INFOGR – Computer Graphics

Jacco Bikker & Debabrata Panja - April-July 2018

Lecture 10: “Shaders”

Welcome!
Operations:
1. Translate head down
2. Rotate towards snowspeeder
3. Translate towards torso

\[
\hat{z} = \text{normalize}(P_{\text{speeder}} - P_{\text{trooper}}) \\
\hat{x} = \text{normalize}(\hat{z} \times (0,1,0)) \\
\hat{y} = \hat{x} \times \hat{z}
\]

\[
M_{\text{final}} = T_{\text{toTorso}} \cdot R \cdot T_{\text{down}}
\]
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 scatter light uniformly in all directions.** *(well... details in ADVGR)*

2. Arriving light however depends on angle.

**Arriving: irradiance**, expressed in Joules per second per \( m^2 \).
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** $L$: $L = lightPos - intersectionPoint$
- **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?

---

These are key points for understanding diffuse lighting in shaders.
Diffuse

Spaces

Default matrix in game.cs:

```csharp
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 it’s 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.
Spaces

Getting model space coordinates to world space:

```csharp
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:

```csharp
public void Render( 
    Shader shader,
    Matrix4 transform,
    Matrix4 toWorld,
    Texture texture 
)
```

Diffuse

Changes

The vertex shader now takes two matrices:

```glsl
// 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;
```

... 

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:

```glsl
#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()`:

```csharp
int lightID = GL.GetUniformLocation(shader.programID, "lightPos");
GL.UseProgram(shader.programID);
GL.Uniform3(lightID, 0.0f, 10.0f, 0.0f);
```
Diffuse
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 \( \hat{R} \).
Phong

Glossy Materials
Phong

Glossy Materials

Let:
\( \hat{L} \) be a vector from the fragment position \( b \) to the light;
\( \hat{R}_V \) be the vector \( \hat{V}_b \) reflected in the plane with normal \( \hat{N} \)

then:
if \( \hat{R}_V = \hat{L} \) then \( \hat{R}_V \cdot \hat{L} = 1 \)

and:
if \( \hat{R}_V \approx \hat{L} \) then \( \hat{R}_V \cdot \hat{L} \rightarrow 1 \)
Glossy Materials

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

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

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

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’.
Phong Lighting

Diffuse Lighting used in Half-Life 2

Specular Lighting

High Quality Lighting in Half-Life 2: Episode One
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

```c
if (n == 0) {
    return 0;
}

// Add any necessary code for mirrors here.
```

---

INFOGR – Lecture 10 – “Shaders”
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.
INFOGR – Lecture 10 – “Shaders”

Mirrors

```c
const float kSpecularExponent = 2.0f;

// The fragment color is determined by the diffuse and specular components.
// The diffuse component is computed using the BRDF, which depends on the
// BRDF type and the material properties. The specular component is computed
// using the Phong model.
const float diffuse = EvaluateDiffuse(L, N, kSpecularExponent);
const float specular = EvaluateSpecular(L, N, kSpecularExponent);
const float color = diffuse * diffuse + specular;
```

```c
// The fragment color is determined by the diffuse and specular components.
// The diffuse component is computed using the BRDF, which depends on the
// BRDF type and the material properties. The specular component is computed
// using the Phong model.
const float diffuse = EvaluateDiffuse(L, N, kSpecularExponent);
const float specular = EvaluateSpecular(L, N, kSpecularExponent);
const float color = diffuse * diffuse + specular;
```
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 maps to lookup a normal at a particular position.

Normal maps generally store normals in *tangent space*. 

```plaintext

```

INFOGR – Lecture 10 – “Shaders”
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 $\Rightarrow x, y, z \in [-1..1]$
Bumps

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 \( z = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \):

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

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

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 $z$:

- $z = \text{ (interpolated) surface normal}$
- $x$ is perpendicular to $z$
- $y$ is perpendicular to $z$ and $x$

See Crytek for details*.

Bumps

```
if (normal < 0)
  n = -n;

float3 view = normalize(-view);
float3 n = normalize(normal);
float3 h = normalize(view + n);
float4 E = (diffuse);  // diffuse

float3 light = normalize(lightposition - position);
float3 n = normalize(normal);
float3 h = normalize(light + n);
float4 L = (diffuse);  // diffuse

float dot = dot(light, n);
float3 H = normalize(light + n);
float3 R = (diffuse);  // diffuse

float2 tex = texCoord * texSize;

float dx = (int(tex.u) + 0.5) / (int(texSize.x - 1) + 1);
float dy = (int(tex.v) + 0.5) / (int(texSize.y - 1) + 1);

if (dot < 0.0)
  E = E * 0.0;
else
  E = E * max(0.0, dot + 0.0);
```

survive = (survival < 0.0) || (survival > 1.0);
```
```
Bumps
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

Fur, Grass etc.
Fur, Grass etc.

```cpp
#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, 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 );
}
```
Fur

```
// vertex shader
vec3 vertexColor = diffuse;

// fragment shader
vec3 fragmentColor = diffuse;
```
INFOGR – Computer Graphics
Jacco Bikker & Debabrata Panja - April-July 2018

END OF lecture 10: “Shaders”

Next lecture: “Visibility”