Pipeline Operations

CS 4620

Pipeline

you are here

APPLICATION

COMMAND STREAM

3D transformations; shading

VERTEX PROCESSING

TRANSFORMED GEOMETRY

conversion of primitives to pixels

RASTERIZATION

FRAGMENTS

blending, compositing, shading

FRAGMENT PROCESSING

FRAMEBUFFER IMAGE

user sees this

DISPLAY

Pipeline of transformations

- Standard sequence of transforms

object space

R

modeling transformation

camera space

R

camera transformation

projection transformation

viewport transformation

screen space

canonical view volume

Hidden surface elimination

- We have discussed how to map primitives to image space
  - projection and perspective are depth cues
  - occlusion is another very important cue
**Back face culling**

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

**Painter’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 sometimes occludes B
  - any sort is valid
    - ABCDEF
    - BADCFe
  - if there are cycles there is no sort

- Useful when a valid order is easy to come by
- Compatible with alpha blending
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

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 ≠ linear interp. in world (eye) space
The precision of z-buffers must be handled with great care when perspective images are created. The value \( \Delta z \) above is used after the perspective divide. Recall from Section 7.3 that the result of the perspective divide is

\[
z = n + f - \frac{f n}{z_w}.
\]

The actual bin depth is related to \( z_w \), the world depth, rather than \( z \), the post-perspective divide depth. We can approximate the bin size by differentiating both sides:

\[
\Delta z \approx \frac{f_n \Delta z_w}{z_w^2}.
\]

Bin sizes vary in depth. The bin size in world space is

\[
\Delta z_w \approx \frac{z_w^2 \Delta z}{f n}.
\]

Note that the quantity \( \Delta z \) is as discussed before. The biggest bin will be for \( z' = f \), where

\[
\Delta z_w^{\text{max}} \approx \frac{f \Delta z}{n}.
\]

Note that choosing \( n = 0 \), a natural choice if we don’t want to lose objects right in front of the eye, will result in an infinitely large bin—a very bad condition. To make \( \Delta z_w^{\text{max}} \) as small as possible, we want to minimize \( f \) and maximize \( n \). Thus, it is always important to choose \( n \) and \( f \) carefully.

### Pipeline for minimal operation

- **Vertex stage** (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color
- **Rasterizer**
  - pass through color
- **Fragment stage** (output: color)
  - write to color planes only if interpolated \( z' < \) current \( z' \)

### Pipeline for basic z buffer

- **Vertex stage** (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color
- **Rasterizer**
  - interpolated parameter: \( z' \) (screen z)
  - pass through color
- **Fragment stage** (output: color, \( z' \))
  - write to color planes only if interpolated \( z' < \) current \( z' \)
**Flat shading**

- Shade using the real normal of the triangle
  - same result as ray tracing a bunch of triangles
- Leads to constant shading and faceted appearance
  - truest view of the mesh geometry

**Pipeline for flat shading**

- **Vertex stage** (input: position / vtx; color and normal / tri)
  - transform position and normal (object to eye space)
  - compute shaded color per triangle using normal
  - transform position (eye to screen space)
- **Rasterizer**
  - interpolated parameters: \( z' \) (screen z)
  - pass through color
- **Fragment stage** (output: color, \( z' \))
  - write to color planes only if interpolated \( z' < \) current \( z' \)
Local vs. infinite viewer, light

- Phong illumination requires geometric information:
  - light vector (function of position)
  - eye vector (function of position)
  - surface normal (from application)
- Light and eye vectors change
  - need to be computed (and normalized) for each face

*Look at case when eye or light is far away:*
  - distant light source: nearly parallel illumination
  - distant eye point: nearly orthographic projection
  - in both cases, eye or light vector changes very little
- Optimization: approximate eye and/or light as infinitely far away

Directional light

- Directional (infinitely distant) light source
  - light vector always points in the same direction
  - often specified by position \([x \ y \ z \ 0]\)
- many pipelines are faster if you use directional lights

Infinite viewer

- Orthographic camera
  - projection direction is constant
- “Infinite viewer”
  - even with perspective, can approximate eye vector using the image plane normal
  - can produce weirdness for wide-angle views
  - Blinn-Phong: light, eye, half vectors all constant!
Gouraud shading

- Often we’re trying to draw smooth surfaces, so facets are an artifact
  - compute colors at vertices using vertex normals
  - interpolate colors across triangles
  - “Gouraud shading”
  - “Smooth shading”

Pipeline for Gouraud shading

- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - compute shaded color per vertex
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: z’ (screen z); r, g, b color
- Fragment stage (output: color, z’)
  - write to color planes only if interpolated z’ < current z’

Result of Gouraud shading pipeline

Vertex normals

- Need normals at vertices to compute Gouraud shading
- Best to get vtx. normals from the underlying geometry
  - e. g. spheres example
- Otherwise have to infer vtx. normals from triangles
  - simple scheme: average surrounding face normals
  
  \[ N_v = \frac{\sum_i N_i}{\|\sum_i N_i\|} \]
Non-diffuse Gouraud shading

- Can apply Gouraud shading to any illumination model
  - it's just an interpolation method
- 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
  - just as easy as interpolating the color
  - but now we are evaluating the illumination model per pixel rather than per vertex (and normalizing the normal first)
  - in pipeline, this means we are moving illumination from the vertex processing stage to the fragment processing stage

Pipeline for Phong shading

- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - transform position (eye to screen space)
  - pass through color
- Rasterizer
  - interpolated parameters: \( z' \) (screen \( z \)); \( r, g, b \) color; \( x, y, z \) normal
- Fragment stage (output: color, \( z' \))
  - compute shading using interpolated color and normal
  - write to color planes only if interpolated \( z' \) < current \( z' \)
Result of Gouraud shading pipeline

Result of Phong shading pipeline