Triangles
Lots of Triangles

Stanford Bunny
69,451 triangles

David (Digital Michelangelo Project)
56,230,343 triangles
Why Triangles?

- Can focus on specializing/optimizing the geometry pipeline for only one geometric primitive
- Software and algorithms can be optimized for one geometric primitive
- Hardware (e.g. GPUs) can be specialized to treat one geometric primitive

- Triangles have many inherent benefits:
  - Even complex objects can be well approximated (piecewise linear convergence) with enough triangles
  - Easy to break other polygons into triangles
  - Triangles are guaranteed to be planar (unlike quadrilaterals)
  - Transformations (from last lecture) only need be applied to the triangle vertices
  - Barycentric interpolation can be used to robustly interpolate information from the triangle’s vertices to the triangle’s interior
  - Etc.
OpenGL

- Blender uses OpenGL for its real-time scanline renderer
- OpenGL was started by SGI in 1991 (went into the public domain in 2006)
- It’s a drawing API for 2D/3D graphics
- Designed to be implemented mostly on hardware
- Many books and other documentation
- Main competitor is DirectX

- OpenGL is highly optimized for triangles:
GPUs and Gaming Consoles

- GPUs and Consoles are highly optimized for the graphics geometry pipeline
- They now support ray tracing, as does Blender
Rasterization

- Use screen space projection to transform triangle vertices to screen space
- Find all the pixels inside the projected 2D triangle
- Color the pixels inside the triangle with the RGB-color of the triangle
Aside: Bounding Box Acceleration

• Checking every pixel against every triangle is computationally expensive
• Calculate a bounding box around the triangle, with diagonal corners:
  \[(\min(x_0, x_1, x_2), \min(y_0, y_1, y_2)) \text{ and } (\max(x_0, x_1, x_2), \max(y_0, y_1, y_2))\]
• Then, round coordinates upward to the nearest integer to find all relative pixels
Implicit Equation for a 2D line

- Compute a directed edge vector $e = p_1 - p_0 = (x_1 - x_0, y_1 - y_0)$
- Compute the 2D normal $n = (y_1 - y_0, -(x_1 - x_0))$, which doesn’t need be unit length
- This 2D normal is *rightward* with respect to the 2D ray direction (leftward normal is $-n$)
- Points $p$ lying exactly on the 2D line have: $(p - p_0) \cdot n = 0$
- This is the same equation used for planes in 3D
(Leftward) “Interior” Side of a 2D Ray

- Points $p$ on the “interior” side of the 2D ray have: $(p - p_0) \cdot n < 0$
- Points $p$ exactly on the 2D line have: $(p - p_0) \cdot n = 0$
- Points $p$ on the “exterior” side of the 2D ray have: $(p - p_0) \cdot n > 0$
- This same concept can be used for planes in 3D
A 2D point is inside a 2D triangle, if it is interior to (to the left of) all 3 rays.

- **Vertex order matters**: backward facing triangles are not rendered, since no points are to the left of all three rays.
Boundary Cases

- Pixels lying exactly on a triangle boundary with \((p - p_0) \cdot n = 0\) for one of the edges won’t be rendered
  - Can cause gaps between adjacent triangles sharing an edge, when that shared edge overlaps a pixel
- Changing the inside test to \((p - p_0) \cdot n \leq 0\) instead of \((p - p_0) \cdot n < 0\) rectifies the problem, but now both triangles attempt to color the same pixel
  - Inefficient, and can cause disagreements that lead to artifacts
- Instead, points on the shared edge should be consistently rendered with one triangle or the other
  - The edge normals point in opposite directions for the two adjacent triangles
  - When \(n_x > 0\) or \((n_x = 0 \text{ and } n_y > 0)\), rasterize pixels on that edge
  - When \(n_x < 0\) or \((n_x = 0 \text{ and } n_y < 0)\), do not rasterize pixels on that edge
  - Note: \(n_x\) and \(n_y\) are never both zero for non-degenerate 2D triangles
Overlapping Triangles

• If one object is in front of another, two triangles may both try to color the same pixel.

• Recall (last lecture), the perspective projection computes depth map values \( z' = n + f - \frac{fn}{z} \) that can be used for occlusion/transparency (via the alpha channel).

• Thus, color the pixel based on which triangle has a smaller \( z' \) value.

• This requires interpolating \( z' \) values from the vertices of the triangle to the pixel locations.

• In order to do this, we use *proper* screen space barycentric weight interpolation.
1D Linear Interpolation

- Given two points \((x_1, y_1)\) and \((x_2, y_2)\) in 1D, one can linearly interpolate between them via 
  
  \[ y(x) = \frac{y_2-y_1}{x_2-x_1} x - \frac{y_2-y_1}{x_2-x_1} x_1 + y_1, \]

  which rearranges to 
  
  \[ y(x) = \left(1 - \frac{x-x_1}{x_2-x_1}\right) y_1 + \frac{x-x_1}{x_2-x_1} y_2 \]

- Alternatively, one can write 
  
  \[ y(t) = (1 - t)y_1 + ty_2 \]

  where 
  
  \[ t = \frac{x-x_1}{x_2-x_1} \]

  ranges from 0 to 1 and can be seen as the fraction of the way from \(x_1\) to \(x_2\)
2D/3D Line Segments

- This concept can be extended to line segments in both 2D and 3D.
- Given endpoints $p_0$ and $p_1$, intermediate points are defined based on the fraction of the distance that point is from $p_0$ to $p_1$ via $p(t) = (1 - t)p_0 + tp_1$.
- Here, $t$ is calculated via $t = \frac{\|p - p_0\|_2}{\|p_1 - p_0\|_2}$ since $p_0$ and $p_1$ are multidimensional points.
- Barycentric weights reformulate this using weights $\alpha_0, \alpha_1 \in [0,1]$ where $\alpha_0 + \alpha_1 = 1$ and $p = \alpha_0 p_0 + \alpha_1 p_1$, i.e. $\alpha_0 = \frac{\|p - p_1\|_2}{\|p_1 - p_0\|_2}$ and $\alpha_1 = \frac{\|p - p_0\|_2}{\|p_1 - p_0\|_2}$.
- Barycentric weights express any point $p$ on the segment as a linear combination of the endpoints of the segment.
2D/3D Triangles

• To extend to triangles with 3 vertices, computes 3 barycentric weights $\alpha_0, \alpha_1, \alpha_2 \in [0,1]$ with $\alpha_0 + \alpha_1 + \alpha_2 = 1$ and $p = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2$

• The weights are computed via areas, i.e.

$$\alpha_0 = \frac{\text{Area}(p,p_1,p_2)}{\text{Area}(p_0,p_1,p_2)} \quad \text{and} \quad \alpha_1 = \frac{\text{Area}(p_0,p,p_2)}{\text{Area}(p_0,p_1,p_2)} \quad \text{and} \quad \alpha_2 = \frac{\text{Area}(p_0,p_1,p)}{\text{Area}(p_0,p_1,p_2)}$$

• Note the triangle area formula: $\text{Area}(p_0, p_1, p_2) = \frac{1}{2} \| \overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2} \|_2$
(Alternative) Algebraic Approach

- Rewrite $\alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2 = p$ as $\alpha_0 \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + \alpha_1 \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} + (1 - \alpha_0 - \alpha_1) \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$

- Assemble into matrix form: 
  $$\begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \\ z_0 - z_2 & z_1 - z_2 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} x - x_2 \\ y - y_2 \\ z - z_2 \end{pmatrix}$$

- In 2D, this is a 2x2 coefficient matrix, but in 3D one has to use the normal equations to reduce to a 2x2 system, i.e. convert $A \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = b$ to $A^T A \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = A^T b$

- The coefficient matrix is rank 1 when the two vectors are colinear, implying infinite solutions for triangles with zero area (one can still embed $p$ on an edge)

- Otherwise, invert the 2x2 coefficient matrix to solve the system of 2 equations with 2 unknowns
Triangle Basis Vectors

- Compute edge vectors \( u = p_1 - p_0 \) and \( v = p_2 - p_0 \)
- Then, any point \( p \) interior to the triangle can be written as \( p = p_0 + \beta_1 u + \beta_2 v \) where \( \beta_1, \beta_2 \in [0,1] \) and \( \beta_1 + \beta_2 \leq 1 \)
- Substitutions and collecting terms gives \( p = (1 - \beta_1 - \beta_2)p_0 + \beta_1 p_1 + \beta_2 p_2 \) implying the equivalence: \( \alpha_0 = 1 - \beta_1 - \beta_2 \), \( \alpha_1 = \beta_1 \), \( \alpha_2 = \beta_2 \)
Perspective Projection

• A triangle in world space with vertices $p_0, p_1, p_2$ is projected into screen space, vertex by vertex, to obtain $p'_0, p'_1, p'_2$ where $x' = \frac{hx}{z}$ and $y' = \frac{hy}{z}$ for all $x$ and $y$

• A point on the triangle in world space $p = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2$ is projected into screen space to a point $p'$

• Notably, $p' \neq \alpha_0 p'_0 + \alpha_1 p'_1 + \alpha_2 p'_2$ because the perspective projection is highly nonlinear

• The barycentric weights that describe the interior of the triangle in world space do not still hold after projecting the vertices into screen space

• Thus, it is unclear how to compute $z'$ at a pixel from the $z'$ values at the vertices of the screen space triangle

• The $z'$ values are not linear with respect to the triangle vertices in screen space, only in world space

• If we knew the location of the pixel on the world space triangle, we could use barycentric interpolation on the world space triangle to compute $z'$ for the pixel
Screen Space Barycentric Weights

- Given a pixel at \( p' \), compute screen space barycentric weights so that \( p' = \alpha'_0 p'_0 + \alpha'_1 p'_1 + (1 - \alpha'_0 - \alpha'_1)p'_2 \)
- Define 2D triangle basis vectors (about \( p'_2 \)) as \( u' = p'_0 - p'_2 \) and \( v' = p'_1 - p'_2 \)
- Then \( p' = \alpha'_0 u' + \alpha'_1 v' + p'_2 = \left( \frac{u'_1}{u'_2} \right) \left( \frac{v'_1}{v'_2} \right) \left( \alpha'_0 \right) + \left( \frac{x'_2}{y'_2} \right) \)
- The unknown point \( p = \alpha_0 p_0 + \alpha_1 p_1 + (1 - \alpha_0 - \alpha_1)p_2 = \alpha_0(p_0 - p_2) + \alpha_1(p_1 - p_2) + p_2 \) that projects to \( p' \) has unknown barycentric weights that need to be determined (once \( \alpha_0 \) and \( \alpha_1 \) are known, \( p \) is then known)
- The coordinates of \( p \) obey \( x = \alpha_0 (x_0 - x_2) + \alpha_1 (x_1 - x_2) + x_2 \), \( y = \alpha_0 (y_0 - y_2) + \alpha_1 (y_1 - y_2) + y_2 \), and \( z = \alpha_0 (z_0 - z_2) + \alpha_1 (z_1 - z_2) + z_2 \)

\[
\begin{pmatrix}
\frac{hx}{z} \\
\frac{hy}{z}
\end{pmatrix} = \left( \begin{array}{c}
\frac{\alpha_0(x_0-x_2)+\alpha_1(x_1-x_2)+x_2}{\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)+z_2} \\
\frac{\alpha_0(y_0-y_2)+\alpha_1(y_1-y_2)+y_2}{\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)+z_2}
\end{array} \right) = \left( \begin{array}{c}
\frac{\alpha_0(z_0x_0'-z_2x'_2)+\alpha_1(z_1x'_1'-z_2x'_2)+z_2x'_2}{\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)+z_2} \\
\frac{\alpha_0(z_0y_0'-z_2y'_2)+\alpha_1(z_1y'_1'-z_2y'_2)+z_2y'_2}{\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)+z_2}
\end{array} \right)
\]

- Or \( p' = \frac{1}{\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)+z_2} \left[ \begin{pmatrix}
z_0x'_0 - z_2x'_2 \\
z_0y'_0 - z_2y'_2
\end{pmatrix} \right] \left( \begin{array}{c}
\alpha_0 \\
\alpha_1
\end{array} \right) + \left( \begin{array}{c}
z_2x'_2 \\
z_2y'_2
\end{array} \right) \)
Screen Space Barycentric Weights

- These two definitions of $p'$ can be equated to obtain:
  \[
  \frac{1}{\alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2} \left[ \left( z_0 x'_1 - z_2 x'_2 \right) \left( z_0 y'_1 - z_2 y'_2 \right) \right] \left( \alpha_0 \right) + \left( z_2 x'_2 \right) \left( \alpha_1 \right) = \left( u'_1 \right) + \left( v'_1 \right) \left( \alpha'_0 \right) + \left( x'_2 \right) \left( y'_2 \right)
  \]
  - Bringing $\left( x'_2 \right) \left( y'_2 \right)$ to the left hand side, and under the brackets as $-\left( \alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2 \right)$ leads to:
    \[
    \frac{\left( z_2 x'_2 - z_0 x'_2 \right) \left( z_2 y'_2 - z_0 y'_2 \right)}{\alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2} \left( z_0 x'_1 - z_1 x'_2 \right) \left( \alpha_0 \right) = \left( u'_1 \right) + \left( v'_1 \right) \left( \alpha'_0 \right)
    \]
  - Importantly, all the terms related to $x$ and $y$ coordinates vanished, leaving dependence only on the $z$ coordinates.
Screen Space Barycentric Weights

• Starting from

\[
\frac{1}{\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)+z_2} (z_0\alpha_0) = \left(\frac{\alpha_0'}{\alpha_1'}\right) or \left(\frac{z_0\alpha_0}{z_1\alpha_1}\right) = (\alpha_0(z_0-z_2) + \alpha_1(z_1 - z_2) + z_2) \left(\frac{\alpha_0'}{\alpha_1'}\right)
\]

• Rewrite to

\[
\left(\frac{z_0 + (z_2 - z_0)\alpha_0'}{(z_2 - z_0)\alpha_1'}\right) \left(\frac{z_2 - z_1)\alpha_0'}{z_1 + (z_2 - z_1)\alpha_1'}\right) = z_2 \left(\frac{\alpha_0'}{\alpha_1'}\right)
\]

• The determinant of this 2x2 matrix is

\[
z_0z_1 + z_1(z_2 - z_0)\alpha_0' + z_0(z_2 - z_1)\alpha_1'
\]

• Thus the inverse is

\[
\frac{1}{z_0z_1 + z_1(z_2 - z_0)\alpha_0' + z_0(z_2 - z_1)\alpha_1'} \left(\frac{z_1 + (z_2 - z_1)\alpha_1'}{(z_0 - z_2)\alpha_1'}\right) = \left(\frac{z_2}{z_0}\right) \left(\frac{z_0\alpha_0'}{z_0\alpha_1'}\right)
\]

• Note that

\[
\left(\frac{z_1 + (z_2 - z_1)\alpha_1'}{(z_0 - z_2)\alpha_1'}\right) = \left(\frac{z_0\alpha_0'}{z_0\alpha_1'}\right)
\]

• Thus,

\[
\left(\frac{\alpha_0}{\alpha_1}\right) = \frac{z_2}{z_0z_1 + z_1(z_2 - z_0)\alpha_0' + z_0(z_2 - z_1)\alpha_1'} \left(\frac{z_1\alpha_0'}{z_0\alpha_1'}\right)
\]

• So, given barycentric coordinates of the pixel, \(\alpha_0'\) and \(\alpha_1'\), we can compute:

\[
\alpha_0 = \frac{z_1z_2\alpha_0'}{z_0z_1 + z_1(z_2 - z_0)\alpha_0' + z_0(z_2 - z_1)\alpha_1'} \quad \text{and} \quad \alpha_1 = \frac{z_0z_2\alpha_1'}{z_0z_1 + z_1(z_2 - z_0)\alpha_0' + z_0(z_2 - z_1)\alpha_1'}
\]

• Then \(\alpha_0\) and \(\alpha_1\) can be used to find the corresponding point \(p\) on the world space triangle

• We use \(\alpha_0\) and \(\alpha_1\) to compute \(z'\) for the pixel, not \(\alpha_0'\) and \(\alpha_1'\)
Ray Tracing

- Ray Tracing works very differently than the Scanline Rendering just discussed.
- The ray tracer creates a ray going through the pixel in question, and subsequently intersects that ray with triangles in world space.
- Since the ray tracer intrinsically operates in world space, as opposed to screen space, it need not worry about dealing with screen space barycentric coordinates.
- Operating in world space is a huge advantage for the ray tracer when it comes to image quality, as it can thoroughly look around in world space to figure out what’s going on.

- A scanline renderer operates in screen space and as such has much more limited information.
- On the other hand, the limited capabilities of a scanline renderer make it a fantastic candidate for real time implementation on hardware.

- Only recently have hardware implementations of some aspects of ray tracing become more feasible!
Lighting and Shading

- After identifying that a pixel is inside a triangle, as discussed above, we set its color to the color of the triangle.
- This ignores all the nuances of how light works (and we’ll discuss that more next week).
- If you rendered a sphere based on this simplistic approach, it would look like this: