Triangle Scan-Conversion

Pixel Coordinate System

Viewport on screen is specified in pixel coordinates

bounded in x by V_{x0}, V_{x1}

bounded in y by V_{y0}, V_{y1}

To map (x_p, y_p, z_p) form $[-1,1]\times[-1,1]\times[0,1]$ into viewport (x_v, y_v, z_v):

$$x_v = x_p \frac{V_{x1}-V_{x0}}{2} + \frac{V_{x1}+V_{x0}}{2}$$

$$y_v = y_p \frac{V_{y1}-V_{y0}}{2} + \frac{V_{y1}+V_{y0}}{2}$$

$$z_v = z_p$$

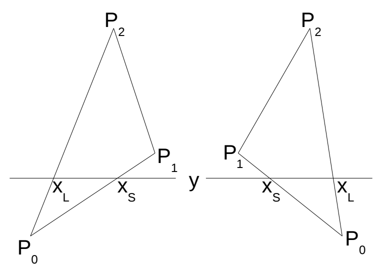
Triangle

Determined by 3 points in floating point pixel coordinates.

$P_0=(x_0, y_0, z_0), P_1=(x_1, y_1, z_1), P_2=(x_2, y_2, z_2)$

Re-ordered so: $y_0 \le y_1 \le y_2$

Edge equations scanline-to-scanline



Given a y scanline, calculate x and z

For edge from (x_i, y_i) to (x_j, y_j):

$$x = m_{xij} y + b_{xij}$$

$$z = m_{zij} y + b_{zij}$$

where

$$m_{xij} = \frac{x_j - x_i}{y_j - y_i} \qquad b_{xij} = \frac{x_i y_j - x_j y_i}{y_j - y_i}$$

$$m_{zij} = \frac{z_j - z_i}{y_j - y_i} \qquad b_{zij} = \frac{z_i y_j - z_j y_i}{y_j - y_i}.$$

Edge equations pixel-to-pixel

On a scanline y with endpoint (x_0, y, z_0) and (x_1, y, z_1):

$$z = m_p x + b_p$$

where

$$m_p = \frac{z_1 - z_0}{x_1 - x_0} \qquad b_p = \frac{z_0 x_1 - z_1 x_0}{x_1 - x_0}.$$

Outline

Split into two regions:

Region1: below the midpoint: $y_0 \le y \le y_1$

Region2: above the midpoint: $y_1 \le y \le y_2$

Rough algorithm:

For each scanline y **in Region1:**

$x_S = \ldots \quad z_S = \ldots \quad$ using points $P_{0,} P_1$

$x_L = \ldots \quad z_L = \ldots \quad$ using points $P_{0,} P_2$

For each pixel x **in span** $x_S \ldots x_L$ **:**

Color pixel (x, y)

For each scanline y **in Region2:**

same as Region1 except use $x_S = m_{12} y + b_{12}$

Many considerations:

Exactly which scanlines and which pixels?

Round or truncate?

Efficient calculations?

How to void double drawing of pixels?

Special cases? Like

$y_0 = y_1$ or $y_1 = y_2$, or

$x_S = x_L$, or

y region has no scanlines, or

x span has no pixels.

Exactly which pixels?

In order to avoid double drawing

(a) a triangle will fill in **only pixels whose centers it contains**

(b) but which triangle contains a pixel on the boundary?

Solution for (a):

careful use of floor and ceil

Solution for (b):

left/bottom filling rule,

and **very** careful use of floor and ceil

Identifying pixels with centers contained

For a region $y_0 \le y \le y_1$

use scanline $\lceil y_0 \rceil$ to $\lfloor y_1 \rfloor$, that is $ceil(y_0)$ to $floor(y_1)$.

For a span of pixels on a scanline $x_S \ldots x_L$:

swap $x_S \ldots x_L$ to ensure $x_S \le x_L$

use pixels $\lceil x_S \rceil$ to $\lfloor x_L \rfloor$, that is $ceil(x_S)$ to $floor(x_L)$.

This is **almost** what we want, except:

Problem: This can draw boundary pixels twice

if they fall on exact integer coordinates.

Next section solves that.

The left/bottom fill rule

If two regions/spans meet **exactly** at an integer:

Let the upper one claim it.

Or equivalently

A triangle draws exact integers on left/bottom edges, but

does not draw exact integers on the other two directions.

Note behavior of floor and ceil

for exact integer y: $floor(y) \neq ceil(y-1)$

for all other y: $floor(y) = ceil(y-1)$

which is exactly the behavior we want

So:

use pixels $ceil(y_0)$ to $ceil(y_1-1)$

and similarly for pixels across a scanline

Full Algorithm

ScanConvert($(x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y_2, z_2)$)

Reorder points so $y_0 \le y_1 \le y_2$

if $y_0 \ne y_1$ **// Region 1**

calculate slopes and intercepts m_{x01}, b_{x01} m_{z01}, b_{z01} m_{x02}, b_{x02} m_{z02}, b_{z02}

for y **from** $ceil(y_0)$ **to** $ceil(y_1 - 1)$

$x_S = m_{x01} y + b_{x01}$

$z_S = m_{z01} y + b_{z01}$

$x_L = m_{x02} y + b_{x02}$

$z_L = m_{z02} y + b_{z02}$

if $x_S < x_L$

$x_0, z_0, \quad x_1, z_1 = x_S, z_S, \quad x_L, z_L$

else

$x_0, z_0, \quad x_1, z_1 = x_L, z_L, \quad x_S, z_S$

calculate slope and intercept m_p, b_p **from** $(x_0, z_0), \quad (x_1, z_1)$

for x **from** $ceil(x_0)$ **to** $ceil(x_1 - 1)$

$z = m_p x + b_p$

SetPixel (x, y, z)

if $y_1 \ne y_2$ **// Region 2**

// Same except change 0 subscripts to 1 and 1 subscripts to 2.

Notes:

Zbuffer is initialized:

to same size as cbuffer

all pixels get depth of 1.0 (since that is max possible depth)

Cbuffer is initialized

all pixels get background color

```
SetPixel(x,y,z):
  if zbuffer[x,y] > z:
    cbuffer[x,y] = Color at pixel (x,y,z) of object being rendered
    zbuffer[x,y] = z
```

Efficiency

Loops in the algorithm calculate linear equations. For instance:

for y **from** y_{first} **to** y_{last}

$x_S = m_{x01} y + b_{x01}$

A more efficient calculation can use an increment

$x_S = m_{x01} y_{first} + b_{x01}$

for y **from** y_{first} **to** y_{last}

$x_S += m_{x01}$