1. Introduction

Our goal for the 18-551 project was to implement a system which could detect a user's motion and recognize the gestures being made. As a motivation for the project, we decided to encapsulate this functionality in a "Simon Says" game for children. If all went well, the system would prompt the user to make a certain gesture. After detecting the user's motion the system would compare it with an ideal version of the motion stored in memory, and then give the user feedback through a simple graphical user interface.

We thought this project was important for several reasons. First, motion tracking and gesture recognition could have applications beyond a simple game. A user's gestures could be used in place of mouse input on a personal computer, to interpret an orchestra conductor's signals, or any of a number of applications. Second, none of us had worked with video before and wanted to try it out. Third, previous 18-551 classes had used a less powerful DSP; we wanted to see if the new C67s were robust enough to handle video.

We decided on a few details about the internals of the system beforehand. First, rather than detecting all motion in the input, the system would follow a target object held in the user's hand. By searching in each frame for this object and then storing the points where the object was found, the system would be able to follow the user's motion. In the final version of the system, we found that a closed fist actually worked quite well as a target object, removing the need for a separate object.

As for the actual gestures that would be recognized, we chose to use simple shapes: triangles, squares, pentagons, and circles. The recognition problem would be accomplished by storing "critical" points. This method will be described in detail later.

To work properly, our system would obviously have to work in real time. We hoped that in its final stage our system would be able to capture images from a camera, search for the user's fist, interpret the motion, and give the user feedback. For various reasons that will be covered later, we fell short of this goal. Our final system had to use as its input sequences of captured images that were stored on disk beforehand.

From looking at the projects of previous 18-551 classes, it seemed that our project was fairly unique. In the spring of 1995, one group (Vishant Bhatia, John Cosnek, Dan Kodesh, Andrew Vaz, and Chris Wise) had attempted a video surveillance tracking system. Some of the requirements of such a system – real time motion tracking, for example – would be similar to ours. Unfortunately, this group had trouble even locating the necessary hardware for their project and had to improvise extensively. We found that their project was not really useful as we went ahead with ours. This was the only project we could find that was remotely similar to ours. No one else had attempted a motion tracking system.

Before we really knew the capabilities of the EVM, we had five frames per second in mind as a goal for the system. By using some creative searching techniques (hierarchical searches, limiting the size of the search window to a certain space around the previous detection point, etc.), we were actually able to achieve much higher frame rates.

2. System Description

Below our system is broken down into several stages. The first is the system input that gets the images and sends them to the EVM. The second is the object detection algorithm that finds the target object in each frame. We considered several different algorithms, which are explained below along with the optimizations we made to speed up the system. Next we discuss the method we used to track the user's motion. Finally, we describe how the different shapes were detected and how the user was given feedback through the GUI. A char that illustrates the system flow is presented in Fig 1.

Fig 1. System Flow

2.1 System Input

Input to the system was accomplished in two stages. First the images had to

be captured using a camera from Logitech and code we found on the web. After converting to RAW format, the images had to be transferred to EVM. Once they were in the EVM's on chip memory, they would be ready for processing.

2.1.1 Hardware

We chose a Logitech QuickCam VC as the video input device for our system. The Quickcam VC can capture images at a variety of resolutions and numbers of colors. We felt that this would be useful if our goals for the image resolution changed or we decided to use color. The camera cost $88.42 directly from the Logitech web site.

The QuickCam family of cameras has become quite popular and we found video capture code for windows on the web quite easily. We chose to use code from a site called www.kolban.com/video/index.htm. The program was called memcap.c and it used the video for windows library (vfw32.lib) from Microsoft. Initially we had trouble compiling this code, but this was due largely to our inexperience with Windows programming. With some effort we were able to compile it and it worked well. The basic function of this program is to pipe the video from the camera to a window. When the user selects "start" from the control menu, a callback function is enabled and called repeatedly until the user presses the escape key. In its given form, the callback function did nothing, but this was the place to add any frame processing. We planned to add in the PC side of the system's code in this function. Unfortunately, this turned out to be the biggest obstacle to our goal of a real time system. Obviously, integration of the image capture code with the PC side of the EVM code would be essential to achieving a real time system. When we attempted to combine the two, we discovered that everything, including frame transfers to the EVM and the object detection, had been tied to very odd window event. Basically, nothing would happen unless the user repeatedly minimized and maximized the window. We spent quite a while attempting to resolve this but we couldn't identify the portion of the code which was causing this. We chose to move ahead with more essential components of the system rather than spend more time on image capture. As a partial solution, we edited the callback function so that it would simply write each frame to disk in a sequence titled "frame0.raw," "frame1.raw," etc.

2.1.2 Software

For simplicity's sake we chose to use RAW formatted images. The RAW format is completely uncompressed. It is simply a long sequence of bytes, with each byte representing a pixel value. Although this generally increases storage requirements, it makes for much simpler EVM code. The memcap.c code stored each frame as an uncompressed bitmap, so the callback function handled the conversion to RAW format.

We chose to use black and white images to keep our storage requirements down. From the available resolutions of the QuickCam VC, we chose 160 x 120. This was mostly because our initial tests on the EVM used 128 x 128 images and 160 x 120 was the closest to that. The size of each image was 19,200 bytes. The target objects were stored in 16 x 16 frames (256 bytes). This meant that our storage requirements could be met completely on chip.

Communication between the PC and the EVM was accomplished with an interface consisting of several commands. The commands are listed below:

1) Send an image to the EVM
2) Send the target image to the EVM
3) Retrieve the detection point from the EVM
4) Exit the program
5) Retrieve the type of shape from the EVM

One of the more frustrating experiences in implementing our project was the transfer of the frames to the EVM. We had hoped to use asynchronous DMA transfers to send each image from the PC. This would allow the EVM to process the current frame while retrieving the next one from the PC. It would also complicate our memory management slightly. The EVM would do its computations on an image stored on chip; while this was happening, the PC would transfer the next one to off chip memory. The image in off chip memory would then be copied to internal memory. However, this was certainly not overly complicated and would save time.

Initially, we believed we had the DMA transfer working. We used a set of test images just to confirm that it was and found that it worked with some consistency. Later, when we were putting the pieces of our project together, we found that the DMA only seemed to work on that particular set of files.

Other sets of files caused the DMA to freeze and the whole system to hang. We could not explain this since the new sets of images were the same size and format. This forced us to change our transfer to use the HPI. Transfers via the HPI are synchronous, so the EVM couldn't process one image while receiving another. Additionally, the memory swapping code would not be necessary. In the end, it seemed that the computation rather than the transfer dominated system performance, so we don't believe this had a hugely detrimental effect on our system.

2.2 Object Detection

Object detection is performed to locate the target object within the given input frame if it presents. Using the target frame and input frame that are stored in the internal memory, detection algorithms are applied to estimate the location of the target object, in our case the user's fist. These algorithms are optimized to achieve a frame rate that is needed to reasonably follow hand movement during live capture. Additional tweaks are also added so that the system is less vulnerable to fist rotation and changes in lighting condition.

2.2.1 Detection Algorithms

Detection algorithms detect the presence of a target object within the input frame and estimate its location when it is present. Two algorithms were considered for this project. They are *correlation* and *frame differencing* .

2.2.1.1 Correlation

Correlation is the operation that is performed to see if the target frame correlates with a section of the input frame. When it is correlated, the result of the correlation operation: the correlation coefficient will be high. When this value is above a certain threshold, it can be assumed that the target object is in that particular section of the input frame.

The correlation operation is described by the following equation:

$$C(\text{section},\text{target}) = S_{k=\{0,T-1\}}\ S_{l=\{0,T-1\}}\ \text{section}(k,l) * \text{target}(k,l)$$

Note that *section* and *target* represents the section of the input frame and the target frame respectively. Additionally, T represents the width and height of the target frame, which are the same in our case.

To find the most correlated section, and thus the best match for our target, the correlation operation needs to be performed across the whole frame, resulting in a search time of $O(MNT^2)$ multiplications where M and N are the width and height of the input frame. One can see from the number of multiplications involved that correlation is an expensive operation. For that reason, another algorithm that performed similar task is considered.

2.2.1.2 Frame Differencing

Frame differencing is the operation of taking the difference between the pixels value of the target frame and sections of the input frame. When the difference is small enough to be below certain threshold, it can be assumed that the target object is inside that particular section of the input frame. In essence, this algorithm is similar to the correlation algorithm mention in the previous section.

The frame differencing operation is described by the following equation:

$$FD(\text{section},\text{target}) = S_{k=\{0,T-1\}}\ S_{l=\{0,T-1\}}\ |\ \text{section}(k,l) - \text{target}(k,l)\ |$$

Similar to correlation, frame differencing also needs to be performed across the whole frame. However, instead of $O(MNT^2)$ multiplications, we now have $O(MNT^2)$ subtractions. Since a subtraction in general takes less time than a multiplication, frame differencing is on average faster than correlation. However, given the system's dimensions of input frame and target frame, the number of operations that needs to be performed per frame is still too great. For that reasons, we looked to further optimize our system.

2.2.2 Optimizations

Several optimization options are explored in order to speed up the search time. These optimizations include hierarchical search, software pipelining, and FFT based correlation. At the end, the former two are implemented, while the latter is not.

2.2.2.1 Hierarchical Search

Hierarchical search is a multilevel search. The basic idea behind it is that the correlation between a target and the input can be obtained even on downsampled version of their image frames. Although doing so will resulted in a less accurate estimate of the location, the estimate should be close enough that a local full resolution search is sufficient to determine the better estimate.

When the search are performed using images downsampled by a factor of X in each dimension, the search time is reduced by a factor of X^4:

$$O(M'N'T'^2) = O(M/X\ N/X\ (T/X)^2) = 1/X^4\ O(MNT^2)$$

Note that this performance increase is offset by a decrease in accuracy. Since the frames now hold less information, it is less likely that a false detection occurred. In the end, we decided to implement a two level hierarchical search. A two level search will give us a performance gain of 16 while offering reasonable accuracy given that our target frame size is 16x16.

Although this optimization presented a great saving in calculation, it did not fully take advantage of the C67 hardware specs. For that reason, parallelization in the form of *loop unrolling* is explored next.

2.2.2.2 Loop Unrolling

Loop unrolling is a coding technique that takes advantage of having multiple ALUs by utilizing as many ALUs as possible in every cycle. An example of this technique is given in Table 1.

Original Loop	**Unrolled Loop**

for(i=0;i<T;i++)
  temp+=x[i];
	temp1=0;temp2=0;
	for(i=0;i<T;i+=2){
	  temp1+=x[i];
	  temp2+=x[i+2];
	}
	temp=temp1+temp2;

Table 1. Example of Loop Unrolling

Notice that in the unrolled loop, two additions are performed in every iteration. Since these additions are written to two different temporary variables, they can be performed in parallel. And at the end, an extra addition can be done to combine the two temporary variables in to one. In contrast, in the unrolled loop, each addition has to be finished before the next addition can be performed.

In our system, the code section that gets executed the most is the inner loop of frame differencing function. Therefore, that particular section is unrolled into four subtractions. The reason that there are only four instead of six (the number of ALUs in C67) operations is because our target size is a multiple of four and not a multiple of six. Additionally, some ALUs are still needed to increment the index. This optimization boosted the frame rate to 5.5.

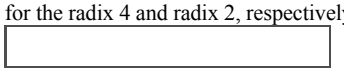
2.2.2.3 FFT based Correlation

Other optimizations that we studied were FFT based correlation and direct correlation. Assembly code had already been written for the C67 and was available on the TI web page at http://www.ti.com/sc/docs/products/dsp/c6000/benchmarks/67x.htm. All of the functions also came with formulas for the number of optimized cycles. The formula for the cross correlation is:

where nb is the length of the first array, and nr is the length of the second. For optimization, nb must be a multiple of 4, greater than or equal to 4, and nr must be a multiple of 2, greater than or equal to 4. This is a one-dimensional correlation, so in order to perform it on the two-dimensional images for our project, the picture must be flattened into a one-dimensional array. The size of our images was 160 X 120 and 16 X 16, for the frame and target, respectively. This results in nb = 19200, and nr = 256. Both of these values meet the criteria for being a multiple of 4 or 2. The total number of cycles would be 2,458,248 per frame processed. However, if we limited the search to a 32 X 32 window, the new total would be 131,720 cycles per frame. The FFT based approach involves multiple benchmarks from the TI web page. First, there is a one-time operation of transforming the target to the frequency domain. Then, for the first frame, the whole image must be transformed, with only the 32 X 32 window needing the FFT for succeeding frames. Next, the transforms need to be multiplied by each other, taking the conjugate of one. Then an inverse FFT needs to be applied.

There are two assembly codes available for the forward FFT, the complex radix 4 FFT and the complex radix 2 FFT. The radix 4 is a decimation in frequency, with digit reversed output for a normal ordered input. The radix 2 is a decimation in time, with bit-reversed output for a normal ordered input. The two equations for the number of cycles are: inverse ((2*N)+16)*log2(N)+25 bit reverse (N/4)*11+9

for the radix 4 and radix 2, respectively. The equations for the radix 2 decimation in



frequency inverse FFT and bit reversal are:

Table 2 lists the calculations for our frame sizes.

	N	Radix 4 FFT	Radix 2 FFT	Radix 2 IFFT	Bit Reversal
16 X 16	256	3696	4286	4249	713
32 X 32	1024	18,055	20,716	20,665	2825
160 X 120	19200	478,272	546,720	546,639	52,809

Table 2. Frame Sizes

Overall, this method would use 3696 + 478,272 + 546,639 = 1,028,607 cycles, in addition to 4,849,920 multiplications (see next paragraph) and the conversion to the conjugate. Following frames would need 196,862 multiplications, plus the conjugate.

The method that we chose, frame differencing, uses only subtractions. For the first frame, when it searches the entire 160 X 120 image, the total number of subtractions would be 256*(19200-256+1) = 4,849,920. Subsequent frames would require 256*(1024-256+1) = 196,864 subtractions in the 32 X 32 window. Since we were only looking to process 5 frames per second, frame differencing worked fine, just fine. It was easy to write code for this, and could be tailored for our individual needs.

2.2.3 Additional Tweaks

Although the above algorithm along with its optimization offers a fast enough search time, the system is still prone to noise. The two most apparent problems: fist rotation and luminance change are addressed by adding code for *rotated target* and *adaptive target* respectively.

2.2.3.1 Rotated Target

It is almost impossible for the user to keep the fist level when he is making a shape with it. If the system simply takes a level target, it might not find the fist when it is rotated during capture.

For this reason, the system is modified so that it can takes as an input several template of the fist, each rotated at a certain angle. Whenever the result of the level two frame differencing over the whole frame too high, the target frame is switched to other rotated version of that target in order to see if a rotated

version of the fist exists instead. If all of the targets is used and the search algorithm still return a high value, then the system assumes that the fist is not present.

The cost of this modification is an increase of the number of searches whenever the target is lost. In our system, we stored three template of fists rotated at different angle. This method decreases the frame rate to 5.2. It is able to find rotate version of the fist, however, from our testing, it seems that it needs more than three version of the fists as it fails to correctly detect the fist in more than half of the frames. Additionally, it is till prone to lighting changes, which is addressed next.

2.2.3.2 Adaptive Target

Another problem that applies to all target objects rather than just human's fist is luminance changes. Frame differencing is vulnerable to luminance change as a slight luminance difference between the input and target frame can result in a huge portion of the frame difference. This is true especially since the difference in luminance normally affect a major portion of the input frame and thus the difference is multiplied by the size of the target frame. For that reason, the system can be set so that it will continually adapt to the target. This is done by overwriting its target frame with the most recent section of the input frame that it acknowledged as having the target in it. This enables the system to adapt to gradual change in lighting. It even enables the system to adapt to other changes such as slight size and orientation changes. The cost for this addition can be substantial, as it needs to be performed on every frame in which the target is found. Additionally, there is the risk of getting stuck with a wrong target, i.e., adapting to a false target object. For that reason, the threshold must be set to be pretty low to avoid adapting to a wrong object. This method resulted in a frame rate of 4.3, which is significantly lower than the previous frame rate of 5.2; however, using this method, our system is able to track the motion of a rotated fist 75% of the time. Given this reasonably accurate and fast object detection system, the next step to be done is to track the movement of the object, which is discussed next.

2.3 Object Tracking

Object tracking is performed in order to follow the movement of the user's fist. Once the object detection code is performed, it is only a matter of streaming the correct coordinates to other part of the system, namely GUI and motion recognition. In order to increase the performance of the tracking, we take into account the extent of our application and came up with two ideas: *limited motion* and *adaptive filter* .

2.3.1 Limited Motion

Given that the user is not going to move his hand to fast, the system can limit the initial search to a smaller window. That is, once a location of the fist is known, the system is going to assume that the fist is not going to be more than X pixels away from it in the next frame. Therefore, it can perform the initial search on a window of size 2X+1 by 2X+1, where 2X+1 is the total range of the movement (-X,-X+1,…,0,…X-1,X). When the initial search fails, a full frame search still needs to be performed, in case the user moves his fist too fast that it went out of the limit window.

For our system, we set X to be 15. This limit allows the system to only search 31 x 31 window instead of 160 x 120 that it originally searches. The performance increases dramatically as it resulted in a frame rate of 34.4.

2.3.2 Adaptive Filter

Another optimization that was not included was the use of linear predictive coding of an adaptive filter. This was also found on the TI web page, at http://www.ti.com/sc/docs/psheets/abstract/apps/spra116.htm, but was optimized for the C30. However, the only usefulness this packet presented was the explanations of the algorithms, since the optimized C code and the assembly code were not included. A basic algorithm in C was included:

```
void lms(float *x,float *h,float *y,int NumH,float *d,float ar, short NumY){
        int i,j;
        float sum;
        float error = 0.0f;
        for (i = 0; i <NumY; i++){
                for (j = 0; j < NumH; j++) {
                        h[j] = h[j] + (ar*error*x[i+j-1]);
                }
```

```
                sum = 0.0f;
                for (j = 0; j < NumH; j++) {
                        sum += h[j] * x[i+j];
                }
                y[i] = sum;
                error = d[i] - sum;
        }
}
```

where y is the output, x is the input, h is the coefficient array, d is the expected output, error is the error, and ar is the mew value. This was implemented in C and MATLAB to test the accuracy of the filter. In MATLAB, random numbers between 0 and 128 were generated and then sorted in ascending order. This was to simulate an unexpected change in acceleration by the user. The size of the filter was 8, and it needed 8 previous inputs. Some problems that were encountered were determining the initial values. The mew value needed to be different for varying inputs, and the initial coefficients were to be determined by us. We ended up using a filter that just averaged the values initially. In the end, the cost of recomputing coefficients for 8 previous inputs was too much, especially since high accuracy could not be obtained. In was more efficient to start the search from the location of the previous point and limit the motion rather than using an LPC filter.

2.4 Motion Recognition

Our system was intended to be able to determine the motion made by the user, given the recognized points, from a predefined list of basic shapes. This was done by in basically three steps: calculating the distance of the points, calculating the angle of the motion vector, and finally recognizing the shape from the number of critical points.

The first step of calculating the distance is performed to filter out the points where the users fist does not move by a lot. The reason to filter these points out is to enable a more accurate measurement of the angle of the motion vectors. In the second step, we determine the angle between two adjacent vectors to find critical points, i.e. points that are considered as corners of a shape. If a critical point is found, it is pushed in to a stack of critical points that the current shape has. The minimum angle between the motion vectors for the point to be considered critical is 45%. The third step involves checking if the last tracking point is close enough to any of the previously stored critical points. If it is within 15 pixels of these points, it means the user has drawn a closed shape. This particular shape can be derived from counting the number of critical points that lies between the last tracked point that the critical points that it closes on. That is, two points corresponded to a line, three to a triangle, four to a square, five to a pentagon, and six or greater to a circle. Once a shape was calculated, the critical point stack is cleared and the process started over again.

2.5 GUI

The best way to observe our project was to use a simple graphical user interface. We found one that could easily be implemented at http://fltk.org. A window, buttons, text output, and frame were defined in main. In order to update the picture captured from the camera, there was a redraw function that was called after each frame had been processed by the EVM. The image needed to be opened from memory and read into an array. This was done for each new frame, while the nine images of the fists in different orientations were static.

In order to show the motion that had been captured, a green line traced the recorded path of the fist. Once the recognition code recognized a point, it was passed to the GUI. This point was then stored in an array that kept tract of the previous 256 points. This was needed since each redraw erased the old lines. This array was then used by a line draw command. Once on object was recognized, the array was reset and all of the old lines on the screen were erased.

The two buttons that were implemented were a simple stop and exit. The exit button just exited the whole program, while the stop button stopped the transfers to the EVM. The buttons used callback functions, which were called whenever a button was pressed.

The text window was used by the recognition code to display the shape it had determined. Once the shape was decided, it updated the string and passed it to the text box display. If the system had been implemented in real time, this window would have prompted the user for a specific shape. If the shape was

correct, it would say that the shape was drawn correctly and ask for a new shape. If it was not correct, it would let them know that as well.

3. Profile

The different optimizations and tweaks resulted in different tracking rates and frame rates. For that reason, it is useful to have a comparison of their profile. We had problems while profiling the code on the EVM side using the profiling options available from Code Composer. Therefore, we used the clock function on the PC side to measure both the transfer time and the calculation time. Transfer time is measured by calculating the difference between the time when the EVM request for data arrived and the time when the EVM indicated that the transfer is completed. The calculation time is measured by calculating the difference between the time when the EVM indicated that the transfer is completed until the EVM requested the next frame. Note that since the units of the value that the clock function is unknown, we ended up calculating the time that it takes to process about 400 frames with a stopwatch to find how the numbers given by the function translates to the number of seconds. The result of these measurements is displayed in Table 3.

	HPI	EVM	Frame Rate
Hierarchical Search	1.9	196.1	4.7
Loop Unrolling	2.1	168.5	5.5
Rotated Target	2.3	179.0	5.2
Adaptive Target	1.8	217.0	4.3
Motion Limit	2.1	27.1	34.4

Table 3. System Profiles

The profiles are listed in the order they appear in this report. Initially, when the system consisted only of a two level hierarchical search the system produces a frame rate of 4.7. The simple addition of software parallelization increases the frame rate to 5.5. When Rotated Target code is added, the frame rate decreases to 5.2. However, the added benefit of being able to track rotated fists seems to justify it. When the Adaptive Target is put in instead, the frame rate drops even more to 4.3. This is due to the cost of copying the target at every frame. However, the tracking rate of this algorithm is really good. In addition to adapting to luminance change, this algorithm is able to adapt to slight size/orientation of the fist. Finally, when we introduced motion limiting, we obtain a huge performance increase of 34.4 frame rate. This frame rate is more than enough to track a fist on real time.

Given the result of this profiling, we decided on using a two level frame differencing search, optimized with loop unrolling and equipped with adaptive target. Additionally we limit the motion to 15 pixels. This EVM signal flow for this system is illustrated in Fig 2.

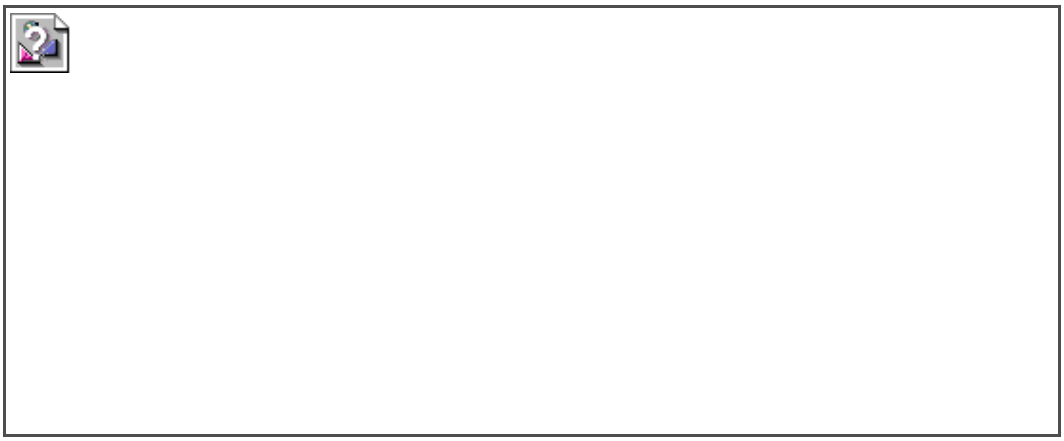


Fig 2. EVM Signal Flow

4. Sumary

Our system met most, but not of our goals. It successfully detected and tracked the user's motion and did a reasonably good job of determining which shape the user was tracing. The GUI did a good job of providing the user with feedback and made it easier to tell if the detection portion of the code was working. The chief failure was in the image capture portion. Our inexperience with Windows programming made it hard to integrate the capture code with the rest of the project. Our workaround made it possible to complete the rest of the project, but for a completely successful system we would need to have real time capability.