Generalized Bump-mapping with Surface Local Coordinates

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
The Most Straightforward Types of Bump-Mapping are Height Fields

Why?

Height Field bump-mapping is straightforward because the underlying coordinate system is constant. Each fragment’s Z points up, each fragment’s X points right, etc.
What if that is not the case? Here, the coordinate system is constantly changing, depending on where you are on the sphere.
This is referred to as **Surface Local Coordinates**

Rather than X-Y-Z, Surface Local Coordinates are **B-T-N**:

- N is the surface Normal vector
- T is a Tangent vector
- B is the Bitangent, the other tangent vector

We will assume that we know the Normal everywhere because of how the shape was modeled. Now, how do we find T and B? And, how do we convert these to X-Y-Z so we can draw as usual?
Generalized Bump Mapping: A Problem

The problem is that we need to do lighting and the lighting information is all in X-Y-Z, but the bump information is in B-T-N!

We need to:

1. Figure out how to determine T and B, and,
2. Figure out how to convert B-T-N coordinates to X-Y-Z for lighting

While we are at it, I like renaming 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 sounds like the way that we have been talking.
Bump Mapping:
Establishing the Surface Local Coordinate System

We need a second piece of information: Pick a general rule, e.g., "Tangent ≈ up (0.,1.,0.)"
We then have two choices:

a. Use two cross-products to correctly orthogonalize it wrt the Normal
b. Use the Gram-Schmidt rule to correctly orthogonalize it wrt the Normal

// the vectors B-T-N form an X-Y-Z-looking
// right handed coordinate system:

vec3 N = normalize( gl_NormalMatrix * gl_Normal );
vec3 Tg, T; // T_guess and corrected T
vec3 B;

#define CROSS_PRODUCT_METHOD

#ifdef CROSS_PRODUCT_METHOD
Tg = vec3( 0.,1.,0.); // guess
B = normalize( cross(Tg,N) );
T = normalize( cross(N,B) );
#endif

#define GRAM_SCHMIDT_METHOD

#ifdef GRAM_SCHMIDT_METHOD
Tg = vec3( 0.,1.,0.); // guess
float d = dot( Tg, N );
T = normalize( Tg - d*N );
B = normalize( cross(T,N) );
#endif
vec3 T_g = vec3( 0.,1.,0.); // initial guess
vec3 B = normalize(cross(T_g,N));
vec3 T = normalize(cross(N,B));

1. Given that N is correct, how do we change T_g to be exactly perpendicular to N?

2. Take the cross product of T_g and N to get a B vector that is perpendicular to both.

3. Take the cross product of N and B to get a T vector that is perpendicular to both.
Gram-Schmidt Orthogonalization

Given that $N$ is correct, how do we change $T_g$ to be exactly perpendicular to $N$?

1. How much of $T_g$ is in the same direction as $N$?

2. $d = T_g \cdot \hat{N}$

3. How much of $T_g$ do we need to get rid of so that none of it is in the same direction as $N$?

4. The resulting $T$ is perpendicular to $N$

$$T = T_g - d\hat{N} = T_g - (T_g \cdot \hat{N})\hat{N}$$

vec3 $T_g = \text{vec3}(0.,1.,0.); // initial guess$
float $d = \text{dot}(T_g, N);$  
vec3 $T = \text{normalize}(T_g - d*N);$  
vec3 $B = \text{normalize}(\text{cross}(T,N));$
Bump Mapping: 
Converting Between Coordinate Systems

Converting from X-Y-Z to s-t-h:

\[
\begin{align*}
\{s\} &= \begin{bmatrix} B_x & B_y & B_z \end{bmatrix} \begin{bmatrix} x \end{bmatrix} \\
\{t\} &= \begin{bmatrix} T_x & T_y & T_z \end{bmatrix} \begin{bmatrix} y \end{bmatrix} \\
\{h\} &= \begin{bmatrix} N_x & N_y & N_z \end{bmatrix} \begin{bmatrix} z \end{bmatrix}
\end{align*}
\]

Converting from s-t-h to X-Y-Z:

\[
\begin{align*}
\{x\} &= \begin{bmatrix} B_x & T_x & N_x \end{bmatrix} \begin{bmatrix} s \end{bmatrix} \\
\{y\} &= \begin{bmatrix} B_y & T_y & N_y \end{bmatrix} \begin{bmatrix} t \end{bmatrix} \\
\{z\} &= \begin{bmatrix} B_z & T_z & N_z \end{bmatrix} \begin{bmatrix} h \end{bmatrix}
\end{align*}
\]

I prefer to use the second one so we can do lighting in X-Y-Z like we are used to doing.
Generalized Bump Mapping:
Establishing the Surface Local Coordinate System

**Vertex shader:**

```glsl
#version 330 compatibility
uniform vec3 uLightPosition;

out vec2   vST; // texture coords
out vec3   vN; // normal vector
out vec3   vL; // vector from point to light
out vec3   vE; // vector from point to eye
out vec3   vBTNx, vBTNy, vBTNz;

void main( )
{
    vN = normalize( gl_NormalMatrix * gl_Normal ); // normal vector
    vec3 Tg = vec3( 0.,1.,0.); // guess
    vec3 B = normalize( cross(Tg,vN) );
    vec3 T = normalize( cross(vN,B) );

    // produce the transformation from Surface coords to Eye coords:
    vBTNx = vec3( B.x, T.x, vN.x );
    vBTNy = vec3( B.y, T.y, vN.y );
    vBTNz = vec3( B.z, T.z, vN.z );
    vST = gl_MultiTexCoord0.st;

    vec4 ECposition = gl_ModelViewMatrix * gl_Vertex; // eye coordinate position
    vL = uLightPosition - ECposition.xyz; // vector from the point to the light position
    vE = vec3( 0., 0., 0. ) - ECposition.xyz; // vector from the point to the eye position
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
Generalized Bump Mapping:  
Using the s-t-h to X-Y-Z Transform

Fragment shader:

```glsl
#version 330 compatibility
uniform vec3 uColor;
uniform vec3 uSpecularColor;
uniform float uKa, uKd, uKs; // coefficients of each type of lighting
uniform float uShininess; // specular exponent
uniform float uBumpDensity; // density of bumps

const float PI = 3.14159265;
in vec2 vST; // texture cords
in vec3 vN; // normal vector
in vec3 vL; // vector from point to light
in vec3 vE; // vector from point to eye
in vec3 vBTNx, vBTNy, vBTNz;

vec3 ToXyz( vec3 sth )
{
    sth = normalize( sth );
    vec3 xyz;
    xyz.x = dot( vBTNx, sth );
    xyz.y = dot( vBTNy, sth );
    xyz.z = dot( vBTNz, sth );
    return normalize( xyz );
}

Look at this closely. It is actually a matrix-multiply!
```

University
Computer Graphics
The basic operation of matrix multiplication is to pair-wise multiply a single row by a single column.

\[
\begin{bmatrix}
1 & 2 & 3
\end{bmatrix}
\times
\begin{bmatrix}
4 \\
5 \\
6
\end{bmatrix}
\rightarrow
4*1 + 5*2 + 6*3
\rightarrow
32
\]
void main() {
    vec3 Normal = normalize(vN);
    vec3 Light = normalize(vL);
    vec3 Eye = normalize(vE);
    vec3 myColor = uColor; // default color

    // locate the bumps based on (s,t):
    float Swidth = (1.-0.) / uBumpDensity;        // s distance between bumps
    float Theight = (1.-0.) / uBumpDensity;        // t distance between bumps
    float numInS = int( vST.s /  Swidth );            // which "checker" square we are in
    float numInT = int( vST.t  /  Theight );           // which "checker" square we are in

    vec2 center;
    center.s = numInS * Swidth + Swidth/2.;  // center of that bump checker
    center.t = numInT * Theight + Theight/2.;  // center of that bump checker
    vec2 st = vST - center;  // st is now wrt the center of the bump

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

vec3 normal = ToXyz( Normal); // 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. ) );
    }
}

...
\[
vec3 \text{ambient} = \text{uKa} \times \text{myColor}; \\
\text{float} \ d = 0.; \\
\text{float} \ s = 0. \\
\text{if} (\ \text{dot(normal, Light)} > 0. \ // \text{only do specular if the light can see the point} \\
\{ \\
\text{d} = \text{dot(normal, Light);} \\
\text{vec3} \ R = \text{normalize( reflect( -Light, normal ) );} \ \ // \text{reflection vector} \\
\text{s} = \text{pow( max( dot(Eye, R), 0. ), uShininess );} \\
\} \\
\text{vec3} \text{diffuse} = \text{uKd} \times d \times \text{myColor}; \\
\text{vec3} \text{specular} = \text{uKs} \times s \times \text{uSpecularColor}; \\
\text{gl\_FragColor} = \text{vec4( ambient + diffuse + specular, 1. );} \\
\]
Changing the Bump Height
Changing the Bump Density
Different Objects

Cow Pox? :-)

Oregon State University
Computer Graphics
Combining Bump and Cube Mapping: 
A Good Reason to Work in X-Y-Z instead of B-T-N
Combining Bump and Cube Mapping: A Good Reason to Work in X-Y-Z instead of B-T-N