Tim Kuehn

Visual Routines Lab Writeup

1. For point radial search, I make the assumption that the size of the image is small enough that it is not terribly inefficient to do exhaustively check each pixel. This is much simpler than worrying about finding the wedge of pixels that fall within the angles in question and proceeding to do an actual radial sweep. Such a method would also run into problems of redundancy: to make sure no pixel is missed in the search, the theta delta between radial lines would be so small that many pixels would overlap from line to line, meaning that the perceived efficiency gain might not actually be real. Thus, the exhaustive method involves 1) iterating over each pixel in the image, 2) checking whether each pixel is flagged on (since we’re dealing with boolean sketches, each pixel is either 0 or 1), then if it is, 3) checking whether it lies within the angles of concern, and if so, 4) checking whether it is radially closer to the initial point than the closest found so far, and if it is, updating the closest found.

2. For shape radial search, a similar method is used. An origin shape is specified, and the search proceeds radially from the origin’s centroid. Instead of checking pixels, centroids of shapes that fulfill a predicate will be compared. An example state machine is created in which an image of eggs is searched. The radial search checks against the predicate IsColor(“red”). Due to problems with getting UnaryShapePred to work as an argument in the function definition, the predicate IsColor is hard-coded such that the prototype is

ShapeRoot radialSearch( ShapeRoot &origin,

const Point &startPt,

const Point &endPt,

const IsColor &pred);

3. For contour search, I implemented what is essentially a breadth-first search of the contour. The prototype is

ShapeRoot contSearch( const Point &origin,

vector<ShapeRoot> shapes,

Sketch<bool> sketch)

The function is relatively simple. It creates a queue of points along the contour and pops points until it finds one that intersects a shape in the shape vector. The trick is that the points are added to the queue in order of distance (closest to farthest). It does so by first enqueuing the origin and marking it as seen. Then, while the queue is not empty, it pops a point and checks if any of its four neighbors are along the contour and unseen. If so, it checks to see if any of those neighbors both along the contour and unseen additionally intersect a shape in the shape vector. If so, that shape is returned. Otherwise, the neighbors who are along the contour and previously unseen are marked as seen and enqueued, and the process continues until a point is found along the contour that has not yet been seen and intersects a shape. This method ensures that the first point found to intersect a shape will also be the closest intersecting point to the search’s origin -- not necessarily, however, closest in terms of Euclidean distance.

At this point, all three methods (both radial searches and contour search) have been implemented. Both radial searches have been tested and proven to work. Contour search has been tested, but currently the test case has a bug (not necessarily the contour search itself, but hard to say without having a working test case). Hopefully, I will be able to fix the bug in short order, but I have handed in my work as it currently stands because at least the three functions are completed.