18-551, Spring 2004

Group #2, Final Report

Where's the ball?

A Ball Tracking System for
The Lazy Sports Fan.

Chris Pearson, cpearson@andrew.cmu.edu
Amy Tsao, atsao@andrew.cmu.edu
Christina Yu, cyu@andrew.cmu.edu
Matthew Theisz, mtheisz@andrew.cmu.edu

Project Intro

Our project, “Where’s the Ball?”, aims to find and track a soccer ball in real time video format using continuous image processing. The purpose of addressing this problem is to develop a tracking system of object(s) that can further be applied to such designs as automated camera tracking, motion sensing security, and other forms of object displacement in real time. We intend to preprocess the captured video on the PC, feed it to the DSP, and then reconstruct the video with the ball highlighted on the PC. All the detection, tracking, and highlighting will be performed on the DSP.

We chose to use blob coloring, correlation, and an adapted version of geometric moments to find where the ball was in each frame. After we have located the position of the ball in the first frame we begin to narrow down our search area dependent on the position of the ball in the previous frame as well as the velocity of the ball. This helps to eliminate false positives in addition to reducing the amount of data we need to send to the EVM. Once we have located the pixels corresponding to the soccer ball we change the color of those pixels to a bright blue, so that it is more visible to viewers.

Prior 551 Work

When we first started doing research on possible algorithms to use in our project, we came across a previous 18551 project from 2000, “Object Tracking via Optical Flow in Video”. Their project was very similar to our proposal, except that we limited our focus to just tracking a soccer ball. Their aim was to implement algorithms for tracking a moving object in an AVI compressed video sequence. They used the KLT (Kanade-Lucas-Tomasi Tracker) algorithm to select and track ‘good’ features in an object. By tracking these features in the video they intended to find the change of velocity vectors in order to perform object distribution analysis. Unfortunately, they were unable to implement their algorithm onto the EVM. In their conclusion they cited that the KLT tracker algorithm was very computation intensive and suggested using Kalman Filtering as a possible improvement to their project.

After reviewing Group 15 of 2000's report, we tried to use the KLT tracker algorithm in our project, and pick up where they left off. We played with the code for the KLT algorithm we found online, but we eventually gave up on it. We found the KLT was not an optimal way to track the soccer ball in our project, especially since the soccer ball in our video was so small. The KLT only assigned one or two 'good' features in which to track the ball. These features were easily lost and hard to reacquire in subsequent frames. Also, in order to find the ball in the first frame we needed to tell the program which feature corresponded to the ball.

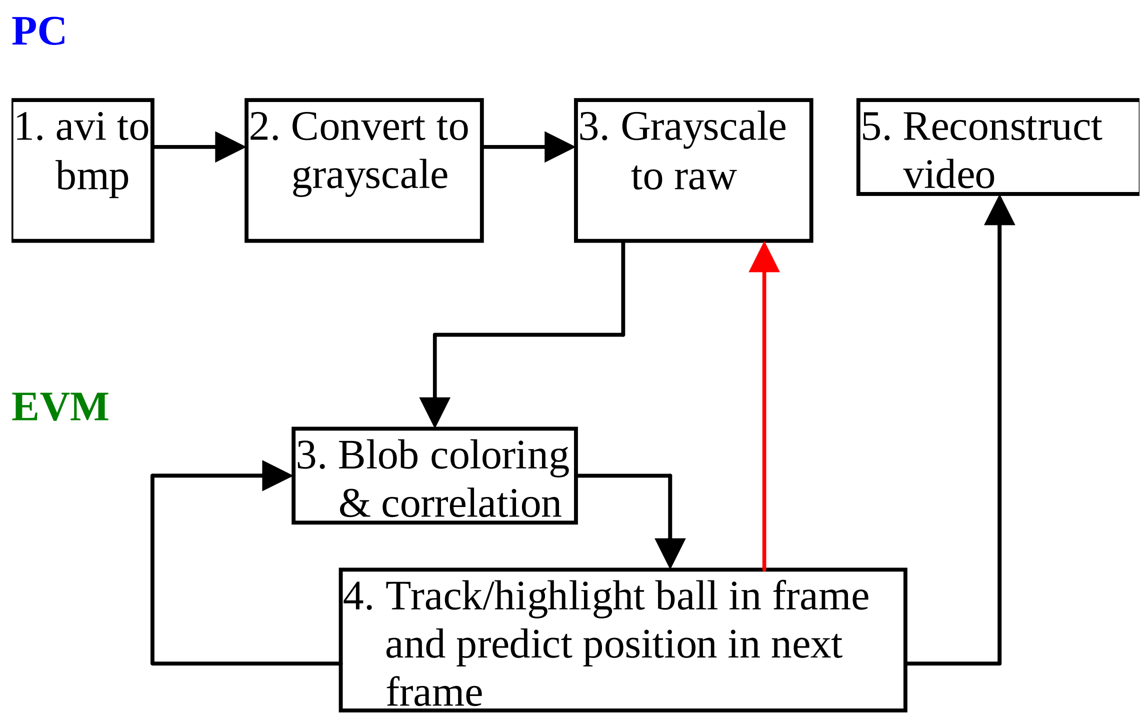
Next we looked into implementing Kalman Filtering. We came across a paper on a trajectory-based algorithm for automatically detecting and tracking the ball in broadcast soccer video, which was almost identical to our project. Instead of looking for features associated with the ball, like in the KLT algorithm, it finds the ball based on its trajectory using the Kalman filter. They claimed that their method achieved 96% accuracy in tracking the ball. Despite the effectiveness of this method, we decided not to implement this algorithm because it was too complicated and we did not think we would be able to successfully apply their method within the time frame of the class.

Process Overview

For our project, the PC side was used to handle AVI files. When called by the EVM the PC would send the appropriate grayscaled image or image section of the frame specified. The EVM would then run all of its processes, which is mentioned in detail below, and return the pixel numbers of which belong to the ball. The PC would then highlight the ball in the non-grayscaled image it stored using the pixel information from the EVM. Afterwards, it added the updated frame to the outputted AVI file.

After sending the pixel information, the EVM would ask for the next frame information. This process would continue until the last frame was called.

The following image is our data flow graph.


PC
1. avi to bmp
2. Convert to grayscale
3. Grayscale to raw
5. Reconstruct video
EVM
3. Blob coloring & correlation
4. Track/highlight ball in frame and predict position in next frame

Algorithms Used

Tracking the ball in the video consisted of several steps. The first step was to locate the initial position of the ball. From there we could use this position to predict the location of the ball in subsequent frames. Once the ball was located in two consecutive frames a velocity could be computed based on the two positions. Using the displacement of the ball a more accurate search area could be created.

PC EVM Communication:

To track the ball using the EVM it was first necessary to get parts of the video in to the memory of the EVM. All memory transfers between the PC and EV were made using HPI transfers. With one exception, the number of frames in the video was sent to the EVM in a transfer mailbox. The PC filled the mailbox with this information as soon as it had opened the video stream. Just before the EVM started its main processing loop it read this mailbox to determine how many iterations to go through. From this point on the EVM initiated all communications. On the PC side a wait function was used to wait for requests, and then serve them. The first communication requested by the EVM was the first frame of the video, either a full 320 x 240 frame, or if an initial position of the ball was given a smaller section. Requesting a frame was accomplished by sending 6 four byte values to the PC using HPI transfer. These 6 values were the starting x and y coordinates of section the EVM was interested in, the height and width of this section, the frame number that the EVM wanted, and a pointer to the section in EVM memory which would hold the frame information. The EVM then waits for two transfers to complete, its request to the PC, and the PC's response. On the PC side once the request from the EVM has been received the first step is to extract the requested frame from the video stream, which will be detailed later. Next the section of the frame requested by the EVM will be grayscaled, also detailed later. Upon completion of the grayscaling process the section requested is sent back to the EVM, to the pointer given in the request.

Locating the Ball:

After the EVM has a frame, or section of a frame the next step is to locate the ball. The location of the ball in a given frame was determined using three steps. First we used thresholding to extract a binary image from the grayscale frame. To perform the thresholding operation the maximum value in the frame was computed and used to set the threshold. Based on initial testing in Matlab the value of the ratio of the maximum to threshold was determined to be 1.45. Accordingly to determine the threshold the maximum value in the frame was divided by 1.45 to get a threshold. Then each pixel was compared to this threshold, those which were great than the threshold were set to 1, those which were less to zero. Thresholding was done in place, the original gray values were written over with the binary values.

Once thresholding was completed blobcoloring was performed. This was accomplished using a recursive algorithm. The image was scanned, one pixel at a time until an on pixel was reached. Upon reaching an on pixel a function was called to color newly encountered blob. The BolbColor function first set the initial pixel to the current blobcolor, which started at 2 for the first blob. The function then checked all eight neighbors of the cell which was it was called for. If the value of any neighbor was 1 the BlobColor function was recursively called on that neighbor. To avoid ping-ponging from one pixel to a neghibor and back the BlobColor function was only called on neghibors that are exactly equal to 1, thus making the previously marked blobs invisible to this function. Once the BlobColor function returned from a blob the area of that blob was checked. To avoid extra processing only blobs of certain sizes were kept. In the initial frame the ball is assumed to be between 5 and 20 pixels. For subsequent frames the ball is taken to be no more than 15 pixels larger, with a maximum size of 40 pixels. The ball is always assumed to be 5 pixels or more, to avoid the noise of small blobs that appear in each frame. Thus, if the ball is found to be 20 pixels in a particular frame no blobs larger than 35 pixels will be considered in the following frame. After the blobs have been colored those not meeting the area restrictions are removed from the image. This is done by calling a recursive BlobErase method. This method takes an initial point in a blob and the numerical “color” of that blob. It then sets the pixel value for each pixel in the blob to 200, so that blobs can be restored at a later time if necessary. Traversing the blob is accomplished in a similar recursive manner to the BlobColor function.

The FindBlob function keeps track of the number of blobs that it has found as well. This is to avoid processing an excessive number of blobs. Only the first 40 blobs encountered are kept as possibly being the ball. This can be done because of the way that the grayscale image is stored. On a Windows system bitmap files are stored such that following the header information is the leftmost pixel of the bottom row of the image. As the camera position is consistently from one side of the field, and always keeps the field in the lower portion of the image we know that if the audience is present in a particular frame they will be at the top of the frame, and thus at the end of the array of pixels. The array of pixels is scanned for blobs from the beginning, and therefore from the bottom of the frame. Because the soccer field has a very low density of blobs it will not have more than 40 blobs, despite its large area. Therefore, we can disregard as audience noise any blobs after 40. In this way we can reduce the interference of the audience. This is essential as various features in the crowd can be mistaken for the ball.

Another source of noise that had to be eliminated was the scoreboard and network label which were superimposed on the upper corners of each frame. These created a more troublesome source of noise than the audience. This is because the blobs on these objects were consistent in size and shape, and if one were mistaken for the ball it could be tracked throughout the entire remainder of the video. To account for this these two objects were isolated in offline preprocessing. The result of this process was a frame which was black at all points not on the labels, and white at all points corresponding to the labels. According to this template the scoreboard and network label are both completely removed from the frame before it is sent to the EVM for processing.

Once all blobs have been found, and inappropriately sized blobs have been removed the blobs are evaluated to determine which one is the ball. There are two different functions used to do this, however both rely on two of the same factors, the height and width of the blob, and the "fullness" of the blob. The height and width are found by determining the maximum and minimum values of the x and y coordinates that make up the blob, then subtracting the minimum from the maximum and adding one, using the maximum and minimum x coordinates for the width, and y coordinates for the height, and adding 1 to this number. Also, as the blob is evaluated a counter is kept representing the number of pixels in the blob. Fullness is calculated as the number of pixels in the blob divided by the area of the bounding box, calculated as the height multiplied by the width. The height to width ratio is also calculated for each blob, to do so height is divided by width. To simplify further analysis it is desirable for this value to be between 0 and 1. If this is not the case the reciprocal of the ratio is used instead. Once all the pixels in a blob have been evaluated the goodness of the blob is calculated. For frames where there is no previous data on the ball goodness is based solely on the ratio and fullness. In this case the goodness value for the blob is calculated with the following formula:

$$\text{Goodness} = \text{Fullness} * (0.8 * \text{Ratio})$$

For cases where the correlation of the current blob and the most recent known shape of the ball is the highest out of all the blobs in the frame 0.2 is added to the

Goodness value of the blob. In cases where the area of the ball in the previous frame is known another value is factored in to the goodness of the blob. In these situations the following formula is used:

$$\text{Goodness} = \text{Fullness} * (0.8 * \text{Ratio}) - (0.005 * \text{abs}(\text{PreviousArea} - \text{CurrentArea}))$$

The same addition is made if the correlation matches for the current blob. The considerations for area are included because the ball is close to a rigid object, thus its area should remain constant, regardless of its orientation because it is a sphere. Unfortunately, the video used to analyze the game shows the ball as blurred when it is moving quickly. Due to this fact the area of the ball changes with its speed. The area consideration therefore assures that the area of the ball does not change too much, and as such is given a very small weight in the formula. The goodness of each blob is evaluated, and the best candidate ball is selected as the blob with the highest goodness.

Highlighting the ball:

After the ball has been selected it must be highlighted for the output frame. This is done by sending the pixels that represent the ball to the PC. However, the pixels that represent the ball in the frame on the EVM are not necessarily the same as those on the full frame. To account for this a FixPixels function was created. Based on the height and width of the current section of the frame, and the location within the section of the ball, the location of the ball in the full frame is returned. These pixel locations are sent to the PC using an HPI transfer. Also sent to the PC is the number of the frame for which the pixel data has been sent. This is necessary because the PC does not keep track of the frame number. Upon being received on the PC the pixel locations are used to modify the outgoing bitmap image. For each pixel that represents the ball the green and red values are set to zero, and the blue value set to 255, or maximum intensity. This modified frame is then put into the video stream, as will be discussed later.

Tracking the ball:

Ball location is a useful tool to have when tracking, however, to make tracking fast and reliable we must not simply look for the ball in each full frame of the video. To improve the speed and reliability of our tracking program we used previous data on the position of the ball to quickly determine its location in the current frame. After the ball is highlighted several parameters are calculated based from the current data. First the exact location and shape of the ball must be determined, so that we may correlate it with the section of the next frame. This is done by analyzing the values in pixels and creating a bounding box on the ball. An array of the size of this box is malloc-ed and the values of the frame for this section are stored to this new array.

After the ball data is stored for the current frame parameters are calculated for the next frame. The first parameters to be set are the x and y coordinates of the ball in the current frame. Next the maximum allowable size for the ball in the next frame is calculated, this value is the current size plus 15 pixels. The minimum size of the ball is always allowed to be five pixels. The next step is to calculate the displacement of the ball from the previous frame. This is done by taking the previous x position, if it is available, and subtracting it from the current x position. The predicted x position is taken to be the current position plus the displacement between the last two frames. A similar procedure is used to predict the position of y. After the predicted positions of the ball are computed a search area is computed. The width and height of the search area are equal to the displacement in the x and y direction, respectively, with a maximum value for height and width of 35, and a minimum of 10. With the size of the search area decided next the position of the upper left corner is calculated.

To be most likely to find the ball we want the predicted position to be in the center of the search area. This is accomplished by making the position of this corner equal to the predicted x position minus the half the width, similarly for y. This being done the x and y values are checked to be sure they remain in the bounds of the complete frame. Finally the global parameters are set to begin the next iteration. Previous positions are set to current positions, the pixels array is overwritten, and the blob color is set back to its initial value of 2. Also, if the ball was not found all parameters are set to their initial or zero values, so that the entire frame will be analyzed to search for the ball.

Speed Issues

Thresholding:

The thresholding function must traverse each element of the section of the frame twice, once as it determines the maximum value, and again as it compares each pixel to the threshold. Factored in to the speed of this function is the location of the section of the frame in the memory hierarchy. This array of pixels varies in size with each successive frame, and there fore cannot be statically defined in on-chip memory, as the maximum size of the frame is larger than the available on-chip memory. Therefore the memory must be allocated with calls to malloc( ). Unfortunately, this places the frame data in the farthest away, and also, slowest section of memory, SDRAM. To counter the fact that constantly accessing many data values in SDRAM would make the function extremely slow we used DMA transfers to get section of the data into on-chip memory for processing. Then, after processing was complete, again used a DMA transfer to send the data back to SDRAM.

Blob Coloring:

Another factor limiting the speed of the implementation is the recursive nature of our ColorBlob function. This implementation was chosen due to its simplicity, it is easy to code since there are no special cases to consider for blobs which are not continuous in each row. Iterative implementations of blob coloring scan each row, and mark blobs as continuous lines of on bits in a row. However, in cases of U shaped blobs bits which are a part of the same blob can appear as separate blobs on certain rows. This means that a more complicated algorithm must be used to check the row above the current row to be sure that these sections are joined. This complication made the recursive implementation more attractive despite its slower running time.

Correlation:

Another speed limiting factor in our tracker is the correlation function. This function is also memory intensive like the thresholding function. However, because of its more complex access pattern it was decided not to use DMA transfers to speed up data accesses. This leads to a relatively time expensive function. However, this cost is offset by the fact that the correlation function is only called on small sections of the image. The section to be considered is considerably smaller than a whole frame for situations where the characteristics of the ball are known; the only time correlation makes sense. Thus instead of performing the correlation over an area of 320 x 240 pixels the correlation is performed on a maximum area of 35 x 35 pixels, reducing the time it takes.

Profile Results:

Due to problems with Code Composer's profiler very little profiling data was able to be obtained. However, the small amount of data that was gathered indicates that there is a great speed gain from using the DMA transfer method in the thresholding function. For a 109 frame video the entire tracking operation took 220 million cycles without DMA transfers. With DMA transfers this number was reduced to 149 million cycles.

Overall speed:

The overall speed of the implementation is rather fast. For cases where the ball is tracked consistently the entire video can be processed in less than the running time of the video, for video at 30 frames per second. Thus, with a few more speed improvements it is theoretically possible to run the tracking algorithm in real time. Also contributing to overall speed is the fact that our EVM code is only 39 KB, allowing it to fit onto the on-chip program memory. Were this not the case instruction fetches would take considerably longer, and slow the program by a great deal.

Other Possible Algorithms:

Based on our observations our over all approach to tracking the ball works very well. The most common source of errors is the program choosing the wrong blob as the ball. The best way to eliminate these errors would be to avoid considering these blobs at all. We considered two main algorithms to do this. First we had considered using morphological processing to try to fully connect the parts of the players so that feet and legs would always be too big to be considered too large to be the ball. Unfortunately, the variability in ball size, and its varying proximity to other objects this method caused more problems than it solved. In cases where the ball was small the structuring element used erode and/or dilate the image had to be so small as to have very little effect. This was to avoid completely eroding the ball away. Also, if the ball was close to a player it would often be merged with that player during morphological dilation. The resulting blob was too big too be the ball, and was thus ignored.

Image differencing was also considered to remove error causing noise. The idea was to subtract the previous frame from the current frame to eliminate the background. This method was effective to eliminate the field from the image, because in nearly every frame it is about the same color, and in the same position. However, the major sources of interference, players and the audience were not eliminated by this method. Because the players run around they are rarely in the same place from frame to frame. The audience is usually in approximately in the same place, however because of the motion of the camera, and the blurred nature of the image in sections corresponding to the audience the pixel values changed considerably from frame to frame over the audience. Under different conditions, such as a stationary camera, frame differencing would be very useful; however, the panning motion of the camera in our sample videos drastically reduces the usefulness of differencing.

Another possible algorithm would have included using the Kanade-Lucas-Tomasi Tracker. This algorithm tracks sets of points that are near big intensity changes form pixel to pixel in adjacent pixels. There are several disadvantages to this algorithm however. First it is very computationally expensive and thus unattractive for a use on the EVM when we are trying for a real time video processing algorithm. Also the properties of the video that we have lead us to believe that to likely would not find good features on the ball. The KLT looks for sharp changes on intensity from pixel to pixel, in the ample videos we have there do not exist around the ball. Due to the motion of the ball and the viewpoint of the camera the pixels which are brightest on the ball are not directly adjacent to green on the field. Instead, the ball blurs into the field over a two pixel range around the ball. These factors lead us to decide against using the KLT feature tracker in our algorithm.

Training and Test Sets

Training and Test Sets were created by using Hauppage WinTV to capture broadcasted soccer games. Ten video clips apiece were taken from two professional soccer games broadcasted by the Fox Sports network. For simplification purposes, all video clips were from the same camera, whose screenshot consisted of field, players, and sometimes the audience. The camera angle was variable as the camera panned the field to follow the ball.

Selections for video were based on the objective of obtaining a collection of video that would represent different scenarios. For simplification purposes, a video clip was taken in which there were few players in the screen with a ball rolling untouched. Other scenarios considered situations in which the audience was included in the screenshot, deflection of the ball occurred causing changes in acceleration and velocity, players surrounded the ball, and player obstruction of the ball.

Our training set consisted of four videos. The following lists the various situations that were tested:

1) Ball rolling by itself
2) Player kicking the ball forward in approximately the same direction as it was moving
3) Player kicking the ball causing it to move in an opposite direction
4) Ball tracking with audience present

Our test set included video clips that contained the same features as above or were more sophisticated in the sense that the clips had a combination of the features.

Video Format

For our project, both the inputted and outputted data was in the form of uncompressed AVI files, compressed AVI files were not used due to its added complexity. During MATLAB testing and preliminary C coding, HandyAVI,[1] a freeware

[1] HandyAVI. <http://www.astroshow.com/handyavi/handyavi.htm>.

program, was used to extract individual frames and store them into bmp files for current use.

To better understand AVI format, NeHe Productions: OpenGL Lesson #35 was used.[2] The following information documents our implementation of retrieving frame information from an AVI, convert to grayscale, and store back into an AVI.

Converting AVI to Grayscaled Images:

Before any AVI functions can be called, the project itself must be a win32 console application that can support an MFC application. In the include statement in the beginning, the header file vfw.h must be included. In addition, for vfw.h to work, the associated library must include vfw32.lib in the links under project settings.

In our code filterpc.c, the openAVI method is called to activate use of AVI. In our openAVI, AVIFileInit is called to begin use of AVI functions. (AVIFileExit is used to terminate AVI function use.) Since our only concern was video and not audio, we then opened the video stream so that we could access the information stored in the video stream. At this time, we also created a new video stream that contained the same stream information as our input to be used later when writing out images.

Our function get_frame handles the actual retrieval of frames. AVIStreamGetFrame is called on the opened streamed video and is returned information that contains an array of information that first contains a header followed by the image information. The header is contains bitmap information (of structure BITMAPINFO) such as the frame's height (biHeight) and width (biWidth). It also contains information on the bitmapinfo such as its size (biSize). Using this information, we set a pointer to point to the array returned by AVIStreamGetFrame incremented by the biSize (to bypass the header information) and are left with the bitmap information.

Problem:

During our testing, we realized that AVIStreamGetFrame did cause one major problem. Since we recorded our information at a frame rate of 30 frames per second and WinTV displayed images at a differing frame rate, certain sequential frames were the

[2] NeHe Productions: OpenGL Lesson #35. <http://nehe.gamedev.net/data/lessosns/lesson.asp?lesson=35>.

same frames. When we tried calling AVIStreamGetFrame on the next frame which was the same as the current one we were on, AVIStreamGetFrame did not rewrite the frame information over the current address as it usually did. Therefore, when we tried to highlight in between frames and then retrieve the next frame which was the same, the ball stayed highlighted. When we tried to perform thresholding the ball would then be lost and the ball would either not be located or improperly detected.

Solution:

To avoid this problem, each new frame image was copied into a new file using memcpy. The size of each new bitmap array created was 320 x 240 x 3. 320 x 240 being our resolution and 3 representing the pixel information of red, blue, and green.

At this point, our image information is contained in 24-bit bitmaps to handle red, blue, and green. To convert these images to 8-bit grayscaled images we used C code from a Raster Data Tutorial webpage. [3] This code uses the formula:

grayValue = 0.299*redValue + 0.587*greenValue + 0.114*blueValue

to evaluate the corresponding gray value. This formula represents how the human eye does not observe all colors equally. An important thing to note is that, Windows stores bitmap information as blue, green, red instead of red, green, blue. This explains why the code first calls the blue value character then green and then red.

The final gray value is stored into a new array. Whereas in RGB, three elements in an array represented a pixel, the grayscaled image only uses one element represent a single pixel.

We use the array information generated here to pass into the EVM. For proper use, it must be noted that Windows stores images backwards, from bottom left to right. Therefore the lower indices start from the bottom left corner and the higher indices are in the upper right corner.

Highlighting Bitmaps:

[3] Raster Data Tutorial. <http://www.pages.drexel.edu/~weg22/colorBMP.html>.

Upon return of the pixel information from the EVM, the highlighting of the ball was done on the original pointer to the bitmap (the non-grayscaled information that was memcpy earlier). Using the same blue, green, red color information, we set corresponding pixels' blue information to 255, green to 0, and red to 0 to achieve the desired blue highlighting effect.

Writing Bitmap to AVIs:

After each frame is evaluated, the frame is written into the new video stream (mentioned above) by first setting the stream format (calling AVIStreamSetFormat) which takes the BITMAPINFOHEADER variable that we first set when we opened and retrieved the first frame information. The actual writing of the bitmap to the stream is done by the AVIStreamWrite function which takes in the bitmap copy and writes it to the specified video stream. [4]

Once all frames are evaluated and been written to streams, both streams must released properly, using AVIStreamRelease. AVIFileExit is used to end the use of AVI functions.

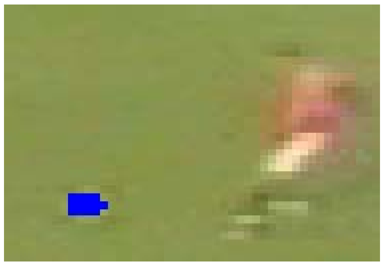
<u>Results Analysis</u>

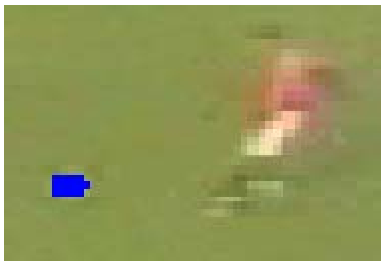
Typical Results:

In a typical sequence the ball is found in a frame, the ball is highlighted; its position compared to the position in the previous frame, and finally, the next section to search is chosen. In most cases this works well, as shown from the transition from frames 33 to 34 in a sample video.

[4] Video for Windows API. <http://www.mapletown.net/~nekora/soft/howto/avi.html>.

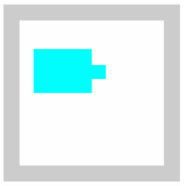
Frame 33: **Frame 34:**

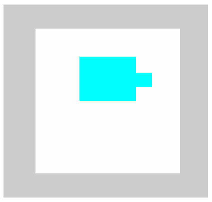




In this case the ball is found it the expected area, both areas are 10 x 10 pixels:

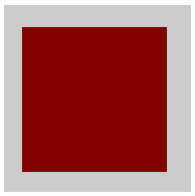
Frame 33: **Frame 34:**

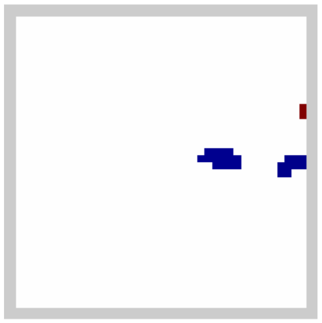




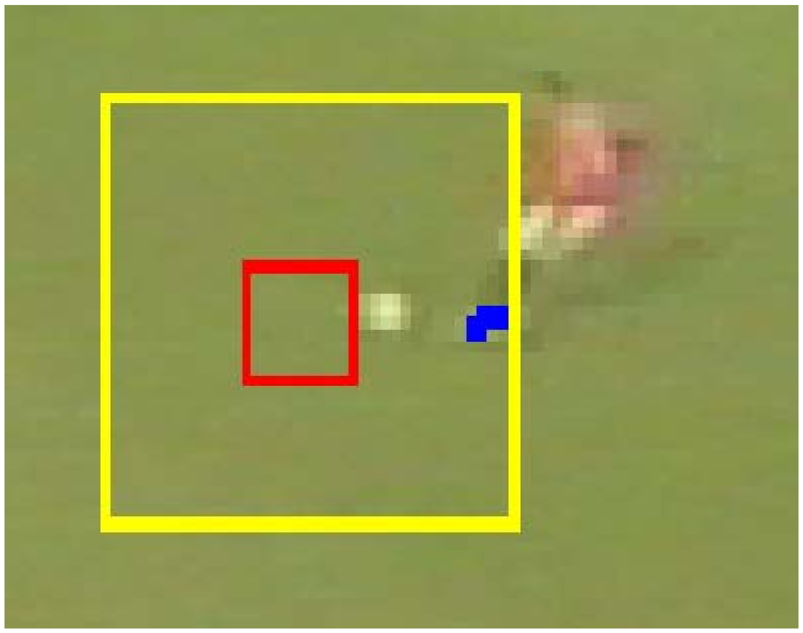
However, because of motion of the camera, and changes in speed in the ball it is not always found in the expected 10 x 10 section, such as the next frame in this video, frame 35. To account for this our algorithm will take a larger area, 40 x 40 pixels to look for the ball.

Frame 35, 10 x 10: **Frame 35, 40 x 40:**





The red in the small section indicates that the entire frame is one blob, far too big to be the ball. As shown in the following picture the original search area (small red box) does not include the ball at all. The subsequent attempt to find that ball (large yellow box) succeeds in enclosing the ball within the search area; however the algorithm chooses the foot of the player as more like a ball than the actual ball. This illustrates one of the many possible problems encountered in trying to track the ball.



Errors:

Despite our best efforts, errors still occurred sometimes, due to the numerous scenarios possible. The following lists the errors that occurred given our test set.

<u>1. Ball obscured by body</u>

If the ball happened to roll behind a player and could no longer be viewed, the program generally detected a part of the player's body as the ball and highlighted it. Proper tracking resumed most of the time after the ball returned to view. Given the constraints of our program, this cannot be avoided, our program is, however designed to recover from such instances and resume tracking. A possible solution to this problem is to track both players and objects in a frame, to better keep track of the key objects in the screen, and highlight accordingly.

<u>2. Audience head was better shaped than ball</u>

This was a problem that only occurred when analyzing the first frame. In certain frames, the ball's height-width ratio would not be equal to 1 whereas a member of the audience’s head would be. Thereby, the audience member’s head would be selected as the ball.

To workaround this problem, our program allows a user to set initial constraints for an area to look for the ball in.

3. Ball was too small and deleted by blob constraints

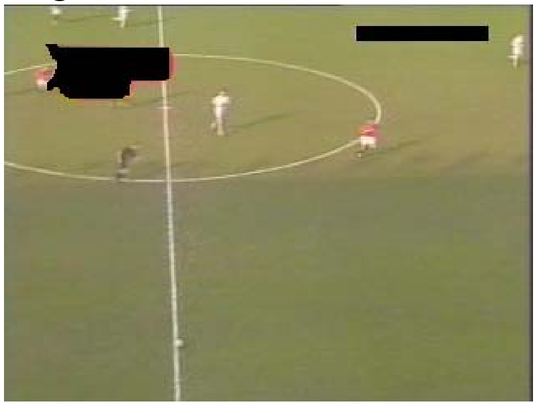
Blob constraints were implemented so that objects too small or too large were not evaluated. Although the maximum blob size could be vary from 20 to 40, depending upon the speed at which the ball was traveling, the minimum size that a blob could be accepted was 5.

Based upon shading effects caused by the sun and camera angle, thresholding sometimes creates a ball blob that is smaller than our acceptable area of 5. Generally, the case was when the ball area was 4. If we were to lower our minimum ball area to 4, however this would cause more random tiny blobs in the audience to be accepted. Therefore this error was left as is, since the probability of this occurrence was slim.

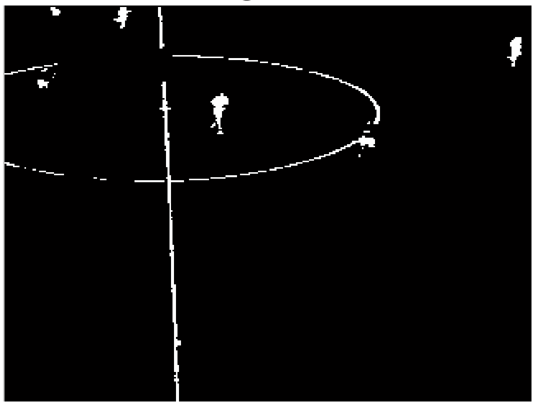
4. Ball blob crossed line and became lost

In one test video, our program successfully tracked the ball freely rolling, but failed to locate the ball once it crossed the line. Examining individual frames, it was found that the ball was rolling at such a pace that in 4 sequential frames, the ball was touching the line. This caused the ball to be included in the line blob and eventually deleted from consideration as a ball.

Original Video Frame



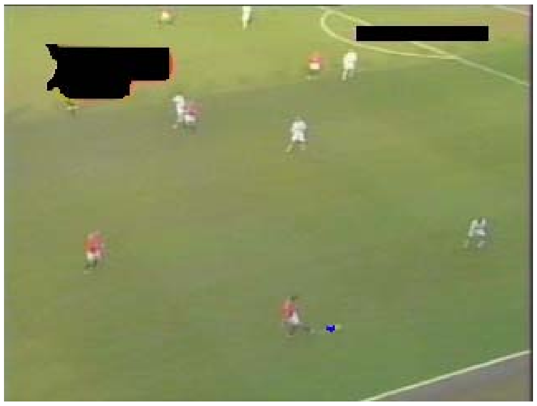
Thresholded Image with Ball Lost



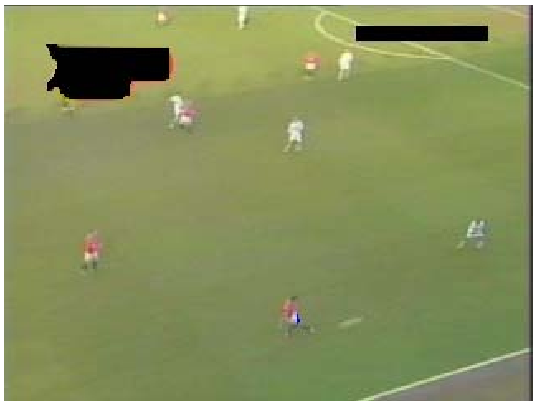
5. Ball was traveling so fast that it was no longer ball shaped but a stretched oval

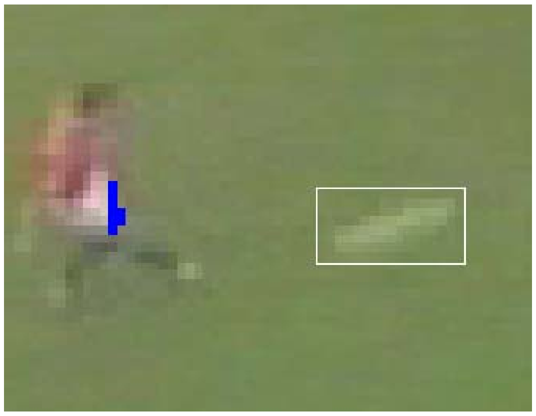
In this case, we were able to track the ball until two frames after a player kicked and accelerated the ball. As seen in the example, the program was able to successfully track and highlight the ball in frame 53, but lost the ball and chose a part of the player's body instead in frame 54.

Frame 53:



Frame 54:





Upon closer examination, it is apparent that the ball is no longer circular shaped but has stretched into what appears to be a thick line (refer to figure – ball is noted by box). This shape would be incorrectly profiled into our function as not being the ball, because of its height width ratio. Despite our best efforts to predict such morphing through correlation and evaluating the ball shape in the previous frame, this error in results could not be avoided.

C code

EVM Side

- Thresholding/Velocity application/Blob Analysis

    - Developed our own code for thresholding, velocity application, and blob analysis, applying the algorithm information listed above into C code
- Blob Coloring
    - http://www.ittc.ukans.edu/~jgauch/research/kuim/html/src/blob_color.c
    - *Content:* C code for blob coloring in 2-Dimensions and 3-Dimensions through recursive functions. Also included functions on erasing blobs in both 2-D and 3-D.
    - *Usage:* The code was used in the EVM side to generate blobs from our thresholded images. Only the 2-D functions were used (BlobColor2D and BlobErase2D). After testing, we kept the 8-bit blob coloring (recursively call cells to the North, Northeast, East, Southeast, South, Southwest, West, and Northwest) that was originally provided rather than 4-bit blob coloring (recursively call cells to the North, East, South, and West).

PC Side

- MSDN Library. "Reading from One Stream and Writing to Another."
    - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/multimed/htm/_win32_reading_from_one_stream_and_writing_to_another.asp
    - *Content:* Tutorial on reading a stream in AVI and writing to another stream
    - *Usage:* The actual code was not used. This webpage served more of as a reference to how we should structure our bmp2stream function.
- "Raster Data Tutorial."
    - http://www.pages.drexel.edu/~weg22/colorBMP.html
    - *Content:* Tutorial on 24-bit BMP format. Addresses raster data location (raw image information) and modification of information to produce 8-bit grayscaled BMP.
    - *Usage:* The code was used in the PC part of our program. We altered it so that the writing out of grayscaled BMPs was optional for debugging/testing purposes.
- "Video for Windows API."

- http://www.mapletown.net/~nekora/soft/howto/avi.html
- *Content:* Tutorial on saving bitmaps to video stream and writing out AVI files
- *Usage:* Despite the documentation for this code being in another language, the actual code itself was very useful in creating video streams, writing to streams, and writing out AVI files.

- NeHe Productions. "OpenGL Lesson #35."
  - http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=35
  - *Content:* Tutorial on retrieving and displaying frame information from AVI files. Details how to open AVI video streams, retrieving header information, and grabbing individual frames.
  - *Usage:* A modification of this code was used to open AVI files and create a function the grabbed a frame, provided a frame number.
- Spring 2004, 18-551 Lab 3
  - *Content:* C code for PC to EVM communication using HPI transfers
  - *Usage:* Used this code as a template for our code to set up HPI transfers

References

- Wilson, Funston, & Darby. "Surprise Visual Object Tracking."
  - http://www.doc.ic.ac.uk/~sdf00e/visual_files/frame.htm
  - *Content:* General information regarding various methods and usage for object tracking.
  - *Usage:* Used for general object tracking understanding.
- Yu, Xinguo, Changsheng Xu, Hon Wai Leong, Qi Tian, Qing Tang, and Kong Wah Wan. "Trajectory-based ball detection and tracking with applications to semantic analysis of broadcast soccer video." International Multimedia Conference.
  - http://portal.acm.org/citation.cfm?id=957018&dl=ACM&coll=GUIDE

  - *Content:* Trajectory-based algorithm that determines whether or not an object is a ball based upon its trajectory. Used the Kalman filter for its candidate verification feature.
  - *Usage:* Compared their trajectory-based method to our object-based method to evaluate potential features in our program.
- KLT: An Implementation of the Kanade-Lucas Tomasi Feature Tracker.
  - http://vision.stanford.edu/~birch/klt/index.html
  - *Content:* C sample code and description on using the KLT algorithm.
  - *Usage:* After further research into uses and application of the KLT algorithm, we decided not to use KLT in our image processing.
- HandyAVI Software
  - http://www.astroshow.com/handyavi/handyavi.htm
  - *Content:* Free software that converts AVI files into individual bitmap frames.
  - *Usage:* Used in preliminary Matlab testing to generate BMP files. This was before we were able to create our own frame extraction function in C code.