Eyepatch: Prototyping Camera-based Interaction through Examples

Dan Maynes-Aminzade, Terry Winograd
Stanford University HCI Group
Computer Science Department
Stanford, CA 94305-9035, USA
[monzy | winograd]@cs.stanford.edu

Takeo Igarashi
University of Tokyo
Computer Science Department
7-3-1 Hongo, Bunkyo, Tokyo, 113-0033 Japan
takeo@acm.org

ABSTRACT
Cameras are a useful source of input for many interactive applications, but computer vision programming is difficult and requires specialized knowledge that is out of reach for many HCI practitioners. In an effort to learn what makes a useful computer vision design tool, we created Eyepatch, a tool for designing camera-based interactions, and evaluated the Eyepatch prototype through deployment to students in an HCI course. This paper describes the lessons we learned about making computer vision more accessible, while retaining enough power and flexibility to be useful in a wide variety of interaction scenarios.

ACM Classification: H.1.2 [Information Systems]: User/Machine Systems — Human factors; H.5.2 [Information Interfaces and Presentation]: User Interfaces — interaction styles, prototyping, theory and methods; I.4.8 [Image Processing and Computer Vision]: Scene Analysis — color, object recognition, tracking; I.4.9 [Image Processing and Computer Vision]: Applications.

General Terms: Algorithms, Design, Human Factors. **Keywords:** Computer vision, image processing, classification, interaction, machine learning.

INTRODUCTION
Many compelling systems have used cameras as an interactive input medium, from the pioneering work by Myron Krueger [22] to projects like Light Widgets [9], EyePliances [29], and Gesture Pendant [30], tangible interfaces such as Illuminating Light [31] and the Designers' Outpost [20], game interfaces [10] including crowd interaction [26] and the Sony Eyetoy [23], and platforms like PlayAnywhere [35], TouchLight [36], and the camera-based SmartBoard. Today, the cost of a digital camera is comparable to that of a traditional mass-market input device like a keyboard or mouse, and cameras are already integrated into many of our devices, such as cell phones, PDAs, and laptop computers. While these cameras are typically only used for video conferencing and taking pictures, recent advances in computing power open the door to their use as an additional channel of input in a wide variety of applications, giving "eyes" to our everyday devices and appliances.

Unfortunately, designing camera-based interfaces is still quite difficult. There are a number of powerful tools for computer vision such as MATLAB and OpenCV [4], but these tools are designed by and for programmers. They require a fairly advanced level of programming skill, and in some cases a sophisticated understanding of the mathematics of image processing and machine learning techniques. Vision prototyping tools like Crayons [8] and Papier-Mâché [19] are an inspiring step in the right direction, but their frameworks place restrictive constraints on the types of applications that can be built, as we shall discuss.

RESEARCH GOALS
With the goal of simplifying the process of developing computer vision applications, we built a tool called Eyepatch that allows novice programmers to extract useful data from live video and stream that data to other rapid prototyping tools, such as Adobe Flash, d.tools [14], and Microsoft Visual Basic. When designing Eyepatch, we made the assumptions that our target community was comfortable using a visual design tool and had some basic

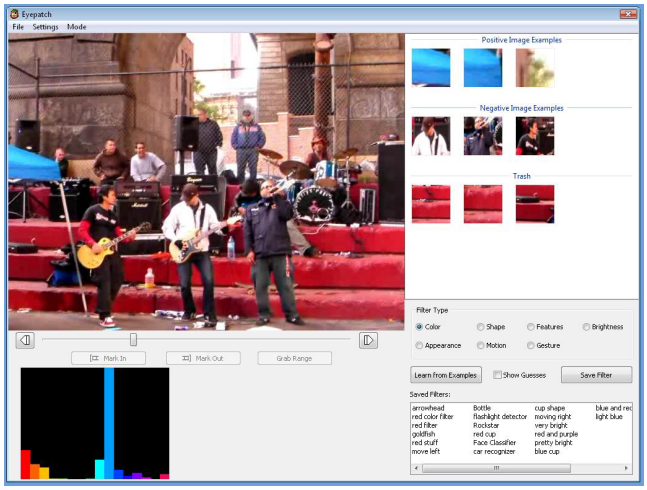

Figure 1 Eyepatch in training mode. Here the user is training a color classifier; the pane at the right shows the examples she has chosen, and the internal state of the classifier is represented by hue histogram in the lower pane.

UIST'07, October 7–10, 2007, Newport, Rhode Island, USA.

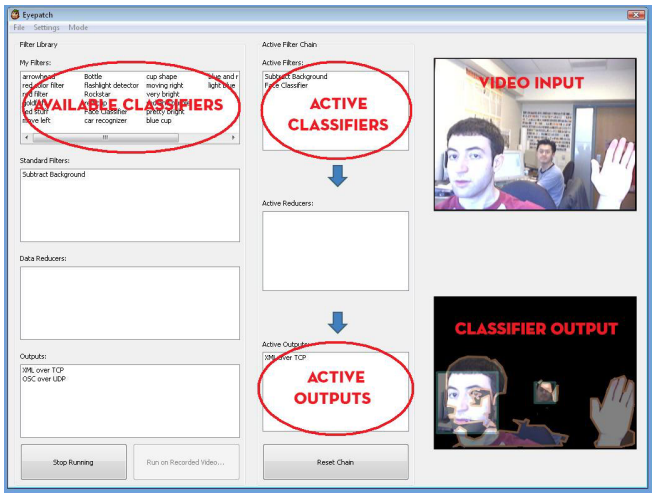

Figure 2 Eyepatch in composition mode, in which users select classifiers to run on live video input. Two classifiers are enabled in this example, and their outputs are shown in the bottom right video pane, outlined in different colors.

programming experience, but little or no knowledge of computer vision, image processing, or machine learning techniques. Our objective was to enable users to get the information they needed from a camera without any specialized computer vision programming.

Our goal was not to support highly complex, data-intensive computer vision techniques such as 3-dimensional reconstruction of object geometry or stereo range finding. Instead, we wanted users to be able to use computer vision techniques to answer simple questions based on video data: Does my houseplant need watering? Is there any coffee left in the coffeepot? What is the weather like outside? Are there any eggs in my refrigerator? Who left those dirty dishes in the sink? Has the mail been delivered yet? How many police cars pass by my window on a typical day? These sorts of simple questions can lead to compelling application scenarios, and answering them does not require detailed, high-level image understanding. Instead, questions such as these can be answered using relatively simple classifiers, provided the designer can train and calibrate these classifiers for the context of the application.

RESEARCH AGENDA

We base our overall research strategy on iterative prototyping. After creating each version of Eyepatch, we provide it to designers, and observe what they can achieve with it and where they encounter problems. This gives us insights that help us improve subsequent versions.

We wrote the first version of Eyepatch as a collection of ActiveX Controls in Visual Basic, each control custom-built to solve a particular computer vision problem, such as tracking faces or finding laser pointer dots. We deployed this first version to students in an HCI class [15] (Figure 3), but found that the highly specific nature of the ActiveX Controls constrained the students to a very narrow range of possible applications, and the ActiveX architecture

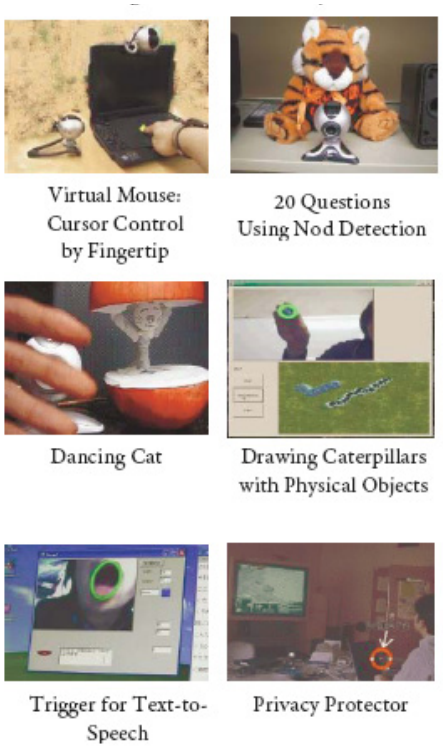

Figure 3 Sample projects created by students at the University of Tokyo using the first version of Eyepatch.

prevented students from using the camera data in other prototyping tools.

This paper describes our second version of Eyepatch, which was designed to allow users more creativity in developing their own computer vision algorithms. We wanted to enable designers to attack a broader range of problems, and use a wider variety of prototyping tools. At the same time, we tried to keep things as clear-cut as possible by making certain simplifying assumptions in our framework. We assumed that the input came from a *camera or recorded video*, the sequence of frames was passed through one or more *binary classifiers*, and the only output was the *set of image regions* yielded by each of the active classifiers. Our expectation was that this framework would still be too limited for some real-world applications. However, by examining what our users could build and what was out of their reach, we hoped to gain an understanding of what features are needed in a camera-based interaction design tool in order for it to achieve widespread use.

We will begin our discussion of Eyepatch with an overview of its framework. We will outline its capabilities and our rationale for designing it as we did. We will then describe our evaluation process, and recount the lessons we learned in our class deployment. We will conclude by framing our contributions in the context of related work.

EYEPATCH OVERVIEW

Eyepatch has two basic modes:

- A **training mode**, where users can train different types of classifiers to recognize the object, region, or parameter that they are interested in.
- A **composition mode**, where the classifiers are composed in various ways and users specify where the output of the classifiers should go.

Composition Mode

The Eyepatch **composition mode** is shown in Figure 2. Its functionality is best explained through an example scenario. Suppose a designer is prototyping an interactive whiteboard system in Flash, and she wants to incorporate input from a camera mounted above the whiteboard. Assume she wants to know (a) when someone is standing in front of the whiteboard and (b) whether or not that person is looking at the whiteboard. The designer enters composition mode and chooses two classifiers, one that recognizes faces and another that performs adaptive background subtraction to identify foreground regions in the image, which in this case are assumed to be people. She then specifies an output path to which this data should be sent, in this case XML over TCP. She clicks "Run on Live Video," which activates the camera and begins running the classifiers on the live video, streaming the data to the specified output. She then returns to her Flash prototype and adds a few lines of ActionScript to connect to a local socket and read this XML data.

Note that although the process of building her application did require some programming in order to read the camera data and do something useful with it, it did not require domain-specific knowledge about computer vision. She could begin using camera input without knowing the details of statistical background models or boosted classifier cascades, and the ActionScript to read the incoming XML data could be copied from one of our toolkit examples.

In this scenario, only built-in classifiers were needed.

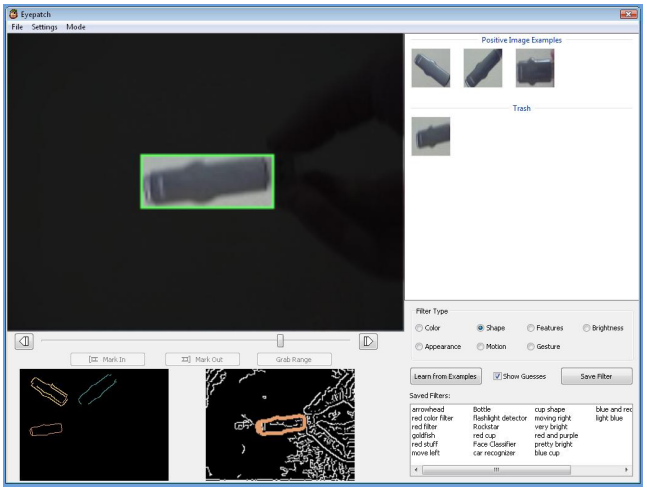

Figure 4 Shape classifiers in Eyepatch use Canny edge detection followed by contour matching to identify the objects of interest. Shape classifiers work best for objects with distinctive outer contours.

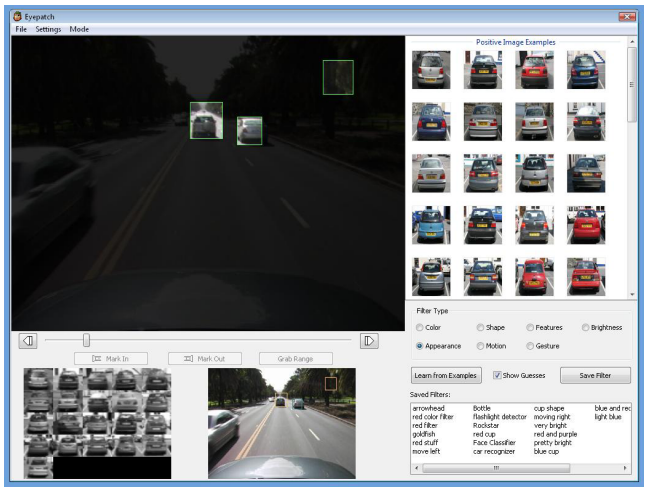

Figure 5 Eyepatch allows users to train a boosted classifier cascade using examples from recorded videos, and immediately observe its performance on live or recorded video.

Eyepatch includes a number of these special-purpose classifiers for common problems such as face detection, background subtraction, and motion detection. However, we expect it will be equally common for users to train their own classifiers that are tailored to their application and its context. The Eyepatch training mode allows users to create their own classifiers using examples drawn from a recorded video.

Training Mode

The Eyepatch **training mode** (Figure 1) uses an interactive learning approach similar to that of Crayons [8], with several key differences. First, Eyepatch operates on *video* instead of still images. This is important for recognizing dynamic motion sequences, and it also simplifies the process of capturing large amounts of training data. The user loads a video in any standard format, or records a video from an attached webcam. The user can scroll through the video, much like in a video editor. She can then use the mouse to highlight regions of frames to use as examples for training a classifier.

Second, while the Crayons system requires Java, Eyepatch allows formatted data to be exported over network sockets to communicate with a wide variety of prototyping tools. We felt that the ability to integrate with many different prototyping tools was an important way to support the common practices of designers.

Third, rather than a single type of classification algorithm, Eyepatch supports a variety of classification strategies. This is important if we wish to support a diverse array of applications. For example, a boosted cascade of Haar classifiers works well for identifying objects of a general class, such as cars or faces, while SIFT features work better for identifying a particular instance of a class, such as a specific book cover or logo. Neither of these feature-based classifiers performs well when attempting to identify smooth, untextured objects like balloons or laser dots, since

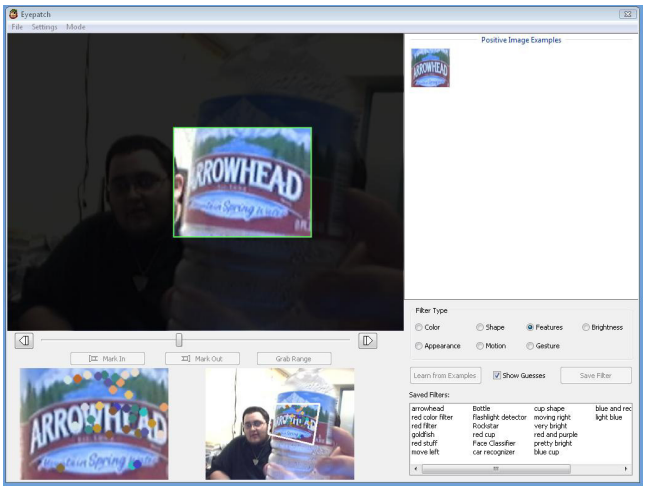

Figure 6 SIFT produces classifiers with better invariance to object pose, scale, and illumination. The colored dots in the classifier viewing pane show the feature correspondences found between the current frame and the training example.

they have few distinguishable features; in this case a classifier based on hue histograms or brightness thresholding is more appropriate. When following a moving object, sometimes the most important factor is the change in the image over time; in this situation the most appropriate approach is to use a technique like frame differencing, motion templates, adaptive background subtraction, or optical flow.

We do not expect our users to have a detailed understanding of the *underlying machinery* behind each classifier type, so they may not always select the appropriate classification strategy on their first try. To make it as painless as possible to find the appropriate classification strategy for the problem at hand, our interface was designed to facilitate *rapid trial-and-error*. In a matter of seconds, the user can switch to a different classification method, train it on the same set of examples, and observe its output. This allows the user to quickly select the optimal strategy.

After selecting some examples from the video feed, the user selects a classification method and clicks "Learn from Examples" to train a classifier. He can then check "Show Guesses" to observe the output of the trained classifier on the current frame. By scrolling through the frames in rapid succession, he can quickly judge the performance of the classifier, and if it is not satisfactory, he can provide more examples or try a different classification method.

An alternate strategy might be to run *all* of the classifier types and ask the computer select the most successful one. At first this strategy seems much easier, but it introduces two challenges. First, it would run much more slowly and eliminate the opportunity for quick iteration, which would violate the *fast-and-focused* UI principle used successfully in the Crayons system. The advantage of the *fast-and-focused* approach is that when the user sees an incorrect classification, he can interactively add the misclassified region to the training set for the next training iteration. Since the misclassified regions are often edge cases, this lets the user quickly select the most relevant examples as input to the classifier. Second, training all of the classifiers at once would ignore any of the knowledge that the user could bring to the table about which classification strategies he thought might be the most appropriate. In practice we found that after enough learning through trial-and-error, users began to develop helpful intuitions about which classifier types worked best for which problems, and this reduced the training time greatly.

ADVANTAGES OF MULTIPLE CLASSIFIER TYPES

One advantage provided by our approach is that users can create several weak classifiers and combine their output into a single classifier with a lower rate of false positives. For example, a color classifier might do a good job of extracting skin-colored regions, but if the user was trying to recognize hand gestures, he would want to ignore faces. By combining the color classifier with a motion classifier that identifies moving regions of the image, he could find only the moving skin-colored regions.

In addition to eliminating false positives, this composition of simple classifiers can allow users to identify more complex types of events. For example, users can train a car classifier, a classifier that recognizes red objects, and a classifier that recognizes objects moving left in the image. They can then combine the outputs of these classifiers to recognize red cars moving left.

Eyepatch currently supports seven classifier types:

1. **Color**, based on hue histograms and backprojection (similar to CAMSHIFT [5]), for identifying distinctively colored objects.
2. **Brightness**, for finding the brightest or darkest regions of an image, such as laser dots or shadows.

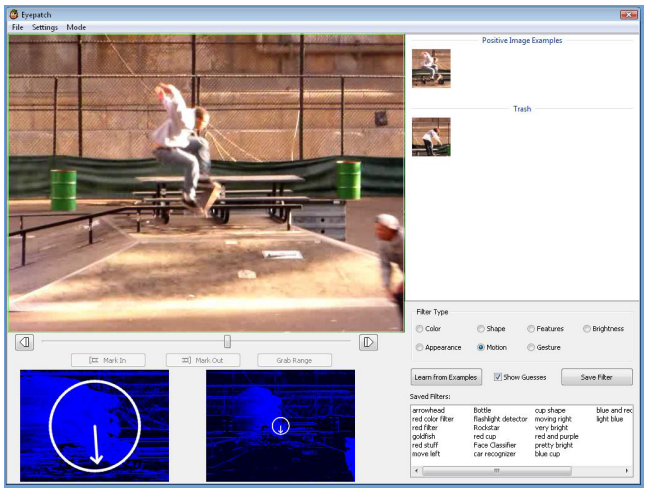

Figure 7 To train a motion classifier, the user selects frames or regions of frames that contain the desired type of motion, and the motion history image is segmented to extract the overall motion directions. These are compared against motion in the current frame. Here, the user is looking for skateboarders who are moving downward.

3. **Shape**, based on Canny edge detection followed by contour matching using pair-wise geometrical histograms [16], for finding objects with distinctive outer contours (Figure 4).
4. **Adaboost**, a machine learning technique that uses a boosted cascade of simple features [32], for recognizing general classes of objects like faces, animals, cars, or buildings (Figure 5).
5. **Scale-Invariant Feature Transforms** [24], for recognizing specific objects with invariance to scale, pose, and illumination (Figure 6).
6. **Motion**, based on segmentation of a motion history image [6], for identifying the directions of moving objects in the scene (Figure 7).
7. **Gesture recognition**, based on blob detection followed by motion trajectory matching [3] using the Condensation algorithm [17], for recognizing particular patterns of motion (Figure 8).

This wide variety of classifier types allows users to tackle problems that cannot be solved with simple pixel-based classifiers, which do not incorporate global image data like contour shapes or the relative positions of image features. SIFT classifiers allow users to build recognizers that are invariant to scale, rotation, and illumination, and Adaboost classifiers let users train recognizers that operate on general classes of objects. The motion and gesture classifiers allow for applications that recognize directions and patterns of motions across time, so that applications are not limited to operating on static images.

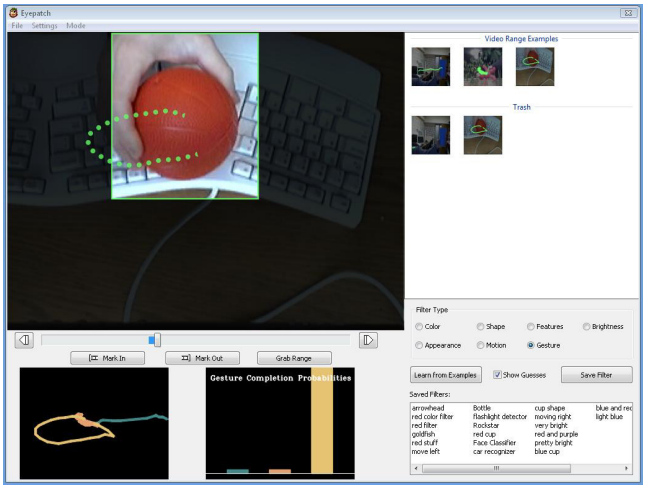

Figure 8 In gesture training mode, the user selects the intervals of video that contain examples of the motion trajectories he wishes to recognize. Motion trails are shown as overlays on the video frame. When run on live or recorded video, a particle filtering technique is used for motion matching, and the match probabilities for each gesture are shown in the classifier viewing pane. In this example, three gestures have been trained, a shaking motion, a straight line, and a circular gesture; the circular gesture is recognized in this frame.

SAMPLE PROJECTS

Describing a few projects we built with Eyepatch will provide a sense of its capabilities. We completed each of the following projects in a few hours, with the majority of this time devoted to components of the projects unrelated to computer vision.

BeiRobot

BeiRobot (Figure 9) is a Lego robot that uses computer vision to play Beirut. Beirut is a party game in which players attempt to throw ping-pong balls into cups. BeiRobot uses a color classifier to detect cup positions in the input image, and it rotates its swiveling base until one of the cups is centered in the image. It then launches a ping-pong ball at the cup, using the size of the detected cup region to estimate the cup's distance from the camera.

Stop Sign Warning Device

The stop sign warning device (Figure 10) warns drivers when they are about to run through a stop sign. It uses an Adaboost classifier to recognize stop signs, and a motion classifier to detect when the car is moving. If a stop sign is detected close to the camera, and the car continues forward without stopping, the device emits a warning tone.

Logo Scoreboard

The logo scoreboard (Figure 11) uses SIFT classifiers to recognize an assortment of five different corporate logos. It watches live television and counts how many times each logo appears. A system like this would allow marketers to assess the brand penetration of their company or evaluate the success of their product placement campaigns.

EVALUATION

Starting with this framework, we began evaluating Eyepatch to see which types of applications it could support, which it could not, and what capabilities were needed for a tool of this nature to be useful to designers. For the purpose of this evaluation, we offered a one-quarter course on computer vision for HCI called "Designing Applications that See" [25]. In a series of hands-on workshop sessions, the students in the class were introduced to a variety of computer vision tools, including MATLAB, OpenCV, JMyron (an extension library for the Processing [28] development environment), and Eyepatch. They were also given a general introduction to the common techniques used in computer vision and image processing, without delving into any complex mathematical details (the lecture notes did not include a single equation). After five weeks of being introduced to the tools, we asked the students to spend five weeks building working prototypes of camera-based applications, using the tools of their choice.

Of the 19 students enrolled in the class, 17 were Masters students in Computer Science or Electrical Engineering and 2 were Computer Science undergraduates. This collection of students had more extensive programming experience than our target community for Eyepatch, but this provided us with a good stress test; because of their technical expertise, the students tackled problems that novice

programmers might not have attempted. As the students hit the limits of what Eyepatch allowed them to achieve, we developed an understanding of what was needed in a computer vision design tool for it to work well on real-world problems.

LESSONS LEARNED

All of the student teams that began their projects using Eyepatch eventually found its limitations. Of the eight project teams, five began their project with Eyepatch, but as their projects progressed, they encountered things that they wanted to do with our framework but could not, and switched to more complex, more general-purpose tools like OpenCV and MATLAB. We shall discuss the specific limitations that the students encountered in more detail, but it is worth noting that many of the teams still found Eyepatch useful in the initial prototyping phase. For example, one group of students built a virtual ping-pong game using real ping-pong paddles as controllers. Although they eventually migrated to OpenCV so that they could stream live video between game opponents, they used Eyepatch to quickly experiment with different strategies for tracking the position of a ping-pong paddle, using shape, color, and feature-based classifiers. Once they had settled on a strategy that worked well, they incorporated it into their OpenCV code, which was more laborious to modify.

Although Eyepatch is intended primarily as a tool for rapid prototyping, ultimately we would like to make its *ceiling* [27] as high as possible. The lessons we learned from the class gave us many ideas for improving Eyepatch, not only for raising its ceiling but also for advancing its interaction model and simplifying the classifier training process. In this section we will describe these lessons, illustrating them with particular examples from the class.

Provide image data in addition to classifier output. Many students wanted to display camera images to the end users of their applications. For example, one group of students built a game in which players moved their heads around to hit targets. The students needed to display an image of the players and overlay graphics on this image. While Eyepatch was capable of detecting face positions, it did not provide a way of exporting the image frames along with the output of the active classifiers. By focusing on translating image data into simple output parameters, we neglected the importance of using the camera image itself as output.

Allow data selection and filtering. One of our simplifying assumptions was to discard data that we thought would not be useful. Although the actual output of the trained classifiers was a set of bounding polygons around the detected regions, we sent only the region centers and areas to the output stream. This is enough data for many applications, but sometimes more detail is required. For example, one group built an animation system using hand tracking, and they wanted to use the shape of the hand

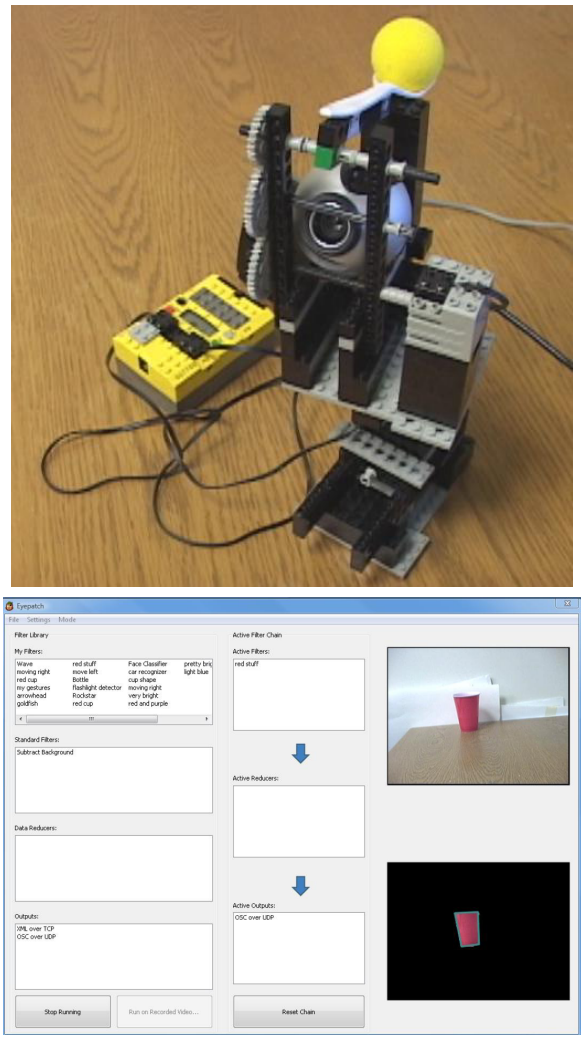

Figure 9 The BeiRobot fires ping-pong balls at red cups, which it finds using a color classifier.

region to recognize certain gestures. This was impossible if they only knew the center and area of each hand.

In our next version, each classifier will allow the user to select among all of its possible outputs. This way the user can decide which data to keep and which to discard, by choosing from a list of region parameters, such as position, bounding box, area, perimeter, eccentricity, image moments, and so on.

Provide a mechanism for data reduction. In other situations, Eyepatch actually provided too much data. For example, one group simply wanted to know whether or not there was something moving in the image. They were essentially trying to reduce the entire image to a single bit of data. In this case, sending out a list of the moving regions in the image was excessive.

We plan to address this problem by providing data reducers that can be added to the output of each classifier. The most commonly requested data reducers were:

- **Simple binary output** (0 if there are no detected regions, 1 otherwise), to tell the user if a certain

condition has been met (for example, tell me when there is someone at my front door).

- **Number of detected regions**, to tell the user how many of the objects of interest are visible in the scene (for example, to count the number of cars on the freeway).
- **Total area of all detected regions**, to inform the user how much of something is in the image (for example, how much coffee is remaining in a coffee pot) or how close an object is to the camera based on its size.

Allow users to combine multiple classifiers of the same type into a single classifier that recognizes multiple objects. One group built a system that recognized pills to help verify correct dosages. They had a database that contained the attributes of each pill, such as size, shape, and color. With the existing framework, they needed to train one classifier for each pill, and then run each of these classifiers in sequence to see which classifiers detected a pill. Since the classifiers were all looking at the same attributes of the image, much better performance could have been achieved in a single pass. What was needed was a way to merge together the pill classifiers, producing a single classifier that output the set of all positive matches against the database, instead of running numerous classifiers, one for each pill, with empty outputs from the majority of the classifiers.

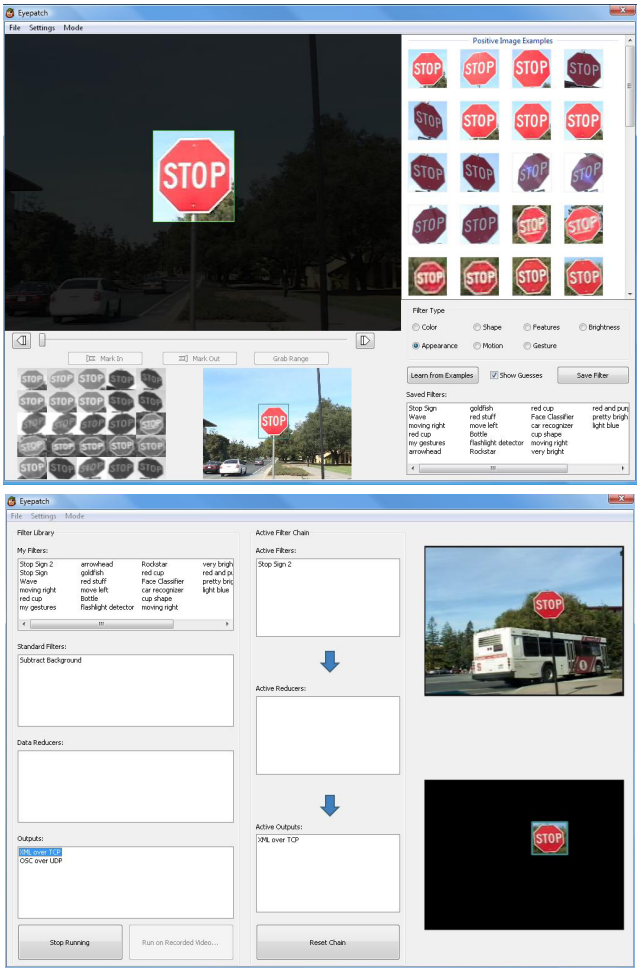

Figure 10 Eyepatch was used to prototype an in-car warning device that used motion detection and Adaboost classifiers to alert drivers to stop signs they might have missed.

Provide the ability to adjust classifier thresholds. When designing any computer vision-based application, there will be some ambiguity in the input. This ambiguity results in a design tradeoff: decrease the recognition threshold and you get more false positives; increase the threshold and you miss detecting valid events. It is important for the designer to be able to adjust the threshold to a level that is appropriate for his application. For example, a "free food detector" that emails users when there is leftover food in the common kitchen should have a high threshold, since an inbox full of spurious food notifications is more annoying than a few missed snacking opportunities. On the other hand, a system that detects fires or warns people that they are about to receive parking tickets deserves a lower threshold, since the penalty of a missed detection is so high.

Our next version will offer a simple slider that allows users to set the detection threshold. While dragging this slider, users will immediately see the detected instances appear and disappear in the input video, allowing them to visualize the effect of the threshold level on their application.

Support temporal filtering for object coherence across frames. A common problem when using a classifier to track an object through a scene is that the classifier will fail on a single frame but then recover the object soon after. Several groups needed to track objects reliably from frame-to-frame, and they had to come up with custom solutions to this problem. The simplest such solution was to assume that the tracked object was in the same position as before if it was not detected in the current frame; only when it was not detected for several frames in a row was it actually assumed to have left the camera view.

Objects being tracked will not generally blink in and out of existence from one frame to the next, so most successful tracking systems use adaptive techniques like Kalman filtering that incorporate the previous positions of a tracked object into the estimate of its current position. Our next version will support this type of temporal filtering, to preserve object coherence across frames and to smooth detected motion paths.

Accelerate the example-collecting process. Eyepatch users can train a cascade of boosted classifiers by selecting examples from video frames. Although our students found this method much simpler to use than the command-line tool bundled with OpenCV (which requires users to type the pixel coordinates of object bounding rectangles into a text file), it can still be a tedious process to step through frames highlighting the positions of the objects of interest. Our students suggested a simpler way of adding multiple examples in rapid succession by following an object with the mouse as the video advanced automatically. In addition,

students did not see the need for providing negative examples to train boosted classifiers, since the system could simply take the inverse regions of their positive examples and use them as negative examples. This would work well provided that users always highlighted all of the positive examples in a frame, and it would eliminate a lot of the busywork involved in training a boosted classifier.

Allow direct manipulation of the classifier model. Eyepatch displays a representation of the internal state of a classifier, but there is no way to directly modify this representation; instead, the user is forced to add or remove examples to modify it indirectly. For example, the color classifier displays a hue histogram of the colors in the training examples. Some users wanted to drag the bars of the histogram manually. This way they could, for example, increase the amount of red in the histogram template without having to find the reddest part of the image to use as a new example. Some classifier types, such as Adaboost, have no obvious directly manipulable representation, since their internal state is a set of weights assigned to different Haar features. Finding a better way to expose the state of this type of complex machine learning algorithm to the user would be an interesting area of future research; for example, it may be useful to produce a visualization indicating to the user which of the examples in the training set were generally the same as the others, and which examples stood out and did a better job of separating the positive and negative elements of the training set.

Provide a plug-in architecture. Some of the students in our class were skilled programmers, and they pointed out that many of the restrictions of Eyepatch could be overcome if more advanced users could simply program their own classifier types. For example, one team of students built a card game that relied on recognizing an assortment of special glyphs, a common technique in augmented reality applications. Although the students could have used one of the standard classifier types in Eyepatch, they would have had to train a separate classifier for each glyph. The problem of recognizing specially-designed glyphs is so specific that using a specialized classifier is a more efficient approach, so the students decided to use the ARToolkitPlus library [33]. If we gave Eyepatch a plug-in architecture, users in a situation like this could simply write a new classifier based on this library. A plug-in architecture would allow Eyepatch to mature and add new functionality as its user base added new classifier types.

RELATED WORK

Our greatest inspiration for this work came from the Crayons design tool for camera-based interaction [8], and we based our interactive learning approach on its "paint, view, and correct" process. Many of our ideas for extending the Crayons model were based on the future work proposed by its authors, such as supporting additional feature types and taking motion into account.

We also drew inspiration from Exemplar [13], which provides a similar example-based approach to training a classifier. The data classified in Exemplar is multiple channels of one-dimensional sensor input, but the approach of building a model through examples, viewing the state of the created model, and adjusting it for better results, is very similar to the approach we adopted in Eyepatch.

The Papier-Mâché toolkit [19] provided a high-level programming abstraction that allowed users to extract certain events from a camera input without worrying about the underlying details of the computer vision algorithms. Its event model was designed to parallel an RFID reader, and could trigger an event when particular objects were added to or removed from the camera view. It also provided the ability to extract certain basic parameters from the objects, such as average color and image moments. This made it very easy to program certain types of computer vision applications, but its single classifier type constrained the types of applications it could produce, and its event model did not adapt well to applications that required dynamic tracking at interactive frame rates.

The Cambience system [7] allowed users to select regions

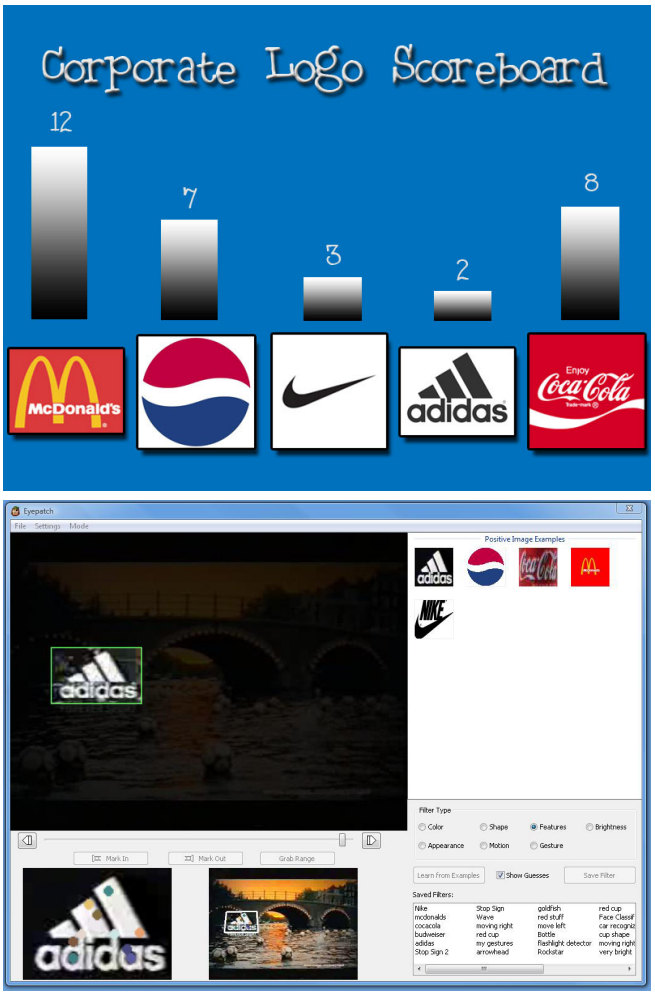

Figure 11 The Logo Scoreboard watches live television and measures the prevalence of some common corporate logos.

of a video frame in which they wanted to detect motion, and then map the motion in each region to a different sound effect. The system also incorporated certain types of control over data flow and filtering, for example to map the intensity of motion to the volume of a sound. We adopted a similar dataflow model, and motion classifiers in Eyepatch have approximately the same functionality. However, Cambience focused on only one type of input (motion) and one type of output (ambient soundscapes).

The Lego Mindstorms [2] "Vision Command" Kit was one of the first systems to attempt to make computer vision accessible to novice programmers. Its visual programming interface allowed users to test for simple conditions in a camera image: when a certain color or a certain brightness threshold was seen in one of five regions of the frame, an event could be triggered. This highly simplified model worked well for certain tasks, like building a robot that followed a white line, or sorted bricks based on their color. Sensing brightness and color are much easier concepts to understand than recognizing particular configurations of image features, and indeed they were popular classifier types in Eyepatch. However, we found that users frequently wanted to detect events or objects that could not be recognized by these simple classifier types alone.

Computer vision programming has been greatly simplified by the many general-purpose software libraries that have been developed over the years, including XVision [11], Mimas [1], OpenCV [4], and the NASA Vision Workbench [12]. There are also various libraries that are very effective at solving particular problems in computer vision, such as GT2K [34] for HMM-based gesture recognition, HandVu [21] for detecting hand pose, and ARToolkit [18] for determining the 3-dimensional position of glyph markers. Although these libraries provide powerful shortcuts when developing camera-based applications, they are designed by and for programmers; they require fairly in-depth programming knowledge to use, and their functionality is generally couched in terms of the underlying mathematical operations rather than the high-level goals. As such, they do not lend themselves to quick iteration by designers, and we believe that the visual, example-based approach used in Eyepatch is better suited to rapid prototyping, especially at the early stages of application development.

CONCLUSION

Deploying Eyepatch to a project class in a longitudinal study was an excellent way to gain insight into its strengths and weaknesses. We will continue to refine and evaluate Eyepatch using our iterative prototyping process. Because the majority of our testing was on Computer Science graduate students, our evaluation was more effective at finding the ceiling of Eyepatch that at measuring its threshold. We hope to learn another set of valuable lessons when we test Eyepatch on undergraduate artists and designers.

Our evaluation process revealed a need for many new features. In incorporating these features into the next version of Eyepatch, we face a design challenge: we must strive to preserve the simplicity of the original design for first-time users, while providing experienced users the advanced functionality they need to develop complex applications. We hope that by providing several levels of progressive disclosure, we can offer this functionality while managing the complexity of the system and adhering to our initial design goals.

Although we learned that Eyepatch still has much room for improvement, we believe that it represents an important step towards making camera input accessible to interaction designers. Eyepatch allows designers to create, test, and refine their own customized classifiers, without writing any specialized code. Its diversity of classification strategies makes it adaptable to a wide variety of applications, and the fluidity of its classifier training interface illustrates how interactive machine learning can allow designers to tailor a recognition algorithm to their application without any specialized knowledge of computer vision.

Software Availability

Eyepatch is open source software licensed under the GPL. A Windows installer for Eyepatch is available for download at http://eyepatch.stanford.edu/.

ACKNOWLEDGMENTS

We thank the students of CS377S for their creative project ideas, their valuable design suggestions, and their helpful bug reports.

REFERENCES

1 Amavasai, B. P. Principles of Vision Systems. *IEEE Systems, Man, and Cybernetics: Principles and Applications Workshop*, 2002.

2 Bagnall, B., *Core LEGO MINDSTORMS Programming*: Prentice Hall PTR Upper Saddle River, NJ, USApp. 2002.

3 Black, M. J. and A. D. Jepson. A probabilistic framework for matching temporal trajectories: Condensation-based recognition of gestures and expressions. *European Conference on Computer Vision* **1**. pp. 909–24, 1998.

4 Bradski, G. The OpenCV Library. *Dr. Dobb's Journal November 2000, Computer Security*, 2000.

5 Bradski, G. R. Computer vision face tracking for use in a perceptual user interface. *Intel Technology Journal* **2**(2). pp. 12-21, 1998.

6 Bradski, G. R. and J. W. Davis. Motion segmentation and pose recognition with motion history gradients. *Machine Vision and Applications* **13**(3). pp. 174-84, 2002.

7 Diaz-Marino, R. and S. Greenberg. CAMBIENCE: A Video-Driven Sonic Ecology for Media Spaces. *Video Proceedings of ACM CSCW'06 Conference on Computer Supported Cooperative Work*, 2006.

8 Fails, J. and D. Olsen. A design tool for camera-based interaction. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 449-56, 2003.

9 Fails, J. A. and D. R. Olsen. Light widgets: interacting in every-day spaces. *Proceedings of IUI'02*, 2002.

10 Freeman, W. T., *et al.* Computer vision for interactive computer graphics. *Computer Graphics and Applications, IEEE* **18**(3). pp. 42-53, 1998.

11 Hager, G. D. and K. Toyama. X Vision: A portable substrate for real-time vision applications. *Computer Vision and Image Understanding* **69**(1). pp. 23-37, 1998.

12 Hancher, M. D., M. J. Broxton, and L. J. Edwards. A User's Guide to the NASA Vision Workbench. *Intelligent Systems Division, NASA Ames Research Center*, 2006.

13 Hartmann, B., L. Abdulla, M. Mittal, and S. R. Klemmer. Authoring Sensor Based Interactions Through Direct Manipulation and Pattern Matching. *CHI: ACM Conference on Human Factors in Computing Systems*, 2007.

14 Hartmann, B., *et al.* Reflective physical prototyping through integrated design, test, and analysis. *Proceedings of the 19th annual ACM symposium on User interface software and technology*. pp. 299-308, 2006.

15 Igarashi, T., *Student projects in the "Media Informatics" course at the University of Tokyo*, 2005. http://www-ui.is.s.u-tokyo.ac.jp/~kwsk/media2005/projects.html

16 Iivarinen, J., M. Peura, J. Sarela, and A. Visa. Comparison of combined shape descriptors for irregular objects. *Proceedings of the 8th British Machine Vision Conference* **2**. pp. 430–39, 1997.

17 Isard, M. and A. Blake. Contour Tracking by Stochastic Propagation of Conditional Density. *Proceedings of the 4th European Conference on Computer Vision-Volume I-Volume I*. pp. 343-56, 1996.

18 Kato, H. and M. Billinghurst. Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System. *Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality* **99**. pp. 85-94, 1999.

19 Klemmer, S. R. Papier-Mâché: Toolkit support for tangible interaction. *CHI: ACM Conference on Human Factors in Computing Systems*, 2004.

20 Klemmer, S. R., M. W. Newman, R. Farrell, M. Bilezikjian, and J. A. Landay. The Designers' Outpost: A Tangible Interface for Collaborative Web Site Design. *The 14th Annual ACM Symposium on User Interface Software and Technology: UIST2001, CHI Letters* **3**(2). pp. 1-10, 2001.

21 Kolsch, M., M. Turk, and T. Hollerer. Vision-based interfaces for mobility. *Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on*. pp. 86-94, 2004.

22 Krueger, M. W., T. Gionfriddo, and K. Hinrichsen. VIDEOPLACE—an artificial reality. *ACM SIGCHI Bulletin* **16**(4). pp. 35-40, 1985.

23 Larssen, A. T., L. Loke, T. Robertson, and J. Edwards. Understanding Movement as Input for Interaction–A Study of Two EyeToy™ Games. *Proceedings of OZCHI 2004*, 2004.

24 Lowe, D. G. Object recognition from local scale-invariant features. *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on* **2**, 1999.

25 Maynes-Aminzade, D., *Website for the course "Designing Applications that See" at Stanford University*, 2007. http://cs377s.stanford.edu/

26 Maynes-Aminzade, D., R. Pausch, and S. Seitz. Techniques for interactive audience participation. *Fourth IEEE International Conference on Multimodal Interfaces*. pp. 15-20, 2002.

27 Myers, B., S. E. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)* **7**(1). pp. 3-28, 2000.

28 Reas, C. and B. Fry. Processing: a learning environment for creating interactive Web graphics. *Proceedings of the SIGGRAPH 2003 conference on Web graphics: in conjunction with the 30th annual conference on Computer graphics and interactive techniques*. pp. 1-1, 2003.

29 Shell, J. S., R. Vertegaal, and A. W. Skaburskis. EyePliances: attention-seeking devices that respond to visual attention. *Conference on Human Factors in Computing Systems*. pp. 770-71, 2003.

30 Starner, T., J. Auxier, D. Ashbrook, and M. Gandy. The Gesture Pendant: A Self-illuminating, Wearable, Infrared Computer Vision System for Home Automation Control and Medical Monitoring. *International Symposium on Wearable Computing*, 2000.

31 Underkoffler, J. and H. Ishii. Illuminating light: an optical design tool with a luminous-tangible interface. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 542-49, 1998.

32 Viola, P. and M. Jones. Rapid object detection using a boosted cascade of simple features. *Proc. CVPR* **1**. pp. 511-18, 2001.

33 Wagner, D. and D. Schmalstieg. Handheld Augmented Reality Displays. *Proceedings of the IEEE Virtual Reality Conference (VR 2006)-Volume 00*, 2006.

34 Westeyn, T., H. Brashear, A. Atrash, and T. Starner. Georgia tech gesture toolkit: supporting experiments in gesture recognition. *Proceedings of the 5th international conference on Multimodal interfaces*. pp. 85-92, 2003.

35 Wilson, A. D. PlayAnywhere: a compact interactive tabletop projection-vision system. *Proceedings of the 18th annual ACM symposium on User interface software and technology*. pp. 83-92, 2005.

36 Wilson, A. D. TouchLight: an imaging touch screen and display for gesture-based interaction. *Proceedings of the 6th international conference on Multimodal interfaces*. pp. 69-76, 2004.