Announcements

- Homework project #4 due Friday, November 2\textsuperscript{nd}
  - Introduction: Oct 29 at 2:30pm

- This Friday:
  - Late grading for project #3
  - Early grading for project #4

- Midterm exam: Thursday, Oct 25, 2-3:20pm in classroom

- Midterm compatible index cards provided today and in instructor’s office

- Reminder: Ted forums exist to discuss homework assignments and midterms
Review

- OpenGL’s Shading Model
Complete Blinn-Phong Shading Model

- Blinn-Phong model with several light sources $I$
- All colors and reflection coefficients are vectors with 3 components for red, green, blue

$$c = \sum_i c_{li} \left( k_d \left( L_i \cdot n \right) + k_s \left( H_i \cdot n \right)^s \right) + k_a c_a$$

diffuse + specular + ambient =
BRDFs

- Diffuse, Phong, Blinn models are instances of bidirectional reflectance distribution functions (BRDFs).
- For each pair of light directions $\mathbf{L}$ and viewing direction $\mathbf{e}$, the BRDF returns the fraction of reflected light.
- Shading with general BRDF $f$

$$
c = \sum_i c_{li} f(\mathbf{L}_i, \mathbf{e})
$$

- Many other forms of BRDFs exist in graphics, often named after inventors: Cook-Torrance, Ward, etc.
Lecture Overview

- **OpenGL Light Sources**
- **Types of Geometry Shading**
- **Shading in OpenGL**
  - Fixed-Function Shading
  - Programmable Shaders
    - Vertex Programs
    - Fragment Programs
  - GLSL
Light Sources

- Real light sources can have complex properties
  - Geometric area over which light is produced
  - Anisotropy (directionally dependent)
  - Reflective surfaces act as light sources (indirect light)

- OpenGL uses a drastically simplified model to allow real-time rendering
OpenGL Light Sources

- At each point on surfaces we need to know
  - Direction of incoming light (the $L$ vector)
  - Intensity of incoming light (the $c_l$ values)

- Standard light sources in OpenGL
  - **Directional**: from a specific direction
  - **Point light source**: from a specific point
  - **Spotlight**: from a specific point with intensity that depends on direction
Directional Light

- Light from a distant source
  - Light rays are parallel
  - Direction and intensity are the same everywhere
  - As if the source were infinitely far away
  - Good approximation of sunlight
- Specified by a unit length direction vector, and a color

\[ \mathbf{L} = -\mathbf{d} \]

\[ c_l = c_{src} \]
Point Lights

- Similar to light bulbs
- Infinitely small point radiates light equally in all directions
  - Light vector varies across receiving surface
  - What is light intensity over distance proportional to?
  - Intensity drops off proportionally to the inverse square of the distance from the light
    - Reason for inverse square falloff:
      Surface area $A$ of sphere:
      $$ A = 4 \pi r^2 $$
Point Lights in Theory

At any point $v$ on the surface:

$$L = \frac{p - v}{||p - v||}$$

$$c_l = \frac{c_{src}}{||p - v||^2}$$
Point Lights in OpenGL

- OpenGL model for distance attenuation:

\[
C_l = \frac{C_{src}}{k_c + k_l |p - v| + k_q |p - v|^2}
\]

- Attenuation parameters:
  - \(k_c\) = constant attenuation, default: 1
  - \(k_l\) = linear attenuation, default: 0
  - \(k_q\) = quadratic attenuation, default: 0

- Default: no attenuation: \(C_l = C_{src}\)

- Change attenuation parameters with:
  - `GL_CONSTANT_ATTENUATION`
  - `GL_LINEAR_ATTENUATION`
  - `GL_QUADRATIC_ATTENUATION`
Spotlights

- Like point source, but intensity depends on direction

Parameters

- Position: location of the light source
- Spot direction: center axis of the light source
- Falloff parameters:
  - Beam width (cone angle)
  - The way the light tapers off at the edges of the beam (cosine exponent)
Spotlights

Light source

Receiving surface

\[ L = \frac{p - v}{\| p - v \|} \]

\[ c_l = \begin{cases} 
0 & \text{if } -L \cdot d \leq \cos(\theta_{max}) \\
\frac{c_{src} (-L \cdot d)^f}{\| p - v \|} & \text{otherwise}
\end{cases} \]
Spotlights

Photograph of real spotlight  Spotlights in OpenGL
Lecture Overview

- OpenGL Light Sources
- **Types of Geometry Shading**
- Shading in OpenGL
  - Fixed-Function Shading
  - Programmable Shaders
    - Vertex Programs
    - Fragment Programs
  - GLSL
Types of Geometry Shading

- Per-triangle
- Per-vertex
- Per-pixel
Per-Triangle Shading

- Known as *flat shading*
- Evaluate shading once per triangle

**Advantage**
- Fast

**Disadvantage**
- Faceted appearance
Per-Vertex Shading

- Known as *Gouraud shading* (Henri Gouraud, 1971)
- Interpolates vertex colors across triangles with Barycentric Interpolation

**Advantages**
- Fast
- Smoother surface appearance than with flat shading

**Disadvantage**
- Problems with small highlights
Per-Pixel Shading

- Also known as Phong Interpolation (not to be confused with Phong Illumination Model)
  - Rasterizer interpolates normals (instead of colors) across triangles
  - Illumination model is evaluated at each pixel
  - Simulates shading with normals of a curved surface

- Advantage
  - Higher quality than Gouraud shading

- Disadvantage
  - Slow

Source: Penny Rheingans, UMBC
Gouraud vs. Per-Pixel Shading

- Gouraud has problems with highlights
- More triangles would improve result, but reduce frame rate
Lecture Overview

- OpenGL Light Sources
- Types of Geometry Shading
- Shading in OpenGL
  - Fixed-Function Shading
  - Programmable Shaders
    - Vertex Programs
    - Fragment Programs
    - GLSL
Shading with Fixed-Function Pipeline

- Fixed-function pipeline only allows Gouraud (per-vertex) shading
- We need to provide a normal vector for each vertex
- Shading is performed in camera space
  - Position and direction of light sources are transformed by GL_MODELVIEW matrix
- If light sources should be in object space:
  - Set GL_MODELVIEW to desired object-to-camera transformation
  - Use object space coordinates for light positions
- More information:
  - http://glprogramming.com/red/chapter05.html
Tips for Transforming Normals

- If you need to (manually) transform geometry by a transformation matrix $\mathbf{M}$, which includes shearing or scaling:
  - Transforming the normals with $\mathbf{M}$ will not work: transformed normals are no longer perpendicular to surfaces

- **Solution:** transform the normals differently:
  - Either transform the end points of the normal vectors separately
  - Or transform normals with $\mathbf{M}^{-1}T$

- **Find derivation on-line at:**

- **OpenGL does this automatically if the following command is used:**
  - glEnable(GL_NORMALIZE)
Lecture Overview

- OpenGL Light Sources
- Types of Geometry Shading
- Shading in OpenGL
  - Fixed-Function Shading
  - **Programmable Shaders**
    - Vertex Programs
    - Fragment Programs
    - GLSL
Programmable Shaders in OpenGL

- Initially, OpenGL only had a fixed-function pipeline for shading
- Programmers wanted more flexibility, similar to programmable shaders in raytracing software (term “shader” first introduced by Pixar in 1988)
- First shading languages came out in 2002:
  - **Cg** (C for Graphics, created by Nvidia)
  - **HLSL** (High Level Shader Language, created by Microsoft)
- **They supported:**
  - **Fragment shaders**: allowed per-pixel shading
  - **Vertex shaders**: allowed modification of geometry
- OpenGL 2.0 supported the OpenGL Shading Language (GLSL) in 2003
- **Geometry shaders** were added in OpenGL 3.2
- **Tessellation shaders** were added in OpenGL 4.0
- Programmable shaders allow real-time:
  - Shadows, environment mapping, per-pixel lighting, bump mapping, parallax bump mapping, HDR, etc., etc.
Demo

- NVIDIA Froggy

- Features
  - Bump mapping shader for Froggy’s skin
  - Physically-based lighting model simulating sub-surface scattering
  - Supersampling for scene anti-aliasing
  - Raytracing shader for irises to simulate refraction for wet and shiny eyes
  - Dynamically-generated lights and shadows
Shader Programs

- Programmable shaders consist of shader programs
- Written in a shading language
  - Syntax similar to C language
- Each shader is a separate piece of code in a separate ASCII text file
- Shader types:
  - Vertex shader
  - Tessellation shader
  - Geometry shader
  - Fragment shader (a.k.a. pixel shader)
- The programmer can provide any number of shader types to work together to achieve a certain effect
- If a shader type is not provided, OpenGL’s fixed-function pipeline is used
Programmable Pipeline

Scene

- Executed once per vertex:
  - Vertex Shader
  - Tessellation Shader
  - Geometry Shader

Modeling and viewing transformation

- Shading

Projection

Rasterization

Fragment processing

Frame-buffer access (z-buffering)

Image

- Executed once per fragment:
  - Fragment Shader
Vertex Shader

- Executed once per vertex
- Cannot create or remove vertices
- Does not know the primitive it belongs to
- Replaces functionality for
  - Model-view, projection transformation
  - Per-vertex shading
- If you use a vertex program, you need to implement behavior for the above functionality in the program!

Typically used for:
- Character animation
- Particle systems
Tessellation Shader

- Executed once per primitive
- Generates new primitives by subdividing each line, triangle or quad primitive
- Typically used for:
  - Adapting visual quality to the required level of detail
    - For instance, for automatic tessellation of Bezier curves and surfaces
  - Geometry compression: 3D models stored at coarser level of resolution, expanded at runtime
  - Allows detailed displacement maps for less detailed geometry
Geometry Shader

- Executed once per primitive (triangle, quad, etc.)
- Can create new graphics primitives from output of tessellation shader (e.g., points, lines, triangles)
  - Or can remove the primitive
- Typically used for:
  - Per-face normal computation
  - Easy wireframe rendering
  - Point sprite generation
  - Shadow volume extrusion
  - Single pass rendering to a cube map
  - Automatic mesh complexity modification (depending on resolution requirements)
Fragment Shader

- A.k.a. Pixel Shader
- Executed once per fragment
- Cannot access other pixels or vertices
  - Makes execution highly parallelizable
- Computes color, opacity, z-value, texture coordinates
- Typically used for:
  - Per-pixel shading (e.g., Phong shading)
  - Advanced texturing
  - Bump mapping
  - Shadows
Lecture Overview

- OpenGL Light Sources
- Types of Geometry Shading
- Shading in OpenGL
  - Fixed-Function Shading
  - Programmable Shaders
    - Vertex Programs
    - Fragment Programs
  - GLSL
Vertex Programs

Vertex Attributes

From Application

Uniform Parameters

Vertex program

To Rasterizer

Output Variables
Vertex Attributes

- Declared using the `attribute` storage classifier
- Different for each execution of the vertex program
- Can be modified by the vertex program
- Two types:
  - Pre-defined OpenGL attributes. Examples:
    ```
    attribute vec4 gl_Vertex;
    attribute vec3 gl_Normal;
    attribute vec4 gl_Color;
    ```
  - User-defined attributes. Example:
    ```
    attribute float myAttrib;
    ```
Uniform Parameters

- Declared by `uniform storage classifier`
- Normally the same for all vertices
- Read-only
- Two types:
  - Pre-defined OpenGL state variables
  - User-defined parameters
Uniform Parameters: Pre-Defined

- Provide access to the OpenGL state
- Examples for pre-defined variables:
  
  ```
  uniform mat4 gl_ModelViewMatrix;
  uniform mat4 gl_ModelViewProjectionMatrix;
  uniform mat4 gl_ProjectionMatrix;
  uniform gl_LightSourceParameters
      gl_LightSource[gl_MaxLights];
  ```
Uniform Parameters: User-Defined

- Parameters that are set by the application
- Should not be changed frequently
  - Especially not on a per-vertex basis!
- To access, use `glGetUniformLocation`, `glUniform*` in application

Example:

- In shader declare
  
  ```
  uniform float a;
  ```

- Set value of `a` in application:
  
  ```
  GLuint p;
  int I = glGetUniformLocation(p,"a");
  glUniform1f(i, 1.0f);
  ```
Vertex Programs: Output Variables

- **Required output:** homogeneous vertex coordinates
  `vec4 gl_Position`

- **varying output variables**
  - Mechanism to send data to the fragment shader
  - Will be interpolated during rasterization
  - Fragment shader gets interpolated data

- **Pre-defined varying output variables, for example:**
  `varying vec4 gl_FrontColor;`  
  `varying vec4 gl_TexCoord[];`
  Any pre-defined output variable that you do not overwrite will have the value of the OpenGL state.

- **User-defined varying output variables, e.g.:**
  `varying vec4 vertex_color;`
Lecture Overview

- OpenGL Light Sources
- Types of Geometry Shading
- Shading in OpenGL
  - Fixed-Function Shading
  - Programmable Shaders
    - Vertex Programs
    - Fragment Programs
    - GLSL
Fragment Programs

Fragment Data

From Rasterizer

Fragment program

Uniform Parameters

To Frame Buffer

Output Variables
Fragment Data

- Changes for each execution of the fragment program
- Fragment data includes:
  - Interpolated standard OpenGL variables for fragment shader, as generated by vertex shader, for example:
    ```
    varying vec4 gl_Color;
    varying vec4 gl_TexCoord[];
    ```
  - Interpolated varying variables from vertex shader
    - Allows data to be passed from vertex to fragment shader
Uniform Parameters

- Same as in vertex programs
Output Variables

- Pre-defined output variables:
  - `gl_FragColor`
  - `gl_FragDepth`

- OpenGL writes these to the frame buffer
- Result is undefined if you do not set these variables!
Lecture Overview

- OpenGL Light Sources
- Types of Geometry Shading
- Shading in OpenGL
  - Fixed-Function Shading
  - Programmable Shaders
    - Vertex Programs
    - Fragment Programs
    - GLSL
GLSL Main Features

- Similar to C language
- `attribute, uniform, varying` `storage classifiers`
- Set of predefined variables
  - Access to per-vertex, per-fragment data
  - Access OpenGL state
- Built-in vector data types, vector operations
- No pointers
- No direct access to data or variables in your C++ code
Example: Treat normals as colors

// Vertex Shader
varying vec4 color;

void main()
{
    // Treat the normal (x, y, z) values as (r, g, b) color components.
    color = vec4(clamp(abs((gl_Normal + 1.0) * 0.5), 0.0, 1.0), 1.0);

    gl_Position = ftransform();
}

// Fragment Shader
varying vec4 color;

void main()
{
    gl_FragColor = color;
}
Creating Shaders in OpenGL

Source: Gabriel Zachmann, Clausthal University
Tutorials and Documentation

- OpenGL and GLSL specifications
  - http://www.opengl.org/documentation/specs/
- GLSL tutorials
  - http://www.lighthouse3d.com/opengl/glsl/
- OpenGL Programming Guide (Red Book)
- OpenGL Shading Language (Orange Book)
- OpenGL API Reference Card