Welcome!
Today’s Agenda:

- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur
Basics of Shading

A diffuse material scatters incoming light equally in all directions.

Two aspects to this:

1. Diffuse materials are view-independent.
2. We need to know how much light is arriving.

Arriving: *irradiance*, expressed in Joules per second per $m^2$. Or, for our purposes, *per unit area*. 

\[
\text{Arriving: } N \text{L} \cos \theta = \mathbf{N} \cdot \mathbf{L}
\]
Basics of Shading

So, in the fragment shader:

- Calculate $L$: $L = lightPos - intersectionPoint$
- Use $N$: already passed via vertex shader
- Apply attenuation, scale by material color
- Multiply by light color
- Emit fragment color.

But wait...
Basics of Shading

So, in the fragment shader:

- Calculate $L$: $L = \text{lightPos} - \text{intersectionPoint}$
- Use $N$: already passed via vertex shader
- Apply attenuation, scale by material color
- Multiply by light color
- Emit fragment color.

But wait...

We have significant problems:

1. How do we specify a light position
2. In which space do we operate?

---

```cpp
#version 330

// shader input
in vec2 vUV;
in vec3 vNormal;
in vec3 vPosition;

// shader output
out vec4 normal;
out vec2 uv;
uniform mat4 transform;

// vertex shader
void main()
{
    // transform vertex using supplied matrix
    gl_Position = transform * vec4( vPosition, 1.0);

    // forward normal and uv coordinate
    normal = transform * vec4( vNormal, 0.0f );
    uv = vUV;
}
```
Spaces

Default matrix in game.cs:

```
Matrix4 transform = Matrix4.CreateFromAxisAngle(new Vector3(0, 1, 0), a);
transform *= Matrix4.CreateTranslation(0, -4, -15);
transform *= Matrix4.CreatePerspectiveFieldOfView(1.2f, 1.3f, .1f, 1000);
```

This produces:

- a teapot that spins around its pivot
- a camera located at (0, 4, 15) or the teapot spins at (0, -4, 15), camera is at (0, 0, 0).

The last line adds perspective.

➡️ We need a ‘base system’ in which we can define a light position: world space.
Diffuse

Spaces

Getting model space coordinates to world space:

```java
Matrix4 transform = Matrix4.CreateFromAxisAngle( new Vector3( 0, 1, 0 ), a );
Matrix4 toWorld = transform;
transform *= Matrix4.CreateTranslation( 0, -4, -15 );
transform *= Matrix4.CreatePerspectiveFieldOfView( 1.2f, 1.3f, .1f, 1000 );
```

We need some additional changes now:

```java
public void Render(
    Shader shader,
    Matrix4 transform,
    Matrix4 toWorld,
    Texture texture
)
```
**Diffuse**

Changes

The vertex shader now takes two matrices:

```cpp
// transforms
uniform mat4 transform;    // full transform: model space to screen space
uniform mat4 toWorld;      // model space to world space

...and uses them:

gl_Position = transform * vec4(vPosition, 1.0 );
worldPos = toWorld * vec4(vPosition, 1.0f );
normal = toWorld * vec4(vNormal, 0.0f );
```
Diffuse

Changes

The shader class needs to know about the two matrices:

```java
public int uniform_mview;
public int uniform_2wrld;
```

... 

```java
uniform_mview = GL.GetUniformLocation( programID, "transform" );
uniform_2wrld = GL.GetUniformLocation( programID, "toWorld" );
```

And the mesh class needs to pass both to the shader:

```java
// pass transforms to vertex shader
GL.UniformMatrix4( shader.uniform_mview, false, ref transform );
GL.UniformMatrix4( shader.uniform_2wrld, false, ref toWorld );
```
Diffuse

Changes

The new fragment shader, complete:

```
#version 330
in vec2 uv; // interpolated texture coordinates
in vec4 normal; // interpolated normal, world space
in vec4 worldPos; // world space position of fragment
uniform sampler2D pixels; // texture sampler
out vec4 outputColor; // shader output

uniform vec3 lightPos; // light position in world space

void main()
{
    vec3 L = lightPos - worldPos.xyz;
    float dist = L.length();
    L = normalize(L);
    vec3 lightColor = vec3(10, 10, 8);
    vec3 materialColor = texture(pixels, uv).xyz;
    float attenuation = 1.0f / (dist * dist);
    outputColor = vec4(materialColor * max(0.0f, dot(L, normal.xyz)) * attenuation * lightColor, 1);
}
```

In game.cs, Init():

```
// set the light
int lightID = GL.GetUniformLocation(shader.programID, "lightPos");
GL.UseProgram(shader.programID);
GL.Uniform3(lightID, 0.0f, 10.0f, 0.0f);
```
Diffuse

```cpp
float3 b = n * (n * (n * n / dot3(n, n) - dot3(n, r) / dot3(n, r)) - dot3(n, r) / dot3(n, r));
float3 brdf = SampleDiffuse(diffuse, n, r1, r2, 0, 0, Survive();
float pdf = 1.0;
E = brdf * (dot3(n, r) / pdf);
```
Today’s Agenda:

- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur
Glossy Materials

A glossy material reflects, but the reflection is somewhat fuzzy:

Using a ray tracer we achieve this effect by sending multiple rays in directions close to the reflection vector $\hat{R}$. 
Glossy Materials

Phong model:

\[
\begin{align*}
\mathbf{L} & = \max(0, \mathbf{I} \cdot \mathbf{N}) \\
\mathbf{R} & = \mathbf{I} - \mathbf{L} \\
\mathbf{F} & = \mathbf{R} \cdot \mathbf{N}
\end{align*}
\]

\[
\begin{align*}
\mathbf{L} & = \max(0, \mathbf{I} \cdot \mathbf{N}) \\
\mathbf{R} & = \mathbf{I} - \mathbf{L} \\
\mathbf{F} & = \mathbf{R} \cdot \mathbf{N}
\end{align*}
\]

where:

- \( \mathbf{I} \) is the incident light.
- \( \mathbf{N} \) is the normal vector to the surface.
- \( \mathbf{L} \) is the reflected light.
- \( \mathbf{R} \) is the remaining light.
- \( \mathbf{F} \) is the fraction of the light reflected.

The Phong model introduces a specular component that depends on the light and the surface normal, as well as a diffuse component that is independent of the light direction. This model is widely used in computer graphics for rendering glossy materials.
Phong

Glossy Materials

Let:
\( \hat{L} \) be a vector from the fragment position \( b \) to the light;
\( \hat{R} \) be the vector \( \hat{V}_b \) reflected in the plane with normal \( \hat{N} \)
then:
if \( \hat{R} = \hat{L} \) then \( \hat{R} \cdot \hat{L} = 1 \)
and:
if \( \hat{R} \approx \hat{L} \) then \( \hat{R} \cdot \hat{L} \to 1 \)
Phong

Glossy Materials

"Locations near b receive almost as much light as b."

But how much? $$(\hat{L} \cdot \hat{V})^\alpha$$
The Full Phong

\[ M_{\text{phong}} = c_{\text{ambient}} + c_{\text{diff}} (\vec{N} \cdot \vec{L}) L_{\text{diff}} + c_{\text{spec}} (\vec{L} \cdot \vec{R})^\alpha L_{\text{spec}} \]

The Phong material model is a combination of:

- ‘Specular’ illumination: \((\vec{L} \cdot \vec{R})^\alpha\), times the ‘specular color’ of the light, times the ‘specular color’ of the material;
- Diffuse illumination: \((\vec{N} \cdot \vec{L})\), times the ‘diffuse color’ of the light, times the ‘diffuse color’ of the material;
- An ‘ambient color’.

```
M_{\text{phong}} = c_{\text{ambient}} + c_{\text{diff}} (\vec{N} \cdot \vec{L}) L_{\text{diff}} + c_{\text{spec}} (\vec{L} \cdot \vec{R})^\alpha L_{\text{spec}}
```
Phong Lighting

Diffuse Lighting used in *Half-Life 2* + Specular Lighting = High Quality Lighting in *Half-Life 2: Episode One*

```
... (depth < FORWARD)... 
... = inside / sf; ...
... nt = nt / nt; ...
... = h * n2t + u * v2t; ...
... t = 1 - (4 * vec4::dot(N, R)); ...
...
E = ((weight * costhe1dot) / direct) ...
... = true; ...
...
if (refr) && (depth < FORWARD) ... D, N ); ...
... = true; ...
```

survive = Survive:Probability (distinguishing between - doing it properly, ...)
Today’s Agenda:

- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur
Mirrors

Reflections

Reflections in a ray tracer are easy:

$$\mathbf{R} = \mathbf{L} - 2(\mathbf{L} \cdot \mathbf{N})\mathbf{N}$$

But what about rasterizers?
Mirrors

```cpp
float s = depth / MAXDEPTH;
if (s < 0.0f) s = 0.0f;
if (s > 1.0f) s = 1.0f;
float n = 0.0f;
float t = 1.0f - s;
float a = n - r;
float b = n - t;
E = ((a + b) / 2.0f);
E = E * diffuse;
if (r < N) { // diffuse
    R = R + E * refr;
}
else { // refractive
    R = R - E * diffuse;
}
```

```cpp
survive = SampleSurvive(1.0f, &r); // first iteration
survive = SampleSurvive(1.0f, &r); // second iteration
radius = SampleLight(brand, L, N, Align); // align with the light source
radius = radius * radius / 2.0f;
E = (weight * cosTheta0) / (direct0); // (random walk) - done properly, closely following Survive()
if (sample < brdf) { // SampleDiffuse(diffuse, N, r1, r2, AA, Survive);
    pdf = E * brdf * (dot(N, r) / pdf);
} else {
    pdf = E * pdf0;
}
```
Mirrors

Planar Reflections

We can fake reflections in a rasterizer by duplicating the scene:

The mirror is not really there; it’s just a hole through which we see a copy of the scene.
Mirrors

```c
float3 brdf = SampleDiffuse( diffuse, N, rt, r2, SSR, true);

E = brdf * (dot(N, R)/pdf);
```
Environment Mapping

Reflections on complex surfaces are faked using an environment map.

This is done exactly as in a ray tracer:

- at the fragment position, we have $\hat{V}$ and $\hat{N}$;
- based on these we calculate the reflected vector $\hat{R}$;
- we use $\hat{R}$ to look up a value in the skydome texture.

Limitations:

- we will not reflect anything but the skydome;
- the reflection is static.
Today’s Agenda:

- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur
Bumps

Recap: Normal Interpolation
Bumps

Normal Maps

A normal map is similar to a texture:

- we use textures to lookup a color at a particular position on a mesh;
- we use normal to lookup a normal at a particular position.

Normal maps generally store normals in *tangent space*. 
Bumps

Normal Map Data

A normal map stores 2 or 3 components per texel.

For 3 components: \(x, y, z \in [0..255]\).

To get this to the correct range:

- Cast to float
- Divide by 128 \(\Rightarrow x, y, z \in [0..2]
- Subtract 1 \(\Rightarrow x, y, z \in [-1..1]\)
Normal Map Data

A normal map stores 2 or 3 components per texel.

For 2 components: \( x, y \in [0..255] \).

To reconstruct the normal, we first apply the same conversion as we did for three components. Then:

\[
\sqrt{x^2 + y^2 + z^2} = 1
\]

\[
x^2 + y^2 + z^2 = 1
\]

\[
z^2 = 1 - (x^2 + y^2)
\]

\[
z = \pm \sqrt{1 - (x^2 + y^2)}
\]

\[
z = \sqrt{1 - (x^2 + y^2)}
\]
Bumps

Tangent Space

Things are easy when \( x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \ y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \) and \( z = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \):

\[ N = M_x x + M_y y + M_z z \]

\[ = M_x \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + M_y \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + M_z \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} M_x \\ M_y \\ M_z \end{pmatrix} \]
Tangent Space

Things are still easy for arbitrary x, y and z:

\[ N = M_x x + M_y y + M_z z \]

Obtaining this x, y and z:

- \( z \) = (interpolated) surface normal
- x is perpendicular to z
- y is perpendicular to z and x

See Crytek for details*.

Bumps

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```

```
Bumps
```
Bumps

Massive Black™
CARNAGOR
Today’s Agenda:

- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur
Fur

Fur, Grass etc.
Fur

Fur, Grass etc.
Fur

Fur, Grass etc.
Fur

INFOGR – Lecture 10 – “Shaders”

Fur, Grass etc.
Fur

Fur, Grass etc.

#version 330

... 

// shell offset
uniform float shellOffset;

// vertex shader
void main()
{
  gl_Position = transform * vec4( vPosition + shellOffset * vNormal, 1.0 );
  worldPos = toWorld * vec4( vPosition + shellOffset * vNormal, 1.0f );
  normal = toWorld * vec4( vNormal, 0.0f );
  uv = vUV;
}
Fur

Fur, Grass etc.

In game.cs:

```csharp
for( int i = 0; i < 30; i++ )
{
    GL.UseProgram( shader.programID );
    GL.Uniform1( offsetID, (float)i * 0.02f );
    mesh.Render( shader, prevMat[19 - i / 4], prevWld[19 - i / 4], fur );
}
```
INFOGR – Lecture 10 – “Shaders”

Fur

```c
// shader code

// Survive = SurviveProbability( diffuse );
if (survive)
    // extinction
    // Now,

    // radiance = SampleLight(brand, I, A, A); E = x + radiance.y + radiance.z > 0 ? E : E;
    if (true)
        // at bnd
        // Scoredf = EvaluateDiffuse( L, N );
        if (true)
            // at factor = diffuse * W;
            // weight = Mss2( directedf, bndprob );
            // E = ((weight * costheta00) / directedf); // random walk - done properly, closely follows

        SampleDiffuse( diffuse, N, r1, r2, LL, Rr, LL, Rr);
```

INFOGR – Computer Graphics
Jacco Bikker & Debabrata Panja - April-July 2017

END OF lecture 10: “Shaders”

Next lecture: “Visibility”