Colorado School of Mines

Computer Vision

Professor William Hoff
Dept of Electrical Engineering & Computer Science
http://inside.mines.edu/~whoff/
Tracking a Planar Object in OpenCV
Previous Lecture on Detecting Planar Objects

• 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](http://inside.mines.edu/~whoff/courses/EENG512/lectures/data/Gonzalez.wmv)
Tracking vs Detecting

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

• If we *track* the object from frame to frame, we can speed up the processing
  — Instead of detecting features separately in both images, and then matching them, we just detect 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

• We covered interest point detection in Lecture 11

```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
```
Run Program “main_v1”

• This program finds interest points in each image
• What frame rate (#frames per second) do you get?

• Detecting interest points in each image is relatively expensive
• A better approach is to track them from frame to frame (using the Lucas-Kanade method) and only add new points if you need them
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
if (pts2D.size() < 0.9*MAXPTS) {

    // Try to add more points ... set mask=1 where we should detect
    cv::Mat mask(imageHeight, imageWidth, CV_8UC1, cv::Scalar(0));

    // Don't detect points too close to borders
    cv::Mat maskROI = mask(cv::Rect(10, 10, imageWidth - 20, imageHeight - 20));
    maskROI = 255;

    // Don't detect too close to the existing points
    for (unsigned int i = 0; i < pts2D.size(); i++) {
        int row = (int)pts2D[i].y;
        int col = (int)pts2D[i].x;
        row = (row < 0 ? 0 : row);
        row = (row >= imageHeight ? imageHeight - 1 : row);
        col = (col < 0 ? 0 : col);
        col = (col >= imageWidth ? imageWidth - 1 : col);
        mask.at<unsigned char>(row, col) = 0;
    }

    // Erode to expand the area of zeros
    cv::Mat strElmt(2 * MINDIST, 2 * MINDIST, CV_8UC1, cv::Scalar(255));
    cv::erode(mask, mask, strElmt);

    // Detect more points
    std::vector<cv::Point2f> morePts2D;
    cv::goodFeaturesToTrack(
        imageInputGray, // input image
        morePts2D, // output points
        MAXPTS, // maximum number of corners to find
        0.01, // minimum quality level
        MINDIST, // minimum allowed distance between points
        mask, // detect where mask != 0
        2 * ISIZE + 1); // blocksize over which to compute derivatives

    for (unsigned int i = 0; i < morePts2D.size(); i++) {
        pts2D.push_back(morePts2D[i]);
        if (pts2D.size() >= MAXPTS) break;
    }
}
Run Program “main_v2”

• This program finds and tracks interest points, and adds more if necessary
• What frame rate (#frames per second) do you get?
Homography

• Some points may be tracked incorrectly
  – Look at the performance of the previous program if you point the camera at a relatively featureless scene

• If we assume that all points lie on a plane, we can enforce the constraint that all point correspondences (from one image to the next) must satisfy a homography

• Keep only the “inlier” points (i.e., the ones that satisfy the homography)
// Fit a homography to the matched points, to find inliers.
if (pts2D.size() < 4) { // Need >=4 points for homography
    printf("Not enough points (%d) to fit a homography\n", pts2D.size());
    continue;
}

// Find homography matrix and get the inliers mask.
std::vector<unsigned char> inliersMask(pts2D.size());
cv::Mat homography = cv::findHomography(pts2D, pts2DNew,
    cv::FM_RANSAC,
    3, // Allowed reprojection error in pixels (default=3)
inliersMask);

// Keep only the inliers.
pts2D.clear();
for (size_t i = 0; i < inliersMask.size(); i++)
    if (inliersMask[i]) {
        pts2D.push_back(pts2DNew[i]);
    }
Run Program “main_v3”

• This program finds and tracks interest points (adding more if necessary), and enforces that all tracked points must fit a homography

• Compare the previous program with this one, if you point the camera at a relatively featureless surface
Computing Pose

• We 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

\[
\frac{f}{x} = \frac{h}{X}
\]
Finding the pose

• Track the image points to the new frame
• 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
```
Run Program “main_v4”

• You will also have to add “drawPose.cpp” to your Visual Studio project
• This program computes the pose from 2D to 3D point correspondences
  – It does not try to add any new points, though
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.
// Estimate corresponding 3D point locations for the new points.
for (unsigned int i = 0; i < morePts2D.size(); i++) {
    // Get unit vector in the direction of this point (in camera coords).
    cv::Mat u_c = (cv::Mat_<double>(3, 1) << pNorm[i].x, pNorm[i].y, 1.0f);
    u_c = u_c / norm(u_c);

    // Rotate the unit vector to convert it to world coordinates.
    cv::Mat u_w = XXXXXXXXXXXXXX

    // Now, the normalized point represents a ray in space, starting from Pc_w,
    // and extending out to infinity along the direction u_w. So a
    // point along the ray is P_w = s*u_w + Pc_w. We want to intersect
    // this ray with the plane Z=0; i.e., 0 = s*u_w.z + Pc_w.z.
    float s = -Pc_w.at<double>(2) / u_w.at<double>(2); // Solve for s = distance
    if (s < 0) continue; // Ray does not intersect plane

    // Get the intersection of the ray with the plane Z=0.
    cv::Mat P_w = XXXXXXXXXXXXXX

    pts2D.push_back(morePts2D[i]);
    pts3D.push_back(cv::Point3f(P_w));
    if (pts2D.size() >= MAXPTS) break;
}
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)