Finding a Planar Object in OpenCV
Approach

• Assume that we have a “reference image” of a planar object, and we want to find the object in a new image

• Approach:
  – Find feature points and their descriptors in the reference image
  – Calculate the 3D locations of the feature points
  – Then, for each new image:
    • Find feature points and their descriptors in the new image
    • Match the descriptors to get a set of tentative matched points
    • Fit a homography (projective transform) to the points, using RANSAC
    • Using the inlier points, find the pose of the object

Example result: http://inside.mines.edu/~whoff/courses/EENG512/lectures/data/Gonzalez.wmv
Reference Image

• Take an image of a planar object at a known height above the object

• Using the camera’s focal length, and similar triangles, calculate the 3D locations of points on the plane

\[
\frac{f}{x} = \frac{h}{X}
\]
SURF

- SURF: Speeded Up Robust Features
- Similar to SIFT – scale and rotation invariant
- We create a “SURF” object and initialize its parameters (see http://docs.opencv.org)

```cpp
cv::_Ptr<cv::Feature2D> f2d = cv::xfeatures2d::SURF::create(1000.0, // threshold (default = 100.0)
4, // number of octaves (default=4)
2, // number of octave layers within each octave (default=2)
true, // true=use 128 element descriptors, false=use 64 element
false); // true=don't compute orientation, false=compute orientation
```

Keypoints

• We detect “keypoints”
  – Each keypoint has location, size, angle, etc

  // Detect keypoints in the input image using the SURF detector.
  std::vector<cv::KeyPoint> keypoints;
  mySurf.detect(imageInput, keypoints);
  printf("Detected %d points in image.\n", keypoints.size());

• We can draw keypoints on the image:

  // Draw keypoints on the image and display it.
  drawKeypoints(imageInput, keypoints, imageOutput,
                cv::Scalar(0, 0, 255), cv::DrawMatchesFlags::DEFAULT);
  cv::imshow("image", imageOutput);

• Next, extract descriptors for all keypoints
Matching points to new image

• “Brute force” matcher – computes the distance between each pair of descriptors
• We find the closest k=2 matches, so that we can apply the “ratio” test (i.e., keep a match only if it is significantly better than the 2\textsuperscript{nd} best match)

// Match descriptors between reference and new image.
// For each, find the k nearest neighbors.
cv::BFMatcher matcher(cv::NORM_L2);
std::vector<std::vector<cv::DMatch>> knnMatches;
matcher.knnMatch(
    descriptors2, // These are the "query" descriptors, in the new image
    descriptors1, // These are the "training" descriptors, from reference image
    knnMatches,   // Output matches
    2);            // Value of k (we will find the best k matches)
Fitting a homography

- Fit a homography transform to the tentative matches
- Use RANSAC to eliminate outliers
- Return a mask to show which points are inliers
  - i.e., inliersMask[i] is true if point i is an inlier

```cpp
// Find homography matrix and get the inliers mask.
std::vector<unsigned char> inliersMask(pts1.size());
cv::Mat homography = cv::findHomography(pts1, pts2,
                      cv::FM_RANSAC,
                      5, // Allowed reprojection error in pixels (default=3)
                      inliersMask);
```
Finding the pose

- Use the function “solvePnP” (i.e., solve the perspective n-point problem)
- Pass in corresponding 3D points and 2D points

```cpp
// Ok, now we have point correspondences from the new (incoming) image, to
// the 3D points on the model. Find the pose using "solvePnP". The
// resulting pose is "model-to-camera".
cv::Mat rotVec, transVec;
bool foundPose = cv::solvePnP(p3, p2,
    K, // intrinsic camera parameter matrix
    cv::Mat::zeros(5, 1, CV_64F), // distortion coefficients
    rotVec, transVec); // output rotation and translation
```