RobonautVision Solution Description

# Contact Information

* First Name : Balazs
* Last Name : Godeny
* Topcoder Handle : walrus71
* Email Address : godeny.b@gmail.com
* Final Code Submission File Name : RobonautVision\_walrus71.java . This is the same as my 2nd full submission with comments added and most of the unused code removed.

# Approach Used

## Approaches Considered

Looking at the scoring function I thought that the match would be decided on how little training information people can work with, so I decided on day 1 to not use any training images. Later I regretted this decision but it was too late to change my approach so I spent the last couple of days of the contest in panic mode, trying to create a non-zero scoring solution. With lots of luck I succeeded, but I am aware of the many weaknesses of my solution and have lots of improvement ideas. I think a follow up contest would be highly justified, we have just scratched the surface of this very interesting problem.

I spent several days trying to generate a simplified 3d mesh so that I could evaluate a candidate placement using, say, 100 vertices instead of ~10000. I tried approaches like descibed in <http://dev.gameres.com/program/Visual/3D/PolygonReduction.pdf> but they didn't work, probably because the given meshes were not proper manifolds.

I also tried to simplify the mesh by vertex clustering (dividing the 3d space into cells and combining vertices within each cell) but the resulting mesh didn't represent the true shape of the target object to a satisfactory degree.

## Approach Ultimately Chosen

Summary of my approach: calculate where the contour points of the target object would be on the image on as many candidate placements as time allows and match them against the edge images generated from the left and right camera photos. Pick the placement which has the highest number of contour points falling on edges.

## Steps to Approach Ultimately Chosen, Including References to Specific Lines of Code

**Preparation: collecting contour points**

This step is done when receiving the 3d model of the target object, see the simplify() method. Uniformly sample the rotation space (following <http://refbase.cvc.uab.es/files/PIE2012.pdf>) and for each rotation draw the object on a hypothetical canvas. Trace the contour of the resulting line drawing to get the external points of the drawing, then pick a couple of points from the contour by uniform sampling (in the 2d space of the drawing canvas)(fillContourPoints() method). Determine the 3d points of the model that correspond to these contour points. All these 3d points are on vertices or on edges. When this step is done, for each rotation we have a handful of points of the model that describe the overall shape of the model as it will appear on a 2d projection. Note that this is all information we'll use, e.g. we don't care about the colour or texture of the object, just the shape of its shadow.

**Image processing**

Recall that nothing is done with training images so the following applies when receiving a test image pair. For both images I remove salt and pepper noise by median filtering, downscale by a factor of 2, find interesting areas, and generate edge image (medianFilterWhereNoisy(), downScale(), makeHeatMap() and makeEdgeMap() methods, resp). Details of the last two operations follow.

The heat map is a black and white image where every pixel represents how interesting the original image looks at that point. A pixel is boring if it looks like its neighbourhood (i.e. it is considered background). To determine the heat of pixel (x,y) I trace a line in each of the 4 main directions starting at (x,y) with maximum length of half the image width and calculate the average colour difference of line pixels to the target pixel. The heat will be taken as the minimum of these 4 values.

The edge image is generated from the above heat image by a standard Sobel edge detection. The resulting edges are blurred a bit so that it will be easier to find them during location search, see later. The intuition behind doing edge detection on the heat image is that such edges should represent boring-to-interesting changes of the image and so they should coincide with the contour of the target object.

**Location search**

Finding the best placement is done in the scan() method which is an ugly nested loop of 6 levels corresponding to the 3 translation and 3 rotation dimensions. For every candidate placement place() is called which returns a fitness score, the highest score wins.

In place() we take all the contour points of the model that correspond to the current rotation, calculate where they would be on the 2d image and count how many of them fall on non-zero areas of the edge image. The placement score will be the ratio of contour points on edges to all contour points.

## Advantages and Disadvantages of the Approach Chosen

Advantages

* Conceptually simple
* No training needed
* Doesn't require colour and texture info about the target object

Disadvantages

* Unreliable if there are many objects on the images
* Throws away lots of useful information like colour of the target object
* Doesn't work if the contour of the object is not enough: e.g. it would completely fail on a dice where each side of the cube is unique but my tool would not recognize it by looking at the shadow only.

## Comments on Libraries

No libraries are embedded into my solution.

The contour tracing code is roughly based on <https://github.com/biometrics/imagingbook/blob/master/src/contours/ContourTracer.java>

The line drawing code is based on <http://tech-algorithm.com/articles/drawing-line-using-bresenham-algorithm/>

## Special Guidance Given to Algorithm Based on Training

NA

## Potential Algorithm Improvements

* Do finer grained search once the approximate location of the object is found. The current approach blindly does a coarse scanning of the 6d grid and hopes for the best. Methods like simulated annealing should be tried.
* The visibility of the contour points change based on the location of the object, currently this is ignored. I planned to extend the contourPoints array with information on where we are in the image (center, top-left corner, etc), but didn't have the time to implement this.
* Get information from a sequence of test images. For some images the environment of the photo is exactly matching a previous one, in these cases by substracting the known background we immediately could see where the object is.