Chapter 14

Rasterization

Go East or Northeast, young man.

Not Horace Greeley

Rasterization is the process of converting vector graphics into raster graphics. For example, vector graphics would describe a triangle as an outline (or a filled region) of three edges specified as a line loop of three vertices. A rasterized triangle outline would consists of the pixels indicating the three edges of the triangle, and a rasterized filled triangle would consist of the pixels representing the interior of the triangle. Since the edges of the triangle might not pass precisely through a pixel location, rasterization is a discretization that approximates the vector graphics representation.

14.1 Rasterizing a Line Segment

The first step for rasterization is to learn how to rasterize a line segment. The vector graphics description of a line segment is given by two endpoints \((x_0, y_0)\) and \((x_1, y_1)\) in viewport coordinates. Modern implementations rasterize lines between floating point endpoints that can fall between pixel centers, but for simplicity, we will assume the endpoints are integer coordinates that lie at pixel centers.

Recall the (explicit) equation for a (non-vertical) line

\[
y = mx + b, \tag{14.1}
\]

which yields a \(y\) coordinate corresponding to every \(x\) coordinate provided. For a line passing through the points \((x_0, y_0)\) and \((x_1, y_1)\) (so long as \(x_0 \neq x_1\)) the slope is

\[
m = \frac{y_1 - y_0}{x_1 - x_0}, \tag{14.2}
\]

(where \(m\) would be infinite for a vertical line). The \(y\)-intercept is found by evaluating (14.1) for \(x = 0\), which yields \(y = b\).
We can simplify the rasterization of a line segment if we only consider line segments that extend from the origin to a point in the first octant (points \((x, y)\) such that \(0 \leq y \leq x\)). Such first-octant lines have slope \(0 \leq m \leq 1\). When given an arbitrary line segment from any point \((x_0, y_0)\) to any other point \((x_1, y_1)\), we will (1) transform the line segment to extend from the origin to a point in the first octant, (2) rasterize this transformed segment into a sequence of first-octant pixels, and (3) apply the inverse transform to move each pixel back to the original line segment.

This transformation consists of a translation followed by a reflection. The translation \(T = T(-x_0, -y_0)\) moves the first endpoint \((x_0, y_0)\) to the origin, and the second endpoint to the position \((x_1 - x_0, y_1 - y_0)\). The reflection \(R\) will transform the position \((x_1 - x_0, y_1 - y_0)\) into the first octant if it isn’t there already, e.g. using one of the seven reflections shown in Figure 14.1. Hence the composite transformation \(RT\) maps the endpoints \((x_0, y_0), (x_1, y_1)\) to the points \((0, 0), (x', y')\) where \(0 \leq y' \leq x'\). This first octant line will be rasterized into a sequence of pixel positions \((x'_i, y'_i)\), and each of these pixel positions is mapped back to the pixels \((x_i, y_i)\) of the original line segment from \((x_0, y_0)\) to \((x_1, y_1)\) using the inverse composite transformation

\[
(x_i, y_i) = T^{-1}R^{-1}(x'_i, y'_i). \tag{14.3}
\]

For a line segment extending from the origin to a point \((x', y')\) in the first quadrant, the line equation (14.1) simplifies to

\[
y = \frac{y'}{x'} x. \tag{14.4}
\]

We can use this equation to find points \((x, y)\) on the segment by plugging in values of \(x\) between 0 and \(x'\), which yield corresponding values of \(y\) between
0 and \( y' \). Because the slope is less than one, we know that if we move along the line changing the \( x \) coordinate by one unit, the corresponding \( y \) coordinate changes by less than one unit. Hence we can find for each \( x \) coordinate from 0 to \( x' \) the integer \( y \) coordinate closest to the line to plot it as a contiguous sequence of pixels. (If we do this for slopes greater than one, we get gaps between pixels.) Hence we rasterize a line by finding for each integer \( x_{\text{pixel}} \) coordinate from 0 to \( x' \), the closest integer \( y_{\text{pixel}} \) coordinate
\[
y_{\text{pixel}} = \text{round}\left(\frac{y'}{x_{\text{pixel}}}x_{\text{pixel}}\right).
\]

(14.5)

Figure 14.3: Rasterization of a line from \((0, 0)\) to \((5, 4)\). The slope is \( m = \frac{4}{5} \), and each of the integer \( x \) coordinates from 0 to 5 yields the \( y \) coordinates 0, \( \frac{4}{5}, 1\frac{3}{5}, 2\frac{2}{5}, 3\frac{1}{5} \) and 4, which round to 0, 1, 2, 2, 3, 4, thus discretizing the line into a contiguous row of pixels.

The \text{round()} function rounds the real \( y \) value from (14.4) to the nearest integer \( y_{\text{pixel}} \) value. When the real \( y \) value is exactly between two integer values, the round function usually “rounds up” to the higher integer value, but this is an arbitrary choice. For example as shown in Figure 14.4, when \( x = 2 \) the line segment from \((0, 0)\) to \((4, 3)\) yields \( y = \frac{3}{4}(2) = 1\frac{1}{2} \). Should pixel \((2, 1)\) or \((2, 2)\) be illuminated to represent the line at this horizontal position? Either pixel can be used, so long as the choice is made consistently. If the choice was random, then as lines are redrawn in an interactive graphics display, the pixel can flicker, which distracts the user as our perceptual system is designed to detect such anomalies.

As stated earlier, our first-octant lines have a slope between zero and one. Even with rounding, when a pixel is plotted at integer coordinates \((x, y)\), the next pixel (if there is one) is either at \((x + 1, y)\) or \((x + 1, y + 1)\).
Hence rasterizing a line is a series of binary choices. If we just plotted \((x, y)\) then the next pixel (if there is one) is either the “East” pixel \(E = (x + 1, y)\) or the “Northeast” pixel \(NE = (x + 1, y + 1)\).

We can express this binary choice between \(E\) and \(NE\) pixels as a function. Recall the (explicit) line equation \(y = mx + b\) yields a \(y\) coordinate when given an \(x\) coordinate. If we subtract \(y\) from both sides, we get the (implicit) line equation

\[
0 = mx + b - y, \quad (14.6)
\]

which states that points \((x, y)\) on a line with slope \(m\) and \(y\)-intercept \(b\) satisfy (14.6), and points \((x, y)\) not on the line do not satisfy (14.6). In fact, we can define a function

\[
f(x, y) = mx + b - y, \quad (14.7)
\]

such that \(f(x, y) = 0\) for a point \((x, y)\) on the line, and \(f(x, y) \neq 0\) for points off the line. In fact \(f(x, y) > 0\) for a point \((x, y)\) below the line, because its \(y\) coordinate is too small and does not subtract enough when negated in (14.7) to overcome the positive \(x\) times the (positive) slope. Similarly, \(f(x, y) < 0\) for a point \(x, y\) above the line, because its \(y\) coordinate is too large and when negated subtracts more than is compensated by the product of the slope and the \(x\) coordinate. More concisely,

\[
\ell \; = \; \{(x, y) : y = mx + b\};
\]

\[
f(x, y) \; = \; mx + b - y, \quad f(x, y) = 0 \implies (x, y) \in \ell, \quad f(x, y) > 0 \implies (x, y) \text{ below } \ell, \quad f(x, y) < 0 \implies (x, y) \text{ above } \ell.
\]

If we just plotted pixel \((x, y)\), we can use a simplified \((m = \frac{y'}{x'}, b = 0)\) version of our line equation (14.7)

\[
f(x, y) = \frac{y'}{x'} x - y \quad (14.8)
\]
as a decision function to tell us whether our line is closer to the East pixel \((x + 1, y)\) or the Northeast pixel \((x + 1, y + 1)\). This decision is made by examining the midpoint \(M = (x + 1, y + \frac{1}{2})\) halfway between the East and Northeast pixel positions. If \(M\) is below the line, then the line is closer to the \(NE\) pixel, if \(M\) is above the line, then the line is closer to the \(E\) pixel, and if \(M\) is on the line, we make an arbitrary but consistent choice between the East and Northeast pixel. For our discussion, we can assume we “round up” and choose the Northeast pixel when the line falls equally between \(NE\) and \(E\). We can thus use the function \(f(x, y)\) from (14.7) on the midpoint \(M\) to decide whether to use the East or Northeast pixel

\[
\text{if } f(M) > 0 \text{ then } NE \text{ else } E. \quad (14.9)
\]
Applying this implicit line function to the next midpoint to decide between $E$ and $NE$ leads to a simple algorithm to draw a line segment from the origin to an endpoint in the first octant.

**Procedure** Line($int \ x', y'$):

```plaintext
float f(x, y) := (y'/x')x - y;
int x, y ← 0;
for (x = 0; x < round(x1); x++) do
    Plot (x, y);
    if f(x + 1, y + 0.5) > 0 then y++;
Plot (x, y);
```

**Algorithm 1:** Rasterize a first octant line from the origin to $x'$, $y'$ using a decision function.

This algorithm is not any faster than evaluating $y = \text{round}((y'/x')x)$ for each $x$ coordinate. We can make it faster by observing that we can use the value of $f$ at the previous midpoint to efficiently find the value at the next midpoint. While there is one midpoint location $M$ used to decide between $N$ and $NE$, there will be a different midpoint used next depending on which choice we made.

Assume our line algorithm has just used pixel $P = (x, y)$. As shown in Figure 14.7, if the line falls below the current midpoint $M = (x + 1, y + \frac{1}{2})$ then the $E = (x + 1, y)$ pixel is chosen. The next decision will take place at a new midpoint location which we denote $M_E = (x + 2, y + \frac{1}{2})$. Similarly, as shown in Figure 14.8, when the line falls above the midpoint, then the $NE = (x + 1, y + 1)$ pixel is chosen. In this case, the next decision will take place at a new midpoint location which we denote $M_{NE} = (x + 2, y + \frac{1}{2})$.

We evaluate $f(M)$ and use the result to choose pixel $E$ or $NE$. Depending on that choice, our next decision will use either $f(M_E)$ or $f(M_{NE})$. Let $M = (x, y)$, then

$$f(M) = f(x, y) = \frac{y'}{x'}x - y. \tag{14.10}$$

If we pick pixel $E$, then the next decision will be at midpoint $M_E = (x + 1, y)$, and

$$f(M_E) = f(x + 1, y) \tag{14.11}$$
$$= \frac{y'}{x'}(x + 1) - y \tag{14.12}$$
$$= \frac{y'}{x'}x - y + \frac{y'}{x'} \tag{14.13}$$
$$= f(M) + \frac{y'}{x'}. \tag{14.14}$$

In other words, since we have already evaluated $f(M)$ we can find $f(M_E)$ by adding the slope $\frac{y'}{x'}$ to $f(M)$. Similarly, if we pick pixel $NE$, then our
next decision will be at midpoint \( M_{NE} = (x + 1, y + 1) \), and

\[
f(M_{NE}) = f(x + 1, y + 1) \quad \text{(14.15)}
\]

\[
= \frac{y'}{x'}(x + 1) - (y + 1) \quad \text{(14.16)}
\]

\[
= \frac{y'}{x'}x - y + \frac{y'}{x'} - 1 \quad \text{(14.17)}
\]

\[
= f(M) + \frac{y'}{x'} - 1. \quad \text{(14.18)}
\]

Hence we can find \( f(M_{NE}) \) by adding the slope \( \frac{y'}{x'} \) minus one to \( f(M) \). We need to initialize this sequence of incremental decision values with an initial value \( f(1, \frac{1}{2}) = \frac{y'}{x'} - \frac{1}{2} \), at the first midpoint. This incremental sequence of decision values leads to an even more efficient algorithm.

**Procedure** Line\((\text{int } x', y')\):

float \( f = \frac{y'}{x'} - 0.5 \);

int \( x, y \leftarrow 0 \);

for \((x = 0; x < x'; x++)\) do

Plot \((x, y)\);

if \( f \leq 0 \) then

\( f+ = \frac{y'}{x'} \);

else

\( f+ = \frac{y'}{x'} - 1 \);

\( y++ \);

Plot \((x, y)\);

**Algorithm 2**: Rasterize a first octant line from the origin to \( x', y' \) with incremental midpoints decision functions.

We can make this algorithm run even faster if we can implement it using only integer operations, in this case the floating point decision variable \( f \). Since we only care whether \( f > 0 \) or \( f \leq 0 \), we could equivalently multiply the decision value by any positive constant and apply the same test. Let

\[
F = 2x'f \quad \text{(14.19)}
\]

define the value of a new decision variable. Because \( x' > 0 \), we have that

\[
f > 0 \Rightarrow F > 0, \quad \text{(14.20)}
\]

\[
f \leq 0 \Rightarrow F \leq 0. \quad \text{(14.21)}
\]

We multiply \( f \) by \( x' \) to remove the denominator from the slope \( \frac{y'}{x'} \), and we multiply by 2 to remove the \( \frac{1}{2} \) used to initialize \( f \) at the first midpoint. Using \( F \), we can implement our incrementalized decision value sequence using only integer values.
Procedure Line(*int* *x′, *y′*):
   *int* *F* = 2*y′ − *x′*;
   *int* *x, y* ← 0;
   for (*x* = 0; *x* < *x′*; *x*++) do
      Plot (*x, y*);
      if *F* ≤ 0 then
         *F* += 2*y′;
      else
         *F* += 2*y′ − 2*x′;
         *y*++;
      end
      Plot (*x, y*)
   end

Algorithm 3: Rasterize a first octant line from the origin to *x′, y′* with incremental midpoints decision functions.

### 14.2 Polygon Rasterization

Line rasterization appears connected because pixels are either horizontal or diagonal neighbors. Polygon rasterization is used to fill in regions of pixels bound by a polygon.

The first step in polygon rasterization is to establish rules of pixel ownership. Similarly, a polygon is rarely drawn by itself. It is usually part of a mesh of neighboring polygons. As meshes are animated by repeated clearing and redrawing, we need to make sure that a pixel belongs to the same polygon each time it is drawn.

Hence we first need to choose some arbitrary but consistent rules for pixel ownership. Recall when drawing a line that falls equally between two vertical pixels, it does not matter which pixel is chosen so long as that choice is consistent to avoid flicker. For polygon rasterization, the goal is to rasterize pixels inside the polygon. The ambiguous cases occur when a polygon edge passes directly through a pixel. For our discussion of polygon rasterization, we will establish the following arbitrary (but consistent) rules of polygon rasterization pixel ownership.

1. **Bottoms.** The polygon owns pixels pierced by *bottom* horizontal edges, whose interior side is above the edge.
2. **Not Tops.** The polygon does not own pixels pierced by *top* horizontal edges, whose interior side is below the edge.
3. **Lefts.** The polygon owns pixels pierced by *left* edges, whose interior side is directly right of the edge.
4. **Not Rights.** The polygon owns pixels pierced by *right* edges, whose interior side is directly left of the edge.
Figure 14.9: Is point A inside or outside the shape? What about points B and C? Use the Jordan curve theorem to find out. Draw a line from the point in any direction to the outside of the shape, making sure the line you draw is not tangent to the boundary of the shape. (If it is, just draw another line in a different direction.) Count the number of times the line crosses the boundary. If it crosses an odd number of times, the point is inside, otherwise it is outside the shape.

We can implement these rules with the following scanline polygon rasterization algorithm. A “scanline” is a horizontal row of display pixels, so this algorithm rasterizes a polygon by finding rows of pixels that represent its interior.

SIMPLY CONNECTED? CONCAVE. OPENGL DOES TRIANGLES. NON-INTERSECTING EDGES.

We first describe the polygon with an edge table. We first remove horizontal edges from the polygon. (We will see why later.) The remaining non-horizontal edges, based on the unequal y-coordinate of their endpoints, will have a lower endpoint and an upper endpoint. Hence each edge rises from its lower vertex \((x_{\text{lower}}, y_{\text{lower}})\) to its upper vertex \((x_{\text{upper}}, y_{\text{upper}})\). We sort the edge table by \(y_{\text{lower}}\), the y coordinate of the lower endpoint of each edge.

We then begin rasterizing the polygon scanline by scanline, working up for increasing values of \(y\). We will use a second datastructure called the active edge table that will keep track of the edges used to rasterize the current scanline’s pixels in the polygon. The active edge table keeps track of:

\[ x, \text{ the } x\text{-coordinate of each edge where it intersects scanline } y, \]
\(\frac{dx}{dy}\), the inverse slope \(1/m = \frac{x_{\text{upper}} - x_{\text{lower}}}{(y_{\text{upper}} - y_{\text{lower}})}\), and \\
y_{\text{upper}}, the y coordinate of the edge’s upper endpoint.

We keep the active edge table sorted in this order, first by \(x\) then by \(1/m\).

We then rasterize the polygon by applying the following algorithm to each scanline \(y\).

1. Delete edges from the active edge table whose \(y_{\text{upper}} = y\).
2. For each active edge table edge, set \(x = x + \frac{dx}{dy}\).
3. Add to the active edge table any edges in the edge table whose \(y_{\text{lower}} = y\).
4. For each pair of active edge table entries \((i, i+1)\), plot pixels from \([x_i]\) to \([x_{i+1}] - 1\), inclusive.

Note that the last step assumes there are an even number of entries in the active edge table. We assume the scanline begins and ends outside the polygon, and each pair of entries in the active edge table denote a row of pixels in the polygon at that scanline. We begin plot from the ceiling of \(x_i\) to one less than the ceiling of \(x_{i+1}\). When \(x_i\) is an integer value the left edge passes through a pixel center, and since \([x_i] = x_i\) the left pixel is plotted. When \(x_{i+1}\) is an integer value the right edge passes through a pixel center, and since \([x_i] = x_{i+1}\) the right pixel is not plotted.

**Attribute Interpolation**

The polygon rasterization process rasterizes polygon edges differently than the Bresenham line algorithm, though there are some similarities. For each edge, we maintain the \(x\) coordinate where the current scanline \(y\) intersects the edge, and the inverse slope \(dx/dy\), used to increment the \(x\) coordinate proportionally when the scanline increments. The derivative \(dx/dy\) indicates the amount of change in \(x\) given a unit change in \(y\), and since this derivative is constant for a straight line, it is simply computed as

\[
\frac{dx}{dy} = \frac{x_{\text{upper}} - x_{\text{lower}}}{y_{\text{upper}} - y_{\text{lower}}}. \tag{14.22}
\]

You may recall that you can set a different color for each vertex of an OpenGL triangle, and the triangle is filled with a gradient that blends each of these vertex colors into each other. This is called Gouraud shading interpolation, and is used to smoothly blend the colors produced by the shading computed at each vertex.

For each polygon edge, we can interpolate the color assigned to the vertices at its two endpoints similar to the interpolation of \(x\) from \(x_{\text{lower}}\) to \(x_{\text{upper}}\). We can represent the RGB color channel values at each endpoint as
\( R_{\text{lower}}, G_{\text{lower}} \) and \( B_{\text{lower}} \) for the lower endpoint, and similarly for the upper endpoint. To interpolate e.g. the red channel from \( R_{\text{lower}} \) to \( R_{\text{upper}} \), we first compute the derivative

\[
\frac{dR}{dy} = \frac{R_{\text{upper}} - R_{\text{lower}}}{y_{\text{upper}} - y_{\text{lower}}}. \tag{14.23}
\]

We then initialize \( R = R_{\text{lower}} \) and increment \( R = R + dR/dy \) for each new scanline. We do this likewise for \( G \) and \( B \), and the result is a color interpolated along each edge of the polygon.

Now we need to interpolate the color through the interior pixels of the polygon. For a given scanline \( y \), we fill in pairs of edge \( x \)-coordinates. Let \((x_{\text{left}}, x_{\text{right}})\) be one of these pairs, and let \( R_{\text{left}} \) be the current red channel value \( R \) for the left edge and \( R_{\text{right}} \) be that for the right edge. For this pair of \( x \)-coordinates on scanline \( y \), we compute

\[
\frac{dR}{dx} = \frac{R_{\text{right}} - R_{\text{left}}}{x_{\text{right}} - x_{\text{left}}}, \tag{14.24}
\]

and initialize\(^1\) \( R = R_{\text{left}} \) and increment \( R = R + dR/dx \) as we set each pixel from \( \lceil x_{\text{left}} \rceil \) to \( \lceil x_{\text{right}} \rceil - 1 \), and similarly for \( G \) and \( B \).

---

\(^1\)To be precise, we should initialize \( R = R_{\text{left}} + (\lceil x \rceil - x)dR/dx \) to account for the difference between the left (integer) pixel and the floating point \( x \)-coordinate of the left edge.