Midpoint Circle Algorithm

Explanation and documentation

Donovan Lay

September 29, 2021

For this second assignment I decided to use C++ as I did in the last assignment. After debating which library to use Simple Directmedia Layer 2 (SDL2). For whatever reason, SDL2 does not include a Stroke or Fill Circle function, it has a rectangle, points, and a line. As odd as this, this meant it was time to learn. How do you draw a circle? Well, after looking I found that there is something called the Midpoint Circle Algorithm, specifically using the integer-based algorithm since we're dealing with pixels. From the article I'll put below we start with a definition for the radius error for any given point on a circle:

$$RE(x_i, y_i) = |x_i^2 + y_i^2 - r^2|$$

This error comes from the fact that we'll be calculating a circle from integers only. The article only shows the solution for a circle centered at the origin, but we can generalize the equation for any center, though first I'll note that to simplify things I will be dropping the *i's* from the x's and y's, but assume they are there. The general equation we would start with is:

$$RE(x, y) = (x - h)^2 + (y - k)^2 - r^2$$

Great, so now that we have this, how can we draw a circle from it? Well essentially what we want is to check if the next integer point that we calculate would be inside or outside of the actual circle, and from that we can determine where to draw said point. The premise is that we are only going to calculate a small portion of the circle, and then reflect the points across the center to draw the full circle. In the article, when starting at the origin, we start with an assumption of constantly increasing y, and occasionally increasing x based on a calculated parameter, the radius error previously mentioned. So the next drawn point should be either (x, y + 1), or (x – 1, y + 1), those are the only two possible options. We could also choose to start at the top of the circle instead, and constantly increase x and occasionally decrease y, this would give the same effect (in fact later on we'll see that for a general solution, we'll have to do that).

Now, to decide between the two points before, the article suggest that we check if the following is true:

$$RE(x - 1, y + 1) < RE(x, y + 1)$$

This will help us determine which point to draw. To determine if the inequality holds, we'll expand it and generalize it:

$$|(x - h - 1)^2 + (y - k + 1)^2 - r^2| < |(x - h)^2 + (y - k + 1)^2 - r^2|$$

Now I will use a few tricks recommended in the article. The absolute values here are useless to us, so we will square both sides since a square will always be positive. So now we have

$$[(x - h - 1)^2 + (y - k + 1)^2 - r^2]^2 < [(x - h)^2 + (y - k + 1)^2 - r^2]^2$$

Now comes a bunch of algebra, I won't explain each step, but I will write them out, you should be able to follow along. I'll start by expanding the inside components. Left hand side:

$$[(x^2 - 2xh - 2x + 2h + h^2 + 1) + (y^2 - 2yk + 2y - 2k + k^2 + 1) - r^2]^2$$

Right hand side:

$$[(x^2 - 2xh + h^2) + (y^2 - 2yk + 2y - 2k + k^2 + 1) - r^2]^2$$

There's a lot of components in the left hand side that are similar to the right hand side. Let's group the left hand side up to look like the right hand side.

$$[(x^2 - 2xh + h^2 + y^2 - 2yk + 2y - 2k + k^2 + 1 - r^2) + (1 + 2h - 2x)]^2$$

Now if we let the right hand side be equal to A and let (1+2h-2x) be B, we have:

$$(A+B)^2 < A^2$$

The insides matter much less at this point. So we can square out the left hand side and get

$$A^2 + 2AB + B^2 < A^2$$
$$2AB + B^2 < 0$$

Now we'll divide B out, which remember is (1+2h-2x). Since we're always going to be dealing with a positive h and x (as we're dealing with pixels), this number will always be less than 0, though we should always make sure it is *not equal* to zero or we'd be dividing by zero here, and the dividing by a negative will flip the less than sign

$$2A + B > 0$$

Or

$$2(x^2 - 2xh + h^2 + y^2 - 2yk + 2y - 2k + k^2 + 1 - r^2) + (1 + 2h - 2x) > 0 \; if \; x - h \neq \frac{1}{2}$$

So what this says, is that *if* this number, the radius error (RE) is greater than 0, then we will increase y and decrease x, otherwise we will just increase y. Unfortunately this isn't the end of the story though. Now we need the actual points to plot. We need 8 points (for the 8 octants of a circle), but it will actually be 4 duplicates since they'll start at the same 4 points and go different ways from there. You might initially think the points we should start with are:

$$x_{right} = h + radius$$
$$x_{left} = h - radius$$
$$y_{right} = k$$
$$y_{left} = k$$
$$Points: (x_{left}, y_{left}), (x_{left}, y_{right}), (x_{right}, y_{left}), (x_{right}, y_{right})$$

And then just flip the x and y on those points to get the other 4 points, right x and right y would work opposite x left and y left (ie. if x left decreases, x right increases), and we'll do this while x > y, and actually you would be right… if h=k, but h isn't always the same as k. So that means we need to generalize even further. So what are the other points? Well, currently we're only using the left and right sides, what if we use the top and bottom?

$$x_{top} = h$$
$$x_{bottom} = h$$
$$y_{top} = k + r$$
$$y_{bottom} = k - r$$

Now we have 4 more combinations to plot. If you try to plot this with the information we have currently, it still won't work. The reason is this: before we were *always* starting with the left and right points, moving up/down on y and sometimes moving on x. Now that we're starting at the top and bottom we need to do the opposite: always increase the x, and sometimes decrease the y (moving opposite directions with top and bottom, same as we did with left and right), and for this part we'll loop while y > x. Now, finally we can get a circle where h and k are not the same! You can try out a bunch of values and find that you will always get a circle… Unless h is greater than k. Again, another generalization we will have to make. Remember how originally we were checking if RE > 0? Well, even I'm not 100% sure how, but this isn't exactly accurate. What we need to do is a little bit of educational guessing. See, here we're checking if RE > 0, and we know this is only working for h <= k, which hints that we need to take h and k into effect. How can we get 0 from h and k if they're equal? Well, subtract them obviously. So now instead of checking RE > 0, what if we check

$$RE > |h - k|$$

This seems it could work, we'll always have a value greater than or equal to 0. There are still some other changes to make. In the original loop, we were looping while x > y, but we need to take h and k into effect here, and after testing, it *does* matter which is greater, and remember we have 2 different loops that are working opposite each other, so we're going to have 4 different equations depending on which is greater. These equations, after some educational guesses, turn out to be

$$Left - Right\ loop: x_{right} > y_{left} + |h - k| \;\; if\ h \geq k$$
$$Top - Bottom\ loop: y_{bottom} > x_{top} - |h - k| \;\; if \;\; h \geq k$$
$$LR\ Loop: x_{right} > y_{left} - |h - k| \;\; if\ h < k$$

$$TB\ Loop\colon y_{bottom} > x_{top} + |h - k| \quad if\ h < k$$

Finally, we have a solution that covers all of our bases! Below, you can see what pseudocode for a circle stroke would look like using SDL2, but you can also see my full implementation in my repo. In order to keep in line with how SDL implemented their other draw functions, SDL_StrokeCircle should return an int (not sure why).

```
#include “SDL.h”
#include “SDL_Circle.h”

int SDL_StrokeCircle(SDL_Renderer *renderer)
{
        WhichCheck = h>= k;
        if(WhichCheck)
        {
                CheckLeft = XRIGHT > YLEFT + abs(h-k);
                CheckTop = YBOTTOM > XTOP – abs(h-k);
        }
        else
        {
                CheckLeft = XRIGHT > YRIGHT – abs(h-k);
                CheckTop = YBOTTOM > XTOP + abs(h-k);
        }
        while(CheckLeft)
        {
                RE = RadiusError(XRIGHT, YLEFT);
                --YRIGHT;
                ++YLEFT;
                if(RE > abs(h-k))
                {
                        --XRIGHT;
                        ++XLEFT;
                }
                if(WhichCheck)
                {
                        //recalculate CheckLeft using same eq as before
                }
                else
                {
                        //recalculate CheckLeft using same eq as before
                }
        }
        while(CheckTop)
        {
                //do similar but opposite as while(CheckLeft) as described in document above
                …
        }
        Return -1;
}
```

For the fill circle, I did a similar thing but tried two different things: The first was to draw circles with a decreasing radius down to the center of the circle, the other way was to draw lines from the center out to the edges. The issue with this is that is leaves gaps. A much simpler implementation (Andy’s recommendation) was to use Dijkstra’s Algorithm to create a fill. This worked out quite well. I left the old fill function in, but it should be considered deprecated and Dijkstra’s Algorithm should be used instead (but still cool to see the learning process).

References:

Midpoint Circle Algorithm: https://en.wikipedia.org/wiki/Midpoint_circle_algorithm
Dijkstra’s Algorithm: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm