Pipeline Operations

COMP575/COMP770 Spring 2016
Pipeline overview

3D transformations; shading

conversion of primitives to pixels

blending, compositing, shading

user sees this

you are here

APPLICATION

COMMAND STREAM

VERTEX PROCESSING

TRANSFORMED GEOMETRY

RASTERIZATION

FRAGMENTS

FRAGMENT PROCESSING

FRAMEBUFFER IMAGE

DISPLAY
Pipeline of transformations

• Standard sequence of transforms
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

[Foley et al.]
Painter’s algorithm

- 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
The z buffer

– 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 ≠ linear interp. in world (eye) space
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
Result of minimal pipeline
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 $z' < \text{current } z'$
Result of z-buffer pipeline
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 $z’$
Result of flat-shading pipeline
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
Local vs. infinite viewer, light

- 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 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
Phong shading

- Bottom line: produces much better highlights
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 Phong shading pipeline