Comp Photography (Fall 2015)

Final Project: Photographic Color Cycling

Kimberly Spasaro
kspasaro3@gatech.edu

1. Background

Color Cycling

The goal of my project is to recreate the effect of color cycling using digital photographs. As Wikipedia explains:

> Color cycling, also known as palette shifting, is a technique used in computer graphics in which colors are changed in order to give the impression of animation. This technique was mainly used in early computer games, as storing one image and changing its palette required less memory and processor power than storing the animation as several frames.

In other words, color cycling creates the experience of motion in a still image by shifting specified colors in the image's color palette. These colors cycle on a loop to create a continuous experience of motion. Joe Huckaby's HTML5 color cycling engine as used on the 8 bit masterpieces of Mark Ferrari served as the inspiration for this project (`http://www.effectgames.com/demos/canvascycle/?sound=0`). I will refer to my project as Photographic Color Cycling (PCC). Python's NumPy and OpenCV libraries were used for this project.

2. Overview

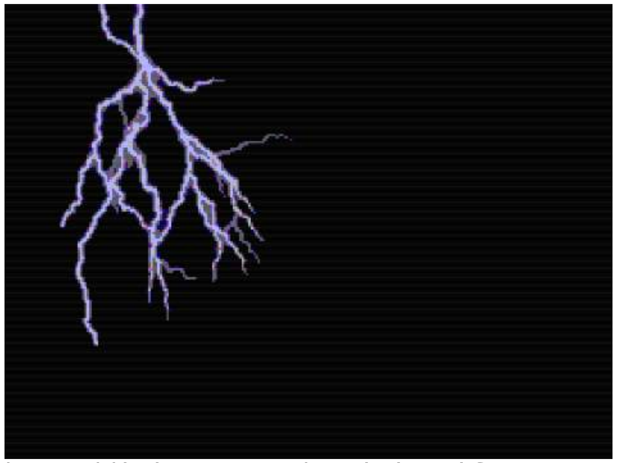
Photographic Color Cycling
The photography pipeline can be understood as:

1. Illumination
2. Optics
3. Sensor
4. Processing
5. Display

PCC alters the processing stage of the photography pipeline. By applying computational processing techniques PCC styles a digital photograph in the manner of an old school video game. PCC, however, does not make true use of color cycling. This is primarily because making aesthetically pleasing color cycling scenes like those of Huckaby and Ferrari requires a large amount of artistic discretion. While Ferrari was able to create pleasing scenes by carefully planning his 8 bit images with color cycling in mind, PCC—due to its nature as a post-processing technique—must work with the input it is given. Thus, a more foolproof means of creating the effect of color cycling in digital photographs is to apply post-processing techniques across a series of photographs over which motion actually occurs. However, the photographer can take some steps to increase the final product's likeness to true color cycling. Because PCC attempts to create a novel image by localizing dynamic change within a scene that is otherwise static, the most desirable images for applying PCC are very similar to the images desired for creating panoramic video textures. The user can best recreate the effect of color cycling by following the guidelines below.

- **Capture a scene with repetitive change** – Color cycling produces the effect of motion by cycling through colors on a loop. Therefore, the photographer should attempt to capture a scene with repetitive motion. For example, a moving stream would capture repetitive change, while a football pass would not.

- **Capture a scene with localized motion** –Because color cycling cannot shift some pixels of a specified color without shifting all pixels of that color, color cycling often localizes motion to one area of an image while the rest of the image remains still. For this reason, the photographer should attempt to capture a scene with motion localized to a single space. For example, the motion of water is generally localized to a small area of space, while a bird flying across a sky is not. The lightning gif linked below demonstrates a scene that does not make use of localized motion, and therefore does not recreate the experience of color cycling.



`http://i.imgur.com/CgIjyjn.gif`

Original Photo
`http://orig08.deviantart.net/fd7f/f/2011/162/9/5/lightning_gif_animation_by_bluehorses-d3in114.gif`

- **Capture a scene with noticeable change** – Because PCC converts images to bit representations or palettes with significantly fewer colors than what are available in modern graphics, it is important that color changes in the digital image are not converted to a single color during PCC. For example, ripples over a colorful reflection are more likely to retain distinct colors after PCC than a waterfall composed of similar shades of white. The photographer may be able to intensify color images for preservation in PCC by adjusting the hue, saturation, or lightness of an image before applying PCC.

- **Create a video texture before applying PCC** – Creating a video texture before applying PCC ensures a smooth transition between cycle repetitions, better recreating an artistically constructed color cycling scene. Creating a video texture which maximizes smoothness between neighboring frames will also prevent choppy frame changes, helping to maintain the appearance of localized motion and further increasing PCC's likeness to true color cycling. The image below demonstrates how using a video texture helps to recreate the effect of color cycling.

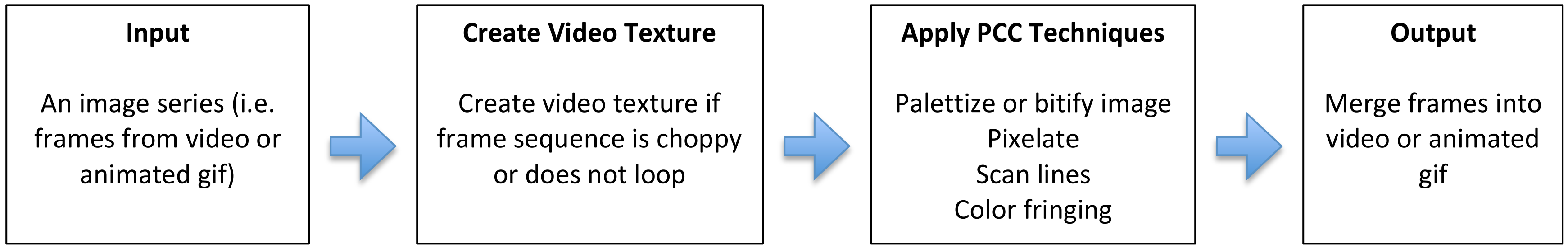
http://i.imgur.com/2eIpCPR.gif

Original Photo
http://rs794.pbsrc.com/albums/yy228/jade95_2010/Scenery%20and%20Landscape/Water-Seas-Waterfalls/anim_waves-j95.gif~c200

While following these guidelines allows PCC to best recreate true color cycling, the techniques available through PCC also allow users to create novel animations and still images which do not attempt to recreate the color cycling effect.

Pipeline

The pipeline for creating a PCC image which best recreates the effect of color cycling is listed below.


Input
An image series (i.e. frames from video or animated gif)
Create Video Texture
Create video texture if frame sequence is choppy or does not loop
Apply PCC Techniques
Palettize or bitify image
Pixelate
Scan lines
Color fringing
Output
Merge frames into video or animated gif

1. **Input** – A series of images portraying motion (for example, frames from a video or animated gif). Input which follows the guidelines above will best recreate the effect of color cycling.

2. **Create a video texture** - This ensures a final product with smooth changes between the beginning and end of the looped frame sequence.

3. **Apply PCC techniques to each frame in the video texture** – The techniques available to PCC are:

    a. **Palettization** – PCC allows the user to convert an image to a specified color palette (for example, to the palette of a specific gaming system or specific game). The NES, Atari 2600 NTSC, Atari 2600 PAL, and Super Mario Bros. 2 palettes are provided. Users may also generate color palettes by providing an image comprised of the desired palette.

   b. **Bitification** – An alternative to palettization, PCC allows users to convert their images to color schemes of a specified number of bits.

   c. **Pixelation** – Users may create a blocky, pixelated look by specifying the desired pixel size.

   d. **Color Fringing** – Users may model the color tuning of older televisions by shifting the image's red, green, and blue channels by slightly different amounts.

   e. **Scan Lines** – Users may recreate the horizontal scan lines common on older televisions by specifying the distance between each scan line and the amount of color difference between the scan line and original image.

4. **Output** – Merge the processed frames into a looped gif or video sequence to create the appearance of color cycling.

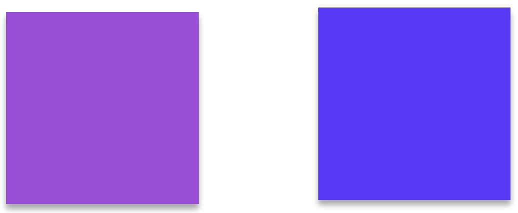
3. Examples and Implementation

Palettization

Palettization is the process of converting an image to a specified color palette. Color comparison is the core of palettization. In order to palettize an image, we need a way to determine which color in a palette is most comparable to the color of a given pixel. The palettization method creates a novel image by finding the closest palette color for every pixel in an image. I have written three different methods for scoring the likeness between two pixels.

RGB Difference

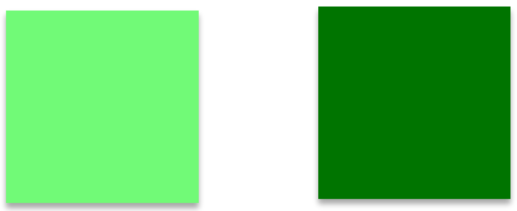
The RGB Difference method scores the likeness between two pixels based on differences between corresponding color channels. The likeness score between two pixels is the sum of the positive differences between the red components, green components, and blue components. For example, the likeness score between pixels with RGB values of (130, 50, 200) and (70, 20, 240) is 130.



Color Score = 130

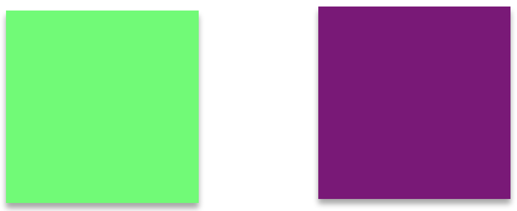
(130, 50, 200)   (70, 20, 240)

While this method is simple to compute, it does not prioritize the dominant color channel. In certain cases this can lead to a color selection which actually appears to be contrary to the original color. For example, the RGB value (100, 255, 100) is expressed as green. While the RGB value (0, 100, 0) is also expressed as green, the RGB value (100, 0, 100), which contains no green, achieves a more comparable color score:



Color Score = 355

(100, 255, 100) (0, 100, 0)



Color Score = 255

(100, 255, 100) (100, 0, 100)

Weighted RGB Difference

To account for the possibility of situations like the one above, I implemented a Weighted RGB Difference function. This function ensures that palette colors with the same order of RGB expression as a given pixel are scored better than pixels that express red, green, and blue components in a different order. Under this method, a pixel which primarily expresses green, like seen in the example above, is necessarily matched by a palette color which also primarily expresses green.

However, this presents a new problem: how do we address pixels that do not express one color component significantly more than the others? For example, the RGB value (100, 120, 110) presents as gray. Under the Weighted RGB Difference method for color comparison, this value is necessarily matched to a palette color whose highest RGB component is blue—even if a different palette color provides a closer representation to gray. To address this, I implemented a threshold parameter. The threshold is used to determine the degree to which a single RGB component must be expressed above its counterparts to be considered dominant. For example, for a threshold value of 50, the highest RGB component must be at least 50 units higher than the second-highest RGB component on a 255 scale in order to be considered dominant. Otherwise, the weighting requirement is overlooked and the generalized RGB Difference color score is returned.

HSV Difference

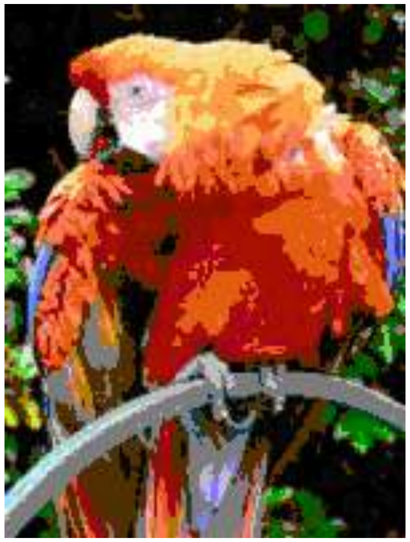
The HSV Difference method is analogous to the generalized RGB Difference method. However, the HSV Difference method uses hue, saturation, and lightness differences between pixels to determine the likeness score between two pixels.

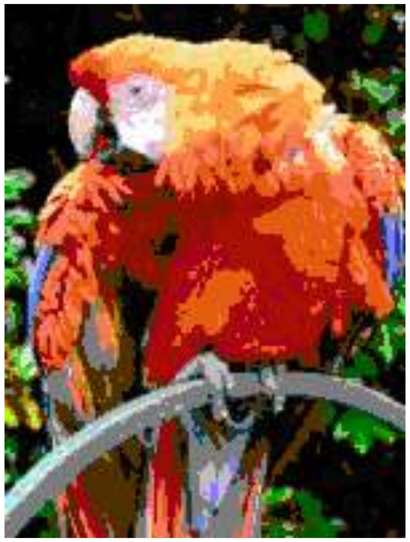
Results

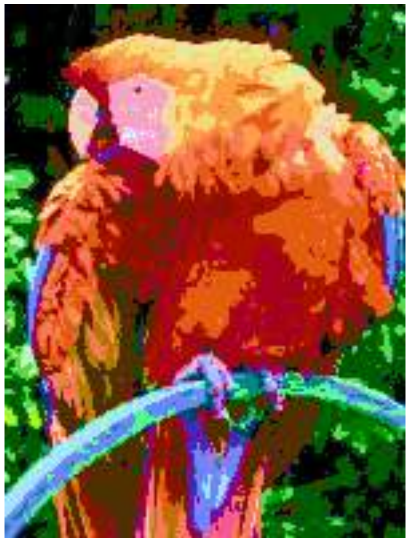
Below are the results for each color comparison method using the NES color palette.

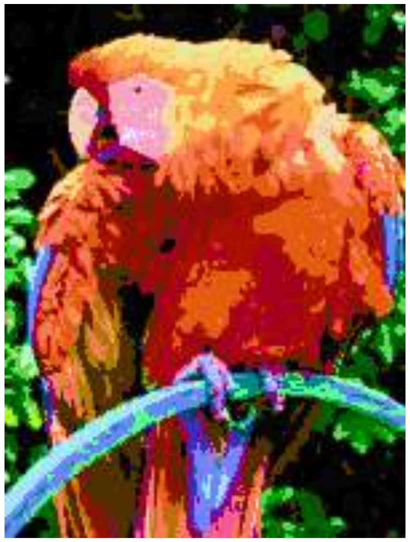
Original Image (`https://en.wikipedia.org/wiki/List_of_video_game_console_palettes`)

Color Determination Methods Using NES Palette


RGB Difference


HSV Difference


Weighted RGB Diff
Threshold: 0


Weighted RGB Diff
Threshold: 50

Analysis
The RGB Difference and HSV Difference methods clearly provide the closer representations of the original image than does the Weighted RGB Difference method. Given the possibility for the generalized RGB Difference method to provide inaccurate color expressions, I was initially surprised by its high performance. Upon reflection, I believe part of the reason the generalized RGB Difference method works so well is because the NES provides an adequate supply of colors. The example I provided for failure in the RGB Difference method is extremely contrived. Realistically, a well-chosen palette is unlikely to result in the sort of extreme failure I presented.

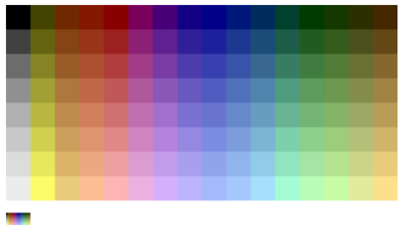
The Weighted RGB Difference method clearly provides a more intense coloration than either the RGB Difference and HSV Difference methods. This is especially visible in the coloration of the leaves, which express more vibrant shades of green than the other methods. We also see green leaves clearly expressed in areas of the Weighted RGB Difference results where the naked eye is not even able to distinguish a green color in the original image (see the bottom right corner). While this intense coloration provides a less accurate representation of the original image, I would hesitate to call is worse, as this could very easily be a desirable effect. However, the results also show that the Weighted RGB Difference method is prone to color bleeding and discoloration. This is visible above the parrot's head, near the parrot's eye, and on the parrot's feet and perch. Comparing the Weighted RGB Difference results to the generalized RGB Difference and HSV Difference results, we can see that these discolorations most often occur in areas of the image that are gray, black, and white. While increasing the threshold alleviates the color bleeding above the parrot's head, we see that the other discolorations persist.

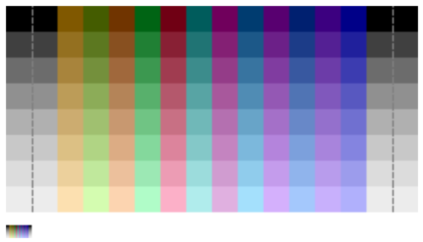
Palette Implementations

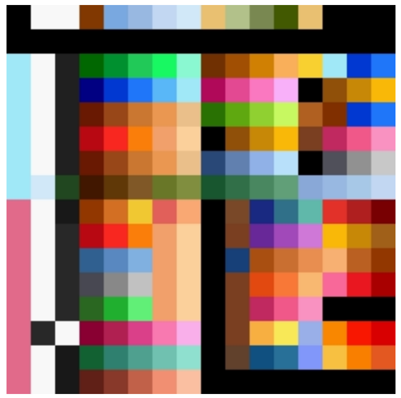
For this project, I implemented the NES, Atari 2600 NTSC, and Atari 2600 PAL palettes as provided on Wikipedia.org, as well as the Super Mario Bros. 2 palette as provided at www.supermariobrosx.org. I have also implemented a function that can convert a specified image into a color palette for palettizing an image. My results for the mentioned color palettes are below.

Game System Palettes

NES [1]


ATARI 2600 NTSC [1]

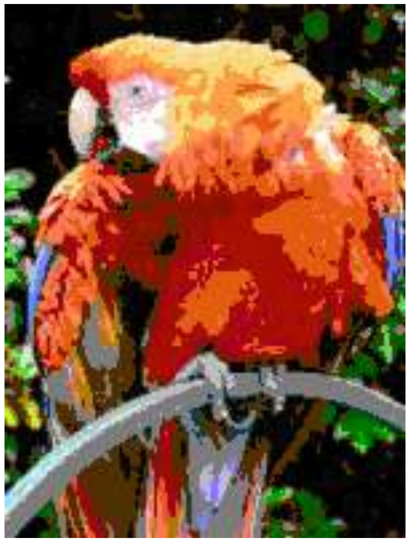

ATARI 2600 PAL [1]

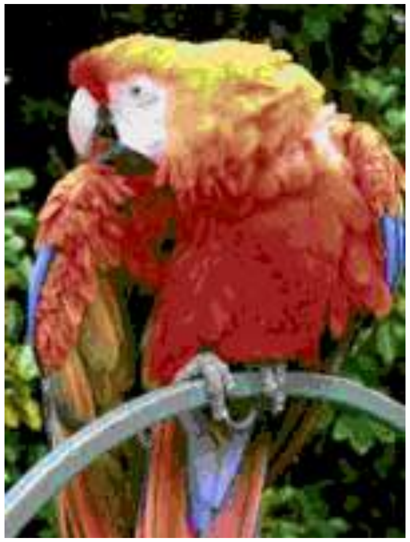

Super Mario Bros. 2 [2]

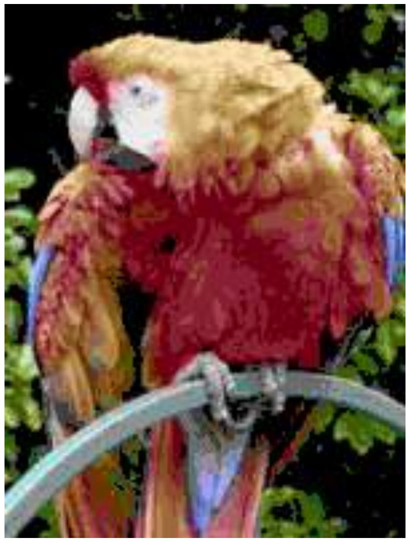
```
[1] https://en.wikipedia.org/wiki/List_of_video_game_console_palettes
[2] http://www.supermariobrosx.org/forums/viewtopic.php?t=2255&p=20896
```

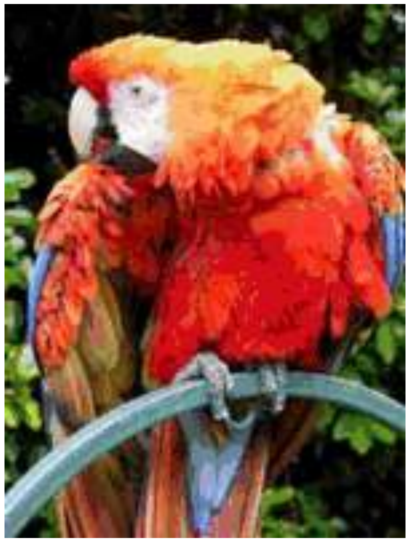
Original Image

Palettized Images


NES Palette


ATARI 2600 NTSC


ATARI 2600 PAL


Super Mario Bros. 2

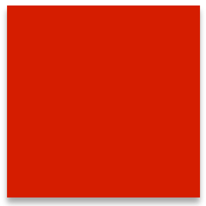
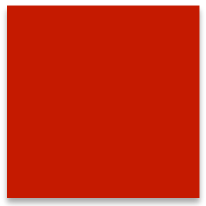
Bitification

As an alternative to palettization, I implemented a bitification method that converts an image to the specified bit scheme. While this method inherently preserves ratios between color channels, thus avoiding the color determination troubles of palettization, it comes at the cost of producing a less authentic representation of old school video game graphics.

The bitification method works by distributing bits across the red, green, and blue color channels. For example, an 8-bit image has a 3 bit red channel, 3 bit green channel, and 2 bit blue channel. This representation allows for 2^3 red values, 2^3 green values, and 2^2 blue values, enabling 2^8 or 256 possible colors. The chart below demonstrates the number of colors available per channel for different bit representations.

Bits per Image	Red Channel	Green Channel	Blue Channel	Colors per Image
3	2 colors	2 colors	2 colors	8
6	4 colors	4 colors	4 colors	64
8	8 colors	8 colors	4 colors	256

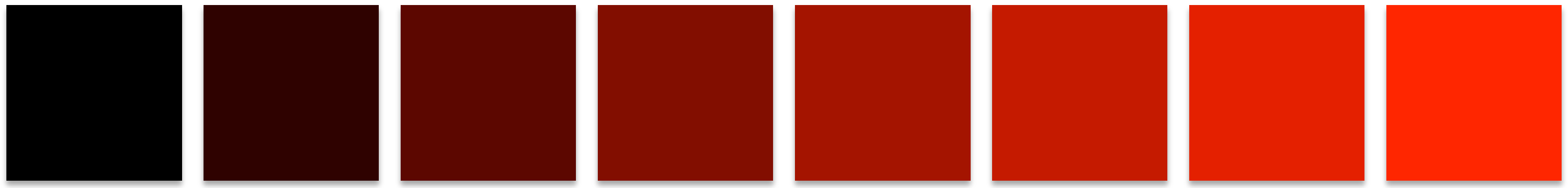
The bitification method overcomes the color comparison challenges of the palettization method by preserving the true ratios between color channels as closely as the specified bit representation allows. This is achieved by converting each channel of the original image to a scale from 0 – 1, determining the corresponding integer value on the range of possible colors for the bitified channel, and then normalizing each channel back to a 0 – 255 scale. For example, *Converting Red Value to 8-bit Color Space*

(200, 0, 0)	200/255 = 0.7843	0.7843 * (8 – 1) = 5.490	5/(8 – 1) * (256 – 1) = 182.1429	(182, 0, 0)
24-bit color space 8-bit R channel	Ratio for R channel	Value on 8-color scale 3-bit R channel	Value on 256-color scale Normalized 3-bit R channel	8-bit color space

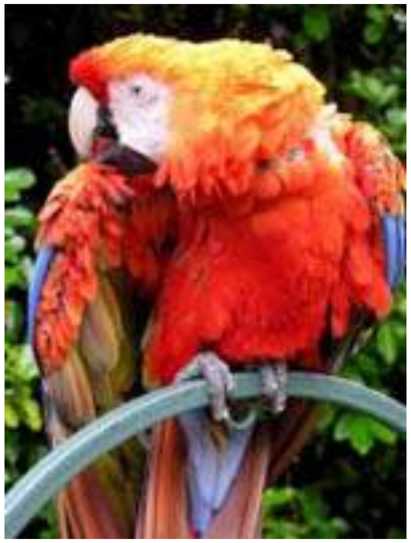
Below are the possible red values for an 8-bit image (8 possible values for red) using my bitification method.

Red Values in 8-bit Color Space

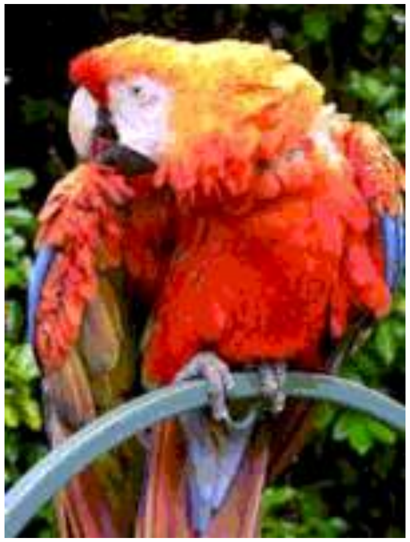


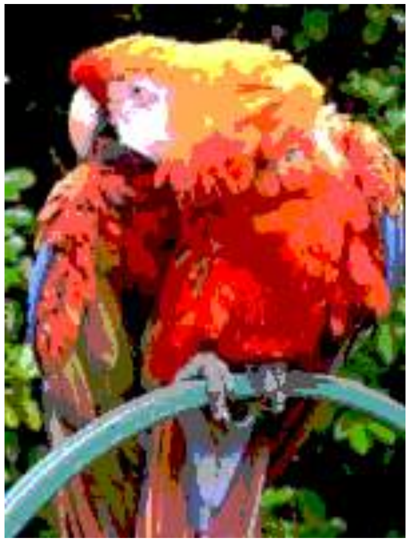
(0, 0, 0)	(36, 0, 0)	(73, 0, 0)	(109, 0, 0)	(146, 0, 0)	(182, 0, 0)	(218, 0, 0)	(255, 0, 0)

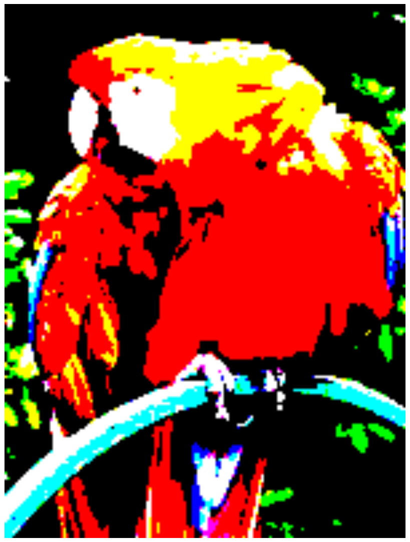
Below are examples of various bitification levels.


Original Image

Bitified Images


8-bit image (256 colors)

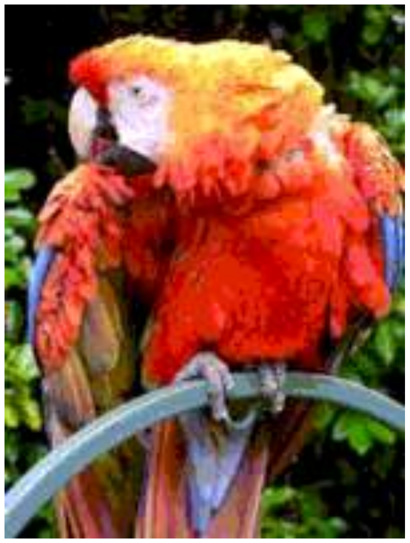

6-bit image (64 colors)

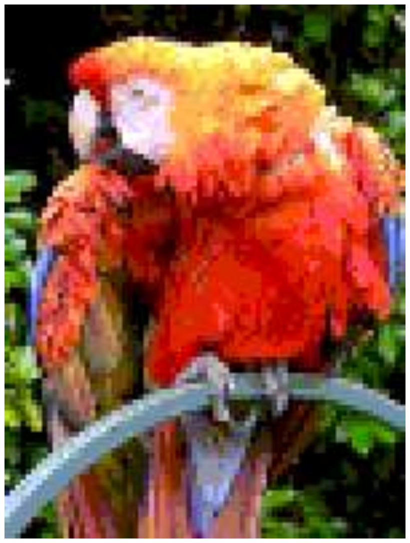

3-bit image (8 colors)

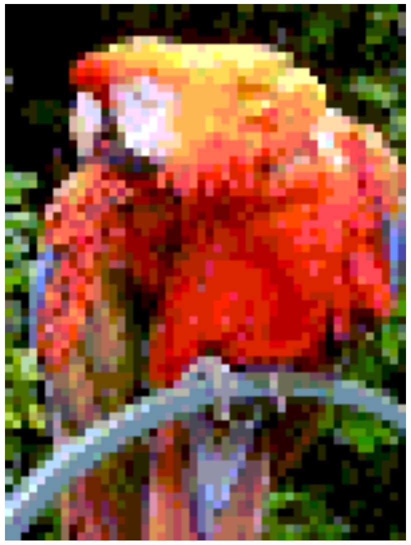
Pixelation

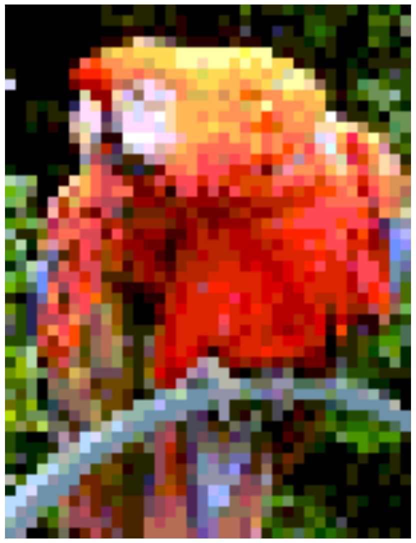
My pixelation method allows the user to increase the perceived pixel size of an image. This creates a blocky, retro feel. Larger pixel sizes should be used with smaller palettes or lower-bit color spaces in order to create a more authentic retro feel.

Pixelation


8-bit image (no pixelation)


2x2 pixel size

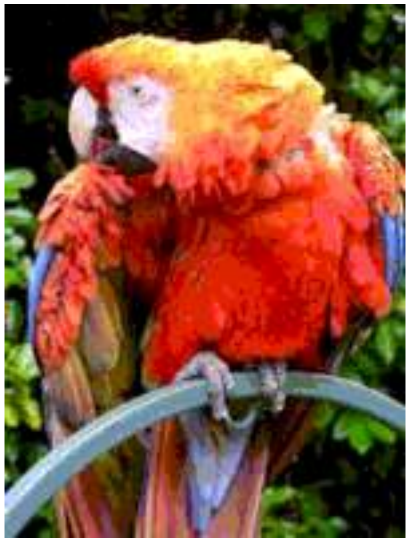

3x3 pixel size

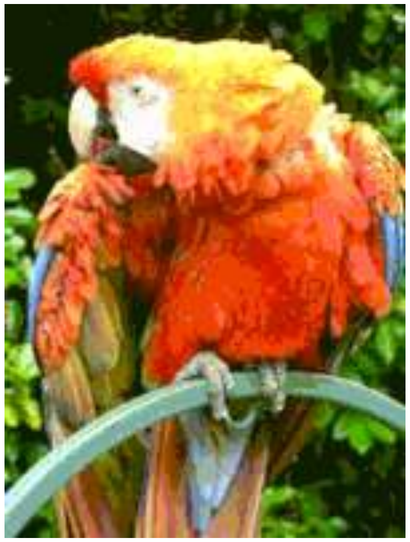

4x4 pixel size

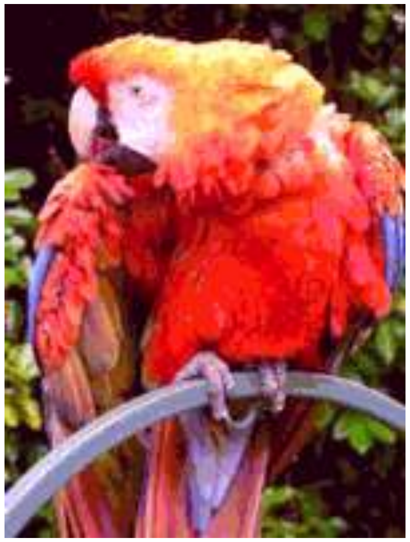
Color Fringing

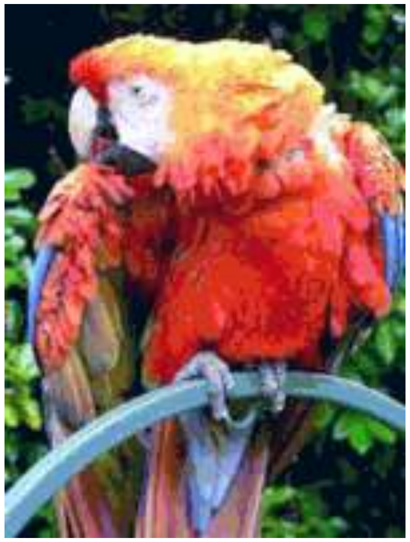
Cadin Batrack at TutsPlus.com suggests color fringing as an additional technique that can add a retro feel to images. Color fringing simulates the appearance of discoloration common to older television sets by slightly shifting an image's RGB values. My color fringing method takes the desired offset values for the red, green, and blue channels and then adjusts each image channel accordingly.

Color Fringing


8-bit image (no fringing)


Red +15
Green + 30
Blue -5

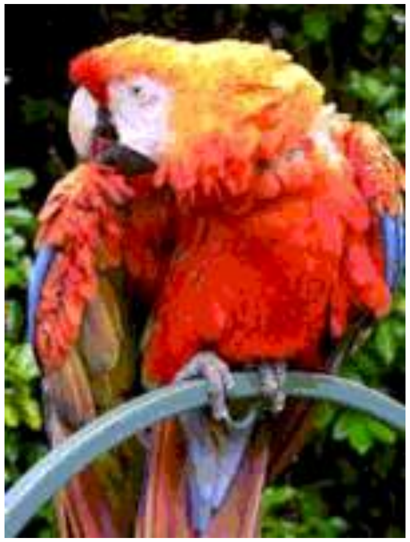

Red +30
Green -5
Blue +15

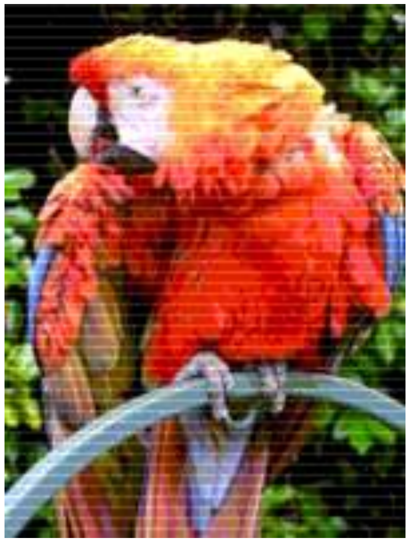

Red -5
Green +15
Blue +30

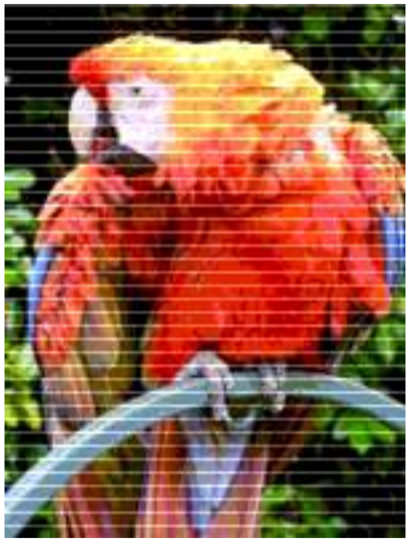
Scan Lines

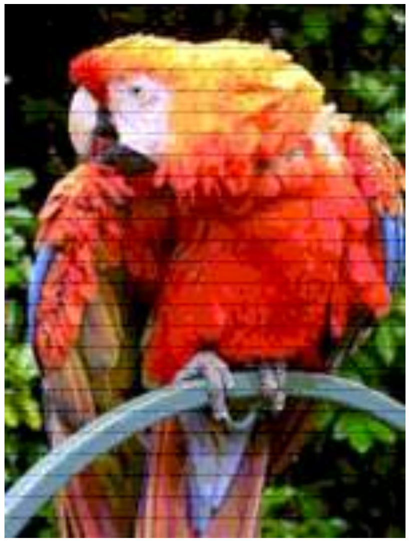
Cadin Batrack at TutsPlus.com suggests horizontal scan lines as an additional feature adding a retro feel to images. This mimics the scan lines common on older television sets. My scan line generator has parameters for setting the distance between scan lines as well as the amount of offset scan line coloration should have relative to the image.

Scan Lines


8-bit image (no scan lines)


16 pixels between lines
30 unit color offset

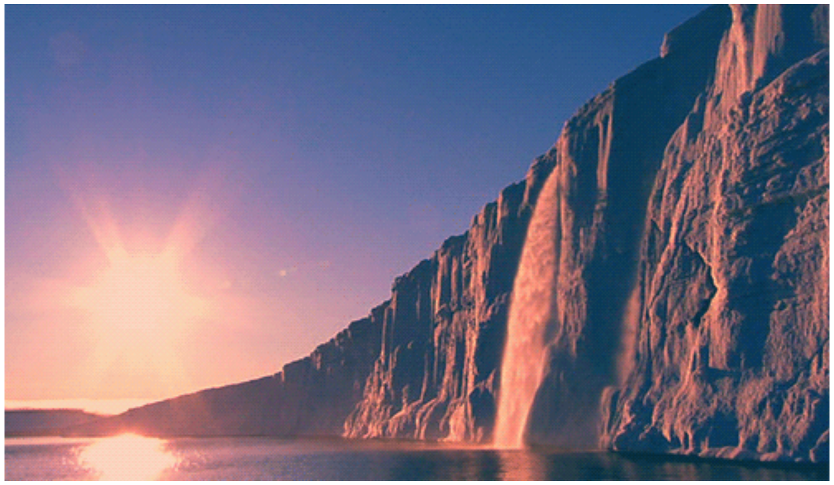

32 pixels between lines
70 unit color offset

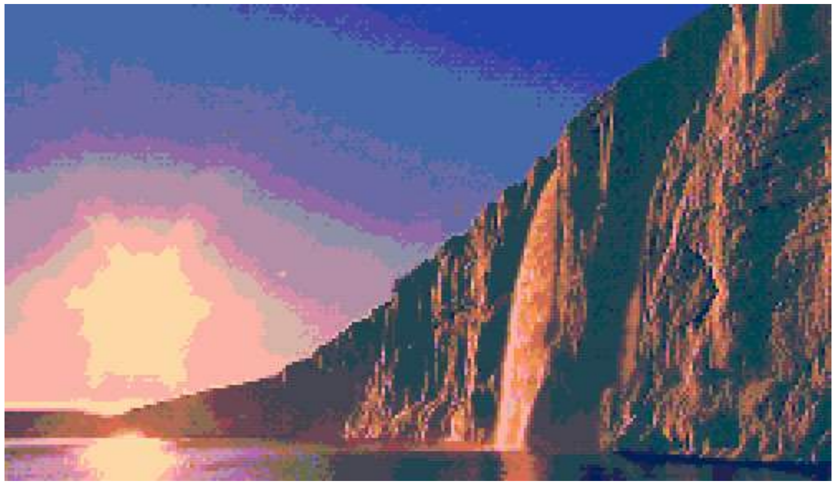

64 pixels between lines
-30 unit color offset

4. Results

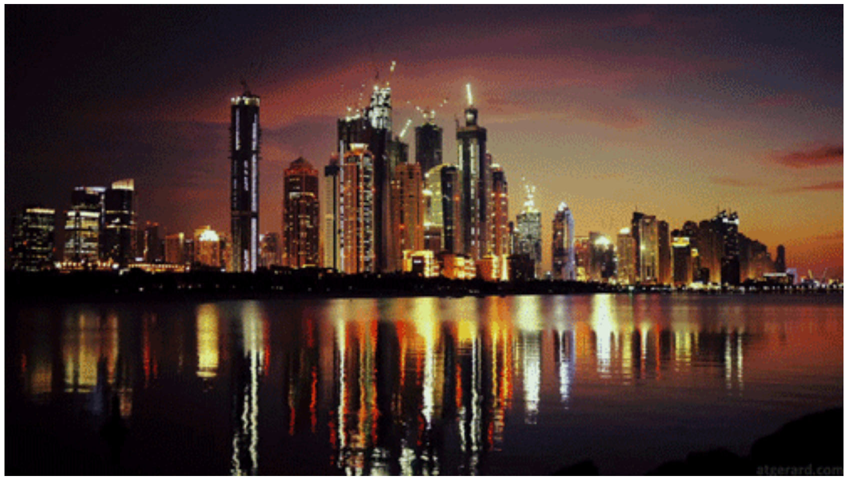
I have already shown the variety of techniques I have implemented for PCC. Therefore, the selected results do not showcase the diverse combinations of these techniques so much as the animations that I find most similar to true color cycling. Also, while this project conceptually makes use of the concept of video textures, I did not actually make use of my code from the video texture assignment. Instead I chose animated gifs that already had the appearance of a video texture so that I could focus on the photo processing techniques for the project.

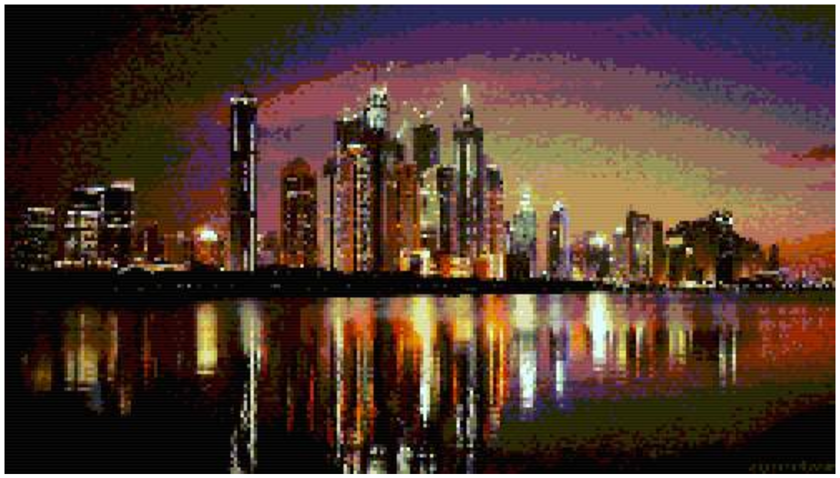
Waterfall


Original photo animation:
`http://media3.giphy.com/media/NPP4Z7wMIf0SA/giphy.gif`


PCC animation
`http://i.imgur.com/As5pdUy.gif`

Night City


Original photo animation [1]

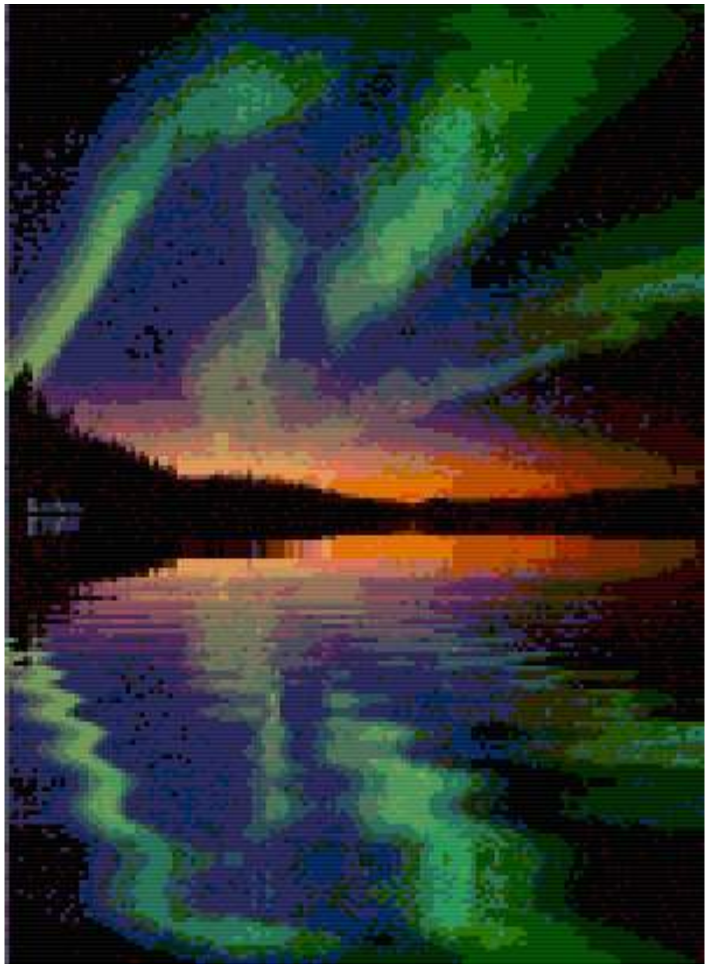

PCC animation [2]

[1] `https://s-media-cache-ak0.pinimg.com/originals/65/ee/e4/65eee49634c8bd54b16846dba90c2238.gif`

[2] `http://i.imgur.com/oG9eWQl.gif`

Northern Lights

Original photo animation [1]



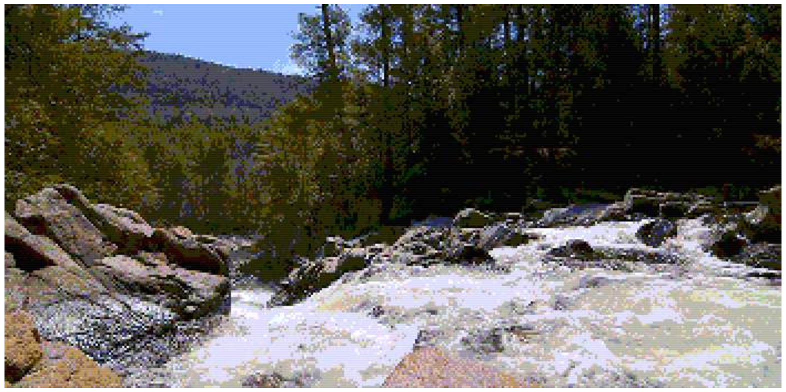
PCC animation [2]

[1] http://i658.photobucket.com/albums/uu308/Keefers_Tx/Keefers_LandscapesAnimated2.gif

[2] http://i.imgur.com/8oCetwm.gif

River

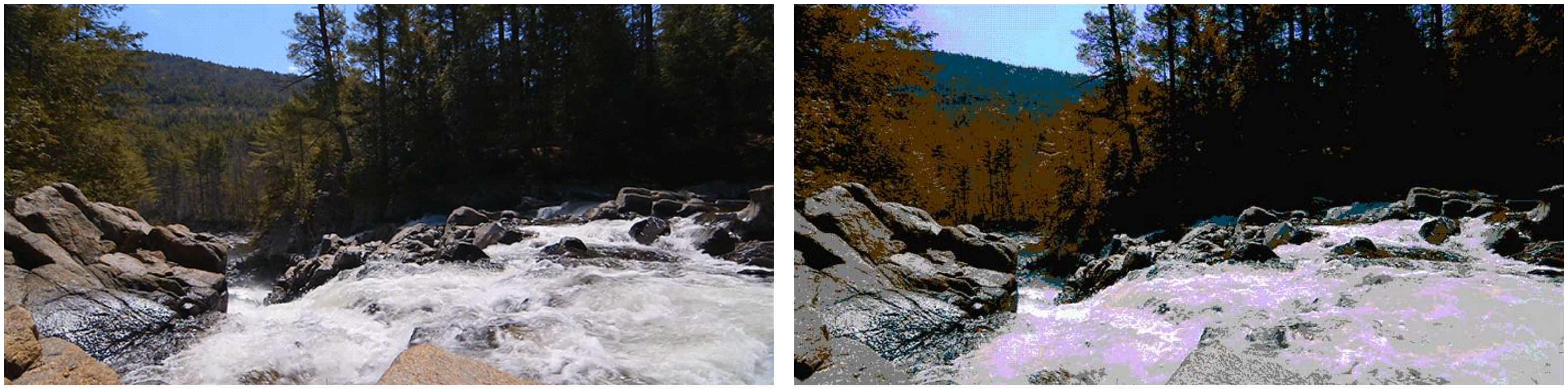
Original photo animation
`http://i.imgur.com/wsPGeJC.gif`


PCC animation
`http://i.imgur.com/XD8SmpZ.gif`

5. Reflection

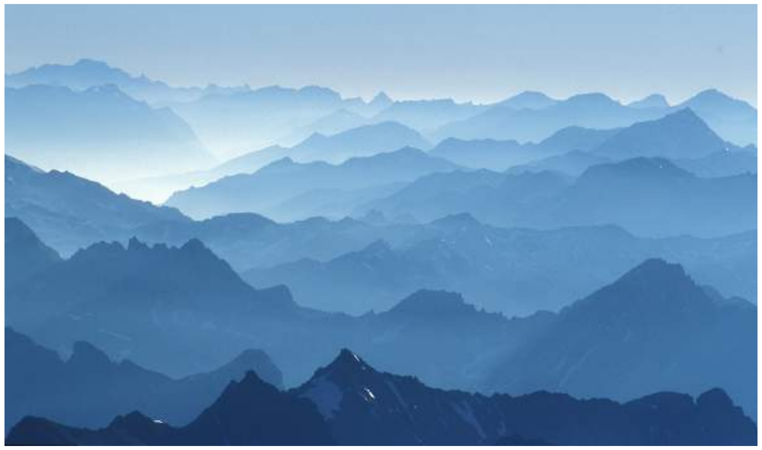
I am happy with how my project turned out! Some challenges I would better address given more time are:

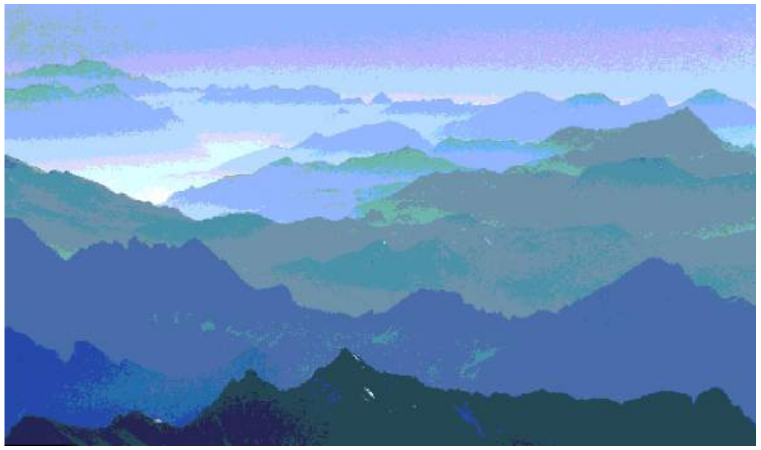
- **Find a better method for color matching**. While a few of my methods for color matching work well on the parrot image provided earlier in this document, I find for most other images my color matching methods perform poorly. For example, the first image in my River output converts to this NES palette using the RGB Difference method:

  While this method worked well for the parrot test image shown earlier, here were see that the trees match to black and brown instead of shades of green, and the sky and water both match to a large amount of purple. This is not ideal. While I was able to give up on color matching and move on to direct bitification, I wish that I had been able to spend more time finding a way to match colors. I think I could figure something out, but ultimately I had to cut my losses. One a positive note, the direct bitification method is immensely faster.

- **Distribute bits across channels.** An 8-bit color space allots 3 bits to the red color channel, 3 bits to the green color channels, and 2 bits to the blue channel. This raises a problem: what if my original picture is composed primarily of shades of blue? Here is an example of an image where the predominant channel allocations do not work well:


Original photo [1]


8 bit image

[1] `http://c0278592.cdn.cloudfiles.rackspacecloud.com/original/71182.jpg`

While the distinctions among the mountains are preserved, the very blue mountain scene presents as shades of purple and green when converted to an 8-bit form. We might be able to represent this image better with 2 bits allotted to the red channel, 2 bits allotted to the green channel, and 4 bits allotted to the blue channel. I would like to write a “smart” bitification method that would examine the color distributions across the RGB channels in an image and determine the best way to distribute bits across the three output channels. However, this function was less essential to the PCC suite, and I ran out of time to implement it. Therefore, a side effect of my implementation is that even though the chosen color space may allow for more accurate and pleasing color conversions, it is currently limited to the default 8-bit schema.

Sources

Batrack, Cadin. “Going Old School: Making Games with a Retro Aesthetic.” TutsPlus. 15 January 2013. http://gamedevelopment.tutsplus.com/articles/going-old-school-making-games-with-a-retro-aesthetic--gamedev-3567.

Huckaby, Joe. “Old School Color Cycling with HTML5.” Effect Games. 2011. http://www.effectgames.com/effect/article.psp.html/joe/Old_School_Color_Cycling_with_HTML5.

“List of Video Game Console Palettes.” Wikipedia. https://en.wikipedia.org/wiki/List_of_video_game_console_palettes.