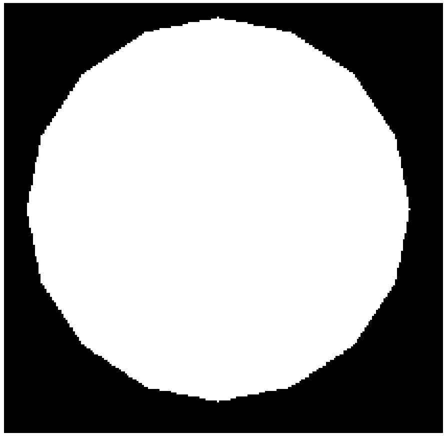
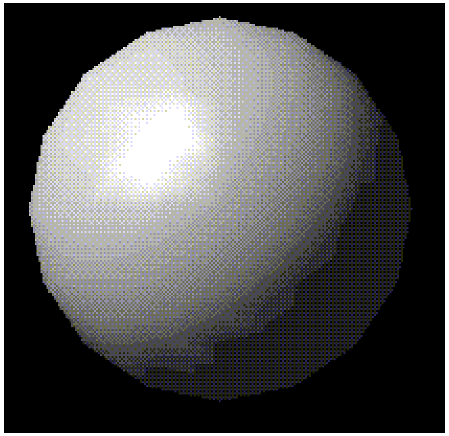
Real-time Graphics

3. Lighting, Texturing

Martin Samuelčík
Juraj Starinský

Scene illumination





Rendering equation

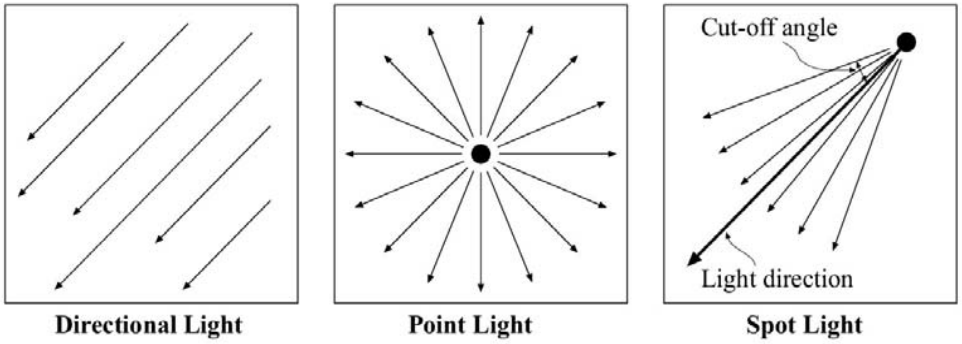
$$L_o(\mathbf{x},\omega,\lambda,t) = L_e(\mathbf{x},\omega,\lambda,t) + \int_\Omega f_r(\mathbf{x},\omega',\omega,\lambda,t)L_i(\mathbf{x},\omega',\lambda,t)(-\omega'\cdot\mathbf{n})d\omega'$$

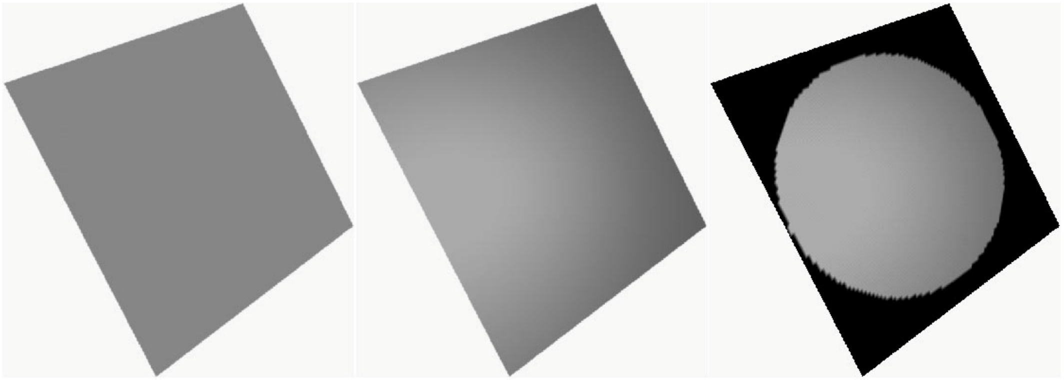
- λ is a particular wavelength of light
- t is time
- $L_o(\mathbf{x},\omega,\lambda,t)$ is the total amount of light of wavelength λ directed outward along direction ω at time t, from a particular position $\mathbf{x}$
- $L_e(\mathbf{x},\omega,\lambda,t)$ is emitted light
- $\int_\Omega \dots d\omega'$ is an integral over a hemisphere of inward directions
- $f_r(\mathbf{x},\omega',\omega,\lambda,t)$ is the proportion of light reflected from ω' to ω at position $\mathbf{x}$, time t, and at wavelength λ
- $L_i(\mathbf{x},\omega',\lambda,t)$ is light of wavelength λ coming inward toward $\mathbf{x}$ from direction ω' at time t
- $-\omega'\cdot\mathbf{n}$ is the attenuation of inward light due to incident angle

- Usually approximating this equation
- Contribution of other scene points:
  - No: Local illumination
  - Yes: Global illumination

Light sources

- Directional lights
- Point lights
- Area lights
- Volume lights



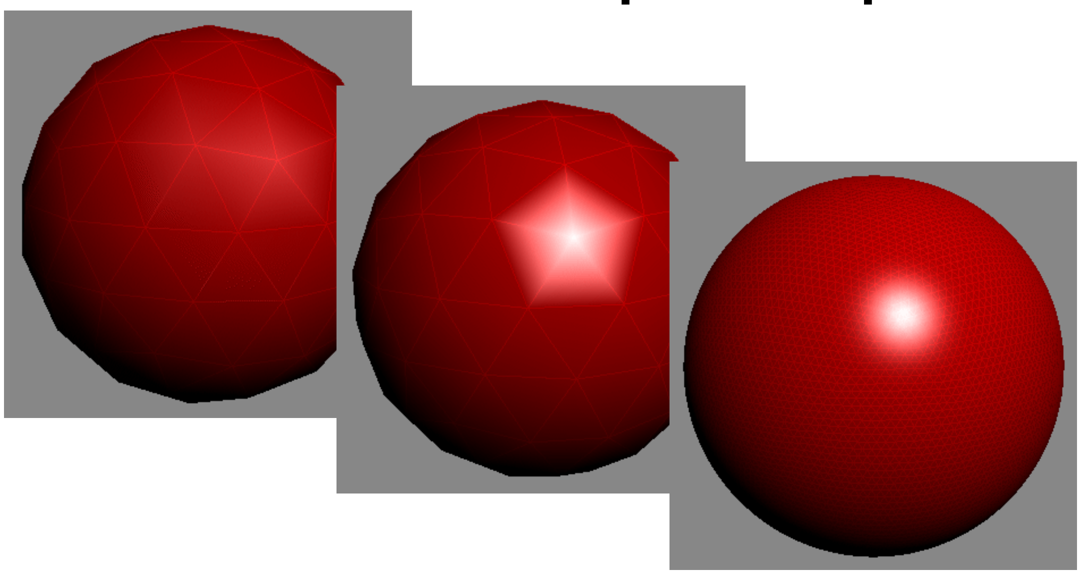


Local illumination models

- Differences mainly in specular form
- Phong
- Blinn-Phong
- Oren-Nayar
- Cook-Torrance
- Ward anisotropic distribution
- Gaussian distribution, …

Phong local illumination

- Illumination of one scene vertex
- Ambient, diffuse, specular components
- Can be computed per-vertex or per-pixel

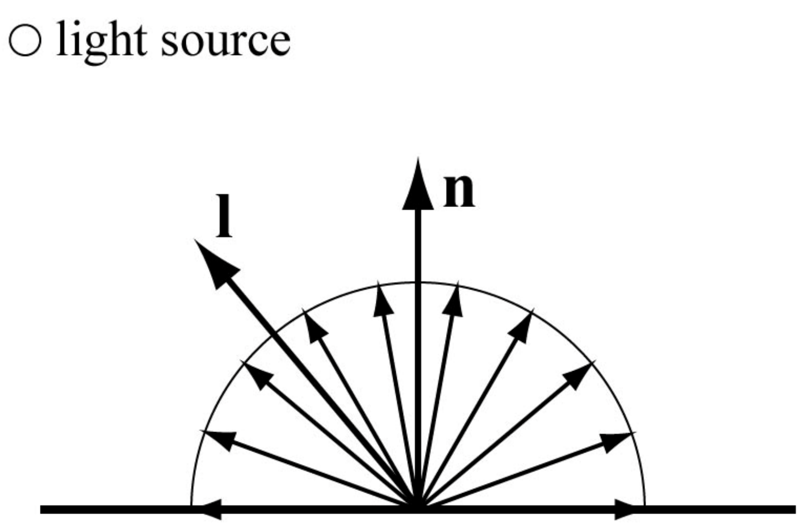


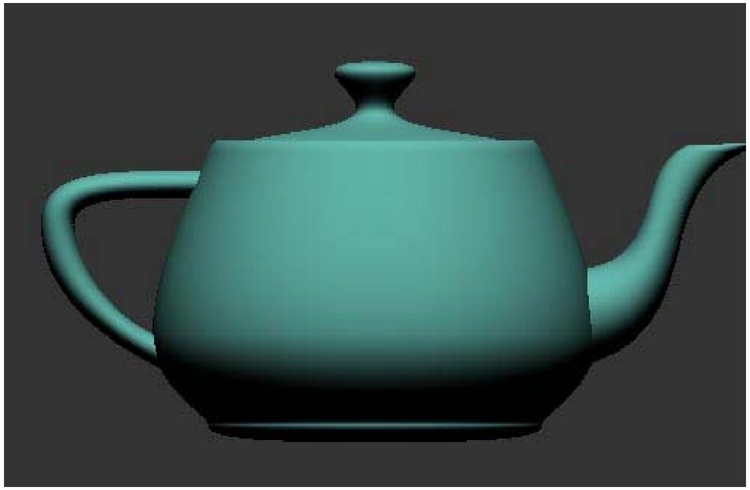
Phong – ambient term

- Constant color
- Simulating light scattered by environment
- Not affected by surface or light direction

Phong – diffuse term

- Simulating scattering of light on micro facets in all directions, intensity is given by angle of incoming light on surface
- Lambert's law $i_{diff} = \mathbf{n} \cdot \mathbf{l} = \cos\phi$



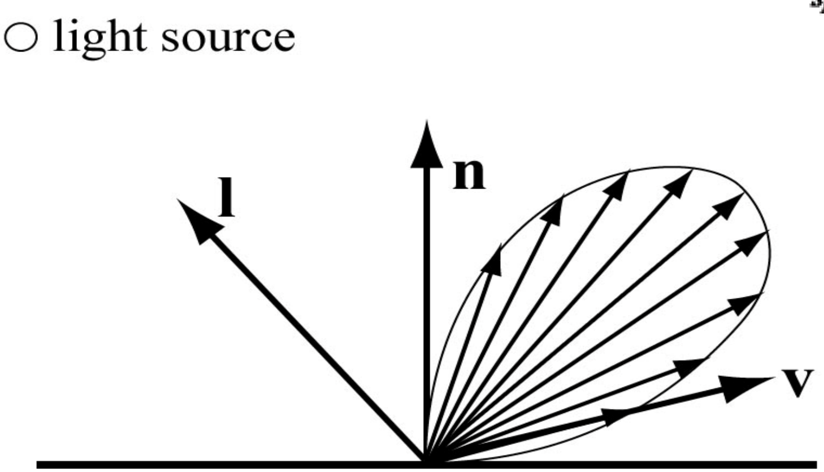


Phong – specular term

- Simulating highlight with maximal intensity in the direction opposite to light direction
- Law of reflection

$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n}$$

$$i_{spec} = (\mathbf{r} \cdot \mathbf{v})^{m_{shi}} = (\cos \rho)^{m_{shi}}$$



Phong computation

$$outputcolor_{vertex} = emission_{material} + ambient_{light\_model} * ambient_{material} +$$

$$\sum_{i=0}^{n-1} \left(\frac{1}{k_c + k_l * d + k_q * d^2}\right) * spotlighteffect_i *$$

$$[ambient_{light}[i] * ambient_{material} + (\max(L.N, 0)) * diffuse_{light}[i] * diffuse_{material} +$$

$$(\max(R.V, 0))^{shininess[i]} * specular_{light}[i] * specular_{material}]_i$$

- n – number of lights
- kc, kl, kq – attenuation factors, parameters of light i
- L – unit vector from vertex to light
- N – unit normal vector at vertex
- R = -L+2*(L.N)N
- V – unit vector from vertex to camera
- $ambient_{material}$, $diffuse_{material}$, $specular_{material}$ – material parameters
- $ambient_{light}$[i], $diffuse_{light}$[i], $specular_{light}$[i], shininess[i] – parameters of light i

Phong GLSL shaders

Vertex shader:

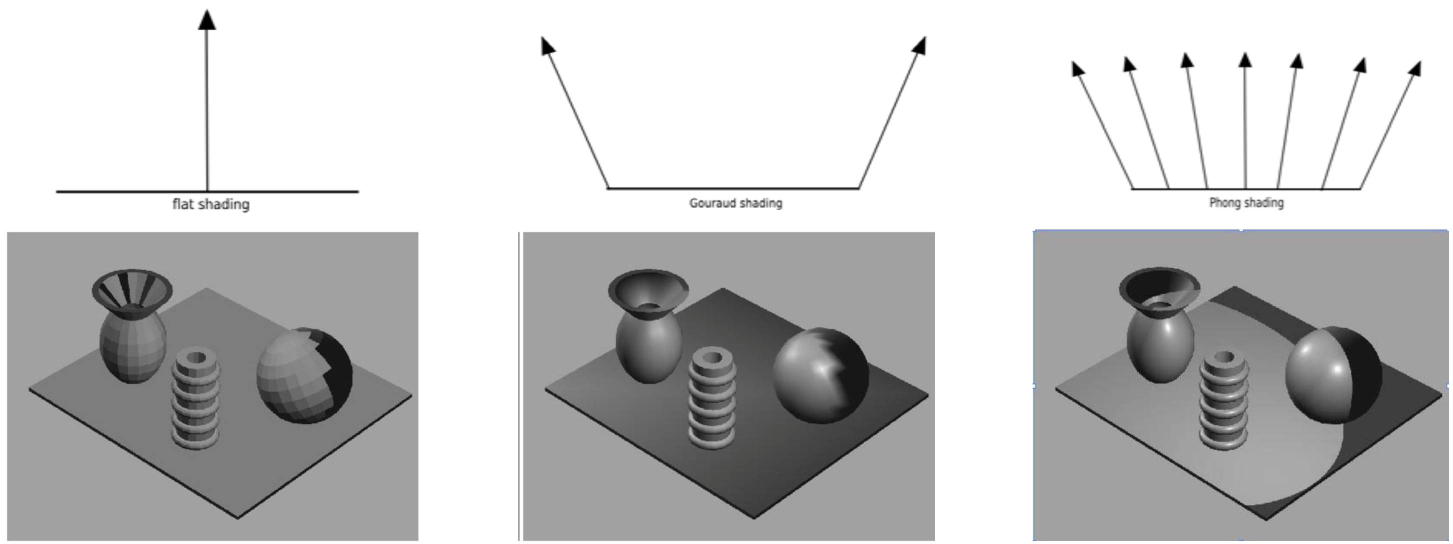
```
uniform vec4 light;

varying vec4 V_eye;
varying vec4 L_eye;
varying vec4 N_eye;
varying vec2  vTexCoord;

void main(void)
{
        V_eye = gl_ModelViewMatrix * gl_Vertex;
        L_eye = (gl_ModelViewMatrix * light) - V_eye;
        N_eye = vec4(gl_NormalMatrix * gl_Normal, 1.0);

        vTexCoord = vec2(gl_MultiTexCoord0);

        gl_Position = gl_ProjectionMatrix * V_eye;
        V_eye = -V_eye;
}
```

Fragment shader:

```
const vec4 AMBIENT = vec4( 0.9, 0.9, 0.1, 1.0 );
const vec4 SPECULAR = vec4( 1.0, 1.0, 1.0, 1.0 );
uniform sampler2D diffuseMap;
varying vec4 V_eye;
varying vec4 L_eye;
varying vec4 N_eye;
varying vec2  vTexCoord;

vec3 reflect(vec3 N, vec3 L)
{
        return 2.0*N*dot(N, L) - L;
}

void main(void)
{
        vec3 V = normalize(vec3(V_eye));
        vec3 L = normalize(vec3(L_eye));
        vec3 N = normalize(vec3(N_eye));

        vec4 Cd = texture2D( baseMap, vTexCoord);
        float diffuse = clamp(dot(L, N), 0.0, 1.0);
        vec3 R = reflect(N, L);
        float specular = clamp(pow(dot(R, V), 16.0), 0.0, 1.0);

        gl_FragColor = AMBIENT + (Cd*diffuse) + SPECULAR*specular);
}
```

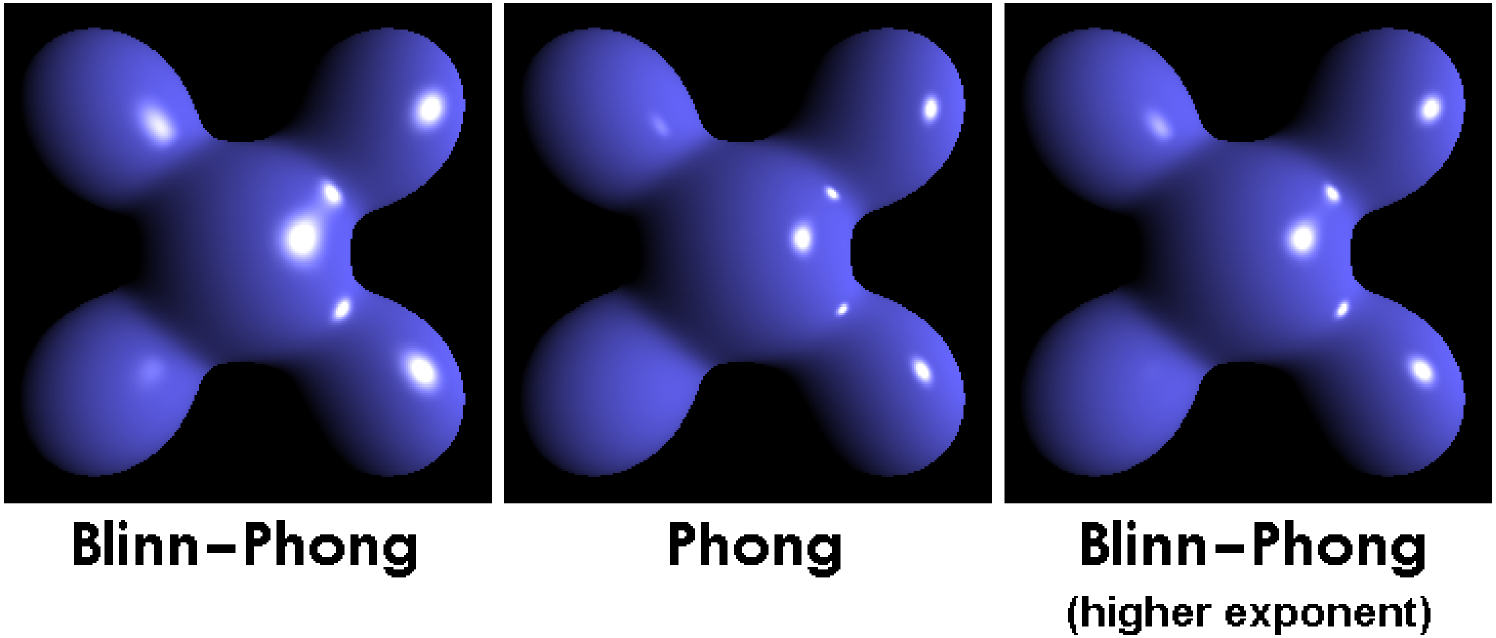
Shading

- Interpolation of input or output values
- Flat, Gourard, Phong
- per-primitive, per-vertex, per-fragment



Blinn-phong model

- Other computation of specular term
- Using half vector $H = \frac{L+V}{|L+V|}$ $\quad i_{spec} = (\mathbf{H} \cdot \mathbf{N})^{m_{shi}}$



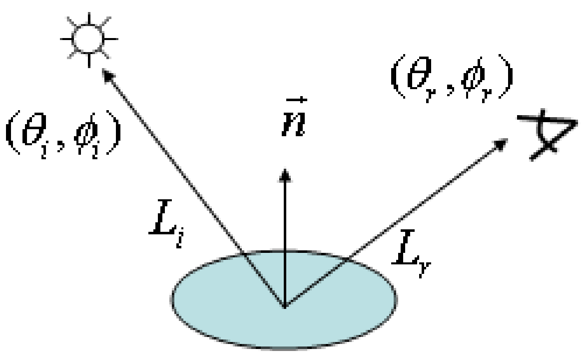
Oren-Nayar model

- Diffuse reflection from rough surfaces
- Rough surfaces are not so dimed

$$L_r = \frac{\rho}{\pi} \cdot \cos\theta_i \cdot (A + B \cdot \max(0, \cos(\phi_i - \phi_r)) \cdot \sin\alpha \cdot \tan\beta) \cdot L_i$$

$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33} \qquad B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09}$$

$\alpha = max(\theta_i,\theta_r)$, $\beta = min(\theta_i,\theta_r)$,
ρ - *albedo of the surface*
σ - *roughness*



Real Image | Lambertian Model | Oren-Nayar Model

Cook-Torrance model

- General model for rough surfaces
- For metal and plastic
- F_0 – index of refraction
- m - roughness
- Geometric term G
- Roughness term R
- Fresnel term F

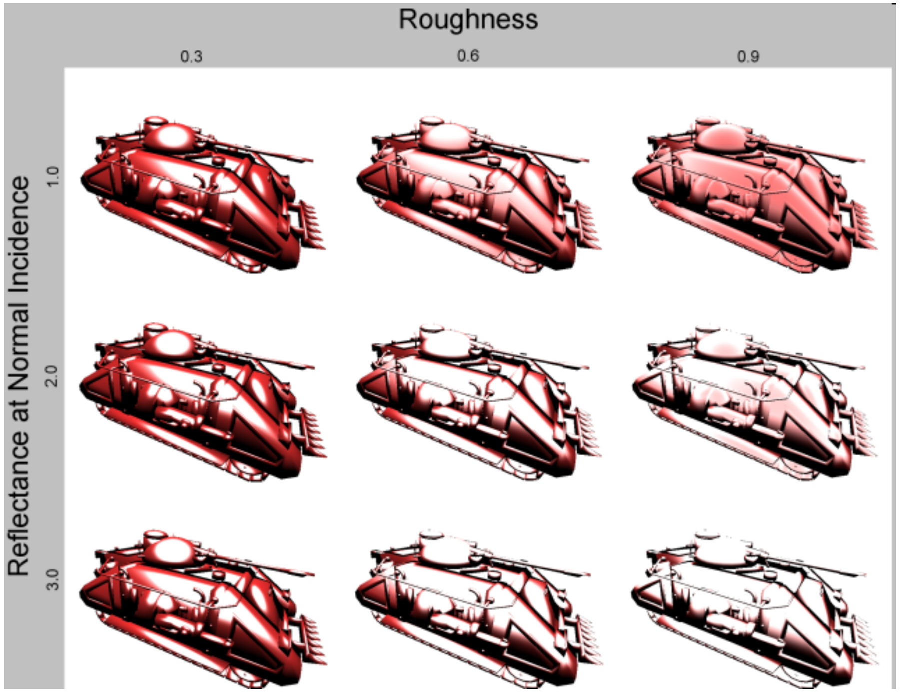
$$i_{spec} = \frac{F*R*G}{(N.V)*(N.L)}$$

$$G = \min(1, \frac{2(H.N)(V.N)}{V.H}, \frac{2(H.N)(L.N)}{V.H})$$

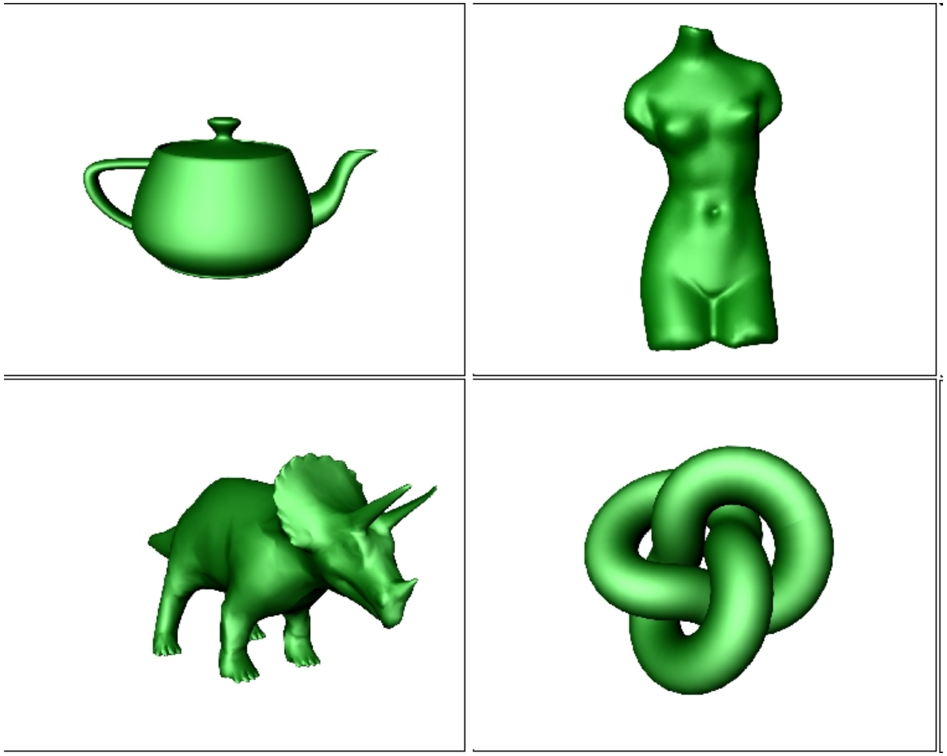
$$R = \frac{1}{m^2*(N.H)^4} * e^{\frac{(N.H)^2-1}{m^2*(N.H)^2}}$$

$$F = F_0 + (1-(H.V))^5 * (1-F_0)$$

Cook-Torrance model



[wiki.gamedev.net]



Materials

- f_r in rendering euqation – BRDF, BTF, …

$$L_o(\mathbf{x},\omega,\lambda,t) = L_e(\mathbf{x},\omega,\lambda,t) + \int_\Omega f_r(\mathbf{x},\omega',\omega,\lambda,t) L_i(\mathbf{x},\omega',\lambda,t)(-\omega'\cdot\mathbf{n})d\omega'$$

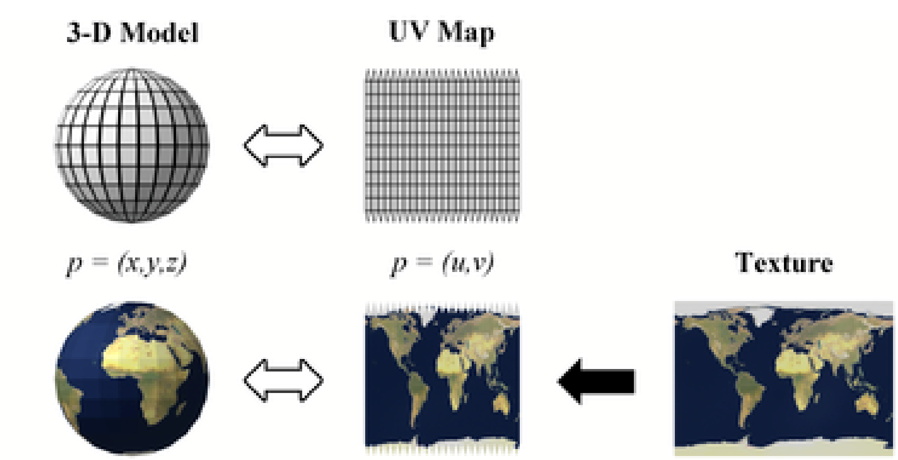
- Approximation using local illumination + materials – properties of surface in vertex
- Components (ambient, diffuse, specular, albedo, shininess, roughness, …)
- Given by value, procedure, texture, …

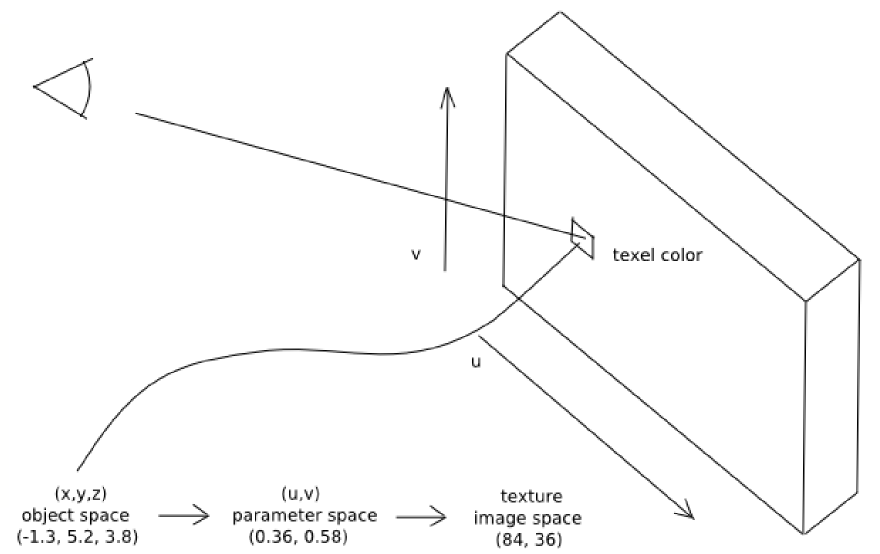
Textures

- Colors of material components stored in large arrays
  - Texture management: *glGenTextures*, *glBindTexture*
  - Texture data: *glTexImage*D*
  - Texture parameters: *glTexParameter**
- Mapping textures = texture coordinates = parameterization of surface
  - Setting coordinates: *glTexCoord**
- Texture application = per-fragment operation based on texture coordinates

Texture coordinates

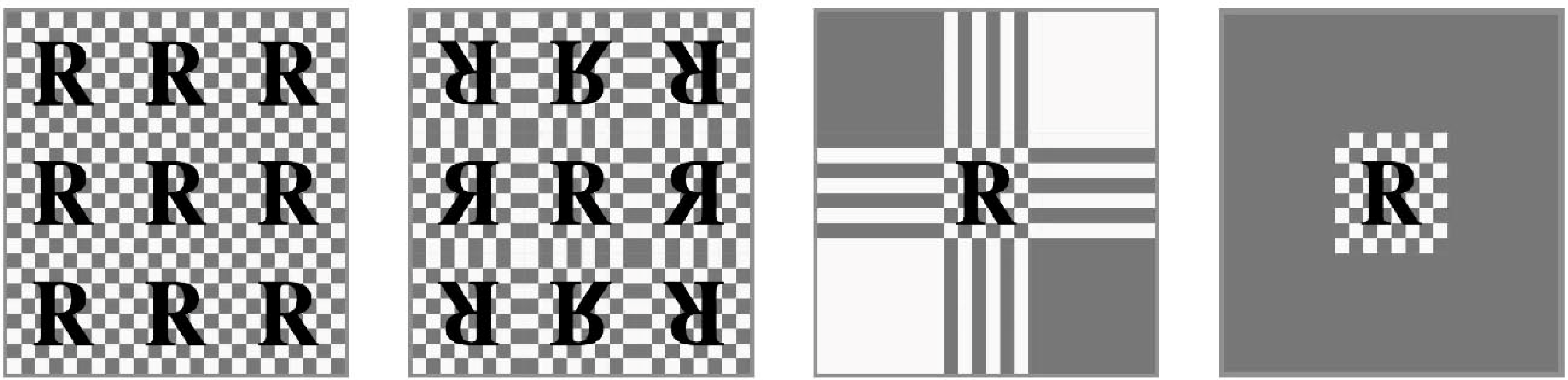
- Given for vertices, telling what is vertex "position" inside texture
- Automatic generation (spherical, planar,...)





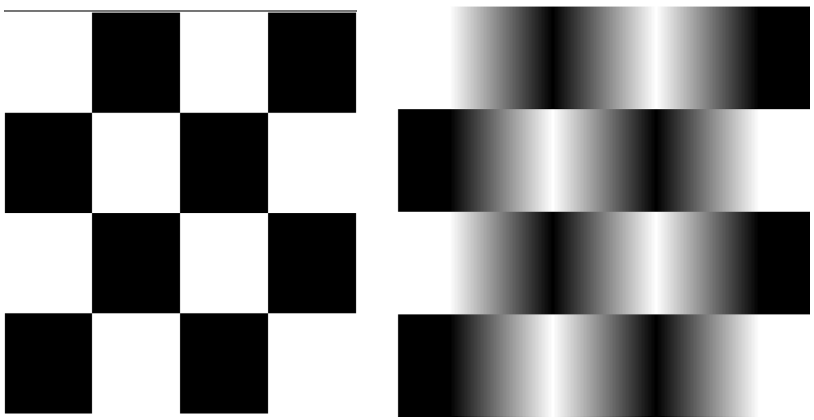
Texture wrap modes

- How to treat texture coordinates outside interval <0,1>
- Modes: repeat, mirror, clamp, border



Texture filtering

- What to do if fragment's texture coordinates are not exactly in the center of texel
- Nearest – take texel which center is nearest
- Linear – linear interpolation of 4 nearest texels
- Bicubic - shaders

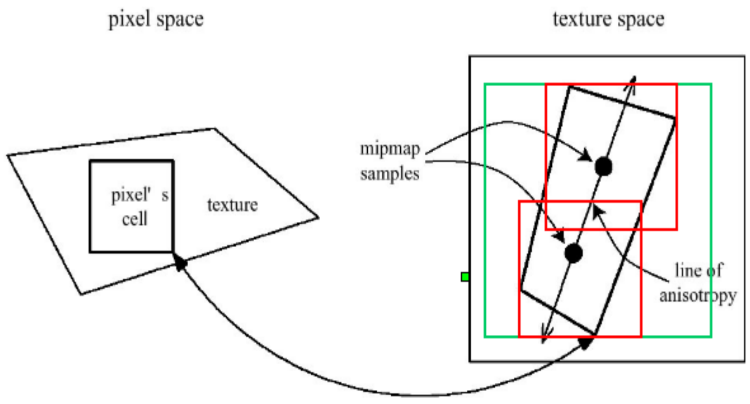


Texture mipmaping

- Undersampling when fetching from texture
- Use several levels of detail for texture
- When rendering, level = $\log_2(\mathrm{sqrt}(\mathrm{Area}))$
- Filtering also between mipmap levels

Anisotropic filtering

- Projecting pixels into texture space
- Taking samples, < 16, Vertical, horizontal
- GL_EXT_texture_filter_anisotropic



Texture compression

- Textures can occupy large part of memory
- Graphics cards – several compression algorithms for textures (S3TC, 3Dc, ...)
- Can be compressed on texture input
- Compression for normal map
- GL_ARB_texture_compression
- OpenGL 1.3

Textures - OpenGL

```
// create a texture object
GLuint textureId;
glGenTextures(1, &textureId);
glBindTexture(GL_TEXTURE_2D, textureId);

// set filtering
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
// enable mipmap generation
glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
// enable anisotropic filtering
GLfloat maximumAnisotropy;
glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &maximumAnisotropy);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, maximumAnisotropy);

// load texture data and tell system that we want use compressed texture, p is pointer to texture data in proper format
glTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGB_ARB, TEXTURE_WIDTH, TEXTURE_HEIGHT, 0, GL_RGB, GL_UNSIGNED_BYTE, p);

// check if texture is compressed
GLint isCompressed;
glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_COMPRESSED_ARB, &isCompressed);
If (isCompressed)
{
		// get compressed texture data
		Glint dataSize;
		glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_COMPRESSED_IMAGE_SIZE, &dataSize);
		unsigned char* compressedData = new unsigned char[dataSize];
		glGetCompressedTexImage(GL_TEXTURE_2D, 0, compressedData);
}
```

Multi-texturing

- Applying several textures to one primitive
- Set of texture coordinates for one vertex
  - *glMultiTexCoord2*ARB*
- Set of active texture objects – texture units
  - *glActiveTextureARB*
- Enable or disable texture units
  - *glClientActiveTextureARB*
- In shaders, sampler is actually texture unit

Multi-texturing - example

```
// create a texture object
GLuint texturesId[2];
glGenTextures(2, &texturesId);

// fill two textures, first texture is diffuse map
glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_2D, texturesId[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, TEXTURE_WIDTH, TEXTURE_HEIGHT, 0, GL_RGBA, GL_UNSIGNED_BYTE, pDiffuseMap);

// second texture is light map
glActiveTextureARB(GL_TEXTURE1_ARB);
glBindTexture(GL_TEXTURE_2D, texturesId[1]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, TEXTURE_WIDTH, TEXTURE_HEIGHT, 0, GL_RGBA, GL_UNSIGNED_BYTE, pLightMap);

// send to shader texture units, we know that texture unit 0 is diffuse map, and texture unit 1 is light map
Glint location = glGetUniformLocationARB(programObject, "diffuseMap");
glUniform1iARB(location, 1, 0);
location = glGetUniformLocationARB(programObject, "lightMap");
glUniform1iARB(location, 1, 1);

// …

// set active texture units
glClientActiveTextureARB(GL_TEXTURE0_ARB);
glClientActiveTextureARB(GL_TEXTURE1_ARB);

// render object with two mapped textures, they are using same texture coordinates
// …
```

Multi-texturing - example

Vertex shader:

```
varying vec2  vTexCoord;

void main(void)
{
        vTexCoord = vec2(gl_MultiTexCoord0);

        gl_Position = ftransform();
}
```

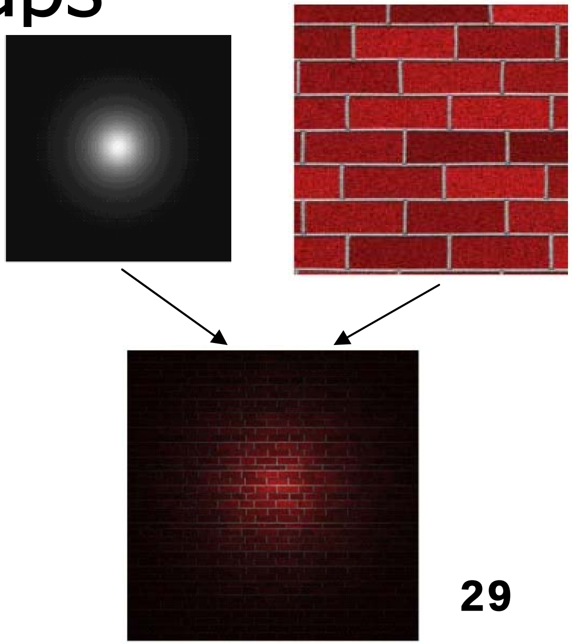
Fragment shader:

```
uniform sampler2D diffuseMap;
uniform sampler2D lightMap;
varying vec2  vTexCoord;

void main(void)
{
        vec4 diffuse = texture2D(baseMap, vTexCoord);
        vec4 light = texture2D(lightMap, vTexCoord);

        //gl_FragColor = clamp(diffuse + light, 0.0, 1.0);
        gl_FragColor = clamp(diffuse * light, 0.0, 1.0);

}
```

Light mapping

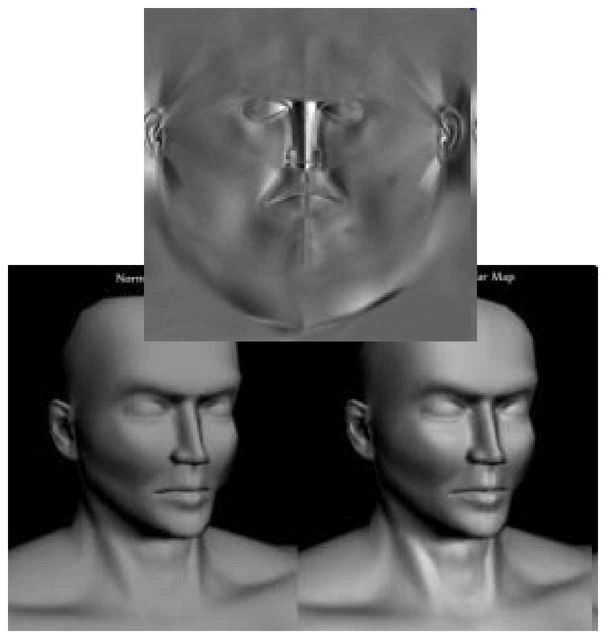
- Diffuse component is view independent
- Precomputed illumination for static lights
- Combination with surface, in separate maps or baked into diffuse maps

Original scene Light-mapped



Gloss & specular mapping

- Specular components of material stored in texture, gloss = shininess, specular = color & intensity of highlights

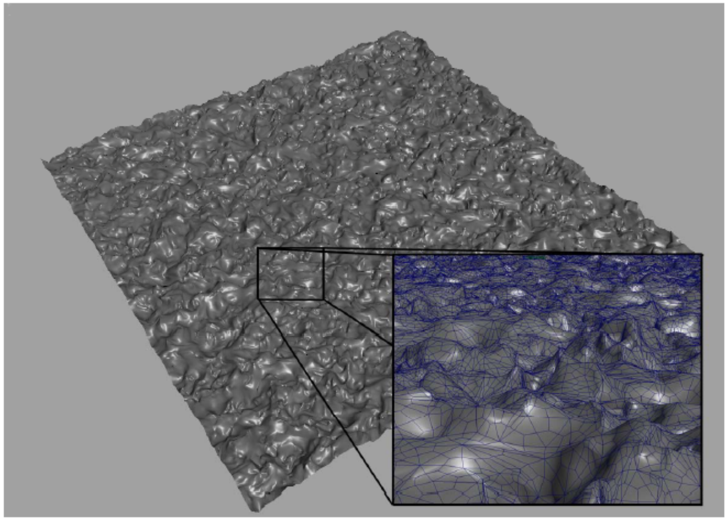


Alpha mapping

- Using alpha component from texture
- Using blending or alpha testing
- Adding transparency to scene – beware of ordering
- Billboards
- Animated

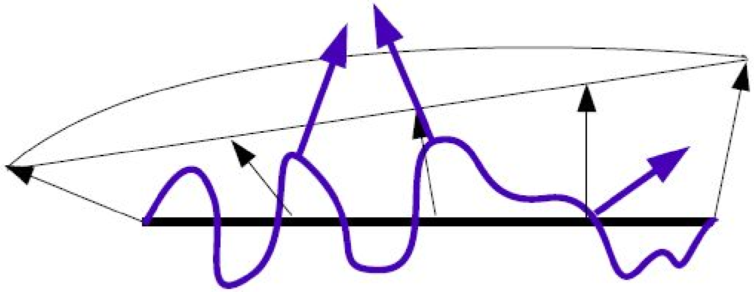
Rough surfaces

- Good geometrical approximation needs lots of triangles -> high bandwidth
- Solution:
  - Geometry in the form of textures
  - "Fake" illumination
  - Hardware tessellation



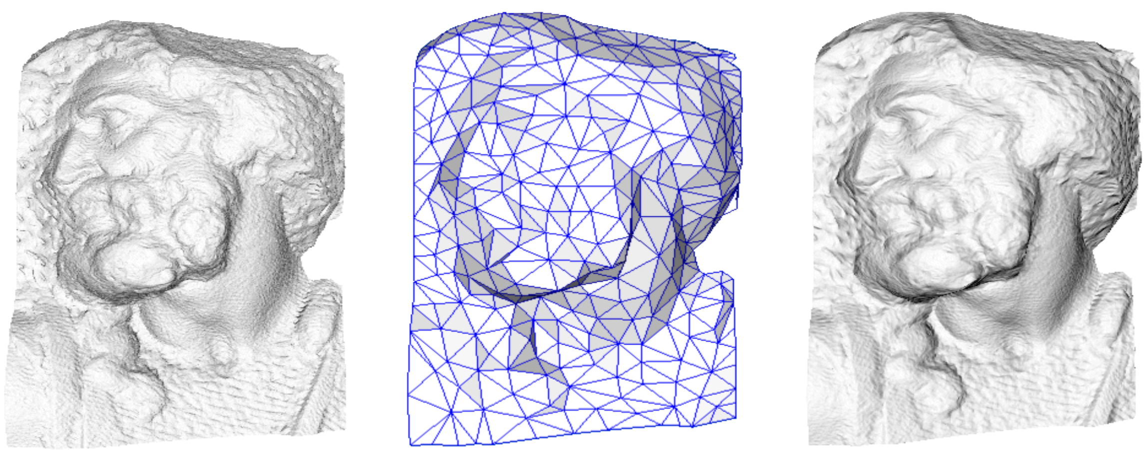
Bump mapping

- Normal is computed from height map, perturbing surface normal by height map normal
- Central differences for height map normal
- More computation, less memory



Normal mapping

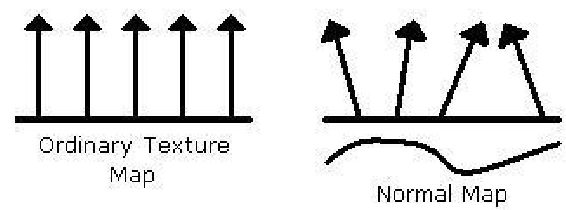
- Normal is stored in texture, 2 or 3 coordinates
- Coordinates normalization [0,1]->[-1,1]
- Normal is in UVW (tangent) space, but view and light vectors are in object (eye) space – conversion needed

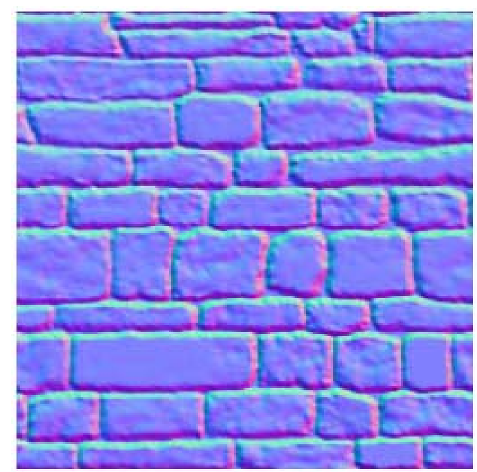


original mesh
4M triangles

simplified mesh
500 triangles

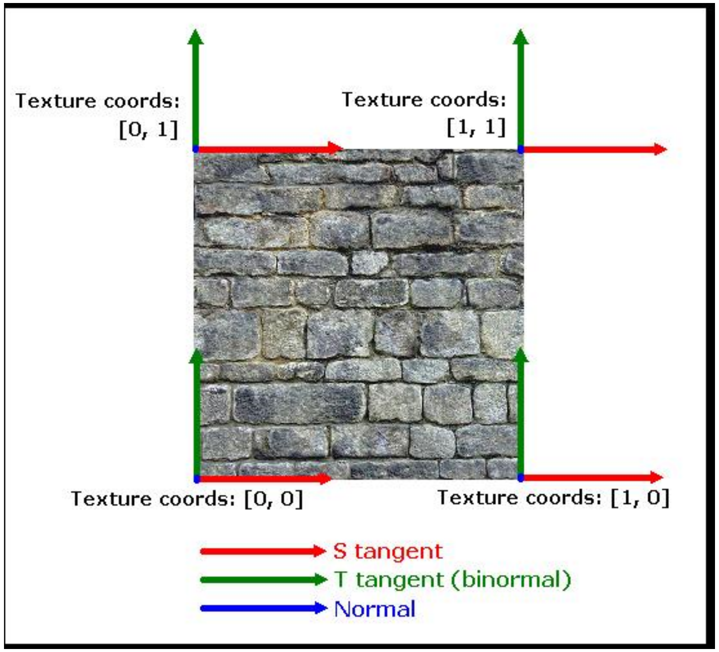
simplified mesh
and normal mapping
500 triangles

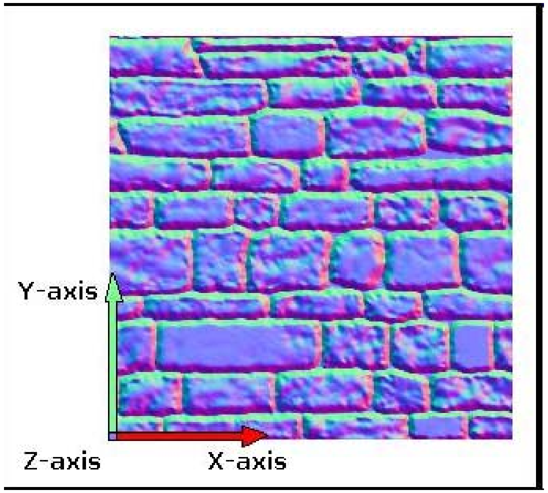


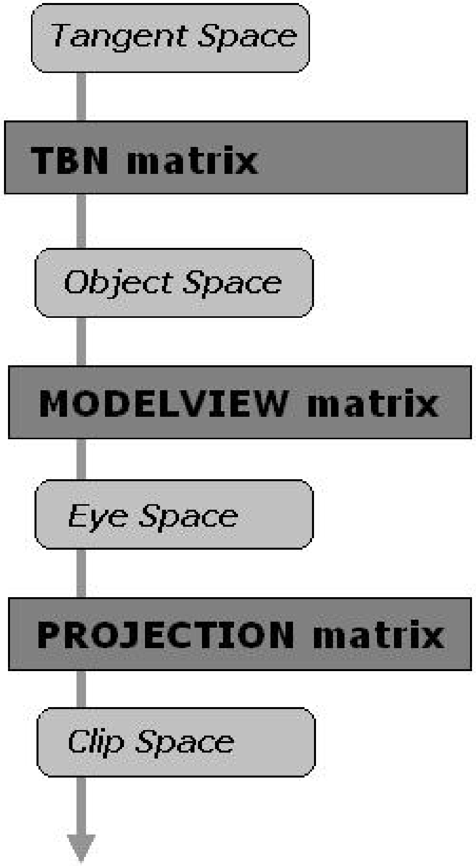


UVW (tangent) space

- Space of texture coordinates of object, using for fetching colors from textures







UVW to object space

- Given triangle ABC:
  - Vertices in object space: A-(x_0,y_0,z_0), B-(x_1,y_1,z_1), C-(x_2,y_2,z_2)
  - Texture coordinates: A-(u_0,v_0), B-(u_1,v_1), C-(u_2,v_2)
- We need transformation P such that
  - $P(u_0,v_0,0,1)=(x_0,y_0,z_0,1)$
  - $P(u_1,v_1,0,1)=(x_1,y_1,z_1,1)$
  - $P(u_2,v_2,0,1)=(x_2,y_2,z_2,1)$
- P is 4x4 matrix, its 3x3 top left submatrix Q has as columns vectors T,B,N
- (T,B,N) is base of UVW space; we only need (T,B,N), because we will transform only vectors - (x,y,z,0)

UVW to object space

- For vectors
  - $Q(u_1-u_0,v_1-v_0,0)=(x_1-x_0,y_1-y_0,z_1-z_0)$
  - $Q(u_2-u_0,v_2-v_0,0)=(x_2-x_0,y_2-y_0,z_2-z_0)$
- $Q=(T,B,N)=((T_0,T_1,T_2)^T, (B_0,B_1,B_2)^T, (N_0,N_1,N_2)^T)$
  - $T_0(u_1-u_0)+B_0(v_1-v_0)=x_1-x_0$
  - $T_1(u_1-u_0)+B_1(v_1-v_0)=y_1-y_0$
  - $T_2(u_1-u_0)+B_2(v_1-v_0)=z_1-z_0$
  - $T_0(u_2-u_0)+B_0(v_2-v_0)=x_1-x_0$
  - $T_1(u_2-u_0)+B_1(v_2-v_0)=y_1-y_0$
  - $T_2(u_2-u_0)+B_2(v_2-v_0)=z_1-z_0$
- N = TxB – cross product

GLSL – normal mapping

- Light computation in eye space

```
attribute vec3 vTangent;
attribute vec3 vBinormal;

uniform vec3 lightPos;

varying vec3 lightVec;
varying vec3 eyeVec;
varying vec2 texCoord;
varying vec3 t;
varying vec3 b;
varying vec3 n;

void main(void)
{
  gl_Position = ftransform();
  texCoord = gl_MultiTexCoord0.xy;

  t = gl_NormalMatrix * vTangent;
  b = gl_NormalMatrix * vBinormal;
  n = cross(t, b);

  vec3 vVertex = vec3(gl_ModelViewMatrix * gl_Vertex);
  lightVec = gl_LightSource[0].position – vVertex;
  eyeVec = -vVertex;
}
```

```
uniform sampler2D colorMap;
uniform sampler2D normalMap;
varying vec3 lightVec;
varying vec3 eyeVec;
varying vec2 texCoord;
varying vec3 t;
varying vec3 b;
varying vec3 n;

void main(void)
{
  vec3 vVec = normalize(eyeVec);
  vec3 lVec = normalize(lightVec);
  vec4 base = texture2D(colorMap, texCoord);
  vec3 bump = texture2D(normalMap, texCoord).xyz;
  vec3 tmpVec = normalize(2.0 * bump – 1.0);
  bump.x = dot(tmpVec, t);bump.y = dot(tmpVec, b);bump.z = dot(tmpVec, n);
  bump = normalize(bump);

  vec4 vAmbient = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
  float diffuse = max(dot(lVec, bump), 0.0);
  vec4 vDiffuse = gl_LightSource[0].diffuse * gl_FrontMaterial.diffuse * diffuse;
  float specular = pow(clamp(dot(reflect(-lVec, bump), vVec), 0.0, 1.0),
          gl_FrontMaterial.shininess );
  vec4 vSpecular = gl_LightSource[0].specular * gl_FrontMaterial.specular *
          specular;

  gl_FragColor = (vAmbient*base + vDiffuse*base + vSpecular);
}
```

GLSL – normal mapping

- Light computation in tangent space

```
attribute vec3 vInvTangent;
attribute vec3 vInvBinormal;
attribute vec3 vInvNormal;

varying vec3 lightVec;
varying vec3 eyeVec;
varying vec2 texCoord;

void main(void)
{
  gl_Position = ftransform();
  texCoord = gl_MultiTexCoord0.xy;

  vec3 vVertex = vec3(gl_ModelViewMatrix * gl_Vertex);

  // transform light vector from object coordinates to tangent space
  vec3 tmpVec = gl_LightSource[0].position.xyz - vVertex;
  tmpVec = gl_ModelViewMatrixInverse * vec4(tmpVec, 0.0);
  lightVec.x = dot(tmpVec, vInvTangent);
  lightVec.y = dot(tmpVec, vInvBinormal);
  lightVec.z = dot(tmpVec, vInvNormal);

  // transform view vector from object space to tangent space
  tmpVec = -gl_Vertex;
  eyeVec.x = dot(tmpVec, vInvTangent);
  eyeVec.y = dot(tmpVec, vInvBinormal);
  eyeVec.z = dot(tmpVec, vInvNormal);
}
```

```
uniform sampler2D colorMap;
uniform sampler2D normalMap;
varying vec3 lightVec;
varying vec3 eyeVec;
varying vec2 texCoord;
varying vec3 t;
varying vec3 b;
varying vec3 n;

void main(void)
{
  vec3 vVec = normalize(eyeVec);
  vec3 lVec = normalize(lightVec);
  vec4 base = texture2D(colorMap, texCoord);
  vec3 bump = texture2D(normalMap, texCoord).xyz;
  bump = normalize(2.0 * bump – 1.0);

 vec4 vAmbient = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
  float diffuse = max(dot(lVec, bump), 0.0);
  vec4 vDiffuse = gl_LightSource[0].diffuse * gl_FrontMaterial.diffuse * diffuse;
  float specular = pow(clamp(dot(reflect(-lVec, bump), vVec), 0.0, 1.0),
         gl_FrontMaterial.shininess );
  vec4 vSpecular = gl_LightSource[0].specular * gl_FrontMaterial.specular *
         specular;

  gl_FragColor = (vAmbient*base + vDiffuse*base + vSpecular);
}
```

Parallax mapping

- Displacing texture coordinates by a function of the view angle and the height map value
- More apparent depth, simulation of rays tracing against height fields
- Calculation:
  - s, b (scale, bias) - based on material
  - V – view vector in tangent space
  - h – value from height map
  - $h_n = s*h-b$
  - $T_n = T_0 + h_n * V.xy$
  - T_n – new texture coordinates

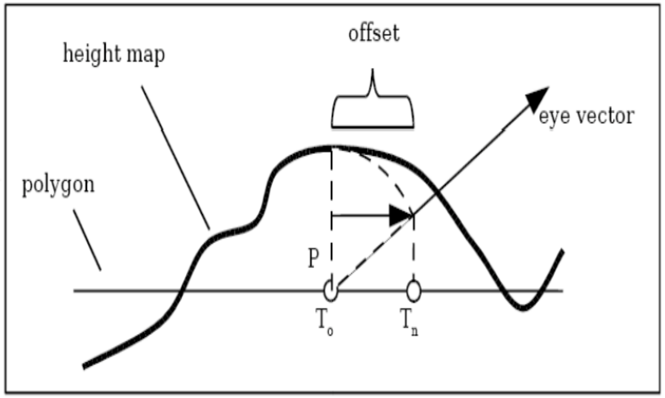


Image by Terry Welsh

GLSL – parallax mapping

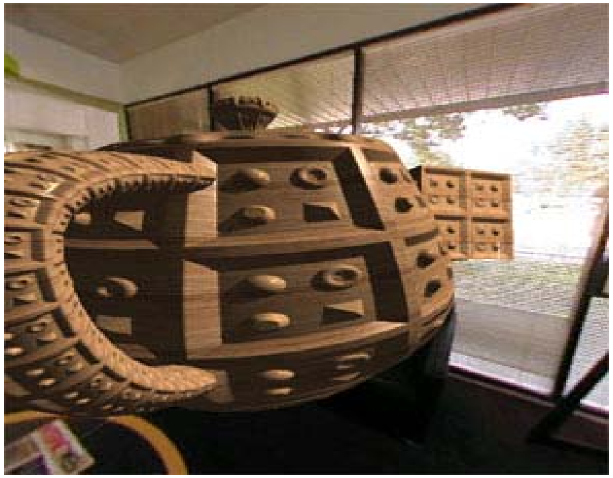
- ShaderDesigner
  - http://www.opengl.org/sdk/tools/ShaderDesigner/

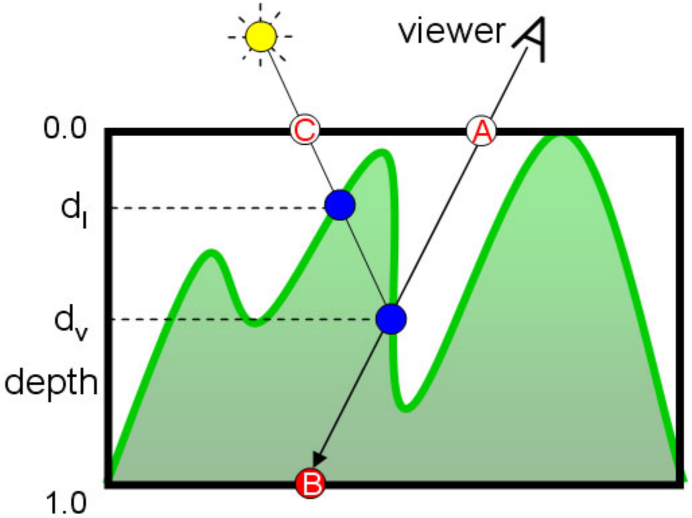
```
attribute vec3 tangent;
attribute vec3 binormal;
varying vec3 eyeVec;

void main()
{
    gl_TexCoord[0] = gl_MultiTexCoord0;

    // compute TBN matrix
    mat3 TBN_Matrix;
    TBN_Matrix[0] =  gl_NormalMatrix * tangent;
    TBN_Matrix[1] =  gl_NormalMatrix * binormal;
    TBN_Matrix[2] =  gl_NormalMatrix * gl_Normal;

    // transform view vector from eye coordinates to UVW coordinates
    vec4 Vertex_ModelView = gl_ModelViewMatrix * gl_Vertex;
    eyeVec = vec3(-Vertex_ModelView) * TBN_Matrix ;

    // default vertex transformation
    gl_Position = ftransform();
}
```

```
uniform vec2 scaleBias;
uniform sampler2D basetex;
uniform sampler2D bumptex;
varying vec3 eyeVec;

void main()
{
    vec2 texUV, srcUV = gl_TexCoord[0].xy;
    // fetch height from height map
    float height = texture2D(bumptex, srcUV).r;
     // add scale and bias to height
    float v = height * scaleBias.x - scaleBias.y;

    // normalize view vector
    vec3 eye = normalize(eyeVec);

    // add offset to texture coordinates
    texUV = srcUV + (eye.xy * v);
    // fetch texture color based on new coordinates
    vec3 rgb = texture2D(basetex, texUV).rgb;

    // output final color
   gl_FragColor = vec4(rgb, 1.0);
}
```

Relief mapping

- Extension of parallax mapping, inclusion of ray-tracing in the height map
- Self-shadowing, self-occlusion, silhouettes
- Various speed-up techniques



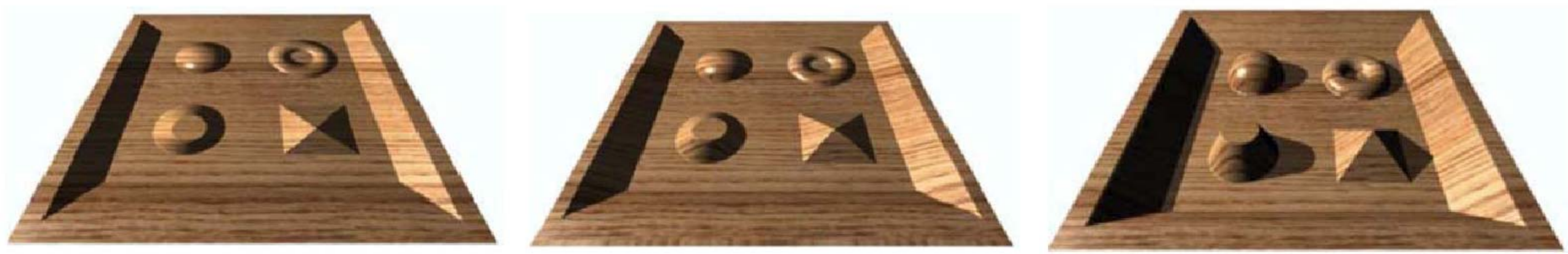


Comparison

Normal mapping

Parallax mapping

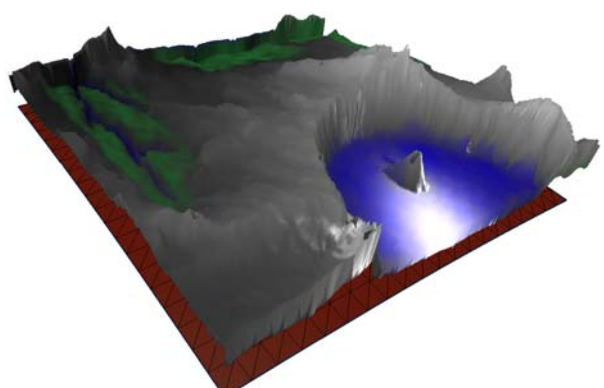
Relief mapping

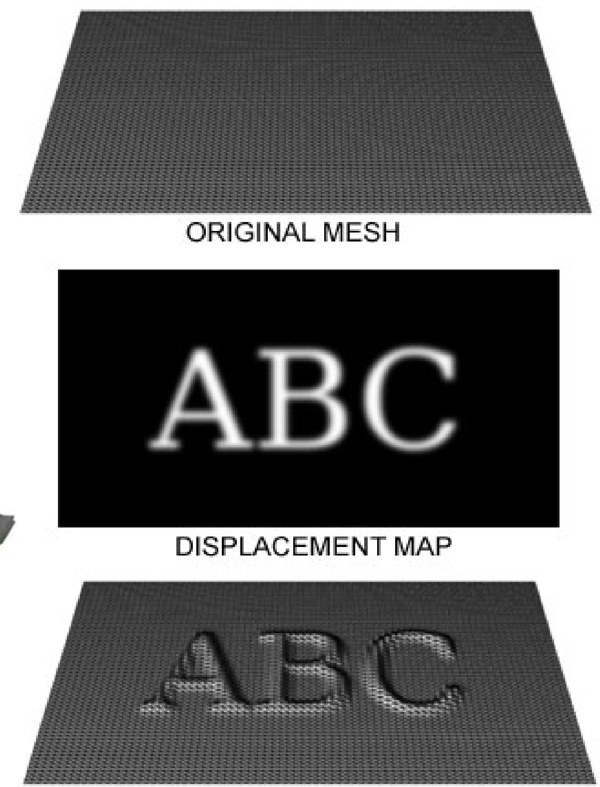


Texture mapping Parallax mapping Relief mapping

Displacement mapping

- Adding offset to vertex along vertex normal
- Offset is given in height map, or computed
- Costly technique
- New GPU
  - Tesselator shader
  - Automatic LOD





Sources

- Normal map generators
  - NVIDIA Melody - http://developer.nvidia.com/object/melody_home.html
  - nDo - http://www.cgted.com/
  - xNormal - http://www.xnormal.net
  - http://normalmapgenerator.yolasite.com/
- Light map generators
  - OGRE FSrad – http://www.ogre3d.org/tikiwiki/OGRE+FSRad
  - 3DS Max, Maya, Blender
  - irrEdit - http://www.ambiera.com/irredit/index.html
- Local illumination models comparison
  - http://www.labri.fr/perso/knoedel/cmsimple/?Work_Experience:DaimlerChrysler_AG

Questions?