The Tenth Planet

If it existed

Followup: Normal Mapping Without Precomputed Tangents

Posted on **20.01.2013** by **christian**

This post is a follow-up to my 2006 ShaderX^5 article [4] about normal mapping without a pre-computed tangent basis. In the time since then I have refined this technique with lessons learned in real life. For those unfamiliar with the topic, the motivation was to construct the tangent frame on the fly in the pixel shader, which ironically is the exact opposite of the motivation from [2]:

> We present a bump mapping method that requires minimal hardware beyond that necessary for Phong shading. We eliminate the costly per-pixel steps of reconstructing a tangent space and perturbing the interpolated normal vector by a) interpolating vectors that have been

Since it is not 1997 anymore, doing the tangent space on-the-fly has some potential benefits, such as reduced complexity of asset tools, per-vertex bandwidth and storage, attribute interpolators, transform work for skinned meshes and last but not least, the possibility to apply normal maps to any procedurally generated texture coordinates or non-linear deformations.

Intermission: Tangents vs Cotangents

The way that normal mapping is traditionally defined is, as I think, flawed, and I would like to point this out with a simple C++ metaphor. Suppose we had a class for vectors, for example called `Vector3`, but we also had a different class for covectors, called `Covector3`. The latter would be a clone of the ordinary vector class, except that it behaves differently under a transformation (EDIT 2018: see this article for a comprehensive introduction to the theory behind covectors and dual spaces). As you may know, normal vectors are an example of such covectors, so we're going to declare them as such. Now imagine the following function:

```
Vector3 tangent;
Vector3 bitangent;
Covector3 normal;

Covector3 perturb_normal( float a, float b, float c )
{
    return a * tangent +
           b * bitangent +
           c * normal;
           // ^^^^ compile-error: type mismatch for operator +
}
```

The above function mixes vectors and covectors in a single expression, which in this fictional example leads to a type mismatch error. If the `normal` is of type `Covector3`, then the `tangent` and the `bitangent` should be too, otherwise they cannot form a consistent frame, can they? In real life shader code of course, everything would be defined as `float3` and be fine, or rather not.

MATHEMATICAL COMPILE ERROR

Unfortunately, the above mismatch is exactly how the 'tangent frame' for the purpose of normal mapping was introduced by the authors of [2]. This type mismatch is invisible as long as the tangent frame is orthogonal. When the exercise is however to reconstruct the tangent frame in the pixel shader, as this article is about, then we have to deal with a *non-orthogonal screen projection*. This is the reason why in the book I had introduced both **T** (which should be called co-tangent) and **B** (now it gets somewhat silly, it should be called co-bi-tangent) as covectors, otherwise the algorithm does not work. I have to admit that I could have been more articulate about this detail. This has caused real confusion, cf from gamedev.net:

> For short, I see Lengyel defining tangent vectors like this:
>
> ```
> PositionDelta = TexCoordDelta * Tangent
> ```
>
> whereas Schuler defines it:
>
> ```
> PositionDelta * Tangent = TexCoordDelta
> ```
>
> Where does this discrepancy come from?

The discrepancy is explained above, as my 'tangent vectors' are really covectors. The definition on page 132 is consistent with that of a covector, and so the frame $(\mathbf{T}|\mathbf{B}|\mathbf{N})$ should be called a cotangent frame.

Intermission 2: Blinns Perturbed Normals (History Channel)

In this section I would like to show how the definition of $\mathbf{T}$ and $\mathbf{B}$ as covectors follows naturally from Blinns original bump mapping paper [1]. Blinn considers a curved parametric surface, for instance, a Bezier-patch, on which he defines tangent vectors $\mathbf{p}_u$ and $\mathbf{p}_v$ as the derivatives of the position $\mathbf{p}$ with respect to u and v.

> The partial derivatives of these functions form two new vectors which we will call Pu and Pv.
>
> Pu = (Xu,Yu,Zu)
>
> Pv = (Xv,Yv,Zv)

In this context it is a convention to use subscripts as a shorthand for partial derivatives, so he is really saying $\mathbf{p}_u = \partial\mathbf{p}/\partial u$, etc. He also introduces the surface normal $\mathbf{N} = \mathbf{p}_u \times \mathbf{p}_v$ and a bump height function f, which is used to displace the surface. In the end, he arrives at a formula for a first order approximation of the perturbed normal:

$$\mathbf{N}' \simeq \mathbf{N} + \frac{f_u \mathbf{N} \times \mathbf{p}_v + f_v \mathbf{p}_u \times \mathbf{N}}{|\mathbf{N}|},$$

I would like to draw your attention towards the terms $\mathbf{N} \times \mathbf{p}_v$ and $\mathbf{p}_u \times \mathbf{N}$. They are the *perpendiculars* to $\mathbf{p}_u$ and $\mathbf{p}_v$ in the tangent plane, and can be seen as the 'offset vectors' that ultimately displace the normal. They are also covectors (why, make the duck test: if it transforms like a covector, it is a covector) so adding them to the normal

does not raise said type mismatch. If we divide these terms one more time by $|\mathbf{N}|$ and flip their signs, we'll arrive at the ShaderX[5] definition of $\mathbf{T}$ and $\mathbf{B}$ as follows:

$$\mathbf{T} = -\frac{\mathbf{N} \times \mathbf{p}_v}{|\mathbf{N}|^2} = \nabla u, \qquad \mathbf{B} = -\frac{\mathbf{p}_u \times \mathbf{N}}{|\mathbf{N}|^2} = \nabla v,$$

$$\mathbf{N}' \simeq \widehat{\mathbf{N}} - f_u \mathbf{T} - f_v \mathbf{B},$$

where the hat (as in $\widehat{\mathbf{N}}$) denotes the normalized normal. $\mathbf{T}$ can be interpreted as the *normal* to the plane of constant u, and likewise $\mathbf{B}$ as the *normal* to the plane of constant v. Therefore we have three normal vectors, or covectors, $\mathbf{T}$, $\mathbf{B}$ and $\mathbf{N}$, and they are the a basis of a cotangent frame. Equivalently, $\mathbf{T}$ and $\mathbf{B}$ are the gradients of u and v, which is the definition I had used in the book. The magnitude of the gradient therefore determines the bump strength, a fact that I will discuss later when it comes to scale invariance.

A Little Unlearning

The mistake of many authors is to unwittingly take $\mathbf{T}$ and $\mathbf{B}$ for $\mathbf{p}_u$ and $\mathbf{p}_v$, which only works as long as the vectors are orthogonal. Let's unlearn 'tangent', relearn 'cotangent', and repeat the historical development from this perspective: Peercy et al. [2] precomputes the values f_u and f_v (the change of bump height per change of texture coordinate) and stores them in a texture. They call it 'normal map', but is a really something like a 'slope map', and they have been reinvented recently under the name of derivative maps. Such a slope map cannot represent horizontal normals, as this would need an infinite slope to do so. It also needs some 'bump scale factor' stored somewhere as meta data. Kilgard [3] introduces the modern concept of a normal map as an *encoded rotation operator*, which does away with the approximation altogether, and instead goes to define the perturbed normal directly as

$$\mathbf{N}' = a\mathbf{T} + b\mathbf{B} + c\widehat{\mathbf{N}},$$

where the coefficients a, b and c are read from a texture. Most people would think that a normal map stores normals, but this is only superficially true. This idea of Kilgard was, since the unperturbed normal has coordinates $(0, 0, 1)$, it is sufficient to store the last column of the rotation matrix that would rotate the unperturbed normal to its perturbed position. So yes, a normal map stores basis vectors that correspond to perturbed normals, but it really is an encoded rotation operator. The difficulty starts to show up when normal maps are blended, since this is

then an interpolation of rotation operators, with all the complexity that goes with it (for an excellent review, see the article about Reoriented Normal Mapping [5] here).

Solution of the Cotangent Frame

The problem to be solved for our purpose is the opposite as that of Blinn, the perturbed normal is known (from the normal map), but the cotangent frame is unknown. I'll give a short revision of how I originally solved it. Define the unknown cotangents $\mathbf{T} = \nabla u$ and $\mathbf{B} = \nabla v$ as the gradients of the texture coordinates u and v as functions of position $\mathbf{p}$, such that

$$\mathrm{d}u = \mathbf{T} \cdot \mathrm{d}\mathbf{p}, \qquad \mathrm{d}v = \mathbf{B} \cdot \mathrm{d}\mathbf{p},$$

where $\cdot$ is the dot product. The gradients are constant over the surface of an interpolated triangle, so introduce the edge differences $\Delta u_{1,2}$, $\Delta v_{1,2}$ and $\Delta \mathbf{p}_{1,2}$. The unknown cotangents have to satisfy the constraints

$$\begin{aligned} \Delta u_1 &= \mathbf{T} \cdot \Delta \mathbf{p}_1, & \Delta v_1 &= \mathbf{B} \cdot \Delta \mathbf{p}_1, \\ \Delta u_2 &= \mathbf{T} \cdot \Delta \mathbf{p}_2, & \Delta v_2 &= \mathbf{B} \cdot \Delta \mathbf{p}_2, \\ 0 &= \mathbf{T} \cdot \Delta \mathbf{p}_1 \times \Delta \mathbf{p}_2, & 0 &= \mathbf{B} \cdot \Delta \mathbf{p}_1 \times \Delta \mathbf{p}_2, \end{aligned}$$

where $\times$ is the cross product. The first two rows follow from the definition, and the last row ensures that $\mathbf{T}$ and $\mathbf{B}$ have no component in the direction of the normal. The last row is needed otherwise the problem is underdetermined. It is straightforward then to express the solution in matrix form. For $\mathbf{T}$,

$$\mathbf{T} = \begin{pmatrix} \Delta \mathbf{p}_1 \\ \Delta \mathbf{p}_2 \\ \Delta \mathbf{p}_1 \times \Delta \mathbf{p}_2 \end{pmatrix}^{-1} \begin{pmatrix} \Delta u_1 \\ \Delta u_2 \\ 0 \end{pmatrix},$$

and analogously for $\mathbf{B}$ with Δv.

Into the Shader Code

The above result looks daunting, as it calls for a matrix inverse in every pixel in order to compute the cotangent frame! However, many symmetries can be exploited to make that almost disappear. Below is an example of a function written in GLSL to calculate the inverse of a 3×3 matrix. A similar function written in HLSL appeared in the

book, and then I tried to optimize the hell out of it. Forget this approach as we are not going to need it at all. Just observe how the adjugate and the determinant can be made from cross products:

```
mat3 inverse3x3( mat3 M )
{
    // The original was written in HLSL, but this is GLSL,
    // therefore
    // - the array index selects columns, so M_t[0] is the
    //   first row of M, etc.
    // - the mat3 constructor assembles columns, so
    //   cross( M_t[1], M_t[2] ) becomes the first column
    //   of the adjugate, etc.
    // - for the determinant, it does not matter whether it is
    //   computed with M or with M_t; but using M_t makes it
    //   easier to follow the derivation in the text
    mat3 M_t = transpose( M );
    float det = dot( cross( M_t[0], M_t[1] ), M_t[2] );
    mat3 adjugate = mat3( cross( M_t[1], M_t[2] ),
                          cross( M_t[2], M_t[0] ),
                          cross( M_t[0], M_t[1] ) );
    return adjugate / det;
}
```

We can substitute the rows of the matrix from above into the code, then expand and simplify. This procedure results in a new expression for $\mathbf{T}$. The determinant becomes $|\Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2|^2$, and the adjugate can be written in terms of two new expressions, let's call them $\Delta\mathbf{p}_{1\perp}$ and $\Delta\mathbf{p}_{2\perp}$ (with $\perp$ read as 'perp'), which becomes

$$\mathbf{T} = \frac{1}{|\Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2|^2} \begin{pmatrix} \Delta\mathbf{p}_{2\perp} \\ \Delta\mathbf{p}_{1\perp} \\ \Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2 \end{pmatrix}^{\mathsf{T}} \begin{pmatrix} \Delta u_1 \\ \Delta u_2 \\ 0 \end{pmatrix},$$

$$\Delta\mathbf{p}_{2\perp} = \Delta\mathbf{p}_2 \times (\Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2),$$
$$\Delta\mathbf{p}_{1\perp} = (\Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2) \times \Delta\mathbf{p}_1.$$

As you might guessed it, $\Delta\mathbf{p}_{1\perp}$ and $\Delta\mathbf{p}_{2\perp}$ are the *perpendiculars* to the triangle edges in the triangle plane. Say Hello! They are, again, *covectors* and form a proper basis for *cotangent space*. To simplify things further, observe:

- The last row of the matrix is irrelevant since it is multiplied with zero.

- The other matrix rows contain the perpendiculars ($\Delta\mathbf{p}_{1\perp}$ and $\Delta\mathbf{p}_{2\perp}$), which after transposition just multiply with the texture edge differences.
- The perpendiculars can use the interpolated vertex normal **N** instead of the face normal $\Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2$, which is simpler and looks even nicer.
- The determinant (the expression $|\Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2|^2$) can be handled in a special way, which is explained below in the section about scale invariance.

Taken together, the optimized code is shown below, which is even simpler than the one I had originally published, and yet higher quality:

```
mat3 cotangent_frame( vec3 N, vec3 p, vec2 uv )
{
    // get edge vectors of the pixel triangle
    vec3 dp1 = dFdx( p );
    vec3 dp2 = dFdy( p );
    vec2 duv1 = dFdx( uv );
    vec2 duv2 = dFdy( uv );

    // solve the linear system
    vec3 dp2perp = cross( dp2, N );
    vec3 dp1perp = cross( N, dp1 );
    vec3 T = dp2perp * duv1.x + dp1perp * duv2.x;
    vec3 B = dp2perp * duv1.y + dp1perp * duv2.y;

    // construct a scale-invariant frame
    float invmax = inversesqrt( max( dot(T,T), dot(B,B) ) );
    return mat3( T * invmax, B * invmax, N );
}
```

SCALE INVARIANCE

The determinant $|\Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2|^2$ was left over as a scale factor in the above expression. This has the consequence that the resulting cotangents **T** and **B** are not scale invariant, but will vary inversely with the scale of the geometry. It is the natural consequence of them being gradients. If the scale of the geomtery increases, and everything else is left unchanged, then the change of texture coordinate per unit change of position gets smaller, which reduces $\mathbf{T} = \nabla u = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}\right)$ and similarly **B** in relation to **N**. The effect of all this is a diminished pertubation of the normal when the scale of the geometry is increased, as if a heightfield was stretched.

Obviously this behavior, while totally logical and correct, would limit the usefulness of normal maps to be applied on different scale geometry. My solution was and still is to ignore the determinant and just normalize $\mathbf{T}$ and $\mathbf{B}$ to whichever of them is largest, as seen in the code. This solution preserves the relative lengths of $\mathbf{T}$ and $\mathbf{B}$, so that a skewed or stretched cotangent space is sill handled correctly, while having an overall scale invariance.

NON-PERSPECTIVE OPTIMIZATION

As the ultimate optimization, I also considered what happens when we can assume $\Delta\mathbf{p}_1 = \Delta\mathbf{p}_{2\perp}$ and $\Delta\mathbf{p}_2 = \Delta\mathbf{p}_{1\perp}$. This means we have a right triangle and the perpendiculars fall on the triangle edges. In the pixel shader, this condition is true whenever the screen-projection of the surface is without perspective distortion. There is a nice figure demonstrating this fact in [4]. This optimization saves another two cross products, but in my opinion, the quality suffers heavily should there actually be a perspective distortion.

Putting it together

To make the post complete, I'll show how the cotangent frame is actually used to perturb the interpolated vertex normal. The function `perturb_normal` does just that, using the backwards view vector for the vertex position (this is ok because only differences matter, and the eye position goes away in the difference as it is constant).

```
vec3 perturb_normal( vec3 N, vec3 V, vec2 texcoord )
{
    // assume N, the interpolated vertex normal and
    // V, the view vector (vertex to eye)
    vec3 map = texture2D( mapBump, texcoord ).xyz;
#ifdef WITH_NORMALMAP_UNSIGNED
    map = map * 255./127. - 128./127.;
#endif
#ifdef WITH_NORMALMAP_2CHANNEL
    map.z = sqrt( 1. - dot( map.xy, map.xy ) );
#endif
#ifdef WITH_NORMALMAP_GREEN_UP
    map.y = -map.y;
#endif
    mat3 TBN = cotangent_frame( N, -V, texcoord );
    return normalize( TBN * map );
}
```

```
varying vec3 g_vertexnormal;
varying       vec3 g_viewvector;  // camera pos - vertex pos
```

```
varying vec2 g_texcoord;

void main()
{
    vec3 N = normalize( g_vertexnormal );

#ifdef WITH_NORMALMAP
    N = perturb_normal( N, g_viewvector, g_texcoord );
#endif

    // ...
}
```

THE GREEN AXIS

Both OpenGL and DirectX place the texture coordinate origin at the start of the image pixel data. The texture coordinate (0,0) is in the corner of the pixel where the image data pointer points to. Contrast this to most 3-D modeling packages that place the texture coordinate origin at the lower left corner in the uv-unwrap view. Unless the image format is bottom-up, this means the texture coordinate origin is in the corner of the first pixel of the last image row. Quite a difference!

An image search on Google reveals that there is no dominant convention for the green channel in normal maps. Some have green pointing up and some have green pointing down. My artists prefer green pointing up for two reasons: It's the format that 3ds Max expects for rendering, and it supposedly looks more natural with the 'green illumination from above', so this helps with eyeballing normal maps.

SIGN EXPANSION

The sign expansion deserves a little elaboration because I try to use signed texture formats whenever possible. With the unsigned format, the value 0.5 cannot be represented exactly (it's between 127 and 128). The signed format does not have this problem, but in exchange, has an ambiguous encoding for -1 (can be either -127 or -128). If the hardware is incapable of signed texture formats, I want to be able to pass it as an unsigned format and emulate the exact sign expansion in the shader. This is the origin of the seemingly odd values in the sign expansion.

In Hindsight

The original article in ShaderX[5] was written as a proof-of-concept. Although the algorithm was tested and worked, it was a little expensive for that time. Fast forward to today and the picture has changed. I am now employing this algorithm in real-life projects for great benefit. I no longer bother with tangents as vertex attributes and all the

associated complexity. For example, I don't care whether the COLLADA exporter of Max or Maya (yes I'm relying on COLLADA these days) output usable tangents for skinned meshes, nor do I bother to import them, because I don't need them! For the artists, it doesn't occur to them that an aspect of the asset pipeline is missing, because It's all natural: There is a geometry, there are texture coordinates and there is a normal map, and *just works*.

TAKE AWAY

There are no 'tangent frames' when it comes to normal mapping. A tangent frame which includes the normal is logically ill-formed. All there is are cotangent frames in disguise when the frame is orthogonal. When the frame is not orthogonal, then tangent frames will stop working. Use cotangent frames instead.

References

[1] James Blinn, "Simulation of wrinkled surfaces", SIGGRAPH 1978
http://research.microsoft.com/pubs/73939/p286-blinn.pdf

[2] Mark Peercy, John Airey, Brian Cabral, "Efficient Bump Mapping Hardware", SIGGRAPH 1997
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.4736

[3] Mark J Kilgard, "A Practical and Robust Bump-mapping Technique for Today's GPUs", GDC 2000
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.537

[4] Christian Schüler, "Normal Mapping without Precomputed Tangents", ShaderX 5, Chapter 2.6, pp. 131 – 140

[5] Colin Barré-Brisebois and Stephen Hill, "Blending in Detail",
http://blog.selfshadow.com/publications/blending-in-detail/

This entry was posted in **Articles** and tagged **gem**, **math**, **normal mapping** by **christian**. Bookmark the **permalink [http://www.thetenthplanet.de/archives/1180]** .

101 THOUGHTS ON "FOLLOWUP: NORMAL MAPPING WITHOUT PRECOMPUTED TANGENTS"

devsh
on **24.05.2018 at 16:08** said:

You do this in a terribly roundabout way, it's much easier to do

"'

vec3 denormTangent = dFdx(texCoord.y)*dFdy(vPos)-dFdx(vPos)*dFdy(texCoord.y);
vec3 tangent = normalize(denormTangent-smoothNormal*dot(smoothNormal,denormTangent));

vec3 normal = normalize(smoothNormal);
vec3 bitangent = cross(normal,tangent);
"'

> christian
> on **30.06.2018 at 15:03** said:
>
> Hi devsh,
> that would be equivalent to what the article describes as the "non-perspective-optimization". In this case you're no longer making a distinction between tangents and co-tangents. It only works if the tangent frame is orthogonal. In case of per-pixel differences taken with dFdx etc, it would only be correct if the mesh is displayed without perspective distortion, hence the name.

Pingback: Normal and normal mapping – One Line to rule them all

Zagolski
on **07.03.2019 at 19:08** said:

Hello!
First I want to thank for the algorithm, this is a very necessary thing. He works on my HLSL!
But there is a bug. When approaching the camera close to the object, a strong noise of pixel texture begins. This is a known problem; no one has yet solved it. Do you have any decisions on this, have you thought about this?

> christian
> on **08.03.2019 at 09:54** said:
>
> Hi Zagolski
> This behaviour is likely that the differences between pixels of the texture coordinate become too small for floating point precision and are then rounded to zero, which leads to a divide by zero down the line.
>
> Try to eliminate any „half“ precision variable that may affect the texture coordinate or view vector, if there are any.

Pingback: Skin Rendering with Texture Space Diffusion – Polysoup Kitchen

Anonymous
on **12.12.2019 at 08:29** said:

Wonderful！！！