# Visual Routines for Spatial Cognition on a Mobile Robot

Neil S. Halelamien
Advisor: Professor David S. Touretzky
May 13, 2004

**Abstract**

In his studies on visual cognition, Ullman (1984) proposed that high-level vision makes use of reusable elemental operators which may be sequentially composed into a "visual routine" specialized for a particular visual task. Seeking to explore Ullman's ideas in the domain of robotics, we have created a framework for robot vision programming which uses reusable operators to parse the visual world. These visual operators act on and transform between two different representations of the world: iconic (pixel-based) and symbolic (shape-based). The framework provides a powerful set of primitives for the construction of visual behaviors, while also providing a GUI which allows for the run-time visualization of intermediate representations organized into a derivation tree. The framework has been implemented in simulation and on a Sony AIBO robot, to perform tasks such as tic-tac-toe board parsing and character recognition. Current research is focused on using camera information to generate a persistent top-down world representation, allowing for more elaborate spatial behaviors.

CONTENTS

**I Introduction** 1

**II Previous work** 1
- II-A Cognitive research into visual routines . . . 1
- II-B Computer implementations of visual routines . . . 2
- II-C Other Vision Frameworks . . . 3

**III Dual representation: iconic and symbolic** 3
- III-A The Iconic Space . . . 3
  - III-A.1 Iconic Operators . . . 4
  - III-A.2 Iconic → Symbolic Operators: Extraction . . . 5
- III-B The Symbolic Space . . . 5
  - III-B.1 Symbolic Operators . . . 5
  - III-B.2 Symbolic → Iconic Operators: Rendering . . . 5

**IV Camera Space and World Space** 6

**V The Visual Routines GUI** 7

**VI Walkthrough of the Tic-Tac-Toe Behavior** 7
- VI-A Iconic → Symbolic: Exracting the Lines and Game Pieces . . . 8
  - VI-A.1 Line Extraction . . . 8
  - VI-A.2 Game Piece (Ellipse) Extraction . . . 10
- VI-B Symbolic → Iconic: Labeling the Board Regions and Specifying a Move . . . 11

VI-B.1 Establish the Pairs of Parallel Lines . . . . . . . . . . . . . . . 11
VI-B.2 Determine Orientation of Line Pairs . . . . . . . . . . . . . . . . 11
VI-B.3 Determine Relative Line Location . . . . . . . . . . . . . . . . . 11
VI-B.4 Determine Borders of Gameboard . . . . . . . . . . . . . . . . . 11
VI-B.5 Find Board Regions . . . . . . . . . . . . . . . . . . . . . . . . . 12
VI-B.6 Determine Board Layout and Specify Desired Move . . . . . . 12

**VII Discussion** 13

**VIII Conclusion** 14

**References** 14

**Index Terms**

Visual routines, spatial cognition, robotics, robot vision, Tekkotsu

# Visual Routines for Spatial Cognition on a Mobile Robot

## I. INTRODUCTION

GENERALLY speaking, programming vision for robots is a difficult task. Often a programmer who wishes to create a robot behavior which relies on vision must begin at a very low level, due to the limited reusability of previously implemented behaviors. It is also often difficult to debug problems in robot vision, as there typically isn't a mechanism available to view the results of intermediate visual computation.

The goal of this project was to make the task of programming robot vision easier, more straightforward, and more effective, by providing a vision programming framework inspired by the notion of "visual routines" from cognitive science. Our framework is implemented within the Tekkotsu framework for AIBO development [1][2], further increasing the ease of programming. Within this framework there are two types of representations: an iconic (pixel-based) representation and a symbolic (shape-based representation). The framework provides operators which take data in one of these representations as input, and similarly generate data in one of these representations as output. The general nature of these operators allows for them to be reused and composed in virtually any way the programmer needs them.

There are additionally two types of coordinate representations in which both iconic and symbolic representations can exist: a first-person "camera space" and a top-down "world space."

Finally, the framework provides a GUI (Figure 1 on the following page) which allows a user at a desktop machine to view the intermediate iconic and symbolic representations generated during the course of a behavior. These representations on the AIBO robot are serialized on request over the network and are viewable as a derivation tree in the GUI. This allows for a better understanding of the operation of a visual behavior, and has in practice proven to be a useful debugging tool.

The iconic and symbolic representations serve two important purposes. First, they are used by a programmer to parse a visual scene in an effective manner. Second, they are a means to convey relevant information over the network to the GUI, allowing a user to better understand the functioning and internal representation of the robot.

## II. PREVIOUS WORK

### A. Cognitive research into visual routines

Ullman [3] proposed that the human visual system's task of perceiving shape properties and spatial relations from visual information is split into two successive stages: an early "bottom-up" stage during which base representations are generated from the visual input, and a later "top-down" stage during which "visual routines" extract the desired information from the base representations. In humans, the base representations generated during the bottom-up stage correspond to retinotopic maps (more than 15 of which exist in the cortex) for properties like color, edge orientation, speed of motion, and direction of motion [4]. These base representations rely on fixed operations performed uniformly over the entire field of visual input, and do not make use of object-specific knowledge or other higher-level information.

The visual routines used to extract spatial information from the base representations are patterns of activation of elementary visual operators. Visual routines are not applied uniformly over the

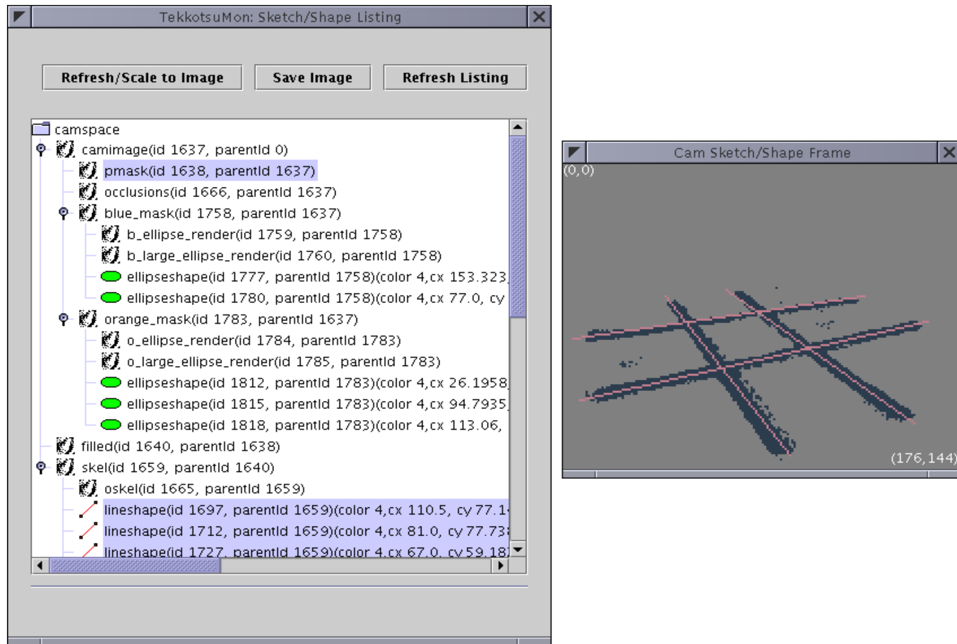


Fig. 1. A screenshot of the Visual Routines GUI. The listing organized into a derivation tree is in the left window, the Viewer is in the right window.

entire visual field, but are rather only applied to objects or areas specified by the routines. Ullman lists the following as examples of elementary operators: shifting the processing focus, indexing a salient item for further processing, spreading activation over an area delimited by boundaries, tracing boundaries, and marking a location or object for future reference. When combined into visual routines, these elementary operators can be used to perform relatively sophisticated spatial tasks such as counting the number of objects satisfying a certain property, or recognizing a complex shape.

Roelfsema et al. [5] conducted both psychophysical and neurophysiological studies to investigate Ullman's hypothesis. Their studies focused on curve tracing, one of the elementary operators which Ullman originally proposed. Their research gave evidence for an incremental spreading of activation along a curve, similar to that hypothesized by Ullman, represented in the primary visual cortex of monkeys. Such evidence increases the plausibility of visual routines as an explanation for visual cognition.

### B. *Computer implementations of visual routines*

Other uses of the term "visual routines" in AI and robotics, inspired by Ullman's work, have influenced our own. Agre and Chapman's [6] project, PENGI, implemented Ullman's concept of visual routines as part of an automated system for playing a real-time two-dimensional video game. Their work also introduced the notion of a "deictic representation": a means for representing entities in the environment in terms of their relations to the perceiver. This notion influenced our design of the entities and interactions within the symbolic space. Rao and Ballard's [7] active vision architecture made use of dual memory systems — one indexed by image coordinates, the other indexed by object identities. This dual memory system is loosely analogous to our framework's dual-coding representations. Forbus et al.'s [8] work, concerned with qualitative

spatial reasoning on maps, provided the basis for many of the iconic operators present in our framework.

### C. Other Vision Frameworks

## III. Dual representation: iconic and symbolic

Within the framework, there are two types of representations: iconic (pixel-based) and symbolic (shape-based). The iconic representation is termed a Sketch and the symbolic representation is termed a Shape. Sketches and Shapes are collectively referred to as VisualObjects.

This dual representation approach is partially inspired by notions from cognitive science. One such notion is Paivio's "dual coding theory" of mental representations [9], which hypothesizes two complementary cognitive subsystems, one of which processes verbal representations (termed "logogens") while the other processes non-verbal representations (termed "imagens"). The verbal subsystem is analogous to our symbolic representation, while the non-verbal subsystem is analogous to our iconic representation.

Sketches and Shapes have a number of operations in common, which are declared as member functions of the abstract VisualObject class:

**getId()** Returns the unique numeric ID of the VisualObject. This ID is used as a unique identifier when communicating over the network.

**getParentId()** Returns the ID of the most recent Viewable ancestor of this VisualObject. The ancestor is a VisualObject which was used to create the current VisualObject. This information is used to draw the derivation tree in the Visual Routines GUI.

**isViewable()** Returns true if the current VisualObject has been set to be Viewable over the network.

**makeViewable()** Makes the current VisualObject Viewable over the network.

**getViewableId()** Returns the ID of the nearest Viewable object in the derivation tree. If the current object is Viewable it returns the current VisualObject's ID; if not, it returns the result of getParentId().

**getNewId()** A static function to return a new unique ID number.

**getName()** Returns a string containing the current VisualObject's name, to be displayed in the Visual Routines GUI.

### A. The Iconic Space

The VisualObjects of the Iconic space are called Sketches. A Sketch is a two dimensional array of pixels of some type (bool, int, vector, etc.), implemented as a templated class in C++. `Sketch<bool>` is used to represent image masks and renderings of symbolic objects, while `Sketch<int>` is used for color-segmented images and labeled regions. `Sketch<float>` is used to represent distance maps.

For purposes of performance and visualization, a Sketch is in actuality a "smart reference" to an internal representation termed a SketchData. These SketchData objects are kept in a SketchSpace, a collection of several type-specific SketchPools (currently there are SketchPools for bools, ints, and floats). Each of these SketchData objects has a reference count, which indicates how many Sketches are currently pointing to it. When a new Sketch is created the appropriate SketchPool is searched for a SketchData with a reference count of 0, and the unused SketchData is associated with the Sketch; if no free SketchData is available, a new one is created. When the Sketch is destroyed or passes out of scope the reference count of the SketchData is decreased, but the SketchData itself is retained in memory for future use. This caching of SketchData objects

improves performance, as memory for the large SketchData objects (a 176x144 Sketch<int> is about 100 Kilobytes) does not have to be repeatedly allocated and deallocated.

*1) Iconic Operators:* The framework provides a variety of standard image processing functions as Sketch operations, in addition to those inherited from VisualObject. These operations take one or more Sketches and zero or more parameters as input. As output, they return a Sketch, which may in turn be used as input to other iconic operations.

*a) Many of these iconic operators are standard arithmetic and logical operators, defined using C++ operator overloading. The following operators are defined as follows:*

**operator=** Sets the values in the current Sketch equal to those in the given Sketch, or sets all of the values in the current Sketch equal to the specified value

**operator*=**
**operator/=**
**operator+=**
**operator-=** Sets the values in the current Sketch equal to the result of the pixel-wise application of the arithmetic operator with the given Sketch, or the specified value

**operator***
**operator/**
**operator+**
**operator-** Returns the result of applying the arithmetic operator to two specified Sketches in a pixel-wise fashion

**operator==**
**operator!=**
**operator**$<$
**operator**$>$
**operator**$<=$
**operator**$>=$
**operator&&**
**operator**$||$ Returns the result of applying the logical operator to two specified Sketches in a pixel-wise fashion, or the result of applying the operator to each Sketch pixel and a specified value

*b) The following operators are defined as separate named functions:*

**bdist(dest,obst,maxdist)** Calculates the bounded distance from each pixel in the image to the closest active pixel in dest, using the wavefront algorithm. Obstacles are indicated by true values in pixels of obst.

**edist(dest)** Calculates the Euclidean distance from each pixel in the image to the closest true pixel in dest, using a linear-time algorithm. (Actually calculates manhattan distance at the moment)

**labelcc(fg,connectivity)** Returns a connected-components labeling of the foreground fg. Each different foreground region will contain a different number.

**neighborSum(im,connectivity)** For each pixel, calculate the sum of its neighbors using the specified connectivity. The connectivity parameter is either 4 or 8, corresponding to 4-way (i.e. up-down-left-right) or 8-way (i.e. up-down-left-right and diagonals) connectivity.

**fillin(im,iter,min_thresh,max_thresh)** Performs morphological fill-in on the input Sketch.

**edge(im)** Performs simple edge detection on the specified image.

**horsym(im)** Returns non-zero values along points of horizontal symmetry, with each of these values equal to the distance to the symmetric points.

**versym(im)** Same as horsym, but with vertical symmetry.

**skel(im)** Returns a skeleton of im, based on combination of results of horsym and versym.

**seedfill(borders,seedx,seedy)** Fills in image starting at (seedx,seedy), with borders obstructing.

*2) Iconic → Symbolic Operators: Extraction:* Several of the symbolic shape classes have functions which take as input a Sketch, and produce as output one (or more) symbolic shapes of the particular class. These functions act as a means to convert from the Iconic space to the Symbolic space. For more information on the types of Shapes, see the section below.

**LineShape::extractLine()** This function takes as input a `Sketch<bool>` containing a skeletonized image, such as that received from applying the skel() operator to a color mask. It returns a LineShape representing the most prominent line, extracted by using moment statistics [10] to determine the overall orientation for the largest skeleton region. The use of moment-based techniques offers a significant performance advantage over using the standard Hough transform. This operator is typically used in conjunction with the clearLine() operator to remove the pixels corresponding to the extracted line, so that a new line may be extracted.

**EllipseShape::extractEllipses()** This function takes as input a `Sketch<bool>` containing the candidate pixels for ellipses, such as that received from applying a color mask to a segmented image. It returns a vector of EllipseShapes within certain area, hollowness, and axis ratio thresholds. The EllipseShape parameters are detected using moment statistics [10].

### *B. The Symbolic Space*

The VisualObjects of the Symbolic space are called Shapes. The classes within the Symbolic space have been developed in collaboration with Jordan Wales.

There are a number of different Shape types:

*a) PointShape:* This is used to represent a single (x,y) point, with each coordinate stored as a floating-point variable.

*b) LineShape:* This is used to represent a line, which may be defined by a pair of points, a point and an orientation, or using r-theta notation.

*c) EllipseShape:* This is used to represent an ellipse, consisting of a centroid, a semimajor axis length, a semiminor axis length, and a principal axis orientation.

*d) AgentShape:* This is used to represent the location and orientation of an agent, typically the AIBO itself.

*e) GlyphShape:* This representation, still under development, is used to represent simple characters and shapes, generally referred to as glyphs (Figure 2 on the next page). Recognition makes use of moment invariants [11], region descriptors which are invariant to translation, scaling, and rotation.

*1) Symbolic Operators:* A number of operators are common to all Shapes, and are defined as member functions of the abstract Shape class (in addition to those inherited from VisualObject):

**centroidX()** Returns the X coordinate of the Shape's centroid.

**centroidY()** Returns the Y coordinate of the Shape's centroid.

**isLeftOf()**

**isRightOf()**

**isAbove()**

**isBelow()** Returns true if the current Shape is in a particular position relative to the Shape argument.

*2) Symbolic → Iconic Operators: Rendering:* Every symbolic shape has a render() function, which produces an iconic representation of the shape. Renderings are cached for efficiency purposes, and are flagged for re-rendering when relevant shape parameters are altered.

Fig. 2. Samples of glyphs

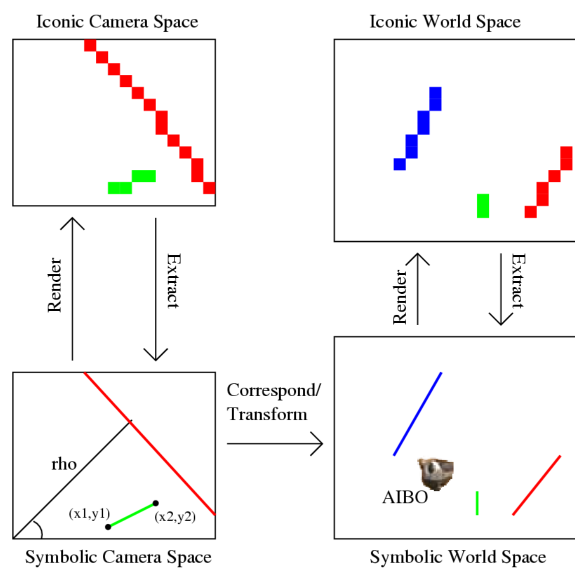


Fig. 3. Interactions among iconic and symbolic representations in camera and world spaces

## IV. Camera Space and World Space

Different visual computations are best performed in different spaces or coordinate frames. The visual routines framework currently offers two different spaces: a camera space and a world space. The camera space corresponds to the coordinate frame of the pixels from the camera, while the world space corresponds to a top-down coordinate frame with units roughly equivalent to millimeter distances. Additionally, objects may persist between camera frames. Iconic and symbolic representations may be created in both the camera space and the world space.

Computation for visual recognition, such as line and ellipse extraction, occurs mostly in the camera space. This space is from the AIBO's point of view, and is closest to the actual image data coming from the camera. VisualObjects in the camera space only persist for the current frame.

Other computations, particularly those involving navigation and spatial reasoning, are best performed in the world space. Since Shape information persists between frames, the world

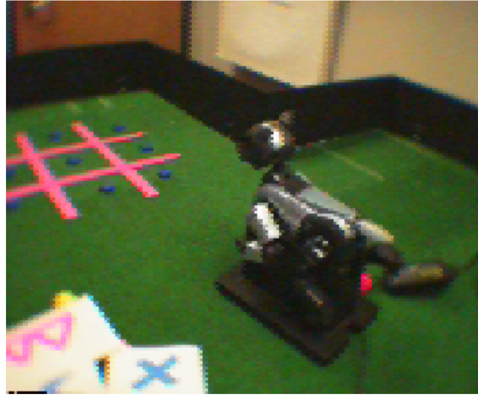

Fig. 4. AIBO stares at the tic-tac-toe board

space allows a programmer to implement behaviors which reason about VisualObjects outside of the AIBOs current view. Although the world space is still under development (in collaboration with Jordan Wales), it will eventually allow programmers to perform tasks like wavefront-based navigation around obstacles with ease.

To generate the world space, Shapes from the camera space are projected onto the world space by making use of a ground plane assumption. These estimated Shape locations are then corresponded with Shapes currently in the world space, removing, adding, and moving Shapes as needed.

## V. The Visual Routines GUI

The visual routines framework includes a GUI (Figure 1 on page 2), implemented in Java, which may be used to display information about Sketches and Shapes. This GUI runs on a desktop computer and communicates over the network to retrieve information about visual objects, which is made available to the user.

Using the GUI, a user is able to click on “Refresh Listing” and retrieve a listing of all the Sketches and Shapes which are currently specified as visible. This listing is displayed as a derivation tree in the GUI, with derivation indicating parentage. For example, a LineShape extracted from a skeleton Sketch is shown as a child of that skeleton Sketch, which is in turn shown as the child of a colormask Sketch. Most of this parentage information is transmitted directly from the input of a visual operator to its outputs. When an object has multiple parents, the first parent is indicated.

Once this listing has been retrieved, a user may then click on individual visual objects, or select multiple visual objects. In the case of clicked Shapes an outline is immediately displayed in the view window; with Sketches, a request for image data is sent over the network and the Sketch image is displayed when it is received.

A “Refresh/Scale to Image” button refreshes the current view and, if the current view is in the world space, rescales to match the extrema of the viewed objects.

The GUI also contains a “Save Image” button, to allow the user to save the current view to an image file.

## VI. Walkthrough of the Tic-Tac-Toe Behavior

We next step through an example behavior implemented using the Visual Routines framework, one which parses a tic-tac-toe game board (Figures 4 and 5 on the following page) when a

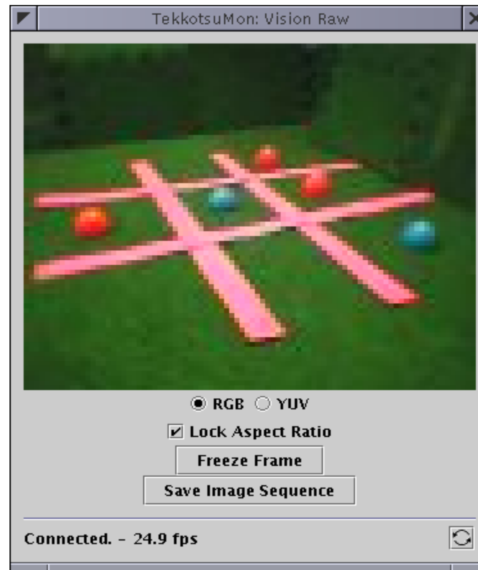


Fig. 5. The tic-tac-toe board from the AIBO's perspective

Fig. 6. The segmented camera image

button is pushed. The robot indicates its next desired move via the Visual Routines GUI. This demonstration task requires the use of many of the operators described above. All operations for this behavior occur in the camera space.

Note that the example code below leaves out some extraneous lines, such as calls to the makeViewable() operator of VisualObjects, and debug messages.

### *A. Iconic → Symbolic: Exracting the Lines and Game Pieces*

Our first step is to obtain the segmented camera image (Figure 6):

```
Sketch<int> cam = sketchFromRle();
```

*1) Line Extraction:* We then pick out all the pink pixels of the segmented color image to generate a pink mask Sketch (Figure 7 on the facing page), then perform a morphological fill-in to remove noise. We also create a mask of all the potential line occluders: the blue and orange game pieces.

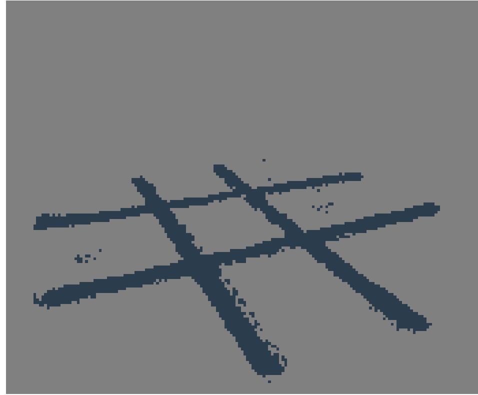

Fig. 7. The mask of pink pixels

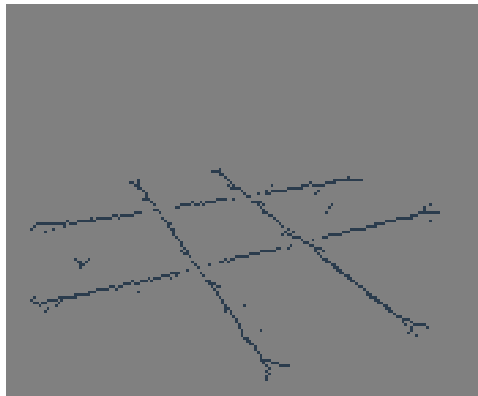

Fig. 8. The result of skeletonizing the pink pixels

```
// cam.getSpace() creates pmask in the camera space, rather than the world space
Sketch<bool> pmask (cam.getSpace(), "pmask");
pmask = (cam == SEG_PINK);
Sketch<bool> filled (cam.getSpace(), "filled");
filled = pmask || (cam == SEG_UNCLASSIFIED
     && visops::fillin(pmask, 1, 2, 8));
Sketch<bool> occlusions (cam.getSpace(), "occlusions", cam.getViewableId());
occlusions = (cam==SEG_BLUE || cam==SEG_ORANGE);
```

We next generate a skeleton of the filled-in mask.

```
Sketch<bool> skel (cam.getSpace(), "skel");
        skel = visops::skel(filled);
        Sketch<bool> oskel (cam.getSpace(), "oskel");
        oskel = skel; // saving for future reference
```

We then extract the four most prominent lines from the skeleton Sketch. This is the first shift from the iconic to the symbolic space. These extracted lines are represented as LineShapes. All of the LineShapes are also added to the `camshapes` ShapeSpace, for visualization purposes.

```
vector<LineShape> lines(4);
for(int i = 0; i < 4; i++) {
```

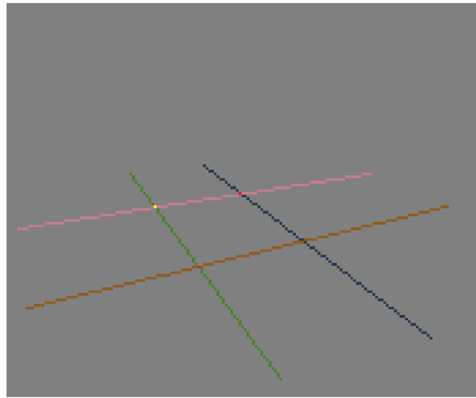

Fig. 9. A rendered Sketch of all of the extracted lines

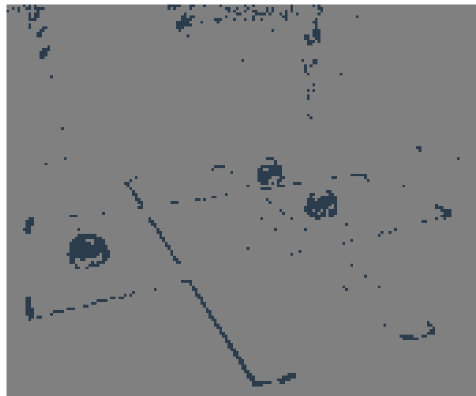

Fig. 10. A Sketch showing all of the orange pixels

```
lines[i] = LineShape::extractLine(skel, filled||occlusions);
lines[i].clearLine(skel);
camshapes.appendShape(&lines[i]);
}
```

*2) Game Piece (Ellipse) Extraction:* We then wish to begin extracting the Easter Egg game pieces, which are roughly elliptical in the image. We first generate masks of the colors blue and orange, the two possible colors for game pieces.

```
Sketch<bool> blue_mask (cam == SEG_BLUE, "blue_mask");
Sketch<bool> orange_mask (cam == SEG_ORANGE, "orange_mask");
```

Now we actually extract the EllipseShapes. We also render the centroids of the extracted EllipseShapes to a separate Sketch, to later use when determining gamepiece intersection with extracted game regions. All of the EllipseShapes are also added to the `camshapes` ShapeSpace, for visualization purposes.

```
Sketch<bool> b_ellipse_render(camspace, "b_ellipse_render",
     blue_mask.getViewableId());
```

```
b_ellipse_render = 0;
vector<EllipseShape> blue_ellipses = EllipseShape::extractEllipses(blue_mask);
typedef vector<EllipseShape>::iterator IT;
for (IT i = blue_ellipses.begin(); i != blue_ellipses.end(); ++i) {
     // use (x,y) subscripts to set ellipse centers in b_ellipse_render
     b_ellipse_render((int)(i->centroidX()),(int)(i->centroidY())) = true;
     camshapes.appendShape(&(*i));
}
Sketch<bool> o_ellipse_render(camspace, "o_ellipse_render",
     orange_mask.getViewableId());
o_ellipse_render = 0;
vector<EllipseShape> orange_ellipses = EllipseShape::extractEllipses(orange_mask);
typedef vector<EllipseShape>::iterator IT;
for (IT i = orange_ellipses.begin(); i != orange_ellipses.end(); ++i) {
     o_ellipse_render((int)(i->centroidX()),(int)(i->centroidY())) = true;
     camshapes.appendShape(&(*i));
}
```

### *B. Symbolic → Iconic: Labeling the Board Regions and Specifying a Move*

Now that all of the lines and game pieces are extracted, we wish to actually determine the bounds of the nine board regions and where the next move should be. This is best done with a symbolic set of representations.

*1) Establish the Pairs of Parallel Lines:* A valid tic-tac-toe board consists of two pairs of parallel lines. To find which ones are parallel to each other, we select one of the lines, then calculate the difference in orientation between it and each of the other lines. The line which is most similar in orientation to the selected line is marked as parallel to it, and the other two lines are marked as parallel to each other. The first pair of lines is referenced as `La0` and `La1`, while the second pair of lines is referenced as `Lb0` and `Lb1`.

*2) Determine Orientation of Line Pairs:* We next determine which of the line pairs is more horizontal than the other, and conversely, which is more vertical than the other. We begin by calculating the average orientation of each of the line pairs. The horizontal pair of lines is referenced as `Lh0` and `Lh1`, while the vertical pair of lines is referenced as `Lv0` and `Lv1`.

*3) Determine Relative Line Location:* We next determine where each of the lines is relative to the other. Specifically, we wish to find which of the two horizontal lines is upper and which is lower, and which of the two vertical lines is left and which is right.

```
// Determine up-down and assign aliases.
bool Lh0_upper = (Lh0.calcMid()).second < (Lh1.calcMid()).second;
LineShape& Lhu = Lh0_upper ? Lh0 : Lh1;
LineShape& Lhd = Lh0_upper ? Lh1 : Lh0;

// Determine left-right and assign aliases.
bool Lv0_left = (Lv0.calcMid()).first < (Lv1.calcMid()).first;
LineShape& Lvl = Lv0_left ? Lv0 : Lv1;
LineShape& Lvr = Lv0_left ? Lv1 : Lv0;
```

*4) Determine Borders of Gameboard:* We now wish to find the borders of the tic-tac-toe board. Game pieces within these borders are considered within play, whereas game pieces outside the borders — such as piles of extra pieces and erroneously detected pieces — are ignored. Guide points are selected from the extrema of the four detected lines, and border lines are drawn through these guide points parallel to the nearby board line.

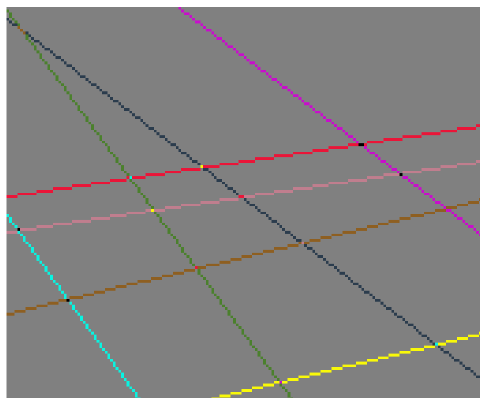

Fig. 11. A Sketch showing the summed renderings of the border lines and the board lines with endpoints turned off (and thus rendered to edge of image)

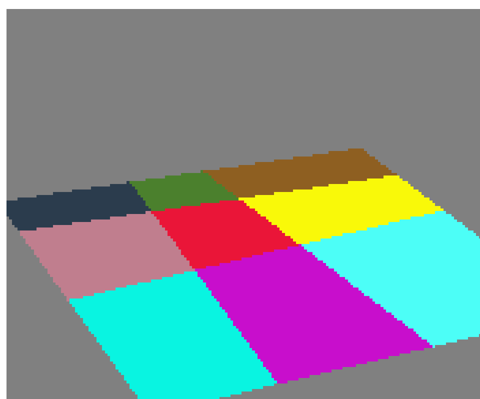

Fig. 12. A Sketch showing the distinctly labeled board regions

*5) Find Board Regions:* We now wish to label each of the board regions, giving each board region a unique number. We turn off the endpoints of the board lines, so that their renderings extend to the bounds of the image. The result of rendering these lines along with the border lines is shown in Figure 11.

```
for(cur_line_i = 0; cur_line_i < NUM_LINES_C; cur_line_i++) {
    lines[cur_line_i].offEnd1();
    lines[cur_line_i].offEnd2();
}
```

We then choose "seed pixels," found by averaging the position of each extreme board line endpoint with that midpoint of the nearby perpendicular board line. These seed pixels are used with the seedfill operator to fill in half-planes (one for each board line) formed by composing each individual board line rendering with the rendering of the borders. This occurs in the iconic space. These half-planes are multiplied by unique powers of two and added together to form a Sketch containing the distinctly labeled regions (Figure 12).

*6) Determine Board Layout and Specify Desired Move:* To determine the board layout, the Sketch containing the region labels is logically ANDed with the previously rendered gamepiece centroids. A running count is made of the number of pieces of each color.

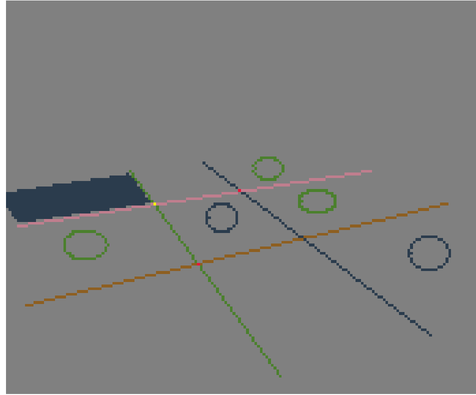

Fig. 13. A Sketch showing the desired move region, along with the detected board lines

```
bool blue_ttt[9]; int blue_count = 0;
bool orange_ttt[9]; int orange_count = 0;
for (int i = 0; i < 9; i++) {
     blue_ttt[i] = (ttt_regions[i] && b_ellipse_render).max();
     blue_count += blue_ttt[i];
     orange_ttt[i] = (ttt_regions[i] && o_ellipse_render).max();
     orange_count += orange_ttt[i];
}
```

If there are more orange pieces on the board than there are blue pieces, it is the AIBO's turn to move. At present a simplistic algorithm is used, where one of the unoccupied labeled regions is randomly chosen. The selected region is indicated in a Sketch (Figure 13).

## VII. Discussion

Compared to conventional robot vision programming techniques, visual behaviors programmed using the framework are easier to understand and visualize. The dual iconic and symbolic representations are a natural way to represent visual information, and the provided operators are an intuitive way to generate new representations from old ones.

Additionally, the Visual Routines GUI allows for greater intuition into the execution of a particular visual behavior, serving not only as a useful debugging tool, but also a means of communicating visual information between the robot and the human user. This has showed its value in practice, with the GUI helping greatly in creating, debugging, and understanding the tic-tac-toe example behavior.

### *Future work*

The full implementation of GlyphShapes in the framework will expand the power of the framework, for example allowing a programmer to create a behavior where the AIBO performs a particular action when a particular shape is viewed. This may include a mechanism for on-line learning of glyphs.

Additional work needs to be done to improve the framework's performance, to allow for the AIBO to smoothly operate in real-time. Many tasks a robot performs require a low latency between a perception and an action, making improved performance crucial.

Currently the ground plane projection is hampered by the lack of a good kinematics model to estimate the camera pose. Having such a model would greatly improve accuracy of the projection.

In the near future the Visual Routines framework will be integrated into the larger Tekkotsu framework for AIBO development [1][2], currently in use by a number of different universities.

## VIII. Conclusion

We have shown how a framework inspired by the notion of visual routines, using a dual-coding iconic/symbolic representation, can make robot vision programming easier and more intuitive. By making the Visual Routines framework available to vision programmers, we hope to greatly facilitate the creation of vision-based behaviors on mobile robots.

## Acknowledgements

This research was supported in part by an NSF REU supplement award to IIS-9978403, and by a grant from the Sony Corporation. Special thanks to Professor David S. Touretzky for his invaluable advising, Ethan Tira-Thompson for the Tekkotsu AIBO development framework, and Jordan Wales for his work on the symbolic representations and the world space.

## References

[1] E. J. Tira-Thompson, "Tekkotsu: A rapid development framework for robotics," Master's thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2004.

[2] (2004) Tekkotsu website. [Online]. Available: http://www.tekkotsu.org/

[3] S. Ullman, "Visual routines," *Cognition*, 1984.

[4] D. Chapman, *Vision, Instruction, and Action*. MA: MIT Press, 1991.

[5] P. R. Roelfsema, V. A. F. Lamme, and H. Spekreijse, "The implementation of visual routines," *Vision Research*, vol. 40, pp. 1385 – 1411, 2000.

[6] P. Agre and D. Chapman, "Pengi: An implementation of a theory of activity," in *Proceedings of AAAI-87*, 1987, pp. 268 – 272.

[7] R. P. N. Rao and D. H. Ballard, "An active vision architecture based on iconic representations," *Artificial Intelligence*, pp. 461 – 505, 1995.

[8] K. D. Forbus, J. V. Mahoney, and K. Dill, "How qualitative spatial reasoning can improve strategy game AIs," in *15th International Workshop on Qualitative Reasoning*, 2001.

[9] A. Paivio, *Mental Representations: A Dual-Coding Approach*. New York: Oxford University Press, 1986.

[10] R. J. Prokop and A. P. Reeves, "A survey of moment-based techniques for unoccluded object representation and recognition," *CVGIP: Graphical Models and Image Processing*, vol. 54, pp. 438 – 460, 1992.

[11] M. K. Hu, "Visual pattern recognition by moment invariants," *IEEE Transactions on Information Theory*, vol. 8, pp. 179 – 187, 1962.