Hidden Surface Removal (HSR)

Hidden surface

Visible surface determination
- Look at a cube that has opaque sides
- We see **at most** 3 sides
- However, all sides exist in the graphics pipeline
- Which sides are visible and which are hidden?

We do not wish to see things that are hidden behind other objects

Why HSR is necessary

Two main classes of HSR
- **Object-space approach**
  - works on object level,
  - comparing objects with each other
- **Image-space approach**
  - works on pixel level,
  - comparing pixel values
- We describe algorithms for scenes of planar polygons, e.g., triangles
Object-space algorithms
- Draw the surfaces in a particular order
- Does not work well with pipeline architectures, since surfaces are passed in an arbitrary order
- All objects must be available for sorting the surfaces (triangles)
- Example: painter’s, depth sort

Sorting
- Triangles should be sorted by their z value
- Sorting:
  1. Pick one of the k triangles
  2. Compare it pairwise with the remaining k-1 triangles
  3. Thereafter we know which part is visible
  4. Next triangle!
- $\mathcal{O}(k^2)$

Sorting
- Insertion sort
- Selection sort
- Bubble sort
  - sorting of an almost sorted list is fast
- $\mathcal{O}(k \log k)$ complexity for more sophisticated sorting algorithms

Painter’s algorithm
Works like a painter
- Draw the farthest object first and the closest last; possibly on top of other objects
- Typically object-based; compares objects with each other
- Makes quite many errors
- We draw unnecessary polygons

Depth sort
- Order polygons and deal with easy cases first and harder later
- Not every polygon is in front of or behind all other polygons

Image-space algorithms
- Work as part of the projection process
- Determine the relationship among object points on each projector
- Fits in well with the rendering pipeline
- Partial information is saved for each object
- Example: z-buffer, ray-tracing
Image-space approach
- Look at each projector and find closest of k polygons
- nm projectors for n x m frame buffer

z-buffer algorithm
- When calculating the z-value interpolate with q=1/z for correct result from a perspective view
- Image/pixel space
- Easy to implement in a pipeline structure (hardware)
- Always correct result (unless numerical errors)
- $O(k)$ worst case

Rendering polygons
- A polygon has two sides: front and back
- front is outward normal side
- Can render front and back differently
- Works nicely for cutaways

Polygons: front and back
- Convention: vertices of front facing polygons appear counter-clockwise on the screen
- front facing
- back facing
- outward facing normals

Front facing polygon
- Eye coordinates
  - Eye at origin
  - Looking at geometry down -z axis
  - $n \cdot v \geq 0$
Culling

- Culling is another form of clipping
  - removing polygons from the pipeline
  - speed up rendering
- Simply remove all surfaces pointing away from the viewer
- Render only the surfaces facing the viewer
- Culling produces a correct image only if we have a solid convex object, e.g., a sphere or a cube

Back-face removal

- $-90^\circ \leq \theta \leq 90^\circ$
- $\cos \theta \geq 0$
- $n \cdot v \geq 0$

Back-face culling

- We will never see the “inside” of solid objects; there is no reason to draw the backsides of the face polygons
- We can check if we see the front side of a polygon by checking if the angle between the normal and the vector pointing towards the viewer is smaller than 90 degrees
- After view transformation and perspective distortion, this simply becomes a check of the z-value of the normal vector

Efficiency analysis

- Assume that at least half of the triangles of a scene model face the viewer
- If we use z-buffer without culling, then $k$ triangles are passed through the pipeline
- If culling is enabled, then less than $\frac{1}{2} k$ triangles are passed