Colorado School of Mines

Computer Vision

Professor William Hoff
Dept of Electrical Engineering & Computer Science
http://inside.mines.edu/~whoff/
Detecting Square Markers using OpenCV
Square Fiducial Markers

• Square markers are popular in augmented reality applications:
  – They are easy to detect
  – The pose of the marker can be estimated
  – Each marker can have a unique id

• There can be hundreds or thousands of different unique id’s.

ArUco Markers

• You can create a “dictionary” of markers for your application, specifying
  – Number of bits (e.g., 4x4, 5x5, 6x6, 7x7)
  – The number of markers in the dictionary (e.g., 50, 100, 250, 1000)
• The “further apart” the markers are (in terms of the number of bits that differ), the more error correction can be done
Marker Recognition

• First detect the black square:
  – Threshold the image (using global or adaptive)
  – Find contours around all black regions
  – Approximate the contours by line segments; keep only contours that have exactly four line segments ... this is a candidate square marker
  – Transform the image of the candidate marker to an “orthophoto”

• Reading the id
  – Threshold the orthophoto
  – Divide into a N x N grid, determine if each cell is 0 or 1
  – Attempt to find that pattern in the dictionary (or one that is close)
Useful OpenCV Function

• **approxPolyDP**

```cpp
void cv::approxPolyDP(
    InputArray curve,
    OutputArray approxCurve,
    double epsilon,
    bool closed)
```

– Approximates a polygonal curve(s) with the specified precision.

• **Example:**

```cpp
std::vector<cv::Point> approxCurve;
// Max allowed distance between original curve and its approximation.
double eps = contours[i].size() * 0.01;
cv::approxPolyDP(contours[i], approxCurve, eps, true);
```
#include <opencv2/opencv.hpp>

// This function tries to find black squares in the image.
// It returns a vector of squares, where each square is represented
// as a vector of four points, arranged in counter clockwise order.
std::vector<std::vector<cv::Point2f>> findSquares(cv::Mat imageInput)
{
    // Convert to gray if input is color.
    cv::Mat imageInputGray;
    if (imageInput.channels() == 3)
    {
        cv::cvtColor(imageInput, imageInputGray, cv::COLOR_BGR2GRAY);
    } else
    {
        imageInputGray = imageInput;
    }

    // Do adaptive threshold ... this compares each pixel to a local
    // mean of the neighborhood. The result is a binary image, where
    // dark areas of the original image are now white (1's).
    cv::Mat imageThresh;
    adaptiveThreshold(imageInputGray, imageThresh,
                      255, // output thresholded image
                      cv::ADAPTIVE_THRESH_GAUSSIAN_C, // local neighborhood
                      cv::THRESH_BINARY_INV, // threshold_type - invert
                      1,  // blockSize (any large number)
                      0); // a constant to subtract from mean

    // Apply morphological operations to get rid of small (noise) regions
    cv::Mat structuringElmt = cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(3, 3));
    cv::Mat imageOpen;
    morphologyEx(imageThresh, imageOpen, cv::MORPH_OPEN, structuringElmt);
    cv::Mat imageClose;
    morphologyEx(imageOpen, imageClose, cv::MORPH_CLOSE, structuringElmt);

This is a function called “findSquares” which finds candidate squares in the image (1 of 3)
// Find contours
std::vector<std::vector<cv::Point>> contours;
std::vector<cv::Vec4i> hierarchy;
cv::findContours(
    imageClose,  // input image (is destroyed)
    contours,    // output vector of contours
    hierarchy,   // hierarchical representation
    CV_RETR_CCOMP,  // retrieve all contours
    CV_CHAIN_APPROX_NONE);  // all pixels of each contours

// Iterate through all the top-level contours and find squares.
std::vector<std::vector<cv::Point2f>> squares;  // output squares

for (int i = 0; i < (int)contours.size(); i++) {
    // Contour should be greater than some minimum area
    double a = contourArea(contours[i]);
    if (!(a > 100)) continue;

    // Reject the ith contour if it doesn’t have a child inside.
    if (hierarchy[i][2] < 0) continue;

    ///// Check the ratio of area to perimeter squared:  R = 16*A/P^2.
    ///// R is equal to 1 for a square.
    //double P = arcLength(contours[i], true);
    //double A = contourArea(contours[i]);
    //if (16 * A / (P*P) < 0.75)
    //    continue;

    // Approximate contour by a polygon.
    std::vector<cv::Point> approxCurve;
    // Maximum allowed distance between the original curve and its approximation.
    double eps = contours[i].size() * 0.01;
    cv::approxPolyDP(contours[i], approxCurve, eps, true);

    // We interested only in polygons that contain only four points.
    if (approxCurve.size() != 4) continue;

This is a function called “findSquares” which finds candidate squares in the image (2 of 3)
// Ok, I think we have a square! Create the list of corner points.
std::vector<cv::Point2f> squareCorners;
for (int j = 0; j < 4; j++)
    squareCorners.push_back(cv::Point2f(approxCurve[j]));

// Sort the points in counter-clockwise order. Trace a line between the
// first and second point. If the third point is on the right side, then
// the points are anticlockwise.
cv::Point v1 = squareCorners[1] - squareCorners[0];
cv::Point v2 = squareCorners[2] - squareCorners[0];
double o = (v1.x * v2.y) - (v1.y * v2.x);
if (o < 0.0)
    std::swap(squareCorners[1], squareCorners[3]);

// Store the square in our list of squares.
squares.push_back(squareCorners);
}

return squares;
}
Transform Marker to “Orthophoto”

• Find the projective transform (homography) that transforms the subimage of the marker to an “orthophoto”
  – Use OpenCV functions “getPerspectiveTransform” and “warpPerspective”

• Example:

```cpp
// Create a list of "ortho" square corner points.
std::vector<cv::Point2f> squareOrtho;
squareOrtho.push_back(cv::Point2f(0, 0));
squareOrtho.push_back(cv::Point2f(120, 0));
squareOrtho.push_back(cv::Point2f(120, 120));
squareOrtho.push_back(cv::Point2f(0, 120));

// Find the perspective transformation that brings current marker to rectangular form.
cv::Mat H = cv::getPerspectiveTransform(squareCorners, squareOrtho);

// Transform image to get an orthophoto square image.
cv::Mat imageSquare;
cv::Size imageSquareSize(120, 120);
cv::warpPerspective(imageInputGray, imageSquare, H, imageSquareSize);
```
Main program to find candidate squares

/* Detect squares in an image. */
#include <iostream>
#include <windows.h>  // For Sleep()
#include <opencv2/opencv.hpp>

// Function prototypes.
std::vector<std::vector<cv::Point2f>> findSquares(cv::Mat imageInput);

int main(int argc, char* argv[]) {
    printf("Hit ESC key to quit\n");

    cv::VideoCapture cap(1);  // open the camera
    //cv::VideoCapture cap("myVideo.avi");  // open the video file
    if (!cap.isOpened()) {  // check if we succeeded
        printf("error - can't open the camera or video; hit any key to quit\n");
        system("PAUSE");
        return EXIT_FAILURE;
    }

    while (true) {
        cv::Mat imageInput;
        cap >> imageInput;
        if (imageInput.empty()) break;

        cv::Mat imageInputGray;
        cv::cvtColor(imageInput, imageInputGray, cv::COLOR_BGR2GRAY);
    }
}
Main program (continued)

```cpp
std::vector<std::vector<cv::Point2f>> squares;
squares = findSquares(imageInputGray);
printf("Number of possible squares found = %d\n", squares.size());

if (squares.size() == 0) {
    // Didn't find any squares. Just display the image.
    cv::imshow("My Image", imageInput);
    if (cv::waitKey(1) == 27) break; // hit ESC (ascii code 27) to quit

    // Continue to the next iteration of the main loop.
    continue;
}

for (unsigned int iSquare = 0; iSquare < squares.size(); iSquare++) {
    std::vector<cv::Point2f> squareCorners = squares[iSquare];

    // Draw square as a sequence of line segments.
    cv::Scalar color = cv::Scalar(0, 255, 0);
    for (int j = 0; j < 4; j++) {
        cv::Point p1 = squareCorners[j];
        cv::Point p2 = squareCorners[(j + 1) % 4];
        cv::line(imageInput, p1, p2, color,
                 2, // thickness
                 8); // line connectivity
    }
```
// Create a list of "ortho" square corner points.
std::vector<cv::Point2f> squareOrtho;
squareOrtho.push_back(cv::Point2f(0, 0));
squareOrtho.push_back(cv::Point2f(120, 0));
squareOrtho.push_back(cv::Point2f(120, 120));
squareOrtho.push_back(cv::Point2f(0, 120));

// Find the perspective transformation that brings current marker to rectangular form.
cv::Mat H = cv::getPerspectiveTransform(squareCorners, squareOrtho);

// Transform image to get an orthophoto square image.
cv::Mat imageSquare;
cv::Size imageSquareSize(120, 120);
cv::warpPerspective(imageInputGray, imageSquare, H, imageSquareSize);
cv::imshow("Marker", imageSquare);
}

// Show the image.
cv::imshow("My Image", imageInput);

// Wait for xx ms (0 means wait until a keypress)
if (cv::waitKey(1) == 27) break; // hit ESC (ascii code 27) to quit

return EXIT_SUCCESS;
}
ArUco Library

• OpenCV has a library module for ArUco marker detection
• Key functions, described on following slides (see documentation for full details):
  – getPredefinedDictionary
  – detectMarkers
  – drawDetectedMarkers
  – estimatePoseSingleMarkers
  – drawAxis
getPredefinedDictionary

Ptr<Dictionary>
cv::aruco::getPredefinedDictionary ( 
    PREDEFINED_DICTIONARY_NAME name )

• Returns one of the predefined dictionaries defined in
  PREDEFINED_DICTIONARY_NAME

• Dictionary name indicates:
  – Size of marker (e.g, 4x4 bits)
  – Number of markers in dictionary (e.g., 100)

<table>
<thead>
<tr>
<th>DICT_4X4_50</th>
<th>DICT_4X4_100</th>
<th>DICT_4X4_250</th>
<th>DICT_4X4_1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>DICT_5X5_50</td>
<td>DICT_5X5_100</td>
<td>DICT_5X5_250</td>
<td>DICT_5X5_1000</td>
</tr>
<tr>
<td>DICT_6X6_50</td>
<td>DICT_6X6_100</td>
<td>DICT_6X6_250</td>
<td>DICT_6X6_1000</td>
</tr>
<tr>
<td>DICT_7X7_50</td>
<td>DICT_7X7_100</td>
<td>DICT_7X7_250</td>
<td>DICT_7X7_1000</td>
</tr>
</tbody>
</table>
detectMarkers

```cpp
void cv::aruco::detectMarkers ( InputArray image,
    const Ptr< Dictionary > & dictionary,
    OutputArrayOfArrays corners,
    OutputArray ids,
    parameters =
    const Ptr< DetectorParameters > & DetectorParameters::create(),
    OutputArrayOfArrays rejectedImgPoints = noArray()
)
```

Basic marker detection.

**Parameters**

- **image**
  - input image

- **dictionary**
  - indicates the type of markers that will be searched

- **corners**
  - vector of detected marker corners. For each marker, its four corners are provided, (e.g. std::vector< std::vector< cv::Point2f > > ). For N detected markers, the dimensions of this array is Nx4. The order of the corners is clockwise.

- **ids**
  - vector of identifiers of the detected markers. The identifier is of type int (e.g. std::vector<int>). For N detected markers, the size of ids is also N. The identifiers have the same order than the markers in the imgPoints array.

- **parameters**
  - marker detection parameters

- **rejectedImgPoints**
  - contains the imgPoints of those squares whose inner code has not a correct codification. Useful for debugging purposes.

Performs marker detection in the input image. Only markers included in the specific dictionary are searched. For each detected marker, it returns the 2D position of its corner in the image and its corresponding identifier. Note that this function does not perform pose estimation.
drawDetectedMarkers

```c
void cv::aruco::drawDetectedMarkers ( InputOutputArray image,
                                   InputArrayOfArrays corners,
                                   InputArray ids = noArray(),
                                   Scalar borderColor = Scalar(0, 255, 0)
)
```

Draw detected markers in image.

**Parameters**

- **image**: input/output image. It must have 1 or 3 channels. The number of channels is not altered.
- **corners**: positions of marker corners on input image. (e.g std::vector<std::vector<cv::Point2f> > ). For N detected markers, the dimensions of this array should be Nx4. The order of the corners should be clockwise.
- **ids**: vector of identifiers for markers in markersCorners. Optional, if not provided, ids are not painted.
- **borderColor**: color of marker borders. Rest of colors (text color and first corner color) are calculated based on this one to improve visualization.

Given an array of detected marker corners and its corresponding ids, this functions draws the markers in the image. The marker borders are painted and the markers identifiers if provided. Useful for debugging purposes.
void cv::aruco::estimatePoseSingleMarkers ( InputArrayOfArrays corners,
    float markerLength,
    InputArray cameraMatrix,
    InputArray distCoeffs,
    OutputArray rvecs,
    OutputArray tvecs )

Pose estimation for single markers.

Parameters

 corners vector of already detected markers corners. For each marker, its four corners are provided, (e.g std::vector<std::vector<cv::Point2f>> ).

For N detected markers, the dimensions of this array should be Nx4. The order of the corners should be clockwise.

See also

detectMarkers

Parameters

 markerLength the length of the markers’ side. The returning translation vectors will be in the same unit. Normally, unit is meters.

cameraMatrix

input 3x3 floating-point camera matrix \( A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \)

distCoeffs

vector of distortion coefficients \((k_1, k_2, p_1, p_2, k_3, [k_4, k_5, k_6], [s_1, s_2, s_3, s_4])\) of 4, 5, 8 or 12 elements

rvecs

array of output rotation vectors

See also

Rodrigues (e.g. std::vector<cv::Vec3d>). Each element in rvecs corresponds to the specific marker in imgPoints.

Parameters

tvecs array of output translation vectors (e.g. std::vector<cv::Vec3d>). Each element in tvecs corresponds to the specific marker in imgPoints.
void cv::aruco::drawAxis ( InputOutputArray image,
    InputArray cameraMatrix,
    InputArray distCoeffs,
    InputArray rvec,
    InputArray tvec,
    float length
)

Draw coordinate system axis from pose estimation.

Parameters

- **image**: input/output image. It must have 1 or 3 channels. The number of channels is not altered.
- **cameraMatrix**: input 3x3 floating-point camera matrix \( \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \)
- **distCoeffs**: vector of distortion coefficients \( (k_1, k_2, p_1, p_2, [k_3, k_4, k_5, k_6], [s_1, s_2, s_3, s_4]) \) of 4, 5, 8 or 12 elements
- **rvec**: rotation vector of the coordinate system that will be drawn.
- **tvec**: translation vector of the coordinate system that will be drawn.
- **length**: length of the painted axis in the same unit than tvec (usually in meters)

See also

Rodrigues

Parameters

- **tvec**: translation vector of the coordinate system that will be drawn.
- **length**: length of the painted axis in the same unit than tvec (usually in meters)

Given the pose estimation of a marker or board, this function draws the axis of the world coordinate system, i.e. the system centered on the marker/board. Useful for debugging purposes.
#include <iostream>
#include <opencv2/opencv.hpp>
#include <opencv2/aruco.hpp>
#include <windows.h>  // For Sleep()

// Length of one side of a square marker.
const float markerLength = 2.0;

int main(int argc, char* argv[]) {
    printf("This program detects ArUco markers.\n");
    printf("Hit the ESC key to quit.\n");

    // Camera intrinsic matrix (fill in your actual values here).
    double K_[3][3] =
        { { 675, 0, 320 },
         { 0, 675, 240 },
         { 0, 0, 1 } };
    cv::Mat K = cv::Mat(3, 3, CV_64F, K_).clone();

    // Distortion coeffs (fill in your actual values here).
    double dist_[4] = { 0, 0, 0, 0, 0, 0, 0, 0 };  // fill in your actual values here
    cv::Mat distCoeffs = cv::Mat(5, 1, CV_64F, dist_).clone();

    cv::VideoCapture cap(0);  // open the camera
    // cv::VideoCapture cap("video.avi");  // or open the video file

    if (!cap.isOpened()) {  // check if we succeeded
        printf("error - can't open the camera or video; hit any key to quit\n");
        system("PAUSE");
        return EXIT_FAILURE;
    }

    // Let's just see what the image size is from this camera or file.
    double WIDTH = cap.get(CV_CAP_PROP_FRAME_WIDTH);
    double HEIGHT = cap.get(CV_CAP_PROP_FRAME_HEIGHT);
    printf("Image width=%f, height=%f\n", WIDTH, HEIGHT);
// Allocate image.
cv::Mat image;
cv::Ptr<cv::aruco::Dictionary> dictionary = cv::aruco::getPredefinedDictionary(cv::aruco::DICT_4X4_100);
cv::Ptr<cv::aruco::DetectorParameters> detectorParams = cv::aruco::DetectorParameters::create();

// Run an infinite loop until user hits the ESC key.
while (1) {
    cap >> image;  // get image from camera
    if (image.empty()) break;

    std::vector<int> markerIds;
    std::vector<std::vector<cv::Point2f>> markerCorners, rejectedCandidates;
    cv::aruco::detectMarkers(
        image,  // input image
        dictionary,  // type of markers that will be searched for
        markerCorners,  // output vector of marker corners
        markerIds,  // detected marker IDs
        detectorParams,  // algorithm parameters
        rejectedCandidates);

    if (markerIds.size() > 0) {
        // Draw all detected markers.
        cv::aruco::drawDetectedMarkers(image, markerCorners, markerIds);

        std::vector<cv::Vec3d> rvecs, tvecs;
        cv::aruco::estimatePoseSingleMarkers(
            markerCorners,  // vector of already detected markers corners
            markerLength,  // length of the marker's side
            K,  // input 3x3 floating-point intrinsic camera matrix K
            distCoeffs,  // vector of distortion coefficients of 4, 5, 8 or 12 elements
            rvecs,  // array of output rotation vectors
            tvecs);  // array of output translation vectors
    }
}
// Display pose for the detected marker with id=0.
for (unsigned int i = 0; i < markerIds.size(); i++) {
  if (markerIds[i] == 0) {
    cv::Vec3d r = rvecs[i];
    cv::Vec3d t = tvecs[i];

    // Draw coordinate axes.
    cv::aruco::drawAxis(image, K, distCoeffs, r, t, 0.5*markerLength); // marker pose
    cv::drawMarker(image, p[0], cv::Scalar(0, 255, 255), cv::MARKER_STAR, 2); // thickness
  }
}

cv::imshow("Image", image); // show image

// Wait for x ms (0 means wait until a keypress).
// Returns -1 if no key is hit.
char key = cv::waitKey(1);
if (key == 27) break; // ESC is ascii 27
return EXIT_SUCCESS;
Extra Stuff

- This function converts the representation of an orientation from angle-axis to XYZ angles (see the lecture slides on 3D-3D transformations)

```cpp
// A function to convert the representation of an orientation from angle-axis to XYZ angles.
std::vector<double> calcVec(cv::Vec3d rvec)
{
    cv::Mat R;
    cv::Rodrigues(rvec, R); // convert to rotation matrix

    double ax, ay, az;
    ay = atan2(-R.at<double>(2, 0),
                pow(pow(R.at<double>(0, 0), 2) + pow(R.at<double>(1, 0), 2), 0.5));
    double cy = cos(ay);
    if (abs(cy) < 1e-9) {
        // Degenerate solution.
        az = 0.0;
        ax = atan2(R.at<double>(0, 1), R.at<double>(1, 1));
        if (ay < 0) ax = -ax;
    } else {
        az = atan2(R.at<double>(1, 0) / cy, R.at<double>(0, 0) / cy);
        ax = atan2(R.at<double>(2, 1) / cy, R.at<double>(2, 2) / cy);
    }

    std::vector<double> result;
    result.push_back(ax);
    result.push_back(ay);
    result.push_back(az);
    return result;
}
```
This code snippet will print the pose (as a text string) onto an image.

```cpp
// Assume that r,t are the rotation vector and the translation; e.g.
// from solvePnP or ArUco's estimatePoseSingleMarkers.
std::vector<double> a = calcVec(r);
std::string label =
  "aX=" + std::to_string(a[0]).substr(0, 5) +
  "aY=" + std::to_string(a[1]).substr(0, 5) +
  "aZ=" + std::to_string(a[2]).substr(0, 5) +
  "tX=" + std::to_string(t[0]).substr(0, 5) +
  "tY=" + std::to_string(t[1]).substr(0, 5) +
  "tZ=" + std::to_string(t[2]).substr(0, 5);

int baseline = 0;
cv::Size textSize = getTextSize(label,
  cv::FONT_HERSHEY_PLAIN, // font face
  1.0, // font scale
  1,   // thickness
  &baseline);

// Draw the background rectangle.
cv::rectangle(image,
  cv::Point(10, 30),  // lower left corner
  cv::Point(textSize.width + 10, 8), // upper right corner
  cv::Scalar(255, 255, 255), // color
  CV_FILLED); // Fill the rectangle

putText(image, label, cv::Point(10, 25),
  cv::FONT_HERSHEY_PLAIN, // font face
  1.0, // font scale
  cv::Scalar(0, 0, 0), // font color
  1); // thickness
```