Triangles
Lots of Triangles

Stanford Bunny
69,451 triangles

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

• Can focus on specializing/optimizing everything for (just) triangles
• Optimize software and algorithms for just triangles
• Optimize hardware (e.g. GPUs) for just triangles

• Triangles have many inherent benefits:
  • Complex objects are well-approximated using enough triangles (piecewise linear convergence)
  • Easy to break other polygons into triangles
  • Triangles are guaranteed to be planar (unlike quadrilaterals)
  • Transformations (from last lecture) only need be applied to triangle vertices
  • Robust barycentric interpolation can be used to interpolate information stored on vertices to the interior (of the triangle)
  • Etc.
OpenGL

- Blender uses OpenGL for real-time scanline rendering
- 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
- Competitors: DirectX (Microsoft), Metal (Apple), Vulkan (Khronos)

- 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

- Transform the vertices to screen space (with the matrix stack)
- Find all the pixels inside the 2D screen space triangle
- Color those pixels 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:
  \((\min(x_0, x_1, x_2), \min(y_0, y_1, y_2))\) 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: \((p - p_0) \cdot n = 0\)
  - Same way planes are defined 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

\[ p_0 = (x_0, y_0) \]

\[ p_1 = (x_1, y_1) \]
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 (edge-sharing) triangles, when an edge overlaps a pixel
  - Can fix by using \((p - p_0) \cdot n \leq 0\) instead of \((p - p_0) \cdot n < 0\), but both triangles aim to color the same pixel
  - Inefficient, and disagreements can cause artifacts
- Instead, render points on the shared edge (consistently) with one triangle or the other:
  - Note: edge normals point in opposite directions for 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 only both zero for degenerate triangle
Overlapping Triangles

• When one object is in front of another, two triangles can aim to color the same pixel

• Recall: screen space projection computes $z' = n + f - \frac{fn}{z}$ for occlusion/transparency (via the alpha channel)

• Color each pixel using the triangle that has the smallest $z'$ value (at that pixel)

• Need to interpolate $z'$ values from triangle vertices to the pixel locations

• In order to do this, we use *proper* screen space barycentric weight interpolation
Linear Interpolation (for functions)

- Linearly interpolate between \((x_1, y_1)\) and \((x_2, y_2)\) via:
  \[
y(x) = \left(\frac{y_2-y_1}{x_2-x_1}\right)(x-x_1) + y_1 \quad \text{or} \quad y(x) = \left(1 - \frac{x-x_1}{x_2-x_1}\right)y_1 + \left(\frac{x-x_1}{x_2-x_1}\right)y_2
  \]

- 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

- Linearly interpolate between points $p_0$ and $p_1$ via $p(t) = (1 - t)p_0 + tp_1$
- $t = \frac{\|p-p_0\|_2}{\|p_1-p_0\|_2}$ is the fraction of the distance from $p_0$ to $p_1$
- **Barycentric weights** reformulate this as $p = \alpha_0 p_0 + \alpha_1 p_1$ with weights $\alpha_0, \alpha_1 \in [0,1]$ having $\alpha_0 + \alpha_1 = 1$, 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

- Express points on the triangle via $p = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2$ with barycentric weights $\alpha_0, \alpha_1, \alpha_2 \in [0,1]$ having $\alpha_0 + \alpha_1 + \alpha_2 = 1$
- The weights are computed via areas:
  
  $$\alpha_0 = \frac{\text{Area}(p, p_1, p_2)}{\text{Area}(p_0, p_1, p_2)}$$
  $$\alpha_1 = \frac{\text{Area}(p_0, p, p_2)}{\text{Area}(p_0, p_1, p_2)}$$
  $$\alpha_2 = \frac{\text{Area}(p_0, p_1, p)}{\text{Area}(p_0, p_1, p_2)}$$
- Note (for triangles): $\text{Area}(p_0, p_1, p_2) = \frac{1}{2} \| p_0 p_1 \times p_0 p_2 \|_2$
(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; in 3D, use the normal equations to convert $A \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = b$ into a 2x2 system $A^T A \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = A^T b$

- The coefficient matrix is rank 1 when the columns (i.e. edges) are colinear, implying infinite solutions for triangles with zero area (one can still embed $p$ on an appropriate edge)

- 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 \( u = p_0 - p_2 \) and \( v = p_1 - p_2 \)
- Points in the triangle have the form \( p = p_2 + \beta_1 u + \beta_2 v \) with \( \beta_1, \beta_2 \in [0,1] \) and \( \beta_1 + \beta_2 \leq 1 \)
- Substitutions and collecting terms gives \( p = \beta_1 p_0 + \beta_2 p_1 + (1 - \beta_1 - \beta_2) p_2 \) implying the equivalence: \( \alpha_0 = \beta_1, \ \alpha_1 = \beta_2, \ \alpha_2 = 1 - \beta_1 - \beta_2 \)
Perspective Projection

- Projecting triangle vertices $p_0, p_1, p_2$ into screen space gives $p'_0, p'_1, p'_2$
  - where $x'_i = \frac{hx_i}{z_i}$ and $y'_i = \frac{hy_i}{z_i}$ for each vertex’s $(x_i, y_i, z_i)$ values ($i = 0, 1, 2$)
- Given a pixel at a location $p'$, we need to compute the $z$ value of the sub-triangle location that projects to it
- Then, the triangle with the smallest such $z$ value will be used to shade the pixel

- Compute 2D barycentric weights for $p' = \alpha'_0 p'_0 + \alpha'_1 p'_1 + \alpha'_2 p'_2$
- Some point $p$ on the world space triangle projects to the pixel location $p'$
- But $p \neq \alpha'_0 p'_0 + \alpha'_1 p'_1 + \alpha'_2 p'_2$ because the perspective projection is highly nonlinear

- The barycentric weights for the interior of a screen space triangle do not correspondingly describe the interior of its corresponding world space triangle (and vice versa)!
Corresponding Barycentric Weights

- Given a pixel at $p'$, compute its 2D screen space barycentric weights: $\alpha'_0$, $\alpha'_1$, $\alpha'_2$
- Also, compute its 2D triangle basis vectors: $u' = p'_0 - p'_2$ and $v' = p'_1 - p'_2$
- Then $p' = p'_2 + \alpha'_0 u' + \alpha'_1 v' = \begin{pmatrix} x'_2 \\ y'_2 \end{pmatrix} + \begin{pmatrix} u'_1 \\ v'_1 \end{pmatrix} (\alpha'_0)$

- Some point $p = p_2 + \alpha_0 (p_0 - p_2) + \alpha_1 (p_1 - p_2)$ projects to $p'$ (barycentric weights for $p$ are unknown)
- The coordinates of $p$ obey: $x = x_2 + \alpha_0 (x_0 - x_2) + \alpha_1 (x_1 - x_2)$, $y = y_2 + \alpha_0 (y_0 - y_2) + \alpha_1 (y_1 - y_2)$, and $z = z_2 + \alpha_0 (z_0 - z_2) + \alpha_1 (z_1 - z_2)$

- Thus, $p' = \begin{pmatrix} hx \\ z \\ hy \end{pmatrix} = \begin{pmatrix} x_2 + \alpha_0 (x_0 - x_2) + \alpha_1 (x_1 - x_2) \\ z_2 + \alpha_0 (z_0 - z_2) + \alpha_1 (z_1 - z_2) \\ y_2 + \alpha_0 (y_0 - y_2) + \alpha_1 (y_1 - y_2) \end{pmatrix} = \begin{pmatrix} z_2 x'_2 + \alpha_0 (z_0 x'_0 - z_2 x'_2) + \alpha_1 (z_1 x'_1 - z_2 x'_2) \\ z_2 + \alpha_0 (z_0 - z_2) + \alpha_1 (z_1 - z_2) \\ z_2 y'_2 + \alpha_0 (z_0 y'_0 - z_2 y'_2) + \alpha_1 (z_1 y'_1 - z_2 y'_2) \\ z_2 + \alpha_0 (z_0 - z_2) + \alpha_1 (z_1 - z_2) \end{pmatrix}$

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

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

$$\frac{1}{z_2 + \alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2)} 
\left[ \begin{pmatrix} z_2' \\ z_2 y'_2 \end{pmatrix} + \begin{pmatrix} z_0 x'_0 - z_2 x'_2 \\ z_0 y'_0 - z_2 y'_2 \end{pmatrix} \begin{pmatrix} z_1 x'_1 - z_2 x'_2 \\ z_1 y'_1 - z_2 y'_2 \end{pmatrix} \right] \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} x'_2 \\ y'_2 \end{pmatrix} + \begin{pmatrix} u'_2 \\ v'_2 \end{pmatrix} \begin{pmatrix} \alpha'_0 \\ \alpha'_1 \end{pmatrix}$$

- Bring $\begin{pmatrix} x'_2 \\ y'_2 \end{pmatrix}$ to the left-hand side, and under the brackets as $-(z_2 + \alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2)) \begin{pmatrix} x'_2 \\ y'_2 \end{pmatrix}$ or equivalently $\begin{pmatrix} -z_2 x'_2 \\ -z_2 y'_2 \end{pmatrix} + \begin{pmatrix} -z_0 x'_2 + z_2 x'_2 \\ -z_0 y'_2 + z_2 y'_2 \end{pmatrix} \begin{pmatrix} \alpha'_0 \\ \alpha'_1 \end{pmatrix}$ leads to:

$$\frac{1}{z_2 + \alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2)} \begin{pmatrix} z_0 x'_0 - z_2 x'_2 \\ z_0 y'_0 - z_2 y'_2 \end{pmatrix} \begin{pmatrix} z_1 x'_1 - z_2 x'_2 \\ z_1 y'_1 - z_2 y'_2 \end{pmatrix} \begin{pmatrix} \alpha'_0 \\ \alpha'_1 \end{pmatrix} = \begin{pmatrix} u'_2 \\ v'_2 \end{pmatrix} \begin{pmatrix} \alpha'_0 \\ \alpha'_1 \end{pmatrix}$$

- Note: all the terms related to $x$ and $y$ coordinates vanished, leaving dependence only on the $z$ coordinates.
Corresponding Barycentric Weights

- Starting from \( \begin{pmatrix} z_0 \alpha_0' \\ z_1 \alpha_1' \end{pmatrix} = \left( z_2 + \alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) \right) \begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix} \)

- Rewrite to \( \begin{pmatrix} z_0 - (z_0 - z_2)\alpha_0' \\ -(z_0 - z_2)\alpha_1' \end{pmatrix} = \begin{pmatrix} \frac{1}{z_0z_1 - z_1(z_0 - z_2)\alpha_0' - z_0(z_1 - z_2)\alpha_1'} \begin{pmatrix} z_1 - (z_1 - z_2)\alpha_0' \\ z_0 - (z_0 - z_2)\alpha_0' \end{pmatrix} \end{pmatrix} \begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix} \)

- Invert the 2x2 matrix: \( \begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix} = \frac{1}{z_0z_2\alpha_0' + z_0z_2\alpha_1' + z_0z_1\alpha_2'} \begin{pmatrix} z_1z_2\alpha_0' \\ z_0z_2\alpha_1' \end{pmatrix} \)

- Simplify: \( \alpha_0 = \frac{z_1z_2\alpha_0'}{z_1z_2\alpha_0' + z_0z_2\alpha_1' + z_0z_1\alpha_2'} \) and \( \alpha_1 = \frac{z_0z_2\alpha_1'}{z_1z_2\alpha_0' + z_0z_2\alpha_1' + z_0z_1\alpha_2'} \)

- In summary, given barycentric coordinates of the pixel, \( \alpha_0' \) and \( \alpha_1' \), we can compute:

- Then \( \alpha_0 \) and \( \alpha_1 \) (and \( \alpha_2 = \frac{z_0z_1\alpha_2'}{z_1z_2\alpha_0' + z_0z_2\alpha_1' + z_0z_1\alpha_2'} \)) can be used to find the corresponding point \( p \) on the world space triangle
- This also allows us to compute \( z = \alpha_0z_0 + \alpha_1z_1 + \alpha_2z_2 \) at the point \( p \)
Depth Buffer

• Since $z = \alpha_0z_0 + \alpha_1z_1 + \alpha_2z_2 = \frac{z_0z_1z_2}{z_1z_2\alpha_0' + z_0z_2\alpha_1' + z_0z_1\alpha_2'}$, we have $\frac{1}{z} = \alpha_0' \left( \frac{1}{z_0} \right) + \alpha_1' \left( \frac{1}{z_1} \right) + \alpha_2' \left( \frac{1}{z_2} \right)$

• That is, $\frac{1}{z}$ can be interpolated correctly with screen space barycentric weights (even though $z$ cannot be)

• Recall, for each vertex: $z_i' = n + f - \frac{fn}{z_i}$, or $\frac{1}{z} = \frac{n+f-z_i'}{fn}$

• This leads to $\frac{1}{z} = \frac{n+f-(\alpha_0'z_0' + \alpha_1'z_1' + \alpha_2'z_2')}{fn} = \frac{n+f-z'}{fn}$ where $z'$ is barycentrically interpolated

• That is, $z' = n + f - \frac{fn}{z}$ for every point on the triangle (not just the vertices)

• Since $\frac{dz'}{dz} = \frac{fn}{z^2} > 0$, comparing interpolated $z'$ values is as valid as comparing $z$ values
Ray Tracing

- Ray Tracing works very differently than the Scanline Rendering just discussed.
- The ray tracer creates a ray going through a pixel, and subsequently intersects that ray with triangles in world space.
- Since the ray tracer intrinsically operates in world space (not screen space), it never uses screen space barycentric coordinates.
- Operating in world space is a huge advantage for the ray tracer when it comes to image quality, since 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 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, its color can be set to the color of the triangle.
- This ignores all the nuances of how light works (we’ll discuss that later).
- If you rendered a sphere using this simplistic approach, it would look like this: