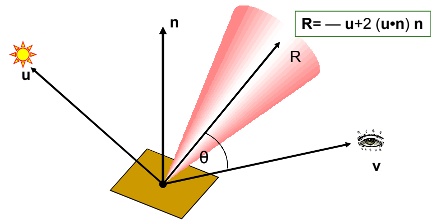
GLSL Per-vertex lighting

Revisit the basic vector elements of illumination



Revisit the basic lighting model

$$Total\_light = Emmisive + Ambient + \alpha(Diffuse + Specular)$$

Emissive material colour

ambient material colour

diffuse material colour

specular material colour

$$\begin{aligned}
Emmisive &= K_e \\
Ambient &= K_a \otimes AmbientLight \\
Diffuse &= K_d \otimes DiffuseLight \times \max(\mathbf{n} \bullet \mathbf{u}, 0) \\
Specular &= K_{spec} \otimes SpecularLight \times \max(\mathbf{r} \bullet \mathbf{v}, 0)^{shininess}
\end{aligned}$$

$$\alpha = \frac{1}{k_c + k_l d + k_q d^2}$$

Implementation options

- Per-vertex vs. per-pixel
  - per-vertex
    - Can lead to artifacts
  - per-pixel
    - Can improve rendering quality
- World space vs. view space
  - Implement in world space
    - Simple and intuitive
  - Implement in view space
    - More efficient for computing specular effects for multiple lights

Vertex shader

- Compute the colour using a vertex shader
- Only calculate colour at vertices
- The colour of each polygon will then be filled automatically within graphics hardware by interpolating colours at its vertices (The Gouraud smooth shading algorithm)

Vertex shader

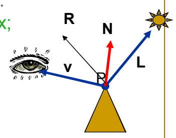
- Uniform variables
  - Light source:
    - uniform vec4 lightPos;
    - uniform vec4 specularLight; //Specular light source
    - uniform vec4 diffuseLight; //Diffuse light source
    - uniform vec4 ambientLight; //Ambient light source
  - Material reflection properties:
    - uniform vec4 Ke;
    - uniform vec4 Ka; //Ambient reflection coefficients
    - uniform vec4 Kd; //Diffuse reflection coefficients
    - uniform vec4 Ks; //Specular reflection coefficients
    - uniform float n_specular; //specular exponent
  - Eye position
    - uniform vec4 eyePos;

Vertex shader

- Attribute variables
  - per-vertex information required for computing the colour at a vertex
    - gl_Vertex; //built-in
    - gl_Normal; //built-in
- Varying variables
  - Per-vertex values will later be interpolated and passed to pixel shader
    - varying vec4 ColorAtVertex;

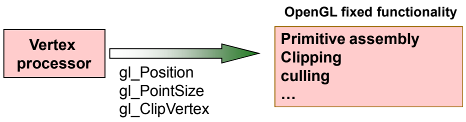
Vertex shader: **main( )**

```
// Output transformed vertex position:
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
//or gl_Position = ftransform();

// Transform vertex position into view space:
vec4 Pos = gl_ModelViewMatrix * gl_Vertex;

// Compute the light vector in view space:
vec3 L = normalize(lightPos.xyz - Pos.xyz);

// Transform normal into view space:
vec3 N = normalize( gl_NormalMatrix * gl_Normal);
```



What is gl_Position?

- A built-in Variable
  - only used in vertex shader
  - used for primitive assembly, clipping, culling, and other fixed functionality operations
    - that operate on primitives after vertex processing has occurred.



Vertex shader **main( )**

```
// Compute light reflection vector view space:
vec3 R = reflect(-L, N);

// Compute view vector in view space:
vec3 V = normalize( -Pos.xyz);

//Compute ambient term:
vec4 ambient = ambientLight * Ka;

// Compute diffuse term:
vec4 vDiff = diffuseLight * Kd * max(0.0, dot(N,   L));
```

Vertex shader **main( )**

```
// Compute specular term:
vec4 vSpec = specularLight * Ks
             * pow(max(0.0, dot(R, V)), n_specular);

ColorAtVertex = Ke + ambient + vDiff + vSpec ;
```

Pixel shader: main( )

```
varying vec4 ColorAtVertex;

void main(void)
{
  gl_FragColor = ColorAtVertex;
}
```

What is *gl_FragColor?*

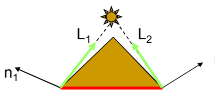
- The **output** of a fragment shader will be processed by the fixed function operations at the back end of the OpenGL pipeline.
- The **way** that a fragment shader outputs values to the OpenGL pipeline is to use the built-in variables:
  - *gl_FragColor,*
  - *gl_FragData,*
  - *gl_FragDepth*;
- *gl_FragColor* specifies the fragment color

GLSL Per-pixel lighting

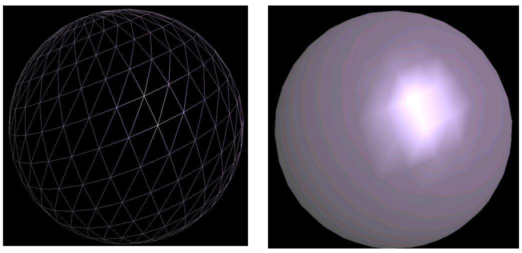
Why per-pixel lighting?

- Problem of per-vertex lighting
  - Can produce noticeable artifacts
    - The shaded objects look like polyhedra when the underlying meshes are coarse
  - Why? Consider rendering a triangle using per-vertex lighting
    - When light is close to one of the vertices of a triangle, the diffuse and specular reflections on the triangle are noticable
    - But when the light close to the center of the triangle, the diffuse and specular reflections on the triangle can be invisible.

$\boxed{dot(L_1, n_1) \approx dot(L_2, n_2)}$



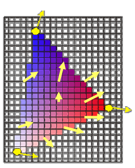
Problem of per-vertex lighting



Per-vertex

Per-Pixel lighting

- Need to apply illumination model at each fragment
- Normal at each fragment is required, which can be estimated by interpolating normals at vertices
- Illumination model is implemented in pixel shader rather than in vertex shader



Vertex shader

- To implement per-pixel lighting, the following information need to be computed in the pixel shader:
  - *direction to light*
  - *direction to the viewer*
  - *normal vector at a fragment:*

```
varying vec3 Pos;
varying vec3 Normal;
```

Vertex shader: main( )

```
varying vec3 Pos;
varying vec3 Normal;

void main( void )
{
    gl_Position = gl_ModelViewProjectionMatrix *
                                        gl_Vertex;
    Pos =  (gl_ModelViewMatrix * gl_Vertex).xyz;
    Normal = gl_NormalMatrix * gl_Normal;
}
```

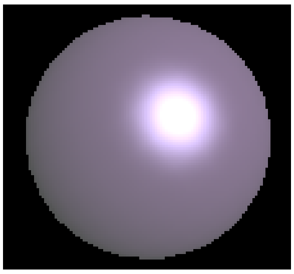
Pixel Shader

```
uniform vec4 lightPos;
uniform vec4 specularLight;
… …
uniform vec4 eyePos;
uniform vec4 Ke;
uniform vec4 Ka;
… …

varying vec3 Pos;
varying vec3 Normal;
```

Pixel Shader: Main( )

```
vec3 L = normalize(lightPos.xyz - Pos);
vec3 N = normalize(Normal);

vec3 R = reflect(-L, N);
vec3 V = normalize( -Pos);

vec4 ambient = ambientLight * Ka;
vec4 vDiff = diffuseLight * Kd * max(0.0, dot(N,   L));
vec4 vSpec = specularLight * Ks
             * pow(max(0.0, dot(R, V)), n_specular);
gl_FragColor = Ke + ambient + vDiff + vSpec;
```

Per-pixel lighting



Per-pixel lighting

Mapping technique

1. Texture mapping
2. Bump mapping
3. Parallax and relief mapping

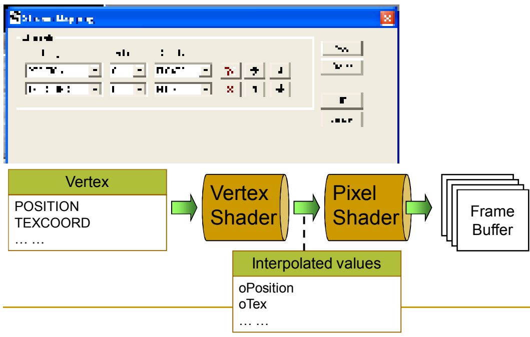
Textures as general purpose memory

- With programmable graphic HW, apart from texture mapping textures can be used for different other purposes
  - can serve as lookup tables for complex functions
  - can store intermediate rendering results
  - Can be used as a kind of general-purpose memory to store
    - normals, heights, visibility information, …
    - position, verlocity, …
    - … …

Access to Texture Maps

- Declaring a **uniform variable** of type **sampler** for each texture to access
- The application must provide a value for the sampler before execution of the shader
- The built-in functions **texture1D**, **texture2D**, **texture3D**, **textureCube**, **shadow1D**, and so on, perform texture access within a shader.

Example: texture mapped sphere —Vertex shader



Example: texture mapped sphere in GLSL —Vertex shader

```
varying vec2 Texcoord;

void main( void )
{
   gl_Position = ftransform();
   Texcoord    = gl_MultiTexCoord0.xy;
}
```

A vertex shader built-in attribute

Example: texture mapped in GLSL —Pixel shader

```
uniform sampler2D baseMap;

varying vec2 Texcoord;

void main( void )
{
   gl_FragColor = texture2D( baseMap, Texcoord );
}
```

texture2D(sampler2D *sampler*, vec2 *coord*)

- A texture lookup function
- Use the texture coordinate *coord* to do a texture lookup in the 2D texture currently bound to a *sampler*
- Return the texture colour at location *coord*

Multitexturing

- With shaders, it is easy to combine several textures together for different purposes

```
uniform sampler2D baseMap;
uniform sampler2D maskMap;
…
vec4  TexColor1= texture2D( baseMap, Texcoord );
vec4  TexColor2 = texture2D(maskMap, Texcoord );
vec4  AddColor= TexColor1+ TexColor2;
vec4  MulColor = TexColor1*TexColor2;
```

Use a texture as a mask

```
uniform sampler2D baseMap;
uniform sampler2D maskMap;
…
vec4  TexCol= texture2D( baseMap, Texcoord );
float  maskVal= texture2D( maskMap, Texcoord ).r;
… …
if (maskVal<0.6)
    gl_FragColor = (Ambient + Diffuse + Specular) * TexCol;
else
    discard
```