Animating Pictures with Motion Textures
Adapted from "Animating Pictures with Stochastic Motion Textures" by Chuang et. al. 2005

Vivien Nguyen
CS 194-26 Fall 2018
UC Berkeley

Abstract

In this project, I implement a simplified version of "Animating Pictures with Stochastic Motion Textures" by Chuang et. al. from 2005 [1]. The system consists of three main steps: segmentation + matting, motion specification, and rendering. We can combine these steps in order to generate animated GIFs from a single image. Despite the simplifications made, the system can still handle a variety of input images and motion specifications, though it may require a significant amount of user input.

1. Introduction

I am largely interested in interesting ways that we can interact with existing photographs and artworks. Though many artists through Impressionism and Absolute Abstraction developed new methods for conveying motion in static images, I wanted to apply the methods in "Animating Pictures with Stochastic Motion Textures" to various pieces of artwork, as well as static photographs.

This system consists of several large pieces drawn from other papers and sources; as the original authors state, the main contribution of this system is the synthesis of these several techniques and a proof of concept.

1.1. Simplifications

In the efforts of making this project tractable in the period given, I made several simplifications to the authors methods. One is to use Photoshop to paint in regions in the Segmentation and Matting portions of the project (elaborated on below). But the major simplification made is the removal of "stochastic" motion textures, and instead using combinations of sinusoids. The original authors specifically warn against the cons of doing so – in fact, their experimentation with this approach and ultimate realization that physics-based simulations looked better is a major contribution of theirs. However, for the sake of this project and implementation, I believe the results gained from combining some hand-tuned sinusoids is sufficient to demonstrate the potential of adding motion to static pictures.

2. System Overview

As mentioned above, this system is quite large, consisting of three major steps and several substeps. I found it was quite natural to implement the project in these several pieces and process images sequentially according to these steps. One major downside to this is that certain steps must strictly happen in serial. This contributes to a substantially large processing time (though largely automatic), especially when using older computational photography algorithms on every day hardware.

Proceeding with an overview of the major steps, let's recall our goal: we want to animate several portions of a static image. Thus, our first step is to segment out the elements of the image that we would like to animate into layers. However, a simple cut and recomposite of these layers in different positions would result in harsh and obvious edges, detracting from the "reality" of the picture. After segmentation, then, we solve for an alpha matte that can be used when recompositing the layers. The final piece of layer generation is to inpaint the region created by segmenting out an element of the image. We now have a new "starting" image, that we can continue to segment out of. We proceed with layer generation moving from front to back, segmenting, matting, and inpainting the layers until we have all the desired elements, each on their own layer L_i.

Once we have as many layers as desired, we need to specify some rules of motion. Again, we use a simple model: sums of sinusoids. We classify several motion types, just like in the original paper: boats (bobbing motion), plants (swaying motion), water (actually the same as boat motion, but distributed across the y-axis), and clouds (simple side to side translation).

We can hand define motion functions for each of these types, as well as for each particular image (i.e. a boat in one

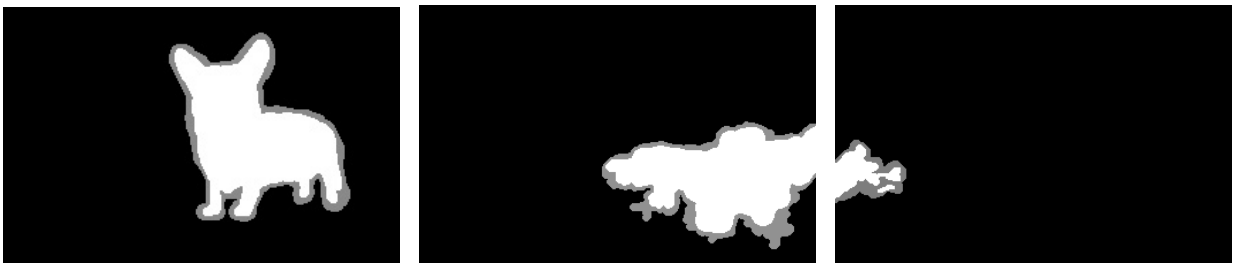

Figure 1: Example of trimaps made in Photoshop

image might behave differently than one in another).

After we have all our layers and motion functions, we can proceed with rendering in the painter's algorithm: back to front. For each layer L_i, we compute a displacement at time t relative to its initial position, apply that displacement to both the foreground image and alpha matte, and composite all the layers.

2.1. Segmentation

The original authors used a cutting method such as "Intelligent Scissors" (finding minimal graph cuts) to segment out foreground elements from the original image. Rather than re-implementing this tool which strictly requires a functional user interface, I decided to simply use Photoshop's scissors tool. However, I soon realized that it was actually much more simple and reliable to use the paintbrush to paint in the foreground, background, and unknown (used for Bayesian Matting, see below) regions.

Though each layer should be segmented after the previous has been inpainted, I found it was more efficient to just paint in all the layers at once, approximating how much the "hidden" foreground layer would take up. See Figure 1 for examples of these created trimaps.

2.2. Matting

Note in the trimap that there are, as the name suggests, three regions: white, black, and gray. The white region represents the "known foreground", the black represents the "known background", and the gray is the "unknown region". How is this useful to us?

Recall that we will be translating our foreground layers around the background. If we do a simple cut and paste, we will end up with some strong edge artifacts. However, we can't make use of something like Gradient Domain Blending from Project 3, since we would definitely like to keep the pixels in the known foreground the accurate color. The original authors therefore use some of their earlier work, "A Bayesian Approach to Digital Matting" (also Chuang et. al. 2001) [2].

Matting is the technique used in green-screen editing. We know in the main foreground region that it is the solid

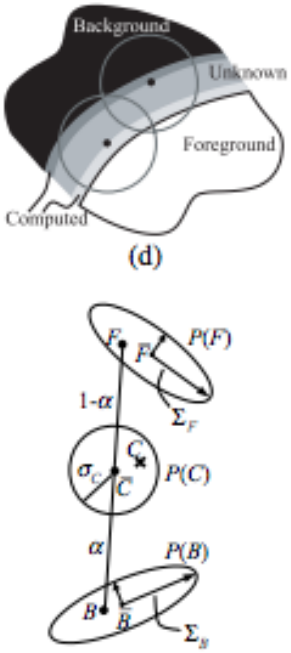

Figure 2: Illustration of Bayesian approach

foreground color, but at the edges, the foreground pixels are influenced by the background they are on. Thus, Bayesian Matting allows us to separate what is the foreground color, the background color, and the alpha matte that we can use the recomposite them.

In Bayesian Matting, we use the distribution of the local neighborhood to a pixel in the unknown region to compute the MAP most likely estimates of the foreground F, background B, and alpha matte a, given the observed color in the unknown region C. See figures 2 and 3, sourced from Chuang's paper.

We initialize alpha values to be 1 in the known foreground and 0 in the known background. We initialize alpha for each pixel in the unknown region to be the average alpha value in the neighborhood, and alternate between solving for the most likely F and B, and for the most likely a.

$$\begin{aligned} & \arg\max_{F,B,\alpha} P(F,B,\alpha \mid C) \qquad (4) \\ = \; & \arg\max_{F,B,\alpha} P(C \mid F,B,\alpha) P(F) P(B) P(\alpha) / P(C) \\ = \; & \arg\max_{F,B,\alpha} L(C \mid F,B,\alpha) + L(F) + L(B) + L(\alpha), \end{aligned}$$

Figure 3: MAP estimate == Max Log Likelihood

In Bayesian Matting, the authors recommend clustering colors in the neighborhood region using Orchard and Bouman's method [3], but I found that a k-means clustering method was sufficient.

The output from this step is an alpha matte, a computed foreground, and a computed background. See figure 4.

2.3. Inpainting

Now that we have a segmented region and its computed alpha matte, we'd like to proceed in generating layers. However, our background image now has a large hole in it! To continue, we will first have to inpaint this region. Moreover, when we animate the foreground layers, pieces of the background will show through, since the foreground is moving.

For inpainting, we will use Criminisi's "Region Filling and Object Removal by Exemplar-Based Image Inpainting" [4] algorithm, which is a patch match based inpainting method. It is relatively naive and straightforward, but since none of our foreground layers move very much (at maximum, approximately 10 pixels in any direction), a simple inpainting method is sufficient.

The key to this inpainting algorithm is that it attempts to propagate existing structures (i.e. edges) in the source region, into the target region. Thus, we identify the fill front by applying a Laplacian to the target mask, then calculate priority values for each location p on the fill front. The priority value is based on the available pixels in the region ("Data"), and the confidence of pixels in the region. This involves calculating gradients in the image to prioritize points with strong isophotes.

Mechanically, we search through all patches in the source region of the same size as our target patch, compute the SSD between the available pixels in the target patch and the corresponding pixels in the source patch, then copy over the needed pixels in the highest matching source patch.

The result of inpainting is a filled in background image, that we can continue to segment layers from (or proceed to motion specification and rendering). See figure 6 for inpainting results.

3. Motion Specification

Now assuming we have N layers, we need to specify a motion for each of them. As in the original paper,

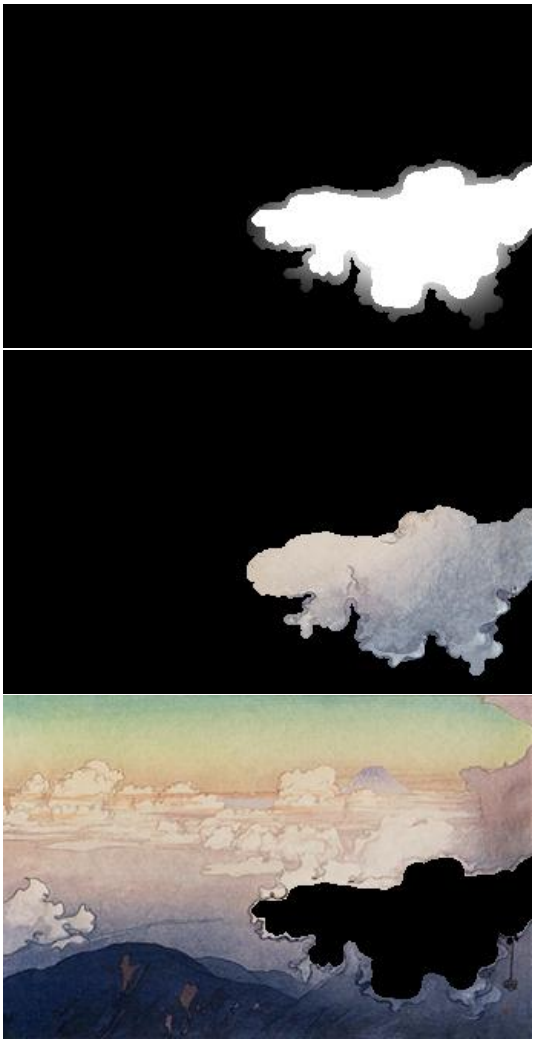

Figure 4: Results from Bayesian matting

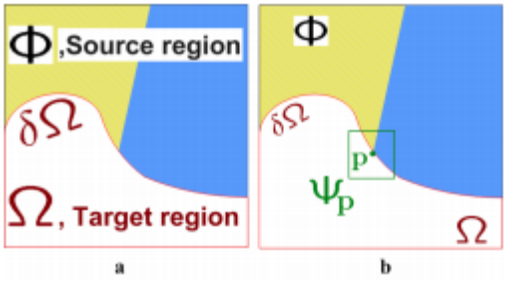

Figure 5: Results from Bayesian matting

Figure 6: Inpainting in process

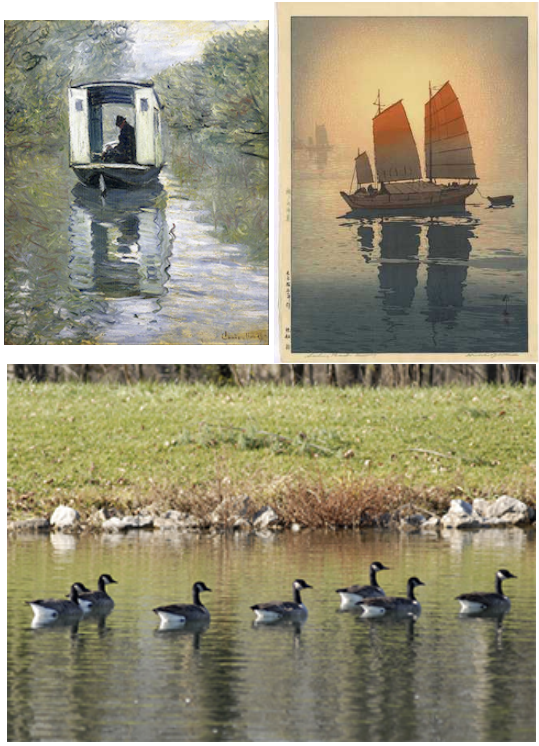

Figure 7: Examples with 'water' texture

we bucket elements into 4 different motion types: **water**, **boats**, **plants**, and **clouds**. Of course, we can also have static layers (no motion). The following sections will go into more detail about the specific motion textures used for each result generated. All the textures (except cloud) are resulting from summed sinusoids, but each has some unique added parameters.

3.1. Water

For water, we would like to create a cascading ripple effect across the body of water. In other words, we don't want the entire water region to bob up and down simultaneously. Since all of these water examples look at the water body head on, we can parameterize our sine functions by using a phase shift relative to the y-coordinate of the pixel we want to displace.

The resulting displacement is then just a simple shift in the y direction according to the sine wave.

3.2. Boats

For boats, we naturally want the boat's motion to be directly in sync with nearby water. To do this, we ask the user to define a *motion armature* for the boat. This only requires the user to click a point at the base of the boat. We then store that y-coordinate, and set the boat's motion to be in phase with the same sinusoid used for the water in that image, at that y-coordinate.

Again, the displacement is just a computed y-shift.

The bobbing motion of a boat extends nicely to other objects as well, such as ducks or circles.

3.2.1 Circles

For circles, we can also approximate these as boats, but we use a different sinusoid to create a bounce effect, more than bobbing. We construct a dampened sine wave, and parameterize the dampening rate and frequency by some value. This value is not the result of a motion armature, but a user specified number corresponding roughly to the size of the circle.

3.3. Clouds

Clouds are the simplest motion texture, as they are just a simple translation. For this, we can compute a shift in the x direction, relative to the time elapsed.

3.4. Plants

For plants, we would like to achieve a swaying motion back and forth (left to right). However, our methods for shifting boats or water up and down can not be directly applied in the x direction. We note that plants are rooted at the base, and thus the swaying motion is actually more akin to a rotation.

Figure 8: "Several Circles", Kandinsky

Figure 9: Sunflower Field

Again, we ask the user to define a motion armature from top to bottom of the plant. Rather than actually implementing a rotation, however, we can map the top coordinate and bottom coordinate to a [0,1] range. Then, for each pixel to be shifted, we translate the y coordinate of this pixel to a u coordinate in [0,1] and linearly interpolate the shift amount.

4. Rendering

To keep track of what we have now, we have N layers. Each layer L_i consists of an alpha matte, foreground element, motion texture, and perhaps some user-generated specification. We now proceed with rendering.

Rendering each frame is actually fairly straightforward. Since we went front to back in segmentation, we now go back to front for rendering. For each time step t, starting at the back layer L_0, we apply its displacement $d_i(t)$ to both the foreground element and the alpha matte, then apply the alpha matte to the foreground and current composite. We then move to layer L_1 and so on, until our composite is complete.

5. Results and Limitations

Overall, the system handles both paintings and photographs indefinitely, with arbitrarily many layers. Since the motion textures are not physics based, there is a lot of room for artistic interpretation on how to animate various elements of the image.

There are semi-notable remaining artifacts resulting from inpainting, likely resulting from an incorrect implementation of priority calculation. Also, the patch size in the algorithm plays a large part in its success. Naive patchmatch inpainting also accumulates error very quickly.

There are also some noise artifacts from Bayesian matting, so perhaps longer iterations for alternately solving for F, B, and a should be used.

Finally, the limitations for using simple sine functions rather than physics-based motion have been discussed at length.

5.1. Time Required

Manual segmentation and defining the trimap takes anywhere from 5-20 minutes, depending on the number of layers and complexity/level of precision desired.

Solving for the alpha matte runs roughly linear in the number of pixels in the unknown region, multiplied by the number of iterations, but is also affected by other factors. If not enough data exists to solve for the pixel at that time, it is passed and returned to later, when more information in the neighborhood becomes available. Matting per layer takes anywhere from 5-40 minutes depending on the size of the unknown region. It is most efficient when the foreground

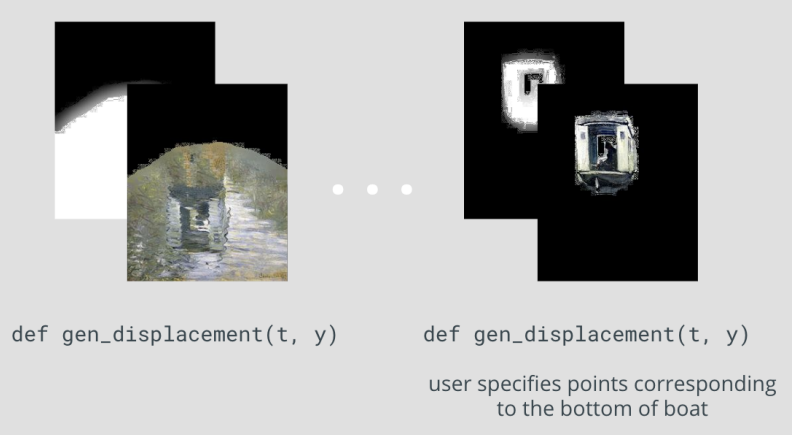

Figure 10: Elements needed for rendering

element takes up a small portion of the overall image, and thus generates a smaller unknown region.

Inpainting is on average the most time consuming part of this pipeline, since it runs in approximately *[num pixels in target region] x [num pixels in source region]*.

Rendering time is dependent on the number of layers and complexity of the motion texture. The most complicated renders take about 1 hour to complete (300 frames), and the simpler ones take about 15-20 minutes.

6. Conclusion, Personal Comments, and Future Work

We now have short clips or GIFs of originally static images, now with animated elements and a unique sense of liveliness to them. Even with simple motion models and relatively few layers of segmentation (the original authors have examples of upwards of 40 layers!!), we can create fun and charming results.

This system is probably the largest and most complicated I've ever implemented from scratch. Though existing source code is available for matting and inpainting (which I referenced, and cite below), it was very unclear how all the pieces should fit together. Thus, I'm quite satisfied with the results I was able to generate!

I hope to extend this work both in automation, and in domain. It would be nice to further automate both segmentation and motion specification (likely using learning techniques) to create a potentially completely automatic end-to-end system.

I would also like to apply these techniques to more images – my attempt on Kandinsky's "Several Circles" was moderately successful, but was incredibly difficult and required a lot of manual input. I'd like to find a way to work on abstract images more easily.

Finally, it would be nice to make these generated clips more interactive, perhaps introducing depth models to create a "Tour into the Picture" effect, or at least some parallax between layers.

7. References and Resources

[1] Yung-Yu Chuang, Dan B Goldman, Ke Colin Zheng, Brian Curless, David Salesin, and Richard Szeliski. Animating Pictures with Stochastic Motion Textures. 2005

[2] A Bayesian Approach to Digital Matting. Yung-Yu Chuang, Dan B Goldman, Brian Curless, David Salesin, Richard Szeliski. 2001

[3] M. T. Orchard, and A. B. Bouman. Color Quantization of Images. 1991

[4] A. Criminisi, P. Perez and K. Toyama. Region Filling and Object Removal by Exemplar-Based Image Inpainting. 2003.

[5] K-Means Clustering source code. https://scikit-learn.org

[6] Bayesian Matting reference. https://github.com/MarcoForte/bayesian-matting/blob/master/bayesian_matting.py

[7] Inpainting reference. https://github.com/igorcmoura/inpaint-object-remover/blob/master/inpainter/inpainter.py