CMSC 335
COMPUTER GRAPHICS

LECTURE 6

• RASTERIZATION ALGORITHMS
• ANTIALIASING
RASTERIZATION

• **Rasterization** is an object order rendering technique (one object at a time) where we determine which pixels and the color of the pixels that an object has an effect on
  • Extremely efficient

• The rasterizer breaks each primitive into **fragments** by enumerating the pixels that are covered by the primitive and interpolates values (attributes) across the primitive
The Rasterizer is typically implemented in hardware. We cannot alter its algorithms, but can control some settings to it. Important to remember:

- All output data from vertex shading is interpolated across primitives and are input into fragment shading.
RASTERIZATION ALGORITHMS
• After the viewing transformations occur, a point \( \langle w_x, w_y, w_z \rangle \) in the viewport will occupy a pixel located at:

- \( \lfloor w_x \rfloor, \lfloor w_y \rfloor \)

- \( w_z \) is retained typically as the depth information to be used in various operations and stored within the framebuffer.
LINE RASTERIZATION

• The basic problem:
  • Given a line in screen coordinates \((x_0, y_0)\) to \((x_1, y_1)\) (assuming it has been projected) with a slope \(m = \frac{y_1 - y_0}{x_1 - x_0}\) in the range \((0, 1]\) for convenience
  • Let \(f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0\) be the equation of a line
  • Determine the pixels you should fill

• Work with a partner to discover a very simple…brute force solution

While, I will present the algorithms on this assumption, it simply represents one of four cases of a slope that can easily be handled by adopting each algorithm
LINE RASTERIZATION

• Brute force algorithm based on line equation:
  
  ```
  for x = x_0 to x_1 do
    fill_pixel(x, [mx + b])
  ```

• Example of ⟨1, 1⟩ to ⟨7, 3⟩

• Pros and cons?
However, we are not being very clever. We recompute the line every time.

Upon a closer look
  - We only highlight \( \langle x_i + 1, y_i \rangle \)
    or \( \langle x_i + 1, y_i + 1 \rangle \)
LINE RASTERIZATION

• Midpoint algorithm:
  
  \[ y = y_0 \]

  \[ \text{for } x = x_0 \text{ to } x_1 \text{ do} \]
  
  fill_pixel(\(x, y\))

  \[ \text{if } f(x + 1, y + 0.5) < 0 \text{ then} \]
  
  \[ y = y + 1 \]

• Example of \((1, 1)\) to \((7, 3)\) with the midpoints shown
LINE RASTERIZATION

• Pros and cons of the midpoint algorithm?
  • Not using the slope information at all
  • Uses floating point arithmetic

• Incremental algorithm built off of the following observations:
  • \( f(x+1, y) = f(x, y) + (y_0 - y_1) \)
  • \( f(x+1, y+1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \)
LINE RASTERIZATION

• Incremental Midpoint Algorithm
  \[ y = y_0 \]
  \[ d = f(x_0 + 1, y_0 + 0.5) \]
  \[ \text{for } x = x_0 \text{ to } x_1 \text{ do} \]
  \[ \text{fill}_\text{pixel}(x, y) \]
  \[ \text{if } d < 0 \text{ then} \]
  \[ y = y + 1 \]
  \[ d = d + (x_1 - x_0) + (y_0 - y_1) \]
  \[ \text{else} \]
  \[ d = d + (y_0 - y_1) \]

• Trace the incremental algorithm on \( \langle 23,30 \rangle \) to \( \langle 27,33 \rangle \)
LINE RASTERIZATION

• Incremental midpoint algorithm pros and cons?
  • Still using floating point arithmetic

• Bresenham's Algorithm is integer only computation

• Midpoint (and Bresenham's) algorithm can easily be extended to other shapes, e.g., circles or curves!
TRIANGLE RASTERIZATION

• Triangles are defined by 3 points \( \vec{p}_0 = \langle x_0, y_0 \rangle, \vec{p}_1 = \langle x_1, y_1 \rangle, \) and \( \vec{p}_2 = \langle x_2, y_2 \rangle \)
• Determining whether a point is within a triangle is an "easy" task using Barycentric Coordinates \( \beta, \gamma \)
• Additionally, Barycentric Coordinates are centric to interpolation
TRIANGLE RASTERIZATION

• Barycentric Coordinates:
  • Any point within the triangle can be written like:
    \[ \vec{p} = \vec{p}_0 + \beta (\vec{p}_1 - \vec{p}_0) + \gamma (\vec{p}_2 - \vec{p}_0) \]
  • Reordered:
    \[ \vec{p} = (1 - \beta - \gamma) \vec{p}_0 + \beta \vec{p}_1 + \gamma \vec{p}_2 \]
  • So let \( \alpha = 1 - \beta - \gamma \) for simplicity
  • Important that \( \alpha + \beta + \gamma = 1 \) and for any point in the triangle \( \vec{p} \),
    \( \alpha \in (0,1), \beta \in (0,1), \) and \( \gamma \in (0,1) \)
TRIANGLE RASTERIZATION

- **Barycentric Coordinates:**
  - Can be represented as signed distances from the lines making up the sides:
    \[
    \beta = \frac{f_{\vec{p}_0\vec{p}_2}(x, y)}{f_{\vec{p}_0\vec{p}_2}(x_1, y_1)}
    \]
  - We take this ratio to normalize the distance of \(\vec{p}\) to \(\vec{p}_0\vec{p}_2\) by the distance of \(\vec{p}_1\) to \(\vec{p}_0\vec{p}_2\), because we want \(\beta = 1\) at \(\vec{p}_1\)
  - Similar for gamma:
    \[
    \gamma = \frac{f_{\vec{p}_0\vec{p}_1}(x, y)}{f_{\vec{p}_0\vec{p}_1}(x_2, y_2)}
    \]
  - Compute \(\alpha = 1 - \beta - \gamma\) and compare these values to the ranges \((0, 1)\)
TRIANGLE RASTERIZATION

• Barycentric Coordinates:
  • Can similarly view $\beta$, $\gamma$, and $\alpha$ as ratios of the areas of sub-triangles to the whole triangle, and compute through cross products
  
  $\beta = \frac{(\vec{p}_0 - \vec{p}_0) \times (\vec{p}_2 - \vec{p}_0)}{(\vec{p}_1 - \vec{p}_0) \times (\vec{p}_2 - \vec{p}_0)}$
  
  $\gamma = \frac{(\vec{p}_1 - \vec{p}_0) \times (\vec{p} - \vec{p}_0)}{(\vec{p}_1 - \vec{p}_0) \times (\vec{p}_2 - \vec{p}_0)}$
  
  $\alpha = 1 - \beta - \gamma$

Defining Barycentric Coordinates is a very common interview question.
TRIANGLE RASTERIZATION

• Interpolating values is a weighted ratio of their vertices, i.e., a weighting of Barycentric coordinates, called Gouraud interpolation
  • Example of color: \( c = \alpha c_0 + \beta c_1 + \gamma c_2 \)

• To avoid edge problems – only fill a pixel when its lies center within the triangle

• Brute force rasterization:
  
  for all \( x \) do
  
    for all \( y \) do
      
        compute \( \alpha, \beta, \gamma \) for \( (x,y) \)
        
        if \( \text{isInTriangle}(\alpha, \beta, \gamma) \) then
          
            interpolateData(\( \alpha, \beta, \gamma \))
            
            fillPixel(\( x, y \))
  

• Reflections? Pros/cons?

Alternatively could draw each of the three lines and perform a filling algorithm, e.g., flood fill.
TRIANGLE RASTERIZATION

• Exercise – build upon the concepts of line drawing, i.e., a move to an incremental algorithm, to write a more efficient filling algorithm
  • What optimizations can you achieve?
TRIANGLE RASTERIZATION

• Reflection
  • Extents of filling – Only box? Or did you consider tracking a start/end to each "scan line"
  • Incremental allows precomputing many values
  • Could you remove floating point values? – unlikely as interpolation needs them
  • Did you handle edges of triangle differently?
    • Consider two adjacent triangles in a mesh – must dedicate one of these for edges…
  • Handle degeneracy?

• See book for some more details.
CLIPPING

• Rasterizing is inherently expensive, so the cost of a pipeline is typically tied to this, i.e., how many triangles are in a scene

• We want to ensure we don't rasterize things that cannot be seen

• Consider a triangle intersecting the view volume
  • What about triangles outside of the view volume?

We should only render this part
CLIPPING

• Clipping is typically performed as a pre-rasterization step before homogeneous coordinates are normalized. Basic approach:

  for each of the six view volume planes do
    if triangle is outside of view volume then
      break (do not rasterize)
    else if triangle spans plane then
      clip triangle
      if a quadrilateral is left then
        split into two triangles

You can also clip in world space, or with arbitrary user defined planes.
What are the six planes? Using homogeneous coordinates, the hyperplanes are:

\[-x + lw = 0,\]
\[x - rw = 0,\]
\[-y + bw = 0,\]
\[y - tw = 0,\]
\[-z + nw = 0,\]
\[z - fw = 0\]
To clip:

- Recall the plane equation: \( f(\vec{p}) = \hat{n} \cdot (\vec{p} - \vec{a}) = 0 \), where \( \vec{a} \) is a point on the plane and \( \hat{n} \) is the plane normal.

- Idea, for each line segment, clip the line against the plane:
  - First test is two points \( \vec{p}_0 \) and \( \vec{p}_1 \) are on opposite sides of the plane, i.e., \( f(\vec{p}) < 0 \) means \( \vec{p} \) is inside the plane, and \( f(\vec{p}) > 0 \) means \( \vec{p} \) is outside of the plane.
  - Convert to a parametric representation:
    \[
    \vec{p} = \vec{p}_0 + t(\vec{p}_1 - \vec{p}_0)
    \]
  - Substitute into the plane equation….wait this looks too familiar!
    - Same as a ray trace
    - Solve for intersection point
CLIPPING

• Breaking a quadrilateral into two triangles
  • Keep a standard convention:
    • $\vec{p}_0, \vec{p}_1', \vec{p}_2'$
    • $\vec{p}_2', \vec{p}_2, \vec{p}_0$
CLIPPING

• Exercise

• Try to develop a clipping algorithm for more complex polygons. What difficulties do you see?
RASTERIZATION OPERATIONS
2D DRAWING

- To mimic 2D
  - Orthographic view
  - Typically very simple shaders that position objects and overwrite color values of fragments
PAINTERS ALGORITHM

• Main problem with rasterization is that it is order dependent
  • Both in terms of efficiency and output

• The painter’s algorithm renders from back to front (handled in software)

• Consider the following
  • What is the problem?
  • What do you do?
Z-BUFFER

• A typical solution is to use z-buffering

• Essentially:
  \[
  \text{if new pixel's z > saved pixel's z then}
  \]
  overwrite pixel

• Depth values are additionally stored in the frame buffer.

• Implemented in post-fragment operations.

• How can you handle alpha blending?
PER-VERTEX SHADING

• Per-vertex shading computes colors during the vertex shader and uses the rasterizer to interpolate colors across a triangle

• Cheap as the vertex shader is run fewer times than fragment shader

• Called Gouraud shading
PER-FRAGMENT SHADING

• Per-fragment shading computes colors during the fragment shader and instead uses the rasterizer to interpolate positions and normals across a triangle

• More expensive than Gouraud shading

• Called Phong shading
WHICH TO USE?

• Depends on sizes of primitives and detailing that you want to achieve. Put another way on how fast the color changes along primitive

  • Large scale affects can be computed in vertex stage
  • Small scale and drastic changes should be computed in fragment stage

• Depends on time allotted for rendering
TEXTURE MAPPING

• Typically done in the fragment stage
• Performed using a texture lookup (more on this after Exam 1)
• Like gluing an image to a face

"Glue" the image on the object
ANTIALIASING TECHNIQUES
SIMPLE ANTIALIASING

• **Supersampling** – creating an image at very high resolution and then downsample each pixel. For example, rasterize at 4x resolution and then average every set of 4 pixels into one color.
  • Works well for objects that are not extremely small relative to the pixel distance

• **Multisampling** – determine how much of a pixel a primitive covers and tone down the color based on this value
CULLING PRIMITIVES
**Basic Culling**

- **Culling** refers to throwing away invisible geometry to save time spent processing it
  - **View volume culling** – the removal of geometry that is outside of the view volume
  - **Occlusion culling** – the removal of geometry that is obscured, or occluded, by other geometry in the scene
  - **Backface culling** – the removal of primitives facing away from the camera
VIEW VOLUME CULLING

• A test to determine if an entire primitive lies outside of the viewing volume because it would not produce any fragments when rasterized.

• Tricky part – individual testing of each primitive could be more expensive than just rasterizing them.

• Solution – test groups of primitives.
VIEW VOLUME CULLING

• One method
  • Use a bounding volume, e.g., a sphere, around a group of primitives
  • If the sphere lies outside of the view volume than all of the object lies outside
    \[(\vec{c} - \vec{a}) \cdot \hat{n} > r\]
  • Otherwise, the primitives may or may not lie outside

• Conservative test

• Could use a bounding volume hierarchy
BACKFACE CULLING

• Polygonal meshes typically are closed and have outwardly facing normal.

• Thus, any triangle with a normal in the opposite direction of the camera direction
  \[ \hat{n} \cdot \hat{w} < 0 \]
  can be removed from rasterizing.

• Culled before the pipeline even starts.
EXAM FORMAT

• Allowed a "hack" sheet (1-page 2 sides handwritten)

• 5 or so questions
  • Q1 – T/F and Fill-in-the-blank like quizzes
  • Q2 – Short 1-line answers like quizzes
  • Q3 – Pipeline diagramming of some sort
  • Q4/Q5 – Application of math/algorithms
  • Bonus

• FDG Chapters 1-8
  • Ray tracing
  • Rasterizing

• No "code" questions