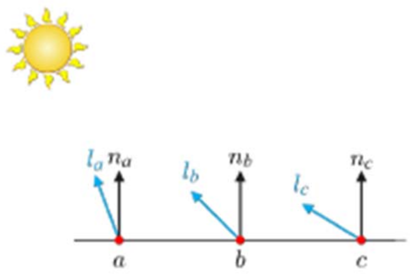
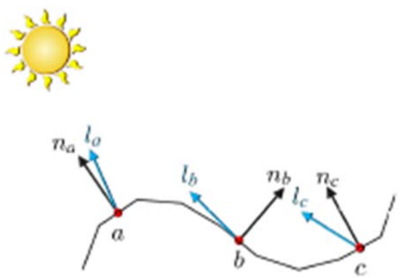
Chapter XIV
Normal Mapping

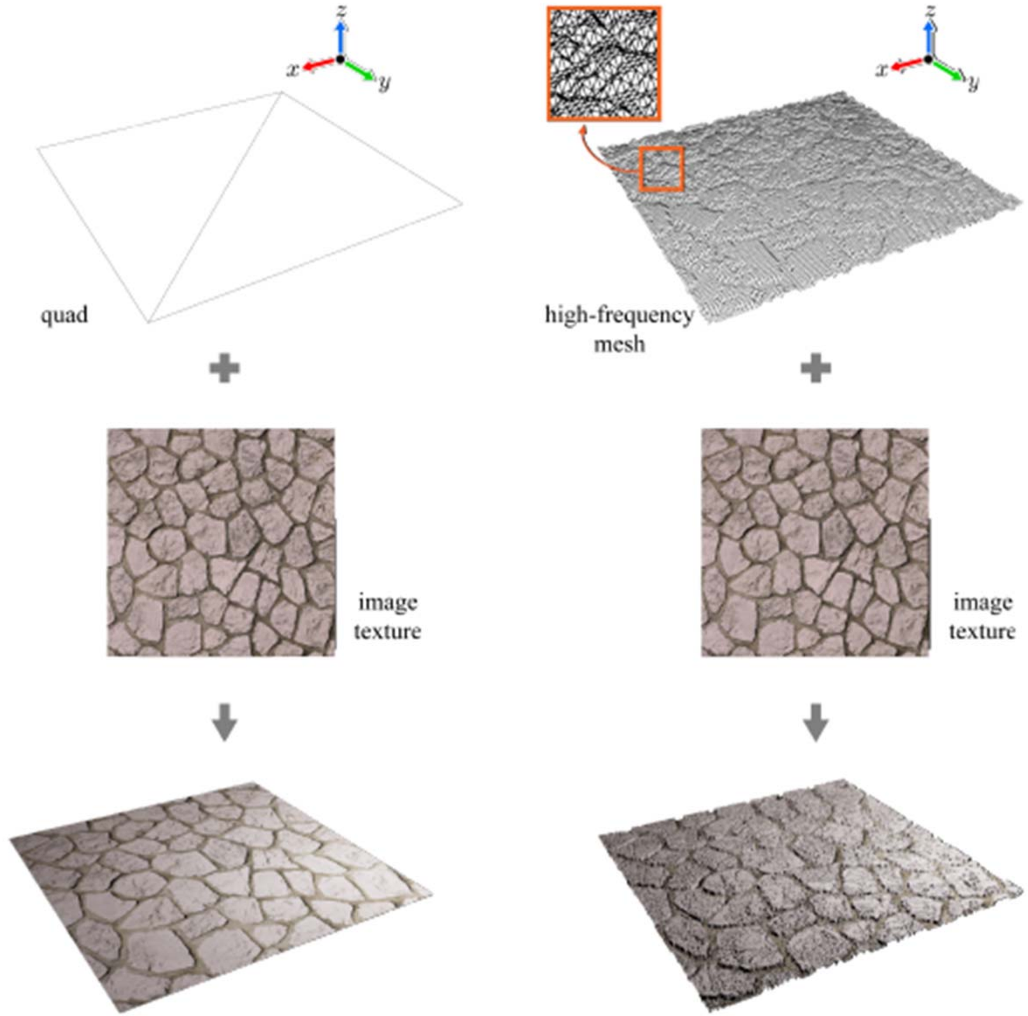
Bumpy Surfaces

- Image texturing only
  - Fast
  - Not realistic



- Highly tessellated mesh
  - Realistic
  - Slow

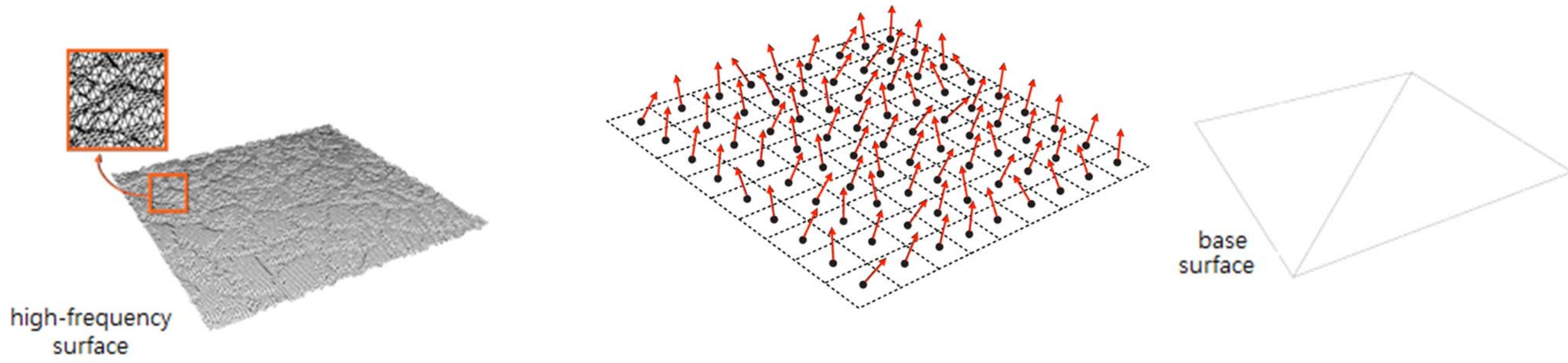


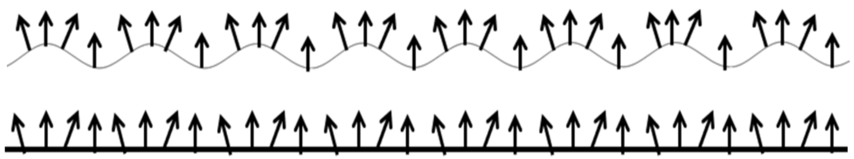


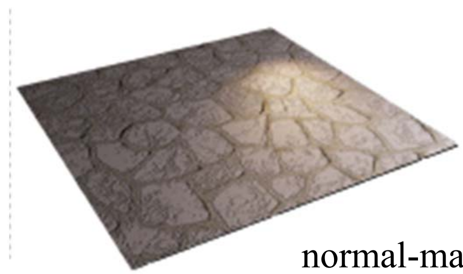
- Surface normals play key roles in lighting.

Normal Mapping

- A way out of this dilemma is
  - to pre-compute and store the normals of the high-frequency surface into a special texture named *normal map*, and
  - to use a lower-resolution mesh at run time which we call *base surface* and fetch the normals from the normal map for lighting.



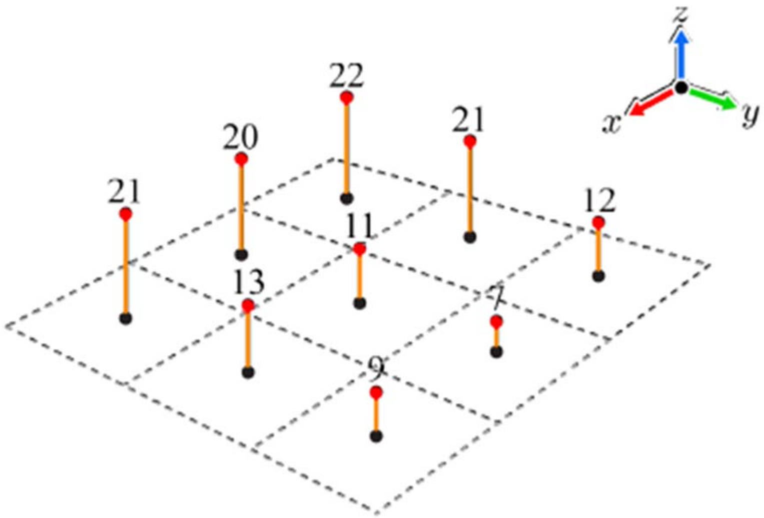




normal-mapped
base surface

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.



Normal Map

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

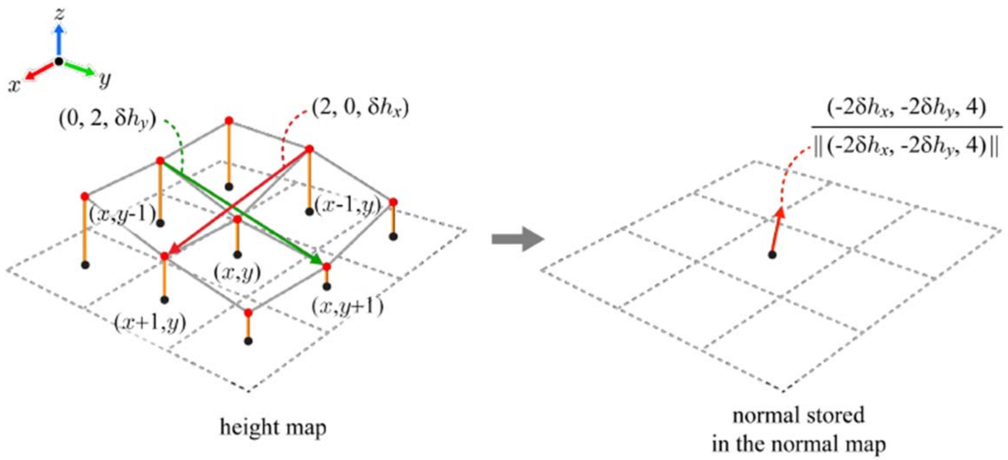
(a) (b) (c)

- The next step from (b) to (c) is done automatically.

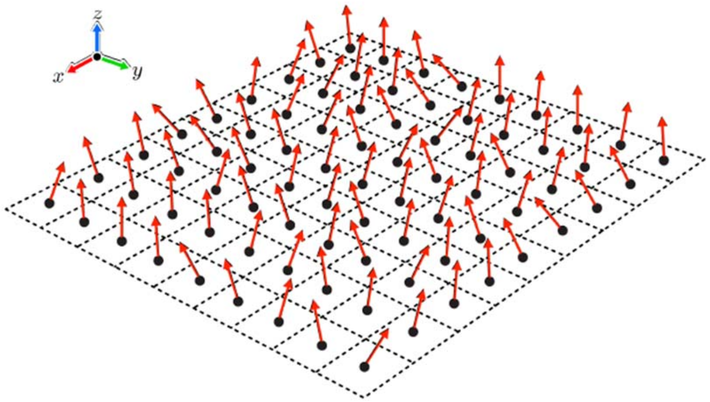
Normal Map (cont'd)

- Creation of a normal map from a height map

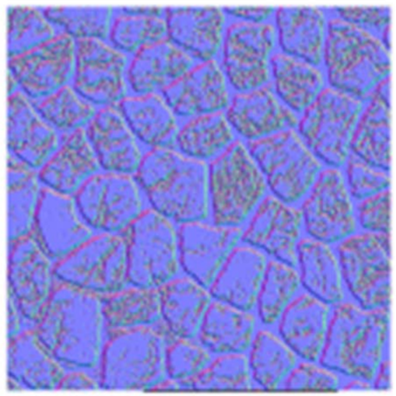
$$\delta h_x = h(x+1,y) - h(x-1,y)$$
$$\delta h_y = h(x,y+1) - h(x,y-1)$$



- Visualization of a normal map

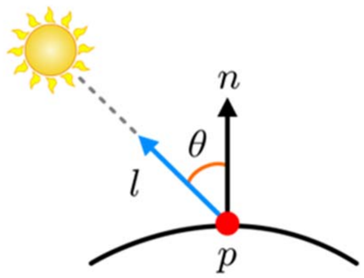


$$R = 255(0.5x + 0.5)$$
$$G = 255(0.5y + 0.5)$$
$$B = 255(0.5z + 0.5)$$



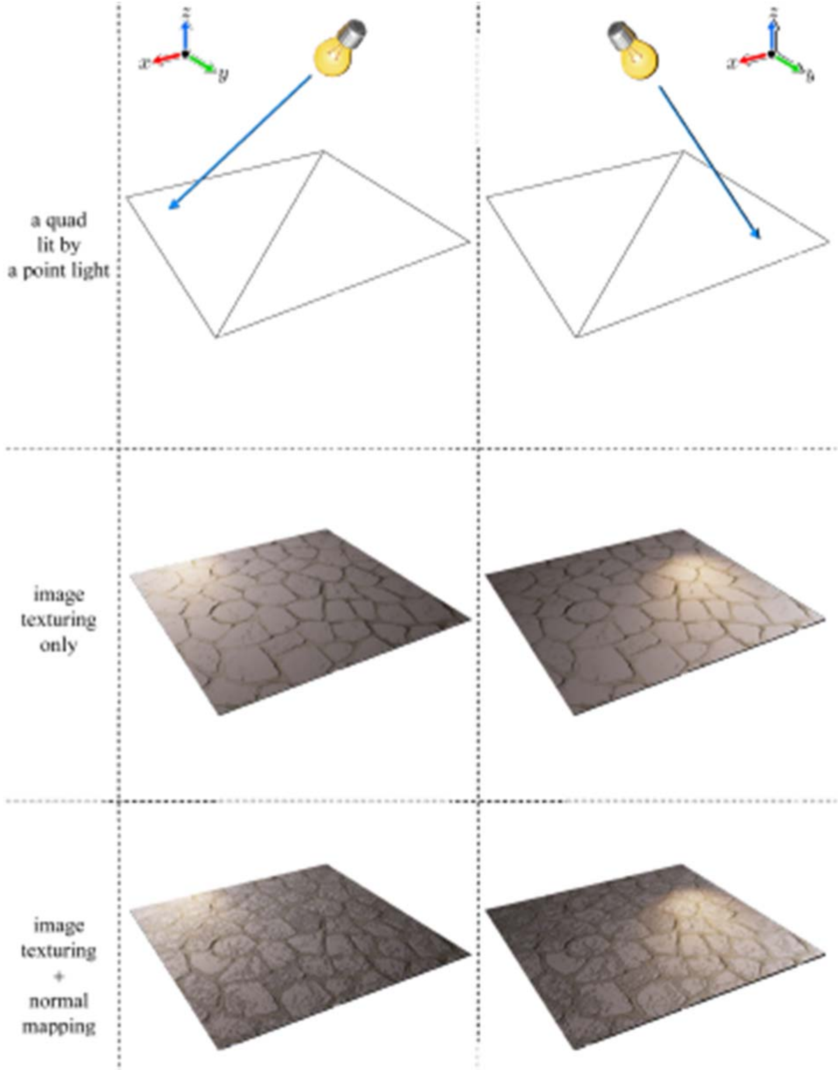
Normal Mapping (cont'd)

- 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, $max(n \bullet l,0)s_d \otimes m_d$.
  - The normal n is fetched from the normal map.
  - m_d is fetched from the image texture.



$$\boxed{max(n \cdot l, 0)s_d \otimes m_d} + (max(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e$$

Normal Mapping (cont'd)

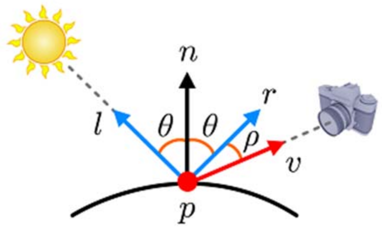


Normal Mapping (cont'd)

- Vertex shader

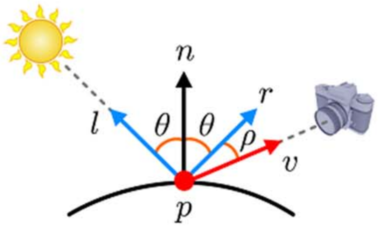
$$\boxed{max(n \cdot l, 0)s_d \otimes m_d + (max(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e}$$

```
#version 300 es

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

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

out vec2 v_texCoord;
out vec3 v_view;

void main() {
    vec3 worldPos = (worldMat * vec4(position, 1.0)).xyz;
    v_view = normalize(eyePos - worldPos);
    v_texCoord = texCoord;
    gl_Position = projMat * viewMat * vec4(worldPos, 1.0);
}
```



Normal Mapping (cont'd)

- Fragment shader $\boxed{max(n \cdot l, 0)s_d \otimes m_d + (max(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e}$



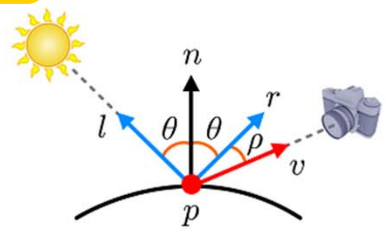

```
#version 300 es

precision mediump float;

uniform sampler2D colorMap;
uniform sampler2D normalMap;
uniform vec3 matSpec, matAmbi, matEmit; // Ms, Ma, Me
uniform float matSh; // shininess
uniform vec3 srcDiff, srcSpec, srcAmbi; // Sd, Ss, Sa
uniform vec3 lightDir; // directional light

in vec3 v_view;
in vec2 v_texCoord;

layout(location = 0) out vec4 fragColor;
```

Normal Mapping (cont'd)

- Fragment shader

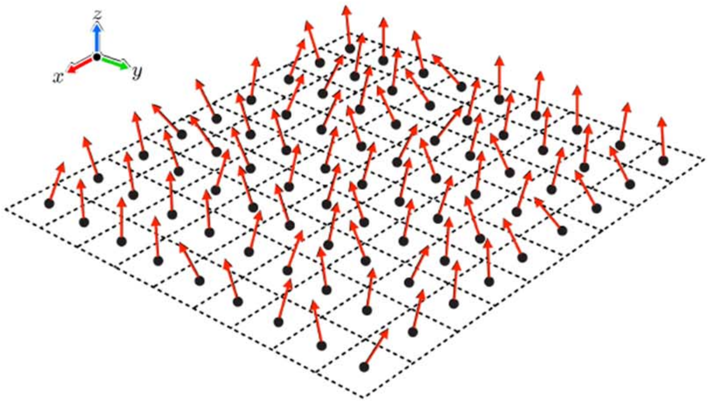
$$\boxed{max(n \cdot l, 0)s_d \otimes m_d + (max(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e}$$

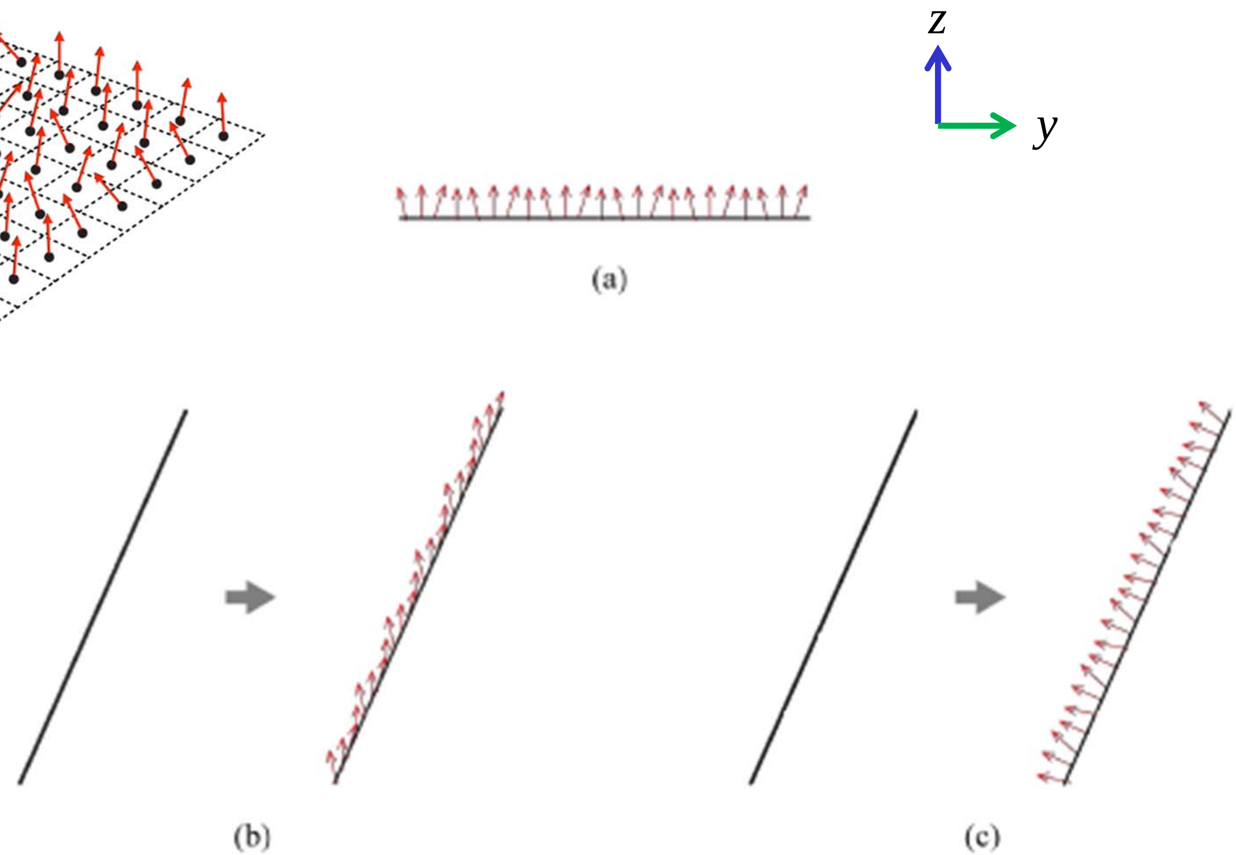
```
void main() {

    vec3 normal = 2.0 * texture(normalMap, v_texCoord).xyz - 1.0;
    vec3 view = normalize(v_view);
    vec3 light = normalize(lightDir);

    // diffuse term
    vec3 matDiff = texture(colorMap, v_texCoord).rgb;
    vec3 diff = max(dot(normal, light), 0.0) * srcDiff * matDiff;

    // specular term
    vec3 refl = 2.0 * normal * dot(normal, light) - light;
    vec3 spec = pow(max(dot(refl, view), 0.0), matSh) * srcSpec * matSpec;

    // ambient term
    vec3 ambi = srcAmbi * matAmbi;

    fragColor = vec4(diff + spec + ambi + matEmit, 1.0);
}
```



Tangent-space Normal Mapping

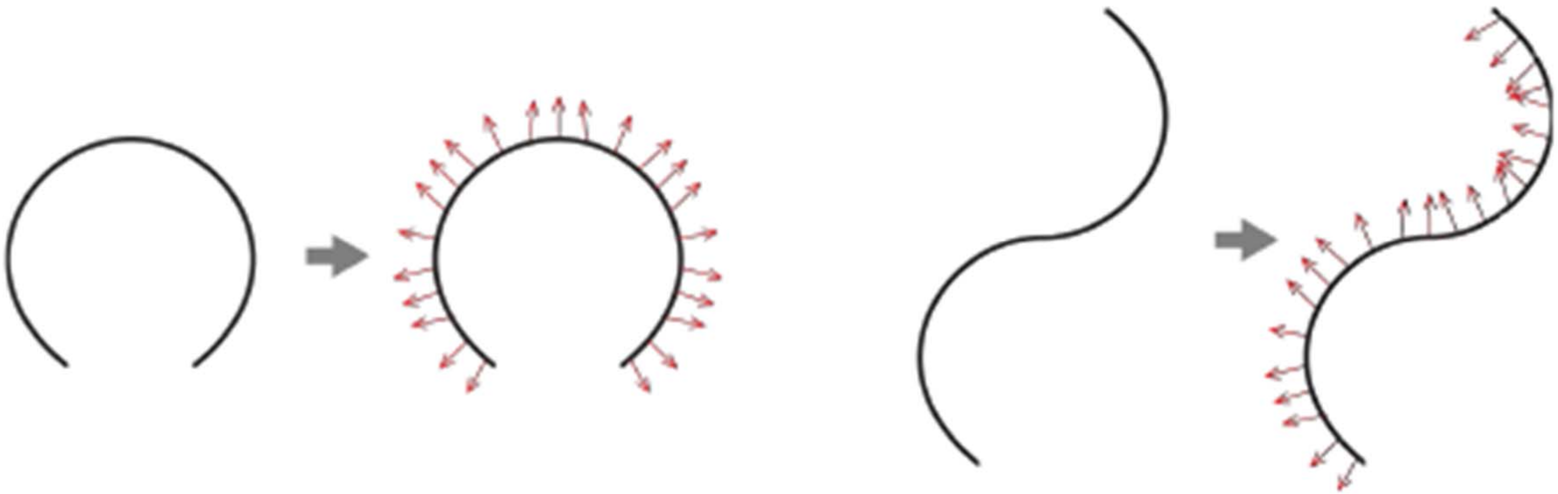
- Recall that texturing is described as wrapping a texture onto an object surface.
- When (a) is given, (b) is incorrect, but (c) is correct.





Tangent-space Normal Mapping (cont'd)

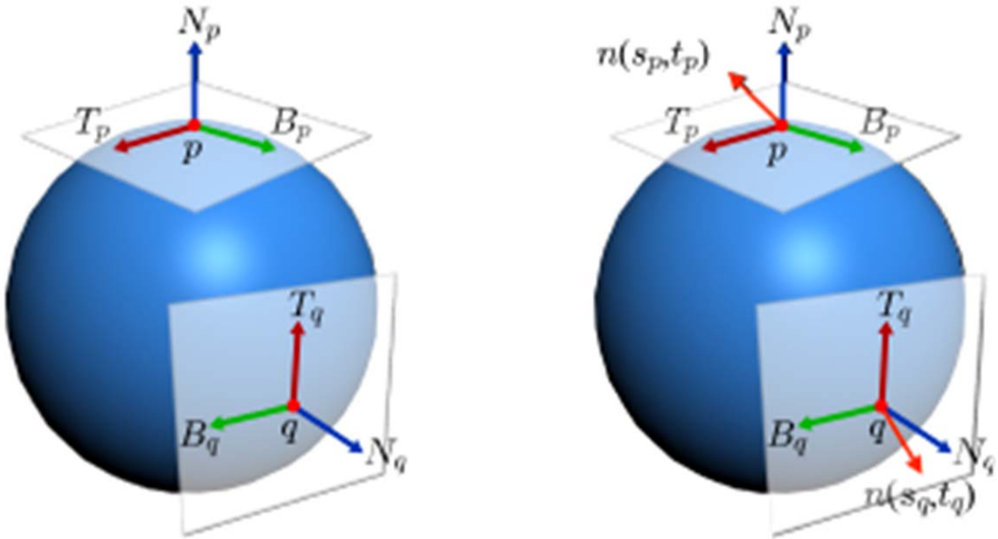
- Other examples.



- The normals fetched from the normal map should replace the surface normals orthogonal to the tangent planes.

Tangent-space Normal Mapping (cont'd)

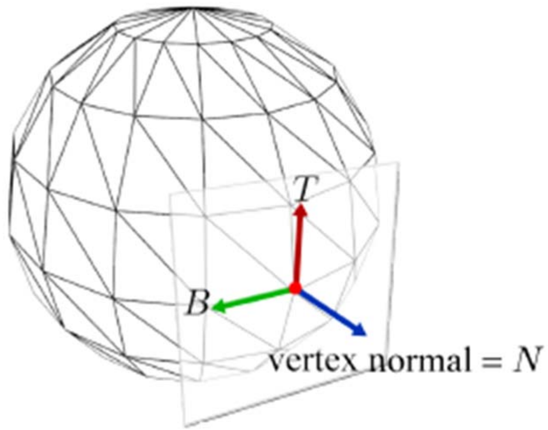
- For a surface point, consider a *tangent space* that is defined by three orthonormal vectors:
  - *T* (for tangent)
  - *B* (for binormal/bitangent)
  - *N* (for normal)



- *T* and *B* lie in the tangent plane, to which *N* is orthogonal.
- The tangent spaces vary across the object surface.
- Assuming that a tangent space can be always defined at a surface point to be normal-mapped, the normal fetched from the normal map is taken as being defined in the tangent space of the point, thereby replacing the surface normal *N*. 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



- There are many utility functions that compute T and B on each vertex of a polygon mesh.

Tangent Space (cont'd)

```
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);
        triTangents.push_back(computedTangent);
        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);
    }
}
```

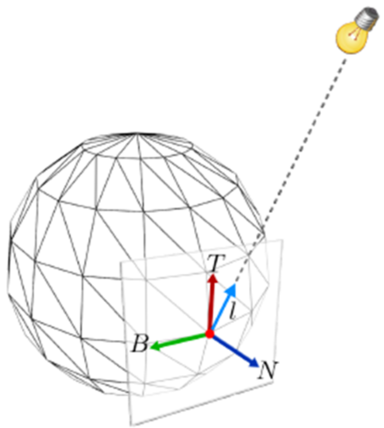
Tangent-space Normal Mapping (cont'd)

- Consider the diffuse term of the Phong lighting model

$$\boxed{max(n\cdot l,0)s_d \otimes m_d} + (max(r\cdot v,0))^{sh}s_s \otimes m_s + s_a \otimes m_a + m_e$$

- 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, or *l* has to be transformed into the tangent space.
- Typically, the per-vertex *TBN*-basis is pre-computed, is stored in the vertex array and is passed to the vertex shader.
- The vertex shader first transforms *T*, *B*, and *N* into the world space and then constructs a matrix that rotates the world-space light vector into the per-vertex tangent space.

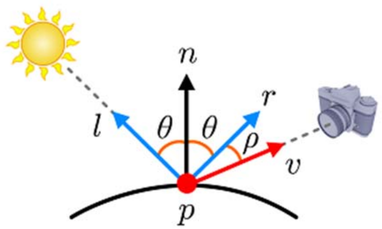
$$\begin{pmatrix} \boxed{T_x \;\; T_y \;\; T_z} \\ \boxed{B_x \;\; B_y \;\; B_z} \\ \boxed{N_x \;\; N_y \;\; N_z} \end{pmatrix}$$



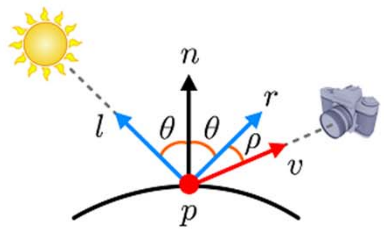
Tangent-space Normal Mapping (cont'd)

```
#version 300 es

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

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

out vec2 v_texCoord;
out vec3 v_viewTS, v_lightTS;

void main() {

    vec3 worldPos = (worldMat * vec4(position, 1.0)).xyz;

    vec3 Nor = normalize(transpose(inverse(mat3(worldMat))) * normal);
    vec3 Tan = normalize(transpose(inverse(mat3(worldMat))) * tangent);
    vec3 Bin = cross(Nor, Tan);
    mat3 tbnMat = transpose(mat3(Tan, Bin, Nor)); // row major

    v_lightTS = tbnMat * normalize(lightDir);
    v_viewTS = tbnMat * normalize(eyePos - worldPos);

    v_texCoord = texCoord;
    gl_Position = projMat * viewMat * vec4(worldPos, 1.0);
}
```

Observe that `normal` is used for defining the transform to the tangent space and is not output.



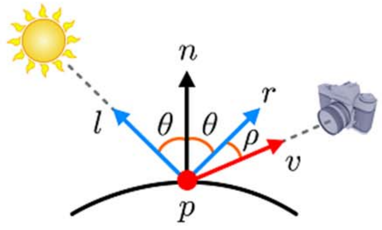
Tangent-space Normal Mapping (cont'd)




```
#version 300 es

precision mediump float;

uniform sampler2D colorMap;
uniform sampler2D normalMap;
uniform vec3 matSpec, matAmbi, matEmit; // Ms, Ma, Me
uniform float matSh; // shininess
uniform vec3 srcDiff, srcSpec, srcAmbi; // Sd, Ss, Sa

in vec3 v_viewTS, v_lightTS;
in vec2 v_texCoord;

layout(location = 0) out vec4 fragColor;
```

Tangent-space Normal Mapping (cont'd)

```
void main() {

    vec3 normal = 2.0 * texture(normalMap, v_texCoord).xyz - 1.0;
    vec3 view = normalize(v_viewTS);
    vec3 light = normalize(v_lightTS);

    // diffuse term
    vec3 matDiff = texture(colorMap, v_texCoord).rgb;
    vec3 diff = max(dot(normal, light), 0.0) * srcDiff * matDiff;

    // specular term
    vec3 refl = 2.0 * normal * dot(normal, light) - light;
    vec3 spec = pow(max(dot(refl, view), 0.0), matSh) * srcSpec * matSpec;

    // ambient term
    vec3 ambi = srcAmbi * matAmbi;

    fragColor = vec4(diff + spec + ambi + matEmit, 1.0);
}
```



Tangent-space Normal Mapping (cont'd)