CSE 167: Computer Graphics

• Rasterization
The graphics pipeline

• The standard approach to object-order graphics
• Many versions exist
  – Software (e.g., Pixar’s REYES architecture)
    • Many options for quality and flexibility
  – Hardware (e.g., graphics cards in PCs)
    • Amazing performance: millions of triangles per frame
• We will focus on an abstract version of hardware pipeline
• Called a “pipeline” because of the many stages
  – Very parallelizable
  – Leads to remarkable performance of graphics cards (many times the flops of the CPU at ~1/5 the clock speed)
Pipeline overview

You are here → APPLICATION

APPLICATION → COMMAND STREAM

COMMAND STREAM → VERTEX PROCESSING

VERTEX PROCESSING → TRANSFORMED GEOMETRY

TRANSFORMED GEOMETRY → RASTERIZATION

RASTERIZATION → FRAGMENTS

FRAGMENTS → FRAGMENT PROCESSING

FRAGMENT PROCESSING → FRAMEBUFFER IMAGE

FRAMEBUFFER IMAGE → DISPLAY

CSE 167, Winter 2020
Primitives

- Points
- Line segments (and chains of connected line segments)
- Triangles
- And that’s all!
  - Curves
    - Approximate them with chains of line segments
  - Polygons
    - Break them up into triangles
  - Curved surfaces
    - Approximate them with triangles
- Trend over the decades towards minimal primitives
  - Simple, uniform, repetitive = good for parallelism
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
  – e.g., texture coordinates
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
Point sampling in action
Bresenham lines (midpoint algorithm)

• 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
Midpoint algorithm in action
Algorithms for drawing lines

- Line equation: \( y = b + mx \)
- Simple algorithm: evaluate line equation per column
- Without loss of generality, \( x_0 < x_1 \) and \( 0 \leq m \leq 1 \)

```
for x = ceil(x0) to floor(x1)
    y = b + m*x
    output(x, round(y))
```
Optimizing line drawing

- Multiplying and rounding is slow
- At each pixel the only options are E and NE
- \( d = m(x + 1) + b - y \)
- \( d > 0.5 \) decides between E and NE
Optimizing line drawing

- \( d = m(x + 1) + b - y \)
- Only need to update \( d \) for integer steps in \( x \) and \( y \)
- Do that with addition
- Known as digital differential analyzer (DDA)
Midpoint line algorithm

\[ x = \text{ceil}(x_0) \]
\[ y = \text{round}(m \cdot x + b) \]
\[ d = m \cdot (x + 1) + b - y \]

while \( x < \text{floor}(x_1) \)
  
  if \( d > 0.5 \)
    
    \[ y += 1 \]
    
    \[ d -= 1 \]
  
  \[ x += 1 \]
  
  \[ d += m \]

output\((x, y)\)
Linear interpolation

• We often attach attributes to vertices
  – For example, computed diffuse color of a hair being drawn using lines
  – Desire for color to vary smoothly along a chain of line segments

• Linear interpolation in 1D
  \[ f(x) = (1 - \alpha) y_0 + \alpha y_1, \text{ where } \alpha = \frac{x - x_0}{x_1 - x_0} \]

• In the 2D case of a line segment, \(\alpha\) is just the fraction of the distance from \((x_0, y_0)\) to \((x_1, y_1)\)
Linear interpolation

- Pixels are not exactly on the line
- Define 2D function by projection on line
  - This is linear in 2D, so DDA can be used to interpolate
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 fragment if pixel is in band

• This makes linear interpolation the primary operation
Pixel-walk line rasterization

\[ x = \text{ceil}(x_0) \]
\[ y = \text{round}(m \times x + b) \]
\[ d = m \times x + b - y \]

while \( x < \text{floor}(x_1) \)

\[ \text{if } d > 0.5 \]
\[ \quad y += 1; \ d -= 1; \]
\[ \text{else} \]
\[ \quad x += 1; \ d += m; \]

\[ \text{if } -0.5 < d \leq 0.5 \]
\[ \quad \text{output}(x, y) \]
Rasterizing triangles

• The most common case in most applications
  – With good antialiasing can be the only case, as 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
Rasterizing triangles

• Input:
  – Three 2D points (the triangle’s vertices in pixel space)
    • \((x_0, y_0); (x_1, y_1); (x_2, y_2)\)
  – Parameter values at each vertex
    • \(q_{00},..., q_{0n}; q_{10},..., q_{1n}; q_{20},..., q_{2n}\)

• Output:
  – A list of fragments, each with
    • The integer pixel coordinates \((x, y)\)
    • Interpolated parameter values \(q_0, ..., q_n\)
Rasterizing triangles

• Summary
  1. Evaluation of linear functions on pixel grid
  2. Functions defined by parameter values at vertices
  3. Use of extra parameters to determine fragment set
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

linEval(xm, xM, ym, yM, cx, cy, ck) {

    // setup
    qRow = cx*xm + cy*ym + ck;

    // traversal
    for y = ym to yM {
        qPix = qRow;
        for x = xm to xM {
            output(x, y, qPix);
            qPix += cx;
        }
        qRow += cy;
    }
}

c_x = .005; c_y = .005; c_k = 0
(image size 100x100)
• **Summary**
  1. Evaluation of linear functions on pixel grid
  2. Functions defined by parameter values at vertices
  3. Use of extra parameters to determine fragment set
Defining parameter functions

- To interpolate parameters across a triangle we need to find the $c_x$, $c_y$, and $c_k$ that define the (unique) linear function that matches the given values at all 3 vertices
  - 3 constraints on 3 unknown coefficients
    
    \[
    \begin{align*}
    c_x x_0 + c_y y_0 + c_k &= q_0 \\
    c_x x_1 + c_y y_1 + c_k &= q_1 \\
    c_x x_2 + c_y y_2 + c_k &= q_2
    \end{align*}
    \]
  
    - In matrix form
      \[
      \begin{bmatrix}
      x_0 & y_0 & 1 \\
      x_1 & y_1 & 1 \\
      x_2 & y_2 & 1
      \end{bmatrix}
      \begin{bmatrix}
      c_x \\
      c_y \\
      c_k
      \end{bmatrix}
      =
      \begin{bmatrix}
      q_0 \\
      q_1 \\
      q_2
      \end{bmatrix}
      \]
Defining parameter functions

• More efficient version: shift origin to \((x_0, y_0)\)

\[
q(x, y) = c_x (x - x_0) + c_y (y - y_0) + q_0
\]
\[
q(x_1, y_1) = c_x (x_1 - x_0) + c_y (y_1 - y_0) + q_0 = q_1
\]
\[
q(x_2, y_2) = c_x (x_2 - x_0) + c_y (y_2 - y_0) + q_0 = q_2
\]

– Which is a 2x2 linear system (since \(q_0\) falls out):

\[
\begin{bmatrix}
(x_1 - x_0) & (y_1 - y_0) \\
(x_2 - x_0) & (y_2 - y_0)
\end{bmatrix}
\begin{bmatrix}
c_x \\
c_y
\end{bmatrix} =
\begin{bmatrix}
q_1 - q_0 \\
q_2 - q_0
\end{bmatrix}
\]

– Solve using Cramer’s rule

\[
c_x = (\Delta q_1 \Delta y_2 - \Delta q_2 \Delta y_1) / (\Delta x_1 \Delta y_2 - \Delta x_2 \Delta y_1)
\]
\[
c_y = (\Delta q_2 \Delta x_1 - \Delta q_1 \Delta x_2) / (\Delta x_1 \Delta y_2 - \Delta x_2 \Delta y_1)
\]
Defining parameter functions

\[
\text{linInterp}(x_m, x_M, y_m, y_M, x_0, y_0, q_0, \\
x_1, y_1, q_1, x_2, y_2, q_2) \{ \\

\text{// setup} \\
\text{det} = (x_1-x_0)*(y_2-y_0) - (x_2-x_0)*(y_1-y_0); \\
cx = ((q_1-q_0)*(y_2-y_0) - (q_2-q_0)*(y_1-y_0)) / det; \\
cy = ((q_2-q_0)*(x_1-x_0) - (q_1-q_0)*(x_2-x_0)) / det; \\
qRow = cx*(x_m-x_0) + cy*(y_m-y_0) + q_0; \\

\text{// traversal (same as before)} \\
\text{for } y = y_m \text{ to } y_M \{ \\
\text{qPix} = qRow; \\
\text{for } x = x_m \text{ to } x_M \{ \\
\text{output}(x, y, qPix); \\
\text{qPix} += cx; \\
\} \\
qRow += cy; \\
\} \\
\}
Interpolating several parameters

linInterp(xm, xM, ym, yM, n, x0, y0, q0[],
         x1, y1, q1[], x2, y2, q2[]) {

  // setup
  for k = 0 to n-1
    // compute cx[k], cy[k], qRow[k]
    // from q0[k], q1[k], q2[k]

  // traversal
  for y = ym to yM {
    for k = 1 to n, qPix[k] = qRow[k];
    for x = xm to xM {
      output(x, y, qPix);
      for k = 1 to n, qPix[k] += cx[k];
    }
    for k = 1 to n, qRow[k] += cy[k];
  }
}
Rasterizing triangles

• Summary
  1. Evaluation of linear functions on pixel grid
  2. Functions defined by parameter values at vertices
  3. Use of extra parameters to determine fragment set
Barycentric coordinates

• A coordinate system for triangles
  – Algebraic viewpoint
    \[ \mathbf{p} = \alpha \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c} \]
    \[ \alpha + \beta + \gamma = 1 \]
  – Geometric viewpoint
    • Areas of subtriangles
• Triangle interior test
  \[ \alpha > 0; \quad \beta > 0; \quad \gamma > 0 \]
Barycentric coordinates

• A coordinate system for triangles
  – Geometric viewpoint
    • Distances
  – Linear viewpoint
    • Basis of edges
      \[ \alpha = 1 - \beta - \gamma \]
      \[ \mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \]
Barycentric coordinates

- Linear viewpoint
  - Basis for the plane

- Triangle interior test
  \[ \beta > 0; \quad \gamma > 0; \quad \beta + \gamma < 1 \]
Clipping to the triangle

- Interpolate three barycentric coordinates across the plane
  - Each barycentric coordinate is 1 at one vertex and 0 at the other two
- Output fragments only when all three are > 0
Pixel-walk (Pineda) rasterization

- Conservatively visit a superset of the pixels you want
- Interpolate linear functions
- Use those functions to determine when to emit a fragment
Rasterizing triangles with shared edge

- Exercise caution with rounding and arbitrary decisions
  - Need to visit these pixels once
  - But it is important not to visit them twice!
Clipping

• Rasterizer tends to assume entire triangles are on screen
• After projection transformation (in normalized device coordinate frame)
  – Clip against the planes $X, Y, Z = 1, -1$
    • 6 planes defining canonical view volume
  – Primitive operation
    • Clip triangle against axis-aligned plane
Clipping a triangle against a plane

- 4 cases, based on sidedness of vertices
  - All inside (keep)
  - All outside (discard)
  - One inside, two outside (one clipped triangle)
  - Two inside, one outside (two clipped triangles)