CS4621/5621 Fall 2015

Basics of OpenGL/GLSL
Shading and Lighting

Professor: Kavita Bala
Instructor: Nicolas Savva

with slides from Balazs Kovacs, Eston Schweickart, Daniel Schroeder, Jiang Huang and Pramook Khungurn over the years.
Today

- Depth Test
- Light sources
- Shading models
Review
Recall: OpenGL Vertex Transformations

- Coordinates specified are transformed
- End result: window coordinates (in pixels)
Recall: OpenGL Vertex Transformation

\[
\begin{bmatrix}
x_w \\
y_w \\
0 \\
1
\end{bmatrix} = \begin{bmatrix}
\text{Viewport Transform} & \text{Perspective Divide} & \text{Projection Transform} & \text{View Transform} & \text{Model Transform}
\end{bmatrix} \begin{bmatrix}
x_o \\
y_o \\
z_o \\
1
\end{bmatrix}
\]
Review: Transforming Normals
Demo: visibility and Rendering Order

What is drawn afterwards overwrites what’s drawn before
in 3D, it should look like this
Depth Test

• Functionality to simulate occlusion due to depth in 3D
  • Nearer objects occlude farther objects

• To turn on:
  • GL11.glEnable(GL11.GL_DEPTH_TEST);

• To turn off:
  • GL11.glDisable(GL11.GL_DEPTH_TEST);

• Algorithm: z-buffer (aka depth buffer)
  • Store depth value at each pixel
  • Keep the fragment from object with the lowest z from viewer
Depth Test and glClear

- Now, we have two buffers to worry about
  - Color buffer
  - Depth buffer

- When calling glClear, must clear both buffers

```java
GL11.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);
```

- Set the value to fill the depth buffer with glClearDepth.
  - Most of the time: GL11.glClearDepth(1.0);
  - 1.0 is the maximum depth used by OpenGL
OpenGL Shading Models

- So far:
  - Specify colors of vertices
  - Vertices of a triangle
  - Same color -> triangle have same color
  - Different colors -> depends on shading model

- Flat shading
  - Use the color of the first vertex for the triangle

- Smooth shading
  - Interpolate color using barycentric coordinate
Specifying Shading Models (legacy)

- `GL11.glShadeModel(GL2.GL_FLAT);`
- `GL11.glShadeModel(GL2.GL_SMOOTH);`
Real World Illumination

- See things that reflect light from a light source or because they emit light
- Classic models involve computations in the fragment or vertex shaders
- Performance vs quality tradeoff
Light Sources

- **Point source**
  - rays originate at a point
  - different incident angles on a plane

- **Directional source**
  - rays are parallel
  - identical incident angles on a plane

- **Area source**
  - finite area in space
  - hybrid of point and directional source

- **Ambient source**
  - equal light from all directions (background illumination)
Directional Light Source

- Light position $w = 0$ component becomes light direction
- Light direction constant for every pixel
- Intensity constant for every pixel
- Used to model far away light sources, e.g. the sun
Directional light vertex shader

#version 330 core
uniform mat4 MVPMatrix;
uniform mat3 NormalMatrix; // to transform normals
in vec4 VertexColor;
in vec3 VertexNormal;      // we now need a surface normal
in vec4 VertexPosition;

out vec4 Color;
out vec3 Normal;         // interpolate the normalized normal
void main()
{
    Color = VertexColor;
    // transform the normal, without perspective, and normalize it
    Normal = normalize(NormalMatrix * VertexNormal);
    gl_Position = MVPMatrix * VertexPosition;
}
in vec4 Color;
in vec3 Normal; // surface normal, interpolated between vertices
out vec4 FragColor;

void main() {
    // compute cosine of the directions, using dot products,

    float diffuse = max(0.0, dot(Normal, LightDirection));
    float specular = max(0.0, dot(Normal, HalfVector));
    specular = pow(specular, Shininess); // sharpen the highlight

    vec3 scatteredLight = Ambient + LightColor * diffuse;
    vec3 reflectedLight = LightColor * specular * Strength;

    vec3 rgb = min(Color.rgb * scatteredLight + reflectedLight, vec3(1.0));
    FragColor = vec4(rgb, Color.a);
}
Point Light Source

• Light position w component = 1
• Intensity can be attenuated by distance from light source (inverse square relationship)
• Can be made a spotlight by specifying more parameters. e.g. ceiling lights and street lights
Point light vertex shader

#version 330 core

uniform mat4 MVPMatrix;
uniform mat4 MVMatrix;     // now need the transform, minus perspective
uniform mat3 NormalMatrix;
in vec4 VertexColor;
in vec3 VertexNormal;
in vec4 VertexPosition;

out vec4 Color;
out vec3 Normal;
out vec4 Position;   // adding position, so we know where we are

void main()
{
    Color = VertexColor;
    Normal = normalize(NormalMatrix * VertexNormal);
    Position = MVMatrix * VertexPosition;     // pre-perspective space
    gl_Position = MVPMatrix * VertexPosition; // includes perspective
}
Point light fragment shader

...
Spotlights

• Light position w component = 1
• Cone of light in a particular direction
• Intensity can be attenuated by distance from light source
e.g. desk lamps and flashlights

Image stolen from Jerry Talton’s slides for UCSC’s CMP 160: Intro to Computer Graphics, 2006
Spotlight vertex shader

// Vertex shader for spotlight computed in the fragment shader
#version 330 core

uniform mat4 MVPMatrix;
uniform mat4 MVMatrix;
uniform mat3 NormalMatrix;

in vec4 VertexColor;
in vec3 VertexNormal;
in vec4 VertexPosition;

out vec4 Color;
out vec3 Normal;
out vec4 Position;

void main()
{
    Color = VertexColor;
    Normal = normalize(NormalMatrix * VertexNormal);
    Position = MVMatrix * VertexPosition;
    gl_Position = MVPMatrix * VertexPosition;
}
uniform vec3 ConeDirection; // adding spotlight attributes
uniform float SpotCosCutoff; // how wide the spot is, as a cosine
uniform float SpotExponent;  // control light fall-off in the spot
in vec4 Color;
in vec3 Normal;
in vec4 Position;
out vec4 FragColor;

void main() {

    // how close are we to being in the spot?
    float spotCos = dot(lightDirection, -ConeDirection);
    // attenuate more, based on spot-relative position
    if (spotCos < SpotCosCutoff)
        attenuation = 0.0;
    else
        attenuation *= pow(spotCos, SpotExponent);

    vec3 halfVector = normalize(lightDirection + EyeDirection);

    vec3 rgb = min(Color.rgb * scatteredLight + reflectedLight,
                   vec3(1.0));
    FragColor = vec4(rgb, Color.a);
Light Position and Transforms

- Light position gets transformed like a vertex point

- Many choices for where to specify:
  - After modeling transform (can specify in light’s object space)
  - After view transform (can specify position in the scene)
  - After projection transform (light moves with the camera)
Surface materials

Broadly there are two main categories of materials:

- **Diffuse / Lambertian**
  - rough surfaces
  - equal reflection in all directions
- **Specular**
  - smooth surfaces
  - reflect light in a well-defined angle
Ambient Shading

Environment with non-directional light

- Same amount of light everywhere
- Object has color of underlying material (silhouette)

\[ I_a = k_a \times L_a \]

- \( k_a \): fraction of light reflected from material surface
- \( L_a \): intensity of ambient light source
Ambient vertex shader

// Vertex shader for ambient light

#version 330 core

uniform mat4 MVPMatrix;

in vec4 VertexColor;
in vec4 VertexPosition;
out vec4 Color;

void main()
{
    Color = VertexColor;
    gl_Position = MVPMatrix * VertexPosition;
}
Ambient fragment shader

// Fragment shader for global ambient lighting

#version 330 core

uniform vec4 Ambient; // sets lighting level, same across many vertices
in vec4 Color;
out vec4 FragColor;

void main()
{
    vec4 scatteredLight = Ambient; // this is the only light
    // modulate surface color with light, but saturate at white
    FragColor = min(Color * scatteredLight, vec4(1.0));
}
Diffuse / Lambertian Shading

- Rough or dull surfaces
- Incident light reflected equally in all directions
- $\cos()$ falloff with increasing angle from the normal direction

\[
I_d = k_d \cdot \text{dot}(l, n) \cdot L_d
\]

$k_d$: fraction of diffuse light reflected from the surface
$L_d$: diffuse light intensity
$\cos(\theta) = \text{dot}(l, n)$ where $l$ is the light position and $n$ is the surface normal
Specular Shading

- Smooth (mirror/metallic) surfaces
- Incident light reflected in preferred direction determined by the surface normal
- \( \cos() \) falloff with increasing angle from the normal direction

\[
l_s = k_s \cdot \text{dot}( r, v )^a \cdot L_s
\]

- \( k_s \): fraction of specular light reflected from the surface
- \( L_s \): light intensity
- \( \cos(\phi) \): angle between the viewer direction and the reflected light direction
- \( a \) : shininess exponent
Total Illumination

- Classic lighting model adds up independently computed components to get the final lighting effect

\[ I = I_a + I_d + I_s \]

Additive superposition for all the light sources
Triangle Shading Algorithms

Once we have the material and lights in the scene we need an approach to compute the shading at a given point in a triangle

- Per polygon: Flat shading
- Per vertex: Gouraud shading
- Per pixel: Phong shading
Triangle Shading Algorithms
Moving Calculations to the Vertex Stage

Expensive square root computations per fragment/pixel

Perform distance calculation per vertex

Then interpolate the result instead of interpolating all the terms involved in the calculation
Point light in vertex shader

// Vertex shader pulling point-light calculations up from the fragment shader.
#version 330 core
uniform mat4 MVPMatrix;
uniform mat3 NormalMatrix;
uniform vec3 LightPosition;    // consume in the vertex shader now
uniform vec3 EyeDirection;
uniform float ConstantAttenuation;
uniform float LinearAttenuation;
uniform float QuadraticAttenuation;
in vec4 VertexColor;
in vec3 VertexNormal;
in vec4 VertexPosition;

out vec4 Color;
out vec3 Normal;
out vec3 LightDirection;    // send the results instead
out vec3 HalfVector;
out float Attenuation;

void main() {
    Color = VertexColor;
    Normal = normalize(NormalMatrix * VertexNormal);
    // Compute these in the vertex shader instead of the fragment shader
    LightDirection = LightPosition - vec3(VertexPosition);
    float lightDistance = length(LightDirection);
    LightDirection = LightDirection / lightDistance;
    Attenuation = 1.0 /
        (ConstantAttenuation +
         LinearAttenuation * lightDistance +
         QuadraticAttenuation * lightDistance * lightDistance);
    HalfVector = normalize(LightDirection + EyeDirection);
    gl_Position = MVPMatrix * VertexPosition;
}
// Fragment shader with point-light calculations done in vertex shader
#version 330 core
uniform vec3 Ambient;
uniform vec3 LightColor;
// uniform vec3 LightPosition; // no longer need this
uniform float Shininess;
uniform float Strength;
in vec4 Color;
in vec3 Normal;
// in vec4 Position; // no longer need this
in vec3 LightDirection; // get these from vertex shader instead
in vec3 HalfVector;
in float Attenuation;
out vec4 FragColor;

void main() {
    // LightDirection, HalfVector, and Attenuation are interpolated now, from vertex shader calculations

    float diffuse = max(0.0, dot(Normal, LightDirection));
    float specular = max(0.0, dot(Normal, HalfVector));

    if (diffuse == 0.0)
        specular = 0.0;
    else
        specular = pow(specular, Shininess) * Strength;

    vec3 scatteredLight = Ambient + LightColor * diffuse * Attenuation;
    vec3 reflectedLight = LightColor * specular * Attenuation;
    vec3 rgb = min(Color.rgb * scatteredLight + reflectedLight,
                   vec3(1.0));
    FragColor = vec4(rgb, Color.a);
}
Gouraud vertex shader (SB example)

version 420 core
// Per-vertex inputs
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
// Matrices we'll need
layout (std140) uniform constants
{
    mat4 mv_matrix;
    mat4 view_matrix;
    mat4 proj_matrix;
};
// Light and material properties
uniform vec3 light_pos = vec3(100.0, 100.0, 100.0);
uniform vec3 diffuse_albedo = vec3(0.5, 0.2, 0.7);
uniform vec3 specular_albedo = vec3(0.7);
uniform float specular_power = 128.0;
uniform vec3 ambient = vec3(0.1, 0.1, 0.1);
// Outputs to the fragment shader
out VS_OUT
{
    vec3 color;
} vs_out;
Gouraud vertex shader
(SB example)

```c
void main(void) {
    // Calculate view-space coordinate
    vec4 P = mv_matrix * position;
    // Calculate normal in view space
    vec3 N = mat3(mv_matrix) * normal;
    // Calculate view-space light vector
    vec3 L = light_pos - P.xyz;
    // Calculate view vector (simply the negative of the view-space position)
    vec3 V = -P.xyz;
    // Normalize all three vectors
    N = normalize(N);
    L = normalize(L);
    V = normalize(V);
    // Calculate R by reflecting -L around the plane defined by N
    vec3 R = reflect(-L, N);
    // Calculate the diffuse and specular contributions
    vec3 diffuse = max(dot(N, L), 0.0) * diffuse_albedo;
    vec3 specular = pow(max(dot(R, V), 0.0), specular_power) * specular_albedo;
    // Send the color output to the fragment shader
    vs_out.color = ambient + diffuse + specular;
    // Calculate the clip-space position of each vertex
    gl_Position = proj_matrix * P;
}
```
#version 420 core

// Output
layout (location = 0) out vec4 color;

// Input from vertex shader
in VS_OUT
{
    vec3 color;
} fs_in;

void main(void)
{
    // Write incoming color to the framebuffer
    color = vec4(fs_in.color, 1.0);
}
Gouraud shading (per vertex lighting)
#version 420 core

// Per-vertex inputs
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

// Matrices we'll need
layout (std140) uniform constants {
    mat4 mv_matrix;
    mat4 view_matrix;
    mat4 proj_matrix;
};

// Inputs from vertex shader
out VS_OUT {
    vec3 N;
    vec3 L;
    vec3 V;
} vs_out;

// Position of light
uniform vec3 light_pos = vec3(100.0, 100.0, 100.0);
void main(void) {
    // Calculate view-space coordinate
    vec4 P = mv_matrix * position;

    // Calculate normal in view-space
    vs_out.N = mat3(mv_matrix) * normal;

    // Calculate light vector
    vs_out.L = light_pos - P.xyz;

    // Calculate view vector
    vs_out.V = -P.xyz;

    // Calculate the clip-space position of each vertex
    gl_Position = proj_matrix * P;
}
#version 420 core

// Output
layout (location = 0) out vec4 color;

// Input from vertex shader
in VS_OUT
{
  vec3 N;
  vec3 L;
  vec3 V;
} fs_in;

// Material properties
uniform vec3 diffuse_albedo = vec3(0.5, 0.2, 0.7);
uniform vec3 specular_albedo = vec3(0.7);
uniform float specular_power = 128.0;

void main(void)
{
  // Normalize the incoming N, L, and V vectors
  vec3 N = normalize(fs_in.N);
  vec3 L = normalize(fs_in.L);
  vec3 V = normalize(fs_in.V);

  // Calculate R locally
  vec3 R = reflect(-L, N);

  // Compute the diffuse and specular components for each fragment
  vec3 diffuse = max(dot(N, L), 0.0) * diffuse_albedo;
  vec3 specular = pow(max(dot(R, V), 0.0), specular_power) * specular_albedo;

  // Write final color to the framebuffer
  color = vec4(diffuse + specular, 1.0);
}
Phong shading (per fragment lighting)
Phong Shininess
Fog shader

The color of distant objects changes as the viewer moves away from the object.
Fog shader

- Medium in which light travels (e.g. air) is not perfectly transparent
- Gases absorb and scatter light as it travels
- Fog caused by particles or vapor interact with light
Fog vertex shader

#version 330 core

uniform mat4 VP;

in vec4 vColor;
in vec4 vPos;

out vec4 Color;
out vec4 ptloc;

void main()
{
    Color = vColor;
    ptloc = VP * vPos;
    gl_Position = ptloc;
}
#version 330 core
uniform int enable_fog = 1;
uniform vec4 fog_color = vec4(0.7, 0.8, 0.9, 0.0);
in vec4 Color;
in vec4 ptloc;
out vec4 FragColor;

vec4 fog(vec4 c) {
    float z = length(ptloc.z);
    float de = 25 * smoothstep(0.0, 6.0, 1.0 - ptloc.y);
    float di = 45 * smoothstep(0.0, 4.0, 2.0 - ptloc.y);

    float extinction   = exp(-z * de);
    float inscattering = exp(-z * di);
    return c * extinction + fog_color * (1.0 - inscattering);
}

void main(void) {
    if (enable_fog == 1)
        FragColor = fog(Color);
    else
        FragColor = Color;
}
Demo: Fog scene