Design and Experimental Validation of a Minimal Adaptive Real-time Visual Motion Tracking System for Autonomous Robots

Virgil Andronache and Matthias Scheutz
Artificial Intelligence and Robotics Laboratory
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN 46556, USA
{vandrona,mscheutz}@cse.nd.edu

Abstract

We present a real-time system for motion detection and tracking for autonomous robots with limited computational resources. The system uses an attentional mechanism to combine color and motion blob information, as well as prediction mechanisms to improve the overall system performance. We demonstrate the functionality of the system in several test scenarios and also discuss solutions to various problems that arise from the limitations of the robotic platform.

Keywords: real-time adaptive tracking, autonomous robot

1 Introduction

Fast and reliable processing of visual information is critical for many applications of autonomous robots. In particular, detection and tracking of moving objects while the robot is moving is of crucial interest to mobile platforms ranging from robotic soccer players to unmanned ground vehicles for military scenarios. While the detection and tracking of moving objects is already a difficult task, the problem is only exacerbated when computational resources are very restricted. Especially in small autonomous robots with slow embedded computers, where the processing speed of the CPU is often traded for the longevity of the battery, standard algorithms for motion detection and tracking are not applicable.

In this paper, we present a system for motion detection and tracking that is especially target at the severe resource limitations of small autonomous robots. It uses an attentional mechanism to select different methods and processing constraints from several modules for the detection and prediction of motion in a scene. We first discuss the overall system architecture and the motivation for the design. Then we present the implementation and report very promising results from several sets of experiments with the system.

2 Background

A significant amount of research has been performed for motion detection and surveillance. Still image analysis results include a variety of approaches ranging from color analysis [6] to the use of genetic algorithms [1]. The active blobs approach relies on building 2-D mesh representations of surfaces. Deformation of a mesh via transforms together with texture mapping are then used to identify deformable objects [5]. Eigenspace approaches [2] use training images and matrix decomposition to create a database of feature images which are used to identify detail of new pictures. An efficient method for a rotating camera was presented by Dellaert and Collins [4]. The algorithm relies on differences between stored images of the surrounding area, taken at various angles.

Several of the algorithms mentioned above (e.g., [5]) are intended for off-line image analysis. As such, they are not designed to function under the type of computational power constraints imposed by an autonomous robotic agent. The algorithms which are designed with a view towards real-time applications (e.g., [4, 2]) require training and may still prove too complex to run on simple, autonomous robots. Others that take resource-constraints seriously (e.g., [3]), fail if colors cannot be reliably detected.

What is needed is an online, *real-time* system that can integrate different information to improve detection and tracking performance despite limited resources. In particular, we require that no assump-

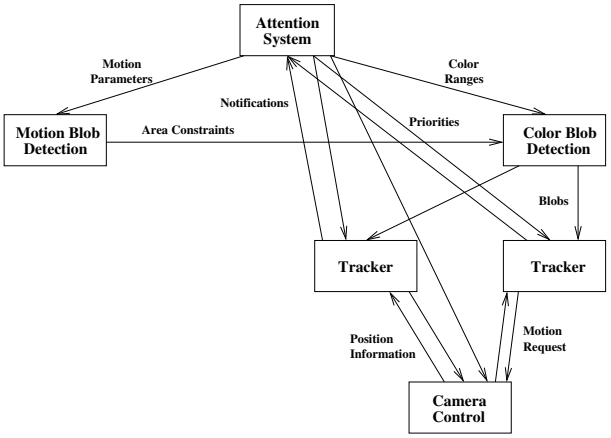

Figure 1: The system architecture

tions be made about the number or quality of the color of an object nor about its uniqueness in a scene. Nor should any restriction be imposed on the lighting

conditions (e.g., to provide glare-free lighting). Finally, no assumptions should be made about temporary occlusions of the objects to be tracked by other, possibly moving objects.

All these criteria together make it very hard, if not impossible to show close-to-perfect performance on high-performance systems, let alone on low performance embedded systems. However, it is possible to achieve a reasonable performance that might be sufficient for many real-world applications. In the following, we first describe the architecture of our proposed system and then show the results of the experimental validation of the design.

3 The System Architecture

The system consists of five major, concurrently active modules: the *color blob detection*, the *motion blob detection*, the *blob tracking*, the *blob selection and camera control*, and the *attentional subsystem* (see Figure 1). In the following, we will describe the functionality of each modules as well as the employed methods for reducing computational resource requirements.

3.1 Color Blob and Motion Blob Detection

Color and motion blob detection are the most expensive subsystems in terms of computational resources (both in time and space): blob detection in the whole image has a time requirement of O($width \cdot height$) (where $width$ and $height$ are the dimensions of the image). Each color is defined in terms of three centroids for each dimensions in RGB space and two threshold values for each centroid, respectively. Any number of colors can be searched for in one pass of the algorithm. Different from other highly optimized blob detection systems (e.g., [3]), any number of overlapping color regions can be defined in RGB space and searched for, resulting in overlapping blobs (this is too allow the system to detect colors rather than having to supply them before hand). The time cost for a search for multiple colors is linear in the number of different colors. The algorithm checks each pixel whether it belongs to one of the target colors and if so, either adds it to the blob of the pixel to the left if the colors match, or to the blob of the upper pixel if colors match. If both the left and the upper pixel match but belong to different blobs, then the two blobs are merged right away. This step reduces the number of blobs that need to be merged and thus improves the performance of the algorithm.[1] After all pixel have been processed, centroids and bounding boxes are computed, and blob filters are applied (e.g., blobs below a minimal size are dropped). The algorithm returns lists of color blobs ordered according to the respective colors parameters.

Motion blob detection is based on the same algorithm, except that pixels in two consecutive images are compared according to their difference along all three dimensions (which are defined in terms of "error intervals"). If a pixel in the current images is not contained in the error intervals of the same pixel in the previous image, it is added to a motion blob. After processing all pixels, motion blobs are merged in the same way as color blobs.

The JAVA implementation of the algorithm has been further optimized in several ways to reduce runtime resources. For example, all data structures are pre-allocated (as "new" operations for class instantiations in JAVA are expensive). Furthermore, it is possible to invoke the algorithm with optional parameters that define a rectangular area in the image, to which the search of blobs should be constrained. Applying constraints is an important way to increase the overall frame rate of the algorithm (see the experiments below).

[1] Still, the flexibility of being able to search any number of possibly overlapping color regions at the same time causes a performance loss of a factor of 2 to 3 depending on the application compared to the highly optimzed, yet less flexible algorithm in [3].

3.2 The Attentional Subsystem

The purpose of the attentional system is to find particular objects in a scene and subsequently track them as well as possible. Objects can be defined in terms of multiple colors, size, and/or motion. Whenever a new request for finding and tracking an object is received by the attentional system, it reconfigures the blob and motion detection systems according to the new requirements. For example, a request to the attentional system like "find a small moving car and track it" will result in the following configuration of the system: the motion tracker will search the whole image for regions with movement and return those blobs. The color blob detection will subsequently be called on those blobs for a range of blue colors. For each blue object that has been identified, a tracker will be instantiated that will independently attempt to track the object. A criterion can be set that will be used to decide which blob to track if inconsistencies arise (e.g., if two blobs move in different directions). The attentional system receives information from each tracker and, by applying the set criterion, assigns one of them a higher priority than the rest. If the camera moves, the attentional system compares the overall number of pixels that are marked as motion pixels and discards an image if this number is above a given threshold. This is a biologically inspired method (performed by the human retina) to determine whether the camera/robot is moving or the objects are moving without requiring proprioceptive feedback from the camera/robot motors. Without dismissing such images taken during camera motion, the tracking mechanism will not work properly, as trackers implicitly assume that the camera has reached its final position when they get blob information from the blob detecting systems. By suppressing the image from the camera, the system's integrity can be preserved.

3.3 The Motion Tracker

Motion trackers are instantiated for each matching region that is found in the image (i.e., color blobs, motion blobs, or combinations). Once instantiated, they will attempt to track the blob defined by the instantiation criteria (e.g., blue moving blob). If the blob enters a critical region on the screen, then the tracker sends a request to the camera control system to move the camera. Similarly, whenever a camera movement occurs, trackers are notified in order to be able to update their state. If a tracker has identified its blob in an image, it will attempt to predict the blob's location in the next image based on the locations of the blob in the past two images. The result of the prediction defines a constraint, which is passed to the color and motion blob detection systems in order to reduce processing time. If a tracker loses its blob, it widens its constraints and attempts to reidentify it in the current image. It also notifies the attentional mechanism, which ensures that wider regions in the images are considered for blob detection. If reidentification of the blob is not possible, the tracker will wait for a predetermined number of images before it terminates (at which time it will also notify the attentional mechanisms).

3.4 The Blob Selection and Camera Control

The blob selection and camera control module is concerned with moving the camera fast enough to keep the blob be designated by the attentional system centered. The camera is controlled by two PID controllers, whose gains have been experimentally determined. Of particular importance for moving objects are the non-zero ID components, which can implement an inertia effect in the camera movement that facilitates the tracking of moving objects (see experiment 2 below).

4 Experimental Set-up

After presenting the details of the functional organization of the system architecture, we now describe the experimental setup to test the system and verify its design. All experiments were conducted with an ActivMedia Pioneer 2 robot with a Sony Pan-Tilt-Zoom camera. The robot has an onboard PC104 computer running RedHat Linux 8.0 with a 850 MHz Pentium III CPU and 128 MB of RAM, and a low-level motor control board with its own embedded processor. Communication with the motor control board (sending commands to the effectors and retrieving sensor information) was achieved using an asynchronously-running server written in JAVA that wraps the low-level controller commands. The tracking system was connected to the JAVA server via JAVA RMI (i.e., remote method invocation), which allows for two different setups: the tracking system can either run on the robot or on a remote host, in which case it connects to the robot through a wireless ethernet. The performance of the tracking system was tested in both setups.

The architecture of the tracking system was developed and tested using the ADE Agent Development Environment (under development in our lab).

Four ADE component types were used to implement the five tracking system components, where motion and color blob detection where merged into one ADE component for efficiency reasons.

4.1 Experiments

We ran three sets of experiments: (1) with a stationary camera on a stationary robot, (2) with a moving camera on stationary robot, and (3) with a moving camera on a moving robot. For each set we considered two environments: one, in which the tracked object was first never occluded, and one, where the object to be tracked was visible in the beginning, and subsequently occluded for short periods of time. We used several objects to test the system, but will report only results from tracking of a single, small blue toy car. This object was not only more difficult to track because of its size than the other objects we used (e.g., the kind of orange soccer ball used in robo-soccer), but also because its color was close to the color of the carpet in our lab, so that it was often impossible to get a unique color blob for the car.

4.1.1 Experiment Set 1: Stationary Camera

In the first experiment, the car had a distinctive blue color which facilitated its detection through color blob image analysis. Blob information was sent to a *Tracker* unit. The *Tracker* then iterated through all blobs, using centroid, area, and boundary information to determine which blob is most likely to represent the car. Using color information as the primary means for tracking led to very reliable tracking of the car in an environment with several moving objects of different colors.

In our second experiment we added an obstacle which occluded the car for half a second to a second at a time. The obstacle was not wide enough to fill the entire image captured by the camera. The camera moved slightly past the beginning of the occlusion and stopped with both ends of the obstacle in view. Thus, when the car emerged on either side - whether continuing its trajectory or reversing its motion, the camera continued its tracking.

4.1.2 Experiment Set 2: Moving Camera

Here, we performed four sets of experiments: the first two were identical to the stationary camera set. However, motion detection was introduced here for the first time into the system. Another new addition to the system was motion estimation: if the car position was known for two iterations through the control cycle, then an extrapolation was made of the probable camera location. The search was then initially limited to this area and, if that failed, generalized to the entire image.

The third and fourth tests, described below, tested the interplay between blob detection and motion detection in tracking a moving object. A PID controller was used to move the camera to the best position for ball tracking. For best results in our system, we have found that a P-gain of 0.26, a I-gain of 0.2 and a D-gain around 0.2 allowed the camera to track the car both in the unoccluded and the occluded experiments.

For the first two experiments we moved the car by holding it in one hand. In our third experiment, we started the experiment in a similar manner. However, at some point we closed the hand around the car, continuing the motion. At that point, the tracker could no longer rely on color information for tracking, relying instead on motion information. In order to better recover the car if/when it reappeared on the screen we kept information about the size of the car. And compared color blob sizes against the known size of the car.

A fourth experiment further tested the limits of our system. As in the third experiment, at some point the car was hidden in the hand. Here, however, the other hand was at this time also moved to and fro in front of the camera. Tracking was satisfactory, though sometimes, depending on thread scheduling and how distinct the moving objects were in the picture, the tracker would latch on to another moving object and start following that (especially if two moving objects came close enough to be identified as a single motion by the algorithm).

4.1.3 Experiment Set 3: Moving Camera + Moving Robot

By adding motion to the robotic platform itself, this became by far the most complex task of the system, as the motion detection became far less useful in limiting the areas which were searched for the ball. However, the motion extrapolation proved to be beneficial in this case, as the motion of the car as it moved (in both a straight line and an a random motion) was followed by the agent.

The additional load of computing motion and controlling the robot wheels led the system to slower response times, and more frequent loss of tracking in the occlusion tests.

With the complete architecture in place, cycle times were taken to evaluate the performance of the architecture. Two measures were considered: the time needed for visual processing and the time

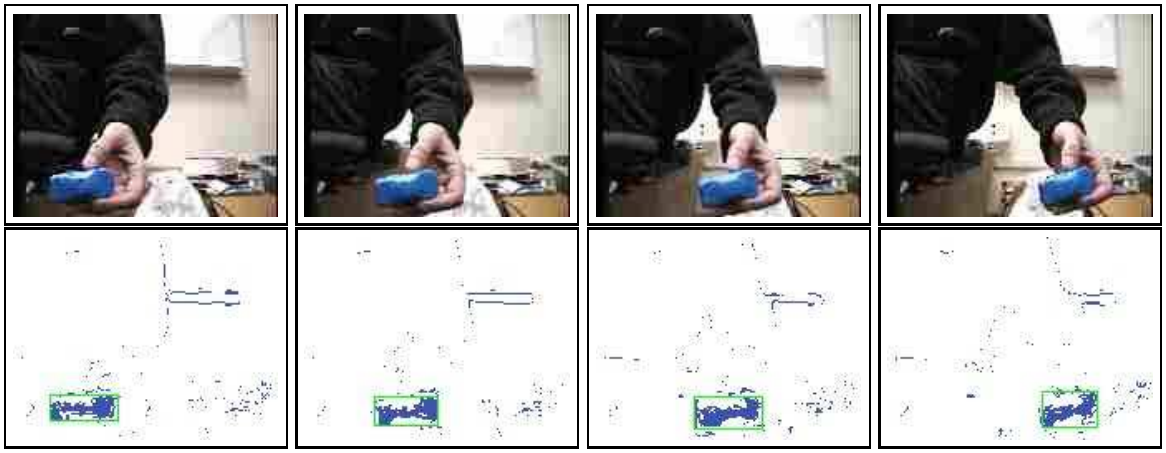

Figure 2: Camera images and tracker information for stationary camera tracking: unprocessed image(top), tracker information (bottom)

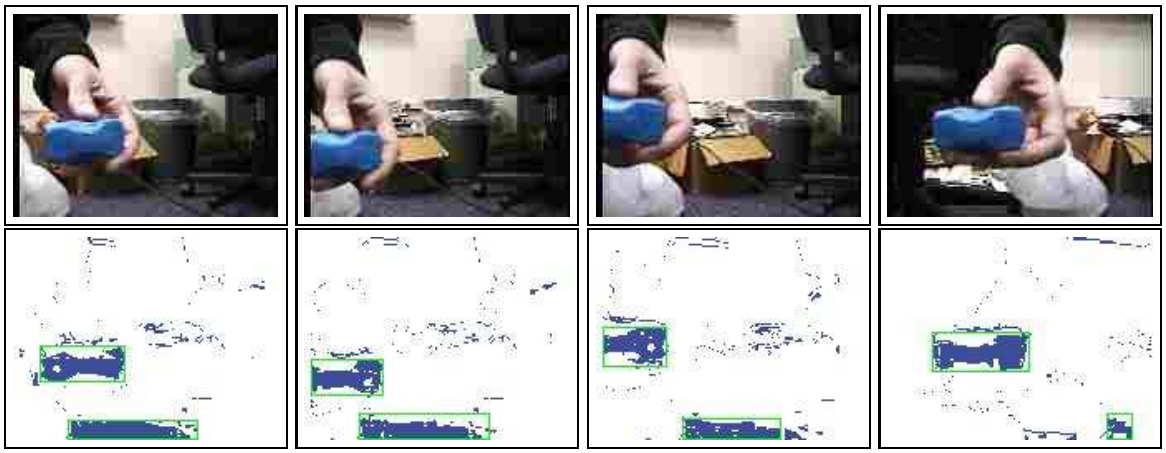

Figure 3: Camera images and tracker information for mobile camera tracking: unprocessed image(top), tracker information (bottom). Two trackers are present in some images; the main blob is moving to the left

needed for one complete update of the architecture. Visual processing averaged 121.65 ms per cycle with a standard deviation of 231.94. Overall cycle times averaged 146.94 ms, with a standard deviation of 103.07.

Three conclusions can be derived:

- Visual processing was responsible for most of the CPU time used by the system (82.79%)
- Individual component processing times tend to vary significantly due to the threaded nature of the system
- The overall system variation in execution times is much less than its component parts, since most thread switches (with such exceptions as the robot server updates) occur among components of the architecture

5 Implementation Details and Optimization Measures

In this section we present some of the ideas, optimizations, and heuristics used in our system in order to maximize performance.

5.1 Threading and Parallelism

In autonomous agents, time is the critical resource: it is essential that operations are performed effectively and in a timely manner. An implicit consequence of this is that threading and parallel execution should be used wherever possible.

5.1.1 Image capture

On the Pioneer 2 robots, the framegrabber uses DMA to transfer images to memory. On an 850

MHz processor, using a 160 pixel by 120 pixel image, each transfer takes roughly 0.1 seconds (hence the highest possible framerate is 10 fps). It is therefore necessary that other processes execute in parallel to the image capture. In JAVA, this translates to image capture being performed in a separate thread in the image server:

```
public void run() {
  byte[] frame = null;
  while (true) {
    frame = frameGrabber.getFrame(quality);
    synchronized(imageGuard) {
      image = frame;
    }
    Thread.yield();
  }
}
```

In the code above, the *image* is accessible to the outside world and access to it is restricted in order to prevent reads being performed during a write. Also notable is the explicit *Thread.yield*() call after each read. We have noticed that the JAVA thread scheduler does not always preempt threads in a timely fashion, leading to severe time lags in camera movement when the *yield* is omitted.

5.1.2 Other threads

Robot sensory updates were run as an asynchronous thread. Each architectural component was also run as a thread, thus allowing the robot updates to interweave with computation. All threads performed a *yield*() at the end of their update cycles, providing a simple mechanism for concurrent updating.

The time to get a new image from the robot using the framegrabber is approximately 100ms. Therefore, the image analysis node should retrieve images from the robot every tenth of a second. This was again an opportunity for threaded execution: in order to minimize both RMI calls and the time spent waiting for images to arrive, a thread in the image analysis node retrieves an image from the robot, sleeps for the remainder of time to 100 milliseconds from the start of the last access and starts a new retrieval.

5.1.3 Image analysis

As mentioned in the Method section, in searching for the car, motion detection was performed first, with color detection only in the regions, where motion was detected. Limitations on the area on which motion detection was performed were sometimes imposed based on estimations of the car motion. This estimate relied on knowing where the car was as two previous points in time. The difference between them was computed and taken as an estimate of how far the car may have moved since last seen. A rectangular search region is created by starting at the previously known location, adding the motion estimate plus an error correction factor (for these experiments 10%) and limiting the search to the region defined by the previous point, the tip of the motion estimate vector and horizontal and vertical lines drawn through the two points. This estimate amounts to allowing for the following types of motion for the car:

- Moving with a horizontal velocity component between 0 and 110% of the estimate
- Moving with a vertical velocity component between 0 and 110% of the estimate

For all but very fast and irregular motion patterns, this estimate proved to be very useful in maintaining a low turnaround time for image analysis.

In the following section we present the environment used for system development and some of the features which facilitated its testing.

6 System Development

The development of the architecture for our system was done in the ADE development environment. ADE is a JAVA-based environment with facilities for both single and multi-computer system development. The JAVA base of the system facilitates the use of graphical tools in the design, testing, and running of a system. Each of the components used in the tracking system was implemented as and ADE component.

The tests were run in two configurations. In the first test, all components of the architecture were run on the robot. In the second, a remote machine was used for visual processing and tracking.

Two steps were taken to ensure a more efficient search for the car: (1) the motion and blob detection codes were optimized, and (2) the colors for the car were optimized.

Due to varying lighting conditions, two color ranges were used to identify the car. The process of identifying and fine-tuning the colors identified as potential car matches was aided by the ease of adding visual interfaces to ADE components.

Figure 4 shows the panel which was added to the vision server and which was used to configure the parameters for blob detection. The panel displays three images: the unmodified camera picture (top), the results of performing blob detection on

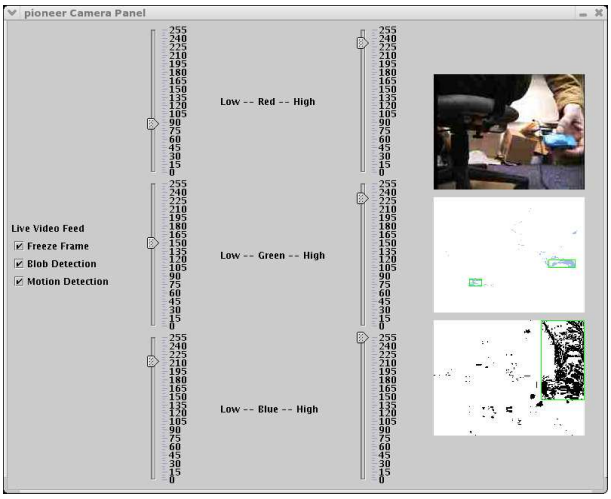

Figure 4: Control panel for dynamic color range adjustment

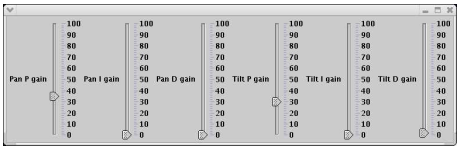

Figure 5: Control panel for dynamic adjustment of PID controller parameters

that image with the parameters set on the sliders (middle), and the results of performing motion detection on the original image.

Camera control was performed through the use of 2 PID controllers - one for horizontal movement and one for vertical movement. A similar calibration process was performed on the parameters of each controller, using another graphical tool (Figure 5), which allowed us to change parameters as the architecture was running.

The architecture for the above experiments was run entirely on the robot. However, due to its ADE-based implementation it was possible to also run the architecture off-board, on a 2.1 GHz PC, leaving only the robot servers and the control node to execute on the robot. A reduction of approximately 35% in the time required for one complete update of the architecture was observed.

7 Conclusion

In this paper we proposed an adaptive tracking system for autonomous robots that uses color and motion information to track moving objects, while the robot and its camera are possibly moving. The system achieves an average frame rate of 7 frames per second on an ActivMedia Pioneer 2 robot (where 10 frames per second is the maximum number of frames that can be obtained from the framegrabber in our setup). The performance evaluation showed that the system, while far from perfect, has good tracking performance under the less than perfect environmental conditions in which it was tested (flickering lights, similar colors in environment to object tracked, motion in the background, etc.). Most importantly, the system can run autonomously on a robot with limited computational power, which is what distinguishes it from most other systems.

We are currently investigating possibilities to improve the system performance by adding optic flow methods, although there seem to be intrinsic computational barriers involved. We are also trying to additional low-cost methods of motion estimation in an effort to add predictive camera movements, which should help in cases where the robot now loses track of an object, because the camera is too slow.

References

[1] A. Bevilacqua. Optimizing parameters of a motion detection system by means of a genetic algorithm. In *Proceedings of the 11-th International Conference in Central Europe on Computer Graphics, Visualization and Computer*, 2003.

[2] M. J. Black and A. D. Jepson. Eigentracking: Robust matching and tracking of articulated objects using a view-based representation. In *ECCV (1)*, 329–342, 1996.

[3] J. Bruce, T. Balch, M. Veloso. Fast and Inexpensive Color Image Segmentation for Interactive Robots. In *Proceedings of IROS 2000*, 2000.

[4] F. Dellaert and R. Collins. Fast image-based tracking by selective pixel integration. In *In Proceedings of The ICCV Workshop on Frame-Rate Vision*, 1999.

[5] S. Sclaroff and J. Isidoro. Active blobs. In *Proceedings of the International Conference on Computer Vision*, 1998.

[6] M. Xu and T. Ellis. Illumination-invariant motion detection using colour mixture models. In *Proceedings of the British Machine Vision Conference BMVC2001*, 2001.