ARTags
ARTags

• ARTag is a fiducial marker system consisting of a 2D bit pattern.
• Computer vision methods can:
  – Read the pattern id
  – Determine the pose of the marker with respect to the camera
• There are thousands of different unique id’s.

• Each pattern consists of a 10x10 square grid.
• The border is 2 cells wide, and is all black.
• The interior region is a 6x6 grid of black or white cells.
From the ARTag paper

• The algorithm for detection first locates quadrilaterals which are perspective views of the marker border, then the interior is sampled into 36 binary ’1’ or ’0’ symbols.
• Some of the bits carry data; others are used as checksums.
• No need to store patterns, since a pattern can be tested for validity.
• The probability of falsely identifying one marker for another, or a piece of the background as a marker, is less than 0.0039%.
Our Algorithm to Find Black Squares

- Threshold the image
- Find contours of the black and white regions
- Keep only contours around black regions, that contain a white region
  
  Similar to algorithm to find CCCs

- Approximate the contour by a polygon
- Keep only contours that have four segments
- Sort the vertices in counter-clockwise order
  
  New
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 a black square in the image.
std::vector<cv::Point2f> findSquare(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,
                      cv::ADAPTIVE_THRESH_GAUSSIAN_C, cv::THRESH_BINARY_INV,
                      cv::ADAPTIVE_THRESH_GAUSSIAN_C, cv::THRESH_BINARY_INV,
                      31, 0);

    // 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);
// 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<cv::Point2f> squareCorners;  // this will hold corners
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;

    // 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;
// Ok, I think we have a square! Create the list of corner points.
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]);
    break;

return squareCorners;
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(100, 0));
squareOrtho.push_back(cv::Point2f(100, 100));
squareOrtho.push_back(cv::Point2f(0, 100));

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

// Transform image to get an orthophoto square image.
cv::Mat imageSquare;
const int cellSize = 10;
cv::Size imageSquareSize(10 * cellSize, 10 * cellSize);
cv::warpPerspective(imageInputGray, imageSquare, M, imageSquareSize);
cv::imshow("Marker", imageSquare);
```
Read the 10x10 bit matrix

```cpp
#include <opencv2/opencv.hpp>

// This function tries to read the bit pattern of a marker.
void readBitMatrix(cv::Mat imageMarker, int cellSize, char bitMatrix[10][10])
{
    // Threshold the marker image.
    cv::threshold(imageMarker, // output thresholded image
                  imageMarker, // output value
                  0, // threshold value (not used; Otsu will compute)
                  255, // output value
                  cv::THRESH_<Otsu | cv::THRESH_BINARY>);

    for (int iy = 0; iy<10; iy++) {
        for (int ix = 0; ix<10; ix++) {
            int x = ix * cellSize;
            int y = iy * cellSize;
            cv::Mat cell = imageMarker(cv::Rect(x, y, cellSize, cellSize));

            int nZ = cv::countNonZero(cell);

            if (nZ > (cellSize*cellSize) / 2)
                bitMatrix[iy][ix] = 1;
            else
                bitMatrix[iy][ix] = 0;
        }
    }
}
```

Function “readBitMatrix”
Matching the pattern

- We have a set of pre-stored patterns.
- We match the input bit pattern against each of the pre-stored patterns.
- If it matches, we return the id of the pattern that matched.
- We also need to allow for a rotated pattern; so check possible rotations of 90, 180, and 270 degrees.
Estimating Pose

• Once we have found a marker, we can estimate its pose
• We have four 3D-2D corresponding points
• We also need to have the camera intrinsic parameters
• We use the OpenCV function “solvePnP”