

Image Analogies – Final project CS 445

Overview

If we have a pair of images A and A' , where A is the original image and A' is the original image processed through a filter, we can use the pair, along with another image B , to compute a target image B' that is analogous to B in the same way that A' is analogous to A (i.e. $A:A' :: B:B'$). In other words, we can use a pair of images to learn the filter applied from A to A' and apply this same process to another image. This algorithm has various applications including texture synthesis and capturing image filters.

Algorithm

We implemented the algorithm using the paper “Image Analogies”, by Hertzmann et al. (<http://www.mrl.nyu.edu/publications/image-analogies/analogies-72dpi.pdf>). The paper contains two pieces of pseudo-code that we must implement to construct B' from A , A' and B .

```
function create_image_analogy( $A$ ,  $A'$ ,  $B$ ):  
    Compute Gaussian pyramids for  $A$ ,  $A'$ , and  $B$   
    Compute features for  $A$ ,  $A'$ , and  $B$   
    Initialize the search structures (e.g., for ANN)  
    for each level  $L$ , from coarsest to finest, do:  
        for each pixel  $q \in B'_L$ , in scan-line order, do:  
             $p \leftarrow \text{BESTMATCH}(A, A', B, B', s, L, q)$   
             $B'_L(q) \leftarrow A'_L(p)$   
             $s_L(q) \leftarrow p$   
    return  $B_{\text{prime}}$  (from finest level)
```

```
function best_match( $A$ ,  $A'$ ,  $B$ ,  $B'$ ,  $s$ ,  $l$ ,  $q$ ):  
     $p_{\text{app}} \leftarrow \text{best\_approximate\_match}(A, A', B, B', L, q)$   
     $p_{\text{coh}} \leftarrow \text{best\_coherence\_match}(A, A', B, B', s, L, q)$   
     $d_{\text{app}} \leftarrow ||F_L(p_{\text{app}}) - F_L(q)||^2$   
     $d_{\text{coh}} \leftarrow ||F_L(p_{\text{coh}}) - F_L(q)||^2$   
    if  $d_{\text{coh}} < d_{\text{app}}(1 + 2^{L-1}\epsilon)$  then  
        return  $p_{\text{coh}}$   
    else  
        return  $p_{\text{app}}$ 
```

Feature vector

Our first task is to generate Gaussian pyramids for all input images. These Gaussian pyramids are necessary to gain finer accuracy in generating the B' image because we first quickly generate B' for the smallest level of the Gaussian pyramid and use the smaller levels to help synthesize the larger levels. We then create feature vectors for all of the input images and their pyramids. Since humans are more

sensitive to changes in luminance, we use the luminance channel (Y in YIQ) as a feature for feature matching. Therefore, a feature vector of a pixel refers to the luminance value of the pixel. To convert RGB to YIQ (Luminance-Inphase-Quadrature), Matlab has a built-in function called “rgb2ntsc”. The transformation would have been relatively easy even if we were not allowed to use this function as seen from the equations below.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Equation 1

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.621 \\ 1 & -0.272 & -0.647 \\ 1 & -1.105 & 1.702 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

Equation 2

Data structure

Next we initialize our structures to store the synthesized gaussian pyramids and feature vectors at each level for B'. We also initialize a matrix, “S”, that stores the mapping of each pixel in the target pair (B and B') to the corresponding pixel in the source pair (A and A'). Therefore, after initialization, we have the following matrices calculated and stored - Gaussian pyramids for A, A' and B; Feature vectors for each pyramid of A, A' and B; and the matrix S.

Best match algorithm

Now we get onto the meat of the algorithm. A high-level explanation of what we do next is as follows: for each level, starting from the coarsest to finest, we fill in B' for that level by finding the coordinates from image A that best matches the pixels in B, and then copying the corresponding pixel with those same coordinates from A' into B'. We keep updating the matrix S as we keep finding best matches for a pixel in B'. Here is the algorithm for finding the best matched pixel -

function best_match(A, A', B, B', s, l, q):

```

    papp ← best_approximate_match(A, A', B, B', L, q)
    pcoh ← best_coherence_match(A, A', B, B', s, L, q)
    dapp ← ||FL(papp) - FL(q)||2
    dcoh ← ||FL(pcoh) - FL(q)||2
    if dcoh < dapp(1 + 2L - Lmax) then
        return pcoh
    else
        return papp

```

We pass in the feature vectors for the current level of A , A' , B , B' (which is partially empty because it's being synthesized), the s matrix for the current level, the current level l , and q which is the coordinates of the current pixel being synthesized in B' . In our implementation, we also additionally had to pass in the feature vectors for the previous level ($l-1$) of A , A' , B and B' . We use two methods to find the best matched pixel - Approximate-Nearest-Neighbor (ANN) algorithm and a coherence formula.

For ANN, we used the Matlab library

(https://github.com/jefferislab/MatlabSupport/tree/master/ann_wrapper). The library uses kd-trees and we found the nearest neighbor using the luminance values of the 5x5 neighborhood (3x3 neighborhood in the smaller, coarser level) around the pixel q in image B .

For coherence matching, we find a pixel r^* in the neighbourhood of a pixel q in the partially synthesized image B' , such that when mapped to the corresponding pixel in A , minimizes the feature difference. We then return $S(r^*) + (q - r^*)$. This formula the best pixel that is coherent with the portion of B' that has already been synthesized. We need to consider coherence so that we can favor pixels that are close together from the source image. In other words, a smoother result is obtained by picking pixels that are near each other rather than pixels from all over the image.

As talked in the paper, $F_L(p)$ denotes the concatenation of all feature vectors within a neighborhood of pixels around pixel p from images A and A' for both the current level and the previous, smaller level. $F_L(q)$ denotes the same concatenation but around pixel q from images B and B' . Note that B' is only partially synthesized. So, to calculate both distances d_{app} and d_{coh} , we must take the norm of the difference between the two concatenations of feature vectors for both images using both candidates p_{app} and p_{coh} . For both of these differences, we compute them as a weighted Gaussian distance so that the differences in feature vectors further from p and q have smaller weight. We do this by applying a Gaussian filter and taking the norm of the difference of feature vectors.

$$d_{coh} < d_{app}(1 + 2^{L-L_{max}} * Kappa)$$

With these distances, we then plug the distance into the above formula, where L is the current level and L_{max} is the highest level. $Kappa$ is a positive, real number chosen by us where we can choose to decrease the influence of ANN matching on choosing which pixel is our best match. As stated before, we then return our best match's pixel coordinates and copy that into our target image B' for the current level at the current pixel q .

Difficulties

A lot of difficulties that we had were from taking the 5x5 and 3x3 neighborhoods of the feature vectors. Our approach pads the feature vectors by adding 2 pixels to the width and height with 0s, so that when we are taking the neighborhood of a border pixel, we don't try to access pixels with index 0 or -1. Another difficulty we had was that color was distorted for some cases. This error was because we copied over the color from A' when we copied the pixel from A' to B' . Some cases, such as the rabbit and the nature picture must preserve the colors of B . Other cases, such as the texture synthesis and city generation needed to preserve the colors of A' . Since the algorithm takes so long to run, fixing the small errors also took a lot of time. Finally, unsurprisingly, there were a lot of index out of bounds & off-by-one errors from random index typos, mistakes, etc. The overall result looks neat though.

Results

All of the results were created using 4 levels of Gaussian pyramids.

Freud

Original set (A and A')



Figure 1 (A)



Figure 2 (A')

Output set (B and B') (Kappa = 1)

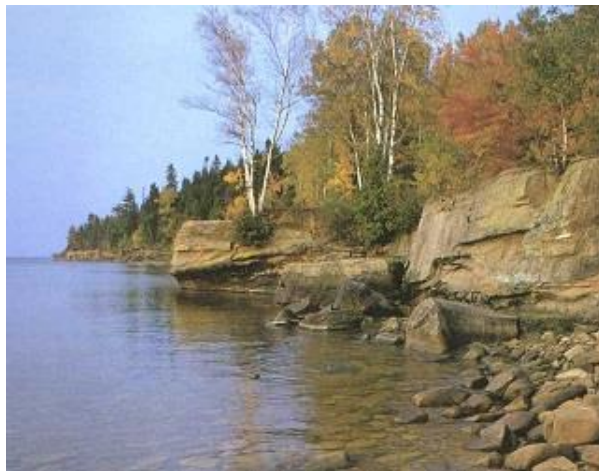


Figure 8 (B)

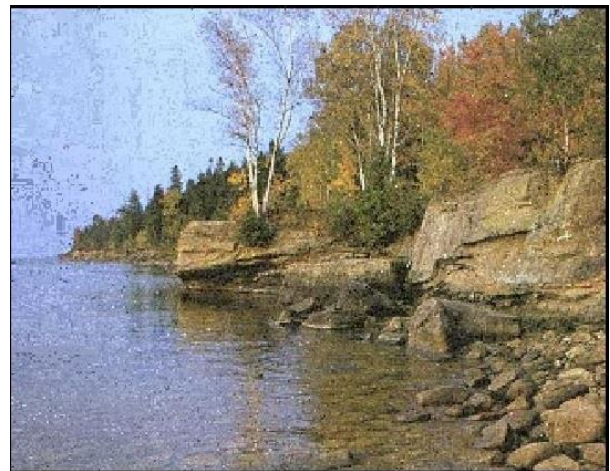

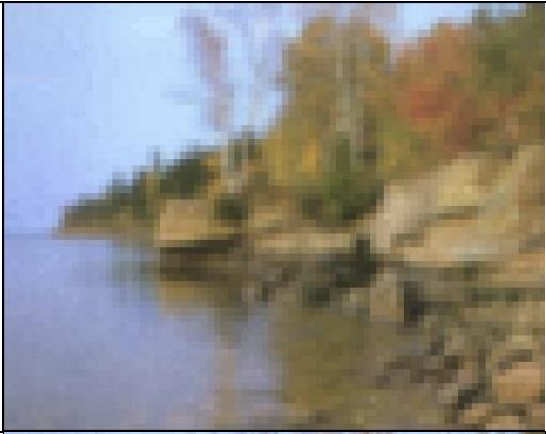
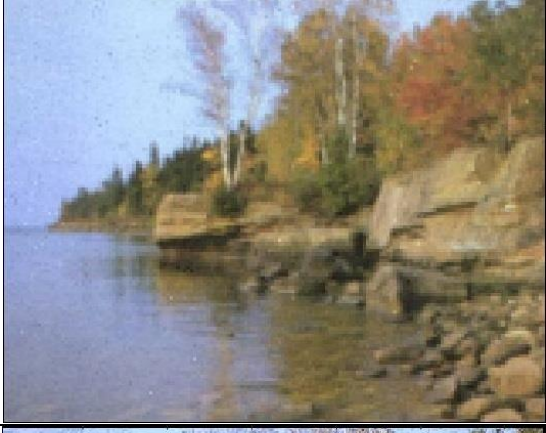
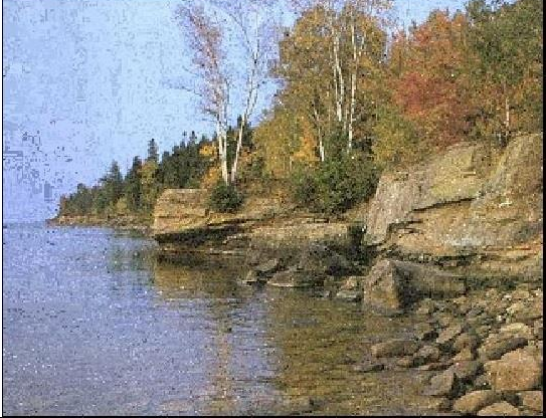


Figure 9 (B')

Gaussian Pyramids

The Gaussian pyramids were created as follows for B'. The pyramids allow for accuracy in the synthesis.

Level 1 (Coarse)			
Level 2			
Level 3			
Level 4 (Fine)			

Rabbit

Original set (A and A')



Figure 3 (A)



Figure 4 (A')

Output set (B and B') (Kappa = 1)



Figure 5 (B')



Figure 6 (B)

We stumbled upon an interesting fact here. Initially, instead of just transferring the luminance from A' to B', we had mistakenly copied the other channels from A' to B' too. As a result, we got a blue-ish image, as shown in Figure 5 (below).

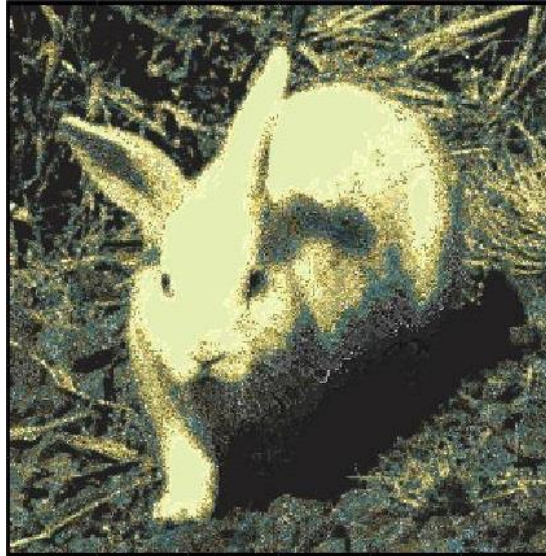
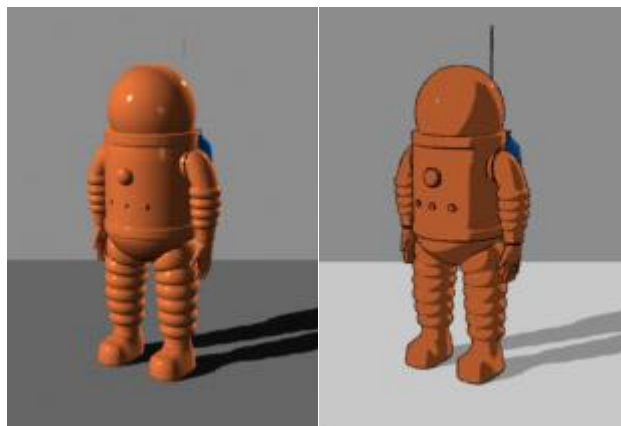
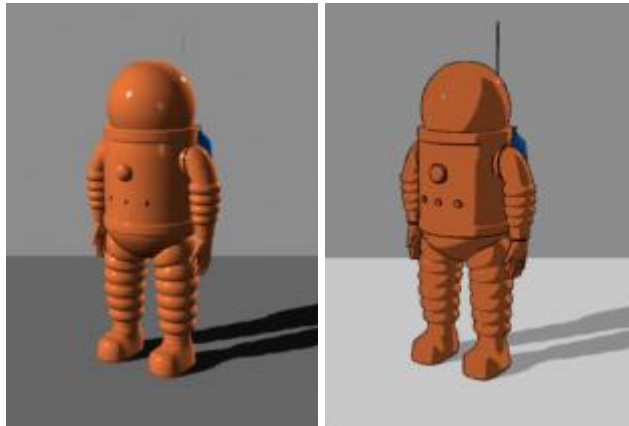


Figure 7

Toon (Cartoon effect) ($\kappa=20$) - Good



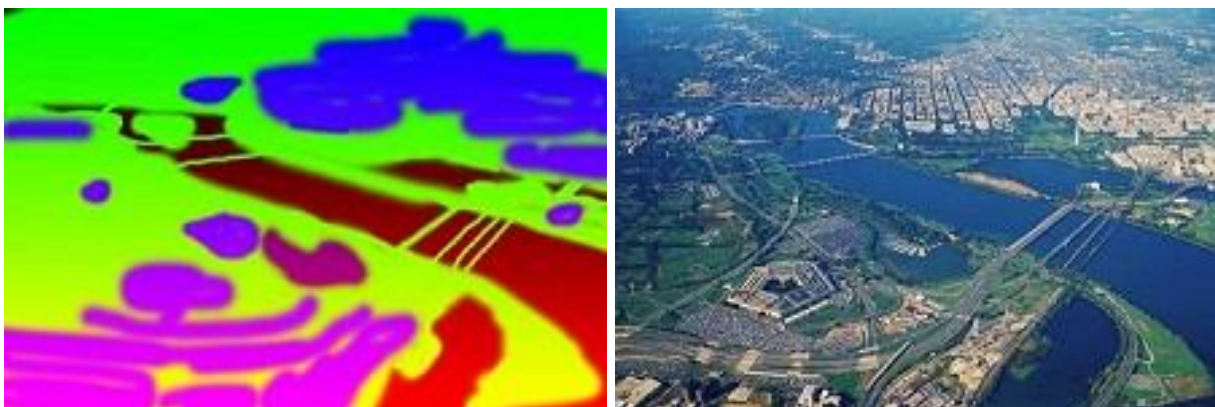
Toon 2 (Cartoon effect) ($\text{Kappa}=20$) – Not so good

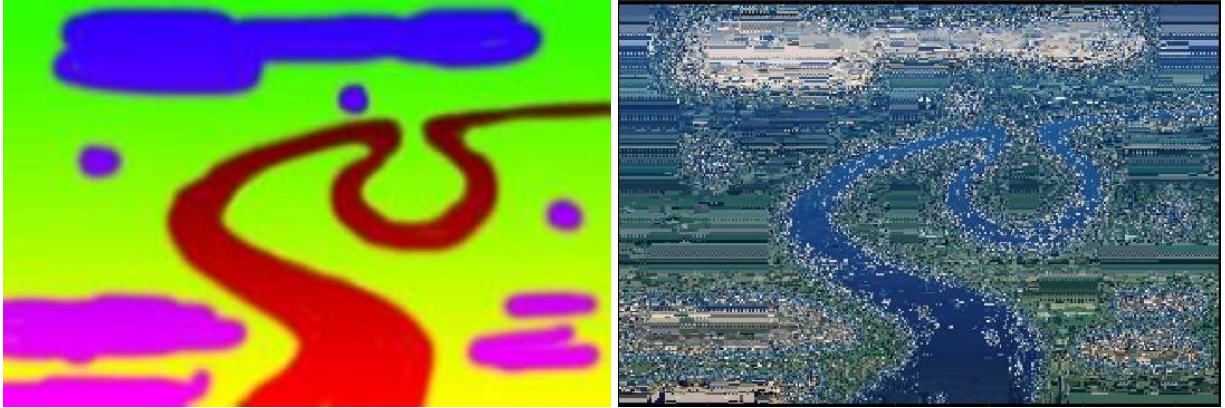


The artefacts occur due to color dissimilarity between A and B. The algorithm fails to find appropriate matches in A, for a pixel in B.

Failure

The following set of images resulted in a failure image as can be seen below.





The failure was mainly because of both poor approximate match and poor coherent results. There are changes in color in B, that cannot be detected clearly from A (and hence A'). Also, there are a lot of matching points for a color in B to A, each of which can map to an entirely different area in A'; hence the unclear image.