

Topcoder Marathon Match - RobonautVision - March/2016

Solution Description

Wladimir Leite (wleite) - 1st place

TRAINING MODEL

My solution heavily depends on the object models, since it doesn't use any training images. My `trainingModel` implementation executes the following tasks:

- **parseModel**: parses the given PLY model, creating `Point3D` and `Face` objects for model's vertexes and faces.
- **findBox**: finds a 3D bounding box that contains all points of given model. It will be used during testing phase to make quick estimation of object's position in the left/right images.
- **interpolateModel**: although the given model is very detailed, (e.g. Drill has 11K vertexes and 21K faces), there are plain surfaces whose vertexes are far from each other (e.g. the screen of RFID Reader). To speed up calculations, my solution uses only 3D Points, no lines (edges) or polygons (faces). Therefore, it is important that points be distributed in a homogeneous way along object's surface. This interpolation step generates random points contained in each face, by taking a weighted average of each vertex position, using random weights. After this step, the number of vertexes (called `modelPoints` in my code) grows a lot (from 7K to 94K, in the case of RFID Reader).
- **reduceModel**: after the interpolation step, the number of model 3D points is reduced, using a simple greedy approach, that adds one point at time, taking the point that is the farthest from the points already included. The parameter `maxModelPoints` controls the number of points after this reduction, and my final submission uses 1,500 points. I tried other values (2K, 3K) that gave a better "quality", but the speed cost didn't pay off in my local tests.

The following figures show the original given model points of the RFID Reader and the points after the interpolation and reduction.

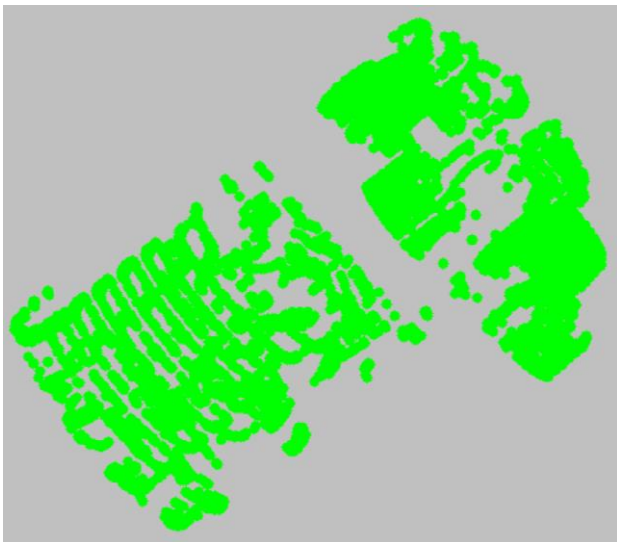


Figure 1 – Original model vertexes.

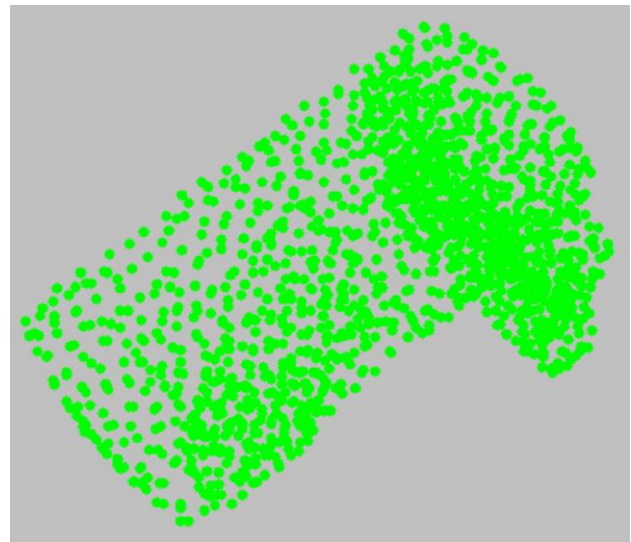


Figure 2 – Model points after interpolation and reduction.

TRAINING IMAGES

My final submission doesn't use training images at all. When I first saw the scoring formula, my idea was to start without using any training images and later use them, if they help achieving better scores. I tried two different approaches (discussed in "Failed Approaches" section), but both didn't help improving my score, so I gave up and returned to the initial solution.

TESTING IMAGES

In short, my approach just tests a lot of different position and takes the best one. The key points were:

- 1) How to test as many positions as possible?
- 2) How to evaluate a position, in a way the correct one (or close to it) produces better scores?

Each testing image processing is independent, i.e. the solution doesn't retain any information that could be used in the remaining images. The main initial steps are as follows:

- **despeckle**: removes single pixels that are clearly not part of the left and right given images. In practice, this doesn't have much effect.
- **stretch**: transforms the original pixels from the left and right images to use all range (0 to 255), for each RGB channel. Also a small effect, since most of given images already use the full range. For few "dark" images though, that improved the result.
- **resize**: as a simple way to allow testing more positions in the given time, images are resized by a factor of 8. During the competition I worked most of the time with 4, but in the end, local tests had slightly better results with 8. By looking at small images (8x reduction), it is clear that too much information was lost, so probably this can be improved. There is `factor` constant in my code that allows changing this behavior easily (since it is used in many calculations).
- **difference**: this is a very important step. It compares the RGB value of each pixel with the most common (median value of each RGB channel) of its row and column, and takes the smallest one between these two comparisons. The idea is to highlight all pixels that are "strange" in the image, so the object of interest be detected. The implemented method is very dependent of image background, i.e. the surface where object is placed and other elements present in the scene, but it worked in most of given example images. Any improvement here, using more advanced methods, or combining with other techniques, would boost the final outcome. The difference between each pixel and the "common" value is normalized in a range between 0 and 255. Ideally, this step would give a value of 255 for all pixels that belong to the object to be detected and 0 for the rest. It is worth noting that left and right images are processed independently. The following figures shows an example, with the original image and the result of the described process.



Figure 3 – Original image.

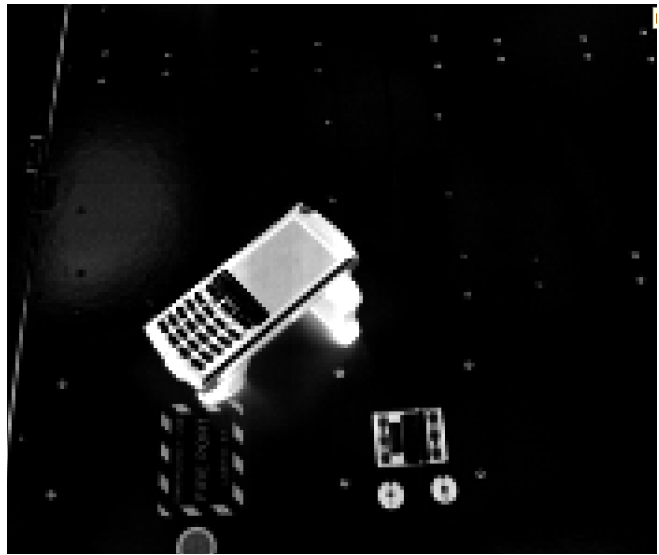


Figure 4 – Generated by “difference” process.

After these initial steps, there are two loops that tests many different positions.

- **FindRect** (not a real method, but a part of `testingPair`): this is done by the first loop, which runs until 10% of allowed time is reached. It tries random positions, evaluating using a method described later. Then a common 2D rectangle, intersection of the bounding box of the best 5 positions found so far, is taken as the “region of interest”, where the object to be detect probably is. If this method fails completely, the following steps won’t be able to recover and the returned position will be wrong.
- **refineRect**: most of the time the previous step returns a good initial guess, but misses part of the object. This step tries to expand (and sometimes to contract) the rectangle previously found, to include areas with a high concentration of “different” pixels and exclude areas with almost none. Figure 5 shows the result of the “FindRect” (in green) and the “refined” rectangle (in yellow). It is fine if the detected rectangle contains only a part of the object, but the larger its area is, more positions will be pruned in the second (main) evaluation loop.

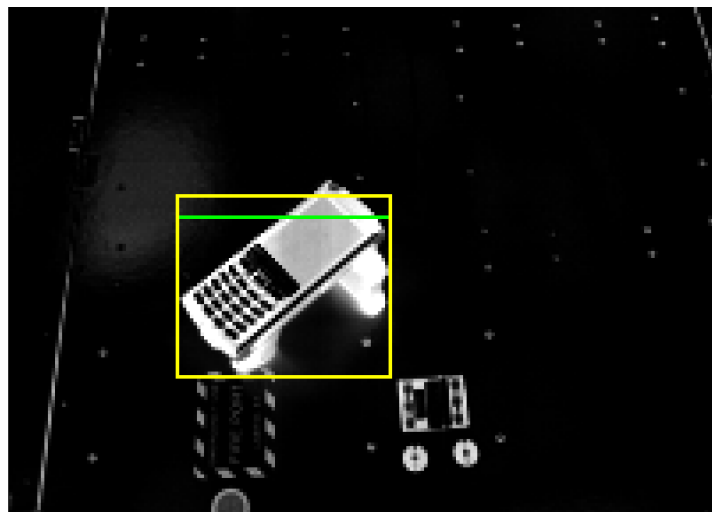


Figure 5 – The rectangle detected by “FindRect” (green) and the refined version (yellow).

- Main Loop:** the second loop uses all remaining time. It has 3 subdivisions: the first one runs until 80% of the time, and tests random positions that contains at least 90% of the ROI rectangle (shown in yellow in Figure 5). This verification is done by translating, rotating and projecting in 2D only the corners of object's 3D bounding box. Also positions that 3D coordinates are outside the "expected" bounds (taken from examples ground truth) are discarded. This pruning is crucial: in the case of the image shown in Figure 5, about 4.5M positions are discarded, while only 90K are evaluated. In the next two phases, instead of using random positions, it uses "neighbor" positions, by translating and rotating by a small value the best position found so far. The third (and last) phase uses an additional step inside the evaluation function, to make it more precise (described later).
- eval1:** this is another key piece of the solution, used in the two loops. It returns a score for a given position. Ideally, better (closer to the ground truth) positions would produce greater values. First it "renders" the model points, adding a border to each point, in order to cover all object, filling the empty space between points that can be observed in Figure 2. Then it compares covered pixels with the "difference" image (Figure 4). Covered pixels should match pixels with high difference values. If it covers a pixel with a low value (which likely doesn't belong to the object), score is penalized. High "difference" values that weren't covered also decrease the score, because the current evaluated position missed them. Figure 6 shows an evaluation of the final position (a successful example), in which blue pixels (model pixels, with a border) covered the high "difference" pixels (and the object). Figure 7 shows only the pixels removed from the evaluation in the third phase of main loop. This is done because the border added to each model point made the considered area bigger than the desired one, as an effect of adding extra pixels close to boundary pixels. Removing them allows a better match, but as this process is slow (at least the way I implemented), it is used only in the last phase.

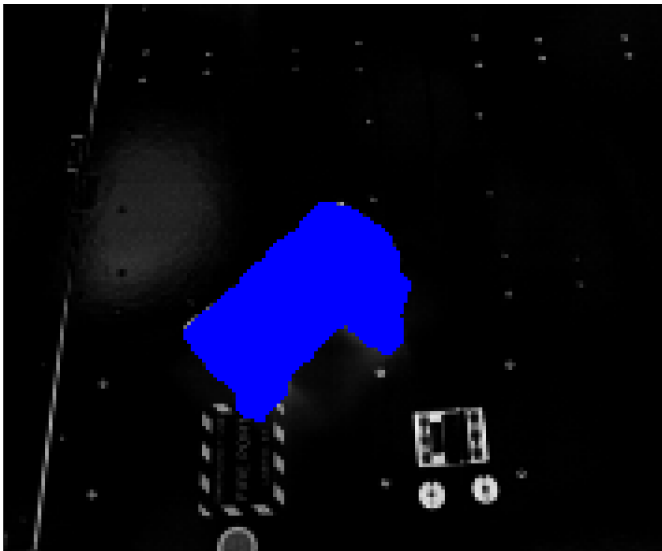


Figure 6 – A successful evaluation.

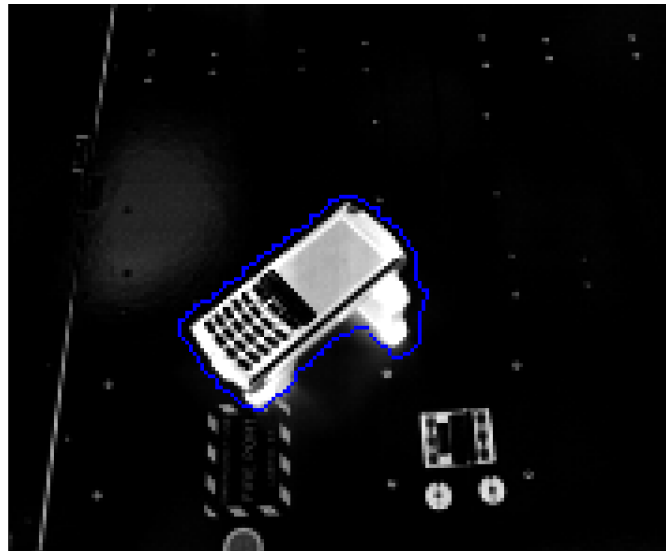


Figure 7 – In blue, pixels discard when a more precise evaluation is used.

- **randomPos**: called by other methods, to generate a truly random position, or a random small change from a given position. This was implemented in a simple way, but it seems to work well enough. Instead of using quaternions, it uses 3 rotation angles (in a way that was easy to generate neighbor rotations). I guess positions generated this way (with 3 rotating angles) are not distributed equally, but didn't have time to figure out a better way of doing this.

FAILED APPROACHES

- I tried to use training images in two different ways: the first one extracted color and texture information from object's pixels, built a random forest to classify pixels from testing images into belong or not to the object. This would be an alternative solution for the "difference" step that detects the object. I guess it didn't work at all because of position, background and lighting conditions variations.
- The second attempt was to use training images to extract the colors of each model point. I spent a lot of time in this path, and the results looked good (two examples in the following figures). The idea was to compare the colored model with the given images (inside the evaluation step, that compares a single on/off color, with difference levels). For some reason, it didn't work well, and I had to abandon this idea.

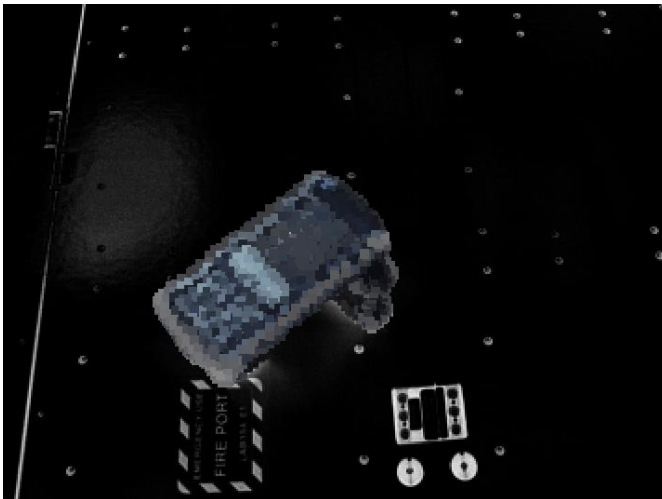


Figure 8 – An evaluation result with colored model points (extracted from training images).

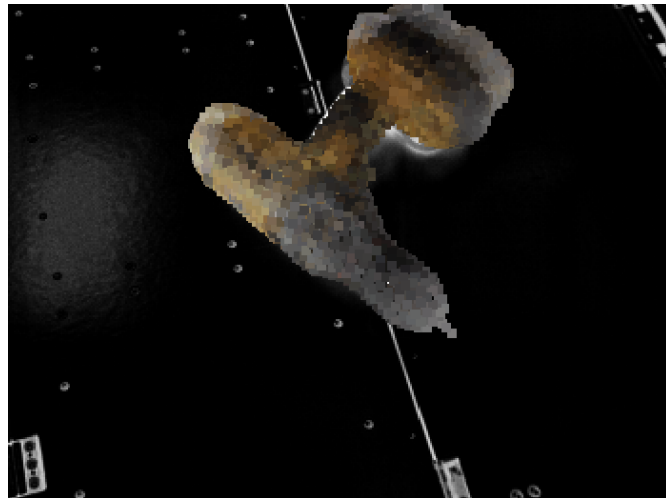


Figure 9 – Same as Figure 8, for the Drill tool.

- Another promising idea was adding a condition that objects must lay in a plain surface. This would reduce a lot the number of possible positions, at least for the given objects. I wasn't able to find an easy (and correct way) to check this.

UNEXPLORED APPROACHES

- Work with polygons (faces) instead of isolated points. That would require more processing time for each attempt, but would give more precision.
- Get information (e.g. color) from testing images (not training) and use in the remaining images.
- Include some kind of comparison between left and right pixels of the same projected model point in the evaluation score.
- Use open source code to detect region of interest.