The Canny Edge Detector

Implementing Canny Edges from scratch

Here's an interesting article - we'll implement canny edges. We won't use any prepackaged functions. I'll be using OpenCV for this article, but I'm sure translating it to some other computer vision package won't be difficult.

> I assume you know how the algorithm works. If not, read up about the Canny edge detection algorithm (/tutorials/canny-edge-detector/)!

Series: The Canny Edge Detector:

1. Introduction the edge detector (/tutorials/canny-edge-detector/)
2. **Implementing Canny Edges from scratch**

Getting Started

We'll start off with the function definition:

```
Mat MyCanny(Mat src, int upperThreshold, int lowerThreshold, double size = 3)
{
```

We send in an image, specify the upper and lower threshold and specify the accuracy of the sobel edge detection with the size parameter (it is the size of the kernel used in sobel).

For this article, I won't be considering error checking and other unrelated things (checking image depth, channels, etc). I assume *src* has one channel (it's grayscale) and is in the CV_8UC1 format (each pixel is a single byte).

Preprocessing

If you want, you can put a Gaussian blur before you even start any work. Or you could simply clone *src*.

```
Mat workImg = Mat(src);

workImg = src.clone();

// Step 1: Noise reduction
cv::GaussianBlur(src, workImg, cv::Size(5, 5), 1.4);
```

Calculating gradients

Next, we'll calculate gradient magnitudes and orientations separately. We start with calculating the sobel of the image:

```
// Step 2: Calculating gradient magnitudes and directions
Mat magX = Mat(src.rows, src.cols, CV_32F);
Mat magY = Mat(src.rows, src.cols, CV_32F);
cv::Sobel(workImg, magX, CV_32F, 1, 0, size);
cv::Sobel(workImg, magY, CV_32F, 0, 1, size);
```

next, we calculate the slope at each point. This is simply dividing the Y derivative by X:

```
Mat direction = Mat(workImg.rows, workImg.cols, CV_32F);
cv::divide(magY, magX, direction);
```

Next we calculate the magnitude of the gradient at each pixel:

```
Mat sum = Mat(workImg.rows, workImg.cols, CV_64F);
Mat prodX = Mat(workImg.rows, workImg.cols, CV_64F);
Mat prodY = Mat(workImg.rows, workImg.cols, CV_64F);
cv::multiply(magX, magX, prodX);
cv::multiply(magY, magY, prodY);
sum = prodX + prodY;
cv::sqrt(sum, sum);
```

We're calculating $\sqrt{G_x^2 + G_y^2}$ for every point. Simple stuff.

Nonmaximum suppression

We need to figure out points that definitely lie on edges - points whose gradient magnitude is greater than upper threshold and are a maxima.

We'll start by initializing a few things:

```
    Mat returnImg = Mat(src.rows, src.cols, CV_8U);

    returnImg.setTo(cv::Scalar(0));         // Initialie image to return to zero

    // Initialize iterators
    cv::MatIterator_<float>itMag = sum.begin<float>();
    cv::MatIterator_<float>itDirection = direction.begin<float>();

    cv::MatIterator_<unsigned char>itRet = returnImg.begin<unsigned char>();

    cv::MatIterator_<float>itend = sum.end<float>();
```

returnImg is the image that will have all canny edges. We've initialized all iterators for the magnitude, direction and return images.

Now, we start going through each pixel:

```
    for(;itMag!=itend;++itDirection, ++itRet, ++itMag)
    {
        const cv::Point pos = itRet.pos();

        float currentDirection = atan(*itDirection) * 180 / 3.142;

        while(currentDirection<0) currentDirection+=180;

        *itDirection = currentDirection;

        if(*itMag<upperThreshold) continue;
```

We store the current pixel in *pos*. Calculate the gradient direction in terms of degrees (and make it positive) and store it back. Then, if the pixel's gradient magnitude is not high enough, we simple skip it. Now add this line:

```
        bool flag = true;
```

This boolean variables indicates if the current pixel is an edge or not. At the end, if it passes all tests, it will still be true and we mark the pixel as an edge. Next, we consider each bin of the direction:

```
        if(currentDirection>112.5 && currentDirection <=157.5)
        {
            if(pos.y>0 && pos.x<workImg.cols-1 && *itMag<=sum.at<float>(pos.y-1, pos.x+1)) flag = false;
            if(pos.y<workImg.rows-1 && pos.x>0 && *itMag<=sum.at<float>(pos.y+1, pos.x-1)) flag = false;
        }
```

If the gradient direction is between 112.5 and 157.5 degrees, then it is rounded off to 135 degrees. This means, the edge is from top left to bottom right. So you need to check the top right and bottom left pixels for the maxima condition - these are (x+1, y-1) and (x-1, y+1). This is done with the *sum.at* call. The extra if's ensure you don't go out of bounds. If the magnitude is lower than the neighbours, flag is set to false.

We do similar things with the other bins:

```
        else if(currentDirection>67.5 && currentDirection <= 112.5)
        {
            if(pos.y>0 && *itMag<=sum.at<float>(pos.y-1, pos.x)) flag = false;
            if(pos.y<workImg.rows-1 && *itMag<=sum.at<float>(pos.y+1, pos.x)) flag = false;
        }
        else if(currentDirection > 22.5 && currentDirection <= 67.5)
        {
            if(pos.y>0 && pos.x>0 && *itMag<=sum.at<float>(pos.y-1, pos.x-1)) flag = false;
            if(pos.y<workImg.rows-1 && pos.x<workImg.cols-1 && *itMag<=sum.at<float>(pos.y+1, pos.x+1)) fl
        }
        else
        {
            if(pos.x>0 && *itMag<=sum.at<float>(pos.y, pos.x-1)) flag = false;
            if(pos.x<workImg.cols-1 && *itMag<=sum.at<float>(pos.y, pos.x+1)) flag = false;
        }
```

After this if-else ladder, we know if the current pixel is a definite edge or not. *flag* indicates this. And based on this, we mark the pixel as white in the return image:

```
        if(flag)
        {
            *itRet = 255;
        }
    }
```

This completes the loop for step 3.

Now we move onto the tricky part of the algorithm.

Thresholding with hysteresis

We keep a flag to indicate if the image was changed in the previous iteration or not. If it was, we need to check again. If not, we've got our final image.

```
    // Step 4: Hysterysis threshold
    bool imageChanged = true;
    int i=0;
    while(imageChanged)
    {
        imageChanged = false;
        printf("Iteration %d\
", i);
        i++;
```

At the beginning of the loop, we set *imageChanged* to false. If new edge pixels are found, this will be changed to true. Next, we initialize iterators again:

```
        itMag = sum.begin<float>();
        itDirection = direction.begin<float>();
        itRet = returnImg.begin<unsigned char>();
        itend = sum.end<float>();
        for(;itMag!=itend;++itMag, ++itDirection, ++itRet)
        {
```

Next, extract the current pixel's gradient direction and position:

```
            cv::Point pos = itRet.pos();
            if(pos.x<2 || pos.x>src.cols-2 || pos.y<2 || pos.y>src.rows-2)
                continue;
            float currentDirection = *itDirection;
```

If we've near the edges, we skip those pixels. A convention we're using for this part is - a pixel with intensity 255 means a 'fresh' edge pixel. It's neighbors have not been checked. After they've been checked, their intensity is changed to 64. Both numbers are arbitrary. I could have selected 178 instead of 64 as well.

```
            // Do we have a pixel we already know as an edge?
            if(*itRet==255)
            {
                *itRet=(unsigned char)64;
```

If a pixel is 255, we change it to a 64 first. Then we check the directions and process them like this:

```
            if(currentDirection>112.5 && currentDirection <= 157.5)
            {
                if(pos.y>0 && pos.x>0)
                {
                    if(lowerThreshold<=sum.at<float>(pos.y-1, pos.x-1) &&
                    returnImg.at<unsigned char>(pos.y-1, pos.x-1)!=64 &&
                    direction.at<float>(pos.y-1, pos.x-1) > 112.5 &&
                    direction.at<float>(pos.y-1, pos.x-1) <= 157.5 &&
                    sum.at<float>(pos.y-1, pos.x-1) > sum.at<float>(pos.y-2, pos.x) &&
                    sum.at<float>(pos.y-1, pos.x-1) > sum.at<float>(pos.y, pos.x-2))
                    {
                        returnImg.ptr<unsigned char>(pos.y-1, pos.x-1)[0] = 255;
                        imageChanged = true;
                    }
                }
                if(pos.y<workImg.rows-1 && pos.x<workImg.cols-1)
                {
                    if(lowerThreshold<=sum.at<float>(cv::Point(pos.x+1, pos.y+1)) &&
                    returnImg.at<unsigned char>(pos.y+1, pos.x+1)!=64 &&
                    direction.at<float>(pos.y+1, pos.x+1) > 112.5 &&
                    direction.at<float>(pos.y+1, pos.x+1) <= 157.5 &&
                    sum.at<float>(pos.y-1, pos.x-1) > sum.at<float>(pos.y+2, pos.x) &&
                    sum.at<float>(pos.y-1, pos.x-1) > sum.at<float>(pos.y, pos.x+2))
                    {
                        returnImg.ptr<unsigned char>(pos.y+1, pos.x+1)[0] = 255;
                        imageChanged = true;
                    }
                }
            }
```

That's a big piece of code! Let me explain what it does.

If the current pixel's direction is between 112.5 and 157.5 (around 135). So the edge is from the top left to the bottom right. So you check if the pixels (x-1, y-1) and (x+1, y+1) are edge pixels or not. You do this by checking:

- The gradient magnitude at these points is greater than the lower threshold
- If that pixel hasn't already been checked (marked as 64)
- The direction at that pixel falls in the same bin (112.5 to 157.5)
- The condition used in nonmaximum suppression still holds
    - For (x-1, y-1) this means checking against (x, y-2) and (x-2, y)
    - For (x+1, y+1) this means checking against the (x, y+2) and (x+2, y)

If all these conditions hold, you mark the neighbour with a 255. This is a fresh edge pixel. And we also change *imageChanged* to true.

We do a similar processing for the other three bins as well:

```
                else if(currentDirection>67.5 && currentDirection <= 112.5)
                {
                    if(pos.x>0)
                    {
                        if(lowerThreshold<=sum.at<float>(cv::Point(pos.x-1, pos.y)) &&
                        returnImg.at<unsigned char>(pos.y, pos.x-1)!=64 &&
                        direction.at<float>(pos.y, pos.x-1) > 67.5 &&
                        direction.at<float>(pos.y, pos.x-1) <= 112.5 &&
                        sum.at<float>(pos.y, pos.x-1) > sum.at<float>(pos.y-1, pos.x-1) &&
                        sum.at<float>(pos.y, pos.x-1) > sum.at<float>(pos.y+1, pos.x-1))
                        {
                            returnImg.ptr<unsigned char>(pos.y, pos.x-1)[0] = 255;
                            imageChanged = true;
                        }
                    }
                    if(pos.x<workImg.cols-1)
                    {
                        if(lowerThreshold<=sum.at<float>(cv::Point(pos.x+1, pos.y)) &&
                        returnImg.at<unsigned char>(pos.y, pos.x+1)!=64 &&
                        direction.at<float>(pos.y, pos.x+1) > 67.5 &&
                        direction.at<float>(pos.y, pos.x+1) <= 112.5 &&
                        sum.at<float>(pos.y, pos.x+1) > sum.at<float>(pos.y-1, pos.x+1) &&
                        sum.at<float>(pos.y, pos.x+1) > sum.at<float>(pos.y+1, pos.x+1))
                        {
                            returnImg.ptr<unsigned char>(pos.y, pos.x+1)[0] = 255;
                            imageChanged = true;
                        }
                    }
                }
                else if(currentDirection > 22.5 && currentDirection <= 67.5)
                {
                    if(pos.y>0 && pos.x<workImg.cols-1)
                    {
                        if(lowerThreshold<=sum.at<float>(cv::Point(pos.x+1, pos.y-1)) &&
                        returnImg.at<unsigned char>(pos.y-1, pos.x+1)!=64 &&
                        direction.at<float>(pos.y-1, pos.x+1) > 22.5 &&
                        direction.at<float>(pos.y-1, pos.x+1) <= 67.5 &&
                        sum.at<float>(pos.y-1, pos.x+1) > sum.at<float>(pos.y-2, pos.x) &&
                        sum.at<float>(pos.y-1, pos.x+1) > sum.at<float>(pos.y, pos.x+2))
                        {
                            returnImg.ptr<unsigned char>(pos.y-1, pos.x+1)[0] = 255;
                            imageChanged = true;
                        }
                    }
                    if(pos.y<workImg.rows-1 && pos.x>0)
                    {
                        if(lowerThreshold<=sum.at<float>(cv::Point(pos.x-1, pos.y+1)) &&
                        returnImg.at<unsigned char>(pos.y+1, pos.x-1)!=64 &&
                        direction.at<float>(pos.y+1, pos.x-1) > 22.5 &&
                        direction.at<float>(pos.y+1, pos.x-1) <= 67.5 &&
                        sum.at<float>(pos.y+1, pos.x-1) > sum.at<float>(pos.y, pos.x-2) &&
                        sum.at<float>(pos.y+1, pos.x-1) > sum.at<float>(pos.y+2, pos.x))
                        {
                            returnImg.ptr<unsigned char>(pos.y+1, pos.x-1)[0] = 255;
                            imageChanged = true;
                        }
```

```
                    }
                }
                else
                {
                    if(pos.y>0)
                    {
                        if(lowerThreshold<=sum.at<float>(cv::Point(pos.x, pos.y-1)) &&
                        returnImg.at<unsigned char>(pos.y-1, pos.x)!=64 &&
                        (direction.at<float>(pos.y-1, pos.x) < 22.5 ||
                        direction.at<float>(pos.y-1, pos.x) >=157.5) &&
                        sum.at<float>(pos.y-1, pos.x) > sum.at<float>(pos.y-1, pos.x-1) &&
                        sum.at<float>(pos.y-1, pos.x) > sum.at<float>(pos.y-1, pos.x+2))
                        {
                            returnImg.ptr<unsigned char>(pos.y-1, pos.x)[0] = 255;
                            imageChanged = true;
                        }
                    }
                    if(pos.y<workImg.rows-1)
                    {
                        if(lowerThreshold<=sum.at<float>(cv::Point(pos.x, pos.y+1)) &&
                        returnImg.at<unsigned char>(pos.y+1, pos.x)!=64 &&
                        (direction.at<float>(pos.y+1, pos.x) < 22.5 ||
                        direction.at<float>(pos.y+1, pos.x) >=157.5) &&
                        sum.at<float>(pos.y+1, pos.x) > sum.at<float>(pos.y+1, pos.x-1) &&
                        sum.at<float>(pos.y+1, pos.x) > sum.at<float>(pos.y+1, pos.x+1))
                        {
                            returnImg.ptr<unsigned char>(pos.y+1, pos.x)[0] = 255;
                            imageChanged = true;
                        }
                    }
                }
            }
        }
    }
```

And that ends the loop for step 4. After the loop is over, all edge pixels are not 'stale' - they're all marked with 64. We need to change that. This is done by a quick little loop:

```
    cv::MatIterator_<unsigned char>current = returnImg.begin<unsigned char>();    cv::MatIterator_<unsigne
    for(;current!=final;++current)
    {
        if(*current==64)
            *current = 255;
    }
    return returnImg;
}
```

Finally we return the image and we're done!

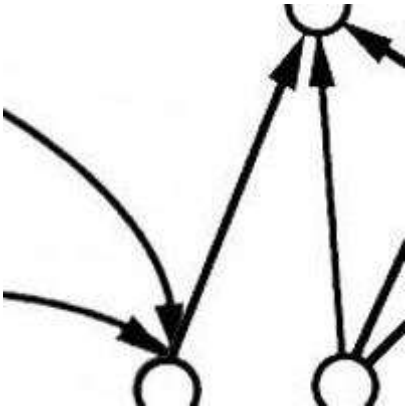
> The OpenCV Canny algorithm produces slightly different results. I'm not sure why that happens. If you figure out why, do let me know!

More in the series

This tutorial is part of a series called **The Canny Edge Detector**:

1. Introduction the edge detector (/tutorials/canny-edge-detector/)
2. **Implementing Canny Edges from scratch**

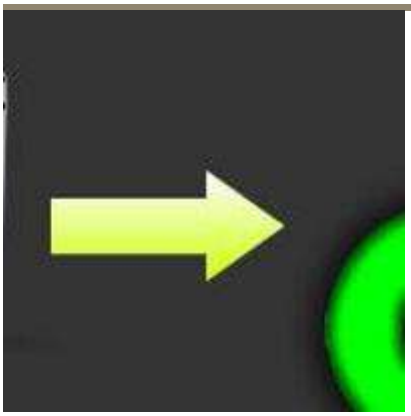
Related posts



(/tutorials/7-unique-neural-network-architectures/)

7 unique neural network architectures (/tutorials/7-unique-neural-network-architectures/)

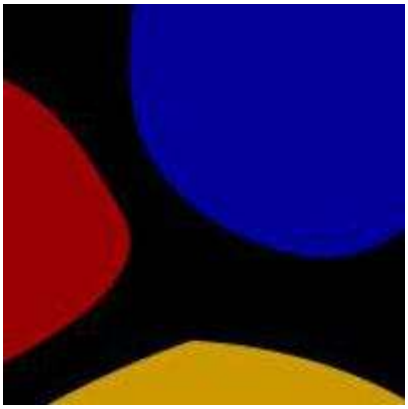
Read more (/tutorials/7-unique-neural-network-architectures/)



(/tutorials/capturing-images/)

Capturing images (/tutorials/capturing-images/)

Read more (/tutorials/capturing-images/)



(/tutorials/connected-component-labelling/)

Connected Component Labelling (/tutorials/connected-component-labelling/)

Read more (/tutorials/connected-component-labelling/)

(/tutorials/convolutions/)

Basics of convolutions (/tutorials/convolutions/)

Read more (/tutorials/convolutions/)

(http://utkarshsinha.com/)

Utkarsh Sinha created AI Shack in 2010 and has since been working on computer vision and related fields. He is currently at Microsoft working on computer vision.

ALSO ON AI SHACK

Obstacle avoidance with the Bug-1 algorithm

6 years ago • 1 comment

An obstacle avoidance algorithm to help your robot reach from one end of the ...

First artificial neurons: The McCulloch-Pitts ...

6 years ago • 1 comment

The McCulloch-Pitts model was an extremely simple artificial neuron. The ...

A super fast thresholding technique

6 years ago • 1 comment

Learn how to implement really fast thresholding - faster than OpenCV! This ...

An introc contours

6 years ago

Introducti you'll get use conto

5 Comments

1 Login ▼

G

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

♡ Share

Best Newest Oldest

Y

Yang

2 months ago

It's really different from the results of opencv, and I also want to know how the canny algorithm in opencv is implemented.

0 0 Reply • Share ›

About AI Shack

Learn about the latest in AI technology with in-depth tutorials on vision and learning!

Follow @utkarshsinha

More... (/about/)

Get started

Get started with OpenCV (/tracks/opencv-basics/)

Track a specific color on video (/tutorials/tracking-colored-objects-opencv/)
Learn basic image processing algorithms (/tracks/image-processing-algorithms-level-1/)
How to build artificial neurons? (/tutorials/single-neuron-dictomizer/)
Look at some source code (http://github.com/aishack/)

AI Shack

Like Page 3.6K likes

Created by Utkarsh Sinha (http://utkarshsinha.com/)