Portland State University
PDXScholar

University Honors Theses | University Honors College

5-23-2019

Glitch Art as an Expression of Medium

Maxwell Ettel
Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: https://pdxscholar.library.pdx.edu/honorstheses

Recommended Citation
Ettel, Maxwell, "Glitch Art as an Expression of Medium" (2019). *University Honors Theses.* Paper 736.

10.15760/honors.753

This Thesis is brought to you for free and open access. It has been accepted for inclusion in University Honors Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

Glitch Art as an Expression of Medium

by Maxwell Ettel

An undergraduate honors thesis submitted In partial fulfillment of the

requirements for the degree of

Bachelor of Science

In

University Honors

and

Graphic Design

Thesis Advisor:

Stephen Lee, M.F.A.

Portland State University

2019

1. Theory

To fully understand and effectively utilize something, you have to understand how it works. The path to comprehension can be paved in many ways, the specific way that this thesis explores is breakages. Breakages or 'glitches' in digital images expose the underlying identity of digital mediums. A 'glitch' is "any error, malfunction, or problem" (*dictionary.com*) in a computational system that results from unintended code interaction or damage. Visual glitches have their own associations in our collective cultural discourse, due to their unique and visually stark appearance.

> "Perspectives on damage to human artifacts have typically fallen into two categories: That of the ideal pristine artifact, in which damage is problematic and to be avoided or repaired; and that of the entropic artifact, in which damage acknowledges mortality, temporality, and serendipity" (Kilker, 52).

Kilker denotes these distinctions in relation to analog photography, in the context of digital imagery they also hold merit. Typically the goal of digital imagery is to create an artifice of reality, where any break in the artifice is undesirable. Counter to this ideal, a break can be viewed as an acknowledgment of the digital foundation of the image. In all, the identity of the image as a digital object is revealed through the break in the visual artifice.

In his work *Camera Lucida,* Roland Barthes introduces and defines the concepts of *Studium* and *Punctum. Studium* is the fundamental human understanding of an image, while *Punctum* is a superseding element that captures an individual viewers attention (Barthes). *Punctum* is a fairly fickle thing due to its very subjective nature, whereas *studium* is involved more in collective human understanding. In essence, *studium* is inherently related to the visual

and formal identity of an image. The visual identity exists as the subject of the image, it lies within our initial understanding of the image as its subject. In essence the surface level *studium.* For example, the visual identity of a photo taken of a building would be found in the understanding of the image *as* the building itself. The formal identity is found in the fundamental elements of the medium used, or the subsurface *studium* of the image. To put this simply, the formal identity *is* the medium. These identities are closely tied, however the degree of balance between the two depends upon both the image itself and its medium.

A medium is a foundation for the image. In the context of this thesis, a medium is a specific means of artistic expression. The two main delineations of a medium (in reference to imagery) are the physical and the digital. While physical mediums such as oil paint, charcoal, or analog photography are tangible; digital mediums are intangible. Secondarily, examples of digital mediums can be seen in regards to the technical vs the conceptual. Technical digital mediums are primarily composed of the distinctions between raster and vector, which are two fundamentally different ways of producing imagery. Conceptual mediums include: digital photography, generative art, 3D Rendering, etc. Despite the difference in tangibility, both physical and digital mediums are a means of representing a subject through the creation of imagery.

The relationship between the visual and formal identities is dependent in part on the medium. This relationship also impacts the *studium* of the image, through the tangibility of the image itself. When you can feel the texture of something that changes how you understand it. It goes from an artifact that exists only to your eyes to something that is truly present within your world. It is the *studium* that paves the way for *punctum.* The *punctum* comes from how the visual

and formal identities interact in the image. Its basis as a concept comes from the minor details of an image. When we view the strokes of a brush in a painting or the gritty texture of pencil, we see the tangibility of the medium. This tangibility sets the stage for *punctum,* by creating visual elements that reinforce the physicality of the image. In glitched imagery the entropic elements call attention to the intangibility of the medium, through their lack of basis in physical imagery. This exposes the formal identity by displaying the inherent digital structures of the image, something that isn't present in physical imagery. The relationship between the visual and formal identity is impacted by the tangibility of the medium in tandem with the specific imagery involved. Another factor in this relationship can be found in how damage affects understanding.

Kilker writes that visual damage can be interpreted in two ways: the ideal and the entropic. The ideal is expressed through the zeitgeist of modern imagery, focusing on attaining the highest and clearest image resolution. This can be seen through the ever increasing resolution of cameras and digital displays. The idea is to create a near perfect artifice, with minimal visual compression and distortion. All in service of prioritizing the visual identity. Counter to that ideal is the movement towards the entropic image in the form of Lo-Fi (Low fidelity) and Glitch art. Where the idea is to actively find or otherwise generate what Kilker calls "entropic artifacts," which are items / images that have been altered through entropic processes (52). In physical mediums the type of wear is that of age: physical damage / deformations, dirt, fading, etc (Kilker, 53). While in digital mediums the entropic artifacts can come as a result of compression and corruption, two opposing methods that each have their own distinct visual language. The nature of these changes is intrinsically tied to the lack of physicality in the medium.

The movement of artists and creatives towards entropic visuals doesn't come without controversy. As with most countercultural movements glitch art and glitch aesthetics have been adopted into situations wherein their inclusion would otherwise be unwelcome. As Carolyn Kane writes:

> "A glitch or technical error can be used to pose questions and open up critical spaces in new and unforeseen ways. Herein lies the appeal of glitch to numerous artists past and present … glitches have also been quickly appropriated back into dominant fashions and styles, moving from political or social critique to commodity, glitch aesthetics bear a fundamentally antagonistic relation even to themselves" (1).

An example that Kane points to is Kanye West's video "Welcome to Heartbreak," which appropriated compression aesthetics from Paul B. Davis' glitch art, while providing none of the greater artistic elements associated with the original work (Kane 2). The idea that glitch art is inherently antagonistic to itself due to its wide commodification and appropriation is valid, and this validity adds new layers to its formal identity in the form of the understanding of this self-antagonism.

In terms of creating digital entropic artifacts the two main visually distinct and attainable means are compression and corruption. Compression at its very basis is a process of removing redundancies, which are pieces of information that aren't necessary for viewing and understanding the image (Arrora et al. 185). In "Review of Image Compression Techniques" by Aurora et al, they detail three types of visual redundancies: "Psycho-Visual Redundancies, Interpixel redundancy (Spatial & Temporal), and Coding Redundancy" (185). These different redundancies all factor into the methods used in a wide variety of image compression methods.

These inform the codec which compress digital images. However, compression in concept is not limited to digital visuals.

Compression, from an art oriented point of view, has its roots in the introduction of photography.

> "Photography could achieve in a fraction of a second what took hours to paint or draw, and thus painters sought new directions for their practice, beyond and away from classical realism, and into the domain of tonality, abstraction, color, frame, format, and medium…The Impressionist and Post-Impressionist painters in particular followed cutting edge research into optics and argued that all one ever perceived was a series of light and abstract color patches" (Kane 3).

These optics are very similar to compression methods of today, with both being processes of removing information redundant to the human eye. The art photographer Thomas Ruff has a series of photographs (called "Jpegs" quite plainly), wherein he takes easily recognizable photographs and distorts them using jpeg compression methods. Ruff takes then takes these photographs and prints them out at a large size, calling attention to the digital formal identity through the excessive jpeg artifacting on the image.

Fig 1. Thomas Ruff, Work Name Unknown (possibly wf02), Cropped.

Accessed: https://davidcampany.com/thomas-ruff-the-aesthetics-of-the-pixel/

Corruption is a lot more varied in terms of the methods of creation. Corruption occurs when something goes wrong, the specifics of which can vary in many ways. Corruption can not only occur on the digital level, but also the physical. Equipment failure is a great example of this, examples include: storage device failure, weak connection / connection failures in cables, capture device failure, etc. This contrasts compression, which works by intended design, whereas corruption exists counter to intended design. Methods utilized to create corrupted images vary from simple code deletion(see Fig. 2), to specialized software that can process images in a specific way (e.g. Photoshop, Processing, etc.).

Fig 2. A Jpeg corrupted through code deletion. Max Ettel (self), 2014.

Corruption can also be carried out by using software in new ways. An example of this is Audio Based Image Manipulation, a process in which a bitmap (bmp), image is imported as raw data into an audio editor (such as Audacity), where it can be modified with audio filter, or cut and pasted into new configurations (see Fig. 3). Audio-based manipulation works similarly to editing image code, albeit through a means not intended for image manipulation. Overall images

can be manipulated in a variety of different ways, though compressing and corrupting are the primary means by which an image can be glitched.

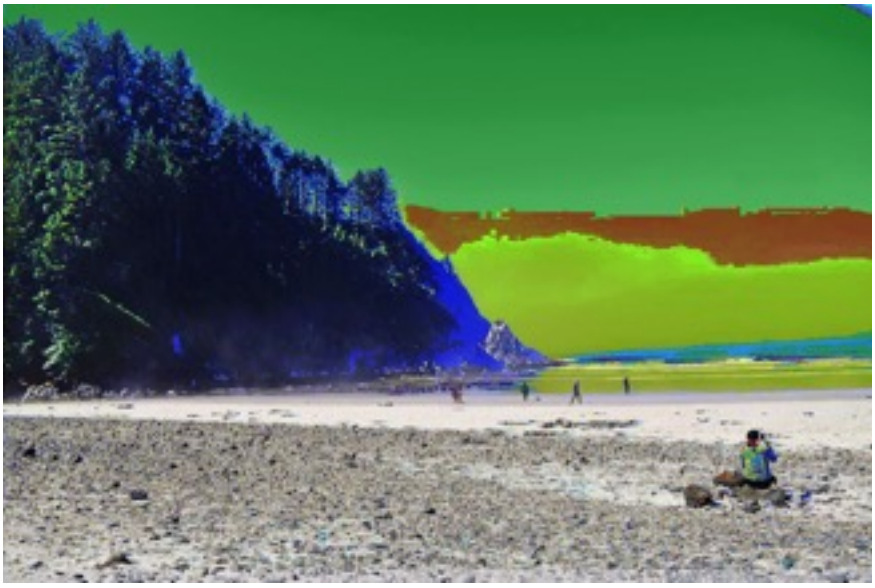

Fig 3. An image corrupted using Audio-based manipulation. Max Ettel (self), 2018.

2. The Project in Code

My goal for this project was to take the concepts and ideas I established and create image processing scripts in p5js that uses them as a creative basis. My approach was primarily to take inspiration from preexisting examples of compression and corruption, and write image processing code that either approximates or takes inspiration from those processes. This led me down several different avenues, the specifics of which will be detailed in section 3.

The foundation of this project is in code. While the results of this will be visual, at the heart of things this project was an exercise in writing p5js code. For the readers who may not be familiar with code (primarily p5js and Processing) the following section will cover the very basics of coding concepts and image processing within p5js.

2.1 Basic Concepts

The primary building blocks of the code are: *variables*, *arrays*, *conditionals*, and *for* loops. These elements are universal between every script that I wrote, and an understanding of how they operate will be essential for understanding the code. *Variables*: a *variable* is at its very basis a value that can be changed (launchschool.com). The specifics of the value can depend on the context in which it is defined. For example in javascript *variables* can store a variety of datatypes. These include: numbers, *booleans*, and *strings*. A *boolean* is primarily a binary value, it can either be true or false (dictionary.com). They are useful if you need to check for a condition that only has two possible states. A *string* is a sequence of characters that will be stored not as units, but as text more often than not (techterms.com). In all, a *variable* can be just about anything that can be singularly stored for use.

Arrays are essentially a list. The list can encompass just about anything, it is primarily a means to store larger blocks of data without using a large amount of separate variables. This project is built off the back of pixel *arrays*, long lists that contain all the pixel values of an image. The *index* of arrays (the number associated with a values location in the array) begins at 0, something which needs to be accounted for when pulling values out of *array*. *Arrays* can also be a way to store objects within the space of object oriented programming (OOP), however that is beyond the scope of this project.

Conditionals are logic oriented lines of code that check for conditions. In this project the *if* conditional was used quite a bit to ensure that the correct action occurred when the required conditions became true. The formatting for an *if* statement looks like so:

```
If (check == true){
 run this code
}
```

There are also *while* Statements, which instead of running code when something becomes true, runs it while something is true.

```
While (check==true){
 run this code
}
```

These can be useful for ensuring that certain things are running congruently with another element. Similar to *while* statements are *for* Loops. The effect of a *for* loop can be achieved with a *while* loop, however *for* loops have built in shorthand that makes them more efficient and easier to code. They are formatted as:

```
for(var variable = 0; variable < maxval; variable++){
 run this code
}
```

The structure of a *for* loops is best summarized by three phases: initialization, condition, incrementation. It begins by initializing a variable and giving it a value, then a *while* check is run against the variable, and if that *while* check is true, it will increment the *variable*s value by the specified amount. The specified incrementation is dependent on what is notated, in the example above '++' indicates an incrementation by 1. This can be used to run code many times within the same frame. Nested (set within each other) *for* loops are used in this project to create the pixel grid for the image.

2.2 p5js script structure

In this section I will be discussing the basic script structure of p5js. No matter the nature of the code, these elements will be ever present in the code. To begin let's discuss the *setup* and *draw* functions. These functions are the primary environment where elements are initialized and run. It is important to note that *setup* is only run once, while *draw* is run on every frame. This means that any code that is constantly updated such as movement or incrementation should be

run in *draw*. While code that only needs to be run once can be written within *setup*. *createCanvas()* will be defined within *setup*, this sets the size of the window for the p5js script. Within draw the *framerate* can also be set. This essentially caps the *framerate*, which is how often *draw* is run. *pixelDensity* can also be set for users of high density displays. This ensures that the resolution of the image matches that of the users display. Within *draw* is where constantly updated code will be written. The primary element to note in p5js is that *setup* and *draw* are essential to any script, if they are omitted the script itself will be useless.

Outside of these primary functions global *variables* can be defined (usually prior to *setup*), so that they can be accessed by both functions and any extraneous custom functions that may be defined for the script.

2.3 Loading & Accessing Image Data

The process for loading and accessing image data within p5js takes several steps. To summarize they are: initialize, access, modify, and update.

```
var img;
var finalimg;
```

To begin, before getting into the primary functions a *variable* named for the loaded image will be initialized, as well as a *variable* for the final image.

```
function preload(){
 img = loadImage('dog200x250.jpg');
 finalimg=createImage(200,250);
}
```

Next a new primary function will be called, named *preload*. This function is used to ensure that the images are loaded before *setup* and *draw* are run. Within preload the image *variables* are redefined as images. The first initial image *variable* is set to a p5.image, which is a special datatype that p5js uses for images (*p5js.org*). This doesn't create a new image, instead it pulls

pixel values and sets them within a pixel array. After the information for the initial image is loaded, the second image *variable* is set to a new blank p5.image.

```
function setup(){
 createCanvas(200,250);
 colorMode(RGB);
 frameRate(30);
}
```

After the image initialization is run within *preload*, *setup* is run. There are no image specific elements within *setup* beside setting the *canvas size*, *framerate*, and *pixel density*. These have little bearing on the final image besides how it's displayed on the webpage.

```
img.loadPixels();
finalimg.loadPixels();
 for (var y=0; y<height; y+=cellsize){
  for (var x=0; x<width; x+=cellsize){
   var index = (x+y*width)*4;
   var r = img.pixels[index+0];
   var g = img.pixels[index+1];
   var b = img.pixels[index+2];
   var c = color(r,g,b);
   finalimg.set(x,y,c);
 }
}
img.updatePixels();
finalimg.updatePixels();
image(finalimg,0,0);
```

Within draw is where the data itself is accessed. Prior to the *for* loops that go through the pixel grid, the pixels for both the initial and final images are made readable and editable. This is done by calling the *.loadPixels()* function. This allows the pixel data to be accessed using *name.pixels[index value]*. Within the *for* loops an *index* value is initialized and calculated. The *index* takes the current *x* and *y* position as defined by the loops, as well as the image width to calculate pixel positions within the pixels *array*. p5js does have some quirks with this, within pixel *arrays* each pixel is comprised of four entries. These values are the *red*, *green*, *blue*, and *alpha* values of the associated pixel. To correctly calculate the *index* for the current pixel the

index calculation must be multiplied by four to correctly read the values. The final *index* calculation looks like this:

```
index = (x+y*width)*4.
```

After the *index* is calculated the *RGB* values are then pulled into their own *variables,* which is done using the pixels *array*. The specific *R*, *G* or *B* value is defined as a variable, and then calculated by reading the images pixel *array* using the *index*. After this is completed, any extraneous calculations can be made to modify the image values within the *for* loop. When those are finished, the variable *C* is defined. *C* is declared as a color data type, and is then set to the final *RGB* values. Then *.set* must be called to write the values to the final images pixel *array*. This is simply done by using dot syntax *finalimg.set(x,y,c)*. The *x* and *y* are the current pixel location as defined by the loops, while *c* contains the final *RGB* values. Directly after all the work in the *for* loops has finished the *arrays* are then fully updated using *.updatePixels()* for both the initial image and the final image. This ensures that the values are fully set within the pixel *arrays*. After that has been completed the final image is displayed within the canvas using *image(finalimage, 0,0)*. This is the basic process for loading and displaying an editable image within p5js.

2.4 Processing Vs. p5js

Initially I was planning on using both p5js and Processing; developing the code in Processing and then porting it into p5js. At the behest of my advisor, I chose to work only in p5js. This proved to be a great idea, due to the intrinsically different ways in which p5js handles image processing vs how Processing handles it.

Both Processing and p5js have marginally different ways of handling pixel data. Processing requires you first define a *PImage*, which is a specific datatype used to store image data within processing (processing.org). After defining a *PImage*, you then point to a local file, whose values are then loaded into the *PImage*. After this has been done the pixel values can be accessed using dot syntax and an *index*. The pixels *array* in Processing is also fairly straightforward, every entry into this array contains the the *RGB* values. This simplifies the means by which you index these values. You only have to think in a very linear manner.

p5js handles pixels differently. As detailed in the previous subsection, p5js does not have *PImage*, instead it uses a *variable* to store the image values. That *variable* is then set to a *p5.image* within the library itself, and not within the script. The pixel *arrays* also behave differently, with individual pixels having 4 entries for the *RGBA* values. Despite the sheer length of the pixel *arrays,* it poses little issue in terms of performance. The methodology of displaying the pixels also differs based on method (which we be detailed in the next subsection), and the final result can change performance drastically based on the method employed.

Overall I found the intrinsic differences between how Processing and p5js handle images initially disorienting. Experience with the nuances of p5js' system proved helpful, and now I find it a bit more intuitive than how Processing handles images.

2.5 .get() vs .pixels[];

In my work on this thesis I have found two different methods for loading and modifying images within p5js, One grossly inefficient, and one fairly efficient. The first of these methods uses the *.get()* function. This function works by accessing the targeted images *array*, *indexing* for the pixel data at the given coordinates, and then saving that out to a short *array*. I would then use

this to get the color values, save them out to a separate *array*, and then use the values that it contains in the image processing. The main issue with using this method is it is not only creating a new *array* on a per pixel basis, but it is essentially doing that twice (alongside any other internal operations that *.get()* conducts). As could be reasonably assumed, this slows the code down by a lot, loading a basic 200x250 image in about 17 seconds.

The *.pixels[]* method is a much faster and more efficient way to access this data. In lieu of using a predefined function to pull the values out of the image, this method directly accesses the images through the *.loadImage()* function followed by the *.loadPixels()* function. Instead of loading small select sections of the image data into an *array*, these functions produce a large *array* containing all values for the image. These values can then be accessed through the *.pixels[] array*. A caveat of this method is that *.set* must be used, which throws current values back into either the original *array*, or a new image. After all of the image has been processed *.updatePixels()* must be called, which essentially glues down all the values that *.set()* threw into the array. After all this processing has been completed the image will have to be displayed, which is done by a simple *image();* function. Overall, despite the few extra priming and finishing steps, this method is significantly faster than the *.get()* method. It loads images in a little over a second.

The main reason why I know of these two different methods is because *.get()* was a necessary evil for my work for a long while. Which was due to the lack of documentation in the specific ways that I wanted to process and display an image. While now I have a better understanding of it, initially I had little idea of how to get it to work in the way that I wanted it

to. It was only after I had developed the majority of the image processing code that I worked out the *.pixels[]* method.

3. The Scripts

The following section will go over the p5js scripts that I wrote for this project. I will be going into the specifics of how they work in regards to how they modify image data.

3.1 Distortion

Fig 4. An image before and after being run through the distortion code.

This code began with a simple question, how can I take the data from one image, and use it to modify another. The visual result of this question came in the form of distortion. This distortion took varying forms based upon the image referenced. My initial tests of this code were using basic gradient images to displace the pixels, however I soon found the best results came from using more complex photos. My focus in this code was to unravel the barrier between formal and visual, using not gross approximations of physical distortion methods, but by using pure raster data to create the visual result.

The code itself is a rather simple modification on the basic image processing code. The main difference in its initialization is that a second image is loaded in the script. For the sake of

brevity I am going to nickname the shift data image the *shiftmap*, and the modified image *img*. Also within the variable declaration is the *dist_limit* variable, this defines the maximum distance that a pixel can be shifted. There is also *shift_direction*, which determines the overall direction in which pixels are moved. This adds a few extra lines of code to the *variable* declaration, *preload*, and *draw* functions.

```
for (var y=0; y<height; y+=cellsize){
for (var x=0; x<width; x+=cellsize){
 var index = (x+y*width)*4;
 var shiftval = (shiftmap.pixels[index]);
 var shift = ceil(map(shiftval,0,255,0,dist_limit));
```

Here we see the initialization and calculation of the *index* variable. Which takes the current *x* and *y* position and converts it into a single value that pulls the correct data from the pixels *array*. Following this we see the initialization and calculation of *shiftval* and shift. *Shiftval* calculates a value from the *shiftmap* image, by averaging the *RGB* values of the corresponding pixel. This is then fed into *shift*, which then *maps* that value from the standard *RGB* color scale (0-255), to a scale of 0 - *distlimit*.

```
var sX;
var sY;
```

These *variables* are then declared. They will store the *x* and *y* values of the shifted pixel once calculated.

```
if (shift_direction == 1){
 sX=x-shift;
 sY=y-shift;
 if (x<shift){
  sX = (width-x)-shift;
 }
 if (y<shift){
  sY = (height-y)-shift;
 }
}
```

This conditional (of which there are two to check for direction), then runs a check to see what value *shift_direction* has, the one shown here checks for 1 and then runs accordingly. *sX* and *sY*

are set to the current *x* or *y* minus *shift*. This is then followed by another set of conditionals, these then check to see if the shifted values will be outside the bounds of the pixel *array*. If they are, it will reflect the pixel across the image.

```
var shiftindex = (sX+sY*width)*4;
var r = dog.pixels[shiftindex+0];
var g = dog.pixels[shiftindex+1];
var b = dog.pixels[shiftindex+2];
var c = color(r,g,b);
finalimg.set(x,y,c);
```

Then *sX* and *sY* are then taken, and a new *index* is calculated. This *index* (*shiftindex*), takes the same calculations from the normal *index*, and instead uses the shifted *x* and *y* values. This *index* is then used to pull the *RGB* values for the shifted pixel, and then sets them to the current pixel location. Finally the *arrays* are updated, and the image is then displayed.

In terms of the theory and ideas that this code represents, my intent was to create an image whose digital identity is displayed through computer originated distortion. While semi-realistic distortion could be produced under the right circumstances, the vast majority of images produced with this code will take on an inherently digital appearance. This digital appearance primarily takes the form of the ghosting that appears when a photographic image is used as a *shiftmap*. When the limit is set to the right amount this ghosting will appear. At one point I felt that it could be a good way to depict a ghost in video, however that has yet to be seen. In all, my intent with this code was to create computer originated distortion, of which displays its digital identity at the forefront.

3.2 Aberration

Fig 5. An image before and after being run through the aberration code.

In the field of photography, chromatic aberration is color fringing that occurs when light rays that pass through a camera lens focus at different points (Nikon). It is an inherently physical artifact of the way that we capture photographs. It carries analog connotations due to its lack of reliance on electronics as an effect. In my previous work I've found that this effect can be emulated fairly easily within digital software. In Adobe Photoshop you can adjust the location of the individual color channels to get sharp fringing (blurring will soften it and give it a more authentic look). My goal with the following code was to take this inherently physical and analog visual artifact, and use it to reinforce the digital elements of the image. This was done through a means that only digital mediums provide, per pixel adjustments.

To begin, the basic image code is initialized as previously described. However new *variables* are initialized, for the individual *RGB* values *lim*, *mode*, and *amt* are set. *Lim* acts as a limit for the amount of per pixel movement. *Mode* determines if x or y is used in shifting calculations. *Amt* is set as the maximum value for a random number generator (*RNG*), which determines how exactly the values are shifted.

```
for (var y=0; y<height; y+=cellsize){
 var r_rng = random(r_amt);
 var g_rng = random(g_amt);
 var b_rng = random(b_amt);
 for (var x=0; x<width; x+=cellsize){
```

Here we can see where the random values are generated. As you can see, as opposed to running on a per pixel level, they are instead run only when *y* increments. This lends consistency to the shifting in the final product.

```
var r_shift;
 if (r_mode ==1){
  if (r_rng<1){
   r_shift  = ceil(sin(x)*r_lim);
  }
  else if (r_rng>1 && r_rng<2){
   r_shift  = ceil(cos(x)*r_lim);
  }
  else if(r_rng>2 && r_rng<3){
   r_shift  = ceil(tan(x)*r_lim);
  }
  else{
   r_shift = 0;
  }
```

Here we can see the conditionals that calculate how much the pixel is shifted. Please note that I am only showing the first set of conditionals, they are all repeated twice for the *RGB* values. The first *if* statement checks for the mode, if passed it will then check for the value generated by the *RNG*. Here we can see, depending on the value generated, the shift value is calculated with either *sine*, *cosine*, or *tangent*. However, if the *RNG* generates a value that is not any of them, it will then set shift to zero.

```
var rX = x-r_shift;
var rY = y-r_shift;
```

After shift is calculated, *rX* and *rY* are then initialized and calculated. *R*, *G*, and *B* all have different ways they apply shift. *R* subtracts its shift, *g* adds its shift, and *b* subtracts its *x* shift, and adds its *y* shift.

```
var index = (x+y*width)*4;
var r_index = (rX+rY*width)*4;
var g_index = (gX+gY*width)*4;
var b_index = (bX+bY*width)*4;
var r = dog.pixels[r_index+0];
var g = dog.pixels[g_index+1];
var b = dog.pixels[b_index+2];
var c = color(r,g,b);
finalimg.set(x,y,c);
```

After all the shift calculations are completed. The *indexes* are then set. First the normal *index* variable is calculated. Then *indexes* for *R*, *G,* and *B* are calculated using the shifted *RGB* values. Then *RGB* is then pulled from the pixels array, and set for the current pixel. After that has completed, the *arrays* are updated and the final image is shown.

The idea in this was to take something that results from a physical process, and make it digital. To display the underlying structures of a digital image by taking both the ways it composes imagery both structurally and chromatically. All digital imagery is inherently composed of color channels. If they work as intended they aren't noticeable. However, if you shift them, either by a little or by a lot, it will begin to expose the digital nature of the image.

3.3 Cell

Fig 6. An image before and after being run through the cell code

The inspiration for this code came from the way in which most digital compression methods operate. Instead of only processing on a per pixel basis, the code delineates cells that contain the pixels. This allows for changes to be made above the pixel level and below the global level. This method of image processing allows for a multitude of different applications. I had ideas that ranged from adaptive resolution, to code that would merge the images in cells. However, the specifics of those evaded my capabilities (though with time I may be able to pull them off). Instead in this section I will go over how the cell shifting code operates. Cell shifting takes the base cell processing code, and shifts the cells over by a random amount, this takes the image and exposes the underlying structure in the form of the cells.

After the basic image processing code, the *for* loops are run.

```
for (var j=0; j<height; j+=cellsize){
 prevCellsX = 0;
 for (var i=0; i<width; i+=cellsize){
```

Between the *j* (*y*) and *i* (*x*) loops, *prevCellsX* is set to zero. This *variable* stores how many previous cells there have been before the current loop. Setting the number of previous horizontal cells to zero ensures that it calculates them out correctly before the *X* cells for that particular row are run.

```
var xShift = ceil(random(move));
var yShift = ceil(random(move));
```

While we are in the first layer of *for* loops, *xShift* and *yShift* are calculated using a random value. Which is then set to an *integer* (whole number) using *ceil*. *Ceil* is a function that takes the value given and rounds it up to the nearest whole number.

```
for (var cY=(prevCellsY*cellsize); cY<(prevCellsY*cellsize)+cellsize; cY+=px){
for (var cX=(prevCellsX*cellsize); cX<(prevCellsX*cellsize)+cellsize; cX+=px){
```

These loops encompass the second layer of the *for* loops. These loops take the information from the cell loop, and uses it to calculate exact pixel position.

```
if (mode==1){
 iX = cX+xShift;
 iY = cY+yShift;
 if(iX > width){
  iX = iX-xShift;
 }
 if(iY > height){
  iY = iY-yShift;
 }
 index = (iX+iY*width)*4;  }
```

This conditional takes the *xShift* and *yShift* values and moves them either by adding them or subtracting them (only mode 1 is shown here, mode 2 subtracts them). There are additional *if* statements within the main one to ensure that no cells will have pixels that *index* outside of the pixels *array*.

```
var r = dog.pixels[index+0];
var g =dog.pixels[index+1];
var b = dog.pixels[index+2];
var c = color(r,g,b);
img.set(cX,cY,c);
```

Here we see the application of the *indexes*, and the setting of color values to the current pixel.

```
  prevCellsX+=1;
 }
 prevCellsY+=1;
}
```

Finally, *prevCellsX* and *prevCellsY* are updated at the end of the cell level *for* loops. After all of the image processing has completed, the *arrays* are then updated and the final image is displayed.

Despite not being able to create more advanced image modification from this concept, I found that the simple shifting of cells creates a compelling effect. The idea is to take this method of processing imagery and display its underlying structure. The relative simplicity of the changes

made point to the very thin line between a perfect and imperfect artifice. Perhaps in the future more advanced methods can be employed to communicate the same message.

3.4 Pseudo Compression

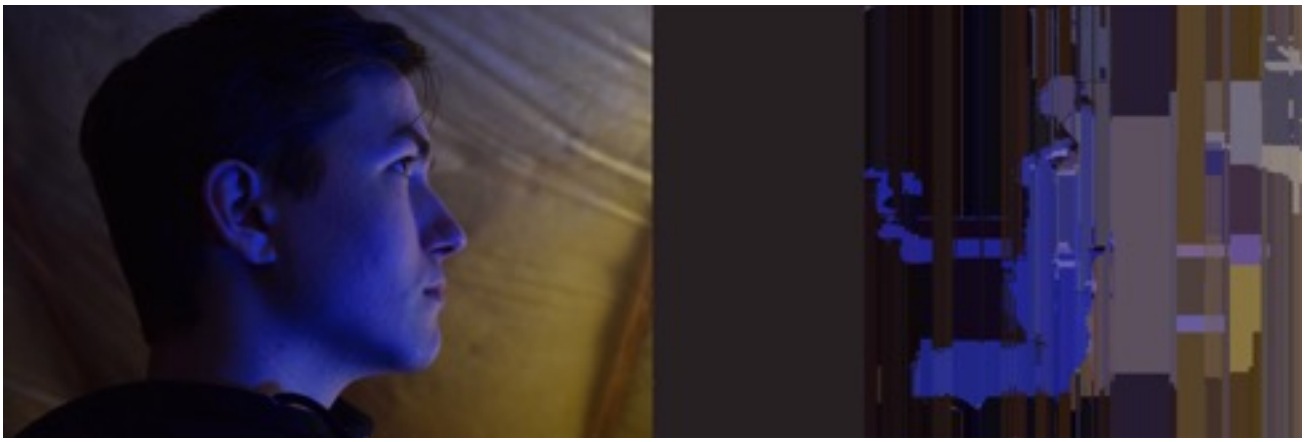

Fig 7. An image before and after being run through the pseudo compression code

This is perhaps some of the most complicated code of this project. The goal was to create a form of compression whose goal is not to remove optical redundancies, but to obscure the original imagery. The initial form of this code simply compressed pixel values either horizontally of vertically. After some development a process to run it both ways became possible, primarily to new methods in how the image data was accessed.

To begin, the image processing was initialized in the standard fashion. New *variables* are initialized globally, these are *thresholds* of various kinds. Depending on the specific version run, it can either use just a basic *threshold*, an *x* and *y threshold*, or individual *R*,*G*,and *B thresholds*. A *boolean* called *copied* is also initialized and set to *false*. This *boolean* ensures that the code knows if it has set the current pixel to the stored color values.

```
var r_prev=0;
var g_prev=0;
var b_prev=0;
```

These *variables* are initialized and set to zero at the beginning of *draw*. They are initialized in *draw* to ensure that the stored values don't carry over when *draw* is run again.

```
var index = (x+y*width)*4;
var r = dog.pixels[index+0];
var g = dog.pixels[index+1];
var b = dog.pixels[index+2];
```

Within the *for* loop: *index*, *R*, *G*, and *B* are initialized and calculated as normal.

```
var r_visual;
var g_visual;
var b_visual;
```

These *variables* are then initialized. They will be used to store the *R*,*G*,*B* values that will be displayed for the current pixel after all the checks have been completed.

```
var r_diff = abs(r_prev-r);
var g_diff = abs(g_prev-g);
var b_diff = abs(b_prev-b);
var avg_diff = (r_diff + g_diff + b_diff)/3;
```

Here difference values are calculated. These take the stored *prev* values, and subtract the current *RGB* from it. They are then calculated as an *absolute.* This is done so that a positive value is always returned from the calculation. These are then averaged to get an average amount of difference in the *RGB* values.

```
if (avg_diff < threshold){
 r_visual = r_prev;
 g_visual = g_prev;
 b_visual = b_prev;
 copied = true;
}
else {
 r_visual = r;
 g_visual = g;
 b_visual = b;
 copied = false;
}
```

Following the calculation of *avg_diff*. A conditional is run where it checks to see if the difference is less than the *threshold*. If it is less than *threshold*, the visual *RGB variables* are set to the

stored value, and *copy* is set to *true*. If it is greater than the *threshold*, the visual *RGB* variables are set to the original *RGB* values for the current pixel.

```
if (x==0 && y==0){
 r_visual = r;
 g_visual = g;
 b_visual = b;
 copied = false;
}
```

This conditional checks to see if *x* and *y* are equal to zero, which ensures that the initial pixel of the image retains its own value, and that its values are saved to the *prev RGB* values.

```
var c = color(r_visual,g_visual,b_visual);
finalimg.set(x,y,c);
```

Here the current pixel is set to the color values stored in the visual *RGB* variables.

```
if (copied == false){
 r_prev = r;
 g_prev = g;
 b_prev = b;
}
```

This final conditional then checks if *copied* is *false*. If it is *false*, the stored *prev* values are set to the value of the current pixel. The *arrays* are then set. If the version only runs the compression once it will display the image. Otherwise it will load the array of the final image and then run it through again with differently oriented *for loops*.

With this code I set out to take a process that works to optimize both file sizes and optical clarity, and warp it. The final result creates imagery that, depending on *threshold*, will create drastically different results. From discussions with my advisor I found that it is similar to *pixel sorting,* which is a well known glitch art process that sorts pixels into specific orientations. Where that code sorts values, this code abbreviates. It takes the original values and destroys them in the name of pseudo optical refinement. In the end I found this to be my favorite of the scripts, and I hope that in the future that I can continue to iterate on it.

4. Design Rationale

In terms of the website home of this project, I tried to use a design vernacular that avoided the conventional glitch visual language. My approach is to emphasize the imagery, while also employing a style that is both stark and unique. Color is extremely limited, with only black and white being employed. This very limited palette is in place so that the color of the imagery will stand out against the sites design. The typographic style is similarly simple, but with some uniqueness in the header typeface. For the body I used Freight Sans Light, which is a sans serif with moderate x-heights as well as mildly humanist forms. The header text uses the typeface Neplus. This typeface has incredibly strong geometric forms, that are reinforced by tiny spaces and counters in the letterforms. The strong forms of this typeface have a boldness that serves to reinforce the equally bold nature of the imagery of the site.

The layout of the site is also fairly simple. Content is arranged into rows separated by dotted lines. The lines are dotted to give it a feel that a solid line won't. Primarily it serves to reinforce the impermanent nature of the imagery. Content is arranged simply into rows based on what it contains, this is done to ensure that the flow of content is paced well for the user in the face of otherwise chaotic content.

4.1 The Images

Before I close I shall address the images I used in the examples for this thesis. All of the photos you saw above were taken by myself. I have a decent back catalog of photos that have been sitting on my external drive. I went through and chose the ones that I thought would best exemplify what the code does. In the sense that the forms of the imagery will work well with the

changes made by the code. Throughout the development process I did use a different test image, however that will not be displayed here.

5. Conclusion

I began this work with one question: Why do I find glitch art compelling? I don't know if I've fully answered that question, but I definitely have come closer to understanding it. Glitch art is counter to the ideal, it is viewed as exotic in the field of perfect imagery. It is, in essence, countercultural. However, as mentioned in the theory section, glitch aesthetics have become widely commodified. There is something that commodified glitched imagery doesn't have, authenticity. There's something about truly glitched imagery that you can't replicate, even with the methods I developed. In them we peer into pure entropy that exists within our electronic systems. That subconscious acknowledgment of entropy is what makes it so compelling. That is the formal identity of glitched imagery.

Works Cited

Arora, Sunny & Kumar, Guarav. “Review of Image Compression” International Journal of Recent Research Aspects. Vol. 5, Issue 1, March 2018, pp 185-188.

Barthes, Roland. Camera Lucida. Translated by Richard Howard, Hill and Wang, 1988.

Berkenfeld, Diane. “Chromatic Aberration.” *Nikon,* www.nikonusa.com/en/learn-and-explore/a/products-and-innovation/chromatic-aberration.html, Accessed 20 May 2019.

“Boolean” *Dictionary.* https://www.dictionary.com/browse/boolean

Christensson, Per. "String Definition." TechTerms. Sharpened Productions, 2006. https://techterms.com/definition/string Accessed 19 May 2019.

“Glitch” *Dictionary*. https://www.dictionary.com/browse/glitch

Kane, Carolyn L. “Photo Noise.” History of Photography, Vol 40, Issue 2, 2016, 129-145.

Kane, Carolyn L. “Compression Aesthetics: Glitch from the Avante-Garde to Kanye West” InVisible Culture. Issue 21, 2014.

Kilker, Julian. “Digital Dirt and the Entropic Artifact: Exploring Damage in Visual Media.” Visual Communication Quarterly, Vol 16, No. 1, 2009, 50-63.

“Reference: loadImage()” *p5js.* https://p5js.org/reference/#/p5/loadImage. Accessed 19 May 2019

“Variables” *LaunchSchool,* https://launchschool.com/books/ruby/read/variables, Accessed 19 May 2019.