Algorithm for drawing outline of image shape in BezierPath (Canny edge detector)

Asked 11 days ago Modified 8 days ago Viewed 69 times

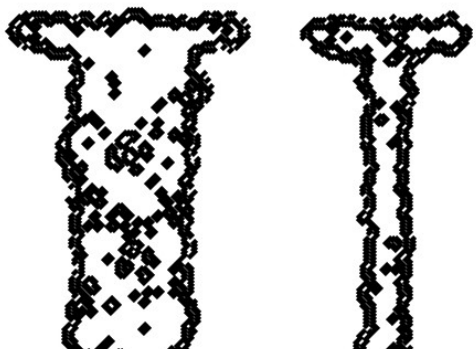
3

I'm trying to draw the outline of an image using BezierPath based on the transparency of each pixel.

However, I'm having an issue with the logic; my logic also draws the internal outlines.

I only want to draw the external outline with BezierPath. What I get (the first shape is the original image, the second is the `bezierPath`):

100



My code:

```
func processImage(_ image: UIImage) -> UIBezierPath? {
    guard let cgImage = image.cgImage else {
        print("Error: Couldn't get CGImage from UIImage")
        return nil
    }

    let width = cgImage.width
    let height = cgImage.height

    // Create a context to perform image processing
    let colorSpace = CGColorSpaceCreateDeviceGray()
    let context = CGContext(data: nil, width: width, height: height, bitsPerComponent:
8, bytesPerRow: width, space: colorSpace, bitmapInfo: CGImageAlphaInfo.none.rawValue)

    guard let context = context else {
        print("Error: Couldn't create CGContext")
        return nil
    }

    // Draw the image into the context
    context.draw(cgImage, in: CGRect(x: 0, y: 0, width: width, height: height))

    // Perform Canny edge detection
    guard let edgeImage = context.makeImage() else {
        print("Error: Couldn't create edge image")
        return nil
    }

    // Create a bezier path for the outline of the shape
    let bezierPath = UIBezierPath()

    // Iterate over the image pixels to find the edges
    for y in 0..<height {
        for x in 0..<width {
            let pixel = edgeImage.pixel(x: x, y: y)

            if pixel > 0 {
                let leftPixel = (x > 0) ? edgeImage.pixel(x: x - 1, y: y) : 0
                let rightPixel = (x < width - 1) ? edgeImage.pixel(x: x + 1, y: y) : 0
                let abovePixel = (y > 0) ? edgeImage.pixel(x: x, y: y - 1) : 0
                let belowPixel = (y < height - 1) ? edgeImage.pixel(x: x, y: y + 1) : 0

                if leftPixel == 0 || rightPixel == 0 || abovePixel == 0 || belowPixel ==
0 {
```

```
                    bezierPath.move(to: CGPoint(x: CGFloat(x), y: CGFloat(y)))
                    bezierPath.addLine(to: CGPoint(x: CGFloat(x) + 1.0, y: CGFloat(y) +
1.0))
                }
            }
        }
    }

    return bezierPath
}

extension CGImage {
    func pixel(x: Int, y: Int) -> UInt8 {
        let data = self.dataProvider!.data
        let pointer = CFDataGetBytePtr(data)
        let bytesPerRow = self.bytesPerRow

        let pixelInfo = (bytesPerRow * y) + x
        return pointer![pixelInfo]
    }
}
```

ios swift algorithm canny-operator Edit tags

Share Edit Follow Close Flag

edited Sep 10 at 16:52
HangarRash
7,472 5 6 32

asked Sep 10 at 13:41
Mickael Belhassen
3,012 1 25 46

You want a flood fill algorithm, starting outside the image to find everything "not" in the shape. Then edge detect on that. en.wikipedia.org/wiki/Flood_fill There are many other approaches, but that one is pretty easy to implement and will work reasonably for many things as long as there are no gaps in your edge. You can also find the fist pixel of an edge as you are, and then try "walking around it" by testing nearby pixels in all directions. – Rob Napier Sep 10 at 14:27

Note that generating thousands of two-pixel lines can be very expensive to work with. You probably will want to simplify the final curve using something like Ramer–Douglas–Peucker (possibly before creating an actual BezierPath). – Rob Napier Sep 10 at 14:38

---

1 That said, how would you want this to work if the above font were the letter P? I would assume there would be a large internal hole that you **would** want to treat as edges, even though you don't want to capture the other, smaller internal holes. Solving this well is likely complex, and require you to detect "features" rather than just adjacent pixels. One approach is to scan with a larger (3x3, 5x5, on that order) overlapping "block" that is considered filled in if any of its pixels are filled in. That will ignore small holes, while capturing larger holes. It's challenging. – Rob Napier Sep 10 at 14:44

---

1 Thank you for your response. I will try to implement the flood fill algorithm; it seems to be what I'm looking for. Indeed, I will probably need to simplify the paths, which I will do in step 2. And to address the example of the letter P, I'm not interested in the hole inside for my goal, only the external outline. I will keep you updated. Thank you very much! – Mickael Belhassen Sep 11 at 9:07

---

1 My goal is to get out the the outline as BezierPath, after more investigation I need to use another algorithm Moore-Neighbor tracing algorithm – Mickael Belhassen Sep 12 at 6:24

---

1 Answer

Sorted by:
Reset to default

Date modified (newest first)

3

From the comments, you found the algorithm (Moore Neighborhood Tracing). Here's an implementation that works well for your problem. I'll comment on some improvements you might consider.

First, you need to get the data into a buffer, with one byte per pixel. You seem to know how to do that, so I won't belabor the point. 0 should be "transparent" and non-0 should be "filled." In the literature, these are often black (1) lines on a white (0) background, so I'll use that naming.

The best introduction I've found (and regularly cited) is Abeer George Ghuneim's Contour Tracing site. Really useful site. I've seen some implementations of MNT that over-check some pixels. I try to follow the algorithm Abeer describes carefully to avoid that.

There is more testing I want to do on this code, but it handles your case.

First, the algorithm operates on a Grid of Cells:

```
public struct Cell: Equatable {
    public var x: Int
    public var y: Int
}

public struct Grid: Equatable {
    public var width: Int
    public var height: Int
    public var values: [UInt8]

    public var columns: Range<Int> { 0..<width }
    public var rows: Range<Int> { 0..<height }

    // The pixels immediately outside the grid are white.
    // Accessing beyond that is a runtime error.
    public subscript (p: Cell) -> Bool {
        if p.x == -1 || p.y == -1 || p.x == width || p.y == height { return false }
        else { return values[p.y * width + p.x] != 0 }
    }

    public init?(width: Int, height: Int, values: [UInt8]) {
        guard values.count == height * width else { return nil }
        self.height = height
        self.width = width
        self.values = values
    }
}
```

There is also the concept of "direction." This is in two forms: the direction from the center to one of the 8 neighbors, and the "backtrack" direction, which is the direction a cell is "entered" during the search

```
enum Direction: Equatable {
    case north, northEast, east, southEast, south, southWest, west, northWest

    mutating func rotateClockwise() {
        self = switch self {
        case .north: .northEast
        case .northEast: .east
        case .east: .southEast
        case .southEast: .south
        case .south: .southWest
        case .southWest: .west
        case .west: .northWest
        case .northWest: .north
        }
    }

    //
    // Given a direction from the center, this is the direction that box was entered
from when
    // rotating clockwise.
    //
    // +---+---+---+
    // + ↓ + ← + ← +
    // +---+---+---+
    // + ↓ +   + ↑ +
    // +---+---+---+
    // + → + → + ↑ +
    // +---+---+---+
    func backtrackDirection() -> Direction {
        switch self {
        case .north: .west
        case .northEast: .west
        case .east: .north
        case .southEast: .north
        case .south: .east
        case .southWest: .east
        case .west: .south
        case .northWest: .south
        }
    }
}
```

And Cells can advance in a given direction:

```
extension Cell {
    func inDirection(_ direction: Direction) -> Cell {
        switch direction {
        case .north:     Cell(x: x,     y: y - 1)
        case .northEast: Cell(x: x + 1, y: y - 1)
        case .east:      Cell(x: x + 1, y: y    )
        case .southEast: Cell(x: x + 1, y: y + 1)
        case .south:     Cell(x: x    , y: y + 1)
        case .southWest: Cell(x: x - 1, y: y + 1)
        case .west:      Cell(x: x - 1, y: y    )
        case .northWest: Cell(x: x - 1, y: y - 1)
        }
    }
}
```

And finally, the Moore Neighbor algorithm:

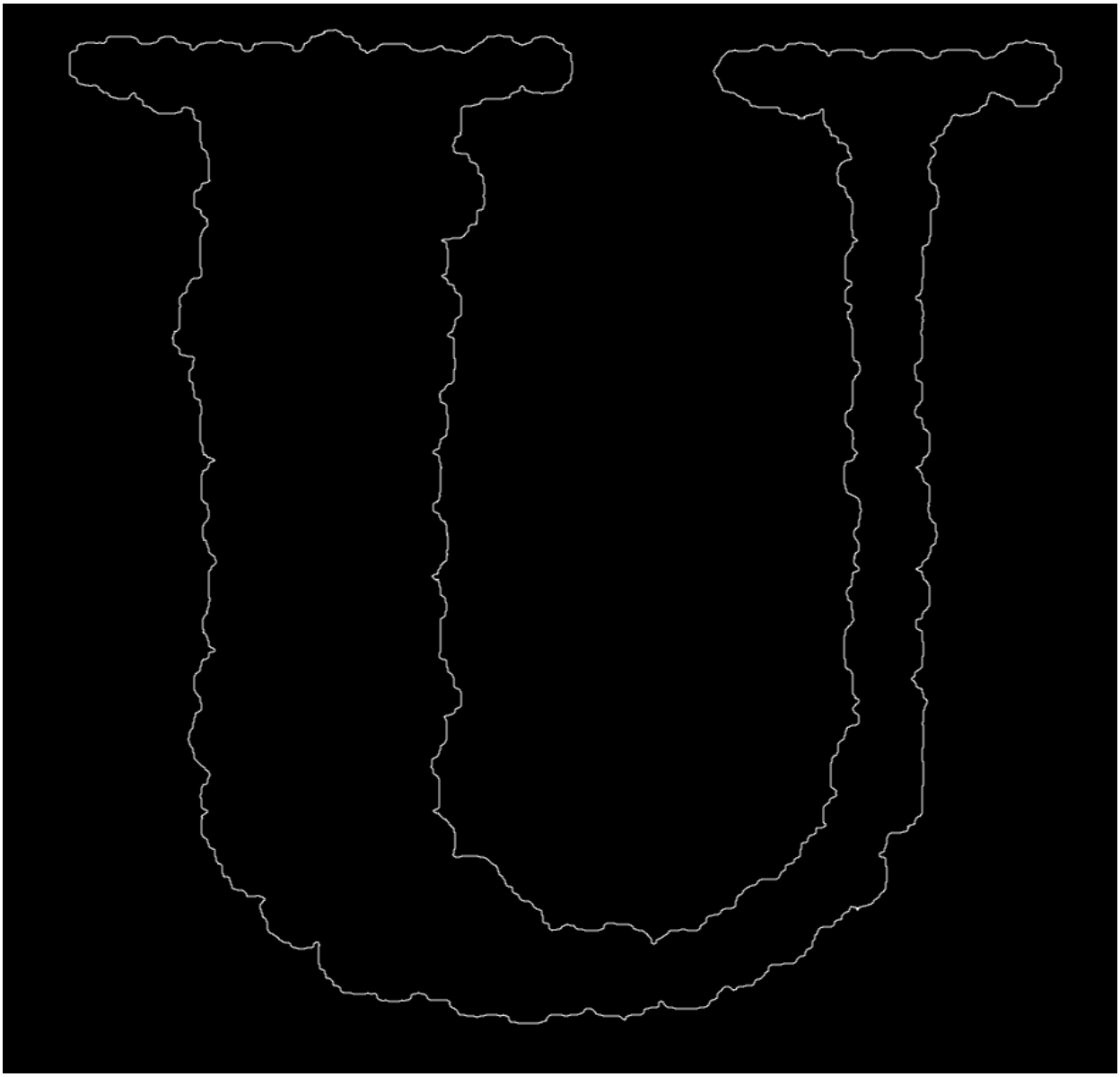
```
public struct BorderFinder {
    public init() {}

    // Returns the point and the direction of the previous point
    // Since this scans left-to-right, the previous point is always to the west
    // The grid includes x=-1, so it's ok if this is an edge.
    func startingPoint(for grid: Grid) -> (point: Cell, direction: Direction)? {
        for y in grid.rows {
            for x in grid.columns {
                let point = Cell(x: x, y: y)
                if grid[point] {
                    return (point, .west)
                }
            }
        }
        return nil
    }

    /// Finds the boundary of a blob within `grid`
    ///
    /// - Parameter grid: an Array of bytes representing a 2D grid of UInt8. Each cell
is either zero (white) or non-zero (black).
    /// - Returns: An array of points defining the boundary. The boundary includes only
black points.
    ///            If multiple "blobs" exist, it is not defined which will be returned.
    ///            If no blob is found, an empty array is returned
    public func findBorder(in grid: Grid) -> [Cell] {
        guard let start = startingPoint(for: grid) else { return [] }
        var (point, direction) = start
        var boundary: [Cell] = [point]

        var rotations = 0
        repeat {
            direction.rotateClockwise()
            let nextPoint = point.inDirection(direction)
            if grid[nextPoint] {
                boundary.append(nextPoint)
                point = nextPoint
                direction = direction.backtrackDirection()
                rotations = 0
            } else {
                rotations += 1
            }
        } while (point, direction) != start && rotations <= 7

        return boundary
    }
}
```

This returns a list of Cells. That can be converted to a CGPath as follows:

```
let data = ... Bitmap data with background as 0, and foreground as non-0 ...
let grid = Grid(width: image.width, height: image.height, values: Array(data))!

let points = BorderFinder().findBorder(in: grid)

let path = CGMutablePath()
let start = points.first!

path.move(to: CGPoint(x: start.x, y: start.y))
for point in points.dropFirst() {
    let cgPoint = CGPoint(x: point.x, y: point.y)
    path.addLine(to: cgPoint)
```

```
}
path.closeSubpath()
```

That generates the following path:



There is a gist of the full sample code I used. (This sample code isn't meant to be a good example of how to prep the image for processing. I just threw it together to work on the algorithm.)

Some thoughts for future work:

- You can likely get good and faster results by first scaling the image to something smaller. Half-scale definitely works well, but consider even 1/10 scale.
- You may get better results by first applying a small Gaussian blur to the image. This will eliminate small gaps in the edge, which can cause trouble for the algorithm, and reduce the complexity of the contour.
- Managing 5000 path elements, each a 2-pixel line, is probably not great. Pre-scaling the image can help a lot. Another approach is applying Ramer–Douglas–Peucker to further simplify the contour.

Share Edit Follow Flag

edited Sep 13 at 13:52

answered Sep 12 at 13:54

Rob Napier
287k 34 457 611

Works perfectly, thank you for your help. You just forgot the 'inDirection' method in your response. But I had understood it: `func inDirection(_ direction: Direction) -> Cell { switch direction { case .north: return Cell(x: x, y: y - 1).... –` Mickael Belhassen Sep 13 at 10:53

Oh yes! Sorry about that. Added. – Rob Napier Sep 13 at 13:52