C.A.V.E.S.

Content Aware Video Expansion and Scaling

Fall 2008, Group 1

Aneeb Qureshi (aneebq@andrew.cmu.edu)
Gregory Tress (gtress@andrew.cmu.edu)
David Xiang (dxiang@andrew.cmu.edu)

1. Problem

Since the proliferation of consumer television technology in the 1940s and 1950s, the American television industry has used the 4:3 aspect ratio as a standard for filming and broadcasting, established by the National Television System Committee. This standard has guided both television manufacturers and content providers. More recently, with the increased availability of high-definition content, the nationwide switch to digital broadcast television, and the introduction of low-cost digital television sets, consumers have demonstrated an interest in viewing content in the 16:9 "widescreen" aspect ratio. This aspect ratio was used by the Advanced Television Systems Committee as part of its standard for high definition television and its basis for new standardized television resolutions. Many televisions in the market today are designed around the 16:9 aspect ratio, and television studios are transitioning toward digital widescreen filming of content. For consumers, digital television generally results in a higher-quality picture, with increased resolution and decreased visible interference.

At the same time, this transition results in a backwards-compatibility issue. The majority of existing television content has already been filmed in 4:3, and in order to display 4:3 aspect ratio content on a 16:9 television, the consumer must choose how to adjust the aspect ratio on the television itself. Widescreen televisions usually have a similar set of user-defined options for this purpose, as described here. One option is to maintain the original aspect ratio of the content and center it on the screen. Because the 16:9 screen is significantly wider than the 4:3 video frame of the same height, black bars will appear on the sides of the video; this phenomenon is known as "pillarboxing." As a result, the consumer loses the benefit of the wider screen; in fact, a widescreen television of the same viewable surface area as a traditional 4:3 television will yield a smaller representation of the same video. A second option is to horizontally "stretch" the 4:3 video, forcing it to fill the entire screen. This results in a significant and noticeable distortion. Yet another option is to "zoom" the 4:3 video, cutting off the top and bottom of the frame, so that the video can fill the entire screen without distorting the content. However, this zoom functionality is generally unintelligent and results in cutting off important parts of the video. Consumers will ultimately continue to face aspect-ratio difficulties as long as 4:3 content is broadcasted. Currently, there is no simple way to view this 4:3 content on a 16:9 television while both utilizing the entire screen and avoiding noticeable distortion.

In this paper we detail a system that intelligently converts video content from one aspect ratio to another. In the case of the television aspect ratio problem described above, this system would modify a 4:3 input video in such a way that the resulting output would fill the entire screen of a 16:9 television but would not appear severely distorted. While our motivation stems specifically from the 4:3 to 16:9 conversion problem, the system is

generalized in such a way that it can convert between any two aspect ratios. Our system does not operate on actual television content in real time, but will still function as a valid proof-of-concept for this, since our algorithm can be used as the basis for a consumer-end device which does perform this function. Such a device in the form of a set-top box would require adequate hardware and a modification to the parameters of our algorithm to operate in real-time. In our case, we can still create a 4:3 to 16:9 conversion with lower resolution and lower frame rate to demonstrate the effectiveness of the system. With the prevalence of mobile devices and web-based video in a variety of physical resolutions, there are many possible applications of aspect ratio conversion in addition to television-specific content.

There are numerous methods explored for content-aware image resizing. For videos in particular, there has been increased research in video retargeting. Video retargeting relies on solving a large system of linear equations in order to determine the desired output aspect ratio. As we will detail in a later section, video retargeting is not suitable for the C67 DSK due to the large amount of computations and memory accesses. Instead, we will be using a modified version of seam carving which takes into account temporal dependency between frames. By using seam carving instead of video retargeting techniques, we create a trade off between quality and speed. Considering the lack of power of the C67 DSK, the tradeoff for not using video retargeting is a suitable sacrifice.

2. Novelty

This project relates closely to Fall 2007 Group 6's project: Content Aware Image Resizing as a video is simply a series of images. However, the algorithm described by their project cannot be directly applied to video due to large artifacts that occur. When the traditional seam carving method is applied to each frame in a video, the result is jerky: parts of the frame appear to jump from one area to another, usually shaking left and right sporadically. This occurs because the seams in one frame are unrelated to the seams in the next frame. Hence, when seams move by enough pixels between 2 given frames, the viewer of the video observes this jerky effect. Throughout the paper, we will refer to a less extreme version of the jerky artifact as a wavy artifact, in which only certain segments of the frame experience mild temporal distortion in their expansion.

With that in mind, we have added new improvements to the seam carving algorithm which allows the algorithm to function properly for video sequences. These tweaks are described in detail in Section 3. In addition, our project has an enormous amount of data to transfer between the DSK and the PC. This introduces memory and speed problems which are not present in Group 6's project.

In regard to creating the prominence scores, we have fully adopted Group 6's face detection algorithm. We felt

that face detection should not be the prominent focus of our project and testing, hence we decided to not explore other face detection algorithms.

3. Algorithms

Edge Detection

The first step in creating the prominence scores for a frame involves detecting high energy areas based on edge detection. Edge detection can simply be implemented by correlating an image with kernels that detect changes in color between adjacent pixels. In order to simplify calculations, the RGB image is converted to grayscale. This reduces the amount of calculations and memory accesses to a third of the original amount. As there are many operators which produce edge detection, we use the Sobel operator as it only uses 3-by-3 kernels. Smaller kernels are more favorable as they reduce the number of calculations. For example, the most times a pixel can be operated on by a 3 by 3 kernel is 9 times. For a 4 by 4 kernel, a pixel can be operated on 16 times. As you can see, the amount of operations on each pixel increases by 7 operations. Considering that the kernel is being correlated with a frame of 320 by 240 (76,800 pixels), saving 7 operations on each pixel is significant.

1	2	1
0	0	0
-1	-2	-1

1	0	-1
2	0	-2
1	0	-1

Figure 3.1: Sobel operator kernels

The Sobel operator uses the kernels defined in Figure 3.1 and the L2-Norm in order to implement edge detection [1]. The L2-Norm is usually implemented by the square root of the sum of squares of each kernel output. However, a faster method is to approximate the L2-Norm by the sum of the absolute value of each kernel output; this allows for the process to be even faster on the DSK. As noted in Figure 3.1, each kernel represents either the gradient in the x-direction or in the y-direction. In actual implementation, these two kernels can be correlated at the same time (meaning that we only need to iterate through the image once).

When implementing the Sobel operator on the DSK, we do not do computations for the "zero" elements in the kernels to save cycles. In addition, we do not multiply values by one, as it is a useless computation. The output of edge detection from the DSK is shown in Figure 3.2.

Figure 3.2: Edge detection

Face Detection

Face detection is an important aspect of the prominence scores in order to ensure that faces are not distorted. In many cases, faces can lack detail (depending on how close the face is to the camera). When this occurs, edge detection will fail to assign high energy to that face. Henceforth, face detection makes up for the shortcomings of edge detection alone.

As noted earlier, we have fully adopted the face detection algorithm by Group 6 from Fall 2007. The approach consists of three sequential stages: creating a binary image via YCbCr thresholding, opening and closing via erosion and dilation, and blob detection and rejection. The face detection process is shown in Figure 3.3.

Figure 3.3: Face detection block diagram

1: YCbCr Thresholding

YCbCr is a color space that is often used in digital video systems. Y represents the brightness component, Cb represents the blue chroma component and Cr represents the red chroma component of the image. The face detection algorithm that we are using focuses on differentiating faces from the rest of the image based on the skin tone. Henceforth, we only use the Cb and Cr components in our thresholding step. It maybe possible to incorporate the Y component to allow the thresholding to identify faces in more situations. But again, we did not want to make face detection the focus of our research and hence we have just simply adopted Group 6's algorithm. The conversion from RGB to YCbCr is simply done through the Matlab command rgb2ycbcr(). We then use the YCbCr color space to create a binary image. The binary image is high for pixels in the range of 100 < Cb < 133 and 140 < Cr < 165.

2: Morphological Opening and Closing

Before explaining the process of opening and closing, we have to understand erosion and dilation. Both erosion and dilation are performed on binary images.

Erosion is simply the process of removing noise and other artifacts by moving a structuring element throughout the image while finding overlaps between the structuring element and high pixels. In each overlap, the central pixel stays high and all other pixels become low [5].

Dilation is the opposite process of erosion. Dilation is performed after erosion in order to attempt to fill in any holes that could have been created by setting high pixels to low pixels in the erosion process. In dilation, we assert high pixels to the entire area of the structuring element for each high pixel in the image; all other pixels become low [5].

When we apply image opening, we erode and dilate the image by a structuring element of size 9 by 9 pixels. This basically means that in image opening, we remove artifacts that are smaller than 9 by 9 pixels. In image closing, we dilate and erode the image by a structuring element of size 7 by 7 pixels. Image closing is used to fill holes, like eyes and lips, which are usually different color than regular skin [5]. Notice that in image opening, we first erode while in image closing, we first dilate. Besides the different structuring elements, those are the key differences between image opening and closing. Both structuring elements used were found by trial and error from Group 6 from Fall 2007.

3: Blob Detection and Rejection

At this point, the noise (if present originally) should be eliminated and we should be left with a series of blobs.

We now need to interpret which blob represents a face and which blob is simply a false detection. By using the method regionprops() in Matlab, we can easily determine the width, height and area of each blob. We then reject all blobs which meet any of the following properties: width < 20, width > 80, height < 25, height > 150. These blobs are rejected based on the idea that they are probably too small to be a face or too large to be a face. Lastly, on the remaining blobs, we examine their width height ratios. If their width-height ratio is between 0.5 and 0.9, then we continue processing; otherwise, that blob is rejected. If something passes the width-height ratio, we then look at the density of the blob, which is defined as the area/(width*height). If the density is less than 0.5, we reject the blob; otherwise, we have successfully detected a face.

Limitations

This face detection algorithm is very limited. Because it is based on color thresholding, it's highly dependent on the lighting conditions of the given frame,. With that in mind, this algorithm will not work for every environment. We have noticed that it works best for indoors environments and often fails in outdoor scenarios. In alternative method for face detection would be to do a feature-based face detection algorithm. Feature-based face detection is independent of the color of the subject; hence, it would work in any lighting conditions. We originally looked at feature-based algorithms and decided not to implement them because they require a very large training set in order for the algorithm to work correctly. To demonstrate a successful face detection, Figure 3.4 and Figure 3.5 each show faces of different sizes detected from different videos. In both cases, the region determined to be a face spreads slightly outside the actual boundaries of the face due to the coloring of the subject's shirt.

Figure 3.4: Input frame and resulting face detection output (with primitive edge detection shown as a guide)

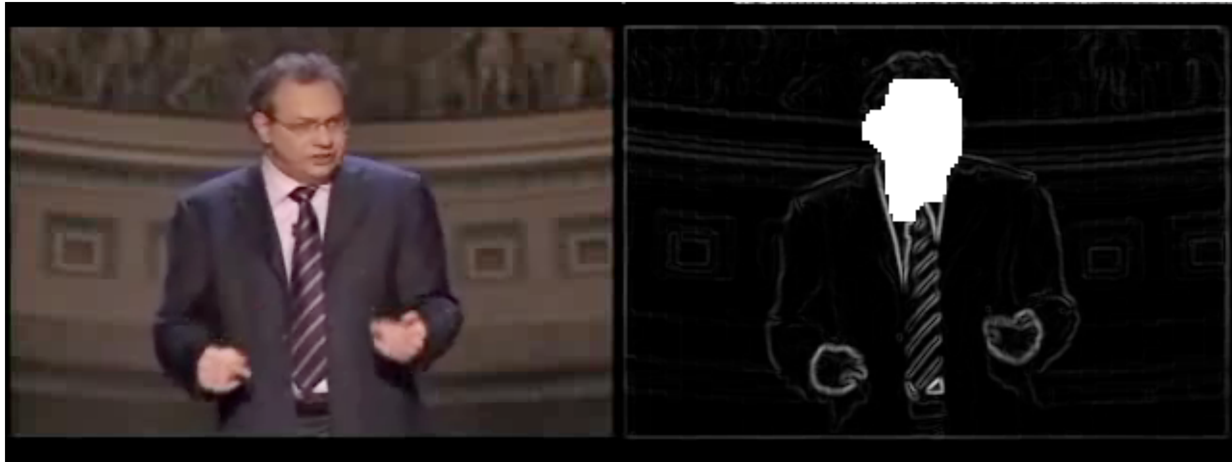

Figure 3.5: Input frame and resulting face detection output (with primitive edge detection shown as a guide)

Motion Detection

The motion algorithm used in the CAVES project is a block based image thresholding algorithm implemented by Liu et al. [2] The goal of motion scoring in video expansion is to give importance to regions of high motion. Assuming that these regions are ones of high interest, the purpose of detecting movement is to give a better viewing experience for expanded videos. The scores of the motion detector is the third contributor to the prominence test matrix.

The motion algorithm used is a proposed way to provide illumination-independent change detection. The algorithm discussion will be broken up into two sections. The first part will discuss special values known as circular shift moments and show how they provide change detection in a noise-free case. The second section will apply a new decision rule on top of this to cope with the effects of noise.

1: Change Detection with Circular Shift Moments (CSM)

For the calculations in this algorithm, the 24bpp images are averaged to give an 8bpp gray scale image. The mapping from RGB to the 8bpp gray scale image is simply an average of the red, green, and blue. From this, the image is partitioned into NxN pixel square blocks. For our implementation, we choose N to be 10 for our 320x240 images. This results in 768 possible areas of motion within one given frame. Choosing smaller values of N resulted in significantly slower computation times, and less accurate decision results by the motion detector. For example, using an N value of 5 results in 3072 possible areas of motion with one frame. For such small values of N, the image sequences are too noisy and result in almost consistent motion detection when it is only noise. For values larger than 10, we found that the motion algorithm would not return specific enough results.

For example, choosing an N value of 20 results in only 192 areas of motion. Even though computation was faster, the square regions were to big to give an accurate estimation of motion regions. Choosing N to be 10, for our given size, resulted in the most promising results.

For every square area of interest, there is a predefined x-direction circular shift moment and a y-directional circular shift moment. Please refer to [2] for the equations. With these equations, the CSM-based change detection can then be applied to detect motion. The steps are as follows. According to the equations given by [2], calculate both the x and y directional circular shift moments for every square block in a reference frame. Along with these calculations, decide upon a predetermined threshold. For every square block area of interest of the kth frame in an image sequence, calculate its x and y directional circular shift moments for every square block. Finally, claim that a change occurs in a square block if the absolute value difference of either the x or y directional circular shift moments between the kth frame and the reference frame is greater than the threshold. Otherwise, there is no motion in that square. Rinse and repeat by re-initializing the reference frame, and move on with the next one.

2: Change Detection with CSM to Cope with Noise

The proposed method in [2] to deal with noise in the video is quite simple. Consider a situation where nothing in the actual video content changes, but pixel values change as a result of noise. Under a noise-corrupted situation, the gray level at a certain position will then be the gray level of that same position in the previous frame with the addition of noise. We assume that this noise is additive white Gaussian noise (AWGN) and can be somewhat accurately encapsulated by its mean and variance. We assume the mean of the noise in our videos is zero with a certain variance. The variance is what will be calculated in order to cope with the noise. Furthermore, the noise is assumed to be independent between pixels.

The goal of these calculations is to intelligently change the threshold from Part 1 in order to properly cope with the noise better. Hypothetically in the noise-free case, the circular shift moments of a certain square block in two consecutive image frames must be identical provided that there is no content change. As we know, the effects of AWGN will make it so that these two CSM moments are different even though the scenes are the same.

The process is simple. For a given reference frame and a kth frame, determine the one square 10x10 region which exhibits the least change in its circular shift moments. Let the 'change' here be defined as the sum of the absolute value differences of both the x and y directional CSM moments. In order words, pick the one square in the entire frame which changes the least. Accumulate the gray scale levels in this area in both the reference

frame and the current frame and create a ratio which will be known as the variation factor. This number right here is then used to estimate the noise variance. The equations and detailed sequential steps are outlined thoroughly in [2]. A basic summary goes as follows. Find the NxN square in a frame which changes the least between two given frames. Assume that this change is due entirely to noise. Calculate the variance of the speculative AWGN and accommodate for the variance in the predetermined threshold. Any change will contribute to noise variance which will make the threshold more stringent. Thus, the final steps are exactly the same as the ones outlined in Part 1, except that the threshold is now more strict, or higher, to filter out noise.

Results

The motion algorithm is implemented entirely in Matlab. The results of the algorithm are good with room for improvement. Motion is detected in square blocks in areas where it should be. This is based on us viewing the videos and actually seeing what moves and then comparing it to the blocks which are detected. All character movement, body movement, and mobile objects are detected well. Results were less accurate during times of intense camera movement. During camera movement, many things such as the background changes a lot when it is in fact not "important" motion. Nonetheless, during the presence of motion, more weight is added in that particular area and is accounted for during the seam carving process. A detailed discussion on how the motion contributed to the results of the seam carving are discussed in Section 8. An example of motion detection on sequential frames is shown in Figure 3.6. In the figure, the two sequential input frames have just a few subtle visible differences, but the motion detection algorithm easily detects the changes in the position of the subject's lightsaber, arm, and head.

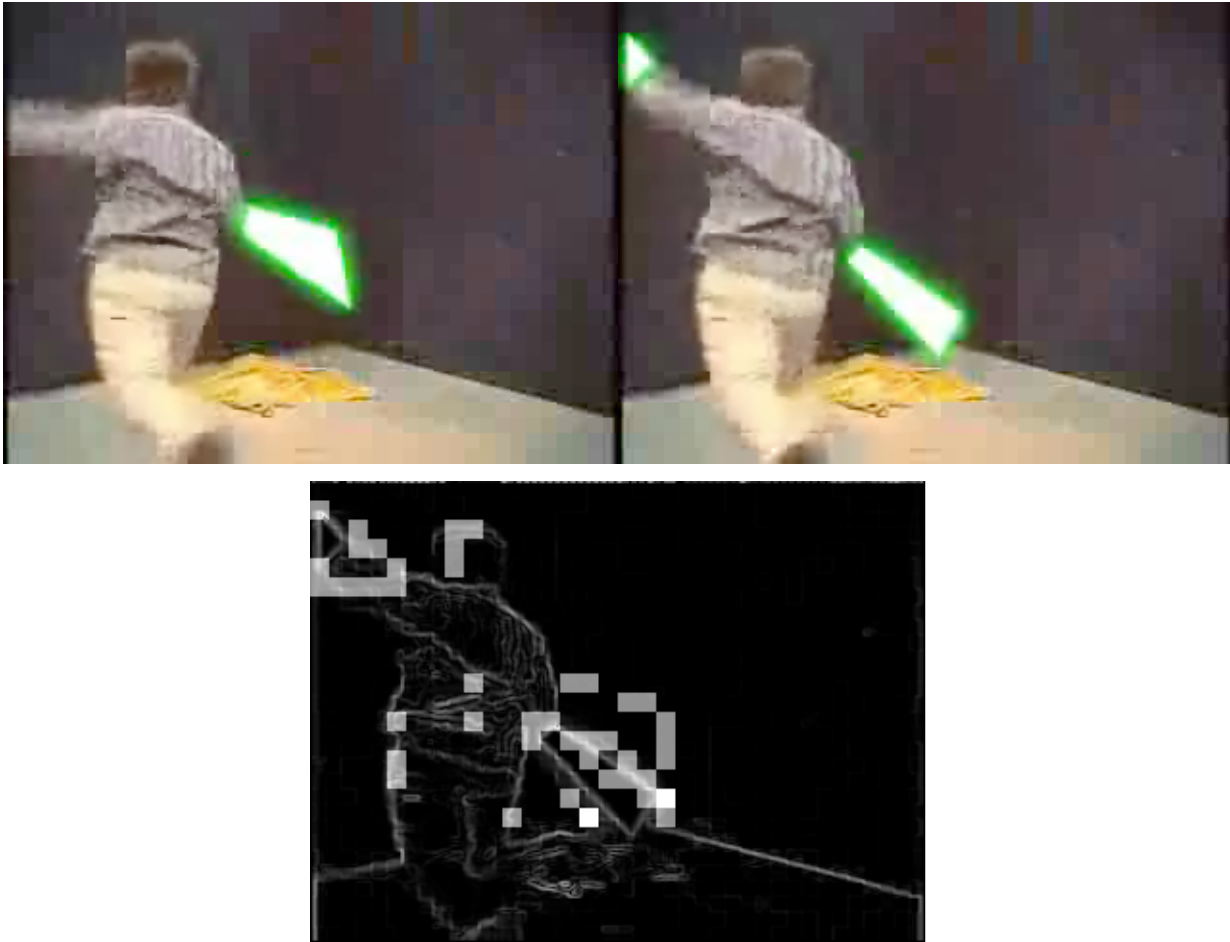

Figure 3.6: Two sequential input frames and the resulting motion detection output (with primitive edge detection shown as a guide).

Seam Carving

Seam carving uses the prominence scores to generate an energy map which shows the "total cost" of a seam at the bottom boundary (for a top-down energy approach). We can't simply use the prominence scores to determine which seams to add due to the fact that each prominence score is independent from each other. Hence, we can't make an educated decision as to where to start seams just from examining one row. This is the motivation behind creating the energy map.

Energy Map

The energy map is calculated by a top-down approach. This means that the total cost of adding a seam is found in the last (bottom) row of the frame. The energy map is created directly from the prominence scores. The very top row of the energy map is equivalent to the top row of the prominence map. Starting from the second row to

the last row, each pixel's value is equivalent to the sum of its prominence score and the maximum prominence score of the three adjacent pixels above it. This algorithm creates values in the last row which carry information from every other row. The nature of this algorithm is clearly best implemented by dynamic programming. Figure 3.7 shows an example energy map.

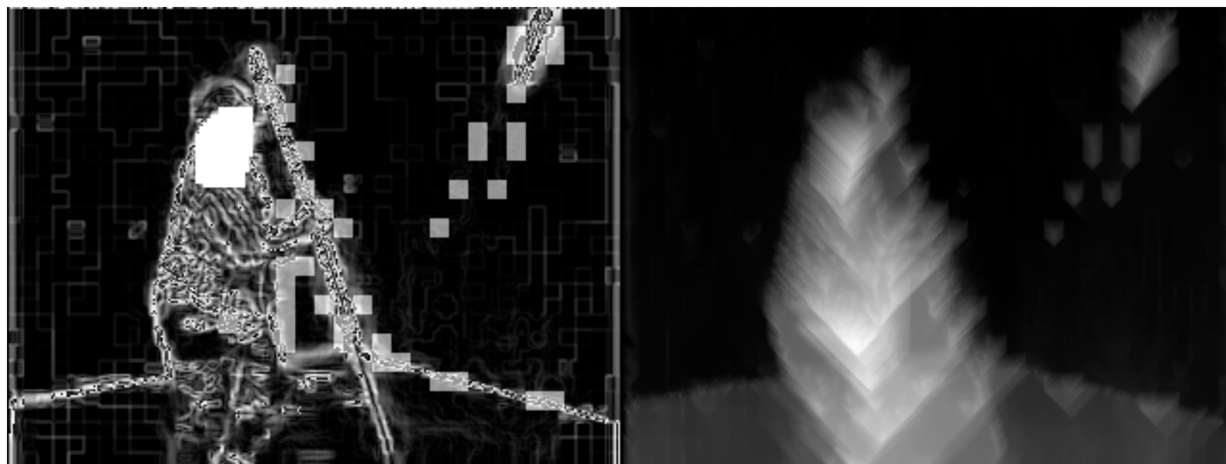

Figure 3.7: A prominence matrix with motion, gradient, and face visible (left) and its corresponding energy (right)

Seams

As mentioned earlier, we use the energy map to determine where to place seams. We simply choose the lowest 100 energy values in the bottom row of the energy map and make these our starting positions for each seam. This is different from seam removal. When seams are being removed, seams are removed one at a time and after each seam is removed, the energy map is recalculated. This is an extremely slow process. In this token, video expansion is much more suitable for implementation on a DSK. The reason why we choose the 100 lowest energy values at once rather than calculating one seam at a time, is to avoid artifacting. Considering calculating seams one at a time. The first seam would be calculated and then duplicated -- causing the image width to increase by 1 pixel. We then try to find the next seam. Because the addition of the first seam simply copies the pixels to the left of it, the energy matrix barely changes. Henceforth, when trying to find the second seam, the algorithm will approximately select the first seam again. This process would keep going until all 100 seams are chosen.

At the end, the viewer will notice that there would be very noticeable artifacting from expanding the same pixels by 100 times The artifacting problem is shown in Figure 3.8.

Figure 3.8: Seam carving artifacting for expansion [3]

Thus, by choosing all 100 seams at the same time, without altering the energy map, we are able to avoid this artifacting problem (as shown in Figure 3.9). After choosing the 100 starting positions, we simply backtrack the energy map to determine the entire seam. This results in seams that are both connective and monotonic [3]. This basically means that a seam can only have one pixel per row and each pixel of the seam must be adjacent to the seam's pixels in the row above and below it. This makes sense, as we want to uniformly change the width of all rows.

Figure 3.9: Proper seam carving for expansion [3]

This method that we described creates seams solely dependent on the energy map. In order to eliminate the jerky artifact and lessen the wavy artifact, we add new restraints on frames. We first compute the first frame of a video as described above. But then for the frames thereafter, we add a new restraint based on the previous

frame. For example, for the second frame, we start off with having the energy map for the second frame and the seams used in the first frame. We then add a restraint for calculating the seams in the second frame. The seams in the second frame must be within 3 pixels to the right or 3 pixels to the left of the seams from the first frame. The value, 3 pixels, was found from experimentation and it gave the best qualitative results. After restraining the seams in the second frame with this 6 pixel window, we then allow the seams to change depending on the energy of the 2nd frame. By restricting the seams of the current frame by the previous frame, we create a temporal dependancy which completely eliminates the jerky artifact.

Problems

By creating a limitation on how much seams can change between two frames, we encounter problems when important regions of the video move fast between frames. For instance, if there is a person in the right side of frame 1 and then he moves progressively to the left side up until frame 10, we often see artifacting. This occurs because the seams are not able to move as fast as the high energy region is moving. When this occurs, the high energy content becomes distorted because it runs into the seams. This can possibly be avoided by not fixing range of the seam's movability window. In future work, we can potentially content-aware calculate the range based on the total energy of the seam. Future work and current problems are discussed in depth in Section 8.

One method we use to ameliorate this problem is the process of keyframing. A keyframe is a frame that has no additional restraints beyond the energy map for seam calculations. If we content-awarely keyframe throughout the video, it gives the seams a chance to fully "reset;" essentially it allows the seams to move to their proper locations disregarding temporal dependancy. This fixes the issue of a person running into seams, as we can just keyframe that instance. However, determining when to keyframe is a task in itself. We naively keyframe when there is a large energy change between frames. When the energy ratio between the current frame and the previous frame is either 200% or 50%, we make the current frame a keyframe. The values 200% and 50% were found by examining the energy ratios of a high-motion 50 frame video. By keyframing only at these large changes, we avoid the problem of keyframing successive seams, as this would break our temporal dependancy.

To further build on using energy ratios, we change the seam's movability window depending on a large [small] enough energy ratio. If the energy ratio is from 126% to 200% or from 50% to 74%, we change the window from 6 pixels to 50 pixels. The idea is that we would like to keyframe at this change in energy ratio, but the change in this energy ratio occurs too many times throughout the video to allow keyframing. Hence, we instead allow more movement for the seams so that they're able to "jump" more. This enhances the wavy artifact but it allows for seams to keep up with moving high important energy areas. Again, these percent ratios were found from the same high-motion 50 frame video.

4. Brief System Overview

The following section will provide a brief system overview of the CAVES project. For more details behind the processing and data transfer between the PC and the DSK please refer to Section 7: Processing, Speeds and Data Rates.

From start to finish, we process an input video on the local machine and output its 420x240 24bpp representation. The whole process can be broken down sequentially into steps with different PC and DSK responsibilities. Figure 4.1 shows a basic representation of these steps. The details of the data flow are described further in this section.

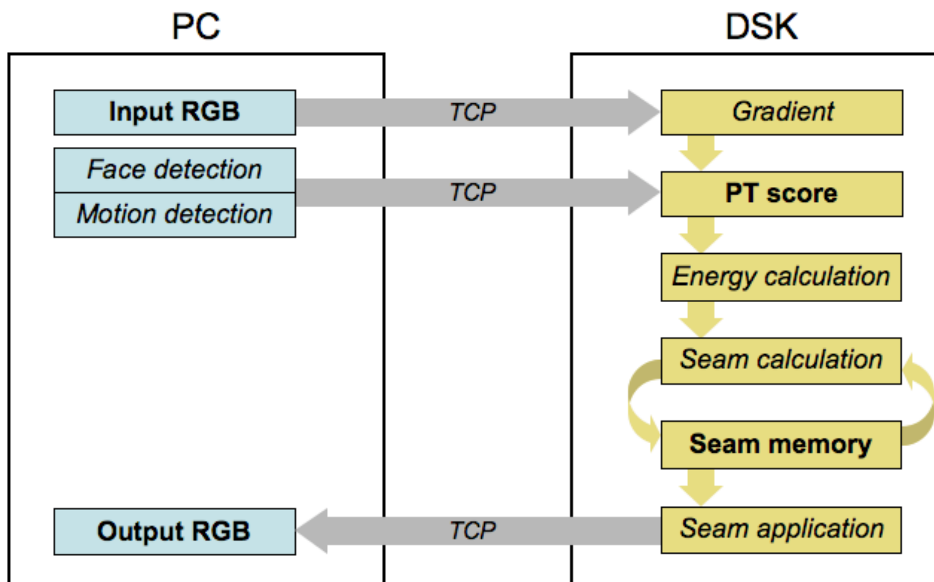

Figure 4.1: Simplified data flow with primary algorithm steps shown

In our system layout, different programs are used in different parts of the data flow (Figure 4.2). Matlab is used for processing on the PC, but a separate network server, written in C, runs on the PC also and handles all communication with the DSK. Both Matlab and the network server read from and write to a common set of image and video files on the PC, but the timing is asynchronous. In the data flow figures in this section, the network server component is excluded from the figures for for clarity, but it is still used for data transfer.

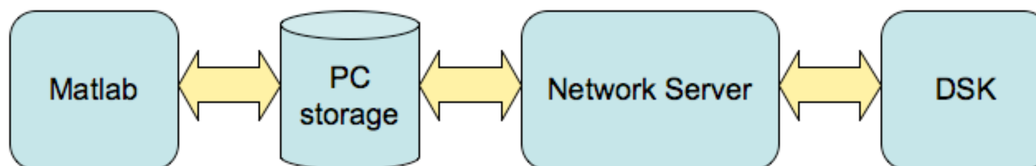

Figure 4.2: Core components used in data transfer and processing

1) PC: The computer takes the input video and does preliminary processing entirely in Matlab. This includes breaking the video down into separate frames, rescaling, recoloring, and preparation for transfer to the DSK. In addition to this, Matlab will perform preprocessing of every frame of the video by running them through the face detection and motion detection algorithms. This will be accumulated into a "partial PT" which will be stored on the PC and sent to the DSK later. (Figure 4.3)

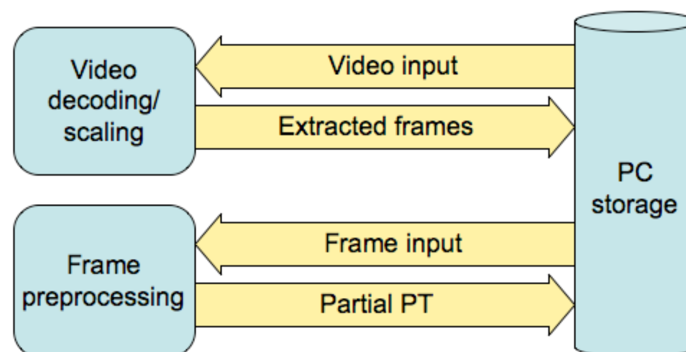

Figure 4.3: Matlab preprocessing

3) DSK: The DSK takes the frame along with the partial PT matrix and computes the gradient. After adding the gradient matrix to the partial PT matrix, the DSK will now hold the final prominence matrix for that particular frame. The DSK computes the energy matrix based on this prominence matrix, and feeds the energy matrix into the seam calculations. All the seams are then routed for this frame. Remember that CAVES calculates seams based on the previous frame's seams along with the current frame's energy. The DSK expands the frame with these calculated seams and sends the expanded frame back to the PC. For debugging and viewing purposes, we also send the expanded frame with red seams, a full PT matrix, and the resulting energy map back to the PC. (Figure 4.4).

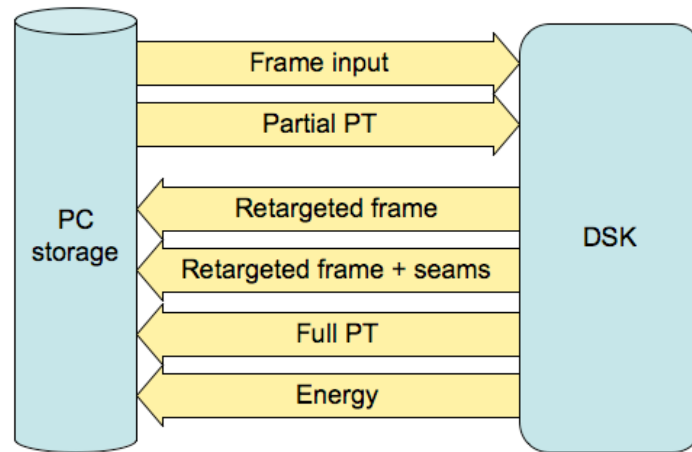

Figure 4.4: DSK processing

To deliver the necessary data to the DSK, a separate PC network server program is used, as described earlier. This program is very simple and performs only network, file-handling, and format-conversion operations. This is not shown in the simplified data flow in Figure 4.4, but is present in the system. The Matlab preprocessing is performed relatively fast (about 5 frames per second) and each frame is saved to a file on the PC's local drive. The DSK processing is started simultaneously and is performed asynchronously from the Matlab processing. The PC network server reads each file recently created by Matlab, converts it to RGB, and sends it to the DSK. When the DSK finishes processing each frame, the PC network server receives the RGB output, converts it to BMP, and saves it on the local PC drive. The expanded frame, the expanded frame with red seams, the final PT matrix, and the energy matrix are each saved as independent BMP files for each frame in a designated folder structure on the PC. Figure 4.5 shows the generalized layout of data delivery to and from the DSK.

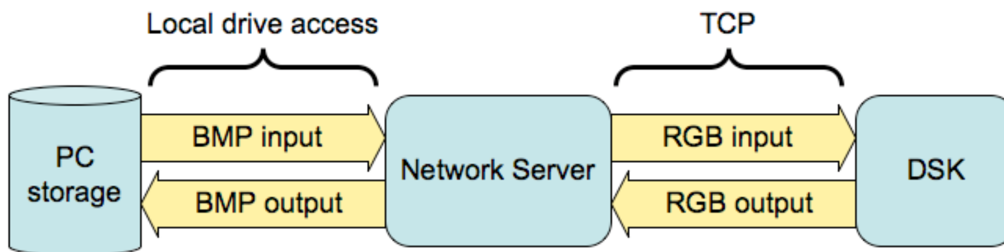

Figure 4.5: Detail of PC-DSK data flow for each data segment

4) PC: The computer receives everything from Step 3 and reassembles it into a video while saving it to the local machine. All the information is viewable on the GUI. (Figure 4.6)

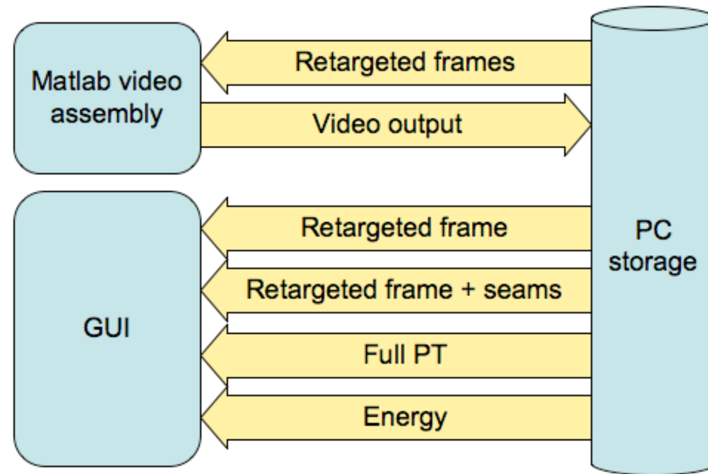

Figure 4.6: Matlab post-processing and GUI

This is a quick summary of the basic system layout of our project. Again, please note that an in-depth analysis of this data flow is included in Section 7: Processing, Speeds and Data Rates.

5. Graphical User Interface

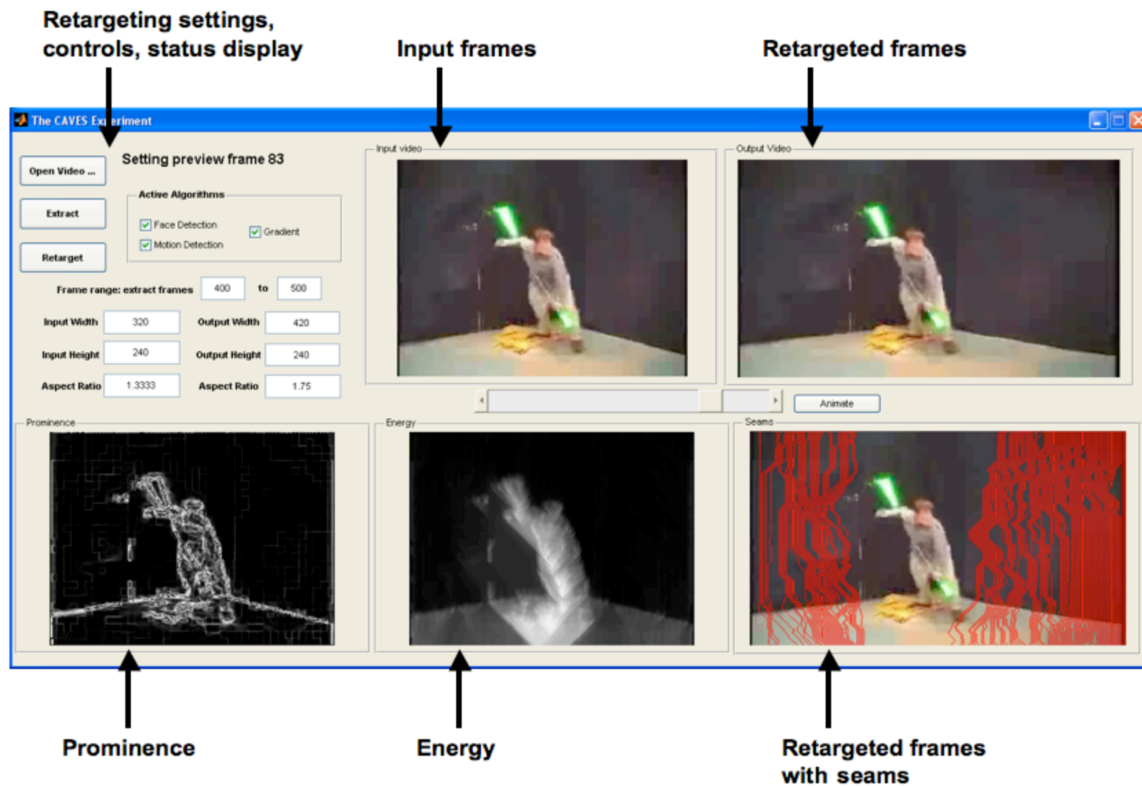

Figure 5.1: GUI overview

For the CAVES project, a graphical user interface was created to allow users to easily use the video resizing algorithm and view the results (Figure 5.1). By displaying a variety of intermediate steps and allowing various setting changes, this GUI was extremely helpful in our debugging process and allows us to analyze various steps in the algorithm. It is also very convenient for demonstrating the behavior of the algorithm in a practical way. The user interface is divided into two main sections: retargetting controls and frame viewing.

Retargetting Controls and Settings:

The retargetting controls and settings are located in the top left of the GUI. (Figure 5.2)

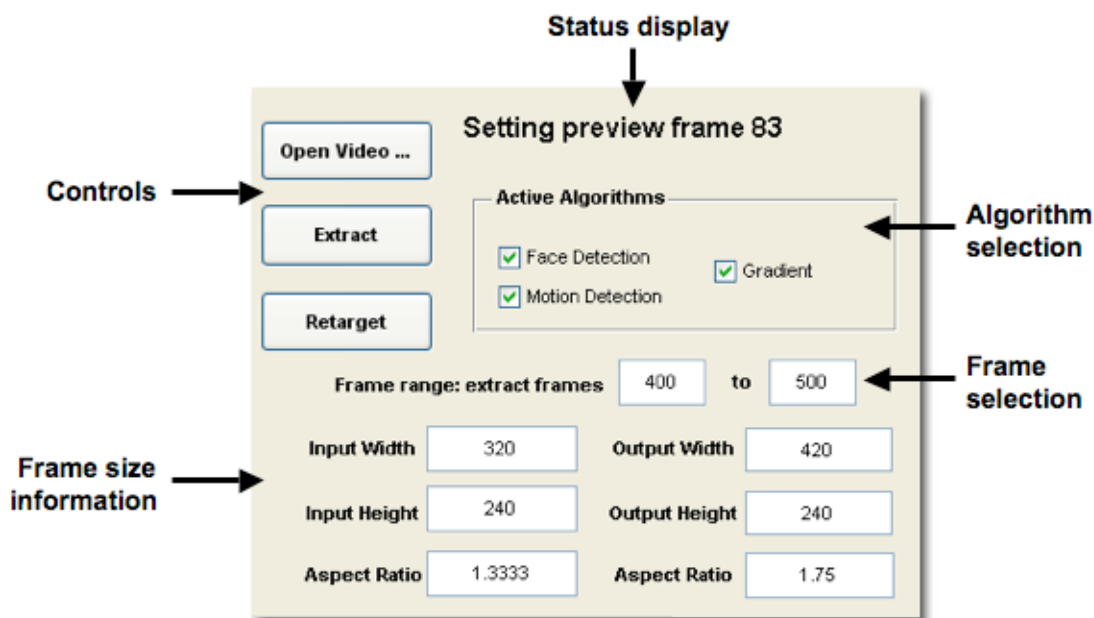

Figure 5.2: Retargeting controls and settings

Control Buttons:

The first step in using the user interface is to use the Open Video button to open up a video with Matlab. These videos are local to the machine and will be read into variables on the Matlab workspace. The second step is to use Extract to determine which sections of the video are to be processed and to perform all the preprocessing required so that the sequence can be sent over to the DSK. The exact processing and details of this are discussed in Section 7: Processing, Speeds, and Data Rates. Finally, the Retarget button will establish the connection between the PC and the DSK hardware unit in order to process our video.

Active Algorithms:

The PT matrix that we calculate per frame is dependent on the gradient of the frame along with motion and face detection. As a default, all three of these attributes are enabled in order to contribute to our prominence

scoring as they are all important. We have added a feature to allow the user to specifically choose among these contributes for the PT matrix calculations. This enables us to analyze how well individual parts are working.

Frame Range:

The user is allowed to input the specific range of frames in the video that he or she would like to retarget. This is useful for debugging as we sometimes tend to analyze particular sequences in videos. This is also very helpful for the viewing experience as users may only want to process certain sections of the video.

I/O Height, Weight, and Aspect Ratio:

These are not writable by the user and are used to merely display the height, weight, and aspect ratio of the input and output video. We have hard coded the input video to be 320x240 pixels and the output video to be 420x240 pixels. This is shown on the GUI to make the user aware of the changes we are making to the video sequence. If the input video does not have an aspect ration of 4:3, its converted size will be smaller than 320x240. The correct converted input dimensions and corresponding output dimensions with 100 seams are displayed, preserving the original aspect ratio of the video during resizing. However, we did encounter some problems with retargeting behavior when using input videos that were not originally 4:3. The problems were caused by various discrepancies between the hard-coded 320x240 dimensions and the actual dimensions of the resized input. Some debugging would be necessary to fix these problems, but we were not particularly concerned with these cases because most of the videos we used for testing and demonstration were 4:3. In addition, we did not have time to make the input or output dimension settings user-customizable. Future work in this area should allow the user to specify arbitrary input and output aspect ratios.

Frame View:

Input Video:

We display the video sequence of our input video. It is important to note that all video sequences and rescaled and recolored to 320x240 24bpp.

Output Video:

This is the final product that CAVES outputs. Our final video is 420x240 24bpp and is the retargetted version of the input video to its left.

Prominence:

For the complete sequence of input frames, we display the prominence matrix of each frame. Remember that the prominence matrix consists of all the checked attributes in the Active Algorithms box. This includes the

gradient, face, and motion by default but may be changed accordingly. The resulting prominence can then be viewed.

Energy:

For the complete sequence of input frames, we display the energy map which is calculated with respect to the prominence. This is useful to see how energy changes in different sections of a frame and between different frames. Changing properties in the Active Algorithms box also enables us to analyze how our various algorithms contribute to the energy matrix as well. The energy matrix is thus used to route seams.

Seams:

For the complete sequence of input frames, we display the retargetting video at its new aspect ratio while including the seams drawn in red. This allows the user to see exactly how each of the seams were routed with respect to the energy matrix.

Scroll Bar / Animate:

In the middle of the GUI, there is a scroll bar and an animate button. The scroll bar scrolls between the frames of the video sequence while the animate button runs through the frames and plays it as a movie. This is essential to see exactly how our algorithm is performing on a video sequence.

Note: In Matlab, behind the scenes, we are not playing a movie. The GUI is stepping through all the frames sequentially. The CAVES program outputs the rescaled input video along with the final output video sequence in a video formatted file on the local machine. The prominence, energy, and seams are outputted as a sequence of images and are not reconstructed into a movie file. These are all outputted nicely into a directory tree.

7. Processing, Speeds, and Data Rates

Video and Image Formats

We chose to use an input video size of 320x240 pixels and an output video size of 420x240 pixels for the system. The input size was chosen because it is exactly a 4:3 aspect ratio and because it is a standardized display resolution, known as Quarter VGA. This is a common size for mobile video displays and is a popular resolution for videos found on the web, including those on the YouTube website. The output size was chosen because it is

very close to 16:9 and allows exactly 100 columns to be added to the video frame. A true 16:9 frame would require approximately 426.67 columns, but this is not an integer and there is no standardized widescreen frame size defined at a height of 240 pixels. For the most part, the code of the system is generalized to allow the frame size to be changed relatively easily if necessary. Using the system with larger standardized resolutions (e.g., 640x480) would require significantly longer processing and network transfer times. We decided that our default size of 320x240 was large enough to visibly demonstrate the results of the seam carving algorithm in a practical way while minimizing overall processing and transfer times.

The PC decoding of the input video is performed entirely in Matlab. Any video format and codec supported by Matlab can be processed. When the video is decoded, its frames are resized to 320x240, which is a 4:3 aspect ratio. If the input video is not 4:3, its existing aspect ratio is preserved and blank data is added around the content so that it fits completely inside a 320x240 frame without aspect ratio distortion. Matlab then saves these frames individually as BMP image files with 24bpp color locally on the PC. We decided to use BMP files because they do not require decompression and we are not concerned with storage space on the PC. A BMP file with 24bpp contains 8 bits each for red, green, and blue color information in each pixel. It also contains header data of variable length and data padding between rows of the image for byte alignment purposes. In order to minimize memory access on the DSK, to eliminate unnecessary format parsing on the DSK, and to reduce network transfer time, we transform the BMP files to a simpler RGB format on the PC before each frame is sent. This RGB format has no header and no internal padding. The DSK performs all processing on 24bpp RGB data (in the case of the input frame) or 8bpp grayscale data (in the case of the PT). When the DSK processing is complete, the RGB output is transferred to the PC. The PC saves this output for each frame as a 420x240 24bpp BMP file by creating the appropriate header information and adding data padding between rows of the image as necessary.

We use Matlab's built-in video processing tools to create AVI files from the bitmap images. The GUI allows the user to select a specific range of frames to retarget from the input video. Matlab creates a duplicate video of only the selected range of frames from the input so that it can be easily compared to the output. When the DSK has finished processing all frames, Matlab converts the output bitmaps into an output video. We chose to use uncompressed video output from Matlab to eliminate distortion caused by compression. We are not concerned with storage space on the PC, so this was not a problem. If the output videos need to be saved permanently or moved, or if storage space is an issue, the compression used by Matlab can easily be changed by adjusting a single parameter in Matlab's movie2avi function call.

Color Depth

Our initial implementation of seam carving used an 8bpp input frame. This color depth was chosen to minimize DSK processing time due to fewer memory accesses. To change the color depth of an input video, we added a simple pixel-by-pixel conversion to our PC code which saved 3 bits of red data, 3 bits of green data, and 2 bits of blue data, in accordance with the standard 8-bit truecolor scheme. In our initial testing, we found that videos originally encoded with higher color depth were converted correctly but experienced a noticeable reduction in quality. This reduction is inevitable because of the smaller number of colors that can be represented with 8bpp compared to other common color depths such as 16bpp or 24bpp. We eventually decided to use 24bpp rather than 8bpp in the DSK processing to eliminate any significant reduction in quality. When retargeting videos that are already significantly compressed, the use of 24bpp is not visibly different than 8bpp. In the future, it would be possible to add support for both formats and allow the user to select high-quality or low-quality conversion as one of the settings. This would require additional code support on the DSK, primarily in the gradient calculation and the applySeams() function, to handle both formats. The use of 24bpp not only requires additional memory accesses on the DSK but also requires additional data to be sent over the network. Ultimately, we decided that the higher quality of the video outweighed the cost of longer processing time and network transfer time.

DSK Memory and Paging

Our retargeting system operates sequentially on one frame at a time. The only information that needs to be saved on the DSK from one frame to the next is the location of the seams. The functions that change the seam locations (getSeams() and getInitialSeams()) operate in-place and simply overwrite the seams of the past frame with the seams of the current frame. Note that seams are formed and updated from left to right and may not overlap. The only temporal constraint of a seam is based on its own location in the past frame, not the location of any other seams. The seam to its left in the current frame provides a spatial constraint. Thus, no extra seam data has to be saved.

Besides the seam data, all other data is overwritten by each subsequent set of frame data. We defined a simple struct called pixel containing three char fields for red, green, and blue. This was used to make the coding more intuitive for 24bpp frames. The following table details the storage requirements for the various sets of data on the DSK.

Data	Type	Dimensions	Size in bytes
Input frame	pixel (3 bytes)	320x240	230,400
PT	char (1 byte)	320x240	76,800
Gradient	char (1 byte)	320x240	76,800
Energy	float (4 bytes)	320x240	307,200
Output frame	pixel (3 bytes)	420x240	302,400
Seams	short (2 bytes)	100x240	48,000
(Total)			1,041,600

Approximately 1 MB of data is stored on the DSK. All of these data fields are stored in external memory. Processing is performed directly on the data in external memory and no paging is used. The L2 cache is set to 32KB. In an early implementation of our algorithm, we fully implemented paging for all processing. At the time, we were using 8bpp color and were only performing seam carving functions on the DSK, not prominence or energy functions. DMA was used for all page transfers. The maximum size of the memory workspace that could fit on-chip was approximately 120 KB with L2 cache turned off. In this implementation, we did not notice a perceptible improvement in overall system speed compared to the same implementation without paging. Later, we upgraded the frame color depth to 24bpp and simultaneously added gradient and energy calculations to the DSK. These changes required adjustmets to the paging mechanism. Due to the added code complexity of paging, the need to handle extra boundary cases in all of the processing functions, and the minimal difference in observed processing time, we chose to eliminate paging in our final implementation. After more detailed testing, we found that network transfer time and various inefficiencies in the PC code (which were later fixed) were contributing to a slow overall system speed. It is reasonable to assume that paging would help memory access time in our final algorithm implementation, but we could not re-implement paging due to time constraints.

One simple oversight of our memory management was that very little on-chip data is used except for automatic variables in functions and a few small permanent arrays. Given that the remaining on-chip memory available is close to 100KB, one of the data fields could be easily moved from external memory to internal memory simply by chaging one line of code. The seam data is a good candidate for this becuase of its small size (48KB) and its consistent use in multiple functions. We did not realize this until after performing all of our timing measurements, so the timing results in this report are based on fully external memory allocation. Anyone who uses our code in the future can easily make this change to on-chip memory if paging is not desired.

Network

For each frame, the data sent from the PC to the DSK is comprised of the scaled input frame and the partial PT matrix, which is comprised of weighted face detection and motion detection scores. Both sets of data are 320x240 pixels. The input frame is 24bpp and the PT is 8bpp grayscale. The resulting size of the input frame is 320 x 240 x 3 = 230,400 bytes and the size of the PT is 320 x 240 x 1 = 76,800. The total amount of data to transfer from the PC to the DSK per frame is 307,200 bytes. Profiling the DSK code for network receiving caused the data transfer to fail, so we could not determine this time exactly. Based on the 2.5 MB/s inbound speed limit of the DSK, we estimate that this transfer takes 123ms. The following table details the time requirements.

Data	Size in bytes	Cycles	Time
Input frame	320 x 240 x 3 = 230,400	20,700,000 estimated	92ms estimated
Partial PT	320 x 240 x 1 = 76,800	6,975,000 estimated	31ms estimated
(Total)	307,200	27,675,000 estimated	123ms estimated

The data sent from the DSK to the PC is comprised of four output images. The primary output is the expanded video frame itself, which is 420x240 and 24bpp. The other outputs are the final PT matrix, the energy matrix, and the expanded frame with visible seams drawn in red. The PT and the energy are 320x240 because they are based on the input image. The expanded frame with visible seams is 420x240. All output data is sent as 24bpp. This makes the process of converting from RGB to BMP consistent for all output images. In the future, it would be possible to reduce network transfer time simply by sending the final PT matrix and the energy matrix as 8bpp, since the data is grayscale. This would also require additional grayscale-to-BMP conversion code to be written on the PC with appropriate adjustments to the BMP header data to create an 8bpp BMP file. Alternatively, the PC could convert the grayscale data to 24bpp RGB and subsequently convert it to 24bpp BMP. We made the initial decision to make all output data 24bpp for consistency, not knowing how this added data transfer would impact out overall system speed. Ultimately, network transfer time was observed to be more significant than we expected, but due to time constraints we did not have the opportunity to create the additional code necessary to handle 8bpp output conversion. When sending data from the DSK to the PC, we were able to use DSK code profiling to perform precise measurements. These measurements essentially confirmed the 10MB/s outbound speed limit of the DSK. The table below shows the DSK-to-PC data sizes and time requirements.

Data	Size in bytes	Cycles	Time
Output frame	420 x 240 x 3 = 302,400	6,350,700 measured	28.2ms measured
Output frame + seams	420 x 240 x 3 = 302,400	6,350,700 measured	28.2ms measured
Final PT	320 x 240 x 3 = 230,400	4,833,900 measured	21.5ms measured
Energy	320 x 240 x 3 = 230,400	4,833,900 measured	21.5ms measured
(Total)	1,065,600	22,369,200 measured	99.4ms measured

The total data size is just over 1MB, and the total time is about 100ms. This matches the predicted 10MB/s. If the final PT and energy were sent as 8bpp instead, they would each require 76,800 bytes of data transfer, or about 7.7ms each. This would reduce the total outbound transfer time to an estimated 72ms per frame, or about 72% of the present transfer time. Note that in a practical application of our system, such as a consumer-end video retargeting device, the only necessary output would be the actual output frame, which has only a 28ms transfer time. If the behavior of the algorithm was determined to be adequate, it would be relatively easy to add an option to the system that would allow it to send only the output frame to the PC. This would increase the overall speed of the system by eliminating information about how the algorithm is working. We did not add this option because we always wanted to have the additional information about the behavior of the algorithm available to us.

Matlab Processing

The full PT is made up of face detection, motion detection, and the image gradient. We implemented the face detection and motion detection algorithms in Matlab as part of the pre-processing of each frame. Since these two detection algorithms contribute a part of the PT score for each pixel, we call the output a "partial PT." When the retargeting process is started, the array of input frames selected by the user is already saved locally in Matlab's memory. The Matlab code is then responsible for performing the face detection and motion detection for each frame and saving the resulting data in a file on the PC hard drive. In addition to the existing BMP file, each frame now has a corresponding partial PT file. The data is an 8bpp grayscale representation of the partial PT. These files will later be read by the network server program and sent to the DSK as needed.

In our measurements, we found that Matlab is able to perform preprocessing at a rate of about 5 frames per second. This speed includes the time required for face detection and motion detection and for saving the file to the hard drive. We did not perform any significant optimizations by hand to the Matlab algorithms since their performance was already adequate on the PC.

After the DSK has finished processing all frames, Matlab reads all BMP files of expanded frames and assembles them into a Matlab-native video structure. The final expanded video is saved to the local drive with the built-in movie2avi() function. Because Matlab can read and assemble the expanded frames relatively fast, we designed the system to wait until all expanded frames were available before starting to read them. This could easily be changed such that, after it completes preprocessing all frames, Matlab starts to asynchronously read the expanded frames while the DSK is still processing. In this scenario, after the final frame is expanded by the DSK, the time required to finish assembling the video in Matlab would be significantly reduced. We believed this issue was relatively unimportant because, for most videos, the total video assembly time is small. For the cases in which this time is significant, it is still relatively small compared to the overall system processing time.

DSK Processing

The processing of each frame on the DSK is performed in distinct stages. The first stage is to complete the PT matrix. The face detection and motion detection results, which are calculated in Matlab and sent to the the DSK, are already saved in memory. The only remaining component of the PT is the gradient, which is calculated on the DSK and added to the existing PT. Once the full PT is formed, the energy is calculated top-down on the frame. Seam carving is based solely on the energy matrix, not the input frame itself. Each seam is drawn in the frame from the bottom up. This is logical because bitmap images are customarily stored on disk and in memory row-by-row starting with the bottom row of the image. This row ordering is preserved in the RGB representation of each frame on the DSK. The energy calculation is performed in the opposite direction of the seam carving algorithm so that the seam carving algorithm can "see ahead" of its current position. This ultimately allows us to use a greedy algorithm when carving seams because all the necessary information for directing the seam at each pixel is contained nearby in the energy matrix. The seams are then applied to the input frame to create the expanded frame.

In general, each major function in the DSK algorithms consists of a constant number of memory accesses for each pixel. Given the fact that the functions tend to take the same amount of time for different input frames and videos, it is reasonable to assume that a significant portion of each stage is limited by memory accesses, although the L2 cache increases the speed. The table below details the cycle counts that we measured for each stage of our algorithm. These cycle counts did vary by several thousand from frame to frame but were always very close.

Function	Cycles (approximate)	Time
gradient	1,850,000	8.2ms
getEnergy	3,540,000	15.7ms
getSeams	4,060,000	18.1ms
applySeams	2,820,000	12.5ms
(Total)	12,280,000	54.6ms

In some cases, the function getInitialSeams() is called in place of getSeams(). The only difference between these functions is the presence of temporal dependency in getSeams(). When getInitialSeams() is called, the algorithm is executed the same way but the perevious seam locations are not used. The cycle count we observed for getInitialSeams() was 3,950,000, or about 110,000 cycles less than getSeams(). This decrease is partially due to the smaller number of memory accesses because the previous seam data does not have to be read. Since the actual time difference is less than 0.5ms, we cosider the difference to be negligible and simply count the values listed in the above table as average-case times.

Performance Optimization

To improve DSK performance, we changed some of the compiler settings in addition to manually modifying our code to make it more optimal. Once our testing demonstrated that the algorithm behavior was correct, we turned off the memory alias safety compiler option. We expected that this would significantly help performance because of the heavy use of heap pointers in function calls and calculations. In reality, this adjustment only significantly increased the speed of the gradient calculation. Figure 7.1 shows the cycle comparison between the stages of the algorithm using alias safety on (the default setting) and off (which we used in our final implementation).

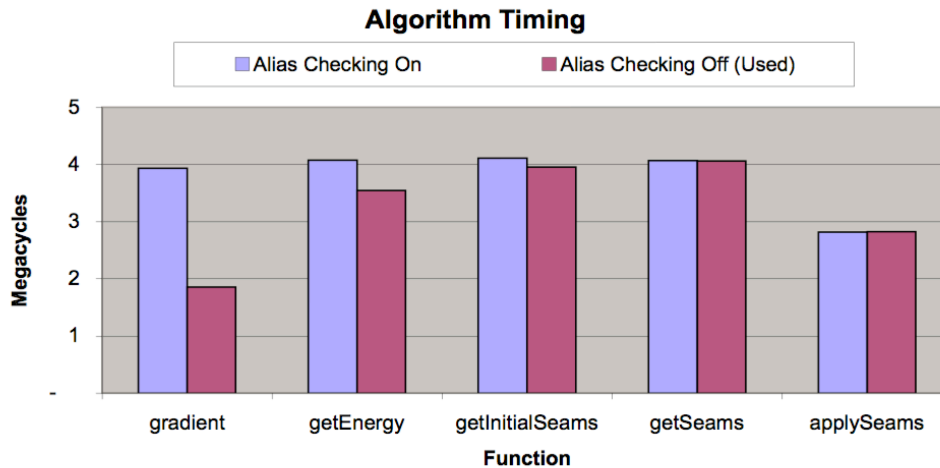

Figure 7.1: Cycle comparison with and without alias safety

Computation time is spread relatively evenly between the primary algorithm functions, so there is no obvious bottleneck in the algorithm. The slowest functions in cycle time are getSeams and getInitialSeams, but we expect this because of the complexity of the seam routing process and the need for many memory accesses (sometimes repeating) in the seam data and energy data. From our initial estimates, we can see that the compiler is able to optimize our functions significantly. For example, the number of external memory accesses (all occurring as 1-byte reads) in the gradient() function alone is over 1.2 million. If each byte was actually read from external memory at each corresponding lookup in the code, this would require nearly 7 million cycles for memory access alone, given an access time of 5.6 cycles. Alternatively, if each byte was read from internal memory with an access time of 1.5 cycles, memory accesses would require 1.8 million cycles. Since the entire optimized gradient() function requires just over 1.8 million cycles, we can assume that a combination of the L1 and L2 caches minimizes the external memory access time for this function. In getEnergy(), we counted about 2.4 million bytes of external memory accesses, suggesting 3.6 million cycles required if internal memory is used (1.5 cycles per byte), and the cycle time of the function is about 3.5 million. It is obvious from these results that both the L1 and L2 caches are used heavily in our processing. Similar cache optimizations occur for the other functions.

After verifying our algorithm behavior, we revised much of our source code by unrolling conditional statements and boundary cases from within important loops when possible. For the getEnergy() function, this change alone reduced the cycle count by over 50% from our initial implementation. We also eliminated unnecessary memory accesses in the getSeams() and getInitialSeams() functions by establishing a validitiy check for possible seam routes before comparing their energy measurements. Based on the compiler's suggestions, we added code speculation option (-mh2) and removed the debug option (-g) to maximize speed in the final tests. Optimization

level 3 was used for the project and no settings were changed on a per-file basis. We did not need to worry about negative effects from aggressive optimization because no interrupt handling is performed in our code.

As a final gauge of our DSK performance, we examined the assembly code generated by the compiler. From this we observed that nearly all of the primary processing loops in the algorithm were scheduled with 3 or 4 iterations in parallel. For those that were not scheduled in parallel, we were able to change the code further by hand, making minor adjustments to create additional parallelism. The only exception to this was the getEnergy() method, which contains many intermediate variables and calculations, and for which the compiler could not find a schedule with any iterations in parallel regardless of our adjustments. Still, we were satisfied with this result.

System Speed

Video and image conversion is not counted in our timing of the system since these times are relatively small and since we simply used the built-in Matlab functions as needed. For our purposes, we assumed that the set of input BMP files is already prepared and that the system only needs to output a BMP file for each frame without considering video reassembly.

The retargeting system runs at about 2.7 frames per second. The corresponding time per frame is about 370ms, which is broken down in detail in Figure 7.2. This includes the time for reading the input file from the local drive and saving the output file to the local drive. These file management times, in addition to the time required for BMP-to-RGB and RGB-to-BMP conversion, are counted in the PC overhead time. As described above, Matlab preprocessing of each frame occurs asynchronously and is not counted here. Network transfer times were more significant than we originally expected and contributed 220ms, or about 60% of the time per frame.

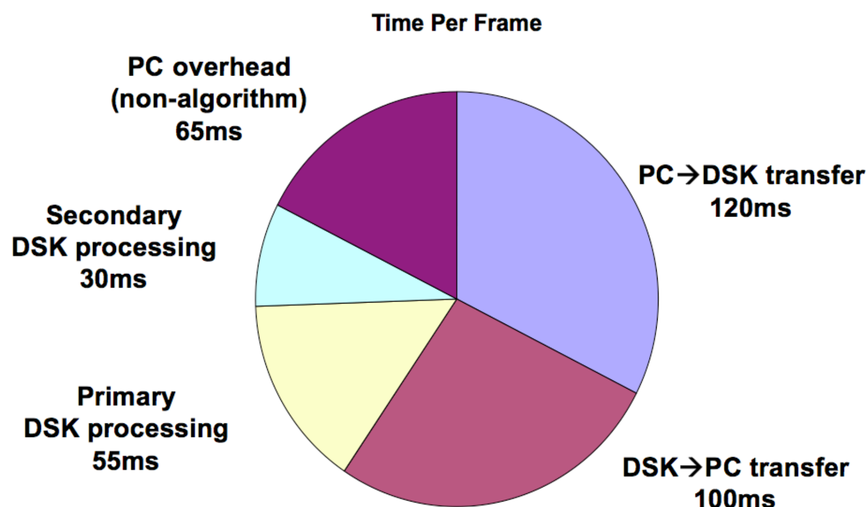

Figure 7.2: Average time for retargeting a single frame

DSK processing time is divided into two categories in Figure 7.2. Primary processing refers to the algorithm functions described in the DSK processing section earlier. These are the functions necessary to generate a single expanded output frame. In addition, the DSK performs some secondary operations to deliver the PT, energy, and visible-seam outputs to the PC. One design choice we made was to send all output data to the PC as 24bpp. This allowed us to use the same RGB-to-BMP conversion function for all outputs on the PC, which simplified our coding but resulted in slower network transfer, as described earlier. In addition, it requires extra time on the DSK to convert the PT matrix (of type char) and the energy matrix (of type float) to RGB. It would be possible to speed up the system by eliminating these DSK-side conversions, but we did not have time to make these changes. The other part of the secondary DSK processing is a second instance of the applySeams() function which draws the seams in red instead of expanding existing pixels in the frame. This is memory intensive and performs mostly identical memory copies as the original call to applySeams(). As a result, calling both functions in series as we do is inefficient, but we we wanted to keep the ability to call one or the other independently if necessary. The more efficient alternative would be to draw the seams over the existing output frame after it is sent to the PC, so that the frame does not have to be expanded again. Alternatively, the DSK could simply send the seam data and the PC could draw them on top of the expanded frame. Either of these approaches could be implemented in the future, but our approach is the most robust at the cost of speed.

Once again, all of the secondary DSK processing, a significant part of the PC overhead, and a significant part of the DSK-to-PC transfer could be eliminated in a practical application of the system because the only important result is the expanded frame itself. To estimate the speed increase in this scenario, we calculate that the new DSK-to-PC transfer time would be 28ms, and we know that the secondary DSK processing would be eliminated completely. The PC overhead would be approximately cut in half to 35ms conservatively. The resulting total time per frame would then be about 240ms, a 35% reduction from the current measured time. This would allow a system throughput of about 4 frames persecond. We did not test this scenario because we were more concerned with the behavior of the algorithm than with ideal-case speed upgrades.

8. Problems, Issues, and Future Work

ISSUE:

The output is "wavy" due to moving seams within unimportant regions.

DISCUSSION:

A variety of different videos displayed wavy-like behavior within seam regions after being processed by CAVES.

Within these unimportant regions, the seams tend to move around quite noticeably and have an adverse affect on the overall output video display experience. The main reason behind this is energy changes due to noise from the video itself. As we rescale input videos to our predetermined height and width, the video quality is not ideal. The existing compression in the videos we used for testing was also significant enough to cause amplified noise in our PT calculations. Subsequently, there is enough noise to cause the energy regions in seam areas to change such that the seams themselves are forced to move based on the nature of the algorithm. The seams change in order to follow the least energy paths, but these paths appear and disappear with the video noise. As a result, our seams tend to exhibit a wavy effect at times in unimportant regions.

A possible improvement could be to account for this video noise within the energy calculation algorithm itself or to filter out the video noise before processing the frame. This way, the energy matrix calculations could be more accurate and only display legitimate energy changes, not random fluctuations in the input video sequence.

ISSUE:

Seams can not move fast enough to accommodate for fast changes in video sequences such as character movement.

DISCUSSION:

When testing videos with CAVES, we were forced to set bounds to seam movement in order to provide temporal smoothness to the video sequences. Without this temporal dependency in seams, video sequences displayed choppy results as seams relocate themselves around the image too drastically. Such choppy behavior was unacceptable in terms of the final viewing experience thus making temporal dependency an essential technique. Nonetheless, this customization came at a cost. During periods of extreme movement during the video sequences, characters sometimes run into the seams and become distorted. Due to our restraints, there is no way for the seam to move out of the way fast enough. As expected, the algorithm does detect the energy change and seams shift accordingly within time. However, in this time the important regions of our video become distorted. There are a couple of potential fixes that we will discuss in order to better accommodate for this.

The first potential fix is to make the seam restraint range change with respect to motion. Currently, motion only adds to the prominence matrix which gets fed into our energy function. The added motion does have its own contribution as it introduces more energy into specific sections of movement, but it does not delegate the range of the seams. A new system could take into account motion specifically, and change the range of specific seam movement dynamically. Ideally, this would result in seams with a large range of motion in areas where the

algorithm detects large quantities of movement.

A second potential fix for this scenario is to force the redrawing of seams in these specific areas of interest. Let us consider an example where a character or object moves into an area densely populated by seams and thus becomes distorted. A proposed fix is to redraw the seams which cause this distortion. If a character moves into a group of seams, that group of seams will be recalculated and drawn in a different area of the frame away from the movement. Since this is only a small section of seams being recalculated at a time, this will not hinder the viewing experience. This is another potential improvement upon our current methods to drawn seams.

ISSUE:

During scenes with a high number of important regions, areas inevitably get distorted in certain areas.

DISCUSSION:

This is a problem with our implementation which has a tough time handling videos with many important regions. The potential fix behind this lies in the fact that we have chosen to hard code the number of seams and expand each seam by exactly one pixel. Aside from this approach, there are a variety of other methods which we have thought about, but did not implement in the CAVES project.

The potential solution to this lies in the fact that it is not necessary to expand a seam by only one pixel. Since we decided to expand our image by one hundred pixels, it was not required that we have to route one hundred specific seams. For example, fifty seams could have been chosen which could have been added twice instead of once. This still expands the image by one hundred pixels and draws less seams which could result in less distortion. Furthermore, smarter techniques could have been developed to expand low energy seams more often than higher energy seams. This way, the overall number of seams could have been reduced but the expansion resizing could still be the same. This would be another potential fix for the issue of having inevitable distortion in "busy" frames.

9. Final Work Schedule

We chose to create the GUI and the PC-DSK network infrastructure early in the project, in parallel with the research and design of our core algorithms, so that we could easily observe the behavior of the DSK code as we transitioned from the Matlab implementation of the algorithm to the C implementation. This allowed us to verify the behavior of the PT and energy calculations and was particularly useful in the early stages of debugging

our seam carving algorithm. Since the components of our system are very modular, it was straightforward to test each part of the data flow either in Matlab or on the DSK itself. We did not encounter any significant obstacles in the Matlab or C implementations of the functions, so we were able to closely follow our original timeframe and to use the last few weeks to tweak the behavior of the algorithm and work on optimization rather than debug behavioral issues in the system. The table below shows the weekly breakdown of tasks we accomplished.

Week	**Task**	**Person**
October 12	Matlab gradients and face detection	Dave
	PC video/image processing	Greg
	Retargeting algorithm research	Aneeb
October 19	GUI creation	Greg
	Matlab motion detection	Dave
	Retargeting algorithm research	Aneeb
October 26	Matlab retargeting implementation	Aneeb
	Network infrastructure	Greg
	Face detection and motion detection improvements	Dave
November 2	Seam carving implementation on DSK	Everyone
November 9	Energy calculation on DSK	Everyone
	Network improvements	Greg
November 16	Seam carving behavioral adjustment and optimization	Greg, Dave
	Energy calculation optimization	Aneeb
November 23	DSK algorithm tweaks; complete system testing	Everyone
November 30	Network and PC code fixes	Greg
	Convert full videos for demonstration	Aneeb
	Final code optimizations and complete system testing	Dave

10. References

•[1] L. Wolf, M. Guttmann, and D. Cohen. Non-homogeneous Content-driven Video-retargeting. IEEE Trans. on Image Processing, 2007.

–Video retargeting based on large computations through systems of linear equations

•[2] S.-C. Liu, C.-W. Fu, and S. Chang. Statistical change detection with moments under time-varying illumination. IEE Trans. On Image Processing, 1998.

–Motion detection algorithm

•[3] S. Avidan and A. Shamir. Seam carving for content-aware image resizing. SIGGRAPH, 2007.

–Original seam carving explanation; we use this information as the basis of our algorithms

•[4] S. Avidan, A. Shamir, and M. Rubinstein. Improved Seam Carving for Video Retargeting. ACM SIGGRAPH, 2008.

–Idea of forward energy calculations and suggestions for temporal considerations

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

–Face detection algorithm resource

•[6] Z. Wolkowicki, J. He, and M. Gonzalez-Rivero. Content Aware Image Resizing. 18-551 Fall 2007 Group 6, 2007.

–General resource; adopted Wolkowicki's face detection algorithm