path from vertex to pixel

- see pipeline fig
- three vertices have passed through the vertex shader
- glPosition has been set to the clip coordinates of each vertex.
- follow their journey to become a bunch of pixels
- we might imagine that openGL needs a divide-by-w step to position the vertices on the screen.
- openGL also needs a few more steps.

Clipping

- process triangles that are fully or partially out of view.
- we don’t want to see behind us
- we want to minimize processing
- the tricky part will be to deal with eye-spanning triangles.
- the openGL API: we keep only points in the range
  \[-1 < x_n < 1\]  \(1\)
  \[-1 < y_n < 1\]  \(2\)
  \[-1 < z_n < 1\]  \(3\)

eye spanners

- see figure
- back vertex projects higher up in the image
- filling in the in-between pixels will fill in the wrong region.
- solution: slice up the geometry by the six faces of the view frustum

coordinates for clipping

- if you wait for NDCs the vertex has flipped, and it’s too late to do the clipping.
- could do in eye space, but then would need to use the camera parameters
- canonical solution: use clip coordinates, post matrix multiply but pre divide.
  - no divide = no flipping
- recall that we want points in the range
  \[-1 < x_n < 1\]  \(4\)
  \[-1 < y_n < 1\]  \(5\)
  \[-1 < z_n < 1\]  \(6\)

- in clip coordinates this is:
  \[-w_c < x_c < w_c\]
  \[-w_c < y_c < w_c\]
  \[-w_c < z_c < w_c\]

- since no divide has been done between eye coordinates and clip coordinates, no flipping has occurred yet!
  - we dealing with (4D) coordinates that are linearly related to eye coordinates.
• clipping is done, openGL can now divide by \( w_c \) to obtain normalized device coordinates!

**Backface Culling**

• when drawing a closed solid object, we will only ever see one “front” side of each triangle.
• for efficiency we can drop these from the processing
• To do this, in openGL, we use the convention of ordering the three vertices in the draw call (IBO/VBO) so that they are counter clockwise (CCW) when looking at its front side.
• during setup, we call `glEnable(GL_CULL_FACE)`, `glFrontFace(GL_CCW)` (the default), and `glCullFace(GL_BACK)` (the default)
• to implement culling, openGL does the following:
  • Let \( \tilde{p}_1, \tilde{p}_2, \text{ and } \tilde{p}_3 \) be the three vertices of the triangle projected to the \((x_n, y_n, 0)\) plane.
  • Define the vectors \( \vec{a} = \tilde{p}_3 - \tilde{p}_2 \) and \( \vec{b} = \tilde{p}_1 - \tilde{p}_2 \).
  • Next compute the cross product \( \vec{c} = \vec{a} \times \vec{b} \).
  • If the three vertices are counter clockwise in the plane, then \( \vec{c} \) will be in the \( z_n \) direction. Otherwise it will be in the positive \(-z_n\) direction.
  • When all the dust settles, this coordinate is
  \[
  (x_n^3 - x_n^2)(y_n^1 - y_n^2) - (y_n^3 - y_n^2)(x_n^1 - x_n^2)
  \]
  (7)

**Viewport**

• now openGL wants to position the vertices in the window. so it is time to move to window coordinates
  • each pixel center has an integer coordinate.
    – this will make subsequent pixel computations more natural.
• openGL wants the lower left pixel center to have 2D window coordinates of \([0, 0]^t\) and the upper right pixel center to have coordinates \([W - 1, H - 1]^t\).
• openGL will think of each pixel as owning the real estate which extends .5 pixel units in the positive and negative, horizontal and vertical directions from the pixel center.
• Thus the extent of 2D window rectangle covered by the union of all our pixels is the rectangle in window coordinates with lower left corner \([-0.5, -0.5]^t\) and upper right corner \([W - 0.5, H - 0.5]^t\).

**viewport matrix**

• openGL needs a transform that maps the lower left corner to \([-0.5, -0.5]^t\) and upper right corner to \([W - 0.5, H - 0.5]^t\).
• the appropriate scale and shift can be done using the viewport matrix
  \[
  \begin{bmatrix}
  x_w \\
  y_w \\
  z_w
  \end{bmatrix}
  =
  \begin{bmatrix}
  W/2 & 0 & 0 & (W - 1)/2 \\
  0 & H/2 & 0 & (H - 1)/2 \\
  0 & 0 & 1/2 & 1/2 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \begin{bmatrix}
  x_n \\
  y_n \\
  z_n
  \end{bmatrix}
  \]
  (8)
  • this does a scale and shift in both \(x\) and \(y\).
  • you can verify that it maps the corners appropriately.
• In openGL, we set up this viewport matrix with the call `glViewport(0,0,W,H)`.
• The third row of this matrix is used to map the \([-1..1]\) range of \(z_n\) values to the more convenient \([0..1]\) range.
• so now (in our conventions), \(z_w = 0\) is far and \(z_w = 1\) is near.
any points out of this range will be clipped away.
so we must also tell OpenGL that when we clear the z-buffer, OpenGL should set it to 0; we do this with the call
\texttt{glClearDepth(0.0)}.

texture Viewport

- the abstract domain for textures is not the canonical square, but instead is the \textit{unit square}
- in this case the coordinate transformation matrix is

$$
\begin{bmatrix}
  x_w \\
y_w \\
1
\end{bmatrix} =
\begin{bmatrix}
  W & 0 & 0 & -1/2 \\
  0 & H & 0 & -1/2 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x_t \\
y_t \\
1
\end{bmatrix}
$$

(9)

Rasterization

- Starting from the window coordinates for the three vertices, OpenGL’s rasterizer needs to figure out which pixel-centers are inside of the triangle.
- Each triangle on the screen can be defined as the intersection of three half-spaces.
- Each such halfspace is defined by a line that coincides with one of the edges of the triangle, and can be tested using an “edge function” of the from

$$
\text{edge} = ax_w + by_w + c
$$

where the \((a, b, c)\) are constants that depend on the geometry of the edge.
- A positive value of this function at a pixel with coordinates \([x_w, y_w]^T\) means that the pixel is inside the specified halfspace.
- If all three tests pass, then the pixel is inside the triangle.

speed up

- only look at pixels in the bounding box of the triangle
- test if a pixel block is entirely outside of triangle
- use incremental calculations along a scanline

detail: boundaries

- for pixel on edge or vertex it should be rendered exactly once
- need special care in the implementation.

important: interpolation

- recall that at each vertex we have a \(z_w\) value.
- and that at each pixel inside the triangle: \(z_n = ax_n + by_n + c\), for some fixed \(a, b\) and \(c\), so \(z_w = ax_w + by_w + c\), for some fixed (but different) \(a, b\) and \(c\).
- we will also have other data values, say \(f\), at each vertex that will be related to, but not identical to the varying variables. (more on this “related to” notion later).
- these \(f\) will have the property that at each pixel inside the triangle, we will want

$$
f = ax_w + by_w + c
$$

for some \((a, b, c)\).
- this form is called an affine function over window coordinates.
- note that the edge functions are also affine functions over window coordinates.

rasterizer’s job

- So it will also rasterizer’s job to evaluate \(z_w\) and each \(f\) at every pixel inside the triangle.
- since these are affine functions over window coordinates, the rasterizer can do this efficiently. (incremental computation).
- we refer to this process as linear interpolation (over the window)