Final Writeup: Clothing Texture Transfer

Project Goal

This project explored clothing re-patterning. After the user uploads a source image, we automatically detect the clothing edges. After the user uploads texture samples, they can use the GUI to draw regions on the edge mask to transfer each texture. We then synthesize a new texture that is a combination of the uploaded texture samples, and replace the pattern on the source clothing while preserving the folds and shadows.

Creating the Clothing Mask

In order to create the clothing mask, we utilized a sequence of processes involving Canny edge detection, binary morphology techniques, and the grabcut algorithm. This process was used to separate the foreground from the background, and then go through an iterative process to refine the separation and create the mask.

Otsu's Method and Canny Edge Detection

We began by first running Canny edge detection by converting the base image to grayscale and gaussian blurring the image to decrease noise in the edge detection. We then used OTSU's method of determining a threshold value to base Canny edge detect on. OTSU's method essentially determines a single intensity threshold that can be used to separate an image into two classes, foreground and background. This threshold is determined by maximizing inter-class variance. We then set a basic lower and upper threshold using that value using a threshold ratio value of 0.3 to return a lower and upper threshold value for Canny edge detection.

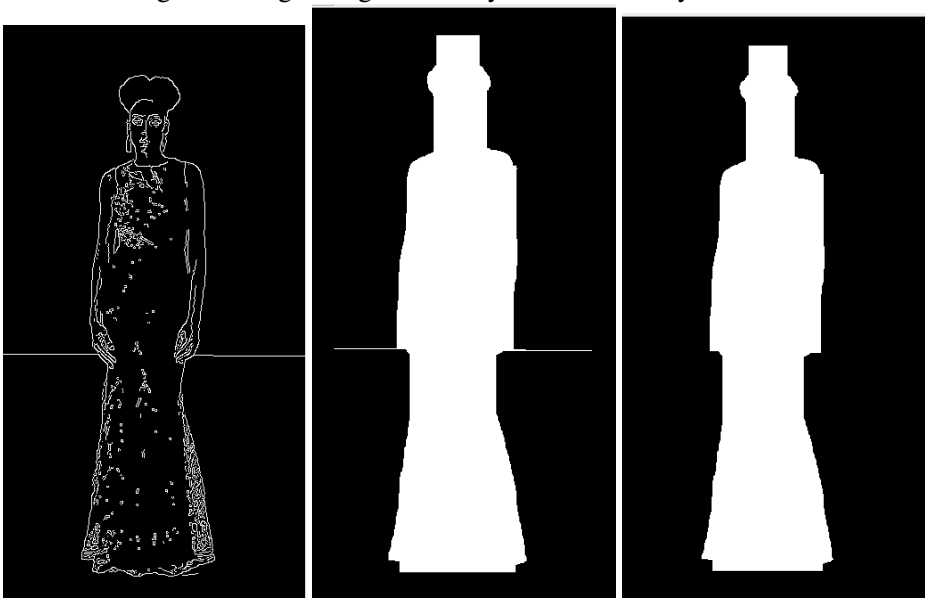
Canny edge detection functions by running a multistage algorithm that begins by denoising the image, and then using a Sobel kernel, similar to Sobel edge detection, to calculate an intensity gradient over the image. This intensity gradient is then further filtered down using non-maximal suppression, which checks if a pixel is a local maximum in its neighborhood in the direction of gradient. If it is, then it is suppressed and set to 0. The result is a binary image of thin edges. Afterwards, the resulting "thin edge" image is then checked for true edges using hysteresis thresholding, by removing any edges that are not connected to "true edges", which are defined by the upper and lower values given by OTSU's method in the previous step. All pixels that fall below the lower limit of the threshold are non-edges and all pixels that are above the threshold are part of a "true edge". Hysteresis thresholding checks the pixels within the threshold based on connectivity. If it is connected to a true edge pixel, then it is also a true edge, while if it is not connected to any true edge or connected to a non edge pixel, then it is removed.

Figure 1: Base image to Edge Image

Binary Morphology

Once we were able to get a proper edge image of the source image, we then proceeded to use binary morphology techniques to essentially fill in the mask. Since the edge image that we received was a binary mask of 1s and 0s, we were able to use a morphological techniques to refine the clothing silhouette mask. We began by first creating a 3x3 binary structure that would be used to define a neighborhood of a pixel that would include not only adjacent pixels but also diagonally adjacent pixels in order to better determine whether we should fill in the pixel within the mask or not. Using this structure, we performed multiple iterations of morphological binary closing, which would in its simplest form fills in the holes within the edges by first performing a binary dilation to expand the edges a pixel in every direction and then a binary erosion to remove the pixels that were just added on the outside of the edges. Performing this multiple times, about 30 iterations, would fill the edges in just enough for grabcut to better refine the mask by supplying enough possible foreground pixels compared to just using the edges as possible foreground pixels. To better refine, the mask, however, we also performed two iterations of binary erosion after the series of binary dilations to remove any noisy lines.

Figure 2: Edge Image to Binary Close to Binary Dilation

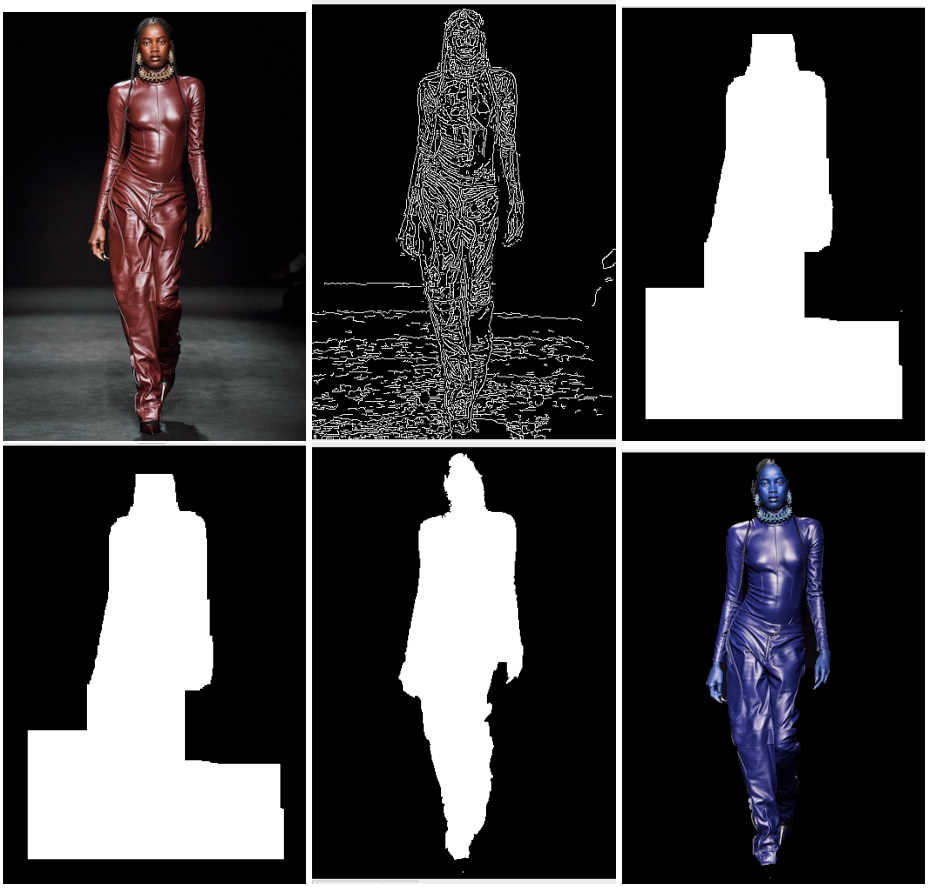


The Grabcut Algorithm

This now somewhat ”filled out” mask would then be used within grabcut. We began by first setting all the current 1 values that were filled out to become possible foreground pixels (the value 3 in grabcut) and all 0 values to become possible background pixels (the value 2). Grabcut then runs a Gaussian mixture model on the pixels to find a distribution of the possible background and foreground pixels. A graph is then built from this pixel distribution, where every pixel is a node in the graph. Two additional two nodes are then added, a Source node and a Sink node. Every foreground pixel is connected to Source node,while every background pixel is connected to Sink node. The weights of edges connecting pixels to source node or sink node are then defined by the probability of that pixel being foreground or background. The weights between the pixels are defined by edge information or pixel similarity. If there is a large difference between pixel values, then the edge between them will get a low weight. Afterwards, the graph is segmented via a minimum cut algorithm. It cuts the graph into two separating source node and sink node using a minimum cost function. The cost function is the sum of all weights of the edges that are cut. After the cut, all the pixels connected to the source node are set as foreground and those connected to the sink node are set as background pixels. This process is run over multiple iterations until the classification converges and the mask becomes more and more refined.

Figure 3: Morphology to Grabcut to Segmented Image

Figure 4: Full Clothing Mask Process



Synthesizing Texture

Afterwards, the clothing mask is then used to define the bounds of where we will be synthesizing textures. The process for this was similar to the image quilting by selectively choosing image patches and quilting them together using the minimum cut seams to create a synthesized texture to alpha blend on to the source image to maintain depth, shadows, and folds.

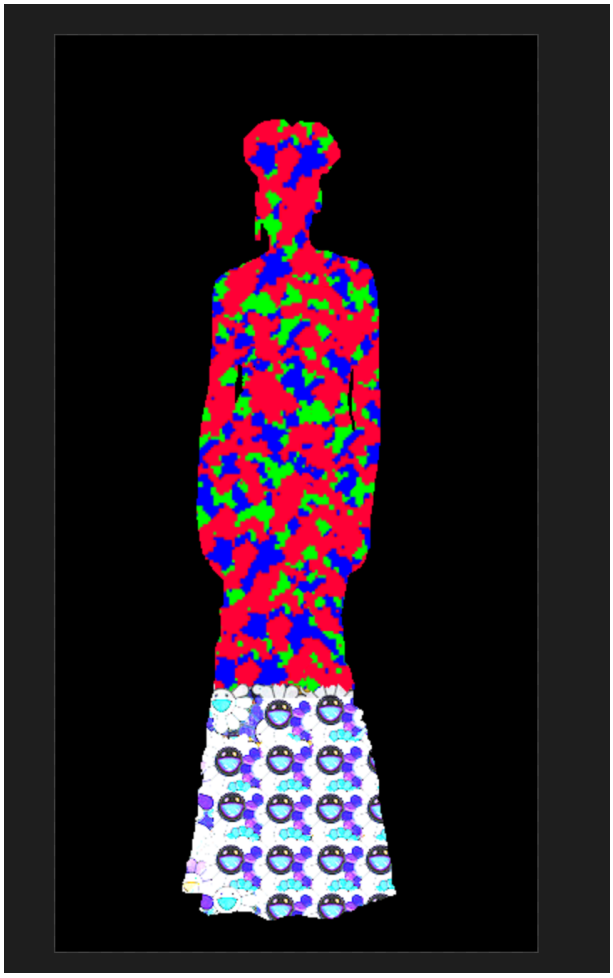


Figure 5: Resulting Multi-Texture Fill

Fast Exhaustive SSD Texture Patch Search

To start the process, we use the drawn in mask to designate specific areas to be used with different textures. The algorithm essentially chooses the correct texture based on the most common texture designated within that to fill patch. By using a combination of windowed views into the numpy array, we were able to drastically decrease the runtime

for an exhaustive texture patch search. We created all the possible patches using numpy strides and stride tricks, while all distance computations were performed by expanding the formula for SSD and then using numpy einsums in order to perform computations on windowed views of the arrays to overall speed up the synthesis of the texture.

Minimum Cut

Afterwards, minimum cut was used to create the edge by which the new patch would begin, preventing a naive overlap over an adjacent tile and instead creating more passable edges. This was done by creating an energy matrix over the overlap space similar to a seam carve energy matrix and then solving for the minimum path from the bottom of the overlap column and the far right of the overlap region. This creates a texture fill that has passable edges that does not look blocky especially between different textures.

Multi-Texture Fill

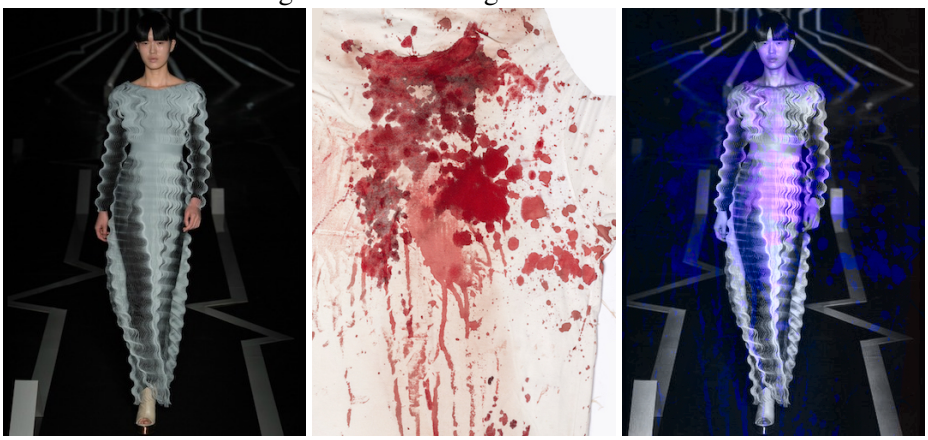
Multi-texture fill was performed by designating specific regions on the mask using the GUI to synthesize with a specific texture. The synthesis would select the most common texture within the patch to use for synthesis. By decreasing tile size over each iteration as well during synthesis, we were able to create better lines and create thinner sections of an individual texture because the algorithm would overwrite a larger texture with a smaller patch of the true most common texture within that region of the mask.

Figure 6: Multi-texture GUI and a Resulting Texture

Transferring Texture

To transfer the texture, we experimented with the LAB color space to separate light and shadow from color. We first tried combining the L channel (perceived brightness) with the A and B color channels from the synthesized texture, which yielding interesting but inaccurate results.

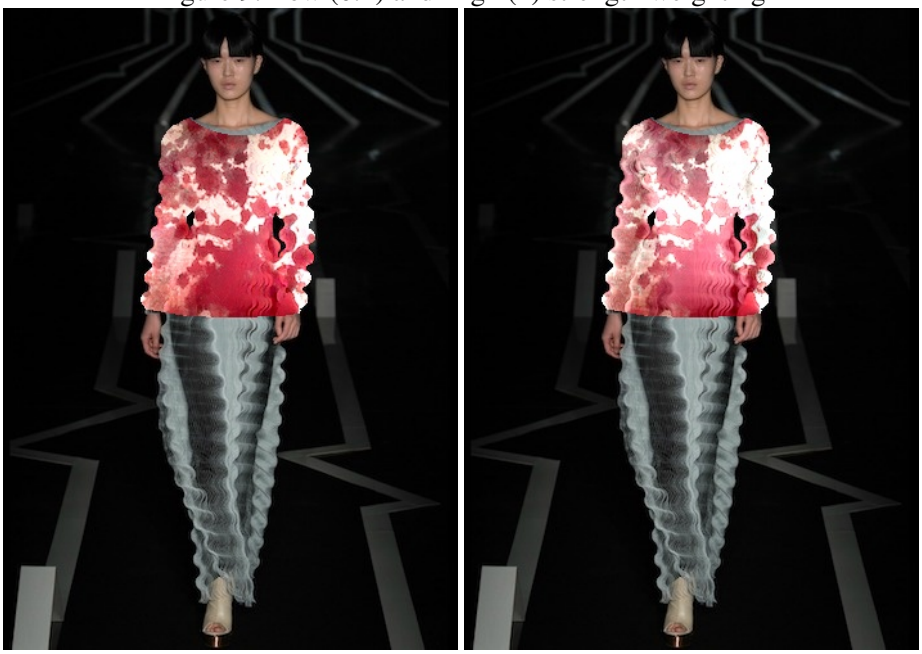
Figure 7: Combining LAB channels



We then tried using gradient domain blending to overlay the shadows from the original garment onto the texture. We used $cv2.createCLAHE$ (Contrast Limited Adaptive Histogram Equalization) to increase the contrast of the L channel, so the shadows and folds would be more clear. We built A and b in a similar way to the Poisson blending project, but we added mixed gradient blending as to only apply the high contrast shadows. When comparing the gradients of the source and target image, we observed that some busy textures overpowered the shadows. To solve this, we added a strength variable that weights the source gradient, and we were able to tweak this variable for extreme transfer cases.

Figure 8: Different textures with the same strength

Figure 9: Low (0.1) and High (2) strength weighting

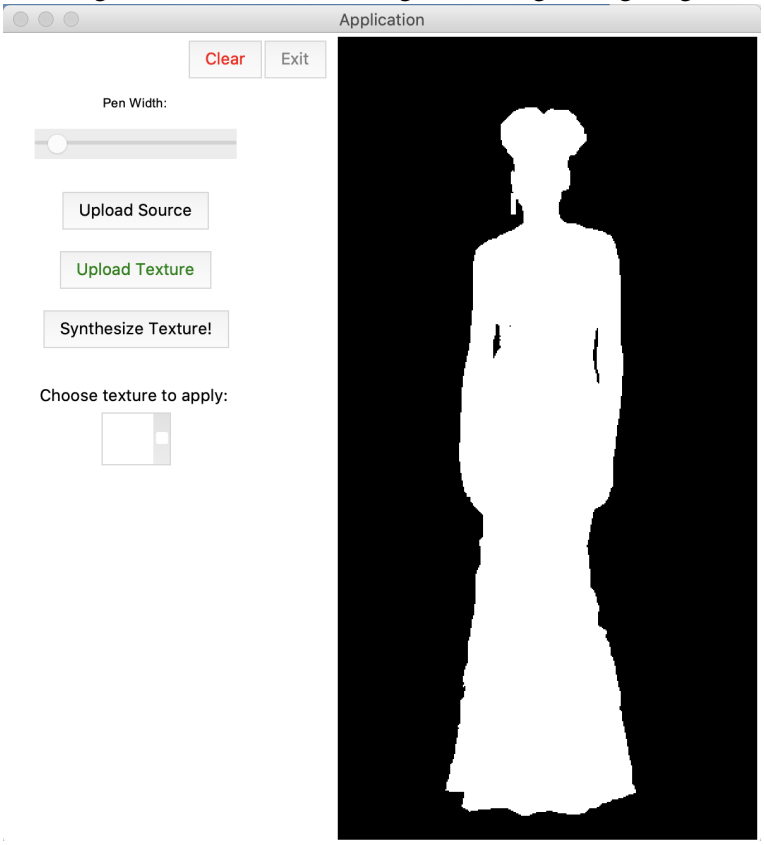


GUI custom mask creation

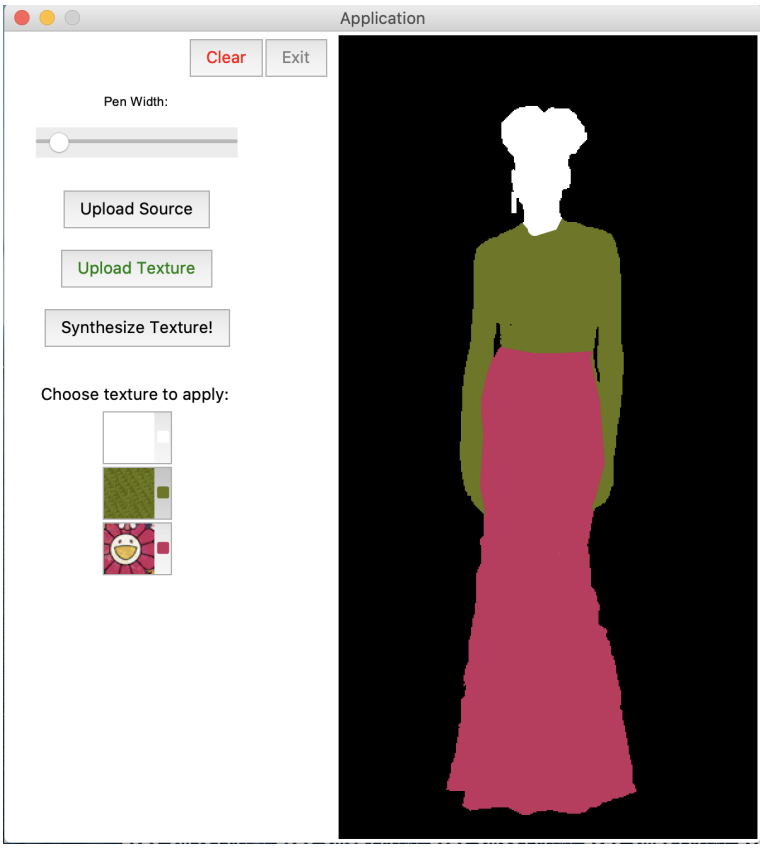
In order to make it easy to test many different sources and textures, we created a GUI using tkinter. The GUI also enables the user to quickly create and modify custom texture masks.

The user first uploads a source image, and the GUI automatically calls create_clothing_mask(), then changes all of the white pixels to transparent. The new transparent mask is continuously drawn on the canvas so the user can not transfer texture to the background.

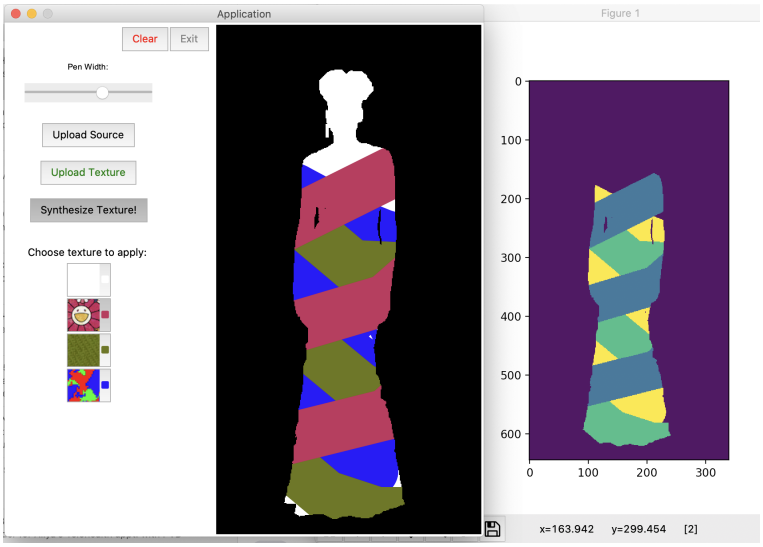
Figure 10: Low (0.1) and High (2) strength weighting



When the user uploads a new texture, we automatically calculate the most common color in the texture to use as an indicator color (sourced from https://stackoverflow.com/questions/3241929/python-find-dominant-most-common-color-in-an-image). The user can then draw on the canvas with the mouse with each texture color (https://www.youtube.com/watch?v=kzp7-0EFrIg). Anything that is left white or ”erased” with the white texture will be left as the original source image.



After adding any number of textures to the mask, the Synthesize Texture button starts the transfer process. First, a final ”multi mask” is calculated by replacing each hex color with an integer corresponding to the order the textures were added. (The first texture is 1, the second is 2, etc). Everywhere not drawn on the mask is zero, even if the original calculated mask included it. This mask is passed to synthesize_texture() and transfer_texture() to create the final output.



Final Results