Pipeline Operations

CS 465 Lecture 16

Operations in the pipeline

• Fundamental to (almost) all 3D applications:
  – vertex stage: coordinate transformation
  – fragment stage: hidden surface elimination

• Examples of additional operations:
  – Flat shading at the vertex stage
  – Gouraud shading at the vertex stage
  – Phong shading at the fragment stage
  – Texture mapping at the fragment stage

Pipeline overview

APPLICATION

COMMAND STREAM

GEOMETRY PROCESSING

TRANSFORMED GEOMETRY

RASTERIZATION

FRAGMENTS

FRAMEBUFFER IMAGE

3D transformations; shading

conversion of primitives to pixels

blending, compositing, shading

user sees this

DISPLAY

you are here

Modeling transformation

• Application specifies primitives in any convenient object coordinates
  – also specifies the transformation to world space (frame-to-canonical for object frame): the modeling matrix
  – e.g. car driving down street
    • car body specified in frame attached to car
    • tire specified in frame attached to wheel
  – often objects’ coordinates can be constant over time

Car body

Car body
### Viewing transformation

- The application also chooses a camera pose (position and orientation)
  - this defines a coordinate frame for the camera
  - transform geometry into that frame for rendering
  - *viewing matrix* is the c.-to-b. transform of the camera frame
  - the resulting coordinates are *eye coordinates*
  - we can now assume that the camera is in standard pose

### Projection transformation

- Projection matrix maps from eye space to *clip space*
  - In this space, the two-unit cube $[-1,1]^3$ contains exactly what needs to be drawn

The view matrix rewrites all coordinates in eye space

### Viewport transformation

- A simple bookkeeping step to scale image
  - clip volume was a simple cube
  - rasterizer needs input in pixel coords
  - therefore scale and translate to map the $[-1,1]$ box to the desired rectangle in window coordinates, or *screen space*
- Also shift $z'$ to the desired range
  - usually that range is $[0,1]$ so that it can be represented by a fixed-point fraction
- Homogeneous divide usually happens here
**Vertex processing: spaces summary**

- Standard sequence of transforms

  \[
  \text{object} \rightarrow \text{clip} \rightarrow \text{screen}
  \]

**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 \( n \cdot v \)

**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

---

**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
- Importance of using 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 \( \text{vtx} \); color \( \text{tri} \))
  - 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 \( \text{vtx} \); color and normal \( \text{tri} \))
  - 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 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

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 \( z' \)

Texture in the graphics pipeline

- Texture coordinates are another attribute
  - the application sets them to control where the texture goes
- **Texturing as a fragment operation**
  - because the whole point is to vary quickly across the surface
- **Interpolating coordinates across triangles**
  - to do texturing at fragment stage, we need interpolated \((u, v)\) coordinates at each fragment
  - but—sad to say—you can’t interpolate \( u \) and \( v \) linearly in screen space
    - not only won’t you get 0.5 at the midpoint, you’ll get different answers depending on the view.

Rasterization: interpolation

- Interp. in screen space \( \neq \) interp. in eye space
  - but perspective preserves lines & planes
  - safe to interpolate screen-z because it lives in screen space
  - not correct to interpolate world \((x,y,z)\), texture \((u,v)\), etc.
Texture coordinate interp example

- Solution: interpolate u/w, 1/w and divide

Texture mapping demo