Spring 2012

Semi-Automated Creation of Cinemagraphs for the Exhibition Still Moving

William T. Wissemann
Bard College, WilliamWissemann@gmail.com

Semi-Automated Creation of Cinemagraphs for the Exhibition *Still Moving*

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
and The Division of the Arts
of
Bard College

by
William Wissemann

Annandale-on-Hudson, New York
May, 2012

Contents

List of Figures

Dedication

To Prabarna, Mom, Dad and all my friends and family for their love and support.

Abstract

This project was inspired by cinemagraphs, a style of animated GIFs made out of a compilation of photographs, and brings my study of computer science and photography together. The aim of the computer science component of the project was to animate specific objects within a set of photographs, while leaving the rest of the frame static. I employed various image processing techniques from computer vision and graphics. This eased and automated some of the repetitive and time consuming portions of the process of generating cinemagraphs. The challenge was to create an algorithm capable of balancing the accuracy of detecting the objects with the speed of processing each frame. The project also explored the limitations and advantages of using Microsoft's Kinect in artistic pursuits. From a photographic standpoint, my aim was to make these cinemagraphs presentable for a photography exhibition, hence the cinemagraphs had to go beyond being eye-catching because of technical trickery. The cinemagraphs had to hold the attention of the onlooker in spite of the novelty of the animated motion within the frame. The exhibition was a platform to explore the boundaries of cinemagraphs as an artistic medium.

Acknowledgments

I would like to thank Keith O'Hara for encouraging me to pursue a joint major and brainstorming with me as to how to integrate computer science and photography. I would also like to thank Stephen Shore for his artistic direction and mentorship while we explored this new medium.

1
Introduction

This project involved taking photographs with a RGBD (red, green, blue, depth) digital camera and producing cinemagraphs, which are represented by animated GIFs. Using image-processing algorithms, I separated and tracked an animated object within the frame. These cinemagraphs utilized formal techniques acquired through my studies at Bard, including framing and visual portrayal of space and incorporated visual cues to maintain a photographic quality. The emphasis of the project was to connect my programming skills and my ability to visualize space and manifest it in photography.

1.1 Cinemagraphs

Cinemagraphs are still images which contain repeated and endless movements within the frame. Kevin Burg, a visual graphic artist, and Jamie Beck, a photographer, coined the term 'cinemagraphs' for the moving images they created. Their images contain a subtle motion within a static image.[1] A cinemagraph is created by selecting all of the images which will be used in the animation and choosing which one will be the background image. Each image is placed on top of a background layer and this current image is edited so that

only the selected portion of this layer is visible. This visible portion of the image becomes the animation. Once all the images have been edited they are compressed into an animated GIF file.

1.2 Personal Artist Statement

My interest in this project was sparked by cinemagraphs created by Jamie Beck, who attempted to create photographs with animated qualities. I was drawn to the unexpected motion within the frame along with their aesthetic qualities. Photography is about capturing a moment in time often conveying a representation of movement through various visual techniques. These techniques include slowing down the shutter speed to create a blur of the moving object and stimulating the viewer's imagination by having the object positioned in such a way as to appear to defy gravity. Both of these methods are incomplete attempts to capture motion. They are merely representations which our brains have been trained to understand as motion. What I have set out to do in this project is capture a single movement within a frame to better understand motion's role in photography.

A majority of cinemagraphs do not hold up as photographs because viewers tend to perceive them as short video clips. My way of avoiding this issue was to include photographic artifacts within each frame. I had these cinemagraphs repeat in an attempt to push beyond the animation and cause the viewer to focus deeper into the frame. This is similar to the way in which depth of field works in photography. When a camera uses a small depth of field, everything but the subject of the photograph is rendered out of focus. This blurry region is something that we cannot see normally because when we look at this region directly, the blur comes into focus. In my cinemagraphs, instead of a part of the

[1]Information obtained from: http://cinemagraphs.com/about/

photograph being out of focus, the blurry region is clear and still, while the subject is in motion.

My animated cinemagraphs explore the changing relationship between a still world and a moving subject. Our eyes are instantly drawn to motion, especially in the city. I chose New York City as the backdrop for this project because it is full of motion. I wanted to isolate those moments of clarity within the chaos that pervade city life. Each cinemagraph emphasizes a moment of stillness in an ever changing world through the association of the viewer with the movement. While most of the time we are too distracted to really perceive all that is happening around us, these cinemagraphs attempt to do just that. Since the world cannot be stopped, these frozen backgrounds stop time and allow the viewer to see interactions that may have been missed and capture what is going in a fleeting moment.

1.3 Image Processing

Computer vision has developed a multitude of strategies for detecting objects within an image. These strategies analyze pixel data to decide which part of the image is foreground (the object being selected) and the background (which is everything else). These processes are known as foreground and background segmentation. Creating a given frame of a cinemagraph is accomplished by isolating the foreground and background of a given frame and then replacing the background of the frame with the background of the cinemagraph. However, one common problem with this method occurs when only color within the image is taken into account (Figure 1.3.1). In certain images, most of the pixels share the same color palette, making it difficult to isolate the foreground from the background.

Recent publications have explored the concept of the using RGBD cameras to capture color and depth in real-time images[1]. Hence, foreground/background segmentation be-

[2]Retrieved from Leaves and Petals shop website: http://www.shopleavesandpetals.com/productimage.php?product_id=53.

Figure 1.3.1. An image that shows the pixels sharing a similar color palette, making the foreground background segmentation challenging.[2]

comes an easier problem to solve because of depth information provided RGBD cameras, making it possible to create GIF images which are both artistically and technically sound.

I used a RGBD camera to augment commonly used algorithmic implementations for object sensing by incorporating the depth component of each pixel. This new implementation created a more reliable system of object detection.

1.4 Related Work

Creating cinemagraphs using conventional means of image editing software, such as Adobe Photoshop, is highly time consuming. I investigated published research to speed up the process of generating cinemagraphs. Recent approaches regarding tracking of non-rigid objects from a moving camera have included an implementation of mean shift which has similarities to a Bayesian framework. This allows for speed and efficiency of the tracking[2]. Paris and Durand approached the segmentation of static images with a slower but more accurate method. They incorporated Morse theory and topology of the images to create

a hierarchical segmentation model[3]. An alternate method of image segmentation uses an adaptive learning technique. Torr *et al* performs interactive image segmentation by using an adaptive learning process called the “Gaussian Mixture Markov Random Field" model[4]. While performing real-time tracking of people, removing the subject and their shadow is essential to the presentation of the final image. This process involves creating a background image in which the person is removed. One approach generates this background using a two-level, discrete wavelet transform[5]. Recent papers have explored the use of incorporating depth values to calculate image segmentation more accurately[1]. This paper focuses on using a combination of user input, data analysis and depth addition to balance processing time with the accuracy of image segmentation.

2
Semi-Automated Creation of Cinemagraphs

This chapter describes the method developed for creating cinemagraphs. The first step collects RGB and depth sensor data, which is read in from the RGBD camera and saved for later use. For this project, the RGBD camera used was a Microsoft Xbox Kinect. In the next step, the saved data was processed by merging foreground and background images to create each frame of a given cinemagraph. The final step was to run each frame at the correct speed to view the cinemagraph.

2.1 Hardware

2.1.1 The Kinect: An RGBD Camera

Kinect is a RGBD camera designed by Microsoft for their gaming console, the Xbox 360. The primary purpose of the Kinect is motion-detection which allows users to interact with the gaming environment without the need for hand-held controllers. After the Kinect's release, open-source hackers wrote drivers for Linux, Mac and Windows, and wrappers for C++, Java and Python programming languages. Later, Microsoft released the official SDK (Software Development Kit).

Figure 2.1.1 shows the breakdown of all the parts of the Kinect. For this project, the components of the Kinect that were used were the infrared projector and sensor, which allows for depth data to be captured, and the RGB sensor, which provides a color image.

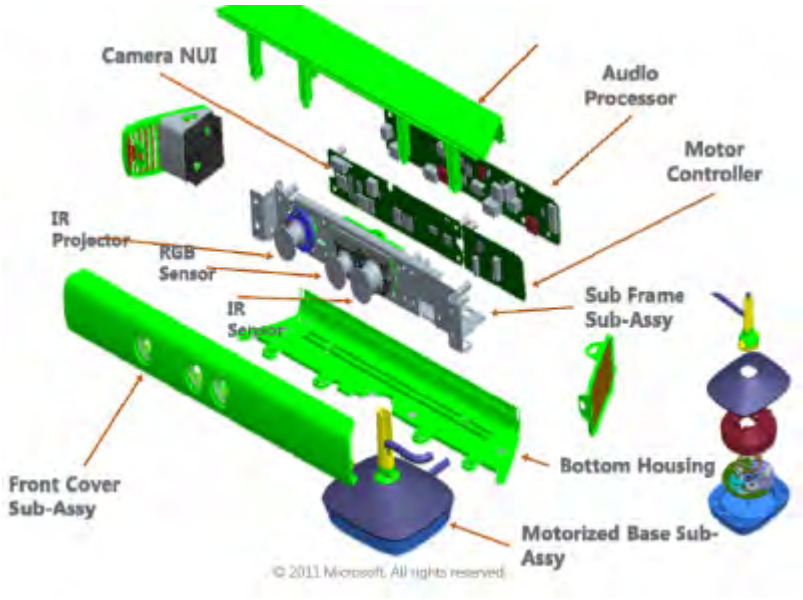

Figure 2.1.1. Breakdown of the Xbox Kinect.[3]

2.1.2 Depth Generation

The depth of an image is generated by a process called stereo triangulation. This process uses two or more images to calculate the 3D position of the points within the image. The first image is a hard coded image in the Kinect's memory. This hard coded image is then projected from the Kinect's IR projector. The second image is generated by the Infrared sensor, which takes a picture of the Infrared projected image. From a comparison of the hard coded image's virtual points to the points in the IR projected image, the 3D positions of the points in the image can be detected. The 3D points are the depth position from the camera. Figure 2.1.2 is an example of the infrared capture image from the Kinect and a visual representation of the depth values.

[3]Retrieved from the Open Kinect website: http://venturebeat.files.wordpress.com/2011/08/kinect-2.jpg

2.1.3 Experimental Setup

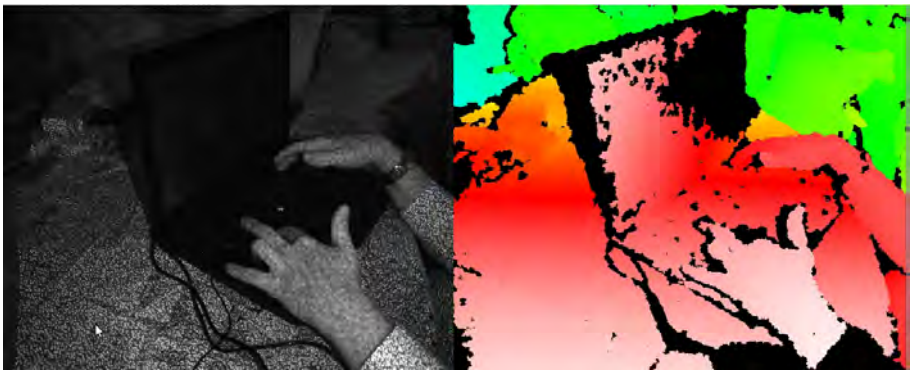

Figure 2.1.2. The IR image ($left$); the generated depth values as an image ($right$).[4]

The Kinect has two auxiliary connections. The first is an input for power. For this project I needed the Kinect to be mobile. Therefore the power cord had to be modified to allow it to be connected to a battery instead of an AC adapter. After a few tests, it was found that the Kinect required a minimum of 10.5 volts to operate which I satisfied with an off-the-shelf 12 volt lithium-ion battery. The second connection is to a USB device. This setup is depicted in Figure 2.1.3.

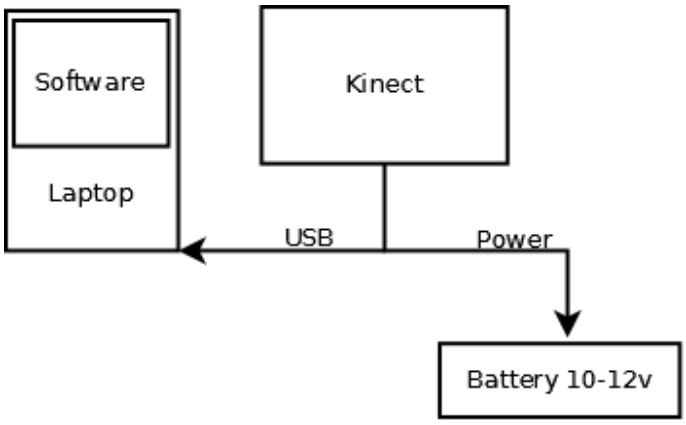

Figure 2.1.3. Schematic diagram of my Kinect setup.

[4]Retrieved from the Nicolas Burrus website: http://nicolas.burrus.name/index.php/Research/KinectRgbDemoV4?from=Research.KinectRgbDemo

2.1.4 Advantages

A Kinect was used for this project for the following reasons:

- The Kinect is a small device which makes it portable and allows for an easy way to take photographs for my senior exhibition.
- It is significantly less expensive compared to other RGBD camera devices.
- It is an accessible commodity for experimenting with the advantages and limitations for making cinemagraphs.
- It does not have significant amount of power demands which enables mobility.
- It collects high quality depth data at 30 frames per second.

2.1.5 Disadvantages

Although the Kinect is a useful way to create cinemagraphs, it has the following drawbacks:

- The Kinect was intended to be used indoors. The result of natural IR interference leads to unreliable depth results outdoors.
- The RGBD camera has extremely low resolution. It only takes images that are 640 by 480 pixels.
- The Kinect has little control over camera functionality, such as focus and f-stop.
- The lack of manual control for light correction results in issues when trying to perform image segmentation.

2.2 Software

Most photographic editing software is meant to edit individual pictures and does not have easy-to-use GUIs for mass editing. While using these tools there is a lot time spent opening, closing and saving each file. In addition, when a new file is opened, all the information about the last image is lost. This is a missed opportunity for the user since the data could be used to make guesses as to what should be selected in the newly opened file. The following three steps are the final processes I came up with to create cinemagraphs.

2.2.1 Data Collection and Synchronization

This step captures and saves the RGB and RGBD data. Since the two cameras are not located in the same place within the Kinect, the two images do not have a one-to-one cor-

respondence between frames. This means that before any analysis can be performed, the depth values must be mapped to their corresponding RGB pixel. During this step a modified *record* program is used, which is a part of the Fakenect library[5]. The unstable branch of the OpenKinect Library had support for retrieving already synchronized depth to RGB images from the Kinect in millimeters. *Record* was modified to dump the synchronized depth instead of the raw depth information.

The *record* program saves a dump of the Kinect sensor data at a rate of approximately 30 frames per second in individual files with names in the form "TYPE-CURRENTIME-TIMESTAMP." Here TYPE is either (d)epth, or (r)gb, TIMESTAMP is the time the image is seen from the Kinect, and CURRENTTIME corresponds to the time the image began to be written to the hard drive. For RGB and DEPTH the dump is just the data provided in PPM and PGM formats, respectively. The RGB data is being stored in a 24-bit format whereas the depth images are stored in a 11-bit format. These data are being stored in their respective formats to prevent data loss. The index file logs all of the PPM and PGM files.

Figure 2.2.1. (*left*) An example of a PPM/RGB image (*right*) An example of a PGM/Depth image

[5]Retrieved from the http://brandynwhite.com/fakenect-openkinect-driver-simulator-experime

2.2.2 Foreground Selection Algorithms - Flood-Fill

Various tools were created to edit the photos, all of which are adaptations of the flood-fill algorithm. The original flood-fill algorithm is recursive in that each neighbor expands its neighbors which in turn expands their neighbors. The neighbors, however, expand only if they meet a specific color criterion. In the end, all the pixels of the same color value are selected. One problem with this approach is that objects appear to be of one color in photographs but are actually multi-tonal. To fix this problem, instead of having the flood-fill be based on a single color, it is based on Euclidean distance. The Euclidean distance equation for an n-dimensional space, where p and q are pixels is:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2} \tag{2.2.1}$$

The first tool I created was based on the x and y location of the pixels and this flood-fill results in a circular selection of pixels:

$$d_1(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} \tag{2.2.2}$$

The next variation was based on color:

$$d_2(p,q) = \sqrt{(p_r - q_r)^2 + (p_g - q_g)^2 + (p_b - q_b)^2} \tag{2.2.3}$$

A third version used depth in addition to color to get a more accurate Euclidean distance value:

$$d_3(p,q) = \sqrt{(p_r - q_r)^2 + (p_g - q_g)^2 + (p_b - q_b)^2 + (p_d - q_d)^2} \tag{2.2.4}$$

I encountered the following problems during the implementation of flood-fill: the maximum heap size in Java's virtual machine was not large enough to handle the number of recursive calls needed for the number of pixels being expanded and each pixel expanded in all four directions. Also, two neighboring pixels caused an infinite expansion loop and therefore will never generate a result. I addressed the first issue by changing the implementing of flood-fill by using one list that contained pixels waiting to be expanded and

another list of pixels that were already expanded. This implementation worked by using a loop that only stopped when the list of waiting pixels had all been processed. This means that the list of waiting pixels continued to grow until there were no more pixels to be expanded. The approach mentioned above resolved the issue of the heap size being too small. The problem of the infinite loop was solved by checking to make sure that the pixel had not already been expanded before adding it to the 'waiting to be expanded' list. As each pixel was processed from the 'waiting to be expanded' list, its Euclidean distance was calculated from the original selected pixel. If the value was smaller than a threshold distance it was turned on. That was maintained by a look up table of on and off as boolean values.

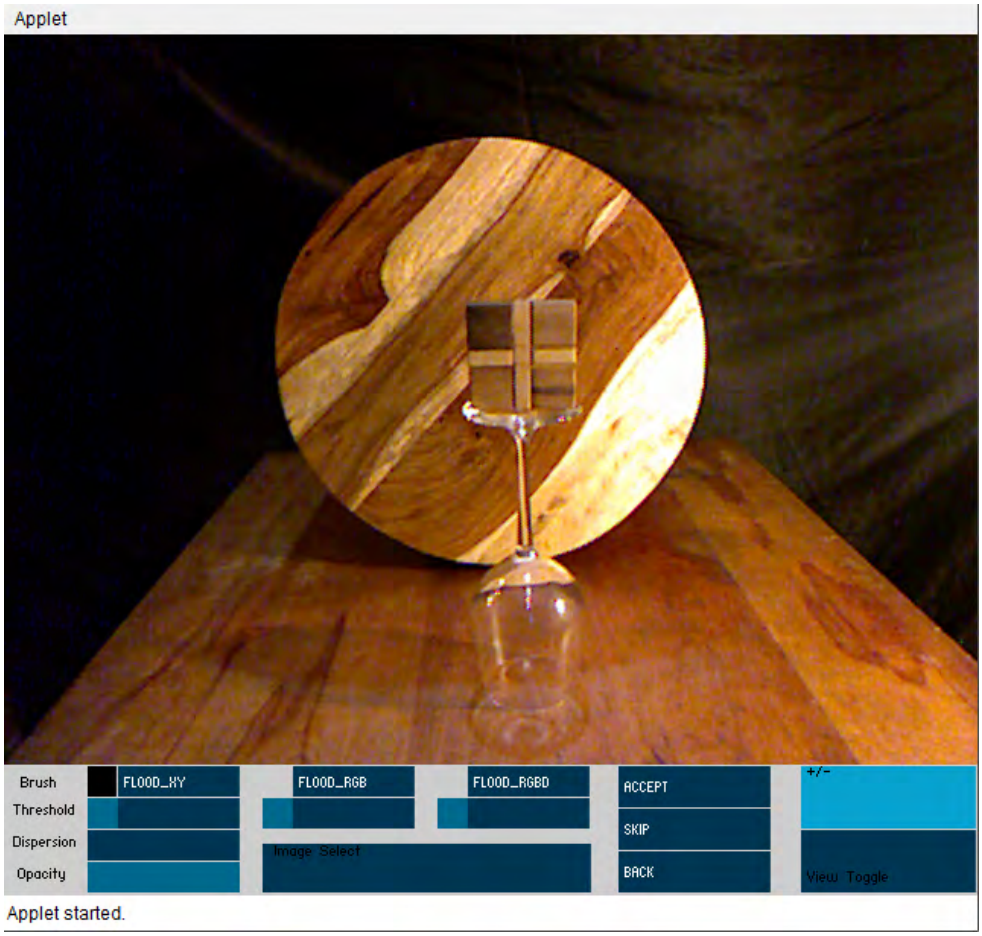

Figure 2.2.2. Screenshot of the GUI for creating cinemagraphs.

Each brush mode (Figure 2.2.2) has associated parameters that the user can control using the scrollbars below each brush. FLOOD_XY (Equation 2.2.2), FLOOD_RGB (Equation 2.2.3), and FLOOD_RGBD (Equation 2.2.4) have a threshold which control the maximum Euclidean distance value accepted. This, in turn, determines how far away, based on Euclidean distance, the pixels can be from the original pixels and still be selected.

The following is the code I implemented for the flood-fill tools FLOOD_RGB and FLOOD_RGBD. The method *isInCircle* calculates the Euclidean distance between the neighboring pixel and the user selected pixel. If the Euclidean distance value is less than the threshold value the method returns true. This code can be used for both tools since Equation 2.2.3 is the same as Equation 2.2.4 where the depth component is ignored in the Euclidean distance calculation.

```
boolean[][] selectPixels(int x, int y, int sr, int sg,
                              int sb, float sd, float threshold):
        ArrayList<int[]> waiting = new ArrayList<int[]>()
        boolean[][] active = new boolean[IMAGE_WIDE][IMAGE_HEIGHT]

        int[] startingPoint = new int[2]
        startingPoint[0] = x
        startingPoint[1] = y

        waiting.add(startingPoint)
        active[x][y] = true
        while (!waiting.isEmpty ()):
                int[] waiting_to_look = (int[])waiting.get(0)
                x = waiting_to_look[0]
                y = waiting_to_look[1]

                if ((x+1) < IMAGE_WIDE && (x-1) >= 0):
                        if (isInCircle(x+1, y, sr, sg, sb, sd, threshold)
                                        && !(active[x+1][y])):
                                int[] point_to_add = new int[2]
                                point_to_add[0] = x+1
                                point_to_add[1] = y
                                waiting.add(point_to_add)
                                active[x+1][y] = true
                        if (isInCircle(x-1, y, sr, sg, sb, sd, threshold)
                                        && !(active[x-1][y])):
```

```
                                int[] point_to_add = new int[2]
                                point_to_add[0] = x-1
                                point_to_add[1] = y
                                waiting.add(point_to_add)
                                active[x-1][y] = true
                        if ((y+1) < IMAGE_HEIGHT && (y-1) >= 0):
                                if (isInCircle(x, y+1, sr, sg, sb, sd, threshold)
                                                && !(active[x][y+1])):
                                        int[] point_to_add = new int[2]
                                        point_to_add[0] = x
                                        point_to_add[1] = y+1
                                        waiting.add(point_to_add)
                                        active[x][y+1] = true
                                if (isInCircle(x, y-1, sr, sg, sb, sd, threshold)
                                                && !(active[x][y-1])):
                                        int[] point_to_add = new int[2]
                                        point_to_add[0] = x
                                        point_to_add[1] = y-1
                                        waiting.add(point_to_add)
                                        active[x][y-1] = true
                                        waiting.remove(0)
        return active
```

Opacity and dispersion are only available to the FLOOD_XY brush. This was a design choice because opacity and dispersion are not required for foreground selection by the other tools. Both of these tools use the alpha channel to control how visible a given pixel will be in an ARGB image. Opacity determines how much of a selected pixel is seen in the final image. This is done by remapping the alpha value from the range 0 to 255 (where 0 is transparent and 255 is opaque) to 0 to the new opacity value. Dispersion allows for softer edges so the selection blends better into the background. This is done by having the alpha values decrease as they get farther away from the original chosen pixel.

2.2.3 User Interaction

The flow chart Figure 2.2.3 shows how the user interaction works. The RGB (PPM and depth(PMG) files are organized in chronological order using TIMESTAMP in their respective file names. Each PPM file is then paired with a PGM file by comparing TIMESTAMP.

A PGM file is then paired with the closest PPM file by their respective TIMESTAMP values. Once a PGM is linked to a PPM file it will not be compared again. If a PPM file does not find a pair it is thrown out and therefore never used. Figure 2.2.2 is an example of the user interface I designed and includes the tools available.

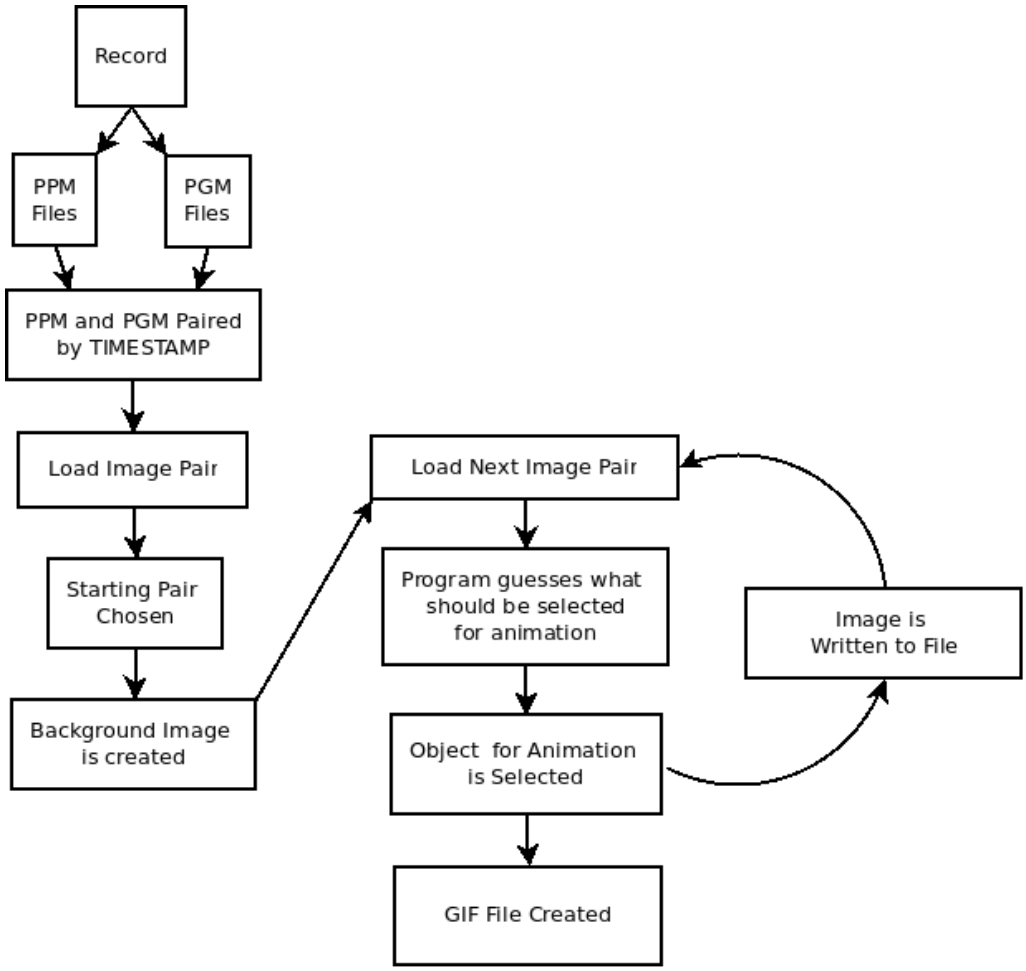

Figure 2.2.3. Breakdown of how the software works.

The skip and back buttons and the image select scrollbar allow the user to move through the image pairs within the sequence. Once an image pair of RGB and depth had been selected, the user could select a specific object for animation using the brush tools. There is a switch view which sets the unselected pixels' opaqueness to zero, and the opposite view where all the selected pixels are tinted red.

After the user hits the ACCEPT button, confirming that they have made their final object selection for a given pair, a foreground layer mask of the select pixels is created. This is done by changing the alpha values of the unselected pixels from the flood-fill to a opaqueness of zero. The first time an image is accepted in a given GIF creation set, the background image is created by changing all the selected pixels to a translucency of 0. These values marked as 0 are replaced by the average of all the corresponding pixels in the rest of the images. This background is maintained throughout the remainder of the program. A new image is then created by superimposing the foreground layer mask onto the background image. This process can be seen in Figure 2.2.4. The final image is then exported as a tiff file. Tiff files were used because they are an uncompressed file format and, therefore, prevent data loss.

Figure 2.2.4. An example of the processes by which the final image is formed. $Left$ the background image, *center* the foreground layer mask, and *right* the image produced.

After the image has been written to file, the next pair is loaded in and an estimate is made based on the previous pixel selections as to which pixels should be selected in the current image. This process is done using an implementation of the k-nearest neighbor algorithm[5] which was adapted to represent each pixel from the previous image as a x, y, R, G, B and D tuple. Each value is made uniform to make it between 0 and 1. The x and y dimensions have a weight of 2, R, G and B values have a weight of 3 and D value has a weight of 1. These weights were found empirically. Each pixel in the current image locates

[5]The library can be found here: http://www.stromberglabs.com/posts/8/k-means-clustering

its approximate closest neighbor within the coordinate system using Euclidean distance. The current pixel is then assigned to selected or unselected based upon the neighbor's selection status. This process is demonstrated in the following pseudocode and illustrated in Figure 2.2.5.

```
kd-tree = new tree()
for each px in classified image:
        x = <px_x, px_y, px_red, px_green, px_blue, px_depth>
        add x to kd-tree
for each px in new image:
        y = <px_x, px_y, px_red, px_green, px_blue, px_depth>
        find the nearest neighbor, n, of y in kd-tree and classify px as fg(n).
```

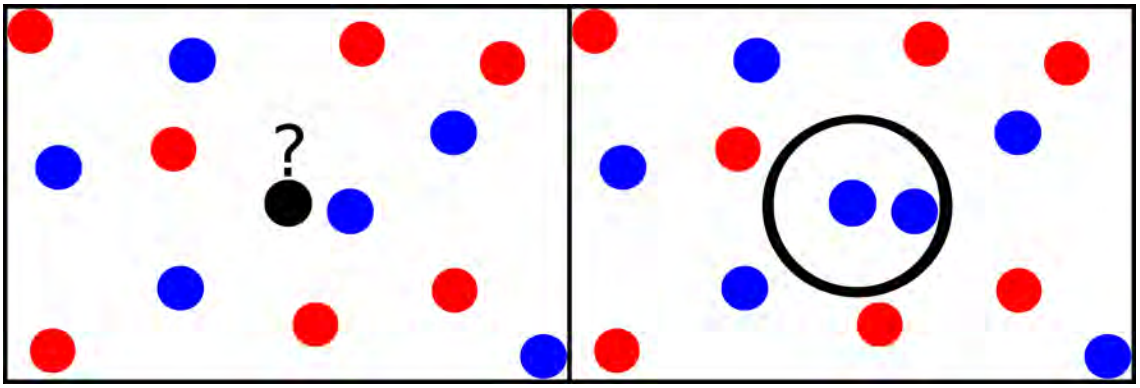

Figure 2.2.5. Example of k-nearest neighbor classification. The pixel's closest neighbor was blue so the unknown pixel is classified blue.

2.2.4 Creating the GIF

Once the user has made a final selection of edited tiffs, they are loaded back in one by one. This method is preferable to loading the selections all at once because all of the images have to exist in memory before the GIF can be written. Java's virtual machine is capped at 2 gigabytes which limits the length of the GIF, which is directly influenced by the number of images that can be imported. In other words, the bigger the images, the shorter the GIF because fewer images are imported. The individual edited tiffs are added to the GIF using

the GifAnimation Processing Library, written by Kevin Weiner[6]. By loading in each image one at a time, instead of all at once, longer GIFs can be created because less memory is needed since all that needs to be stored into memory is the GIF itself and the current image being added.

[6]The library can be found here: http://www.extrapixel.ch/processing/gifAnimation/

3
Results

An image-based evaluation method was used to analyze how well the three different components of the tools performed. The components were flood-fill, automated next frame selection using the k-nearest neighbor algorithm and the creation of each frame used to produce a given cinemagraph.

3.1 Analysis of Selection Algorithms

Figure 3.1.1 - 3.1.4 illustrate flood-fill. In 3.1.1 the upper left shows a flood-fill selection based on depth. The upper right and lower left show the selection based on the same points but use RGB and RGBD flood-fills respectively. The figure indicates that RGB and depth data separately are not sufficient to create a selection that is comparable with the ground truth. However, we get a clear indication that by combining RGB and depth data, it is possible to get an selection within 2 - 10 flood-fill selections that is reasonably close to the ground truth. The lower right is the optimal selection created using Adobe Photoshop.

Figure 3.1.2 is the image pair used to generate Figures 3.1.3 and 3.1.4 where the top is RGB and the bottom is depth. Figure 3.1.3 is a RGB flood-fill and 3.1.4 is a RGBD

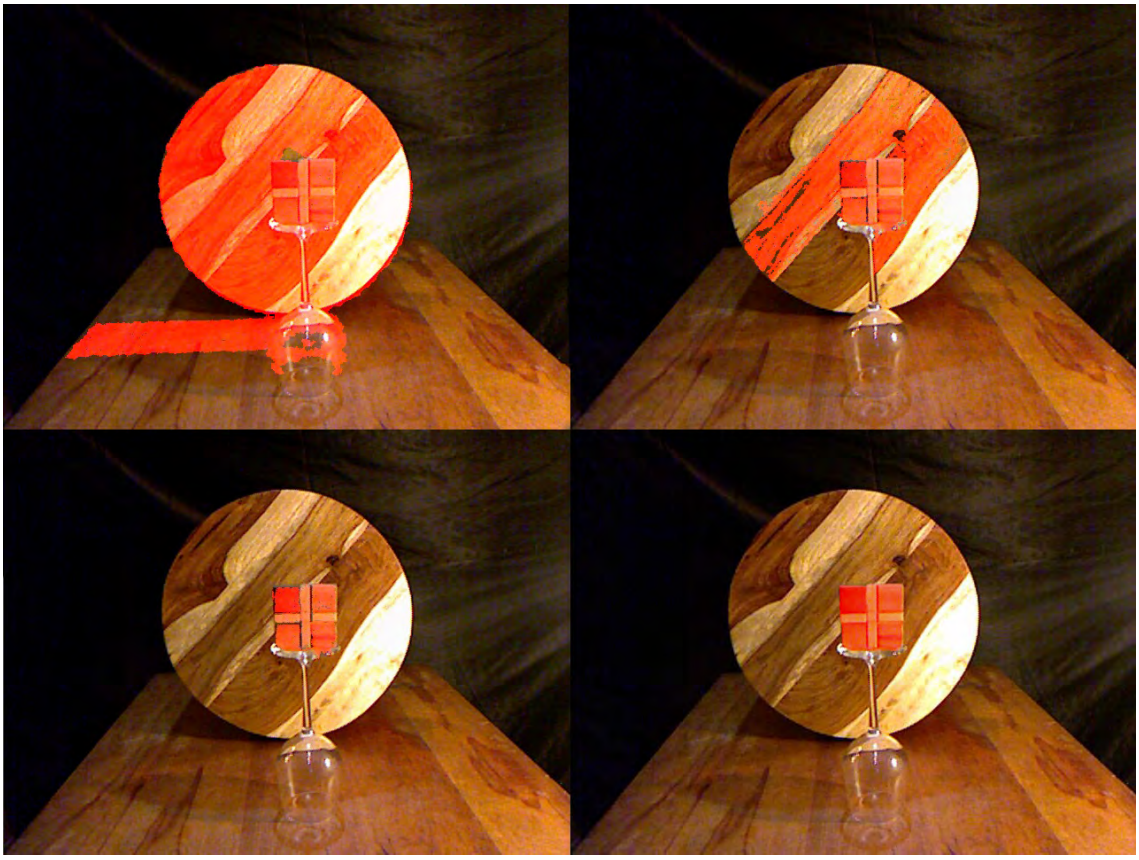

Figure 3.1.1. Illustration of optimal flood-fill selections against the ground truth image.

flood-fill. In both figures, each column represents a specific threshold value, corresponding to 20, 40, 60, 80 and 100 from left to right. The upper row represents the selection, the middle represents the isolated foreground and the lower row represents a mock cinemagraph frame. Figures 3.1.2 - 3.1.4 illustrate how much of an image was highlighted for a given threshold for different flood-fill versions. In Figure 3.1.3, the flood-fill, when only RGB is taken into account, shows that although the threshold value highlights the intended red within the boy's sweatshirt, parts of the background are highlighted as well. Although in RGBD flood-fill, Figure 3.1.4, the boy's sweatshirt is not completely filled with the maximum threshold value within the figure, the spread outside of the sweatshirt is within an acceptable region. Another aspect of adding the depth component is that it adds a level

of usability for picking a threshold. This is because a range of thresholds leads to the same resulting image, allowing the user to not have to find an exact threshold value to get a particular image.

Figure 3.1.2. The image pair used by Figure 3.0.3-3.0.4.

Figure 3.1.3. RGB flood-fill selection at various threshold intervals.

Figure 3.1.4. RGBD flood-fill selection at various threshold intervals.

3.2 Efficacy of Automatic-Selection

In Figures 3.2.1 - 3.2.3, the top two rows contain five contiguous input image pairs (t_0, t_1, t_2, t_3, t_4). From these input image pairs, the lower three rows of results are generated. The rows from top to bottom are RGB, depth, foreground, selection and the final cinemagraph image. The first frame is selected by the user. Successive frame selections are produced based on selections in the previous frames using k-nearest neighbor algorithm. In Figures 3.2.1 and 3.2.2 when the object selected moves very little the automated selection process works well, albeit with some noise around the object. Figure 3.2.3 is an example that shows that when there is greater movement of the selected object, the further the frames gets from the user input, and the accuracy of the selection of the object decreases. However, the automation does an acceptable approximation of what the next frame should be. As long as there is some user input for each of the frames, the resulting frames are acceptable.

3.3 Resulting Cinemagraphs

In Figures 3.3.1 - 3.3.9 the top frames contain five key image pairs. The lower two rows use the upper rows to generate results. The rows from top to bottom are RGB, depth, foreground and the final cinemagraph image. These images show that the selection tools overcome varying ranges of situations such as different amounts of depth data, similar color palettes and varying degrees of motion. The final cinemagraphs produced from those frames can be viewed in the supplement.

Figure 3.2.1. Results of automated selection without user intervention over time.

Figure 3.2.2. Results of automated selection without user intervention over time.

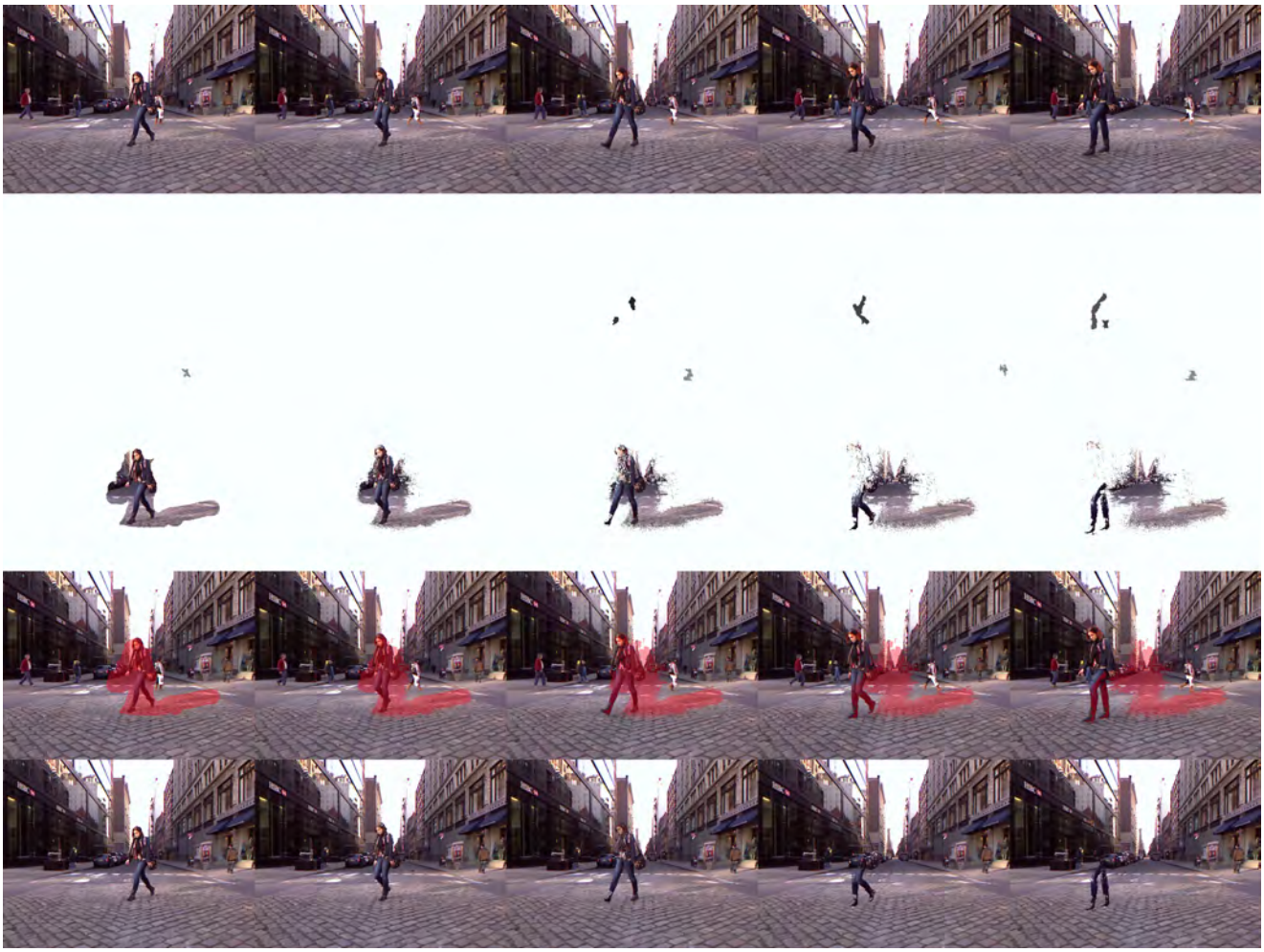

Figure 3.2.3. Results of automated selection without user intervention over time.

Figure 3.3.1. Final cinemagraph frame created from key image pairs.

Figure 3.3.2. Final cinemagraph frame created from key image pairs.

Figure 3.3.3. Final cinemagraph frame created from key image pairs.

Figure 3.3.4. Final cinemagraph frame created from key image pairs.

Figure 3.3.5. Final cinemagraph frame created from key image pairs.

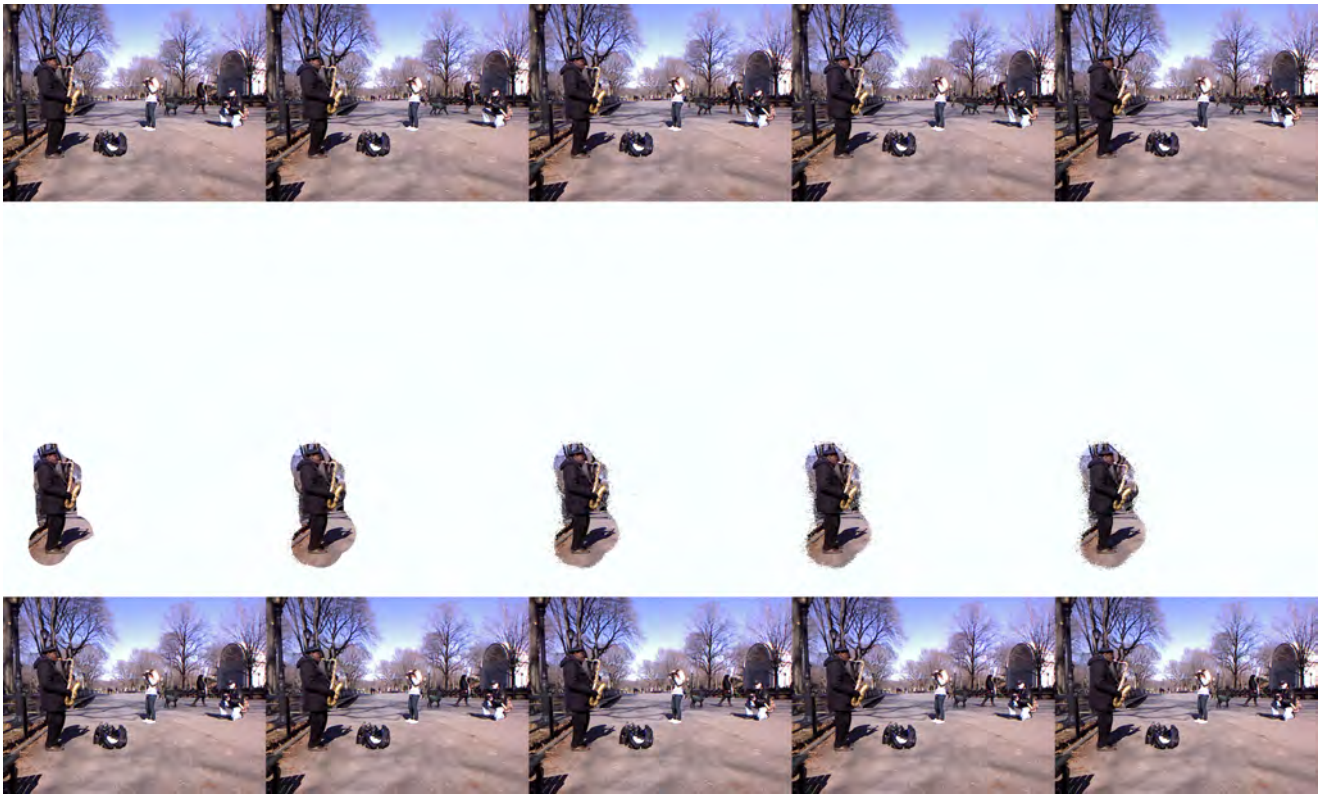

Figure 3.3.6. Final cinemagraph frame created from key image pairs.

Figure 3.3.7. Final cinemagraph frame created from key image pairs.

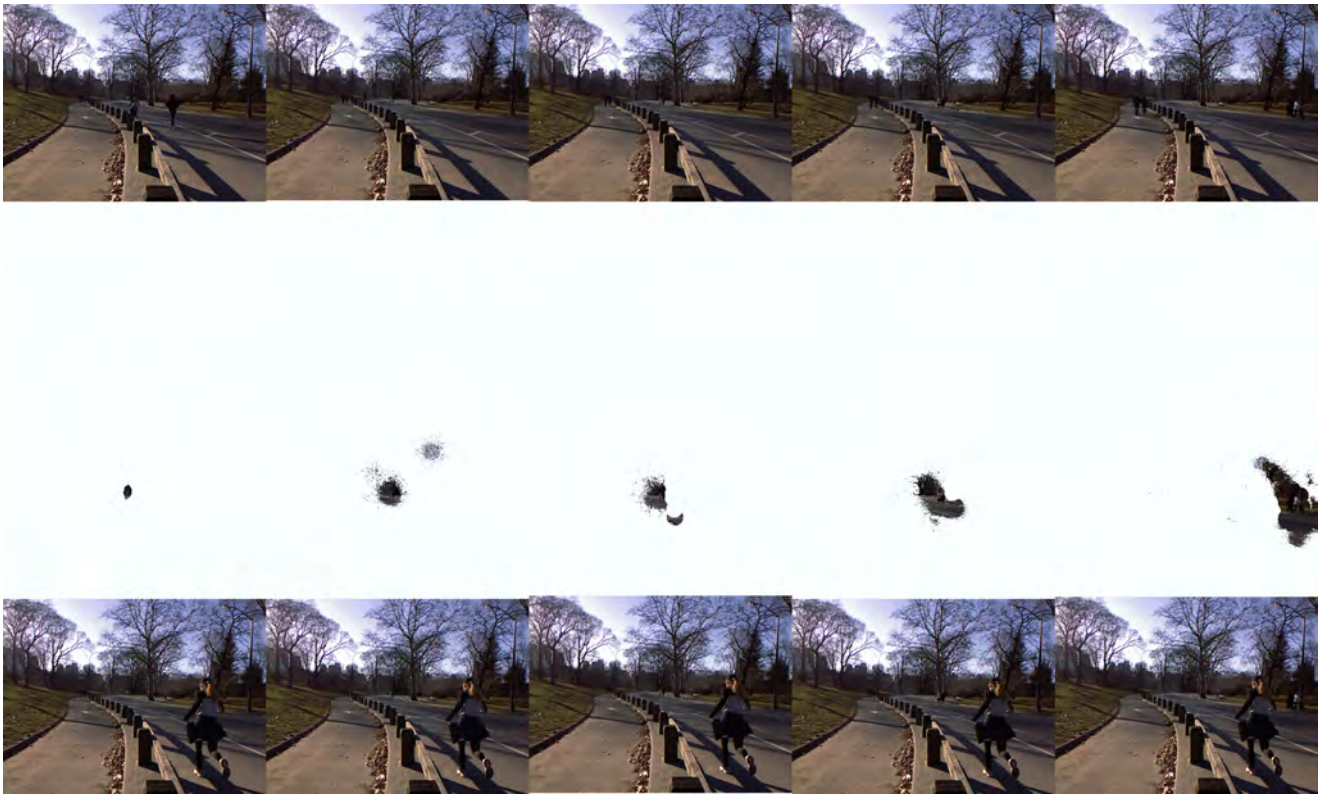

Figure 3.3.8. Final cinemagraph frame created from key image pairs.

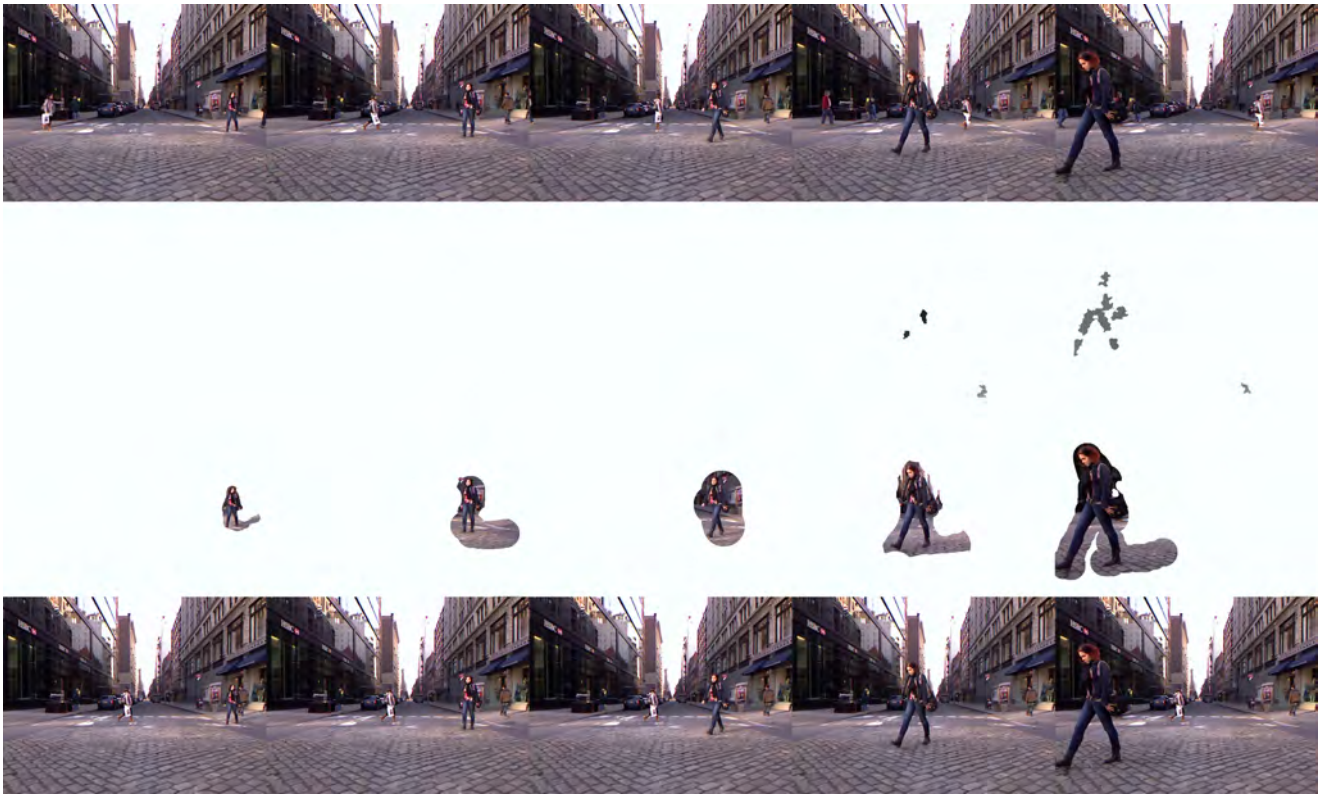

Figure 3.3.9. Final cinemagraph frame created from key image pairs.

4
Discussion

The objective of this project was to reduce the amount of user inputs required to create cinemagraphs. The first step was to include depth data within the process of image segmentation. The second step was to predict the next frame selection based on previous foreground selections. The implication of our results is that more accurate depth data can significantly improve the ability to perform image segmentation at a faster rate. This is because less clicks would be needed to make an accurate selection if more depth data was available to the user. However, the resulting depth images from the Kinect indicate that the depth images were highly affected by natural IR. This led to a lack of usable depth data in moderate to severe sunlight, hence showing the limitations of using the Kinect in an outdoor setting.

By using a RGBD camera to create cinemagraphs we see that depth data collected indoors leads to a highly automated process for generating cinemagraphs. The process works for an outdoor environment but is not highly optimal in its results. However, in most indoor conditions the color image has low tonality due to dim lighting whereas the outdoor setting provides good lighting conditions to generate high tonality color images. Kinect has

a built-in auto adjust property to deal with changing light conditions. This, however, gets in the way of the auto select process because the animated selection changes in darkness or brightness while the background remains static. This also applies to view ability in that the animation brightness varies in relation to the background image, which in some cases makes it difficult to seamlessly integrate the animated object into the cinemagraph's background image. It would be preferable to have more control over the final image through normal photographic controls, such as f-stop and shutter speed. This could be accomplished by mounting a stronger IR depth camera on top of a SLR camera.

The images tended to have a color cast from the auto-correction of the Kinect. I did not explore a method of correction because the color cast changed from frame to frame. A solution might be to create a tool similar to color curves in Adobe Photoshop. Since depth does not perform very reliably outdoors, a future pursuit would entail structuring the program to ignore depth when unreliable and unavailable and only using depth data when present to avoid wasting time with unnecessary computations.

In terms of using the Processing library, the Java applet on the screen refreshes only when the computation has completed and the code returns to the draw method. This means that when computationally heavy methods are used, there is a lag between the on-screen representation and what is occurring in the background. This problem affected usability when it came to the auto-selection using k-nearest neighbor algorithm because it is slow with more input dimensions. For example, while the k-nearest neighbor algorithm is assigning pixels to its closest neighbor the applet is frozen for 1-2 seconds. During that time span, if the user clicks anywhere on the screen it results in the program skipping to the next image pair and a second wait for 1-2 seconds occurs. An improvement in usability would be to find a way to have any visual changes rendered in real-time to the screen while also allowing the user to update the applet in real-time.

The animated cinemagraphs created for the exhibition *Still Moving* were successful on a technical level exceeding my expectations. I knew before I began that due to the size of the images produced by the Kinect, the quality of the cinemagraphs would be slightly compromised. That said, the cinemagraphs I created for the show conquered a range of technical issues. This joint project has provided me with a new academic avenue of study and a larger visual vocabulary to explore the role of movement in photography.

Bibliography

[1] Ryan Crabb, Colin Tracey, Akshaya Puranick, and James Davis, *Real-time Foreground Segmentation via Range and Color Imaging*, European Conference on Computer Vision (2008), 1-5.

[2] Dorin Comaniciu, Visvanathan Ramesh, and Peter Meer, *Real-Time Tracking of Non-Rigid Objects using Mean Shift*, Computer Vision and Pattern Recognition **Vol. 2** (2000), 142-149.

[3] Sylvain Paris and Fredo Durand, *A Topological Approach to Hierarchical Segmentation using Mean Shift*, Conference on Computer Vision and Pattern Recognition (2007), 1-8.

[4] Andrew Blake, Carsten Rother, M. Brown, Patrick Perez, and Philip Torr, *Interactive Image Segmentation using an adaptive GMMRF model*, European Conference on Computer Vision **Vol. 1** (2004), 428-441.

[5] Ching-Tang Hsieh, Eugene Lai, Yeh-Kuang Wu, and Chih-Kai Liang, *Robust, Real Time People Tracking with Shadow Removal in Open Environment*, Control Conference. 5th Asian **Vol. 2** (2004), 901-905.