2.6

Normal Mapping without Precomputed Tangents

Christian Schüler, **Phenomic**

Introduction

Tangent-space normal mapping is a well-known technique to compute per-pixel lighting with small surface perturbations stored in a texture map, as pioneered by [Kilgard00] and [Peercy97]. In virtually every implementation today, a tangent frame needs to be precomputed from partial derivatives of texture coordinates at each vertex of a surface and stored as vertex attributes.

This article shows how the tangent frame can be generated inside the pixel shader using derivative instructions. Since the tangent frame is computed per pixel, there is no need to store precomputed tangent frames. No constraints are placed on the way texture coordinates are assigned to vertices. For instance, UV-mirroring and procedural texture coordinates are perfectly possible.

Review of Tangent Space Normal Mapping

A normal map is a texture that stores information about how to rotate a normal vector into a perturbed position. The operation is

$$\mathbf{N}' = \underline{\mathbf{R}}(\mathbf{N}),$$

where **N**' is the perturbed normal we want to produce, **N** is the interpolated surface normal we know from the vertices, and $\underline{\mathbf{R}}$ is a rotation operator somehow stored in the normal map. From the many ways possible to encode a rotation operator, a certain representation is particularly easy in tangent space.

Tangent space is a coordinate system that is locally aligned with the surface, such that the surface normal **N** is a constant unit vector (usually in the z-direction). In this space one can express a rotation matrix acting on **N** with just its bottom row, since the first two components of **N** are zero. Assuming that **R** is a rotation matrix in tangent space, it follows that [Kilgard00]

$$\mathbf{N}' = \mathbf{N}\,\mathbf{R} \rightarrow$$
$$\mathbf{N}' = (0,0,1)\,\mathbf{R} \rightarrow$$
$$\mathbf{N}' = (\mathbf{R}_{31}, \mathbf{R}_{32}, \mathbf{R}_{33}).$$

In essence, a tangent space normal map contains the bottom row of the rotation matrix **R** as a three-component vector.

The coordinate basis for the tangent space, the *tangent frame*, is usually given as a set of three vectors: the *tangent* **T**, *binormal* **B**, and surface normal **N**. The first two (the *tangents*, for short) lie in the surface plane. A code snippet to perform normal perturbation using a tangent-space normal map could look like this:

```
float3 perturb_normal( float3 T, float3 B, float3 N, float2 texcoord )
    {
        // build a tangent frame as float3x3 for convenience
        float3x3 tangent_frame = float3x3( T, B, N );

        // read the perturbed normal from the normal map
        float3 perturbed_normal = tex2D( normalmap, texcoord );

        // sign-expand (for a normal map in unsigned RGB format)
        perturbed_normal = 2 * perturbed_normal - 1;

        // and transform the perturbed normal out of tangent space
        // (into whatever space T, B and N were originally expressed in,
        // usually world).
        return normalize( mul( perturbed_normal, tangent_frame ) );
    }
```

Computation of Tangent Frames

As mentioned in the introduction, normal mapping involves precomputing tangent frames and storing them as vertex attributes. Usually **T** is taken from the partial derivative of the u texture coordinate, and **B** is taken from the partial derivative of the v texture coordinate with respect to a point $\mathbf{p}(u,v)$ in world space $x,\ y,\ z$:

$$\mathrm{T} \propto \frac{\partial u}{\partial \mathbf{p}} = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}\right), \qquad \mathrm{B} \propto \frac{\partial v}{\partial \mathbf{p}} = \left(\frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}, \frac{\partial v}{\partial z}\right).$$

In other words, the tangent vector points to the direction of increasing u, and the binormal points toward increasing v (see Figure 2.6.1). As long as the surface $\mathbf{p}(u,v)$ consists of a set of planar polygons with interpolated u and v, the partial derivatives are constant across each polygon and can be calculated once per face. The partial derivatives are unbounded in magnitude and in general not orthogonal. The tangent frame, on the other hand, is usually orthonormalized and averaged at each vertex from adjacent polygons.

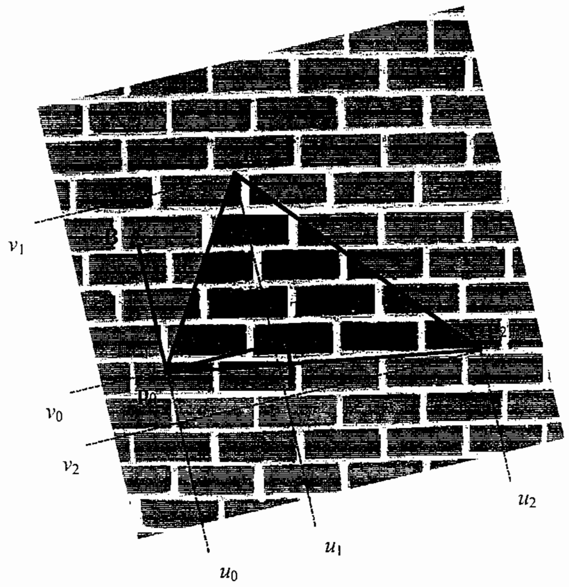

FIGURE 2.6.1 Textured triangle with texture coordinates, tangent, and binormal.

If all that is known is a triangle with vertices $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$ and texture coordinates at these vertices, u_0, u_1, u_2, and v_0, v_1, v_2, how are tangents computed? First, we introduce edge differences:

$$\begin{matrix} \Delta\mathbf{p}_1 = \mathbf{p}_1 - \mathbf{p}_0 \\ \Delta\mathbf{p}_2 = \mathbf{p}_2 - \mathbf{p}_0 \end{matrix}, \quad \begin{matrix} \Delta u_1 = u_1 - u_0 \\ \Delta u_2 = u_2 - u_0 \end{matrix}, \quad \begin{matrix} \Delta v_1 = v_1 - v_0 \\ \Delta v_2 = v_2 - v_0 \end{matrix}.$$

In the next step we observe that $\Delta\mathbf{p}$ can be related to Δu and Δv via dot products. For example, Figure 2.6.1 has the angle α between the lower triangle edge $\Delta\mathbf{p}_2$ and the tangent vector T. Assume for this time that the length of **T** corresponds to the magnitude of the *u*-gradient. Then we can write

$$\Delta u = \Delta\mathbf{p}\,|\mathrm{T}|\cos\alpha \quad \rightarrow \quad \Delta u = \Delta\mathbf{p}\cdot\mathrm{T},$$

and similarly for the binormal and Δv. It is therefore easy to formulate three conditions that we can use to solve for both tangent and binormal. Two conditions stem from the dot product relations, and the third condition constrains the vector in question to the triangle plane; the latter is expressed as a dot product with a perpendicular vector $\Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2$ (the plane normal). Taken together, all conditions that must be met, are.

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

These systems of equations can be solved in matrix form. We construct a matrix with the rows $\Delta\mathbf{p}_1$, $\Delta\mathbf{p}_2$ and $\Delta\mathbf{p}_1 \times \Delta\mathbf{p}_2$ to be multiplied with one of T or **B**, and an inversion then yields the solution. Following **T** as an example we have

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

A similar expression yields **B** when *u* is replaced with *v*.

Since each polygon has its own set of tangents, no vertex containing a tangent frame could be shared with another polygon. An involved procedure usually follows, which averages tangent frames from adjacent polygons while respecting hard edges between polygons and duplicating vertices at borders where regions of left- and right-handed (mirrored) texture mapping intersect. All this complexity is of course not needed if tangent frames are not stored as vertex attributes.

Moving to the Pixel Shader

The computation of the tangent frame can be done in the pixel shader. All we need are edges of triangles at which to throw our algorithm. It turns out that we can construct such triangles at every pixel by means of the `ddx` and `ddy` derivative instructions. On current hardware, if a position vector and a pair of texture coordinates are fed to the pixel shader, the `ddx` and `ddy` instructions will calculate the edge differences of triangles over a 2×2 pixel area (see Figure 2.6.2).

A straightforward implementation of the algorithm from the previous section using pixel triangles is very easy. The only advanced prerequisite we need is a function to invert a 3×3 matrix, since there is no built-in function to perform this task in a pixel shader. Later we will find ways to optimize this expensive operation to something cheaper, but for now let's consider a full matrix inversion:

```
float3x3 invert_3x3( float3x3 M )
{
    float det = dot( cross( M[0], M[1] ), M[2] );
    float3x3 T = transpose( M );
    return float3x3(
        cross( T[1], T[2] ),
        cross( T[2], T[0] ),
        cross( T[0], T[1] ) ) / det;    }
```

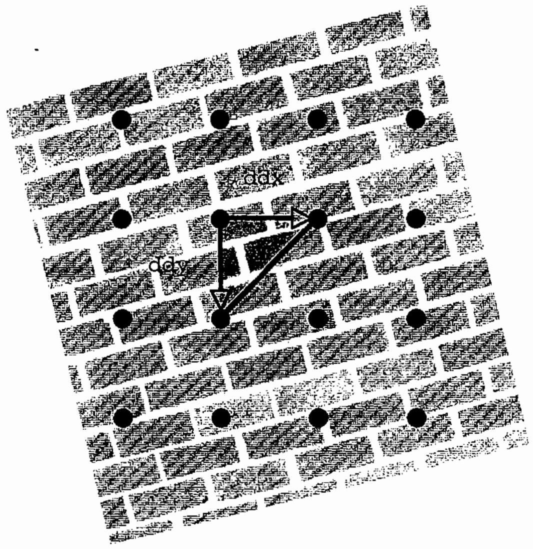

FIGURE 2.6.2 A 2 × 2 pixel triangle spanned by the ddx and ddy instructions on the pixel grid.

Using this function, we create a function that takes an interpolated surface normal **N**, a position vector **p**, and texture coordinates *u* and *v* and returns a complete tangent frame that is ready to be used in lighting calculations:

```
float3x3 compute_tangent_frame( float3 N, float3 p, float2 uv )
{
    // get edge vectors of the pixel triangle
    float3 dp1 = ddx( p );
    float3 dp2 = ddy( p );
    float2 duv1 = ddx( uv );
    float2 duv2 = ddy( uv );

    // solve the linear system
    float3x3 M = float3x3( dp1, dp2, cross( dp1, dp2 ) );
    float3x3 inverseM = invert_3x3( M );
    float3 T = mul( inverseM, float3( duv1.x, duv2.x, 0 ) );
    float3 B = mul( inverseM, float3( duv1.y, duv2.y, 0 ) );

    // construct tangent frame
    // (* see discussion regarding the square patch assumption)
    float maxLength = max( length(T), length(B) );
    return float3x3( T / maxLength, B / maxLength, N );
}
```

Note that it is possible to replace **p** with a vector other than the proper vertex position. Since only differences matter, a constant offset on **p** does not change the result. The view vector **V**, which is often used in lighting calculations, already contains the vertex position subtracted by the camera position. Since the camera position is constant, the view vector is a suitable replacement for **p** with respect to tangent frame computation, so no additional attributes are needed if texture coordinates and **V** are already fed to the pixel shader.

Figure 2.6.3a shows a shot from a normal-mapped model with per-pixel Blinn-Phong lighting, with orthonormalized per-vertex tangent frames provided per default in NVIDIA's FX Composer [NVIDIA06]. Figure 2.6.3b shows the same model using per-pixel tangent frames computed by the algorithm discussed above. By and large, both methods tend to produce very similar results. A notable difference, however, can be seen at regions of high texture anisotropy. Compare the region around the nozzle of the teapot where the texture is severely squashed (magnified respectively in the inlets). In Figure 2.6.3a, the light direction is wrong on some of the circular spots, and the stripes seem to fade out toward higher anisotropy. In Figure 2.6.3b, no such problems are apparent. The reason behind this is that we have relaxed the requirement of an orthonormal tangent frame, which is discussed in the next section.

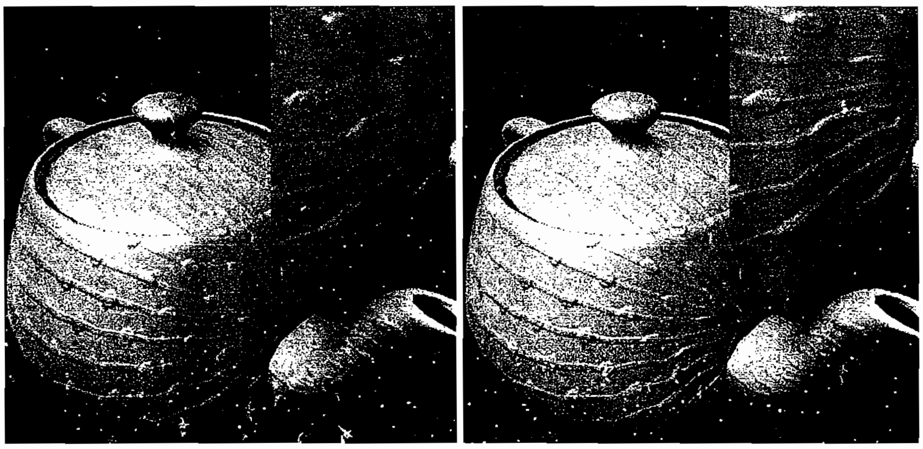

FIGURE 2.6.3 Normal mapping under differently calculated tangent frames. **(A)** Tangent frame from interpolated vertex attributes. **(B)** Tangent frame calculated inside pixel shader.

The Square Patch Assumption

The authors [Peercy97] and [Kilgard00] employed the *square patch assumption*, which states that the u- and v-gradients are assumed to be equal in magnitude and orthogonal to each other. In other words, the texture image on the mapped surface should look like a square patch, neither stretched nor sheared. With this assumption, the tangent frame was constrained to be orthonormal.

It is not strictly necessary for us to keep up the square patch assumption; quite to the contrary, it may be beneficial to give it up for special applications. Consider Figure 2.6.4, which shows an array of embossed points, lit from the upper-left corner. In Figure 2.6.4a, the square patch assumption holds. In Figures 2.6.4b to 2.6.4d, the texture image was stretched and sheared by means of texture coordinate manipulation inside the pixel shader. An orthonormal tangent frame becomes obsolete, and the produced lighting is wrong (Figure 2.6.4b). A tangent frame not bound to orthogonality has no problems adjusting to the stretched texture, and the lighting is correct (Figure 2.6.4c; observe the location of the specular highlights).

The alert reader might have noticed that our pixel shader from the previous section contains a quasi normalization for **T** and **B** by their common maximal length. This is a hack, but if it was left out, he bump depth would also have been dependent on object scale—the larger the object in world space, the fainter the bumps. In terms of partial derivatives, this is a correct behavior (think of a height field being stretched), but probably not desired.

A last word on nonorthogonal tangent frames: you have probably been freely using tangent frames given in **T**, **B**, **N** form as transforms for going to and from tangent space. In the general case, the transpose does not equal the inverse, and you can use the tangent frame only to transform out of tangent space, not into it.

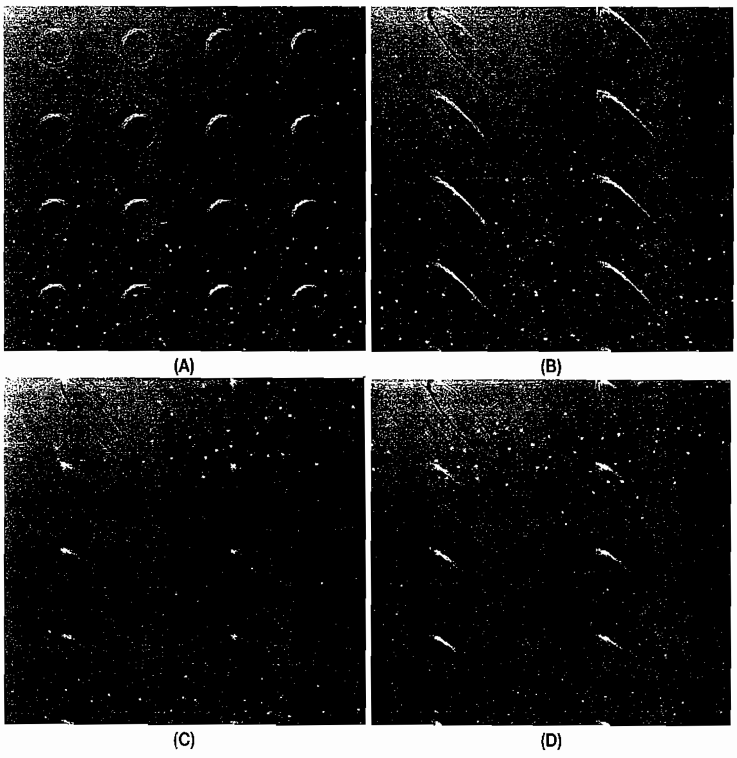

FIGURE 2.6.4 Discussion of the square patch assumption. (**A**) Square patch condition; (**B**) lighting with orthonormalized tangent frame; (**C**) lighting with general tangent frame; (**D**) lighting with normalized tangent frame.

Optimization

The function `compute_tangent_frame` as presented above accounts for an overhead of 46 pixel shader instructions when compared to the case when the tangent frame is simply read and normalized from vertex attributes.

The first step in optimization is to independently normalize the tangents **T** and **B** to unit length, while still allowing them to be nonorthogonal with each other. This produces lighting as shown in Figure 2.6.4d. The specular highlight on the stretched texture is a little off when compared to the general solution in Figure 2.6.4c, but this error is well worth the performance gain. Since the tangents are constrained to unit length at the end of the computation, we can disregard any scale factors on the way up to the result. This allows us to eliminate the determinant from the matrix inverse. The first optimized version of the `compute_tangent_frame` function thus might look like this:

```
float3x3 compute_tangent_frame_01( float3 N, float3 p, float2 uv )
{
    ...
    // no determinant since result gets normalized
    float3x3 inverseM = invert_3x3_nodet( M );
    ...

    // construct tangent frame
    return float3x3( normalize(T), normalize(B), N );
}
```

Here, the function `invert_3x3_nodet` is just a variant of the matrix inversion code that doesn't compute a determinant. With this optimization, the cost of tangent frame computation has come down to 31 instructions.

The next step of the optimization exploits the zero components found in the solution vectors and collapses a hidden double transpose. Consider these lines:

```
    float3 T = mul( inverseM, float3( duv1.x, duv2.x, 0 ) );
    float3 B = mul( inverseM, float3( duv1.y, duv2.y, 0 ) );
```

If these multiplications could be transposed, the zero components could eliminate an entire matrix row. To achieve this, we need the inverse transpose of **M**, instead of just the inverse, which turns out to be less expensive to compute, since we can eliminate the transpose already happening in the inverse. The modified code should then look like this:

```
float3x3 compute_tangent_frame_02( float3 N, float3 p, float2 uv )
{
    ...
    // solve linear system
```

```
        // hidden double transpose revealed, only needs float2x3
        float3x3 M = float3x3( dp1, dp2, cross( dp1, dp2 ) );
        float2x3 inversetransposeM =
            float2x3( cross( M[1], M[2] ), cross( M[2], M[0] ) );
        float3 T = mul( float2( duv1.x, duv2.x ), inversetransposeM );
        float3 B = mul( float2( duv1.y, duv2.y ), inversetransposeM );
        ...
    }
```

In this function the calculation of the inverse transpose has been inlined into the function body since it is nothing more than two cross products. Note also how the inverse transpose matrix has been reduced to a `float2x3`, since the last row doesn't contribute. This step of optimization reduces the cost of tangent frame computation to 17 pixel shader instructions and is visually equivalent to the previous step.

The last step of optimization exploits the fact that we have made an inverse transpose explicit. An inverse transpose reduces to a nonoperation (up to a scale factor, but scale doesn't matter already) if the underlying matrix is orthogonal. Let's assume that **M** is orthogonal:

```
    float3x3 compute_tangent_frame_03( float3 N, float3 p, float2 uv
)
    {
        ...
        // solve linear system
        // (not much solving is left going here)
        float2x3 M = float2x3( dp1, dp2 );
        float3 T = mul( float2( duv1.x, duv2.x ), M );
        float3 B = mul( float2( duv1.y, duv2.y ), M );
        ...
    }
```

In this minimalist version the cross products have been eliminated and the edge differences just multiply with themselves. This version has a cost of 14 pixel shader instructions for the tangent frame computation, not much less than the version before, but how likely is it that the matrix M will be orthogonal anyway? The matrix **M** consists of the edge differences of the vertex positions of the screen triangles in world space. So **M** is reasonably orthogonal if the square patch assumption holds for the *screen projection*. In other words, as long as perspective distortion is low, the assumption holds. We have to expect artifacts, however, if there is screen-space anisotropy. Compare Figure 2.6.5a (rendered with `compute_tangent_frame_02`) to Figure 2.6.5b (rendered with `compute_tangent_frame_03`) on a perspective plane. While the close-up region of the figures agrees very well, there are notable differences in the farther away region.

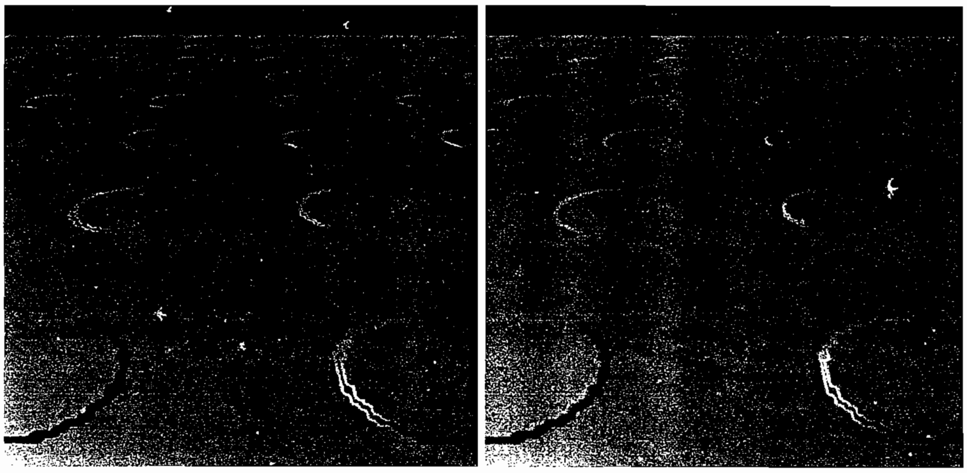

FIGURE 2.6.5 Dependency of optimized tangent frame conputation of perspective distortion. (**A**) Normalized tangent frame. (**B**) Normalized plus assuming the inverse transpose to be a non-operation.

Conclusion

The main contribution of this article is to show a way to transfer the tangent frame computation into the pixel shader. This eliminates the need to store precomputed tangent frames as vertex attributes and their associated complexity. A general solution for the pixel shader is presented that can handle arbitrary texture space configurations, such as procedurally distorted texture coordinates. Several optimizations from the general solution are discussed both on a theoretical and practical level. Per-pixel tangent frames can be bought for as low as 14 pixel shader instructions. The complete source code of the Shader can be found on the CD-ROM.

ON THE CD

References

[Kilgard00] Mark J. Kilgard. "A Practical and Robust Bump-mapping Technique for Today's GPUs." Game Developers Conference 2000. Available online at *http://www.nvidia.com/object/Practical_Bumpmapping_Tech.html.*

[NVIDIA06] FX Composer 1.8. NVIDIA Corporation. Available online at *http://developer.nvidia.com.*

[Peercy97] Mark Peercy, John Airey, and Brian Cabral. "Efficient Bump Mapping Hardware." *Computer Graphics* (Proc. Siggraph '97) (1997): pp. 303–306. Available online at *http://citeseer.ist.psu.edu/peercy97efficient.html.*