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:
  • Complex objects are well approximated (piecewise linear convergence) using 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 it’s 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

- Screen Space Projection transforms triangle vertices from 3D 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 \textbf{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 \textbf{exterior} side of the 2D ray have: \((p - p_0) \cdot n > 0\)
- This same concept can be used for planes in 3D

\[ p_0 = (x_0, y_0) \]

\[ p_1 = (x_1, y_1) \]
A 2D point is considered inside a 2D triangle, when it is interior to (to the left of) all 3 rays.

Vertex ordering 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
  • Causes gaps between adjacent (sharing an edge) triangles, 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 both triangles attempt to color the same pixel
  • Inefficient, and can cause disagreements that lead to artifacts
• Instead, points on the shared edge can 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): screen space projection computes \( z' = n + f - \frac{fn}{z} \) that can be used for occlusion/transparency (via the alpha channel)

• Color the pixel based on which triangle gives the smallest \( z' \) value (closest to the camera)

• 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, linearly interpolate between them via:
  \[
y(x) = \frac{y_2 - y_1}{x_2 - x_1} x - \frac{y_2 - y_1}{x_2 - x_1} x_1 + y_1 \quad \text{or} \quad y(x) = \left(1 - \frac{x - x_1}{x_2 - x_1}\right) y_1 + \frac{x - x_1}{x_2 - x_1} y_2
\]
- Alternatively, \(y(t) = (1 - t)y_1 + ty_2\) 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 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 $p(t) = (1 - t)p_0 + tp_1$
- $t = \frac{\|p - p_0\|_2}{\|p_1 - p_0\|_2}$, 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

- Extend to triangles with 3 vertices by computing 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:

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

- Note the triangle area formula: $\text{Area}(p_0,p_1,p_2) = \frac{1}{2} \| p_0 p_1 \times p_0 p_2 \|_2$
(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 appropriate edge)

• Otherwise, invert the 2x2 coefficient matrix to solve the system of 2 equations with 2 unknowns (for $\alpha_0$ and $\alpha_1$, and set $\alpha_2 = 1 - \alpha_0 - \alpha_1$)
Triangle Basis Vectors

- Compute edge vectors $\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$ and $\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$
- Any point $\mathbf{p}$ interior to the triangle can be written as $\mathbf{p} = \mathbf{p}_0 + \beta_1 \mathbf{u} + \beta_2 \mathbf{v}$ with $\beta_1, \beta_2 \in [0,1]$ and $\beta_1 + \beta_2 \leq 1$
- Substitutions and collecting terms gives $\mathbf{p} = (1 - \beta_1 - \beta_2)\mathbf{p}_0 + \beta_1 \mathbf{p}_1 + \beta_2 \mathbf{p}_2$ implying the equivalence: $\alpha_0 = 1 - \beta_1 - \beta_2$, $\alpha_1 = \beta_1$, $\alpha_2 = \beta_2$
Perspective Projection

- Project a world space triangle (vertices $p_0$, $p_1$, $p_2$) into screen space, vertex by vertex, to obtain $p'_0$, $p'_1$, $p'_2$ via $x' = \frac{hx}{z}$ and $y' = \frac{hy}{z}$ for each vertex $(x, y, z)$
- A point $p = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2$ on the world space triangle is projected into screen space to a corresponding 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

- Need a way of computing $z'$ 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 (so can’t use barycentric interpolation!)
- However, 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 $z$ and $z'$ for the pixel
Screen Space Barycentric Weights

• Given a pixel at $p'$, find valid 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 $p' = \alpha'_0 u' + \alpha'_1 v' + p'_2 = \begin{pmatrix} u'_1 \\ v'_1 \end{pmatrix} \begin{pmatrix} \alpha'_0 \\ \alpha'_1 \end{pmatrix} + \begin{pmatrix} x'_2 \\ y'_2 \end{pmatrix}$

• 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 $z = \alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2$

• Thus, $p' = \begin{pmatrix} \frac{hx}{z} \\ \frac{hy}{z} \end{pmatrix} = \begin{pmatrix} \alpha_0(x_0 - x_2) + \alpha_1(x_1 - x_2) + x_2 \\ \alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2 \end{pmatrix} \begin{pmatrix} \alpha'_0 (x'_0 - z_2 x'_2) + \alpha'_1 (z_1 x'_1 - z_2 x'_2) + z_2 x'_2 \\ \alpha'_0 (z_0 y'_0 - z_2 y'_2) + \alpha'_1 (z_1 y'_1 - z_2 y'_2) + z_2 y'_2 \end{pmatrix}$

• Or $p' = \frac{1}{\alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2} \left[ \begin{pmatrix} (z_0 x'_0 - z_2 x'_2) \\ (z_0 y'_0 - z_2 y'_2) \end{pmatrix} (\alpha'_0) + \begin{pmatrix} z_1 x'_1 - z_2 x'_2 \\ z_1 y'_1 - z_2 y'_2 \end{pmatrix} (\alpha'_1) \right]$
Screen Space Barycentric Weights

- These two definitions of $p'$ can be equated to obtain:

$\frac{1}{\alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2} \begin{bmatrix} (z_0 x_0' - z_2 x_2') & (z_1 x_1' - z_2 x_2') \\ (z_0 y_0' - z_2 y_2') & (z_1 y_1' - z_2 y_2') \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} + \begin{bmatrix} z_2 x_2' \\ z_2 y_2' \end{bmatrix} = \begin{bmatrix} u_1' & v_1' \end{bmatrix} \begin{bmatrix} \alpha_0' \\ \alpha_1' \end{bmatrix} + \begin{bmatrix} x_2' \\ y_2' \end{bmatrix}$

- Bringing $\left( \begin{array}{c} x'_2 \\ y'_2 \end{array} \right)$ to the left hand side, and under the brackets as $-(\alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2) \left( \begin{array}{c} x'_2 \\ y'_2 \end{array} \right)$ or equivalently $\left( \begin{array}{c} z_2 x'_2 - z_0 x'_2 \\ z_2 y'_2 - z_0 y'_2 \end{array} \right) \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} - \left( \begin{array}{c} z_2 x'_2 \\ z_2 y'_2 \end{array} \right)$ leads to:

$\frac{1}{\alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2} \begin{bmatrix} (z_0 x_0' - z_0 x'_2) & (z_1 x_1' - z_1 x'_2) \\ (z_0 y_0' - z_0 y'_2) & (z_1 y_1' - z_1 y'_2) \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} = \begin{bmatrix} u'_1 & v'_1 \end{bmatrix} \begin{bmatrix} \alpha_0' \\ \alpha_1' \end{bmatrix}$

$\frac{1}{\alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2} \begin{bmatrix} u'_1 & v'_1 \end{bmatrix} \begin{bmatrix} z_0 \alpha_0 \\ z_1 \alpha_1 \end{bmatrix} = \begin{bmatrix} \alpha_0' \\ \alpha_1' \end{bmatrix}$

- Importantly, all the terms related to $x$ and $y$ coordinates vanished, leaving dependence only on the $z$ coordinates
Screen Space Barycentric Weights

- Starting from $\frac{1}{\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)+z_2} (z_0\alpha_0) = (\alpha'_0, \alpha'_1)$ or $(z_0\alpha_0) = (\alpha_0(z_0-z_2) + \alpha_1(z_1-z_2) + z_2)(\alpha'_0, \alpha'_1)$

- Rewrite to
  $$(z_0 + (z_2-z_0)\alpha'_0 \quad (z_2-z_1)\alpha'_0 \quad (z_2-z_1)\alpha'_0) (\alpha_0) = z_2 (\alpha'_0, \alpha'_1)$$

- 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 $\frac{1}{z_0z_1 + z_1(z_2-z_0)\alpha'_0 + z_0(z_2-z_1)\alpha'_1} (z_1 + (z_2-z_1)\alpha'_0 \quad (z_1-z_2)\alpha'_0 \quad z_0 + (z_2-z_0)\alpha'_0)$

- Note that
  $$(z_1 + (z_2-z_1)\alpha'_0 \quad (z_1-z_2)\alpha'_0 \quad (z_1-z_2)\alpha'_0) (\alpha'_0) = (z_1\alpha'_0, \alpha'_1)$$

- Thus, $\left(\frac{\alpha'_0}{\alpha_0}\right) = \frac{z_2}{z_0z_1 + z_1(z_2-z_0)\alpha'_0 + z_0(z_2-z_1)\alpha'_1} (z_1\alpha'_0, \alpha'_1)$

- 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 $\alpha_1$ (and $\alpha_2$) can be used to find the (unknown) corresponding point $p$ on the world space triangle

- We use $\alpha_0$ and $\alpha_1$ to compute $z$ (as well as $z' = n + f - \frac{fn}{z}$) 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: