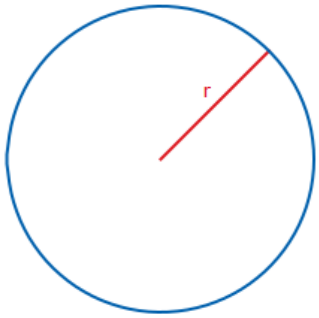
Everett Gaius Vergara

I'm a Programmer and this is my Programming Journal (This website is not yet live)

How to Draw a Circle (Midpoint and Bresenham's Algorithm)

Submitted by Everett Gaius on Sun, 07/05/2020 - 10:26



With trigonometry functions such as sine and cosine, it is very easy to compute for the value of x and y for any given radius and teta.

```
px = cx + radius * cosine(0);
py = cy + radius * sine(0);

for (teta = 1; teta <= 360; ++teta)
{
    px = cx + radius * cosine(teta);
    py = cy + radius * sine(teta);
    line(px, py, x, y);
    px = x;
    py = y;
}
```

We've all learned this from highschool trigonometry class and it will not be tacked in this article. However, drawing a circle using these functions requires the usage of floating-point numbers which is slow, not to mention the typecasting from floating-point to integers as plotting a pixel requires integral datatype. And finally, the big turnoff of this script is using the line segment function to connect the dots.

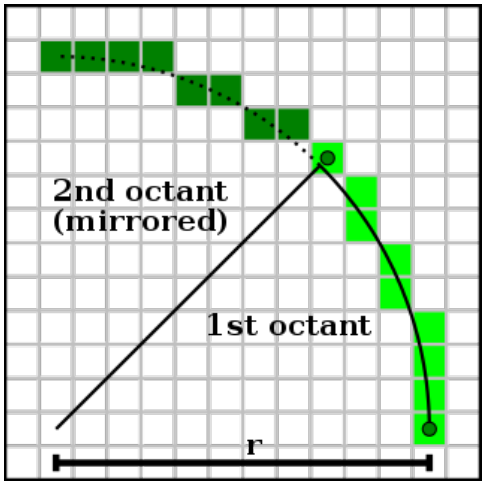
You can improve the above code by reducing the computational complexity by dividing the circle to quadrants, octants, or whatever.

Midpoint Circle Algorithm

Understanding the algorithm starts with the circle formula

- $x^2 + y^2 = r^2$

For simplicity of computation, we don't have to compute for the x and y of the entire circle given r. We can divide it into quadrants, octants, or whatever suits your preference. In our case, we will divide the circle by 8, which means we only need to compute 0 degrees to 45 degrees. Also to further simplify, our circle will be centered at coordinates (0, 0).



Notice on the first octant, that y continually increases while x_{i+1} is either on the same position as x_i or x_i - 1. We'll consider y as fast direction and x as the slow direction in this case. This means that x decreases by some factor of the circle. But how do we get x based on the previous values of x and y?

For each point in the circle, the following formula holds, where i is the iteration starting from 0.

- $x^2_i + y^2_i = r^2$

To get the x we can rearrange the above to:

- $x^2_i = r^2 - y^2_i$, the next point would be $x^2_{i+1} = r^2 - y^2_{i+1}$

Since y is the fast direction, y increases for every iteration in the first octant, hence:

- $y^2_{i+1} = (y_i + 1)^2$

Let's substitute this to x^2_{i+1}

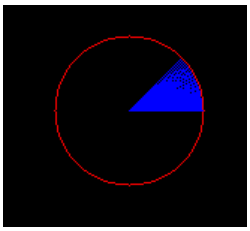
- $x^2_{i+1} = r^2 - (y_i + 1)^2$
- $x^2_{i+1} = r^2 - y^2_i - 2y_i - 1$, we can simplify by the computation by replacing $r^2 - y^2_i$ by x^2_i, which gives us:
- $x^2_{i+1} = x^2_i - 2y_i - 1$
- $x_{i+1} = \sqrt{[\, x^2_i - 2y_i - 1\,]}$

Now that we a formula of x that is dependent on the previous value, we can start writing our program. The following is written in C/C++ and SDL (actually all other programs below are written using the same language and framework):

```
void drawCircle1(SDL_Surface* surface, Sint32 cx, Sint32 cy, Uint32 r, Uint32 rgba)
{
    Sint32 x, x2, xn, y, xsurface, ysurface;

    // Center position the pointer
    Uint32* pixels = (Uint32*)surface->pixels + cx + (cy * surface->w);

    // Starts plotting (0, 0)
    x2 = r * r;
    x = sqrt(x2);
    y = 0;

    while (x > y)
    {
        xn = (x2 - (2 * y) - 1);
        x = sqrt(x2);
        xsurface = x * surface->w;
```

```
        ysurface = y * surface->w;

        // First Octant
        *(pixels + x - (ysurface)) = rgba;

        // The rest of the octants
        *(pixels + y - (xsurface)) = rgba;
        *(pixels - y - (xsurface)) = rgba;
        *(pixels - x - (ysurface)) = rgba;
        *(pixels + x + (ysurface)) = rgba;
        *(pixels + y + (xsurface)) = rgba;
        *(pixels - y + (xsurface)) = rgba;
        *(pixels - x + (ysurface)) = rgba;

        x2 = xn;
        ++y;
    }
}
```

Output:



It is important to take note that the upper left coordinates of the screen is (0, 0), and x and y increase as we move southeastwards, therefore adjustments are made to plot the first octant in the cartesian plane. The rest of the octants are computed accordingly.

Although the above script works, it could have been more elegant if floating-point computation such as square root, and typecasting have been eliminated.

Bresenham's Derivation of Drawing a Circle

To completely remove the floating-point computation, we need to start with the so-called Radius Error (RE). RE is can be thought of as a deviation of computation of points. If we've detected a deviation then we need to do correction in x position (assuming we're working on the first octant). If we're starting again with the first octant at (r, 0), we can say that RE_i, where i = 0, is zero (0), because we're sure that x = r and y = 0 at this point, and we can plot these two values using integral data type, hence we can write:

- $RE(x_i, y_i) = | x^2_i + y^2_i - r^2 | = 0$, where | | denotes absolute value.

when i = 0, on first octant,

- $RE(x_i, y_i) = | x^2_i + 0 - r^2 | = 0$

Just like the Midpoint algorithm, and since we're starting with the first octant, we know that y is the fast direction and x is the slower. Therefore, as y increments, x can either stay in the same position or decrement by one (1). To give a solution to this, we define a decision statement to determine if we need to retain the value of x or decrement the value of x as y increases.

$RE(x_i - 1, y_i + 1) < RE(x_i, y_i + 1)$, if the value of the left hand side is less than the right, then we plot $RE(x_i - 1, y_i + 1)$, otherwise $RE(x_i, y_i + 1)$.

To start the determination, let's substitute the value of RE. This gives us the following equation:

- $= | (x_i - 1)^2 + (y_i + 1)^2 - r^2 | \quad < \quad | x_i^2 + (y_i + 1)^2 - r^2 |$

Expanding the equation gives us

- = | $(x^2_i - 2x_i + 1) + (y^2_i + 2y_i + 1) - r^2$ | < | $x_i^2 + (y^2_i + 2y_i + 1) - r^2$ |

Rearranging will give us

- = | $x^2_i - 2x_i + 1 + y^2_i + 2y_i + 1 - r^2$ | < | $x_i^2 + y^2_i + 2y_i + 1 - r^2$ |
- = | $x^2_i + y^2_i - r^2 + 2y_i + 1 + 1 - 2x_i$ | < | $x_i^2 + y^2_i - r^2 + 2y_i + 1$ |
- = | $(x^2_i + y^2_i - r^2 + 2y_i + 1) + (1 - 2x_i)$ | < | $(x_i^2 + y^2_i - r^2 + 2y_i + 1)$ |

Since absolute function | | requires additional library, we replace it by squaring both sides of the equation.

- = $[(x^2_i + y^2_i - r^2 + 2y_i + 1) + (1 - 2x_i)]^2$ < $[(x_i^2 + y^2_i - r^2 + 2y_i + 1)]^2$

Expanding the left hand side of the equation gives us:

- = $(x^2_i + y^2_i - r^2 + 2y_i + 1)^2 + 2(1 - 2x_i)(x^2_i + y^2_i - r^2 + 2y_i + 1) + (1 - 2x_i)^2$
- < $(x_i^2 + y^2_i - r^2 + 2y_i + 1)^2$

Canceling out $(x_i^2 + y^2_i - r^2 + 2y_i + 1)^2$

- = $2(1 - 2x_i)(x^2_i + y^2_i - r^2 + 2y_i + 1) + (1 - 2x_i)^2 < 0$

And since we know that x > 0 in the first octant, the term $(1 - 2x_i) < 0$, dividing both sides will cancel out the negative, giving us

- (a) = $2 [x^2_i + y^2_i - r^2 + 2y_i + 1] + (1 - 2x_i) > 0$
- (b) or simply = $2 [RE (x_i, y_i) + Y_{change}] + X_{change} > 0$

If the statement is true, then we decrement the value of x, otherwise, x stays at the same position.

Consider statement (a), to translate the decision statement to program, we write:

```
void drawCircle2(SDL_Surface* surface, Sint32 cx, Sint32 cy, Sint32 r, Uint32 rgba)
{
    Sint32 x, x2, xn, y, y2, r2, re, xsurface, ysurface;


    // Center position the pointer
    Uint32* pixels = (Uint32*)surface->pixels + cx + (cy * surface->w);
    r2 = r * r;
    x = r;
    x2 = x * x;
    y = 0;

    while (x > y)
    {
        xsurface = x * surface->w;
        ysurface = y * surface->w;

        y2 = y * y;
        re = x2 + y2 - r2;

        if ((2 * re) + (4 * y + 2) + (1 - (2 * x)) > 0)
        {
            --x;
            x2 = x * x;
        }

        // First Octant
        *(pixels + x - (ysurface)) = rgba;
```

```
        // The rest of the octants
        *(pixels + y - (xsurface)) = rgba;
        *(pixels - y - (xsurface)) = rgba;
        *(pixels - x - (ysurface)) = rgba;
        *(pixels + x + (ysurface)) = rgba;
        *(pixels + y + (xsurface)) = rgba;
        *(pixels - y + (xsurface)) = rgba;
        *(pixels - x + (ysurface)) = rgba;

        ++y;
    }
}
```

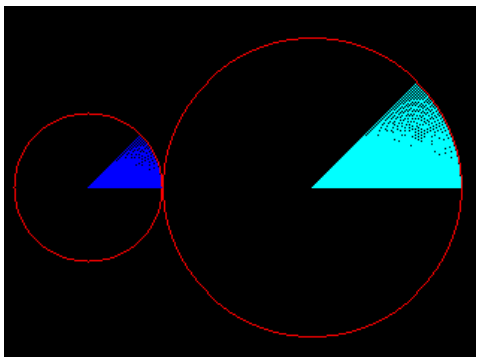
The above script can still be further improved by eliminating multiplication in the script by the following:

- bit shifting to the left one time multiplies the integer by 2, bit shifting to the left two times multiples the integer by 4 (refer to drawCircle3() function in attached .cpp file)
- Consider the 3 quantities in the equation at close inspection and determine their values at different values of x_i and y_i. Let's find a way to do it iteratively to reduce the computational complexity (removal of squares - multiplication).
    - RE $(x_i, y_i) = x^2_i + y^2_i - r^2$
    - $Y_{change} = 2y_i + 1$
    - $X_{change} = 1 - 2x_i$
- Since we're starting with x =r, y = 0, and RE $(x_0, y_0) = 0$, let's set RE initially to 0, $Y_{change} = 1$, and $X_{change} = 1 - (2 * r)$;
- We increment RE byY_{change} every iteration since, y is in the fast direction.
- We increment Y_{change} by 2 based on the formula.
- We compare (2 x RE + Xchange > 0), if true then that's the time we decrement x and increment RE by X_{change} and increment X_{change} y 2 based on the formula.

The complete program of plotting a circle using integer arithmetic is shown below:

```
void drawCircle4(SDL_Surface* surface, Sint32 cx, Sint32 cy, Sint32 r, Uint32 rgba)
{
    Sint32 x, y, xsurface, ysurface;
    Sint32 xchange, ychange, re;

    // Center position the pointer
    Uint32* pixels = (Uint32*)surface->pixels + cx + (cy * surface->w);

    x = r;
    y = 0;
    xsurface = x * surface->w;
    ysurface = y * surface->w;

    xchange = 1 - (r << 1);
    ychange = 1;
    re = 0;

    while (x > y)
    {
        // First Octant
        *(pixels + x - (ysurface)) = rgba;

        // The rest of the octants
        *(pixels + y - (xsurface)) = rgba;
        *(pixels - y - (xsurface)) = rgba;
```

```
        *(pixels - x - (ysurface)) = rgba;
        *(pixels + x + (ysurface)) = rgba;
        *(pixels + y + (xsurface)) = rgba;
        *(pixels - y + (xsurface)) = rgba;
        *(pixels - x + (ysurface)) = rgba;

        ++y;
        re += ychange;
        ychange += 2;
        if ((re << 1) + xchange > 0)
        {
            --x;
            xsurface -= surface->w;
            re += xchange;
            xchange += 2;
        }
        ysurface += surface->w;
    }
}
```



So there you go, an elegant solution (with an explanation of the derivation) as to how you can draw a circle efficiently. Feel free to play around with the codes and import it to your programming language of choice.

Attachments:

Circle.cpp

Tags:

GFX Algorithms

Bresenham's Algorithm C++ Circle SDL Midpoint Circle Algorithm

Everett Gaius's blog