Announcements

• Find homework scores on Gradesource

• Homework Project 3 due this Friday, Oct 23, demo in lab 250 between 2-5pm

• Homework late submission policy:
  - This Friday accepting late submissions of project 2

• Midterm exam: Thursday, Oct 29

• No lab hours next Friday! If you need to get graded late you must submit by next Thursday.
Midterm Exam

• Midterm exam Thursday Oct 29 during lecture slot, 2-3:20pm
• Use previous exams on course web site to study
• No homework project for next Friday!
• Midterm tutorial with Jason:
  - Mon Oct 26, 2-5pm, Atkinson Hall room 4004
  - Walk in/out at any time
• TAs will answer questions about midterm during all lab hours next week
Shading

- Computes interaction of light with surfaces
- Interactive applications
  - Local illumination
  - Simplified model for reflection at surfaces
  - Sum of 3 components
- Covers a large class of real surfaces

\[
\text{diffuse} + \text{specular} + \text{ambient} = \text{result}
\]
Diffuse reflection

- Ideal diffuse material reflects light equally in all directions

- View-independent

- Matte, not shiny materials
  - Paper
  - Unfinished wood
  - Unpolished stone

- Example: the moon
Specular reflection

- Mirror-like reflection
  - Glossy materials
- Shading models:
  - Phong
  - Blinn

Glossy teapot
**Ambient light**

- In real world, light is bounced all around scene
- Could use global illumination techniques to simulate
- Simple approximation
  - Add constant ambient light at each point $k_a c_a$
  - Ambient light $c_a$
  - Ambient reflection coefficient $k_a$
- Without ambient light, areas with no direct illumination would be completely dark/black
**Complete model**

- Blinn model with several light sources $i$

\[ c = \sum_{i} c_{li} (k_{d} (L_{i} \cdot n) + k_{s} (h_{i} \cdot n)^{g}) + k_{a} c_{a} \]

- diffuse
- specular
- ambient
Today

Shading

- Light sources
- Shader programming
Light sources

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

- Interactive rendering is based on simple, standard light sources
Light sources

• At each point on surfaces we need to know
  - Direction of incoming light (the $\mathbf{L}$ 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 the 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

\[ L = -d \]
\[ c_l = c_{src} \]
Point lights

- Simple model for light bulbs
- Point that radiates light in all directions equally
  - Light vector varies across the surface
  - 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

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

\[
c_l = \frac{c_{src}}{||\mathbf{p} - \mathbf{v}||^2}
\]

Light source

Receiving surface
Attenuation

- Sometimes, it is desirable to modify the inverse square falloff behavior of point lights
  - Common (OpenGL) model for distance attenuation
    \[ c_l = \frac{c_{src}}{k_c + k_l |p - v| + k_q |p - v|^2} \]
  - Not physically accurate
Spotlights

- Like point source, but intensity depends on direction

Parameters

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

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

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

Photograph of spotlight

Spotlights in OpenGL
Per-triangle, -vertex, -pixel shading

- Shading operations
  - Once per triangle
  - Once per vertex
  - Once per pixel

Scene data

Modeling and viewing transformation

Shading

Projection

Rasterization, visibility

Image
Per-triangle shading

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

- **Advantages**
  - Fast

- **Disadvantages**
  - Faceted appearance
Per-vertex shading

- Known as *Gouraud shading* (Henri Gouraud 1971)
- Interpolate vertex colors across triangles
- OpenGL default

Advantages
- Fast
- Smoother than flat shading

Disadvantages
- Problems with small highlights
Per-pixel shading

- Also known as Phong interpolation (not to be confused with Phong illumination model)
  - Rasterizer interpolates normals across triangles
  - Illumination model evaluated at each pixel
  - Implemented using fragment shaders (later today)

- Advantages
  - Higher quality than Gouraud shading

- Disadvantages
  - Much slower
Gouraud vs. per-pixel shading

- Gouraud has problems with highlights
- More triangles would improve result, but impact frame rate
Shading in OpenGL

// Somewhere in the initialization part of your program...
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

// Make sure vertex colors are used as material properties
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glColorMaterial(GL_FRONT, GL_SPECULAR);

// Create light components
GLfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
GLfloat diffuseLight[] = { 0.8f, 0.8f, 0.8, 1.0f };
GLfloat specularLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };

// Assign created components to GL_LIGHT0
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
glLightfv(GL_LIGHT0, GL_POSITION, position);
Shading in OpenGL

// Somewhere in the initialization part of your // program...
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

// Make sure vertex colors are used as material properties
glEnable(GL_COLOR_MATERIAL);

glColorMaterial(GL_FRONT, GL_DIFFUSE);
glColorMaterial(GL_FRONT, GL_SPECULAR);

// Create light components
GLfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
GLfloat diffuseLight[] = { 0.8f, 0.8f, 0.8f, 1.0f };
GLfloat specularLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };

// Assign created components to GL_LIGHT0
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);

Shading computations (diffuse, specular, ambient) performed automatically (unless you use shader programs)
Shading in OpenGL

- Need to provide per vertex normals
- Shading is performed in camera space
  - Position, direction of light sources is transformed by GL_MODELVIEW matrix
- If light sources should be fixed relative to objects
  - Set GL_MODELVIEW to desired object-to-camera transform
  - Choose object space coordinates for light position
  - Will be transformed using current GL_MODELVIEW
- Lots of details, highly recommend OpenGL programming guide
  - http://glprogramming.com/red/chapter05.html
Transforming normals

- If the object-to-camera transformation $M$ includes shearing or non-uniform scaling, transforming normals using $M$ does not work:
  - Transformed normals are not perpendicular to surfaces any more

- To avoid the problem, we need to transform the normals differently:
  - by transforming the end points of the normal vectors separately
  - or using $M^{-1}T$

- Find derivation on-line at:

- OpenGL does this automatically for us on the GPU
Today

Shading

• Light sources

• Shader programming
Configurable pipeline

Before 2002

- APIs (OpenGL, Direct3D) to configure the rendering pipeline
- Enable/disable functionality
  - E.g., lighting, texturing
- Set parameters for given functionality
  - E.g., light direction, texture blending mode
Configurable pipeline

Disadvantages

• Restricted to preset functionality
  - Limited types of light sources (directional, point, spot)
  - Limited set of reflection models (ambient, diffuse, Phong)
  - Limited use of texture maps

• More flexibility desired for more photorealistic effects
Programmable pipeline

- Replace functionality in parts of the pipeline by user specified programs
- Called shaders, or shader programs
- Not all functionality in the pipeline is programmable
Shader programs

- Written in a shading language
  - Cg: early shading language by NVidia
  - Shading languages today:
    - GLSL for OpenGL (GL shading language)
    - HLSL for DirectX (high level shading language)
  - Similar to C

- Novel, quickly changing technology
- Driven by more and more flexible GPUs
Two types of shader programs

1. Vertex program

2. Fragment program
   (fragment: pixel location inside a triangle and interpolated data)
GPU architecture

Pipeline
- Scene data
  - Modeling and viewing transformation
    - Shading
  - Projection
  - Rasterization
  - Fragment processing
    - Frame-buffer access (z-buffering)
- Image

GPU Architecture
NVidia NV80 (GeForce 8800 Series)
- 128 stream processors

Programmable pipeline (2006)

Not programmable:

• Projective division

• Rasterization
  - Determination of which pixels lie inside a triangle
  - Vertex attribute interpolation (color, texture coordinates)

• Access to frame buffer
  - Texture filtering
  - Z-buffering
  - Frame buffer blending
Shader programming

- Application programmer can provide:
  - No shaders, standard OpenGL functions are executed
  - Vertex shader only
  - Fragment shader only
  - Vertex and fragment shaders

- Each shader is a separate piece of code in a separate text file

- Output of vertex shader is interpolated at each fragment and accessible as input to fragment shader
Vertex programs

- Executed once for every vertex
- Replaces functionality for
  - Modelview, projection transformation
  - Per-vertex shading
- If you use a vertex program, you need to implement this functionality in the program
- Vertex shader often used for animation
  - Characters
  - Particle systems
Fragment programs

- Executed once for every fragment
- Implements functionality for
  - Texturing
  - Per pixel effects
  - Per pixel shading
  - Bump mapping
  - Shadows
  - Blending
  - Look-up tables
  - Etc.
Creating shaders in OpenGL

[OpenGL programming guide]
**Vertex programs**

**Vertex attributes**
Coordinates in object space, additional vertex attributes

From application

**Uniform parameters**
OpenGL state, application specified parameters

**Vertex program**

To rasterizer

Transformed vertices, processed vertex attributes
Types of input data

- **Vertex attributes**
  - Change for each execution of the vertex program
  - Predefined OpenGL attributes (color, position, etc.)
  - User defined attributes

- **Uniform parameters**
  - Normally the same for all vertices
  - OpenGL state variables
  - Application defined parameters
**Vertex attributes**

- “Data that flows down the pipeline with each vertex”
- Per-vertex data that your application specifies
- E.g., vertex position, color, normal, texture coordinates
- Declared using `attribute storage` classifier in your shader code
  - Read-only
Vertex attributes

- OpenGL vertex attributes accessible through **predefined** variables

  ```
  attribute vec4 gl_Vertex;
  attribute vec3 gl_Normal;
  attribute vec4 gl_Color;
  ```

  etc.

- Optional user defined attributes
OpenGL state variables

- Provide access to state of rendering pipeline, which you set through OpenGL calls in application

- Predefined variables

  ```
  uniform mat4 gl_ModelViewMatrix;
  uniform mat4 gl_ModelViewProjectionMatrix;
  uniform mat4 gl_ProjectionMatrix;
  uniform gl_LightSourceParameters
     gl_LightSource[gl_MaxLights];
  ```

  etc.

- Declared using `uniform` storage classifier
  - Read-only
Uniform parameters

- Parameters that are set by the application
- Should not change frequently
  - Not on a per-vertex basis!
- Will be the same for each vertex until application changes it again
- Declared using `uniform storage classifier`
  - Read-only
Uniform parameters

• To access, use `glGetUniformLocation` and `glUniform*` in application.

• Example
  - In shader declare
    ```
    uniform float a;
    ```
  - In application, set `a` using
    ```
    GLuint p;
    //... initialize program p
    int i = glGetUniformLocation(p, "a");
    glUniform1f(i, 1.0f);
    ```
Output variables

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

- **varying outputs**
  - Mechanism to send data to the fragment shader
  - Will be interpolated during rasterization
  - Interpolated values accessible in fragment shader (using same variable name)

- **Predefined varying outputs**
  ```
  varying vec4 gl_FrontColor;
  varying vec4 gl_TexCoord[];
  ```
  etc.

- **User defined varying outputs**
Next Lecture

- Fragment shaders
- Texture mapping