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.

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:
- Use two cross-products to correctly orthogonalize it wrt the Normal
- Use the Gram-Schmidt rule to correctly orthogonalize it wrt the Normal

Bump Mapping:

```c
// 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
vec3 B;
#define CROSS_PRODUCT_METHOD
#define GRAM_SCHMIDT_METHOD

// 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
vec3 B;

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

//define GRAM_SCHMIDT_METHOD
#define GRAM_SCHMIDT_METHOD
vec3 Tg = vec3( 0.,1.,0.); // guess
float d = dot( Tg, N );
T = normalize( Tg - d*N );
B = normalize( cross(T, N) );
```
Given that \( N \) is correct, how do we change \( T_g \) to be exactly perpendicular to \( N \)?

1. Take the cross product of \( T_g \) and \( N \) to get a \( B \) vector that is perpendicular to both.

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

3. The resulting \( T \) is perpendicular to \( N \).

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

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

2. How much of \( T_g \) is in the same direction as \( N \)?

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 \( N \).

I prefer to use the second one so we can do lighting in X-Y-Z like we are used to doing.
```cpp
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. );
}
```

Changing the Bump Height

Changing the Bump Density

Different Objects

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