From Vertices to Fragments: Rasterization

Reading Assignment: Chapter 7

Frame Buffer

- Special memory where pixel colors are stored.

System Bus

CPU Main Memory Graphics Card

-- Graphics Processing Unit (GPU)
-- Frame Buffer
The Graphics Pipeline

- **Modeling Transformations**
- **Illumination** (Shading)
- **Viewing Transformation** (Perspective / Orthographic)
- **Clipping**
- **Projection** (to Screen Space)
- **Scan Conversion** (Rasterization)
- **Visibility / Display**

**Input:**

- **Geometric model:**
  - Description of all object, surface, and light source geometry and transformations
- **Lighting model:**
  - Computational description of object and light properties, interaction (reflection)
- **Synthetic Viewpoint (or Camera):**
  - Eye position and viewing frustum
- **Raster Viewport:**
  - Pixel grid onto which image plane is mapped

**Output:**

- **Colors/Intensities** suitable for framebuffer display
  (For example, 24-bit RGB value at each pixel)

---

Modeling Transformations

- 3D models defined in their own coordinate system (object space)
- Modeling transforms orient the models within a common coordinate frame (world space)

![Object space and World space](image)
### Illumination (Shading) (Lighting)

- Vertices lit (shaded) according to material properties, surface properties (normal) and light sources
- Local lighting model (Diffuse, Ambient, Phong, etc.)

#### Modeling Transformations
#### Illumination (Shading)
#### Viewing Transformation (Perspective / Orthographic)
#### Clipping
#### Projection (to Screen Space)
#### Scan Conversion (Rasterization)
#### Visibility / Display

#### Viewing Transformation

- Maps world space to eye space
- Viewing position is transformed to origin & direction is oriented along some axis (usually z)
**Clipping**

- Transform to Normalized Device Coordinates (NDC)
- Portions of the object outside the view volume (view frustum) are removed

**Projection**

- The objects are projected to the 2D image place (screen space)
Scan Conversion (Rasterization)

- Rasterizes objects into pixels
- Interpolate values as we go (color, depth, etc.)

Visibility / Display

- Each pixel remembers the closest object (depth buffer)
- Almost every step in the graphics pipeline involves a change of coordinate system. Transformations are central to understanding 3D computer graphics.
From Geometry to Display

Rasterization

- How to draw primitives?
  - Convert from geometric definition to pixels
    - *Rasterization* = selecting the pixels
- Will be done frequently
- Must be fast:
  - use integer arithmetic
  - use addition instead of multiplication
Next

• Line-drawing algorithm
  – naïve algorithm
  – Bresenham algorithm
• Circle-drawing algorithm
  – naïve algorithm
  – Bresenham algorithm

Scan Converting 2D Line Segments

• Given:
  – Segment endpoints (integers $x_1, y_1; x_2, y_2$)
• Identify:
  – Set of pixels $(x, y)$ to display for segment
Line Rasterization Requirements

- Transform **continuous** primitive into **discrete** samples
- Uniform thickness & brightness
- Continuous appearance
- No gaps
- Accuracy
- Speed

Simple Line

Based on *slope-intercept algorithm* from algebra:

\[ y = mx + h \]

Simple approach:
- increment x, solve for y

Floating point arithmetic required
Naive Line Rasterization Algorithm

- Simply compute $y$ as a function of $x$
  - Conceptually: move vertical scan line from $x_1$ to $x_2$
  - What is the expression of $y$ as function of $x$?
  - Set pixel $(x, \text{round}(y))$

$$y = y_1 + \frac{x - x_1}{x_2 - x_1} (y_2 - y_1) = y_1 + m(x - x_1)$$

$$m = \frac{dy}{dx}$$

Efficiency

- Computing $y$ value is expensive
  $$y = y_1 + m(x - x_1)$$
- Observe: $y \ += m$ at each $x$ step ($m = dy/dx$)
Does it Work?

- It seems to work okay for lines with a slope of 1 or less.
- Doesn’t work well for lines with slope greater than 1 – lines become more discontinuous in appearance and we must add more than 1 pixel per column to make it work.
- Solution? - use symmetry.

Modify Algorithm per Octant

OR, increment along x-axis if dy<dx else increment along y-axis.
Bresenham's Algorithm

• Select pixel vertically closest to line segment
  – intuitive, efficient,
    pixel center always within 0.5 vertically
• Same answer as naive approach

Bresenham's Algorithm

• Observation:
  – If we're at pixel P \((x_p, y_p)\), the next pixel must be
    either E \((x_p+1, y_p)\) or NE \((x_p+1, y_p+1)\)
  – Why?
Bresenham Step

• Which pixel to choose: E or NE?
• Error associated with a pixel:

![Diagram of Bresenham algorithm]

- Pick the pixel with error < \( \frac{1}{2} \)
- The sum of the 2 errors is _____

Bresenham Step

• How to compute the error?
• Line defined as \( y = mx + h \)
• Vertical distance from line to pixel \((x, y)\):

\[
e(x, y) = mx + h - y
\]

  - negative if pixel above L
  - zero on L
  - positive below L

\( e \) is called the *error function*. 
Bresenham's Algorithm

• How to compute the error?
• Error Function:
  \[ e(x, y) = mx + h - y \]
• Initialize error term \( e = 0 \)

• On each iteration:
  update \( x \): \( x' = x + 1 \)
  update \( e \): \( e' = e + m \)
  if \( e \leq 0.5 \): \( y' = y \) (choose pixel E)
  if \( e > 0.5 \): \( y' = y + 1 \) (choose pixel NE) \( e' = e - 1 \)

Summary of Bresenham

• Initialize \( e = 0 \)
• for \( (x = x_1; x \leq x_2; x++) \)
  – plot \( (x, y) \)
  – update \( x, y, e \)

• Generalize to handle all eight octants using symmetry
• Still using floating point! \( e' = e + m \)
• Can we use only integer arithmetic?
Bresenham with no Floating Point

- Error function \( e(x, y) = mx + h - y \), \( m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{dy}{dx} \)
- At selected pixel \((x, y)\): \( e(x, y) \leq \frac{1}{2} \) \( 2mx + 2h - 2y - 1 \leq 0 \) \( 2x \cdot dy + 2h \cdot dx - 2y \cdot dx - dx \leq 0 \)
- If \( e \leq 0 \) stay horizontal
  If \( e > 0 \) move up
- Update for next pixel:
  - If stay horizontal: \( x += 1 \), \( e += 2dy \)
  - If move up: \( x += 1 \), \( y += 1 \), \( e += 2(dy - dx) \)
- Algorithm = single instruction on graphics chips!

Line Anti-aliasing

- Note: brightness can vary with slope \( \sqrt{2} * L \)
  - What is the maximum variation?
- How could we compensate for this?
  - Answer: antialiasing
How do we remove aliasing?

• Solution 1 - Area Sampling:
  – Treat line as a single-pixel wide rectangle
  – Color pixels according to the percentage of each pixel that is “covered” by the rectangle

• Solution 2 – Super Sampling:
  – Divide pixel up into “sub-pixels”: 2x2, 3x3, 4x4, etc.
  – Pixel color is the average of its sub-pixel colors
  – Easy to implement (in SW and HW). But expensive.
OpenGL Antialiasing

- Can enable separately for points, lines, or polygons
- For points and lines:
  
  ```c
  glEnable(GL_POINT_SMOOTH);
  glEnable(GL_LINE_SMOOTH);
  ```

- For triangles:
  
  ```c
  glEnable(GL_POLYGON_SMOOTH);
  ```

Next:

- Circle Rasterization
Circle Rasterization

• Generate pixels for 2nd octant only
• Slope progresses from 0 → -1
• Analog of Bresenham Segment Algorithm

Circle Rasterization: *Naïve* algorithm

• Circle equation: \(x^2 + y^2 - R^2 = 0\)
• Simple algorithm:
  
  ```
  for x = xmin to xmax
      y = sqrt(R*R - x*x)
      draw pixel(x,y)
  ```

• Work by octants and use symmetry
Circle Rasterization: Bresenham

• Choice between two pixels, E and SE
• Mid-point algorithm:
  – If the midpoint between pixels is inside the circle,
  – E is closer, draw E
  – If the midpoint is outside, SE is closer, draw SE

Circle Rasterization: Bresenham

• In/Out decision function:
  \[ d(x, y) = x^2 + y^2 - R^2 \]
• Compute \( d \) at midpoint btw E, SE

If the last pixel drawn is \((x, y)\), then E = \((x+1, y)\), and SE = \((x+1, y-1)\).
Hence, the midpoint = \((x+1, y-1/2)\).

\[ e(x,y) = (x+1)^2 + (y - 1/2)^2 - R^2 \]
\( e < 0 \): draw E
\( e \geq 0 \): draw SE
Circle Rasterization: Bresenham

• Error Function:
  \[ e(x, y) = (x+1)^2 + (y - 1/2)^2 - R^2 \]

• On each iteration:
  
  update x: \[ x' = x + 1 \]
  update e: \[ e' = e + 2x + 3 \]
  if (e < 0): \[ y' = y \] (choose E)
  if (e \geq 0): \[ y' = y - 1 \] (choose SE), \[ e' = e' - 2y + 2 \]

• Two multiplications, two additions per pixel
• Can you do better?

---

Circle Rasterization: Better Bresenham

• On each iteration:
  
  update x: \[ x' = x + 1 \]
  update e: \[ e' = e + 2x + 3 \]
  if (e < 0): \[ y' = y \] (choose E)
  if (e \geq 0): \[ y' = y - 1 \] (choose SE), \[ e' = e' - 2y + 2 \]

• The error is not linear
• However, what gets added to the error is
• Keep \( \Delta x \) and \( \Delta y \). At each step:
  \[ \Delta x += 2, \Delta y += -2 \]
  \[ e += \Delta x, \text{ and if } y \text{ gets decremented, } e += \Delta y \]
• 4 additions per pixel
Extensions to Other Functions

• Midpoint algorithm easy to extend to any curve defined by: \( f(x,y) = 0 \)
• If the curve is polynomial, can be reduced to only additions using n-order differences

2D Scan Conversion

• Geometric primitive
  – 2D: point, line, polygon, circle...
  – 3D: point, line, polyhedron, sphere...
• Primitives are continuous; screen is discrete
Use line rasterization

• Compute the boundary pixels

Scan-line Rasterization

• Compute the boundary pixels
• Fill the spans
• Requires some initial setup to prepare
Modern Rasterization

For every triangle

Compute Projection
Compute bbox, clip bbox to screen limits
For all pixels in bbox

\textbf{If pixel in triangle}

\texttt{Framebuffer}[x,y]=triangleColor