CSE167: Introduction to Computer Graphics

Lecture #8

Jürgen Schulze
University of California, San Diego
Fall 2008
Project 3, Midterm Exam

• Project 3 due this Friday

• Late policy:
  - This Friday accepting late submissions of project 2
  - Project 3+: Late credit only on Monday after due date

• Midterm exam Thursday Oct 30 during lecture
  - Use previous exams on course web site to study

• No homework project for next Friday!

• Midterm Q&A session:
  - Fri Oct 24, 4-5:30pm, Atkinson Hall room 4004
  - Okay to walk in/out at any time

• Additional session with Robert next week:
  - Wed at 2pm, lab 250
Today

Shading

• Light sources

• Shader programming
Shading

- Compute interaction of light with surfaces
- Interactive applications
  - Local illumination
  - Simplified models reflection at 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
Diffuse reflection

- Lambert’s cosine law
- Lambertian surfaces

- Example: the moon
Specular reflection

- Mirror like reflection
  - Glossy materials

Glossy teapot
Phong model

- Specular reflectance coefficient $k_s$
- Phong exponent $p$
  - Higher $p$, smaller (sharper) highlight

\[ c = k_s c_l (\mathbf{R} \cdot \mathbf{e})^p \]
Blinn model (Jim Blinn, 1977)

- Define unit halfway vector
  \[ h = \frac{L + e}{||L + e||} \]

- Halfway vector represents normal of micro-facet that would lead to mirror reflection to the eye
Blinn model

- The larger the angle between micro-facet orientation and normal, the less reflection
- Use cosine of angle between them
- Shininess parameter $s$
- Very similar to Phong model

\[ c = k_s c_l \left( \mathbf{h} \cdot \mathbf{n} \right)^s \]
Local illumination

Simplified model

- Sum of 3 components
- Covers a large class of real surfaces

\[ \text{diffuse} + \text{specular} + \text{ambient} = \text{total illumination} \]
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 $k_a$
- Areas with no direct illumination are not completely dark
Complete model

- Blinn model with several light sources $i$

\[ c = \sum_{i} c_{i} (k_{d} (\mathbf{L}_{i} \cdot \mathbf{n}) + k_{s} (\mathbf{h}_{i} \cdot \mathbf{n})^{g}) + k_{a} c_{a} \]

- diffuse
- specular
- ambient
Questions?
Light sources

- Light sources can have complex properties
  - Geometric area over which light is produced
  - Anisotropy in direction
  - 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 $L$ 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

Light source

Receiving surface

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

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

Let $p$ be the light source, $d$ be the direction to the receiving surface, and $\theta_{\text{max}}$ be the maximum angle of acceptance. The vector $L$ is defined as:

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

The component $c_l$ is given by:

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

Photograph of spotlight

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

- May compute shading operations
  - Once per triangle
  - Once per vertex
  - Once per pixel
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
- Could use more triangles...
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);

// 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 object-to-camera transformation $M$ includes shearing, transforming normals using $M$ does not work
  - Transformed normals are not perpendicular to surface any more

• To avoid problem, need to transform normals by $M^{-1T}$

• No derivation here, but remember for rotations $R^{-1T} = R$

• OpenGL does this automatically for us
Questions?
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 reflectance 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
• Recent, quickly changing technology
• Driven by more and more flexible GPUs
Programmable pipeline (2006)

Scene data

Modeling and viewing transformation

Shading

Projection

Rasterization

Fragment processing

Frame-buffer access (z-buffering)

Image

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)

Still fixed functionality

- 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 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

• 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]
Questions?
Vertex programs

Vertex attributes
Coordinates in object space, additional vertex attributes

Uniform parameters
OpenGL state, application specified parameters

Vertex program

From application

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
“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

```cpp
attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;
attribute vec4 gl_Color;
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`, `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.f);
    ```
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**
Output variables

Note

- Any predefined output variable that you do not write will assume the value of the current OpenGL state

- E.g., your vertex shader does not write varying vec4 gl_TexCoord[]
  - Your fragment shader may still read it
  - The value will be the current OpenGL state
“Hello world” vertex program

• main() function is executed for every vertex

• Use predefined variables

```c
void main()
{
    gl_Position = // required output
    gl_ProjectionMatrix * // predefined uniform
    gl_ModelViewMatrix * // predefined uniform
    gl_Vertex; // predefined attribute
}

• Alternatively, use
gl_ModelViewProjectionMatrix or ftransform()
```
Vertex programs

Current limitations

- Cannot write data to memory accessible by application
- Cannot pass data between vertices
  - Each vertex is independent
- For each incoming vertex there is one outgoing vertex
  - Cannot generate new geometry
Examples

• Animation
  - Character skinning
  - Particle systems
  - Water
Fragment programs

Fragment data
Interpolated vertex attributes, additional fragment attributes

From rasterizer

Fragment program

Fragment color, depth

To fixed framebuffer access functionality (z-buffering, etc.)

Uniform parameters
OpenGL state, application specified parameters

Primitives
Modeling and viewing transformation
Shading
Projection
Rasterization
Fragment processing
Frame-buffer access (z-buffering)
Image
Types of input data

• Fragment data
  - Change for each execution of the fragment program
  - Interpolated from vertex data during rasterization, *varying* variables
  - User defined *varying* variables
  - Predefined OpenGL *varying* variables

• Uniform parameters
  - Do not change from fragment to fragment
  - OpenGL state variables
  - Application defined parameters
Fragment data

- E.g., interpolated fragment color, texture coordinates
- Standard OpenGL fragment data accessible through predefined variables
  ```glsl
varying vec4 gl_Color;
varying vec4 gl_TexCoord[];
```
  etc.
- Note varying storage classifier, read-only
- Optional user defined data
Uniform parameters

• Same as in vertex shader
• OpenGL state
• Application defined parameters
  - Use `glGetUniformLocation`, `glUniform*` in application
Output variables

- Predefined outputs
  - `gl_FragColor`
  - `gl_FragDepth`
- Result is undefined if you don’t write these variables
“Hello world” fragment program

- `main()` function is executed for every fragment
- Use predefined variables
- Example draws everything in green

```c
void main()
{
    gl_FragColor = vec4(0.0,1.0,0.0,1.0);
}
```
Examples

• Fancy per pixel shading
  - Bump mapping
  - Displacement mapping
  - Realistic reflection models
  - Cartoon shading
  - Shadows
  - Etc.

• Most often, vertex and fragment shader work together to achieve desired effect
Fragment programs

Current limitations

• Cannot read frame buffer
• Can only write to frame buffer pixel that corresponds to fragment being processed
  - No random write access to frame buffer
• Number of \textit{varying} variables passed from vertex to fragment shader is limited
• Number of application-defined uniform parameters is limited
Summary

• Shader programs replace part of the rendering pipeline

• Written in special shading language (GLSL in OpenGL)

• Sequence of OpenGL calls to compile/activate shaders

• Two types of shaders
  - Vertex shaders
  - Fragment shaders
GLSL main features

• Similar to C

• *attribute*, *uniform*, *varying*
  storage classifiers

• Set of predefined variables to access OpenGL state

• Built in vector data types, vector operations

• No pointers
Questions?
Tutorials and documentation

- OpenGL and GLSL specifications
  http://www.opengl.org/documentation/specs/

- GLSL tutorials
  http://www.lighthouse3d.com/opengl/gls1/
  http://www.clockworkcoders.com/oglsl/tutorials.html

- OpenGL Programming Guide (red book)
- OpenGL Shading Language (orange book)
Shader development tools

- ATI/AMD render monkey (OpenGL and DirectX shaders)

- NVidia FX composer (DirectX shaders only)
  http://developer.nvidia.com/object/fx_composer_home.html
Next time

- Texture mapping