Developing an Application to
Convert Real Images into Cartoons using
De-pixelization, Segmentation and Rescaling with Restricted Colour Palette

Course: COMP4102

Members:

Benjamin Melone	101034994
Ryan Kane	101041831
Eimhin McDonald	101043694
Joseph Dooley	101040004

Date: 18/04/2020

Abstract

This project turns real life pictures into their cartoon counterparts. This project uses an interactive segmentation tuning and selection, high frequency enhancements, scaling, colour quantization and finally de-pixelization. The project deliverables are a 5-6 minute presentation with slides about this project, the code base, a final write up document, input and output image examples, and finally a video of the application running. This project has many challenges to overcome, the main challenges are to separate the object from the background, maintaining distinct colours between segments after downscaling and quantizing the colours, and finally transforming the small quantized coloured image into a scalable vector graphic.

1- Introduction

Our project implements three main algorithms to achieve the cartoon image output. The first step is a segmentation algorithm combined with a segment selection and parameter fine-tuning application. The second step is to pixelate without losing important edges, and quantizing the colours. The last step of the algorithm is to depixelate the output from the previous step into a scalable vector image. As a final optional step, a background can be added behind the cartoonized foreground. This project is a good candidate for vision applications because it processes image data to determine areas for cartoonification and manipulates those areas in the image and outputs them as a new image. The manipulation of the specified areas is a good candidate for vision applications because it can be used to take any pixel art or image, find its vectors and create a new image at any scale. This project is challenging because it requires a lot of preprocessing to create a pixel art usable for the de-pixelization, and the de-pixelization itself is challenging as it needs to create a clean planar similarity graph and detect shape boundaries.

2 - Background

Three papers were referenced while developing this project. The paper titled Efficient Graph-Based Image Segmentation from Felzenszwalb and Huttenlocher describes a method a segmentation algorithm that uses graph-based methods to segment an image [1]. This segmentation algorithm serves as the basis for many graph-based segmentation algorithms, and was applied to this project as a base example for how to develop a segmentation algorithm. Next, the paper titled Efficient Algorithms for Hierarchical Graph-Based Segmentation Relying on the Felzenszwalb–Huttenlocher Dissimilarity describes methods that improve the segmentation [2]. The methods described in this paper were not applied in this project as they were too complicated to reproduce, however ideas from this paper were used to improve the segmentation. Finally, the paper titled Depixelizing pixel art was used to determine how to convert pixel-art into a scalable vector image [3]. This conversion is the core of how pixelated images are turned into cartoonized images. Two software packages were used, the OpenCV library version 3.4.9 and code references provided by Felzenszwalb on his personal website.

3 - Approach

3.1 - Segmentation

The first objective of this project was to segment the image and extract the foreground. This is a notorious problem and the best solution is probably to use machine learning algorithms. For this project, the chosen approach was to use a graph-based segmentation algorithm and then have the user select the foreground with their mouse. This process emphasizes correctness by forcing interactivity from the user. The segmentation algorithm chosen was the algorithm described by Felzenszwalb and Huttenlocher because it was easy to understand, it has satisfactory results, and it runs quickly [1]. This algorithm works as described in Algorithm 1 below.

Algorithm 1: Felzenszwalb and Huttenlocher's Segmentation algorithm
Input: An input image *Img*, a sigma value σ, a threshold value *c*, a minimum segment size *min*
Output: A segmented image, a disjoint set

1. Perform a Gaussian blur on *Img* based on σ
2. For each unique pair of adjacent pixels in *Img*: build a list of edges *E* that contains the locations of of the pixels that make up the edge, and the dissimilarity of these pixels
3. Sort the edges by their dissimilarity in increasing order
4. Create a disjoint set *S* with each vertex (pixel) in its own set. These represent the segments
5. Create a list of thresholds *T* that defines a threshold for each vertex, initialized to *c*
6. Iterate through *E* and merge sets if the vertices belong to different sets, and also both have thresholds less than the vertex thresholds held in *T*. When merging sets, update the new threshold value in *T* to be the edge weight plus the threshold divided by the size of the set
7. For each edge that spans different sets, merge the sets if both of the sets' sizes are less than *min*
8. Create an image where every pixel in a set is given the same random colour, for display purposes

This algorithm creates an image that displays the segmentation, and also a disjoint set which allows the user to check which segment a pixel belongs to. This algorithm was taken from sample code provided by Felzenszwalb and Huttenlocher, but was modified to provide compatibility with OpenCV, compatibility with png and jpg image types, and to return the disjoint set after segmentation. Originally, it was intended to improve this algorithm with the hierarchical segmentation techniques described in [2] but it was decided that these improvements were outside the scope of this project. With the segmented image, the user can then interactively click on the segments that make up the foreground, and the selected segments are stored. When the user presses a key, the non-selected segments are removed from the image, leaving only the foreground. With the foreground extracted, the foreground image is passed to the Pixelate function.

3.2 - Pixelation

The Pixelate function was created in order to preprocess the inputted image for the de-pixelization step. The de-pixelization takes a small pixel-art image that has a simplified colour palette to perform vector calculations and render a de-pixelated image. So Pixelate needs to resize an image but avoid aliasing, and convert pixels to a small colour palette.

The process originally was to gaussian blur then rescale by a given factor but this resulted in segments merging together in the colour quantization step. To prevent this issue, first a canny edge detection is used on the original image. Next the output from the edge detection is put through a contour finder, and the contour is redrawn onto the image with a line thickness of 1. This allows the important edge features to survive the gaussian blur and resizing. This is the best approach compared to the other methods tested. Using a sharpening kernel gave a slightly increase in high frequency but was almost all removed when blurred and scaled down. Another option was to subtract the low frequency from the image to highlight the high frequency, this resulted in the features still disappearing when scaled down.

3.3 - Colour Palette Restriction

Once the image was scaled down the colour needs to be quantized so that it looks like pixel art for the depixelizer. There are three methods for quantizing the colour in the scaled image: *basic*, *nBit*, and *k-means*. Each method can be chosen and passed into the Pixelate procedure to choose which quantization method should be used. These three have pros and cons for quantizing colour: *basic* requires a custom colour palette to be inputted; *nBit* can only quantize to a constant number; *k-means* can take longer on bigger images.

The approach for *basic* is to take in an image and a list of colours and to map each pixel in the image to a colour in the palette of colours. To do this the first step is to initialize a minimum distance to a very high number and a chosen colour to black. Then iterate through every pixel in the original image and compute the colour distance between the original image colour and every colour in the palette. The colour distance is:

$$colourDist = \sqrt{(origB - PaletteB)^2 + (origG - PaletteG)^2 + (origR - PaletteR)^2}$$

If the colour distance is smaller than the stored minimum distance then the new colour distance is stored as the new minimum distance and the chosen colour set to that colour palette. After all colour iterations the colour in the colour palette with the smallest colour distance will be chosen and finally it replaces the old colour. The big o notation of this algorithm is O(n*m*c) where n is the rows, m is the columns and c is the amount of colours in the colour palette. This is the best approach to quantizing colour to a specified palette because it allows for 0 to infinite possibilities of colours to apply. Normal quantizing of colours either keeps the same colour theme or is all the same hue, however this algorithm can match any set of colours to an image making it more similar to a piece of pixel art. The biggest limitation of this algorithm is that an image that is made from many similar colours may have all of its pixels mapped to the same colour. For example an image of a person wearing a hat that is the same colour as their skin tone, although they are two distinct surfaces, the colours may be merged together.

The approach for *nBit* was to convert the image from a 24 bit colour image to a smaller bit format and then back to 24 bit. After pixelating the image, the method was to simply convert the colour values to a smaller bit format system and then back to the original 24 bit format. This compresses the amount of colours because the format with less bits stores less unique colours so the colours that cannot be expressed are lost. What is essentially happening is the pixels are mapped into equal intervals where the amount of intervals is 2 to the power of n. The intervals each represent a unique colour in 24 bit format. This idea came from the fact that older video games were limited not only in their resolution, but also limited in the amount of colours that they could display. The SNES for example used a 15 bit colour system, but this amount of colours still appeared to be too many and did not solve the problem. We found that the best results from using this method were achieved by converting the colour format to 6 bits, making the number of total unique colours 64. This approach is very fast as it runs in linear time. However since the colours are basically sorted into equal intervals that are merged into one colour through the conversion, this approach falls also victim to the same problem as the *Basic* method where images with many similar colours are merged together into one colour.

The approach for *k-means* is to quantize the colour using kmean from OpenCV. This algorithm was created to give variety to quantizing colours, as this one keeps the colours of the original image but reduces the total amount of colour based on the input K. First the image is converted to 32 byte float and a data matrix is created so that every pixel is in its own row. Then the k-means procedure from open cv is called with K splits, team criteria of 10 iterations to an accuracy of 1.0. The resulting center, label and data are reshaped to have 4 channels, the fourth is for the alpha channel. Then for each row in the data matrix, the label matrix's colour is set to the corresponding row in data. Finally the data image is returned with the colour quantized by the parameter K, which is the number of colour clusters required. This approach is good for keeping the same colour style as the original image, and finds the best K colours to match the image. It also solves the problem from the previous two methods, where similar, but distinct colours are merged together. Since the kmean procedure varies the colours to the best representation of the image, similar, but distinct colours are preserved. A downside is that in very large images this process can be slower.

3.4 - De-pixelization

Once the image has been down-scaled and the colour space has been reduced, the de-pixelation algorithm can be used to create a cartoon-like effect. The first step is to create a similarity graph for the pixels of the foreground image. The similarity graph is a representation of the image where each pixel in the image is represented as a node. The edges for a node are determined by checking the 8 neighbouring pixels. If a neighbouring pixel passes some similarity constraint with the current pixel then an edge is added between those to nodes in the graph. The similarity constraint used in the algorithm is that the pixel colours are exactly identical. This differs from the de-pixelization paper, as our algorithm had previously reduced the colour space, thereby setting similarly coloured pixels to the same colour. The resulting similarity graph has a number of nodes equal to image length times width, and a number of edges up to image length times width times 8. This graph is efficiently stored in an grayscale OpenCV matrix with the same length and width of the down-scaled image, with the existence of an edge to a neighbouring node stored in the grayscale pixel byte. Thus each byte in the matrix represents a bit-set mask of the 8 edges for a given pixel. For the next step of the de-pixelization process, the similarity graph must be planar. Currently, planarity of the graph is not met because diagonal edges may cross each other. If the crossed edges are from the same pixel colours, they can be removed freely. Otherwise, If a checkered pattern appears in the pixel data then one of the diagonal edges must be removed. Remove one of these edges will break a diagonally connected colour region.

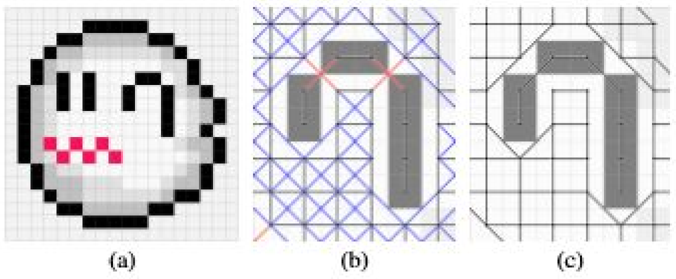


Figure 1: Depiction of the similarity graph [3]. (a) The input image. (b) The non-planar similarity graph of the ghost hand. All blue diagonals can be removed freely. Red diagonals form a checkered pattern where one must be removed. (c) Planar similarity graph for the ghost hand.

The next step of the de-pixelization process is to extract the dual graph from the planar similarity graph. If the similarity graph had nodes in the center of the pixels and edges from between the centers of two nodes, then the dual graph will have nodes in between these edges. Ignoring the diagonal edges in the similarity graph, the nodes in the dual graph could have positions quantized on the corners of the pixels of the image. When a diagonal edge exists in the similarity graph, the dual graph node cannot be quantized on the corner of a pixel because the diagonal edge lies on this corner point. Thus, diagonal edges split a corner node in the dual graph onto opposite sides of the edge. These split nodes are quantized such that they are shifted both ¼ pixel vertically and horizontally away from the corner on both sides of the edge. Furthermore, nodes in the dual graph that fall on a corner that is shared by similar pixels are removed. This leaves only nodes that fall on the border of two differing pixel colours. The edges of the dual graph are added to connect the nodes that are shared by neighbouring pixels. Thus, the constructed dual graph can be plotted on a 2-D cartesian plane and will resemble the borders between differently colour pixel regions, but diagonally connected pixel regions have been smoothed. The dual graph is then processed so that it is split into a list of edge chains, by splitting edge junctions on nodes with degree greater than two. This essentially ensures the graph is represented as a list of varying length node sequences. These sequences can be interpolated and rendered on the screen to display the borders of the de-pixelated image. The de-pixelization paper uses b-spline curves to interpolate the sequences, but our algorithm uses a linear interpolation as the result is generated more quickly and visually looks sufficiently good when upscaling the down-scaled image to its original size.

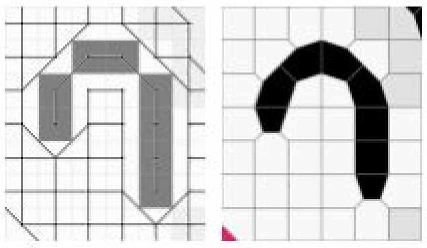
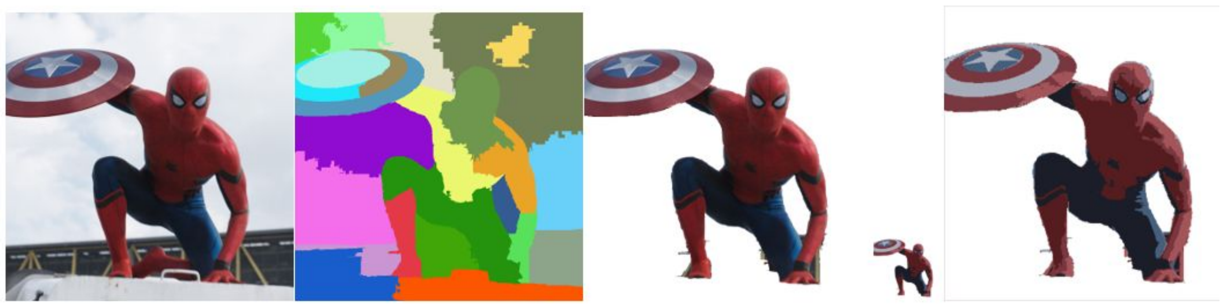


Figure 2: Example of the planar similarity graph on the left, with its corresponding dual graph on the right. The dual graph has nodes on the corners of the pixels from the similarity graph image [3].

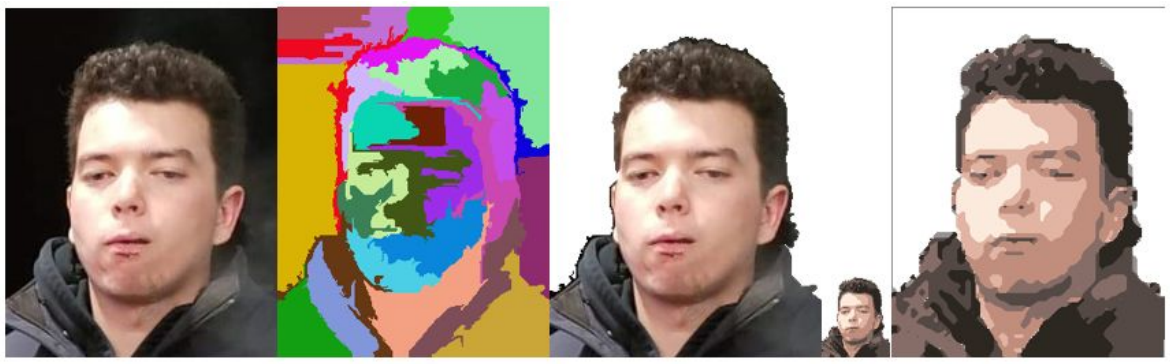
The final step in the algorithm is to render the image. To do this, a simple flood fill algorithm was used to fill the correct colours from the down-sized palette into the areas between the interpolated colour border sequences. This starts at positions where the colour is known and floods in the colours until it reaches a border. Finally, a background image can be placed behind the cartoon foreground by layering the background over the fully alpha pixels that were determined in the segmentation process. This results in the ability to place a newly cartoonized image onto any background scene.

4 - Results

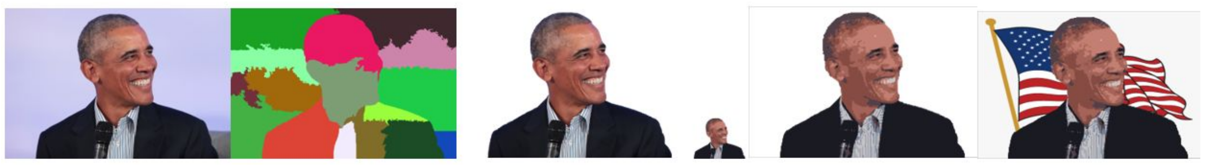
Figures 1, 2, 3, and 4 show the results at each stage of the application. The steps begin with the left-most image (the original) and moving right. In the second outputted image, the original is segmented and the user can select the foreground. The third image shows the selected foreground extracted from the original image. Fourth, in the small image, the image is pixelated and the image's colour palette is restricted to fewer colours. Finally the right-most image shows the output after the pixelization and de-pixelization steps. In Result 3, the optional step of adding a background image is shown in the fifth image. These results show the cases where this application works well. This application does not work so well in cases where the background is similar to the foreground, the background is very detailed, or the images are very large.



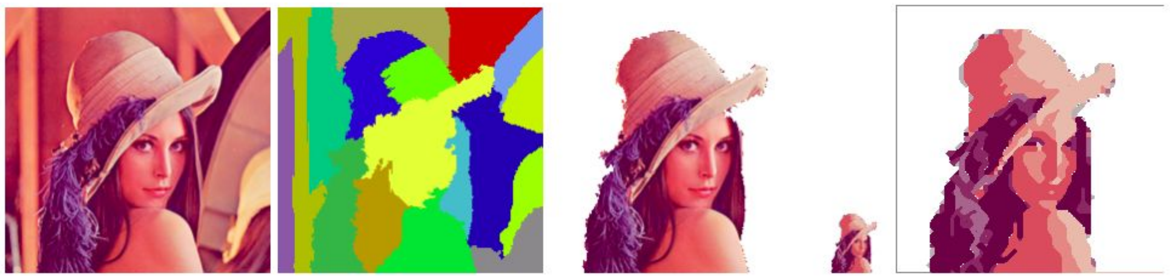
Result 1



Result 2



Result 3



Result 4

List of Work

Equal work was performed by all project members.

GitHub Page

Implementation was written in C++ with external sources cited where used. All code on github: https://github.com/BMelone/COMP4102-Project

References

[1] Felzenszwalb, P.F., Huttenlocher, D.P. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision* 59**,** 167–181 (2004).
http://cs.brown.edu/people/pfelzens/papers/seg-ijcv.pdf

[2] E. Cayllahua Cahuina, J. Cousty, et al. Efficient Algorithms for Hierarchical Graph-Based Segmentation Relying on the Felzenszwalb–Huttenlocher Dissimilarity.
Published April 9 2019:
https://www.worldscientific.com/doi/pdf/10.1142/S0218001419400081

[3] Johannes Kopf and Dani Lischinski. 2011. Depixelizing pixel art.
https://johanneskopf.de/publications/pixelart/paper/pixel.pdf