Printing complex geometry with complex materials

Polyjet Voxel Printing Exploration

Jack Crane *jack.crane@slu.edu*
Saint Louis University Center for Additive Manufacturing *slu.cam@slu.edu*

Contents

SAINT LOUIS UNIVERSITY Center for Additive Manufacturing

SAINT LOUIS UNIVERSITY

1 Introduction & Background

I wanted to be able to print random objects that could be mathematically defined and 3d printed on our PolyJet 3d printing system. Once parts become complex, colorful, or more unique, it is necessary to begin using customized slicers for more advanced solutions. My first objective is to replicate a part that SLUCAM had when I was looking at schools. It was several inches long, about an inch wide, about $\frac{1}{10}$ of an inch thick. It featured a gradient from white to a color, and from flexible to hard as it went along, the white as flexible and the color as hard. After some discussion, it was made clear that the only way to achieve this is to write some code that would generate the voxel cloud that we could then parse into a print job. The GrabCAD voxel printing documentation[1] served as a good place to start, providing background on the printer setup, such as resolution, layer height, colors, and formats.

1.1 Colors

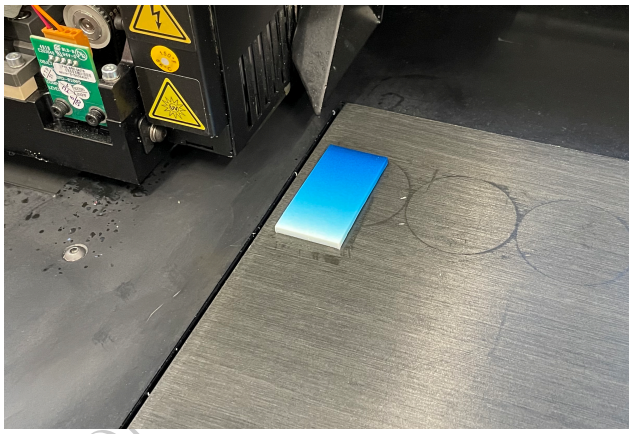
From previous internal explorations and the documentation, I have learned that the printer cannot extrapolate the colors to use from merely an imported image, the image must already be split into colors. The Voxel Print Utility expects a small number of colors to be present in the provided PNG sequence, and assumes materials to use based on colors provided. Color assumptions are made based on this table excerpt:

Base Resin	Red	Green	Blue	Hex
Agilus30Wht	255	255	255	#FFFFFF
VeroClear	227	233	253	#E3E9FD
VeroMGT-V	198	0	88	#C60058
VeroYL-V	240	197	0	#F0C500
VeroCY-V	0	137	166	#0089A6
(Void)	0	0	0	#000000

This table comes from the GrabCAD Voxel Printing Documentation. I have removed information on resins we do not have loaded in our machine for brevity. How we have the printer configured, the slicer expects these (and only these colors) be supplied. While these specific colors are not *required* as colors can be re-mapped from inside the Voxel Print utility, using the assigned colors helps the utility to automatically assume resins without manual configuration.

1.2 Resolution & DPI

The printer is configured strangely in how it handles resolution. The Voxel Print documentation calls out that a 600 x 300 pixel image will serve as a 1 inch square, and a 600 x 600 pixel image will draw a rectangle. I found it hard to understand this configuration, but a 600 x 600 pixel print will print oriented as such:



Beware while designing that once you prepare your PNG layers, you cannot change their orientation in the software. You are able to translate, but cannot rotate your print, so the software must be written to take this into account while slicing.

2 System Design

This is not SLUCAM's first exploration into learning and perfecting the use of the voxel printing methods. Past explorations used Matlab to create the contents, but I chose to use JavaScript in my explorations due to my relative comfort in JavaScript and out of wanting to define methods in another language.

2.1 File & Organization

The voxel print utility expects a folder of PNG files, each with the same dimensions and color schema. The files must be consistently named using a prefix convention and each PNG is a layer. My prints use a 'slice_{layer number}.png' naming scheme, but any file names using a consistent prefix and a sequential value symbolizing the layer height.

```
`./output/slice_${String(i).padStart(3, "0")}.png`
```

[1] https://help.grabcad.com/article/230-guide-to-voxel-printing?locale=en

For ease of management, the files live in an output folder that can be zipped and transferred elsewhere or directed into GrabCAD Print Voxel Print Utility directly.

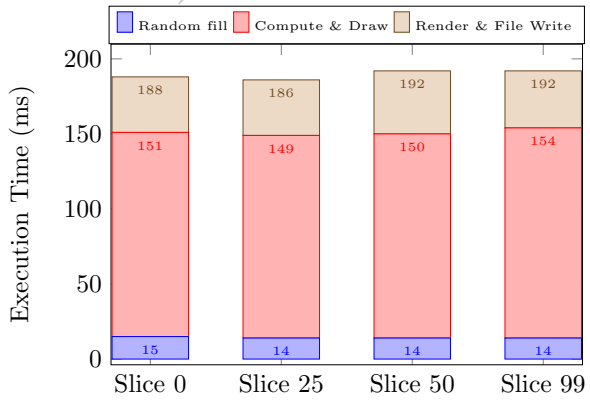
2.2 Language Choice

For this project I elected to use JavaScript to implement these features. Unlike previous work done in SLUCAM I opted to use JavaScript over Matlab. Previous work has been done in Matlab because it is a powerful math/vector language. I opted to use JavaScript because of familiarity and it's relative efficiency thanks to its ability to use the GPU to write image files. (which may be put to the test). Although cross-platform support is not something we care about now, the flexibility and universality of JavaScript made it alluring because of the option to deploy this work to other devices and platforms. Because JavaScript is freely available and non-proprietary, a potential user does not need to have Matlab installed and own a license.

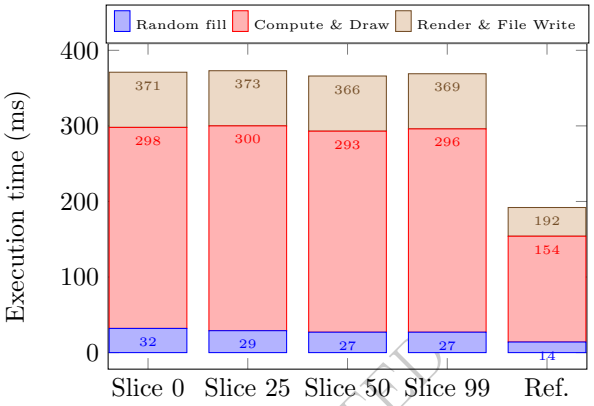
2.3 Performance

While JavaScript may not be the most performant language, especially when dealing with large amounts of data or vectors, it is still able to hold its own against most languages and has nice benefits when datasets become large. Figures 2.3.1-2 highlight how the performance changes across layers. Figure 2.3.1 shows how performance worsens more-or-less linearly with the size of the image. These performance graphs have not been overly systematically generated, but are accurate enough to provide an approximate understanding of how performance will change. Performance was measured on a 2022 M2 MacBook Air with 8gb of memory.

2.3.1 Performance on a 600x600 image

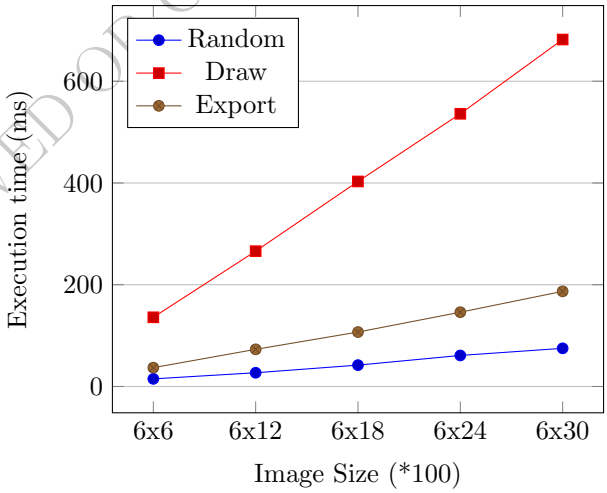


2.3.2 Performance on a 1200x600 image



The "Ref." entry is Slice 99 from the 600x600 image provided for comparison.

2.3.3 Performance on n-sized images



3 Strategy & Algorithms

There are a few sub-algorithms that need designed and strategized to make sure the part can be generated. There were several strategies considered including a "cascading-cell" pixel fill algorithm that would generate recurring, consistent $n \times n$ cells that would hold a predictable pattern of colored voxels, and a "random-fill" algorithm that fills a helper array with random values then uses a loop's progress to determine the color to display on a per-pixel base. I decided to go with the random-fill algorithm because of it's relative simplicity and it's basis seeming simpler and more flexible.

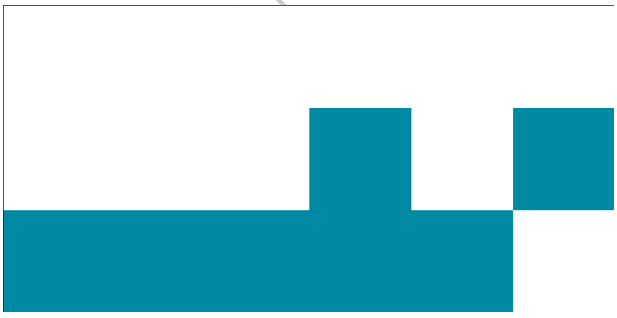
3.1 Gradient Generation

3.1.1 Linear Gradient Generation

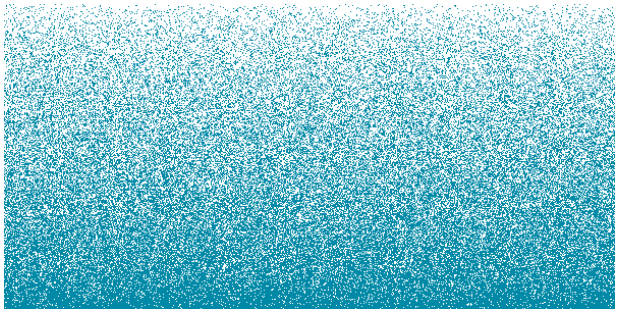
A linear gradient is the simplest and most barebones approach to generating a gradient and is the first strategy I chased. The algorithm worked by creating a "ghost" array that is filled with random numbers uniformly distributed between 0 and 1 (represented as the large non-italicized numbers in the number grid). Then as the loop iterates through the "rows" of the array, it computes it's progress by $row_{current}/row_{total}$, where row_{total} comes from the resolution configuration. This is the small italicized numbers in the grid. Finally, it compares the progress to the random, and if the progress is larger, the pixel gets filled.

0.09 *0.00 <*	0.34 *0.00 <*	0.21 *0.00 <*	0.16 *0.00 <*	0.87 *0.00 <*	0.31 *0.00 <*
0.58 *0.33 <*	0.82 *0.33 <*	0.48 *0.33 <*	0.00 *0.33 >*	0.92 *0.33 <*	0.16 *0.33 >*
0.24 *0.67 >*	0.58 *0.67 >*	0.32 *0.67 >*	0.44 *0.67 >*	0.11 *0.67 >*	1.00 *0.67 <*

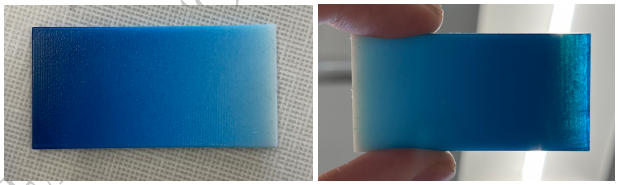
Each square represents a pixel of this 6x3 example. The first pixel (top left) has a random value of 0.09, and a progress value of 0.00. Because 0.09 is greater than 0.00, that pixel does not get colored and is set to the color denoting Agilus 30 White, #FFFFFF. Now looking at the bottom left pixel, it has a random value of 0.24 and a progress value of 0.67. Because the progress value is greater than the random number, the pixel gets filled.



Once the algorithm is defined, the resolution can just be increased and the gradient pixel fill will scale.



However, this highlights a multitude of issues based in color theory and the corrections that need to be applied to force colors to show as intended. This above gradient looks *perfect*. It features a smooth, linear transition from blue to white. However, colors and materials do not mesh like this in the real world, as when this gradient was printed, we were met with this[2]:

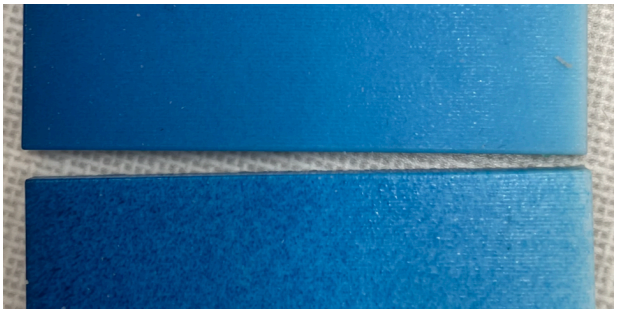


A distinctly *nonlinear* gradient from blue to white. This helps illustrate how the darker blue color overpowers the more gentle white. Color implementations are also affected by the specific materials loaded in our printer, in this case Agilus 30 White and Vero Cyan Vivid. As highlighted by the picture holding the print up to the light, Vero Cyan is slightly transparent and when not backed by white is an extremely dark blue. This is typically solved digitally using a technique called ICC Color Correction to help translate between different color spaces. However, we are not able to use existing color correction techniques without thought and modification because existing color correction techniques are either based on the RGB (or RGB based) color space, or the CMYK color space. Neither of these apply as they are because RGB is *emissive*, and CMY(+K) assume they are being laid on a bright and somewhat reflective white surface. When Polyjet 3d printing, we have neither to work with. The approach we will have to hone in on will be based on the CMY(+W) architecture because those are the colors that are available in our printer.

This gradient may also be pushed into a nonlinear space because the more powerful dark blue is trans-

[2]This print was generated with the same algorithm as the gradient image, but was printed at a 600x600, not 600x300 resolution like the large-resolution image above.

parent and there are subsurface layers that have blue material in overlapping positions and the random pixel fill algorithm generates a randomly seeded gradient on each layer. I Also wanted to experiment with using a consistent random gradient for each layer, where the position of the blue and white voxels would be either the same on each layer (bottom), or randomly shuffled for each layer (top). This resulted in a distinctly different appearance, durometer, and flexibility.

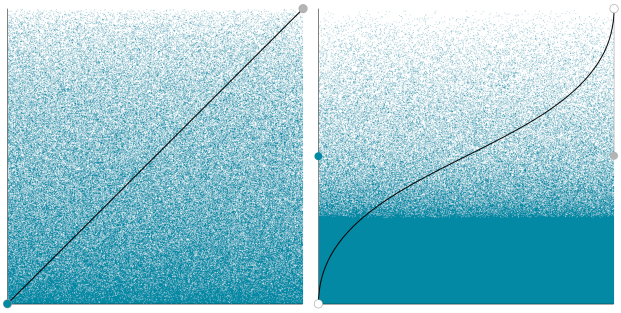


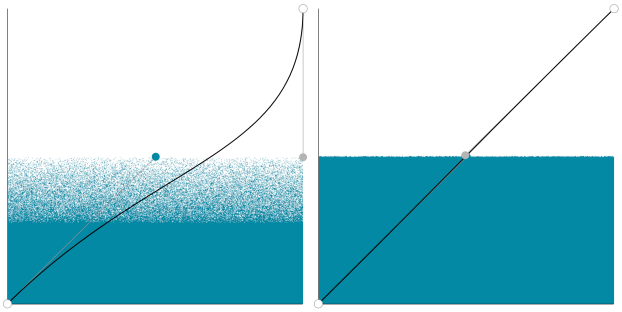
Somewhat unsurprisingly, the part made without the random dispersion was extremely grainy, driven from the fact that there is no offset provided by the colors behind. Interestingly, with eyes closed, their flexibility is nearly indistinguishable.

3.1.2 Nonlinear Bézier-driven Gradient

In my first approach to creating a nonlinear approach to creating a smooth gradient, I decided to take a "stop layer" between a straight linear gradient and a mathematically, color-theory-heavy approach, I wanted to experiment and tweak the gradient generation between print jobs. There are several approaches to this kind of experimentation-friendly program, but I chose to create an implementation of a standard 2-node bézier curve.

Bézier point	Material	Attribute	Default
p_0	Color	Falloff	0
p_1	Color	Basis	0
p_2	White	Falloff	1
p_3	White	Basis	1

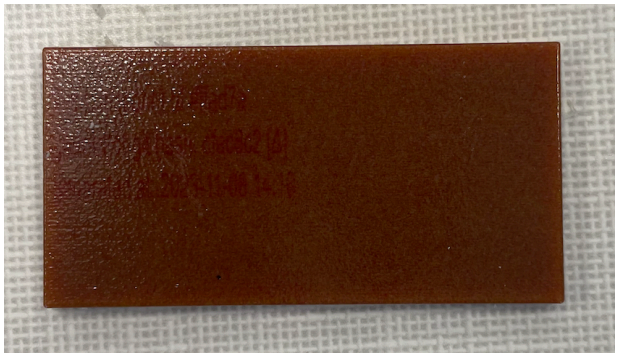




These images are examples of the gradient will react to different curve configurations. These images are based on bézier attributes of *(0,0,1,1)*, *(0,5,1,5)*, *(5,5,1,5)*, and *(0,5,5,5)* respectively. The "vertical" position of each handle (corresponding to p_1 and p_3) controls the *basis*, or the overall strength of the color. Functionally, this tends to influence where the solid-color part of the gradient ends and the transition begins. The "horizontal" component controls the *falloff* of the color, or how aggressively it transitions into the gradient color.

4 Generating more colors

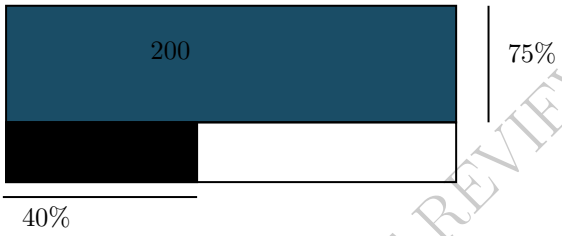
The printer only has the cyan, magenta, yellow, and white resins installed, so if we wanted to print something that is not one of the pre-installed colors (say, #FFAD7A), we need to merge voxels of other colors in ratios that will end up producing the "end goal" color. There are a few steps to getting to these ratios. The result of my initial exploration into generating a part with that color resulted in a partial success.



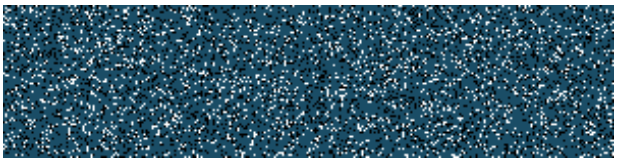
4.1 Converting to HSV

The HSV color space provides the algorithm more context on the objective color because it gives us a hue, which serves as a color attribute without lightness or saturation having come into play, saturation, which serves as a ratio of how much color vs an illumination ratio, and the value, which serves as said illu-

mination ratio. For the sake of explanation, consider the objective color as #1a4d66. The first step is to convert the hexadecimal value into an RGB color, resulting in [26, 77, 102]. The next step is to convert the RGB values into HSV components. I used existing color conversion logic, but the conversion from RGB to HSV is neither complex no novel, so will not be covered in further detail. The output from the HSV conversion turns out to be [200°, 75%, 40%]. In practice, the HSV implementation in a physical ratio format is as depicted below. The first component (the hue) is the position on the outer rim of the color wheel that the objective color falls, or in this case, 200 degrees counter-clockwise from the 12 o'clock position. The saturation, or how strong the color is serves as the ratio of the colored portion to the colorless portion, in this case 75%, meaning that 3/4 voxels should be dedicated to reaching the desired color, and 1/4 is dedicated to the value component. The value, or amount of darkness in the color, is the percentage of "black" to white in the mix, or in this case 40%. In our real slices, these all should be mixed up, but for the sake of demonstration, this graphic shows the relative amount of each color group.



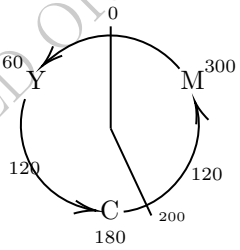
Using the same logic as we used to distribute colors randomly in the gradient, we can go about distributing pixels. This is done by creating the randomly seeded random value array, and (in this case), if that random number is below 0.75, assign the blue color. Otherwise, we go into a nested random checker where a new random number is generated for that pixel. If the random number is below 0.4, the pixel gets assigned black, and otherwise, the pixel is assigned white. The image on the left is an example of the output generated by this algorithm.



This works great except it doesn't actually solve our problem all the way. It is able to normalize the color so that the lightness[3] or contrast of the color is not a factor and allows it to be more granularly controlled in the part, but it still uses the blue color which is not installed, and still uses black to describe darkness, which is likewise not installed.

4.2 Converting from hue to CMY

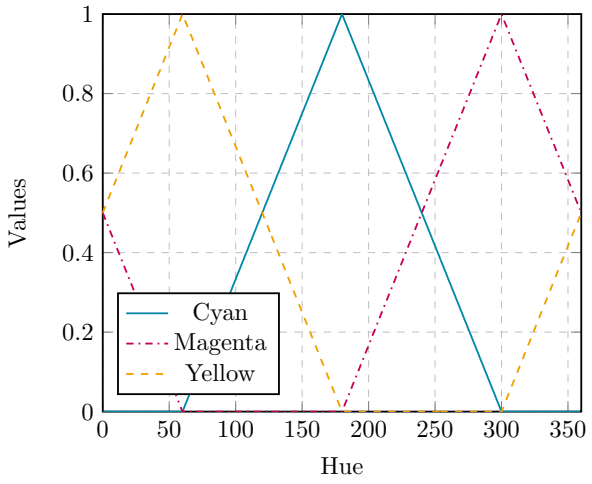
To convert from hue (the first component returned from the HSV conversion) to CMY, it is important to understand the color wheel. This extremely simple rendition of the color wheel highlights the positions of yellow, magenta, and cyan, the degree positions of each, the degree travels between each color, and the position of our color of focus at the 200° position.



The logic for converting the directional position of our color of focus to a ratio of CMY values, we are able to infer a lot from the color wheel to build an intuition. The CMY values for a given position are just a ratio between the closest 2 cardinal colors (yellow at 60, cyan at 180, and magenta at 300). In this case, yellow is the furthest away from 200 and thus will be 0, leaving us to find the ratio of cyan to magenta. The magenta value is set at $(hue - 180)/120$, or in this case $20/120 = 0.167$, and cyan is $1-$magenta, or $1 - 0.167 = 0.833$, leading us to the conclusion that 0% of our true color section should be yellow, 17% should be magenta, and 83% should be cyan, all of which are colors available to the printer. However, there is one more catch: the printer does not have black resin installed, but as any kindergartener can tell you, you can get black from mixing all the colors together in equal proportions.

The color ratios connect to hue positions as follows:

[3]Not to be confused with the lightness component of an HSL color. We are using HSV, not HSL, and while they are similar, it is important to note that anytime lightness or darkness is referenced in this paper, it is in reference to the value component.



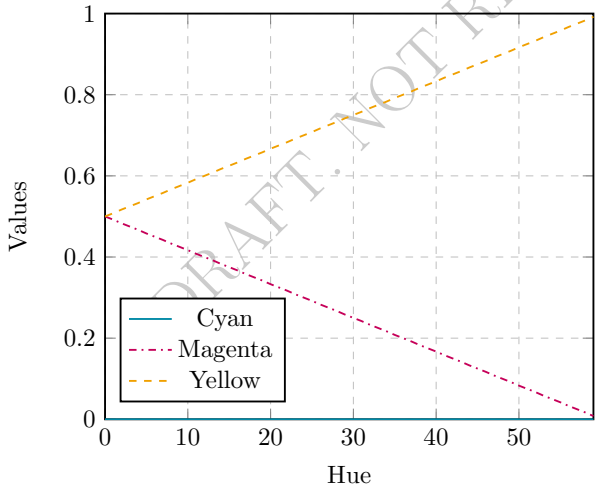
This is achieved by breaking the computation into 4 blocks:

4.2.1 Hue is between 0 and 60

When the hue is between 0 and 60, it is transforming between an equal mesh of magenta and yellow to a fully yellow color set. Cyan will always be zero.

$$magenta = \frac{(60 - hue)}{60} * 0.5$$
$$yellow = 1 - magenta$$
$$cyan = 0$$

When supplied with 0, the function should return [0, .5, .5], and supplied with 60, should return [0, 0, 1]

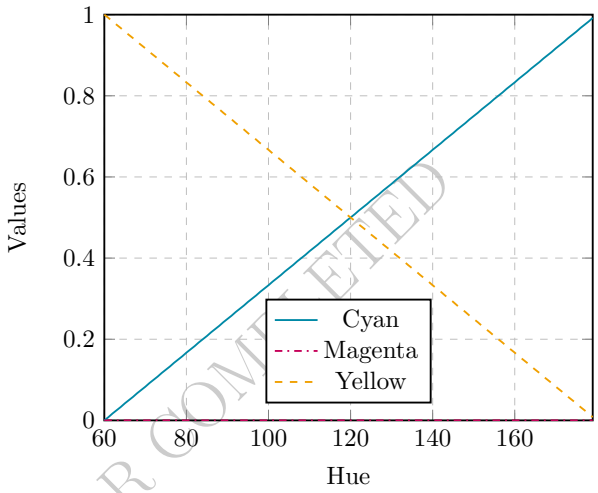


4.2.2 Hue is between 60 and 180

When the hue is between 60 and 180, the colors are transforming from completely yellow to completely cyan. Magenta will always be zero.

$$magenta = 0$$
$$yellow = \frac{180 - hue}{120}$$
$$cyan = 1 - yellow$$

When supplied with 60, the function should return [0, 0, 1], and when supplied with 180, it should return [1, 0, 0]. At 120, it should return [0.5, 0, 0.5].

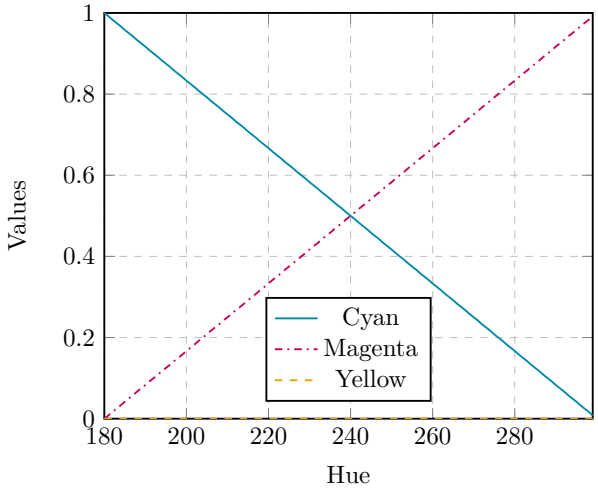


4.2.3 Hue is between 180 and 300

When the hue is between 180 and 300, the colors are moving from completely cyan to completely magenta.

$$cyan = \frac{300 - hue}{120}$$
$$magenta = 1 - cyan$$
$$yellow = 0$$

When supplied with 180, the function should return [1, 0, 0]. At 300, it should return [0, 1, 0], and at 240 it should return [0.5, 0.5, 0].

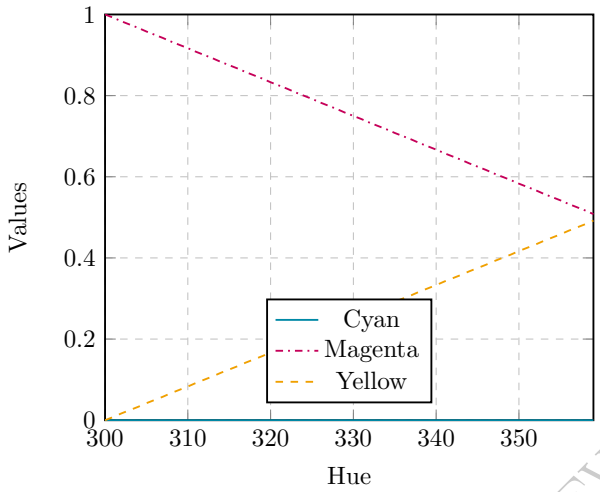


4.2.4 Hue is between 300 and 360

When the hue is between 300 and 360, it is moving from completely magenta to an even split between magenta and yellow.

$$magenta = \frac{420 - hue}{120}$$
$$yellow = 1 - magenta$$
$$cyan = 0$$

When supplied with 300, the function should return [0, 1, 0]. 360 should be [0, 0.5, 0.5], and 330 should be [0, 0.75, 0.25].



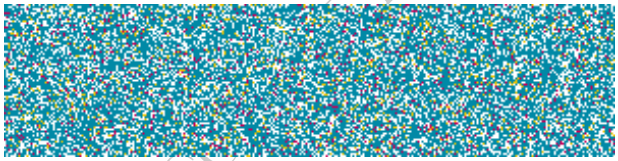
4.2.5 Code Implementation

```
// Context for this code block available within GitHub commit hashed 21858d9
export const hueToCMY = (hue) => {
  hue = hue % 360; // Normalize the hue to be within 0-360 degrees
  let c = 0, m = 0, y = 0; // Initialize variables to hold color attributes

  if (hue < 60) {
    c = 0;
    m = ((60 - hue) / 60) * 0.5;
    y = 1 - m;
  } else if (hue < 180) {
    m = 0;
    y = (180 - hue) / 120;
    c = 1 - y;
  } else if (hue < 300) {
    c = (300 - hue) / 120;
    m = 1 - c;
    y = 0;
  } else {
    m = (420 - hue) / 120;
    y = 1 - m;
    c = 0;
  }

  // Ensuring values are between 0 and 1
  c = Math.max(0, Math.min(1, c));
  m = Math.max(0, Math.min(1, m));
  y = Math.max(0, Math.min(1, y));

  return [c, m, y];
};
```

4.3 Filling a solid color

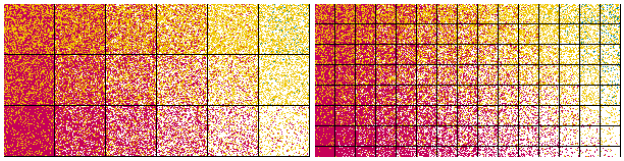
The fill algorithm extends on the algorithm defined in 4.1, but instead of simply assigning blue to any pixel who's random value is less than 0.75, we have to randomly distribute cyan and magenta in the ratios calculated above. This is accomplished by generating another random number, and if it is less than 0.83, a cyan pixel is placed, and if it is greater than 0.83, a magenta pixel is placed.

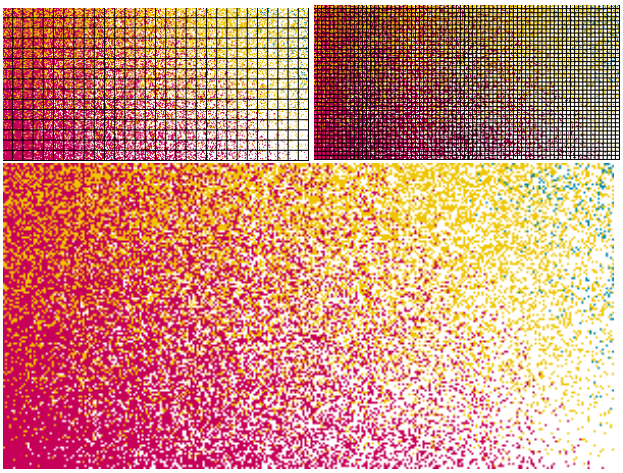
We use the exact same method to fill the required black sections, but rather than placing black pixels, generate another random number, if it is less than 1/3, place a cyan pixel, between 1/3 and 2/3 a magenta pixel, and over 2/3 a yellow pixel. This will generate the "final" slice image that is ready for printing. The final step is to generate a ton of them and to compile them in the Voxel Print Utility.



After printing 100 randomized layers generated by this algorithm, A dark blue swatch came out of the printer.

At a simple level, this works just like the gradient in how it runs through data and uses random numbers to place colors. Originally I had thought I would need to break whatever image or part into groups of pixels, but after exploring with where the best threshold is, I discovered that I could use 'groups' of 1x1 pixels, and the random number-based color generation strategy would take care of getting the accurate color. In order, these images are explorations of the end product of 50x50, 20x20, 10x10, 4x4, and 1x1 pixel groups:





4.4 Color Accuracy & Error

The original gradient swatches in section 3.1 look decently accurate to the generated color slices, but as the colors get more complex, the appearance of the swatch and slice begin and continue to diverge. There are many sources of error in the current strategy.

4.4.1 Lack of color calibration

Printers and screens have color calibration logic that shifts hues such that they look accurate to their theoretical counterparts. This can be solved by attempting to implement the ECC calibration values provided by Stratasys, but on the short-term will be accomplished by printing a color wheel and comparing the physical color wheel to a real-world theoretical color wheel. This should result in a hue rotation value that will give us some level of "easy" color calibration.

4.4.2 Change in color space behavior

Most existing color translation algorithms are designed for inkjet style paper printers where 2d ink is deposited on a comparatively extremely bright and reflective white surface, but Polyjet prints darkening dyes on a void, causing the colors to be driven darker.

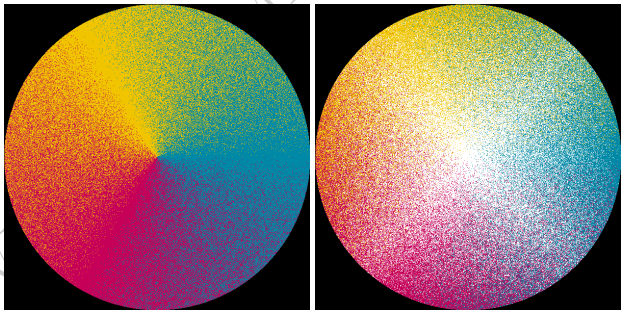
4.4.3 Challenges in predicting an outcome

Due to the fact that screens are "light-additive" and prints are "light-subtractive", a print looks nothing like how it will come out when looking on a screen.

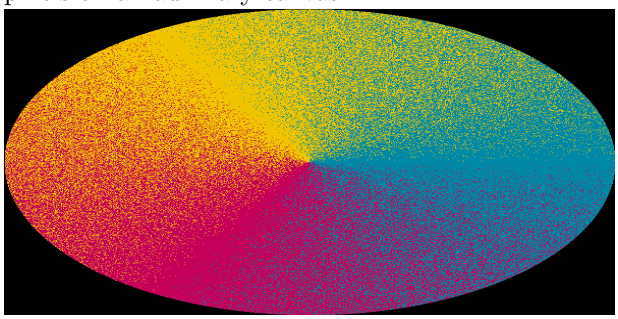
5 Generating colorful parts

5.1 Generating a color wheel

Once the basis for printing colors was developed, it was time to move into blending colors and learning more about their interactions with each other. My strategy for generating the color wheel is to iterate through each pixel and convert it from cartesian coordinates to polar coordinates relative to the center of the part. This returns our radius and theta (θ) that can be almost directly converted to colors. After ensuring the radius is less than the shortest distance from the edge of the square image to the center (this ensures the color wheel is round), we can use the radius as our saturation and the θ as the hue in degrees. After using the solid color fill algorithm on each pixel, we have a "perfect" color wheel.



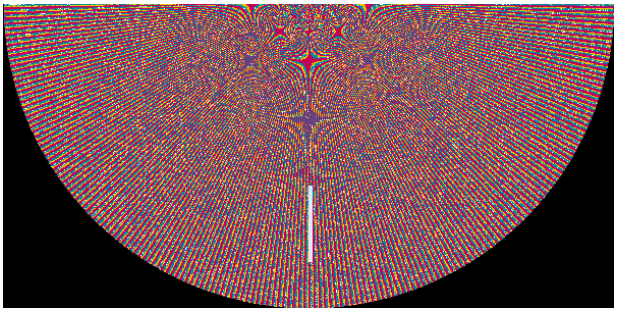
There are 2 color wheels pictured: one with no consideration put towards the saturation and one who's saturation ranges from 0 to 1 moving from the center to the outer edge of the circle. The final step was to double the width to match the printer's DPI expectations by iterating through each pixel and writing 2 pixels on an auxiliary canvas.



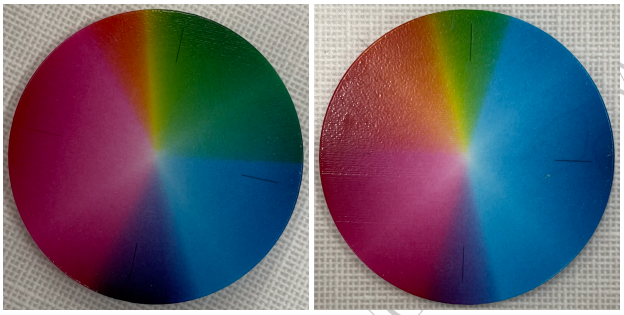
5.1.1 Challenges

Although theoretically simple, generating the color wheel was an extremely challenging task to accomplish. Between the theoretical basis of colors and it being difficult to properly predict the outcome of a test before printing it, we ended up with a few

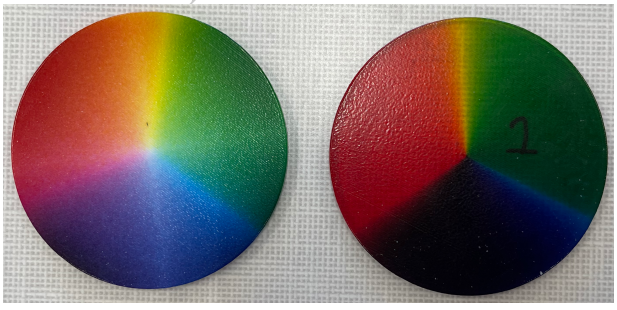
unsuccessful prints and a few interesting artifacts of generation like wild fractals generated by an order-of-operations issue.



I also battled mild logic and implementation errors, leading to a multitude of strange errors including a few order of operations errors including setting $magenta = \frac{(60-hue)}{60*0.5}$ in 4.2.1 which led to incorrect ratios being generated. These errors were extremely difficult to diagnose because they were not error bugs but were issues in the very heart of the algorithms. Issues like this led to color wheels that looked *close* but not quite there.



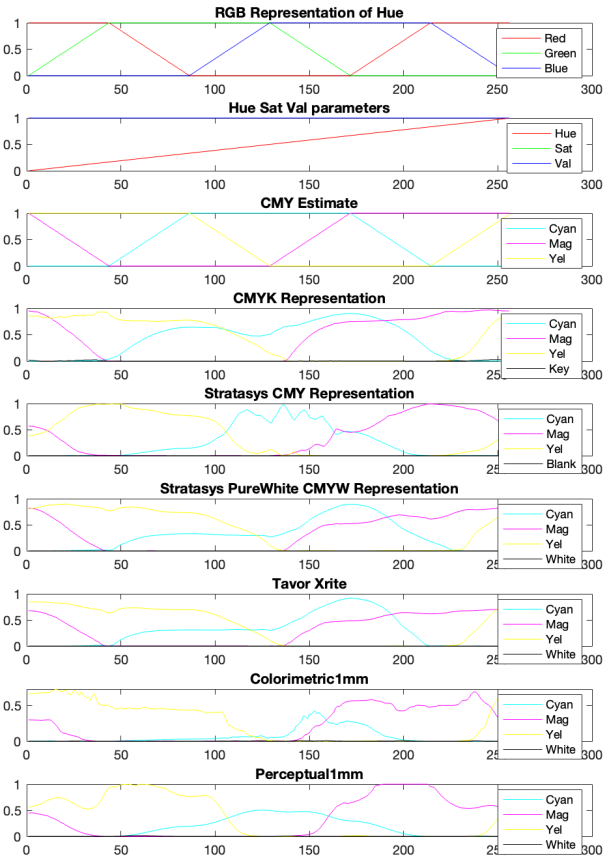
This is close, but is still problematic. After a ton of work exploring and debugging the color conversion algorithm I discovered I had made a few mistakes implementing the logic, but that still resulted in an imperfect wheel:



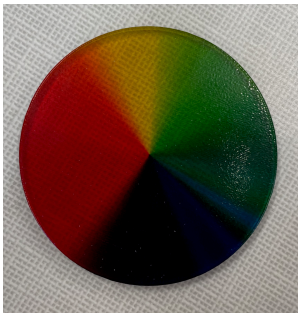
The first has a little opaque white mixed in and the right is pure color. This is clearly much closer but is still not quite good enough. Because we were convinced that the algorithm and implementation were perfect in their own scope, but clearly the scope was lacking.

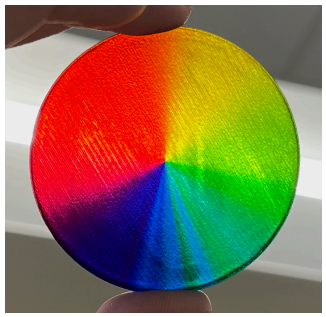
5.1.2 Enter color correction

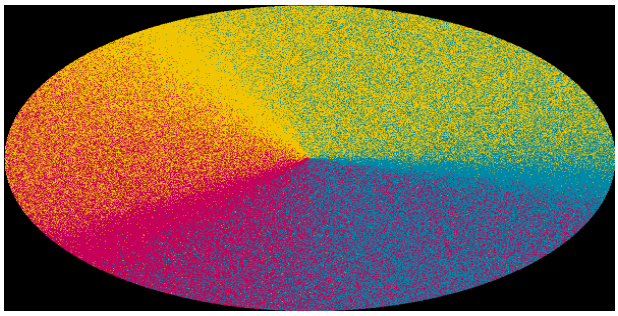
After more exploration, I found out that there are color correction (ICC) profiles made by Stratasys for the Polyjet system with different materials installed, and after plotting the lines in Matlab it is obvious that there is some error introduced by color handling:



On these graphs, the y axis is the amount of the given color required to meet the hue as specified by the x axis (scaled from 1-360 to 0-255). I attempted to use the "Stratasys CMY Representation" to build a wheel because it most closely matched the materials we were using, but it's result left a lot to be desired:

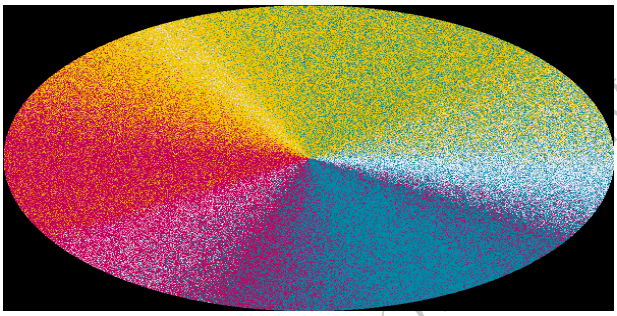






Notice the cyan banding in the blue section and how it reflects the cyan spikes in the "Stratasys CMY Representation" from about x=110:170. This is confirmation that the translation between the ICC profile and the codespace works, but that the implementation is still incorrect.

We attempted to use a different profile, the "Stratasys PureWhite CMYW Representation" as the CMYW color space most closely matches the materials we have loaded in our machine. By applying the color correction curve, we got the following image:



Because we are trying to avoid usisng white while figuring out the color blend logic, we normalized each pixel's color such that it sums to 1:

$$sum = red + blue + green \tag{1}$$

$$red = red/sum \tag{2}$$

$$blue = blue/sum \tag{3}$$

$$green = green/sum \tag{4}$$

This turned out to what looked like a promising slice file

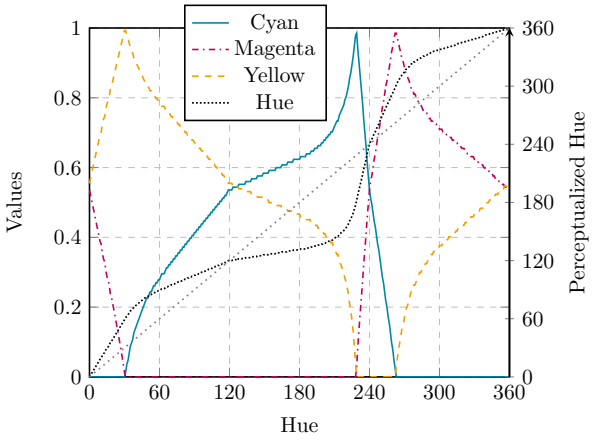
Again this did not work and left us with a disc with mostly green and dark purple, leaving almost no blue and yellow.

5.1.3 Perceptual color space

A UCSB presentation[4] provided some guidance on converting between color spaces and we used some coefficients provided in a section dedicated to converting from color to black and white. The thought is that this will help us account for the difference in strengths of colors without attempting to implement some complicated algorithm to match a graph. From here we can grow into further complexity. The presentation provides the following equation to get a luminance value from a RGB triple:

$$Y = 0.299R + 0.587G + 0.114B \tag{5}$$

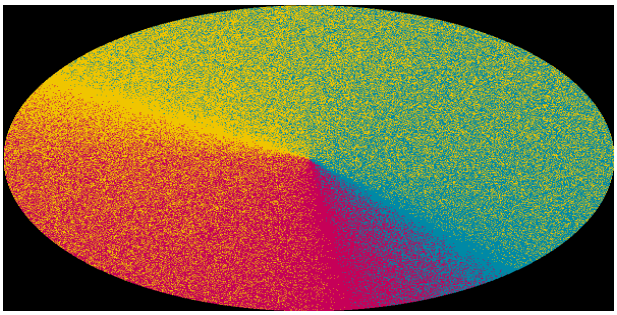
I changed the color wheel algorithm to incorporate these ratios by converting the hue to a RGB triple, multiplying each value by its coefficient, then converting back into hue, then continuing with the original color decision logic.



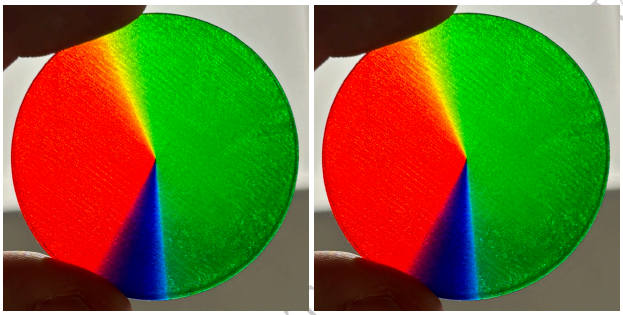
This frustratingly busy graph shows the conversion from the perfect theoretical hue (shown on this graph as the straight grey dotted line). This line goes from 360 to 360 degrees and serves as a reference for what

[4]https://sites.cs.ucsb.edu/ yfwang/courses/cs181b/ps/Dis6.pdf

the perceptualized hue (shown on this graph as the black densely dotted line). On this line, a point's x-position serves as the 'input' value to the hue conversion function, and it's y-position is the conversion function's output. From there, the colors are following the same hue to CSV function written in the beginning of this project. Graphed here are the CSV outputs of the conversion function when given the perceptualized hue. We used this to generate a wheel that looked like this:



Looking at it now, we we can predict that there will be very little blue and yellow but lots of orange and a ton of green. We still gave it a shot and printed it with relatively unsurprising results:



While obviously far from an ideal result, there is no banding and the colors do look solid and they look very true to the slice file provided. A postmortem of this print realized what went wrong. Looking back to equation 5 from the USCB presentation, there is a pretty obvious result. The coefficient on red (0.299) has a very obvious realization in the part: it is roughly 29.9% red. Even more obvious, the part is 58.7% green, leaving about 11.4% to blue.

5.2 Printing a photograph

After printing a color wheel, the next logical stop is to print a photograph using the Polyjet system. I picked a photo from my camera roll that showcases lots of colors and is a good picture, but does not have too many tiny details that could be problematic for an early stab at printing photographs.

By continuing to use the same functions defined in creating the color wheel, it became extremely simple to create new implementations, and resulted in a prepared image in very little time: