INTERPOLATING PIXEL ART

by

Anson Rutherford
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Science

Committee:

_____ Dr. Yotam Gingold, Thesis Director

_____ Dr. Zoran Durić, Committee Member

_____ Dr. Jyh-Ming Lien, Committee Member

_____ Dr. Sanjeev Setia, Department Chair

_____ Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Fall Semester 2018
George Mason University
Fairfax, VA

Interpolating Pixel Art

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Anson Rutherford
Bachelor of Fine Arts
George Mason University, 2016

Director: Dr. Yotam Gingold, Professor
Department of Computer Science

Fall Semester 2018
George Mason University
Fairfax, VA

Table of Contents

List of Figures

Abstract

INTERPOLATING PIXEL ART

Anson Rutherford, M.S.

George Mason University, 2018

Thesis Director: Dr. Yotam Gingold

We present a method of generating plausible interpolations between frames from animated pixel art. We do this in the hopes of helping artists working in the style by providing a simple method of interpolation, which would reduce the amount of by-hand work required by the artists. We adapt an existing method for image interpolation, which uses energy cost minimization to define motion for each pixel position in the image. The specific challenges inherent in interpolating pixel art are discussed and examined. We describe the reasons the original paper's methods won't adequately work for our specific problem. Then, we describe the changes we make to the original paper's energy cost function to attempt to accommodate those challenges provided by pixel art. These changes focus on gathering more information from the source images, to make up for the limited amount of color information. We examine these changes and their results by interpolating the same images with different weights for the various factors we propose. We discuss which methods work best in general, and in specific cases.

Chapter 1: Introduction

1.1 Description of Pixel Art

We describe an interpolation technique designed to work with pixel art, a specific style of image that has unique properties that set it apart from others. We start with an existing method of image interpolation, and modify it to overcome pixel art's specifics challenges. When we talk about pixel art, we are referring to small raster images, usually not more than a few thousand pixels, rendered in a very limited palette. Pixel art is typically created not only by hand, but pixel by pixel. In addition, pixel art often depicts a single figure on a solid background.

Because pixel art requires so much concentrated effort to create, animated pixel art often animates at a rather infrequent rate. Artists will often animate only the bare minimum number of frames to get the point of an animation across. By developing a method that allows two frames to be interpolated between, we can lessen the amount of work required by an artist, and can allow artists to produce pixel art animations with a higher frame rate. Tweening, which uses a type of shape interpolation, is a tool utilized by artists who work in vector art. This tool can be used to generate more frames in between those handcrafted by the artist. We hope to develop our technique to fill the same role in the medium of pixel art, and bring the same benefits to those who work in the style.

Despite the small size and limited palette of pixel art, these features actually make it more difficult to interpolate. Less information in the image makes it harder to confidently assert what motion is happening between those two frames. A smaller image size means that errors and noise are more noticeable. Lastly, because of the infrequent frame rate, the frames we want to interpolate between are typically further apart temporally than you would see in other applications of this sort of algorithm.

Figure 1.1: Frames from an example of a hand-drawn pixel art animation. **Image courtesy of user ArMM1998 at Open Game Art (CC0).**

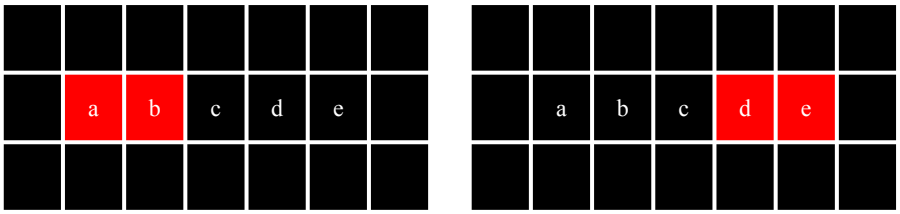

Figure 1.2: Two frames to be interpolated between. Positions of importance labeled for future reference.

1.2 Overview of Method

We base our method on the technique used in the work of Mahajan et al. [1]. We begin with two source images to interpolate between. We think of these two frames as being snapshots of a continuous motion. We consider the first image a sample at beginning of this motion, and the second image a sample at the very end. We use this algorithm to generate a description of the motion occurring between these two samples. The method uses energy cost minimization to assign a vector of motion to each position in the frame. This vector is not meant to represent the direction the pixel at this position is moving in, rather it represents the flow of pixels through this position between the two source images. Interpolation is done by sampling the pixels that lie along this vector of motion. Because of this, we want to assign this vector to be opposite the motion at this position. In Figure 1.2 we see an example of two frames we might want to interpolate between. Figure 1.3 depicts a well suited labelling of this example. Note that though the object depicted in these frames

moves to the right, the paths assigned, that we will interpolate along, move to the left.

During interpolation, we sample along the path defined by the assigned label at each position. We start by interpolating from the original position along the labelled path in the first frame. However, at some point, to better describe more complex scenarios, we want to be able to switch to sampling from the second frame. To do this, we pick some point, at which we swap to interpolating in the second frame. We want to follow the same motion, but at the end of this motion return to the original position. To facilitate this, when we swap to sampling from the second frame, we also move to a new position in the second frame, so that the remainder of the path of motion will return us to the original position. Essentially we split the total vector of motion into two colinear segments, which define how we sample between the two source frames. See Figure 1.4 for an example of this in practice.

These labels of motion are assigned by the energy cost minimization algorithm. We define a finite number of possible labels, covering every possible motion. Then we define two costs to be assessed by the minimization algorithm. We define a "unary" cost, which, for each position and for each label, gives the cost of assigning that label to that position. Secondly, we define a "coherency" cost, which for each pair of labels, gives the cost of those two labels neighboring each other. We then use a multi-label optimization algorithm to produce a solution to this energy minimization problem. As each position has been assigned a label, we can now produce any interpolation for any time between the two original images. For each position, we linearly interpolate along the defined vector to find the position between the beginning and end. We then copy the color at this position from the associated frame to the original position. Doing this for each position produces our interpolation.

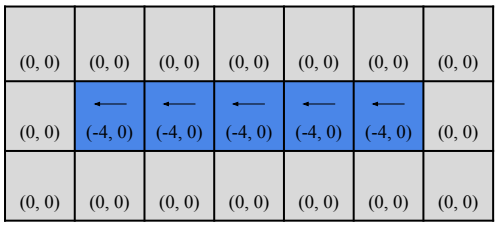

Figure 1.3: An ideal labelling of our example from Figure 1.2. Positions a, b, c, d, and e are each labelled in accordance to the motion happening in that region, and the remaining pixels are labelled corresponding to their lack of motion.

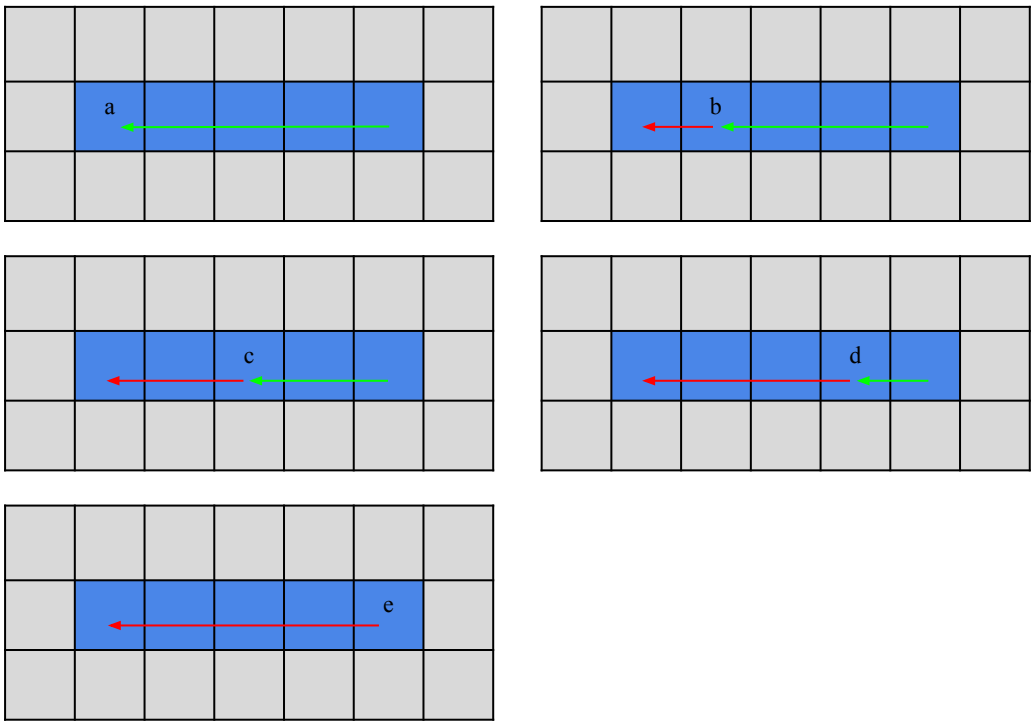

Figure 1.4: A depiction of a potential way of splitting the vectors for the positions labelled in the previous figures. The red vector refers to the motion in the first frame, while the green vector refers to the motion in the second. Note that while each pair of vectors sums to the same total, some pairs couldn't be placed in certain positions without leading outside the size of the frame.

Chapter 2: Previous Work

2.1 Image Interpolation

There exist numerous methods of interpolating between frames of animation. We base our methods most closely on the work of Mahajan et al. [1], which uses optical flow to interpolate images. Rather than interpolating directly, their method derives a description of the motion occurring between the two images by finding correspondence between the two images. While this method could be applied as-is to pixel art, it would suffer due to the specifics of the style. This is because the technique relies on the relative uniqueness of color information in the source images to derive the correspondence it requires to interpolate confidently. In pixel art, where we have fewer unique colors, it is harder to derive this. In addition, the small size of pixel art, and large gaps between frames, highlight any noise or errors in the solution, in a way typically not visible in larger images with a bigger palette. The work done by Browning et al. [2] is one example of image interpolation that attempts to maintain a specific style. They use an fluid simulation to generate new frames between hand-drawn key frames. These interpolated images follow the underlying simulation, and match the style of the hand-drawn frames of fluid simulation. Their work produces frames that are plausibly fluid based, in contrast to the results of other methods. Whited et al. [3] offer another example of stylized image interpolation. Their work focuses specifically on interpolating between stroke-based vector images. These images are generated and manipulated by an artist to save time during the inbetweening process, where the artist produces "in between" frames between two more disparate "key frames".

2.2 Pixel Art

Other work has been done on the subject of generating or modifying pixel art. The method developed by Inglis and Kaplan [4] generates pixel art from vector art. Their algorithm, Superpixelator, generates rasterized images that exhibit the stylistic properties of pixel art. This is done by detecting and incorporating aspects of the original vector art into the output, as well as correcting certain features typically avoided by pixel artists. In contrast, the method developed by Kopf and Lischinski [5] generates vector art from pixel art. Their method works by resolving ambiguities in the original image so that vectorization respects the intended features of the image. Similar to our efforts, the work done there takes an existing technique for vectorization and customizes it to accommodate the specific challenges of pixel art. Gerstner et al. [6] developed a technique to translate high resolution raster images into pixel art. Their algorithm works with an optimization method to establish and modify the palette and pixels of their output image iteratively, until their result converges. Again, more naive methods for the task of down-sampling exist, but were improved upon for the specific case of pixel art. We hope to follow suit by adapting an existing image interpolation technique to cooperate with pixel art.

Chapter 3: Algorithm

3.1 Description of Original Algorithm

We now describe the method used by Mahajan et al. [1], and describe the motivation and implementation of the major innovations we test. To interpolate between two images, we determine the motion occurring at every position in frame. We have a set of labels to describe all possible motions, and use an energy minimization algorithm to assign a label to each pixel position. For every position in the image, we associate a cost with assigning each possible label to that position. We also associate a cost to assigning two given labels in cardinally adjacent positions. This first cost, the unary cost, helps ensure the motions chosen respect information present in the source frames. The second, the coherency cost, ensures that the motions respect *each other*, rather than each choosing independently. The energy minimization algorithm ensures that labels are chosen for every position that minimize the total cost.

In order to assign motion to pixels, we must define a label for each possible motion. So that we can deal with a discrete number of labels, we deal only with motions described by whole number vectors. This motion defines where each position samples the original images. Each motion samples the first frame, and then the second. We first limit ourselves to motions that fit within the size of the original image. For example, in an image 10 pixels wide, there's no sense in allow for horizontal motion outside the range of $(-10, 10)$. For an image with width w and height h, we only consider vectors (x, y), where x is a whole number within $(-w, w)$, and y is similarly a whole number within $(-h, h)$. Secondly, we'd like to be able to split this total motion into two colinear vectors. Because we only want whole number vectors, we allow for pairs of vectors that aren't quite colinear. To facilitate this, we run our original vector through Bresenham's line algorithm, to produce a list of

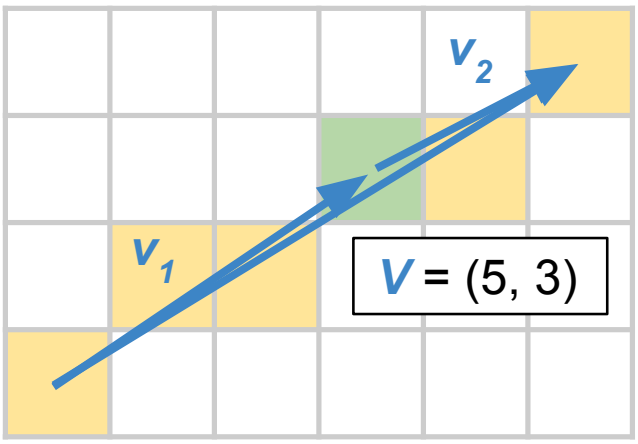

Figure 3.1: Here we see our full vector of motion V. Running V through Bresenham's line algorithm produces a set of positions $B(V)$, which are highlighted. We set our first frame vector v_1 to any of the pixels within $B(V)$, and set our second frame vector v_2 to $V - v_1$. Doing this ensures v_1 and v_2 sum to V and are roughly colinear. Here we highlight in green one potential splitting position. This sets v_1 to $(3, 2)$, and v_2 to $(2, 1)$.

whole number positions that approximate our vector. We then select any position from this list for our first vector, and set the remainder as our second. See Figure 3.1 for an example of this splitting.

Rather than declaring one label for each possible pairs of vectors, we only create one label for each full vector of motion. Our coherency cost is defined as the magnitude of difference between the two full vectors. As such, no information about the split is taken into account. However, the location of this split does change the unary cost. To simplify things, when calculating unary cost for a given full vector of motion, we calculate the cost for each possible split, and then store the minimum split along with its cost. When doing energy minimization, we use this minimum cost for each unary cost. If that label is selected, we then split that full vector of motion according to the minimum cost split we stored earlier. This saves time during energy minimization by lowering the total number of labels we must consider per position.

When we calculate coherency cost, we also implement a "seam cap", which lowers any

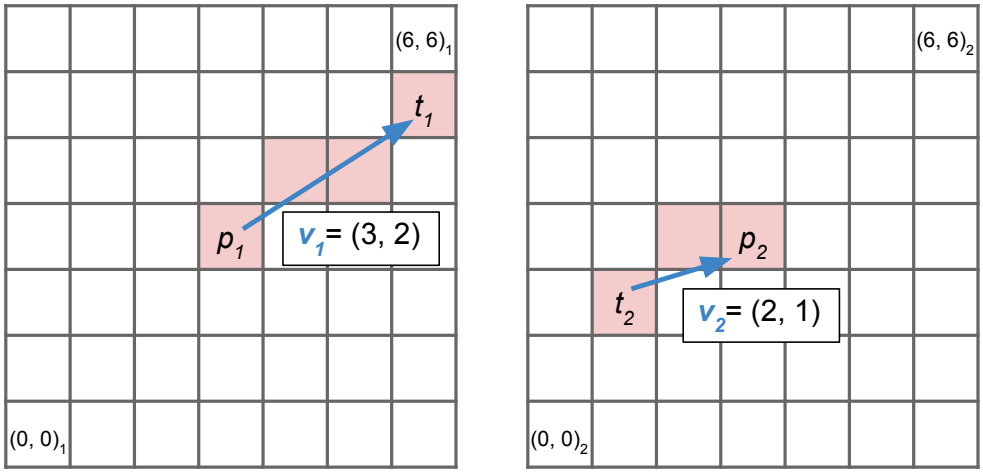

Figure 3.2: Here we illustrate a potential path for a given position p through the first and second frames we interpolate. p_1 and p_2 are at the same position $(3, 3)$ in their respective frames. The positions we sample along this path are highlighted in red.

coherency cost above it down to match it. We use a coherency cost to prevent noisy choices of motion, but we do expect there to be larger patches of motions that are dissimilar. We don't want to excessively penalize this expected behavior. So we cap the coherency cost, in an attempt to only penalize smaller inconsistencies. As such, our full coherency cost is

$$C(l_1, l_2) = min(|l_1 - l_2|, \delta) \tag{3.1}$$

where l_1 and l_2 are two labels, each representing a vector, and δ is our seam cap. This coherency cost is identical to the one employed in the work done by Mahajan et al. [1].

When we discuss unary cost, we talk about the cost of assigning a *pair* of vectors to a given position, even though pairs with the same sum will fall under the same label for energy cost minimization. This split drastically changes the unary cost defined in the original paper, and many of the innovations we describe below. The original paper compares the color at the end of the first frame vector to the color at the beginning of the second frame vector. This discontinuity, referred to as the "transition point" by the original paper,

is obviously entirely dependent on the choice of split. We can see clearly in Figure 3.2 why this is important. We use the motion described in Figure 3.1, with v_1 defining the motion in frame 1, and v_2 defining the motion frame 2. As we interpolate from the first frame to the second, we sample colors along the path formed by following v_1 from p_1 the point labeled t_1, equal to $p_1 + v_1 = (6, 5)$. At this transition point, we stop sampling from frame 1, and switch to sampling from frame 2, starting from t_2. t_2 is defined as $p - v_2$, which here is $(1, 2)$. We follow the path defined by v_2 back to p_2, the same position we started at, but now in the second frame. When we jump from sampling one image to the other, we want the transition to be as smooth as possible. Therefore, it is important that the colors at the points t_1 and t_2 match.

3.2 Issues with Original Methods

In the original paper, the unary cost is calculated from the magnitude of difference in color at the transition points. This method of determining energy cost is not as effective at interpolating pixel art, because it relies heavily on the comparatively high amount of information available in a full color image. In pixel art, a single color may make up half the image. As a result, simply finding two closely matching colors between the two frames is rarely enough to provide confidence in our choice of motion. In fact, often we can't rely on the similarity between two colors at all. Because of this, we instead choose to simply compare if the two match or don't, and add a penalty to the unary cost when they don't match.

This boolean comparison of color reduces the utility of this unary cost greatly, potentially too far to glean enough information from the source image. To remedy this, many of our innovations add terms to the unary cost which compare other pixels in the image as well. We also attempt to take advantage of the presence of a flat background, and attempt to make use of the knowledge that the background is non-moving to discourage certain options. The small size and high contrast colors of pixel art images highlight what would otherwise be minor errors or noise. In addition, frames of pixel art are usually fairly spaced out, and so

two adjacent frames of animation may look wildly different. Because of these factors, there may be many cases where a perfect solution either doesn't match any general function, or is only at best approximated. Below we see the possible additions to the unary cost we might choose to experiment with when interpolating between frames.

3.3 Modifications to Original Methods

3.3.1 Boolean Transition Point Comparison

As mentioned above, we switch to only confirming whether or not the two transition points match, rather than paying attention to the magnitude of difference in color data. Specifically, we associate a cost of 1,000 between a given position, and a label that places the transition points and pixels with mismatching color, and a prohibitively high cost of 10,000 if one or both of those transition points are out of bounds.

$$u(p, v_1, v_2) = \begin{cases} 10000, & \text{if } p_1 + v_1 \text{ or } p_2 - v_2 \text{ is outside the bounds of the image} \\ 0, & \text{if } p_1 + v_1 \text{ matches } p_2 - v_2 \\ 1000, & \text{otherwise} \end{cases} \tag{3.2}$$

In our figure 3.2 example, you can see a better representation of why $p_1 + v_1$ and $p_2 - v_2$ give us our transition points, there labelled as t_1 and t_2. In that example, because t_1 and t_2 are both within bounds, we will return a cost of 1,000 only if the colors at those positions, $(6, 5)$ and $(1, 2)$, don't match.

3.3.2 Transition Point Proximity Comparison

One way we attempt to improve upon existing results is to investigate the area around the transition point. We calculate this cost by comparing the pixels neighboring the two transition points to each other. We only consider the pixels adjacent in each cardinal direction for this. The cost is determined by the number of pixels that match between the

two frames. As with the transition point, we penalize each mismatch, adding a cost of 250 for each of potentially four total mismatches. Any out of bounds pixels are ignored, in the cases where the transition point is on the border of one of the images. This cost helps get a better sense of how well two transition points match, hoping to make up for some of the information loss in color. For our example in Figure 3.2, we would compare the points around t_1 to those around t_2. We would add 250 to this cost for each pair of $(6,4)_1$ and $(1,1)_2$, $(5,5)_1$ and $(0,2)_2$, $(6,6)_1$ and $(1,3)_2$. We would not add any cost for the comparison of the neighbors to the right of the transition point, as the point to the right of t_1 is outside the bounds of the image.

3.3.3 Frame Path Comparison

This energy cost is calculated by comparing the values at all points along the full vector of motion in both frames. Essentially, we calculate the transition point cost for all possible transition points along the full vector of motion. This helps us to get a better idea if vector chosen works in general, or if it is only suitable for a few chosen transition points. Barring the interference of other motion in the frame, you'd expect all pairs of vectors to match, since they represent the same spatial area. We sum up the number of mismatches along the vector, and then divide by the number of points. Because of this, the final cost is proportional to the percentage of mismatches, not the total sum. This helps us to avoid penalizing longer motions unduly. We do not penalize pairs of pixels when one pixel is out of bounds. Once again, we scale our cost in such a way that a complete mismatch produces a cost of 1,000. For each point b on the Bresenham line algorithm representation of V, we add

$$\begin{cases} 0, & \text{if } p_1 + b \text{ and } p_2 - V + b \text{ match, or are out of bounds} \\ 1000/|B(V)|, & \text{otherwise} \end{cases} \tag{3.3}$$

In Figure 3.2, this would involve comparing $p_1 + b = (3,3) + b$ to $p_2 - V + b = (-2,0) + b$ for each value for b in $\{(0,0),(1,1),(2,1),(3,2),(4,2),(5,3)\}$, and adding $1,000/6$ for each

in bounds mismatch.

3.3.4 Path Consistency Measure

This cost is defined as the number of pixels that don't match the end points of the frame paths they exist within. For each pixel along the path in frame 1, we penalize pixels for not matching the first pixel, and again for not matching the last pixel. Then we do the same with the pixels in frame 2 with the first and last pixels along that path. This encourages simpler paths, ones that don't move through very many areas of different color. This is done in the hopes of preventing unnecessarily complex motions being chosen. A path that remains the same color is less likely to produce odd noise during interpolation. We again divide the energy by the length of the path, as to not penalize longer motion. Because we check against the first and last pixel, even if all but the last pixel of a frame's path are the same color, it'll still be penalized at half the full possible amount, as none of those middle pixels will match the final pixel, The worst case, which produces the "full" cost, is one in which the first and last pixels of a frame path are different colors from all other pixels on that path, and the first and last pixels are different colors as well. Each pixel mismatch is penalized at 500/the length of the full motion vector. It will be penalized twice if it matches neither the beginning or end of its frame's vector. Again, the full possible penalty here would be 1000, though the beginning and end pixels of each vector will obviously match themselves. Figure 3.2 highlights the pixels in question in red. In frame 1, we'd compare the color at each red position to the color at p_1, and to the color at t_1. In frame 2, we'd compare the colors at those highlighted positions to those at t_2 and p_2.

3.3.5 Motion Length Penalty

This cost simply adds the squared magnitude of the vector of motion. We use this to encourage shorter movements. There are cases where motion is ambiguous in an image. We typically want to encourage the lowest amount of movement. There are numerous cases where two frames may be equally explained by two or more possible motions, where one

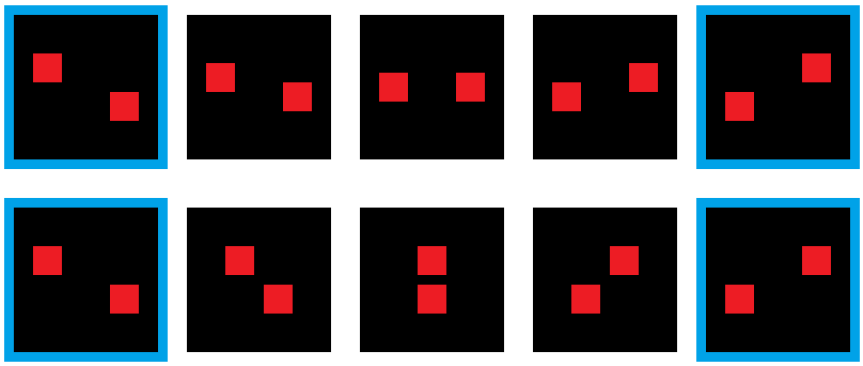

Figure 3.3: Two possible interpolations between two source frames (highlighted in blue). We can use the Motion Length Penalty to encourage the former be chosen over the latter.

motion is considerably longer than the other. See Figure 3.3 for an example where this penalty prevents a less expected movement from being chosen. We scale this cost by 1,000, and divide it by the squared magnitude of the size of the image.

$$u(p, V) = 1000|V|^2/|(w, h)|^2 \tag{3.4}$$

In our example, our vector of motion $(5, 3)$ has a squared magnitude of $5^2 + 3^2 = 34$, which we then multiply by 1,000 and divide by $|(7, 7)|^2 = 98$.

3.3.6 Background Distance Penalty

This variant of the above cost applies this length penalty only to motions which place their transition points in the background of the image. Because of what we know about the background of the image, namely that it is static, we can choose penalize labels that posit that the background does shift. In a way, placing a transition point in the background is "cheating". It disregards the initial intent of the idea, which is to find and transition between two pixels that represent the same spatial object in the frame. We know the background doesn't move, and so we can say that two transition points in the background some distance apart are assuming incorrectly by the magnitude of that distance. The cost associated is

derived from the distance between these two transition points, which is the same length as the full vector of motion. We simply apply the motion length penalty, though selectively so it only penalizes transition points in the background.

$$u(p,V) = \begin{cases} 1000|V|^2/|(w,h)|^2, & \text{if } p_1 + v_1 \text{ and } p_2 - v_2 \text{ are background positions} \\ 0, & \text{otherwise} \end{cases} \tag{3.5}$$

If, in our example, t_1 and t_2 were both background pixels, we'd apply the same cost we calculate for the Motion Length Penalty.

3.3.7 Background Exclusion

Due to the odd and counterproductive nature of the background, it may be best to avoid dealing with it whenever possible. This cost simply adds a prohibitively high cost of 10,000 to any pair of vectors that places both its transition points in the background of the image. This is a more drastic option that arguably might prohibit certain solutions from being found, but may help identify others in some cases. We make an exception for labels that assign a motion of $(0,0)$. Background pixels outside the moving parts of the image should have this label, and so we encourage it strongly here.

$$u(p,V) = \begin{cases} 10000, & \text{if } V \neq (0,0), \text{ and } p_1 + v_1 \text{ and } p_2 - v_2 \text{ are background positions} \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

If, in our example, t_1 and t_2 were both background pixels, we'd apply a penalty of 10,000, since our vector of motion isn't $(0,0)$.

Chapter 4: Evaluation

4.1 Method of Experimentation

To compare the inclusion and exclusion of various factors, we show interpolated frames alongside the original source frames. We ran our algorithm on a variety of animations, including two simple ones drawn for demonstration purposes, as well as other animations sourced externally. We used a prior developed library [7–9] for our multi-label optimization, using their provided expansion method with no iteration limit.

We tested each of our proposed factors at several different weights, with a weight of 1 referring to the cost described in the previous chapter, and other weights scaling those costs accordingly. At a weight of 1, most of these factors can produce a cost between 1 and 1,000, with the exception of Background Exclusion, which penalizes at 10,000 at a weight of 1, and the original transition point matching cost, which returns 10,000 for out-of-bounds transition points. For the coherency cost we set our seam cap to be 20, meaning no two labels could have a higher coherency cost than two with a magnitude of difference greater than 20. We also tried scaling this cost by various amounts, expecting higher scalars to bring it into comparable range to the penalties calculated from the unary cost.

4.2 Naive Results

If we run our algorithm naively, using just the Transition Point Mismatch penalty, the algorithm immediately finds a trivial solution. This solution is possible because there is a static, single color background that reaches to the borders of the image. When we only derive cost from the similarity of the transition points, and the coherency of motion, the algorithm can "cheat". By placing the transition points of all pixel positions at the far edges

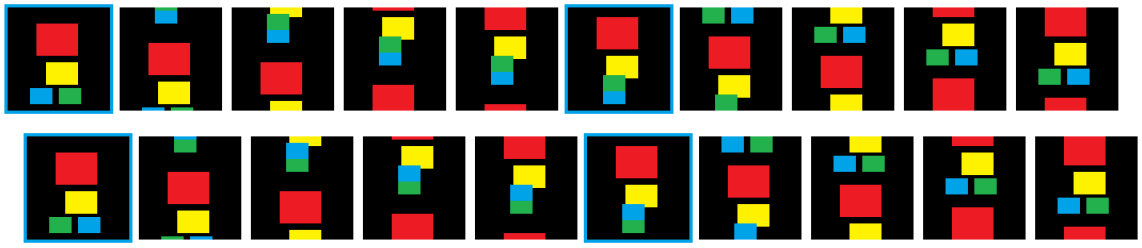

Figure 4.1: The results of an interpolation attempted with naive methods. We interpolate four new frames for each source frame. Four original source frames highlighted in blue.

of the image, each transition point can be placed in the background, while keeping the full vector of motion the same for each pixel. This produces a "scrolling" effect, where the entirety of the first image is moved out of frame to make way for the second. We illustrate this result in Figure 4.1.

To prevent this, we can simply incorporate the Motion Length Penalty, to penalize longer motions, and encourage the algorithm to work "within" the frame. Or we can attempt to force the algorithm not to abuse the background, as is the goal of the Background Distance Penalty and Background Exclusion factors. Additionally, the Frame Path Comparison and Path Consistency Measure prevent free choice of background pixels, by taking additional information from other areas of the image. When we apply any combination of these factors, we successfully prevent this trivial solution.

4.3 Specific Examples

Figure 4.2 depicts various interpolation methods of our first dummy animation. Each row depicts a different weighting. Each weighting multiplied coherency cost by 10, and weighted Motion Length Penalty, Background Distance Penalty, and Background Exclusion at weight 1. In the first row alone, we omit the Transition Point Proximity Comparison, for comparison. In the third row we incorporate the Frame Path Comparison at weight 1, and in the fourth row we instead add the Path Consistency Measure. In the final row, we utilize

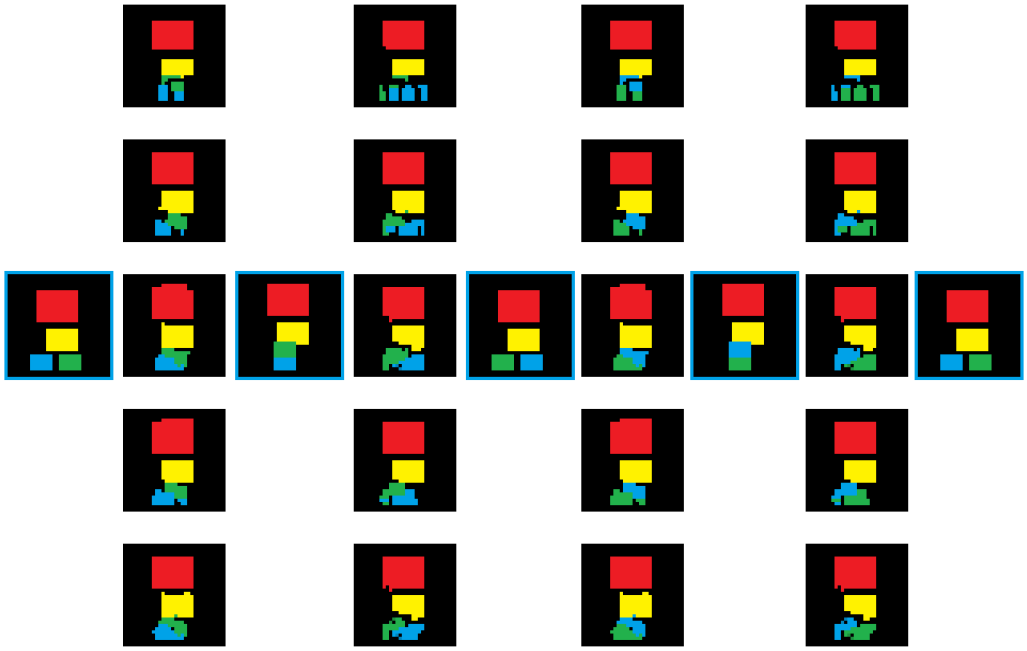

Figure 4.2: Interpolations between the four frames that make up one of our looping dummy animations. Source frames highlighted in blue. The fifth source frame represents the return to the first frame.

both of these together. We see that in all cases where we use the Transition Point Proximity Comparison, the cycling blue and green "feet" of the animation retain more of their cohesiveness, rather than being split into smaller chunks as is the case in the first row.

Figure 4.3 compares the differences between weightings of Motion Length Penalty, Background Distance Penalty, and Background Exclusion. Coherency cost was not multiplied. The first row represents our naive case, where we only use the base factor at weight 1. Rows two and three show Motion Length Penalty used at weight 0.1 and weight 1, respectively. Rows four and five show Background Distance Penalty at weight 0.1 and weight 1, respectively. Rows six and seven show Background Exclusion at 0.1 and 1, respectively, though there are only minimal differences in out put in these two cases. Each of these working independently isn't nearly enough to suit our needs. It's interesting to note that at a weight of 0.1, Motion Length Penalty and Background Distance Penalty look very similar, though at weight 1 they diverge further. Background Exclusion is the most noticeably incorrect on

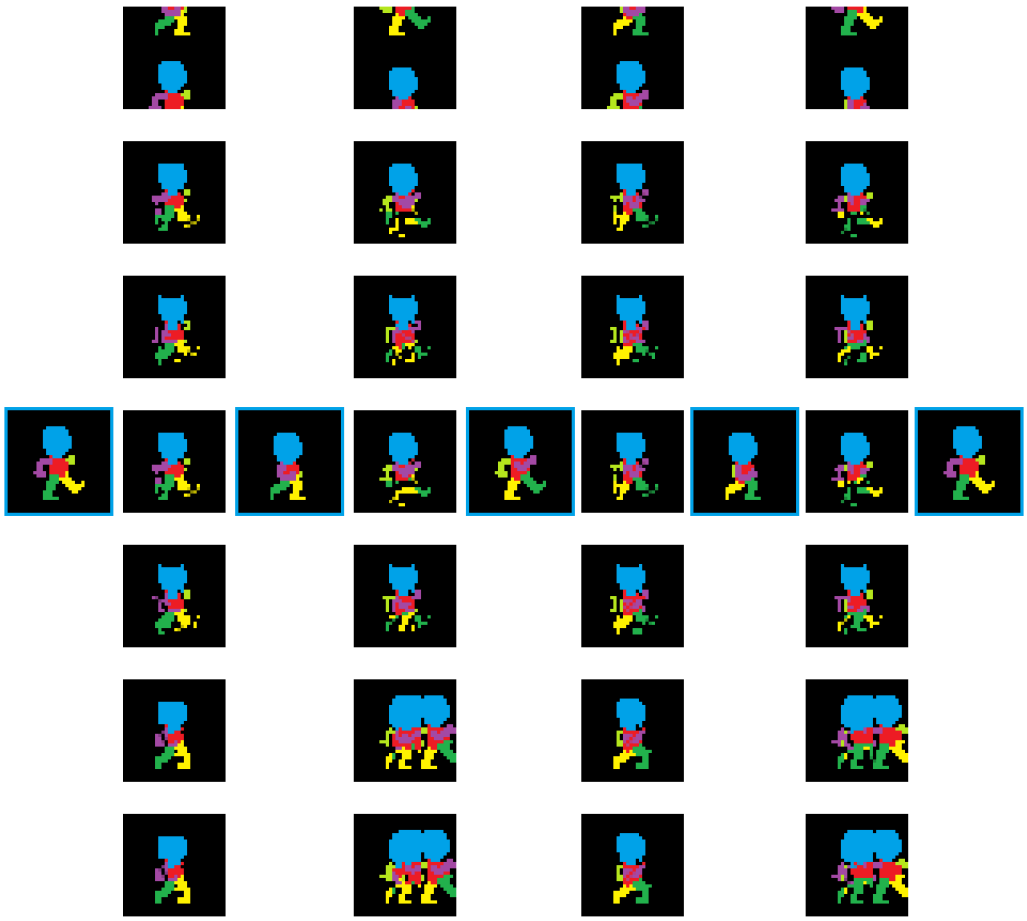

Figure 4.3: Interpolations between the four frames that make up one of our looping dummy animations. Source frames highlighted in blue. The fifth source frame represents the return to the first frame.

its own.

In the remaining examples, we show exemplary outputs of our algorithm. In Figure 4.4, we see a case where things worked better without Frame Path Comparison and Path Consistency Measure. Because the motion here is not linear, we wouldn't expect this to be handled appropriately by our algorithm, but in this case things work reasonably well. The extra noise in these frames probably contributes to the ineffectiveness of the Frame Path Comparison and Path Consistency Measure, and the rapid motion of parts around the frame probably prevents Background Exclusion from working well.

Figure 4.4: Interpolation of example animation. Coherency cost was not multiplied, all factors included at weight 1, except for Frame Path Comparison, Path Consistency Measure, and Background Exclusion. Several other sets of weights produced similar results. **Image courtesy of user ArMM1998 at Open Game Art (CC0).**

Figure 4.5: Interpolation of looping example animation. Coherency cost was multiplied by 100, all factors included at weight 1. **Image courtesy of user ArMM1998 at Open Game Art (CC0).**

Figure 4.5 shows an example of an animation handled exceptionally well by our algorithm. The motion here is extremely linear, which means a higher coherency cost helps things greatly, though we were still able to achieve good results with lower coherency cost. The abstract design gives us more leeway in our intermediate frames, and the relative simplicity of the shape leave little room for mistakes. Still, we're able to maintain that shape between frames, and shift colors around in a plausible manner.

Figure 4.6 depicts a more complex source. Note that the interpolation simultaneously keeps the shape of the head intact while approximately moving the sword along its intended path. Other weights produced results that rendered the head inconsistent during interpolation, or had the sword jump between frames, or disappear altogether. We expected the contrasting types of motions would complicate things too far, but we were able to produce plausible results regardless.

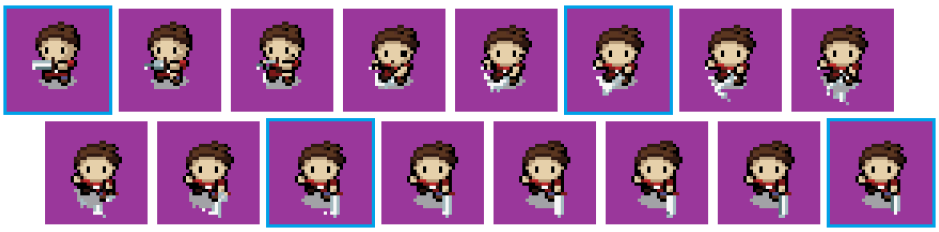

Figure 4.6: Interpolation of example animation. Coherency cost was multiplied by 10, all factors included at weight 1, excluding Frame Path Comparison and Motion Length Penalty. **Image courtesy of user ArMM1998 at Open Game Art (CC0).**

4.4 General Assessment

The Motion Length Penalty and Background Distance Penalty produced very similar results. This seems to follow from the fact that the latter is a conditional implementation of the former. In certain cases, the difference is apparent, though not always predictable. The Background Exclusion factor seemed prone to causing issues. It seemed to cause problems most often when the position of foreground pixels changed the most. It caused less trouble in cases where the foreground pixels stayed in the same area between frames, even if they swapped positions wildly within those bounds. These three factors seemed most suited to avoid the trivial case, and in some cases, some combination of just these factors was enough to produce suitable results.

The Transition Point Proximity Comparison factor seems to generally improve interpolation in most cases. We did not find any cases where its inclusion, at weighted cost 1, was noticeably worse than omitting it. The Frame Path Comparison and Path Consistency Measure both can be beneficial, but not always. Their effectiveness is highly dependent on the noisiness of the source frames. In simpler cases, like our two dummy examples, they are mostly beneficial. However, in more textured cases, they can force odd choices of motion.

Higher weighting of Coherency cost obviously helped in cases with simpler motion. It became a hindrance in cases where motions were more varied, for example when a region expanded, contracted, or rotated, rather than shifting linearly. In general, it did not seem

to be the case that a higher coherency cost was required to find a suitable solution, it merely made the choice of factors less important. Most cases could be solved with an unweighted coherency cost, so long as the correct factors were chosen.

Chapter 5: Conclusions and Future Work

Our algorithm takes in two frames of pixel art and generates any number of interpolations between these two images. We also specify a number of factors to be used for this interpolation. These factors are chosen in an attempt to mimic the style of hand-drawn pixel art, and to generate plausible motion between the two source images. This algorithm successfully avoids a trivial solution found by a simpler form of the same algorithm. The algorithm allows for an existing animation to be increased in frame rate.

While we've shown the use of the different factors we introduced, determining if an optimal set of weights exists, and if it is generally applicable, is of particular interest. Additional by-hand investigation could be sufficient to make stronger claims. However, as the set of weights and other properties is easily represented as a vector of numbers, it could possibly be computed approximated by a neural network or similar optimization technique. One could use a hand-drawn or more strongly derived intermediate frame as a ground truth, and seek to find a set of weights that best approximates the truth for a wide variety of different problems.

Though pixel art does have a very limited palette, there are cases where two approximate colors may still have some spatial relationship with one another. In shaded images, it's common to see a color appear in a few different levels of brightness. Incorporating this relationships between similar colors explicitly could be used to get more information from the images provided.

In cases where we want to interpolate between more than just two frames, there is additional information that could be put to use. Rather than treating the interpolation of the first and second frame, the second and third frame, the third and fourth frame, etc. as different problems, information could be shared between them. This may make it easier to discover correspondence, or to produce more coherent motion.

Because we end up defining the motion between two frames, its possible this could be used to render intermediate frames at real time. A shader could be developed that would use this motion, and the original source frames, to render the exact intermediate frame for any given time, rather than quantizing to one of however many provided frames and interpolations. This could be used to create "real time" animation from just a few images and motion sets.

Lastly, because we generate the motion, rather than interpolations directly, it's possible to apply motion derived from two images to different source images. One could apply the same motion to multiple different images, so long as the motion was the same. This could be used to save effort when interpolating multiple recolors or re-texturings of similar animations. Additionally, more complex images could be reduced to something better handled by the algorithm, and then those results could be applied back to the original images.

Bibliography

[1] D. Mahajan, F.-C. Huang, W. Matusik, R. Ramamoorthi, and P. Belhumeur, "Moving gradients: a path-based method for plausible image interpolation," in *ACM Transactions on Graphics (TOG)*, vol. 28, no. 3. ACM, 2009, p. 42.

[2] M. Browning, C. Barnes, S. Ritter, and A. Finkelstein, "Stylized keyframe animation of fluid simulations," in *Proceedings of the Workshop on Non-Photorealistic Animation and Rendering*. ACM, 2014, pp. 63–70.

[3] B. Whited, G. Noris, M. Simmons, R. W. Sumner, M. Gross, and J. Rossignac, "Betweenit: An interactive tool for tight inbetweening," in *Computer Graphics Forum*, vol. 29, no. 2. Wiley Online Library, 2010, pp. 605–614.

[4] T. C. Inglis and C. S. Kaplan, "Pixelating vector line art," in *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*. Eurographics Association, 2012, pp. 21–28.

[5] J. Kopf and D. Lischinski, "Depixelizing pixel art," in *ACM Transactions on graphics (TOG)*, vol. 30, no. 4. ACM, 2011, p. 99.

[6] T. Gerstner, D. DeCarlo, M. Alexa, A. Finkelstein, Y. Gingold, and A. Nealen, "Pixelated image abstraction with integrated user constraints," *Computers & Graphics*, vol. 37, no. 5, pp. 333–347, 2013.

[7] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 23, no. 11, pp. 1222–1239, 2001.

[8] V. Kolmogorov and R. Zabih, "What energy functions can be minimizedvia graph cuts?" *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 2, pp. 147–159, 2004.

[9] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, no. 9, pp. 1124–1137, 2004.

Biography

Anson Rutherford began his collegiate education by dual enrolling at Northern Virginia Community College while finishing up his final year of homeschooling. After graduating high school in 2012, he transferred to George Mason University. He graduated *cum laude* with a Bachelor of Fine Arts in Computer Science from George Mason University in 2016. During that time, he also completed the requirements to be accepted into George Mason University's Computer Science Masters program.