Tie-Dye Pixel Art Generation and Acceleration

HENRY LUENGAS
MARIA PANTOJA, California Polytechnic State University San Luis Obispo

In this study, we discuss a method for rendering tie-dye like pixel art images shown in Figure 1. The naive implementation for this process is far too slow to be useful for large images, so we evaluate acceleration of the process using methods including: CPU parallelism, GPU parallelism, just in time compilation, and use of a spatial data structure.

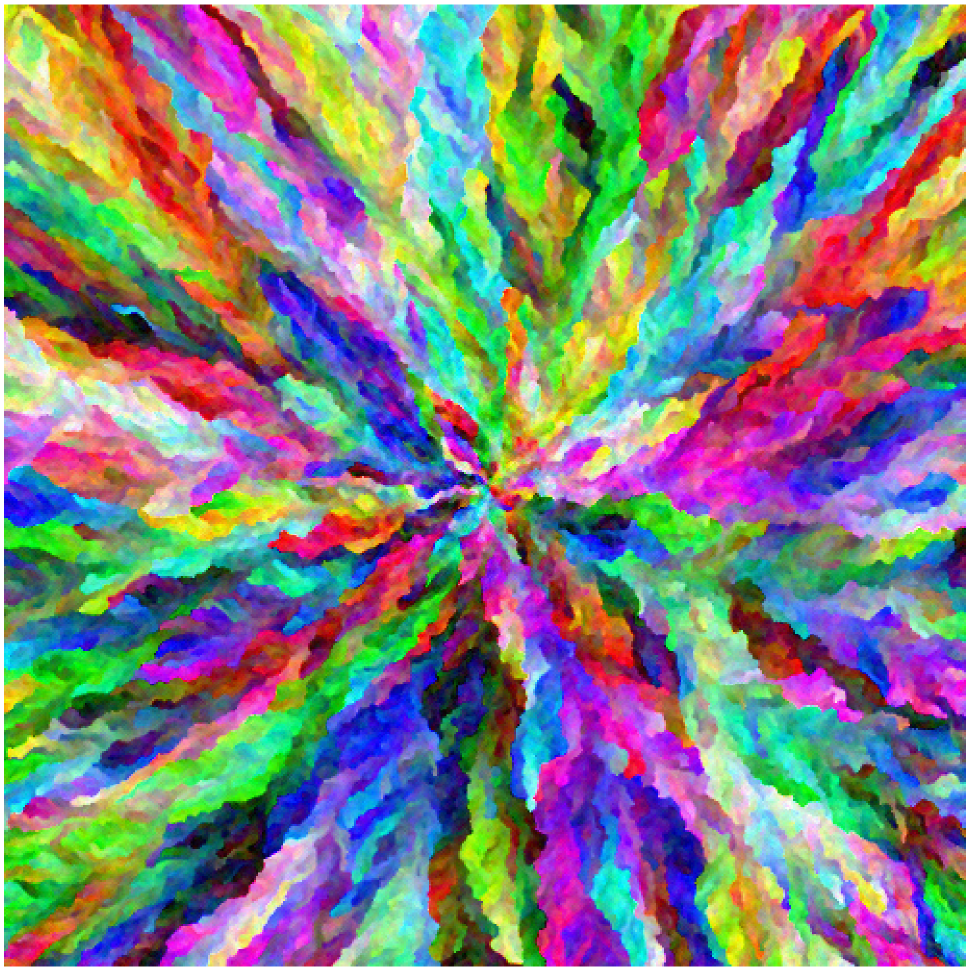


Fig. 1. 500x500 Tie-Dye Pixel Art Image.

Authors' addresses: Henry Luengas, hluengas@calpoly.edu, henryluengas@gmail.com; Maria Pantoja, mpanto01@calpoly.edu, California Polytechnic State University San Luis Obispo, 1 Grand Ave, San Luis Obispo, California, 93405.

1 INTRODUCTION

The project was inspired by a Code Golf challenge, "Images with all Colors" [4]. The goal of the challenge was to make images where each pixel is a unique color. Many algorithms in a variety of languages were submitted, but the images produced by one really caught our eyes. As the poster noted, the produced images look like paintings and are aesthetically interesting. Unfortunately, using the method described takes dozens or even hundreds of hours to create a wallpaper sized HD image, thus it would be completely unfeasible for the even higher resolutions needed for print media. The style of the images lessens the impact of this issue, because images can be rendered smaller and scaled up as needed. Pixel art images show no loss of image quality at scale so long as no interpolation algorithm is used for scaling. Despite this, the naive approach is clearly lacking. The purpose of this project is to explore various methods of accelerating this process.

2 ALGORITHM DESCRIPTION

The images are produced in the following way. Given a set of distinct colors, shuffle them into a random order and pick the first color from the set. Place this color on the image as the starting colored pixel. From the set of colors, add one color to the image at a time such that the new color is placed adjacent to a previously colored pixel where it "best" fits in the image. We will refer to the color being placed as the "Target Color" going forward. Since each colored pixel must be adjacent to neighboring colored pixels, a boundary region of black pixels naturally forms. This boundary layer must be searched through exhaustively to place each color, and it changes with the addition of each new pixel.

2.1 Best Fit Strategies

The original poster of the Code Golf response proposed two strategies for determining best fit. We came up with a third one as well. All three involve using the Euclidean distance formula over an RGB color-space. Any further references to the Euclidean distance formula are always over RGB, never spatial dimensions. All strategies start with the consideration of every available location; for each of which, a color-distance will be computed. The only requirement for an available location is the adjacency of at least one colored pixel. Each strategy computes this color-distance differently. Ultimately the Target Color is placed at the location with the minimum computed color-distance. A small output image for each strategy is shown in Figure 2.

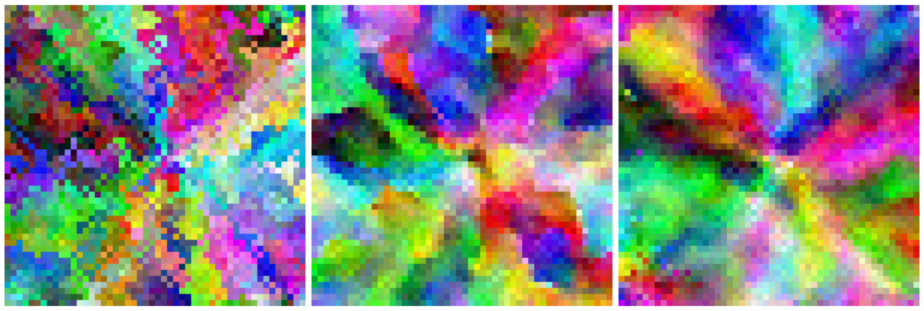


Fig. 2. 50x50 Output for Minimum, Average, and Neighborhood Strategies Respectively.

2.1.1 Minimum Color-Distance. The first strategy is to compute the distance between the Target Color and each colored neighbor of the available location. The minimum of these distances is taken to be the computed color-distance for this available location.

This method creates a highly pixelated output, and places colors with no regard for the color of the surrounding area. Every color will end up adjacent to the nearest possible color to itself.

2.1.2 Average Color-Distance. The second strategy is very similar. For each colored neighbor of the available location, compute the distance. Instead of the minimum, the average of these distances is taken to be the computed color-distance for this available location.

This method creates a much softer edged image. It gives weight to each location's surroundings, resulting in colors grouping up to form larger, smoother tracks out from the center.

2.1.3 Neighborhood Color-Distance. The final strategy is to first find the average value of the colored neighbors. Sum up all the respective red, green, and blue channels, then scale down the resulting color vector by the number of neighbors. This results in a single color that we will refer to as "Neighborhood Color" going forward. The distance between the Target Color and the Neighborhood Color is taken to be the computed color-distance for that available location.

The output of this method is similar to that of the Average Strategy. At small sizes, it can be hard to differentiate them. However, one may notice that the Neighborhood Strategy produces even softer borders between the various color streaks. Despite producing similar images, these two methods vary wildly in their run-time. This is entirely due to the way the boundary region forms using each of the two strategies, as at this point, all three strategies must consider every neighbor of every available location.

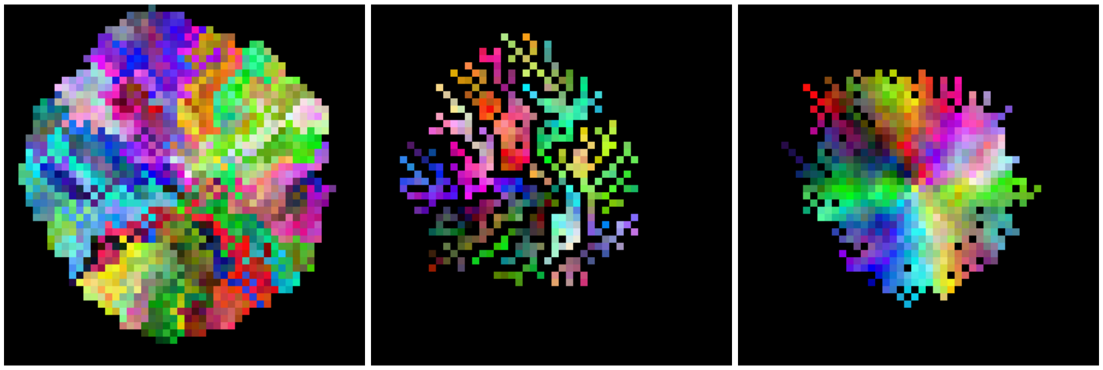


Fig. 3. 50x50 Colored-Uncolored Boundary for Minimum, Average, and Neighborhood Strategies Respectively.

2.2 Strategy's Effect on Boundary Formation

Figure 3 shows how each of the three strategies create very different naturally forming boundaries between colored and uncolored pixels. The Minimum Strategy produces the smoothest and most uniform front, growing like the circumference of a circle grows with its area. While the other two strategies both consider the entire area surrounding a location, the Average Strategy leads to a boundary region that grows in size much faster. This mode biases placement towards locations with fewer valid neighbors. The bias is the result of the average distance being reduced by the presence of more neighbors, because having more neighbors usually means having some distant (color-wise) neighbors. The Neighborhood Strategy generates a front somewhere between the other two, but closer to that of the Minimum Strategy. It remains roughly circular, but with salients encroaching both inward and out. A few dark enclaves remain in the ball of color and a few color shoots protrude outward like feelers.

The growth of the boundary region is one factor that strongly affects the generation time of these images. The time it takes to place each pixel depends only on the number of available locations at that time. As an example for a 100x100 image, render times were: 2 minutes (Minimum Strategy), 19 minutes (Average Strategy), and 6 minutes (Neighborhood Strategy). The growth rates of the boundary regions are represented in Figure 4, where the number of pixels available is graphed against the number of pixels placed for each strategy.

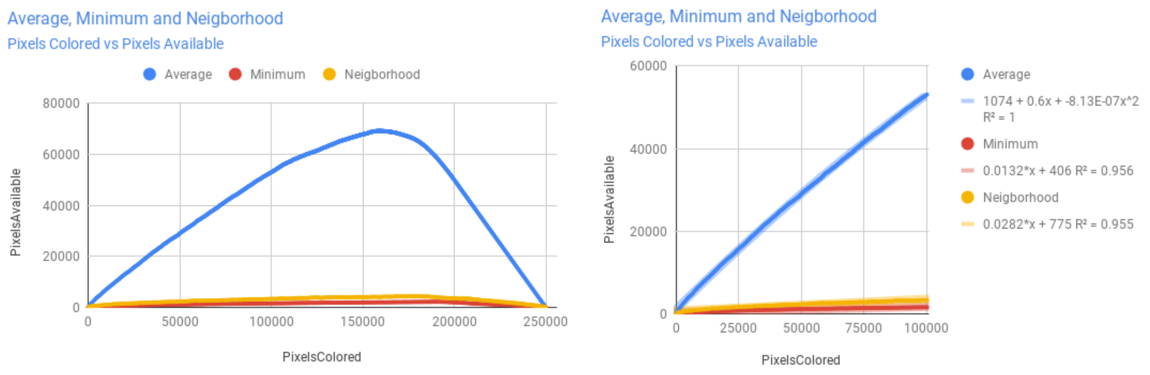


Fig. 4. Boundary Region Growth Rate by Strategy - 500x500 Render.

In the graph on the left, we can see that each strategy goes through three phases: the first being unrestricted growth, followed by limited growth when the outer edge of the image is first reached, and finally a sharp drop off in available locations as the image is finished. The graph on the right shows the same relationship but for only the first phase of growth. This growth phase is best approximated by a linear relation for the Minimum and Neighborhood Strategies, and by a quadratic relation for the Average Strategy. For small images this also matches a linear relation well, as the coefficient for the second degree term is minuscule.

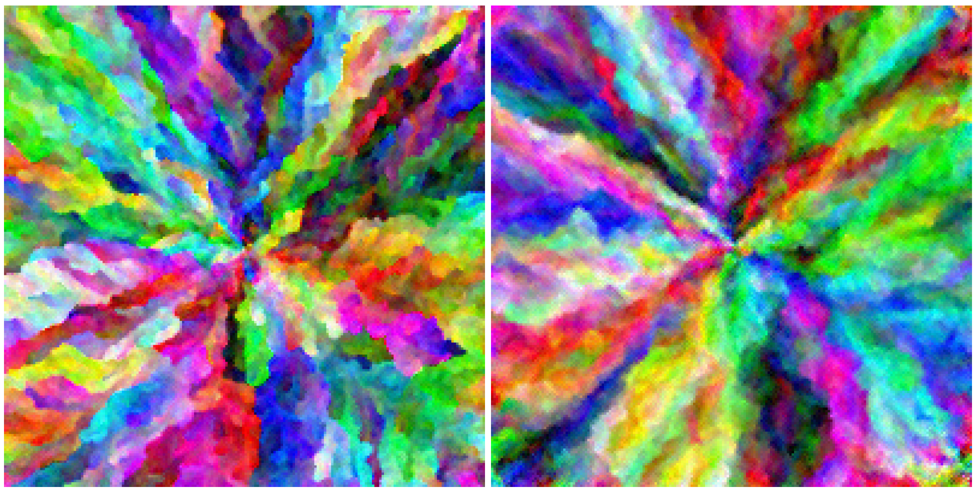


Fig. 5. 500x500 Render - Average Strategy Left, Neighborhood Strategy Right.

We can conclude that if a pixelated image is desired, the Minimum Strategy is the best choice as it is not only fastest, but also the only choice that produces pixelated output. If a smoother output is desired, the Neighborhood Strategy can greatly improve the total run-time; however, especially with larger images, the aesthetic difference between the Minimum and Neighborhood Strategies becomes more obvious. Therefore, the Average Strategy cannot be entirely discarded. Figure 5 shows the visual difference between the two strategies. Most notably, the borders between different streaks of color are very sharp when using the Average Strategy.

2.3 Spatial Data Structures

The ballooning search space is a primary concern for improving the speed of the painter. Trees allow for sorted data to be partitioned, and many data structures exist that optimize searching through such data. A few relevant structures are briefly explained in this section as they are well explained in computer science literature.

2.3.1 B-Trees. The simplest such data structure is a Binary Search Tree (BST), which cuts the search time of the sorted data from $O(n)$ to $O(\log n)$. The next notable variation is the B-Tree, which generalizes the concept of a BST, allowing tree nodes to have more than two children [3]. B-Trees have the same time characteristics as BSTs, but they are self balancing and have other features that make them ideal for large serialized data sets like those used by databases.

One problem with BSTs is that our search space is multi dimensional. We could sort colors by their hue, but our best fit algorithm strictly considers all 3 RGB values. Therefore, the best position may not be adjacent to a color of the same hue if brightness or saturation vary wildly. Luckily, many data structures exist for partitioning data by arbitrary numbers of dimensions.

2.3.2 Quad/Oct-Trees. The first of these is a Quad-Tree; it is a type of B-Tree that partitions 2D space at each node into 4 quadrants. One step up from that gives the Oct-Tree, which works the same way on 3D space, dividing it into 8 cubic regions at each node. Oct-Trees are often favored for their uniformity and simplicity, and use of one will allow us to search an RGB color space in $O(\log n)$ time [5].

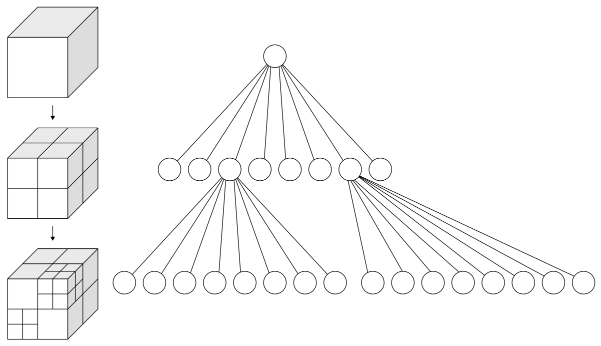


Fig. 6. [Public domain], via Wikimedia Commons. (https://en.wikipedia.org/wiki/Octree)
Oct-Tree Splitting Representation.

2.3.3 KD-Trees. Another space partitioning data structure that we considered is the KD-Tree. It works by dividing N-Dimensional space into two half spaces at each node, alternating the splitting dimension each time. In three dimensional space, this has the effect of splitting 3D space with a 2D plane at each node [2]. This generalizes to any number of dimensions like the 2D example split by 1D lines shown in Figure 7.

KD-Trees have the best performance for nearest neighbor search when they are properly maintained and balanced. Unfortunately while both insertion and deletion are possible and take $O(\log n)$ time, the tree is not self balancing and doing so has an $O(n \log n)$ time penalty [2]. This makes KD-Trees less useful in the case of rapid insertion and deletion that we will encounter.

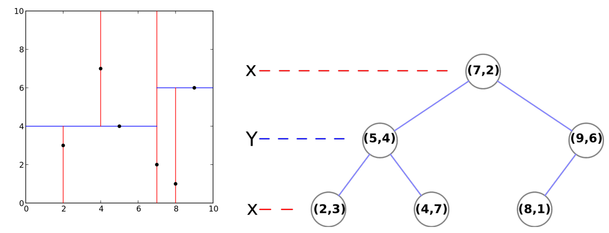


Fig. 7. [Public domain], via Wikimedia Commons. (https://en.wikipedia.org/wiki/K-d_tree)
Left: 2D Points To Be Indexed
Right: Resulting KD-Tree With Split Axis Labeled

2.3.4 R-Trees. The final spatial indexing algorithm we investigated is the R-Tree. It works like a B-Tree, but where each node consists of bounding "hyper"-rectangles (N Dimensional Rectangles) for its children. Leaves can either be rectangles themselves or points which are just rectangles where the min coordinate is equal to the max coordinate. A 2D example of bounding rectangles, and the corresponding R-Tree is shown in Figure 8.

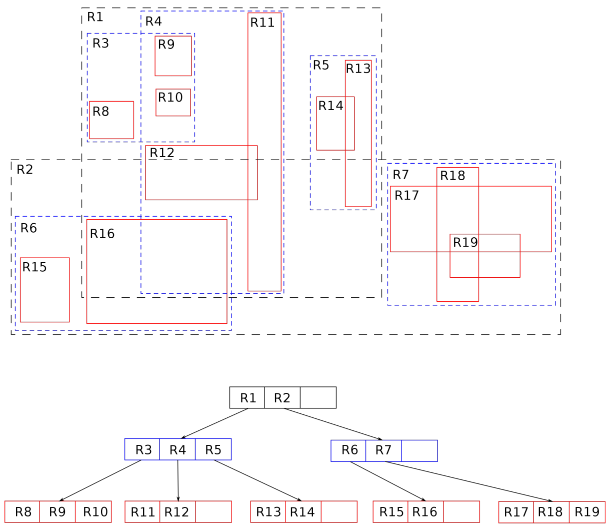


Fig. 8. [Public domain], via Wikimedia Commons. (https://en.wikipedia.org/wiki/R-tree)
Top: 2D Points/Rectangles To Be Indexed
Right: Resulting R-Tree

R-Trees have $O(\log n)$ insertion, deletion, and nearest neighbor search performance. If presorted, many points can be "bulk-loaded" with a single insertion. R-Trees may be experimentally faster than Oct-Trees [5] for nearest neighbor search, but this is highly dependent on the coordinate data itself and the tuning of the trees. The only way to tell conclusively would be to test both.

For this project only the R-Tree structure was implemented; however, in future work we would like to investigate Oct-Trees for the same application. KD-Trees could be used as well but would require a novel strategy like the one used in this project [1], that utilizes a forest of KD-Trees of increasing size. The author rebuilds sub-trees at intervals to avoid deletion and maintain good search performance.

2.3.5 R-Trees.* We ended up choosing to use a variant of the R-Tree known as the R*-Tree for this project. The search and deletion algorithms from R-Tree remain unchanged for the R*-Tree variant; however, with R*-Tree, the heuristics used for insertions and splitting of full nodes are improved. In addition, the R*-Tree tries to avoid splits by reinserting objects and sub-trees into the tree. The effect of this improved insertion and splitting strategy is a minimization of overlap. Overlap meaning that on data query or insertion, more than one branch of the tree needs to be expanded. These effects are easy to see in Figure 9 which depicts German postal codes represented using the original Guttman R-Tree and again using the R* Variant.

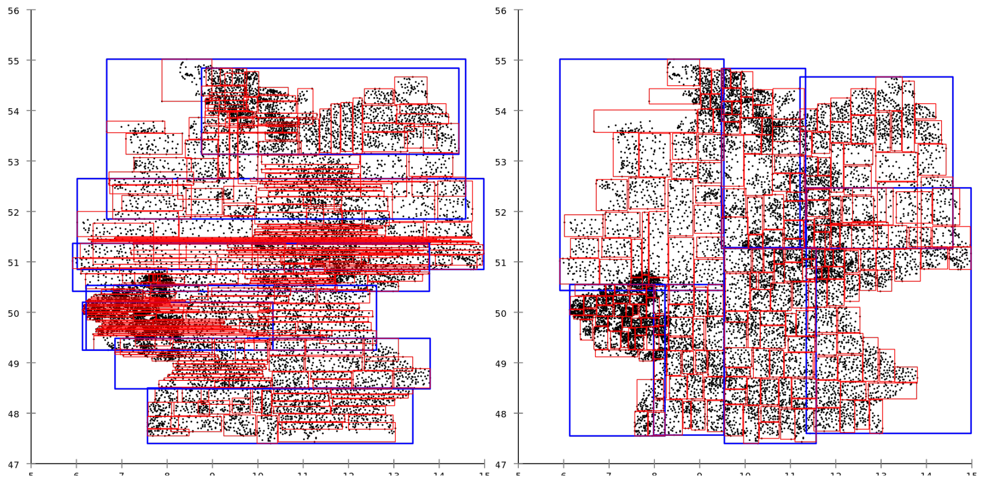


Fig. 9. [Public domain], via Wikimedia Commons. (https://en.wikipedia.org/wiki/R*_tree)
German Postal Codes: Guttman R-Tree on left, R*-Tree on right.

3 ACCELERATION

We chose to implement the renderer using Python because it is simple to prototype with and has great library support. It was not critical to use a compiled language as the goal was to investigate ways of improving the process, not create the absolute fasted possible render engine. We planned to parallelize the process using Python threads on the CPU, implement a data structure to cut down on the number of positions evaluated for each placement, and scale up the parallelization to run on a GPU using OpenCL. Along the way, we decided to implement Numba, a Python library that compiles functions to CPU machine-code on the fly using LLVM.

3.1 Concurrent Futures - Python Multiprocessing

Parallelism was the first feature we added to accelerate the painting process. Multiprocessing in Python is done through the use of the threading and multiprocessing libraries. It is simplified further by the "concurrent.futures" module. This module provides a high level interface for creating and dispatching thread or process pools. The strategy we came up with for parallelizing the painting process is simple.

If 8 workers are desired, then eight colors rather than one with be drawn from the random color set; one color is given to each worker. The eight workers are dispatched as their own processes to find the best location given the current canvas. Each worker must now return its decided best location and the target color it was asked to place. As the workers finish, the main process checks that the desired location on the canvas is indeed still available. If available, the canvas is altered and a pixel has been placed. If it is not available, then one of the previous workers must have snatched that location before the worker that just completed. We refer to this as a collision, and we record collided colors in a separate list to be placed sequentially at the end. Since colors are being selected and placed in a completely random order, collisions have no effect on how the output image looks.

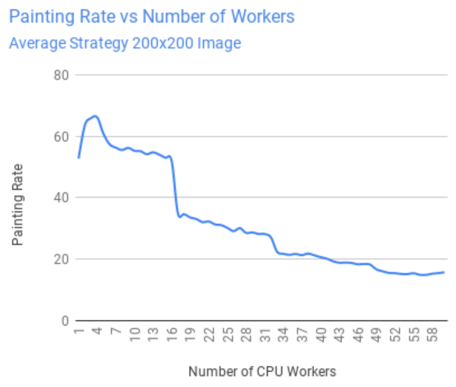


Fig. 10. Painting Rate vs Progress - Various Strategies

It is still desirable to limit the rate of collisions as they do represent wasted work even if they don't cause any aesthetic damage. For this reason, the program will always begin rendering with a single worker. For every 200 pixels available, it will add another worker until it is capped the number of threads detected by the system. Figure 10 illustrates that the first 16 threads added limit the loss in performance due to the expanding search space. After, this there is little benefit to using more threads. We expected 16 to be the optimal number as the CPU used has 8 cores (16 threads). This strategy keeps the rate of collisions down around 10% until the thread count cap is reached at which point the rate will steadily decline.

Strategy	Sequential	Parallel
Minimum	2 min	2 min
Average	19 min	3 min
Neighborhood	6 min	3 min

Table 1. Render Times for a 100x100 image.

Table 1 lists the render times for each strategy with and without parallel processing. The Minimum Strategy sees almost no improvement because it only gets a second worker at the very end of its render due to the slow growth rate of its boundary region. This is shown in Figure 11 as the Rate and Rate-per-worker diverge only briefly. The boundary region grows much faster using the Average Strategy allowing for CPU to hit full utilization before finishing the render, leading to a 6x decrease in render time. Finally, the Neighborhood Strategy sees less impressive gains as it is able to use multiple threads but finishes before being able to fully utilize the CPU.

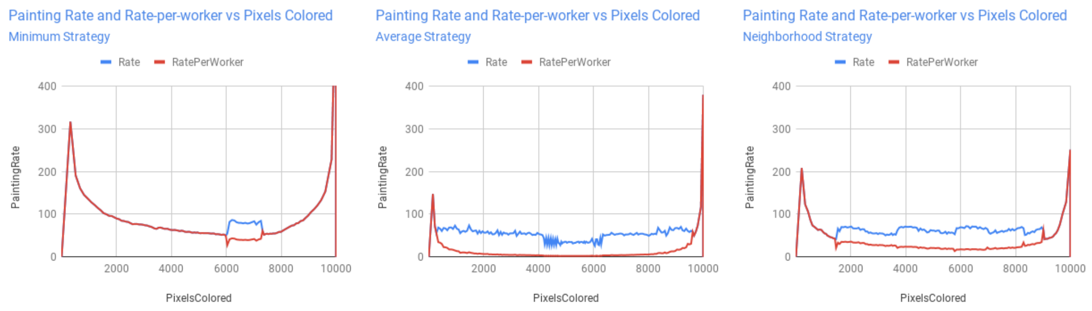


Fig. 11. Painting Rate vs Progress - Various Strategies

Figure 12 illustrates that CPU multiprocessing is able to flatten the loss of performance up until the CPU becomes fully utilized, at which point the performance characteristic becomes a constant multiple of that of the sequential brute-force approach. On the graph this is represented by the linear decrease in print rate right up until about the 3200 available pixels mark (16 threads * 200 locations), where the rate begins to decay rapidly.

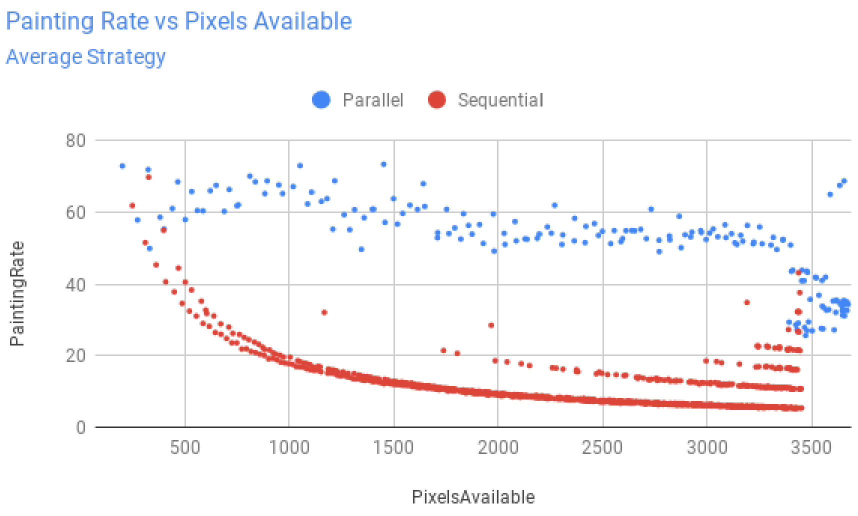


Fig. 12. Painting Rate vs Available Pixels for 500x500 Render using OpenCL

All this indicates that for large enough renders each strategy could see up to 6 fold speed improvement, but that certain strategies achieve this speed increase much quicker than others. If we were to do a 200x200 render instead of 100x100, then the Neighborhood Strategy would now beat the Minimum Strategy. Having a search space that grows faster allows it to use more workers faster, ultimately leading to a faster completion time. Unfortunately this is only effective to a point, and the multi-threaded painter is only practically fast enough to make small thumbnail sized images. Beyond these sizes the rate per worker soon drops precipitously.

3.2 R*-Tree - Limiting Search Space

Distinct RGB colors are distinct coordinate points along red, green, and blue color axes. Therefore for a set of colors loaded into an R-Tree, querying for the nearest neighbor of a target color necessarily returns the color in the structure that has the minimum Euclidean color distance to the target. This is very similar to what we need, and in fact two strategies can be easily adapted to use an R-Tree data structure.

3.2.1 R-Tree Minimum Strategy.* For the Minimum Strategy, we must switch from tracking available black boundary pixels to tracking their adjacent colored counterparts. The R-Tree is queried with a given target color, and the result is the least distant color. One of the up to 8 black neighbors of this location is chosen at random as the best position. Each of these neighbors would have tied for best location using the the brute-force searching strategy. Some systemic changes would need to be made in order to track the colored boundary region rather than the black available pixels. This would certainly improve the time taken to get the best position from $O(n)$ to $O(\log n)$. Because of the required systemic changes and a general lack of interest in the more pixelated images, we did not implement this feature.

3.2.2 R-Tree Average Strategy.* The same approach falls short for the Average Strategy. This strategy requires finding the distance to each neighbor of a location, and you cannot use the same assumption and shortcut as the Minimum Strategy. You cannot pre-calculate, average, and store these distances because they will rely on an as of then unknown target color. We were unable to conceive of a method for cutting down the search space for this strategy. This is unfortunate because this method produces perhaps the most interesting images, and we would clearly see the most benefit from a search space reduction as its search space grows much faster than the other strategies.

3.2.3 R-Tree Neighborhood Strategy.* Luckily the final strategy allows us to render the smoother looking images using the benefits of an R-Tree. For this strategy, we are back to the standard method of accounting for the available boundary pixels. Now not only are available locations in a list, they are also inserted into the R-Tree. Despite actually being black, the available location is stored with its Neighborhood Color, as calculated by averaging the color of each colored neighbor. Recall that this strategy tries to minimize distance between Target and Neighborhood Colors. Therefore to find the best location, we query the R-Tree for the nearest neighbor of the Target Color, giving us the location with the minimum color distance between Neighborhood Color of the available location and the Target Color. Thus the strategy is fulfilled and the location is painted.

After painting a pixel, each implementation must decide which of it's neighbors are now available. The brute-force implementation only requires knowing that a location is available, now we also need to know the location's neighborhood color. So under this implementation, if a neighbor is already being tracked; then it must be removed from the R-Tree, its Neighborhood Color must be recalculated, and then it must be reinserted into the R-Tree. Several different schemes could be used to limit this added overhead, such as flagging updated locations as unavailable, and making use of the bulk-loading insert feature of R-Trees.

3.2.4 Conclusions. The Neighborhood Strategy was implemented using an R-Tree, but no suitable method was found for the Average Strategy. The Minimum Strategy can utilize an R-Tree but this was not implemented due to time constraints. To evaluate its performance, we will only look at metrics for Neighborhood Strategy renders. For a 100x100 render using sequential processing, brute-force painting took 6 minutes; using R-Tree sequentially took only about 10 seconds. This is a surprising result because the search space never gets very big during a 100x100 render. The R-Tree should be slower or equivalent to the brute-force search for sizes of n < 100 or so.

The reason for the over thirty fold increase in render speed comes down to compilation. The R-Tree Python module that we used is actually a wrapper for the libspatialindex C library. So we are inadvertently hugely benefiting from C compilation over Python's interpretation. This made it clear that a compiled brute-force version of the algorithm would be necessary for a true comparison with the data structure. Because of this, more R-Tree testing results will be mentioned in the following sections.

There is nothing preventing use of a spatial data structure in conjunction with multiple workers. Using the same strategy as discussed earlier, we would expect good scaling. Unfortunately, the libspatialindex library that we used for this project is explicitly not thread safe for nearest neighbor queries, even though this should be a read-only operation for an R-Tree.

3.3 Numba - Just in Time Python Compilation

Due to the unexpected performance gain that use of a C library provided, we decided Python may not be sufficient for this project. C++ and Rust were considered, but ultimately we decided to salvage our Python work with a package called Numba. Numba provides function decorators that when encountered by the Python interpreter, cause a version of the function to be compiled on the fly using the LLVM compiler. Any subsequent calls will now use the fast machine code version of the function. Numba has limitations and only works on a narrow subset of Python code using NumPy arrays. These types of code simplifications were going to be needed for painting with OpenCL as well, so they were implemented at this stage. All of the work for getting the best location was consolidated into a single function. Pyhton style for loops were simplified into more general C style for loops. Finally, arguments were all converted into NumPy arrays, and any computation done with NumPy function calls was altered to be done manually. This was done since NumPy itself optimizes with compilation so we wanted to isolate any benefits. The result was a monolithic pure Python work function, ready to be paired with Numba.

Table 2. Render Times for a 100x100 Image.

Strategy	Pure Python	Numba Accelerated
Minimum	2.22 min	0.06 min
Average	19.49 min	0.30 min
Neighborhood	5.51 min	0.09 min

Table 2 lists render times for a 100x100 image with and without Numba acceleration. All strategies see a huge 30-60 fold speedup. The true value is likely greater than a 60 fold performance boost because the first method call incurs a roughly 1-2 second compilation time penalty; therefore, the results are skewed for the shortest renders. At this point in development, the multiprocessing and Numba features were usable in conjunction. Unfortunately it seems that Python cannot keep pace scheduling accelerated workers, as scaling was almost non-existent. This must be the reason Numba provides its own multiprocessing and thread pooling functionality. We chose not to implement this feature as it seemed clear that it would provide somewhere between half and full linear scaling, not providing any new information.

Now that Numba was implemented, it was possible to draw more direct comparisons between the brute-force and R-Tree implementations. To make this comparison, a 500x500 image was rendered once using Numba, then again using the R-Tree. The Numba implementation completed the render in 10.4 minutes; the R-Tree implementation completed in 7.4 minutes.

In Figure 13 one can see that the R-Tree implementation produces the same style output image. In fact, the R-Tree implementation does have one small difference. Normally the algorithm handles ties in color distance in a uniform way leading to a slight bias in placement towards the top left corner. This means the top left corner is painted naturally and that the and the bottom right corner gets painted by the very last colors to be placed. For this reason the R-Tree implementation (Right) produces better, more uniform corners.

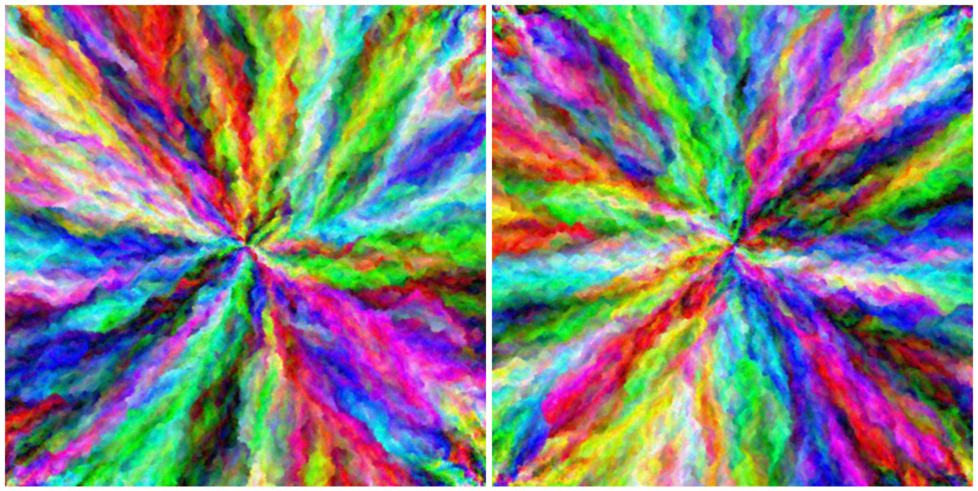


Fig. 13. Numba vs R-Tree Renders and Printing Rates Graphed Against Available Locations

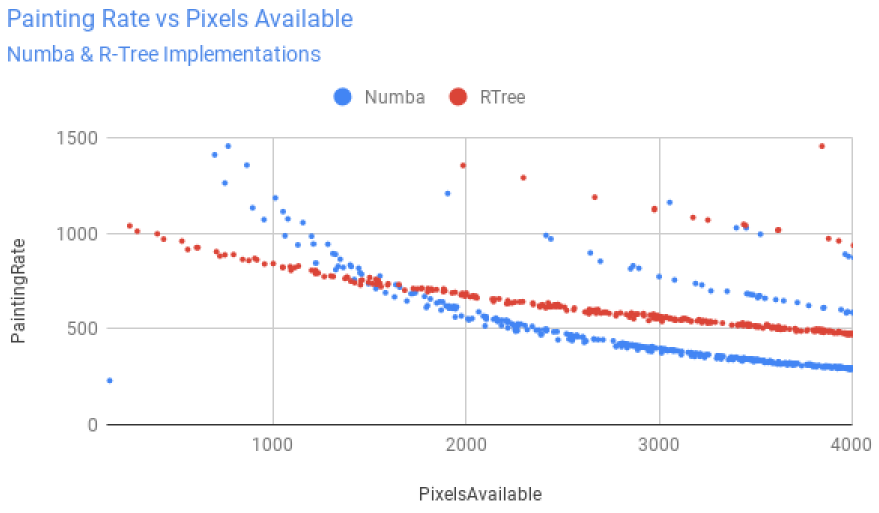


Fig. 14. Numba vs R-Tree Renders and Printing Rates Graphed Against Available Locations

Figure 14 shows that the Numba implementation begins the render with a greater painting rate. This makes sense because R-Trees are no better than brute-force for small search spaces, and because the R-Tree requires extra work for each painted pixel to maintain the data structure. It takes less then 2000 available locations for performance of the R-Tree implementation to overtake that of the Numba implementation. The decay of its painting rate is superior to that of the brute-force implementation.

It was difficult to specifically quantify this decay, as we can't explain why both implementations seem to bounce between a primary curve and then a secondary or tertiary curve with greater printing rates. Each curve is itself two curves on top of each other as the number of pixels available sweeps from low to high and then back down to low as the render completes. In conclusion use of an R-Tree greatly improves the renderer's speed for larger images, but comes at the cost of precluding the Average Strategy.

3.4 OpenCL - GPU Multiprocessing

Despite being unable to use the R-Tree, there was still hope for speeding up the Average Strategy. The final tool used for accelerating the project was OpenCL. OpenCL is an open standard that facilitates executing compute tasks across a wide range of different hardware including CPUs and GPUs. Using a GPU could allow us to scale the problem up to more than 16 workers, potentially allowing for larger images to be rendered even using the Average Strategy.

PyOpenCL is a Python module that lets you access GPUs and other massively parallel compute devices from Python. Compute kernels are written in a variation of C used by OpenCL, but the rest of the complex OpenCL setup and boilerplate code is managed by a handful of high-level Python commands. Similarly to the Numba decorated function, data is passed to the kernel from special PyOpenCL buffers filled by NumPy arrays. The only difference, which caused a fair bit of trouble, is that the arrays must be flattened to one dimension. After implementing these changes and translating the work function to OpenCL kernel code, all three strategies could be run on the GPU.

The strategy for choosing the number of workers is largely the same as that used for CPU parallelism. Additional workers are added one at a time for every 200 locations available, but now up to a maximum of 256. This number was chosen experimentally as four times the number of compute units (64) on the GPU we used during development. Greater numbers did not have a negative effect, but there was a clear benefit from using multiples of 64. Despite this, we were unable to achieve any better performance by incrementing by multiples of 64. The work-group size as well as many other parts of the GPU implementation could use far more optimization.

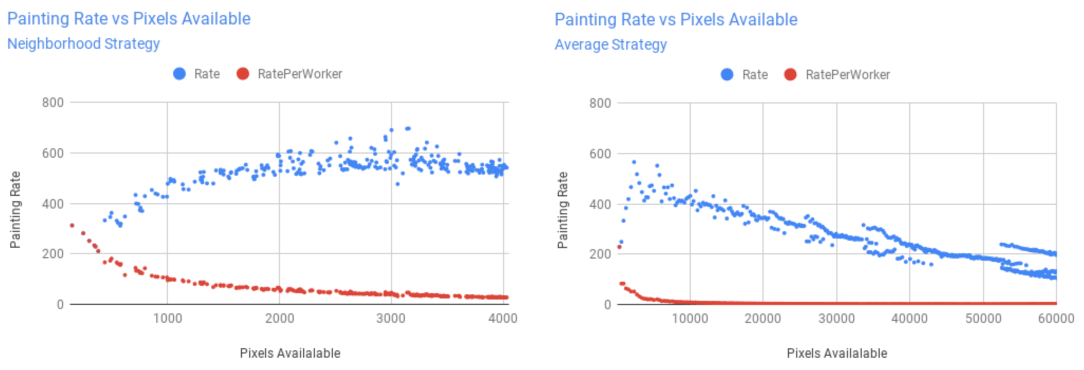


Fig. 15. Painting Rate vs Available Pixels for 500x500 Render using OpenCL

Despite the lack of optimization, this implementation performed very well in our testing. In Figure 15, one can see the painting rate and the rate per worker for the Neighborhood and Average Strategies respectively. The left graph displays how the rate grows as we add workers to a max of 20 for 4000 locations. The right graph shows the same behavior we observed with CPU multiprocessing, a slight rate increase, followed by a now much longer period of linear decay. A count of 256 workers are in use near the 50,000 pixels available mark, as more workers are added beyond this that we see the same odd behavior arise where painting rate will fluctuate between two or more independent rates. This printing rate is calculated very infrequently, once per second, but by this point the rate per worker is well below 1 per second. Despite this, the overall rate is high compared to our other implementations.

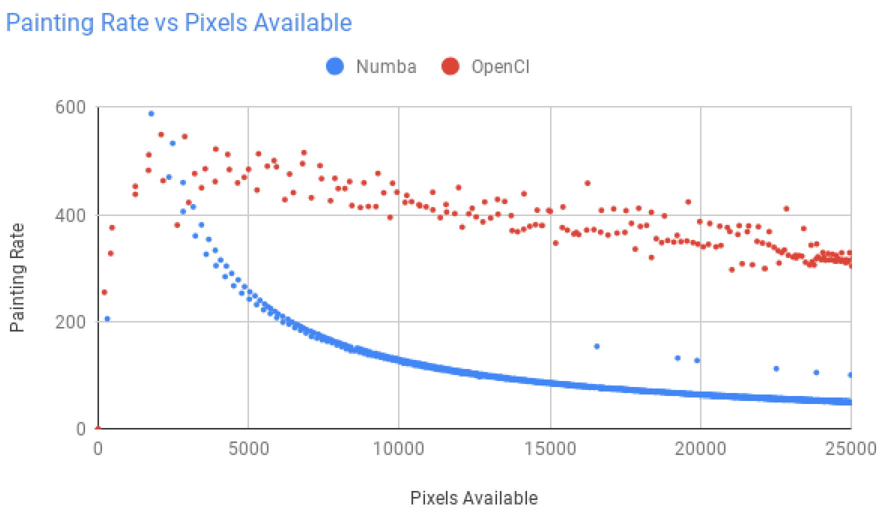


Fig. 16. Painting Rate vs Available Pixels for 500x500 Render using OpenCL

Compared to our sequential implementation (Numba), the parallel compiled OpenCL solution is far quicker. Figure 16 demonstrates this performance uplift. The Numba render starts out far in the lead but the painting rate drops precipitously, whereas the OpenCL render starts modestly, quickly accelerates, and then maintains its lead. Notably by the end of the render when 25,000 locations are available, Numba's rate is still near 50 pixels per second, while the rate per worker for OpenCL has fallen to about 1 pixel per second. This is part of the reason we feel the OpenCL implementation still has plenty of room for further optimization.

Using the Neighborhood Strategy with OpenCL allows us to compare its performance to that of our R-Tree implementation. This comparison is shown in Figure 17. Although the OpenCL render ends with a higher rate, we know that it will soon begin to drop off linearly. At some point the R-Tree render's rate will again overtake that of the OpenCL render, but for images smaller than 500x500 their performance is similar.

Despite having clear room for improvement, the OpenCL renderer is competitive with the other implementations. It is the champion when it comes to rendering images with the Average Strategy, which cannot use the R-Tree. Unlike with the CPU which we could utilize fully, we don't think we even scratched the surface of GPU utilization. Some of the ways this could have been improved upon will be discussed in the Next Steps section of this paper.

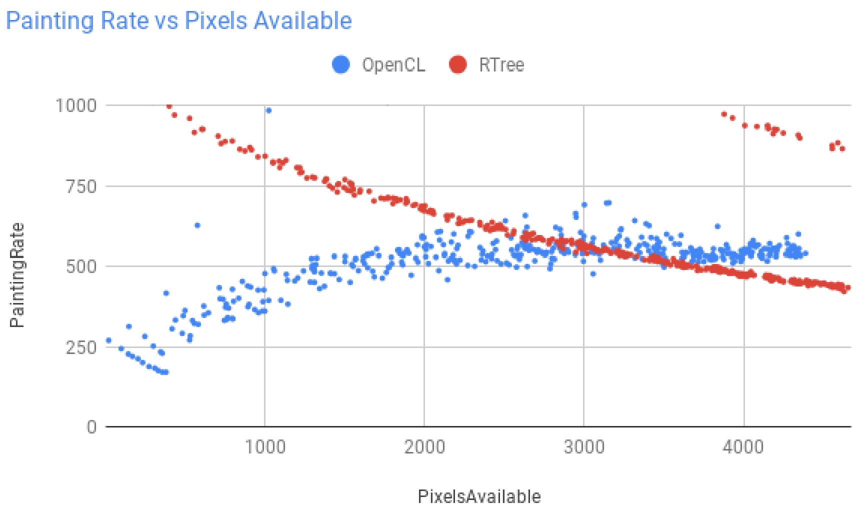


Fig. 17. Painting Rate vs Available Pixels for 500x500 Render Using OpenCL

4 EXTRA RENDERER FEATURES

4.1 CLI

The program has an easy to use Command Line Interface that allows access to all features and many configurable options. No input is required; all options have default values. There is a toggle for each implementation: -multi, -numba, -rtree, and -opencl. The user can choose which best fit strategy to use. The user may input height and width dimensions for the image, as well as chose the starting coordinates. The bit depth of the generated color space can be chosen. Alternate generation color spaces can be selected: -HLS and -HSV. The last shuffling step for color generation can be disabled, allowing the user to group colors by a color channel (this is explained further in "Colors"). Finally, the relative path filename can be selected for the output files.

4.2 Output Files

The program creates a "filename.txt" file that records the options used for this render. This file will also include the final render time after completion. Once per second, stats will be saved to "filename.csv" and "filename.png" will be updated. A separate temporary PNG is also stored. The stats recorded include: Pixels Placed, Pixels Available, Percent Complete, Total Collisions, Rate, Worker Count, and Rate-per-Worker. At the end of the render, the program will use FFmpeg to construct an animated GIF from the temporary PNGs that were stored once per second. Upon Completion, the user receives the settings TXT file with the final render time, a CSV full of stats, a PNG of the final painting, and a GIF of the whole painting process.

4.3 Colors

As mentioned above, the program has options for color generation. The program does not use a static list of colors, it generates them at the beginning of the render. A list of integers from 0 to the maximum value of a color channel (255 for an 8-bit example) is shuffled. Each of these values is given to a color worker to generate colors in its own process. We will use an example where our worker receives 53 as its given value. Each worker has

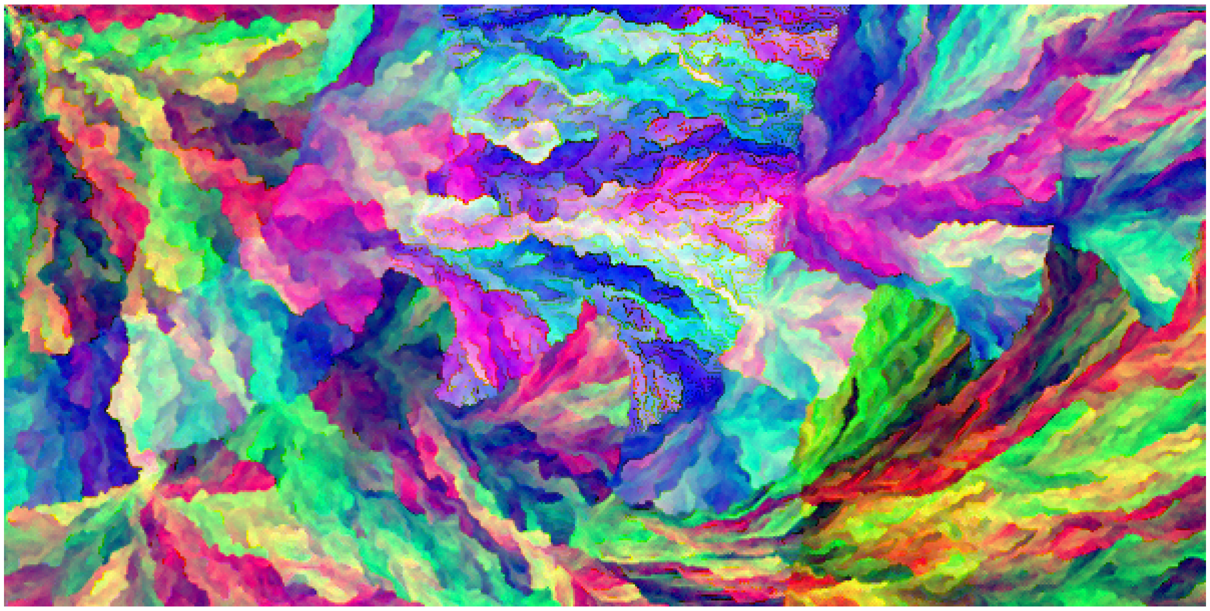


Fig. 18. RGB Average Strategy - Grouped by Channel 1.

2 nested loops that iterate up to the max value of a color channel. In the inner loop, final colors are made and recorded in the form [worker-given-value, outer-loop-index, inner-loop-index]. When the worker is finished, it has a list of colors with one channel having a constant value, for example all RGB colors with an R value of 53. The channel that is held constant is actually selected by the user and does not have to be the first color channel.

The worker then optionally converts the color space. So this list could instead be interpreted as all HSV colors with an H value of 53, and these would be converted to RGB values. Finally, the worker shuffles the resulting list so that the non-constant values become non-ordered, and it returns this list to the main process.

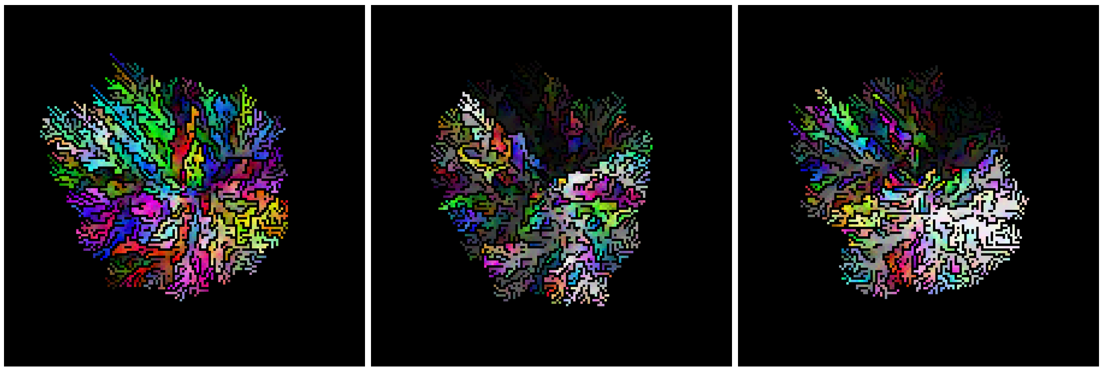


Fig. 19. RGB, HSV, and HLS Randomized Order Renders.

Once all workers have completed their task, the main color process will now have all RGB colors (or all HSV/HLS colors converted to RGB) grouped by value for one color channel. At this point, it does the final shuffle which can be disabled by the user. If this shuffle is done, the order of the final color list is completely random. This type of random output is shown for each color space in Figure 18. If the final randomize is not done, the colors will still come randomly, as ensured by the initial shuffle of the worker's given color, but they will be grouped by this random value. The simplest way to understand this is to consider the HSV color space grouped along various channels; this is shown in Figure 20.

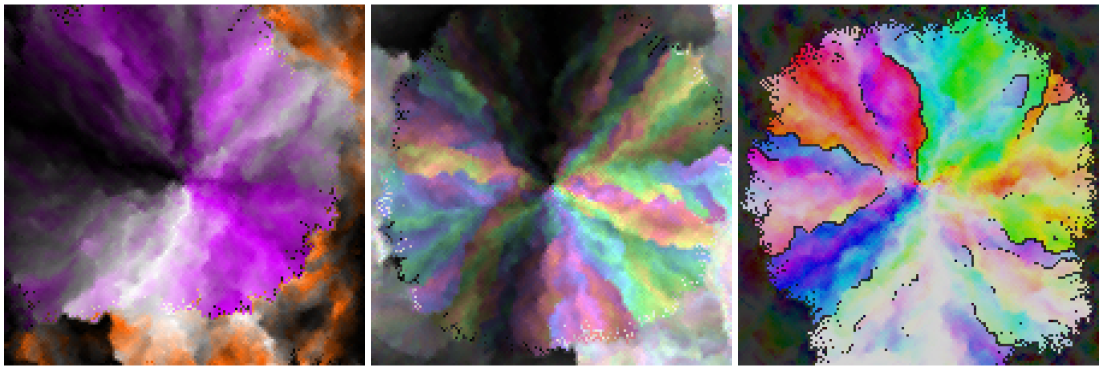


Fig. 20. HSV Neighborhood Strategy - Grouped by Channel 1, Channel 2, Channel 3.

The first image is grouped by channel one, the hue channel. Hue is spectral color; the first color we received was of a magenta hue. Notice how this first, innermost, region sweeps through all saturation and brightness values of magenta. Next the renderer received an orange hue and repeated the process. This is what is meant by grouped, but not ordered. The first hue received could be anything, but you can be sure that every possible color of that hue will be placed before the next hue is received. The second image is grouped by channel 2, the saturation channel. This image begins with a lightly saturated region. It sweeps through every color and brightness value of that saturation level. Then moves on to a very unsaturated value, and so on. The third image is the same but grouped by brightness value.

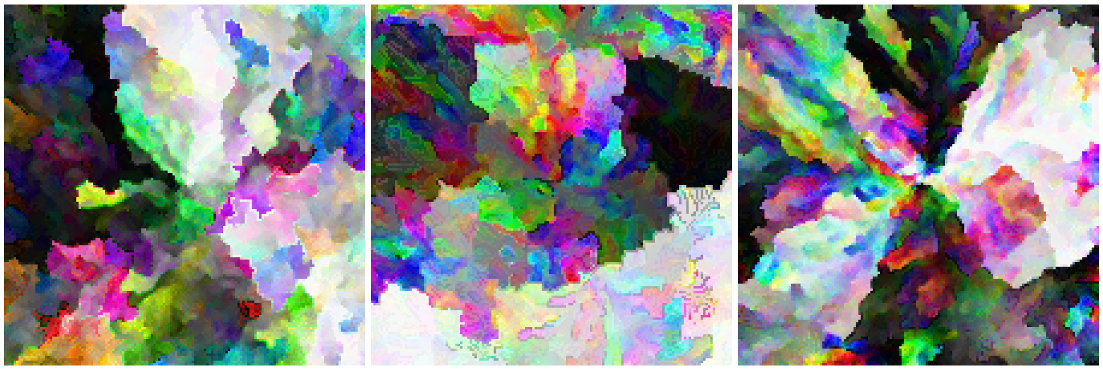


Fig. 21. Average Strategy - Top HSV Color Space, Bottom RGB Color Space

Figure 21 similarly shows 3 renders grouped by varying color channels using an HLS color space. Figure 22 shows 3 renders grouped by varying color channels using an RGB color space. The first image is grouped by constant red values; the second is grouped by green values, and the third is grouped by blue values.

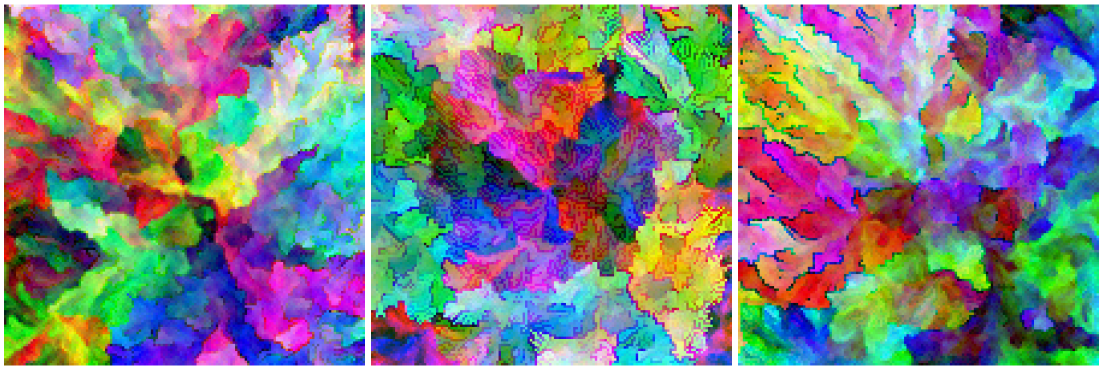


Fig. 22. Average Strategy - Top HSV Color Space, Bottom RGB Color Space

In Figure 23, the each image used a larger bit depth than previous images. This, as a result, increases the size of color groups. In the first image we see just a single hue value used, in the second a single saturation value is used, and in the third a single red value is used.

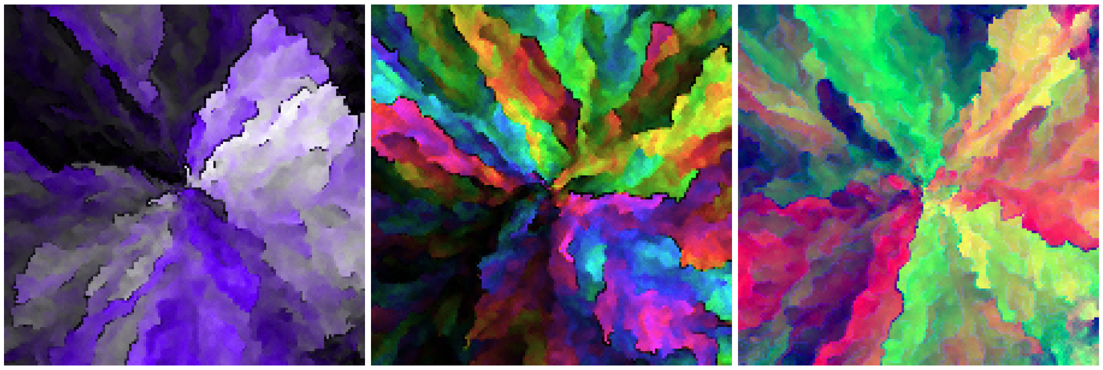


Fig. 23. Average Strategy.

5 CONCLUSIONS

The original goals of this project were finding a data structure that could accelerate the the various best fit strategies, and learning about parallelism, a topic that we have not gotten an opportunity to explore at Cal Poly. Both of these goals and more were achieved, but with some notable shortcomings. For the R-Tree implementation, these shortcomings were an inability to optimize the Average Strategy, as well as the library limitation making multiprocessing with the R-Tree unnecessarily difficult. Despite this, when it can be used, the R-Tree implementation is the fastest and can practically produce the largest images by far.

Through this project we have gained much more familiarity with using parallelism. Some limitations of the GPU implementation include: an unoptimized work-group size, no data type optimization, and no use of specialized GPU texture hardware for passing or manipulating our canvas data. When the R-Tree cannot be used, OpenCL and the GPU greatly extend what can be accomplished with the brute-force implementation. Learning about Numba was beneficial as well. The performance difference of the compiled code was unexpected, but

Numba allows one to do prototyping in Python and then squeeze as much performance as possible out of any bottle-necking functions.

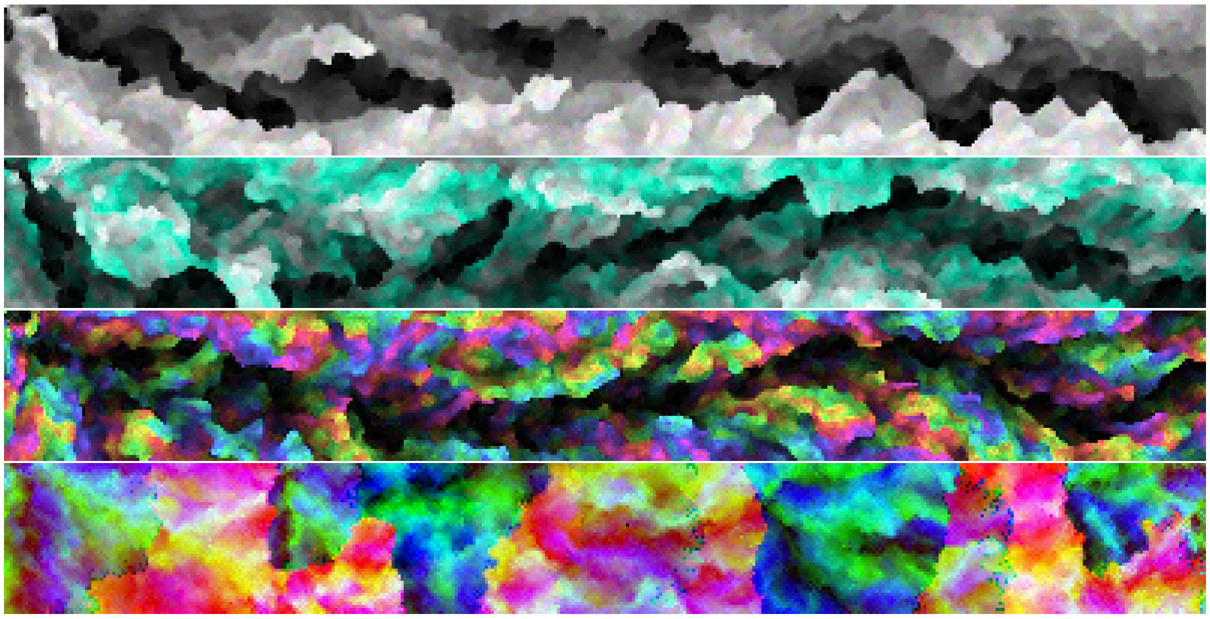
6 FUTURE WORK

Taking what we have learned from this project forward; we plan to re-implement the renderer with a number of changes in the Rust language. Rust is a performance and multiprocessing focused compiled language, that also has R-Tree and OpenCL packages available. Native and safe multiprocessing are strengths of Rust that should allow for and extremely fast parallel approach using an R-Tree on the CPU. It interests us to see how a KD-Tree and Oct-Tree based implementation would perform.

One issue that arose up regularly during development was the schema used for storing colors. They were stored as unsigned integers in Python lists or NumPy arrays. The PNG writing module expects 8-bit unsigned integers, but we wanted to keep track of them internally as at least 16-bit so that higher color depth spaces could be supported. There were lots of issues with the distance values overflowing 8-bits in the OpenCL kernel and overall many headaches would have been prevented by going with floats clamped between 0 and 1. This would have simplified color generation and many other interactions dealing with color.

This would also be the first step in further optimizing an OpenCL implementation, as GPU architectures are designed to operate on floating point values. We were unable to thoroughly test the effect of work-group sizes. Despite having 64 compute units, the GPU accepts work-group dimensions up to 1024x1024x1024 which makes a single dimension of 256 workers seem tiny. Its unknown if the GPU could be made to utilize a serialized R-Tree. At least for the brute-force approach, all data passed to the OpenCL kernel could have been encoded as visual textures. GPUs have specialized hardware for this, but it’s unclear if that would necessarily provide a benefit.

Overall, this has been a project that required investigation of new methods and techniques to accomplish our overarching goal. The resulting program exceeds the initial scope of the project, but still has ample room for additional improvement. The project successfully generates large, beautiful pixel art tie-die paintings, and served as an excellent instructional tool for learning about parallelization and data structures.



REFERENCES

[1] Tavian Barnes. 2014. *KD-Forests.* tavianator. Retrieved April 26, 2020 from https://tavianator.com/k-d-forests/

[2] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517. https://doi.org/10.1145/361002.361007

[3] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137. https://doi.org/10.1145/356770.356776

[4] Fejesjoco. 2014. *Images with all colors.* StackExchange. Retrieved April 26, 2020 from https://codegolf.stackexchange.com/questions/22144/images-with-all-colors

[5] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. 2002. Quadtree and R-Tree Indexes in Oracle Spatial: A Comparison Using GIS Data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02).* Association for Computing Machinery, New York, NY, USA, 546–557. https://doi.org/10.1145/564691.564755