Object Tracking and Solving Image Puzzles using CmuCam2+ Vision Sensor

Ramazan S. Aygün
University of Alabama in Huntsville
Computer Science Department
College of Science,
Technology Hall, N360, Huntsville, AL 35899

Abstract

Object tracking is the ability to take an image, isolate a particular object and extract information about the location of a region of that image that contains just that object. Solving image puzzles is the problem of computing the similarities between consecutive frames, alignment of the frames based on these similarities, and then blending the image pixels on a region named as sprite or mosaic. This paper combines both researches by using CmuCam2+ vision sensor. Reduction of redundancy carries important role for especially for mobile environments where the energy of the system is limited. As a part of this research, a traxter robot is built using Serilazer .NET robot controller to track color objects using CmuCam2+ vision sensor. In addition, the images that are captured from CmuCam2+ are merged to give a larger view of the environment. To handle this, a revised hierarchical translational motion estimation has been used.

1 Introduction

Real-time image processing is one of the hot research problems nowadays. Especially, video surveillance systems require real-time storage as well as real-time processing of the captured data. The cameras may be mounted on a hall for security or under unmanned aerial vehicle for information gathering. If the camera is mobile, the region that is tracked may appear in multiple frames due to similarity between the consecutive frames. This redundancy may be reduced by solving image puzzles. Solving image puzzles is the process of aligning the common areas in the consecutive frames and then building a big picture of the environment. CmuCam2+ vision sensor is used to capture images. Cmucam2+ vision sensor is also used for color object tracking. A robot to track an object based on its color characteristics has been built. Before providing our approach, brief related work on solving image puzzles and object tracking is provided.

1.1 Solving Image Puzzles

The solving image puzzle problem can be related to research on background generation [Lu, 2003], panoramic image generation [Szeliski, 1997], mosaic generation [Peleg, 2000], and sprite generation [Farin, 2006]. The sprite generation has been an important component of the video compression standard MPEG-4 [Dufaux, 2000], [Sikora, 1997]. The current systems implement simpler versions of MPEG-4 without sprite coding. A camcorder is used to capture the environment by panning and tilting the camera. As a result, a video that is composed of frames (images or pictures) is generated. Since capturing is a continuous event, the overlapping areas in consecutive frames are determined. Each image in the video is aligned on top of the previous image such that overlapping areas in two images map to each other. By combining and merging all images in the video, the big picture of the environment is extracted. The sprite generation has 3 major steps: Global Motion Estimation, Warping, and Blending. Global motion estimation estimates how much camera moved in consecutive images of a video. Warping aligns the images on top of each other such that overlapping areas are aligned. Warping may resize and rotate the images so that images are aligned. Blending decides how to integrate the values on the overlapping areas.

1.2 Object Tracking

Object-tracking is one of the important problems in image processing. Tracking can be defined as the problem of estimating the trajectory of an object in the image plane as it moves around a scene. Object tracking is the ability to take an image, isolate a particular object, and extract information about the location

of a region of that image that contains the object. Furthermore, given multiple images at different times from the same or similar view points, it should be possible to isolate objects that may have moved. Object tracking has very important applications [Merl, 2004]: security & surveillance, medical therapy, retail space instrumentation, video abstraction, traffic management, video editing, and interactive games. The tracking methods can be broadly divided into three main categories [Yilmaz, 2006]: point tracking, kernel tracking, and silhouette tracking. In *point tracking* method, object tracking is considered as a correspondence problem in which the detected objects are represented by points across frames [Shafique, 2003], [Veenman, 2001]. In *kernel tracking*, the conformal, translational or affine motion of the object in subsequent frames is computed. In *silhouette tracking*, the objects that cannot be easily represented by simple geometric shapes are given an accurate shape description [Sato, 2004], [Chen, 2001].

1.3 Our Approach

We have used object tracking based on color characteristics using CmuCam2+ Vision sensor [Rowe, 2003], since color is one of the most robust methods of tracking if the color of the object can be distinguishable from the background. The vision sensor has been installed on a traxter robot so that the robot can follow a colored object. The same vision sensor is again used for solving image puzzles. The motion between consecutive frames is estimated using a hierarchical global motion estimation algorithm. However, due to low resolution of the images, the search space to compute the motion vectors turned out to be larger than the traditional approaches. The frame-rate of CmuCam2+ vision sensor was not as high as expected. Our results show that our approach provides robust results at low resolution and low frame rates.

This paper is organized as follows. The following section introduces CmuCam2+ vision sensor. Our color tracking methodology is explained in Section 3. Section 4 explains the communication with the camera. Solving image puzzles is discussed in Section 5. The last section concludes our paper.

2 CMUcam2+ Vision Sensor and Color Tracking

CmuCam2+ vision sensor is very similar to CmuCam2 sensor. CmuCam2+ vision sensor is smaller in size and has no serial port. It has a TTL port. This TTL port can be transformed from TTL to serial port by a converter. CMUCam2+ has the following properties: track user defined color blobs at up to 50 Frames Per Second (fps); track motion using frame differencing at 26 fps; find the centroid of any tracking data; gather mean color and variance data; gather a 28 bin histogram of each color channel; process horizontally edge filtered images; transfer a real-time binary bitmap of the tracked pixels in an image; arbitrary image windowing; image down sampling; adjust the camera's image properties; dump a raw image (single or multiple channels); up to 176 x 255 resolution; supports baudrates of: 115,200 57,600 38,400 19,200 9,600 4,800 2,400 1,200; automatically use servos to do two axis color tracking; B/W Analog video output (PAL or NTSC); and multiple pass image processing on a buffered image. The CmuCam2+ Vision sensor of our robot is shown in Figure 1.

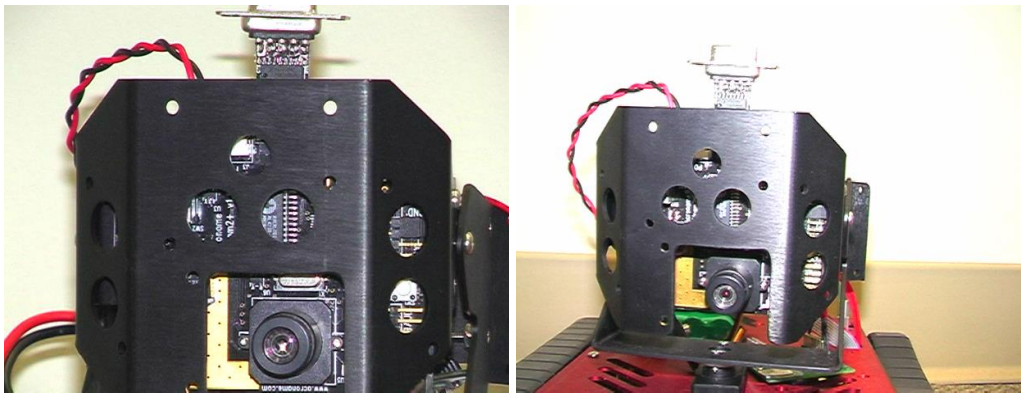

Figure 1. CmuCam2+ Vision Sensor

Color tracking is the ability to take an image, isolate a particular color, and extract information about the location of a region of that image that contains just that color. For example, assume 64x64 Lena image against black background where the objective is to track the Lena image and adjust the robot movement and the camera motion such that the Lena image is always centered in the image frame.

In CMUcam2+, each of the red, green and blue channels is converted into a number in the range 16 and 240. The upper limit and lower limits on each of the red, green and blue channels are to be specified to efficiently track the color space. The histograms are used in obtaining the upper and lower bounds for each of the red, green and blue channels. Hence six values are to be used to constrain the entire color space that is being tracked. The CMUcam2+ takes these bounds and processes the image. The CMUcam2+ uses a simple one pass algorithm that processes each new image frame from the camera independently. It starts at the top left of the image and sequentially examines every pixel row by row. If the pixel it is inspecting falls inside the range of colors specified, it marks that pixel as being tracked. It also examines the position of the current tracked pixel to see if it is the top most, bottom most, left most or right most position of the entire tracked pixels found thus far in the image. If it finds that the pixel is outside of the current bounding box of the tracked region, it grows the bounding box to contain this new pixel. Because the location of even a single tracked pixel can change the bounding box, the bounding box can sometimes fluctuate quite a bit from frame to frame.

3 Color Tracking Methodology

There are three components of color tracking: identification of the colors of the object, interpreting tracking data from CmuCam2+, and sending commands Serializer .NET robot controller to track the object.

3.1 Determining the Color of an Object

Light is not perfectly uniform as well as the color of an object. These variations need to be accommodated. However, these bounds cannot be relaxed too much, since many undesired colors may be accepted. Because of this, CmuCam2+ accepts a range of values for each channel: red, green, blue. It is very hard to estimate the red-green-blue (RGB) values of a colored object. Java CmuCam2GUI is used to determine the range of colors. This GUI allows grabbing frames and visualizing the pixel values. When the mouse pointer is moved over the colored object, GUI provides the colors on the object.

The next step is to verify whether we have chosen the correct colors or not. The track color option of CmuCam2GUI is used to verify the colors. Given upper and lower bounds for each channel, the CMUcam2+ is commanded to identify the region of the object. In Figure 2, sample results of tracking are shown where rectangular blue box provides the bounding box of the object.

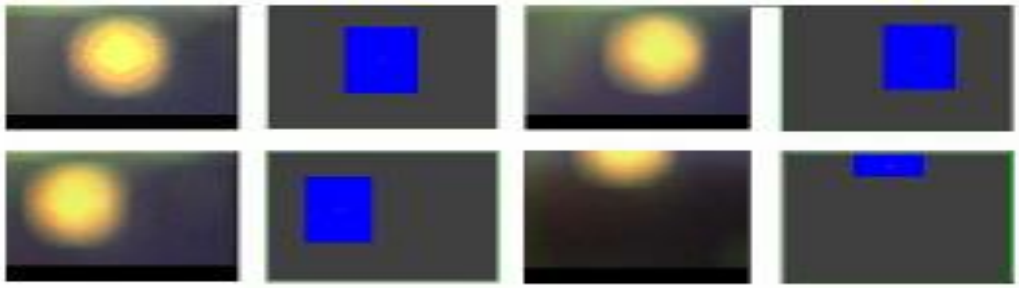

Figure 2: First row (left-to-right): reference image, tracked object, reference image when camera moved right, tracked object. Second row (left-to-right) reference image, tracked object, reference image, tracked object

3.2 Interpreting Tracking Information from CmuCam2+

After determining the range of colors, the range of colors is provided to CmuCam2+ using the command: "**ST** Rmin Rmax Gmin Gmax Bmin Bmax \r". If this command is followed by TC command, the tracking starts. The tracking can also be set and started with TC command by providing range of colors to TC command as: "**TC** Rmin Rmax Gmin Gmax Bmin Bmax \r".

TC command returns a T packet in the following format: "*T mx my x1 y1 x2 y2 pixels confidence\r".* T represents that the data that is sent is a T packet. The values x1, x2, y1, and y2 represent the borders of the bounding box of the object. The values x1 and x2 represent the leftmost border and rightmost border of the bounding box, respectively. The values y1 and y2 indicate the top and bottom borders of the bounding box, respectively. The pair (mx, my) represent the centroid of the rectangle. In our experiments, it is observed that mx=(x1+x2)/2 and my=(y1+y2)/2. This information helped us verify whether the data is read from CmuCam2+ properly or not. The value *pixels* returns the number of pixels that are tracked. The confidence value indicates the confidence of tracking the object. This confidence value is used to determine when to start the traxter (robot). When tracking starts, it takes time for CmuCam2+ to capture the bounding box properly. In the beginning of tracking, confidence is either 0 or less than 50. The confidence is expected to be at least 50 to start tracking.

3.3 Controlling the Robot

While tracking the color object, a continuous and steady motion by the robot is expected. In other words, the robot should continuously move towards the object as fast as possible. As stated in the previous section, the parameters that are used for color tracking are the centroid of the bounding box and the confidence. The robot actually starts tracking when the confidence is high enough. There are two parameters for the centroid: mx and my. The parameter *mx* represents the centroid in the x dimension (or horizontal dimension) and the parameter *my* denotes the centroid in the y dimension (or vertical dimension). The robot can only move in the horizontal dimension if two dimensions are available (horizontal and vertical). The robot cannot change its elevation. Therefore, the parameter *mx* is used in our experiments in terms of centroid.

One difficulty that is faced during our experiments is the optimizations that are performed by CmuCam2+. Although the images (or frames) that are captured from CmuCam2+ have size of 176x144, actually CmuCam2+ sends data as 88x144. Instead of sending values for every column, CmuCam2+ skips a column and sends the values for the next column. When the image needs to be generated, the values of each column are repeated. Hence the *mx* parameter's maximum value becomes 88 rather than 176. To centralize the colored object, the *mx* parameter must be close to 44 rather than 88. Since this was not clear in our earlier experiments, the robot was usually turning away from the object.

The *pwm* (power motor) command is used to give directions to the robot. There are only two motors that control each side of the robot. The *pwm* command expects the speeds of each motor as a parameter. To turn in one direction, one of the values is set to a negative value and the other one is set to a positive value. For example, "pwm 1:10 2:-10" commands the robot to turn left whereas "pwm 1:-10 2:10" commands the robot to turn right.

As stated earlier, a continuous motion by the robot to track a colored object is desired. Initially, the parameter *mx* is compared with 88/2=44. If *mx* is less than 44, the robot is expected to turn left, otherwise it is expected to turn right to centralize the object. However, such a comparison slowed the motion of the robot continuously. The robot was taking a small step forward and then turning either left or right to centralize the object. When it turned left, the centroid stayed on the right; and when it turned right the centroid stayed on the left. The number of left-turns and right-turns had to be minimized for continuous motion. To do this, the condition for turns has been made flexible. If the centroid of the object is really far away from the center, then the robot makes a turn. If *mx* is less than 24, the robot makes a left turn. If *mx* is greater than 64, the robot makes a right turn. In other words, it is assumed that the object is in the center if mx is within at least 20 pixels away from the center. In this version of our experiments, the stopping condition of the robot is decided as the disappearance of the colored object.

Several videos are captured for color tracking. These videos are available at http://www.cs.uah.edu/~raygun/colortracking.htm website. It can be seen that our robot is able to track static objects as well as mobile objects. From these videos, it is also noticeable that our robot has a continuous motion towards the colored object. The pictures of our robot are shown in Figure 3.

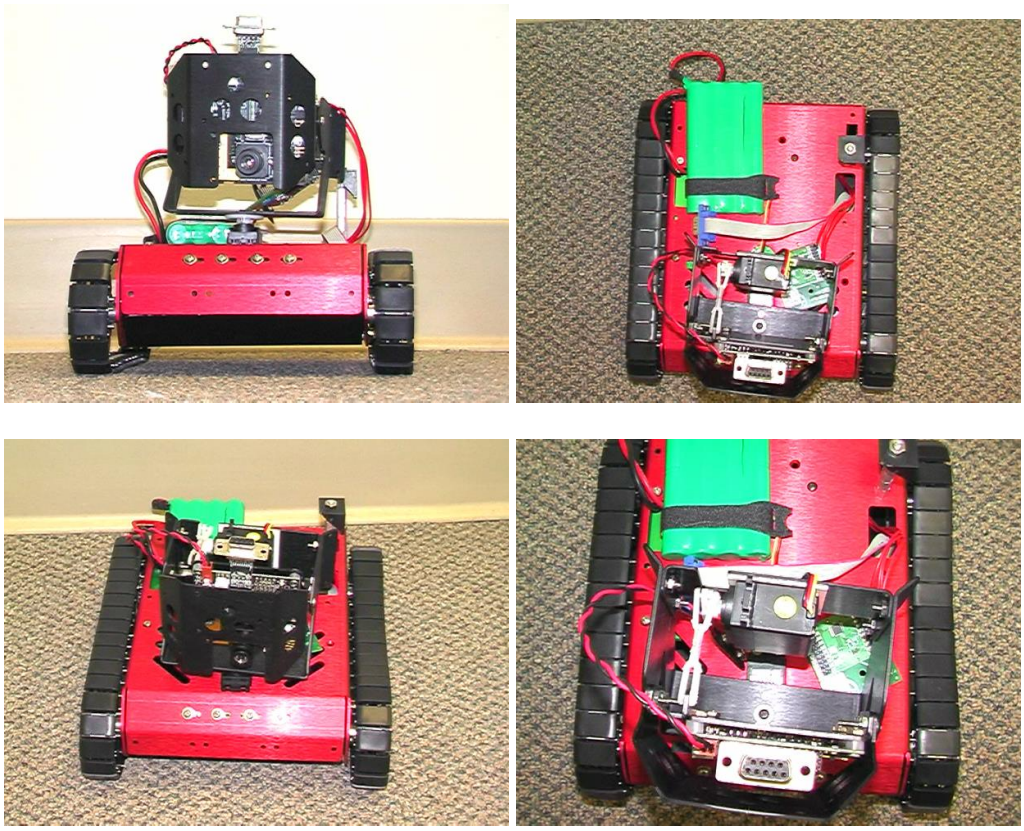

Figure 3. Our traxter robot

4 Setting-up the Camera and Communicating with the Camera

CmuCam2+ does not come up with a serial port. However, it has a port that supports TTL logic. To convert TTL to RS-232 we use a serial interface connector. On one end of the connector, there is a 4-pin female connector. This is connected to the camera board.

4.1 Serial Communication with CMUCam2+

The camera communicates using serial RS-232 cable. The communication with CmuCam2+ should have been established in C# using .Net. Initially, Microsoft Robotics Studio is used to experiment our robot. In the early stages of this project, the Microsoft Robotics Studio had version 1.0. After completing several phases, Microsoft Robotics Studio released new version 1.5. Some compatibility issues between these versions have been experienced. Our running code with version 1.0 stopped running with version 1.5. Microsoft Robotics Studio requires *manifest* file for each hardware. They had some sample manifest files for Serializer .NET robot controller available by roboticsconnection.com. However, there was no available manifest file for CMuCam2+. Moreover, it has also been realized that Microsoft Robotics Studio is good for experimenting hardware but it has limitations to perform a full-fledged research due to limitations by manifest files.

There was code available only in Java as a GUI for accessing the robot. As stated before, this was not suitable since Serializer .NET was compliant with C#. The necessary components of the Java files had to be converted into C#. Java GUI was studied and analyzed carefully to map it into C#. The serialport java file is not needed in C# .NET since .Net itself has a serial port package called IO.Ports which can be used to set up serial object. The major components of our system are listed as follows:

(a) *Serial Communication (serialcomm.cs):* This deals with defining a serial port object, reading data from it, writing data to it. The exceptions are handled. When opening the serial port the parameters to be passed are 115,200 baud, 8 data bits, 1 stop bits, no parity, and no flow control.

Constructor **SerialComm** opens the serial port with the specified parameters mentioned above. From this point on, assume that the serial port is represented with object *sp*. Function **ReadByte** reads a byte from the camera using sp.ReadByte. The exceptions are checked (.Net already has timeout exception. So, timeout does need to be checked separately). Function **writestr** writes a byte to the camera. It takes in a string as a parameter and using the function **StrToByteArray,** it converts the string into bytes and stores it in a byte array. This byte array is then written to the camera.

(b) *Communicating with the camera (cameraserial.cs):* This includes the above class for serial communication. The class serialcomm can be thought as a class that implements 'primitive instructions' such as readbyte, write byte whereas cameraserial class actually sends commands to the camera using primitive instructions defined in serialcomm class. Constructor **CameraSerial** has two phases. In the first phase, the serial communication object is defined and opened as an instance of serialcomm class. In the second phase, the commands are given to the camera that resets it and gives back the version and type.

(i) '\r': This command resets the camera and gives ACK on success and NCK on failure. It then waits for further commands by showing a colon **':'.** So our code first writes \r to the camera using writestr (in serialcomm). After writing this, the camera if successful would have end on a colon (waiting for further commands). Thus after writing \r, a byte is read continuously from the camera until a colon is detected. The line of code that corresponds to this is 'While data! = 58, readbyte'. The ASCII value for colon is 58.

(ii) After this, the camera version and type needs to be determined. From the manual, it can be seen that the command 'RS\r' resets the vision board and returns the camera version and type.

This can be illustrated as follows:

: RS
ACK
CMUcam2 v1.0 c6
:

In this case, 2 is the camera version and 6 is the type. Thus after writing 'RS\r', bytes are read from the camera until 'm' is detected. 'm' is before the camera version. After this the next byte is checked whether it is 1, 2 etc. For Cmucam2+, 2 is expected. Once camera version is confirmed, bytes are read from the camera until camera type is detected (i.e. bytes are read until 'c' is encountered). The next byte is the camera type.

4.2 Converting Data Packets into Image

The data obtained from the camera is in form of packets. The type and content of packets depends upon the command sent to the camera. Just as **'RS\r'** resets the camera and sends out the type and camera version, **'SF [channel]\r'** command needs to be sent to get the image in form of packets. Then the packets need to be converted into to the image. This is done in Java GUI code, but it is spread through different segments. Most of our code is put together as one logical stream of code. The major class for reading image data is maintained in ImageCreation.cs file.

ImageCreation class is responsible for creating an image object and initializing the pixel values. The command sf/r givea the data in the format as a Type F data packet that has the following format:

1 - new frame followed by X size and Y size
2- new col
3 - end of frame

RGB (CrYCb) values range from 16 to 240. The data appears in the following format:

1 xSize ySize 2 r g b r g b ... r g b r g b 2 r g b r g b ... r g b r g b 3.

This information is decoded in this class and then converted to image. In this class, the command 'SF/r' is written to the camera. On successful reception of this command, the camera returns 'ACK'. The next byte is checked whether it is 1 (i.e. this means the start of data bytes as mentioned above). Another byte is read. If data is '0', then the frame grabbing fails. If data is '1', then the height and width are stored. If data is '2', it indicates the start of new row. If data is '3' then the code has successfully read the bytes. If data is neither of '2','3' or '1' then it means that the camera is still sending bytes of data for the current row. This is verified by checking whether new-frame variable is 1 or not.

The pixel information is stored in such a way that one 'RGB' corresponds to 2 pixel values. Hence upon a receipt of 'RGB' these values are assigned to 6 positions in the bytes array: first for 'red', second for 'blue', and third for 'green'. The same pattern is repeated for the next three positions. Detailed information about this can be found in CMUcam2+ manual. Finally, the image is written to a file in PPM format.

4.3 Image Processing Application Programming Interface

Image processing application programming interface provides the necessary functions for image processing such as color model conversion, file operations, and reading series of images from CmuCam2+. This API is written under class ImageManipulation. In this section, a set of functions that are built for image processing are briefly described.

The functiopn setnoOfFrames(int) determines the number of consecutive frames to be grabbed from CmuCam2+. The function setFileHeader(String) determines the header of the file name to write the grabbed images. The function setActualWidthHeight(int, int) sets the actual height and width of the images that are grabbed from the CmuCam2+. The function initialize() allocates the necessary storage to maintain the images in the main memory. The function getFrames reads a set of number frames set by setnoOfFrames(int) function into the buffer.

There are four functions available for file operations: readRGB(String, int[,,]), readRGBFiles(String, int), writePPMFiles(), and writePPMImage(String, int[, ,], int, int). The function readRGBFiles reads a sequence of images from files given a header of file names. The function readRGB reads a single image from a file in RGB format and is invoked by the readRGBFiles function. The function writePPMFiles writes a sequence of images to the disk in PPM format. The function writePPMImage writes a single file in PPM format to a file and is invoked by the function writePPMFiles.

Our API also supports color model conversion between RGB and YUV models. The function rgbToyuv converts an image in RGB format into YUV format. The functions getRfromyuv, getGfromyuv, and getBfromyuv extract R, G, and B components, respectively, from Y,U, and V values. Among these color components Y component has significance over the others. Y component indicates the luminance or gray-scale values of the image. To improve performance, most computations are only performed on one channel (i.e. Y component). For that purpose, we have two functions: createYImages(int[][,]) and createYImageFromRGB(int[, ,], int[,],int,int). The function createYImageFromRGB converts a single RGB image into Y image whereas the function createYImages(int[][,]) creates Y images from a sequence of RGB images.

The function downSample is used to generate the lower resolutions of an image. The function generateMeanVarianceFeatures(int[][,], int[][][,], int[][][,], int[][][,], int, int) extracts the mean and variance values of pixels in 8x8 block. There are several functions specifically for solving image puzzles such as initializePuzzle() and solvePuzzle(). This will discused in the section of Solving Image Puzzles. A code snipplet is provided how simple to retrive images from CmuCam2+. The following code retrieves 5 images from CmuCam2+ and writes into files having file header as cmutest..

```
imageManipulation.setnoOfFrames(5);
imageManipulation.setFileHeader("cmutest");
imageManipulation.initialize();
imageManipulation.getFrames();
```

imageManipulation.writePPMFiles();

5 Solving Image Puzzles

Solving image puzzles have three steps: motion estimation, warping (alignment), and blending. We have used images that are generated by CmuCam2+ using our image processing API. However, two problems are faced with when using CmuCam2+. One of them is that images captured by CmuCam2+ have very low resolution. The second is that CmuCam2+ is extremely slow for image grabbing. It took several seconds to grab an image. Normally we expect to get at least several images per second for proper video processing. Because of this, it can be stated that CmuCam2+ is better for tracking of objects.

The motion estimation is kept as simple as possible. It is assumed that the motion of the camera can be represented with translational motion model. Hierarchical motion estimation is applied for estimation of the motion. For solving image puzzle, a class named ImagePuzzle is created. The class ImagePuzzle has two functions related to extracting features, three functions for alignment, two functions for motion estimation, two functions for error computation for each block during motion estimation, and one function to generate low resolution images. The functions for extracting features are extractMeanVarianceFeaturesForLevel(int[,] frame, int[][,] mean, int[][,] variance, int level, int width, int height) and extractMeanVarianceFeatures(int[,] frame, int[][,] lowpass, int[][,] mean, int[][,] variance, int width, int height). These two functions basically extracts mean and variance for each block at each level. The second function calls the first function for each level. In our experiments, there are only three levels. The function downSample(int[,] frame, int [][,] lowpass, int width, int height) creates lower resolutions of the original image.

The motion estimation is perfomed on all consecutive frames by calling the function estimateMotionFrameSequence(int[][,] framesY, int[][][,] lowPass,int width,int height,int noOfFrames,float[][] motionVectors). This function calls estimateMotion function to determine the motion parameters between two consecutive frames. The function coarseMatch(int[,] lowPass1, int[,] lowPass2, int width, int height, float[] a, int[] searchBoundary) finds the best matching block by calling the computeSAD function that computes the sum of absolute differences. When the motion parameters are found for a level, the motion parameters are updated for the next level by calling the function computeNewSearchBoundariesForUpperLevel(int[] searchBoundary, float[] mv). When motion parameters are determined they need to be aligned on the puzzle based on the previous image. The first image is centered on the center. The function alignImages calls alignImage function to align all images the sequence. The first image is aligned by the alignFirstImage sequence. The following is a basic code for solving image puzzles.

```
public void solvePuzzle()
 {
     initializePuzzle();
     imagePuzzle = new ImagePuzzle();
     createYImages(framesY);
     downSample(framesY, lowPass, actualWidth, actualHeight);
     generateMeanVarianceFeatures(framesY,lowPass, mean, variance, actualWidth, actualHeight);
      imagePuzzle.estimateMotionFrameSequence(framesY, lowPass, actualWidth, actualHeight,
              noOfFrames, motionVectors);
      imagePuzzle.alignFirstImage(frames[0], puzzle, puzzleWidth, puzzleHeight);
      imagePuzzle.alignImages(frames, motionVectors, puzzle, actualWidth, actualHeight,
             puzzleWidth, puzzleHeight, noOfFrames);
      writePPMImage("mypuzzle.ppm", puzzle, puzzleWidth, puzzleHeight);
 }
```

As mentioned earlier in this section, a hierarchical search method is used for motion-vectors. The hierarchical seach implies that motion vector for an upper level can be computed from a lower-level. Normally, if the motion vector at a level k is (u^k,v^k), the motion vector for level k+1 is expected to satisfy the following:

$$(2u^{k+1} - 1 \le u^k \le 2u^{k+1} + 1, 2v^{k+1} - 1 \le v^k \le 2v^{k+1} + 1) \quad (1).$$

This indicates that motion vector should be within ±1 of the motion vector at the previous level. However, this usually failed in our experiments since the images are low-resolution images. To resolve this problem, the search space for the next levels is increased as ±3:

$$(2u^{k+1} - \mathbf{3} \le u^k \le 2u^{k+1} + \mathbf{3}, 2v^{k+1} - \mathbf{3} \le v^k \le 2v^{k+1} + \mathbf{3}) \quad (2).$$

The figure 4 displays sample sequences that are captured by CmuCam2+. The image puzzle is displayed at the bottom of the figure.

6 Conclusion

In this paper, we have provided solution on solving image puzzles using CMUCam2+ vision sensor. CmuCam2+ vision sensor provides lower resolution and lower frame-rate than traditional cameras. By extending the search space in hierarchical motion estimation, we were able to determine the correct motion parameters for the alignment. On the other hand, CmuCam2+ vision sensor is very powerful for color tracking. We have built a robot using Serializer .NET robot controller to track colored objects. We would like to extend our research by using multiple cameras for collecting important information from the environment.

Acknowledgements

The author would like to thank to the Office of the Vice President for Research for supporting this project. The author would also like to thank to the following students who have contributed to this project: Mallikarjun Avula (funded), Manisha Deodhar (funded), Divya Panati (funded), Sharanya Krishnan (Independent Study), Venkatesh Supreeth (Independent Study and Course Project), Shubra Koshta (volunteer), and Arun Venkatachaliah (volunteer).

References

Chen, Y., Rui, Y., and Huang, T (2001). Jpdaf based hmm for real-time contour tracking. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 543–550.

Dufaux, F. and Konrad, J. (2000) "Efficient, robust, and fast global motion estimation for video coding," *IEEE Transactions on Image Processing*, vol. 9, no. 3, pp. 497–501.

Farin D. and de With, P. H. N. (2006). "Enabling arbitrary rotational camera motion using multisprites with minimum coding cost.," *IEEE Trans. Circuits Syst. Video Techn.*, vol. 16, no. 4, pp. 492–506.

Li, Z-N. and Drew, M.S. (2003). "*Fundamentals of Multimedia*", Prentice Hall; 2003.

Merl, (2004) Mitsubishi Electric Research Labaratories, http://www.merl.com/projects/ObjectTracking/

Peleg, S., Rousso, B., Rav-Acha, A., and Zomet, A. (2000). "Mosaicing on adaptive methods," *IEEE Trans. onPattern Analysis and Machine Intelligence*, vol. 22, no. 10, pp. 1144–1154.

Rowe, A. (2003) ."*CMUcam2 Vision Sensor*", Carnegie Mellon University http://www.cs.cmu.edu/~cmucam

Sato, K. and Aggarwal, J. (2004). Temporal spatio-velocity transform and its application to tracking and interaction. *Comput. Vision Image Understand. 96*, 2, 100–128.

Shafique, K. and Shah, M. (2003). A non-iterative greedy algorithm for multi-frame point correspondence. In *IEEE International Conference on Computer Vision (ICCV)*. 110–115.

Sikora, T. (1997) "The mpeg-4 video standard verification model," *IEEE Trans. Circuits Syst. Video Technology*, vol. 7, pp. 19–31.

Szeliski, R. and Shum, H-Y. (1997) "Creating full view panoramic image mosaics and environment maps," in *Computer Graphics Proceedings, Annual Conference Series*, pp. 251–258.

Veenman, C., Reinders, M., and Backer, E. (2001). Resolving motion correspondence for densely moving points. *IEEE Trans. Patt. Analy. Mach. Intell. 23*, 1, 54–72.

Yilmaz, A., Javed, O., and Shah, M. (2006). Object tracking: A survey. *ACM Comput. Surv.* 38, 4.

Frame 1 Frame 2 Frame 3 Frame 4 Frame 5

Frame 6 Frame 7 Frame 8 Frame 9 Frame 10

The Image Puzzle

Figure 4. Solving image puzzles.