Fast and Inexpensive Color Image Segmentation for Interactive Robots

James Bruce **Tucker Balch** **Manuela Veloso**

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{jbruce,trb,mmv}@cs.cmu.edu

Abstract

Vision systems employing region segmentation by color are crucial in real-time mobile robot applications, such as RoboCup[1], or other domains where interaction with humans or a dynamic world is required. Traditionally, systems employing real-time color-based segmentation are either implemented in hardware, or as very specific software systems that take advantage of domain knowledge to attain the necessary efficiency. However, we have found that with careful attention to algorithm efficiency, fast color image segmentation can be accomplished using commodity image capture and CPU hardware. Our paper describes a system capable of tracking several hundred regions of up to 32 colors at 30 Hertz on general purpose commodity hardware. The software system is composed of four main parts; a novel implementation of a threshold classifier, a merging system to form regions through connected components, a separation and sorting system that gathers various region features, and a top down merging heuristic to approximate perceptual grouping. A key to the efficiency of our approach is a new method for accomplishing color space thresholding that enables a pixel to be classified into one or more of up to 32 colors using only two logical AND operations. A naive approach could require up to 192 comparisons for the same classification. The algorithms and representations are described, as well as descriptions of three applications in which it has been used.

1 Introduction

An important first step in many color vision tasks is to classify each pixel in an image into one of a discrete number of color classes. The leading approaches to accomplishing this task include linear color thresholding, nearest neighbor classification, color space thresholding and probabilistic methods.

Linear color thresholding works by partitioning the color space with linear boundaries (e.g. planes in 3-dimensional spaces). A particular pixel is then classified according to which partition it lies in. This method is convenient for learning systems such as neural networks (NNs), or multivariate decision trees (MDTs) [2].

A second approach is to use nearest neighbor classification. Typically several hundred pre-classified exemplars are employed, each having a unique location in the color space and an associated classification. To classify a new pixel, a list of the K nearest exemplars are found, then the pixel is classified according to the largest proportion of classifications of the neighbors [3]. Both linear thresholding and nearest neighbor classification provide good results in terms of classification accuracy, but do not provide real-time[1] performance using off-the-shelf hardware.

Another approach is to use a set of constant thresholds defining a color class as a rectangular block in the color space [4]. This approach offers good performance, but is unable to take advantage of potential dependencies between the color space dimensions. A variant of the constant thresholding has been implemented in hardware by Newton Laboratories [5]. Their product provides color tracking data at real-time rates, but is potentially more expensive than software-only approaches on general purpose hardware.

A final related approach is to store a discretized version of the entire joint probability distribution. In this method, to check whether a particular pixel is a member of the color class, its individual color components are used as indices to a multi-dimensional histogram. When the location is looked up in the the returned real number indicates probability of membership. This technique enables a modeling of arbitrary distribution volumes and membership can be checked with reasonable efficiency. The approach also enables the user to represent unusual membership volumes (e.g. cones or ellipsoids) and thus capture de-

[1]We define "real-time" as full frame processing at 30 Hz or faster with bounded running time.

pendencies between the dimensions of the color space. The primary drawback to this approach is high memory cost — for speed the entire probability matrix must be present in memory.

The approach taken in our work is a combination of the methods described above, but with a special focus on efficiency issues. Thus we are able to provide effective classification at real-time rates. The method is best described as constant thresholding, but with a projected color space when needed. Above this is a layer that converts the frame into a more geometric representation suitable for high level processing. In the next section the outline of our approach is presented. The remaining sections describe the performance of a system using the method, and provide examples of its use in several applications.

2 Description of the Approach

2.1 Color Space Transformation

Our approach involves the use of thresholds in a three dimensional color space. Several color spaces are in wide use, including Hue Saturation Intensity (HSI), YUV and Red Green Blue (RGB). The choice of color space for classification depends on several factors including which is provided by the digitizing hardware and utility for the particular application.

RGB is a familiar color space often used in image processing, but it suffers from an important drawback for many robotic vision applications. Consider robotic soccer for instance, where features of the environment are marked with identifying colors (e.g. the ball might be painted orange). We would like our classification software to be robust in the face of variations in the brightness of illumination, so it would be useful to define “orange” in terms of a ratio of the intensities of Red Green and Blue in the pixel. This can be done in an RGB color space, but the volume implied by such a relation is conical and cannot be represented with simple thresholds.

In contrast, HSI and YUV have the advantage that chrominance is is coded in two of the dimensions (H and S for HSI or U and V for YUV) while intensity is coded in the third. Thus a particular color can be described as “column” spanning all intensities. These color spaces are therefore often more useful than RGB for robotic applications.

Some digitizing hardware provides one or more appropriate color spaces directly (such as HSI or YUV). In other cases, the space may require transformation from the one provided by hardware to something more appropriate. Once a suitable projection is selected, the resulting space can be partitioned using constant valued thresholds, since most of the significant correlations have been removed.

The commodity digitizer we initially used provides images coded in RGB. We found that rotating the RGB color space provides significantly more robust tracking. Much of the information in an RGB image varies along the intensity axis, which is roughly the bisecting ray of the three color axes. By calculating the intensity and subtracting this component from each of the color values, a space in which the variance lies parallel to the axes is created, allowing a more accurate representation of the region space by a rectangular box.

Another, more robust (but more expensive) transformation is a nonlinear fractional RGB space, where each of the component colors is specified as a fraction of the intensity, and the intensity is added as another dimension. This projection into a 4 dimensional space proved accurate, but with the extra dimension to process and three divides per pixel to calculate the fractions, it proved to be too slow for currently available hardware.

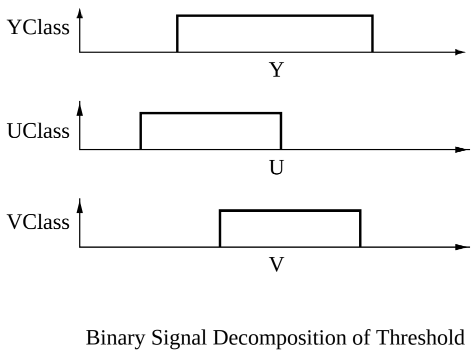
We later moved to a system which provided YUV colors in hardware. This combines the power of a robust color space without the performance penalty of a software color space transformation. Thus systems can take advantage of hardware with good native color spaces, but even without them, a suitable transformation can lead to a reasonable solution.

2.2 Thresholding

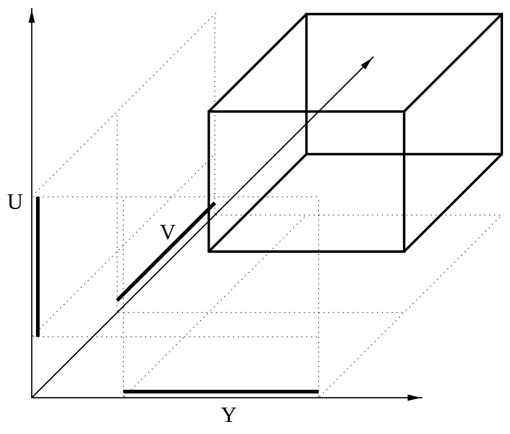
The thresholding method described here can be used with general multidimensional color spaces that have discrete component color levels, but for the purposes of discussion the YUV color space will be used as an example. In our approach, each color class is specified as a set of six threshold values: two for each dimension in the color space, after the transformation if one is being used. The mechanism used for thresholding is an important efficiency consideration because the thresholding operation must be repeated for each color at each pixel in the image. One way to check if a pixel is a member of a particular color class is to use a set of comparisons similar to

```
if ((Y >= Ylowerthresh)
   AND (Y <= Yupperthresh)
   AND (U >= Ulowerthresh)
   AND (U <= Uupperthresh)
   AND (V >= Vlowerthresh)
   AND (V <= Vupperthresh))
   pixel_color = color_class;
```

to determine if a pixel with values `Y, U, V` should be grouped in the color class. Unfortunately this approach is rather inefficient because, once compiled, it could require as many as 6 conditional branches to determine membership in one color class for each pixel. This can be especially inefficient on pipelined processors with speculative instruction execution.



Binary Signal Decomposition of Threshold



Visualization as Threshold in Full Color Space

Figure 1: A three-dimensional region of the color space for classification is represented as a combination of three binary functions.

Instead, our implementation uses a boolean valued decomposition of the multidimensional threshold. Such a region can be represented as the product of three functions, one along each of the axes in the space (Figure 1). The decomposed representation is stored in arrays, with one array element for each value of a color component. Thus class membership can be computed as the bitwise `AND` of the elements of each array indicated by the color component values:

```
pixel_in_class = YClass[Y]
             AND UClass[U]
             AND VClass[V];
```

The resulting boolean value of `pixel_in_class` indicates whether the pixel belongs to the class or not. This approach allows the system to scale linearly with the number of pixels and color space dimensions, and can be implemented as a few array lookups per pixel. The operation is much faster than the naive approach because the the bitwise `AND` is a significantly lower cost operation than an integer compare on most modern processors.

To illustrate the approach, consider the following example. Suppose we discretize the YUV color space to 10 levels in each each dimension. So "orange," for example might be represented by assigning the following values to the elements of each array:

```
YClass[] = {0,1,1,1,1,1,1,1,1,1};
UClass[] = {0,0,0,0,0,0,0,1,1,1};
VClass[] = {0,0,0,0,0,0,0,1,1,1};
```

Thus, to check if a pixel with color values `(1,8,9)` is a member of the color class "orange" all we need to do is evaluate the expression `YClass[1] AND UClass[8] AND VClass[9]`, which in this case would resolve to 1, or `true` indicating that color is in the class "orange."

One of the most significant advantages of our approach is that it can determine a pixel's membership in multiple color classes *simultaneously*. By exploiting parallelism in the bit-wise `AND` operation for integers we can determine membership in several classes at once. As an example, suppose the region of the color space occupied by "blue" pixels were represented as follows:

```
YClass[] = {0,1,1,1,1,1,1,1,1,1};
UClass[] = {1,1,1,0,0,0,0,0,0,0};
VClass[] = {0,0,0,1,1,1,0,0,0,0};
```

Rather than build a separate set of arrays for each color, we can combine the arrays using each bit position an array element to represent the corresponding values for each color. So, for example if each element in an array were a two-bit integer, we could combine the "orange" and "blue" representations as follows:

```
YClass[] = {00,11,11,11,11,11,11,11,11,11};
UClass[] = {01,01,01,00,00,00,00,10,10,10};
VClass[] = {00,00,00,01,01,01,00,10,10,10};
```

Where the first (high-order) bit in each element is used to represent "orange" and the second bit is used to represent "blue." Thus we can check whether `(1,8,9)` is in one of the two classes by evaluating the single expression `YClass[1] AND UClass[8] AND VClass[9]`. The result is `10`, indicating the color is in the "orange" class but not "blue."

In our implementation, each array element is a 32-bit integer. It is therefore possible to evaluate membership in 32 distinct color classes at once with two `AND` operations. In contrast, the naive comparison approach could require

32×6, or up to 192 comparisons for the same operation. Additionally, due to the small size of the color class representation, the algorithm can take advantage of memory caching effects.

2.3 Connected Regions

After the various color samples have been classified, connected regions are formed by examining the classified samples. This is typically an expensive operation that can severely impact real-time performance. Our connected components merging procedure is implemented in two stages for efficiency reasons.

The first stage is to compute a run length encoded (RLE) version for the classified image. In many robotic vision applications significant changes in adjacent image pixels are relatively infrequent. By grouping similar adjacent pixels as a single "run" we have an opportunity for efficiency because subsequent users of the data can operate on entire runs rather than individual pixels. There is also the practical benefit that region merging need now only look for vertical connectivity, because the horizontal components are merged in the transformation to the RLE image.

The merging method employs a tree-based *union find* with path compression. This offers performance that is not only good in practice but also provides a hard algorithmic bound that is for all practical purposes linear [6]. The merging is performed in place on the classified RLE image. This is because each run contains a field with all the necessary information; an identifier indicating a run's parent element (the upper leftmost member of the region). Initially, each run labels itself as its parent, resulting in a completely disjoint forest. The merging procedure scans adjacent rows and merges runs which are of the same color class and overlap under four-connectedness. This results in a disjoint forest where the each run's parent pointer points upward toward the region's global parent. Thus a second pass is needed to compress all of the paths so that each run is labeled with its the actual parent. Now each set of runs pointing to a single parent uniquely identifies a connected region. The process is illustrated in Figure 2).

2.4 Extracting Region Information

In the next step we extract region information from the merged RLE map. The bounding box, centroid, and the size of the region are calculated incrementally in a single pass over the forest data structure. Because the algorithm is passing over the image a run at a time, and not processing a region at a time, the region labels are renumbered so that each region label is the index of a region structure in the region table. This facilitates a significantly faster lookup while computing the incremental statistics.

Other statistics can optionally be computed with additional passes. Currently the only additional option is to compute the average color of a region, which can be useful as a method of double thresholding to verify if a region has the same color as the desired object. Other features that may be included in the future are color variance within a region, and convex hulls and edge points which could be useful for geometric model fitting.

After the statistics have been calculated, the regions are separated based on color into separate threaded linked lists in the region table. Finally, they are sorted by size so that high level processing algorithms can deal with the larger (and presumably more important) regions and ignore relatively smaller ones which are most often the result of noise.

2.5 Density-Based Region Merging

In the final layer before data is passed back up to the client application, a top-down merging heuristic is applied that helps eliminate some of the errors generated in the bottom up region generation. The problem addressed here is best introduced with an example. If a detected region were to have a single line of misidentified pixels transecting it, the lower levels of the vision system would identify it as two separate regions rather than a single one. Thus a minimal change in the initial input can yield vastly differing results.

One solution in this case is to employ a sort of grouping heuristic, where similar objects near each other are considered a single object rather than distinct ones. Since the region statistics include both the area and the bounding box, a density measure can be obtained. The merging heuristic is operationalized as merging pairs of regions, which if merged would have a density is above a threshold set individually for each color. Thus the amount of "grouping force" can be varied depending on what is appropriate for objects of a particular color. In the example above, the area separating the two regions is small, so the density would still be high when the regions are merged, thus it is likely that they would be above the threshold and would be grouped together as a individual region.

3 Results and Applications

The first implementation was a prototype for a group of inexpensive autonomous robots based on the Probotics Cye platform[7]. These robots are based on commodity hardware to keep the cost low and aid in simplicity. They still require high performance vision however because it will serve as their primary hazard sensor. The platform uses a conventional NTSC color camera linked to a AMD K6 based PC-104 computer and a Winnov digitizer. The op-

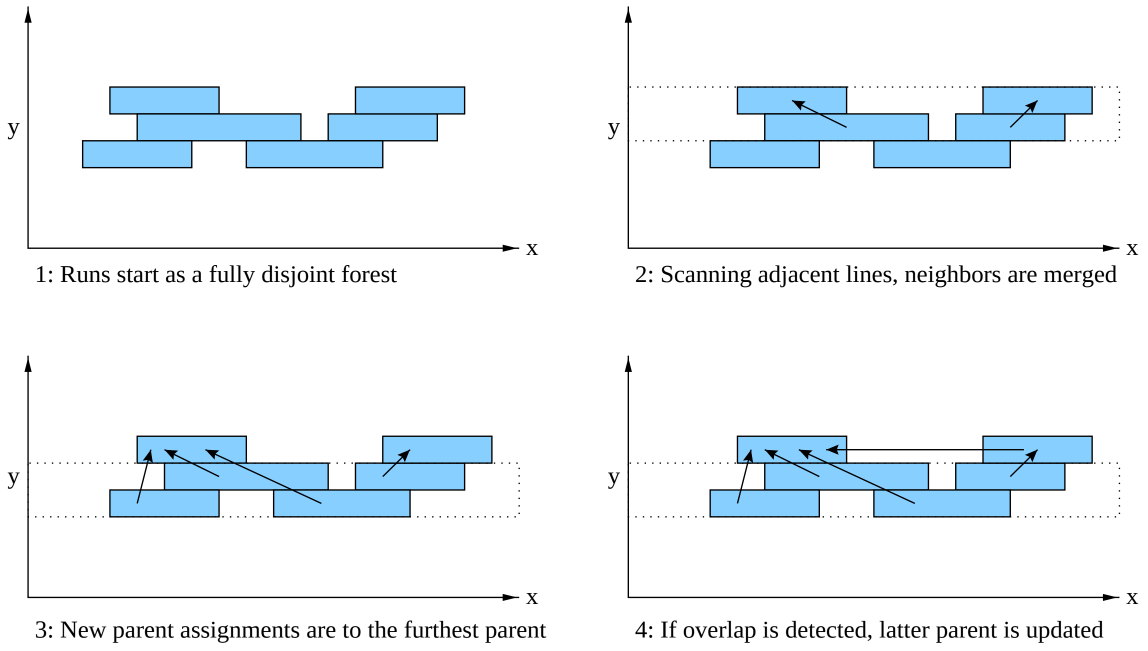

y
x
1: Runs start as a fully disjoint forest
y
x
2: Scanning adjacent lines, neighbors are merged
y
x
3: New parent assignments are to the furthest parent
y
x
4: If overlap is detected, latter parent is updated

Figure 2: An example of how regions are grouped after run length encoding.

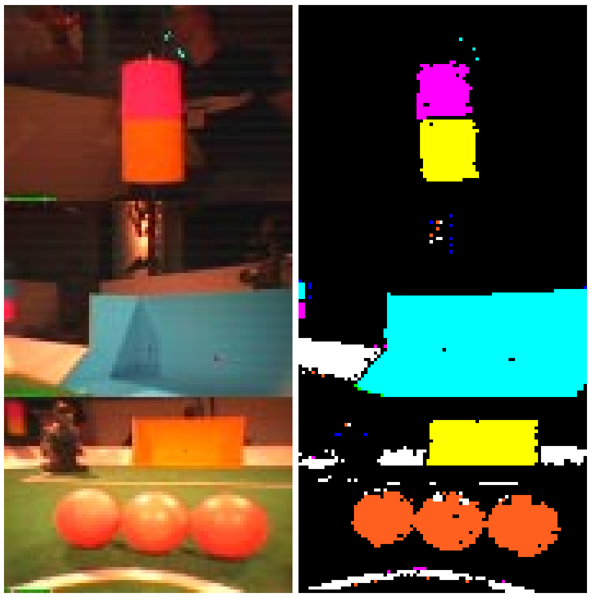


Figure 3: An example image classified using the approach presented in the paper. The image on the left is a composite of objects tracked by a soccer robot at RoboCup-99: a position marker (top), a goal area (middle) and three soccer balls (bottom). The classified image is on the right. (Color versions of these images are available online at `http://www.coral.cs.cmu/cmvision`.)

erating system is a standard Linux distribution and kernel with Video for Linux drivers for video capture. In its current form the system can process 320x240 images at 30 Hz with 30% utilization of the 350 MHz CPU.

The second successful application was for Carnegie Mellon's entry into the RoboCup-99 legged-robot league[8]. These robots, provided by Sony, are quadrupeds similar to the commercially available Aibo entertainment robot. The robots play a game of three versus three soccer against other teams in a tournament. To play effectively, several objects must be recognized and processed, including the ball, teammates and opponents, two goals, and 6 location markers placed around the field. The hardware includes a camera producing 88x60 frames in the YUV color space at about 15Hz. In this application color classification is done in hardware, removing the need for this step in the software system. Even with one step of the algorithm handled in software however, limited computational resources require an optimized algorithm in order to leave time for higher-level processes like motion control, team behaviors, and localization. The system included the density based region merging heuristic to overcome excessively noisy images that simple connectivity could not handle. The system proved to be robust at the RoboCup-99 competition, enabling our team to finish 3rd in the international competition.

The third application of the system is as part of an entry for the RoboCup small-size league (F180). This domain involves a static camera tracking remotely controlled robots playing soccer on a small field. Our system uses a single camera above the center of the field. The system can now track the ball, and ten robots. The five teammate robots contain a standard marker, as well as special orientation and unique identification color patches. The opponent robots use a standard marker and additional patterns of their choice for their tracking systems. Full frame processing is beneficial in this domain due to temporary occlusion of the ball or robots, and in accurate tracking of the ball and robots moving at speeds up to 4m/s. At a resolution of 640x480 and 30Hz capture frequency, the system uses 60% of a 700MHz Pentium III.

4 Conclusion

We have presented a new system for real-time segmentation of color images. It can classify each pixel in a full resolution captured color image, find and merge regions of up to 32 colors, and report their centroid, bounding box and area at 30 Hz. The primary contribution of this system is that it is a software-only approach implemented on general purpose, inexpensive, hardware (in our case 350MHz or 700MHz x86 compatible systems with $200 image digitizers). Among full frame processing systems, this provides a significant advantage over more expensive hardware-only solutions, or other slower software approaches.

The system operates on the image in several steps:

1. Optionally project the color space.
2. Classify each pixel as one of up to 32 colors.
3. Run length encode each scanline according to color.
4. Group runs of the same color into regions.
5. Pass over the structure gathering region statistics.
6. Sort regions by color and size.

The speed of our approach is due to a focus on efficient algorithms at each step. Step 1 is accomplished with a linear transformation. In Step 2 we discard a naive approach that would require up to 192 comparisons per pixel in favor of a faster calculation using two bit-wise `AND` operations. Step 3 is linear in the number of pixels. Step 4 is accomplished using an efficient *union find* algorithm. The sorting in Step 5 is accomplished with radix sort, while Step 6 is completed in a single pass over the resulting data structure.

The approach is intended primarily to accelerate low level vision for use in real-time applications where hardware acceleration is either too expensive or unavailable. Functionality is appropriate to provide input to higher level routines which encode geometric and/or domain-specific processing. This tool enables formerly offline processes to run as a part of a real-time intelligent vision system. The current system and its variants have been demonstrated successfully on three hardware platforms.

References

[1] H. Kitano, Y. Kuniyoshi, I. Noda, M. Asada, H. Matsubara, and E. Osawa. RoboCup: A challenge problem for AI. *AI Magazine*, 18(1), pages 73–85, 1997.

[2] C. E. Brodley and P. E. Utgoff. Multivariate decision trees. *Machine Learning*, 1995.

[3] T. A. Brown and J. Koplowitz. The weighted nearest neighbor rule for class dependent sample sizes. *IEEE Transactions on Information Theory*, pages 617–619, 1979.

[4] R. Jain, R. Kasturi, and B. G. Schunck. *Machine Vision*. McGraw-Hill, 1995.

[5] Newton Laboratories. Cognachrome image capture device. `http://www.newtonlabs.com`, 1999.

[6] R. E. Tarjan. Data structures and network algorithms. *Data Structures and Network Algorithms*, 1983.

[7] The Probotics Cye Personal Robot. `http://www.probotics.com`, 2000.

[8] M. Veloso, E. Winner, S. Lenser, J. Bruce, and T. Balch. Vision-Servoed Localization and Behaviors for an Autonomous Quadruped Legged Robot *Artificial Intelligence Planning Systems*, 2000.