Bump Mapping

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University
What is Bump-Mapping?

Bump-mapping is the process of creating the illusion of 3D depth by using a manipulated surface normal in the lighting, rather than actually creating the extra surface detail. You saw this before in RenderMan like this:

Displacement-mapped

\[ P = P + \text{normalize}(N) \times \text{disp}; \]
\[ N = \text{calculatenormal}(P); \]

Bump-mapped

\[ N = \text{calculatenormal}( P + \text{normalize}(N) \times \text{disp} ); \]
The Most Straightforward Type of Bump-Mapping is *Height Fields*
Definition of Height Fields -- Think of the Pin Box!
terrain.vert

```cpp
#version 330 compatibility
out vec3 vMCposition;
out vec3 vECposition;
out vec2 vST;

void main() {
    vST = gl_MultiTexCoord0.st;
    vMCposition = gl_Vertex.xyz;
    vECposition = ( gl_ModelViewMatrix * gl_Vertex ).xyz;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
#version 330 compatibility

uniform float uLightX, uLightY, uLightZ;
uniform float uExag;
uniform vec4 uColor;
uniform sampler2D uHgtUnit;
uniform bool uUseColor;
uniform float uLevel1;
uniform float uLevel2;
uniform float uTol;
uniform float uDelta;

in vec3 vMCposition;
in vec3 vECposition;
in vec2 vST;

const float DELTA = 0.001;
const vec3 BLUE = vec3(0.1, 0.1, 0.5);
const vec3 GREEN = vec3(0.0, 0.8, 0.0);
const vec3 BROWN = vec3(0.6, 0.3, 0.1);
const vec3 WHITE = vec3(1.0, 1.0, 1.0);

const float LNGMIN = -579240./2.;  // in meters, same as heights
const float LNGMAX =  579240./2.;
const float LATMIN = -419949./2.;
const float LATMAX =  419949./2.;

Floating-point texture whose .r components contain the heights (in meters)
void main( )
{
    vec2 stp0 = vec2( DELTA, 0. );
    vec2 st0p = vec2( 0., DELTA );
    float west = texture2D( uHgtUnit, vST-stp0 ).r;
    float east  = texture2D( uHgtUnit, vST+stp0 ).r;
    float south = texture2D( uHgtUnit, vST-st0p ).r;
    float north = texture2D( uHgtUnit, vST+st0p ).r;

    vec3 stangent = vec3( 2.*DELTA*(LNGMAX-LNGMIN), 0., uExag * ( east - west ) );
    vec3 ttangent = vec3( 0., 2.*DELTA*(LATMAX-LATMIN), uExag * ( north - south ) );
    vec3 normal = normalize( cross( stangent, ttangent ) );

    float LightIntensity = dot( normalize( vec3(uLightX,uLightY,uLightZ) – vMCposition ), normal );
    if( LightIntensity < 0.1 )
        LightIntensity = 0.1;
    if( uUseColor )
    {
        float here = texture2D( uHgtUnit, vST ).r;
        vec3 color = BLUE;
        if( here > 0. )
        {
            float t = smoothstep( uLevel1-uTol, uLevel1+uTol, here );
            color = mix( GREEN, BROWN, t );
        }
        if( here > uLevel1+uTol )
        {
            float t = smoothstep( uLevel2-uTol, uLevel2+uTol, here );
            color = mix( BROWN, WHITE, t );
        }
        gl_FragColor = vec4( LightIntensity*color, 1. );
    }
    else
    {
        gl_FragColor= vec4( LightIntensity*uColor.rgb, 1. );
    }
}
Terrain Height Bump-mapping: Exaggerating the Height

No Exaggeration

Exaggerated
Terrain Height Bump-mapping: Coloring by Height
Terrain Height Bump-mapping: Coloring by Height

No Exaggeration

Exaggerated
Terrain Height Bump-mapping: Even Zooming-in Looks Good

- Portland
- Salem
- Corvallis
- Eugene
- Crater Lake
Terrain Height Bump-Mapping on a Globe

Visualization by Nick Gebbie
The Second Most Straightforward Type of Bump-Mapping is
*Height Field Equations*

This is the coordinate system we will be using. The plane is X-Y with Z pointing up.
In 2D, a slope \( m = \frac{dy}{dx} \). It can be expressed as the vector \([1, m]\).

The normal to the shape is the vector perpendicular to the vector slope:

\[ [-m, 1] \]

Note that \([1, m] \cdot [-m, 1] = 0\), as it must be.

So, if \( z = -Amp \cdot \cos\left(\frac{2\pi x}{Pd} - 2\pi \text{Time}\right) \), then the slope \( \frac{dz}{dx} \) is:

\[
\frac{dz}{dx} = Amp \cdot 2\pi / Pd \cdot \sin\left(\frac{2\pi x}{Pd} - 2\pi \text{Time}\right),
\]
and the vector slope is:

\[
\text{Slope} = [1., 0., Amp \cdot 2\pi / Pd \cdot \sin\left(\frac{2\pi x}{Pd} - 2\pi \text{Time}\right)]
\]
Bump-mapping to Create Polar Ripples

Following the pattern from before, the normal vector is:

\[
\text{Normal } = \begin{bmatrix} -\text{Amp} \times \frac{2\pi}{\text{Pd}} \times \sin(\frac{2\pi x}{\text{Pd}} - 2\pi \text{Time}) \end{bmatrix}, \quad 0., \quad 1. \]

This is true along just the X axis. The trick now is to rotate the normal vector into where we really are. Because we are just talking about a rotation, the transformation is the same as if we were rotating a vertex.

\[
\begin{align*}
\text{Nx'} &= \text{Nx} \times \cos \Theta - \text{Ny} \times \sin \Theta = \text{Nx} \times \cos \Theta \\
\text{Ny'} &= \text{Nx} \times \sin \Theta + \text{Ny} \times \cos \Theta = \text{Nx} \times \sin \Theta \\
\text{Nz'} &= \text{Nz} = 1.
\end{align*}
\]

In the final code, you would substitute \( R \) for \( x \) in the slope and normal equations.

(Also note that you could include some exponential decay to make this behave more like real ripples.)
Combining Bump and Cube Mapping
The Most Generalized Type of Bump Mapping, but the Trickiest is **Surface Local Coordinate Systems**

- N is the surface Normal
- T is the Tangent, which must be consistently oriented from vertex to vertex (glman does this automatically in the Sphere primitive)
- B is the Bitangent
Bump Mapping: A Problem

The problem is that lighting information is in Eye Coordinates, but the bump information is in Surface Local Coordinates!

We need to:

1. Figure out how to convert from one to the other, and,
2. Decide which of light information or bump information gets converted to the other’s coordinate system

While we are at it, let’s also rename the Surface Local coordinates to (s, t, h) for (texture_s, texture_t, bump_height). This is the same as (B, T, N), but uses terminology that is more bump-specific.
Bump Mapping:  
Converting Between Coordinate Systems

Converting from Eye Coordinates to Surface Local Coordinates:

\[
\begin{align*}
\begin{pmatrix} s \\ t \\ h \end{pmatrix} &= \begin{pmatrix} B_x & B_y & B_z \\ T_x & T_y & T_z \\ N_x & N_y & N_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \\
(\text{The “Orange Book” uses this to convert the light vector to Surface Local Coordinates.})
\end{align*}
\]

Converting from Surface Local Coordinates to Eye Coordinates:

\[
\begin{align*}
\begin{pmatrix} x \\ y \\ z \end{pmatrix} &= \begin{pmatrix} B_x & T_x & N_x \\ B_y & T_y & N_y \\ B_z & T_z & N_z \end{pmatrix} \begin{pmatrix} s \\ t \\ h \end{pmatrix} \\
(\text{I prefer to use this one to convert the bump normal to Eye Coordinates.})
\end{align*}
\]
Bump Mapping:  
Two Ways to Establish the Surface Local Coordinate System

There are 2 good ways to get the tangent and binormal vectors:

1. Have the Tangent already defined (glman's Sphere does this)
2. Pick a general rule, e.g., "Tangent ≈ up"
   2a. Use Gram-Schmidt to correctly orthogonalize it wrt the Normal
   2b. Use two cross-products to correctly orthogonalize it wrt the Normal

Note: 2a and 2b give the same result, but some people find 2b easier to understand

// the vectors B-T-N form an X-Y-Z-looking right handed coordinate system:
vec3 N = normalize( gl_NormalMatrix * gl_Normal );
vec3 T;
vec3 B;

#define GRAM_SCHMIDT_METHOD
#ifdef HAVE_TANGENT_METHOD
T = normalize( vec3( gl_ModelViewMatrix*vec4(Tangent,0.) ) );
B = normalize( cross(T,N) );
#endif

#define CROSS_PRODUCT_METHOD
#ifdef GRAM_SCHMIDT_METHOD
T = vec3( 0.,1.,0.);float d = dot( T, N );
T = normalize( T - d*N );
B = normalize( cross(T,N) );
#endif

#endif CROSS_PRODUCT_METHOD
T = vec3( 0.,1.,0.);
B = normalize( cross(T,N) );
T = normalize( cross(N,B) );
#endif
Gram-Schmidt Orthogonalization

1. Given that \( \mathbf{N} \) is correct, how do we change \( \mathbf{T} \) to be exactly perpendicular to \( \mathbf{N} \)?

2. How much of \( \mathbf{T} \) is in the same direction as \( \mathbf{N} \)?

3. How much of \( \mathbf{T} \) to get rid of so that none of it is in the same direction as \( \mathbf{N} \)

4. The resulting \( \mathbf{T}' \) is exactly perpendicular to \( \mathbf{N} \)

\[
\mathbf{T} = \text{vec3}(0.,1.,0.);
\]
\[
\text{float } d = \text{dot}(\mathbf{T}, \mathbf{N});
\]
\[
\mathbf{T} = \text{normalize}(\mathbf{T} - d \mathbf{N});
\]
\[
\mathbf{B} = \text{normalize}(\text{cross}(\mathbf{T}, \mathbf{N}));
\]
Bump Mapping:
Establishing the Surface Local Coordinate System

// Produce the transformation from Surface coords to Eye coords:

BTN = vec3( B.x, T.x, N.x );
BTNy = vec3( B.y, T.y, N.y );
BTNz = vec3( B.z, T.z, N.z );

// where the light is coming from:

vec3 LightPosition = vec3( LightX, LightY, LightZ );
vec3 ECposition = ( gl_ModelViewMatrix * gl_Vertex ).xyz;
DirToLight = normalize( LightPosition - ECposition );

gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
Bump Mapping:
Using the Surface-Local-to-Eye-Coordinate Transform

```cpp
vec3 ToXyz( vec3 sth )
{
    sth = normalize( sth );

    vec3 xyz;
    xyz.x = dot( BTNx, sth );
    xyz.y = dot( BTNy, sth );
    xyz.z = dot( BTNz, sth );
    return normalize( xyz );
}
```
Bump Mapping:
Using the Surface Local Transform

void main()
{
    const float PI = 3.14159265;

    vec2 st = vST; // locate the bumps based on (s,t)

    float Swidth = 1. / BumpDensity;
    float Theight = 1. / BumpDensity;
    float numInS = floor( st.s / Swidth );
    float numInT = floor( st.t / Theight );

    vec2 center;
    center.s = numInS * Swidth + Swidth/2.;
    center.t = numInT * Theight + Theight/2.;
    vec3 stp = st - center; // st’ is now wrt the center of the bump

    float theta = atan( stp.t, stp.s );
}
Bump Mapping:
Using the Surface Local Transform

```c
vec3 normal = ToXyz( vec3( 0., 0., 1. ) ); // un-bumped normal
if( abs(stp.s) > Swidth/4. || abs(stp.t) > Theight/4. )
{
    normal = ToXyz( vec3( 0., 0., 1. ) );
} else
{
    if( PI/4. <= theta && theta <= 3.*PI/4. )
    {
        normal = ToXyz( vec3( 0., Height, Theight/4. ) );
    } else if( -PI/4. <= theta && theta <= PI/4. )
    {
        normal = ToXyz( vec3( Height, 0., Swidth/4. ) );
    } else if( -3.*PI/4. <= theta && theta <= -PI/4. )
    {
        normal = ToXyz( vec3( 0., -Height, Theight/4. ) );
    } else if( theta >= 3.*PI/4. || theta <= -3.*PI/4. )
    {
        normal = ToXyz( vec3( -Height, 0., Swidth/4. ) );
    }
}
float intensity = Ambient + (1.-Ambient)*dot(normal, DirToLight);
vec3 litColor = SurfaceColor.rgb * intensity;
gl_FragColor = vec4( litColor, SurfaceColor.a );
```
Changing the Bump Height
Changing the Bump Density
It's Handy to not need a Program-supplied Tangent Vector

Cow Pox? :-(
Combining Bump and Cube Mapping: A Good Reason to Work in Eye Coordinates instead of Surface Local Coordinates
Combining Bump and Cube Mapping:
A Good Reason to Work in Eye Coordinates instead of Surface Local Coordinates