BASIC DRAWING ALGORITHMS

Almost all systems that have graphics capabilities have built-in functions or libraries to perform primitive drawing operations, such as lines, rectangles, and circles. However, if the only drawing capability the system has is to set the value at an individual pixel, then it is necessary to implement the basic drawing algorithms in your program. There are also times when your application needs to perform an operation similar to one of the basic drawing algorithms, and it is useful to be familiar with how they operate. As an example, a communication application may be required to perform a Line of Sight (LOS) calculation from one point to determine if another point is visible. One approach is to step along the line using a standard line drawing algorithm to determine if the ground altitude at any point along the line is higher than the height of the line.

Four basic algorithms will be described: line drawing, circle drawing, curve drawing, and area filling.

Drawing Lines

Perhaps the most basic operation in computer graphics, other than setting the value of a single pixel, is to draw a line from point A to point B. The first approach one might take to accomplish this is to determine the equation of the line from A to B, and simply solve the equation for every point in-between. For example, if you use the slope intercept form of the line equation, \( y = m \times x + b \), you can calculate the values of \( m \) and \( b \) from the endpoints of the line:

\[
\begin{align*}
  m &= \frac{y_2 - y_1}{x_2 - x_1} \\
  b &= y_1 - m \times x_1
\end{align*}
\]

Once calculated, the value for \( Y \) can be computed for every point \( X \) along the line. Figure 1 shows a simple example. The filled squares in the grid represent pixels that would be colored ON to represent the line. Since pixel coordinates are integers, the calculated values for \( Y \) have to be rounded to determine which pixel coordinate to fill.

![Figure 1. Simple line drawing approach](image)

Note that this simple approach only works if the slope of the line is between -1 and 1. If the slope is greater than 1, then incrementally stepping \( X \) can result in skipped pixels. Figure 2 shows this result.
Figure 2. Incremental X approach with slope > 1.

The solution is to incrementally step the Y value and calculate X based on the line equation, as shown in Figure 3. In general, using this simple approach, select which coordinate to increment based on the slope of the line. Special checks should be made for horizontal and vertical lines to avoid division by zero.

Figure 3. Incremental Y approach with slope > 1.

A variation on this simple approach is to calculate each new value as an incremental change from the previous value. For example, in Figure 1, the slope of the line was 0.4. That means for every step of 1 in the X direction, the Y changes by +0.4, as can be seen in the table of values. Rather than calculating the value of Y from the line equation, simply adding 0.4 to the previous Y value within a loop quickly provides the next value. If the slope of the line is greater than 1, and the Y value is being incremented, than the incremental X change is 1 / slope. This approach to calculating line points is known as a digital differential analyzer or DDA algorithm.

A famous variation of the DDA algorithm was developed in 1965 by Jack Bresenham, known as Bresenham’s algorithm. What is significant about Bresenham’s algorithm is that it requires only integer addition, subtraction, and bit shifting. Thus, it is very fast and can easily be implemented in graphics displays and output devices, such as plotters. It uses an incremental approach similar to DDA, but rather than calculating the exact value at each point, it calculates an error value which is the difference
between the rounded and exact values. If the error value exceeds .5, the next value is incremented by 1. The details of the algorithm can be found online:

http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

As previously described, one issue that needs to be addressed, especially with lines, is the problem of aliasing. While a technique, such as supersampling, can be used with lines, a more common approach is to use an algorithm that specifically handles anti-aliasing, such as Wu’s algorithm which was presented in 1991. Wu’s algorithm is also an incremental algorithm, but rather than making a determination at each point as to which pixel to fill, it plots two points that straddle the line. Each point is filled with a gray scale that is based on how close the center of the pixel is to the line. This algorithm, while slower than Bresenham’s, is much faster than traditional anti-aliasing techniques. Figure 4 shows a sample set of pixels weighted by distance to the line using Wu’s approach and a zoomed in version of the resulting line. The reader is referred to the literature for details of the algorithm.

![Figure 4. Wu's algorithm for drawing anti-aliased lines.](image)

**Drawing Circles**

Similar to a line, one way of drawing circles is to start with a simple equation. The equation for a circle with a radius of \( r \) centered at \( x_0, y_0 \) is:

\[
(x - x_0)^2 + (y - y_0)^2 - r^2 = 0
\]

You can solve this equation for \( y \), and then for each value of \( x \) within the range of the circle, calculate the two \( y \) values (there are two because of the square root) and plot them. Figure 5 shows the resulting circle using this approach. Unfortunately, as a result of only using integer values of \( x \), there are places where there are “gaps” in the circle. A simple check can determine if there are any missing values between the current versus previous \( y \) value, and fill in any gaps, or lines could be drawn between successive points.
Figure 5. Simple circle drawing using equation.

An alternative formulation is to use a parametric form of the circle equation:

\[ x = r \cos t + x_0 \quad y = r \sin t + y_0 \quad 0 \leq t \leq 2\pi \]

The value of \( t \) can be incremented from 0 to \( 2\pi \), values of \( x \) and \( y \) can be calculated, and lines between successive points drawn to the screen. As seen in Figure 6, the size of the \( t \) increment determines the number of points calculated on the circle. Too few points can result in a “blocky” appearance. An advantage of using the parametric equation approach is that generated points are equidistant apart. Thus, the number of points required to create a “smooth” approximation of a circle can be determined at run-time based on the radius and minimum desired distance between points. Note that you typically have to deal with the last line segment as a special case due to round-off errors.

Figure 6. Drawing a circle with the parametric equation with 5, 10, and 25 points (\( t=1.5708, 0.7854, 0.3142 \)).

Both of these approaches to drawing circles require a more complex math operation, either a square root or a trigonometric function. With modern computing speeds, these do not exact the performance penalty of earlier processors. However, if speed is an issue, or if the algorithms are being put into simple hardware, there are alternative drawing schemes, similar to lines, which only require integer arithmetic. There is a variation on both Bresenham’s and Wu’s algorithms that can handle circles. Another incremental algorithm is the midpoint algorithm which is similar to Bresenham’s. This is illustrated in Figure 7. If the upper left pixel (column 0) is part of the circle, then when \( x \) is incremented, the next \( y \) value in column 1 is either the same, or \( y - 1 \). To determine which it is, the midpoint between each pixel, shown by the \( X \), is checked as to whether it is contained in the circle or not. If it is, as in columns 1 and 3, than \( y \) remains the same; if not, as in column 2, then \( y \) is decremented. The determination of containment in the circle can be achieved by substituting the \( x, y \) values for the midpoint into the initial equation for the circle shown above. If the results are less than zero, the point is contained; if equal to zero, the point is on the circle; if greater than zero, the point is outside the circle. This calculation requires no special math operations.
Methods for drawing curves depend on the types of curves and how they are defined. There are some constructive and incremental methods for certain types. A common general approach is to simply compute points along the curve and draw line segments to connect them. If the curve is parametrically defined, such as Bezier curves, then the parameter can be incremented from 0 to 1 to compute x and y coordinates similar to the circle above. A challenge is to draw line segments that are close enough together to provide a smooth appearance. In areas of high curvature, more points are needed, while relatively straight areas require fewer points. Unlike the parametric form of the circle, in a Bezier curve coordinates calculated from equal increments of t are not equidistant apart, as shown in Figure 8.

Filling an Area

Oftentimes, it is desirable to display shapes as solidly shaded areas versus just the outlines. For many systems, including Processing, if you want to fill a shape, you simply set the color using a fill() function before drawing the shape. In paint systems, flooding an area with a color is a common operation. It can be implemented as changing all connected pixels of a specified color, known as flood fill, or flooding up to a specified boundary color, known as boundary fill. Figure 9 shows how the two approaches differ. In the original image, the cursor is pointing at a yellow pixel. When flooding a color, as shown in the lower left, all connected yellow pixels turn into the flood color. In the lower right image, when filling to a boundary color (black in this example) everything up to that boundary is turned to the flood color.
Fill Algorithms

A simple approach to flood filling is to use a recursive routine that checks the current pixel, and if it is equal to the target color, change it and check its neighbors. Sample code for this in Processing, along with the output is shown in Figure 10.

```java
void Flood(int Px, int Py){
    int PixelLoc = Py * width + Px;
    if(pixels[PixelLoc] == targetColor) {
        pixels[PixelLoc] = finalColor;
        flood(Px-1,Py);
        flood(Px+1,Py);
        flood(Px,Py-1);
        flood(Px,Py+1);
    }
}
```

In this example, the target color was yellow, the final color was green, and the starting point was in the yellow area. The size of the face was 80 x 80 pixels, and it required 4,069 calls to the Flood routine to complete. In systems in which stack space is limited, this approach is not practical or efficient. For example, when the face size above was increased to 150 x 150, the program failed with a StackOverflowError stating it was attempting too much recursion. To change this from a flood fill to a boundary fill, each pixel would be checked if it was not equal to the boundary color before being placed on the stack.

An alternative approach uses a queue implementation and does not run into stack problems. Any of several queue implementation methods could be used to implement the algorithm, a version using the Java ArrayList structure and a simple Point class is shown below:
void flood(int Px, int Py) {
    // flood fill using queue
    Point pt = new Point(Px, Py); // uses a class Point, that contains x, y
    ArrayList queue = new ArrayList(); // use the Java ArrayList for the queue
    if(pixels[Py*width+Px] != targetColor) // if the starting point is not the
        return; // right color, quit
    queue.add(pt); // otherwise put the first pt on the queue
    while(!queue.isEmpty()) { // keep checking points until queue empty
        Pt = (Point)queue.remove(queue.size()-1); // pull a pt off the queue
        Px = pt.Px; // get the x, y coordinates of the point
        Py = pt.Py;
        int currPt = Py*width+Px; // compute indexes for surrounding points
        int eastPt = currPt+1;
        int westPt = currPt-1;
        int northPt = (Py-1)*width+Px;
        int southPt = (Py+1)*width+Px;
        if(pixels[currPt] == targetColor) // if this pt the right color, change
            pixels[currPt] = finalColor;
        if(pixels[westPt] == targetColor) { // check the point to the west
            pixels[westPt] = finalColor; // if it is, change and put on queue
            queue.add(new Point(Px-1, Py));
        }
        if(pixels[eastPt] == targetColor) { // check the point to the east
            pixels[eastPt] = finalColor;
            queue.add(new Point(Px+1, Py));
        }
        if(pixels[northPt] == targetColor) { // check the point to the north
            pixels[northPt] = finalColor;
            queue.add(new Point(Px, Py-1));
        }
        if(pixels[southPt] == targetColor) { // check the point to the south
            pixels[southPt] = finalColor;
            queue.add(new Point(Px, Py+1));
        }
    }
}

An optimization can be made to this approach by filling lines, rather than putting each pixel on the
queue. When checking east and west pixels, continue checking until a non-fill color is found, then fill in
all pixels in-between. North and south pixels for all points filled in still need to be checked for adding to
the queue.

**Scanline Algorithms**

In addition to drawing lines and curves, there are many occasions in which you want to draw filled
polygons to the screen. When rendering a polygon, one approach would be to draw the edges using
one of the line drawing algorithms, and then use an area fill algorithm to fill it in. Care would be
necessary if drawing over existing lines and filled areas, that the fill algorithm would work correctly. A
more straightforward approach is to combine these two steps into a single algorithm. A common way to
accomplish this is to order the edges of the polygon by vertical position, and for each horizontal row of
pixels, or scanline, calculate where each edge crosses the scanline, order the crossing points in pairs, and
fill in-between. This type of algorithm is known as a scanline algorithm. Figure 12 shows a conceptual
picture of how the algorithm would work. For the dotted scanline shown, the edges of the polygon
intersect it four times. If the x values of the crossings are ordered left to right, then pairwise values can
be used to fill in interior points. As long as the polygon is well-formed (e.g., no crossing lines), this
approach will work. Even if the polygon has holes, the pairwise filling will work correctly.
Oftentimes, to simplify the scanline approach, rather than working with arbitrary polygons, the polygons are first broken into triangles. There are well-defined algorithms for converting any arbitrary polygon into a set of triangles. This process, known as tiling (or tessellation), can use different criteria for determining the “best” set of triangles to fill an area. Figure 13 shows two possible tilings of the polygon in Figure 12. The advantages of working with triangles are that a) they are guaranteed to be planar in three dimensions, since three points define a plane, and b) there are only three edges which allow for fast scanline algorithms that can easily be translated into hardware.

A single triangle can be broken further into two parts based on where a scanline from the “middle vertex” intersects the opposite side, as shown in Figure 14. Now each “half triangle” consists of two edges that can be quickly filled by stepping from scanline to scanline. For each edge, calculate the amount the x changes from scanline to scanline, add that amount to the x crossing to get each x intersection, and fill in-between. The pseudocode for this, assuming the vertices are labeled as shown in Figure 14, would be:
DeltaX0 = (C.x – A.x) / (C.y – A.y)
DeltaX1 = (C.x – B.x) / (C.y – B.y)
x0 = A.x
x1 = B.x
for( y = A.y to C.y)
    for( x = x0 to x1)
        FillPixel(x,y)
    x0 += DeltaX0
    x1 += DeltaX1

Figure 14. Breaking a triangle into two halves and scan converting.

Other variations of this algorithm can be used to only use integer arithmetic, similar to line drawing algorithms. This same approach can be used to interpolate other values across a triangle. For example, in the example above, assume each of the three triangle points had a different color. In addition to calculating a delta X value for each edge, you can calculate a delta-Red, delta-Green, and delta-Blue value that change the color as you move from scanline to scanline and across a scanline. Figure 15 shows an example of such a color interpolation. In future readings, we will use this technique of interpolating values across a triangle to create smooth shading with realistic lighting.

Figure 15. Interpolating color across a triangle.

Lagniappe – Rotoscopy

An animation technique that is occasionally seen in movies and television commercials is known as rotoscoping, or taking live-action footage and tracing the outline of objects so that the result appears
hand-drawn. This technique, while still prevalent today, is not new. The original concept was invented and patented by Max Fleischer in 1915! A pre-recorded live-action film was projected, frame by frame, onto a frosted panel and redrawn by an artist. The results were an animated film that looked hand-drawn, but whose motion was very life-like. The device to perform this was called a *rotscope*. Max used his brother, dressed up as a clown, as the live-film reference for the character Koko the Clown who appeared in many early cartoons.

Several animation studios used rotoscoping for all or part of their animations. Disney used it in *Snow White*, *Cinderella*, and *One Hundred and One Dalmatians*. The rotoscope was used mainly for studying human and animal motion rather than the actual tracing of characters. In more recent times, rotoscoping has been used in several video games, television commercials, and for visual effects in live-action movies. A classic example of traditional rotoscoping was in the original three *Star Wars* films in which it was used to create the glowing lightsaber effect. Oftentimes today, the animator is replaced with computer software that can automatically generate the drawing from the film. One of the best known recent uses of rotoscoping was to create the 2006 film, *A Scanner Darkly*. 