Announcements

- HW 2 extension till next Monday

Pipeline overview

- Application
- Command stream
- Vertex processing
- Transformed geometry
- Rasterization
- Fragments
- Fragment processing
- Framebuffer image
- Display

you are here

Primitives

- Points
- Line segments
- Triangles
- And that’s all!
  - Curves? Approximate them with chains of connected line segments
  - Polygons? Break them up into triangles
  - Curved regions? Approximate them with triangles
- Trend has been toward minimal primitives
  - Simple, uniform, repetitive: good for parallelism
  - And of course, cyclical; now you can send curves, and the vertex shader will convert to primitives

Rasterization

- First job: enumerate the pixels covered by a primitive
  - Simple, aliased definition: pixels whose centers fall inside
- Second job: interpolate values across the primitive
  - E.g. colors computed at vertices
  - E.g. normals at vertices
  - Will see applications later on

Rasterizing lines

- Define line as a rectangle
- Specify by two endpoints
- Ideal image: black inside, white outside
**Point sampling**

- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: sometimes turns on adjacent pixels

---

**Bresenham lines (midpoint alg.)**

- Point sampling unit width rectangle leads to uneven line width
- Define line width parallel to pixel grid
- That is, turn on the single nearest pixel in each column
- Note that 45° lines are now thinner

---

**Algorithms for drawing lines**

- line equation: 
  \[ y = b + mx \]
  \[ d = mx + b - y \]
- Simple algorithm: evaluate line equation per column
- W.l.o.g., \( x_0 < x_1 \)
  \[ 0 \leq m \leq 1 \]
  for \( x = \text{ceil}(x_0) \) to floor(\( x_1 \))
  \[ y = b + m \times x \]
  output(\( x, \text{round}(y) \))

---

**Bresenham lines (midpoint alg.)**

- round(\( y \))?
  - cutoff at midpt
  \[ y = mx + b \]
  \[ d = mx + b - y \]
- \( d(x+1,y+0.5) \)
  \[ = m(x+1) + b - (y+0.5) \]
- \( d > 0 ? E : NE \)
  - what does \( d > 0 \) mean?
Optimizing line drawing

- Multiplying and rounding: slow
- At each pixel
  - only options are E and NE

Midpoint line algorithm

\[
\begin{align*}
x &= \text{ceil}(x_0) \\
y &= y_0 = \text{round}(m \cdot x + b) \\
d &= m \cdot (x + 1) + b - y \\
\text{while } x < \text{floor}(x_1) \\
& \quad \text{if } d > 0.5 \\
& \quad \quad y += 1 \\
& \quad \quad d -= 1 \\
& \quad \quad x += 1 \\
& \quad \quad d += m \\
& \quad \text{output}(x, y)
\end{align*}
\]

Linear interpolation

- We often attach attributes to vertices
  - e.g., computed diffuse color of a hair being drawn using lines
- Want color to vary smoothly along a chain of line segments
- Recall basic definition
  - 1D: \( f(x) = (1 - \alpha) y_0 + \alpha y_1 \)
  - where \( \alpha = \frac{x - x_0}{x_1 - x_0} \)
- In the 2D case of a line segment, alpha is just the fraction of the distance from \( (x_0, y_0) \) to \( (x_1, y_1) \)

Alternate interpretation

- We are updating \( d \) and \( \alpha \) as we step from pixel to pixel
  - \( d \) tells us how far from the line we are
  - \( \alpha \) tells us how far along the line we are
- So \( d \) and \( \alpha \) are coordinates in a coordinate system oriented to the line
Alternate interpretation

- View loop as visiting all pixels the line passes through
  - Interpolate \( d \) and \( \alpha \) for each pixel
  - Only output frag. if pixel is in band
- This makes linear interpolation the primary operation

Pixel-walk line rasterization

\[
x = \text{ceil}(x) \\
y = \text{round}(m \times x + b) \\
d = m \times x + b - y
\]
while \( x < \text{floor}(x) \)
- if \( d > 0.5 \), \( y += 1; d -= 1; \)
- else \( x += 1; d += m; \)
  - if \(-0.5 < d \leq 0.5\) output \((x, y)\)

Rasterizing triangles

- The most common case in most applications
  - with good antialiasing can be the only case
  - some systems render a line as two skinny triangles
- Triangle represented by three vertices
- Simple way to think of algorithm follows the pixel-walk interpretation of line rasterization
  - walk from pixel to pixel over (at least) the polygon’s area
  - evaluate linear functions as you go
  - use those functions to decide which pixels are inside

Incremental linear evaluation

- A linear (affine, really) function on the plane is:
  \[
  q(x, y) = c_x x + c_y y + c_k
  \]
- Linear functions are efficient to evaluate on a grid:
  \[
  q(x + 1, y) = c_x (x + 1) + c_y y + c_k = q(x, y) + c_x \\
  q(x, y + 1) = c_x x + c_y (y + 1) + c_k = q(x, y) + c_y
  \]

Incremental linear evaluation

\[
\text{linEval}(x_l, x_h, y_l, y_h, c_x, c_y, c_k) \{
  // setup 
  qRow = c_x * x_l + c_y * y_l + c_k; \\
  // traversal 
  for y = y_l to y_h { 
    qPix = qRow; 
    for x = x_l to x_h { 
      output(x, y, qPix); 
      qPix += c_x; 
    } 
    qRow -= c_y; 
  }
\}
\]

\( c_x = .005; c_y = .005; c_k = 0 \)
(image size 100x100)