18-551, Spring 2005

Group #4 Final Report

Get in the Game

Nick Lahr (nlahr)

Bryan Murawski (bmurawsk)

Chris Schnieder (cschneid)

Table of Contents

1. Introduction

2. Project Overview

3. Data Rates and Flow

4. Previous 551 work

5. PC Side

6. EVM Side

7. The Game

8. Conclusion

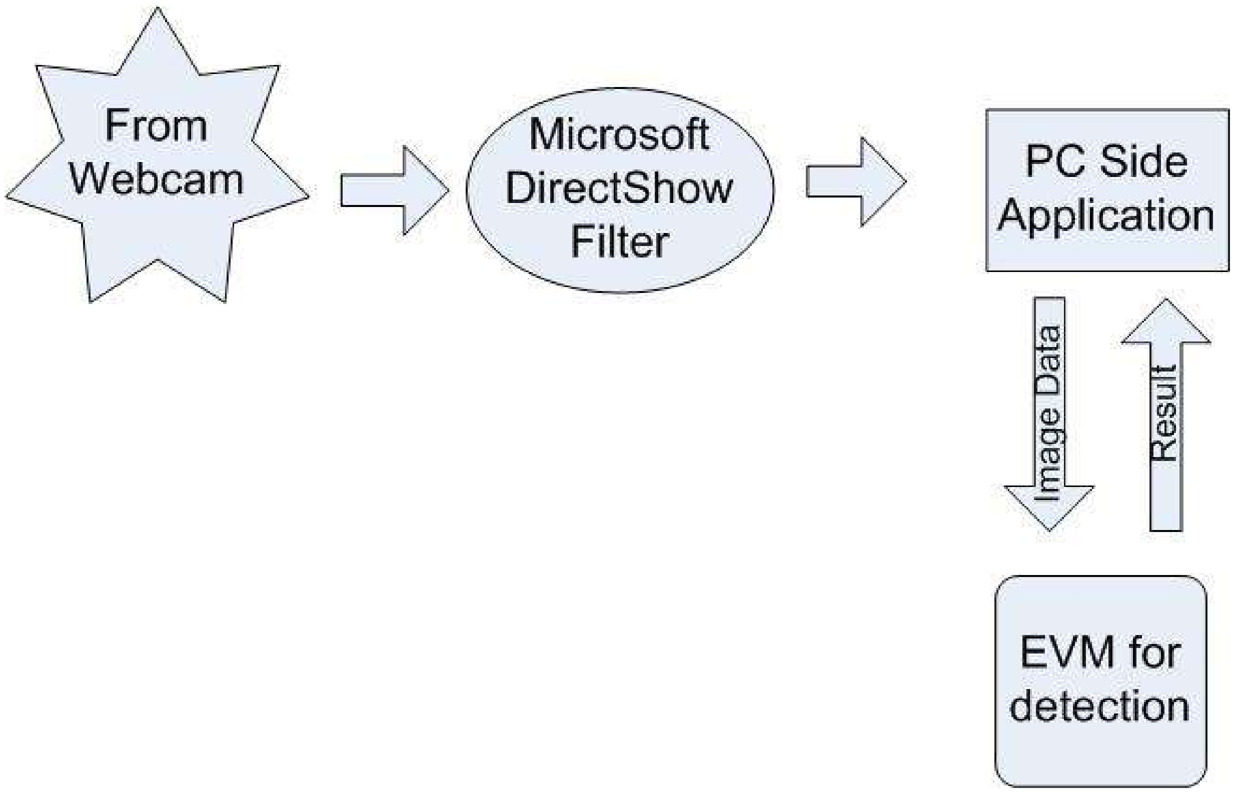
9. References

Introduction

Currently most input devices to a computer require the user to perform some unnatural motion, which is then mapped to an action. This problem is most evident in the area of gaming, where the user interacts via a hand-held controller. This type of interface is awkward, unrealistic, and weakens the impact and realism of the virtual environment. The gaming industry is currently attempting to address this issue with more interactive products such as the Playstation2 eyeToy. The eyeToy is a real time device, which observes a user's motions and uses them as the input to a game, allowing fluid input control. We designed and implemented an eyeToy-like device capable of detecting motion in a video stream in real time. To demonstrate our system, we designed a simple memory game, which the user plays through means of realistic gestures.

Overview

The system consists of a webcam connected to a computer. The image from the webcam is mirrored onto the computer screen, so that users can see their movements. This image from the camera is then split into a 3 x 3 grid of evenly spaced sectors, which are namely the different areas in which we recognize motion. Furthermore, in each sector a differentiation is made between horizontal and vertical motion. The video stream is then down sampled, converted from color to grayscale, and sent to the EVM for processing. On the EVM, a background image is generated. The background is then subtracted from subsequent frames, so that we have the difference between frames. This is then converted to a binary image based on a threshold constant, and if a large enough number of pixels in a sector are white, the sector is marked as interesting. Interesting sectors are then correlated with the corresponding sector of the previous frame. If the frames are identical, the peak value of the correlation will be located at the center of the correlation. Differences in the location of the actual peak and the peak if the frames are identical can be used to track motion. Observing the difference in terms of rows and columns rather than just a raw magnitude allows us to differentiate between horizontal and vertical motion. When enough motion in a sector has been observed, the EVM then sends a signal to the game to that effect.



The data flow of our application. 1) The webcam is captured using a DirectShow Filter. 2) The image is routed through our PC side application to the EVM. 3) The EVM runs our motion detection algorithm, and 4) sends the result to the game, located on the PC.

Data Rates and Flow

The raw data coming from the webcam is a video stream consisting of 24bit color images, each at a resolution of 320 x 240 pixels, with a frame rate of 30 frames per second. Before sending the data to the EVM, the video stream is down sampled to a rate of 15 frames per second, and the resolution is reduced to 160 x 120 pixels. Our overall data transfer rate is:

$$15 \text{ fps} * (160 * 120) \text{ pixels} * 1 \text{ byte /pixel} = 288{,}000 \text{ Bytes/sec}$$

sent to the EVM every second. This is well within the EVM's HPI transfer capabilities. The real limit in our project is not the ability to transfer data, but the speed at which we can process the data on the EVM, as we detect motion in real-time.

Previous 551 Work

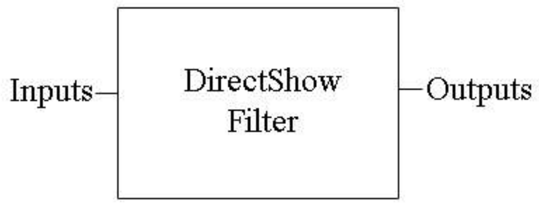
Our project has some minor surface similarities to previous 551 projects. The spring 2004 group "Car Eye for the Drunk Guy" also deals with streaming video from a webcam. Their project was of little use to us however, as they worked with pre-recorded AVI files, and we worked with real time input data.

The spring 2004 group "Where's the Ball" uses a related thresholding process for motion detection. The algorithm used by this group, Kalman Filtering is not suitable for our purposes. However, their project did point us in the direction of a grayscale formula, available via the Raster Data Tutorial. As the eye perceives the different color channels to have varying degrees of darkness, we implemented this formula in our projects grayscale conversion.

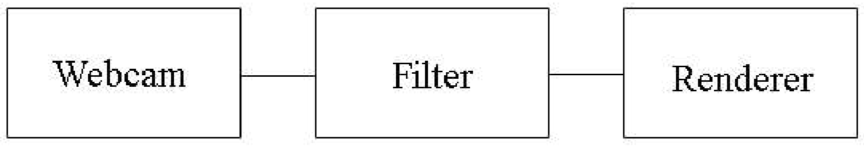
PC Side

One important part of our project was capturing the webcam data. To get this data, we first needed to find a method of communicating with the webcam. In our initial attempt, we tried to gain direct access to the webcam's AVI stream. This proved to be difficult, as we could not simply gain a pointer to this stream, and we weren't sure that such a stream even existed. Other groups that had used this technique in the past appeared to have saved the webcam data and then processed it offline. We wanted to keep our project in real-time so we chose to use a Microsoft DirectShow filter to grab webcam data. The design and implementation of this filter was not a trivial procedure and required us to learn a large amount about Microsoft programming. In learning this, the Microsoft Developer's Network (MSDN) proved invaluable.

To start, we needed to learn what exactly how to create a DirectShow filter. The DirectShow SDK, which is available from Microsoft, had ample amounts of source code that we looked over to find out how this filter works. Basically, a DirectShow filter is an ActiveX object. These objects are compliant with Microsoft's Component Object Model (COM) standard, which is best defined as a set of services that allow you to create modular, object-oriented, customizable and upgradeable, distributed applications using a number of programming languages. Microsoft chose to use COM so that these filters are more versatile and future compatible. The DirectShow filter can be instantiated when any program makes a call to the "CoCreateInstance" function using a unique identifier to that COM object. These identifiers are known as CLSID's and are used to identify different COM objects that are registered with your system. This is one of the reasons why you must register new filters with your system before they will work. Once a DirectShow filter has been initialized, it has to be connected to other objects so that it can be used. To understand how filters are connected it's best to think of them boxes like the one pictured below.

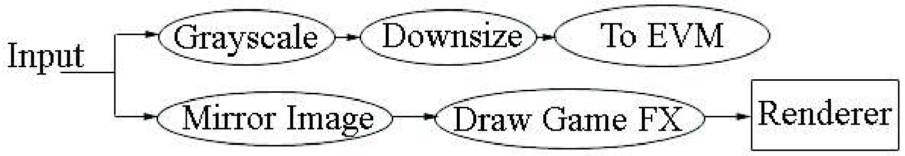


In this picture, there may be any number of input and output pins. This setup is nice because you are able to daisy chain many filters together to transform the image input or output as is necessary. In the high level sense, you need to instantiate as many filter objects, video source objects, and video rendering objects as are necessary for your project and then just connect them into a graph-like structure. In fact, this "graph" is referred to as a FilterGraph by Microsoft. For our purposes, we created a graph that was relatively simple and is pictured below.



Actually creating this filter graph was a little bit trickier than simply drawing these lines. Using the Microsoft DirectShow "PlayCap" project as our starting point, we found how to instantiate all of the filters, create pointers to their pins, and finally use a call to the "Connect" function to link them all together. In the end, we had the webcam feeding a stream of 24bit color bitmap images into our filter and the output of the filter being rendered on the screen using Microsoft DirectX. Because we had the main executable, PlayCap, running and our filter at the same time we now had concurrent processes to deal with. This will be an important point later since it's important to deal with concurrency issues when transferring data between these two programs. For now, however, the next major step was to design the functionality needed by the actual filter.

This filter had to alter the input image, pass the altered data to the EVM, and draw the game graphics that would be needed for playing the game. This was broken into two basic paths, shown below.



First, the input image is altered for the EVM and sent to it; next the original input image is altered for the game and output to the rendering pin. To alter the image for the EVM it is necessary to transform the 24bit color bitmap into 8bit grayscale, and drop the resolution from 320x240 down to 160x120. To do these tasks we used some simple equations. First, to convert 24bit color to 8bit grayscale, we used the equation below, which was found online at the "Raster Data Tutorial" website listed in the references.

$$gray = 0.299*red + 0.587*green + 0.114*blue$$

This equation takes into account how important each color is overall and is able to provide clear grayscale images. To shrink the size of the image, every other pixel and every other row are dropped. This reduced each dimension by a factor of two, making the final image only 160x120 as opposed to the original 320x240. To use the webcam image in the game it needed to be mirrored and have the game data drawn on it. Mirroring an image is done by taking a reflection through the y-axis. Drawing the game's effects and actually transferring to the EVM, however, require the filter to communicate with PlayCap, since it handles game processing.

It seemed best to have only one process, PlayCap, handle the EVM transfers and game playing. This meant that our filter had to get the frame data to PlayCap so it could send to the EVM. Also, since PlayCap was running the game, we needed it to communicate to the filter what types of effects should be drawn. The simplest solution to both of these inter-process communication problems was using shared memory. Shared memory is simply a pointer to a chunk of memory that you initialize with a unique string of your choosing. This string allows the OS to give multiple programs a pointer to this same memory, hence making it shared among them. This adds an aspect of concurrency into our project, as we must now make sure that only one application is accessing the shared memory at any given time. To solve our concurrency issue, we were able to just create a pair of mutexes. Specifically, a producer-consumer method was used so that we could ensure that PlayCap would read each frame placed in the memory exactly once. For clarification, this method is shown below.

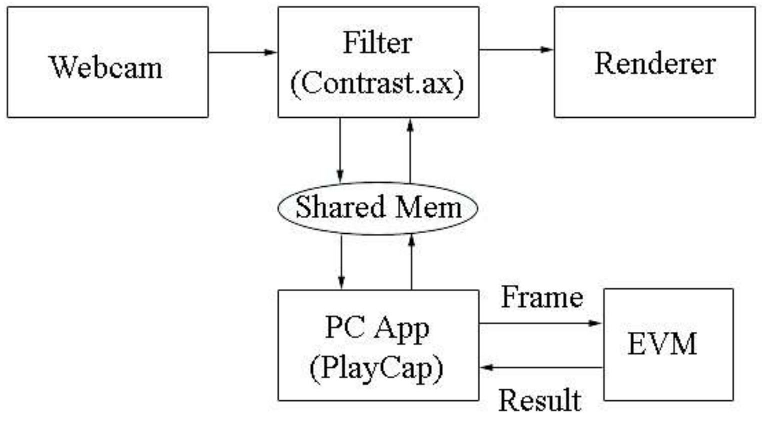
```
MutexP = InitSemaphore(0);  //Initialize Producer to 0
MutexC = InitSemaphore(1); //Initialize Consumer to 1

PlayCap.exe                         Contrast.ax

Wait(MutexP);                       Wait(MutexC);
…                                   …
Signal(MutexC);                     Signal(MutexP);
```

This method allowed for perfect communication of the frames from the webcam to PlayCap. The shared memory was also a useful tool in getting information from PlayCap about what to paint on the screen. For our game's purposes, a simple integer could be used to represent any move that needed to be drawn. We were able to just place this integer into the shared memory and then have the filter read this value when transferring a frame. It would then render the proper image on all subsequent frames until a different value was placed into the shared memory. The last thing that PlayCap needed to do was to transfer data to the EVM.

For the PC to EVM transfers, we used a strategy very similar to that in Lab3. The PC creates an event that is triggered when an EVM message is received. The PC then simply waits for this to be triggered indicating that the EVM is ready to receive information. Also, in its message, the EVM sends a pointer to the buffer where we should write the frame data and sets the second mailbox with an integer that tells us if and where motion was detected. The pointer is used in a call to an HPI transfer that will copy the frame stored on the PC into the EVM's memory. The mailbox value can be a number of values, where zero is no motion and the other constants can be found defined in the move_flags.h file. The EVM sends this synchronization message every time it completes processing the last frame and is ready for another. After it sends this message, the EVM waits until the PC sends a message indicating that the transfer is complete, and proceeds to process the new frame data. Since PlayCap reads every single frame from the webcam, which is going at around 30 fps, it only actually sends every other frame, which reduces our frame rate to 15 fps. This communication method effectively synchronized our transfers from the PC to the EVM. The overall system's communication scheme looks as is pictured below.



EVM Processing

The program on the EVM runs in a continuous loop and recognizes three commands from the PC side. The first special message is the background generation

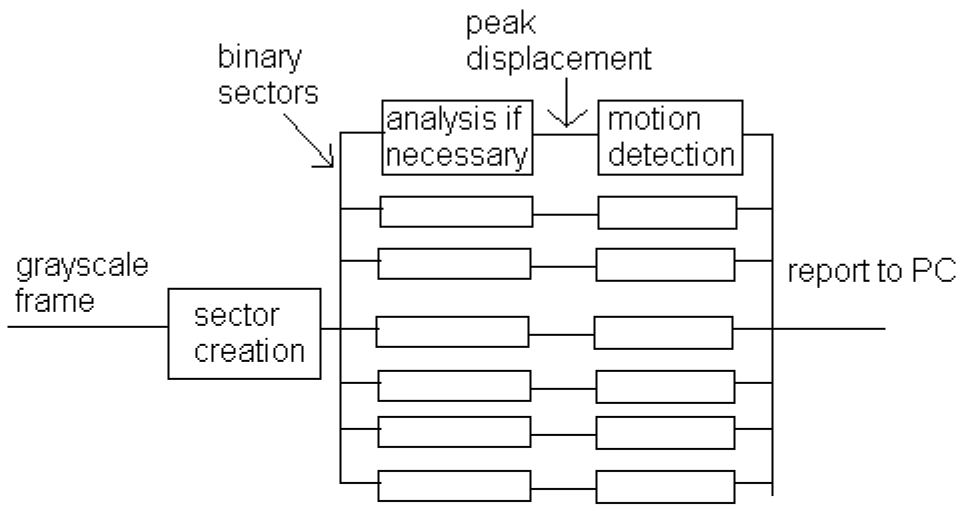
message, which is sent when the game begins. The first frame becomes the base for the background and the next fourteen frames are each averaged with the background up until that frame. Please note that the user must be in the image. The picture on the left is an example of a typical background. This minimizes the loss of data due to the truncation of integer division. The other special message is the quit message, which causes the program to terminate. The majority of frames received by EVM are processed normally, which is broken up into two major parts, sector creation and sector analysis.

For processing, information is stored for both the current frame and the previous frame. In the off chip memory arrays store both the character and float representations of each frame. The location of the new information is alternated between to sets of arrays so no extra memory transfers are required. The character array holds the binary form of the sectors created for each frame. The float array stores the FFT of the sector if the sector was processed so that it can be used in the future. In order to minimize processing time, almost all operations are performed on data that has been moved to a 40kB buffer of on chip memory. The order data is processed is designed to minimize transfers to on chip memory. The only other memory used in the EVM process is for small arrays that store data for motion detection.

The first step in the processing of a frame is the creation of the corresponding sectors. Both the current frame from the PC and the background image are placed in on chip memory. First each pixel in the frame is replaced with the absolute value of the difference between the frame and the background. Next, each pixel in a sector is set in an on chip memory copy of the sector which is then transferred to off chip memory for storage. A pixel is set to a value of 1 if the average value of it and the eight neighboring pixels pass a threshold; otherwise it is set to 0. This threshold of 24 was experimentally determined to generate a binary image that best matched the location of the user in the frame. A count of how many pixels are set to 1 is kept and if more than 100 pixels are set to 1 it is assumed part of the user is present in the sector and the sector is marked for processing. This is repeated for the seven sectors that are actually used in the game.

After all sectors have been created, those marked as interesting are analyzed. This first part of the analysis is a correlation with the corresponding sector in the previous frame. The character version of the sector is copied to a 64x64 float array with all the extra data set to 0. If necessary an array of the sector for the previous frame is also created, but if the sector was analyzed for the last frame the FFT will already have been performed and will simply be reused. Once the float arrays are constructed a correlation in frequency is performed. The result of the correlation overwrites the FFT of the previous sector and the FFT of the current sector is preserved for use with the next frame. Because the original data is only 1s and 0s, performing the correlation in space was considered, but it proved to be slower than in frequency. If there is no difference in the frames being correlated, the peak will be located in the center of the correlation. For each correlation performed, the difference between the actual peak and this location is calculated in terms of the change in rows and columns of the sector. This displacement is used in the motion detection algorithm. The correlation is not a true correlation as the values at the extremes are not calculated to minimize processing time, but it is assumed the peak will not be closer to the edge of the correlation than the center and this has not caused any problems with the calculations. To save processing, the FFTs in the correlation are not shifted and just the location of the peak is adjusted accordingly.

Once the change of the correlation peak is calculated, the sector is checked for motion. Motion should be detected when the user is moving continuously in a sector for at least a second. The square of the magnitude of changes of the peak both horizontally and vertically over the last fifteen frames, one second of data, is stored in a buffer corresponding to each sector. When the sector is analyzed the new value overwrites the oldest one. If the sector is not analyzed, the oldest value is overwritten with zero. If the total of the values in the buffer passes a threshold, the PC side is told that motion occurred in the corresponding direction. An appropriate threshold was determined to be 300 for both directions of motion. Squaring the values of the change both makes them positive and gives more weight to larger changes which are associated with larger motions. To prevent the threshold from being reached in just a few frames with extremely fast motions, the maximum amount any single correlation can add to the total is capped at 100, the equivalent of 10 rows or columns. Experimental data showed normal motion generally created changes smaller than this. To prevent motion from being triggered repeatedly, the total is reset to zero and the buffer is cleared whenever motion occurs in a sector.



EVM dataflow

Frames from the PC are first converted to sectors. A correlation peak displacement is calculated for interesting sectors. The motion total for that sector is then checked against a threshold and results are reported to the PC.

Threshold Calculation

All thresholds used in the algorithm were calculated experimentally. To determine the differencing and interest level thresholds used in sector creation, various values were tried with a video stream saved in a file. To determine the motion thresholds, which were eventually set at 300 for both horizontal and vertical motion, people used the system. The thresholds were adjusted so that motion could be triggered when desired, but would not occur too easily. These thresholds work as desired regardless of the user.

Optimizations

The only real optimizations performed on the code were to move all the data to on chip memory before performing extensive calculations on it. The following profiling data was obtained for some functions in the code:

160 x 120		
Function	**Cycles before optimization**	**Cycles after optimization**
make_sectors	5,500,000	620,000
make_fft_array	167,000	140,000
find_peak	100,000	170,000
Correlation2D	2,200,000	900,000

The correlation took to long to perform when off chip and the PC side would crash when the EVM was unable to accept more frames so no profiling data was performed. With all the optimized code, processing a frame always took less than 7 million cycles, although the exact value depends on how many sectors are analyzed and how many of the FFTs were already performed on the previous frame.

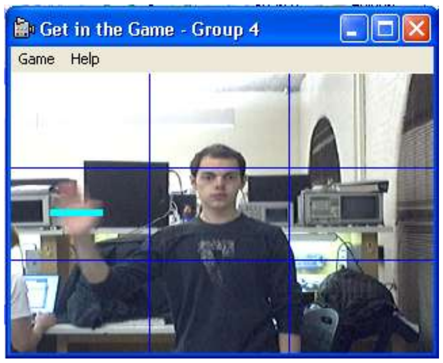
The Game

The game that was developed was a simple memory based game. Basically, the player has to imitate the computer generated sequence. This sequence starts one move long, but becomes more elaborate by building on itself as the player continues to play. There are 14 different moves that can be made by the player. The number 14 comes from how the screen is divided. There are 9 squares on the screen, out of which, only the outside 7 can register hits. This is because we chose to ignore the middle and bottom middle sectors since making motion in there proved difficult and awkward. In each square that can register a hit, you may have either horizontal or vertical motion. Seven squares with two moves per square gives us 14 different moves. A typical playing of the game could go as follows.

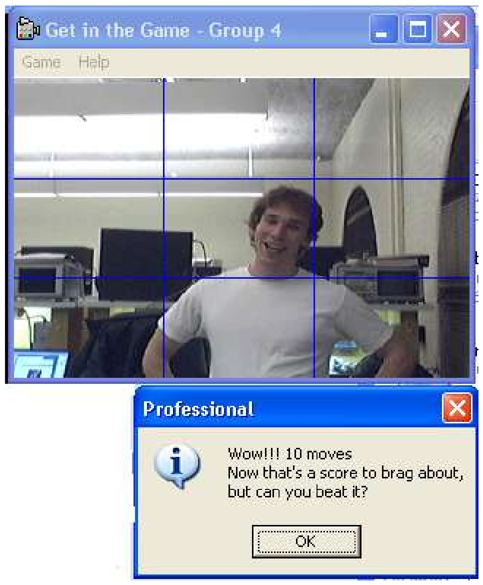
Round	***Computer (goes first)***	***User (repeats computer)***
1	Left-Horizontal	Left-Horizontal
2	Left-Horizontal, Right-Vertical	Left-Horizontal, Right-Vertical
3	Left-Horizontal, Right-Vertical, Top-Horizontal	Left-Horizontal, Right-Vertical, Top-Horizontal
4	Left-Horizontal, Right-Vertical, Top-Horizontal, Left-Vertical	Left-Horizontal, Right-Vertical, Top-Horizontal, Left-Horizontal

In the example above, the user follows the computer correctly up until the 4th round of the game, and would, therefore, be informed that their score was a 3. The score is the number of rounds that were done correctly, so it must exclude the last one since they failed on that one. This gives a general overview of how the game is played. There are, however, more specific details on our exact implementation that are important to mention.

The actual game is implemented in a thread which is created by the PlayCap program. This thread handles all game related issues and is terminated by a call to exit(0) when the application is done running. After this thread is launched by PlayCap, you are greeted with a picture of yourself on the webcam. The game will start when you choose the Game→Start from the menu bar. At this point a background is generated by sending a special message to the EVM. Once the background is generated, the game will commence. The PC knows that background generation has been completed because it receives a special result value from the EVM. To indicate where you are supposed to move colored bars will appear on the screen. These bars are either horizontal or vertical and in the sector that you are supposed to move in. The colors are cyan, pink, or yellow. Below is a picture of one such bar being triggered by user motion.



Each time the computer shows the new pattern it should be repeated by the user until failure. When they do fail, one of several customized victory messages will be displayed on the screen. On the next page is a screenshot showing one of the 6 possible victory messages.

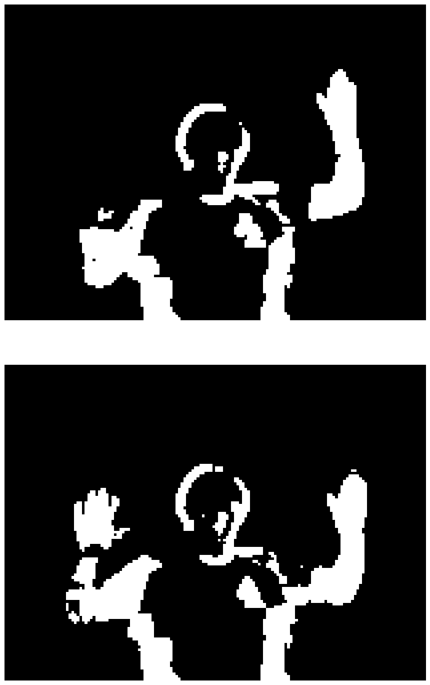


Wow!!! 10 moves
Now that's a score to brag about,
but can you beat it?

After this point the user can choose to play again or to quit the game. If you continue the start button will have to be pressed again before the game will resume. This was done so that different players could take over after a round and have time to get ready. Once start is pressed, a new background will be generated and the game sequence described above will be repeated. The game is a simple, but effective demonstration of how our detection scheme works and can be used in the real world.

Conclusions

The project successfully met the original objectives, but there are limitations. The success of the motion detection algorithm relies heavily on the proper generation of the background image. In the lab setting, we often experienced problems triggering motion in sectors 0 and 2, that is to say the sectors in the lower left and lower right corners of the grid. Due to the varied and cluttered nature of the background in these sectors, a moving hand does not contrast as sharply as it does in the other sectors, and it washes out. This phenomenon can be clearly observed in the binary images shown. Please note how the hand on the left is clearly visible in the lower image, but not the upper.



A second issue with the background is that if an object enters or leaves the background, it looks acts a stationary part of the user and the correlation peak displacements will not correspond to the user's motion. Some sort of background updating process could alleviate this, but it would be difficult to implement with the current structure of the EVM code. This issue is present because the algorithm takes into account the entire user. This also limits the type of motion that can be detected. A possible expansion to this project is to isolate just the user's hands. This would both limit the problems caused by background changes and possibly allow tracking more complex movement patterns. The goal of the project was to detect motion in various sectors of the screen, though, and this has been achieved.

References

Green, Bill. “The Raster Data Tutorial”. http://www.pages.drexel.edu/~weg22/raster.html 2002. Drexel University.

Green, Bill. “The Raster Data Tutorial (24 Bit)”. http://www.pages.drexel.edu/~weg22/colorBMP.html . 2002. Drexel University.

These two references were very useful for working with bitmap images. The first explains the bmp file structure, i.e. how many bits the header is, where the size information goes, etc. Internally we process only raw pixel data. Therefore, for debugging purposes we needed to create bitmaps of the various stages in our algorithm, so that we could view the background image, the subtracted image, the binary images, etc and verify their correctness. This site explained the file information, allowing us to create actually bitmaps which could in turn be opened by Windows Picture Viewer or other suitable program.

The second reference details the algorithm we used for the conversion to grayscale, how the red, green, and blue color channels are weighted differently.

Jones, Douglas. “Decimation-in-time (DIT) Radix-2 FFT”. Rice University. **http://cnx.rice.edu/content/m12016/latest**.

This site contains an easy to understand implementation of a radix 2 FFT, with code in c that works as is. This code is slow, and we eventually rejected it in favor of TI’s own radix 4 FFT code (this is the same as the FFT code used in lab 2). However, this code is much easier to use than the TI code, as the user of this code does not need to worry about either twiddle factors or bit reversing. This was very useful in verifying the correctness of the other parts of our algorithm.

“Motion and Video Analysis”. Metaverselab.
http://metaverselab.org/classes/635/lectures/published/motion.pdf

The idea of background subtraction is well known, but this where we were introduced to the concept. Though we ultimately used an algorithm of our own design, and not anything shown here, our algorithm uses background subtraction.

Microsoft Developer’s Network (MSDN) **http://msdn.microsoft.com/**

The data provided in the MSDN was used for everything related to the PC side. It provided much of the information necessary for creating the DirectShow filter. On top of this it contained documentation for every function used pertaining to Shared Memory, Mutexes, GUI design, and COM programming.