Computer Graphics

Lecture 23
Shading in the Graphics Pipeline
Announcements

• Class tomorrow in CF 420 - OpenGL Lab

• Artifact voting is open through tomorrow night.
Graphics Pipeline: Overview

- **You are here**: Application
- **Command Stream**: Command Stream
- **Vertex Processing**: 3D transformations; shading
- **Transformed Geometry**: Conversion of primitives to pixels
- **Rasterization**: Rasterization
- **Fragments**: Blending, compositing, shading
- **Fragment Processing**: Fragment processing
- **Framebuffer Image**: Framebuffer image
- **Display**: User sees this
Last time

3D transformations; shading

conversion of primitives to pixels

blending, compositing, shading

user sees this

Backface culling
Clipping
Z buffering
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
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

  2. Set model, view, and projection matrices

  3. Set GL to triangle mode

  4. Send vertices to GPU one at a time.

  5. Call draw function to draw to the screen.

```python
glMatrixMode(GL_PROJECTION); glLoadIdentity();
glMatrixMode(GL_MODELVIEW); glLoadIdentity();
gBegin(GL_TRIANGLES);
gVertex2f(-0.5f, -0.5f);
gVertex2f(0.5f, -0.5f);
gVertex2f(0.5f, 0.5f);
gEnd();
```
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**
- GL executes 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.
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

- GL **rasterizes** primitives into pixel-shaped **fragments**

(write fragment shader)
- GL executes 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_22f/pipeline_demo/
Rendering Realistic Images

• We have a pipeline that gives us access to the compute power of shaders and does a bunch of nice things for us.

• Tomorrow we'll learn how to get data in and out

• How do we realistic-looking images using shading models like Lambertian and Blinn-Phong?
Rendering Realistic Images

• We have a pipeline that gives us access to the compute power of shaders and does a bunch of nice things for us.

• Tomorrow we'll learn how to get data in and out

• How do we realistic-looking images using shading models like Lambertian and Blinn-Phong?

but first, a rant about terminology
Phong shading Blinn-Phong shading in the fragment shader

• Shade (v.): determine color of a pixel
  *basically all of computer graphics...*

• Shader (n.): a program that runs on GPU
  *vertex shader, fragment shader*

• Shading model (*reflection* or *illumination model*):
  light interaction model that determines a pixel's color
  *Lambertian reflection, Blinn-Phong reflection*

• Shading algorithm (*interpolation technique*):
  when, and in which shader, is the reflection model computed, and using what normals?
  *flat shading, Gouraud shading, Phong shading*
Let's call this "not shading"

https://facultyweb.cs.wwu.edu/~wehrwes/courses/csci480_22f/pipeline_demo/
Flat shading (interpolation)

- Shade using the real normal of the triangle
  - same result as ray tracing a bunch of triangles without normal interpolation
- 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
## Summary: Shading and Interpolation Techniques

### Pipeline Stage

<table>
<thead>
<tr>
<th>Interpolation</th>
<th>Vertex Shader</th>
<th>Rasterizer</th>
<th>Fragment Shader</th>
</tr>
</thead>
</table>
| **Flat**      | p, n, l --> cam space  
color = n * l * vtx_color  
p --> screen space |  | Write color if z buffer says so |
| **Gouraud**   | p, n, l --> cam space  
color = n * l * vtx_color  
p --> screen space |  | Write color if z buffer says so |
| **Phong**     | p, n, l --> cam space  
p --> screen space  
pass through vtx_color |  | color = n * l * frag_color  
Write color if z buffer says so |
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
# Summary: Shading and Interpolation Techniques

## Pipeline Stage

<table>
<thead>
<tr>
<th>Interpolation</th>
<th>Vertex Shader</th>
<th>Rasterizer</th>
<th>Fragment Shader</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flat</strong></td>
<td>p, n, l --&gt; cam space&lt;br&gt;color = n * l * vtx_color&lt;br&gt;p --&gt; screen space</td>
<td>Interpolate z'</td>
<td>Write color if z buffer says so</td>
</tr>
<tr>
<td><strong>Gouraud</strong></td>
<td>p, n, l --&gt; cam space&lt;br&gt;color = n * l * vtx_color&lt;br&gt;p --&gt; screen space</td>
<td></td>
<td>Write color if z buffer says so</td>
</tr>
<tr>
<td><strong>Phong</strong></td>
<td>p, n, l --&gt; cam space&lt;br&gt;p --&gt; screen space&lt;br&gt;pass through vtx_color</td>
<td></td>
<td>color = n * l * frag_color&lt;br&gt;Write color if z buffer says so</td>
</tr>
</tbody>
</table>
Some possible efficiency hacks:

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

- Hack: use directional lights sodoesn'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

(*not the same thing as Blinn-Phong reflection)
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
## Summary: Shading and Interpolation Techniques

### Pipeline Stage

<table>
<thead>
<tr>
<th>Interpolation</th>
<th>Vertex Shader</th>
<th>Rasterizer</th>
<th>Fragment Shader</th>
</tr>
</thead>
</table>
| **Flat**      | p, n, l → cam space
   color = n * l * vtx_color
   p → screen space |   | Write color if z buffer says so |
| **Gouraud**   | p, n, l → cam space
   color = n * l * vtx_color
   p → screen space |   | Write color if z buffer says so |
| **Phong**     | p, n, l → cam space
   p → screen space
   pass through vtx_color |   | color = n * l * frag_color
   Write color if z buffer says so |
### Summary: Shading and Interpolation Techniques

<table>
<thead>
<tr>
<th>Interpolation Type</th>
<th>Lambertian</th>
<th>Blinn-Phong</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat</td>
<td><img src="image" alt="Flat Lambertian" /></td>
<td><img src="image" alt="Flat Blinn-Phong" /></td>
</tr>
<tr>
<td>Gouraud</td>
<td><img src="image" alt="Gouraud Lambertian" /></td>
<td><img src="image" alt="Gouraud Blinn-Phong" /></td>
</tr>
<tr>
<td>Phong</td>
<td><img src="image" alt="Phong Lambertian" /></td>
<td><img src="image" alt="Phong Blinn-Phong" /></td>
</tr>
</tbody>
</table>
# Summary: Shading and Interpolation Techniques

## Pipeline Stage

<table>
<thead>
<tr>
<th>Interpolation</th>
<th>Vertex Shader</th>
<th>Rasterizer</th>
<th>Fragment Shader</th>
</tr>
</thead>
</table>
| **Flat**      | p, n, l --> cam space  
color = n * l * vtx_color  
p --> screen space | Interpolate z'  
Pass through color | Write color if z buffer says so |
| **Gouraud**   | p, n, l --> cam space  
color = n * l * vtx_color  
p --> screen space | Interpolate z'  
Interpolate color | Write color if z buffer says so |
| **Phong**     | p, n, l --> cam space  
p --> screen space  
pass through vtx_color | Interpolate z'  
Interpolate vtx_color  
Interpolate normal | color = n * l * frag_color  
Write color if z buffer says so |