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. Z always points up, X always points the right, etc.

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
- T is the Tangent
- B is the Bitangent

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?

Bump Mapping: A Problem

The problem is that lighting information is 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 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: Converting Between Coordinate Systems

Converting from X-Y-Z to s-t-h:
\[
\begin{bmatrix}
  s \\
  t \\
  h \\
\end{bmatrix} =
\begin{bmatrix}
  B_x & B_y & B_z \\
  T_x & T_y & T_z \\
  N_x & N_y & N_z \\
\end{bmatrix}^{-1}
\begin{bmatrix}
  x \\
  y \\
  z \\
\end{bmatrix}
\]

Converting from s-t-h to X-Y-Z:
\[
\begin{bmatrix}
  x \\
  y \\
  z \\
\end{bmatrix} =
\begin{bmatrix}
  B_x & B_y & B_z \\
  T_x & T_y & T_z \\
  N_x & N_y & N_z \\
\end{bmatrix}
\begin{bmatrix}
  s \\
  t \\
  h \\
\end{bmatrix}
\]

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

We need a second piece of information: Pick a general rule, e.g., “Tangent = up”
a. Use the Gram-Schmidt rule to correctly orthogonalize it wrt the Normal
b. Use two cross-products to correctly orthogonalize it wrt the Normal

1. If the vectors B-T-N form an X-Y-Z looking right handed coordinate system:
   - \( \text{vec3 } N = \text{normalize( gl\_NormalMatrix } \times \text{gl\_Normal )} \)
   - \( \text{vec3 } T; \)
   - \( \text{vec3 } B; \)
   - Define GRAM_SCHMIDT_METHOD
     - \#ifdef GRAM_SCHMIDT_METHOD
     - \( \text{vec3 } T = \text{vec3} (0., 1., 0.); \)
     - \( \text{float } d = \text{dot( T, N )}; \)
     - \( \text{vec3 } T = \text{normalize( } T - d \times N \text{ )}; \)
     - \( \text{vec3 } B = \text{normalize( cross( T, N ) )}; \)
     - \#endif
   - \#ifdef CROSS_PRODUCT_METHOD
     - \( \text{vec3 } T = \text{vec3} (0., 1., 0.); \)
     - \( \text{vec3 } B = \text{normalize( cross( T, N ) )}; \)
     - \( \text{vec3 } T = \text{normalize( cross( N, B ) )}; \)
     - \#endif

2. We need a second piece of information: Pick a general rule, e.g., “Tangent ≈ up”
   a. Use the Gram-Schmidt rule to correctly orthogonalize it wrt the Normal
   b. Use two cross-products to correctly orthogonalize it wrt the Normal

Gram-Schmidt Orthogonalization

T = vec3(0., 1., 0.);
float d = dot( T, N );
T = normalize( T - d\times N );
B = normalize( cross( T, N ) );

ˆ
de T
dN

T' = T - d\hat{N} = T - (T \cdot \hat{N})\hat{N}

How much of T to we need to get rid of so that none of it is in the same direction as \( \hat{N} \)?

Given that \( N \) is correct, how do we change T to be exactly perpendicular to \( N \)?

The resulting \( T' \) is exactly perpendicular to \( N \).

The resulting \( T' \) is exactly perpendicular to \( N \).

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

Generalized Bump Mapping:
Establishing the Surface Local Coordinate System

// Produce the transformation from Surface coords to Eye coords:
vec3 BTNx = vec3( B.x, T.x, N.x );
vec3 BTNy = vec3( B.y, T.y, N.y );
vec3 BTNz = vec3( B.z, T.z, N.z );
// where the light is coming from:
vec3 LightPosition = vec3( LightX, LightY, LightZ );
vec3 ECposition = ( gl\_ModelViewMatrix \times gl\_Vertex ).xyz;
DirToLight = normalize( LightPosition - ECposition );
gl\_Position = gl\_ModelViewProjectionMatrix \times gl\_Vertex;

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

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 );
}

Generalized 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 );
    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 = ToXyx( 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)\times dot( normal, DirToLight );
    vec3 litColor = SurfaceColor.rgb \times 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

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