Chapter XI
Normal Mapping
Bumpy Surfaces

- Image texturing only
  - Fast
  - Not realistic
- Highly tessellated mesh
  - Realistic
  - Slow
Surface Normal and Lighting

- Recall the interaction among light sources, surfaces, and camera/eye.

- Why light and why dark?

- Surface normals play key roles in lighting.
Normal Mapping

- Store the surface normals of the high-frequency surface into a texture, called the *normal map*.
- At run time, use the normal map for lighting.
**Height Field**

- A popular method to represent a high-frequency surface is to use a *height field*. It is a function $h(x,y)$ that returns a height or $z$ value given ($x,y$) coordinates.
- The height field is sampled with a 2D array of regularly spaced ($x,y$) coordinates, and the height values are stored in a texture named *height map*.
- The height map can be drawn in gray scales. If the height is in the range of $[0,255]$, the lowest height 0 is colored in black, and the highest 255 is colored in white.
Height Map for Terrain Rendering

(a)  
(b)  
(c)  
(d)  
(e)  

Computer Graphics with OpenGL ES (J. Han)
Normal Map

- Simple image-editing operations can create a gray-scale image (height map) from an image texture (from (a) to (b)).

- The next step from (b) to (c) is done automatically.
Normal Map (cont’d)

- Creation of a normal map from a height map

\[
\frac{\partial h}{\partial x} = \frac{h(x + 1, y) - h(x - 1, y)}{2} \\
\frac{\partial h}{\partial y} = \frac{h(x, y + 1) - h(x, y - 1)}{2}
\]

- Visualization of a normal map

\[
R = 255(0.5x + 0.5) \\
G = 255(0.5y + 0.5) \\
B = 255(0.5z + 0.5)
\]
Normal Mapping (cont’d)

- The polygon mesh is rasterized and texture coordinates \((s, t)\) are used to access the normal map.
- The normal at \((s, t)\) is obtained by filtering the normal map.
- Recall the diffuse reflection term, \(\max(n \cdot l, 0)s_d \otimes m_d\).
  - The normal \(n\) is fetched from the normal map.
  - \(m_d\) is fetched from the image texture.
Normal Mapping (cont’d)
Normal Mapping (cont’d)

```cpp
#version 300 es

uniform mat4 worldMat, viewMat, projMat;
uniform vec3 eyePos, lightPos;

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
layout(location = 2) in vec2 texCoord;

out vec2 v_texCoord;
out vec3 v_lightDir, v_viewDir;

void main() {
    vec4 posWS = worldMat * vec4(position, 1.0);
    gl_Position = projMat * viewMat * posWS;
    v_texCoord = texCoord;

    v_lightDir = normalize(lightPos - posWS.xyz);
    v_viewDir = normalize(eyePos - posWS.xyz);
}
```
Normal Mapping (cont’d)

```glsl
#version 300 es

precision mediump float;

uniform sampler2D s_tex0;
uniform sampler2D s_texNor;

uniform vec3 materialSpec, materialAmbi, materialEmit // Ms, Ma, Me
uniform float materialSh;
uniform vec3 sourceDiff, sourceSpec, sourceAmbi: // Sd, Ss, Sa

in vec2 v_texCoord;
in vec3 v_lightDir, v_viewDir:

out vec4 fragColor:

struct Material {
    float sh: // shininess (specular power)
    vec3 diff, spec, ambi, emit: // material colors: diffuse, specular,
    ambient, emission
};

struct Light {
    vec3 dir, diff, spec, ambi: // light and colors: diffuse, specular,
    ambient
};
```
Normal Mapping (cont’d)

```cpp
vec3 phongLight(vec3 view, vec3 normal, // view direction and normal
    Material M, Light S) { // material and source light
    float diff = max(dot(normal, S.dir), 0.0);
    vec3 refl = 2.0 * normal * dot(normal, S.dir) - S.dir;
    float spec = (diff > 0.0) ? pow(max(dot(refl, view), 0.0), M.sh) : 0.0;

    vec3 sum = vec3(0.0);
    sum += diff * S.diff * M.diff;
    sum += spec * S.spec * M.spec;
    sum += S.ambi * M.ambi + M.emit;
    return sum;
}

void main() {
    vec3 materialDiff = texture(s_tex0, v_texCoord).xyz;
    vec3 norTS = normalize(texture(s_texNor, v_texCoord).xyz * 2.0 - vec3(1.0));

    Material material =
        Material(materialSh, materialDiff, materialSpec, materialAmbi, materialEmit);

    Light source =
        Light(normalize(v_lightDir), sourceDiff, sourceSpec, sourceAmbi);

    vec3 color = phongLight(normalize(v_viewDir), normalize(norTS), material, source);

    fragColor = vec4(color, 1.0);
}
```
Normal Mapping (cont’d)
Tangent-space Normal Mapping

- Recall that texturing is described as wrapping a texture onto an object surface.
**Tangent-space Normal Mapping (cont’d)**

- A point at any surface has a surface normal. The normals of the normal map should be considered as perturbed instances of the *surface normals*.

- In the example, \( n(s_p,t_p) \) will replace \( n_p \), and \( n(s_q,t_q) \) will replace \( n_q \).
- A tricky problem to face: Normals are assumed to be defined at the world space, but does the normal map contain world-space normals? The answer is NO.
Tangent-space Normal Mapping (cont’d)

- For a surface point, consider a tangent space, the $z$-axis of which corresponds to the surface normal.

  ![Diagram](c) Tangent spaces

  ![Diagram](d) Normal map normals in the tangent spaces

- Assuming that a tangent space is defined for a surface point to be normal-mapped, the normal fetched from the normal map is taken as defined in the tangent space of the point, not in the world space.
- In this respect, the normal is named the tangent-space normal.
**Tangent Space**

- The basis of the tangent space \( \{T,B,N\} \)
  - Vertex normal \( N \) – defined per vertex at the modeling stage.
  - Tangent \( T \) – needs to compute
  - Binormal \( B \) – needs to compute
void ImlRenderer::ComputeTangent() {
    vector<vec3> triTangents;
    // Compute Tangent Basis
    for (int i=0; i< mIndexArray.size(); i += 3) {
        vec3 p0 = mVertexArray.at(mIndexArray.at(i)).pos;
        vec3 p1 = mVertexArray.at(mIndexArray.at(i+1)).pos;
        vec3 p2 = mVertexArray.at(mIndexArray.at(i+2)).pos;
        vec3 uv0 = vec3(mVertexArray.at(mIndexArray.at(i)).tex, 0);
        vec3 uv1 = vec3(mVertexArray.at(mIndexArray.at(i+1)).tex, 0);
        vec3 uv2 = vec3(mVertexArray.at(mIndexArray.at(i+2)).tex, 0);
        vec3 deltaPos1 = p1 - p0;
        vec3 deltaPos2 = p2 - p0;
        vec3 deltaUV1 = uv1 - uv0;
        vec3 deltaUV2 = uv2 - uv0;

        // Compute the tangent
        float r = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV1.y * deltaUV2.x);
        vec3 computedTangent = (deltaPos1 * deltaUV2.y - deltaPos2 * deltaUV1.y) * r;
        triTangents.push_back(computedTangent);
    }
    // Initialize mTangents
    for (int i=0; i < mVertexArray.size(); ++i)
        mTangentArray.push_back(vec3(0));
    // Accumulate tangents by indices
    for (int i=0; i < mIndexArray.size(); ++i) {
        mTangentArray.at(mIndexArray.at(i)) = mTangentArray.at(mIndexArray.at(i)) + triTangents.at(i);
    }
}
Tangent-space Normal Mapping (cont’d)

- The diffuse term of the Phong lighting model

\[
\text{max}(n \cdot l, 0) s_d \otimes m_d + (\text{max}(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_c
\]

- A light source is defined in the world space, and so is \( l \). In contrast, \( n \) fetched from the normal map is defined in the tangent space. To resolve this inconsistency, \( n \) has to be transformed into the world space.

- Typically, the per-vertex TBN-basis is pre-computed, is stored in the vertex array and is passed to the fragment shader. Then, the fragment shader constructs a matrix that rotates the world-space light vector into the per-vertex tangent space.

(f) Light vectors transformed into the tangent spaces
Tangent-space Normal Mapping (cont’d)

```cpp
#version 300 es

uniform mat4 worldMat, viewMat, projMat;
uniform vec3 eyePos, lightPos;

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
layout(location = 2) in vec2 texCoord;
layout(location = 3) in vec3 tangent;

out vec2 v_texCoord;
out vec3 v_lightTS, v_viewTS;

void main() {
    vec4 posWS = worldMat * vec4(position, 1.0);
    gl_Position = projMat * viewMat * posWS;
    v_texCoord = texCoord;

    vec3 nor = mat3(worldMat) * normal;
    vec3 tan = mat3(worldMat) * normalize(tangent);
    vec3 bin = cross(nor, tan);
    mat3 worldToTangentSpace = mat3(tan.x, bin.x, nor.x,
                                    tan.y, bin.y, nor.y,
                                    tan.z, bin.z, nor.z);

    v_lightTS = worldToTangentSpace * (lightPos - posWS.xyz);
    v_viewTS = worldToTangentSpace * (eyePos - posWS.xyz);
}
```
Tangent-space Normal Mapping (cont’d)

```glsl
#version 300 es

precision mediump float;

uniform sampler2D s_tex0;
uniform sampler2D s_texNor;

uniform vec3 materialSpec, materialAmbi, materialEmit; // Ms, Ma, Me
uniform float materialSh;
uniform vec3 sourceDiff, sourceSpec, sourceAmbi; // Sd, Ss, Sa

in vec2 v_texCoord;
in vec3 v_lightTS, v_viewTS;

out vec4 fragColor;

struct Material {
  float sh;
  vec3 diff, spec, ambi, emit;
};

struct Light {
  vec3 dir, diff, spec, ambi;
};
```
Tangent-space Normal Mapping (cont’d)

```c
vec3 phongLight(vec3 view, vec3 normal, vec3 S, M, dir, light) {
    float diff = max(dot(normal, S, dir), 0.0);
    vec3 refl = 2.0 * normal * dot(normal, S, dir) - S, dir;
    float spec = (diff > 0.0) ? pow(max(dot(refl, view), 0.0), M, sh) : 0.0;

    vec3 sum = vec3(0, 0, 0);
    sum += diff * S, S, diff * M, M, diff;
    sum += spec * S, spec * M, spec;
    sum += S, ambL * M, ambL * M, emit;
    return sum;
}

void main() {
    vec3 materialDiff = texture(s Tex0, v_texCoord), xyz;
    vec3 normal = normalize(texture(s TexNor, v_texCoord), xyz * 2.0 - vec3(1.0));

    Material material =
        Material(materialSh, materialDiff, materialSpec, materialAmbi, materialEmit);

    Light source =
        Light(normalize(v_lightTS), sourceDiff, sourceSpec, sourceAmbi);

    vec3 color = phongLight(normalize(v_viewTS), normalize(normalTS), material, source);
    fragColor = vec4(color, 1.0);
}
```
Tangent-space Normal Mapping (cont’d)
Tangent-space Normal Mapping (cont’d)

- Unreal demo at 2005.