Computer Graphics

Lecture 21
The Graphics Pipeline
Announcements

- A2 artifact voting is open - see Canvas announcement; voting closes Monday night
Mid-Late term Exam

• Take-home exam out Friday 2/19
  Due Monday 2/22 at 10pm.

• Similar to the homeworks, but no collaboration and no google.
  • Book is ok. Writing code is ok.
Final Project

• Proposals due in 1 week.
Questions?
Graphics Pipeline: Overview

you are here ➔ APPLICATION

3D transformations; shading ➔ VERTEX PROCESSING

conversion of primitives to pixels ➔ RASTERIZATION

blending, compositing, shading ➔ FRAGMENTS

user sees this ➔ FRAMEBUFFER IMAGE ➔ DISPLAY
Application sends geometric primitives to renderer (e.g., to GPU)

Vertices are transformed to image space (we've done lots of this!)

Primitives are converted into pixel-shaped "fragments"; values are interpolated across primitives.

Fragments are shaded, blended, and composited to determine pixel colors.

Pixel colors written to the framebuffer appear on the screen.
Command Stream

Application sends geometric primitives to renderer (e.g., to GPU)

What primitives?

• Points
• Line segments
  – and chains of connected line segments
• Triangles
• And that’s all!
  – Curves? Approximate them with chains of line segments
  – Polygons? Break them up into triangles
  – Curved surfaces? Approximate them with triangles
• Trend over the decades: toward minimal primitives
  – simple, uniform, repetitive: good for parallelism
Vertex Processing

Vertices are transformed to clip space
Vertex values are computed
(we've done much of this part!)
Rasterization

**Rasterization algorithms: next week**

- **First job:** enumerate the pixels covered by a primitive
  - which pixels fall inside triangle?
  - includes "clipping" content outside view volume
- **Second job:** interpolate values across the primitive
  - e.g. colors computed at vertices
  - e.g. normals at vertices
  - e.g. texture coordinates

Rasterization algorithms: next week
Fragment Processing

*Painter's algorithm; Z buffering: today*

- Hidden surface removal (occlusion) - only the closest object is drawn
- Per-fragment shading:
  - determine color of the pixel based on a shading model
  - diffuse color might come from a texture
- Blending, compositing - e.g.:
  - anti-aliasing
  - transparency / alpha blending
Hidden Surface Removal

Two motivations: realism and efficiency
Back face culling

• For closed shapes you will never see the inside
  —therefore only draw surfaces that face the camera
  —implement by checking $\mathbf{n} \cdot \mathbf{v} > 0$
Handling Occlusion

• What if multiple triangles are facing the viewer at different depths?

• **Painter's algorithm:** draw them back-to-front

• Topological sort on the occlusion graph:
  • if A ever occludes B, it must come after B in the drawing order

Works great if the ordering is easy to find...

... but often it isn't.

**Example:** z.obj
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
– store z as an integer for speed and memory efficiency (at the expense of precision!)
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

instead of using the smallest $z$, use the largest $1/z$
\[ P = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n + f & -f n \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ \frac{v}{z} \end{bmatrix} \]

\[ \mathbf{z}' = \mathbf{z} = n + f - \frac{fv}{z} \]

\[ \mathbf{z}' = k - k \frac{1}{z} \]

\[ \frac{-1}{\mathbf{z}^2} \mathbf{z} \]

\[ \mathbf{z}' \propto \frac{-1}{z} \]
Graphics Pipeline: Overview

you are here ➔ APPLICATION

APPLICATION ➔ COMMAND STREAM

COMMAND STREAM ➔ VERTEX PROCESSING

VERTEX PROCESSING ➔ TRANSFORMED GEOMETRY

TRANSFORMED GEOMETRY ➔ RASTERIZATION

Rasterization ➔ FRAGMENTS

FRAGMENTS ➔ FRAGMENT PROCESSING

Fragment Processing ➔ FRAMEBUFFER IMAGE

Framebuffer Image ➔ DISPLAY

DISPLAY ➔ user sees this

you are here ➔ APPLICATION

Application ➔ COMMAND STREAM

Command Stream ➔ VERTEX PROCESSING

Vertex Processing ➔ TRANSFORMED GEOMETRY

Transformed Geometry ➔ RASTERIZATION

Rasterization ➔ FRAGMENTS

Fragments ➔ FRAGMENT PROCESSING

Fragment Processing ➔ FRAMEBUFFER IMAGE

Framebuffer Image ➔ DISPLAY

DISPLAY ➔ user sees this

3D transformations; shading ➔ VERTEX PROCESSING

Conversion of primitives to pixels ➔ RASTERIZATION

Rasterization ➔ FRAGMENTS

Fragments ➔ FRAGMENT PROCESSING

Fragment Processing ➔ FRAMEBUFFER IMAGE

Framebuffer Image ➔ DISPLAY

DISPLAY ➔ user sees this

3D transformations; shading ➔ VERTEX PROCESSING

Conversion of primitives to pixels ➔ RASTERIZATION

Rasterization ➔ FRAGMENTS

Fragments ➔ FRAGMENT PROCESSING

Fragment Processing ➔ FRAMEBUFFER IMAGE

Framebuffer Image ➔ DISPLAY

DISPLAY ➔ user sees this

user sees this ➔ DISPLAY

Display ➔ user sees this
OpenGL: One implementation of the graphics pipeline.

And now: a highly abridged and only somewhat accurate history of OpenGL.
OpenGL: The Bad Old Days

- OpenGL was (still is) a state machine.

- Basic usage:

  1. Set flags for shading mode (Lambertian or Blinn-Phong), interpolation methods, depth buffer, ...
  2. Set GL to triangle mode
  3. Send vertices to GPU one at a time.
  4. Call draw function to draw to the screen.
OpenGL: Nowadays

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written vertex shader program on each vertex (to determine its location in clip space) = normalized device coordinates
- GL rasterizes primitives into pixel-shaped fragments
- Execute custom-written fragment shader program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.
You are a user of OpenGL

Your tasks are:

• Send geometry

• Write vertex shader

• Write fragment shader

• Sit back and watch the pretty pictures happen.
OpenGL: Your job, conceptually

(send geometry)
- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, ...)

(write vertex shader)
- GL executes custom-written vertex shader program on each vertex (to determine its location in clip space) = normalized device coordinates

(rasterize)
- GL rasterizes primitives into pixel-shaped fragments

(write fragment shader)
- Execute custom-written fragment shader program on each fragment to determine its color.

- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.
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

https://facultyweb.cs.wwu.edu/~wehrwes/courses/csci480_21w/pipeline_demo/

From drop-down select "constant color without depth".
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 $z'$
Result of z-buffer pipeline

From drop-down select "constant color with depth".
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 $z’ < \text{current } z’$
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 / 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
Some possible efficiency hacks:

- Blinn-Phong model requires knowing
  - normal
  - light direction
  - view direction

- Hack: use directional lights so \( I \) doesn't change

- Hack: pretend viewer is infinitely distant so view direction doesn't change either.
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
  – (demo)
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)
  – in pipeline, this means we are moving illumination from the vertex processing stage to the fragment processing stage
Per-pixel (Phong) shading

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