Karadeniz Teknik Üniversitesi

Bilgisayar Mühendisliği

Ders: Görüntü İşleme

Ders Sorumlusu: Prof. Dr. Murat Ekinci

Öğrenci Adı: Fatih Küçükbıyık

Öğrenci No: 413055

İçindekiler

Ödev Hakkında

Ödevin birinci kısmını aldığı resimdeki çizgileri ve çemberleri tespit eden C++ dilinde bir uygulamanın yazılması oluşturuyor. Ödev boyunca resimleri içeri aktarırken ve ekranda gösterimi yapılırken OpenCV kullanılmasına müsaade ediliyor. Onun dışında herhangi bir kısımda kütüphane kullanılmaması gerekiyor.

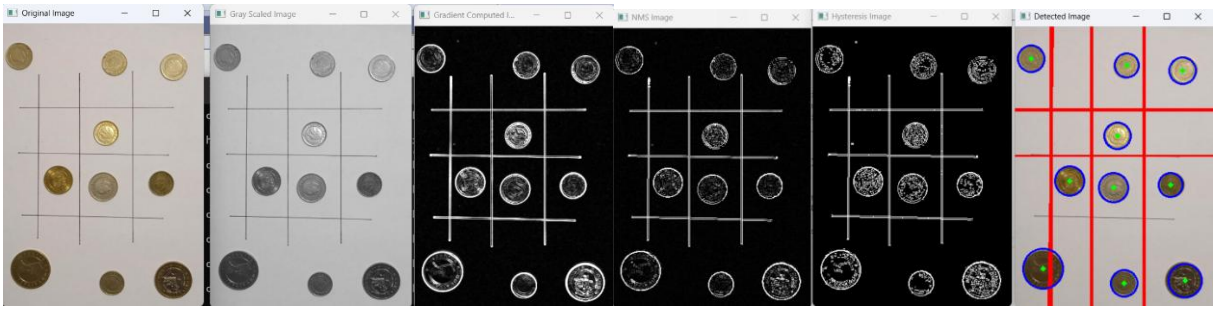
İkinci kısım ise farklı açılardan çekilen fotoğrafların ilk çektiğimiz düz fotoğrafa göre kalibre edilmesidir. Kalibrasyon işleminde ilk ödevde kullandığımız kenar tespit fonksiyonu kullanılacaktır.

Çizgilerin ve Çemberlerin Tespit Edilmesi

Main Fonksiyonu ve Uygulamanın Genel İşleyişi

```
221 int main()
222 {
223     string image_path = "../resim3.jpg";
224     Mat image = imread(image_path, IMREAD_COLOR);
225     if (image.empty()) {
226         cerr << "Görüntü Yüklenemedi!" << endl;
227         return -1;
228     }
229     Mat detected_image = image.clone();
230
231     Image new_image = ConvertToImage(image);
232     Image gray_scale_img = ConvertToGrayScale(new_image);
233     Image gradient_img = ComputeGradient(gray_scale_img);
234     Image NMS_img = NonMaximumSupression(gradient_img);
235     Image hysteresis_img = HysteresisThreshold(NMS_img);
236
237     LineDetection(hysteresis_img, detected_image);
238     CircleDetection(hysteresis_img, detected_image);
239
240     imshow("Original Image", image);
241     imshow("Gray Scaled Image", ConvertToMat(gray_scale_img));
242     imshow("Gradient Computed Image", ConvertToMat(gradient_img));
243     imshow("NMS Image", ConvertToMat(NMS_img));
244     imshow("Hysteresis Image", ConvertToMat(hysteresis_img));
245     imshow("Detected Image", detected_image);
246     waitKey(0);
247 }
```

Görüntü OpenCV kütüphanesi kullanarak içeriye aktarılmıştır ve OpenCV matrisine dönüştürülmüştür. Daha sonra bu görüntü uygulamada kullanılmak üzere kendi ***Image*** sınıfımıza dönüştürülmüştür. ***Image*** nesnesi kenarların tespit edilebilmesi için Canny Edge Detection aşamalarına sokulmuştur. Fakat Gaussian Blur olmadan daha iyi bir sonuç alındığı için bu kodda kullanılmamıştır. Canny Edge Detection sonucunda elde edilen Hysteresis Threshold görüntüsündeki çizgiler ve çemberler OpenCV de bulunan ***HoughLine*** ve ***HoughCircle*** fonksiyonları kullanılarak tespit edilmiştir. Son olarak aşamalar boyunca elde edilen resimler ekranda gösterilmiştir.



Resim Açma

Resim açma işlemi OpenCV de bulunan ***imread*** fonksiyonu ile yapılmıştır. Elde edilen resim ***Image*** nesnesine dönüştürülmüştür.

```
9   class Image {
10  public:
11      int w, h, c;
12      unsigned char* data;
13      vector<pair<int, int>> gradyan;
14
15      Image(int width, int height, int color) {
16          w = width; h = height; c = color;
17          data = new unsigned char[h * w * c];
18      }
19
20      ~Image() = default;
21  };
```

Image sınıfı resmin genişlik, yükseklik ve kanal bilgilerini tutmaktadır. Resimdeki piksel bilgileri bir boyutlu ***unsigned char*** dizisinde tutulmuştur. Burada ekstradan gradyan vektörü gradyan değerleri hesaplandıktan sonra Non-Maximum Supressionda da açıyı hesaplayabilmek için eklenmiştir.

```
29  Image ConvertToImage(Mat img) {
30      Image image(img.cols, img.rows, img.channels());
31      int k = 0;
32      for (int i = 0; i < image.h; i++) {
33          for (int j = 0; j < image.w; j++) {
34              Vec3b pixel = img.at<Vec3b>(i, j);
35              image.data[k++] = pixel[2];
36              image.data[k++] = pixel[1];
37              image.data[k++] = pixel[0];
38          }
39      }
40      return image;
41  }
```

ConvertToImage fonksiyonu ile OpenCV Mat matrisindeki veri Image nesnesine dönüştürülür. OpenCV BGR formatında kullanmaktadır burada görüntünün ***Image*** sınıfında RGB formatında tutulması sağlanır.

Canny Edge Detection

Elde edilen Image nesnesi resmin kenarlarının tespit edilmesi için Canny Edge Detection aşamalarına sokulur.

Gray-Scale Dönüşümü

```
43  Image ConvertToGrayScale(const Image& input_image) {
44      Image gray(input_image.w, input_image.h, 1);
45
46      for (int i = 0; i < input_image.h; i++) {
47          for (int j = 0; j < input_image.w; j++) {
48              int idx = (i * input_image.w + j) * input_image.c;
49              unsigned char r = input_image.data[idx];
50              unsigned char g = input_image.data[idx + 1];
51              unsigned char b = input_image.data[idx + 2];
52              gray.data[i * input_image.w + j] = 0.3 * r + 0.59 * g + 0.11 * b;
53          }
54      }
55      return gray;
56  }
```

Burada RGB yani 3 kanallı görüntü tek kanala indirilir. Bu işlem kenarların tespitini kolaylaştırmaktadır.

int idx = (i * input_image.w + j) * input_image.c;

Buradaki indekslemenin mantığı Image sınıfında pikselleri tek boyutlu bir dizide tutmamızdan gelmektedir. i değişkeni ile satır başına gidilir ve j ile sütunlar taranmış olur. Elde edilen RGB değerleri(sırasıyla) 0.3, 0.59, 0.11değerleri ile çarpılır böylece her piksel teker teker Gray-Scale formata dönüştürülür. Sonuç olarak fonksiyon Gray-Scale bilgiler tutan bir Image nesnesi dönderir.

Gaussian Blur Uygulanması

Bir pikselin değerini, kendisi ve komşularının ağırlıklı ortalamasına göre değiştirir. Bu ağırlıklar, bir **Gaussian fonksiyonu** ile belirlenir. Böylece komşulara olan uzaklık arttıkça, ağırlıkları azalır.

Gaussian fonksiyonu şu şekildedir:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

x, y: Pikselin merkezden uzaklığı

σ: Standart sapma

Görüntüdeki her piksel için bir kernel uygulanır. Bu Kernel 2D Gaussian fonksiyonu ile oluşturulmuş ağırlıklı bir matristir. Kernel görüntü üzerinde gezdirilir ve her piksel etrafındaki değerlerle çarpılarak yeni bir piksel değeri elde edilir.

Görüntü üzerindeki küçük parazitleri yok ederek daha temiz kenar tespiti yapılmasını sağlar.

Burada Gaussian Blur Fonksiyonu eklenmeden daha iyi sonuç alındığı için kodda gösterilmemiştir.

Gradyan Değerlerinin Hesaplanması

```
58  Image ComputeGradient(const Image& gray_image) {
59      Image gradient(gray_image.w, gray_image.h, 1);
60
61      gradient.gradyan = std::vector<std::pair<int, int>>(gray_image.w * gray_image.h, { 0, 0 });
62      int sobel_x[3][3] = { {-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1} };
63      int sobel_y[3][3] = { {-1, -2, -1}, {0, 0, 0}, {1, 2, 1} };
64
65      for (int y = 1; y < gray_image.h - 1; y++) {
66          for (int x = 1; x < gray_image.w - 1; x++) {
67              int gx = 0, gy = 0;
68
69              for (int i = -1; i <= 1; i++) {
70                  for (int j = -1; j <= 1; j++) {
71                      int pixel = gray_image.data[(y + i) * gray_image.w + (x + j)];
72                      gx += pixel * sobel_x[i + 1][j + 1];
73                      gy += pixel * sobel_y[i + 1][j + 1];
74                  }
75              }
76
77              gradient.gradyan[y * gray_image.w + x] = { gx, gy };
78              int magnitude = sqrt(gx * gx + gy * gy);
79              magnitude = min(255, magnitude); // aşırı değerleri kırp
80              gradient.data[y * gray_image.w + x] = magnitude;
81          }
82      }
83
84      return gradient;
85  }
```

Gray-Scale resme Sobel filtresi uygulanarak düşey ve yatay eksende gradyan değerleri yani x ve y ekseni boyunca değişimler hesaplanır. Bu değerler Non-Maximum Supression işleminde açı hesabında da kullanılacağı için Image sınıfında bulunan gradyan vektörüne eklenir. Daha

sonra bu değerleri birleştirerek kenarın büyüklüğünü(şiddetini) buluruz:

$$\text{magnitude} = \sqrt{G_x^2 + G_y^2}$$

Eğer bu değer büyükse orada bir kenar vardır denilir.

Non-Maximum Supression Uygulanması

```cpp
Image NonMaximumSupression(const Image& gradient_image) {
    Image result(gradient_image.w, gradient_image.h, 1);
    for (int y = 1; y < gradient_image.h - 1; y++) {
        for (int x = 1; x < gradient_image.w - 1; x++) {
            int gx = gradient_image.gradyan[y * gradient_image.w + x].first; // X yönündeki gradient
            int gy = gradient_image.gradyan[y * gradient_image.w + x].second; // Y yönündeki gradient

            float angle = atan2(gy, gx) * 180.0 / CV_PI;
            if (angle < 0) angle += 180;

            int current = gradient_image.data[y * gradient_image.w + x];
            int neighbour1 = 0, neighbour2 = 0;

            // Komşuları belirle
            if ((angle >= 0 && angle < 22.5) || (angle >= 157.5 && angle <= 180)) {
                neighbour1 = gradient_image.data[y * gradient_image.w + (x - 1)];
                neighbour2 = gradient_image.data[y * gradient_image.w + (x + 1)];
            }
            else if (angle >= 22.5 && angle < 67.5) {
                neighbour1 = gradient_image.data[(y - 1) * gradient_image.w + (x + 1)];
                neighbour2 = gradient_image.data[(y + 1) * gradient_image.w + (x - 1)];
            }
            else if (angle >= 67.5 && angle < 112.5) {
                neighbour1 = gradient_image.data[(y - 1) * gradient_image.w + x];
                neighbour2 = gradient_image.data[(y + 1) * gradient_image.w + x];
            }
            else if (angle >= 112.5 && angle < 157.5) {
                neighbour1 = gradient_image.data[(y - 1) * gradient_image.w + (x - 1)];
                neighbour2 = gradient_image.data[(y + 1) * gradient_image.w + (x + 1)];
            }

            if (current >= neighbour1 && current >= neighbour2) {
                result.data[y * gradient_image.w + x] = current;
            }
            else {
                result.data[y * gradient_image.w + x] = 0;
            }
        }
    }
    return result;
}
```

Bu işlemle birlikte gradyan büyüklüğü hesaplandıktan sonra kenar kalınlığını 1 piksel inceliğine indirilir. Her piksel komşuluğu ile karşılaştırılır. Komşuluğunda bulunan piksellerden büyük ise kenar kabul edilir diğer durumda değer sıfırlanır.

Hysteresis Threshold Uygulanması

```cpp
Image HysteresisThreshold(const Image& nms_img, int low_thresh = 20, int high_thresh = 70) {
    Image result(nms_img.w, nms_img.h, 1);
    for (int y = 1; y < nms_img.h - 1; y++) {
        for (int x = 1; x < nms_img.w - 1; x++) {
            int idx = y * nms_img.w + x;
            int val = nms_img.data[idx];
            if (val >= high_thresh) { result.data[idx] = 255;  }// güçlü kenar
            else if (val >= low_thresh) {
                // 8 komşusuna bak, biri güçlü kenar mı?
                bool connected_to_strong = false;
                for (int dy = -1; dy <= 1; dy++) {
                    for (int dx = -1; dx <= 1; dx++) {
                        if (dy == 0 && dx == 0) continue;
                        int n_y = y + dy;
                        int n_x = x + dx;
                        if (n_y >= 0 && n_y < nms_img.h && n_x >= 0 && n_x < nms_img.w) {
                            int n_idx = n_y * nms_img.w + n_x;
                            if (nms_img.data[n_idx] >= high_thresh) { connected_to_strong = true; }
                        }
                    }
                }
                result.data[idx] = connected_to_strong ? 255 : 0;
            }
            else { result.data[idx] = 0; }
        }
    }
    // Kenar piksellerini sıfırla
    for (int x = 0; x < nms_img.w; x++) {
        result.data[x] = 0;
        result.data[(nms_img.h - 1) * nms_img.w + x] = 0;
    }
    for (int y = 0; y < nms_img.h; y++) {
        result.data[y * nms_img.w] = 0;
        result.data[y * nms_img.w + nms_img.w - 1] = 0;
    }
    return result;
}
```

Non-Maximum Supression' dan sonra elimizde ince, potansiyel kenarları içeren bir görüntü olur. Ancak bu kenarların hangisi gerçekten önemli bir kenardır, hangisi rastgele gürültüdür, bunu anlamamızı sağlar. Pikseller düşük eşik ve yüksek eşik sınırına göre kontrol edilir. Eğer yüksek eşik değeri üzerinde ise kenar korunur, arada bir değerde ise güçlü kenar komşuluğu olup olmamasına göre korunur onun dışındakiler sıfırlanır.

Hough Transform

Kenar tespiti sonrası elde edilen dağınık kenar noktalarını, bir geometrik şekil (örneğin doğru) altında gruplayarak nesne tanıma sağlar.

Line Detection

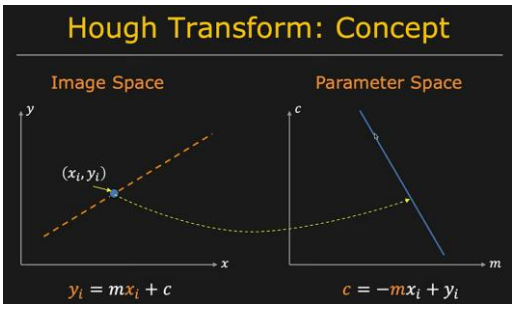
```cpp
void LineDetection(Image hysteresis_img, Mat image) {
    // Hough Line Detection
    vector<Vec2f> lines;
    Mat img = ConvertToMat(hysteresis_img);
    HoughLines(img, lines, 1, CV_PI / 180, 132); // 132
    for (size_t i = 0; i < lines.size(); i++) {
        float rho = lines[i][0];
        float theta = lines[i][1];
        Point pt1, pt2;
        double a = cos(theta), b = sin(theta);
        double x0 = a * rho, y0 = b * rho;
        pt1.x = cvRound(x0 + 1000 * (-b));
        pt1.y = cvRound(y0 + 1000 * (a));
        pt2.x = cvRound(x0 - 1000 * (-b));
        pt2.y = cvRound(y0 - 1000 * (a));
        line(image, pt1, pt2, Scalar(0, 0, 255), 2);
    }
}
```

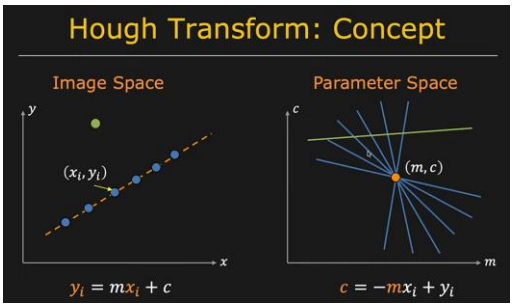
Bir doğru, y = mx + b formunda ifade edilir. Ancak Hough dönüşümünde daha sağlam olan kutupsal (polar) gösterim kullanılır:

$$\rho = x \cos\theta + y \sin\theta$$

- ρ: Doğrunun orijine uzaklığı
- θ: Doğrunun eğimi (açısı)

Canny ile kenar noktaları tespit edildikten ve her piksel için belirli açılar(θ) denendikten sonra ρ değeri hesaplanır. Kenar uzayında Accumulator matrisinde (θ, ρ) bu değerler için oy verilir. Accumulator matrisinde yüksek oy alan noktalar, gerçek doğrulara karşılık gelir.





Circle Detection

```cpp
void CircleDetection(Image hysteresis_img, Mat image) {
    // Hough Circle Detection
    Mat img1 = ConvertToMat(hysteresis_img);
    vector<Vec3f> circles;
    HoughCircles(img1, circles, HOUGH_GRADIENT, 1, img1.rows / 8, 100, 20, 5, 45);
    for (size_t i = 0; i < circles.size(); i++) {
        Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
        int radius = cvRound(circles[i][2]);
        circle(image, center, radius, Scalar(255, 0, 0), 2);
        circle(image, center, 2, Scalar(0, 255, 0), 3); // center
    }
}
```

Bir daire, merkezi (a, b) ve yarıçapı r olmak üzere şu şekilde ifade edilir:

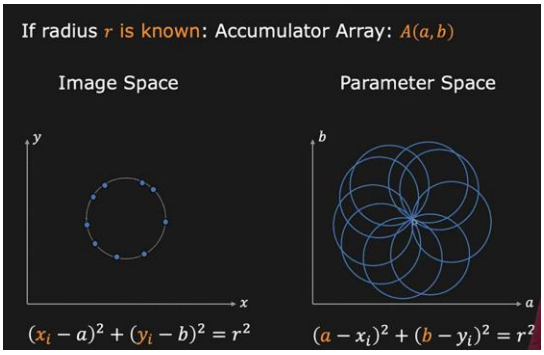
$$(x - a)^2 + (y - b)^2 = r^2$$

Her kenar pikseli (x, y) için, bu pikselin bir dairenin çevresi üzerinde olduğu varsayılır.
O zaman merkez (a, b) ve yarıçap r olmak üzere:

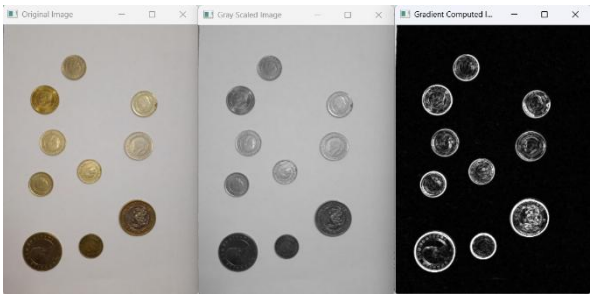
Mümkün olan tüm merkezler hesaplanır. Her olası merkez için bir “oy” verilir (Accumulator).

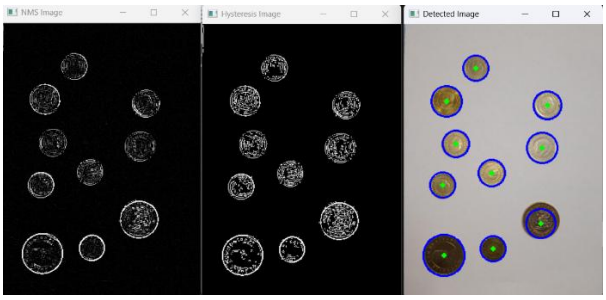
- a: dairenin x koordinatlı merkezi
- b: dairenin y koordinatlı merkezi
- r: yarıçap

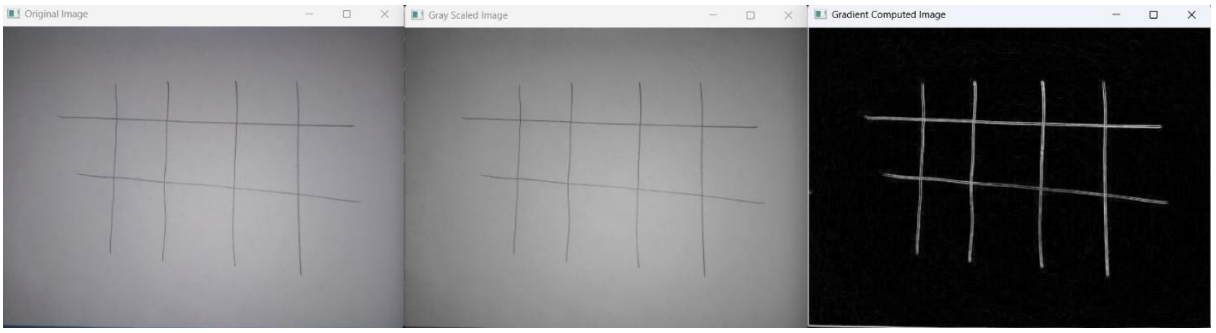
Bu nedenle Hough uzayı: **(a, b, r)** şeklindedir → 3D oylama gerekir.

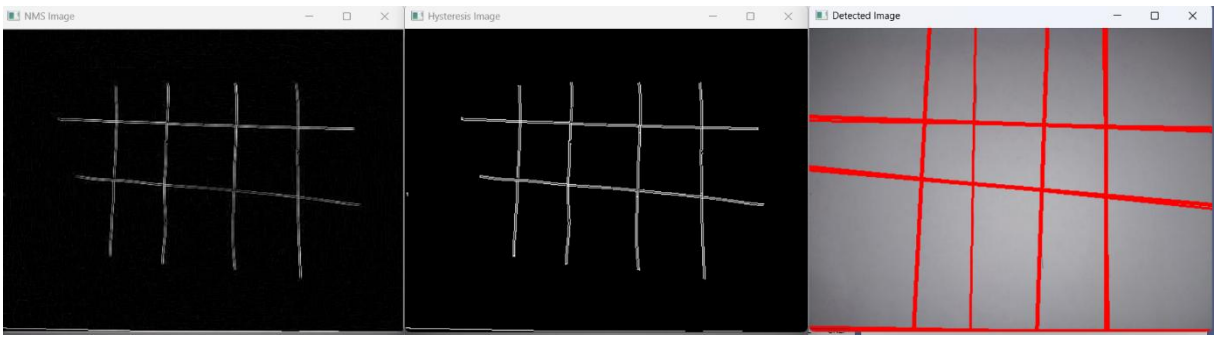


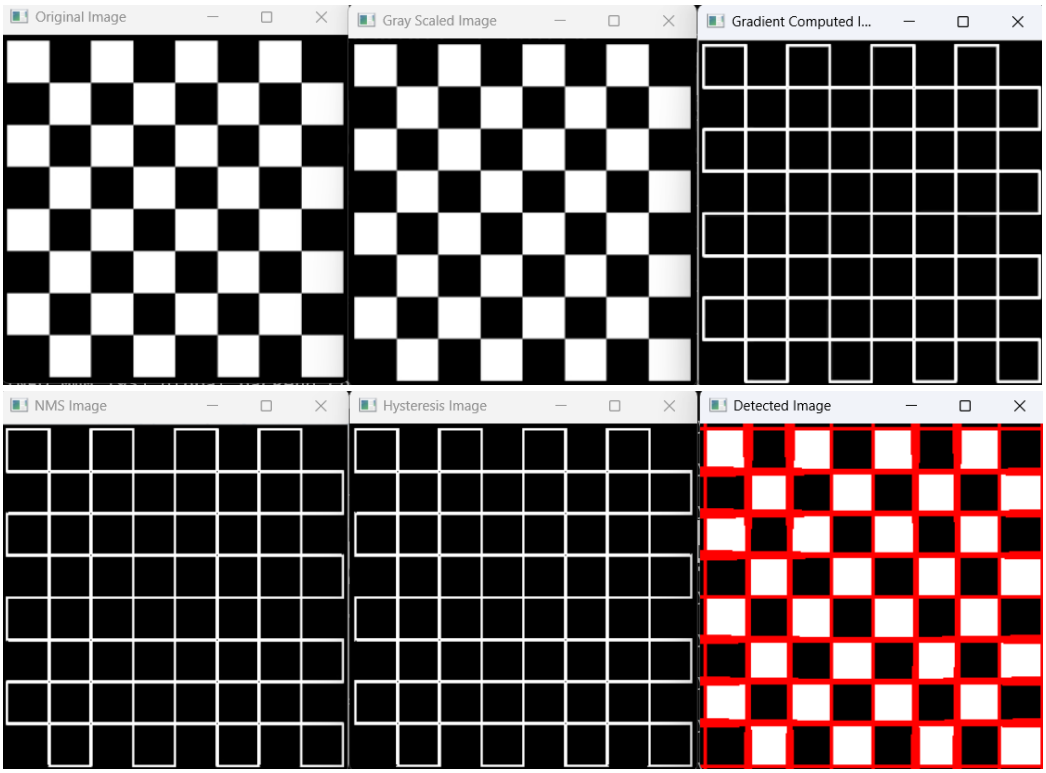
Sonuçlar











Calibration

Main Fonksiyonu ve Kodun Genel İşleyişi

```cpp
int main()
{
    string image_path1 = "../1.jpg"; // Orijinal resim
    string image_path2 = "../2.jpg"; // Farklı açıdan çekilmiş resim


    Mat image1 = imread(image_path1, IMREAD_COLOR);
    Mat image2 = imread(image_path2, IMREAD_COLOR);


    vector<Vec2f> lines1 = LineDetection(image_path1);
    vector<Vec2f> lines2 = LineDetection(image_path2);

    vector<Point2f> corners = findPoint(lines1, image1);
    vector<Point2f> dst_corners = findPoint(lines2, image2);

    vector<pair<int, int>> a = { {0, 255}, {255, 0}, {255, 255} };

    // Köşegenleri görmek için
    for (int i = 0; i < 3; i++) {
        line(image1, corners[i], corners[i], Scalar(0, a[i].first, a[i].second), 5);
    }

    for (int i = 0; i < 3; i++) {
        line(image2, dst_corners[i], dst_corners[i], Scalar(0, a[i].first, a[i].second), 5);
    }

    Mat H = findHomography(dst_corners, corners);
    Mat aligned;
    warpPerspective(image2, aligned, H, image1.size()); // orijinal boyuta göre hizala

    imshow("Orijinal Resim", image1);
    imshow("Perspektif Resim ", image2);
    imshow("Aligned Resim", aligned);
    waitKey(0);
}
```

İlk ödevde kullanılan Line Detection aşamaları burada da aynı şekilde kullanılmıştır. İlk resim orijinal resim olup kalibrasyon bu resme göre yapılacaktır. İkinci resim ise aynı sahnenin farklı açılardan çekilmiş resmi olup kalibre edilecek fotoğraftır.

Line Detection

```cpp
vector<Vec2f> LineDetection(string path_name) {
    Mat image = imread(path_name, IMREAD_COLOR);
    if (image.empty()) {
        cerr << "Görüntü Yüklenemedi!" << endl;
    }

    Image new_image = ConvertToImage(image);
    Image gray_scale_img = ConvertToGrayScale(new_image);
    Image gradient_img = ComputeGradient(gray_scale_img);
    Image NMS_img = NonMaximumSupression(gradient_img);
    Image hysteresis_img = HysteresisThreshold(NMS_img);


    // Hough Line Detection
    vector<Vec2f> lines;
    Mat img = ConvertToMat(hysteresis_img);
    HoughLines(img, lines, 1, CV_PI / 180, 120); // 132
    for (size_t i = 0; i < lines.size(); i++) {
        float rho = lines[i][0];
        float theta = lines[i][1];
        Point pt1, pt2;
        double a = cos(theta), b = sin(theta);
        double x0 = a * rho, y0 = b * rho;
        pt1.x = cvRound(x0 + 1000 * (-b));
        pt1.y = cvRound(y0 + 1000 * (a));
        pt2.x = cvRound(x0 - 1000 * (-b));
        pt2.y = cvRound(y0 - 1000 * (a));
        line(image, pt1, pt2, Scalar(0, 0, 255), 2);
    }

    //imshow("Image", image);
    //waitKey(0);
    return lines;
}
```

LineDetection fonksiyonu ilk ödevde kenar tespit etmek için kullandığımız aşamaları kullanıp tespit edilen kenarları bir vektör değişkeni olarak döndürmektedir.

Köşelerin Tespit Edilmesi

```cpp
Point2f computeIntersection(Vec2f line1, Vec2f line2) {
    float rho1 = line1[0], theta1 = line1[1];
    float rho2 = line2[0], theta2 = line2[1];

    float sin1 = sin(theta1), cos1 = cos(theta1);
    float sin2 = sin(theta2), cos2 = cos(theta2);

    float det = cos1 * sin2 - sin1 * cos2;
    if (fabs(det) < 1e-10) // çizgiler paralelse
        return Point2f(-1, -1);

    float x = (sin2 * rho1 - sin1 * rho2) / det;
    float y = (-cos2 * rho1 + cos1 * rho2) / det;

    return Point2f(x, y);
}
```

computeIntersection fonksiyonu ***findPoint*** fonksiyonu içerisinde önceden tespit edilmiş kenarların birbirleriyle kesiştikleri yerleri tespit eder ve kesişim noktalarını dönderir.

```cpp
vector<Point2f> findPoint(vector<Vec2f> lines, Mat image) {
    vector<Point2f> intersections;
    for (size_t i = 0; i < lines.size(); i++) {
        for (size_t j = i + 1; j < lines.size(); j++) {
            Point2f pt = computeIntersection(lines[i], lines[j]);
            if (pt.x >= 0 && pt.y >= 0 && pt.x < image.cols && pt.y < image.rows) {
                intersections.push_back(pt);
            }
        }
    }

    Point2f topLeft, topRight, bottomLeft, bottomRight;
    float minSum = FLT_MAX, maxSum = -FLT_MAX;
    float minDiff = FLT_MAX, maxDiff = -FLT_MAX;

    for (auto& pt : intersections) {
        float sum = pt.x + pt.y;
        float diff = pt.y - pt.x;

        if (sum < minSum) { minSum = sum; topLeft = pt; }
        if (sum > maxSum) { maxSum = sum; bottomRight = pt; }
        if (diff < minDiff) { minDiff = diff; topRight = pt; }
        if (diff > maxDiff) { maxDiff = diff; bottomLeft = pt; }
    }

    vector<Point2f> corners = { topLeft, topRight, bottomRight, bottomLeft };
    for (const auto& corner : corners) {
        cout << "Corner: " << corner << endl;
    }
    return corners;
}
```

findPoint fonksiyonu kenarların kesiştiği noktalar belirlendikten sonra büyük olasılıkla köşe olabilecek noktaları dönderir. Köşe noktaları hem ilk hem de ikinci resim içinde hesaplanır.

Homografi Matrisi ve Görüntü Kalibrasyonu

Homografi Matrisi

Homografi, iki düzlem (örneğin iki farklı görüntü) arasındaki perspektif dönüşüm ilişkisini ifade eder. Bir düzlemdeki noktaları, başka bir düzlemdeki karşılıklarına dönüştürmek için 3×3'lük bir dönüşüm matrisi kullanılır.

Bir nokta dönüşümü şöyle yazılır:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \sim H \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Burada:

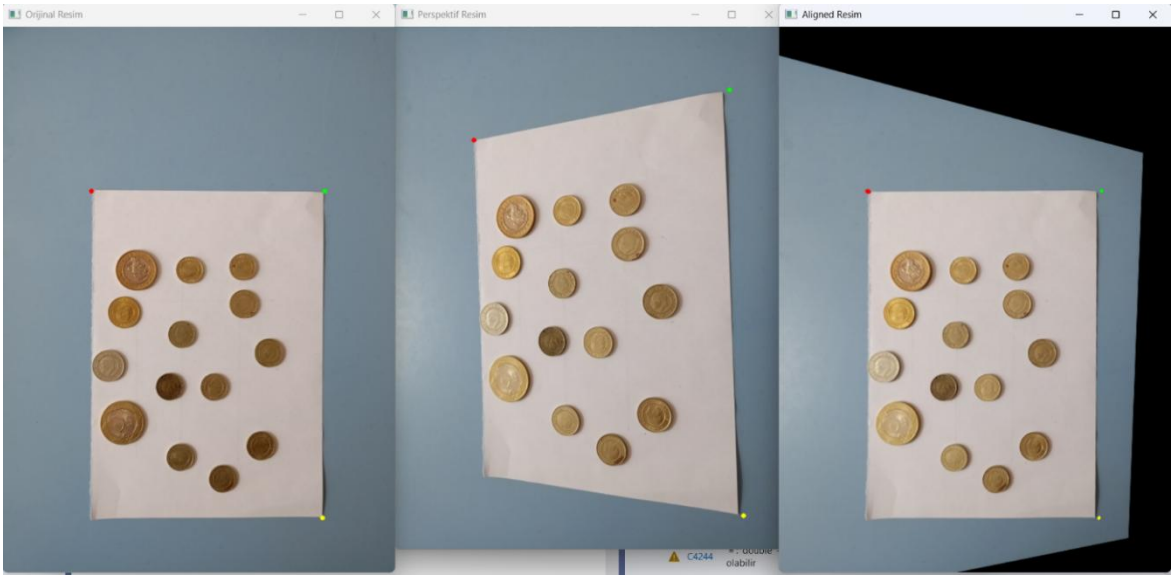
- (x, y): Kaynak görüntüdeki nokta,
- (x', y'): Hedef görüntüdeki karşılık gelen nokta,
- H: 3×3 Homografi matrisi (8 serbest parametreli çünkü 9. parametre genellikle 1'e sabitlenir),
- ∼: Eşdeğerlik, çünkü Homografide ölçek farkı olabilir

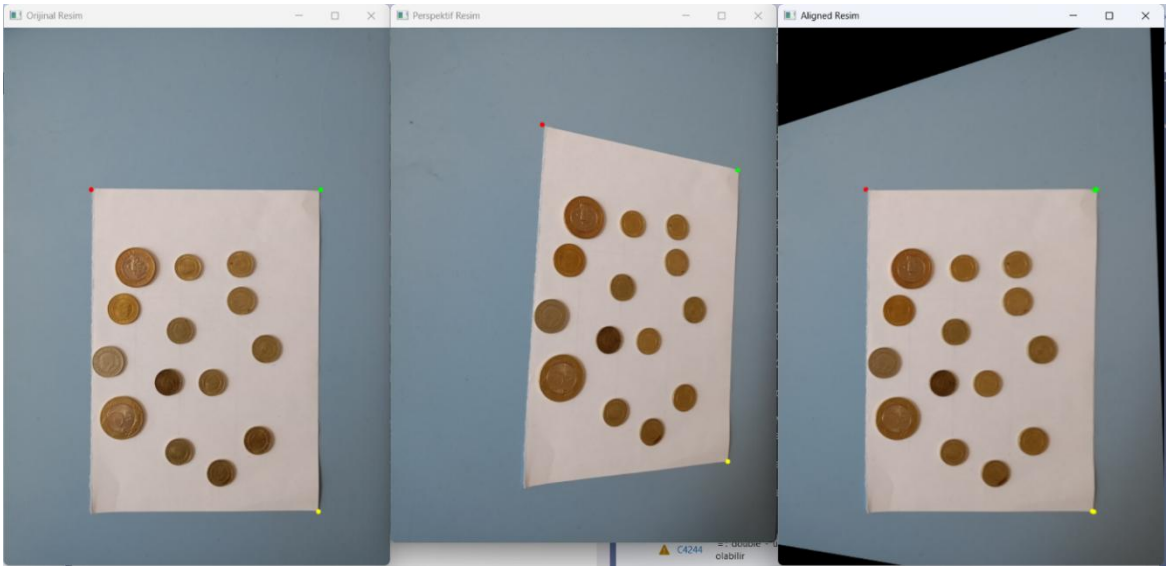
İki görüntüden elde edilen köşe noktaları OpenCV' de bulunan ***findHomography*** fonksiyonun parametre olarak verilip Homografi matrisi elde edilir.

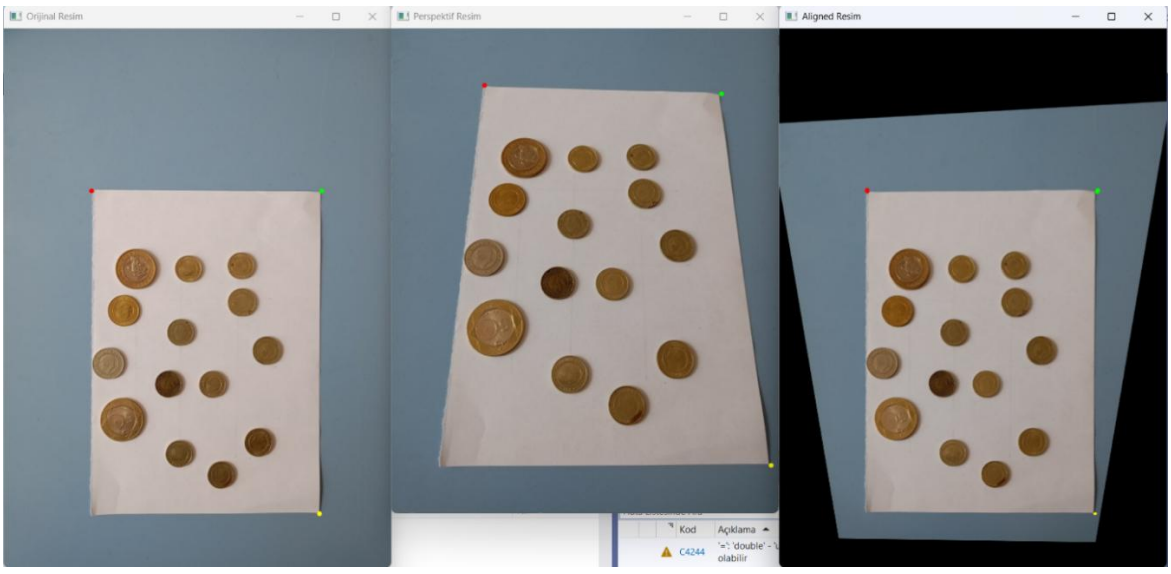
Resmin Hizalanması

Elde ettiğimiz Homografi matrisini, ilk resmi ve hizalanacak resmi OpenCV' de bulunan ***warpPerspective*** fonksiyonuna parametre olarak vererek hizalanmış resmi elde ederiz.

Sonuçlar


Orijinal Resim
Perspektif Resim
Aligned Resim


Orijinal Resim
Perspektif Resim
Aligned Resim


Orijinal Resim
Perspektif Resim
Aligned Resim

Kaynakça

1. **Gonzalez, R. C., & Woods, R. E. (2018).**
   Digital Image Processing (4th ed.). Pearson.
2. **Wikipedia – Gaussian Blur**
   https://en.wikipedia.org/wiki/Gaussian_blur
3. **Shree K. Nayar Hough Transform (First Principle of Computer Vision)**
   https://www.youtube.com/watch?v=XRBc_xkZREg