Buffers and Pipelines

Based on slides from Steve Marschner

Hidden surface elimination

• We have discussed how to map primitives to image space
  • projection and perspective are depth cues
  • occlusion is the MOST important depth cue

[Diagram of cubes]
Back face culling

- For closed shapes you will never see the inside
- therefore only draw surfaces that face the camera
- implement by checking $n \cdot v$

Painters’s algorithm

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer
Painter’s algorithm

- Amounts to a topological sort of the graph of occlusions
  - that is, an edge from A to B means A is occluded by B
- any sort is valid
  - ABCDEF
  - BADCFE
- if there are cycles there is no sort

Painter’s algorithm

- Useful when a valid order is easy to come by
- Compatible with alpha-blended transparency
The z buffer

- In many (most) applications maintaining a z sort is too expensive
- changes all the time as the view changes
- many data structures exist, but complex
- Solution: draw in any order, keep track of closest
- allocate extra channel per pixel to keep track of closest depth so far
- when drawing, compare object’s depth to current closest depth and discard if greater
- this works just like any other compositing operation

- another example of a memory-intensive brute force approach that works and has become the standard
Precision in z buffer

- The precision is distributed between the near and far clipping planes
- this is why these planes have to exist
- also why you can’t always just set them to very small and very large distances
- Generally use $z'$ (not world $z$) in z buffer

Interpolating in projection

linear interp. in screen space $\neq$ linear interp. in world (eye) space
Pipeline for minimal operation

- Vertex stage (position and color)
  - transform position (object -> screen space)
- Rasterize stage
  - fill in shape color
- Fragment stage
  - write color to framebuffer

Result of minimal pipeline
Pipeline for basic z buffer

- **Vertex stage** (position and color)
  - transform position (object -> screen space)
- **Rasterize stage**
  - interpolate \( z' \) (screen \( z \))
  - fill in shape color
- **Fragment stage**
  - write color to framebuffer if interpolated \( z' < \) current \( z' \)

Result of z-buffer pipeline
Flat shading

- Shade using the triangle normal
- Leads to constant shading and faceted appearance

Pipeline for flat shading

- Vertex stage (position and color)
  - transform position (object -> screen space)
  - compute shaded color per triangle using normal
- Rasterize stage
  - interpolate z’ (screen z)
  - fill in shape color
- Fragment stage
  - write color to framebuffer if interpolated z’ < current z’
Result of flat-shading pipeline

Lighting

- Phong illumination requires:
  - light vector
  - eye vector
  - surface normal
Directional light

- Directional (infinitely distant) light source
  - light vector always points in the same direction
  - often specified by position \([x\ y\ z\ \theta]\)
  - many pipelines are faster if you use directional lights

Gouraud shading

- Often draw smooth surfaces
  *GL_SMOOTH
  - compute colors at vertices using vertex normals
  - interpolate colors across triangles
  - “Gouraud shading”
  - “Smooth shading”
Pipeline for Gouraud shading

- Vertex stage (position and color)
  - transform position and normal (object -> screen space)
  - compute shaded color per triangle using normal
- Rasterize stage
  - interpolate z’ (screen z), and color
  - fill in shape color
- Fragment stage
  - write color to framebuffer if interpolated z’ < current z’

Result of Gouraud shading
Vertex normals

- Need normals at vertices to compute Gouraud shading
- Best to get vertex normals from the geometry
  - e.g. spheres
- Otherwise have to infer vertex normals from triangles
  - simple scheme: average surrounding face normals

\[ N_v = \frac{\sum_i N_i}{\| \sum_i N_i \|} \]

---

Non-diffuse Gouraud shading

- Results are not so good with fast-varying models like specular ones
- problems with any highlights smaller than a triangle
Phong shading

- Get higher quality by interpolating the normal
- as easy as interpolating the color
- evaluating the illumination model per pixel rather than per vertex
- in pipeline, this means moving illumination from the vertex processing stage to the fragment processing stage

Phong shading

- Produces much better highlights
Pipeline for Phong shading

- Vertex stage (position and color)
  - transform position and normal (object -> screen space)
  - compute shaded color per triangle using normal
- Rasterize stage
  - interpolate $z'$ (screen $z$), color, and $x, y, z$ normal
  - fill in shape color
- Fragment stage
  - compute shading using interpolated and color
  - write color to framebuffer if interpolated $z' <$ current $z'$

Result of Phong shading

Not implemented in OpenGL. You must write your own shaders to do this.