Surface Gradient–Based Bump Mapping Framework

Morten S. Mikkelsen
Unity Technologies, USA

Figure 1. The left pair of images on the top row show a low-resolution model in wireframe and a lit version using a normal map baked from a high-resolution model to the primary set of texture coordinates. In the top right pair of images, we see a normal map of generic detail applied to the clothes using the primary and secondary set of texture coordinates, respectively. Furthermore, we apply details to the hair and beard using the 3D procedural bump map known as `fractalsum` noise. The corresponding close-ups are shown in the bottom row. As we can see, by using the secondary set of texture coordinates, the artist is able to reorient the tiling direction.

Abstract

In this paper, a new framework is proposed for the layering and compositing of bump maps, including support for multiple sets of texture coordinates as well as procedurally generated texture coordinates and geometry. Furthermore, we provide proper support for bump maps defined on a volume, such as decal projectors, triplanar projection, and noise-based functions.

 ISSN 2331-7418

1. Introduction

Since the year 2000, as described in [Kilgard 2000], the approach to normal mapping in real-time 3D graphics has remained largely the same. That is, a transformation by a 3×3 TBN (tangent, bitangent, and normal) matrix, where the tangent and bitangent are precomputed offline at the vertex level. Today, this conventional approach is beginning to show its age:

1. In modern computer graphics, material layering is critical to achieving rich and complex environments. However, conventional normal mapping does not extend to multiple sets of texture coordinates, since game engines typically only store one tangent space per vertex (for memory and performance reasons).

2. Traditional normal mapping does not allow for proper order-independent blending between different forms of bump influences, such as separate sets of texture coordinates, object-space normal maps, and volume bump maps (such as triplanar projection, decal projectors, and volumetric noise).

3. Geometry of a more procedural nature does not work well with traditional normal mapping either, since generating a per-vertex tangent space in real time—for every frame—is impractical. Some examples are blend shapes, tessellation, cloth, water, and procedural trees.

For blend shapes, one might alternatively store a pregenerated tangent and sign bit with each vertex of a blend shape. However, memory footprint is a concern, plus linearly blending sets of tangent vectors across several shapes will often produce incorrect results. In the context of tessellation, generating a vertex-level tangent space directly from the control mesh is not meaningful, since it may be very coarse and deviate significantly from the limit surface. Furthermore, linear interpolation of tangents will not provide a vector that is tangent to the limit surface.

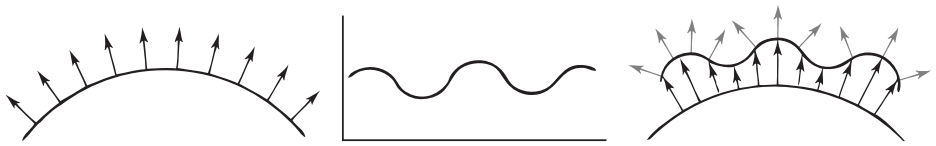
The reader of this paper is expected to be familiar with shader authoring and normal mapping, and also possess a basic understanding of differential geometry.

2. Previous Work

Bump Mapping was originally introduced by Blinn [1978] to mimic high-frequency detail on curved surfaces. The surface S consists of a collection of charts where each chart has a known parameterization $\sigma : \mathbb{R}^2 \to \mathbb{R}^3$ and partial derivatives $\frac{\partial \sigma}{\partial s}$ and $\frac{\partial \sigma}{\partial t}$, where $(s, t) \in \mathbb{R}^2$. Furthermore, we are given a height map $H : \mathbb{R}^2 \to \mathbb{R}$, which is used to assign a value $H(s, t)$ to each corresponding point on S. A new virtual surface τ is defined as the displacement of σ by H along the surface normal $\vec{n}$ (see Figure 2):

$$\tau = \sigma + H \cdot \vec{n}. \tag{1}$$

However, rather than rendering the highly detailed surface τ, bump mapping uses a cheaper alternative, whereby the initial surface σ is rendered using the corresponding normal of τ, which creates the illusion of a detailed surface. This replacement of the initial normal $\vec{n}$ is referred to as the *perturbed normal*, $\vec{n}'$.



(a) Initial surface (b) Height map (c) Combined

Figure 2. (a) The initial surface σ, (b) the height map H, and (c) the combined result after displacement using Equation (1) and the new surface normals.

The normal of the displaced surface can be computed as the cross product of the partial derivatives of the parameterization for τ:

$$\begin{aligned}
\frac{\partial \tau}{\partial s} &= \frac{\partial \sigma}{\partial s} + \frac{\partial H}{\partial s} \cdot \vec{n} + H \cdot \frac{\partial \vec{n}}{\partial s}, \\
\frac{\partial \tau}{\partial t} &= \frac{\partial \sigma}{\partial t} + \frac{\partial H}{\partial t} \cdot \vec{n} + H \cdot \frac{\partial \vec{n}}{\partial t}, \\
\frac{\partial \tau}{\partial s} \times \frac{\partial \tau}{\partial t} &\simeq \frac{\partial \sigma}{\partial s} \times \frac{\partial \sigma}{\partial t} + \frac{\partial H}{\partial s} \cdot \left(\vec{n} \times \frac{\partial \sigma}{\partial t} \right) + \frac{\partial H}{\partial t} \cdot \left(\frac{\partial \sigma}{\partial s} \times \vec{n} \right).
\end{aligned} \tag{2}$$

Blinn's unnormalized perturbed normal is given by Equation (2) and uses approximations for $\frac{\partial \tau}{\partial s}$ and $\frac{\partial \tau}{\partial t}$ by omitting their last term. In this formulation, $\vec{n}$ is defined as normalizing $\frac{\partial \sigma}{\partial s} \times \frac{\partial \sigma}{\partial t}$. This presents a problem when shading, since the texture parameterization does not determine the forward-facing direction of the normal $\vec{n}$. This is solved in [Mikkelsen 2008] via an alternate derivation, where the normal to the graph of H,

$$\begin{aligned}
g(s,t) &= (s, t, H(s,t)), \\
\frac{\partial g}{\partial s} \times \frac{\partial g}{\partial t} &= \left(1, 0, \frac{\partial H}{\partial s}\right) \times \left(0, 1, \frac{\partial H}{\partial t}\right) \\
&= \left(-\frac{\partial H}{\partial s}, -\frac{\partial H}{\partial t}, 1\right),
\end{aligned} \tag{3}$$

is transformed to the surface σ using the transpose inverse of the matrix

$$N = \left[\begin{array}{c|c|c} \frac{\partial \sigma}{\partial s} & \frac{\partial \sigma}{\partial t} & \vec{n} \end{array} \right],$$

where the vector $\vec{n}$ is known to be unit length and perpendicular to the surface σ but can face in either direction. This transformation yields the vector $\vec{n}'$, and using the

scalar triple product it can be expressed as

$$\begin{aligned}\vec{n}' &= (N^{-1})^T \cdot \begin{bmatrix} -\frac{\partial H}{\partial s} \\ -\frac{\partial H}{\partial t} \\ 1 \end{bmatrix} \\ &= \vec{n} + \frac{(\vec{n} \times \frac{\partial \sigma}{\partial t})\frac{\partial H}{\partial s} + (\frac{\partial \sigma}{\partial s} \times \vec{n})\frac{\partial H}{\partial t}}{\vec{n} \bullet (\frac{\partial \sigma}{\partial s} \times \frac{\partial \sigma}{\partial t})}, \end{aligned} \tag{4}$$

where the symbol $\bullet$ represents the dot product. Furthermore, the denominator in the second term of Equation (4) is equal to

$$\begin{aligned}\det(N) &= \vec{n} \bullet \left(\frac{\partial \sigma}{\partial s} \times \frac{\partial \sigma}{\partial t} \right) \\ &= \pm \left\| \frac{\partial \sigma}{\partial s} \times \frac{\partial \sigma}{\partial t} \right\| \end{aligned}$$

and is positive when the parameterization is orientation preserving. When both equations are normalized, the formulation given here is identical to that of Blinn's Equation (2) except that there is no assumption that $\det(N)$ is positive.

A more compact and generalized version of Equation (4) was discovered by Mikkelsen [2010] in the form of

$$\vec{n}' = \vec{n} - \nabla_S H, \tag{5}$$

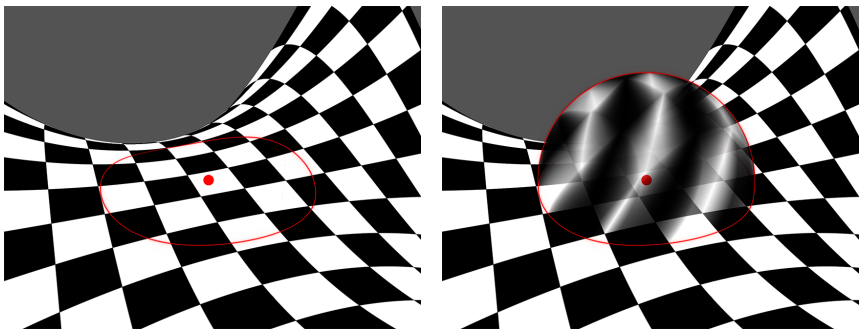
where $\nabla_S H$ is the surface gradient. More specifically, for a given point p on the surface S and an arbitrarily small open volume $B \subset \mathbb{R}^3$ containing p, if there exists a local extension $\widetilde{H} : B \to \mathbb{R}$ such that $\widetilde{H}$ is smooth and obeys $\widetilde{H}|_{B \cap S} = (H \circ \sigma^{-1})|_{B \cap S}$ (see Figure 3), then the surface gradient at p is by definition

$$\nabla_S H = \nabla \widetilde{H} - \vec{n} \cdot \left(\vec{n} \bullet \nabla \widetilde{H} \right), \tag{6}$$

which is the projection of the regular gradient $\nabla \widetilde{H} = (\frac{\partial \widetilde{H}}{\partial x}, \frac{\partial \widetilde{H}}{\partial y}, \frac{\partial \widetilde{H}}{\partial z})$ onto the tangent plane at p.

Any two such local extensions at p will produce the same surface gradient. To see this, we rotate the scene such that the tangent plane at p aligns with the XY-plane. In this case, for any choice of a local extension, the gradient $\nabla \widetilde{H}$ will only differ in the third component $\frac{\partial \widetilde{H}}{\partial z}$, which will be zero after projection onto the XY-plane to produce $\nabla_S \widetilde{H}$.

Equation (5) tells us that the perturbed normal depends on the shape of the surface S and the distribution of displacement values assigned by H, but not the underlying parameterization. However, to see that this formulation is in fact identical to Equation (4) and thus Blinn's perturbed normal, we can equate the second term of



(a) Cut-out region (b) Local extension

Figure 3. (a) A surface S with a 2D texture map H applied to it. (b) A local extension $\widetilde{H}$ at the point p (red dot) on S. The domain of this extension is an open volume $B \subset \mathbb{R}^3$, containing p, and is outlined in red. The corresponding red outline in (a) represents $B \cap S$.

Equation (5) with the second term of Equation (4), which leads to the following:

$$\begin{aligned} \nabla_S H &= \frac{(\frac{\partial \sigma}{\partial t} \times \vec{n})\frac{\partial H}{\partial s} + (\vec{n} \times \frac{\partial \sigma}{\partial s})\frac{\partial H}{\partial t}}{\vec{n} \bullet (\frac{\partial \sigma}{\partial s} \times \frac{\partial \sigma}{\partial t})} \\ &= \left(\begin{bmatrix} \frac{\partial H}{\partial s} & \frac{\partial H}{\partial t} & 0 \end{bmatrix} \cdot N^{-1} \right)^T . \end{aligned} \tag{7}$$

The vector given by Equation (7) is in the tangent plane of σ. However, to be the surface gradient, the dot product between it and some given unit-length tangent vector $\vec{v}$ must give the rate of change in the direction of $\vec{v}$. This property holds, since for a given linear combination $\vec{v} = a \cdot \frac{\partial \sigma}{\partial s} + b \cdot \frac{\partial \sigma}{\partial t}$, where $a, b \in \mathbb{R}$, we obtain $\nabla_S H \bullet \vec{v} = a \cdot \frac{\partial H}{\partial s} + b \cdot \frac{\partial H}{\partial t}$ when using Equation (7).

An important subtlety to Equation (4) is that the formulation is scale dependent. Though this is the expected behavior, it can be impractical (from a workflow standpoint) for an artist to have to adjust the scale of H relative to the extent of the surface. In real-time graphics, a very common scale-independent variant was introduced in [Peercy et al. 1997], which as an approximation assumes N is an orthogonal matrix stored at the vertex level, which implies $N = (N^{-1})^T$. We will refer to this matrix as M, where

$$\begin{aligned} \vec{t} &= \frac{\frac{\partial \sigma}{\partial u}}{\left\| \frac{\partial \sigma}{\partial u} \right\|}, \\ \vec{b} &= \text{sign}\left(\det(N)\right) \cdot \left(\vec{n} \times \vec{t}\right), \\ M &= \left[\begin{array}{c|c|c} & & \\ \vec{t} & \vec{b} & \vec{n} \\ & & \end{array} \right]. \end{aligned} \tag{8}$$

Furthermore, the partial derivatives are defined on the domain $(u, v) \in \mathbb{R}^2$, which represents an *unnormalized texture space* such that one unit corresponds to one pixel in the texture map for $H(u, v)$. This allows the partial derivatives of H to remain independent of the height map resolution $(\text{width}, \text{height})$. In the corresponding normalized parameter space $(s, t) \in \mathbb{R}^2$, one unit corresponds to the full texture, and the relation between the two is given by

$$(s, t) = \left(\frac{u + \frac{1}{2}}{\text{width}}, \frac{v + \frac{1}{2}}{\text{height}} \right). \tag{9}$$

By replacing $(N^{-1})^T$ in Equation (4) with M, we arrive at

$$\begin{aligned} \vec{m}_z \cdot \vec{n}' &\simeq M \cdot \vec{m} && (10) \\ &= \vec{t} \cdot \vec{m}_x + \vec{b} \cdot \vec{m}_y + \vec{n} \cdot \vec{m}_z, && (11) \end{aligned}$$

where

$$\vec{m} = \frac{\begin{bmatrix} -\frac{\partial H}{\partial u} & -\frac{\partial H}{\partial v} & 1 \end{bmatrix}^T}{\sqrt{\frac{\partial H}{\partial u}^2 + \frac{\partial H}{\partial v}^2 + 1}} \tag{12}$$

is the normalized version of Equation (3), hence the scaling of $\vec{n}'$ by the factor $\vec{m}_z$ in Equation (10). The vector $\vec{m}$ is referred to as the *tangent-space normal* and is stored per pixel in a normal map. Alternatively, we may think of $\vec{m}$ as the *perturbed normal* $\vec{n}'$ but normalized and transformed into the orthogonal frame of reference represented by M. An important observation is that although M is orthogonal at the vertex, the interpolated matrix is not, so $\vec{n}'$ must be renormalized before use in the pixel shader.

3. Modern Framework

What we want is a practical, modern framework for normal mapping that solves the limitations described in Section 1 and provides a uniform processing of all bump influences. The basis for such a framework is implicitly given by the surface gradient–based formulation of Equation (5), which provides a unified solution because of two important properties of the surface gradient:

1. It is a linear operator.
2. It depends on the shape of the surface and the given distribution of bumps across the surface but does not depend on the underlying parameterization.

The first property tells us that if a bump map is expressed as a linear combination of several bump maps, then the surface gradient of the function is equal to the corresponding linear combination of surface gradients:

$$\begin{aligned} H &= s_0 \cdot H_0 + s_1 \cdot H_1 + \cdots + s_n \cdot H_n, && (13) \\ \nabla_S H &= s_0 \cdot \nabla_S H_0 + s_1 \cdot \nabla_S H_1 + \cdots + s_n \cdot \nabla_S H_n. && (14) \end{aligned}$$

To be clear, we do not need the actual displacement maps $H_1, H_2, \ldots, H_n$ to use this framework: they are conceptual priors to the normal maps.

The second property tells us that it does not matter what the specific unwrap of texture coordinates looks like, or even if the bumps were mapped to the surface via a volume bump map.

Together, the two properties allow us to accumulate different forms of bump contributions as surface gradients provided that we can establish the surface gradient in each case. Concretely, the following structure for a bump mapping framework emerges from Equations (5) and (14):

1. Bump scales are applied by scaling each surface gradient by a user-specified displacement value.

2. All surface gradients are added together into a final surface gradient (Equation (14)).

3. Finally, a *resolve* is done by normalizing the difference between the initial unit length surface normal and the accumulated surface gradient (Equation (5)).

We can perform the resolve step using the function in Listing 1, and we can adjust the intensity of a bump map by modulating its surface gradient (or alternatively the derivative $(\frac{\partial H}{\partial u}, \frac{\partial H}{\partial v})$). The value `nrmBaseNormal` corresponds to $\vec{n}$ and is calculated and written to a static global during the prologue step outlined in Section 3.7.

```
float3 ResolveNormalFromSurfaceGradient(float3 surfGrad)
{
    return normalize(nrmBaseNormal - surfGrad);
}
```

Listing 1. Resolve function to establish the final perturbed normal.

As an aside, when a negative value is used to weight a surface gradient, the effect corresponds to inverting the bump map, since $\nabla_S(-H) = -\nabla_S H$. Doing so will reflect $\vec{n}'$ by $\vec{n}$, as indicated by Equation (5). We can even perform this operation directly on an already perturbed normal `v` by using `reflect(-v, nrmBaseNormal)`.

While the framework is straightforward in concept, it requires knowledge of how to obtain a surface gradient for each relevant scenario. This is what we will focus on in the following subsections.

3.1. Derivatives vs. Tangent-Space Normals

Tangent-space normal maps are the most common representation for bump mapping. This standard is compatible with our proposed framework, but as we shall see, on the

shader side it is more practical to convert the sampled value into a two-component derivative, on the fly, rather than use a three-component vector.

As indicated by Equation (12), we may convert the tangent-space normal $\vec{m} \in [-1;1]^3$ to the derivative $\vec{d} = (\frac{\partial H}{\partial u}, \frac{\partial H}{\partial v})$ using Equation (15) and use Equation (16) when only two components $(\vec{m}_x, \vec{m}_y)$ are available due to texture compression:

$$\vec{d} = \left(-\frac{\vec{m}_x}{\vec{m}_z}, -\frac{\vec{m}_y}{\vec{m}_z}\right) \tag{15}$$

$$= \frac{(-\vec{m}_x, -\vec{m}_y)}{\sqrt{\max\left(\varepsilon, 1 - \vec{m}_x^2 - \vec{m}_y^2\right)}}. \tag{16}$$

```
// Input: vM is tangent space normal in [-1;1].
// Output: convert vM to a derivative.
float2 TspaceNormalToDerivative(float3 vM)
{
    const float scale = 1.0/128.0;

    // Ensure vM delivers a positive third component using abs() and
    // constrain vM.z so the range of the derivative is [-128; 128].
    const float3 vMa = abs(vM);
    const float z_ma = max(vMa.z, scale*max(vMa.x, vMa.y));

    // Set to match positive vertical texture coordinate axis.
    const bool gFlipVertDeriv = false;
    const float s = gFlipVertDeriv ? -1.0 : 1.0;
    return -float2(vM.x, s*vM.y)/z_ma;
}
```

Listing 2. Conversion of tangent-space normal $\vec{m}$ to a derivative $\vec{d}$.

The implementation in Listing 2 returns the derivative $\vec{d} \in [-128;128]^2$, which corresponds to a maximum angle of 89.55°. The main motivation for using the derivative form rather than tangent-space normals in the shader is that it implicitly allows all bump contributions to conform to Equation (5), as we shall see in Section 3.2.

It is important to note that the correct setting for `gFlipVertDeriv` in Listing 2 is always a constant but is specific to your codebase. When `gFlipVertDeriv` is set to `false`, if $(0,0)$ represents the upper-left corner of the texture, then positive green in the normal map must represent down. Alternatively, if $(0,0)$ represents the lower-left corner, then positive green must represent up. If this is not the case, you will need to set `gFlipVertDeriv` to `true` in Listing 2. You will also, most likely, need to negate `sign_w` in Listing 9 depending on your existing build process and shaders. This is because the derivative must be with respect to the positive axis for s and for t.

3.2. TBN-Style Surface Gradient

Normal mapping in real-time graphics uses the scale-independent approach described in Section 2 and as given by Equation (11), where the vectors $\vec{t}$ and $\vec{b}$ represent the tangent and bitangent, respectively. As previously defined, $\vec{n}$ is the initial normalized surface normal and $\vec{m}$ is the tangent-space normal (Equation (12)). By leveraging Equation (15) and rearranging terms in Equation (11), we arrive at a derivative-based formulation, which corresponds to Equation (5):

$$\begin{aligned}\vec{n}' &= \frac{1}{\vec{m}_z} \cdot \left(\vec{t} \cdot \vec{m}_x + \vec{b} \cdot \vec{m}_y + \vec{n} \cdot \vec{m}_z\right) \\ &= \vec{n} - \left(\vec{t} \cdot \frac{-\vec{m}_x}{\vec{m}_z} + \vec{b} \cdot \frac{-\vec{m}_y}{\vec{m}_z}\right) \\ &= \vec{n} - \left(\vec{t} \cdot \vec{d}_x + \vec{b} \cdot \vec{d}_y\right). \end{aligned} \tag{17}$$

Note that the scale by $\frac{1}{\vec{m}_z} > 0$ will cancel out in the final resolve when $\vec{n}'$ is normalized before use (see Listing 1).

The majority of existing tools employed by artists to author normal maps use a standard known as *mikkTSpace* (see [Golus 2020]) to generate the matrix M per vertex, as given by Equation (8). The motivation for this is to ensure that the same transformation M is used when generating a normal map and when rendering with it. For more information, the reader is referred to [Mikkelsen 2011].

The mikkTSpace standard dictates that the interpolated but *unnormalized* vertex attributes for $\vec{t}$, $\vec{b}$, and $\vec{n}$ must be used in the shader to match the normal map. However, for the surface gradient–based formulation, the interpolated vertex normal must be *normalized*. We can address this conflict by reusing the same trick as before, which is that a positive uniform scale of the perturbed normal cancels out in the final resolve, so we can multiply all three vectors by the reciprocal magnitude of the unnormalized but interpolated vertex normal:

$$\vec{t} \leftarrow \frac{1}{\|\vec{n}\|} \cdot \vec{t}, \quad \vec{b} \leftarrow \frac{1}{\|\vec{n}\|} \cdot \vec{b}, \quad \vec{n} \leftarrow \frac{1}{\|\vec{n}\|} \cdot \vec{n}.$$

This allows us to use a surface gradient–based formulation that is mikkTSpace compliant, where $\vec{n}$ is normalized and $\vec{t}$ and $\vec{b}$ have been scaled accordingly. We perform this adjustment to the basis `mikktsTangent` and `mikktsBitangent` in Listing 9. The final TBN-style surface gradient is summarized in Listing 3 and can be used with the adjusted basis vectors as parameters for `vT` and `vB`. The first parameter, `deriv`, corresponds to $(\frac{\partial H}{\partial u}, \frac{\partial H}{\partial v})$, as described in Section 3.1.

Though mikkTSpace compliance requires vertex attributes for $\vec{t}$ and $\vec{b}$, game engines typically only provide one tangent space per vertex, which does not account for multiple sets of texture coordinates or even procedural texture coordinates. We address this limitation in the next section.

```
float3 SurfgradFromTBN(float2 deriv, float3 vT, float3 vB)
{
    return deriv.x*vT + deriv.y*vB;
}
```

Listing 3. TBN-style surface gradient.

3.3. TBN Basis without Vertex-Level Tangent Space

For a scenario in which the tangent $\vec{t}$ and bitangent $\vec{b}$ do not exist at the vertex level for a given set of texture coordinates, the vectors must be generated on the fly in the pixel shader. The following function depends on `sigmaX`, `sigmaY`, `flip_sign`, and the initial unit-length normal `nrmBaseNormal`, which are cached and initialized at the beginning of the pixel shader, as described in Section 3.7. The values `sigmaX` and `sigmaY` represent the first-order derivatives of the surface position with respect to screen-space `dPdx` and `dPdy` but perpendicular to `nrmBaseNormal`. By using the chain rule, we obtain the tangent as $\vec{t} = \texttt{sigmaX} \cdot \texttt{dXds} + \texttt{sigmaY} \cdot \texttt{dYds}$.

```
void GenBasisTB(out float3 vT, out float3 vB, float2 texST)
{
    float2 dSTdx = ddx_fine(texST);
    float2 dSTdy = ddy_fine(texST);
    float det = dot(dSTdx, float2(dSTdy.y, -dSTdy.x));
    float sign_det = det < 0.0 ? -1.0 : 1.0;

    // invC0 represents (dXds, dYds), but we don't divide
    // by the determinant. Instead, we scale by the sign.
    float2 invC0 = sign_det*float2(dSTdy.y, -dSTdx.y);
    vT = sigmaX*invC0.x + sigmaY*invC0.y;
    if (abs(det) > 0.0) vT = normalize(vT);
    vB = (sign_det*flip_sign)*cross(nrmBaseNormal, vT);
}
```

Listing 4. Procedural TBN basis.

The function in Listing 4 delivers the tangent frame as an orthonormal set that may be used as parameters to the function `SurfgradFromTBN()` in Listing 3. Furthermore, the function works regardless of whether the texture coordinates are defined clockwise or counterclockwise.

3.4. Surface Gradient from Object- or World-Space Normal

In some cases, it is convenient for an artist to use an object- or world-space normal map that represents the normal after it has already been perturbed. However, we want to integrate these into the surface gradient–based framework so that we can adjust the bump scale and combine them with other bump mapping contributions, such as

tangent-space normal maps, decals, triplanar projection, etc. In other words, we need a method to convert the object- or world-space normal to a surface gradient.

We receive as input the sampled object- or world-space normal $\vec{v}$ of arbitrary positive magnitude, and the initial normal $\vec{n}$ is also known. The vector $\vec{v}$ represents the direction of $\vec{n}'$ but not its magnitude. Thus, there exists some factor $k \in \mathbb{R}_{\neq 0}$ such that $\vec{v} = k \cdot \vec{n}'$. We can solve for $\nabla_S H$ by reordering terms in Equation (5) and substituting $\vec{n}'$ with $\frac{1}{k} \cdot \vec{v}$, where the factor k is determined using the following:

$$\begin{aligned} \vec{n} \bullet \vec{v} &= k \cdot \left(\vec{n} \bullet \vec{n}'\right) \\ &= k \cdot \left((\vec{n} \bullet \vec{n}) - (\vec{n} \bullet \nabla_S H)\right) \\ &= k \cdot (1 - 0) \\ &= k. \end{aligned}$$

The final utility function is given in Listing 5.

```
// Surface gradient from a known "normal" such as from an object-
// space normal map.  This allows us to mix the contribution with
// others, including from tangent-space normals. The vector v
// doesn't need to be unit length so long as it establishes
// the direction. It must also be in the same space as the normal.
float3 SurfgradFromPerturbedNormal(float3 v)
{
    // If k is negative then we have an inward facing vector v,
    // so we negate the surface gradient to perturb toward v.
    float3 n = nrmBaseNormal;
    float k = dot(n, v);
    return (k*n - v)/max(FLT_EPSILON, abs(k));
}
```

Listing 5. Run-time conversion of object- or world-space normal to surface gradient.

3.5. Scale-Dependent Bump Mapping

It is important to distinguish between *scale-dependent* and *TBN-style* bump mapping. With the latter, the intensity of the bump effect does not depend on the tiling rate of the normal map across the surface. The reason for this is a combination of tradition, artist workflow convenience, and the fact that it reduces the memory footprint of the tangent frame by assuming it represents an orthonormal basis at the vertex level.

When applying displacement-style bump mapping, the perturbed normal is meant to represent the new vertex normal of the actual displacement-mapped mesh. In this case, we cannot ignore the tiling rate, nor can we assume that the bitangent is perpendicular to the tangent.

The surface gradient for this scenario is summarized in Listing 6 using Equation (7). The values for `dPdx`, `dPdy`, and `nrmBaseNormal` are cached in the prologue given in Listing 9 and must all be in the same frame of reference. An important

subtlety to note is that `dPdx` and `dPdy` must be the screen-space derivatives of the initial surface position *before displacement*. The only exception to this is when performing a separate bump mapping pass on the new surface after displacement mapping, in which case `nrmBaseNormal` would also need to be replaced with the normal of the displaced surface.

When `isDerivScreenSpace` is set to `true`, `deriv` represents the screen-space derivatives of the height: `dHdx` and `dHdy`. A crude single-sample approach for calculating these in the pixel shader is to use `ddx_fine(height)` and `ddy_fine(height)` directly. However, since this is based on numerical differencing across a block of 2×2 pixels, better quality is achieved by using the three-tap approach provided in Listing 7.

```
// dim: resolution of the texture deriv was sampled from.
// isDerivScreenSpace: true if deriv is already in screen space.
float3 SurfgradScaleDependent(float2 deriv, float2 texST, uint2 dim,
                              bool isDerivScreenSpace = false)
{
    // Convert derivative to normalized (s,t) space.
    const float2 dHdST = dim*deriv;

    // Convert derivative to screen space by applying
    // the chain rule. dHdx and dHdy correspond to
    // ddx_fine(height) and ddy_fine(height).
    float2 texDx = ddx(texST);
    float2 texDy = ddy(texST);
    float dHdx = dHdST.x*texDx.x + dHdST.y*texDx.y;
    float dHdy = dHdST.x*texDy.x + dHdST.y*texDy.y;

    if (isDerivScreenSpace)
    {
        dHdx = deriv.x;
        dHdy = deriv.y;
    }

    // Equation 3 in [Mikkelsen 2010].
    float3 vR1 = cross(dPdy, nrmBaseNormal);
    float3 vR2 = cross(nrmBaseNormal, dPdx);
    float det = dot(dPdx, vR1);

    const float eps = 1.192093e-15F;
    float sign_det = det < 0.0 ? -1.0 : 1.0;
    float s = sign_det/max(eps, abs(det));

    return s*(dHdx*vR1 + dHdy*vR2);
}
```

Listing 6. Surface gradient of scale-dependent bump mapping.

```
    // 3 taps:  better quality than float2(ddx_fine(H), ddy_fine(H)).
    float2 dSTdx = ddx(st), dSTdy = ddy(st);
    float Hll = hmap.Sample(sampler, st).x;
    float Hlr = hmap.Sample(sampler, st + dSTdx).x;
    float Hul = hmap.Sample(sampler, st + dSTdy).x;

    float2 deriv = float2(Hlr - Hll, Hul - Hll);
```

Listing 7. Forward differencing: three-tap screen-space derivative of the height.

3.6. Volume Bump Maps

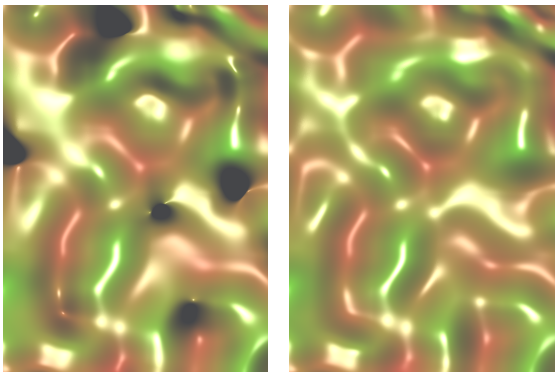
The concept of a *space function* was introduced by Perlin [1985] and is defined as a function H that varies over a three-dimensional domain $H : (x, y, z) \rightarrow \mathbb{R}$. We can assign colors to a surface embedded in this volume by sampling the space function using the point p on the surface associated with the pixel being shaded. Perlin refers to this as a *solid texture*. The same approach can be used to assign displacement values to a surface when using a scalar space function H. In the following, we will refer to this as a *volume bump map*.

It is suggested in [Perlin 1985] that a normal perturbing effect may be achieved by adding the volume gradient $\nabla H = (\frac{\partial H}{\partial x}, \frac{\partial H}{\partial y}, \frac{\partial H}{\partial z})$ to the initial normal $\vec{n}$ and renormalizing. This does not produce the correct normal, as pointed out by Worley in [Ebert et al. 2003], and instead the correct solution is described as using the traditional bump mapping method of Blinn [1978]. However, this requires a known surface parameterization and known partial derivatives of the surface position, and generally requires storing the sampled displacements in a 2D texture, performing a precomputation step to evaluate the derivatives of H with respect to u and v, and generating MIP maps, which is presumably why this approach is not used in [Perlin 1985] nor in [Ebert et al. 2003].

A significantly simpler way to achieve the correct result is to leverage the discovery made by Mikkelsen [2010] that Blinn's perturbed normal has a surface gradient–based formulation, as given by Equation (5) and also provided as a shader function in Listing 1. We can produce the surface gradient $\nabla_S H$ from ∇H and the initial normal $\vec{n}$ using Equation (6), which is summarized as a function in Listing 8.

```
// Used to produce a surface gradient from the gradient of a volume
// bump function such as 3D Perlin noise. Equation 2 in [Mik10].
float3 SurfgradFromVolumeGradient(float3 grad)
{
    return grad - dot(nrmBaseNormal, grad)*nrmBaseNormal;
}
```

Listing 8. Surface gradient produced from volume gradient.



(a) Perlin's perturbation (b) Surface gradient–based

Figure 4. (a) Perlin-based normal perturbation can produce visual artifacts. (b) These are avoided by using our method.

The difference between Perlin's method in [Perlin 1985] and the correct approach is visually significant. Figure 4 provides a comparison using Perlin noise–based bump mapping (see [Perlin 2002]), where the elevation level is indicated in green and reddish-brown. As a correction, ∇H is negated in Figure 4(a) to invert the displacement direction.

If ∇H is above the tangent plane, the intensity of the bump effect becomes understated when using Perlin's method. If the gradient is below the tangent plane, the perturbation impact is increased until it finally pulls the normal inward, resulting in the black spots we see in Figure 4(a). Using the surface gradient–based method, we get the artifact-free result shown in Figure 4(b).

It is worth emphasizing that resampling the displacements, storing them in a 2D texture map, and using traditional bump mapping is mathematically equivalent to using the method in Listing 8 and resolving using Listing 1.

3.7. Pixel Shader Prologue of Framework

The code snippet in Listing 9 calculates various parameters that are used by many of the other listings. Thus, to use the framework functions without further modification, it is necessary to retrofit this prologue into your own shader.

In a scenario where one resolve has already been performed using Listing 1, it is possible to perform a second pass of bump mapping on top. We refer to this as *post-resolve bump mapping*, and in this case the values `nrmBaseNormal`, `dPdx`, `dPdy`, `sigmaX`, `sigmaY`, and `flip_sign` must be updated. However, in some cases, `ddx_fine()` and `ddy_fine()` will not produce useful values because there is no longer a correspondence between the pixels in each 2×2 block. An example of this is *parallax occlusion mapping* (POM) (see Section 4.3), since different intersection points may be found during the ray-marching step at each of the four pixels.

Deferred rendering is another common case that exhibits divergence, at regional discontinuities such as silhouette boundaries. In such cases, we can calculate `dPdx` and `dPdy` analytically from the position and normal, as shown in Listing 10. Using this method, `dPdx` and `dPdy` will already be perpendicular to `nrmBaseNormal`, so they will be identical to `sigmaX` and `sigmaY`, respectively.

One issue with using analytical `dPdx` and `dPdy` values is that the interpolated vertex normal $\vec{n}$ may be backfacing with respect to the view vector $\vec{v}$. When this is the case, a workaround is to correct the normal in Listing 10 by reflecting it toward the observer as $\vec{n} - 2 \cdot (\vec{v} \bullet \vec{n}) \cdot \vec{v}$.

```
// Cached bump globals, reusable for all UVs including procedural.
static float3 sigmaX, sigmaY, nrmBaseNormal, dPdx, dPdy;
static float  flip_sign;
// Used for vertex-level tangent space (one UV set only).
static float3 mikktsTangent, mikktsBitangent;

float4 PixelShader(VertexOutput In) : SV_Target
{
    // NO TRANSLATION! Just 3x3 transform to avoid precision issues.
    float3 relSurfPos = ToRelativeWorldSpace(In.surfPos.xyz);

    // mikkts for conventional vertex-level tangent space
    // (no normalization is mandatory). Using "bitangent on the fly"
    // option in xnormal to reduce vertex shader outputs.
    float sign_w = In.tangent.w > 0.0 ? 1.0 : -1.0;
    mikktsTangent = In.tangent.xyz;
    mikktsBitangent = sign_w*cross(In.normal.xyz, In.tangent.xyz);

    // Prepare for surfgrad formulation w/o breaking mikkTSpace
    // compliance (use same scale as interpolated vertex normal).
    float renormFactor = 1.0/length(In.normal.xyz);
    mikktsTangent   *= renormFactor;
    mikktsBitangent *= renormFactor;
    nrmBaseNormal    = renormFactor*In.normal.xyz;
    // Handle two-sided materials. Surface gradients, tangent, and
    // bitangent don't flip as a result of flipping the base normal.
    if (IsFlipNormal()) nrmBaseNormal = -nrmBaseNormal;

    // The variables below (plus nrmBaseNormal) need to be
    // recomputed in the case of post-resolve bump mapping.
    dPdx = ddx_fine(relSurfPos);
    dPdy = ddy_fine(relSurfPos);
    sigmaX = dPdx - dot(dPdx, nrmBaseNormal)*nrmBaseNormal;
    sigmaY = dPdy - dot(dPdy, nrmBaseNormal)*nrmBaseNormal;
    flip_sign = dot(dPdy, cross(nrmBaseNormal, dPdx)) < 0 ? -1 : 1;
```

Listing 9. Bump mapping prologue to initialize reusable parameters.

```
// Calculate ddx(pos) and ddy(pos) analytically.
// Practical when ddx and ddy are unavailable.
// pos: surface position of the pixel being shaded.
// norm: represents nrmBaseNormal of the position being shaded.
// mInvViewProjScr: transformation from the screen
// [0;width] x [0;height] x [0;1] to the space pos and norm are in.
// x0, y0: the current fragment coordinates (pixel center at .5).
void ScreenDerivOfPosNoDDXY(out float3 dPdx, out float3 dPdy,
                            float3 pos, float3 norm,
                            float4x4 mInvViewProjScr,
                            float x0, float y0)
{
    float4 plane = float4(norm.xyz, -dot(pos, norm));
    float4x4 mInvViewProjScrT = transpose(mInvViewProjScr);
    float4 planeScrSpace = mul(mInvViewProjScrT, plane);

    // Ax + By + Cz + D = 0 --> z = -(A/C)x - (B/C)y - D/C.
    // Intersection point at (x, y, -(A/C)x - (B/C)y - D/C, 1).
    const float sign_z = planeScrSpace.z < 0.0 ? -1.0 : 1.0;
    const float nz = sign_z*max(FLT_EPSILON, abs(planeScrSpace.z));

    const float ac = -planeScrSpace.x/nz;
    const float bc = -planeScrSpace.y/nz;
    const float dc = -planeScrSpace.w/nz;

    float4 C2 = mInvViewProjScrT[2];
    float4 v0 = mInvViewProjScrT[0] + ac*C2;
    float4 v1 = mInvViewProjScrT[1] + bc*C2;
    float4 v2 = mInvViewProjScrT[3] + dc*C2;

    // 4D intersection point in world space.
    float4 ipw = v0*x0 + v1*y0 + v2;

    // Use derivative of f/g --> (f'*g - f*g')/g^2.
    float denom = max(FLT_EPSILON, ipw.w*ipw.w);
    dPdx = (v0.xyz*ipw.w - ipw.xyz*v0.w)/denom;
    dPdy = (v1.xyz*ipw.w - ipw.xyz*v1.w)/denom;

    // If mInvViewProjScr is in normalized screen space [-1;1]^2,
    // then scale dPdx and dPdy by 2/width and 2/height,
    // respectively. Also, negate dPdy if there's a Y-axis flip.
    // dPdx *= 2.0/width; dPdy *= 2.0/height;
}
```

Listing 10. Calculation of `dPdx` and `dPdy` when `ddx` and `ddy` are unavailable.

4. Applications and Testing

In this section, we cover some examples of practical use cases for the surface gradient–based framework and compare with existing traditional methods where applicable.

4.1. Multiple Sets of Texture Coordinates

Reusing generic detail in the form of tileable 2D textures on different surfaces requires an unwrap of each mesh—as texture coordinates—with careful seam placement, orientation, and scale. However, with the emergence of modern workflows such as photogrammetry, projecting detail from a high-resolution model to a low-resolution model, or painting in 3D directly onto the surface, the authored texture is uniquely tied to the mesh. In this case, the unwrap is tailored toward utilizing as much of the texture resolution as possible rather than supporting reuse. In the following, we will refer to these as the *tileable unwrap* and the *unique unwrap*, respectively.

The ability to combine multiple layers on a surface allows us to mimic complex and expansive surface detail that would otherwise require excessive texture resolution and become impractical to author from an artist workflow standpoint. In some cases, 3D models may even leverage a third and fourth unwrap, yet for normal mapping game engines only provide one tangent space per vertex, which is specific to the first unwrap of the model. For this reason, artists are traditionally required to keep all normal maps on the mesh assigned to the same unwrap, which is far from ideal.

An example of this is shown in Figure 1 as a low-resolution model of a gnome. As a preprocess, the surface normals of a high-resolution model of the gnome are projected onto the low-resolution version and stored in a normal map using the unique unwrap. In the top row, starting from the left, we first see the gnome in wireframe, then lit using the aforementioned normal map. Continuing to the right we see generic tileable detail added on top using the unique and tileable unwraps, respectively. In the case of the latter, we used Listing 4 to produce a tangent space. In the bottom row are the corresponding close-ups. When using the unique unwrap for both contributions, we get an implausible result where the generic details maintain the same orientation on the shoulder strap as on the shirt. In contrast, using the tileable unwrap allows the details to align with the shoulder strap itself.

4.2. Alternative Methods for Accumulating Bump Maps

Over the years, different methods to combine normal maps have surfaced, and several are described by Barré-Brisebois and Hill in their survey [2012]. In the following, we will highlight some of the most common methods described in their survey and correlate these to the framework we are proposing here. These methods are *partial derivatives* (PD), *whiteout* (WO), *Unreal Developer Network* (UDN), and *reoriented normal mapping* (RNM).

The problem is defined thus: given two tangent-space normals $\vec{a}, \vec{b} \in [-1; 1]^3$, how do we combine these in a way that gives the impression of stacking up the bump maps but gives the desired aesthetic appearance? Given this somewhat fuzzy goal, it is unsurprising that two of these methods, WO and UDN, are empirical. In the survey, they are described as the result of normalizing $\begin{bmatrix} \vec{a}_x + \vec{b}_x & \vec{a}_y + \vec{b}_y & \vec{a}_z \cdot \vec{b}_z \end{bmatrix}$ and $\begin{bmatrix} \vec{a}_x + \vec{b}_x & \vec{a}_y + \vec{b}_y & \vec{a}_z \end{bmatrix}$, respectively. In contrast, the PD approach simply adds the corresponding partial derivatives for $\vec{a}$ and $\vec{b}$, as given by Equation (15).

An important observation to make is that PD, WO, and UDN, as described, assume that both normal maps are assigned to the same unwrap. Now, using our framework, we can easily extend all three to work across multiple unwraps.

Compositing Method	First Tangent Normal	Second Tangent Normal
PD	$\vec{a}$	$\vec{b}$
WO	$\begin{bmatrix} \vec{a}_x & \vec{a}_y & \vec{a}_z \cdot \vec{b}_z \end{bmatrix}$	$\begin{bmatrix} \vec{b}_x & \vec{b}_y & \vec{a}_z \cdot \vec{b}_z \end{bmatrix}$
UDN	$\vec{a}$	$\begin{bmatrix} \vec{b}_x & \vec{b}_y & \vec{a}_z \end{bmatrix}$

Table 1. Surface gradient–based variants of surveyed methods for PD, WO and UDN.

In any of the three cases, we convert both the first and second tangent-space normals in Table 1 into a derivative using Listing 2 and then proceed to produce a surface gradient from each of them using any of the applicable methods described in Section 3. Finally, we add the surface gradients together and resolve using Listing 1.

The final method we will cover from the survey is RNM, given here by Equation (18), as derived in [Barré-Brisebois and Hill 2012]:

$$\vec{m} = \frac{1}{\vec{a}_z + 1} \begin{bmatrix} \vec{a}_x \\ \vec{a}_y \\ \vec{a}_z + 1 \end{bmatrix} \cdot \left(\begin{bmatrix} \vec{a}_x & \vec{a}_y & \vec{a}_z + 1 \end{bmatrix} \cdot \begin{bmatrix} -\vec{b}_x \\ -\vec{b}_y \\ \vec{b}_z \end{bmatrix} \right) - \begin{bmatrix} -\vec{b}_x \\ -\vec{b}_y \\ \vec{b}_z \end{bmatrix}. \quad (18)$$

This method corresponds directly to post-resolve bump mapping described in Section 4.6. In this case, $\vec{a}$ represents the first perturbation step and $\vec{b}$ represents the second. However, unlike our post-resolve approach, the RNM method does not provide a solution that supports multiple unwraps, object-space normal maps, or volume bump maps. Despite these limitations, it is very efficient, thus it remains useful in the context of post-resolve bump mapping when using two tangent-space normal maps that are assigned to the same unwrap. In this case, we can convert the result from RNM into a surface gradient, allowing us to combine the bump contribution with other forms of bump mapping within our framework.

4.3. Parallax Mapping

Techniques that are strongly related to displacement mapping (described in Section 3.5) are *parallax mapping* [Kaneko et al. 2001; Welsh 2004] and *parallax occlusion map-*

ping (POM) [Brawley and Tatarchuk 2005]. Just as with displacement mapping, these techniques are sensitive to the tiling rate, so using a TBN-style frame of reference is incorrect.

A detailed explanation of parallax mapping and POM is beyond the scope of this paper. But, in short, you transform the view vector into texture space and search for the first intersection with the height map along the ray, starting at the current pixel.

The function given in Listing 11 transforms the unit-length input direction `dir` and establishes the 2D line segment in texture space as a vector $\vec{v}$ along which we must search for the intersection between the ray and the height field.

When `skipProj` is `false`, the search range is extended $(\vec{v}_x, \vec{v}_y, \vec{v}_z) \rightarrow (\frac{\vec{v}_x}{\vec{v}_z}, \frac{\vec{v}_y}{\vec{v}_z}, 1)$ such that the third component is always one unit along `nrmBaseNormal`. This projection is not used with the more basic approach in [Welsh 2004] (as explained in their Section 4.3), so in this case `skipProj` should be set to `true`. A correction is applied at the end of Listing 11, where we scale by the user parameter `bumpScale` to account for this factor in the displacement vector `bumpScale*H(s,t)*nrmBaseNormal`. This implies that a height level of `1.0` corresponds to the displacement distance `bumpScale` along the normal, in world space.

```
// dir: normalized vector in same space as surface pos and normal.
// bumpScale: p' = p + bumpScale*DisplacementMap(s,t)*normal.
float2 ProjectVecToTextureSpace(float3 dir, float2 texST,
    float bumpScale, bool skipProj = false)
{
    float2 texDx = ddx(texST);
    float2 texDy = ddy(texST);
    float3 vR1 = cross(dPdy, nrmBaseNormal);
    float3 vR2 = cross(nrmBaseNormal, dPdx);
    float  det = dot(dPdx, vR1);

    const float eps = 1.192093e-15F;
    float sgn = det < 0.0 ? -1.0 : 1.0;
    float s = sgn/max(eps, abs(det));

    float2 dirScr  = s*float2(dot(vR1, dir), dot(vR2, dir));
    float2 dirTex  = texDx*dirScr.x + texDy*dirScr.y;
    float  dirTexZ = dot(nrmBaseNormal, dir);

    // To search heights in [0;1] range use dirTex.xy/dirTexZ.
    s = skipProj ? 1.0 : 1.0/max(FLT_EPSILON, abs(dirTexZ));
    return s*bumpScale*dirTex;
}
```

Listing 11. Transformation of project vector to texture space for parallax mapping.

A useful feature, provided by the function in Listing 12, is the ability to evaluate the corrected position at the surface, which corresponds to the corrected offset for

the texture coordinates. From the initial surface position, the offset in the tangent plane, and the height value at the corrected location, it is trivial to evaluate the new displaced surface position. It is worth noting that the same thing cannot be done using a traditional vertex-level tangent space, since we need to take the ratio between world-space units and texture-space units into account for a proper conversion.

When the initial surface is not flat, a similar correction should ideally be applied to the normal `nrmBaseNormal`, where the correction vector is $\texttt{vx}\cdot\texttt{dNdx}+\texttt{vy}\cdot\texttt{dNdy}$. The values `vx` and `vy` are as given in Listing 12, and `dNdx` and `dNdy` represent `ddx()` and `ddy()` on `nrmBaseNormal`.

```
// initialST: initial texture coordinate before parallax correction.
// corrOffs: parallax-corrected offset from initialST.
float3 TexSpaceOffsToSurface(float2 initialST, float2 corrOffs)
{
    float2 texDx = ddx(initialST);
    float2 texDy = ddy(initialST);
    float det = texDx.x*texDy.y - texDx.y*texDy.x;

    const float eps = 1.192093e-15F;
    float sgn = det < 0.0 ? -1.0 : 1.0;
    float s = sgn/max(eps, abs(det));

    // Transform corrOffs from texture space to screen space.
    // Use 2x2 inverse of [ddx(initialST) | ddy(initialST)].
    float vx = s*( texDy.y*corrOffs.x - texDy.x*corrOffs.y);
    float vy = s*(-texDx.y*corrOffs.x + texDx.x*corrOffs.y);

    // Transform screen-space vector to the surface.
    return vx*sigmaX + vy*sigmaY;
}
```

Listing 12. Transformation of the POM chosen correction vector from texture space to the surface.

Once the correction to the texture coordinates has been performed by adding the offset vector, the POM or parallax mapping process proceeds by sampling all remaining textures of the material (with the same unwrap) using these new coordinates.

A notable problem is the way in which the GPU determines the fractional MIP level to sample, which is based on the difference in texture coordinates between adjacent pixels on the screen within a block of 2×2 pixels.[1] A better result is achieved by using the HLSL intrinsic `CalculateLevelOfDetail()` with the texture coordinates *before correction*, where this chosen level of detail is used together with the texture coordinates *after correction* to perform subsequent texture sampling with `SampleLevel()`.

[1] The problem was also discussed in Section 3.7 in the context of `ddx_fine()` and `ddy_fine()`.

4.4. Triplanar Bump Mapping

Triplanar projection [Geiss 2007a] involves parallel-projecting three images H_x, H_y, and H_z $(u, v) \rightarrow \mathbb{R}$ onto the rendered surface, where each image is assigned to a separate axis $\vec{x}$, $\vec{y}$, or $\vec{z}$. The parameter (u, v) represents unnormalized coordinates such that one pixel represents one unit. For triplanar bump mapping with a left-hand coordinate system, the final height map is summarized as follows:

$$H(x, y, z) = w_x \cdot H_x(z, y) + w_y \cdot H_y(x, z) + w_z \cdot H_z(x, y), \tag{19}$$

where $w(\vec{n}) = (w_x, w_y, w_z)$ is a smooth function of the surface normal $\vec{n}$ and is used to weight the contributions from each of the three projections. As noted earlier in Section 3, we do not need the actual displacement maps for images H_x, H_y, and H_z in practice.

Note that in Equation (19) we have implicitly assumed that $(0, 0)$ represents the lower-left corner of a height map. A change to the upper-left corner can be achieved by inverting the second sampling coordinate for each height map: $H(s, 1 - t)$, in normalized coordinates. But in this case, we must also negate the second coordinate of each of the three derivatives that we receive in Listing 13. Similarly, a change to a right-hand coordinate frame is straightforward. In this case, $H_x(-z, y)$ and $H_y(x, -z)$ are sampled using negated coordinates for z, which means that we must negate `deriv_xplane.x` and `deriv_yplane.y` in Listing 13, which in practice is achieved by negating the third component of `grad`.

To use triplanar bump mapping within our framework, we need to establish the surface gradient $\nabla_S H$ for Equation (19). At this point we should acknowledge that $w(\vec{n})$ depends on the initial normal $\vec{n}$, which means that the derivative of w depends on the derivative of $\vec{n}$. However, similar to [Blinn 1978], we assume that the derivative of $\vec{n}$ is sufficiently close to zero. For a deeper analysis of this approximation, the reader is referred to [Mikkelsen 2008]. Thus, we can proceed under the assumption that $w(\vec{n}) = (w_x, w_y, w_z)$ is sufficiently constant at each point p being shaded. Although Equation (19) does not represent a valid volume bump map at a global level due to its dependency on $\vec{n}$—which in turn depends on the surface—we can think of it as a tiny local volume bump map containing p on the surface as we shade it with a fixed w. The gradient on this volume is then given as

$$\nabla H \simeq \left(w_z \cdot \frac{\partial H_z}{\partial u} + w_y \cdot \frac{\partial H_y}{\partial u}, w_z \cdot \frac{\partial H_z}{\partial v} + w_x \cdot \frac{\partial H_x}{\partial v}, w_x \cdot \frac{\partial H_x}{\partial u} + w_y \cdot \frac{\partial H_y}{\partial v} \right).$$

Finally, we can determine the surface gradient $\nabla_S H$ using Equation (6).

It is worth noting that an alternative solution is to establish the surface gradient for each two-dimensional mapped image separately, using Equation (7), and then accumulate the three surface gradients as a weighted average using the coordinates of w as weights. However, this results in exactly the same surface gradient $\nabla_S H$.

The complete implementation of triplanar bump mapping is given in Listing 13, and the implementation for establishing weights is given in Listing 14. Note that triplanar projection can also be performed on a surface that has already been bump mapped. In this case, we simply replace `nrmBaseNormal` with the perturbed normal in Listings 1, 8, and 14.

```
// Triplanar projection is considered a special case of volume
// bump map. Weights are obtained using DetermineTriplanarWeights()
// and derivatives using TspaceNormalToDerivative().
float3 SurfgradFromTriplanarProjection(float3 triplanarWeights,
    float2 deriv_xplane, float2 deriv_yplane, float2 deriv_zplane)
{
    const float w0 = triplanarWeights.x;
    const float w1 = triplanarWeights.y;
    const float w2 = triplanarWeights.z;

    // Assume deriv_xplane, deriv_yplane, and deriv_zplane are
    // sampled using (z,y), (x,z), and (x,y), respectively.
    // Positive scales of the lookup coordinate will work
    // as well, but for negative scales the derivative components
    // will need to be negated accordingly.
    float3 grad = float3(w2*deriv_zplane.x + w1*deriv_yplane.x,
                         w2*deriv_zplane.y + w0*deriv_xplane.y,
                         w0*deriv_xplane.x + w1*deriv_yplane.y);

    return SurfgradFromVolumeGradient(grad);
}
```

Listing 13. Triplanar normal mapping using a surface gradient–based formulation.

```
// Adapted from
// http://www.slideshare.net/icastano/cascades-demo-secrets.
float3 DetermineTriplanarWeights(float k = 3.0)
{
    float3 weights = abs(nrmBaseNormal) - 0.2;
    weights = max(0, weights);
    weights = pow(weights, k);
    weights /= (weights.x + weights.y + weights.z);
    return weights;
}
```

Listing 14. Determination of blend weights for triplanar mapping.

The traditional implementation for triplanar projection by Geiss [2007b] is similar to the one we give here, but there are two key differences. First of all, the traditional method uses the first two components of the tangent-space normal directly in the implementation instead of the corresponding derivatives. Second, Geiss uses Perlin's method to perturb the surface normal, as described in Section 3.6.

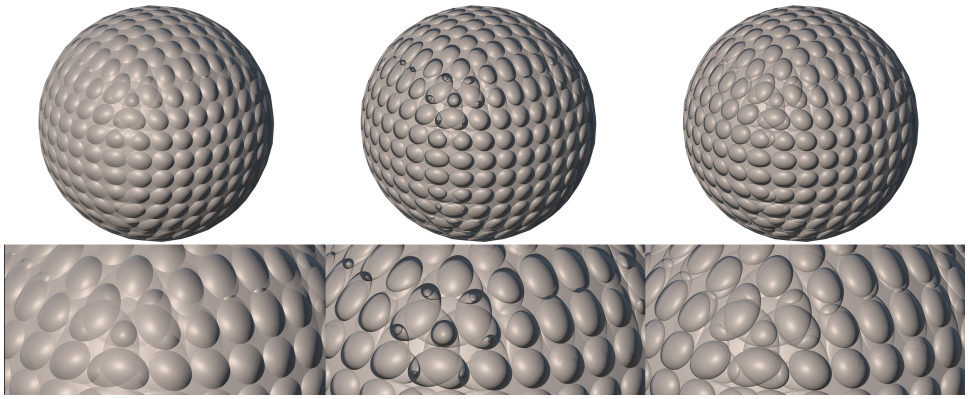


Figure 5. Left: The result of using the traditional method for triplanar projection, which flattens the intended details. Middle: The result of using a more correct derivative-based approach but still using Perlin's perturbation method, which gives artifacts at the crossover between planar projections. Right: Our method. The corresponding close-ups are shown in the bottom row.

Figure 5 shows an example of triplanar projection, which is used to add detail from a tiling hemisphere normal map. The traditional method of Geiss is shown on the left, and as we can see, the hemispheres become flattened as a result of omitting the conversion to a derivative, which gives us a peak slope of 45° rather than a near-vertical limit.

In the middle image, we apply the conversion to a derivative but still use Perlin's method to perturb the normal. This results in the same type of artifact described in Section 3.6 and shown in Figure 4(a). The artifact appears mainly near the crossover between the planar projections, since this is where the volume gradient will stray the most from the tangent plane of the surface. Finally, the image on the right shows the correct result, which we get by using the surface gradient–based method described in this section.

4.5. Decal Projector

Similar to triplanar projection, the *decal projector* is a special case that is ideally processed as a volume bump map. It is defined as parallel-projecting an image $H : (u, v) \to \mathbb{R}$ onto the rendered surface but relative to a user-specified orthonormal frame of reference $\vec{x}$, $\vec{y}$, and $\vec{z}$. Within this local frame of reference, H is sampled using (x, y) as the sampling coordinates, which means that the image is conceptually repeated along the $\vec{z}$-axis, so the volume gradient transformed into a common frame of reference is equal to

$$\nabla H = \frac{\partial H}{\partial u} \cdot \vec{x} + \frac{\partial H}{\partial v} \cdot \vec{y}.$$

Once again, we can readily determine the surface gradient $\nabla_S H$ using Equation (6), leading to the simple implementation provided in Listing 15.

```
// axisX, axisY: X- and Y-axes of decal projector. Same space as
// normal.
float3 SurfgradFromDecal(float2 deriv, float3 axisX, float3 axisY)
{
    // Transform volume gradient.
    float3 grad = deriv.x*axisX + deriv.y*axisY;

    // Produce a surface gradient.
    return SurfgradFromVolumeGradient(grad);
}
```

Listing 15. Decal projector as volume bump map.

As in Section 4.4, we have assumed that $(0, 0)$ represents the lower-left corner of the bump map. Like before, in order to convert to the upper-left corner, we would need to invert the second sampling coordinate and negate `deriv.y` in Listing 15.

At this point, an important observation to make is that, just as the surface gradient is a linear operator, the same is true for the volume gradient. Thus, we do not have to perform the projection to the surface, `SurfgradFromVolumeGradient()`, per decal. Instead, we can accumulate the volume gradient of each decal one by one and only project onto the surface at the very end to obtain the surface gradient. This is particularly useful in a scenario where decals are processed prior to having access to the initial normal $\vec{n}$, such as before running a G-buffer pass.[2]

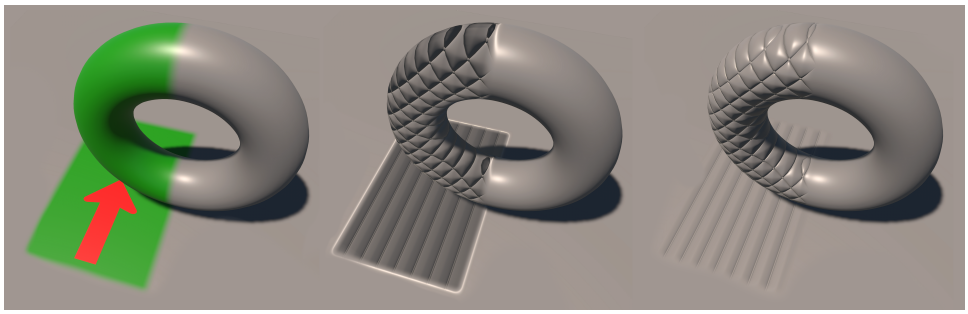


Figure 6. Left: A decal placed such that the ground plane and a torus intersect with it, as illustrated by the green region, and where the direction of projection is shown with a red arrow. Middle: The result of using the negated direction of projection as the normal to be perturbed. Right: The improved result achieved when using a surface gradient–based approach to perturb the normal of the embedded surface.

[2]The same feature allows us to modulate by the spatial falloff associated with the projector volume as a bump scale applied to either the surface gradient or the volume gradient.

Today's game engines typically apply decal projectors by using the negated projector direction as the normal to be perturbed, then later blend the result with the normal of the embedded surface to which the decal is applied. The problem with this method is that it leads to inferior quality when applied to a curved surface or when the projector direction is misaligned with the normal of the embedded surface, as shown in the middle of Figure 6. When we use the surface gradient–based approach described in this section, we are able to perturb the surface normal of the actual surface, which gives the improved result seen on the right.

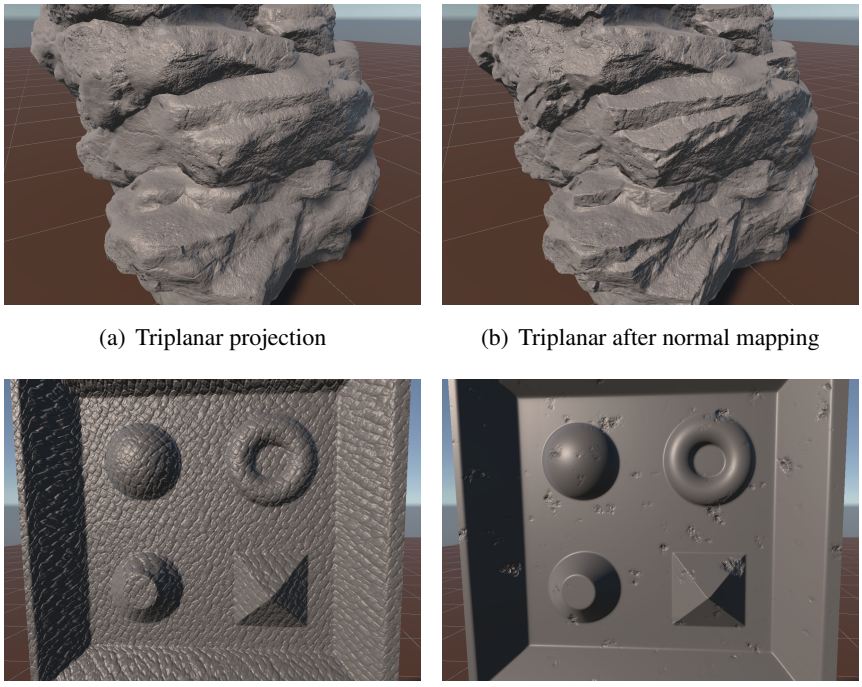
4.6. Post-Resolve Bump Mapping

In Section 3.7, we briefly discussed using bump mapping *back-to-back*, where we perturb the surface normal of the virtual surface that we obtained from a previous iteration of bump mapping. This process, which we refer to as *post-resolve bump mapping*, is particularly useful for adding fine detail to a surface, where macro-scale detail is first established in the initial iteration.

It is common in games to see examples of triplanar projection used to texture large-scale geometry such as terrain. However, for more complex shapes such as the rock shown in Figure 7(a), this approach does not produce convincing results, since the interpolated vertex normal is not a good representation of the tangent plane of the surface. Fortunately, using our framework, we can achieve significantly improved results by using post-resolve bump mapping. The initial step is to use a conventional normal map to establish the macro-scale normal of the surface, followed by a resolve using Listing 1. Once this resolve has been performed, we must replace `nrmBaseNormal` with the new resolved normal, which will then be used in the triplanar projection as well as in the second resolve that follows. The improved result is shown in Figure 7(b).

A good use case for our framework is to add fine detail on top of parallax mapping (or POM) via post-resolve bump mapping. The reasoning is that after parallax mapping the original surface is conceptually replaced with a new virtual surface, so we want to perturb the normal of this new surface rather than composite the detail via blending. An example is shown in Figure 7(c).

Alternatively, we can use a volumetric bump map to introduce fine detail (such as in Figure 7(d)), where the surface position of the virtual surface is used as a sampling point. A traditional vertex-level tangent space represents an orthonormal frame (see Equation (8)), and so no record is retained for world-space units across the surface per unit texture along s and t ($\|\frac{\partial P}{\partial s}\|$ and $\|\frac{\partial P}{\partial t}\|$). Rather than requiring an artist to dial in this ratio, we solve the problem by using the approach described in Section 4.3, where the transformation to texture space is established in the pixel shader.



(a) Triplanar projection (b) Triplanar after normal mapping

(c) Parallax with details (d) Parallax with volumetric detail

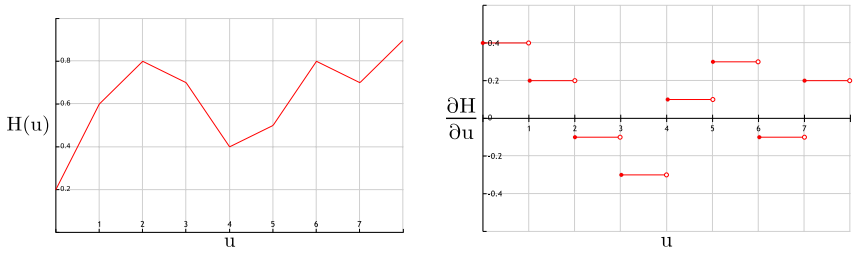
Figure 7. (a) The result of using triplanar projection, where blending is determined using the interpolated vertex normal. (b) The effect of using a traditional baked normal map followed by triplanar projection. (c) Fine detail added to the virtual surface resulting from parallax mapping. (d) Additionally, a procedural bump map, on a volume, used to produce erosion and dents.

In our example, we use the function `turbulence()` as described in [Perlin 1985], with a similar frequency clamping method to filter the signal, since MIP mapping is not an option for such a procedural approach. Based on the `dented()` shader in [Upstill 1989], we use $g(x) = x^k$ as a function to composite with `turbulence()`. We evaluate the volume gradient of the composite function $\nabla(g \circ f)$ by leveraging that $(g \circ f)' = g'(f)f'$. Finally, we establish the surface gradient using Listing 8 and resolve using Listing 1, where `nrmBaseNormal` is the normal of the virtual surface.

4.7. Bump Mapping from a Height Map

In some scenarios, it is convenient to be able to establish the derivative $(\frac{\partial H}{\partial u}, \frac{\partial H}{\partial v})$ directly from a height map in a shader: for instance, when sculpting a large-scale terrain either in code or by painting displacements to a texture, in which case maintaining a corresponding synced normal map is impractical. Another example is a case where one already uses a height map (such as with POM) but wishes to avoid the additional memory footprint of a normal map.

Figure 8. Four different ways to determine $(\frac{\partial H}{\partial u}, \frac{\partial H}{\partial v})$ for the normal. In the leftmost image, forward differencing is used. The second image also uses forward differencing, but with a minimum step size of one pixel. The third image bilinearly samples from a 2×2 block of derivatives, and the fourth image is a normal map.



(a) Height map (b) Derivative

Figure 9. (a) A piecewise-linear function $H(u)$. (b) The corresponding derivative function $\frac{\partial H}{\partial u}$.

While we can use the method in Listing 7 to perform bump mapping from a height map, it produces objectionable faceting under texture magnification, as shown in the leftmost image of Figure 8. This happens because, during upsampling, the GPU will bilinearly filter a block of 2×2 pixels at the top MIP level, which gives a signal of piecewise-bilinear patches. This is illustrated in 2D in Figure 9, where we see that the derivative function of a piecewise-linear function is a piecewise-constant function.

The problem occurs during forward differencing when the adjacent samples are taken from the same 2×2 block as the center sample. A trick for solving this is to force the adjacent samples to be taken from a minimum distance of one pixel relative to the top MIP level. We can do this implicitly by using the HLSL intrinsic `CalculateLevelOfDetail()` to determine the adjacent sampling locations, since it returns a level-of-detail value greater than or equal to zero. An implementation for this is given in Listing 16; note that the bottom half of the function is skipped when `isUpscaleHQ` is set to `false`. The corresponding result is shown in the second image in Figure 8.

The result of using an actual normal map is shown in the fourth image of Figure 8. We can achieve a closer match to this by bilinearly interpolating a 2×2 block of

derivatives $\frac{\partial H}{\partial u}$. A common discrete method to compute derivatives is to use the Sobel operator, which can be expressed as a matrix multiplication such that we obtain 2×2 derivatives from 4×4 height values. The bilinear interpolation is also expressible as a matrix multiplication. Let $Q = [H_{ij}]$ be the 4×4 matrix of pixels from the top MIP level such that the sampling location is within the center 2×2 submatrix of Q. The fractional sampling location used in the bilinear interpolation is $\vec{t} \in [0; 1]^2$, and the final transformation sequences to obtain $(\frac{\partial H}{\partial u}, \frac{\partial H}{\partial v})$ are given by Equations (20) and (21), respectively:

$$D = \frac{1}{2}\begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}, \quad B = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{bmatrix},$$

$$\frac{\partial H}{\partial u} = \begin{bmatrix} (1-\vec{t}_y) & \vec{t}_y \end{bmatrix} \cdot B \cdot Q \cdot D^T \cdot \begin{bmatrix} (1-\vec{t}_x) & \vec{t}_x \end{bmatrix}^T, \tag{20}$$

$$\frac{\partial H}{\partial v} = \begin{bmatrix} (1-\vec{t}_y) & \vec{t}_y \end{bmatrix} \cdot D \cdot Q \cdot B^T \cdot \begin{bmatrix} (1-\vec{t}_x) & \vec{t}_x \end{bmatrix}^T. \tag{21}$$

These are used in Listing 16 when `isUpscaleHQ` is set to `true` and only during texture magnification. As an optimization, we fetch the 4×4 heights using just four HLSL `Gather()` calls. The result is shown in the third image of Figure 8, and as we can see, it more closely resembles the normal-mapped result shown in the last image. The corresponding tangent-space normal can be obtained by applying Equation (12) to the derivative returned by the function in Listing 16.

5. Conclusion

We have presented a comprehensive framework of utility functions for performing advanced compositing of bump maps, with support for multiple sets of texture coordinates, as well as correct normal perturbation with volumetric bump mapping.

With this framework, we have reconciled with traditional normal mapping by introducing an alternative yet equivalent surface gradient–based formulation, which allows us to remain compliant with existing baked normal maps.

Strictly speaking, full compliance is only achieved when a vertex-level tangent space is provided by the application. In practice, this is primarily a requirement for low-polygon hard surface modeling, and in our experience the replacement of a vertex-level tangent space with a procedural approach does not exhibit a loss in fidelity in the majority of cases. It seems likely that the reliance on a vertex-level tangent space will lessen over time; until then, our procedural approach provides a strong alternative for multiple sets of texture coordinates as well as procedural texture coordinates.

```
// Returns dHduv where (u,v) is in pixel units at the top MIP level.
float2 DerivFromHeightMap(Texture2D hmap, SamplerState samp,
                          float2 texST, bool isUpscaleHQ = false)
{
    float lod = hmap.CalculateLevelOfDetail(samp, texST);
    uint2 dim; hmap.GetDimensions(dim.x, dim.y);
    float2 onePixOffs = float2(1.0/dim.x, 1.0/dim.y);
    float eoffs = exp2(lod);
    float2 actualOffs = onePixOffs*eoffs;
    float2 st_r = texST + float2(actualOffs.x, 0.0);
    float2 st_u = texST + float2(0.0, actualOffs.y);

    float Hr = hmap.Sample(samp, st_r).x;
    float Hu = hmap.Sample(samp, st_u).x;
    float Hc = hmap.Sample(samp, texST).x;
    float2 dHduv = float2(Hr - Hc, Hu - Hc)/eoffs;
    float start = 0.5, end = 0.05; // start-end fade
    float mix = saturate((lod - start)/(end - start));

    if (isUpscaleHQ && mix > 0.0)
    {
        float2 f2TexCoord = dim*texST - float2(0.5, 0.5);
        float2 f2FlTexCoord = floor(f2TexCoord);
        float2 t = saturate(f2TexCoord - f2FlTexCoord);
        float2 cenST = (f2FlTexCoord + float2(0.5, 0.5))/dim;
        float4 sampUL = hmap.Gather(samp, cenST, int2(-1,-1));
        float4 sampUR = hmap.Gather(samp, cenST, int2( 1,-1));
        float4 sampLL = hmap.Gather(samp, cenST, int2(-1, 1));
        float4 sampLR = hmap.Gather(samp, cenST, int2( 1, 1));

        // float4(UL.wz, UR.wz) represents first scanline and so on.
        float4x4 H = {sampUL.w, sampUL.z, sampUR.w, sampUR.z,
                      sampUL.x, sampUL.y, sampUR.x, sampUR.y,
                      sampLL.w, sampLL.z, sampLR.w, sampLR.z,
                      sampLL.x, sampLL.y, sampLR.x, sampLR.y};

        float2 A = float2(1.0 - t.x, t.x);
        float2 B = float2(1.0 - t.y, t.y);
        float4 X = 0.25*float4(A.x, 2*A.x + A.y, A.x + 2*A.y, A.y);
        float4 Y = 0.25*float4(B.x, 2*B.x + B.y, B.x + 2*B.y, B.y);
        float4 dX = 0.5*float4(-A.x, -A.y, A.x, A.y);
        float4 dY = 0.5*float4(-B.x, -B.y, B.x, B.y);
        float2 dHduv_ups = float2(dot( Y, mul(H, dX)),
                                  dot(dY, mul(H,  X)));
        dHduv = lerp(dHduv, dHduv_ups, mix);
    }

    return dHduv;
}
```

Listing 16. Derivative on the fly from a height map.

Acknowledgements

My deep gratitude goes to both Stephen Hill as editor and to the anonymous reviewers, for their dedication and active role in improving the exposition of this paper. I would also like to thank Evgenii Golubev for his many constructive comments and proofreading, Martin Vestergaard Kümmel for allowing me to use production content, and Aras Pranckevičius for his review and constructive comments.

References

BARRÉ-BRISEBOIS, C., AND HILL, S., 2012. Blending in detail. *Self Shadow*, July 10, https://blog.selfshadow.com/publications/blending-in-detail. 76, 77

BLINN, J. 1978. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, 286–292. 61, 72, 80

BRAWLEY, Z., AND TATARCHUK, N. 2005. Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing. In *ShaderX3: Advanced Rendering with DirectX and OpenGL*, W. Engel, Ed., ShaderX Series. Charles River Media, 135–154. 78

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 2003. *Texturing and Modeling: A Procedural Approach*, third ed. Morgan Kaufmann. 72

GEISS, R., 2007. Cascades demo secrets. *SlideShare*, September 10, http://www.slideshare.net/icastano/cascades-demo-secrets. 80

GEISS, R. 2007. Generating complex procedural terrains using the GPU. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Publishing, ch. 1, 7–38. 81

GOLUS, B., 2020. Generating perfect normal maps for unity (and other programs). *Medium*, Feburary 3, https://medium.com/@bgolus/generating-perfect-normal-maps-for-unity-f929e673fc57. 68

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., AND TACHI, S. 2001. Detailed shape representation with parallax mapping. In *Proceedings of the ICAT 2001*, The Virtual Reality Society of Japan, 205–208. 77

KILGARD, M. J. 2000. Advanced opengl game development: A practical and robust bump-mapping technique for today's gpus. Game Developers Conference. http://developer.download.nvidia.com/assets/gamedev/docs/GDC2K_gpu_bump.pdf. 61

MIKKELSEN, M. S. 2008. *Simulation of Wrinkled Surfaces Revisited*. Master's thesis, Department of Computer Science at the University of Copenhagen. 62, 80

MIKKELSEN, M. S. 2010. Bump mapping unparametrized surfaces on the gpu. *J. Graphics, GPU, & Game Tools 15*, 1, 49–61. 63, 72

MIKKELSEN, M. S., 2011. Tangent space normal maps. http://www.mikktspace.com. 68

PEERCY, M. S., AIREY, J., AND CABRAL, B. 1997. Efficient bump mapping hardware. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM, 303–306. 64

PERLIN, K. 1985. An image synthesizer. *SIGGRAPH Comput. Graph. 19*, 3, 287–296. doi:10.1145/325165.325247. 72, 73, 85

PERLIN, K. 2002. Improving noise. *ACM Trans. Graph. 21*, 3 (July), 681–682. doi:10.1145/566654.566636. 73

UPSTILL, S. 1989. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison–Wesley Longman Publishing Co. Dented shader, Listing 16.16, pages 350–352. 85

WELSH, T. 2004. Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces. 77, 78

Index of Supplemental Materials

A standalone demo is provided at https://github.com/mmikk/surfgrad-bump-standalone-demo. Snapshot: http://jcgt.org/published/0009/03/04/surfgrad-bump-standalone-demo-1.0.zip.

Author Contact Information

Morten S. Mikkelsen
Unity Technologies SF
mikkelsen7@gmail.com
http://mmikkelsen3d.blogspot.com/p/3d-graphics-papers.html

Morten S. Mikkelsen, Surface Gradient–Based Bump Mapping Framework, *Journal of Computer Graphics Techniques (JCGT)*, vol. 9, no. 3, 60–90, 2020
http://jcgt.org/published/0009/03/04/

Received:	2020-05-20		
Recommended:	2020-07-09	Corresponding Editor:	Stephen Hill
Published:	2020-10-21	Editor-in-Chief:	Marc Olano