Pipeline Operations

CS 465 Lecture 17

Pipeline overview
- you are here

APPLICATION

COMMAND STREAM

GEOMETRY PROCESSING

TRANSFORMED GEOMETRY

RASTERIZATION

FRAGMENTS

FRAMEBUFFER IMAGE

DISPLAY

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

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

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

Viewing transformation
- the view matrix rewrites all coordinates in eye space
**Projection transformation**

- With geometry in eye space, projection is simple:
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z'
  \end{bmatrix} =
  \begin{bmatrix}
  -dx/z \\
  -dy/z \\
  -z
  \end{bmatrix} \sim
  \begin{bmatrix}
  dx \\
  dy \\
  -z
  \end{bmatrix} =
  \begin{bmatrix}
  d \\
  0 \\
  0
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z
  \end{bmatrix}
  \begin{bmatrix}
  0 \\
  0 \\
  -1
  \end{bmatrix}
  \begin{bmatrix}
  x' \\
  y' \\
  z'
  \end{bmatrix}
  \]

- To enable hidden surface removal, want to keep a pseudo-depth \( z' \) that increases with \( z \):

\[
\begin{bmatrix}
  x' \\
  y' \\
  z'
  \end{bmatrix} \sim
  \begin{bmatrix}
  \frac{x}{z} \\
  \frac{y}{z} \\
  -\frac{z}{z}
  \end{bmatrix} =
  \begin{bmatrix}
  d \\
  0 \\
  0 \\
  0 \\
  a \\
  b \\
  0
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z
  \end{bmatrix}
  \begin{bmatrix}
  0 \\
  0 \\
  -1
  \end{bmatrix}
  \begin{bmatrix}
  x' \\
  y' \\
  z'
  \end{bmatrix}
  \]

(recall this means “in a scalar multiple of”)

**Projection transformation**

- Just like \( x' \) and \( y' \) run from \(-1\) to \(1\), we’d like \( z' \) to run from \(-1\) to \(1\)

\[
\begin{align*}
\bar{z}(z) &= az + b \\
z'(n) &= -1 \
\Rightarrow \bar{z}(n) &= n \\
z'(f) &= 1 \
\Rightarrow \bar{z}(f) &= -f
\end{align*}
\]

- solving for \( a \) and \( b \) leads to

\[
\begin{align*}
a &= \frac{n+f}{n-f} \\
b &= \frac{2n_f}{f}
\end{align*}
\]

---

**Projection transformation**

- Thus the projection matrix for projection plane distance \( d \) and near and far distances \( n \) and \( f \) is:

\[
\begin{bmatrix}
  x' \\
  y' \\
  z'
  \end{bmatrix} \sim
  \begin{bmatrix}
  \frac{x}{z} \\
  \frac{y}{z} \\
  -\frac{z}{z}
  \end{bmatrix} =
  \begin{bmatrix}
  d \\
  0 \\
  0 \\
  0 \\
  a \\
  b \\
  0
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z
  \end{bmatrix}
  \begin{bmatrix}
  0 \\
  0 \\
  -1
  \end{bmatrix}
  \begin{bmatrix}
  x' \\
  y' \\
  z'
  \end{bmatrix}
  \]

---

**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 $\mathbf{n} \cdot \mathbf{v}$

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

**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**

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

**Painter’s algorithm**

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

**Painter’s algorithm**

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

**Painter’s algorithm**

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

- 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
    - BADC FE
  - if there are cycles there is no sort

**Diagram:**

- A
  - B
  - C
  - D
  - E
  - F
**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
    - BACDFE
  - 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
Interpolating in projection

![Diagram](image1)

linear interp. in screen space # linear interp. in world (eye) space

Interpolating in projection

![Diagram](image2)

linear interp. in screen space # linear interp. in world (eye) space

Interpolating in projection

![Diagram](image3)

linear interp. in screen space # linear interp. in world (eye) space

Interpolating in projection

![Diagram](image4)

linear interp. in screen space # linear interp. in world (eye) space

Interpolating in projection

![Diagram](image5)

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 $z' < \text{current } z'$
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’

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’