Generalized Bump-mapping with Surface Local Coordinates

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. Thus, the tangent vectors always involve $\mathbf{v}_x$ and $\mathbf{v}_y$.

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

To call these moving axes X-Y-Z would be confusing. Rather than X-Y-Z, Surface Local Coordinates are B-T-N:
- $\mathbf{N}$ is the surface Normal vector, which we usually know already
- $\mathbf{T}$ is a Tangent vector
- $\mathbf{B}$ 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 $\mathbf{T}$ and $\mathbf{B}$? And, how do we convert these to X-Y-Z?

Generalized Bump Mapping: A Problem

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

We need to:
1. Figure out how to determine $\mathbf{T}$ and $\mathbf{B}$, and,
2. Figure out how to convert B-T-N coordinates to X-Y-Z for lighting

We will refer to the coordinates in the B-T-N system as $(b,t,n)$.

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
Bump Mapping: Converting Between Coordinate Systems

Converting from X-Y-Z to b-t-n:

\[
\begin{bmatrix}
b_x \\
b_y \\
b_z
\end{bmatrix} =
\begin{bmatrix}
B_x & B_y & B_z \\
T_x & T_y & T_z \\
N_x & N_y & N_z
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}
\]

Converting from b-t-n to X-Y-Z:

\[
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix} =
\begin{bmatrix}
B_x & T_x & N_x \\
B_y & T_y & N_y \\
B_z & T_z & N_z
\end{bmatrix}
\begin{bmatrix}
b_x \\
b_y \\
b_z
\end{bmatrix}
\]

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: Using the s-t-h to X-Y-Z Transform

Fragment shader:

```
vec3 T = normalize(cross(vN,B) );
vec3 B = normalize(cross(T,N) );
vec3 N = normalize( gl_NormalMatrix * gl_Normal ); // normal vector

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

uniform float uBumpDensity; // density of bumps
uniform float uShininess; // specular exponent
uniform float uKa, uKd, uKs; // coefficients of each type of lighting
uniform vec3 uSpecularColor;
uniform vec3 uColor;

#version 330 compatibility
```

Matrix Multiplication is Really Row-by-Row Dot Products

The basic operation of matrix multiplication is to pair-wise multiply a single row by a single column:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\begin{bmatrix}
4 \\
5 \\
6
\end{bmatrix}
= 32
\]

1x3 3x1 1x1
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 );

    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 ambient = uKa * myColor;
    float d = 0.;
    float s = 0.;
    if( dot(normal,Light) > 0. // only do specular if the light can see the point
    {
        d = dot(normal,Light);
        vec3 R = normalize(  reflect( -Light, normal )  );  // reflection vector
        s = pow(  max( dot(Eye,R), 0. ),  uShininess );
    }

    vec3 diffuse = uKd * d * myColor;
    vec3 specular = uKs * s * uSpecularColor;
    gl_FragColor = vec4( ambient + diffuse + specular, 1. );
}
Combining Bump and Cube Mapping: A Good Reason to Work in X-Y-Z instead of B-T-N