CENG – 477
Introduction to Computer Graphics
Rasterization
Rasterization

- Rasterization is concerned with creating **fragments** from vertices
- It works in screen coordinates – thus it is next step after the viewport transform
- **Goal:** Given a set of vertices which fragments must be “turned on” to create the primitive:
Rasterization

- **Subproblems:**
  - How to deal with different primitives
  - How to make it fast
  - How to interpolate color and other attributes
- We’ll start with line rasterization
Line Rasterization

- Several methods exist
- **Option 1:** Treat the line as a thin rectangle and turn on pixels that are inside this rectangle

  - May cause gaps in the line
  - Too thin rectangle

  - Line thickness varies
  - Too thick rectangle
Line Rasterization

- What must be the minimum thickness?

- May not draw anything if it is too thin

- Must be at least one to draw a horizontal line in some cases
Line Rasterization

- But a thickness of 1 may result in too thick lines

- This segment is $>1$
  - thus may contain more than 1 pixel
Line Rasterization

• Therefore we need another solution
• We can use line equations to decide which pixels belong the line
• Assume that we want to draw a line between two screen coordinates: \((x_0, y_0)\) to \((x_1, y_1)\)
• Assume that the slope of the line, \(m\), is in range \((0, 1]\)

\[ m = \frac{y_1 - y_0}{x_1 - x_0} \]

Which pixels should we draw?
Line Equation

• Let’s first remember the implicit line equation:

\[ f(x, y) = x(y_0 - y_1) + y(x_1 - x_0) + x_0y_1 - y_0x_1 \]

• We can derive it from geometry:

\[ \mathbf{n} \cdot [x_1 - x_0, y_1 - y_0]^T = 0 \]
\[ \mathbf{n} = [y_0 - y_1, x_1 - x_0] \]
\[ f(p) = (p - p_0) \cdot \mathbf{n} = 0 \]

What is the meaning of \( f(p) = 0 \), \( f(p) > 0 \) and \( f(p) < 0 \)
The Basic Algorithm

• The basic algorithm is as follows:

\[
\begin{align*}
y & = y_0 \\
\text{for } x = x_0 \text{ to } x_1 \text{ do:} \\
& \quad \text{draw}(x, y) \\
& \quad \textbf{if} \ (\text{some condition}) \ \textbf{then}: \\
& \quad \quad y = y + 1
\end{align*}
\]

• Because the slope is in (0, 1], we always go right and sometimes go up (assuming that \(x_0 < x_1\) and \(y_0 < y_1\))
The Midpoint Algorithm

- Assume we have just drawn \((x, y)\). Which pixel to draw next?

- Compute the midpoint of the next pixel
- If the midpoint is on or above the line choose right pixel (E: east)
- If the midpoint is below the line, choose the top right pixel (NE: north-east)

\[
y = y_0 \\
\text{for } x = x_0 \text{ to } x_1 \text{ do:} \\
\quad \text{draw}(x, y) \\
\quad \text{if } f(x+1, y+0.5) < 0 \text{ then:} \\
\quad \quad y = y + 1
\]

We call these pixels E and NE
The Midpoint Algorithm

- This algorithm works well but requires evaluating the line equation at every iteration
- Can be optimized with an **incremental** algorithm:

  - If we selected E, in the next iteration we need the value of \( f(M_1) \)
  - If we selected NE, in the next iteration we need \( f(M_2) \)
  - Both can be computed from \( f(M_0) \)

\[
f(M_1) - f(M_0) = y_0 - y_1
\]
\[
f(M_2) - f(M_0) = (y_0 - y_1) + (x_1 - x_0)
\]
The Midpoint Algorithm

• This algorithm works well but requires evaluating the line equation at every iteration
• Can be optimized with an incremental algorithm:

\[ f(M_0) = f(x_0 + 1, y_0 + 0.5) \]
\[ = (y_0 - y_1) + 0.5(x_1 - x_0) \]
The Midpoint Algorithm

• So the overall algorithm is:

\[
y = y_0 \\
d = (y_0 - y_1) + 0.5(x_1 - x_0) \\
\textbf{for } x = x_0 \textbf{ to } x_1 \textbf{ do:} \\
\hspace{1cm} \text{draw}(x, y) \\
\hspace{1cm} \textbf{if } d < 0 \textbf{ then: } // \text{choose NE} \\
\hspace{2cm} y = y + 1 \\
\hspace{2cm} d += (y_0 - y_1) + (x_1 - x_0) \\
\hspace{1cm} \textbf{else: } // \text{choose E} \\
\hspace{2cm} d += (y_0 - y_1)
\]
The Midpoint Algorithm

• For max efficiency, \( f(x, y) = 0 \) is written as \( 2f(x, y) = 0 \). This entirely eliminates floating point operations:

\[
\begin{align*}
y &= y_0 \\
d &= 2(y_0 - y_1) + (x_1 - x_0) \\
\text{for } x = x_0 \text{ to } x_1 \text{ do:} \\
\quad \text{draw}(x, y) \\
\quad \text{if } d < 0 \text{ then: } // \text{ choose NE} \\
\quad \quad y &= y + 1 \\
\quad \quad d &= d + 2[(y_0 - y_1) + (x_1 - x_0)] \\
\quad \text{else: } // \text{ choose E} \\
\quad \quad d &= d + 2(y_0 - y_1)
\end{align*}
\]
The Midpoint Algorithm

- The presented algorithm works when the slope of the line, \( m \), is in range \((0, 1]\)
- For other slope ranges, minor modifications to the algorithm is required
- For instance if \( m \in (1, \infty) \), we need to swap the roles of \( x \) and \( y \)
- Other cases must be adapted similarly
Float vs Integer

• The midpoint algorithm was originally developed by Pitteway in 1967
• Floating point arithmetic was very expensive at the time
• Does it still matter?
• We implemented both algorithms and drew one million lines each between 1000 and 1400 pixels long:
  – Test run on Intel Core i7 CPU at 3.2 GHz
  – Compiled with g++ and –O2 option
  – Basic algorithm: 7.2 seconds
  – Optimized algorithm: 3 seconds
  – So it still makes a difference!
Interpolating Attributes

• What if the two end points of the line have a different color?
• The color across the line must smoothly change:
Interpolating Attributes

• What if the two end points of the line have a different color?
• The color across the line must smoothly change:
Interpolating Attributes

For $m \in [0,1]$ it is better to interpolate in the $x$ direction as it will produce a smoother variation.
Interpolating Attributes

• Assume that the color of the endpoints are \( c_0 \) and \( c_1 \)
• The color of an intermediate point should be:

\[
c = (1 - \alpha)c_0 + \alpha c_1
\]

where \( \alpha \) is the interpolation variable

• At any pixel \((x, y)\), we can compute it based on the horizontal or vertical distance of the pixel to the first endpoint:

\[
\alpha = \frac{x - x_0}{x_1 - x_0}
\]
Interpolating Attributes

• Assume that the color of the endpoints are \( c_0 \) and \( c_1 \)
• The color of an intermediate point should be:

\[
c = (1 - \alpha)c_0 + \alpha c_1
\]

where \( \alpha \) is the interpolation variable

• It can also be computed incrementally

\[
\alpha(x_0 + 1) = \alpha_1 = \frac{1}{x_1 - x_0}
\]

\[
\alpha(x_0 + 2) = \alpha_2 = \alpha_1 + \frac{1}{x_1 - x_0}
\]
Algorithm with Interpolation

- Algorithm with color interpolation:

\[
\begin{align*}
  y &= y_0 \\
  d &= (y_0 - y_1) + 0.5(x_1 - x_0) \\
  c &= c_0 \\
  dc &= (c_1 - c_0) / (x_1 - x_0) \quad // \text{skip } \alpha; \text{ directly compute color increment}
\end{align*}
\]

\[
\text{for } x = x_0 \text{ to } x_1 \text{ do:}
\]
\[
\begin{align*}
  \text{draw}(x, y, \text{round}(c)) \\
  \text{if } d < 0 \text{ then: } // \text{choose NE} \\
  & \quad y = y + 1 \\
  & \quad d += (y_0 - y_1) + (x_1 - x_0) \\
  \text{else: } // \text{choose E} \\
  & \quad d += (y_0 - y_1) \\
  & \quad c += dc
\end{align*}
\]
Triangle Rasterization

- Initially all we have is the screen coordinates of three vertices

\[(x_0, y_0)\]

\[(x_1, y_1)\]

\[(x_2, y_2)\]
Triangle Rasterization

• From them we compute line equations for the edges:

\[ f_{01}(x, y) = x(y_0 - y_1) + y(x_1 - x_0) + x_0 y_1 - y_0 x_1 \]
\[ f_{12}(x, y) = x(y_1 - y_2) + y(x_2 - x_1) + x_1 y_2 - y_1 x_2 \]
\[ f_{20}(x, y) = x(y_2 - y_0) + y(x_0 - x_2) + x_2 y_0 - y_2 x_0 \]
Triangle Rasterization

- We then walk across the viewport to determine inside pixels:
Triangle Rasterization

- For efficiency we may only walk within the bounding box of the triangle:

  - For each pixel we visit, we must make an inside test with respect to all three edges
  - We can simply plug-in the (x, y) value of the visited pixel to each line equation
  - If all are negative, the pixel is inside the triangle
Triangle Rasterization

• However, using barycentric coordinates will help us with interpolation of attributes

What are the barycentric coordinates of \((x, y)\)?

\[
\alpha = \frac{A_0}{A}, \quad \beta = \frac{A_1}{A}, \quad \gamma = \frac{A_2}{A}
\]
Triangle Rasterization

- We don’t actually need to compute the areas as the bases are shared and will cancel out

\[ \alpha = \frac{A_0}{A}, \beta = \frac{A_1}{A}, \gamma = \frac{A_2}{A} \]

\[ \alpha = \frac{f_{12}(x, y)}{f_{12}(x_0, y_0)} \]
\[ \beta = \frac{f_{20}(x, y)}{f_{20}(x_1, y_1)} \]
\[ \gamma = \frac{f_{01}(x, y)}{f_{01}(x_2, y_2)} \]

What are the barycentric coordinates of \((x, y)\)?
Overall Algorithm

\[
\text{for } y = y_{\text{min}} \text{ to } y_{\text{max}} \text{ do:} \\
\quad \text{for } x = x_{\text{min}} \text{ to } x_{\text{max}} \text{ do:} \\
\quad \quad \alpha = \frac{f_{12}(x, y)}{f_{12}(x_0, y_0)} \\
\quad \quad \beta = \frac{f_{20}(x, y)}{f_{20}(x_1, y_1)} \\
\quad \quad \gamma = \frac{f_{01}(x, y)}{f_{01}(x_2, y_2)} \\
\quad \quad \text{if } \alpha \geq 0 \text{ and } \beta \geq 0 \text{ and } \gamma \geq 0 \text{ then:} \\
\quad \quad \quad \quad c = \alpha c_0 + \beta c_1 + \gamma c_2 \\
\quad \quad \quad \quad \text{draw}(x, y, \text{round}(c))
\]

- Note that the computation of the barycentric coordinates can also be made incremental for greater efficiency
Summary

• The result of rasterization is a set of fragments (pixel-to-be) for each primitive

• Each fragment has interpolated values of attributes:
  – Color values
  – Texture coordinates
  – Depth value
  – Normals
  – Or any user-defined attribute for vertices
CENG – 477
Introduction to Computer Graphics
Fragment Processing
Fragment Processing

• The previous stages of the pipeline (up to rasterization) is generally known as the vertex pipeline
• **Rasterization** creates a set of **fragments** that make up the interior region of the primitive
• The rest of the pipeline which operates on these fragments is called the **fragment pipeline**
• Fragment pipeline is comprised of several operations
• The end result of fragment processing is the update of corresponding locations in the **framebuffer**
Fragment Processing

Vertices → Vertex Pipeline → Rasterization → Fragment Pipeline → Framebuffer

Framebuffer → Color Buffer → Depth Buffer → Stencil Buffer
Fragment Processing

- Fragment pipeline is comprised of many stages:
  - Following is OpenGL’s handling of the fragment pipeline
  - Different renderers may implement a different set of stages
Depth Buffer Test

• Among these, the **depth buffer test** is very important to render primitives in correct order

• Without depth buffer, the programmer must ensure to render primitives in a back to front order
  – Known as painter’s algorithm:

From wikipedia.com
Depth Buffer Test

- Binary space partitioning (BSP) trees may be used for this purpose
- However, they are costly to generate and may require splitting primitives due to impossible ordering cases:

From wikipedia.com
Depth Buffer Test

• When memory was a very valuable resource, such algorithms were implemented
• Quake3 was one of the main games that used painter’s algorithm using BSP trees
• Each game level was stored as a huge BSP tree
  – Read more at: https://www.bluesnews.com/abrash/chap64.shtml
Depth Buffer Test

• **Main Idea:**
  – At each pixel, keep track of the distance to the closest fragment that has been drawn in a separate buffer
  – Discard fragments that are further away than that distance
  – Otherwise, **draw** the fragment and **update** the z-buffer value with the z value of that fragment

• Requires an extra memory region, called the **depth buffer**, to implement this solution

• At the beginning of every frame, the depth buffer is reset to **infinity** (1.0f in practice if the depth range is [0.0f, 1.0f])

• Depth buffer is also known as **z-buffer**
### Example

<table>
<thead>
<tr>
<th>Initial state of depth buffer</th>
<th>z-values of the first triangle</th>
<th>Resulting depth buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="wikipedia.com" alt="Initial state" /></td>
<td><img src="wikipedia.com" alt="First triangle" /></td>
<td><img src="wikipedia.com" alt="Resulting depth buffer" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial depth buffer</th>
<th>First triangle</th>
<th>Resulting depth buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="wikipedia.com" alt="Initial depth buffer" /></td>
<td><img src="wikipedia.com" alt="First triangle" /></td>
<td><img src="wikipedia.com" alt="Resulting depth buffer" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>z-values of the second triangle</th>
<th>Resulting depth buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="wikipedia.com" alt="Second triangle" /></td>
<td><img src="wikipedia.com" alt="Resulting depth buffer" /></td>
</tr>
</tbody>
</table>
Depth Range

• The range of values written to the depth buffer can generally be controlled by the programmer.

• In OpenGL, the command `glDepthRange(zMin, zMax)` is used.

• The default depth range is [0, 1].

• The z-value in the canonical viewing volume (CVV), which is in range [-1, 1] is scaled to this range during the viewport transform.

• `glDepthRange` is to the z-values what `glViewport(x, y, width, height)` is to the x- and y-values.
Z-Fighting

- Remember that the z-values get compressed to [0, 1] range from the [-n:-f] range after projection and viewport transforms
- Observe how it looks for n = 10 and f = 50
Z-Fighting

- Remember that the z-values get compressed to [0, 1] range from the [-n:-f] range after projection and viewport transforms
- Observe the same for n = 10 and f = 200
Z-Fighting

- With a limited precision depth buffer, fragments that are close in depth may get mapped to the same z-value
Z-Fighting

• The compression is more severe for with larger depth range
• This may cause a problem known as z-fighting:
  – Objects with originally different (but close) z-values get mapped to the same final z-value (due to limited precision) making it impossible to distinguish which one is in front and which one is behind
Z-Fighting

• To avoid z-fighting, the depth range should be kept as small as possible for keeping the compression less severe

• Alternatively, a floating point depth buffer can used
  – Unavailable in older hardware
  – Supported in all modern GPUs

• Finally, the command `glPolygonOffset` can be used to push and pull polygons a little to avoid z-fighting
Scissor Test

- Scissor test is a per-fragment operation that discards fragments outside a certain rectangular region.

In OpenGL, `glScissor` command is used for this purpose. Note that this operation is different from clipping.
Stencil Test

- While scissor test can be used to mask out rectangles, stencil test can be used to mask arbitrary fragments.
- Requires a different buffer known as the stencil buffer.

Original  Stencil Buffer  Result

From research.ncl.ac.uk
Stencil Test

- Typically depth and stencil buffers are combined to produce single buffer made of 24-bit depth and 8-bit stencil information for each pixel

<table>
<thead>
<tr>
<th>Pixel 0</th>
<th>Pixel 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>S</td>
</tr>
<tr>
<td>D</td>
<td>S</td>
</tr>
<tr>
<td>D</td>
<td>S</td>
</tr>
<tr>
<td>D</td>
<td>S</td>
</tr>
<tr>
<td>:</td>
<td>...</td>
</tr>
<tr>
<td>D</td>
<td>S</td>
</tr>
<tr>
<td>D</td>
<td>S</td>
</tr>
<tr>
<td>D</td>
<td>S</td>
</tr>
</tbody>
</table>

Pixel 0

Pixel 1
Stencil Test

- Stencil buffer and stencil test can also be used to implement one type of shadowing algorithms (we’ll learn this later)
Alpha Blending

- Alpha blending is another fragment operation in which new objects can be blended with the existing contents of the color buffer for a variety of effects.

Without blending  With blending
Summary

- At the end of the pipeline, input vertices with connectivity information end up populating certain regions of the framebuffer.
- This pipeline can be implemented on the software (CPU), hardware (GPU) or both (CPU + GPU).