again and use the Color Picker to find a better color. The Color Sliders work well for this. Any set of three equal components will be a shade of gray, so avoid that.
6. When you find a color you like, exit the Color Picker and hit Return to select it.
7. Finally, move the new color up between the two adjacent colors (in this example, the first and second colors).

Now the first two gradients should look good.

Now for a little math to help explain what's going on.

How are these pictures generated? We start with an iterative function: $z_{n+1} = z_n^2 + c$, where z and c are complex numbers. Since our computers don't work with complex numbers, and they seem a little scary, we use a little math to get to real numbers. Then we can use the real x and y values of geometry. For each pixel location in the image, the program starts with that location and computes a new location. The new location is tested to see if it exceeds some maximum value that we choose. If it does, the program records the number of iterations required to get there. If not, it checks to see if the number of iterations has reached the maximum allowable number of iterations. If it does, it records that number and goes on to the next pixel. If not, it computes a new location.

The distance from the origin to a point (x, y) is the square root of $(x^2 + y^2)$. It has been shown that the distance from the origin must grow to infinity if that distance ever exceeds 2. Since square roots are an unnecessary and expensive calculation, we just use the distance squared, so the **distance** must grow to infinity if the **distance squared** exceeds 4. We call the distance from the origin, r, so we compute r^2 and call it rSq.

We use the number of iterations to help to define a color for that location.

What if we iterate many times and r^2 never exceeds the limit of 4? We call that limit rSqLimit. In general, we can't be sure that r^2 will ever exceed 4, so we pick a large number and say that, if we don't exceed rSqLimit in that number of iterations, we never will. We generally color that location black.

We use the variable, iterationsMax, for the maximum number of iterations to try. We start out with iterationsMax = 10,000. Any time you find a black shape that doesn't look like a Mini-Mand, it means you need to increase iterationsMax. You define the RGB colors used in the image and their order. You also define the numbers of blocks of color to be used, nBlocks. We start with six colors and 60 blocks. The colors we used are, in order: lime, yellow, red, magenta, blue, and cyan. These are six of the eight corners of the RGB color cube. The program finds the minimum number of iterations used at any location in the picture so that our coloring can start somewhere on the edge of our picture.

Why would we ever choose a value for rSqLimit greater than 4? This has to do with color smoothing. We find the number of iterations required to exceed rSqLimit at each location in our image. We then assign a color to that location based on that number of iterations. However, there will be many locations nearby that have the same number of iterations. We would be left with bands of a uniform color, which may not look good.

To get color smoothing, we need another piece of information. What if, instead of picking 4 as the value for rSqLimit, we picked a much higher value? We could then adjust the number of iterations by a fraction depending on how high r^2 got to in exceeding rSqLimit. We call that value of r^2, rSqMax. We can calculate just how large rSqMax could possibly be for a given value of rSqLimit. rSqMax is simply $(rSqLimit + 2)^2$.

We want to define a value that acts like a fractional value for the number of iterations required to exceed rSqLimit, so we defined something that acts like a differential iteration as:

$$dIter = (r^2 - rSqLimit)/(rSqMax - rSqLimit)$$

and a fractional iteration as:

$$fIter = iter - dIter.$$

The program puts each value of fIter into a two-dimensional array of values covering the width and height of the image.

A plot of fIter against location would show a very non-linear curve. That would not be good for smoothing out the colors, so an equation using logarithms of logarithms computes dIter from r^2, rSqLimit, and, rSqMax.

This helped to makes dIter much more linear and looks better. Then we just linearly vary the R, G, and B color components to get the desired color smoothing.