Computer Vision

Professor William Hoff
Dept of Electrical Engineering & Computer Science
http://inside.mines.edu/~whoff/
OpenCV Examples: Detection and Tracking
Reference Image

• First 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

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

• 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](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 elementfalse); // true=don't compute orientation, false=compute orientation```

Keypoints

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

```cpp
// 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:

```cpp
// 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 2nd best match)

```cpp
// 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

```c++
// 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
```
Tracking
Tracking vs Detecting

• The example simply *detected* the object in each new image, without using any information from previous images
  – Detecting SURF keypoints and computing their descriptors is relatively slow

• If we *track* the object from frame to frame, we can speed up the processing
  – Instead of detecting complex features separately in both images, and then matching them, we just detect simple features in the first image, and then track the features to the 2nd image
Interest Points

• Instead of using SURF or SIFT, we will use Shi-Tomasi “interest points”, and track them using the Lucas-Kanade optical flow method

```cpp
std::vector<cv::Point2f> pts2D;
   cv::goodFeaturesToTrack(
      imageInputGray, // input image
      pts2D, // output points
      MAXPTS, // maximum number of corners to find
      0.01, // minimum quality level
      MINDIST, // minimum allowed distance between points
      cv::Mat(), // detect where mask != 0
      2 * ISIZE + 1); // blocksize over which to compute derivatives
```
Lucas-Kanade Optical Flow Tracking

```cpp
std::vector<unsigned char> status;
std::vector<float> err;
cv::Size winSize(21, 21);
cv::TermCriteria criteria(
    cv::TermCriteria::COUNT | cv::TermCriteria::EPS,
    30, // terminate after this many iterations, or
    0.01); // when the search window moves by less than this

calcOpticalFlowPyrLK(
    imageInputGrayPrevious, // previous image
    imageInputGray, // next image
    pts2D, // points in previous image
    pts2DNew, // output tracked points
    status, // output status vector (1 = match found)
    err, // output vector of errors
    winSize, // size of the search window at each pyramid level
    3, // use up to maxLevel levels in pyramid
    criteria, // termination criteria
    0, // flags
    0.001); // minEigThreshold
```
Adding More Points

• If the number of tracked points falls below a threshold, detect some more

• Don’t detect new points that are too close to existing points

• We’ll use the “mask” input to “goodFeaturesToTrack”, to keep it from detecting points that are close to existing ones
Adding New Points

• When we add new 2D points, we also need to estimate their 3D locations, assuming that they lie on the same plane.
A Note on Feature Drift

• Our program uses image templates from the previous image to track to the new image
  – The templates are constantly being updated
  – It allows large changes in appearance (as long as the changes happen slowly)
  – This means that the location of the point may drift over time

• For better accuracy, you should track from the original (reference) image to the current image
  – Need to allow for appearance changes (e.g., affine)