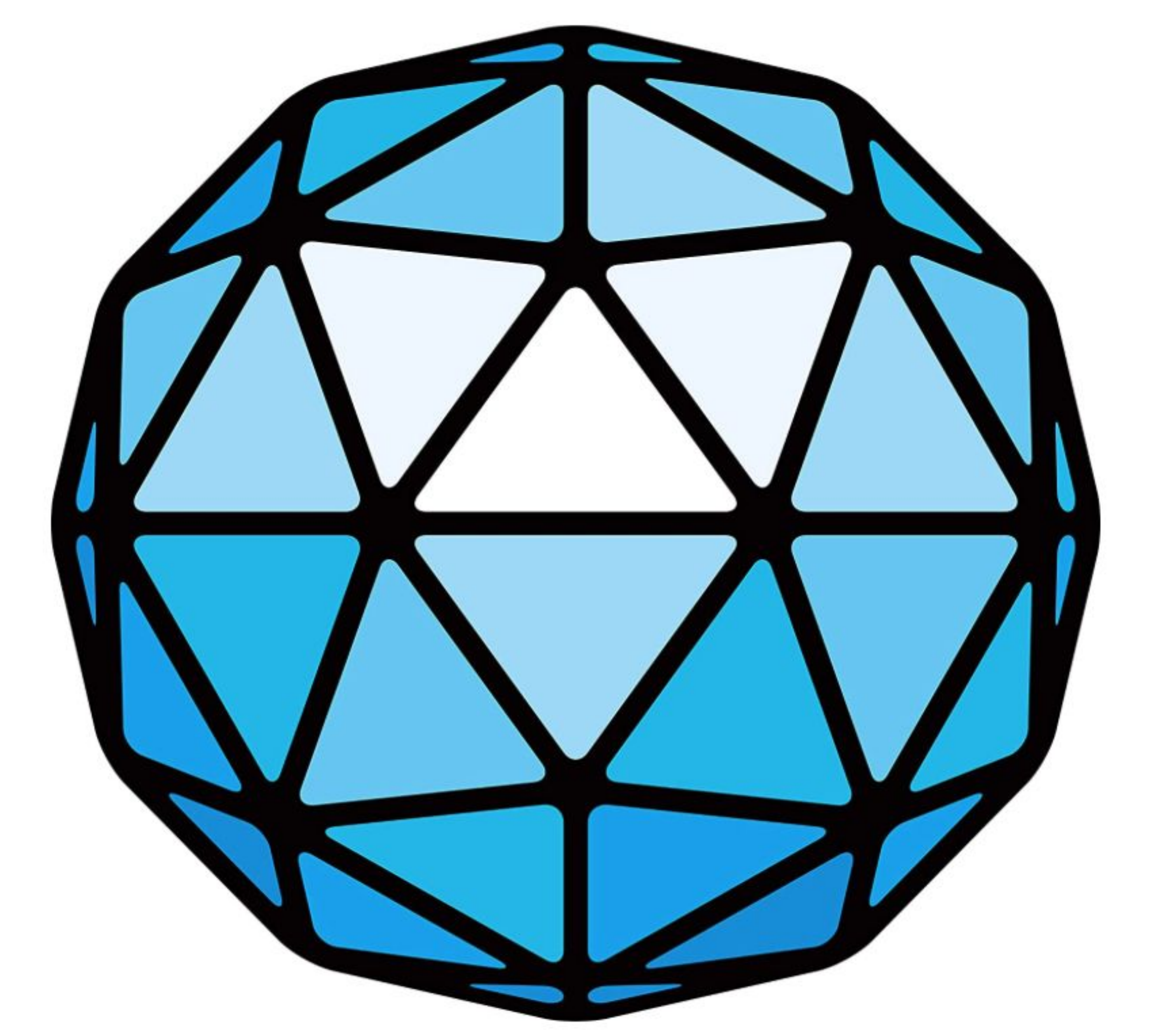
Triangles



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

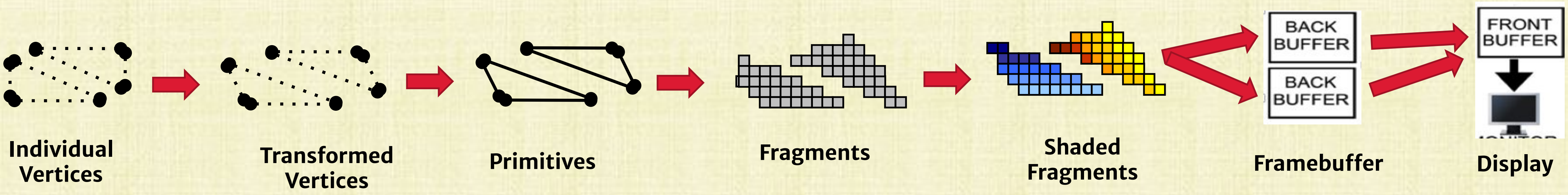
- Triangles have many inherent benefits:
  - Complex objects are well-approximated (piecewise linear convergence) using enough triangles
  - Easy to break other polygons into triangles
  - Triangles are guaranteed to be planar (unlike quadrilaterals)
  - Transformations (from last lecture) only need be applied to 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 <u>scanline renderer</u>

- 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:

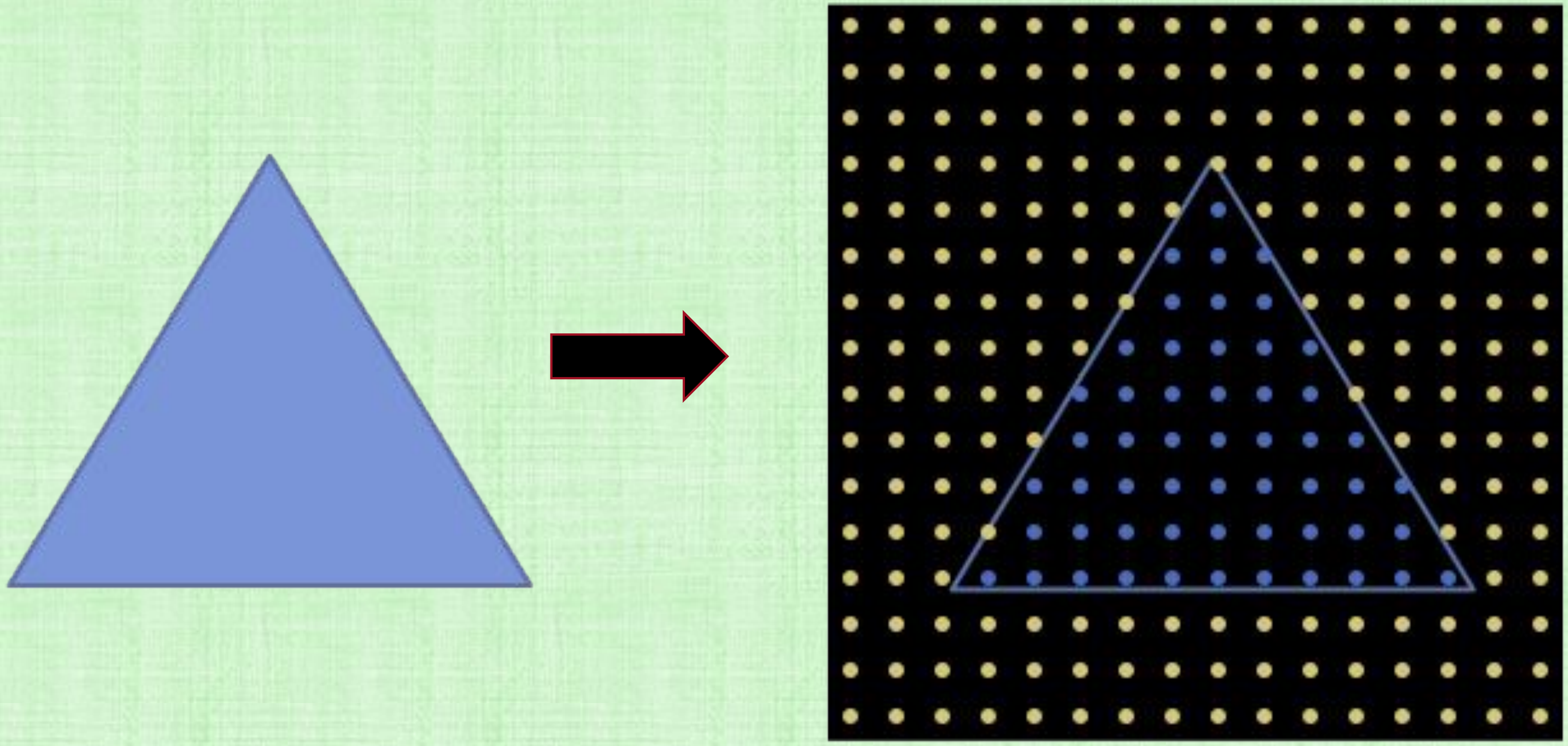


GPUs and Gaming Consoles

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

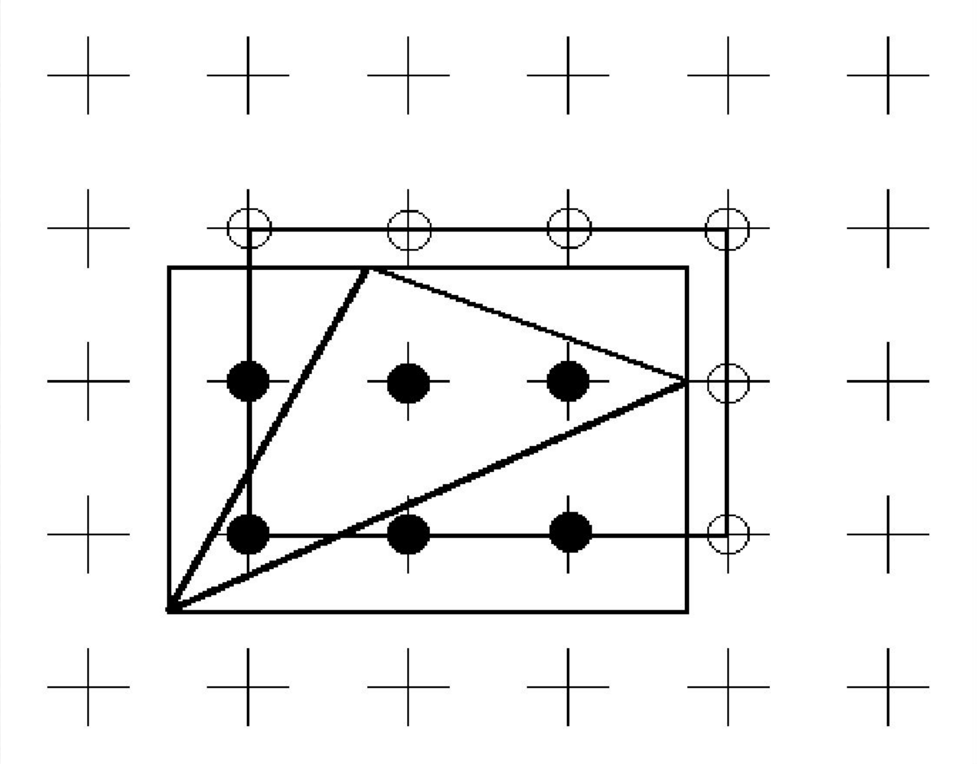
Rasterization

- 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



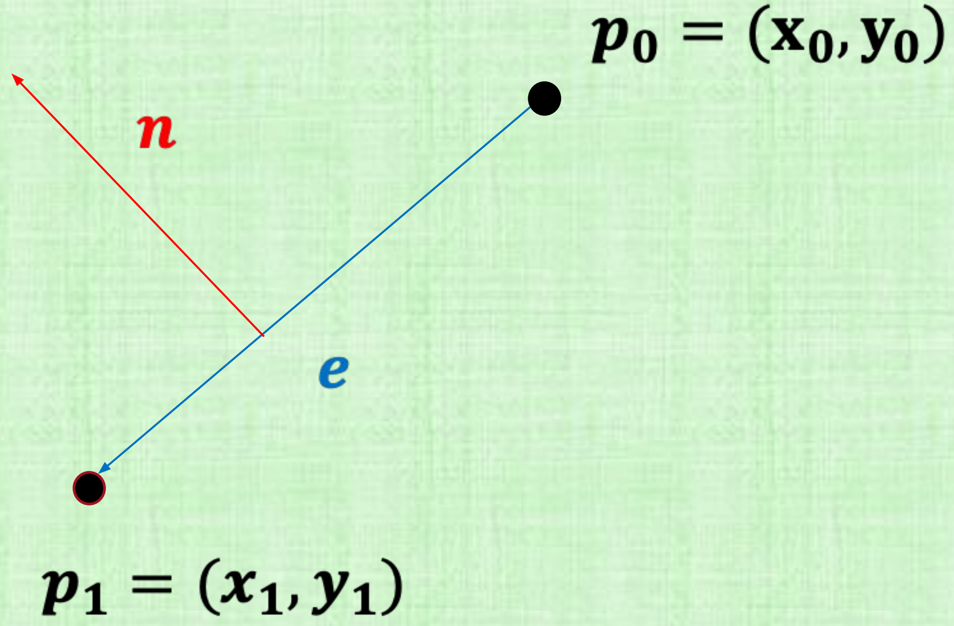
Aside: Bounding Box Acceleration

- Checking every pixel against every triangle is computationally expensive
- Calculate a bounding box around the triangle, with diagonal corners:
  $(\min(x_o, x_1, x_2), \min(y_0, y_1, y_2))$ and $(\max(x_o, x_1, x_2), \max(y_0, y_1, y_2))$
- Then, round coordinates upward to the nearest integer to find all relative pixels



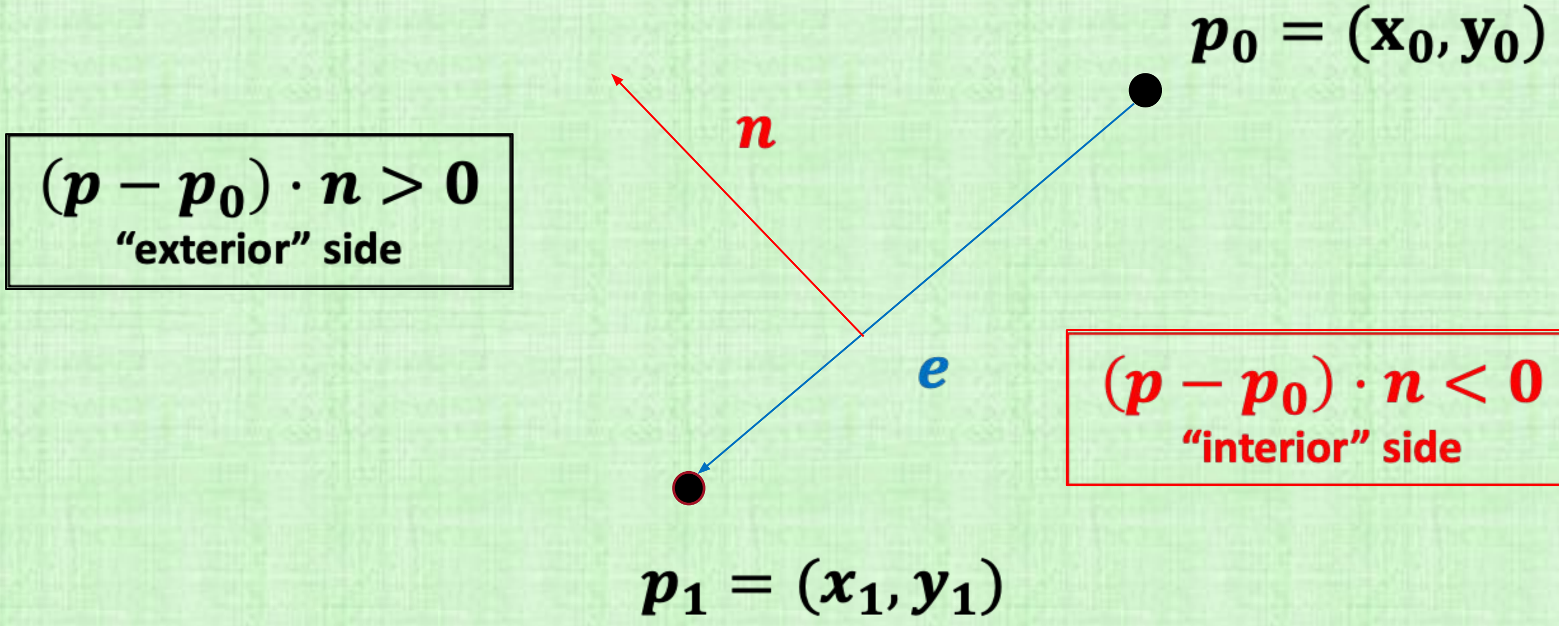
Implicit Equation for a 2D line

- 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

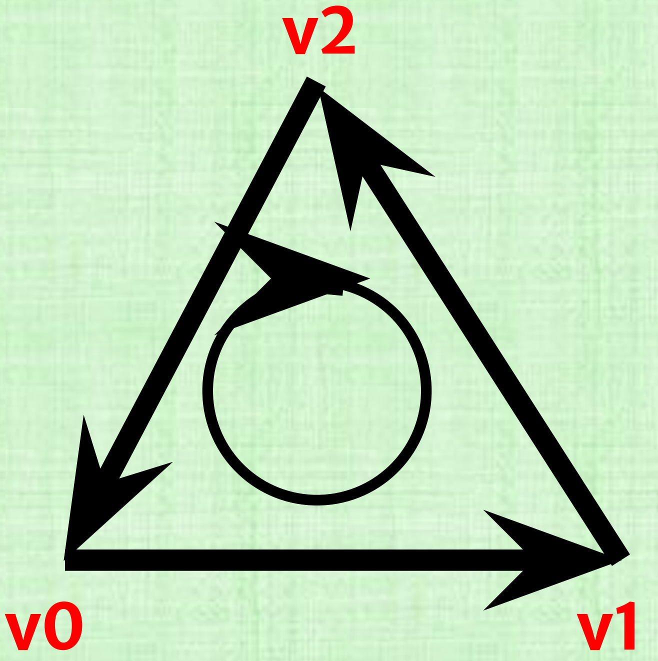


("Leftward") Interior Side of a 2D Ray

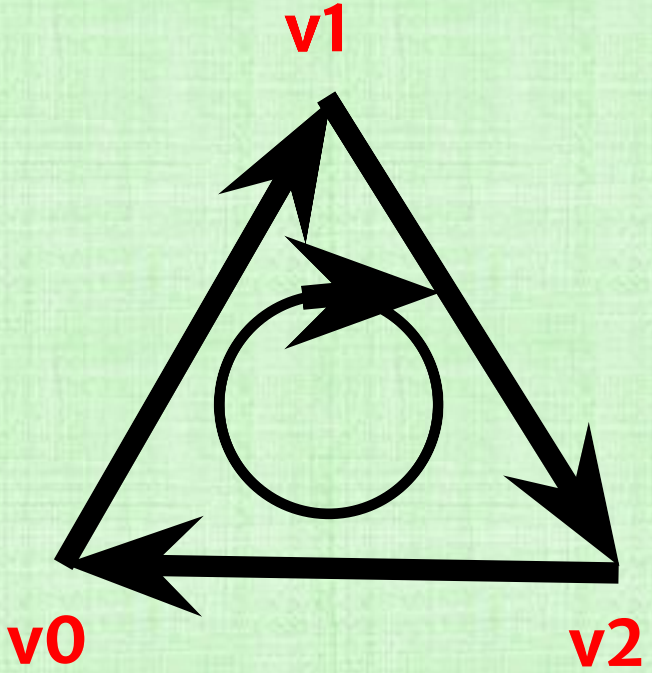
- Points p on the interior side of the 2D ray have: $(p - p_0) \cdot n < 0$
- Points p exactly on the 2D line have: $(p - p_0) \cdot n = 0$
- Points p on the exterior side of the 2D ray have: $(p - p_0) \cdot n > 0$
- This same concept can be used for planes in 3D



2D Point Inside a 2D Triangle



Counter-Clockwise vertex ordering
(facing camera)



Clockwise vertex ordering
(facing away from camera)

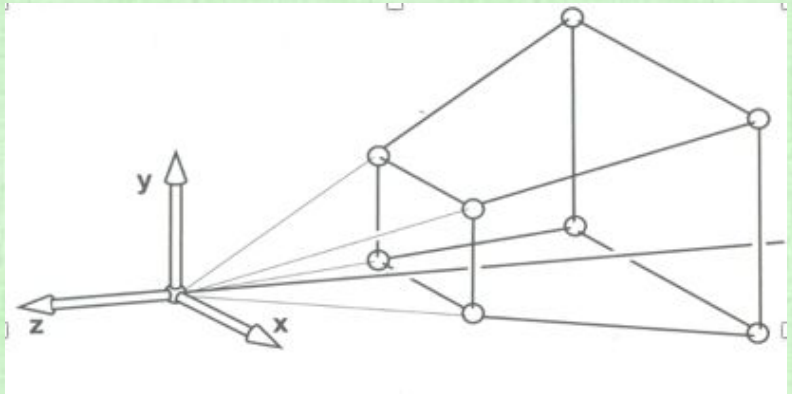
- A 2D point is considered inside a 2D triangle, when it is interior to (to the left of) all 3 rays
- <u>Vertex ordering matters</u>: backward facing triangles are not rendered, since no points are to the left of all three rays

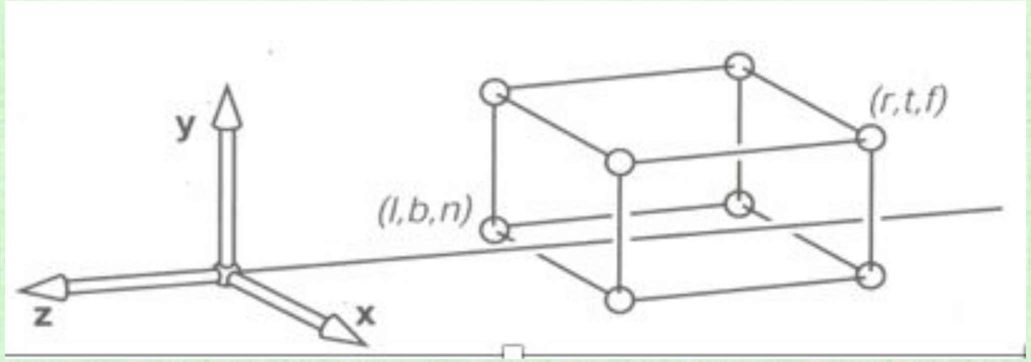
Boundary Cases

- 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 an edge overlaps a pixel
- Changing the inside test to $(p - p_0) \cdot n \leq 0$ instead of $(p - p_0) \cdot n < 0$ fixes this, 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 ($n_x = 0$ and $n_y > 0$), rasterize pixels on that edge
  - When $n_x < 0$ or ($n_x = 0$ and $n_y < 0$), do not rasterize pixels on that edge
  - Note: n_x and n_y are never both zero (unless the triangle is degenerate)

Overlapping Triangles

- If one object is in front of another, two triangles may both aim to color the same pixel
- Recall (last lecture): screen space projection computes $z' = n + f - \frac{fn}{z}$ for use in occlusion/transparency (via the alpha channel)





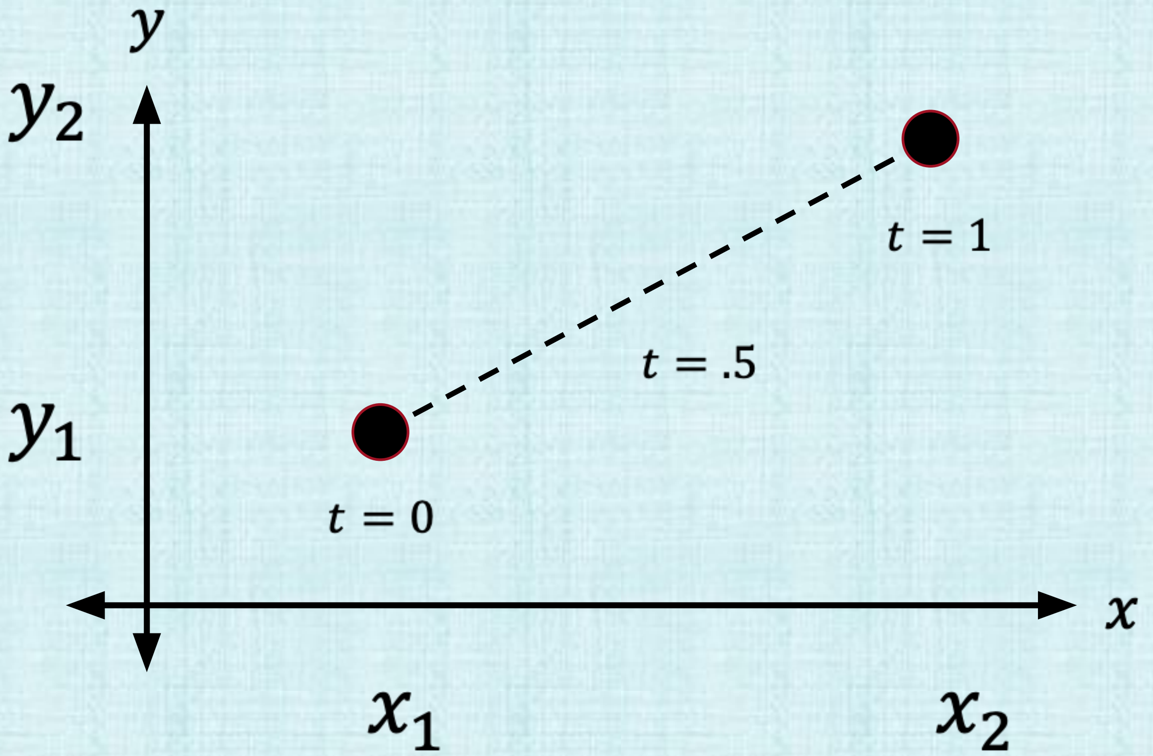
- Color each pixel using the triangle that gives the smallest z' value (for that pixel)
- This requires interpolating z' values from triangle vertices to the pixel locations
- In order to do this, we use *proper* screen space barycentric weight interpolation

Linear Interpolation (for functions)

- Given two points (x_1, y_1) and (x_2, y_2), linearly interpolate between them via:

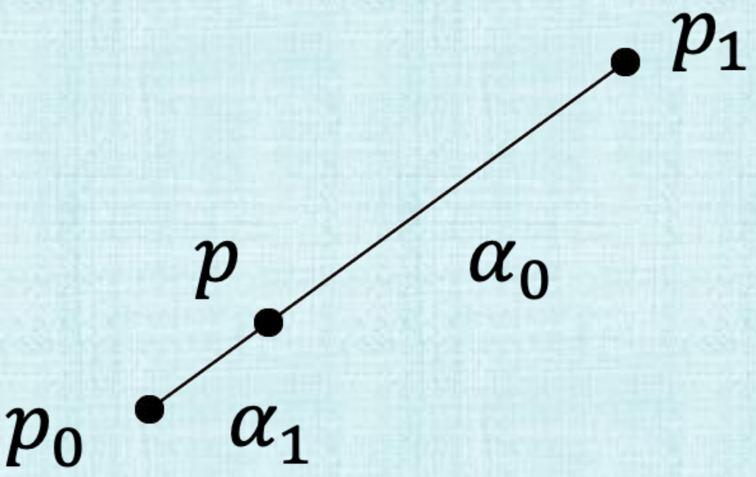
$$y(x) = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x - x_1) + y_1 \quad \text{or} \quad y(x) = \left(1 - \frac{x - x_1}{x_2 - x_1}\right) y_1 + \left(\frac{x - x_1}{x_2 - x_1}\right) y_2$$

- 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

- 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

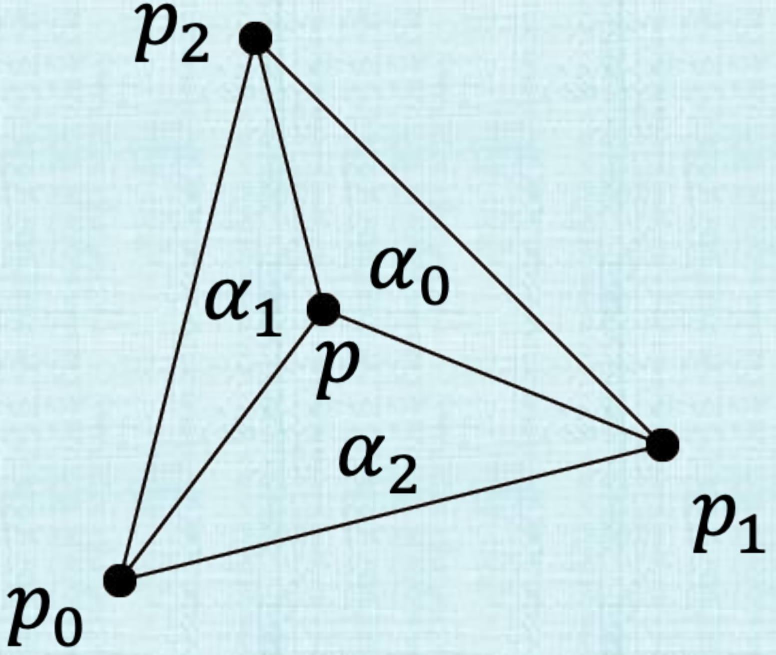


2D/3D Triangles

- Given endpoints p_0, p_1, p_2, compute 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:

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

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

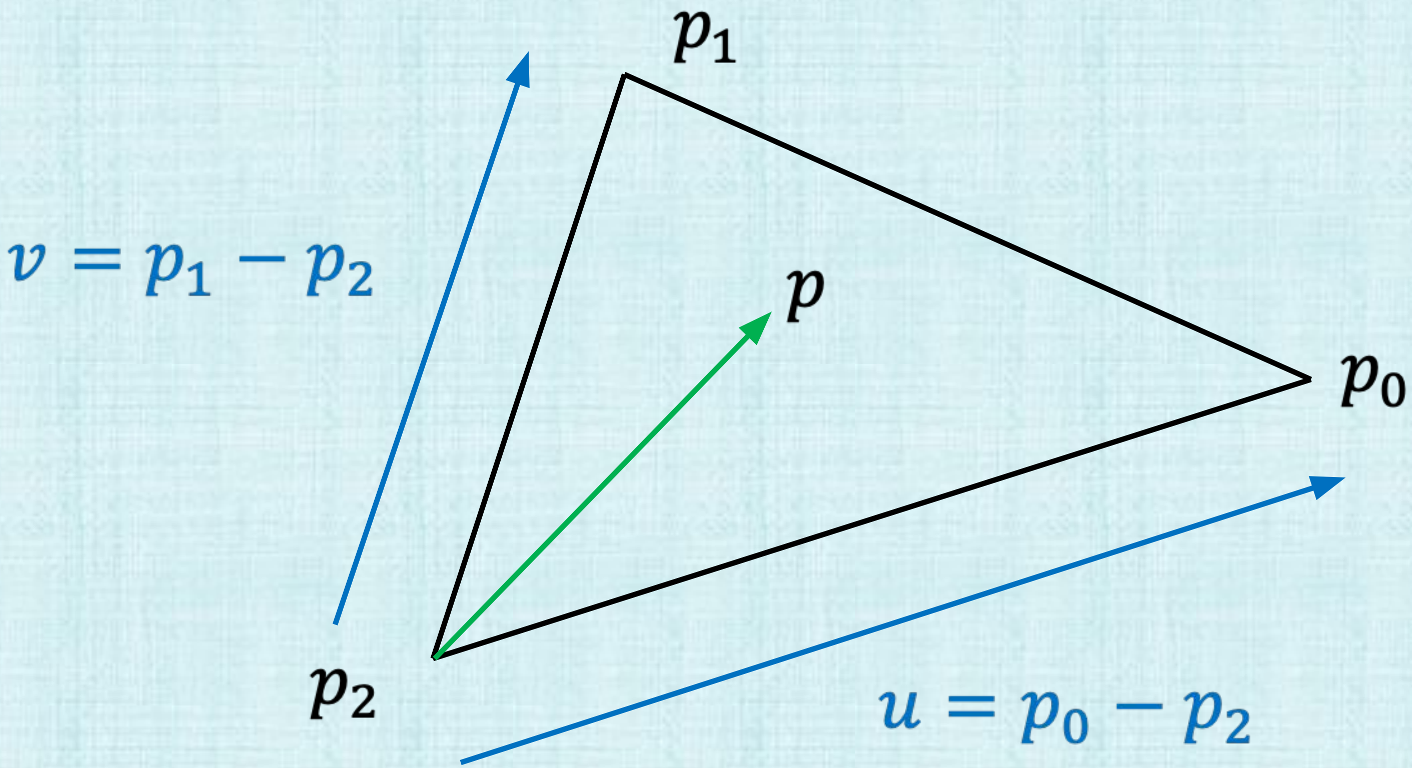


(Alternative) Algebraic Approach

- 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 obtain 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 α_0 and α_1, and set $\alpha_2 = 1 - \alpha_0 - \alpha_1$)

Triangle Basis Vectors

- Compute edge vectors $u = p_0 - p_2$ and $v = p_1 - p_2$
- Any point p interior to the triangle can be written as $p = p_2 + \beta_1 u + \beta_2 v$ 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

- Triangle vertices p_0, p_1, p_2 are projected into screen space (vertex by vertex) to obtain p_0', p_1', p_2' via $x_i' = \frac{hx_i}{z_i}$ 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 want to know the z value of the sub-triangle location that projects to it
- We want to use the triangle with the smallest such z value (when triangles overlap)

- Can compute 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 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 $p' = p_2' + \alpha_0' u' + \alpha_1' v' = \begin{pmatrix} x_2' \\ y_2' \end{pmatrix} + \begin{pmatrix} u_1' & v_1' \\ u_2' & v_2' \end{pmatrix}\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$

- 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, $p' = \begin{pmatrix} \frac{hx}{z} \\ \frac{hy}{z} \end{pmatrix} = \begin{pmatrix} h\frac{x_2+\alpha_0(x_0-x_2)+\alpha_1(x_1-x_2)}{z_2+\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)} \\ h\frac{y_2+\alpha_0(y_0-y_2)+\alpha_1(y_1-y_2)}{z_2+\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)} \end{pmatrix} = \begin{pmatrix} \frac{z_2x_2'+\alpha_0(z_0x_0'-z_2x_2')+\alpha_1(z_1x_1'-z_2x_2')}{z_2+\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)} \\ \frac{z_2y_2'+\alpha_0(z_0y_0'-z_2y_2')+\alpha_1(z_1y_1'-z_2y_2')}{z_2+\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)} \end{pmatrix}$

- Or $p' = \frac{1}{z_2+\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)}\left[\begin{pmatrix} z_2x_2' \\ z_2y_2' \end{pmatrix} + \begin{pmatrix} z_0x_0' - z_2x_2' & z_1x_1' - z_2x_2' \\ z_0y_0' - z_2y_2' & z_1y_1' - z_2y_2' \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}\right]$

Corresponding Barycentric Weights

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

$$\frac{1}{z_2+\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)}\left[\begin{pmatrix} z_2x_2' \\ z_2y_2' \end{pmatrix}+\begin{pmatrix} z_0x_0'-z_2x_2' & z_1x_1'-z_2x_2' \\ z_0y_0'-z_2y_2' & z_1y_1'-z_2y_2' \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}\right]=\begin{pmatrix} x_2' \\ y_2' \end{pmatrix}+\begin{pmatrix} u_1' & v_1' \\ u_2' & v_2' \end{pmatrix}\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$$

- 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 $\begin{pmatrix} -z_2x_2' \\ -z_2y_2' \end{pmatrix}+\begin{pmatrix} -z_0x_2'+z_2x_2' & -z_1x_2'+z_2x_2' \\ -z_0y_2'+z_2y_2' & -z_1y_2'+z_2y_2' \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$ leads to:

$$\frac{1}{z_2+\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)}\begin{pmatrix} z_0x_0'-z_0x_2' & z_1x_1'-z_1x_2' \\ z_0y_0'-z_0y_2' & z_1y_1'-z_1y_2' \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}=\begin{pmatrix} u_1' & v_1' \\ u_2' & v_2' \end{pmatrix}\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$$

$$\frac{1}{z_2+\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)}\begin{pmatrix} u_1' & v_1' \\ u_2' & v_2' \end{pmatrix}\begin{pmatrix} z_0\alpha_0 \\ z_1\alpha_1 \end{pmatrix}=\begin{pmatrix} u_1' & v_1' \\ u_2' & v_2' \end{pmatrix}\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$$

$$\frac{1}{z_2+\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)}\begin{pmatrix} z_0\alpha_0 \\ z_1\alpha_1 \end{pmatrix}=\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$$

- 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 $\begin{pmatrix} z_0 - (z_0 - z_2)\alpha_0' & -(z_1 - z_2)\alpha_0' \\ -(z_0 - z_2)\alpha_1' & z_1 - (z_1 - z_2)\alpha_1' \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} z_2\alpha_0' \\ z_2\alpha_1' \end{pmatrix}$
- Invert the 2x2 matrix: $\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \frac{1}{z_0z_1 - z_1(z_0 - z_2)\alpha_0' - z_0(z_1 - z_2)\alpha_1'}\begin{pmatrix} z_1 - (z_1 - z_2)\alpha_1' & (z_1 - z_2)\alpha_0' \\ (z_0 - z_2)\alpha_1' & z_0 - (z_0 - z_2)\alpha_0' \end{pmatrix}\begin{pmatrix} z_2\alpha_0' \\ z_2\alpha_1' \end{pmatrix}$
- Simplify: $\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \frac{1}{z_1z_2\alpha_0' + z_0z_2\alpha_1' + z_0z_1\alpha_2'}\begin{pmatrix} z_1z_2\alpha_0' \\ z_0z_2\alpha_1' \end{pmatrix}$

- So, given barycentric coordinates of the pixel, α_0' and α_1', we can compute:

$$\alpha_0 = \frac{z_1z_2\alpha_0'}{z_1z_2\alpha_0' + z_0z_2\alpha_1' + z_0z_1\alpha_2'} \quad \text{and} \quad \alpha_1 = \frac{z_0z_2\alpha_1'}{z_1z_2\alpha_0' + z_0z_2\alpha_1' + z_0z_1\alpha_2'}$$

- Then α_0 and α_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
- In particular, we want $z = \alpha_0z_0 + \alpha_1z_1 + \alpha_2z_2$

Summary

- Express the pixel p' terms of its screen space barycentric weights: $\alpha_0', \alpha_1', \alpha_2'$
- Express the point p that projects to p' in terms of unknown world space barycentric weights: $\alpha_o, \alpha_1, \alpha_2$
- Project p into screen space and set the result equal to p'
- Solve for $\alpha_o, \alpha_1, \alpha_2$ to obtain:

$$\alpha_0 = \frac{z_1 z_2 \alpha_0'}{z_1 z_2 \alpha_0' + z_0 z_2 \alpha_1' + z_0 z_1 \alpha_2'}$$

$$\alpha_1 = \frac{z_0 z_2 \alpha_1'}{z_1 z_2 \alpha_0' + z_0 z_2 \alpha_1' + z_0 z_1 \alpha_2'}$$

$$\alpha_2 = \frac{z_0 z_1 \alpha_2'}{z_1 z_2 \alpha_0' + z_0 z_2 \alpha_1' + z_0 z_1 \alpha_2'}$$

Depth Buffer

- Since $z = \alpha_0 z_0 + \alpha_1 z_1 + \alpha_2 z_2 = \frac{z_0 z_1 z_2}{z_1 z_2 \alpha'_0 + z_0 z_2 \alpha'_1 + z_0 z_1 \alpha'_2}$, 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 barycentrically interpolated in screen space

- Recall, for each vertex: $z'_i = n + f - \frac{fn}{z_i}$, or $\frac{1}{z_i} = \frac{n+f-z'_i}{fn}$
- This means that $\frac{1}{z} = \frac{n+f-(\alpha'_0 z'_0 + \alpha'_1 z'_1 + \alpha'_2 z'_2)}{fn}$, and thus $z = \frac{fn}{n+f-(\alpha'_0 z'_0 + \alpha'_1 z'_1 + \alpha'_2 z'_2)} = \frac{fn}{n+f-z'}$
- That is, the interpolated z' and corresponding z value obey the same pointwise equation: $z' = n + f - \frac{fn}{z}$

- BTW: $\frac{dz}{dz'} = \frac{fn}{(n+f-z')^2} > 0$ implies that 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, as opposed to screen space, it can ignore 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 next week)
- If you rendered a sphere using this simplistic approach, it would look like this: