Shadow Vision

Mark Huang
Shishir Mehrotra
Flavia Sparacino

6.837
Introduction to Computer Graphics
Fall 1999

1.0 Abstract

Shadow Vision attempts to create a virtual shadow puppet theater A user's hand motions over an overhead projector are used to direct object creation and manipulation in a 3D OpenInventor scene.

The first task of recognizing the hand puppets is accomplished using contour modeling. Neural network based approaches are also attempted. Then, the object classification and salient features are packaged into standardized datagrams and passed over a network socket to the graphics engine. The graphics engine uses the object's classification as well as other features such as rotation to generate a 3D model. The user can then apply virtual manipulations to the object by altering their hand puppet or by using a suite of tool images.

The system successfully differentiates between a suite of approximately seven objects and manipulators but is rather unsuccessful with the "finger" objects.

2.0 Introduction

Shadow Vision is intended to be a fun demonstration of a few fairly powerful contour recognition algorithms. Using only a small computer video camera, an overhead projector, and some creativity with the fingers, the user can make shadow pictures on a wall and have them realized in 3-D by the computer (see right). If the computer recognizes the image the user is trying to create, it generates a 3-D model of it. The user may then interact with the model by forming various tools out of shadows as well. The model can be scaled, rotated, skewed, or discarded with these virtual tools. When the user is finished editing it, the model can then be added to a scene.

Because the easiest shadows to make (and recognize) are classic animal shapes, Shadow Vision currently recognizes a bird, a crocodile, a dragon, and a cow.

3.0 Goals

The project broke roughly into three components: the computer vision components, the computer graphics components, and the communication between the two. Each of these components has their own objective.

- **Vision Goal:** Correctly classify a set of shapes represeting animals and various virtual manipulators. Because of the use of an overhead projector, the hand puppets can be treated as 2 dimensional black and white images. The classifcation system should be invariant to rotation and scale, but should be able to retrieve this information when necessary. The set of objects to be classified range from hand made shadow puppets represeting various animals to drawings on overheads representing manipulators.
- **Graphics Goal:** Offer an interactive shadow theater, capable of displaying and manipulating a suite of different animal models. The interface to these models should be rather simple, since it must be responsive to directives from the vision component. Thus, for the most part, interactivity could be limitted to a handful of functions per object.
- **Communications Layer Goals:** Standardize an interface between the graphics and vision components of the project. Paying special attention to this section was necessary in order to allow parallel development of the application as well as to produce a disrtibuted runtime environment.

The next section describes in detail how each of these components were implemented and how the associated objectives were acheived.

4.0 Implementation

4.1 Computer Vision: Recognizing the Shadow Puppets

The computer vision components breaks down into the following chain of stages. First, during the **acquisition** stage, the image is acquired from the camera and converted into a suitable form. Then, during **segmentation** the image is thresholded andteh pixels are marked as being part of the object or being part of the background. Also during this stage, a contour of the object is computed. Next, during **feature extraction**, a number of transformations are applied to the contour in order to extract salient features. Finally, these features are compared to training data during the **recognition** stage. Here, the features of the object are compared to the models known to the system, and if a match or sufficient quality is found, then it is reported. Together, these 4 stages acheive the goal of recognizing a hand puppet and returning the correct classification.

4.1.1 Acquisition

The first task for the computer vision program was to acquire and display the incoming video stream.We chose to capture the images to process using an infrared camera. Any video projection in the area seen by the camera will be invisible to the infrared camera. This has the advantage for us to be able to project other computer generated images on top or around the shadow without any interference with the computer vision processing.

We were not able to use any existing library as we were originally hoping to, as most of the existing code we had access to was targeted to color processing of images.

We also needed to make sure that the images were captured inYUV space. We chose YUV because all we need to process is the luma information, as the incoming infrared image is black and white.

In this process we found out two things:

- The only YUV encoding natively supported by the O2s is a 422 packing which alternates a U and V values for every acquired Y byte. Any other encoding will have to be converted in software and therefore slows down the acquisition.
- In some other cases we noticed that the O2 would automatically use RGB8 acquisition and therefore we were only touching the red pixel instead of the expected luma pixel. Although these are each one byte, the unpacked information was wrong: it did not correspond to the calculated luma value we compared with when calculating luma computationally from RGB.

4.1.2 Display

Another bit of a challenge was encountered in displaying the acquired YUV image in real time on the screen. A possibility was to use X and the *XCreateImage()* function. We wanted instead to use GL or OpenGL. We found out it is not possible to display a YUV image in GL in real time on the screen without the introduction of a delay. If forced to have a pixmode with a luma setting, GL does a conversion from YUV to RGB in software and this slows down the display. It can otherwise be forced to display the luma through the red channel of the graphics display. This works in real time but it is ugly to look at.

We then moved to OpenGL, in which it is possible to use *glDrawPixels()* with the GL_YCRCB_422_SGIX option, which displays the YUV image on the screen in real time.

Working with OpenGL rather than X, makes it easier to later on add a more elaborate graphical representation of the contour of the segmented portion of the image. One could use splines for example to represent graphically the contour of the segmented image or use OpenGL to compute histograms or do more elaboate image processing on the fly.

4.1.3 Segmentation

Once the image is acquired we need to flag the pixels of the projected shadow-hand. Given that the shadows are generated with an overhead projector, and that the resulting image has such a high contrast, thresholding is sufficient to get the hand's pixels. However all the area around the projected area is also dark. We therefore implemented a crop function which alowed us to limit the processing only in the chosen area of the incoming video image. In order to select an appropriate threshold value we considered a variety of techniques. One of them, which seemed appropriate was to use background subtraction and later on to calculated mean and standard deviation of the segemnted pixels of the foreground. These would provide us with an ideal value to use for thresholding. For time constraint reasons, we ended up selecting the lower and upper values of the threshold by hand: we simply use the mouse to click on different areas of shadow in the image and come up with an empirical estimate of the values to use.

Although the segmentation is obtained by rastering the entire full resolution video image, or the cropped area inside which the shadows are produced, the program works in real time and with no visible delay. Other segmentation techniques find first a seed object belonging to the foreground or target object and then use a grow-region or fill-algorithm to find the remanining pixels of the object. This is useful in cases in which computation time is important or in which it is important to find multiple non connected region within the image. The continuous raster approached worked well for us as we were segmenting only one object at a time and because it already worked in real time.

4.1.4 Feature Extraction

Once the image is segmented it is easy to calculate its centroid and bounding box, in parallel with the segmentation itself. For the bounding box we keep track of the min and max x and y coordinates of the flagged pixels. The centroid is given by the mean of x and y for all flagged pixels. We used these features as a basis for a first prototype of our system. With these we used the centroid to move around a CG object from a connecting program. The bounding box information was used to initially flap the wings of a CG bird.

We also computed the contour of the segmented object to use it later in the recongition phase of the program. The litterature [Ballard, Brown] suggests using a chain code algorithm to find the pixels at the boundary of the segmented region. However, given that we obtain the segmented object by doing a continuous raster of the incoming image, we also obtain the contour points "for free". They are those for which the segmenting algorithm changes its setting from labeled to unlabeled and viceversa. This is a bit like IN and OUT points of a CG scan algorithm.

The main difference between this technique for contour extraction and the chain code is that our contour points are not ordered. Also our methods has problems with adges which are perfectly aligned with the X-axis—which would have to be handled as a special case. Given that this case does not arise in practice for hands, we have not dealt with it.

4.1.5 Recognition

The intuitive notion of a shape is not easily captured by a formal definition that can be translated into a computer program. In order to recognize simple shapes the litterature usually refers to Hu's moment invariants. These are obtained first by calculating central moments, i.e. moments with respect to the centroid, and then by normalizing them by the area. Hu's moments are invariant for translation, rotation and scale. Our choice was not to use Hu's technique but to still use centralized moments are our feature set. We believe that it is pointless to perform the moment computation for all points inside the shape. The contour of the shape summarizes already all we need to know for simple shape recognition. Driven by this belief we implemented instead Mertzios' and Tsirikoloas' technique which used moments on a centralized contour. Although the points of our contour are not given in order it was possible for us to use this technique because the moment calculation is a series of products and sum which do not depend on order. This algorithm provides shape recognition which is also invariant for translation, rotation, and scale.

4.1.6 One-dimensional moments algorithm

For all contour points we first compute the euclidean distances from these points to the contour. This is our feature set. We then compute their 1-D moments up to the desired "resolution". The first 2 moments are:

$$m_1 = \frac{1}{N}\sum_{j=1}^{N} a_j$$

$$m_2 = \left[\frac{1}{N-1}\sum_{j=1}^{N} (a_j - m_1)^2\right]^{1/2}$$

For moments higher than 2 we use the formula:

$$m_k = \frac{1}{N}\sum_{j=1}^{N} \left(\frac{a_j - m_1}{m_2}\right)^k$$

Mertzios' and Tsirikoloas' provide us with a cost function to use to compare the stored moments of the object we want to recognize with the moments of the currenly analized object.

$$C(n) = \sum_{j=3}^{k} (m_j - p_{nj})^2$$

We then added a “train option” to our program so that it could be adapted to either record the moments of an object to recognize or to actually be in recognition mode.

The method has two free parameters: 1. the choice of the resolution to use i.e. how many moments to use to describe the shape of an object 2. the lower threshold to use to evaluate the number given by the cost function. This threshold needs to be adapted to the number of moments used (resolution).

4.1.7 Evaluation of the one-dimensional algorithm

The chosen object recognition techniques works reliably for a variety of shapes. However in order to get a 100% recognition rate we decided to produce shadows by cutting an outline of the shadow we wish to produce on paper and pasting it onto an overhead transparency. This was also very useful as neither of us turned out to be a good shadow theater puppet-master. By using this technique and by choosing shadows not too similar to one another we were able to obtain a 100% recognition rate for 7 shadows.

We also wanted to be able to recognize hands making numbers but the algorithm was not too good at discriminating a hand making one-two-three- and four symbols. To do this we used the information at the edge of the cropping region to tell us how many fingers were entering the recognition area.

The system is not good at discarting negative examples. We could spend some additional effort in classifying negative example into a new “unrecognized class”. However instead of doing this we decided to try and improve the overall shadow recognition with the use of a neural net.

4.1.8 Neural network algorithm

While the feature set returned from the moment calculations was rich, using a cost-based recognition function had a number of disadvantages.

First, for every object, it required a single set of ideal moments for which to compare. Not only was this extremely time consuming to generate, but it did not lend itself to certain objects which perhaps had two different sets of ideal moments. In other words, limiting the recognition space to single peak functions significantly cut down on the types of objects that could be recognized. Second, incorporating negative and likely false positive examples of objects was difficult. Third, adding additional features (such as axis of inertia characteristics) was not possible since determining how to standardize these features and properly weight them was difficult.

Neural networks offered a good solution to these problems. We could easily add features and objects to our system and let the network figure out how to distinguish them.

The implementation of the neural net based on some sample code from Carnegie Mello University (see Bibliography). It is a simple back propagating three layer network. The inputs currently correspond to the seven moments of the object as well as the length, angle, and endpoints of the axis of inertia. With only 13 inputs, the network could perform remarkably well. The output of the network was a binary encoding (using 5 nodes) of the classifcation number of the object. The training of the network was based on a set of varying parameters that were fine tuned to produce the best results. Some of these

parameters included the number of hidden nodes (in the second layer), number of epochs of training, and learning momentum.

The net was integrated into the application in a way that would facilitate easy training of the network. The user is presented with a set of objects for which he can supply training data. All the user needs to do is select which object he is currently displaying and then start making the shape. Also, the user can present false positive and negative examples by marking them to be classified as “unclassified”. Once the user feels he has collected enough data, he stops the collection and allows the network to begin training. The trained network is stored away and used from then on to classify shapes.

4.1.9 Additional feature extraction

Using the moment features extracted from the object turned out to be sufficient to classify the object. However, some of the shapes required having some additional features. One example is the rotation manipulator which can control the rotation of the last created object. Obviously, maintaining rotation information for this obejct would be very useful. Similarly, other objects called for a scale feature.

Scale and rotation features were extracted after the classification of the object using the same contour. From this contour, the axis of inertia was determined. This axis reflected the longest axis of the object which passed through the centroid. Once this axis was found, its slope was used as an angle to represent the rotation of the object. Also, its length could be used as a scaling feature.

4.2 Communication Between Graphics and Vision Components

Because of the computational demands of the recognition engine, it was decided to separate the engine from the actual model display code. The communication protocol between the recognition engine and the model display code is flexible. Because it uses the standard UNIX sockets interface, the client (the recognition engine) and the server (the display) can run on two different networked machines (or the same machine if *localhost* is specified as the server). At compile time, the client and the server agree on the size and composition of the datagram to be used. For instance, the datagram may be a structure that lists the type of object, its position and/or orientation, and its current actions. At run time, the latency between recognition and movement of the model is unnoticeable. The chief advantage of this strategy is that it allows for smooth thirty frames-per-second animation of the model by off-loading the non-interactive recognition engine to another machine.

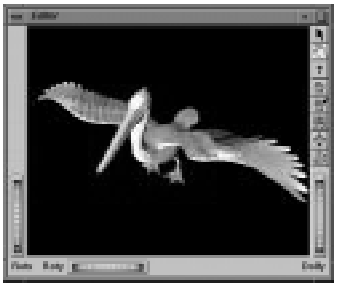
An additional advantage of this choice was that it allowed the code base to be easily split among the team members. Dealing with the Open Inventor models, coding the graphics and event callbacks, and implementing the networking code were separated from the more isolated problem of recognizing images from the camera input. Sockets made for cleaner code, faster performance, and fewer bugs.

4.3 Computer Graphics: Making the Virtual Theater

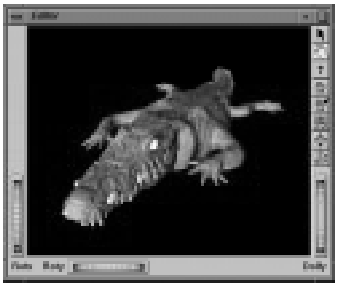
Once it is determined what type of shadow has been displayed for the camera, a 3-D model is generated and displayed. The Open Inventor API is used to support rudimentary display and editing of the models once they are displayed.

4.3.1 Open Inventor Models

Because of a painful lack of artistic creativity, most of the models used in the project were downloaded from third parties, edited to add animation access points, and are displayed using standard Open Inventor API viewers. The four models used (and consequently the four animal shapes recognized by the engine) are of a bird, a crocodile, a dragon, and a cow.

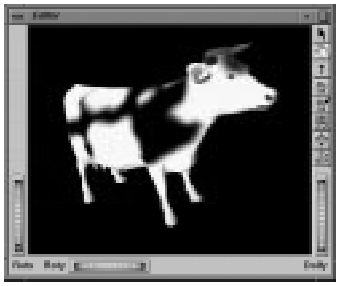


The bird (more accurately, a pelican) was converted from Alias|Wavefront to Open Inventor using *ObjToIv*, then edited by hand to separate the wings from the body. Giving credit to its makers, the original Alias|Wavefront model was obtained from the “Free 25 Models” section at *http://www.viewpoint.com*. Its wings are animated via an *action()* callback installed by the server. “Flapping” is recognized by a change in the length of the major axis of its bounding box.



The crocodile was also downloaded from Viewpoint and colored by hand. Its inherent features did not allow for easy animation, so it remains a static model.

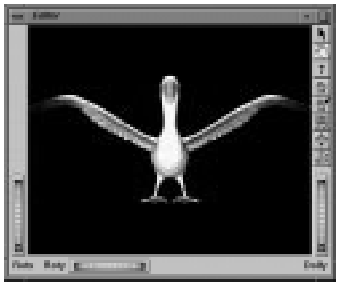
The dragon was—you guessed it—downloaded from Viewpoint and modified with a simple texture map and coloring.



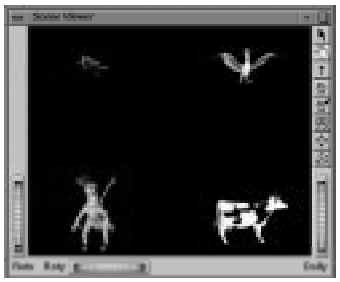
The cow was downloaded from the Avalon free public image archive (*http://avalon.viewpoint.com*) and texture-mapped using the GNU Image Manipulation Program.

The code for creating and displaying the models is about 40% boilerplate code for creating an *SoXtExaminerViewer* and about 60% multithreading and callback code. Because Open Inventor really seems to hate it when external threads mess with the display list, a separate timer callback manually refreshes the screen thirty times a second (usually the main Inventor loop will refresh the screen automatically when a node in the display list is altered). The main callback responds to commands passed in from a *Server* object. This function manages the creation and actions of models, as well as the action of tools.

4.3.2 User Interface



The user interface consists of two windows (instances of *SoXtExaminerViewer*). The first window is an object space editor. Shadows that are recognized by the engine are transformed into live 3-D models in the editor. Here the user can watch the effects of tool transformations in real-time.



The second window is a scene viewer. After the user is done editing the models in the editor with the virtual shadow manipulators, they can be inserted into a scene with other objects.

4.3.3 Mouse Test

While the main client is the shadow recognition engine, a second client was coded for testing purposes. To isolate problems with the camera and the contour recognition code from problems with the networking and display interface, this mouse-driven client simulates the operation of the main recognition engine. An “input pad” is provided with which the user can drag out bounding boxes. Several buttons line the side of the client to simulate operations such as changing tools, creating new shadows, or committing shadows to the scene viewer. While not at all very polished, the mouse client nevertheless helped enormously in debugging the early stages of coding.

5.0 Individual Contributions

5.1 Mark Huang

I was responsible for the front-end graphics, the networking code, and the modeling. It was pretty fun. I learned all about the internal limitations of the MIPS Pro C++ compiler, several known but unfixed bugs in the Open Inventor 2.1 API, and how bad coding standards get adopted.

5.2 Shishir Mehrotra

I worked on the initial Inventor models, implemented the neural network recognition, and worked on feature extraction.

5.3 Flavia Sparacino

I worked on the computer vision code for image grab, OpenGL display and processing. I also implemented the one dimensional object recognition algorithm.

6.0 Acknowledgements

We would like to thank Charles Lee, Thomas Minka, and Professor Seth Teller for helpful comments and assistance throughout the duration of the project.

7.0 Bibliography

Ballard, D. H. & Brown, C. M. *Computer Vision*. Prentice-Hall, 1982.

Hu, M. *Visual pattern recognition by moment invariants*. IRE Trans. Inform. Theory, 8:179-187, 1962.

Mertzios, B. G. & Tsirikolias, K. D. *Fast shape discrimination using one-dimensional moments*. IEEE CH2977-7/91/0000-2473.

Mina, A. *Ombres Chinoises*. Editions De Vecchi, 1997.

Shufelt, J. *A Neural Network Face Recognition Assignment.* Available at http://www.cs.cmu.edu:80/afs/cs.cmu.edu/user/avrim/www/ML94/face_homework.html.

Sockets Tutorial. Available at http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html.

Wernecke, J. *The Inventor Mentor: Programming Object-Oriented 3D Graphics With Open Inventor, Release 2*. Addison-Wesley, 1994.