# Content Aware Image Resizing
# 18-551 Fall 2007
# Group 6

**Ziv Wolkowicki** (zwolkowi@andrew)
**Jim He** (jjhe@andrew)
**Manuel Gonzalez-Rivero** (mgonzale@andrew)

## Introduction

There are many times when resizing an image is necessary. In such a media driven age, images speak louder than words, and portraying that image as effectively as possible is a high priority. Resizing is important when trying to fit several images to a web page, or an academic document such as this one. When trying to preserve the aspect ratio of an image, cropping is often used – however, what should be done if the desired content of an image will not fit into a very specific frame?

Modern resizing techniques work effectively when preserving the aspect ratio of an image. For example, to resize a 640 by 480 image down to a 320 by 240 image, sets of 2 by 2 pixels are grouped and averaged out. Figure 1 is an example of this technique. The output image appears acceptable to the eye because people often identify objects through their height to width ratios as well as the distances between them. Important characteristics of an image can be kept intact by scaling down to a smaller image while still preserving the same aspect ratio.

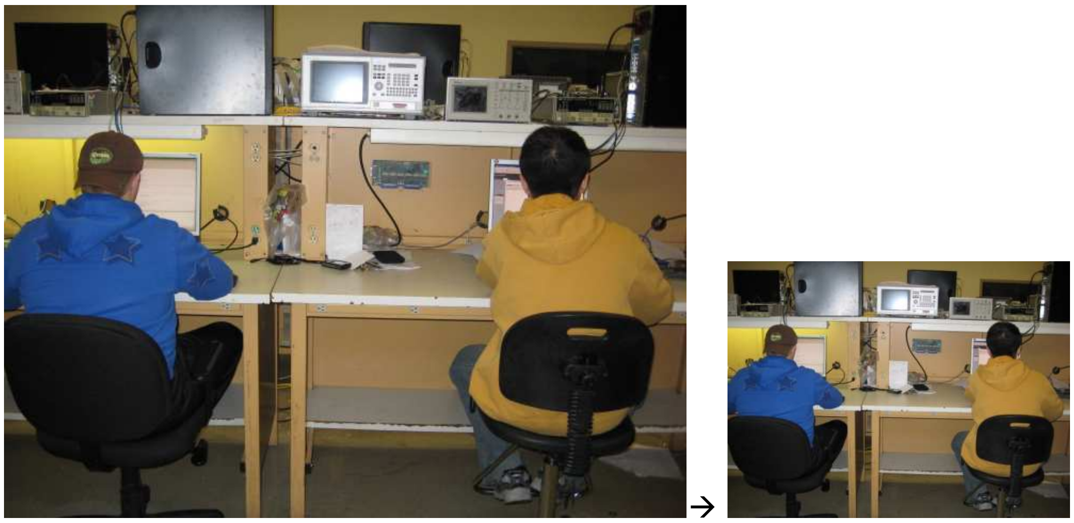


**Figure 1: Resampling (Scaling)**

We can also make our 640 by 480 image fit on a 320 by 480 display, as is commonly seen on modern cell phones. Such cropping could be used to leave only the important part of the image. In Figure 2 it is clear that both the aspect ratio and the distances between important objects are maintained, assuming that the person on the right is the important content. The problem with cropping arises when our 320 by 480 frame is unable to capture all the important objects in the image. As can be seen in Figure 3, cropping no longer suffices if we want to keep both people in the frame of the image.

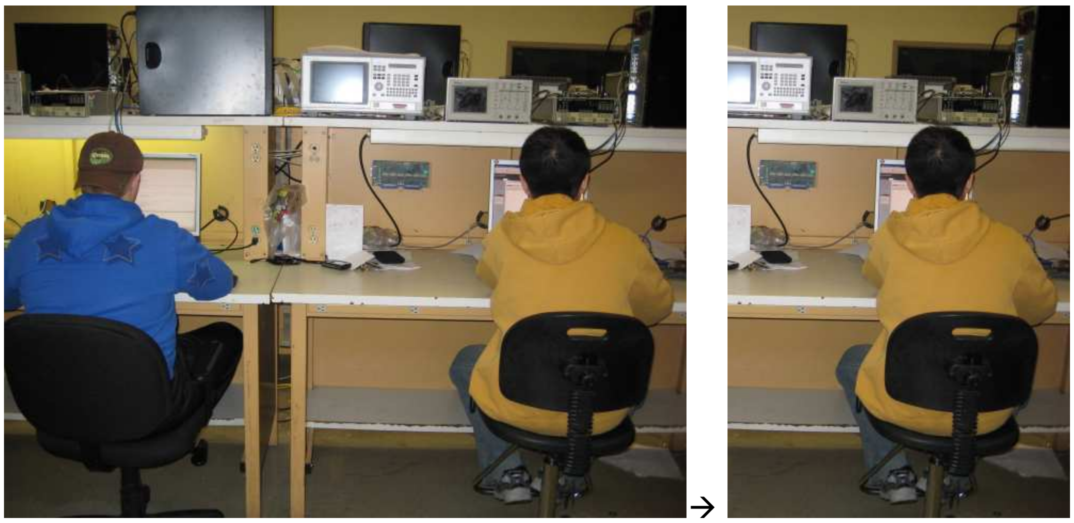


Figure 2: Cropping

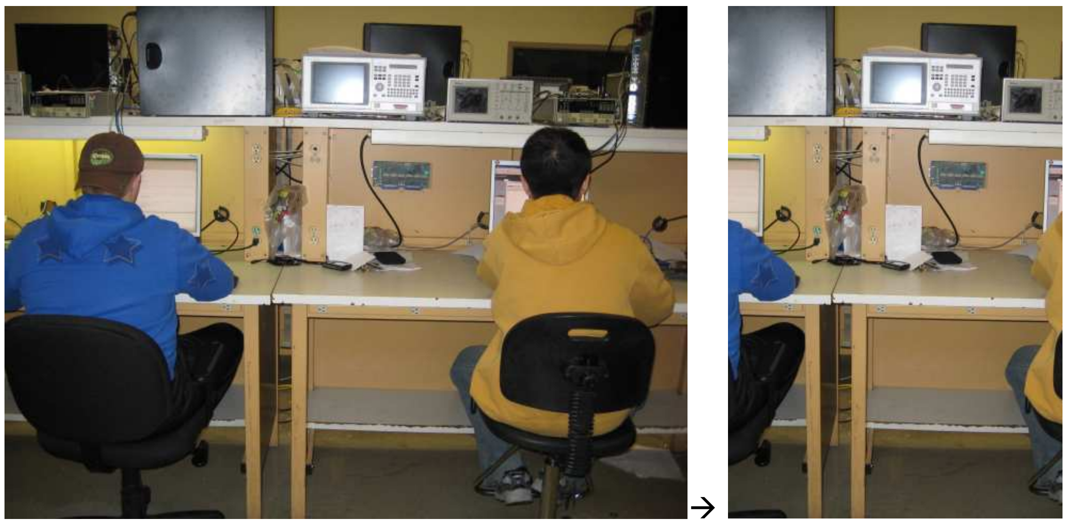


Figure 3: Cropping Gone Wrong

In an attempt to fix the issue, we may try resorting to resampling techniques as described above. For instance, we may take blocks of 2 by 1 and averaging those out into one pixel. However, this method also leads to a distorted image, as seen in Figure 4. The changed aspect ratio has caused many important features of the image to become far too narrow. Additionally, the distance between the two bodies has diminished to an unsatisfactory level. This approach to resizing is also unacceptable.

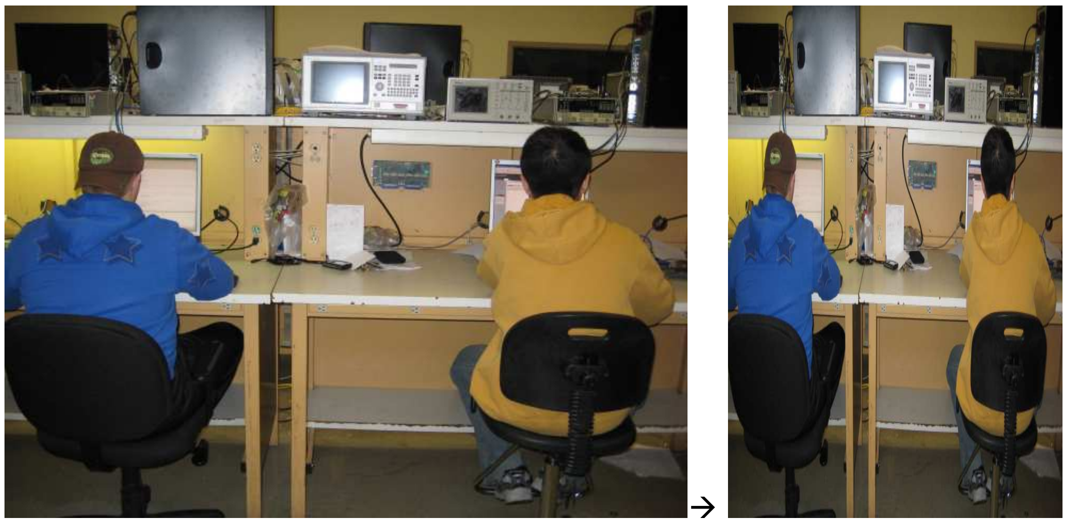


Figure 4: Resampling Gone Wrong

In order to address problems associated with cropping and scaling, Shai Avidan and Ariel Shamir suggested an approach they named *content aware image resizing* at SIGGRAPH 2007[1].The aim of their algorithm is to remove the least noticeable seams of the image. A seam can be horizontal or vertical, and is defined as a connected path of pixels, which contains no more than one pixel per column or no more than one pixel per row. Shamir and Avidan quantify *noticeability* as the sum of the energies associated with the pixels in the seam. An image can be resized to a new aspect ratio while leaving its important content untouched by removing a combination of vertical and horizontal seams. Figure 5 shows the application of this technique to an image of several islands. The islands contain relatively high energy, and their aspect ratios are preserved. The ocean, on the other hand, is associated with relatively low energy due to the little change in color between adjacent pixels. As a result, most of the seams removed are from within the water. Fortunately, the low energy content is less noticeable, so it is difficult to see how the seam removal affected the water.

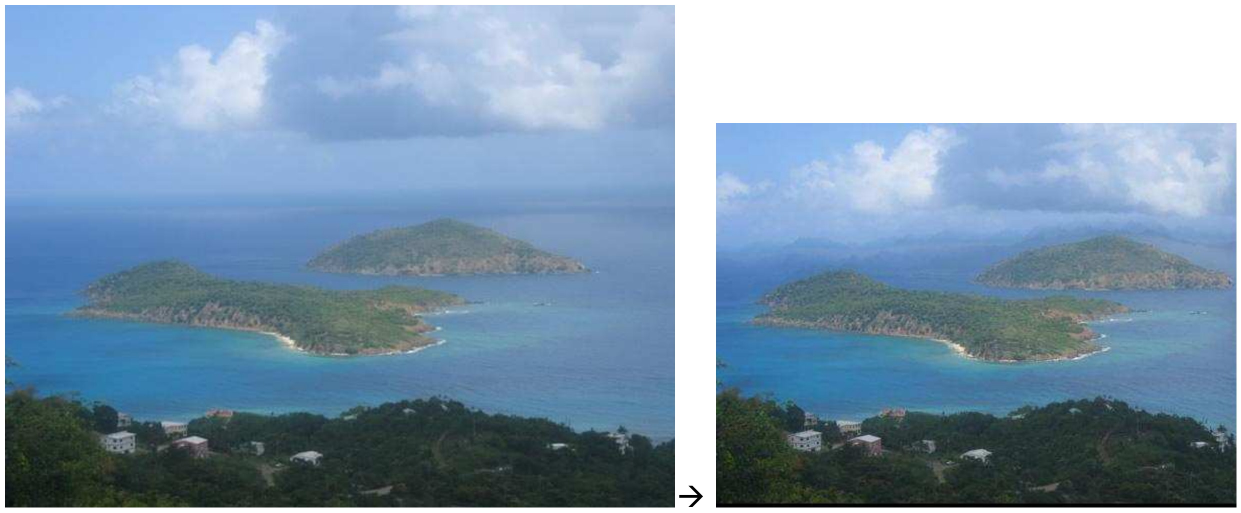


Figure 5: Content Aware Image Resizing

[1] (Avidan & Shamir, 2007)

This project explorers and modifies many of the ideas behind the algorithm described in the Shamir and Avidan paper. Although there were several aspects of the algorithm that we were not able to replicate, we were still able to match their results and the results of an online implementation[2] closely. The majority of the signal processing related computation takes place on a TI C67 DSK, a DSP microprocessor embedded on a development board. The highly specific optimization of the DSP makes it optimal for such an arrangement.

One of the major problems with the Shamir/Avidan algorithm has been its poor performance with faces. Our implementation addresses this issue in order to increase the desirability of this research. Through implementation of this algorithm, our work has yielded several new parameters that have a significant effect on the aesthetic appeal of the output. Although the scope of our research has not covered all of these issues, we will be discussing those of which it has. Additional suggestions will also be put forth for solutions to issues we were unable to explore at a sufficient depth.

**Previous 18-551 Work and Novelty**

Due to the novelty of this algorithm, nothing like this has ever been attempted in prior 18-551 projects. The algorithm invented in August of 2007, and as such we were the first semester to have been exposed to it. As a result, we had to rely on outside research to guide us through our development stages. Fortunately, we were able to get in contact with Shamir and Avidan personally to receive some assistance in our implementation. In addition, online services such as rsizr.com have already completed their own implementations of the algorithm. However, the implementations that we have seen did not combine face detection into their package, nor did they look into different edge detectors. Because our research looks into these two major factors, we believe that we have taken steps make this algorithm even more desirable to commercial and academic purposes. In terms of face detection, we closely follow the work of Group 2 in Spring 2005. This will be discussed in greater detail later on in the paper.

## **THE ALGORITHM**

As mentioned earlier, the technique we implemented attempts to produce visually appealing images that are resized to fit other aspect ratios. The algorithm identifies areas of importance and preserves those areas. Areas of importance are simply the areas of the image that catch the eyes first – perhaps an island floating in the ocean, or an individual with a landscape behind her. Such content is normally associated with being of higher energy, meaning that there is a lot of change in color or light intensity in those areas. For example, in Figure 5 the water has a blue gradient very little change. As a result, there should be strong *edges* between the water and the islands, and also within the trees and dirt on the island itself. Unfortunately, areas that are of importance to us are not always characteristic of this. For example, faces are often associated with less energy than a tree (as in Figure 15), except that a viewer would far prefer to maintain the face and cut out the tree if given a choice. We address this issue by

[2] (rsizr.com - intelligent image resizing, 2007)

detecting the faces and protecting them differently, as will be discussed later in this paper. Several steps are involved in protecting the areas of greater energy; we begin by detecting the edges.

## Edge Detection

The first step involved in resizing an image is to detect areas of high energy by means of edge detection. In order to do this, we simply convolve an image with a kernel that detects changes in color or luminosity between adjacent pixels. In an attempt to simplify the problem, the color (RGB) image is first converted into a grayscale image to detect differences in luminosity only. This process decreases the size of our input data, in turn decreasing the transfer and computation times. Afterwards, we convolve the grayscale image with a kernel that will enhance edges. Since edges occur where luminosity changes, the horizontal edges are detected by subtracting the value of every pixel with the pixel to its immediate right. The vertical edges can be computed in a similar fashion, except we subtract the value of the pixel below the one being considered. Summing up the vertical and horizontal edges creates a map of all our edges, referred to as an *edge map*. Regions that have a lot of change in luminosity will have larger areas of high values to represent the strong edges. In terms of 2D signal processing, we are convolving the input image with the kernel seen in Figure 6 to produce the edge map. Figure 7 displays an example of this computation on a smaller scale.

| | |
|---|---|
| **2** | **-1** |
| **-1** | **0** |

**Figure 6: Edge Kernel**

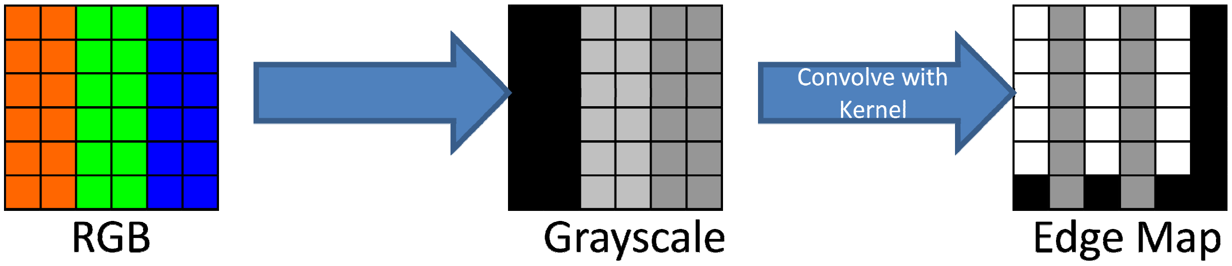


**Figure 7: Edge Detection**

There are several edge detection algorithms that involve a combination of filtering noise, edge enhancement and thresholding. However, these algorithms are intended to find absolute, well defined edges. They create a binary map depicting the absolute presence of edges. Due to their binary nature, these edge detectors are not suitable for content aware resizing. For our purposes, it would be preferable to use a range of values works because it allows the edge map to contain a range of edge strength. As a result, the weaker edges are more likely to be removed than the stronger edges. Thus, the additional features provided algorithms like Canny Edge Detection make no contribution to our resizing algorithm and may perhaps be detrimental. Using our simple convolution method to create the edge map, we gain much needed flexibility and computational efficiency.

The edge kernel seen in Figure 6 proves to be effective in finding edges, but may not necessarily be ideal for our application. Images in which there are lots of trees (Figure 15) have an overwhelming amount of energy associated with them such that it overpowers the amount of energy associated with the parts of the image we would like to protect. This issue is addressed by smoothing out the image first and then applying the edge detection kernel, so areas like thick forest will no longer contain so much energy. Due to the associative and commutative properties of convolution, the smoothing and edge detection kernels can be combined into one large kernel. This would allow for greater computational efficiency and less memory necessary to complete the process. To further improve computation time, we round and scale each value in the kernel so that simple fixed point math can be used in our calculations. The resulting kernel seen in Figure 8 was used to produce all further results. Of course, there are many other kernels that may perform better for specific types of images, but the kernel in Figure 8 proved to yield the most desirable results overall. Later on in this paper we will provide a comparison of some different kernels and discuss potential further research potential in this area.

| | | | |
|---|---|---|---|
| 6 | 5 | -5 | -6 |
| 5 | 4 | -4 | -5 |
| -5 | -4 | 4 | 5 |
| -6 | -5 | 5 | 6 |

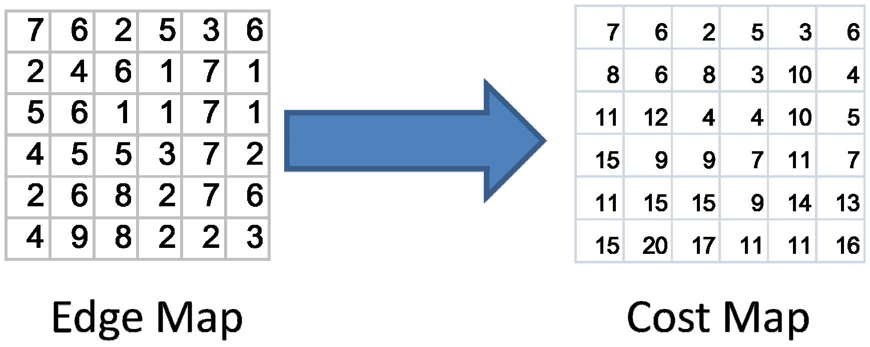
Figure 8: Smoothing Edge Kernel

Now that we have decided on which areas of the image are of **higher importance**, we can begin the process of finding which seams to remove. To accomplish this, we apply a fairly simple dynamic programming algorithm to determine which seam contains the lowest energy.

**Seam Removal**

In order to detect which seam has the lowest energy associated with it, we perform a pixel-by-pixel calculation that is similar to our edge finding technique. However, this process cannot occur in the form of a convolution, making it somewhat more computationally intensive for a DSP processor. Nevertheless, we will demonstrate calculations that show this is not a significant detriment on runtime.

In this portion of the algorithm, the first row of pixels of the input edge map populates the first row of pixels in our output *cost map*. For every pixel in the succeeding rows, we take the minimum of the three above adjacent pixels (top left, directly above, and top right) and add it to the value of the pixel in the original image as can be seen Figure 9. This process is continued for every row until the cost map is completed.



Edge Map

| | | | | | |
|---|---|---|---|---|---|
| 7 | 6 | 2 | 5 | 3 | 6 |
| 2 | 4 | 6 | 1 | 7 | 1 |
| 5 | 6 | 1 | 1 | 7 | 1 |
| 4 | 5 | 5 | 3 | 7 | 2 |
| 2 | 6 | 8 | 2 | 7 | 6 |
| 4 | 9 | 8 | 2 | 2 | 3 |

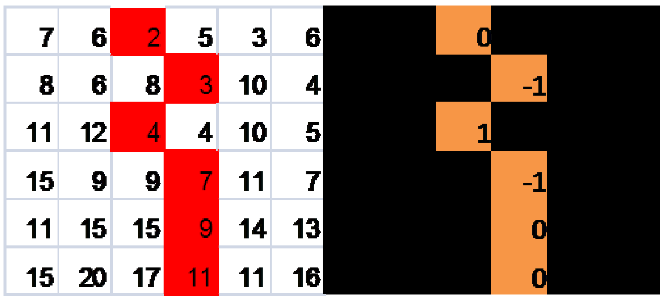
Cost Map

| | | | | | |
|---|---|---|---|---|---|
| 7 | 6 | 2 | 5 | 3 | 6 |
| 8 | 6 | 8 | 3 | 10 | 4 |
| 11 | 12 | 4 | 4 | 10 | 5 |
| 15 | 9 | 9 | 7 | 11 | 7 |
| 11 | 15 | 15 | 9 | 14 | 13 |
| 15 | 20 | 17 | 11 | 11 | 16 |

Figure 9: Cost Map

Each pixel in bottom row of the cost map represents the energy associated with the seam that would be removed if the seam began at that pixel. To achieve our stated goal of removing the least important seam, we want to start at the lowest pixel in the bottom row. Once we find the lowest bottom pixel, we trace back towards the top of the image. In doing so, we once again look at the three above adjacent pixels, and we continue on to the lowest of the three. Each pixel we touch on the way towards the top is a pixel within our seam. Once we finish tracing through a whole seam, we can remove it in the corresponding color image (note that we have been working with the grayscale image up until now). We have then successfully resized our image by one unit in width as defined by the algorithm.

Note that under the above implementation, the minimum value of pixels is calculated twice unnecessarily. To save some computation, it is possible to build the cost map using values of either –1, 0, or +1 to mark the above pixel that we came from. By doing so, we can trace back much more quickly since we no longer need to calculate the minimum value a second time around. This method is demonstrated in Figure 10. Since caching works based upon pre-fetching and spatial locality, calculating the minimum pixel we came from above will cause memory latency on any caching processor (since it is in a stride perpendicular to our cache direction). However, building the cost map on the way down will not be any less efficient in this direction since we are applying the dynamic programming portion of the algorithm in the same stride direction.



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 2 | 5 | 3 | 6 | | | 0 | | | |
| 8 | 6 | 8 | 3 | 10 | 4 | | | | -1 | | |
| 11 | 12 | 4 | 4 | 10 | 5 | | | 1 | | | |
| 15 | 9 | 9 | 7 | 11 | 7 | | | | -1 | | |
| 11 | 15 | 15 | 9 | 14 | 13 | | | | 0 | | |
| 15 | 20 | 17 | 11 | 11 | 16 | | | | 0 | | |

Figure 10: Trinary Cost Map

It is important to note that we have only found the optimal seam to remove in the horizontal dimension (it is a vertical seam). We can simply apply the same algorithm by building a cost map going from left to right as opposed to top-down. In the interest of memory access latency on our processor and overall code size, we actually remove seams only in the top-down direction. To take care of horizontal seams, we input the transpose of the image into our system.

This is the final step in one iteration of the core algorithm. Afterwards we ideally want to input the new one-seam-removed image into our system again and loop until we remove as many seams as we originally wanted to. To produce the optimal outputs we must calculate the cost map in both the vertical and horizontal dimension (assuming we want to remove seams from both dimensions) and remove the lowest seam out of a set of all seams in both dimensions, until we complete resizing in both dimensions. Through personal contact, Avidan and Shamir suggested directly to us to recalculate the cost map and edge map seam we remove at every iteration. However, such an implementation would have significant implications on the runtime of this algorithm, especially our specific DSK TCP/IP setup. Because of this, we deploy a collection of approximation techniques that produce very similar outputs when compared to the optimal implementation, allowing for a speedup factor greater than 2. We will discuss these approximations, their runtime benefits, and their visual pitfalls.

## DSK and PC Flow

As can be expected, the algorithm requires a significant amount of computation per seam removed. In addition, the target processor is a DSK which implies that there will be transfer back and forth between it and a PC. We must deal with multiple issues, such as the transfers necessary to get data to and from the processor as well as memory latency to ensure timely computation of our algorithm.

Figure 11 shows the interaction between the PC and the DSK and which portions of the algorithm were allocated to which computer. Due to the DSP's poor compiler optimization and small memory space, we had to place specific portions of the algorithm on the PC in order to avoid poor computation time.

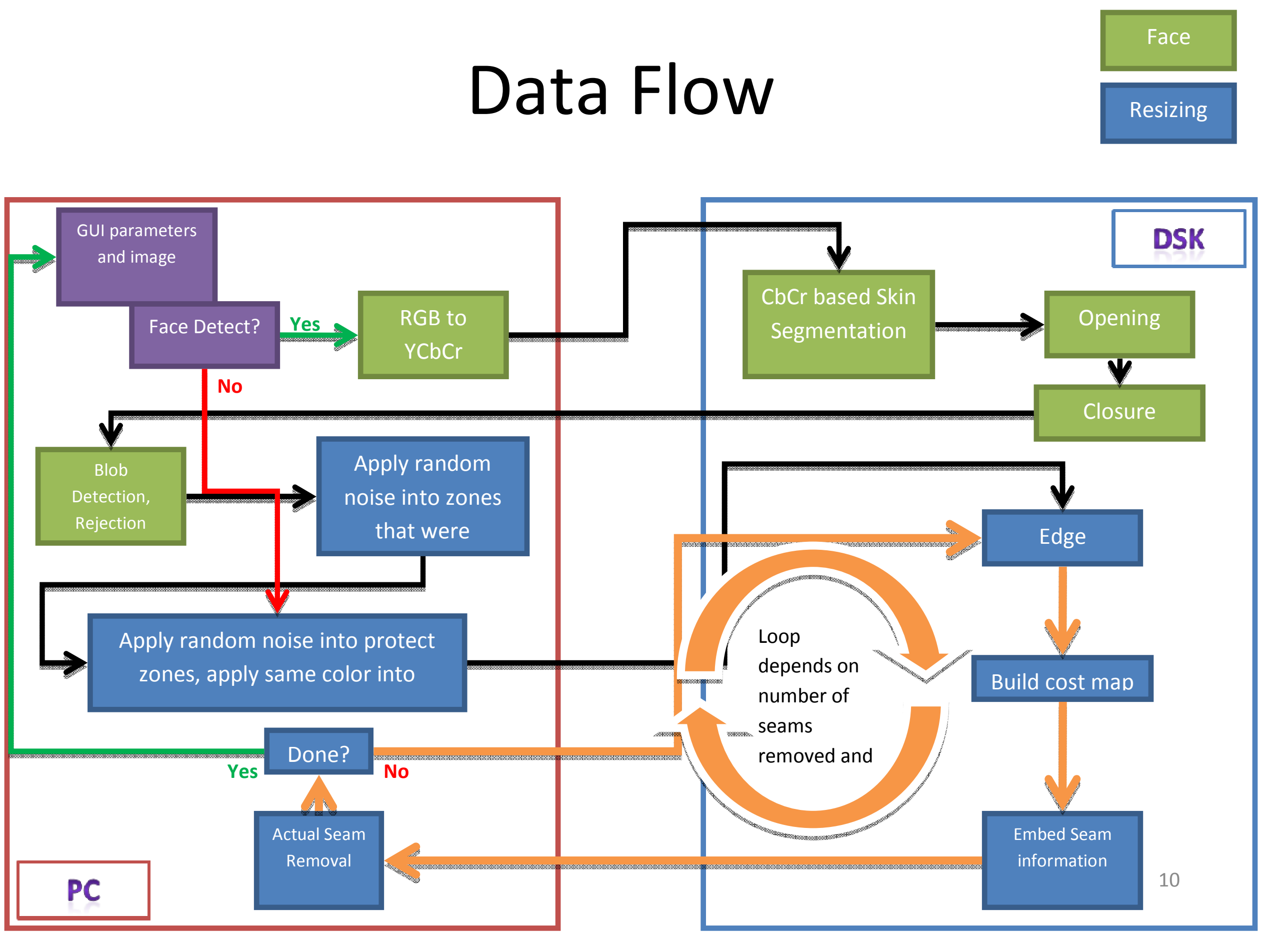

Data Flow
Face
Resizing
DSK
PC
GUI parameters and image
Face Detect?
Yes
No
RGB to YCbCr
CbCr based Skin Segmentation
Opening
Closure
Blob Detection, Rejection
Apply random noise into zones that were
Apply random noise into protect zones, apply same color into
Done?
Yes
No
Actual Seam Removal
Edge
Build cost map
Embed Seam information
Loop depends on number of seams removed and
10

Figure 11

Before seams can be removed from an image, we first preprocess with face detection. We then begin to execute our core algorithm. In order to minimize the amount of data transferred to the DSK and the amount of data the DSK needs to process, we convert our 1 byte/pixel * 3 channel (RGB) image into a 1 byte/pixel image on the PC side. Following this, the data is sent to the DSK to be processed over a standard TCP/IP connection.

We perform edge detection by applying the integer kernel as mentioned earlier (Figure 8). Afterwards we build the cost map per criteria set by the dynamic programming we explained. As mentioned, we only calculate the cost map from the top down in the interest of code size, code complexity, and cache access times. In unison with building the cost map we also embed the seam information, and send this output back to the PC by means of the same TCP/IP connection.

The DSP processor is well suited for the types of computations we wish to perform. Edge detection in itself commonly uses multiplications and additions while cost map building is characterized by comparisons and additions – all of which are high throughput operations. Additionally, matrix operands are frequently free of dependencies. As such, it works well that the DSP possesses 8 functional units that give it capability of parallelizing operations free of dependencies. As a result, the calculations we which to perform are ideal for the DSP environment.

On the other hand, purely memory-access based processes (requiring little to no actual signal processing), are completed on the PC. Once the PC receives the seam information from the DSK, it can simply copy all pixels from the color image to a new color image, excluding the pixels to be removed. Since our DSK returns an output that does not contain actual image information (only seam removal information) it is important to keep a temporary color image on the PC that will eventually be our final result. This operation would require a large amount of additional time and memory on the DSK to complete, justifying its location on the PC/DSK allocation.

Since the PC holds the color image with a seam removed, it reconverts this image into grayscale, sends it to the DSK, and we start all over again. This process is looped until we reach our desired width. We then transpose our temporary color image on the PC (once again a memory-access only operation) and loop the same process until we reach the desired height. Finally, we transpose our color image one last time to the correct orientation, yielding our end result.

## Approximations

As mentioned earlier, it would be ideal to loop this processor one seam at a time to produce the most visually appealing output. Unfortunately, given our transfer rates and computation times (to be analyzed later in this paper), the overall execution time associated with doing so is undesirable. Calculating a horizontal and vertical cost map per every seam removed would be even more computationally expensive.

Due to the computational costs, we introduce a collection of approximations, the first of which is already apparent. The SIGGRAPH paper had discussed that it would be optimal to remove the lowest cost seam in either vertical or horizontal dimension[3], but we can speed up computational performance by at least a factor of 2 if we do one at a time. It turns out that by first calculating and removing all the vertical seams and then the horizontal seams, we can produce visually identical results when compared to removing the seams in the optimal fashion suggested by the paper. Given little to no visual artifacts as a result of this approximation, it is clearly desirable to implement such an approximation.

The most significant speedup is introduced by how often we calculate the cost map, quantified by seam removals per recalculation. All of the demonstrated results at the end of the paper (unless otherwise noted) have been produced by removing 5 seams/recalculation. Since the quality of the output decreases the more we approximate, we give the user an option of choosing either 1, 5, 10, or 15 seams to be removed per recalculation, as depicted in the GUI options.

3 (Avidan & Shamir, 2007)

## PERFORMANCE

There is clearly an array of applications for the algorithm, but to confirm its feasibility of being used in media and by home users we must quantify its performance. Since the target machine of this implementation is an embedded DSP processor, industry grade development tools were available to determine the computation time and transfer rates involved in the algorithm.

As mentioned earlier, data transfers occur over a standard Ethernet connection using TCP/IP. The documentation for this particular board, the TI C67 DSK, states that it can receive data at a rate of approximately 3MB/s and send at about 10MB/s. The difference in these speeds is due to the hardware architecture and the way the development kit interfaces with its Ethernet daughterboard. These data transfer rates have been replicated. However, our algorithm only requires sending and receiving an amount smaller than one megabyte (assuming an input image of 640 by 480). Given these short spurts of data transfer, the board is unable to reach its full potential since transfer rates of the first several kilobytes of data tend to be slower on average. The measurements we yield from our data transfer benchmarks are as follows:

| | |
|---|---|
| PC ➔ DSK | 1.161953 MB/s |
| DSK ➔ PC | 3.085707 MB/s |

Fortunately, the amount of bandwidth required for these applications is not overwhelming. However, a performance issue associated with the TCP/IP transfers does arise. Nevertheless, problem is not due to the transfer itself. Given the scope of our project and the architecture of our system, it is required the TCP/IP socket closes connection and opens again on both the DSK and PC side between iterations of cost map calculation. The time required to complete this action and for the GUI to execute the PC side communications is minimal individually. However, these computations accumulate over several iterations into a significant amount of latency.

Another performance measurement analyzed carefully in our implementation is cycle count of the common case. The common case is the edge detection and cost map building, which applies a certain set of calculations to every pixel in the image. Obviously, these calculations make up almost the entirety of the time spent processing on the DSP processor. Therefore, it is imperative that the calculations and inputs are optimized to ensure desirable runtimes.

A significant portion of execution time in any processor can be attributed to memory accesses. Modern day processors are faster by orders of magnitude than the memory that serves them. In order to address this, many processors – including the TI C67 – have a cache that stores memory accessed recently in effort to cut down access times. We have optimized our algorithm to exploit this to the fullest extent. The application of our kernel is performed in the same stride direction as the cache. This allows increasing memory addresses to lay in the same row which is optimal when using a cache. The cost map is also populated in the same memory stride direction as this for the same reason. Since the cost map must be built in the dimension perpendicular to which the seam is being carved, we limit the processor to only finding vertical seams. This justifies the need to transpose the input image before it is transferred to the DSK to remove horizontal seams, as described earlier. In addition, paging has been made use of to ensure memory accesses remain within internal memory with the exception of the EDMA copy itself. In order to reduce memory accesses, the coefficients involved in applying our kernel is hard-coded into the algorithm.

Below is a diagram that shows the array of operations done by the processor in order to execute our algorithm. Predictions have been made as to how many cycles per pixel would be necessary as seen below:

EDMA: $3 bytes * \ 0.5625 \frac{cycles}{byte} = \ 1.6875 \ cycles$

Memory Access = 144.6 cycles

- Input: 5 $accesses \ * \frac{1.23 cycles}{byte} * \ (16 \ + \ 3 \ + \ 1) \ bytes$
- Output: 16 $bytes * \ 1.35 \frac{cycles}{byte}$

Indexing = 114 cycles

- $4 \frac{additions}{pixel} \ * \ (16 \ + \ 3) pixels \ * \frac{1 cycle}{addition}$
- $2 \frac{multiplications}{pixel} \ * \ (16 \ + \ 3) pixels \ * \frac{1 cycle}{multiplication}$

Control Flow = 3 cycles

- $3 \frac{subtractions}{additions} * \frac{1 cycle}{\frac{sub}{add}}$

Dynamic Programming = 16 cycles

- $(7 \ ands \ + \ 3 \ ors \ + \ 2 \ compares \ + \ 2 \ abs() + \ 2 \ shifts) * \frac{1 cycle}{op}$

**Total predicted cycle count = 280 cycles/pixel**

Profiling Results:

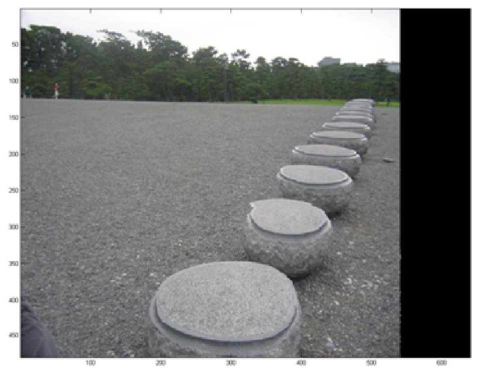
Code Size: 5856 Bytes

$$\frac{36{,}202{,}894 \ cycles}{370x278 \ pixels} == \ 351.96 \frac{\text{cycles}}{\text{pixel}}$$

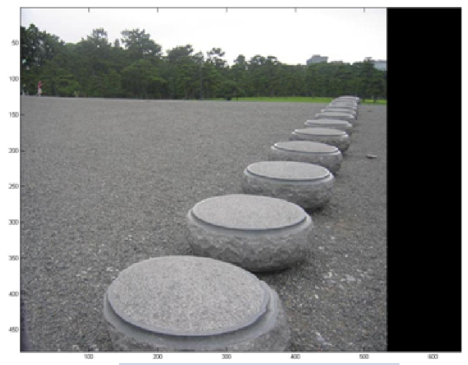
$$\frac{108{,}370{,}619 \ cycles}{640x480 \ pixels} = \ 352.8 \frac{cycles}{pixel}$$

An error of 20% can be seen between our predictions and our measurements. Given the difficulty involved to determine exactly what the compiler is optimizing, an error of this magnitude is not very surprising.

**Kernels**



| | | |
|---|---|---|
| 1 | -2 | 1 |
| -2 | 4 | -2 |
| 1 | -2 | 1 |



| | | | |
|---|---|---|---|
| 6 | 5 | -5 | -6 |
| 5 | 4 | -4 | -5 |
| -5 | -4 | 4 | 5 |
| -6 | -5 | 5 | 6 |

Figure 12: Edge and Smoothing Kernel Results

We provide a quick demonstration on the effects of different edge detections when used with our algorithm. The left figure demonstrates a kernel that is a very sensitive edge detector, while the figure on the right demonstrated the edge detection with smoothing that we implement to yield the results in our paper. It may be difficult to see on this scale, yet the background of the image consists of very small pebbles, with the horizon being lined with trees. Both of these are areas of high energy and are likely to be protected, while the areas that are important to us are the lower-energy stone pillars. Due to their relatively low energy, the sensitive edge detector will make these objects far more likely to me removed. On the other hand if we smooth out these very fine edges first, the field of tiny rocks becomes a fairly smooth solid-colored body, and the trees in the background become a smooth green body. If edge detection is executed upon this smoothed image, which the “smoothing edge detector” simulates, the stone pillars will now have a relatively high energy when compared to the rest of the image. This difference in sensitivity of edge detection accounts for the appearance of artifacts in the use one kernel, and that these artifacts are no longer present with the second kernel.

### Artifacts due to JPEG Compression

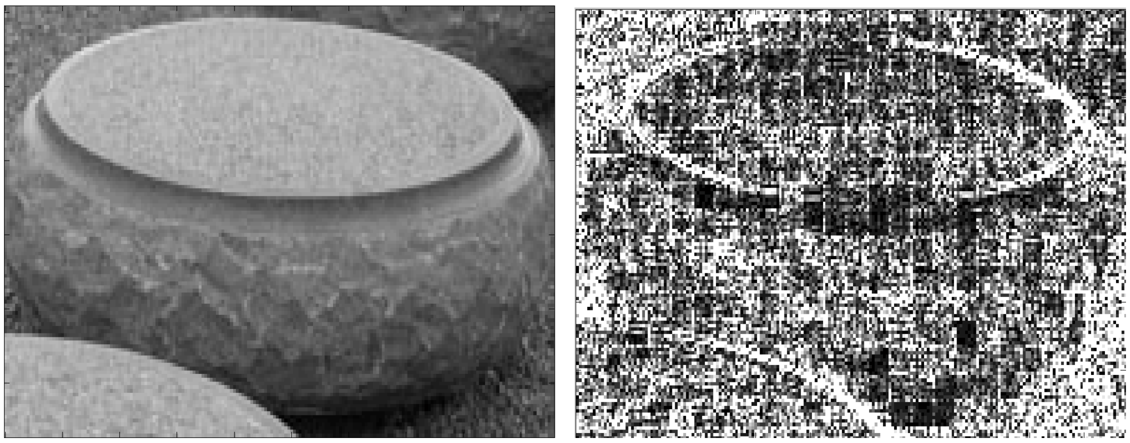


Figure 13: JPEG Compression

In this day in age, it is uncommon to store images in an uncompressed format. Chances are that images being input into the algorithm will be loaded from one of the prevalent image compression format such as JPEG. Due to the nature of the JPEG compression algorithm, the image has been broken down into smaller square chunks that are compressed individually. Since two adjacent pixels may be compressed separately, slight differences in the variable compression in JPEG may cause edge detectors to pick up edges that do not exist, such in the figure shown above. After one of many potential edge detection kernels is applied to the image on the left, faux-edges in a grid shape become evident as demonstrated in the image on the right.

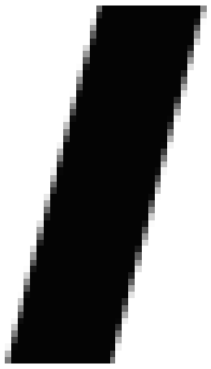


Figure 14: Anti-Aliased Line

Anti-aliasing techniques implemented by JPEG compression also has negative effects on seam-carving algorithms. In the figure above, it can observed that anti-aliasing has been applied to provide a smooth gradient on the interface between the white document and the black line. This has negative implications on edge detectors because it decreases the strength of a potentially important edge. This is a real problem that has been observed in our research and isn't simply speculation.

Fortunately, for the issues involved with the blocking and variable compression of JPEG, the smoothing-edge detection kernel does not suffer as much from these artifacts. Due to the fact that anti-aliasing is a form of smoothing, the smoothing-edge detector sometimes yields lower quality results.

**FACE DETECTION**

Face detection is required because of the relative importance of faces in photographs. In most situations, users would prefer not to distort the faces in their images during a resize, simply because the point of many pictures is to display the people present in it. However, the paradox with our algorithm is that faces may be more likely to get distorted due to their smooth nature. In images with high background energy, it is possible that the face is completely removed after we remove just enough x pixels to cover the width of the face, or just enough y pixels to cover the length of the face!

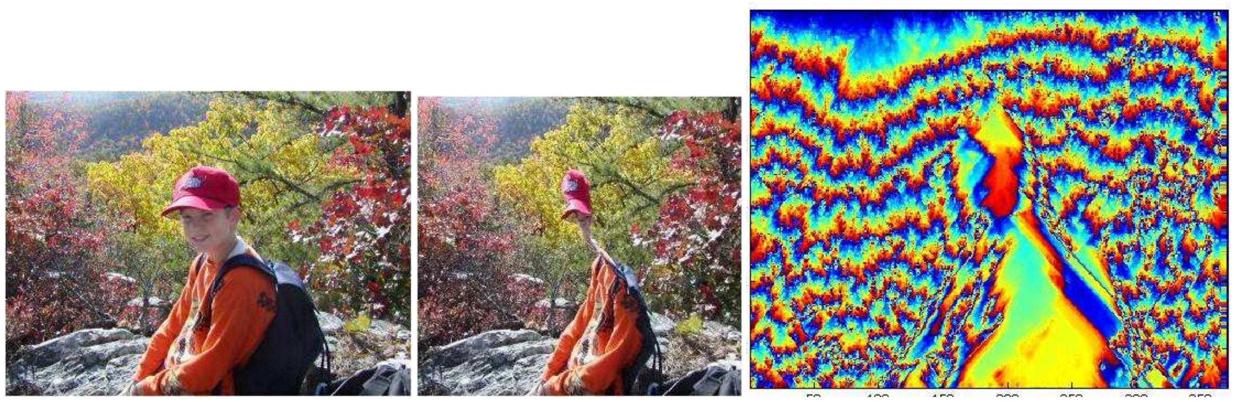


**Figure 15: The Boy with No Face**

As can be seen in this Figure 15, the face is the first thing that our algorithm chose to remove from the image. Looking at the cost map, we can easily see why. There is simply not enough energy in the boy's body or face with respect to the rest of the image, and the vertical seams will no doubt go right through him!

Fortunately, there are ways around this problem without using face detection at all. Many other implementations simply allow for a masking feature which would protect a user specified area within the image from being cut out. To save this image, the user could simply highlight the boy with a mask, and services such as rsizr.com[4] would keep his poor body in tact. In fact, our implementation also allows for such *protection masks* to be put in place. It is a good thing that protection masks allow for a manual protection of the boy. However, it would still make sense to automate the entire process using a quick and simple face detection mechanism. In our implementation it is done rather quickly (within 10 seconds) and essentially serves as a preprocessing calculation for the main resizing procedure.

[4] (rsizr.com - intelligent image resizing, 2007)

## Face Detection Algorithm

Our face detection algorithm is essentially the same one as implemented by Group 2 from Spring 2005. Both implementations relied on novel concepts as developed by Stanford's Diedrick Marius[5]. Our approach consisted of three stages:

1. Convert from RGB to YCbCr color space and threshold it to create a binary image.
2. Open and Closing, using morphological algorithms of erosion and dilation.
3. Detect and reject blobs using a simple blob rejection scheme.

The face detection addition adds a new feature to our algorithm that previous implementations lacked. On paper, it should automatically solve the issues with faces that other implementations often fail on when they do not manually protect the faces from being carved through. However, the capability of our face detection turns out to be rather limited due to the simplicity of the algorithm, and the limited computation time that it is allowed to have. It is, after all, nothing more than a preprocessing calculation performed before the resizing algorithm. Nevertheless, in the end it still adds a novel feature to our algorithm when used with simple portraits or landscape poses.

## YCbCr Thresholding

YCbCr is a color space that is generally used in video and digital photography systems, and it would make perfect sense to use it in our thresholding step for face detection. Y represents the luma, or brightness, component and is not used in our color thresholding. Instead, we use the Cb and Cr color values, which represent blue and red chroma components respectively. In our implementation, MATLAB performs the RGB to YCbCr conversion and sends the image to the DSK. The DSK only requests the Cb and Cr color values from the PC server, so from the onset it takes up 2 bytes per pixel in external memory. Once the image is stored on the DSK, we create a binary image that is high for pixels in the range: $100 < Cb < 133$ and $140 < Cr < 165$. These values were taken directly from Group 2's project. In their research, they claimed that these ranges were chosen to detect all types of skin under most conditions. The numbers were found after using a training set of 30 images. However, it is possible that there is a better range that would provide greater flexibility in lighting conditions and differing skin colors. In addition, perhaps the Y component can also be incorporated in thresholding to further improve accuracy and increase flexibility.

5 (Marius, Pennathur, & Rose)

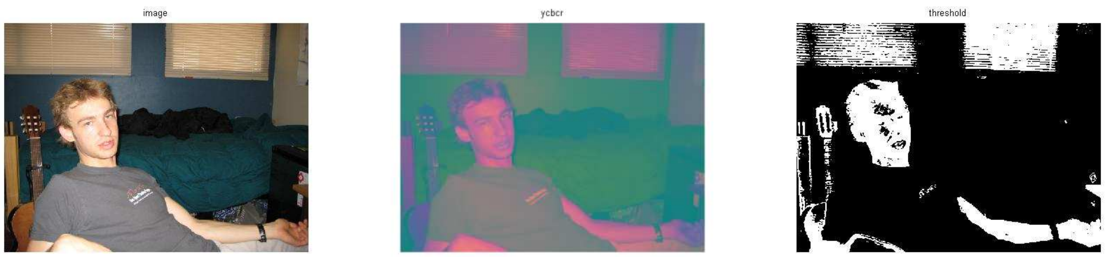


Figure 16: YCbCr Thresholding

## Morphology: Erosion

Erosion is the process of morphing an image so that insignificant artifacts will be removed. It is typically performed on a binary black and white image. The basic algorithm involves moving a structuring element through the entire image and finding overlaps between the structuring element and high regions within the binary image. In each overlap, only the central pixel may stay high. All other pixels are asserted low.

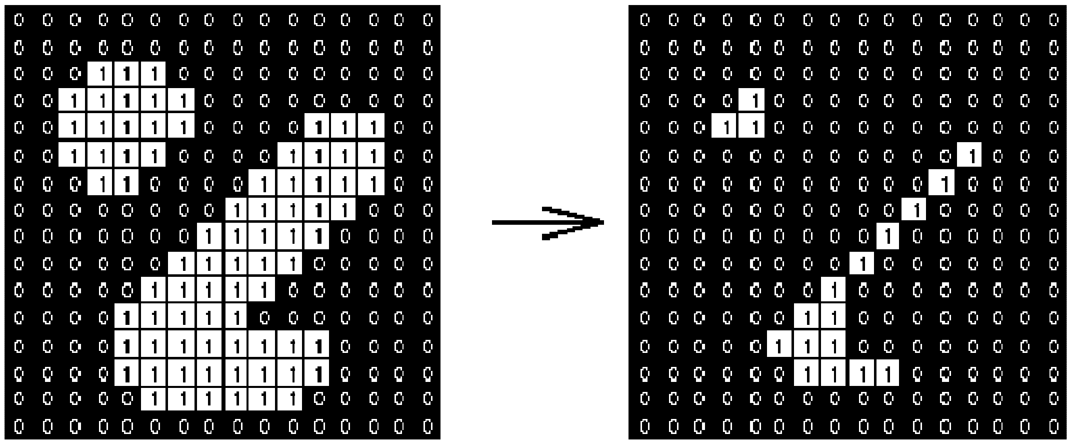


Figure 17: Erosion[6]

As can be seen in Figure 17, the large blob and fat stick on the left are reduced to a small group of dots and a skinny stick after erosion is performed using a square of 3x3 pixels as the structuring element. The process of erosion is performed on the DSK. In our Opening, we free the Cb and Cr images for the sake of saving DSK memory and operate solely on the thresholded binary image. In our Closing, we free the dilated image from our Opening sequence and operate solely on the dilated image from our Closing.

## Morphology: Dilation

Dilation is the opposite process of erosion. We perform dilation after erosion in an attempt to fill in the holes that may have been created by taking away too many high pixels. The algorithm involves applying

[6] (Fisher, Perkins, Walker, & Wolfart, Morphology - Erosion, 2003)

high pixels to the entire area of a structuring element for each high pixel in the image. All other pixels remain asserted low.

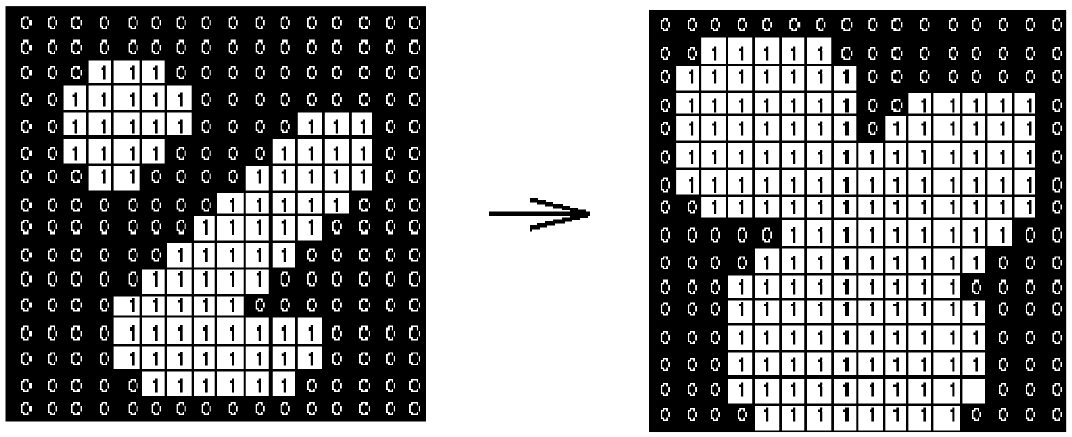


Figure 18: Dilation[7]

In this case, the large blob and fat stick on the left are increased to a single large blob that takes up most of the image after dilation is performed using a square of 3x3 pixels as the structuring element. The process of dilation is performed on the DSK. In our Opening, we free the thresholded binary image and operate solely on the eroded image. In our Closing, we free the eroded image from our Opening sequence and use the dilated image from our Opening.

## Opening Sequence

In our opening sequence, we simply erode the image and dilate it by a square of size 9x9. This value was obtained through trial and error using a small set of images. Given more time, it is possible to perfect the structuring element and its size. However, for our purposes, this number seems to work rather well. Essentially, we are removing artifacts that are smaller than 9x9 pixels in this Opening process.

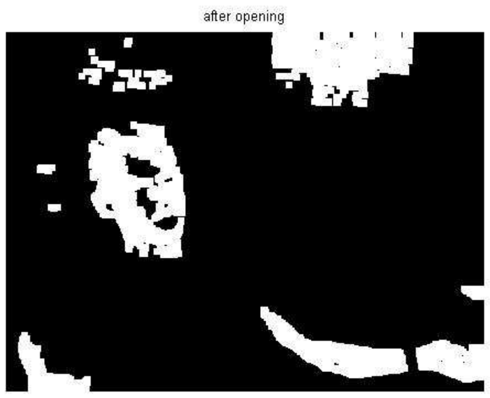


Figure 19: Image After Opening

[7] (Fisher, Perkins, Walker, & Wolfart, Morphology - Dilation, 2003)

## Closing Sequence

In our closing sequence, we simply dilate the image and erode it by a square of size 7x7. Once again, this value was obtained through trial and error, and can probably be improved upon. Closing is mainly used to fill holes that are created by eyes and lips, which are generally of a different color than regular skin. As can be seen in the Figure 19, there are plenty of holes to fill in. However, the closed image in Figure 20 does not fill in these holes completely, due to the smaller 7x7 structuring element. This problem will be dealt with later on.

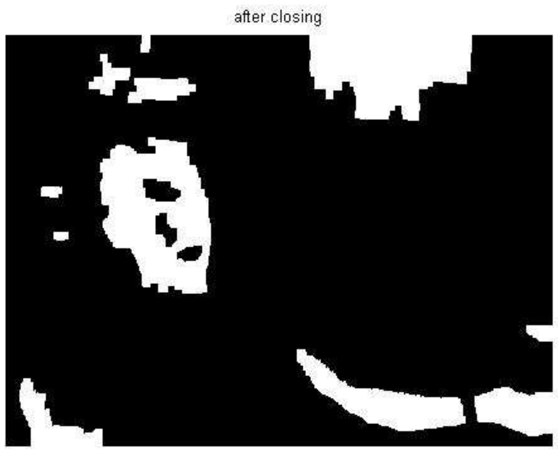


Figure 20: After Closing

## Blob Detection and Rejection

At this point, we should ideally have a series of blobs that are a superset of the faces we are looking for. We need to detect each of these blobs and separate them in order to examine if each one is a face. This is the most time consuming algorithm in our face detection series, and there was not enough time to come up with a more suitable technique for the DSK. Therefore, blob detection is performed in MATLAB on the image that has been closed by the DSK and transferred back to the PC.

In our algorithm, we search for a high pixel in the image. This pixel is placed into a queue. We then place all of its neighbors that are also high into the same queue. While the queue is not empty, we continue searching for high neighbors. Once the queue empties, we have one full blob. This blob is marked, and we continue searching for more high pixels that are outside of the blob. If the blob meets our rejection scheme, it is considered to be a face and kept. The process continues until we have finished finding all the high pixels that have not been marked as part of another blob.

Our blob rejection scheme is rather simple: we keep only the blobs that have a width-to-length ratio of between 0.7 and 0.9. These numbers were deliberately chosen to be less than 1 due to the fact that faces are always longer than they are wide. Once again, this range was obtained through trial and error, and there is probably a better combination of numbers from among Cb/Cr ranges, structuring element sizes, and width-to-length ratio ranges. However, as can be seen in the Figure 21, our methods worked flawlessly for a relatively noisy and complex picture.

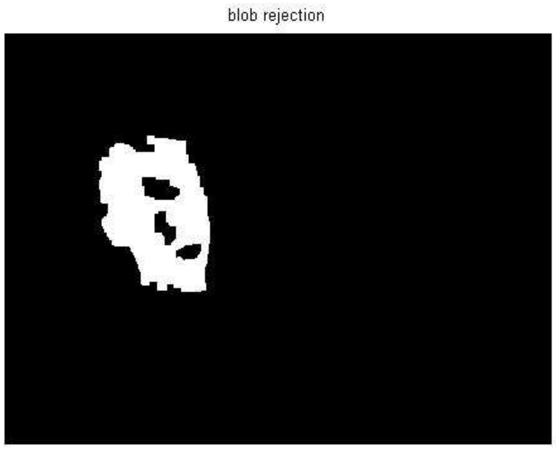


**Figure 21: After Blob Detection**

### Finishing Face Detection

To deal with the problem of remaining holes after our Closing sequence, we simply create a rectangle with dimensions: $Length = y_{min} - y_{max}$ of accepted blobs, $Width = x_{min} - x_{max}$ of accepted blobs. This rectangle is then filled with random noise in preparation for the resizing algorithm.

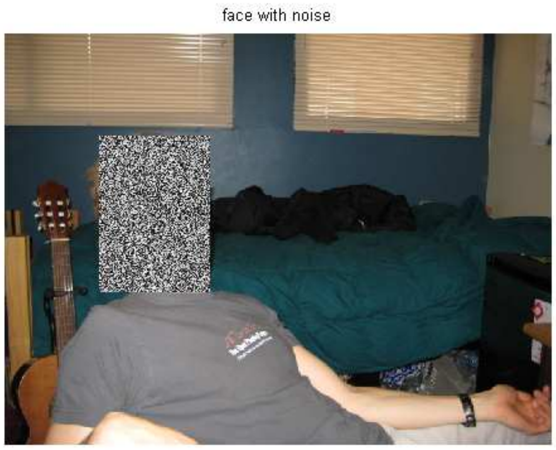


**Figure 22: Noise Rectangle**

The random noise is applied in MATLAB, since blob detection and rejection is already performed here. It is created by applying MATLAB's rand() to the entire rectangular region that was perceived to be the face. Intuitively, this random region would create a cost map of extremely high energy in the face region. As a result, the seams will generally stay away from any such region.

**Performance Analysis**

Our performance for face detection is relatively poor mainly because no paging or improvements to the algorithms were successfully completed within the given timeframe. In these estimations, we use the external memory access cycles of 7.125 cycles/access as given in the class handout[8]. In addition, we assume that the comparisons are done through additions, which cost 1 cycle each.

YCbCr thresholding involves testing each input pixel four times: twice in the Cb domain and twice in the Cr domain. There must be two tests per each domain due to the lower and upper bounds. In addition, we access the output threshold array once, resulting in 5 accesses per pixel in total. Similarly to the four accesses, there are four additions per pixel. This results in a total sum of:

$$5\frac{accesses}{pixel} * 7.125\frac{cycles}{access} + 4\frac{additions}{pixel} * 1\frac{cycle}{addition} = 39.625\frac{cycles}{pixel}$$

The actual measured value of thresholding was 35.473 cycles/pixel, resulting in an error of 11.7%.

Erosion involves 81 reads per pixel, due to the 9 by 9 size of our structuring element. This is assuming that the compiler does not perform any accessing optimizations whatsoever. In addition, there must be one write per pixel, resulting in 82 memory accesses in total. For each of the 81 reads, there must be 4 additions performed, and so we end up with 324 additions/pixel. This results in a total sum of:

$$82\frac{accesses}{pixel} * 7.125\frac{cycles}{access} + 81 * 4\frac{additions}{pixel} * 1\frac{cycle}{addition} = 908.25\frac{cycles}{pixel}$$

The actual measured value of erosion was 880.79 cycles/pixel, resulting in a small error of 3.12%. This indicates that there were indeed no optimizations performed by the compiler. This is probably due to an excessively large a block size (at 9 by 9). There were simply not enough registers in the DSK to handle so many variables. A faster way to implement this may be to use nine 3 by 3 blocks so that the compiler can perform some optimizations on the code.

Dilation involves 81 reads per pixel, due to the same 9x9 structuring element. There is also one write per pixel, resulting in the same 82 total memory accesses. It performs its comparisons outside of the two inner for loops, so there are only 4 additions performed per pixel. This results in a total sum of:

$$82\frac{accesses}{pixel} * 7.125\frac{cycles}{access} + 4\frac{additions}{pixel} * 1\frac{cycle}{addition} = 588.25\frac{cycles}{pixel}$$

The actual measured value of dilation was 549.12 cycles/pixel, resulting in an error of 7.13%.

8 (18-551, Fall 2007)

| Phase | Measured Cycles/Pixel | Estimated Cycles/Pixel | Estimation Error (%) |
|---|---|---|---|
| YCbCr Thresholding | 35.473 | 39.625 | 11.7 |
| Erosion (9x9 square) | 880.79 | 908.25 | 3.12 |
| Dilation (9x9 square) | 549.12 | 588.25 | 7.13 |

Figure 23: Estimations for Face Detection

**Face Detection Pitfalls**

There are some obvious limitations in the algorithm as described above. It does not use Eigenimage Template Matching as discussed in the Stanford paper, so it is relying solely on colors and blob aspect ratios, hoping to get lucky by dealing with an idealized image. As a result, it can only function under good lighting conditions, with clearly separated faces. The faces must be evenly lit throughout in order to assure a well formed blob after Opening and Closing. There must not be any items in the background with a similar color in the Cb and Cr domain and a length-width aspect ratio of less than 1 in order to prevent false positives from showing up. Also, the people must be clothed (at least with shirts), so that the high pixels after the Closing sequence maintain the target aspect ratio. In addition, the choice of a 9 by 9 square in Opening means that faces which are smaller than 9 pixels in length will be rejected in the Opening sequence, and therefore will ultimately not be detected. This means pictures of large crowds, where there are an abundance of small faces, would not fare well with our algorithm.

Despite numerous limitations to our face detection algorithm, it generally works very well for photographs in which a relatively small number of people pose, fully clothed, in front of a low energy landscape background. Luckily for us, this is a very common type of photograph, so the algorithm should serve its purpose quite well in general.

**White Balance Revisited**

In an attempt to determine exactly how much our face detection algorithm relies on a well lit, white balanced image, we tried to run it on a couple pictures taken with and without white balance. As expected, the results show that we simply must operate on a white balanced image to expect face detection to work. Our implementation relies very heavily on color within the image, and conditions must be close to ideal for the thresholding to function properly.

Take an image that was taken without white balancing, such as the one shown in Figure 24.

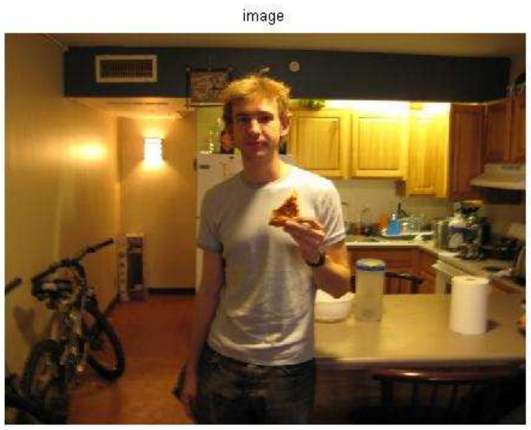


Figure 24: Photograph without White Balance

By inspection, we can already tell that this image is not properly lit. The pizza-eater’s face is half dark, and there are rays of yellow light coming from multiple sources from the kitchen. We should see a large amount of high pixels from the threshold due to this scattered light. As expected, the thresholded image in Figure 25 looks atrocious. The pizza-eater’s face is only half illuminated. His shirt is almost fully illuminated.

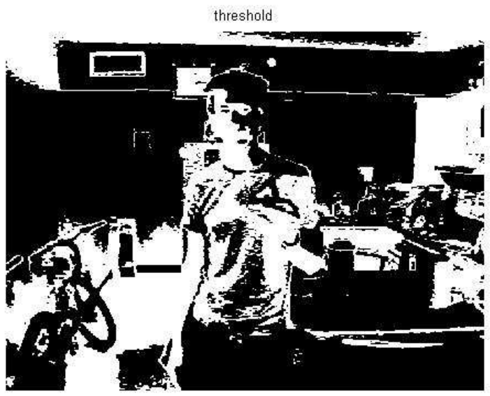


Figure 25: Thresholding without White Balance

In addition, there are high pixels everywhere – there is no way that an Opening and Closing sequence will leave us with a small number of reject-able blobs along with a perfect high face of appropriate aspect ratio. Not surprisingly, we end up with numerous false positives in the final detection (Figure 26). In fact, the face is not even detected at all. This image is clearly not friendly to our ambitious algorithm.

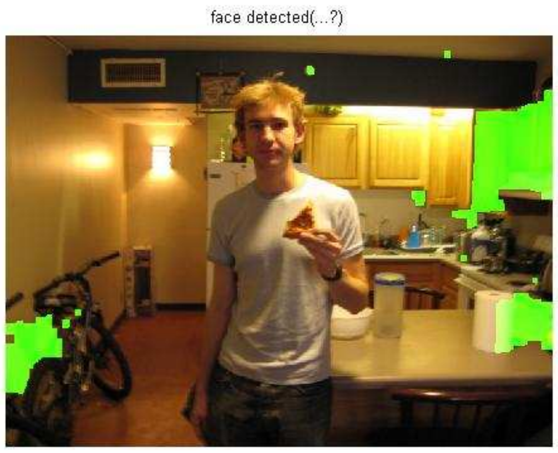


Figure 26: Face Detection without White Balance

Using a similar photograph with white balance enabled, we see drastically different results. As can be seen in Figure 27, the white balanced image is much more friendly to the pizza-eater's face and, as a result, to our algorithm. In addition, the bright lights in the background are impressively reduced, resulting in a clear, human-friendly photograph.

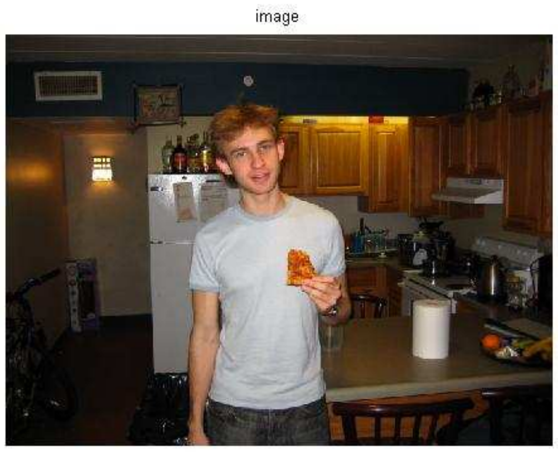


Figure 27: Photograph with White Balance

As expected, the thresholded image (Figure 28) is also much more manageable. The only foreseeable problem is that the cabinet right behind the pizza-eater's face has also been selected as high. This may create a face that is too wide to be accepted by the blob rejecter, because both cabinet and face will combine into one blob. However, we were lucky. As can be seen in Figure 29, the face had a good enough aspect ratio to be an accepted blob in the end.

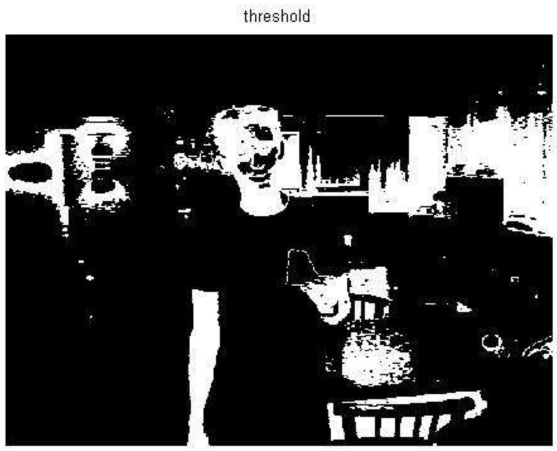


Figure 28: Thresholding with White Balance

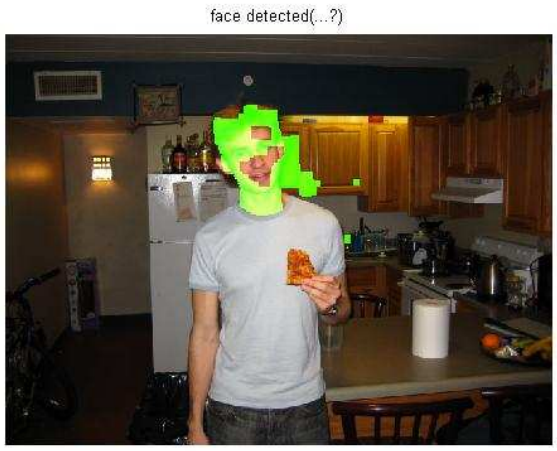


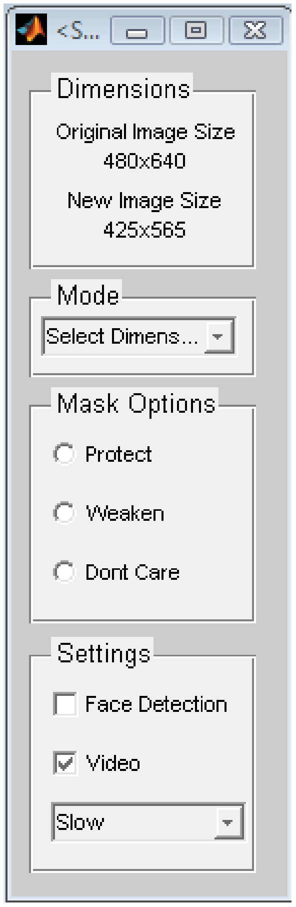
Figure 29: Face Detection with White Balance

It is clear from this isolated study that white balance is necessary for our face detection algorithm to function properly. In our examples above, the image that was not white balanced (Figure 24) experienced a numerous amount of *color cast,* in which the yellow light in the various sources was too heavy. Color cast caused the walls, cabinets, and the pizza-eater's shirt to be shifted towards a yellow tint. Unfortunately, this color of yellow happened to be within the Cb and Cr ranges that were being thresholded under and we were left with a threshold that contained too many high pixels. The white balanced image (Figure 27) fixed the vast majority of the color cast, with the exception of the small amount of yellow light on the right of the pizza-eater's face. However, our face detection algorithm happened to be flexible enough to work on this particular image. Fortunately enough, most cameras these days implement some sort of white balancing automatically, so there should not be too many problems in this regard.

## GUI

A Graphic User Interface (GUI) was required to allow users to take advantage of the complex nature of the content aware resizing algorithm. Without a GUI, many of the options available to the user, such as mask creation, would be virtually inaccessible due to the laborious nature of creating accurate masks computationally. Additionally, users can gain better understanding of how to modify the settings they currently have because they are given access to intermediate steps. Thus the GUI can be seen as a tool used for development and demonstration. Here we will discuss the features of the GUI and how to properly utilize them for the best results.

### Design Decisions



Original Image Size
480x640

New Image Size
425x565

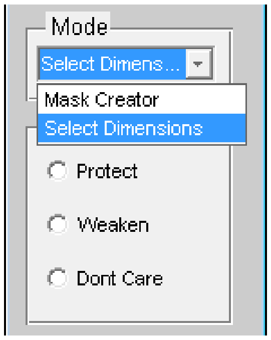
Mode: Select Dimens...

Face Detection: ☐
Video: ✓

Slow

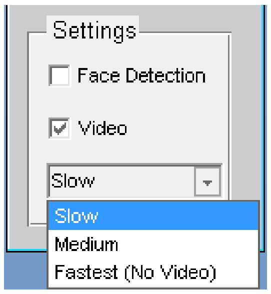
During the process of creating the GUI, we decided that it would be most beneficial to give users the ability to move objects around the workspace as they pleased. Furthermore, it was important the user need not perform math to indicate regions of interest that lie on the image or interact with the GUI. There were several obstacles given these design stipulations. Since each window itself is considered an independent GUI in MATLAB, a system by which variables could be shared among each of the GUI's was utilized. Also, the only feasible alternative to keyboard controlled image selection turned out to be mouse controlled selection. The end result was a dynamic system in which a control panel could be moved to any mouse controlled area while still maintaining command of each aspect of the GUI.

### Resizing an Image



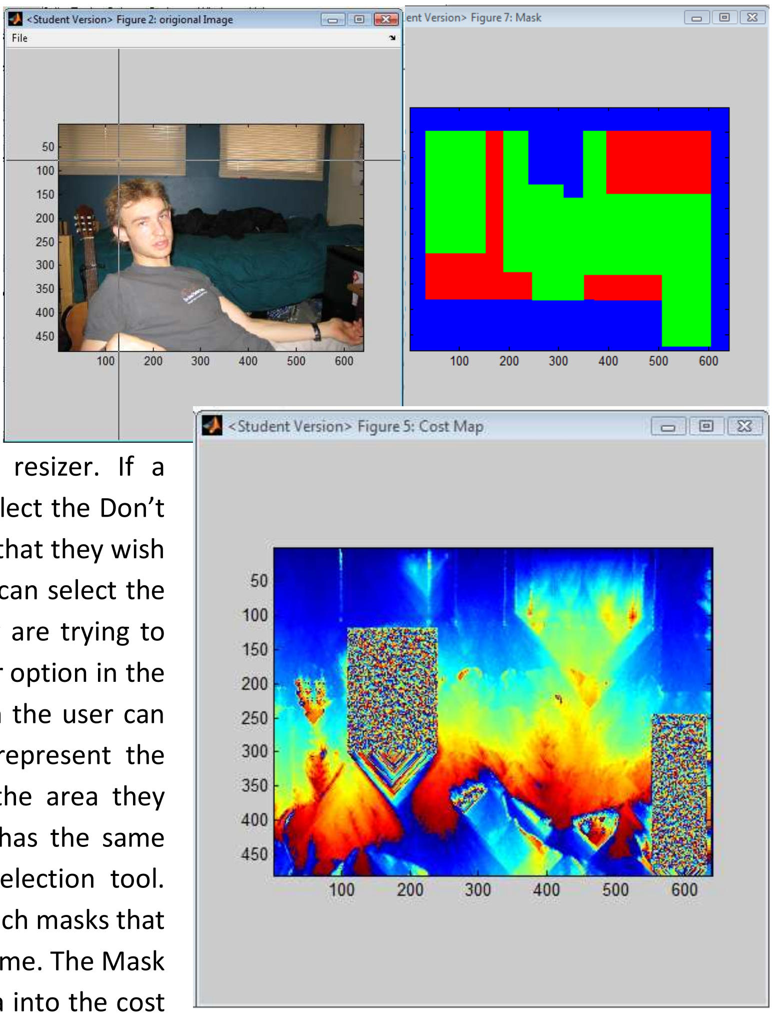
The user can make a variety of selections before performing resizing upon a desired image. To start the process, the user can select any image they like by using the file menu system on the figure labeled *Original Image*. The open command under the file menu is interfaced with windows such that a user can simply navigate to any image under the windows file system. Selecting an image will result in the display of that image under the *Original Image Figure* as well as an initialization of the *Mask Figure* (discussed below). The *Control Panel Figure* is then used for the remainder of the procedures.

The Control Panel figure is the main center of control for the user. It is used to display information about original and processed images, as well as for specification of algorithmic options. The Control Panel Figure will display the size of the image selected in the Dimensions panel below the *Original Image Size* label. An image with N rows and M columns will be displayed as "NxM". Beneath it we can see the label *New Image Size* with value *Not Resized Yet*. This label will change once a user specifies the new desired dimensions of the image. In order to do so, the mode can be changed to Select Dimensions in the pop-



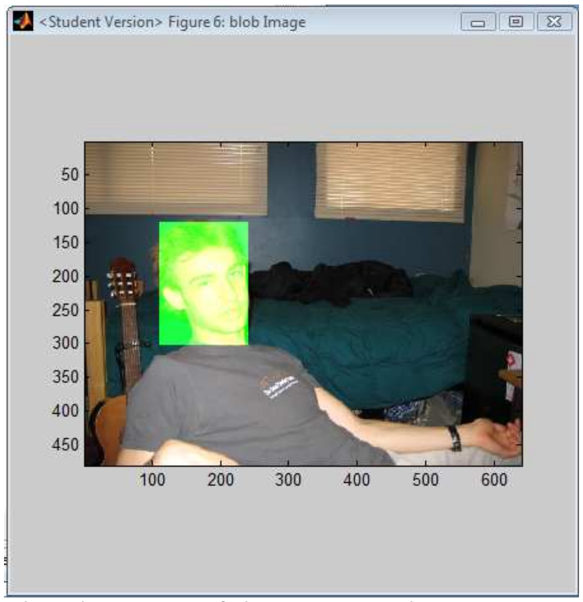
up menu on the Mode panel. Once selected, the program waits until the user specifies the new dimensions desired. To do so, the mouse can be used to click and drag a region of interest on the image itself. The program is robust enough to recognize any invalid input, even selections made outside the area of the image space. Once a new dimension is selected, the *New Image Size* value should have changed accordingly. It uses the same format as the *Original Image Size* label. At this point, the user has now done the bare minimum amount required to resize an image via GUI. To do so, the user can simply press the resize button for the output image to appear in the *Resized Image* window. If the user wishes a faster run-time, he may select a different speed in the drop down bar within *Settings*. The tradeoff to speed is the quality of the resizing algorithm. The fastest setting uses a resizing algorithm that removes 30 lines per cost map calculation. Though this improves speed, the output quality suffers. The slowest setting removes 5 lines per cost map calculation (for more information on the speed settings read the passages on seam removal). As a rule of thumb, speed and resize quality are negatively correlated.

**Mask Creation**


<Student Version> Figure 2: origional Image
File
ent Version> Figure 7: Mask
<Student Version> Figure 5: Cost Map

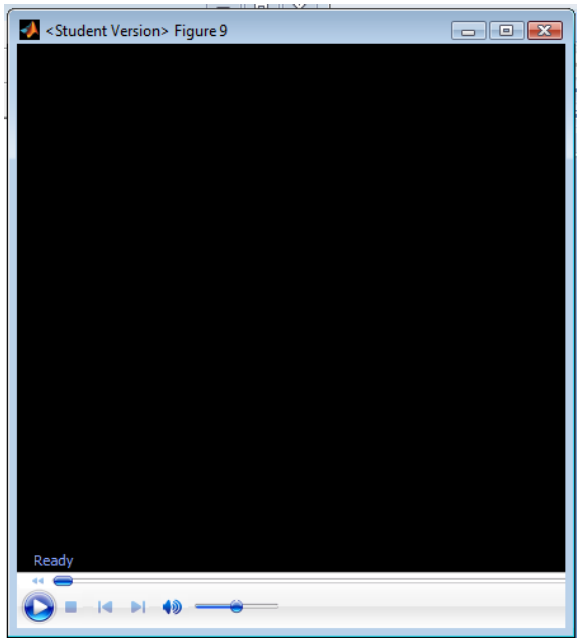
The GUI can be used to create a mask that can protect or weaken portions of the image. This tool is called the *Mask Creation Tool* and can be selected in the Control Panel on the Mode pop-up menu. The mask that a user creates contains the *weaken*, *protect*, and *don't care* states. These states can be used to make a region of the image inaccessible to the content aware resizer or to make that region particularly susceptible to the resizer. If a mistake is made, the user can simply select the Don't Care radio button and select the region that they wish to reinitialize. To achieve this, the user can select the radio button for the type of mask they are trying to create and then select the *Mask Creator* option in the pop-up menu on the mode panel. Then the user can select two points that they want to represent the upper left and lower right corner of the area they wish to affect. The mask creator also has the same robust safeguards as the dimension selection tool. This process can be repeated to create rich masks that protect and weaken areas at the same time. The Mask creator then inserts high frequency data into the cost map for protected areas and low frequency data for weakened areas.

**Face Detection Integration**



Many interesting images that people may want to resize can contain faces or smooth human skin tones, which are highly susceptible to a content aware resizer. Thus, it would be beneficial for the user to be able to have these regions automatically protected from removal. In order to do this, the user can select the face detection checkbox under the settings panel. This will set a flag in the algorithm to first run a face detection algorithm that will mask regions where faces may exist. Once a face is detected, the content aware resize protects faces in the same way that the protection mask protects a region of interest. Once the resize button has been clicked, the GUI will now contain a new figure labeled *Blob Image*. This figure contains the face detection results. The user should also notice that the region of the cost map that corresponds to this face detection mask has very high energy levels now.

**Video Creation**



One of the more interesting aspects of this project has not been the transformation of the original image to its final state, but rather the transition period that leads to the results. In order to view these results, a video option is available that allows the user produce a video after resizing. This video is created frame by frame at the point at which each seam is about to be removed. The seam to be removed is highlighted with a red line. Each of these frames are concatenated in time to produce an AVI video. When the user selects the video checkbox on the settings panel the GUI creates an ActiveX module and allows the video to play in a MATLAB figure in an instance of Windows Media Player®. From this display, the user can both play the video and control the display of specific frames if preferred.

## Future Work

Given that this algorithm has only been introduced to the public at SIGGRAPH 2007, and the limited time available for our particular research, there has not been an opportunity to explore all the issues associated with the algorithm and potential input images at a sufficient depth. There is a plethora of different techniques that can be implemented in the interest of robustness, computational efficiency, and visual appeal. To further progress this research, some of the ideas that we formulated are discussed below.

### Removing Unnecessary Calculation and Data Structures

It has been mentioned earlier that recalculating the entire edge and cost maps per every seam removed produces the optimal solution. These calculations are responsible for the majority of the runtime. Questions arise when observing Avidan and Shamir's demonstration of the algorithm[9], as to how they successfully resize images so quickly. It has been discovered that pre-computation of optimal seams to be removed are embedded in the image file, leaving only the removal of these seams to be done once the user inputs their destination dimensions. Unfortunately this is not necessarily ideal for many applications that have real-time demands such as cameras and video. In order to address this issue we suggested a collection of approximations to constant recalculation.

Instead of approximating we believe that there is a way to organize the pixels of the image such that recalculating only the necessary edges and costs can be completed efficiently. It turns out that edge and cost maps only change minimally between iterations of seam removal. If there was a way to identify which pixels would remain the same and which would not, we could significantly decrease computation time. If we were to try to calculate only necessary areas in our current data structure – a linear array of pixels – we would have to keep track of which seams overlap so that we can update only the necessary seams. Unfortunately we have not been able to explore the feasibility of such a data structure, yet there are many ways in which we can avoid recalculating the unnecessary portions of the cost map.

### Faster Average Transfer

Upon examining the results of data transfer rates and the amount of time associated with setting up connections, it is evident that removing this unnecessary overhead can provide significant speedup. By rewriting the existing communications between the PC and the DSK we could potentially achieve such a goal. Another alternative is preparing a workspace in which a general purpose CPU and a DSP processor share a common bus in an efficient manner. Such a workspace would provide a whole new world of possibilities for performance.

We have discussed a collection of pitfalls of our current implementation as well due to time restrictions that require no elegant solution, just additional time. There is much room for exploration within this algorithm to make it more desirable to home consumer and commercial applications, and we hope that our research has taken a big step towards this goal.

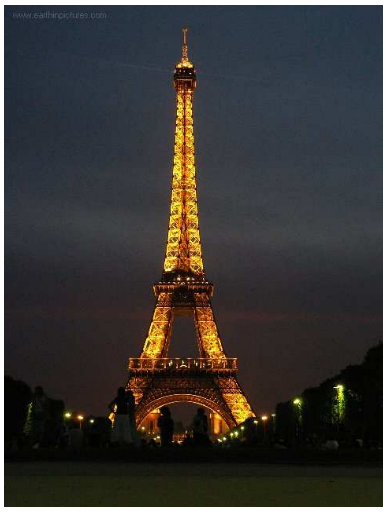
[9] (Avidan & Shamir, Video: Seam Carving for Content-Aware Image Resizing )

## RESULTS

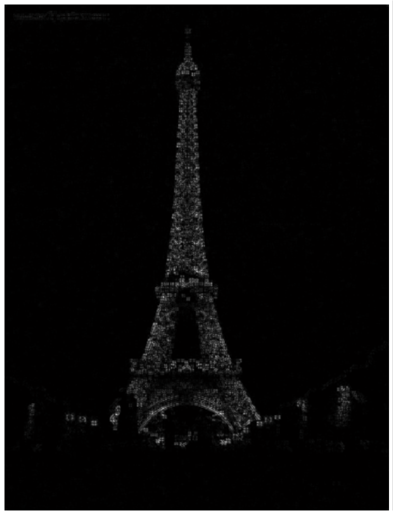
The following are some results yielded by the final implementation of our algorithm, removing 5 seams per recalculation of cost map. We also provide a comparison of our results with rsizr.com[10] using the same resized height and width to provide an accurate comparison of our performance in terms of visual appeal. All images are scaled with referenced to the original image as displayed in this document in order to provide an accurate comparison as well.

[10] (rsizr.com - intelligent image resizing, 2007)

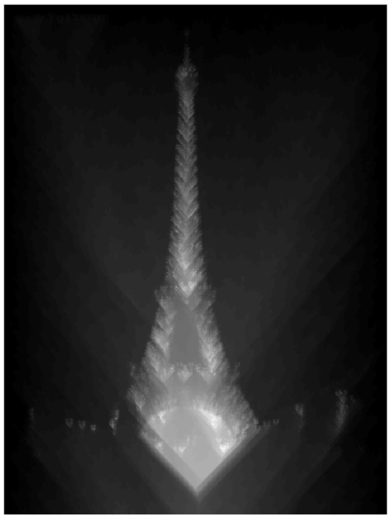
**EXAMPLE 1: THE EIFFEL TOWER**



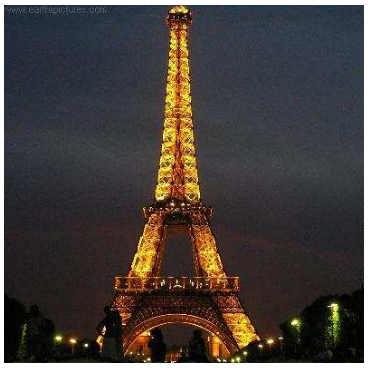
Original Image



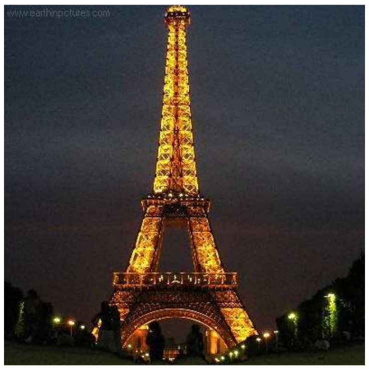
Edge Map



Cost Map



DSK Output

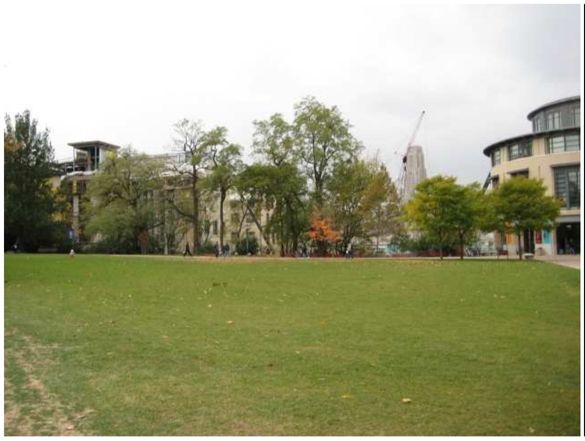


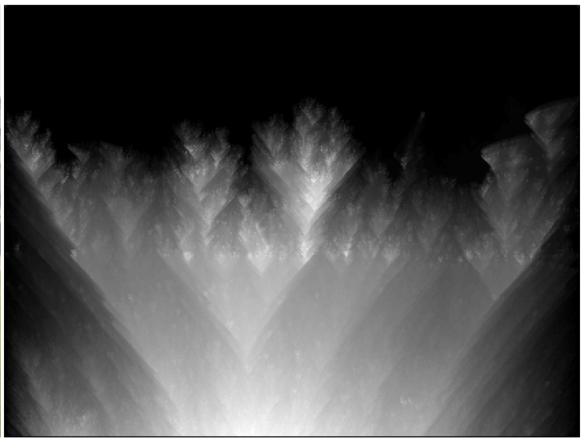
rsizr.com Output

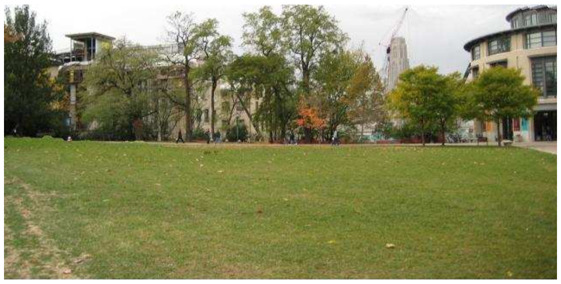
Above is an example of the different stages and the output of our algorithm when inputting an image of the Eiffel Tower. This image is ideal since all of our important content – i.e. the Tower – is centered in it, and it makes for a good illustration of the algorithm. The second image shows the result of edge detection, which clearly marks the entirety of the Eiffel tower as an area of high energy. Next we build the cost map, which shows seams of higher energy associated under the Eiffel tower. This results in a tendency to carve seams around the tower, not through the middle. We push the algorithm to its limits by resizing smaller than the object itself, so the algorithm actually makes the tower shorter. Fortunately, it still looks very much like the Eiffel Tower in the end. In fact, the shortening is almost unnoticeable due to smart seam carving. We compare the results of commercial implementation resizr.com[11] with the result of our algorithm. They are nearly identical.

[11] (rsizr.com - intelligent image resizing, 2007)

**EXAMPLE 2: CARNEGIE MELLON CUT**


Original Image

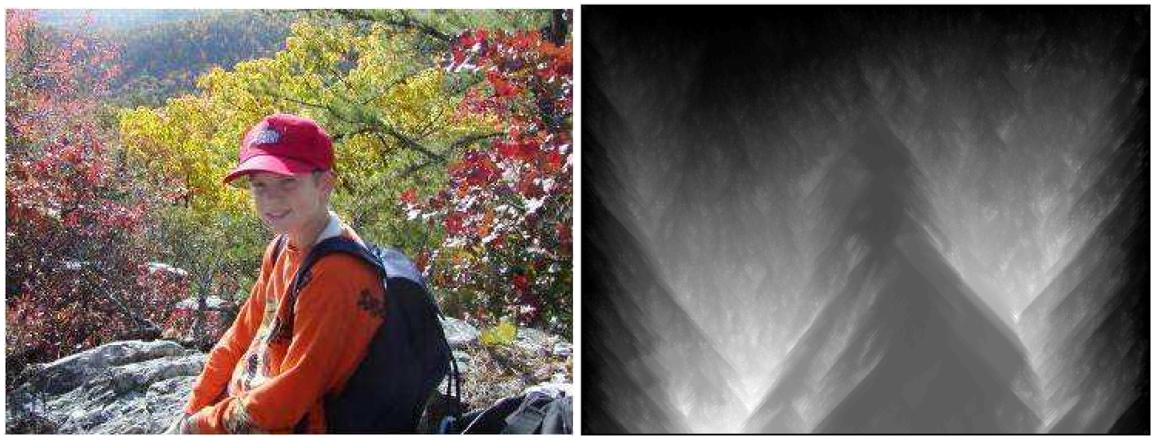

Cost Map

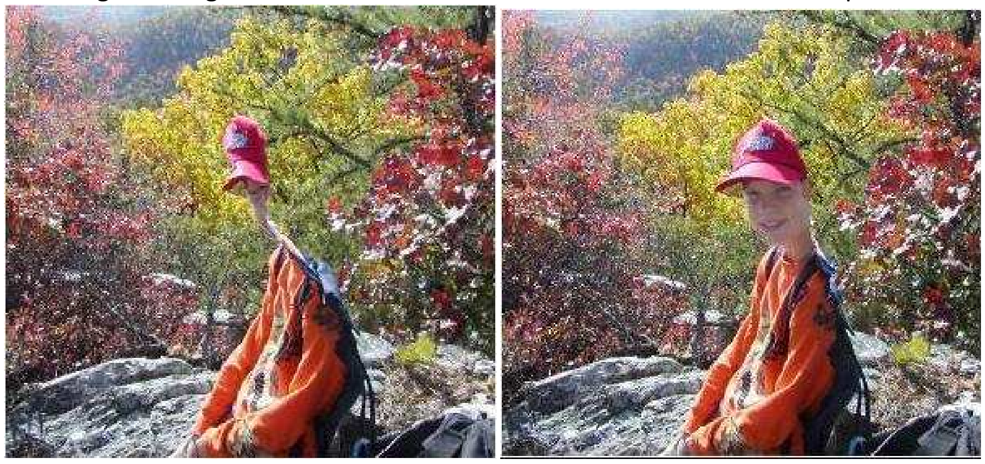

DSK Output

In the example above we have in image in which we can easily predict from where the seams in the image will be removed. As we can see in the cost map, the sky is very low in energy. Therefore, we simply remove horizontal seams from the sky. Since we attempted to resize the image by more than the height of the sky, the algorithm begins to remove seams of pixels in the grass, another area of low energy. Although not depicted, our algorithm indeed yields a result visually identical to that of rsizr.com[12]. Once again, this is simply a demonstration to promote understanding of the algorithm. In the following example we will see how our algorithm deals with a more complicated image.

12 (rsizr.com - intelligent image resizing, 2007)

**EXAMPLE 3: THE BOY WITHOUT A FACE**

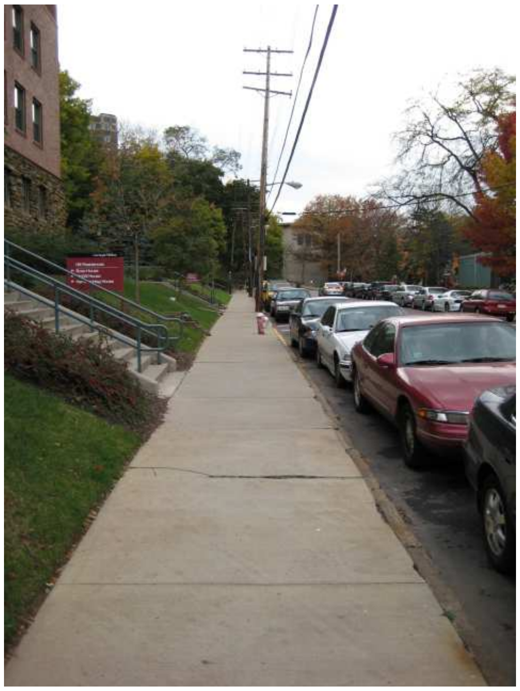

Original Image Cost Map

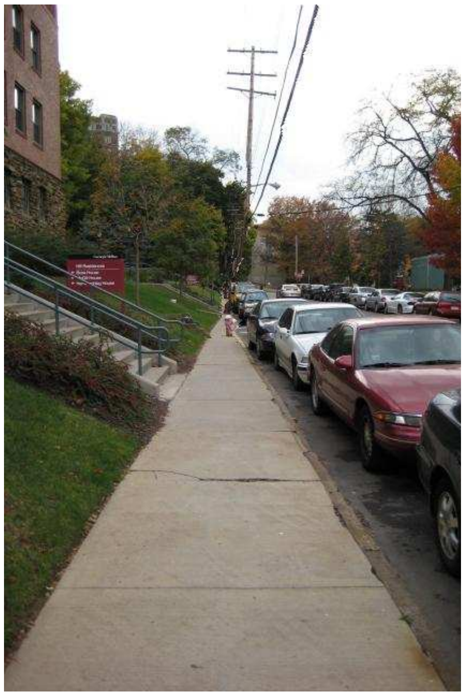

rsizr.com Output DSK Output

We have a leg up on the competition is due to our ability to handle faces. As we can see above, we have an image of a relatively solid colored boy sitting in a very high energy surrounding. The cost map shows that the trees around the boy are associated with far more energy than the boy himself, leaving him nothing more than a silhouette in the map. As we can see, rsizr.com[13] is unable to handle this case. The first and only seams that can be removed go directly through the boy's face. This is a case in which Shamir and Avidan[14] have predicted, yet have not continued to carry out a solution that has been made public. With our automated face detection we are able to effectively protect his face as seen above.

13 (rsizr.com - intelligent image resizing, 2007)
14 (Avidan & Shamir, Seam Carving for Content-Aware Image Resizing , 2007)

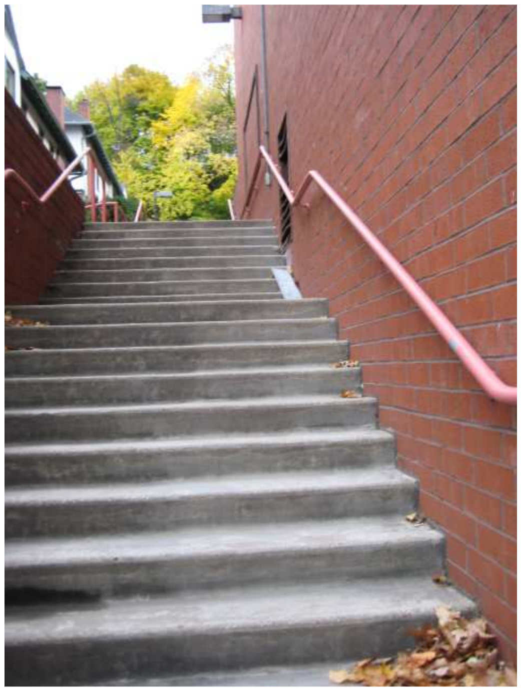
## EXAMPLE 4: NARROW SIDEWALK

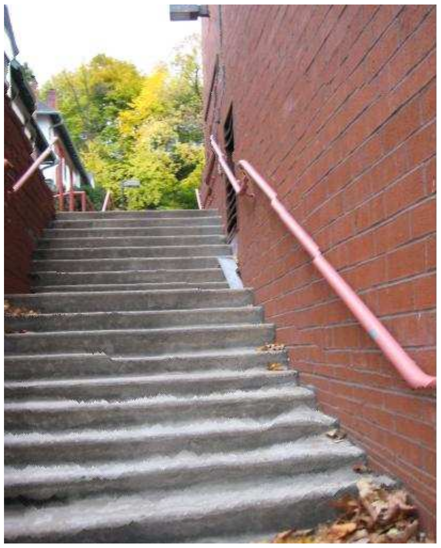

Original Image


DSK Output

The purpose of this case is to demonstrate that seams can be removed from the middle of the image and do not have to extend to both ends (for example a sky often hits the right and left edge of the image). In this case, the low-energy sidewalk eventually ends in an area of very high energy trees. Our implementation of the algorithm is able to produce a resized image within seconds by means of approximation, leaving a result once again comparable to other implementations.

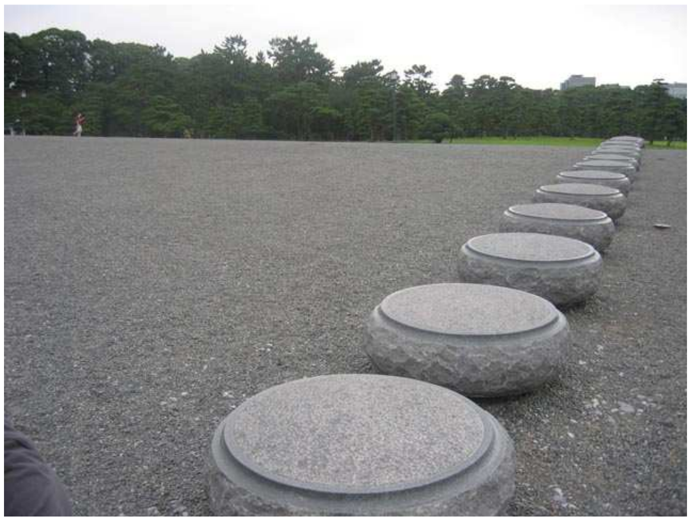
## EXAMPLE 5: ONE STEP AT A TIME

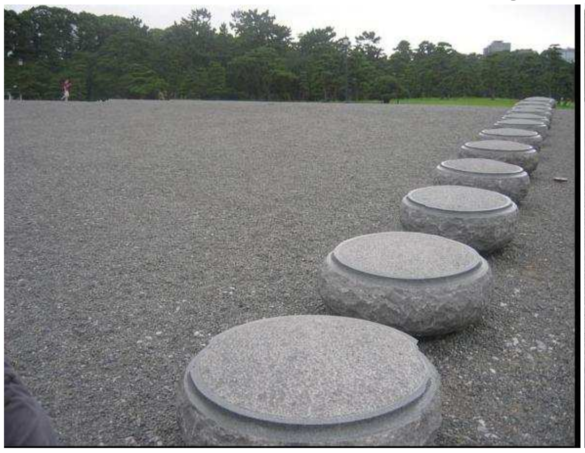

Original Image

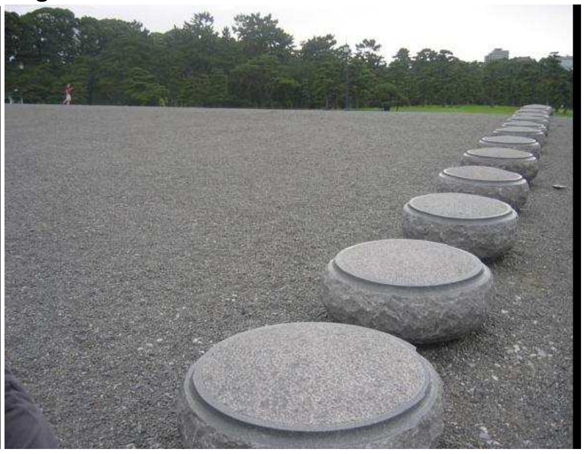

DSK Output

As robust as our algorithm may prove to be, it still suffers some pitfalls. Images that tend to cause problems are those with several parallel and perpendicular lines, such as stairs, bricks, and window shades. The reason this occurs is because these edges are not as strong as those involved in other objects in the image, such as trees and people. Other implementations of this algorithm yield similar artifacts for similar reasons. We have not yet formulated a method to address this issue, but it is indeed a case that needs to be addressed.

### Example 6: Approximately Right

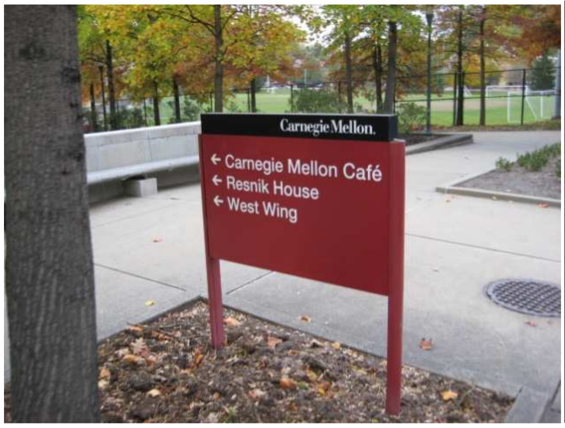

Original Image


5 seams/cost map calculation

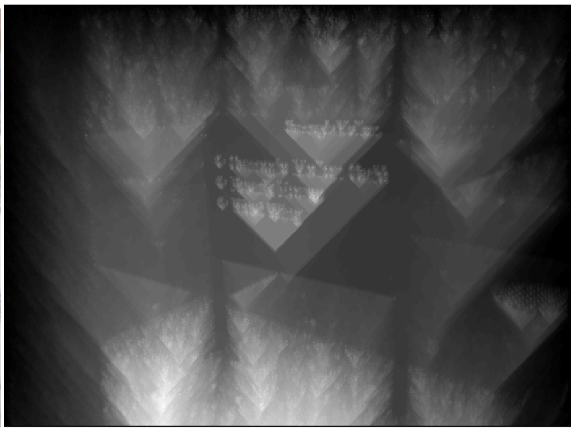

1 seam/cost map calculation

In the above example, we demonstrate the potential negative effects of our approximations. Due the majority of the image being associated with high energy, our approximation causes us to be more likely to remove bundles of seams from the same area. This causes artifacts as seen the on bottom left image. Our round pillar is no longer so round when removing 5 seams per recalculation. However, our "optimal" solution of one seam per cost map recalculation yields more ideal results, as can be seen in the bottom right image.

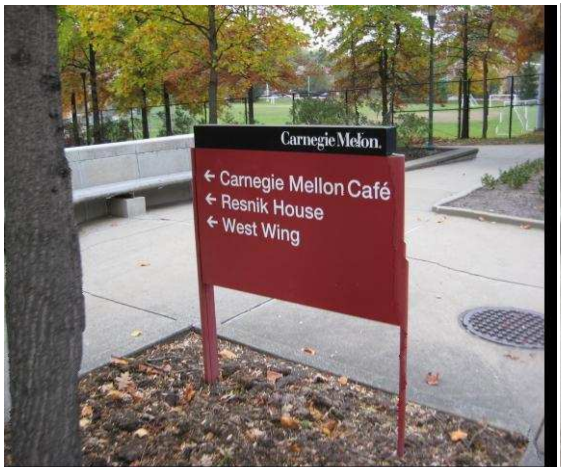
**Example 7: It's a Sign…**



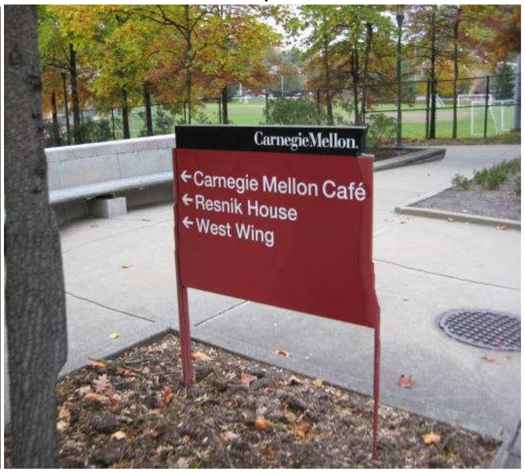
Original Image



Cost Map



DSK Output

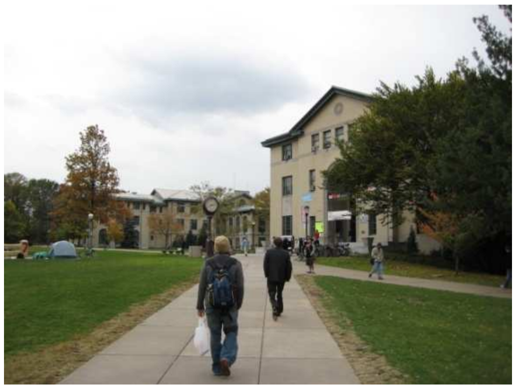


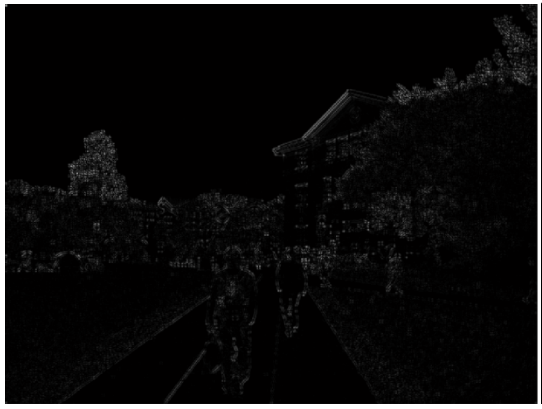
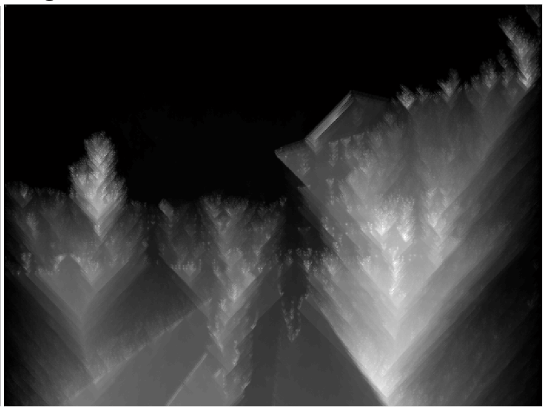
rsizr.com Output

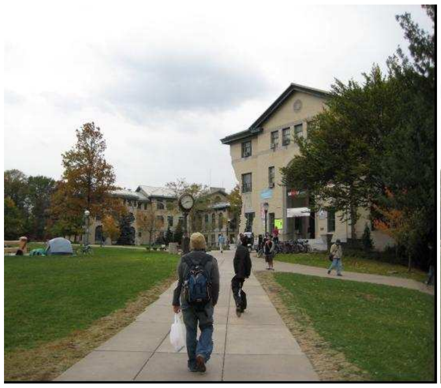
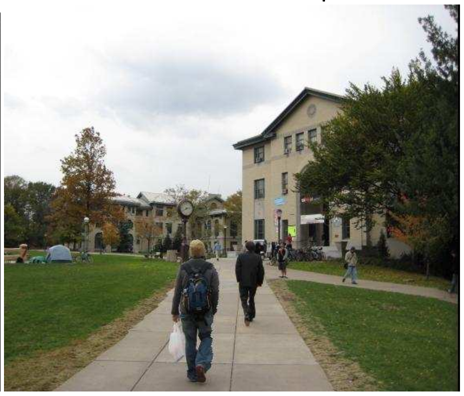
During the initial stages of developing our algorithm, we had hypothesized that text would be an issue in our resizing due to its solid colors. But as it turns, the edges associated with text itself are able to negate the detrimental effects of its solid colored background. These edges can be observed in the cost map. Although we are more likely to carve through the sign itself (which yields the artifacts visible in rsizr.com[15]), the areas around the text are less likely to be removed. Our implementation actually produces a more visually appealing output than rsizr.com as it yielded far less deformation of the tree shape and of the right leg of the sign.

[15] (rsizr.com - intelligent image resizing, 2007)

**Example 8: Oh Doherty!**


Original Image



Edge Map Cost Map



DSK Output without Protection Mask DSK Output with Protection Mask

Above we see another undesirable output of our algorithm. In the edge map and the cost map, we see that the roof of Doherty Hall is associated with high energy, since there is a significant amount of color difference between green and white. However, the sky is of similar color to the building itself, which weakens the edge between them severely. As a result, we are likely to carve through this wall from the sky causing the artifact seen on the bottom right. We acknowledge this issue and protect the left boundary of Doherty Hall with our GUI's protection masks, yielding a far more desirable output seen on the right. Unfortunately, we are not yet able to autonomously handle this case.

## Division of Labor

Ziv:

- Resizing Algorithm in MATLAB and DSK
- PC server to send images and receive re-sized images
- MATLAB backend that runs the PC server
- Looping to allow DSK to resize to any value
- Use mask information (from GUI and face detection) to make random noise or uniform color in targeted regions

Jim:

- Face Detection in MATLAB and DSK
- PC server to perform face detection on DSK
- MATLAB backend that runs the face server
- Combine resizing algorithm with face detection

Many:

- GUI
- Mask to take user input specifying protect and remove regions in the image
- Take user input specifying the desired dimensions of the image
- Connect GUI to MATLAB backend

## Conclusion

Our implementation of the content aware image resizing algorithm on the DSK has shown serious results. By tweaking the kernel used for cost map calculation, we have improved performance on images with widespread high energy content significantly. Through the use of face detection techniques, the algorithm can now automatically protect important information that previously required manual user input. An interactive GUI also makes all aspects of the algorithm accessible and comprehensible to any user. Concluding our semester's long hard work, we believe that we have brought this technology a couple steps closer to commercial application.

## Works Cited

18-551. (Fall 2007). Caches, Memory Access, Our Timing Estimates, etc.

Avidan, S., & Shamir, A. (2007). Seam Carving for Content-Aware Image Resizing . *SIGGRAPH.*

Avidan, S., & Shamir, A. (n.d.). *Video: Seam Carving for Content-Aware Image Resizing* . Retrieved from http://www.faculty.idc.ac.il/arik/IMRet-All.mov

Fisher, R., Perkins, S., Walker, A., & Wolfart, E. (2003). *Morphology - Dilation*. Retrieved from http://homepages.inf.ed.ac.uk/rbf/HIPR2/dilate.htm

Fisher, R., Perkins, S., Walker, A., & Wolfart, E. (2003). *Morphology - Erosion*. Retrieved from http://homepages.inf.ed.ac.uk/rbf/HIPR2/erode.htm

Marius, D., Pennathur, S., & Rose, K. (n.d.). Face Detection Using Color Thresholding, and Eigenimage Template Matching.

*rsizr.com - intelligent image resizing*. (2007). Retrieved from http://rsizr.com/