Graphics pipeline

Computer Graphics
CSE 167
Lecture 7
Pipeline overview

- You are here
  - APPLICATION
  - COMMAND STREAM
  - VERTEX PROCESSING
  - TRANSFORMED GEOMETRY
  - RASTERIZATION
  - FRAGMENTS
  - FRAGMENT PROCESSING
  - FRAMEBUFFER IMAGE
  - DISPLAY

3D transformations, shading

Conversion of primitives to pixels

Blending, compositing, shading

User sees this

Based on slides courtesy of Steve Marschner
Transformations

Object (or Model) Coordinate Frame

Model Transformation $M$

World Coordinate Frame

View Transformation $V$

Camera (or Eye) Coordinate Frame

Projection Transformation $P$

Normalized Device Coordinate Frame

Viewport Transformation $D$

Window (or Screen) Coordinate Frame

$$X_{\text{window}} = \begin{bmatrix}DPVMX_{\text{object}}\end{bmatrix}$$

- Object (or Model) coordinates
- World coordinates
- Camera (or Eye) coordinates
- Normalized device coordinates
- Window (or Screen) coordinates
Transformations

Object (or Model) Coordinate Frame

World Coordinate Frame

Camera (or Eye) Coordinate Frame

Normalized Device Coordinate Frame

Window (or Screen) Coordinate Frame

Model Transformation $M$

View Transformation $V$

Projection Transformation $P$

Viewport Transformation $D$

View frustum in camera (or eye) coordinate frame

Canonical view volume in normalized device coordinate frame

Window (or screen) coordinate frame

$x_0, y_0$

$x_1, y_1$

CSE 167, Winter 2018
Pipeline for minimal operation

• Vertex stage (input: position per vertex; color per triangle)
  – 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

But order is incorrect; blue sphere is closer to the eye
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

[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 an 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

[Foley et al.]
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 cannot simply set them to very small and very large distances
Interpolating in projection

Linear interpolation in screen space ≠ Linear interpolation in eye space
Pipeline for basic z buffer

• Vertex stage (input: position per vertex; color per triangle)
  – 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’
Result of z-buffer pipeline

Order is correct; blue sphere is closer to the eye
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 per vertex; color and normal per per triangle)
  – 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 \(z'\)
Transforming normal vectors

• Tangent vector $\mathbf{t}$ at surface point $\mathbf{X}$ is orthogonal to normal vector $\mathbf{n}$ at $\mathbf{X}$

$$\mathbf{t}^T \mathbf{n} = \mathbf{n}^T \mathbf{t} = 0$$

• Transformed tangent vector and transformed normal vector must also be orthogonal

$$\mathbf{t'}^T \mathbf{n'} = \mathbf{n'}^T \mathbf{t'} = 0$$
Transforming normal vectors

• Tangent vector can be thought of as a difference of points, so it transforms the same as a surface point

\[ t' = At \]

We are only concerned about direction of vectors, so do not add translation vector

• Normal vector does not transform the same as tangent vector

\[ n' \neq An \]

\[ n' = Mn \]

How is \( M \) related to \( A \)?
Transforming normal vectors

• How is $M$ related to $A$?

$$t'^{\top}n' = 0$$

$$(At)^{\top}Mn = 0$$

$$t^{\top}A^{\top}Mn = 0$$

$$t^{\top}n = 0 \text{ if } A^{\top}M = I$$

• Solve for $M$

$$M = (A^{\top})^{-1} = (A^{-1})^{\top} = A^{-T}$$

• Transform normal vectors using

$$n' = A^{-T}n$$
Result of flat-shading pipeline
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 per vertex)
  – 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
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 vertex
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!
Vertex normals

• Need normals at vertices to compute Gouraud shading
• Best to get vertex normals from the underlying geometry
  – For example, spheres
• Otherwise, have to infer vertex normals from triangles
  – Simple scheme: average surrounding face normals
Non-diffuse Gouraud shading

- Can apply Gouraud shading to any illumination model
  - It is an general interpolation method
- Results are not so good with fast-varying models like specular ones
  - Problems with any highlights smaller than a triangle

[Plate II.31 Shutterbug. Gouraud shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar’s PhotoRealistic RenderMan™ software.)]
Per-pixel (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)
Per-pixel (Phong) shading

- Bottom line: produces much better highlights
Pipeline for per-pixel shading

- **Vertex stage** (input: position, color, and normal per vertex)
  - 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 per-pixel shading pipeline
Programming hardware pipelines

• Modern hardware graphics pipelines are flexible
  – Programmer defines exactly what happens at each stage
  – Do this by writing shader programs in domain-specific languages called shading languages
  – Rasterization is fixed-function, as are some other operations (depth test, many data conversions, ...)
• One example: OpenGL and GLSL (GL Shading Language)
  – Several types of shaders process primitives and vertices; most basic is the vertex program
  – After rasterization, fragments are processed by a fragment program
GLSL Shaders

Application

Triangles → Attributes

Vertex program

Attributes → Varying parameters

Rasterizer

Varying parameters

Fragment program

Varying parameters

Depth → Color

Framebuffer

Uniform variables