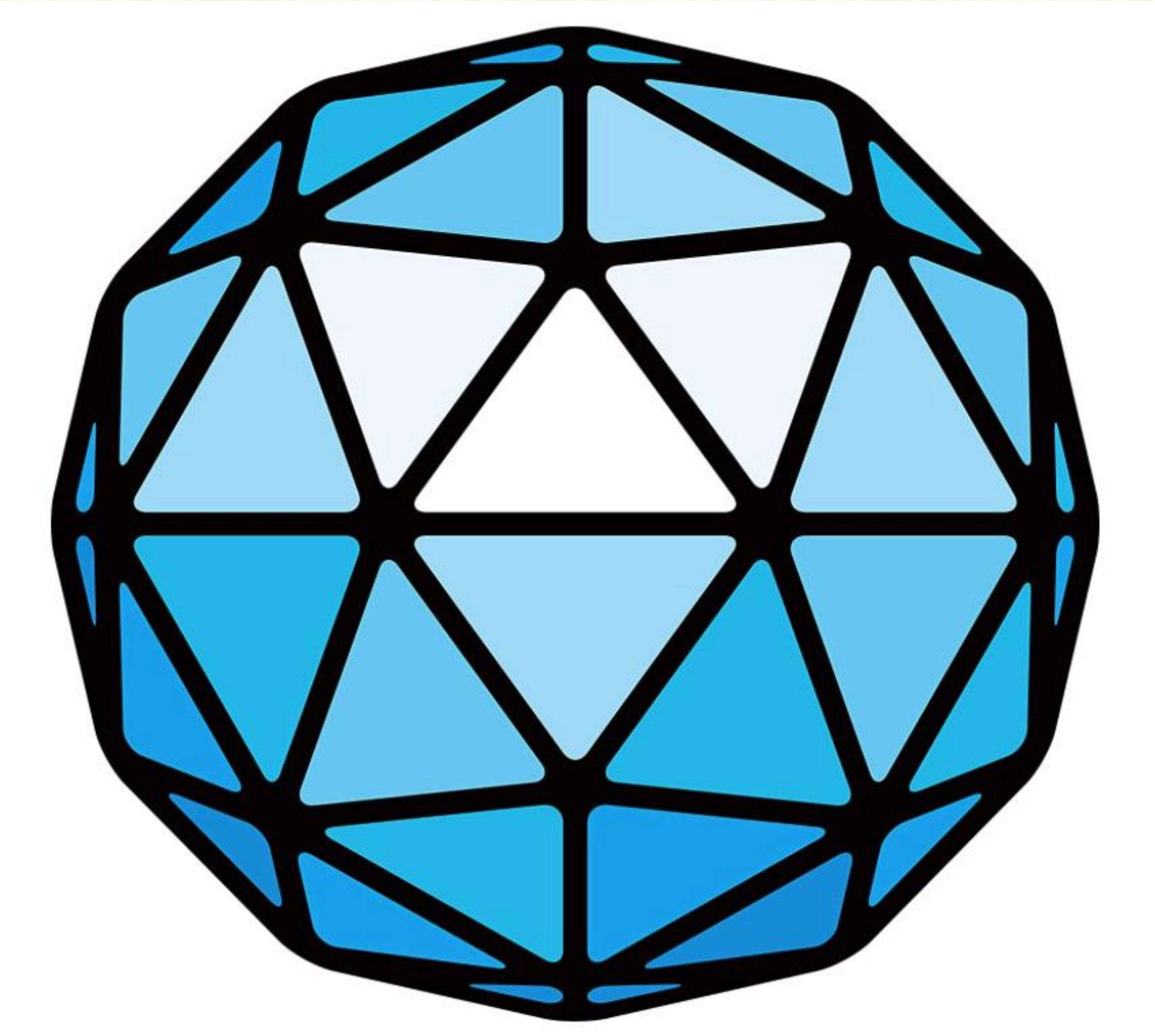
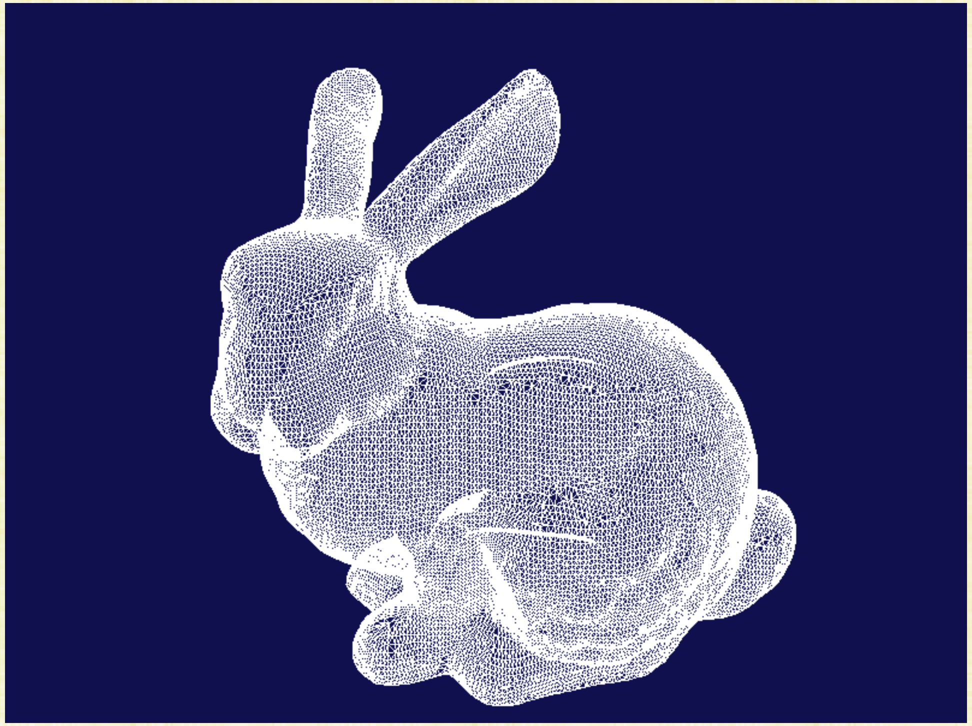
Triangles



Lots of Triangles



Stanford Bunny
69,451 triangles

David (Digital Michelangelo Project)
56,230,343 triangles

Why Triangles?

- Can specialize/optimize for just triangles
  - Optimize software and algorithms for just triangles
  - Optimize hardware (e.g. GPUs) for just triangles

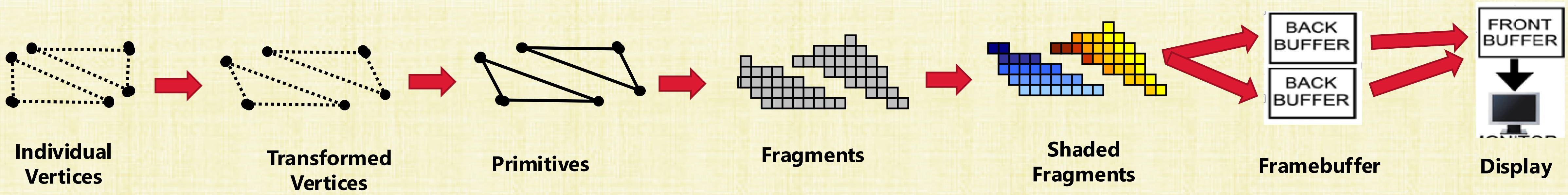
- Triangles have many inherent benefits:
  - Complex objects are well-approximated using enough triangles (piecewise linear convergence)
  - Easy to break other polygons into triangles
  - Triangles are guaranteed to be planar (unlike quadrilaterals)
  - Transformations (from last lecture) only need be applied to triangle vertices
  - 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 <u>scanline rendering</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
- Competitors: DirectX (Microsoft), Metal (Apple), Vulkan (Khronos)

- OpenGL is highly optimized for triangles:

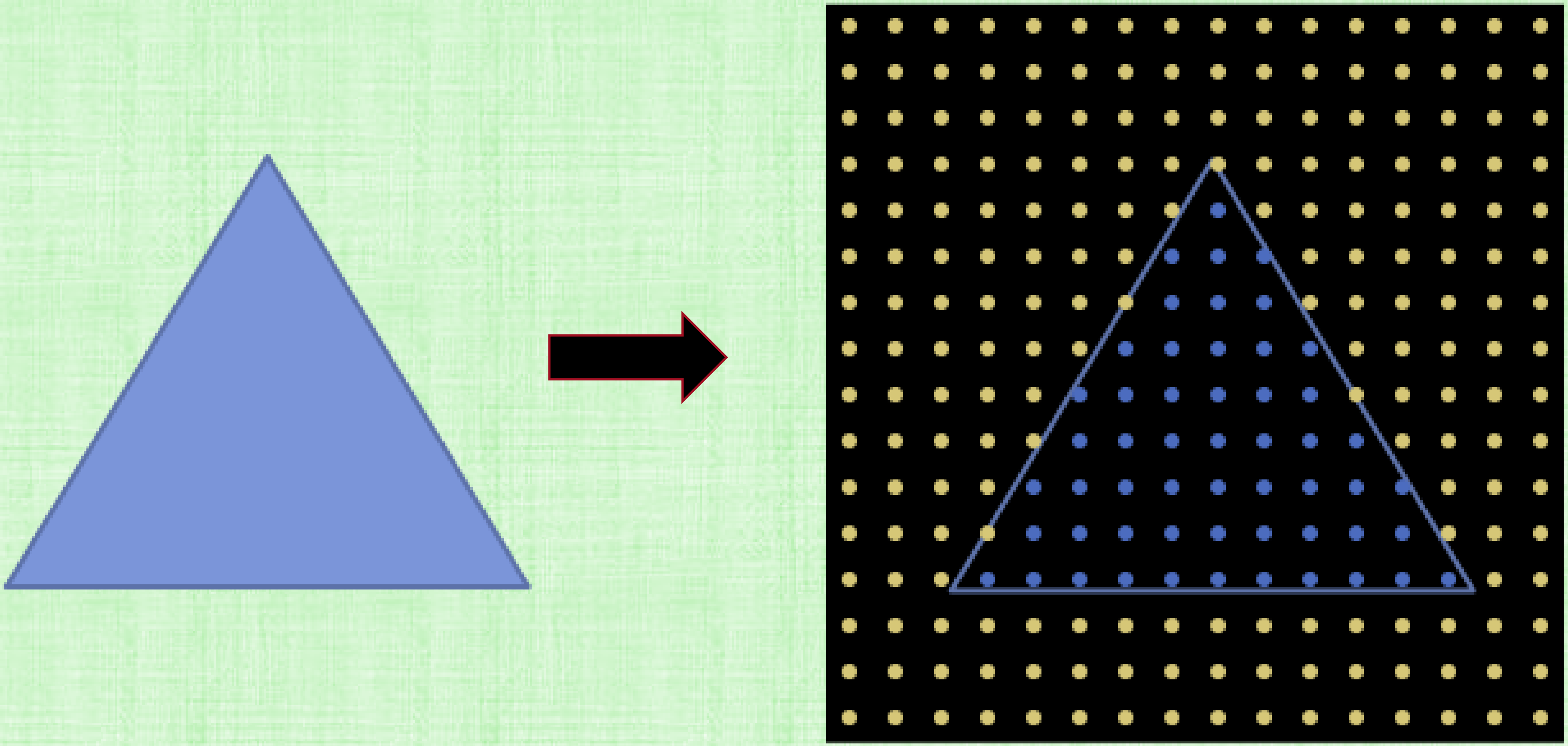


GPUs and Gaming Consoles

- 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

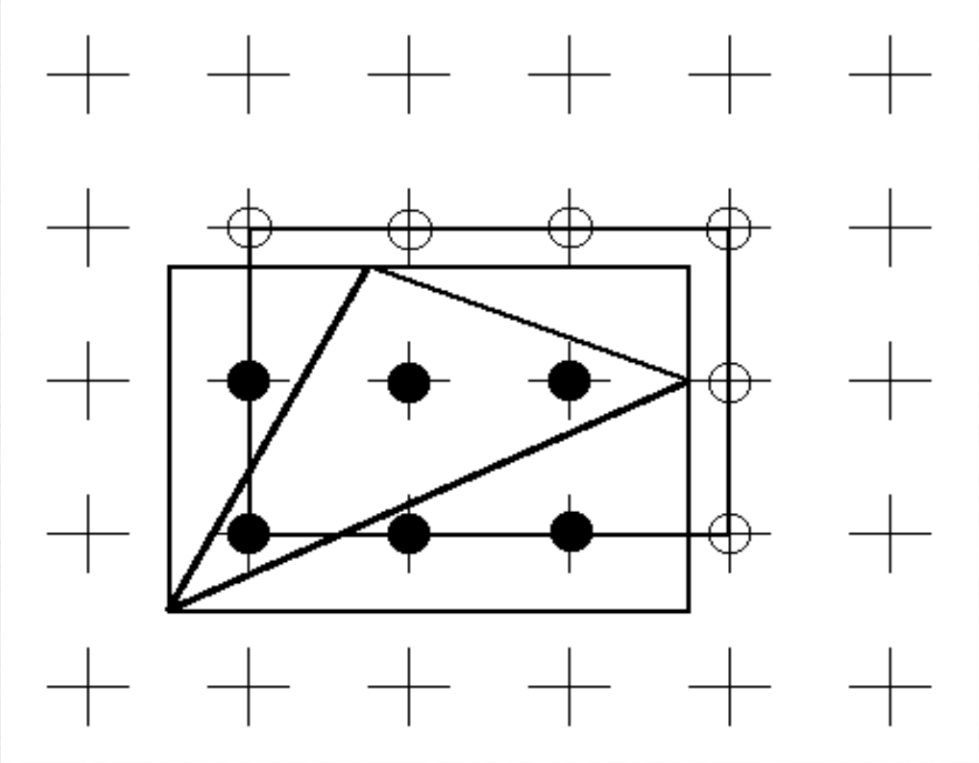


Aside: Bounding Box Acceleration

- Checking every pixel against every triangle is computationally expensive
- Calculate a bounding box around the triangle, with diagonal corners:

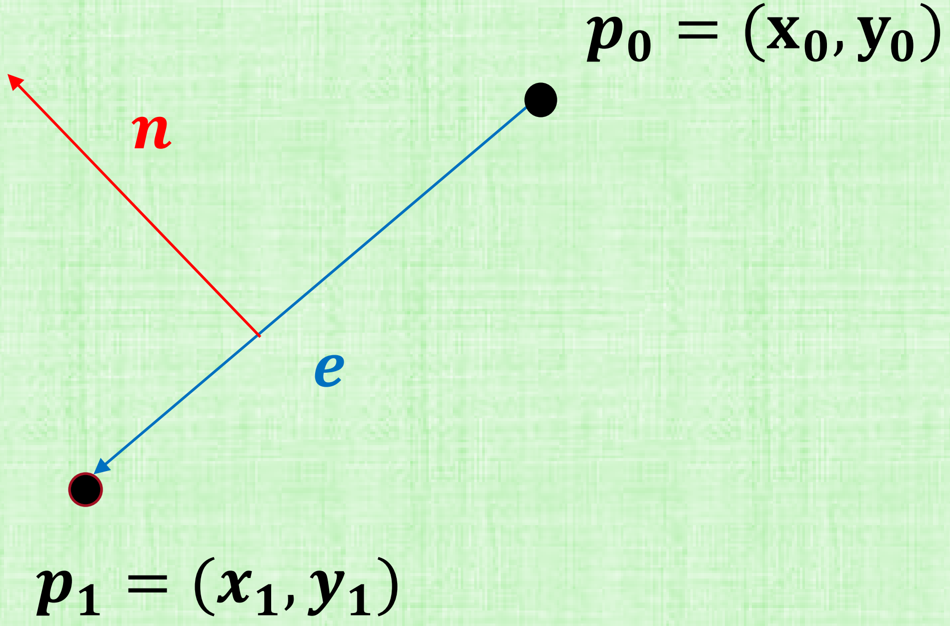
$$(\min(x_o, x_1, x_2), \min(y_0, y_1, y_2)) \text{ and } (\max(x_o, x_1, x_2), \max(y_0, y_1, y_2))$$

- Then, round coordinates upward to the nearest integer to find all relative pixels



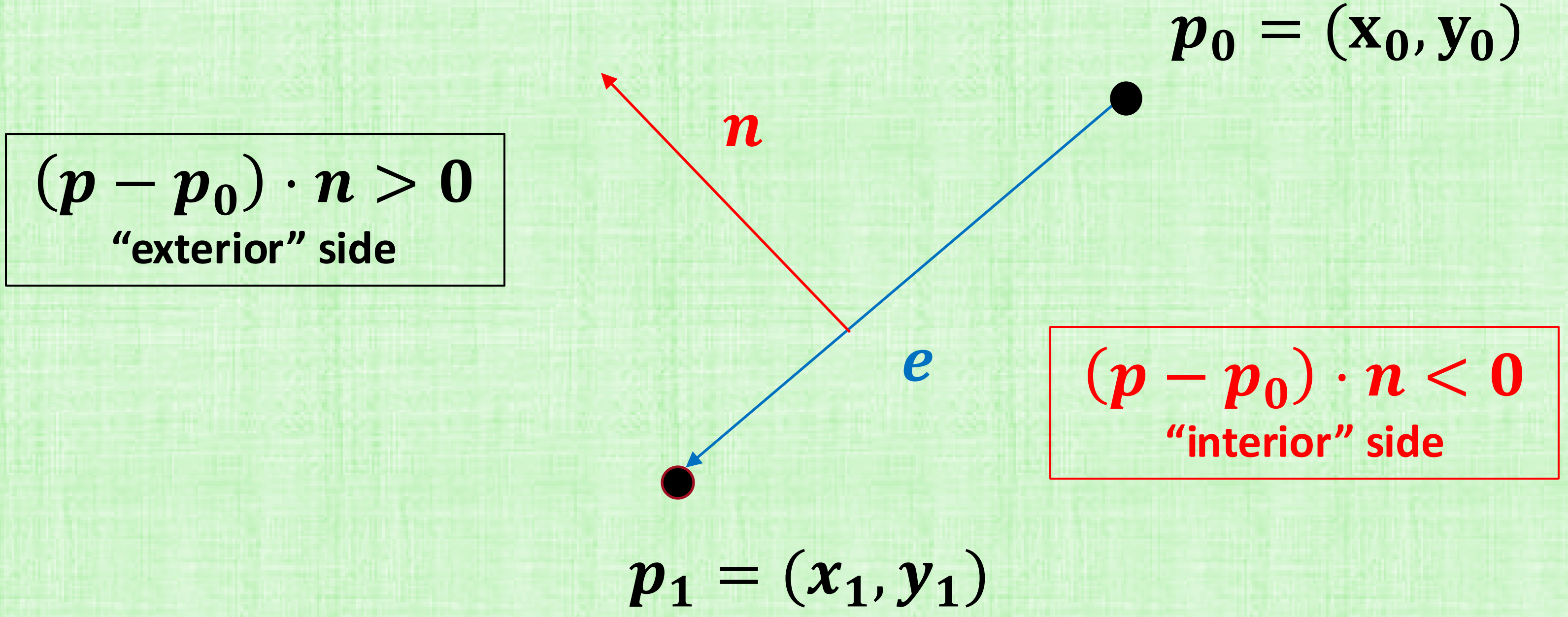
Implicit Equation for a 2D line

- Compute a directed edge vector $e = p_1 - p_0 = (x_1 - x_0, y_1 - y_0)$
- Compute a 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

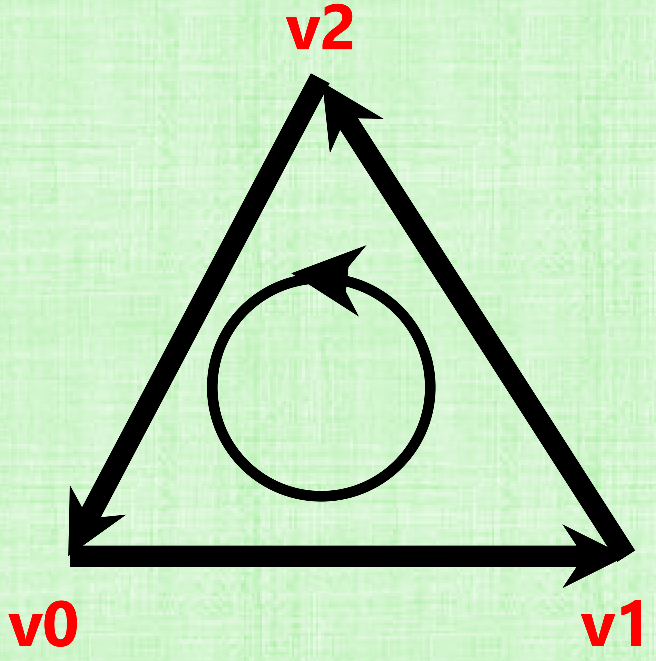


("Leftward") Interior Side of a 2D Ray

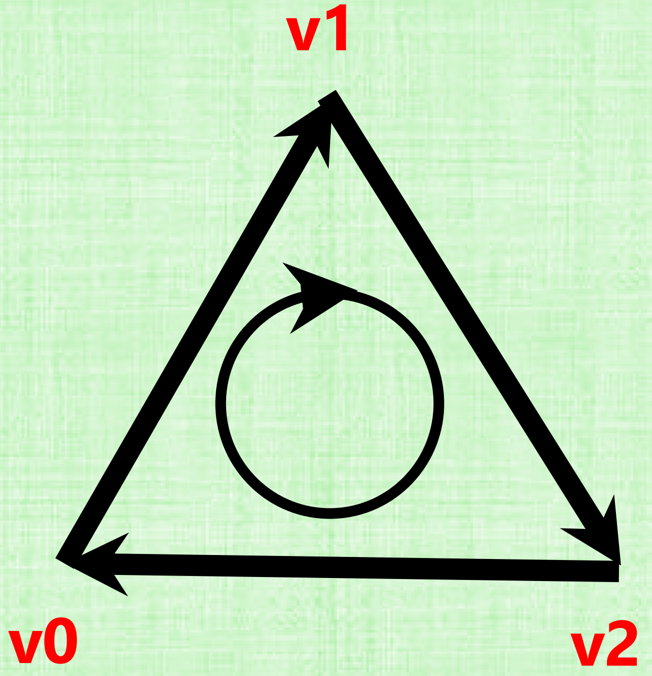
- Points p on the interior side of the 2D ray have: $(p - p_0) \cdot n < 0$
- Points p exactly on the 2D line have: $(p - p_0) \cdot n = 0$
- Points p on the exterior side of the 2D ray have: $(p - p_0) \cdot n > 0$
- This same concept can be used for planes in 3D



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
- Vertex ordering matters: backward facing triangles are not rendered, since no points are to the left of all three rays

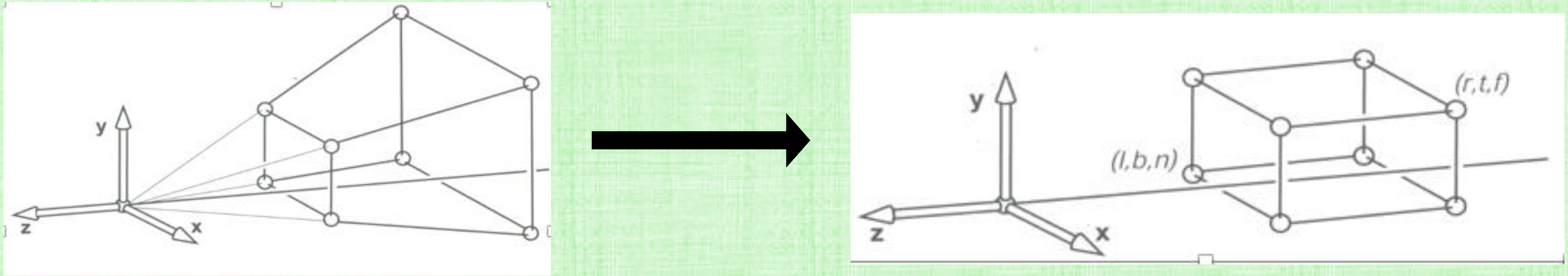
Adjacent Triangles

- Pixels on the boundary with $(p - p_0) \cdot n = 0$ for one of the edges won't be rendered
  - This causes gaps between adjacent edge-sharing triangles, when the edge overlaps a pixel
- Can fix by using $(p - p_0) \cdot n \leq 0$ instead of $(p - p_0) \cdot n < 0$, but then both triangles aim to color the same pixel
  - Inefficient, and disagreements can cause artifacts
- Instead, render points on the shared edge (consistently) using only one of the two triangles:
  - Note: edge normals point in opposite directions for 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 only both zero for a degenerate triangle

Overlapping Triangles

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

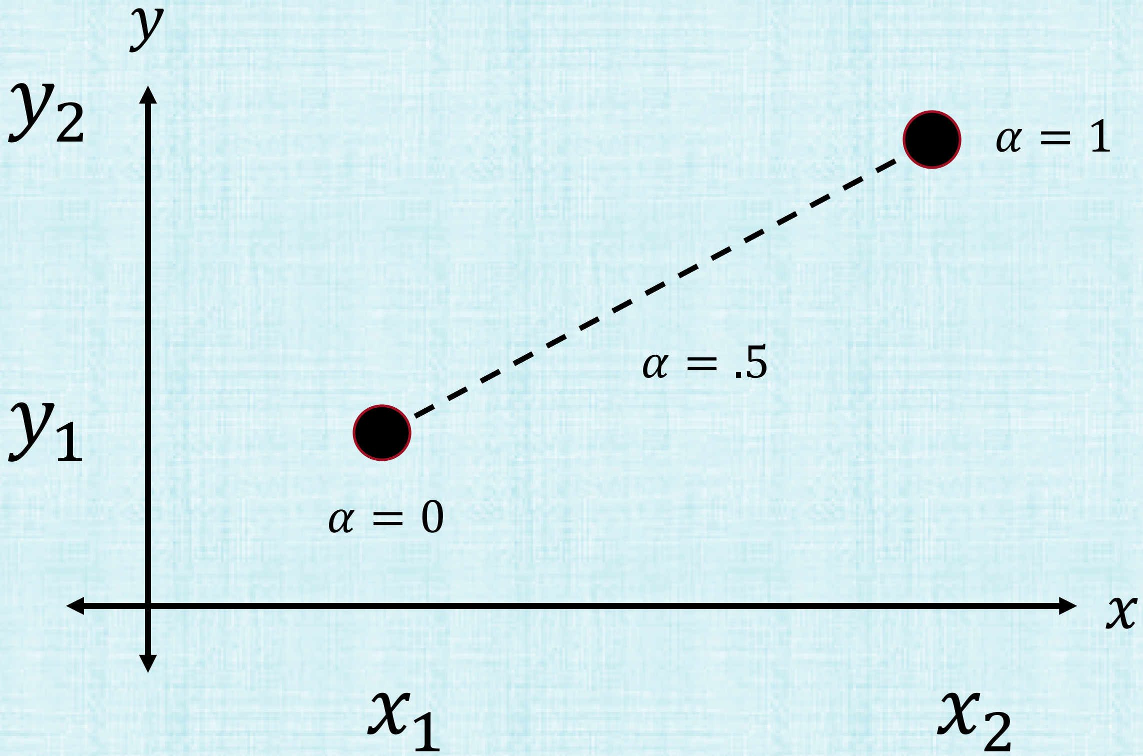
- Recall: screen space projection computes $z' = n + f - \frac{fn}{z}$ for occlusion/transparency



- Color each pixel using the triangle with the smallest z' value at that pixel
  - Need to interpolate z' values from triangle vertices to pixel locations
  - In order to do this, we use <u>screen space barycentric weight interpolation</u>

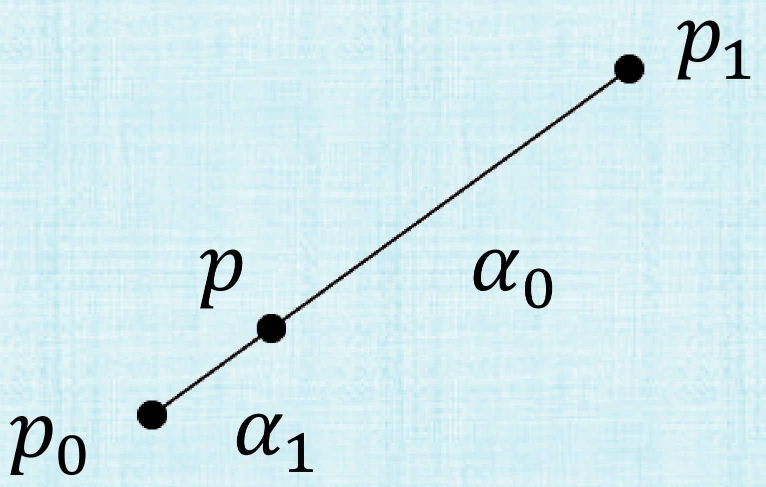
Linear Interpolation for Functions

- 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, $y(\alpha) = (1 - \alpha) y_1 + \alpha y_2$ where $\alpha = \frac{x - x_1}{x_2 - x_1} \in [0,1]$ is the fraction of the way from x_1 to x_2 or equivalently from y_1 to y_2



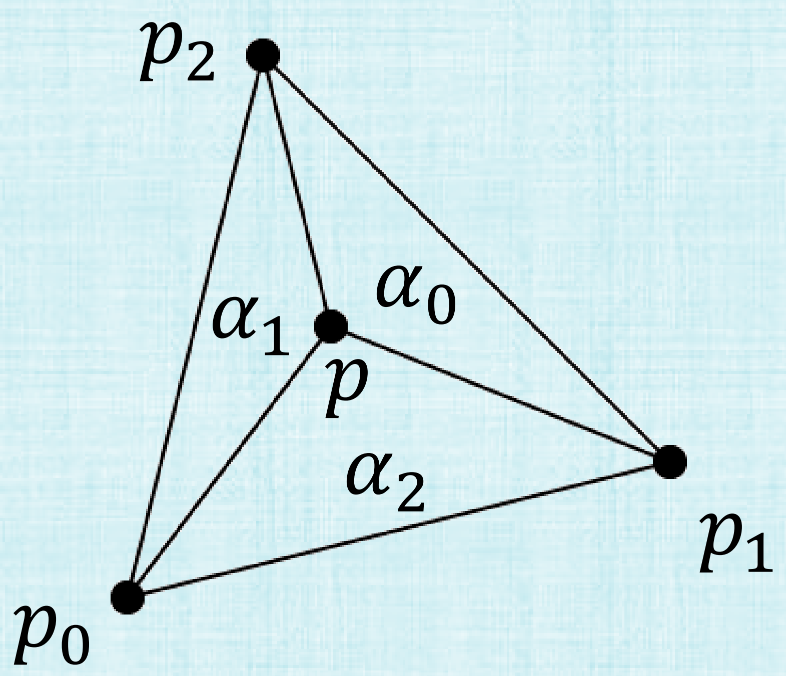
2D/3D Line Segments

- Linearly interpolate between p_0 and p_1 via $p(\alpha) = (1-\alpha)p_0 + \alpha p_1$ where $\alpha = \frac{\|p-p_0\|_2}{\|p_1-p_0\|_2}$ is the fraction of the distance from p_0 to p_1
- Barycentric weights: Any point p on the segment can be written as $p(\alpha_0, \alpha_1) = \alpha_0 p_0 + \alpha_1 p_1$ with $\alpha_0, \alpha_1 \in [0,1]$ and $\alpha_0 + \alpha_1 = 1$
- Weights are computed via lengths: $\alpha_0 = \frac{\|p-p_1\|_2}{\|p_1-p_0\|_2} = 1-\alpha$ and $\alpha_1 = \frac{\|p-p_0\|_2}{\|p_1-p_0\|_2} = \alpha$



2D/3D Triangles

- Barycentric weights: Points p in the triangle can be written as $p(\alpha_0, \alpha_1, \alpha_2) = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2$ with $\alpha_0, \alpha_1, \alpha_2 \in [0,1]$ and $\alpha_0 + \alpha_1 + \alpha_2 = 1$
- Weights are computed via areas: $\alpha_0 = \frac{Area(p,p_1,p_2)}{Area(p_0,p_1,p_2)}$, $\alpha_1 = \frac{Area(p_0,p,p_2)}{Area(p_0,p_1,p_2)}$, $\alpha_2 = \frac{Area(p_0,p_1,p)}{Area(p_0,p_1,p_2)}$
  - Triangle Area: $Area(p_0, p_1, p_2) = \frac{1}{2} \| \overrightarrow{p_0 p_1} \times \overrightarrow{p_0 p_2} \|_2$



Algebraic (instead of Geometric) Approach

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

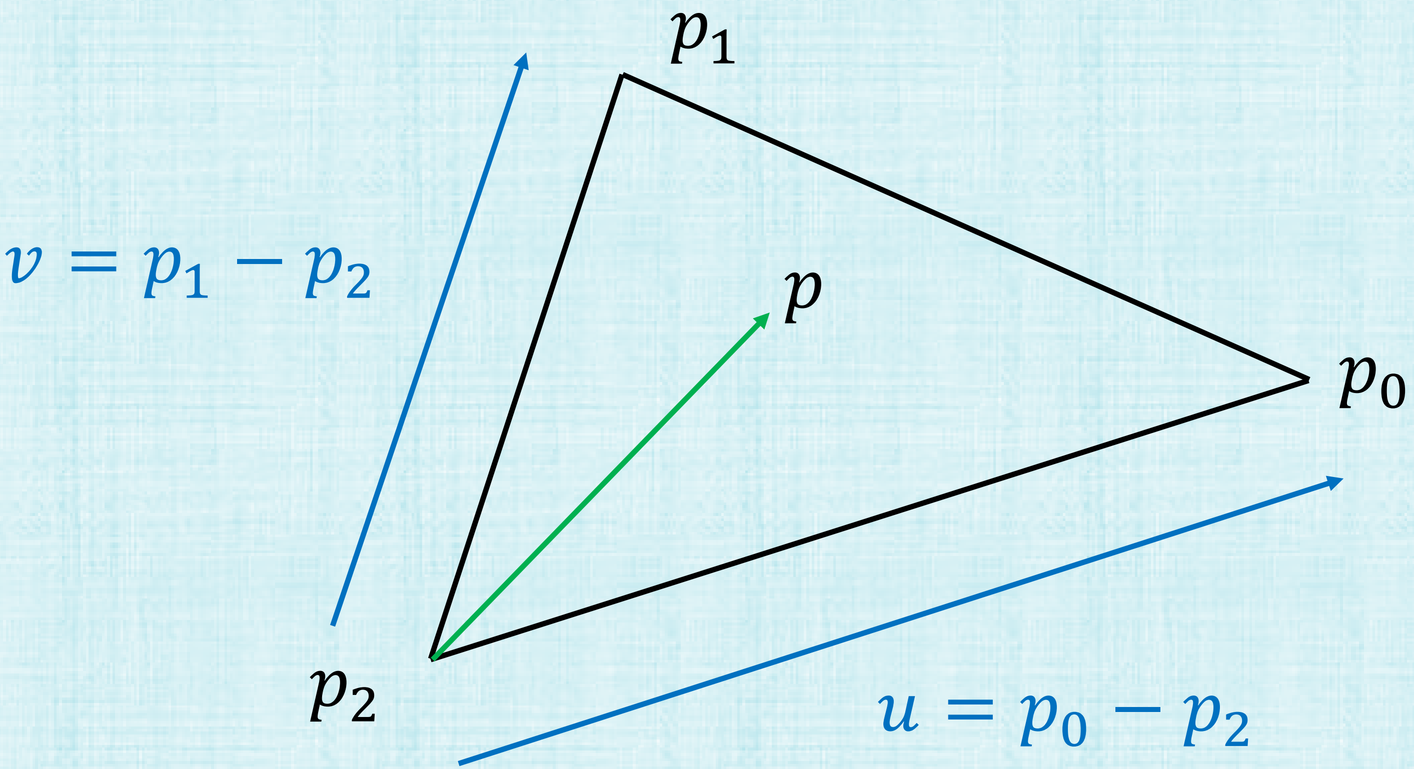
- Assemble into matrix form: $\begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \\ z_0 - z_2 & z_1 - z_2 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} x - x_2 \\ y - y_2 \\ z - z_2 \end{pmatrix}$

- The coefficient matrix is rank 1 when the columns (i.e. triangle edges) are colinear, implying infinite solutions for triangles with zero area (one can still embed p on an appropriate edge)
- 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 $A^T A \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = A^T b$ with a 2x2 coefficient matrix $A^T A$
- Invert the 2x2 coefficient matrix to solve the system of 2 equations with 2 unknowns to obtain α_0 and α_1; then, $\alpha_2 = 1 - \alpha_0 - \alpha_1$

gives the same answer
as the previous slide

Triangle Basis Vectors (a Linear Algebra approach)

- Compute edge vectors $u = p_0 - p_2$ and $v = p_1 - p_2$
- Points in the triangle have the form $p = p_2 + \beta_0 u + \beta_1 v$ with $\beta_0, \beta_1 \in [0,1]$ and $\beta_0 + \beta_1 \leq 1$
- Substitutions give $p = \beta_0 p_0 + \beta_1 p_1 + (1 - \beta_0 - \beta_1) p_2$ implying that: $\alpha_0 = \beta_0,\ \alpha_1 = \beta_1,$ $\alpha_2 = 1 - \beta_0 - \beta_1 = 1 - \alpha_0 - \alpha_1$ equivalent to the previous two slides



Ray Tracing vs. Scanline Rendering

Ray Tracing:

- Creates a ray for each a pixel, and intersects that ray with world space triangles
- Barycentric weights can be used to interpolate z values to the point of intersection
- When two triangles intersect a ray, the one with the smaller z value is used
- Aside: operating in world space is a huge advantage for the ray tracer, as it can "look around" in world space to figure out what's going on (resulting in higher quality images)

Scanline Rendering:

- Operates in screen space, where triangles have been distorted by the screen space projection
- Thus, barycentric weight interpolation of z values gives incorrect results
- Aside: operating in screen space with more limited information makes scanline rendering a good candidate for hardware acceleration (only recently have hardware implementations of ray tracing become more feasible)

Working in Screen Space

- Project triangle vertices p_0, p_1, p_2 into screen space to get p_0', p_1', p_2'
  - For each vertex $i = 0, 1, 2$, write (x_i, y_i, z_i) and $(x_i', y_i', z_i') = \left(\frac{hx_i}{z_i}, \frac{hy_i}{z_i}, n + f - \frac{fn}{z_i}\right)$

- Given a pixel at p' with barycentric weights $p' = \alpha_0' p_0' + \alpha_1' p_1' + \alpha_2' p_2'$, we need to compute $z \neq \alpha_0' z_0 + \alpha_1' z_1 + \alpha_2' z_2$ in order to handle occlusion/transparency

- Let p be the world space point that projects to p', noting that world space barycentric weight interpolation can be used to correctly calculate z at p
- Using triangle basis vectors:

$$p = p_2 + \alpha_0(p_0 - p_2) + \alpha_1(p_1 - p_2) = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} + \begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \\ z_0 - z_2 & z_1 - z_2 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$$

where α_0 and α_1 are unknowns that need to be determined

Working in Screen Space

- Since the point p projects to the pixel p':

$$p' = \begin{pmatrix} \frac{hx}{z} \\ \frac{hy}{z} \end{pmatrix} = \frac{h}{z}\left[\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}\right]$$

- Pixel p' is in a screen space triangle with basis vectors $u' = p'_0 - p'_2$ and $v' = p'_1 - p'_2$ where it has screen space barycentric weights $\alpha'_0, \alpha'_1, \alpha'_2$; thus,

$$p' = p'_2 + \alpha'_0 u' + \alpha'_1 v' = \begin{pmatrix} x'_2 \\ y'_2 \end{pmatrix} + \begin{pmatrix} u'_1 & v'_1 \\ u'_2 & v'_2 \end{pmatrix}\begin{pmatrix} \alpha'_0 \\ \alpha'_1 \end{pmatrix}$$

- Set these two expressions for p' equal to each other:

$$\frac{h}{z}\left[\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}\right] = \begin{pmatrix} x'_2 \\ y'_2 \end{pmatrix} + \begin{pmatrix} u'_1 & v'_1 \\ u'_2 & v'_2 \end{pmatrix}\begin{pmatrix} \alpha'_0 \\ \alpha'_1 \end{pmatrix}$$

Solve for α_0 and α_1
in terms of α'_0 and α'_1

Working in Screen Space

- Rewrite the first term on the right hand side as

$$\begin{pmatrix} x_2' \\ y_2' \end{pmatrix} = \frac{h}{z_2}\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \frac{h}{z}\left[\frac{z}{z_2}\begin{pmatrix} x_2 \\ y_2 \end{pmatrix}\right] = \frac{h}{z}\left[\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} \frac{z_0}{z_2}x_2 - x_2 & \frac{z_1}{z_2}x_2 - x_2 \\ \frac{z_0}{z_2}y_2 - y_2 & \frac{z_1}{z_2}y_2 - y_2 \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}\right]$$

- After cancellation of the blue terms and moving the green terms to the left hand side:

$$\frac{h}{z}\begin{pmatrix} x_0 - \frac{z_0}{z_2}x_2 & x_1 - \frac{z_1}{z_2}y_2 \\ y_0 - \frac{z_0}{z_2}y_2 & y_1 - \frac{z_1}{z_2}y_2 \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} u_1' & v_1' \\ u_2' & v_2' \end{pmatrix}\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$$

$$\frac{1}{z}\begin{pmatrix} x_0' - x_2' & x_1' - x_2' \\ y_0' - y_2' & y_1' - y_2' \end{pmatrix}\begin{pmatrix} z_0\alpha_0 \\ z_1\alpha_1 \end{pmatrix} = \begin{pmatrix} u_1' & v_1' \\ u_2' & v_2' \end{pmatrix}\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$$

$$\frac{1}{z}\begin{pmatrix} z_0\alpha_0 \\ z_1\alpha_1 \end{pmatrix} = \begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$$

Notice that all the x and y terms vanished!

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

$$\begin{pmatrix} z_0 - (z_0 - z_2)\alpha_0' & -(z_1 - z_2)\alpha_0' \\ \cdots & \cdots \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \cdots \end{pmatrix} = \begin{pmatrix} z_2\alpha_0' \\ \cdots \end{pmatrix}$$

Finding the World Space Barycentric Weights

- Invert the 2x2 coefficient matrix:

$$\begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} = \frac{1}{z_0 z_1 - z_1(z_0 - z_2)\alpha'_0 - z_0(z_1 - z_2)\alpha'_1}\begin{pmatrix} z_1 - (z_1 - z_2)\alpha'_1 & (z_1 - z_2)\alpha'_0 \\ (z_0 - z_2)\alpha'_1 & z_0 - (z_0 - z_2)\alpha'_0 \end{pmatrix}\begin{pmatrix} z_2\alpha'_0 \\ z_2\alpha'_1 \end{pmatrix}$$

$$= \frac{1}{z_1 z_2 \alpha'_0 + z_0 z_2 \alpha'_1 + z_0 z_1 \alpha'_2}\begin{pmatrix} z_1 z_2 \alpha'_0 \\ z_0 z_2 \alpha'_1 \end{pmatrix}$$

- In summary, given the barycentric weights α'_0 and α'_1 of pixel p', we can compute:

$$\alpha_0 = \frac{z_1 z_2 \alpha'_0}{z_1 z_2 \alpha'_0 + z_0 z_2 \alpha'_1 + z_0 z_1 \alpha'_2} \quad \text{and} \quad \alpha_1 = \frac{z_0 z_2 \alpha'_1}{z_1 z_2 \alpha'_0 + z_0 z_2 \alpha'_1 + z_0 z_1 \alpha'_2}$$

- In addition, $\alpha_2 = \frac{z_0 z_1 \alpha'_2}{z_1 z_2 \alpha'_0 + z_0 z_2 \alpha'_1 + z_0 z_1 \alpha'_2}$ from $\alpha_2 = 1 - \alpha_0 - \alpha_1$

- Then, α_0, α_1, α_2 can be used to find $p = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2$ on the world space triangle

Depth Buffer

- Given word space barycentric weights $\alpha_0, \alpha_1, \alpha_2$ for the point p that projects to a pixel located at p', the z value at p can be computed via $z = \alpha_0 z_0 + \alpha_1 z_1 + \alpha_2 z_2$
- When two triangles overlap the same pixel, there are two different points p (one on each triangle) and the one with the smaller z value is used to color the pixel

- 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 interpolated correctly with screen space barycentric weights (in contrast to most other quantities)
- Using $z'_i = n + f - \frac{fn}{z_i}$, we have:

$$z' = n + f - \frac{fn}{z} = n + f - fn\left[\alpha'_0\left(\frac{1}{z_0}\right) + \alpha'_1\left(\frac{1}{z_1}\right) + \alpha'_2\left(\frac{1}{z_2}\right)\right] = \alpha'_0 z'_0 + \alpha'_1 z'_1 + \alpha'_2 z'_2$$

  - That is, z' can also be interpolated correctly with screen space barycentric weights!
- Since $\frac{dz'}{dz} = \frac{fn}{z^2} > 0$, comparing z' values is as valid as comparing z values

- Note: Even though the $\alpha_0, \alpha_1, \alpha_2$ aren't needed for the final result (only to derive it), we will need them for texture mapping (later in the quarter)

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: