Visibility

• in the real world, opaque objects block light.
• we need to model this computationally
• one idea is to render back to front and use overwriting
  – this will have problem with visibility cycles
• we could explicitly store everything hit along a ray and then compute the closest
  – makes sense in a ray tracing setting, where we are working one pixel/ray at time, but not for OpenGL, where we are working one triangle at a time.

z-buffer

• we will use use z-buffer
• triangles are drawn in any order
• each pixel in framebuffer stores “depth” value of closest geometry observed so far
• When a new triangle tries to set the color of a pixel, we first compare its depth to the value stored in the z-buffer. Only if the observed point in this triangle is closer do we overwrite the color and depth values of this pixel.
• this is done per-pixel, so no cycle problems.
• there are a optimizations where z-testing is done before the fragment shading is done

Other Uses of Visibility Calculations

• visibility to a light source is useful for shadows
  – we will talk about shadow mapping later
  – we will also discuss shadow calculations in a ray tracer
• Visibility computation can also be used to speed up the rendering process.
  – If we know that some object is occluded from the camera, then we don’t have to render the object in the first place.
  – can use a conservative test

Basic Mathematical Model

• for every point we define its \([x_n, y_n, z_n]^T\) coordinates using the following matrix expression.

\[
\begin{bmatrix}
x_n \, w_n \\ y_n \, w_n \\ z_n \, w_n \\ w_n
\end{bmatrix} =
\begin{bmatrix}
x_e \\ y_e \\ z_e \\ 1
\end{bmatrix} =
\begin{bmatrix}
s_x & 0 & -c_x & 0 \\ 0 & s_y & -c_y & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
x_e \\ y_e \\ z_e \\ 1
\end{bmatrix}
\]

(1)

• we now also have the value \(z_n = -\frac{1}{z_e}\).
• Our plan is to use this \(z_n\) value to do depth comparisons in our z-buffer.

correct ordering

• Given two points \(\tilde{p}^1\) and \(\tilde{p}^2\) with eye coordinates \([x_1^e, y_1^e, z_1^e, 1]^T\) and \([x_2^e, y_2^e, z_2^e, 1]^T\).
• Suppose that they both are in front of the eye, i.e., \(z_1^e < 0\) and \(z_2^e < 0\).
• And suppose that \(\tilde{p}^1\) is closer to the eye than \(\tilde{p}^2\), that is \(z_1^e < z_2^e\).
• Then \(-\frac{1}{z_2^e} < -\frac{1}{z_1^e}\), meaning \(z_n^2 < z_n^1\).

projective transform
we can now think of the process of taking points given by eye coordinates to points given by normalized device coordinates as an honest to goodness 3D geometric transformation.

This kind of transformation is generally neither linear nor affine, but is something called a 3D *projective transformation*.

projective transformations preserve co-linearity and co-planarity of points

**projective figure**

- first map film plane
  - iso-$z_e$ so iso $z_n$
- map the red segment
  - some straight segment
- then note that rays must hit same pixel, with same $(x_n, y_n)$, so map to parallel lines

$z_n$ **interp is right**

- preservation of coplanarity: for points on a fixed triangle, we will have $z_n = ax_n + by_n + c$, for some fixed $a$, $b$ and $c$.
- Thus, the correct $z_n$ value for a point can be computed using linear interpolation over the 2D image domain as long as we know its value at the three vertices of the triangle
- (see fig): projective transforms are funny
- linear interpolation of $z_e$ values over the screen would produce wrong answer.
- “red” should win for entire bottom half of image.
- suppose I really wanted to have $z_e$ at each fragment, how could I get it?... more later.

**Numerics**

- there can be numerical difficulties when computing $z_n$. As $z_e$ goes towards zero, the $z_n$ value diverges off towards (positive) infinity.
- Conversely, points very far from the eye have $z_n$ values very close to zero. The $z_n$ of two such far away points may be indistinguishable in a finite precision representation, and thus the z-buffer will be ineffective in distinguishing which is closer to the eye.

solution: near/far

- solution: replacing the third row of the matrix with the more general row $[0, 0, \alpha, \beta]$.
- it is easy to verify that if the values $\alpha$ and $\beta$ are both positive, then the z-ordering of points (assuming they all have negative $z_e$ values) is preserved under the projective transform.
- To set $\alpha$ and $\beta$, we first select depth values $n$ and $f$ called the near and far values (both negative), such that our main region of interest in the scene is sandwiched between $z_e = n$ and $z_e = f$.
- Given these selections, we set $\alpha = \frac{f + n}{f - n}$ and $\beta = -\frac{2fn}{f - n}$.
- We can verify now that any point with $z_e = f$ maps to a point with $z_n = -1$ and that a point with $z_e = n$ maps to a point with $z_n = 1$
- Any geometry not in this [near..far] range is clipped away by OpenGL and ignored
- see fig

**Code**

- In OpenGL, use of the z-buffer is turned on with a call to `glEnable(GL_DEPTH_TEST)`.
- We also need a call to `glDepthFunc(GL_GREATER)`, since we are using a right handed coordinate system where “closer to the eye” equals less-negative equals greater.
- in real life, you may see other conventions (for how to interpret $n$ and $f$, some of the signs of the matrix, and the handedness of the ultimate z-test.