Surface Gradient Based Bump Mapping Framework

Morten S. Mikkelsen
Unity Technologies, USA

October 7, 2019

Abstract

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

1 Introduction

Since year 2000 as described in [MJ00] the approach to bump/normal mapping in real-time 3D graphics has remained largely the same. That is a transformation by the 3x3 TBN (tangent, bitangent and normal) matrix where the tangent and bitangent are precomputed offline at vertex level. Today this conventional approach is beginning to show its age.

1. In modern computer graphics, material layering is critical to achieve rich and complex environments. However, conventional normal mapping does not extend to multiple sets of texture coordinates since game engines do not store more than one tangent space per vertex.

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 per vertex tangent space in real-time, in every frame, is impractical. Some examples are blendshapes, tessellation, cloth, water and potentially trees.

For blendshapes one might alternatively store a pre-generated tangent and sign bit with each vertex of a blendshape, but footprint in memory is a concern,

plus linearly blending sets of tangent vectors across several shapes will often provide incorrect results. In the context of tessellation generating 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, barycentric/bilinear interpolation of tangents will not provide a vector which is tangent to the limit surface.

The reader of this paper is expected to be familiar with shader authoring and traditional normal mapping, but beyond this, a novice to intermediate level in math is sufficient.

2 Modern Framework

What we want is a practical modern approach/framework to 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 in the paper "Bump Mapping Unparametrized Surfaces on the GPU" [Mik10]. It follows from the discovery made in this paper that Jim Blinn's perturbed normal [Bli78] is identical to the initial unit length normal $\vec{n}$ minus the surface gradient ∇_S of the bump map $H : (x, y, z) \rightarrow \mathbb{R}$ (eq. 4 in [Mik10])

$$\vec{n}' = \frac{\vec{n} - \nabla_S H}{\|\vec{n} - \nabla_S H\|} \tag{1}$$

It is not necessary to understand what a surface gradient is to leverage the framework presented in this paper but one way to think of it is as an orthographic projection, along $\vec{n}$, of the volume gradient $\nabla H = \left(\frac{\partial H}{\partial x}, \frac{\partial H}{\partial y}, \frac{\partial H}{\partial z}\right)$ onto the tangent plane of the surface at the point being shaded (eq. 2 in [Mik10]). This definition of the surface gradient also applies to 2D bump maps as explained in the appendix.

The surface gradient based formulation of bump mapping provides a unified solution because of two important properties of the surface gradient.

1. The surface gradient is a linear operator.

2. The surface gradient depends on the shape of the surface and the given distribution of bumps across the surface but does not depend on the underlying parametrization.

The first property tells us 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

$$H = s_0 \cdot H_0 + s_1 \cdot H_1 + \cdots + s_n \cdot H_n \tag{2}$$

$$\nabla_S H = s_0 \cdot \nabla_S H_0 + s_1 \cdot \nabla_S H_1 + \cdots + s_n \cdot \nabla_S H_n \tag{3}$$

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

The second property tells us it does not matter what the specific UV layout looks like or even if the bumps were mapped to the surface via a volume bump map. The two properties allow us to accumulate different forms of bump contributions as surface gradients provided we can establish the surface gradient in each case. The following structure for a bump mapping framework emerges from equations (1) and (3)

1. Bump scales are applied by scaling each surface gradient by a user-defined displacement value.
2. All surface gradients are added together into one final surface gradient.
3. Finally the resolve is done by normalizing the difference between the initial unit length surface normal and the accumulated surface gradient.

We can perform the resolve step which is given by equation (1) using the function in listing 1 and we can adjust the intensity of a bump map by modulating it's surface gradient using the function in listing 2 or alternatively the derivative directly as shown in listing 3. The values *nrmBaseNormal* and *resolveSign* are calculated and written to static globals during the prologue step outlined in section 2.11.

```
float3 ResolveNormalFromSurfaceGradient(float3 surfGrad)
{
  // resolve sign +/-1. Should only be -1 for double sided
  // materials when viewing the back-face and the mode is
  // set to flip.
  return normalize(nrmBaseNormal - resolveSign*surfGrad);
}
```

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

```
float3 ApplyBumpScale(float scale, float3 surfGrad)
{
  return scale * surfGrad;
}
```

Listing 2: Bump scale applied to the surface gradient.

```
float2 ApplyBumpScale(float scale, float2 deriv)
{
  return scale * deriv;
}
```

Listing 3: The bump scale can be applied to the derivative.

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

2.1 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 shader side it is more practical to convert the sampled value at runtime into a two-component derivative rather than a three-component vector.

The complexity to convert the usual two channels of a tangent space normal $\vec{m} \in [-1;1]$ to a derivative $\vec{d} = \left(\frac{\partial H}{\partial u}, \frac{\partial H}{\partial v}\right)$ is equal to

$$\vec{d} = \left(-\frac{\vec{m}_x}{\vec{m}_z}, -\frac{\vec{m}_y}{\vec{m}_z}\right) \tag{4}$$

$$= \frac{(-\vec{m}_x, -\vec{m}_y)}{\sqrt{\max\left(\varepsilon, 1 - \vec{m}_x^2 - \vec{m}_y^2\right)}} \tag{5}$$

which, thanks to the intrinsic $rsqrt()$, is comparable to the usual reconstruction of the third component of the tangent space normal from the two sampled components $\vec{m}_z = \sqrt{\max\left(\varepsilon, 1 - \vec{m}_x^2 - \vec{m}_y^2\right)}$.

```
// input: vM is channels .xy of a tangent space normal in [-1;1]
// out: convert vM to a derivative
float2 TspaceNormalToDerivative(float2 vM)
{
  const float fS = 1.0/(128*128);
  float2 vMsq = vM*vM;
  const float mz_sq = 1-vMsq.x-vMsq.y;
  const float maxcompxy_sq = fS*max(vMsq.x,vMsq.y);
  const float z_inv = rsqrt( max(mz_sq,maxcompxy_sq) );

  // set to match positive vertical texture coordinate axis
  const bool gFlipVertDeriv = false;
  const float s = gFlipVertDeriv ? -1 : 1;
  return -z_inv*float2(vM.x,s*vM.y);
}
```

Listing 4: Convert from two channels of a tangent space normal $\vec{m}$ to a derivative $\vec{d}$.

The implementation in listing 4 returns the derivative $\vec{d} \in [-128; 128]$ which corresponds to a maximum angle of 89.55°. Unlike tangent space normals we can do proper blending/mixing of derivatives when they are sampled using the same uv set. However, as discussed in section 2 we can also do this at surface gradient level which allows us to blend across all bump influences regardless of uv set.

The main motivation for using the derivative form rather than tangent space normals in the shader is it implicitly allows all bump contributions to conform to equation (1) as we shall see in section 2.2.

It is important to note that the correct setting for *gFlipVertDeriv* in listing 4 is always a constant but is specific to your codebase. When *gFlipVertDeriv* is set to *false* then if $(0, 0)$ represents upper-left corner in the texture then positive green in the normal map must represent down or alternatively if $(0, 0)$ represents lower-left corner then positive green must represent up. If this is not the case then you need to set *gFlipVertDeriv* to *true* in listing 4 and most likely also need to negate *signw* in listing 16 depending on your existing build process and shaders. This is because the derivative must be wrt. to the positive axis for s and for t.

2.2 TBN–style surface gradient

A traditional TBN formulation is given as $\vec{t} \cdot \vec{m}_x + \vec{b} \cdot \vec{m}_y + \vec{n} \cdot \vec{m}_z$ where $\vec{t}$ and $\vec{b}$ are the tangent and bitangent. As previously defined, $\vec{n}$ is the initial normalized surface normal and $\vec{m}$ is the tangent space normal. By leveraging equation (4) and rearranging terms, we can do a derivative–based formulation of TBN style normal perturbation which corresponds to equation (1).

$$\begin{aligned} \frac{1}{\vec{m}_z} \cdot \left(\vec{t} \cdot \vec{m}_x + \vec{b} \cdot \vec{m}_y + \vec{n} \cdot \vec{m}_z \right) &= \vec{n} - \left(\vec{t} \cdot \frac{-\vec{m}_x}{\vec{m}_z} + \vec{b} \cdot \frac{-\vec{m}_y}{\vec{m}_z} \right) \\ &= \vec{n} - \left(\vec{t} \cdot \vec{d}_x + \vec{b} \cdot \vec{d}_y \right) \end{aligned} \tag{6}$$

Note that the scale by $\frac{1}{\vec{m}_z} > 0$ will get canceled in the final resolve as indicated by eq. (1) and listing 1. In order to achieve full compliance with **mikktspace** [Mik11] we must also take into account this transformation is defined such that the interpolated but **unnormalized** vertex attributes for $\vec{t}$, $\vec{b}$ and $\vec{n}$ must be used in the shader to match the baked normal map. However, for the surface gradient based formulation, the interpolated vertex normal must be normalized. We can solve this conflict by reusing the same trick which is a positive uniform scale of the perturbed normal gets canceled in the final resolve so we can multiply all three by the reciprocal magnitude of the unnormalized but interpolated vertex normal.

$$\begin{aligned} \vec{t} &\leftarrow \frac{1}{\|\vec{n}\|} \cdot \vec{t} \\ \vec{b} &\leftarrow \frac{1}{\|\vec{n}\|} \cdot \vec{b} \\ \vec{n} &\leftarrow \frac{1}{\|\vec{n}\|} \cdot \vec{n} \end{aligned}$$

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 *mikktsTang* and *mikktsBino* in listing 16. The final TBN style surface gradient is summarized in listing 5 and can be used with the adjusted basis vectors as parameters for vT and vB.

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

Listing 5: TBN–Style Surface Gradient.

Strict mikktspace compliance is required for cases such as baked low polygonal hard surface modeling to obtain artifact free renderings [Mik08]. However, since game engines only provide one tangent space per vertex this does not account for multiple sets of texture coordinates or even procedural texture coordinates which we cover in the next section 2.3.

2.3 TBN basis without vertex level tangent space

For a scenario where the tangent $\vec{t}$ and bitangent $\vec{b}$ do not exist at vertex level for the given texture coordinate, 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 2.11. The values *sigmaX* and *sigmaY* represent the first order derivatives of the surface position wrt. screen–space *dPdx* and *dPdy* but perpendicular to *nrmBaseNormal*. By using the chain rule we obtain the tangent as $\vec{t} = sigmaX \cdot dXds + sigmaY \cdot dYds$.

```
void GenBasisTB(out float3 vT, out float3 vB, float2 texST)
{
  float2 dSTdx = ddx_fine(texST), dSTdy = ddy_fine(texST);

  float det = dot(dSTdx, float2(dSTdy.y,-dSTdy.x));
  float sign_det = det<0 ? -1 : 1;

  // invC0 represents (dXds, dYds); but we don't divide by
  // determinant (scale by sign instead)
  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 6: Procedural TBN basis.

The function in listing 6 delivers the tangent frame as an orthonormal set that may be used as parameters given to the function SurfgradFromTBN() in listing 5. Furthermore, the function works both when texture coordinates are defined clockwise or counterclockwise.

2.4 Surface Gradient from Object/World–space normal

In some cases it is convenient for an artist to use an object/world space normal map which 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 these with other bump/normal mapping contributions such as tangent space normal maps, decals, triplanar projection, etc. In other words, we need a method to convert the object/world–space normal to a surface gradient.

Since we already know the initial normal of the surface, and since we know the object/world–space normal, we can solve for the surface gradient by reordering terms in equation (1) which is how we arrive at the utility function in listing 7.

```
// 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. v does not need
// to be unit length as long as it establishes the direction.
// The vector v and the normal must be in the same space.
float3 SurfgradFromPerturbedNormal(float3 v)
{
  float3 n = nrmBaseNormal;
  float s = 1.0/max(FLT_EPSILON,abs(dot(n,v)));
  return s * ( dot(n,v)*n - v);
}
```

Listing 7: Runtime object/world–space normal to surface gradient.

2.5 Scale–Dependent Bump Mapping

It is important to distinguish between scale–dependent bump mapping and TBN style bump mapping. In the latter case 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 convenience in workflow for the artist, tradition, but also reducing memory footprint of the tangent frame by assuming it represents an orthonormal basis at vertex level.

When doing 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, and should not assume the bitangent is perpendicular to the tangent, either.

The surface gradient for this scenario is summarized in listing 8 using equation 3 in [Mik10]. The first parameter *deriv* in listing 8 corresponds to $\left(\frac{\partial H}{\partial u}, \frac{\partial H}{\partial v}\right)$, where (u, v) represents unnormalized texture coordinates meaning one unit is one pixel in the displacement map. Additionally, the magnitude of the derivative must fit the applied displacements.

The values for *dPdx*, *dPdy* and *nrmBaseNormal* are cached in the prologue given in listing 16 and must all be in the same frame of reference. An **important** subtlety to notice is *dPdx* and *dPdy* must be the screen space derivatives of the initial surface position **before displacement**. The only time you would want

to use the screen-space derivative of the position after displacement is if you are doing a separate bump mapping pass on the new surface after displacement mapping in which case *nrmBaseNormal* would also have to be replaced with the normal of the displaced surface.

When *isDerivScreenSpace* is set to true, *deriv* represents the screen–space derivative of the height *dHdx* and *dHdy*. A crude single sample approach to calculate these in the pixel shader is to use *ddx_fine(height)* and *ddy_fine(height)* directly. However, since this is based on numerical differencing in a block of *2x2* pixels better quality is achieved using the 3–tap approach shown in listing 9.

```
// dim must be the 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 st space
  const float2 dHdST = dim*deriv;

  // convert derivative to screen space by 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 "bump mapping unparametrized surfaces
  // on the gpu".
  float3 vR1 = cross(dPdy, nrmBaseNormal);
  float3 vR2 = cross(nrmBaseNormal, dPdx);
  float det = dot(dPdx, vR1);

  float sgn = det<0.0 ? (-1.0f) : 1.0f;
  float s = sgn / max(FLT_EPSILON, abs(det));

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

Listing 8: Surface gradient of scale–dependent bump mapping.

```
  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;

  // 3 taps - better quality than: float2(ddx_fine(H), ddy_fine(H))
  float2 deriv = float2(Hlr-Hll, Hul-Hll);
```

Listing 9: Forward Differencing. 3–tap screen–space derivative of the height.

2.6 Parallax Mapping

Techniques which are strongly related to displacement mapping (described in section 2.5) are parallax mapping [Wel04] and parallax occlusion mapping (POM) [KTI+01]. Just like displacement mapping, these techniques are sensitive to tiling rate so using a TBN style frame of reference is incorrect.

A detailed explanation of parallax mapping/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 10 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 $(\vec{v}_x, \vec{v}_y, \vec{v}_z) \rightarrow \left(\frac{\vec{v}_x}{\vec{v}_z}, \frac{\vec{v}_y}{\vec{v}_z}, 1\right)$ such that third component is always one unit along *nrmBaseNormal*. This projection is not used with the more basic approach in [Wel04] 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 10 where we scale by the user parameter bumpScale. That is to account for this factor in the displacement vector *bumpScale * H(s,t) * nrmBaseNormal*. This implies a height level of 1.0 corresponds to the displacement distance *bumpScale*, along the normal, in world space.

A useful feature is the ability to evaluate the corrected position at the surface which corresponds to the corrected offset for the texture coordinate as provided by the function in listing 11. 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 the same thing could not be done using 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 $vx \cdot dNdx + vy \cdot dNdy$. The values vx and vy are as given in listing 11 and $dNdx$, $dNdy$ represent *ddx()* and *ddy()* on *nrmBaseNormal*.

```
// dir: must be a normalized vector in same space as the surface
// position and normal.
// bumpScale: p' = p + bumpScale * DisplacementMap(st) * 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);

  float sgn = det<0.0 ? (-1.0f) : 1.0f;
  float s = sgn / max(FLT_EPSILON, abs(det));
```

```
  // transform
  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
  sgn = dirTexZ<0.0 ? (-1.0f) : 1.0f;
  s = sgn / max(FLT_EPSILON, abs(dirTexZ));
  return bumpScale * (skipProj ? 1.0f : s) * dirTex;
}
```

Listing 10: Project vector to texture space for parallax mapping.

```
// initialST: Initial texture coordinate before parallax correction
// corrOffs: the 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;
  float sgn = det<0.0 ? (-1.0) : 1.0;
  float s = sgn / max(FLT_EPSILON, 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 11: Transform the POM chosen correction vector from texture space to the surface.

2.7 Volume Bump Maps

The concept of a *space function* was introduced in [Per85] and is defined as a function H that varies over a three dimensional domain $H : (x, y, z) \to \mathbb{R}$. 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. The author 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 this will be referred to as a *volume bump map*.

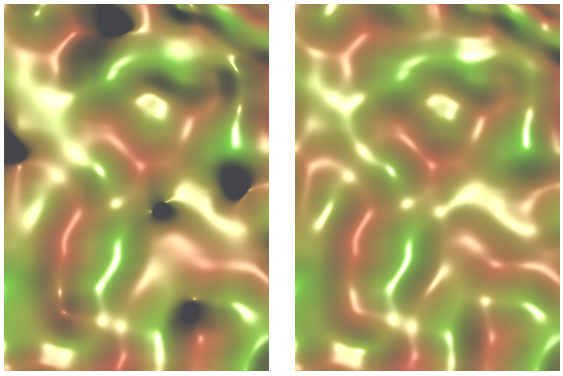
It is suggested in [Per85] that a normal perturbing effect may be achieved by adding the volume gradient $\nabla H = \left(\frac{\partial H}{\partial x}, \frac{\partial H}{\partial y}, \frac{\partial H}{\partial z}\right)$ to the initial normal $\vec{n}$ and renormalizing. This does not produce the correct normal, which is pointed out by Steve Worley in [EMP+03], and instead the correct solution is described as

resampling the space function but using the traditional bump mapping method of Jim Blinn [Bli78]. However, this requires a known surface parametrization, known partial derivatives of the surface position, and generally requires storing the sampled displacements in a texture, performing a precomputation step to evaluate the derivatives of H wrt. u and v, and generate mip maps, which is presumably why this approach is not used in [Per85] nor in [EMP+03].

A significantly simpler way to achieve the correct result is to leverage the discovery made in [Mik10] that Jim Blinn's perturbed normal has a surface gradient based formulation, as given by equation 4 in their paper, and is provided here 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 2 in [Mik10] which is summarized here as a function in listing 12.

```
// used to produce a surface gradient from the gradient of a volume
// bump function such as a volume of Perlin noise.
// equation 2. in "bump mapping unparametrized surfaces on the GPU"
float3 SurfgradFromVolumeGradient(float3 grad)
{
  return grad - dot(nrmBaseNormal, grad)*nrmBaseNormal;
}
```

Listing 12: Volume gradient to surface gradient.



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

Figure 1: The result of Perlin based normal perturbation is shown in figure 1(a). There are visual artifacts which are gone using our method shown in figure 1(b).

The difference between Perlin's method in [Per85] and doing it correctly is visually significant. A comparison using Perlin noise (see [Per02]) based bump mapping can be seen above in figure 1 where elevation level is indicated in green and reddish–brown. As a correction, ∇H is negated in 1(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 1(a). Using the surface gradient based method we get the artifact–free result shown in figure 1(b).

It is worth reiterating the point made in section 2 that the surface gradient does not depend on the underlying parametrization. What matters is the shape of the surface and the distribution of the displacement values across the surface. Storing the displacements in a 2D texture map and using traditional bump mapping is mathematically equivalent to using the method in listing 12 and resolving using listing 1. For a more detailed explanation on this read the appendix.

2.8 Triplanar Bump Mapping

Triplanar projection [Gei07] 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:

$$H(x, y, z) = w_x \cdot H_x(z, y) + w_y \cdot H_y(x, z) + w_z \cdot H_z(x, y) \tag{7}$$

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. Note that we do not need the actual displacement maps for these images H_x, H_y and H_z they are conceptual priors.

In equation (7) we have implicitly assumed $(0, 0)$ represents the lower–left corner of the height maps. A change to 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 we receive in listing 13.

Similarly, a change to a right–hand coordinate frame is easy. In this case $H_x(-z, y)$ and $H_y(x, -z)$ are sampled using negated coordinates for z which means we must negate *deriv_xplane.x* and *deriv_yplane.y* in listing 13 which corresponds to negating the third component of *grad* in the listing.

The goal is to establish the surface gradient $\nabla_S H$ for eq. (7). Note that $w(\vec{n})$ depends on the initial normal $\vec{n}$, which means the derivative of w depends on the derivative of $\vec{n}$. However, similar to [Bli78], we assume the derivative of $\vec{n}$ is sufficiently close to zero. For a deeper analysis on this approximation the reader is referred to [Mik08]. Thus we can proceed under the assumption that $w(\vec{n}) = (w_x, w_y, w_z)$ is sufficiently constant at each point p we are shading. Though eq. (7) does not represent a valid volume bump map at a global level due to it's dependency on $\vec{n}$, which 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

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

We can finally determine the surface gradient $\nabla_S H$ using equation 2 in [Mik10]. It is worth pointing out that an alternative path is to establish the surface gradient for each two–dimensional mapped image separately using equation 3 in [Mik10] and then accumulate the three surface gradients as a weighted average using the coordinates of w as weights. However as previously stated this results in exactly the same surface gradient $\nabla_S H$.

The final implementation of triplanar bump/normal 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 simply replace *nrmBaseNormal* with the perturbed normal in listings 1, 12 and 14.

```
// triplanar projection considered special case of volume bump map
// derivs obtained using TspaceNormalToDerivative() and weights
// using DetermineTriplanarWeights().
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 sampled
  // sampled using (z,y), (x,z) and (x,y) respectively. Positive
  // scales of the look-up 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.

```
// http://www.slideshare.net/icastano/cascades-demo-secrets
float3 DetermineTriplanarWeights(float k=3.0)
{
  float3 blend_weights = abs(nrmBaseNormal) - 0.2;
  // seems pointless with the normalization at the end
  //blend_weights *= 7;
  blend_weights = max(0, blend_weights);
  blend_weights = pow(blend_weights, k);
  blend_weights /=
      (blend_weights.x+blend_weights.y+blend_weights.z);

  return blend_weights;
}
```

Listing 14: Determine weights for triplanar mapping.

2.9 Decal Projector

Similar to triplanar projection as described in section 2.8 the decal projector is a special case which is ideal to process 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 coordinate, which means the image is conceptually repeated along the $\vec{z}$ axis so the volume gradient transformed into a common frame of reference is equal to

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

again we can determine the surface gradient $\nabla_S H$ using equation 2 in [Mik10] which is summarized in code in listing 15.

```
// axisX: X axis of decal projector. Same space as normal
// axisY: Y axis of decal projector. Same space as normal
float3 SurfgradFromDecalProjector(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 2.8 we have assumed $(0, 0)$ represents the lower–left corner of the bump map. As before to convert to upper–left corner invert the second sampling coordinate and negate *deriv.y* in listing 15.

An important observation to make is just as the surface gradient is a linear operator the same is true for the volume gradient. We do not have to perform the projection to the surface, *SurfgradFromVolumeGradient()*, per decal. We can accumulate the volume gradient of each decal one by one and then project onto the surface at the end to get 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). The same feature allows us to modulate by the spatial fall-off associated with the projector volume as a bump scale applied to either the surface gradient or to the volume gradient.

2.10 Double Sided Materials

When rendering double–sided materials there are three possible modes when viewing the back–facing side of a triangle.

1. **None** – Perturbed normal remains exactly same as front-side.
2. **Mirror** – The interpolated vertex normal flips and the bump map is mirrored.

3. **Flip** – Perturbed normal is flipped.

The tangent and bitangent represent the first–order derivative of the surface position wrt. the texture coordinates s and t so these directions do NOT change on the back–facing side. The interpolated vertex normal is flipped/negated when the mode is either *Mirror* or *Flip*.

Conceptually bump/displacements follow the interpolated vertex normal. This implies when the height map is unchanged, and the interpolated vertex normal is negated, the displacements will follow the negated direction. This corresponds to the second scenario *Mirror* and in this case according to equation 3. in [Mik10] the surface gradient remains identical to it's value on the front–face side. If on the other hand we negate the surface gradient this corresponds to negating the height map. If we do this and negate the interpolated vertex normal this gives us scenario *Flip* which follows from eq. 4 in [Mik10]. In other words we only need to negate the surface gradient in scenario *Flip* and this should be done just before the resolve using listing 1.

We handle these specifics in the prologue in listing 16 and in listing 1. The value for *g_bfaceMode* must be provided by the application and is expected to have one of the following values: *MODE_NONE*, *MODE_MIRROR* or *MODE_FLIP*. Notice when *nrmBaseNormal* is negated the same is true for *flip_sign* which preserves the direction of the generated bitangent in listing 6. The final surface gradient will only be negated when *resolveSign* is set to -1 in the prologue when we observe a back-facing triangle and *g_bfaceMode* is set to *MODE_FLIP*.

An alternative to handling scenario *Flip* directly in listing 1 during the resolve is to convert from scenario *Mirror* to scenario *Flip*. This is done at the end once all resolves are complete by reflecting the perturbed normal $\vec{n}'$ by the initial normal $\vec{n}$ as it was before the first resolve. In hlsl we can do this using the intrinsic *reflect*$(-\vec{n}', \vec{n})$.

2.11 Pixel Shader Prologue of Framework

The code snippet in listing 16 calculates various parameters which are used throughout several of the listings provided in this document. In order 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 but a second pass of bump mapping needs to be applied on top 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 the correspondence between the pixels in each block of *2x2* has diverged. An example of this is POM, since different intersection points may be found during the ray–marching step at each of the four pixels. In this case we can calculate *dPdx* and *dPdy* analytically from the position and normal as shown in listing 17. Using this method, *dPdx* and *dPdy* will already be perpendicular to the *nrmBaseNormal* so these will be identical to *sigmaX* and *sigmaY*.

```
// smallest such that 1.0+FLT_EPSILON != 1.0
#ifndef FLT_EPSILON
  #define FLT_EPSILON     1.192092896e-07F
#endif

#define MODE_NONE     0
#define MODE_MIRROR   1
#define MODE_FLIP     2

// should be one of the above
uniform uint g_bfaceMode;


// cached bump globals reusable for all UVs including procedural
static float3 sigmaX;
static float3 sigmaY;
static float3 nrmBaseNormal;
static float flip_sign;
static float3 dPdx, dPdy;
static float resolveSign;

// used for vertex level tangent space (one UV set only)
static float3 mikktsTang;
static float3 mikktsBino;

float4 pixshader(VertexOutput In,
         bool isFrontFacing : SV_IsFrontFace) : SV_Target
{
  // cache anything reusable
  float3 relSurfPos = toRelativeWorldSpace( In.surfPos.xyz );
  float renormFactor = 1.0/length(In.normal.xyz);

  // mikkts for conventional vertex level tspace (no normalizes
  // is mandatory).
  // Using bitangent on the fly option in xnormal to
  // reduce vertex shader outputs.
  float signw = In.tangent.w>0 ? 1.0 : (-1.0);
  mikktsTang = In.tangent.xyz;
  mikktsBino = signw * cross(In.normal.xyz, In.tangent.xyz);

  // prepare for surfgrad formulation without breaking compliance
  // (use exact same scale as applied to interpolated vertex normal
  // to conform to mikktspace standard).
  mikktsTang *= renormFactor; mikktsBino *= renormFactor;

  bool isBFace = !isFrontFacing;
  float fs = (isBFace && g_bfaceMode!=MODE_NONE) ? (-1.0) : 1.0;
  nrmBaseNormal = (fs*renormFactor) * In.normal.xyz;
  resolveSign = (isBFace && g_bfaceMode==MODE_FLIP) ? (-1.0) : 1.0;

  // the values below including nrmBaseNormal will require new
  // values if there's back to back bump mapping in the shader
  // (ie. post-resolve bump mapping).

  // NO TRANSLATION! Just 3x3 transform to avoid precision issues
  dPdx = ddx_fine( relSurfPos );
  dPdy = ddy_fine( relSurfPos );
```

```
  // already in world space
  sigmaX = dPdx - dot(dPdx, nrmBaseNormal)*nrmBaseNormal;
  sigmaY = dPdy - dot(dPdy, nrmBaseNormal)*nrmBaseNormal;
  flip_sign = dot(dPdy, cross(nrmBaseNormal, dPdx))<0 ? -1 : 1;



  // ... whatever other unrelated work the shader needs to do ....
```

Listing 16: Bump mapping prologue to initialize reusable parameters.

```
// Calculate ddx(pos) and ddy(pos) analytically.
// Practical when ddx/ddy is not an available option.
// 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 coordinate (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));

  // plane in screen space
  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 sgn = planeScrSpace.z<0.0 ? -1.0 : 1.0;
  const float nz = sgn * 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 on normalized screen space [-1;1]^2
  // then scale dPdx and dPdy by 2/width and 2/height respectively
```

```
  // and negate dPdy if there's a Y axis flip.
  // dPdx *= (2.0/width); dPdy *= (2.0/height);
}
```

Listing 17: Calculate dPdx and dPdy when ddx/ddy is not available.

One issue with using analytical *dPdx* and *dPdy* is the interpolated vertex normal $\vec{n}$ may be back–facing relative to the view vector $\vec{v}$. When this is the case it may (or may not) provide better results to replace the normal in listing 17 by reflecting it toward the observer as $\vec{n} - 2 \cdot dot(\vec{v}, \vec{n}) \cdot \vec{v}$.

3 Discussion

In section 2.11 we discussed, briefly, the issue of doing a second round of normal perturbation after a resolve has already been performed following a first round of normal perturbation. In the following we refer to this form of back–to–back bump mapping as "post–resolve" bump mapping.

Using post–resolve bump mapping is particularly useful for adding fine detail to a surface where macro–scale detail is established in the first round as a normal map combined, potentially, with techniques such as tessellated displacement mapping/parallax mapping/POM/imposters etc.

In the context of parallax mapping/POM a problem in real-time computer graphics is our traditional vertex level tangent space represents an orthonormal frame which means there is no information retained on 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 relying on an artist to dial in what this ratio is we solve the problem by computing the transformation to texture space at runtime without relying on a precomputed vertex level frame of reference as shown in listing 10.

By having the proper magnitudes for $\frac{\partial P}{\partial s}$ and $\frac{\partial P}{\partial t}$, we are able to convert the chosen offset in texture–space to the corresponding offset in 3D space within the tangent plane as given in listing 11. From this offset and by sampling the height/displacement value at the corrected location we are able to establish the new position in 3D space on the virtual surface. By being able to obtain the correct new surface position we are also able to leverage this in subsequent calculations for shading and lighting but also to do post–resolve bump mapping as shown in the next section 4.

Finally since certain methods, such as POM, lead to discontinuity, between adjacent pixels on the screen this implies *ddx_fine()* and *ddy_fine()* will not produce useful results. In such a scenario and when the screen–space derivative for the new surface position is required for subsequent calculations, we provide an alternative method in listing 17 which does not rely on these built–in intrinsics. Note the same problem exists for mip level selection based on the corrected texture coordinate. A better result may be achieved by using the hlsl intrinsic *CalculateLevelOfDetail()* with the texture coordinate **before correction**, where this chosen lod is used together with the texture coordinate **after correction** to do subsequent texture sampling with *SampleLevel()* in HLSL.

4 Results

The ability to combine multiple normal maps 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 artist workflow standpoint. An example from "Book of the Dead" [Tec18] is shown in figure 2. Support for multiple sets of texture coordinates can be leveraged in many ways but one way is to have one set be an unwrap of the 3D model and a second set which is tileable. The first set is used for baking normals while the second set is used for tiling generic surface detail.

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 3(a) this approach does not produce convincing results since the interpolated vertex normal is not a good representation for the tangent plane of the surface. Using our framework, and thus correct math for volumetric bump maps, 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 is done, 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 3(b).

When doing parallax mapping (or POM), a good usecase for our framework is to introduce fine detail with post–resolve bump mapping as described in section 3. After parallax/POM the original surface is conceptually replaced with a new virtual surface. Rather than blending, we want to perturb the normal of this new surface. An example of this using a detail normal map is shown in figure 3(c).

Alternatively, we can use a volumetric bump map to introduce fine detail such as shown in figure 3(d), where the surface position of the virtual surface (as described in section 3) is used as a sampling point. In this example, we use the function *turbulence()* as described in [Per85] 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 work of Larry Gritz [Gri94] 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 12 and resolve using listing 1, where *nrmBaseNormal* is the normal of the virtual surface.

5 Conclusion

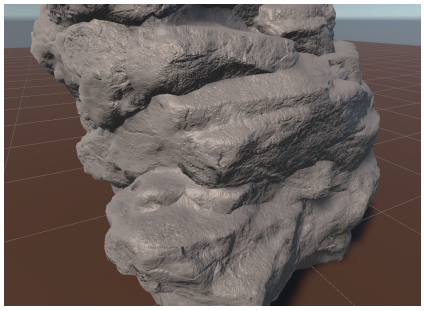
In this paper we have presented a comprehensive framework of utility functions to perform advanced compositing for bump/normal mapping. This includes the ability to use multiple sets of texture coordinates for normal mapping and correct normal perturbation with volumetric bump mapping.

In our framework we have reconciled with traditional normal mapping by

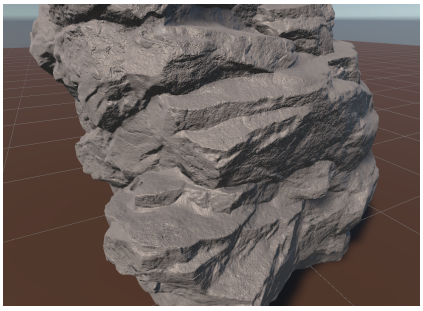
(a) Normal mapped with detail map

(b) Normal mapped

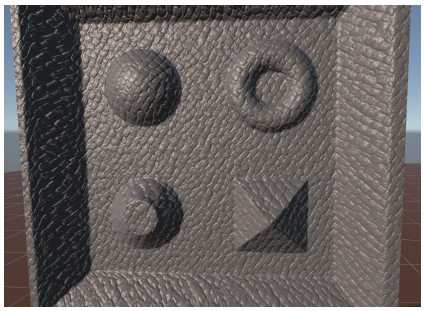
Figure 2: A normal mapped rock is shown in figure 2(b). The same asset is shown in figure 2(a) but using an additional normal map to enhance the appearance of detail on the surface.



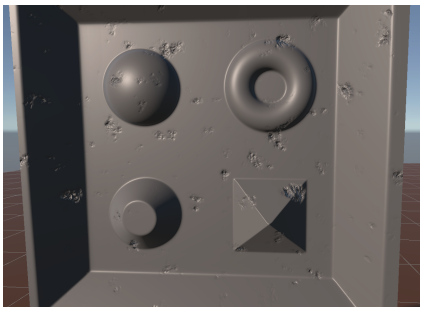
(a) Triplanar projection



(b) Triplanar after normal mapping



(c) Parallax with details



(d) Parallax with volumetric detail

Figure 3: In figure 3(a) we see the result of using triplanar projection where blending is determined using the interpolated vertex normal. The effect of using a traditional baked normal map followed by triplanar projection on top is shown in figure 3(b). Fine detail added to the virtual surface resulting from parallax mapping shown in figures 3(c) and 3(d). A procedural bump map, on a volume, was used to produce erosion and dents in figure 3(d).

introducing an alternative, yet equivalent, surface gradient based formulation to replace it with, which allows us to remain compliant with existing baked normal maps. However, strict compliance is only achieved when vertex level tangent space is provided by the application. In practice, strict compliance is primarily necessary for low polygonal hard surface modeling and, in our experience, the replacement of vertex level tangent space with a procedural approach does not exhibit loss in fidelity in the majority of cases. It seems likely the reliance on vertex level tangent space will lessen over time, but until then, our procedural approach provides a strong alternative for multiple sets of texture coordinates as well as procedural texture coordinates.

Acknowledgements. This author would like to thank Evgenii Golubev for constructive comments and proof reading, Veselin Efremov and Martin Vestergaard Kümmel for allowing me to use their production content and to Aras Pranckevicius for his review and constructive comments.

References

[Bli78] J.F. Blinn. Simulation of wrinkled surfaces. In *ACM Computer Graphics (SIGGRAPH '78)*, pages 286–292, 1978.

[EMP+03] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley. *Texturing & Modeling: A Procedural Approach: Third Edition*, chapter 5, page 171. Morgan Kaufmann, 2003.

[Gei07] Ryan Geiss. Cascades demo secrets, 2007. `http://www.slideshare.net/icastano/cascades-demo-secrets`.

[Gri94] Larry Gritz. dented.sl – displacement shader for dents, 1994. `http://www.cs.utah.edu/~schmelze/gibbon/example.htm\#dented.sl`.

[KTI+01] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *Proceedings of the ICAT 2001*, pages 205–208, 2001.

[Mik08] Morten S. Mikkelsen. Simulation of wrinkled surfaces revisited. Master's thesis, Department of Computer Science at the University of Copenhagen, 2008.

[Mik10] Morten S. Mikkelsen. Bump mapping unparametrized surfaces on the gpu. *J. Graphics, GPU, & Game Tools*, 15(1):49–61, 2010.

[Mik11] Morten S. Mikkelsen. Tangent space normal maps, 2011. `http://www.mikktspace.com`.

[MJ00] Kilgard Mark J. Advanced opengl game development: A practical and robust bump–mapping technique for today's gpus. Game Developers Conference, 2000. `http://developer.download.nvidia.com/assets/gamedev/docs/GDC2K_gpu_bump.pdf`.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[Per02] Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, July 2002.

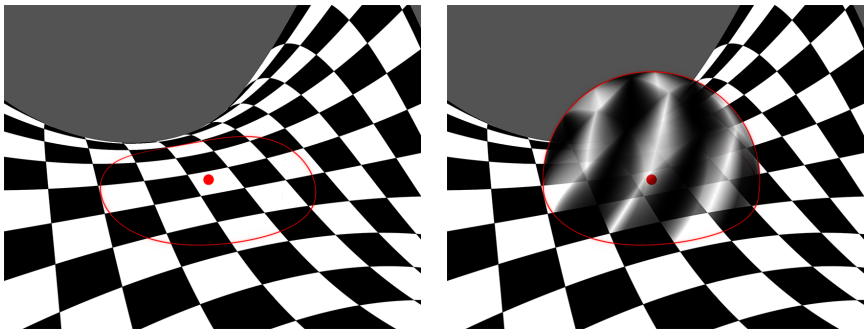
[Tec18] Unity Technologies. Book of the dead, 2018. `https://unity3d.com/book-of-the-dead`.

[Wel04] Terry Welsh. Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces. 01 2004.

Appendix: From 2D Map to a Smooth Extension

In this appendix, we show that the surface gradient, $\nabla_S H$, does not depend on the underlying parametrization of the surface. Furthermore, whether we use a 2D texture or a volume texture, to produce the same distribution of heights, H, across the surface $\nabla_S H$ will be the same.

For a given point p on some surface S, with surface normal $\vec{n}$, and a height map $H : (x, y, z) \rightarrow \mathbb{R}$ defined on an arbitrarily small open volume, containing p, we can obtain the surface gradient $\nabla_S H$, at p, by projecting $\nabla H = \left(\frac{\partial H}{\partial x}, \frac{\partial H}{\partial y}, \frac{\partial H}{\partial z}\right)$ onto the tangent plane, at p, along $\vec{n}$ (eq. 2 in [Mik10]).



(a) Cut–out region (b) Smooth extension

Figure 4: Figure 4(a) shows a surface with a cut–out region with a 2D texture map on it. The red dot represents the point p on the surface. A smooth extension of the same region to a volume texture is shown in figure 4(b). Any smooth extension containing the point p will produce the same surface gradient at p.

For a 2D bump map defined using a specific parametrization of the surface S we know we can find $\nabla_S H$ using eq. 3 in [Mik10]. However, to see that $\nabla_S H$ does not depend on the parametrization of S we can use the concept of a *smooth extension* shown in figure 4. The texture mapped to the cut–out region, containing p, is expanded to a volume texture. The surface gradient $\nabla_S H$, at p, will be the same no matter what this extension looks like as long as it is smooth and preserves values, H, when resampled at the intersection between

the volume and the cut–out region on S. This becomes clear if we rotate the scene such that the tangent plane, at p, aligns with the XY–plane. In this case for any choice of a smooth extension the gradient, ∇H, will only differ in the third component $\frac{\partial H}{\partial z}$ which will be zero after projection onto the XY–plane to produce $\nabla_S H$.