Understanding ArUco Markers

1 Why Do We Need ArUco?

Imagine we want to identify objects by writing numbers on them. We could write simple numbers like 1, 2, 3, 4, 5, 6, 7, 8, 9.

But what happens when we write 6 and 9? If the paper gets rotated, 6 looks like 9 and 9 looks like 6! The computer gets confused.

So we need something smarter than just writing numbers.

2 Making a Square for ArUco

Instead of writing numbers, we make a square pattern with black and white bits:

```
# Each black square = 1, each white square = 0
# A 6x6 grid has 36 bits total
pattern = [
    [1, 0, 1, 0, 1, 0],
    [0, 1, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 0],
    [0, 1, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 0],
    [0, 1, 0, 1, 0, 1]
]
```

3 Dictionary Size Problem

With 36 bits, we could make 2^{36} different patterns. That's about 68 billion different IDs!

But look at this code:

```
self.aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.
   DICT_6X6_250)
```

Why only 250 patterns instead of 68 billion? Because we need error margin!

4 Error Margin

When you shout to a friend across a noisy room, you don't just say "six" once. You say "S-I-X, six!" to make sure they hear correctly.

ArUco does the same thing. It uses many bits to represent one ID, so if the camera makes mistakes reading some squares, it can still figure out the correct ID.

Only 250 out of billions possible patterns are "valid" because:

- Each valid pattern is very different from others
- Small mistakes can be detected and corrected
- Invalid patterns get thrown away

5 Code Explanation

5.1 Setting Up the Detector

```
class ArUcoDetector:
    def __init__(self, marker_size=0.05):
        self.marker_size = marker_size
```

This tells the computer that our marker is 5 centimeters big in real life.

5.2 Camera Matrix

```
self.camera_matrix = np.array([
    [600, 0, 300],
    [0, 600, 300],
    [0, 0, 1]
], dtype=np.float32)
```

This is like the "prescription" of the camera. It tells the computer how the camera sees the world:

- 600 = focal length (how "zoomed in" the camera is)
- 300 = center point of the image

5.3 Loading the Dictionary

```
self.aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.
   DICT_6X6_250)
self.aruco_params = cv2.aruco.DetectorParameters()
```

This loads the "rulebook" of 250 valid marker patterns that the computer knows how to read.

5.4 Detection Parameters

```
self.aruco_params.adaptiveThreshWinSizeMin = 3
self.aruco_params.adaptiveThreshWinSizeMax = 23
self.aruco_params.minMarkerPerimeterRate = 0.03
self.aruco_params.maxMarkerPerimeterRate = 4.0
```

These numbers tell the computer:

- How to decide if a square is black or white
- How big or small markers can be in the image
- How to filter out things that aren't markers

5.5 Finding Markers

```
def detect_markers(self, image_path):
    image = cv2.imread(image_path)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    detector = cv2.aruco.ArucoDetector(self.aruco_dict, self.
       aruco_params)
    corners, ids, rejected = detector.detectMarkers(gray)
```

Step by step:

1. `cv2.imread()` loads the picture from file
2. `cv2.cvtColor()` makes it black and white (ArUco works better this way)
3. `detectMarkers()` looks for square patterns that match our 250 valid patterns

The computer returns:

- `corners` = where the marker corners are in the image
- `ids` = which ID number each marker has
- `rejected` = square things that looked like markers but weren't valid

5.6 Finding Position and Rotation

```
rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(
    corners, self.marker_size, self.camera_matrix, self.dist_coeffs
)
```

This answers the questions:

- Where is the marker? (`tvecs` = translation = position)
- How is it rotated? (`rvecs` = rotation vectors)

5.7 Converting Rotation to Angles

```
rotation_matrix, _ = cv2.Rodrigues(rvecs[i])
roll, pitch, yaw = self.rotation_matrix_to_euler_angles(
    rotation_matrix)
```

The computer gives us rotation in a complicated math format. We convert it to simple angles:

- **Roll** = tilting left or right (like rolling a ball)
- **Pitch** = tilting forward or backward (like pitching a baseball)
- **Yaw** = turning left or right (like saying "no" with your head)

5.8 Storing Results

```
pose_info = {
    'id': int(marker_id),
    'translation': {
        'x': float(x), 'y': float(y), 'z': float(z),
        'distance': float(np.linalg.norm(tvecs[i]))
    },
    'rotation': {
        'roll': float(np.degrees(roll)),
        'pitch': float(np.degrees(pitch)),
        'yaw': float(np.degrees(yaw))
    }
}
```

For each marker we found, we save:

- Its ID number
- Where it is (x, y, z coordinates and distance)
- How it's rotated (roll, pitch, yaw in degrees)

6 Why Are We Studying This?

Let's say in a factory, a robot needs to identify a box. Think of an Amazon warehouse - there are thousands of packages moving around.

Now there are some packages in the warehouse. Though Amazon doesn't send cakes, let's say one box has a cake for you or someone special (certainly not your bf/gf).

The robot needs to place this cake box precisely on a platform. If the robot makes a mistake and drops it or places it wrong, the cake gets spoiled!

Now you realize why you need ArUco tags - to avoid spoiling the party (or rather save yourself from angry customers).

The robot uses our code:

```python
results = detector.detect_markers(image_path)
for pose in results['poses']:
    x = pose['translation']['x']
    y = pose['translation']['y']
    z = pose['translation']['z']
    # Robot knows exactly where to grab the box!
```

With ArUco markers on the box, the robot knows:

- Exactly where the box is (x, y, z coordinates)
- How the box is oriented (roll, pitch, yaw)
- How far away it is (distance)

So the robot can grab the cake box gently and place it perfectly on the platform. No spoiled cake, no angry customers, and you save yourself!

7 Summary

ArUco markers solve the "6 looks like 9" problem by:

1. Using square patterns of black and white bits instead of numbers
2. Having only 250 valid patterns out of billions possible (for error protection)
3. Letting computers read them accurately and find their exact position and rotation

The code loads pictures, finds square patterns, checks if they match valid ArUco patterns, and calculates where they are in 3D space.