Efficient axis and circle detection through Hough transformation

Karissa E. Hobert
University of Northern Iowa

Efficient Axis and Circle Detection Through Hough Transformation

Karissa E. Hobert

Presidential Scholar Senior Project
University of Northern Iowa

May 1995

Abstract

Computer vision and human vision are quite different. While a person can look at an object and easily recognize it, a computer is does not understand the the objects it is seeing. Finding axes and circles in an image is an intermediate step in designing a machine with the ability to recognize shapes and objects. Through the utilization of contour and smoothed local symmetry information, the main axes of an image are determined and may be used to facilitate shape and object analysis. A Hough transform algorithm detects the most popular lines from a set of candidate axis points. This conversion process also detects the optimal circles in an image. Through the use of Hough transform algorithms, these axes and circles are designated as being the best suited for the computer to recognize what it is seeing. These extensions bridge part of a gap between a collection of outlines and symmetric points to shape and eventual object recognition[1].

[1]Primary research Summer 1994 as REU Fellow supported through NSF at University of Iowa under Dr. Margaret Fleck. Implementation built in conjunction with the Vision Course Code written in Lisp developed by the Computer Vision and Simulation group at the University of Iowa. Vision Course Code available from Dr. Margaret M. Fleck, University of Iowa

1 Shape Analysis and Object Recognition

Consider a simple stick figure. These drawings come in a multitude of shapes, sizes and colors. Even when drawn by a kindergartner, the obvious characteristics of the long torso with head and arms extending from the top and legs grounding the figure immediately spark recognition. What makes a figure's lines and circles a coherent collection of shapes? One reason is that exposure to this icon since childhood makes the figure recognizable. Even so, it is remarkable that a stick figure represents basic features of the human figure in such a simple manner. A more complete explanation of the recognition of a stick figure is that the representation of the joints and connections in proper places mimic a person's basic structure to create the likeness of a person. However, while the relation of a circle at the top of a vertical stick to someone's head start to become clear upon initial perusal, similar representations of other objects may not lend themselves to such easy interpretation.

Consider next the problem of distinguishing between the kindergartner's drawing of a bear and a person. The stick figure representations may be somewhat similar, but the bear will most likely have its characteristic ears and perhaps even a tail. Representation of characteristic regional information and recognition of axes and their intersections may present enough information to allow for rudimentary shape analysis and ultimately object recognition. Gaining insight about the shapes that make up an object, in this case a bear, aids in the recognition process. A reliable description about the lines, shapes and their intersections make sure the bears and people don't become mixed up. These descriptions are especially important if part of an object is occluded by another object, such as a bear peeking out from behind a tree.

"The ultimate aim in a large number of image processing applications is to extract important features from image data, from which a description, interpretation, or understanding of the scene can be provided by the machine."[4, p. 342] In the case of the kindergartner, some explanation on their part may be the deciding factor in figuring out whether the picture is of a person or a bear. Through careful observation, the child will eventually learn the distinguishing characteristics of a bear or a human figure. Similarly, through analysis of characteristic shapes, regional axes, and their intersections, a computer may recognize objects in two-dimensional images. Researchers pursue many avenues in the problem of finding and representing these basic characteristics. Both child and computer require instruction in order grasp the key characteristics and analyze their meaning.

The framework for a computer-based solution requires solving many nontrivial intermediate problems. An important step toward recognition of an object in a two-dimensional image involves finding its major axes. These axes are extremely important in figuring out what shapes are represented, coming to conclusions about how object shapes are joined together, and finding the general orientation of the objects the computer is attempting to recognize[2]. Once candidate axis points are determined, applying a Hough transform algorithm allows for efficient detection of major axes in the image. Application of similar Hough transforms identify circles, which aid in shape and region analysis of an image. These intermediate solutions provide stepping stones towards solving the larger shape and object recognition problem.

This study describes the basic building blocks of an existing vision system that uses contours and smoothed local symmetries (SLSs) to represent characteristics of objects in two-dimensional images. Desire for simplification of axis information leads to the search for optimal axes through the use of a Hough transform algorithm. Shape analysis provides the motivation for use of a similar Hough transform for detecting circles in the image.

2 Vision System Basics

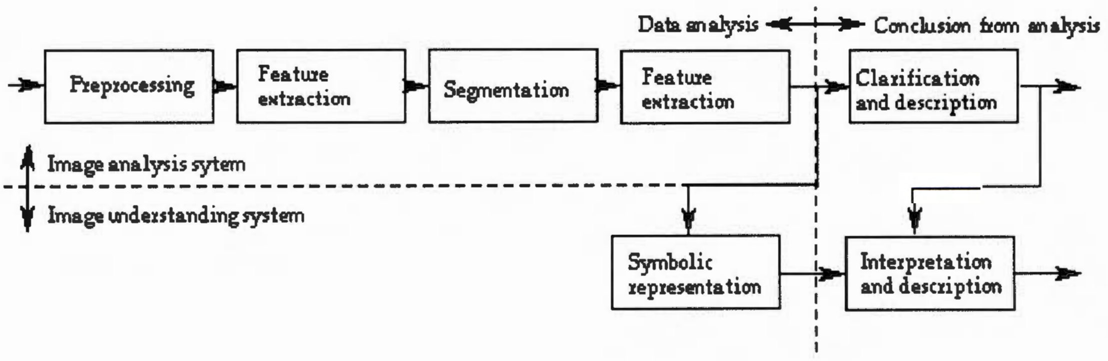

Figure 1: A computer vision system[4, p. 343].

Systems that attempt to analyze shapes and recognize objects go through a somewhat generic collection of steps. This process, pictured in Figure 1, includes gathering information and determining what it can from that information. Image anlysis basically involves three major steps: feature extraction, segmentation, and classification techiniques[4, p. 343]. For example, after some preprocessing involving enhancements of an image and internal representation, certain features are extracted for segmentation into components. The conclusions from analysis help aid in determining relationships between different objects in a scene which aids in description[4].

An existing vision system[7] contains a substantial body of image analysis tools. These tools include the basic utilities to display, print, store and read two-dimensional images. Also included are many algorithms that manipulate and display complex information in order to provide meaningful analysis for both color and black-andwhite images. At each stage of manipulation, visual output provides direct feedback for inspection by the user.

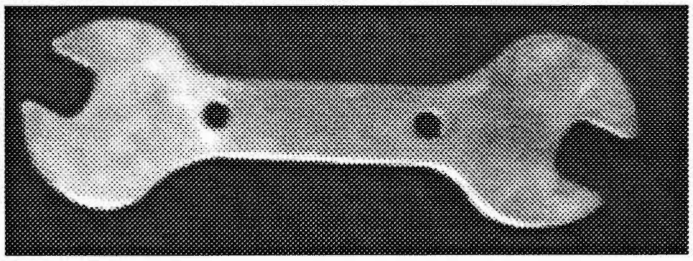

Figure 2: Gray-scale image of a spanner.

Relatively simple images are used for beginning analysis. A relatively small example image pictured in Figure 2 contains only one object, fairly cleanly captured. The entire object exists within the bounds of the image and there is a high contrast between the object and its background. Note the variety of shapes that make up this simple spanner. Even for a single object in a fairly small image, analysis is complex. The problem gets substantially more difficult as one increases the uncertainty in position, orientation or contrast[1]. Multiple objects in an image provide even more complexity, as the overlap and relationship between the objects must be represented and understood.

Key starting points for image analysis include the outlines and general shapes of objects in an image. On a high level, contours are viewed as the outlines of an image. They mark the start towards image understanding.

2.1 Contours

A major step towards image recognition involves finding the contours in an image. Contour-based representations encode the bounding contour of a shape[1]. For a casual onlooker, contours appear simply as the outlines of an image. It appears as if someone placed the object on top of a piece of paper and drew the lines tracing its outline. Contours represent places where a large contrast exists in image intensities. The computer looks for places where the brightness of the image changes abruptly, in magnitude or in the rate of change of magnitude[6, p. 75].

Detection of contours is performed as a two-step process: first, the detection of short linear edge segments where image intensities change, followed by the aggregation of these into extended edges with added analysis[6, p. 78]. Several edge detection methods have been developed such as the Marr-Hildreth and the Canny[6, p. 87-90] edgefinding algorithms based in Gaussian smoothing methods.

Conceptually, contours are the outlines of the objects in the image. This simplified

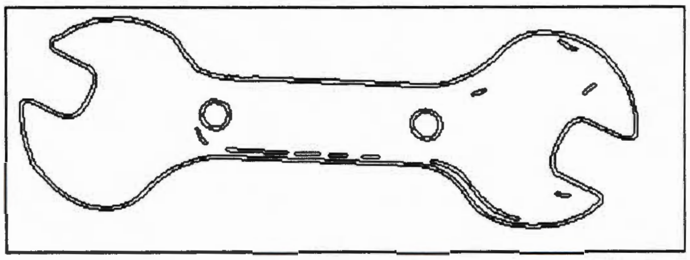
Figure 3: Contours of the spanner (found using Canny edgefinder).

view comes through analysis of simple scenes with high contrast between object and background such as the example image, Figure 2. The contours seen in Figure 3 are produced using the Canny operator on the example image. The contours appear to be long lines that closely mimic the places where image intensities change. However, each contour consists of hundreds of an edgesegment with an associated orientation describing the direction the contour is situated. Contours which outline the entire main object as cleanly as pictured in Figure 3 are rare. Often a specific contour delineates only a small portion of the border, with another one picking up from where it left off. The contours provide a simple starting place for image analysis. In addition, reflections from the object in the picture may cause areas that appear in the contours which are part of the the object, not a separate object. Lighting and noise in the input images greatly affects the output from the contour finding analysis.

It is difficult to compute or determine important regional properties, such as symmetry, or such descriptions as "elongated and curved," in a purely contour-based representation of shape[1]. These descriptions provide useful keys in shape and object analysis systems. Especially when dealing with partially occluded (or hidden) objects, contours cannot provide insight about object overlap and overall shape. Even with these shortcomings, contours provide a useful starting point. While they do not provide all the information needed, finding the contours is an important step in analysis of an image.

2.2 Smoothed Local Symmetries

Since basic contour information does not seem adequate, suggesting shape features as the next analysis step seems logical based on the earlier discussions of the differences between stick figures and bears. Shape features are useful, can be computed efficiently, and are reasonably insensitive to noise and quantization[1]. Again the complex images picturing multiple objects cause a problem because "descriptions of the visible portion of occluded objects bear an arbitrary relationship to the value that would be computed for the whole object" and it follows that it is at best difficult, and usually impossible, to recognize occluded parts using global features[1].

While a contour-based representation demonstrates only part of the information needed for overall shape analysis and object recognition, smoothed local symmetries (SLSs) provide the needed combination of shape and contour information. Smoothed local symmetries provide an improved two-dimensional shape representation because each represents both the bounding contours of a shape fragment and the region that it subtends or encloses[1]. This makes representation and adequate analysis of partially occluded objects possible.

Detecting symmetry in pieces of contour compares to analysis of ink blot paintings. Creasing a piece of paper, splattering ink down the center, and pressing the two sheets together creates an image that is symmetric about the central fold. Just as the two halves mirror each other across that center fold, small pieces of contour demonstrate symmetry about an imaginary axis. These axes are extremely important in figuring out what shapes are represented, coming to conclusions about how object shapes are joined together, and finding the general orientation of the objects the computer is attempting to recognize.

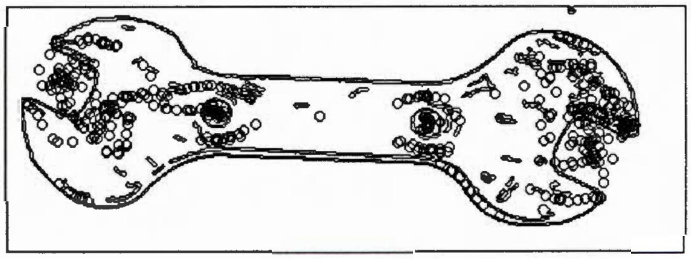

Figure 4: Candidate axis points with contours.

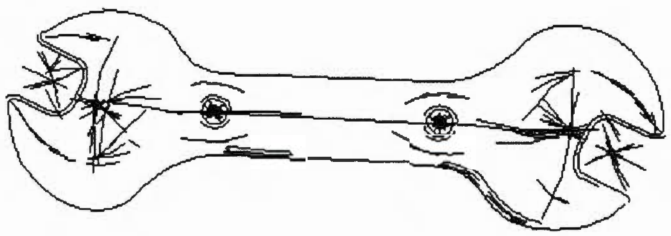

Figure 5: Smoothed local symmetries: axes with contours of the spanner.

Searching for smoothed local symmetries uses the contour information, breaking contours into smaller curvesections, and analyzing the symmetry between various pieces. The points that lie on the axes of symmetry are included in the output and are shown superimposed on the contours of the image shown in Figure 4. Further analysis of the symmetry information yields the complex set of candidate axes seen in Figure 5.

3 Axis Simplification

After the smoothed local symmetries have been found, a whole collection of candidate axis points exist. These axis points are connected into a complex tree-like structure based on the pieces of contour about which they are symmetric. The resulting axes are in Figure 5. The output is a huge collection of possible axes, each one a good candidate. However, the set of potential axes is far too complex for even simple global and regional information for shape and object recognition functions to use this information. The goal is to find a small number of the best axes, not the multitude of axes as seen in Figure 5. These axes are important in determining orientation of images and determining how various shapes are joined together.

Determining the best axes is similar to playing connect-the-dots in an attempt to figure out the small set of axes which represents the needed information. A problem arises because the candidate axis points are not numbered and therefore the final goal remains a mystery. The simple trial and error is not a timely option. A large collection of candidate points are imported from the smoothed local symmtry analysis. An optimal axis must represent the figure well from both the local and regional perspective with the bigger picture the ultimate goal.

4 Hough Transform for Axis Detection

In the search for the optimal axes in a picture one must determine a sizable series of collinear points that closely match the orientation of the symmetries on either side of them. This can be viewed as a popularity contest to decide which line contains the most points. The Hough transform provides for efficient detection of straight lines by replacing the problem of finding collinear points with that of finding collinear lines [3, p. 11]. Each of the figure's axis points is transformed into a straight line in parameter space and votes for the lines that pass through it.

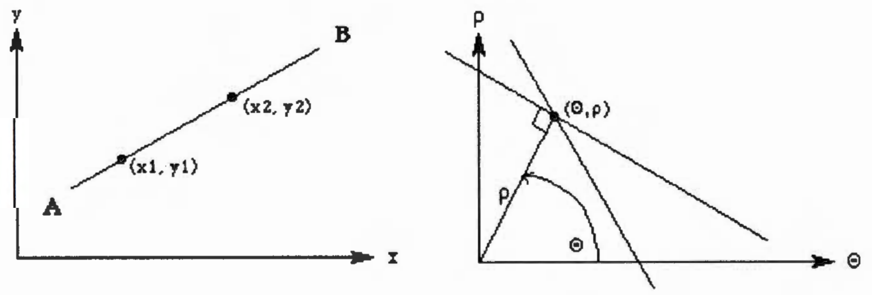

Figure 6: The points $(x1, y1)$ and $(x2, y2)$ which are collinear in image space, intersect at a single (θ, ρ) point in parameter space.

The set of candidate axis points of the smoothed local symmetries are used as input and each point is converted to the possible lines in parameter space it could lie on based on an angle θ and a distance ρ from the origin figured for each (x, y) coordinate (See Figure 6). The equation enabling the conversion: $\rho = x \cos\theta + y \sin\theta$.

While an infinite number of lines pass through a given point (x, y), thresholds are set for the θ values so that a finite number of lines are swept out for each point, perhaps every five degrees. In addition, as the ρ values are determined for each point using the chosen θ values, additional thresholds are needed for the values to group them in the accumulator. The number of divisions and range of the accumulator determine this information. While defaults are provided, both these parameters are tunable by the user, allowing for the best mix of thresholds for a given set of smoothed local symmtries.

It is possible to search out the entire 360 degree range for each point, allowing them to vote for lines that pass in all directions through them. Constraining the orientation of the lines for which each points votes provides more accurate results. The basis for this orientation vector is a series of vectors through associated candidate axis points. Each smoothed local symmetry has three axis points associated with it. A vector between two of these is likely to mimic the axis through that small region. So, for each set of candidate axis points, several vectors are chosen between these candidate axis points thus providing the needed starting orientaion.

A small range of surrounding θ values may be swept out after determining the starting orientation. This allows for slight inaccuracies in the starting orientation while tightly constraining the axes for which a point votes. A tunable threshold is predetermined for the range of values swept out for each voting point. This assures consideration of those lines that closely match the starting orientation. For each (θ, ρ) value computed for the (x, y) axis point, a vote is recorded in the accumulator. This accumulator tallies the votes as each point is considered.

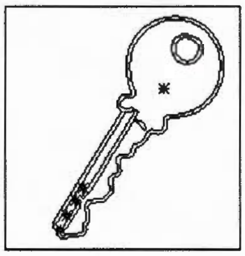

Figure 7: Points along a popular axis with the contours of the key.

After voting is completed for each candidate axis point, the accumulator contains the number of votes for each possible axis in the image. The greater the number of points within a threshold of a line, the more axis points which lie along the line and thus more likely the line is one of the optimal axes in the image. The accumulator

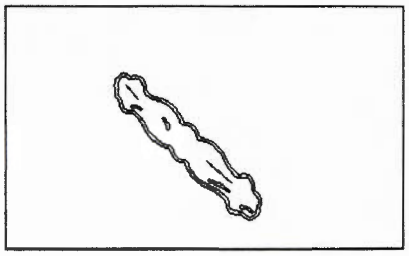

Figure 8: Two optimal axes with contours of the image.

used is to discover which lines are the best candidate axes in the image. This is done with a smoothing array that determines the biggest value in a neighborhood of n values. This provides a rough approximation of the "best" axes. Additional filtering includes line fitting to assure the axes found best represents the points along each axis.

A collection of points along an optimal axis are shown in Figure 7. These points are chosen after all candidate axis points have voted and the accumulator has been analyzed. In Figure 8, the points along the optimal axes have been converted to the axis each represents. These axes are the desired output from the application of the Hough transform algorithm to find optimal axes.

5 Hough Transform for Circle Detection

Because regional shape information is as important as axis information, application of a Hough transform algorithm for efficient circle detection is also useful. The problem is transformed from finding the minimum error over all points to searching for curves that fit a maximum number of points[6, p. 113]. Thus, the goal is changed from finding a few large circles that reflect the entire set of points to finding the collection of points that best represent circles. Circle detection is based on roughly the same principles as line detection, but the equations are more complex: $x = r\cos\theta, y = r\sin\theta$[5, p. 120]. The accumulator now must hold three values: (a, b, r), where (a, b) is the center of the circle voted for, and r is the radius.

Each point along the contour must vote for the circles on which it may lie. Sweeping out this added third dimension of values appears it is extremely time consuming. Finding circles that may have a radius in any direction from the given point would be time consuming and present highly complex results. However, the set of contours used as input contains orientation information for each point along the contour. This orientation describes the direction the contour is moving at that point and follows the image. This orientation serves as a vector tangent to any circles passing through the (x, y) input point. The centers of candidate circles must lie along the gradient, the line perpendicular to this tangent vector. This gradient information allows simplification of equations from three degrees of freedom to only one[6, p. 109]. There is only one line along which the radii can lie for a given point. Thus, there are relatively few points to sweep out for each input point.

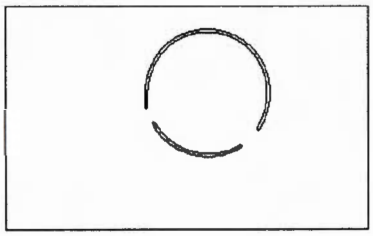

Figure 9: Contours of a single circle in an image.

In the simple example seen in Figure 9, there is only one circle in the image. This points along the contour each vote for the radii that extend perpendicular to their orientation vector. The points that are swept out for the image are shown in Figure 10. The ordered appearance of these points is due to the threshold values which cause points within similar areas to cast votes for the same accumulator entry.

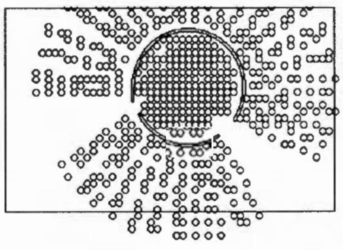

Figure 10: The contours of the circle with candidate circle centers swept out through the Hough transform.

Chosen thresholds limit the radius size swept out based on the maximum size of circles expected or may be based on maximum image size to find any possibilities. Only a finite number of radii will be voted for in each case. In small images, searching the maximum range is feasible, while in larger images this becomes time consuming and limiting the circle size is necessary.

As with the line detection, an accumulator entry with more votes is more likely to be the center of a circle. The entries with the most votes are collected through a series of filters applied to the accumulator. A sample of the output is included in Figure 11. Several of the circles that received the highest number of votes are pictured. Squares mark the approximate center of each circle. As with the line detector, there are many directions to proceed with output filters that further fine tune the circle accumulator output. If a finite curve is desired, with endpoints that match the placement of the

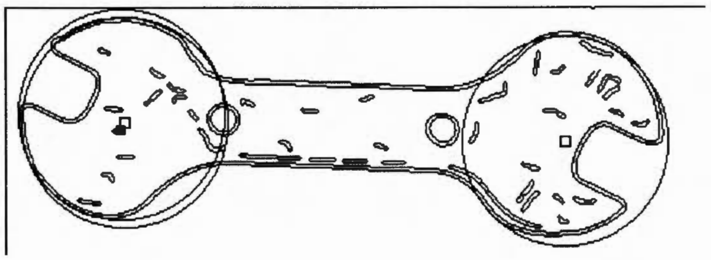

Figure 11: Contours of the spanner, the most popular circles found, and their centers.

voting point, or if curve fitting for portions of the circle is desired, more complex analysis is required.

6 Analysis

The Hough Transform algorithm accurately finds the most popular lines and circles with parameters that can be fine tuned for better results. In each case, it finds many top lines and circles that need further processing to be useful. The axes found are useful in determining the type of intersection of pieces of an image and provide information needed as the next layer is added in the journey toward a system that accurately provides shape and object recognition.

Application of a Hough Transform provides useful methods for axis and circle detection in the images being considered. However, applying it to more complex figures breaks down in both efficiency and accuracy rather quickly due to the number of parameters that must be swept for each point. Generalizing to other curves is relatively easy. Unfortunately exponential growth of both the accumulator size, computation time, and the associated analysis, make this solution to the problem of finding more complex curves impractical. Therefore, applications of a Hough transform algorithm in axis and circle detection remain useful without pushing the limits of the accuracy of more complex figure analysis using the Hough transform.

7 Conclusions

The axis and circle information provide a stepping stone towards recognition of the stick figure drawn by the kindergartner. With fewer axes to deal with and accurate circle finding to delineate the head of the figure, the goal of recognition can be reached. Eventually such a machine may be able to recognize bears and stick figures, as well as more complex scenes.

References

[1] Michael Brady and Harua Asada. Smoothed local symmetries and their implementation. *The International Journal of Robotics Research*, 3(3):36 – 61, 1984.

[2] Jonathan H. Connell and Michael Brady. Generating and generalizing models of visual objects. Technical Report A.I Memo 823, Massachusetts Institute of Technology Artificial Intelligence Laboratory, July 1985.

[3] Richard O. Duda and Peter E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15, January 1972.

[4] Anil K. Jain. *Fundamentals of Digital Image Processing.* Prentice Hall, 1989.

[5] Carolyn Kimme, Dana Ballard, and Jack Sklansky. Finding circles by an array of accumulators. *Communications of the ACM*, 18(2):120 – 122, February 1975.

[6] Vishvjit S. Nawla. *A Guided Tour of Computer Vision.* Addison-Wesley Publishing Company, 1993.

[7] Vision and University of Iowa Simulation Group. Vision course code. This code is the product of ongoing research under Dr. Margaret Fleck, Professor of Computer Science at the University of Iowa.

Appendix A Compute-Hough Code

```
;; Hough transform top level function
;; Karissa Hobert
;; Summer 1994

(in-package "VISION")
(export '(HOUGH-TRANSFORM))

;; Top level call to setup accumulator,
;; get axis points into the needed array,
;; and make the call to compute-hough which does the actual work.
;;  Takes the symmetries (symmetry regions) returned by compute-symmetries.
;; Default vars defined with rest of hough-utils.
(defun hough-transform (symmetries &key (threshold *hough-line-threshold*)
   (angle-range *hough-angle-range*)
   (angle-increment *hough-angle-increment*)
   (debug *debug*))

  (format t "Computing hough transform with
line threshold ~d,~% " threshold)
  (format t "angle-increment ~f degrees and angle-range ~f."
angle-increment angle-range)
  (let* ((accumulator (hough-accumulator symmetries
 :angle-increment angle-increment
 :threshold threshold
 :angle-range angle-range
 :debug debug)) )
    (format t "~%~d axis points to consider."
(length
(get-all-keys
(accumulator-axis-hash accumulator))) )
    (format t "~%Sweeping out ~d theta values for each point."
(* 2 (floor (/ (accumulator-angle-range accumulator)
 (accumulator-angle-increment accumulator)))) )
    ;; must sweep out on each side of the point
    ;; (so *2), range/increment = number to sweep out
   (setq accumulator (compute-hough accumulator))
   (format t "~%~d unique lines voted for."
(length (get-all-keys (accumulator-votes accumulator))))
   (setq accumulator (smooth-hough-accumulator accumulator :destructive t))
   (format t "~%~d lines left after smoothing."
(length (get-all-keys (accumulator-votes accumulator))))
   accumulator
  ))
```

```
;;; Utilities for hough transform algorithm
;;; Karissa Hobert
;;; Summer 1994
;; Revised version, using hash table instead of array for accumulator...

(in-package "VISION")
(export '(SMOOTH-LINE-ACCUMULATOR ))

;;; In this file:  (...just so I can keep track!)
;;;    defstruct accumulator
;;;    defun normalize-coord
;;;  (converts (x,y) -> (theta, rho) parameter space
;;;    hough-accumulator (makes accumulator)
;;;    computer-hough (takes votes using hough-transform algorithm)
;;;    get-line-points
;;;  (gets coordinates that voted for line from accumulator)
;;;    smooth-line-accumulator (smooths accumulator in NxN chunks)
;;;    hough-find-axes (simplified filter to find "good" axes)
;;;    get-max-line (gets the max line....)
;;;    sort-keys (not yet completed....)


;;; Default vars:
;;;     *angle-increment*
;;;     *hough-line-threshold*
;;;     *hough-angle-range*

(defun lo ()
  (load "~/h2/hu.lisp")
  (load "~/h2/points.lisp")
  (load "~/h2/hough-top.lisp"))
;;; =================
;;; structures
;;; =================

;; This file contains the basic functions needed
;; to compute the hough transform
;; on the axis point structure returned
;; by compute-symmetries.  It converts the
;; axis points in that structure and an angle
;; being swept out to points in linespace
;; that correspond to that line in 2-space.
;; equation for conversion: (x, y) and Theta -----> (theta, rho)
;;;   Theta = angle / angle-increment
;;;   rho = (rho / length-increment) + zero-offset)
;; Each point converted to a line then "votes" for the line it lies on.
;; The most popular lines are then collected
;; and added computation is done on them
;; to determine the best fitting line for that
;; collection of points which voted for
;; it.  Further tests are completed
;; (and need to be added) to determine if this line
;; that has a lot of axis points lying near it,
;; is a "good axis" and should be kept
;; in the final output.  (These last steps need improving and tuning.....)

(defvar *debug* nil)  ;; t turns on extra output features.

(defvar *default-scaler* 2)
       ;; used to keep the votes in range of the 8 byte smoothing array

(defvar *min-votes* 2)

(defvar *hough-angle-increment* 5)
       ;; degrees, default value to increase theta
       ;; value by when voting, ex: 10 --> it will sweep out every ten
       ;; degrees (within it's angle-range).

(defvar *hough-line-threshold* 5)  ;; degrees, just _how_ close the
       ;; values will be to each other, the smaller the value, the more
       ;; precise the values are in voting.

(defvar *hough-angle-range* 20)  ;; degrees, default value to sweep out
       ;; around starting place when "voting", sweeps out 20 degrees on
       ;; either side of the starting place (based on orientations)
       ;; Therefore, 40 degrees are swept out by default here....
```

```
(defstruct accumulator
  (angle-range nil)
;; The range to sweep out, in degrees
        ;; ex: 20 --> sweep out 20 degrees each way from start angle
  (votes nil)
;; hash table that stores "votes" for each line
  (angle-increment nil)
;; size of  each angle  (determines number of angles considered)
        ;; actual angle increment value to test how close
;; needs to be to theta value is 1/2 this
  (threshold nil)
;; how close each point needs to lie to the line it's near
        ;; used to measure how close the rho values need to be to each line
  (axis-hash nil)
;; hash of axis points in asix-point structure.
        ;;  each entry here is an array of num-theta values for tallying

  (stats nil)  )
;; filled by hough-stats structure in hough-stats.lisp
```

```
;;; =================
;;; utilities
;;; ================

;;; First it should be noted that there are two
;;; different representations of the linespace coordinates.

;;; One is simply the straightforward representation
;;; that would normally be figured (i.e. by using normalize-coord) and the
;;; other is the linespace representation used in the array representation.
;;; Theta = angle / angle-increment
;;; rho = (rho / length-increment) + zero-offset)
;;; This allows for easy access to the 2D accumulator array.
;;; Functions should specify which type they take.
;;; Often the accumulator array format
;;; requires the accumulator be passed along
;;; with the arguments so the increment and
;;; zero-offset values needed can be accessed.
;;; This also provides a clue about which coords rep is being used.

;; Converts to line space (theta rho) value.
;;  (or  (angle length) as specified here)
;; Takes coordinates in the (x y) format and the angle and returns the pair
;; of "normalized parameterizations" for the pair.  (angle length)
;; Angle is expected in degrees.
(defun normalize-coord (coordinates angle)
  (let* ((first (* (car coordinates) (sin (degreetoradian angle))))
 (second (* (cadr coordinates) (cos (degreetoradian angle))))
 (length (+ first second)))
    (list angle length)  ))

;; Make-accumulator-array makes the array used
;; to tally the "votes" for various
;; lines.  Sets up the bounds of the array
;; Length-range is the approximate size of the image.
;; Used to determine array bounds

;; Note: only values for angle from 0 <= angle < 180 are considered.
;; Therefore there
;; is no duplication of lines.  i.e. every  line is unique.
;; angle-increment is the number of degrees represented by one aray value.
;; length-increment is the distance between each rho value.
;; Sets up the constants to be used in accessing the accumulator array.
;; Also determines the size of the bins (threshold of values)
;; for the lines.
;; Returns an accumulator array.

(defun hough-accumulator (symmetries
&key (angle-increment *hough-angle-increment*)
       (threshold *hough-line-threshold*)
       (angle-range *hough-angle-range*)
       (debug *debug*))
  (format t "~%Building the accumulator. ")
  (let* (acc)
    (setf acc (make-accumulator))
    (setf (accumulator-axis-hash acc)
;; make the hash table of axis points
  (make-axis-point-hash symmetries :debug debug))
    (setf (accumulator-threshold acc) threshold)
    (setf (accumulator-angle-increment acc) angle-increment)
    (setf (accumulator-angle-range acc) angle-range)
    (setf (accumulator-votes acc) (make-hash-table :test #'equal))
  acc))

;; Takes an accumulator array as argument.
;; Uses the votes hash table stored there to take 'votes' for each of the
;; axis points in the array -- as to which of the lines it lies near.  How
;; near it must be depends on the size
;; of the increment values and therefore
;; the bins setup in the hough-accumulator function.
;; Converts each (x y) coordinate to its
;; equivilent linespace representation
;; and uses the array access formula to cast a vote for the line.
;; key format: (x y radius) point

(defun compute-hough (acc)
  (format t "~%Transforming coordinates to linespace
and voting for closest line.")
  (let* ((axis-point-hash (accumulator-axis-hash acc))
 (keys (get-all-keys axis-point-hash))
 ;; accumulator of votes -- hash table
 (vote-hash (accumulator-votes acc))
 starting-angle1
 starting-angle2
 (angle-inc (accumulator-angle-increment acc))
 ;; number of values to sweep out
 (number-angles (floor (/ (accumulator-angle-range acc)
  angle-inc)))
 (threshold (accumulator-threshold acc))
 (angle 0) )
```

```
    ;; For each key in the hash table of axis points (point)
    (dolist (key keys acc)
      ;; For each entry under that key
      (dolist (entry (gethash key axis-point-hash))

      ;; using only two of the orientations to vote
      ;;  (could easily be expanded to use others)
      ;; the first two (used ones) are:  Axis1 -> Axis2  AND Axis1 -> Axis3
      ;; These are the starting angles from which
      ;; to sweep out the range of angles

      ;; Why so complex?
      ;;   ....need to make sure the starting angle is along an angle
      ;; to be swept out --> a multiple of angle-inc
      (setq starting-angle1
(* angle-inc
    (round (degree-from-orientation
(car (axis-point-orientations entry)))
      angle-inc)) )
      (setq starting-angle2
(* angle-inc
     (round (degree-from-orientation
(cadr (axis-point-orientations entry)))
      angle-inc)) )


      (vote key angle starting-angle1 vote-hash threshold)
      (vote key angle starting-angle2 vote-hash threshold)
      ;; Sweep out all angle values considered (for each point)
      (dotimes (number number-angles acc)
       (setq angle (* (1+ number) angle-inc))
       (vote key angle starting-angle1 vote-hash threshold)
       (vote key angle starting-angle2 vote-hash threshold)
     )
 )
      )
    )
  )


;; Setup increment value for the votes hash entry.  Based on the
;; aspect value currently.  Can be as complex a weighting procedure as
;; needed. (simple for now)
;; vote-point
```

```
(defun vote (axis-point angle starting-angle vote-hash threshold)
 (let* ((line-point1 (normalize-coord axis-point (+ angle starting-angle)))
 (line-point2 (normalize-coord axis-point (- angle starting-angle)))
 ;; first point already along correct threshold, need to make sure
 ;; that first one also lies along a well-known rho value so those
 ;; within the threshold will vote for the same line
 (vote-point1 (list (car line-point1)
    (* threshold (round (cadr line-point1) threshold))))
 (vote-point2 (list (car line-point2)
    (* threshold (round (cadr line-point2) threshold))))
 (entries (list vote-point1 vote-point2))  )

    ;; Does an entry already exist for this linespace point?  If
    ;; so, push this new point onto the list, otherwise, create
    ;; a new hash table entry and and make the list the new point
    (dolist (vote-point entries vote-hash)
    (setq vote-entry (gethash vote-point vote-hash))
    (cond (vote-entry (setf (gethash vote-point vote-hash)
    (cons axis-point vote-entry)))
  (t (setf (gethash vote-point vote-hash)
   (list axis-point)))) )
    ))

;; This function takes the (cos, sin) vector
;; and changes it to the number of degrees from 0.
;; It will always return a value between -90 and 90 degrees.
(defun degree-from-orientation (orientation-vector)
  ;; if (0 0) then return 0 degrees
  (cond ((and (zerop (car orientation-vector))
      ;; will avoid division error....
      (zerop (cadr orientation-vector))) 0)
(t (radiantodegree (atan (/ (float (cadr orientation-vector))
    (car orientation-vector)))))  ))
```

```
;; Smooth the accumulator in NxN chunks.
;;  (The smaller the n, the faster it works.)
;; Radius is the var that determines the size of the NxN chunk
;;        (i.e 1 = radius of 1 = 3x3 chunks)
;; NOTE: changes accumulator contents
;;  to lists of (radius value) stored in each key
;; only IF the destructive flag is set to t.
;; Those not the "max" in the neighborhood
;; are not transfered to a new hash accumulator.
(defun smooth-hough-accumulator (acc &key (debug *debug*) (radius 1)
    (scaler *default-scaler*) (destructive nil)
    (min-votes *min-votes*) )
  (let* ((N (1+ (* 2 radius)))
 (vote-hash (accumulator-votes acc))
         (result-array (make-array (list N N)
:element-type '(unsigned-byte 8)))
         (keys (get-all-keys vote-hash))
         (smoothed-hash (make-hash-table :test 'equal))   )

    (format t "~%Smoothing accumulator with a radius of ~d." radius)
    (format t "~%~d keys to consider." (length keys))
    (cond (debug (format t "~%         Point   #votes")))

    (dolist (key keys)
    (setf temp-array (setup-smoother-array key N acc))
    (setq result-array (max-filter temp-array radius))

    ;; if it's the max in the neighborhood, add to new smooth-hash
    ;; else let it be
    ;; The value of (aref radius radius array) is
    ;; the center element of the array created and smoothed over

    (cond ((and (= (aref temp-array radius radius)
   (aref result-array radius radius))
(> (aref temp-array radius radius) min-votes))
   (setf (gethash key smoothed-hash)
 (gethash key vote-hash))
   (cond (debug
  (format t "~%max key ~7A  ~4d " key
  (aref result-array radius radius))))
  )))

    (cond (destructive
   (setf (accumulator-votes acc) smoothed-hash) acc)
  (t smoothed-hash))   ))
```

```
;; Sets up the temporary array of values
;; needed to smooth the voting accumulator
;; Based on a theta-rho sort of array (similar to (x y) used to)
(defun setup-smoother-array (start-point N acc)
  (let* ((temp-array (make-array (list N N)
:element-type '(unsigned-byte 8)))
 (angle-inc (accumulator-angle-increment acc))
 (rho-threshold (accumulator-threshold acc))
 (vote-hash (accumulator-votes acc))
 (start-theta (car start-point))
 (start-rho (cadr start-point))
 (radius (floor (/ N 2)))
 theta-val  rho-val)

    (dotimes (theta N temp-array)
       (dotimes (rho N)
(setf theta-val (- start-theta
   (* theta angle-inc)))
(setf rho-val (- start-rho
 (* rho rho-threshold)))
(setf (aref temp-array theta rho)
      (length (gethash (list theta-val rho-val) vote-hash)))
))
))

;; Simplified filter to determine if the lines found are "good" axes.
(defun hough-find-axes (accumulator scratchpad &key (radius 1) (debug nil))
  (let* ((max-hash (smooth-line-accumulator
accumulator :radius radius :debug debug))
 (keys (get-all-keys max-hash))
 line-points line-equation)
    (setq keys (sort-keys keys max-hash))  ))

(defun get-max-line (accumulator linespace-point scratch)
  (let* (line-points line-equation)
    (setq line-points (get-line-points linespace-point accumulator))
    (setq line-equation (hough-best-fit line-points))
    (draw-line line-equation line-points scratch)))


(defun sort-keys (keys max-hash)
  (format t "~%Sorting keys.")
  (let* ((keylist keys)
 (max-list nil)
 (maxval 0)
 max-array max-entry
 (count 0))

    ;; get total list of the keys
    (dolist (key keys)
 (dolist (entry (gethash key max-hash))
(push entry max-list)))
    (setq max-array (make-array (length max-list)))
    (setq max-entry (car max-list))
    (dotimes (count (length max-list) max-array)
(dolist (entry max-list)
(cond ((> (car (last entry)) maxval)
       (setf max-entry entry)
       (setf maxval (car (last entry))) )))
(setf (aref max-array count) max-entry)
(remove max-entry max-list)
(setf max-val (car (last (car max-list))))
(setf max-entry (car max-list))))
```

```
(in-package "VISION")
(export '(HOUGH-BEST-FIT))

;; Takes the near-indices field of a candidate and the array of axis
;;  points and returns the line which is the best fit to all of the
;;  points represented by near-indices
;;  represented by (slope x-intersection).

(defun hough-best-fit (pair-list)
  (let* ((best-line (multiple-value-list (least-squares pair-list)))
 (old-slope (cadr best-line))
 (old-int (car best-line))
 (new-slope (/ 1 old-slope))            ; transpose to rotated axes
 (new-int (- 0 (/ old-int old-slope))))
    (list new-slope new-int)))

(defun get-hough-axes (accumulator)
  (let* ((maxpoints (smooth-accumulator accumulator))
 line-list)
    (dolist (x maxpoints line-list)
    (setf line-list (cons (hough-best-fit (get-line-points x accumulator))
  line-list) ))))


;; Changes the coordinates to near horizontal to pass to the least-squares
;; line-fit algorithm.  Avoids problems with near vertical lines.
;; Args include the pointlist of (x y) coordintates found along the line
;; with normal of angle theta (passing through the origin -- in linespace
;; coord) and the accumulator.
;; Please note when using with hough transform that the angle passed to
;; the function must be in degrees
;;  (array offset of theta * angle-increment)
;; Returns list of rotated axis points AND the angle theta (needed for
;; conversion back.)
;; Rotation to nearly vertical is desired, so an additional 90 degrees...
(defun rotate-axis-points (pointlist theta)
  (let* ((angle (degreetoradian theta))
 (sin-theta (sin angle))
 (cos-theta (cos angle))
 (rotated-pointlist nil)  )
    (format t "~% angle ~d degrees" (radiantodegree angle))
    (format t "~% sin ~f = ~f" angle sin-theta)
    (format t "~% cos ~f = ~f" angle cos-theta)
    (dolist (point pointlist rotated-pointlist)
    (setf rotated-pointlist (cons (list
    (difference-cos cos-theta sin-theta
     (cadr point) (car point))
    (- (difference-sin cos-theta sin-theta
     (cadr point) (car point))) )
     rotated-pointlist)))  ))

;; Takes (theat rho) coordinates and
;; returns a slope based on that line in linespace.
;; x = rho (sin theta)     y = rho (cos theta)
;; x-int = rho / (sin (90 - theta))
;; Returns (slope , x-intercept)
(defun slope-from-linespace (linespace-point)
  (let* ((x (* (cadr linespace-point) (sin (car linespace-point))))
 (y (* (cadr linespace-point) (cos (car linespace-point)))) )
    (list (/ x y) (/ (car linespace-point)
     (sin (degreetoradian (- 90 (cadr linespace-point))))) )))

;; Takes the accumulator as an argument.  Returns
;; Draw flag specifies whether the results
;; will be drawn to a scratchpad or not.

(defun get-axes (accumulator &key (radius 3) (method :c) (draw nil)
     (scratchpad nil))
  (let* ((max-pointlist (smooth-accumulator accumulator :radius radius
                                                       :method method))
        line-points
 rough-line
 correction
 best-fit-line
 (line-list nil)
 line2
 scratch2)
    (setf scratch2 (make-scratchpad))
    (dolist (point max-pointlist)
    (setf line-points (get-line-points point accumulator))
    (setf rough-line (slope-from-linespace point))

    (setf correction (/ 1
(hough-best-fit
      (rotate-axis-points line-points
    (+ 90 (* (car point)
       (accumulator-angle-increment accumulator))) ))))
    (format t "~%Line through point ~A" point)
    (setq best-fit-line (list (+ (car rough-line) (car correction))
  (+ (cadr rough-line) (cadr correction))))
```

```
    (format t "~%rough est ~A ~%correction ~A ~% *** new line equation ~A "
    rough-line correction best-fit-line)
    (setq line-list (cons best-fit-line line-list))
    (setq line2 (hough-find-line point line-points :scratchpad scratch2))

    (cond ((and draw scratchpad)
;;     (draw-line rough-line line-points scratchpad)
   (draw-line best-fit-line line-points scratchpad)
   (draw-pointlist line-points scratchpad) )
  (draw ;; (draw-line rough-line line-points
;;    (setq scratchpad (make-scratchpad))
   (draw-line best-fit-line line-points
      (setq scratchpad (make-scratchpad )))
   (draw-pointlist line-points scratchpad))))
    (values line-list scratchpad))


(defun print-votes-list (votes-hash)
  (let* ((keys (get-all-keys votes-hash))
 (n (length keys))  )

    (format t "~%Printing list of the keys and number of votes.~%")
    (dolist (key keys nil)
    (format t "~A ~d ~%" key (length (gethash key votes-hash)))
    )))
```

Appendix B: Hough-Circle Code

```
;; This file loads the necessary functions for the circle finding code.
(in-package "VISION")

(load "~/circle/circle-finder-utils.lisp")
(load "~/circle/draw-circle.lisp")

;; Hough Transform circle-finder code.
;; Karissa Hobert
;; Summer 1994 (REU program)

(in-package "VISION")
(export '(CIRCLE-FINDER SMOOTH-CIRCLE-ACCUMULATOR HIGHEST-CENTER-VALUE))

;; Hough Transform circle finder.
;;   important equations:  (these do the conversion for radius center (a b)
;;       a = x + r (cos t)
;;       b = y + r (sin t)
;; Also note: the orientation is stored with each edgesegment
;; (accurate to about 20 degrees, I'm told)  Since this is perpendicular to
;; the gradient value and stored in a (cos sin) form, makes for easy
;; manipulation and use.
;; Works on contours the list of edgesegments returned by image-to-contours
;; Takes every nth edgesegment and votes for the circle center lying
;; along the gradient vector through the edgesegment or radius r.

;; Important to note is the change in the
;; accumulator after smooth-circle-accumulator
;; is run.  Only the points with the most votes in the
;; neighborhood are kept, and those
;; are in the form of list of (radius value) stored under each key.

(defvar *num-r-values* 6)   ;;default number of radii to search through.
(defvar *nth-contour-default* 10) ;; n = 10
    ;;every nth edgesegment that's considered.
(defvar *r-min* 10) ;; smallest radius to search for.

(defstruct hough-circle
  (contour-hash nil)  ;; hash table of contour values
;; (only every nth stored there)
  (r-min nil)         ;; min radius value to search for
  (r-increment nil)   ;; value for radius increase through search
  (num-r nil)         ;; number of radii to seach  through
  (accumulator nil)   ;; accumulator for votes
;; (set to nil after no longer needed...)
  (threshold nil) )  ;; threshold to search within


;; Top level function.
;; Calls necessary helpers to compute the circles.


;; Takes the list of contours returned by
;; image-to-contours or similar algorithm
;; and puts them in a hash table with keys
;; of the form (x y) with the list of values
;; that have the key being stored in the list.
;; Only each nth-contour is put into the hash table
;; and will therefore eventually be considered.
;; Also takes x-max and y-max as args.  These prevent the edge values from
;; being included in the hash of contours being considered.
(defun contours-to-hash (contour-list nth-contour x-max y-max)
  (let* ((hash-table (make-hash-table :test #'equal))
 key
 edgesegment )
    (decf x-max 5)
    (decf y-max 5)
    (dolist (contour contour-list hash-table)
 (dotimes (n (array-dimension contour 0))
     (cond ((= (mod n nth-contour) 0)
   (setf edgesegment (aref contour n))
   (setf key (list (edgesegment-x edgesegment)
     (edgesegment-y edgesegment)))
   ;; test each value to make sure it doesn't lie along the
   ;; border of the picture (not a useful contour)
   (cond ((and (> (car key) 5)
       (< (car key) x-max)
       (> (cadr key) 5)
       (< (cadr key) y-max) )

  (setf (gethash key hash-table)
   (cons (copy-edgesegment edgesegment) (gethash key hash-table))
  )) )))))))

;; Uses Hough Transform algorithm to compute the "votes"
;; for every nth point along
;; the contours for a circle of each radii being considered.
;; Votes kept in a hash table that serves as the accumulator.
;; The hash table accumulator consists of radii-value arrays where each
;; entry must be voted for by not only the key, but also the radii.
;; Entries are initialized to zero, then incremented.
;; If the debug flag is set, a scratchpad is made if not passed as an arg,
;; and the values of the centers that are being considered is printed.
(defun circle-finder (contours &key (r-min-value *r-min*)
       (num-radii *num-r-values*)
       (nth-contour *nth-contour-default*)
       (debug nil)
       (scratchpad nil))
  (let* ((edges (multiple-value-list (image-edges-from-contours contours)))
 (x-max (car edges))
 (y-max (cadr edges))
 (c-accum (make-hough-circle))
 (contour-hash (setf (hough-circle-contour-hash c-accum)
     (contours-to-hash contours nth-contour x-max y-max)))
 (accumulator (setf (hough-circle-accumulator c-accum)
    (make-hash-table :test #'equal)))
 (keys (get-all-keys contour-hash))
 threshold xmax ymax entry
 a b
 gradient
 max-r r-increment radius)

     ;; max radius is set to one third of the max value for the image.
     (cond ((> x-max y-max)
   (setf max-r (truncate (/ x-max 3.0))) )
  (t (setf max-r (truncate (/ y-max 3.0)))))

    ;; increment to search through
    ;; based on the min and max-r values and num-radii
    (setq r-increment (truncate (/ (- max-r r-min-value) num-radii)))

    (setq threshold (round (/ r-increment 2.0)))
    (format t "~%Finding circles with radii ~d to ~d
    for every ~dth contour." r-min-value
    (+ r-min-value (* r-increment num-radii)) nth-contour)
    (format t "~%Radius increment = ~d" r-increment)
    (format t "~%Threshold = ~d~%" threshold)

    (cond ((and debug (not scratchpad))
   (setq scratchpad (make-scratchpad))
   (display-contours contours scratchpad)))

    (setf (hough-circle-r-min c-accum) r-min-value)
    (setf (hough-circle-r-increment c-accum) r-increment)
    (setf (hough-circle-num-r c-accum) num-radii)
    (setf (hough-circle-threshold c-accum) threshold)
    (dolist (point keys (values c-accum scratchpad))
    ;; gradient is perpendicular to the tangent orientation vector
    (setf gradient (rotate-90-clockwise
      (edgesegment-orientation (car (gethash point contour-hash)))))
 (dotimes (r num-radii)
  (setf radius (+ r-min-value (* r r-increment)))
  ;; a =  x + r (cos t)    (where (car gradient) = cos t)
   (setf a (floor (round (+ (car point) (* radius (car gradient))))
  threshold))
   ;; b = y + r (sin t)    (car gradient = sin t)
   (setf b (floor (round
   (+ (cadr point) (* radius (cadr gradient))))
  threshold))
   ;; If no entry in hash table for this point, creat one.
   (cond ((not (setf entry (gethash (list a b) accumulator)))
  (setf entry
  (setf (gethash (list a b) accumulator)
  (make-array num-radii :element-type '(unsigned-byte 16)
      :initial-element 0)))) )
   ;; "vote" for the value of that radius at that point.
   (incf (aref entry r) 1)
   (cond (debug
  (highlight-location (* a threshold)
 (* b threshold) scratchpad) ))
   ;; get circles of radii on either side, reverse gradient, recompute
   (setf a (floor (round
   (+ (car point) (* radius (- (car gradient)))))
   threshold))
   (setf b (floor (round
   (+ (cadr point) (* radius (- (cadr gradient)))))
  threshold))
   (cond ((not (setf entry (gethash (list a b) accumulator)))
  (setf entry
  (setf (gethash (list a b) accumulator)
  (make-array num-radii :element-type '(unsigned-byte 16)
      :initial-element 0)))) )
   ;; "vote" for the value of that radius at that point.
   (incf (aref entry r) 1)
   (cond (debug
  (highlight-location (* a threshold)
      (* b threshold) scratchpad)  ))
))))
```

```
;; Smooth the accumulator in 3x3 chunks.
;; NOTE: changes accumulator contents
;; to lists of (radius value) stored in each key
;; value = the number of votes for that key of the given radius.
;; radius = array reference radius notation.
;; correct radius (needed later) ---> (r = radius * increment + min-r)
(defun smooth-circle-accumulator (circles &key (debug nil))
  (let* ((radius 3)
 (accumulator (hough-circle-accumulator circles))
 (temp-array (make-array
(list radius radius) :element-type '(unsigned-byte 8)))
 (keys (get-all-keys accumulator))
 (newhash (make-hash-table :test 'equal))
 entry value near-val result output)

    (format t "~%Smoothing accumulator with a radius of 3.")
    (format t "~%~d keys to consider." (length keys))
    (cond (debug (format t "~%         Point   radius  #votes")))

    (dolist (key keys)
    (dotimes (radii (hough-circle-num-r circles))
      (cond ((= 0 (aref (gethash key accumulator) radii)) )
    (t
     (dotimes (x-count radius)
       (dotimes (y-count radius)
          ;; current val to small smoother array
      (setf entry (gethash (list (- (+ (car key) x-count) 1)
 (- (+ (cadr key) y-count) 1))
 accumulator))
      (cond (entry
     (setf (aref temp-array x-count y-count)
   (aref entry radii)))
    (t (setf (aref temp-array x-count y-count) 0)) )))

      (setf result (max-filter temp-array radius))
      ;; If it's the max in the neighborhood.... then add that
      ;; entry to the new hash table, else let it be.
      ;; NOTE: hash table entries are changed to lists of
      ;; (radius value) in this exchange.
      (cond ((and (= (aref temp-array 1 1)
     (aref result 1 1))
  (> (aref temp-array 1 1) 2))
     (setf (gethash key newhash)
   (cons (list radii
       (aref temp-array 1 1))
 (gethash key newhash)))
     (cond (debug
    (format t "~%max key ~A ~4d ~4d " key radii
     (aref result 2 2))))
      ))))))
    (setf (hough-circle-accumulator circles) newhash)))

;; Test function to print the values of the centers found.
;; Call only after running smooth-circle-accumulator (or else the
;; accumulator will not be in the correct format).
(defun print-new-accumulator-centers (circles scratchpad &key
      (xoffset 0)
      (yoffset 0)
      (figure :opencircle) )
  (let* ((keys (get-all-keys (hough-circle-accumulator circles)))
 (threshold (hough-circle-threshold circles)) )
    (dolist (key keys)
    (highlight-location (* threshold (car key))
(* threshold (cadr key)) scratchpad
 :xoffset xoffset
 :yoffset yoffset
 :figure figure))))

;; Finds max value in accumulator.
;; Also called *after* the call to
;; smooth-circle-accumulator.
;; Remember the keys are stored in the (radii value) format.
;; Passes back values as actual vars with radius,
;; and keys in proper format.
(defun highest-center-value (circles)
  (let* ((accumulator (hough-circle-accumulator circles))
 (keys (get-all-keys accumulator))
 (max-val 0)
 max-info
 circle-points)

    ;; Find max value in the hash table (in terms of "votes")
    (dolist (key keys)
    (dolist (entry (gethash key accumulator))
    (cond ((> (cadr entry) max-val)
   (setf max-info (append (list key) entry))))))

    ;; once collected the max-value of the accumulator,
    ;;get rid of it.
    ;; (so can search for next biggest one easily.
    (setf (gethash (car max-info) accumulator) nil)

    ;; Get the points that voted for the center
    ;; to try to better approximate the actual center.
    (setf circle-points (get-circle-center-point
(car max-info) (cadr max-info) circles))
   ))


;; Still being built.  Doesn't search out radii from hough-transform yet.
;;  ---> just does the conversion.  Can easily be added later.
;; (along with other test to help assure
;; this is the "best" center and radii)

;; Gets the value of the points that voted for the center (w/in threshold)
;; and finds their average to be the center value returned.
;; Center and radii returned in a list.
;; Not converted to screen coord yet.
(defun get-circle-center-point (point radius circles &key (threshold nil))
  (let* ((r-increment (hough-circle-r-increment circles))
 (min-r (hough-circle-r-min circles))
 (threshold (hough-circle-threshold circles))  )

  ;; convert the center point to "real" screen coordinates
  ;; (from its "floored form" used for scaling it
  ;; according to the threshold value

  (list (list (* threshold (car point))
      (* threshold (cadr point)))
(+ min-r (* radius r-increment)))) )
```

```
;; Pick the "most popular" circles that received most votes.
;; Takes the hough-circle struct as arg.  (output of circle-finder)
;; Returns a 3-D array of the form
;;  -- total -- those voting (val > 0) -- radius
;; Debug flag simply turns on a listing of points through processing.
;; the debug makes use of the scratchpad.
;; (to make it slightly faster, remove this)

(defun find-max-circles (circles &key (debug nil) (scratchpad nil))
  (format t "Finding circles with the most votes. ~%")
  (let* ((num-radii (hough-circle-num-r circles))
 (keys (get-all-keys (hough-circle-accumulator circles)))
 ;; 3 fields for each radii: total, num voting, and (maxvalue point)
 (med-array (make-array (list num-radii 3)
:initial-element 0
:element-type '(unsigned-byte 16)))
 (max-val-array (make-array num-radii))
 (times -1)
 (acc-hash (hough-circle-accumulator circles))
 value
 (threshold (hough-circle-threshold circles)))

    (cond ((and debug (not scratchpad))
(setq scratchpad (make-scratchpad))))

    (dolist (point keys med-array)
    (dotimes (x num-radii)
     (cond (debug
    (highlight-location (* (car point) threshold)
(* (cadr point) threshold) scratchpad)))
     (cond ((= 0 (setf value (aref (gethash point acc-hash) x))))
   (t (incf (aref med-array x 0) value)
      (incf (aref med-array x 1) 1) ))
     ;; new max value?
     (cond ((> value (aref med-array x 2))
    (setf (aref med-array x 2) value)))
     ))))
```

```
;;; This is the no-doubt-about-it long and timely way to do this.
;; Much more efficiently done using algorithm in circle-finder...
;; hough-transform put to good use.

;; Takes the contour-has and determines
;; what points lie on the circle of radius
;; r, centered at (a b)
;; (from the hash table of contour edgesegments).
;; Also passed in is the list of keys.
;; This avoids having to recompute them
;; numberous times.
;; Returns the number lying on the circle
;; and the list of the points found. (x y)
;; If no points found, returns nil.
(defun get-points-on-circle (keys a b r
   &key (threshold *circle-point-threshold*))
  (let* ((points-on-circle nil)
 (numpoints 0) )
    (dolist (point keys)
    ;; If the difference between the distance between the point and
    ;; the center of the circle and the radius is less than threshold...
    (cond ((> threshold (abs (- r (vector-dist (list a b) point))))
   (setf  points-on-circle (cons point points-on-circle))
   (incf numpoints 1) )
  ))
    (cond (points-on-circle (values numpoints points-on-circle))
  (t nil)) ))


;; Also the *tough* way to do this.  Timely.  Lots of extra calculation.
;; Hough Transform can be put to good use to do the same task much faster.
(defun get-num-points-on-circle (keys a b r
      &key (threshold *circle-point-threshold*))
  (let* ((numpoints 0) )
    (dolist (point keys)
    ;; If the difference between the distance between the point and
    ;; the center of the circle and the radius is less than threshold...
    (cond ((> threshold (abs (- r (vector-dist (list a b) point))))
   (incf numpoints 1) )
  ))
    numpoints) )
```

```
;; Finds the centers and given radii for the circles with the most votes.

(defun find-max-center (circles)
  (let* ((max-array (find-max-circles circles))
 a b
 (max-list nil)
 (max-val 0)
 (keys (get-all-keys (hough-circle-contour-hash circles))) )
    ;; first figure which radius val has most votes.
    (dotimes (x (array-dimension max-array 0) max-list)
     (cond ((<= max-val (aref max-array x 2))
    (push (list x (setf max-val (aref max-array x 2))))) )
    ;; next find the center of those circles.
;;    (dolist (point keys)
;;    (setf
    ))
)


(in-package "VISION")
(export '(DRAW-ARC))

;; This function is built on top of the xlib draw-arc
;; command and provides
;; a simple interface for drawing circles and arcs.
;; Args are in screen coordinates
;; and angle1 and angle2 must be in radians.
(defun draw-arc (x y scratchpad radius &key (xoffset 0)
   (yoffset 0) (filledp nil) (angle1 0) (angle2 (* 2 pi)))
  (xlib:draw-arc (slot-value scratchpad 'windowcontents)
 (slot-value scratchpad 'gcontext)
 (round (+ (- radius) yoffset y))
 (round (+ (- radius) xoffset x))
 (* 2 radius) (* 2 radius) angle1 angle2 filledp)
  (show-limited-result scratchpad (- x (* 2 radius)) (- y (* 2 radius))
      (* 4 radius) (* 4 radius)))
```