Vectors in Action: Backface Culling

• Your first look at applying vectors and their operations in computer graphics is also a first look at an algorithm that can be done transparently for you by the graphics subsystem: hidden surface removal (HSR)

• Full-blown HSR will be tackled later, but there is one aspect that we can look at now: backface culling

• The principle behind backface culling is that, if a polygon on a 3D model is facing “away” from you, then you probably will not see it

Culling Exceptions

• It might already be apparent to you that, based on this principle, backface culling is not the full solution to HSR—exceptions are easy to identify:
  ◇ Non-convex polyhedrons
  ◇ Multiple objects
  ◇ “Flat” objects (e.g., a single triangle)

• However, it may “cull” a good number of polygons from the picture, thus serving as a kind of optimization
So, Those Vectors...

Determining whether a polygon faces front is easier than it may seem, thanks to some vector math: a polygon faces front if the angle between its normal and the vector toward the camera/eye/viewer is between –90 and 90 degrees, or

\[ \cos \theta \geq 0 \]

- Hmmm…we’ve seen cosines related to vectors before… oh yes, this:

\[ \cos \theta = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}| |\mathbf{v}|} \]

- Now, assuming WLOG that the camera/eye/viewer and normal vectors in the diagram above are unit vectors, this gives us:

\[ \mathbf{e} \cdot \mathbf{n} \geq 0 \]

- But, in a right-handed coordinate system, \( \mathbf{e} \) is simply \( <0, 0, 1> \), meaning \( \mathbf{e} \cdot \mathbf{n} = \text{the } z \text{ component of } \mathbf{n} \)

- Thus, backface culling is a matter of checking whether the \( z \) component of \( \mathbf{n} \) is greater than zero!
Yay! Oh, wait...

• So, this is great—just look at the normal vector and we’re done!

But...we don’t have the normal vector...

• Fortunately, we can compute it—vectors to the rescue one more time:

\[ u \times v = (y_u z_v - z_u y_v, z_u x_v - x_u z_v, x_u y_v - y_u x_v) \]

• Recall that this cross product computes the normal vector to the plane defined by its two operand vectors—this is exactly what we need!

But I Have Vertices!

• Digging one step deeper, you probably noticed that we don’t have vectors—we have vertices

• Again, no problem: vectors can be derived from the vertices by subtracting them from each other

• Given the vertices of a polygon, what vectors do we use to calculate the normal vector?

• Well...let’s just pick two that we have for sure: the vector from the first to the second vertex, then the one from the second to the third
And Finally...

- So we have our vectors, but you might have realized that there is one last detail: for a given plane defined by these two vectors, there are two possible normals—one for each side of the plane!

- As it turns out, the direction of the vertices influences the vectors that we get, and thus which normal gets computed when multiplying our two selected vectors.

- So we must not only standardize on our vectors, but the order in which we list our vertices.

- A little study reveals that the normal vector will have $z > 0$ when the vertices are listed counterclockwise as viewed from the polygon’s “front”.

- Thus, we finally meet the level of our vertex data and code with this rule:

  ```
  In order to use backface culling, we standardize on listing our polygons’ vertices so that they “wind” counterclockwise when viewed from their front.
  ```

- Once we have this, we can then just “turn it on:”

  ```
  gl.enable(gl.CULL_FACE);
  ```

  …thus gaining this bit of optimization!