# Line/Circle Detection

Test edilen girdi görüntüsü 1:

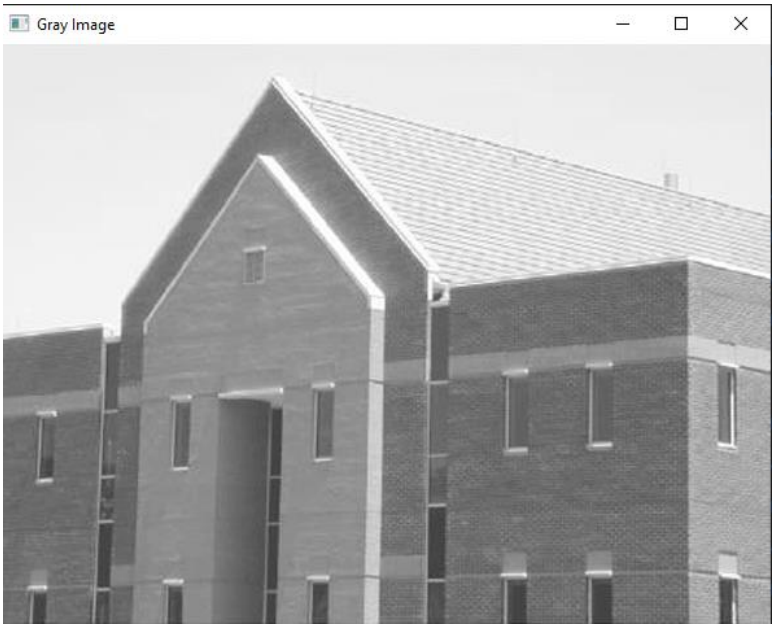
Test edilen girdi görüntüsü 2:

1. **RGB görüntünün griye dönüştürülmesi**

```cpp
// RGB to Intensity dönüşümü
unsigned char rgbToGray(int r, int g, int b) {
    return static_cast<unsigned char>(0.299 * r + 0.587 * g + 0.114 * b);
}

// RGB'den griye çevirme fonksiyonu
Mat convertToGray(const Mat& img) {
    Mat grayImg(img.rows, img.cols, CV_8UC1); // Gri görüntü için matris

    unsigned char* grayData = grayImg.data;
    unsigned char* imgData = img.data;
    int channels = img.channels();
    int width = img.cols;
    int height = img.rows;

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int index = (i * width + j) * channels; // Pikselin başlangıç indexi
            int grayIndex = i * width + j;

            int blue = imgData[index];
            int green = imgData[index + 1];
            int red = imgData[index + 2];

            // RGB'den gri tona çevir ve yeni matrise ata
            grayData[grayIndex] = rgbToGray(red, green, blue);
        }
    }

    // Görüntüyü göster
    imshow("Gray Image", grayImg);
    waitKey(0);
    //kaydet
    imwrite("resource/test_gray.png", grayImg);

    return grayImg;
}
```

Fonksiyonun temel amacı, her pikselin kırmızı (R), yeşil (G) ve mavi (B) bileşenlerini kullanarak karşılık gelen gri değerleri hesaplamak ve yeni bir gri tonlamalı görüntü oluşturup bu görüntüyü kullanıcıya göstermek ve kaydetmektir.

Giriş görüntüsünü parametre olarak alır. Gri görüntü için matris oluşturulur. Her pikselin başlangıç indeksi, renkli görüntüde 3 kanal (BGR) olduğu için “(i * width + j) * channels” şeklinde hesaplanır. convertToGray fonksiyonunun içerisinde rgbToGray(red, green, blue) fonksiyonu çağrılarak gri ton değeri hesaplanır.



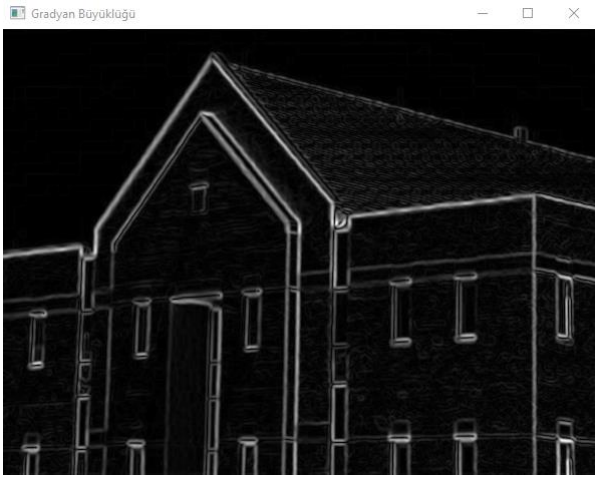


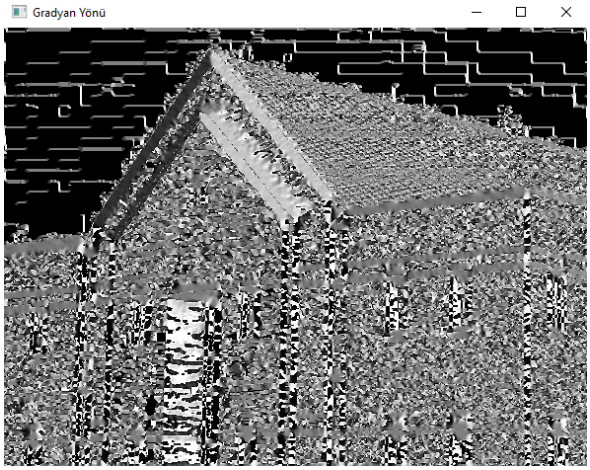
## 2. Gradient görüntünün oluşturulması

```cpp
Mat GradientImage(const Mat& grayImg, Mat& gradientDirection) {
    // Gradyanları hesapla
    Mat gradX, gradY;

    int sobelX[3][3] = {
            {-1, 0, 1},
            {-2, 0, 2},
            {-1, 0, 1}
    };

    int sobelY[3][3] = {
        {-1, -2, -1},
        { 0,  0,  0},
        { 1,  2,  1}
    };

    // Çıkış matrislerini oluştur
    gradX = Mat::zeros(grayImg.rows - 2, grayImg.cols - 2, CV_32F);
    gradY = Mat::zeros(grayImg.rows - 2, grayImg.cols - 2, CV_32F);

    // Görüntünün iç piksellerini dolaş
    for (int r = 0; r < gradX.rows; r++) {
        for (int c = 0; c < gradX.cols; c++) {
            float gx = 0, gy = 0;

            // 3x3 çekirdeği uygula
            for (int m = 0; m < 3; m++) {
                for (int n = 0; n < 3; n++) {
                    uchar pixel = grayImg.at<uchar>(r + m, c + n);
                    gx += pixel * sobelX[m][n];
                    gy += pixel * sobelY[m][n];
                }
            }

            gradX.at<float>(r, c) = gx;
            gradY.at<float>(r, c) = gy;
        }
    }
```

```
    // Gradyan büyüklüğünü ve yönünü hesapla
    Mat gradientMagnitude;

    gradientMagnitude = Mat(gradX.size(), CV_32F);
    gradientDirection = Mat(gradX.size(), CV_32F);

    for (int r = 0; r < gradX.rows; r++) {
        for (int c = 0; c < gradX.cols; c++) {
            float gx = gradX.at<float>(r, c);
            float gy = gradY.at<float>(r, c);

            // Büyüklüğü hesapla
            gradientMagnitude.at<float>(r, c) = sqrt(gx * gx + gy * gy);

            // Açıyı hesapla
            float theta = atan2(gy, gx) * (180.0 / PI);
            // Açıyı 0-180 aralığına getir (çizgilerin yönü değil doğrultusu önemli)
            if (theta < 0) theta += 180;

            gradientDirection.at<float>(r, c) = theta;

        }
    }
    // Görselleştirme için normalize et
    Mat gradMagVis, gradDirVis;
    normalize(gradientMagnitude, gradMagVis, 0, 255, NORM_MINMAX);
    gradMagVis.convertTo(gradMagVis, CV_8U);

    normalize(gradientDirection, gradDirVis, 0, 255, NORM_MINMAX);
    gradDirVis.convertTo(gradDirVis, CV_8U);

    // Sonuçları görüntüle
    imshow("Gradyan Büyüklüğü", gradMagVis);
    imshow("Gradyan Yönü", gradDirVis);

    // Görüntüyü kaydet
    imwrite("resource/gradient_magnitude.jpg", gradMagVis);
    imwrite("resource/gradient_direction.jpg", gradDirVis);

    waitKey(0); // Kullanıcının tuşa basmasını bekle

    return gradientMagnitude;
}
```

Bu fonksiyon, gri görüntüdeki kenar bilgilerini elde etmek amacıyla Sobel operatörünü kullanarak gradyan büyüklüğü ve yönü hesaplamaktadır. Görüntünün yatay (gradX) ve dikey (gradY) gradyanları, 3x3 Sobel filtreleriyle hesaplanır. gradX'in ve gradY'nin mutlak değeri alınarak toplanır ve böylece her pixelin gradyan büyüklüğü hesaplanır. Gradient yönünün hesaplanabilmesi için gradY'nin gradX'e oranının arctanjantı alınır, 0-180 arasına sınırlanır ve gradientDirection matrisine atanır.

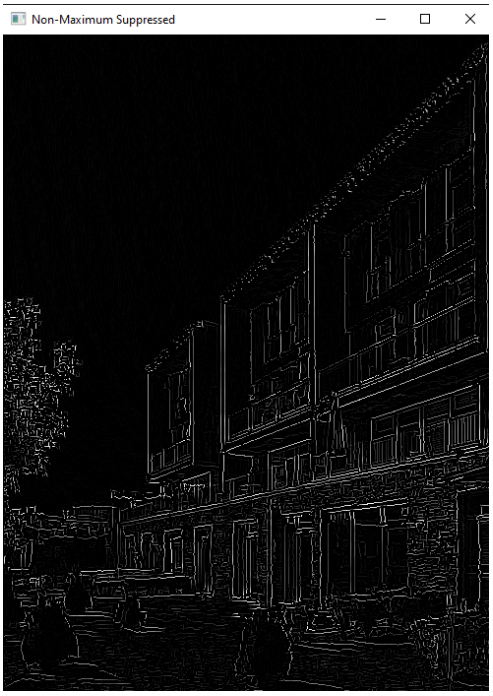




## 3. Canny Edge Detection

```cpp
Mat nonMaximumSuppression(const Mat& magnitude, const Mat& angle) {
    Mat suppressed = Mat::zeros(magnitude.size(), CV_8UC1);

    for (int i = 1; i < magnitude.rows - 1; ++i) {
        for (int j = 1; j < magnitude.cols - 1; ++j) {
            float angleDeg = angle.at<float>(i, j);
            float mag = magnitude.at<float>(i, j);

            // Normalize angle to [0,180)
            if (angleDeg < 0) angleDeg += 180;
            else if (angleDeg >= 180) angleDeg -= 180;

            float neighbor1 = 0, neighbor2 = 0;

            if ((angleDeg >= 0 && angleDeg < 22.5) || (angleDeg >= 157.5 && angleDeg < 180)) {
                // 0 derece (yatay)
                neighbor1 = magnitude.at<float>(i, j - 1);
                neighbor2 = magnitude.at<float>(i, j + 1);
            }
            else if (angleDeg >= 22.5 && angleDeg < 67.5) {
                // 45 derece (çapraz)
                neighbor1 = magnitude.at<float>(i - 1, j + 1);
                neighbor2 = magnitude.at<float>(i + 1, j - 1);
            }
            else if (angleDeg >= 67.5 && angleDeg < 112.5) {
                // 90 derece (dikey)
                neighbor1 = magnitude.at<float>(i - 1, j);
                neighbor2 = magnitude.at<float>(i + 1, j);
            }
            else if (angleDeg >= 112.5 && angleDeg < 157.5) {
                // 135 derece (çapraz)
                neighbor1 = magnitude.at<float>(i - 1, j - 1);
                neighbor2 = magnitude.at<float>(i + 1, j + 1);
            }

            if (mag >= neighbor1 && mag >= neighbor2)
                suppressed.at<uchar>(i, j) = static_cast<uchar>(mag); // Orijinal magnitude değerini sakla
        }
    }
    Mat suppressedNorm;
    normalize(suppressed, suppressedNorm, 0, 255, NORM_MINMAX);
    suppressedNorm.convertTo(suppressedNorm, CV_8U);

    imshow("Non-Maximum Suppressed", suppressedNorm);
    waitKey(0);
    imwrite("resource/suppressed.jpg", suppressedNorm);
    return suppressed;
}
```


nonMaximumSuppression fonksiyonu, kenar tespiti sonrası **yalnızca en belirgin (maksimum) kenar noktalarını** korumak amacıyla **non-maximum suppression** algoritmasını uygular. Her piksel için gradyan yönü (0–180°) 4 temel doğrultuya ayrılır: 0° (yatay), 45° (çapraz), 90° (dikey), 135° (çapraz).

Mevcut pikselin büyüklüğü (magnitude), kendi yönündeki iki komşusuyla karşılaştırılır. Eğer büyüklük, iki komşusundan da büyük veya eşitse, bu piksel korunur. Daha küçük olan piksel değerleri bastırılarak kenar kalınlığı azaltılır ve yalnızca belirgin kenarlar korunur. Elde edilen sonuç normalize edilerek görselleştirilir ve kaydedilir.

```
Mat Histogram(const Mat& suppressedNorm) {
    Mat hist = Mat::zeros(1, 256, CV_32SC1); // 0-255 arası değerler için 256

    for (int i = 0; i < suppressedNorm.rows; i++) {
        for (int j = 0; j < suppressedNorm.cols; j++) {
            int pixel_value = suppressedNorm.at<uchar>(i, j);
            hist.at<int>(0, pixel_value)++;
        }
    }

    return hist;
}
```

Bu fonksiyon, non-maximum suppressed görüntüye ait **histogram** oluşturur. 0–255 arası parlaklık seviyeleri için 256 hücreli bir matris (hist) oluşturulur. Her hücre, ilgili parlaklık seviyesindeki piksel sayısını tutar. Görüntüdeki her pikselin değeri okunur ve karşılık gelen histogram hücresinin değeri bir artırılır.

```cpp
int CalculateThresholdFromHistogram(const Mat& hist, float percentage) {
    int totalPixels = 0;
    for (int i = 0; i < 256; i++) {
        totalPixels += hist.at<int>(0, i);
    }

    int targetPixels = static_cast<int>(totalPixels * percentage);

    int cumulative = 0;
    for (int i = 255; i >= 0; i--) { // En parlaklardan başla
        cumulative += hist.at<int>(0, i);
        if (cumulative >= targetPixels) {
            return i;
        }
    }

    return 0;
}
```

Bu fonksiyon, histogram verisini kullanarak threshold değerini belirler. Histogramdaki tüm yoğunluk değerleri toplanarak toplam piksel sayısı hesaplanır. Histogram 255'ten 0'a doğru taranarak kümülatif toplam alınır. Hedef piksel sayısına ulaşıldığında, o parlaklık seviyesi eşik değeri olarak döndürülür. Döndürülen değere göre kullanılacak low ve high treshold değerleri belirlenir.

```cpp
Mat ApplyThresholdWithHysteresis(const Mat& suppressedNorm, int lowThreshold, int highThreshold) {
    Mat binary = Mat::zeros(suppressedNorm.size(), CV_8UC1);

    for (int i = 0; i < suppressedNorm.rows; i++) {
        for (int j = 0; j < suppressedNorm.cols; j++) {
            uchar pixel = suppressedNorm.at<uchar>(i, j);

            if (pixel >= highThreshold) {
                binary.at<uchar>(i, j) = 255; // Güçlü kenar
            }
            else if (pixel >= lowThreshold) {
                binary.at<uchar>(i, j) = 100; // Zayıf kenar
            }
            else {
                binary.at<uchar>(i, j) = 0; // Arka plan
            }
        }
    }

    imshow("Threshold", binary);
    waitKey(0);
    imwrite("resource/Threshold.png", binary);

    return binary;
}
```

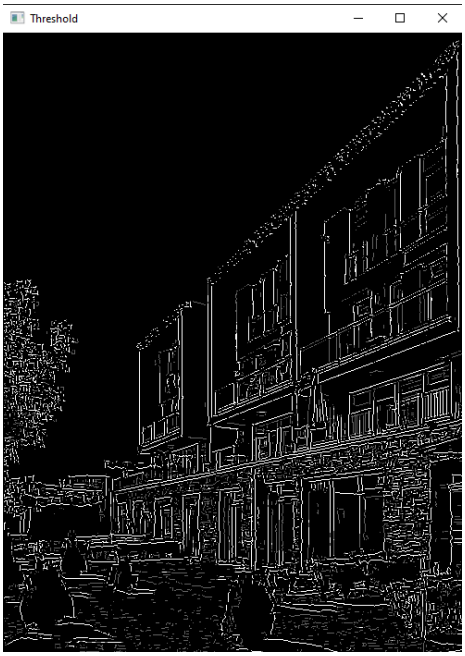
Bu fonksiyon, histerezisli kenar tespiti yöntemiyle baskılanmış gradyan görüntüsünden binary bir kenar haritası oluşturur. Her pikselin değeri:

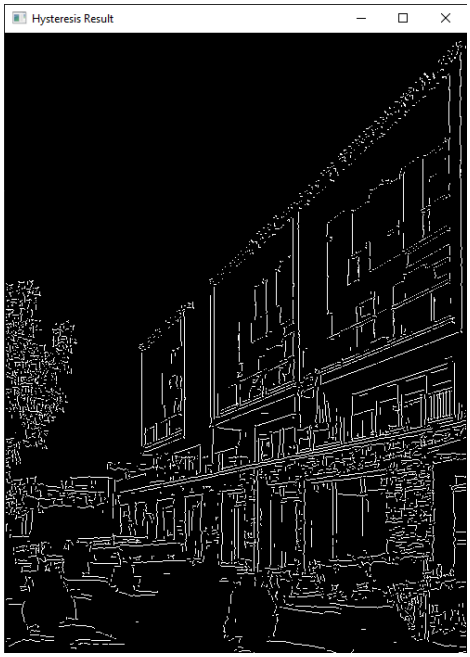
- highThreshold değerinden büyükse güçlü kenar (255) olarak işaretlenir.
- lowThreshold ile highThreshold arasında ise zayıf kenar (100) olarak atanır.
- lowThreshold'dan küçükse arka plan (0) kabul edilir.

```cpp
void edgesWithHysteresis(Mat& binary) {
    Mat output = binary.clone();
    int rows = binary.rows;
    int cols = binary.cols;

    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            if (binary.at<uchar>(i, j) == 100) { // Zayıf kenar
                // 8 komşuya bak
                if (
                    binary.at<uchar>(i - 1, j - 1) == 255 || binary.at<uchar>(i - 1, j) == 255 || binary.at<uchar>(i - 1, j + 1) == 255 ||
                    binary.at<uchar>(i, j - 1) == 255 || binary.at<uchar>(i, j + 1) == 255 ||
                    binary.at<uchar>(i + 1, j - 1) == 255 || binary.at<uchar>(i + 1, j) == 255 || binary.at<uchar>(i + 1, j + 1) == 255
                    ) {
                    output.at<uchar>(i, j) = 255; // Güçlü kenarla bağlantılıysa kenar olarak kabul et
                }
                else {
                    output.at<uchar>(i, j) = 0; // Bağlantısızsa arka plana at
                }
            }
        }
    }

    binary = output; // Güncellenmiş matrisi geri ata

    imshow("Hysteresis Result", binary);
    waitKey(0);
    imwrite("resource/hysteresis_edges.png", binary);
}
```

Bu fonksiyon, histerezis ile kenar izleme işlemini gerçekleştirerek, zayıf kenarların gerçekten kenar olup olmadığını belirler. Her zayıf kenar pikseli (100), 8 komşuluğu kontrol edilerek değerlendirilir. Eğer komşularından en az biri güçlü kenarsa (255), o piksel de kenar olarak kabul edilir (255). Aksi takdirde, arka plan kabul edilir (0).





## 4. Line Detection için Hough Space

```cpp
Mat houghTransform(const Mat& binaryImage) {
    int rows = binaryImage.rows;
    int cols = binaryImage.cols;
    // Uzaklık max köşegen kadar değer alabilir
    int d = (int)sqrt(rows * rows + cols * cols);
    Mat houghSpace = Mat::zeros(d, 180, CV_32SC1);

    // PI sabitini tanımlayalım
    const double PI = CV_PI;
    double distance = 0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (binaryImage.at<uchar>(i, j) == 255) {
                for (int k = 0; k < 180; k++) {
                    distance = i * cos(k * PI / 180) + j * sin(k * PI / 180);
                    int rho = cvRound(distance);

                    if (rho >= 0 && rho < d) {
                        houghSpace.at<int>(rho, k) += 1;
                    }
                }
            }
        }
    }

    Mat houghSpaceVis;
    normalize(houghSpace, houghSpaceVis, 0, 255, NORM_MINMAX);
    houghSpaceVis.convertTo(houghSpaceVis, CV_8U);

    imshow("Hough Transform", houghSpaceVis);
    waitKey(0);

    imwrite("resource/HoughTransform.png", houghSpaceVis);
    return houghSpace;
}
```

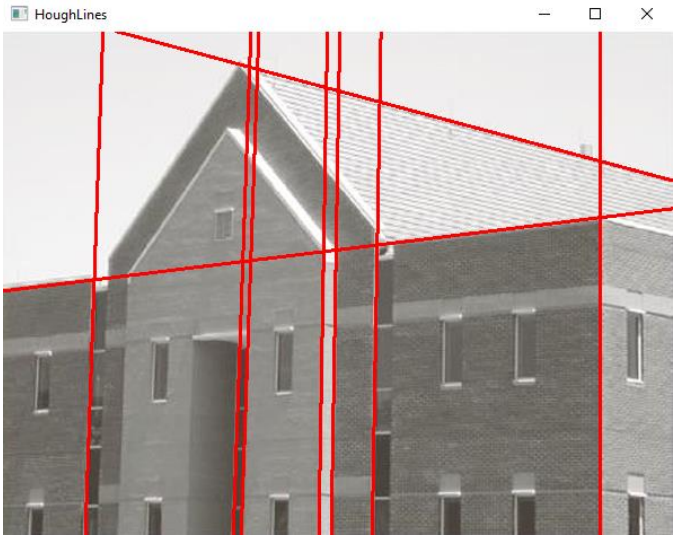

houghTransform fonksiyonu, binary görüntüyü parametre olarak alır. Binary görüntüdeki değeri 255 olan pixeller için d = c*cos(theta) + r*sin(theta) denklemi kullanılarak 0-180 arasındaki açılar için uzaklık değeri hesaplanır (c sütun değerini r satır değerini temsil eder) ve houghSpace matrisinde açının ve uzunluk değerinin karşılık geldiği indisin içeriği bir arttırılır. Uzaklığın maximum değeri köşegen uzunluğu kadar olabilir. HoughSpace matrisinin değerleri ekranda göstermeye uygun olmadığı için NormalizeValue fonksiyonu ile değerler 0-255 arasına sınırlanarak ekranda gösterilir.

```cpp
void manualHoughLines(const Mat& binary, vector<Vec2f>& lines, int rho_res, double theta_res, int threshold) {
    int rows = binary.rows, cols = binary.cols;
    int max_rho = sqrt(rows * rows + cols * cols);
    int rho_bins = max_rho / rho_res;
    int theta_bins = CV_PI / theta_res;

    Mat accumulator = Mat::zeros(rho_bins * 2, theta_bins, CV_32S);

    // Piksel taraması
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < cols; x++) {
            if (binary.at<uchar>(y, x) > 0) {
                for (int theta_idx = 0; theta_idx < theta_bins; theta_idx++) {
                    double theta = theta_idx * theta_res;
                    int rho = cvRound(x * cos(theta) + y * sin(theta)) + rho_bins;
                    accumulator.at<int>(rho, theta_idx)++;
                }
            }
        }
    }

    // Eşik değerini geçen noktaları toplama
    for (int rho = 0; rho < accumulator.rows; rho++) {
        for (int theta = 0; theta < accumulator.cols; theta++) {
            if (accumulator.at<int>(rho, theta) >= threshold) {
                lines.push_back(Vec2f((rho - rho_bins) * rho_res, theta * theta_res));
            }
        }
    }
}
```

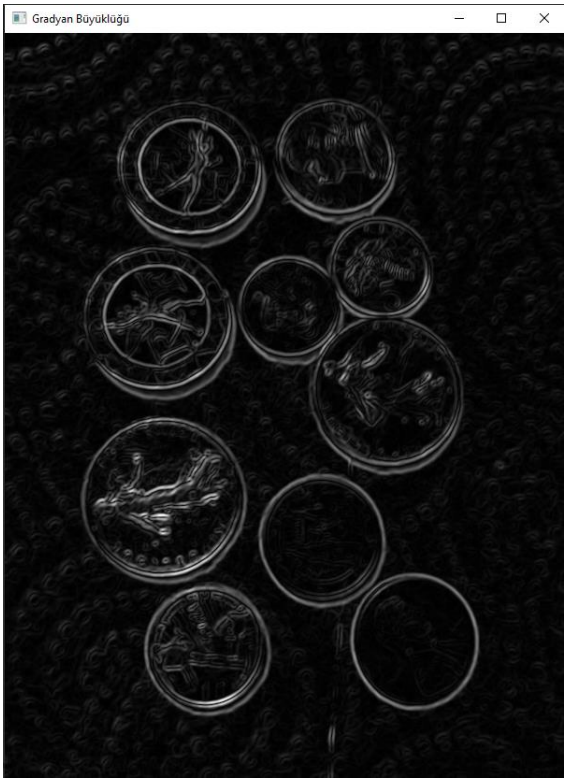
manualHoughLines fonksiyonu, kenarları belirlenmiş bir görüntüde doğruları bulmak için Hough Dönüşümünü gerçekleştirir. Öncelikle her beyaz piksel (kenar pikseli) için 0 ile 180 arasındaki açı değerlerine karşılık gelen rho (uzaklık) hesaplanır ve bu değerler bir akümülatör matrisinde tutulur. Yani hangi açıda hangi uzaklıkta ne kadar çok piksel varsa, bu matris içinde o hücrenin değeri artar. Daha sonra, bu akümülatör matrisinde belirli bir eşik değerinden (threshold) büyük olan hücreler, bir doğruya karşılık geldiği düşünülerek lines listesine eklenir. Bu sayede görüntüde yoğun şekilde hizalanmış kenarların oluşturduğu doğrular bulunmuş olur.

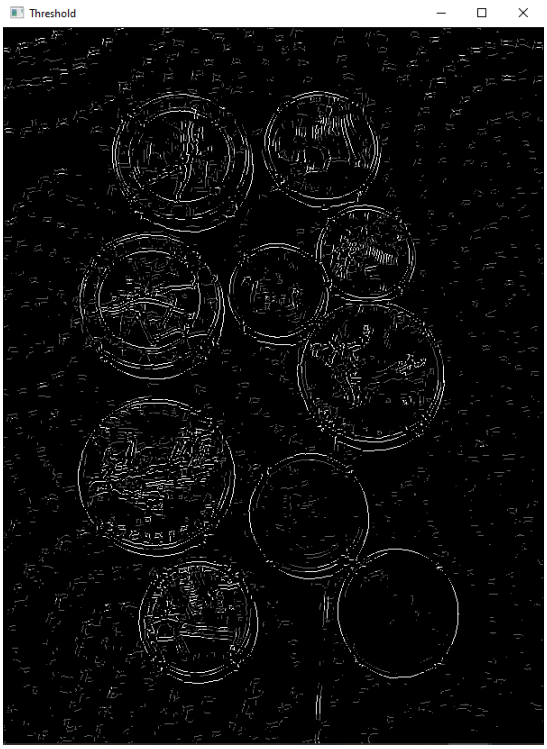
```cpp
void drawHoughLines(Mat& image, const Mat& binary) {
    vector<Vec2f> lines;
    manualHoughLines(binary, lines, 1, CV_PI / 180, 110);

    for (size_t i = 0; i < lines.size(); i++) {
        float rho = lines[i][0], 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("HoughLines", image);
    waitKey(0);
    imwrite("resource/houghlines.png", image);
}
```

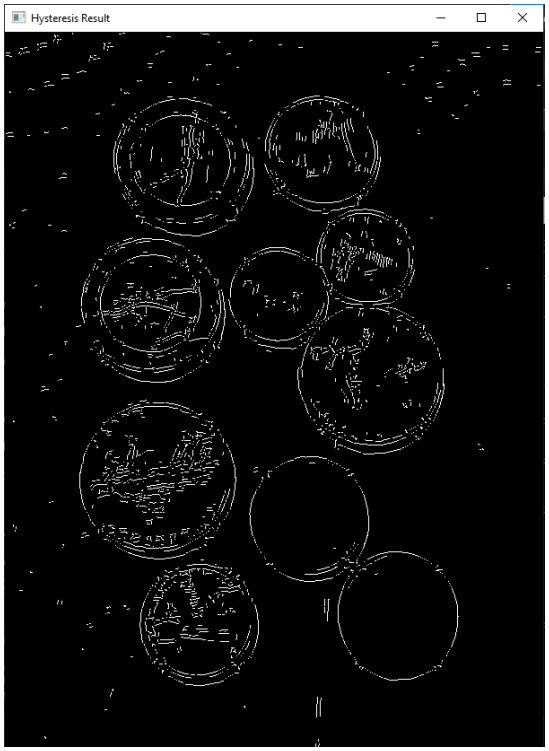
Bu fonksiyon, daha önce manualHoughLines ile tespit edilen doğruları orijinal görüntü üzerine çizer. manualHoughLines fonksiyonu ile elde edilen her bir (rho, theta) çifti, bir doğruyu temsil eder. Bu doğrular line fonksiyonu ile çizilir. Her doğru, görüntü üzerine kırmızı renk (BGR formatında (0, 0, 255)) ve 2 piksel kalınlıkla eklenir.




## 5. Circle Detection Fonksiyonların Çıktısı

## 6. Circle Detection için Hough Space

```cpp
vector<Vec3f> detectCircles(const Mat& binary, int minRadius, int maxRadius, int threshold) {
    vector<Vec3f> circles;
    int rows = binary.rows;
    int cols = binary.cols;
    int radiusRange = maxRadius - minRadius + 1;

    // 3 boyutlu akümülatör oluştur
    vector<Mat> houghSpace(radiusRange);
    for (int r = 0; r < radiusRange; r++) {
        houghSpace[r] = Mat::zeros(rows, cols, CV_32SC1);
    }

    // Kenar piksellerini dolaş ve oyları topla
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < cols; x++) {
            if (binary.at<uchar>(y, x) == 255) {
                // Her bir kenar pikseli için olası tüm çember merkezlerini hesapla
                for (int r = 0; r < radiusRange; r++) {
                    int radius = r + minRadius;

                    // Açı artış miktarını yarıçapa göre ayarla (daha büyük yarıçaplar için daha az açı)
                    int angleStep = max(1, 360 / (4 * radius)); // Daha verimli hesaplama

                    // Her bir açı için
                    for (int theta = 0; theta < 360; theta += angleStep) {
                        // Radyana çevir
                        double rad = theta * PI / 180.0;

                        // Potansiyel merkez koordinatlarını hesapla
                        int a = cvRound(x - radius * cos(rad));
                        int b = cvRound(y - radius * sin(rad));

                        // Görüntü sınırları içinde mi?
                        if (a >= 0 && a < cols && b >= 0 && b < rows) {
                            houghSpace[r].at<int>(b, a)++;
                        }
                    }
                }
            }
        }
    }

    // Geçici sonuçları sakla
    vector<Vec4f> tempCircles; // (x, y, radius, votes)
```

```cpp
for (int r = 0; r < radiusRange; r++) {
    int radius = r + minRadius;
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < cols; x++) {
            int votes = houghSpace[r].at<int>(y, x);
            if (votes > threshold) {
                // Yerel maksimum kontrolü - daha geniş alan (5x5)
                bool isLocalMax = true;

                int windowSize = 5; // Pencere boyutu
                int halfWindow = windowSize / 2;

                // Aynı yarıçap için daha büyük pencerede maksimum kontrolü
                for (int ny = max(0, y - halfWindow); ny <= min(rows - 1, y + halfWindow) && isLocalMax; ny++) {
                    for (int nx = max(0, x - halfWindow); nx <= min(cols - 1, x + halfWindow) && isLocalMax; nx++) {
                        if ((ny != y || nx != x) && houghSpace[r].at<int>(ny, nx) > votes) {
                            isLocalMax = false;
                        }
                    }
                }

                // Farklı yarıçaplar için de kontrol
                if (isLocalMax) {
                    for (int nr = max(0, r - 2); nr <= min(radiusRange - 1, r + 2) && isLocalMax; nr++) {
                        if (nr != r) {
                            // Civardaki yarıçaplarda daha güçlü bir merkez var mı?
                            if (houghSpace[nr].at<int>(y, x) > votes) {
                                isLocalMax = false;
                            }
                        }
                    }
                }

                // Eğer yerel maksimumsa, geçici listeye ekle
                if (isLocalMax) {
                    tempCircles.push_back(Vec4f(x, y, radius, votes));
                }
            }
        }
    }
}
```

```cpp
    sort(tempCircles.begin(), tempCircles.end(),
        [](const Vec4f& a, const Vec4f& b) { return a[3] > b[3]; });

    // Yakın daireleri birleştir
    for (size_t i = 0; i < tempCircles.size(); i++) {
        bool shouldAdd = true;

        // Bu merkez daha önce eklenen başka bir daireye çok yakın mı?
        for (size_t j = 0; j < circles.size(); j++) {
            float centerDist = sqrt(pow(tempCircles[i][0] - circles[j][0], 2) +
                pow(tempCircles[i][1] - circles[j][1], 2));
            float radiusDiff = abs(tempCircles[i][2] - circles[j][2]);

            // Eğer merkezler yeterince yakın ve yarıçaplar benzer ise
            if (centerDist < tempCircles[i][2] * 0.5 && radiusDiff < tempCircles[i][2] * 0.5) {
                shouldAdd = false;
                break;
            }
        }

        if (shouldAdd) {
            circles.push_back(Vec3f(tempCircles[i][0], tempCircles[i][1], tempCircles[i][2]));
        }
    }

    // En iyi N daire ile sınırla
    const int maxCircles = 12;
    if (circles.size() > maxCircles) {
        circles.resize(maxCircles);
    }

    return circles;
}
```

detectCircle fonksiyonu, bir görüntüdeki daireleri tespit etmek için Hough daire dönüşümünü uygular. Görüntüdeki her kenar pikseli için, farklı yarıçaplarda ve açılarda olası daire merkezleri hesaplanır ve bu merkezlere oy verilir. Oy sayısı eşik değerini geçen noktalar için, komşularda yerel maksimum olup olmadıkları kontrol edilir. Yerel maksimumlar geçici listeye alınır, ardından oy sayısına göre sıralanır. Yakın merkezli ve benzer yarıçaplı daireler birleştirilerek tekrar eden tespitler engellenir.


```cpp
void drawHoughCircles(Mat& image, const vector<Vec3f>& circles) {
    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]);
        // Merkezi çiz
        circle(image, center, 3, Scalar(0, 255, 0), -1, 8, 0);
        // Daireyi çiz
        circle(image, center, radius, Scalar(0, 0, 255), 2, 8, 0);
    }

    imshow("Detected Circles", image);
    waitKey(0);
    imwrite("resource/detected_circles.png", image);
}
```

Bu fonksiyon, tespit edilen daireleri verilen görüntü üzerine çizmek için kullanılır. Her bir daire için önce merkezi bir yeşil nokta olarak işaretlenir, ardından yarıçapına göre kırmızı renkli bir daire çizilir.