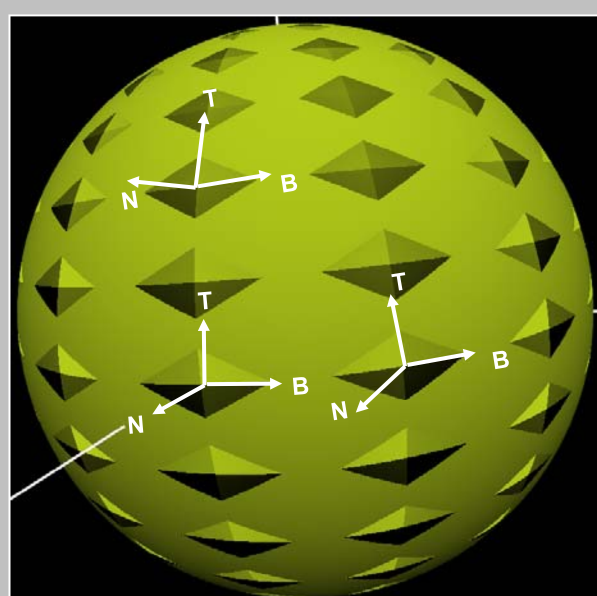
Bump Mapping

Mike Bailey

Oregon State University

Bump Mapping: Surface Local Coordinate System



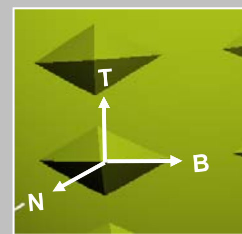
- N is the surface normal
- T is the tangent, which must be consistently oriented from vertex to vertex (glman does this automatically in the Sphere primitive)
- B is the Binormal

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_**h**eight). 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:

$$\begin{Bmatrix} s \\ t \\ h \end{Bmatrix} = \begin{bmatrix} B_x & B_y & B_z \\ T_x & T_y & T_z \\ N_x & N_y & N_z \end{bmatrix} \begin{Bmatrix} x \\ y \\ z \end{Bmatrix}$$

(The "Orange Book" uses this to convert the light vector to Surface Local Coordinates.)

Converting from Surface Local Coordinates to **Eye Coordinates:**

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{bmatrix} B_x & T_x & N_x \\ B_y & T_y & N_y \\ B_z & T_z & N_z \end{bmatrix} \begin{Bmatrix} s \\ t \\ h \end{Bmatrix}$$

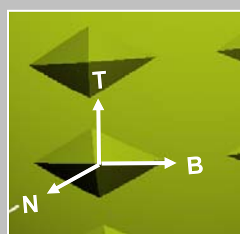
(I prefer to use this one to convert the bump normal to Eye Coordinates.)

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

#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

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

Oregon State University
Computer Graphics

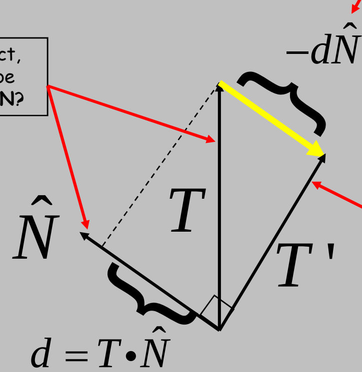
mjb – April 24, 2007

Gram-Schmidt Orthoganalization

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

(3) How much of **T** to get rid of so that *none* of it is in the same direction as **N**

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



(4) The resulting **T'** is exactly perpendicular to **N**

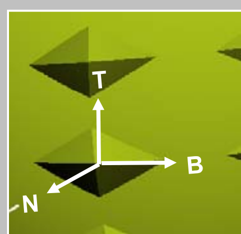
(2) How much of **T** is in the same direction as **N**

Oregon State University
Computer Graphics

mjb – April 24, 2007

Bump Mapping:
Establishing the Surface Local Coordinate System

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

BTNx = 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;
```

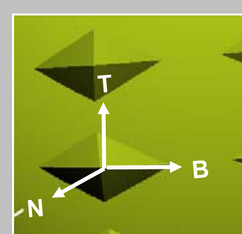


Oregon State University
Computer Graphics

mjb – April 24, 2007

Bump Mapping:
Using the Surface-Local-to-Eye-Coordinate 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 );
}
```

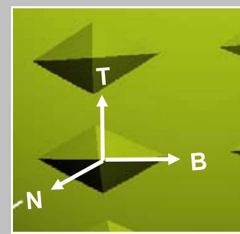


Oregon State University
Computer Graphics

mjb – April 24, 2007

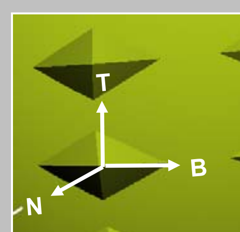
Bump Mapping:
Using the Surface Local Transform

```
void main()
{
	const float PI = 3.14159265;

	vec2 st = gl_TexCoord[0].st;    // 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 );
```

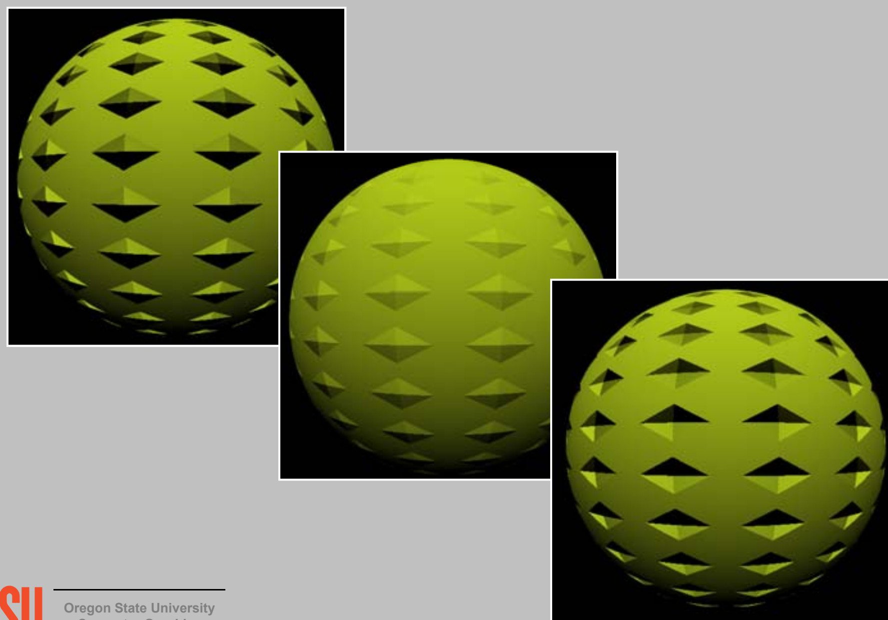


Bump Mapping:
Using the Surface Local Transform

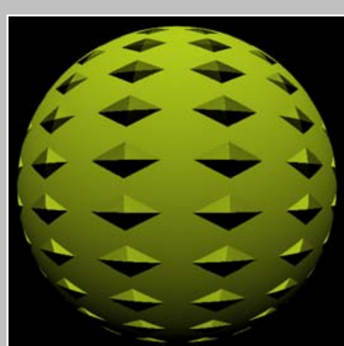
```
	vec3 normal = ToXyz(  vec3( 0., 0., 1. )  );        // un-bumped normal

	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 );
}
```

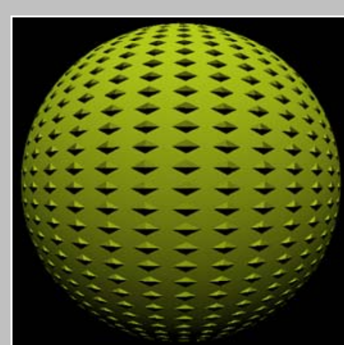


Changing the Bump Height

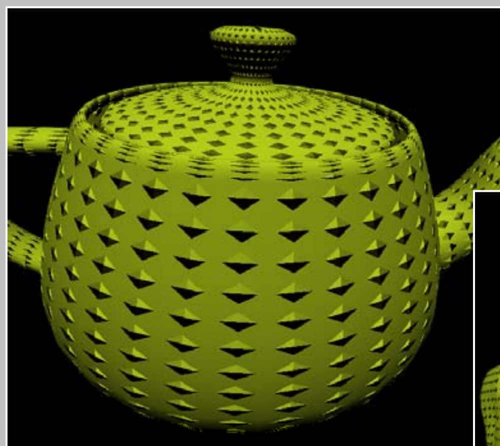


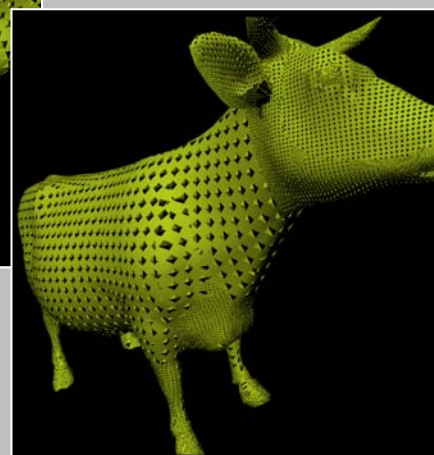
Changing the Bump Density





It's handy to not need a Program-supplied Tangent Vector





Cow Pox? :-)

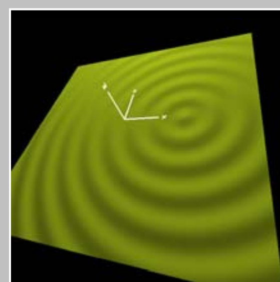
Combining Bump and Cube Mapping: A Good Reason to Work in Eye Coordinates instead of Surface Local Coordinates

Combining Bump and Cube Mapping:
A Good Reason to Work in Eye Coordinates instead of Surface Local Coordinates

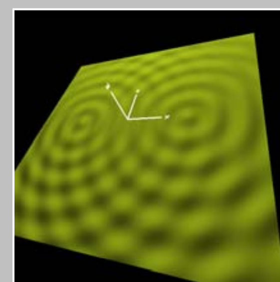
Combining Bump and Cube Mapping:
A Good Reason to Work in Eye Coordinates instead of Surface Local Coordinates

It's Even Easier When You Know the Bump Equation in World Coordinates: Bump-mapping to Create Ripples

Rock A Dropped

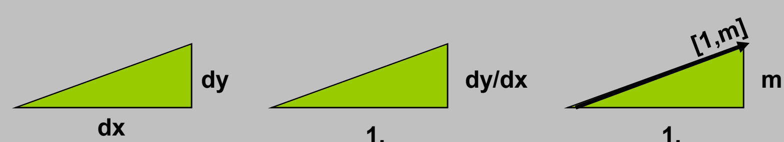


Rock B Dropped



Both Rocks Dropped

In 2D, a slope m = dy/dx. It can be expressed as the vector [1,m].



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

Note that [1,m] · [-m,1] = 0, as it must be.

So, if z = -Amp * cos(2πx/Pd - 2πTime), then the slope dz/dx is:

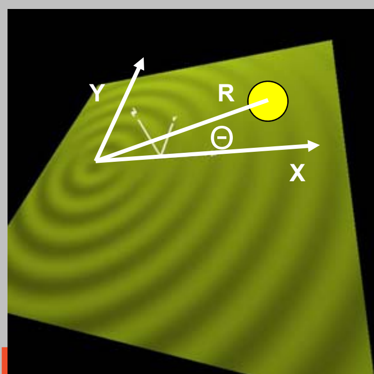
dz/dx = Amp * 2π/Pd * sin(2πx/Pd - 2πTime), and the vector slope is:

Slope = [1., 0., Amp * 2π/Pd * sin(2πx/Pd - 2πTime)]

Following the pattern from before, the normal vector is:

[Normal] = [-Amp * 2π/Pd * sin(2πx/Pd - 2πTime), 0., 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.



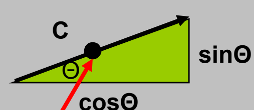
Nx' = Nx * cosΘ - Ny * sinΘ = Nx * cosΘ

Ny' = Nx * sinΘ + Ny * cosΘ = Nx * sinΘ

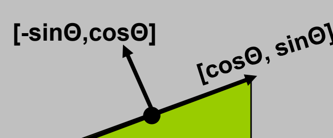
Nz' = Nz = 1.

(Note that in the final version, you will substitute R for x in the slope equation)

Because each linear ripple has an angle Θ, we can think of its direction and perpendicular normal like this:

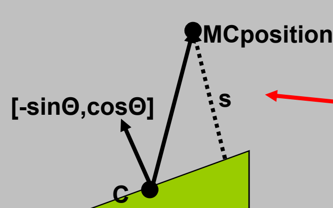


The linear ripple goes through the point C in the direction [cosΘ, sinΘ]



The normal is then [-sinΘ, cosΘ]

(Note that slope · normal = 0, as it must be.)



The distance, s, of a Model Coordinate position perpendicular to the linear ripple is:
s = (MCposition-C) · (sinΘ, cosΘ)

The amplitude of the wave, z, is:

z = - Amp * cos(2πs/P - 2πTime)

(where P is the wave period)

And the slope dz/ds is:

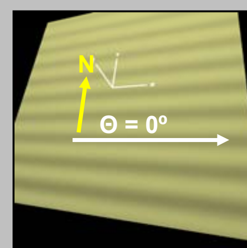
dz/ds = Amp * 2π/P * sin(2πs/P - 2πTime)

If we start by assuming that the ripple angle is 0º (i.e., the wave is propagating in y), then the vector slope of the wave is:

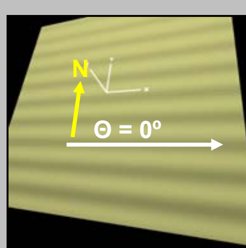
slope = [0., 1., dz/dy]
= [0., 1., Amp * 2π/P * sin(2πs/P - 2πTime)]

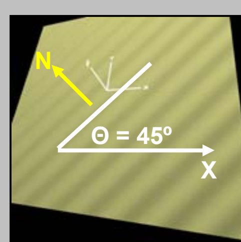
So the wave's vector normal while propagating in y is:

normal = [0., -Amp * 2π/P * sin(2πs/P - 2πTime), 1.]



This is true if the wave is propagating in y, i.e., the ripple angle is 0º. 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.





Nx' = Nx * cosΘ - Ny * sinΘ

Ny' = Nx * sinΘ + Ny * cosΘ

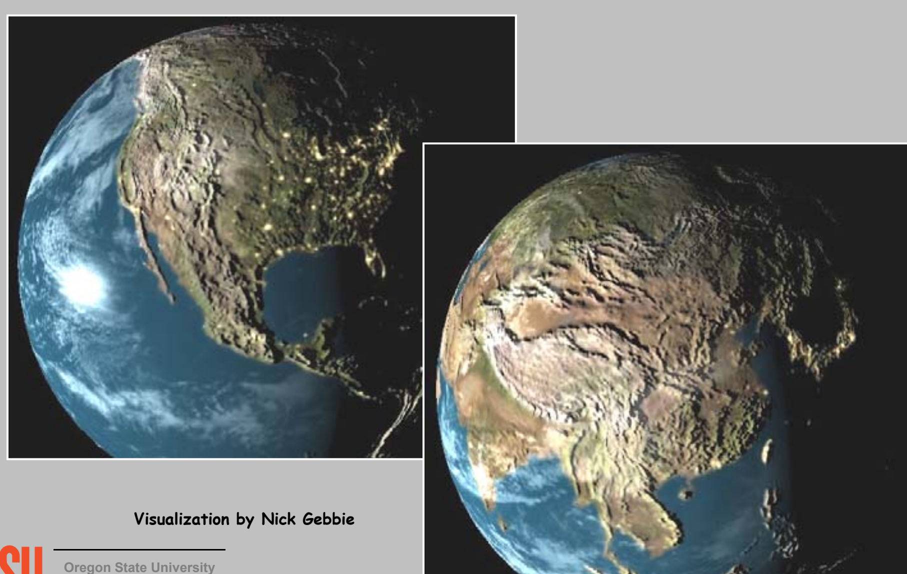
Nz' = Nz

```
vec3 normal = normalize( vec3(Nx',Ny',Nz') );
```

So, for any MCposition of a fragment, we compute the normal vector to the simulated rippled surface. We then make this interact with the light source location to make variations in intensity give the rippled appearance.

Combining Bump and Cube Mapping: A Good Reason to Work in Eye Coordinates instead of Surface Local Coordinates

Visualization: Terrain Height Bump-Mapping



Visualization by Nick Gebbie