Computer Graphics

Lecture/Lab 23
WebGL, continued
Flat, Gouraud, and Phong "shading"
Announcements

• Final project groups due tonight.
  • Still looking for a group? Let's meet up after class.

• HW3 graded
  • If you submit late (today or after), let me know so we can go back and grade it.

• Final project report due Friday
A2 Artifact Results

A 4-way tie for first place!
3rd Place (tie): Sam Burgess
07 - Lucas Binder
12 - Jonathan Derr
3rd Place (tie): Sam Burgess

03 - Carter Schmidt
19 - Raiden Van Bronkhorst
Graphics Pipeline: Overview

you are here ➔ APPLICATION

3D transformations; shading ➔ VERTEX PROCESSING

conversion of primitives to pixels ➔ RASTERIZATION

blending, compositing, shading ➔ FRAGMENT PROCESSING

user sees this ➔ DISPLAY
OpenGL: Your job, conceptually

• Send buffers full of data to GPU up front.

• Tell GL how to interpret them (triangles, ...)

• 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.
OpenGL: Your job, conceptually

(send geometry)

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

**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.
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

(write fragment shader)
- 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.
Last time: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader in **GLSL**, the GL shader language
- Write a fragment shader
Last time: Hello, Triangle!

- Send geometry by calling gl functions
- Write a vertex shader in GLSL, the GL shader language
- Write a fragment shader
Shader Responsibilities

The **vertex shader's job** is to:

- assign a value to `gl_Position`, which specifies the vertex's position
- assign values to any **varying** parameters needed later

The **fragment shader's job** is to:

- assign a value to `gl_FragColor`, which specifies the fragment's color
Shader Responsibilities

The **vertex shader's job** is to:

- assign a value to `gl_Position`, which specifies the vertex's position
- assign values to any **varying** parameters needed later

Lab code so far:

```glsl
gl_Position = vec4(Position, 1.0)
```

The **fragment shader's job** is to:

- assign a value to `gl_FragColor*`, which specifies the fragment's color
Shader Responsibilities

The **vertex shader's job** is to:

- assign a value to `gl_Position`, which specifies the vertex's position
- assign values to any **varying** parameters needed later

```
Lab code so far:  
gl_Position = vec4(Position, 1.0)
```

The **fragment shader's job** is to:

- assign a value to `gl_FragColor*`, which specifies the fragment's color

```
Lab code so far:  
gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0)
```
WebGL Data Plumbing: Overview

See also: Tuesday lecture notes
WebGL Data Plumbing

application

vertices

uniform variables

rasterizer

fragment program

depth

framebuffer

See also: Tuesday lecture notes
WebGL Data Plumbing

application

vertex program

uniform variables

rasterizer

fragment program

framebuffer

sent in vertex buffers

See also: Tuesday lecture notes
WebGL Data Plumbing

Application

Vertex Program

Fragment Program

Framebuffer

Uniform Variables

Triangles

Attributes

Varying Parameters

Rasterizer

Depth

Color

Sent in an index buffer

Sent in vertex buffers

See also: Tuesday lecture notes
WebGL Data Plumbing

- uniform variables
- triangles
- attributes
- vertex program
- rasterizer
- fragment program
- framebuffer
- depth
- color

See also: Tuesday lecture notes
GLSL - GL Shader Language

• Built-in types for small vectors/matrices (e.g., \texttt{vec3}, \texttt{mat4}). They have friendly constructors:

\[
\begin{align*}
\text{vec3 } a &= \text{vec3}(1.0, 1.0, 1.0) \\
\text{vec4 } b &= \text{vec4}(a, 1.0)
\end{align*}
\]

• Multiplication does matrix multiplication:

\[
\begin{align*}
\text{mat2 } A &= \text{mat2}(1.0, 2.0, 3.0, 4.0) \\
\text{vec2 } x &= \text{vec2}(1.0, 0.0)
\end{align*}
\]
\[
\text{vec2 } a = A \times x; \quad // \quad a = (1,2)
\]
Task 2: Add a uniform

- Add a uniform variable called Matrix containing a 4x4 matrix

- In the vertex shader, multiply the Position attribute of the vertex by the Matrix to move the triangle vertices.
Terminology: data plumbing

See also: Tuesday lecture notes
GLSL - GL Shader Language

• *varyings* are declared in both the Vertex shader and in the Fragment shader.

  • The vertex shader sets their values for each vertex, then the rasterizer *interpolates* their values for each fragment and passes to the fragment shader.

• By convention, *varying* names are usually chosen to begin with *v*, such as *vColor* or *vNormal*.
Task 3: Add a varying

- Set up a varying parameter to set the color at each vertex

- Use the interpolated values in the fragment shader to set each fragment's color.
Rendering Realistic Images
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.
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.

• We know how to get data in and out
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.

• We know 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.

• We know 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 Lambertian shading in the fragment shader
Phong shading Lambertian shading in the fragment shader

- Shade (v.): determine color of a pixel
Phong shading Lambertian shading in the fragment shader

• Shade (v.): determine color of a pixel

  basically all of computer graphics...
Phong shading Lambertian shading in the fragment shader

- Shade (v.): determine color of a pixel
- Shader (n.): a program that runs on GPU
Phong shading Lambertian 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
Phong shading Lambertian 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
Phong shading Lambertian 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*
Phong shading Lambertian 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?
Phong shading Lambertian 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*
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
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”
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 \( \mathbf{l} \) doesn't change

- Hack: pretend viewer is infinitely distant so view direction doesn't change either.
Some possible efficiency hacks:

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

• Hack: use directional lights so light 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

(*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

<table>
<thead>
<tr>
<th>Interpolation</th>
<th>Lambertian</th>
<th>Blinn-phong</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gouraud</td>
<td><img src="image1" alt="Gouraud Lambertian" /></td>
<td><img src="image2" alt="Gouraud Blinn-phong" /></td>
</tr>
<tr>
<td>Phong</td>
<td><img src="image3" alt="Phong Lambertian" /></td>
<td><img src="image4" alt="Phong Blinn-phong" /></td>
</tr>
</tbody>
</table>

**reflection**