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.

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

\[
\begin{align*}
-1 & < x_n < 1 \\
-1 & < y_n < 1 \\
-1 & < z_n < 1 \\
\end{align*}
\]

- in clip coordinates this is:

\[
\begin{align*}
-w_c & < x_c < w_c \\
-w_c & < y_c < w_c \\
-w_c & < z_c < w_c \\
\end{align*}
\]

divide

- clipping is done, OpenGL can now divide by w 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\texttt{glEnable(GL\_CULL\_FACE)}, \texttt{glFrontFace(GL\_CCW)} (the default), and \texttt{glCullFace(GL\_BACK)} (the default)

to implement culling, OpenGL does the following:

Let $\tilde{p}_1$, $\tilde{p}_2$, and $\tilde{p}_3$ be the three vertices of the triangle projected down to the $(x, y, 0)$ 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 $z$ direction. Otherwise it will be in the positive $-z$ direction.

When all the dust settles, this coordinate is

$$\begin{align*}
(x_n^3 - x_n^2)(y_n^1 - y_n^2) - (y_n^3 - y_n^2)(x_n^1 - x_n^2)
\end{align*}$$

Viewport

now OpenGL wants to position the vertices in the window. so it is time to move to window coordinates

- 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 $[-.5, -.5]^t$ and upper right corner $[W - .5, H - .5]^t$.

viewport matrix

OpenGL needs a transform that maps the lower left corner to $[-.5, -.5]^t$ and upper right corner to $[W - .5, H - .5]^t$.

the appropriate scale and shift can be done using the viewport matrix

$$\begin{pmatrix}
\frac{x}{w} \\
\frac{y}{w} \\
\frac{z}{w}
\end{pmatrix} = \begin{pmatrix}
\frac{W}{2} & 0 & 0 & \frac{(W - 1)}{2} \\
0 & \frac{H}{2} & 0 & \frac{(H - 1)}{2} \\
0 & 0 & 1/2 & 1/2 \\
0 & 0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
\frac{x}{n} \\
\frac{y}{n} \\
\frac{z}{n} \\
1
\end{pmatrix}$$

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 \texttt{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.

- 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

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

(6)

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 form

\[
\text{edge} = ax_w + by_w + c
\]

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

interpolation

• As input to rasterization, each vertex also has some auxiliary data associated with it.
  – This data includes a \(z_w\) value,
  – as well as other data that is related, but not identical to the varying variables

• It is also the job of the rasterizer to linearly interpolate this data over the triangle.

• Each such value \(f\) to be linearly interpolated can be represented as an affine function over screen space with the form

\[
f = ax_w + by_w + c
\]

(7)

• An affine function can be easily evaluated at each pixel by the rasterizer.

• Indeed, this is no different from evaluating the edge test functions just described.

boundaries

• for pixel on edge or vertex it should be rendered exactly once

• need special care in the implementation.