Bump Mapping Unparametrized Surfaces on the GPU

Morten S. Mikkelsen
Naughty Dog Inc., USA

May 24, 2010

Abstract

Original bump mapping is only defined for surfaces with a known surface parametrization. In this paper a new method, for the GPU, is proposed which does not use such a given parametrization. To compute the perturbed normal the only inputs used are the surface position, the height value and the original normal.

The method decouples bump mapping from the primitive type which allows for a higher degree of proceduralism in both generation of the height value and the surface.

1 Introduction

In 1978, James Blinn published his paper [Bli78] on simulation of wrinkled surfaces. The aim of his paper was to mimic the high frequency surface irregularities that we see in reality on objects everywhere. Today the effect is known as *bump mapping* and is widely used in the computer graphics industry. In real–time computer graphics a variant known as *tangent space normal mapping* is used. This is the same effect but with height derivatives precomputed. For a proof the reader is referred to [Mik08].

The effect is defined based on the premise that a surface parametrization is known. For a triangular mesh this is defined by texture coordinates which represent some *unwrap* of the surface onto a plane. The normal perturbation expression contains the first order derivatives of the surface parametrization. For triangular meshes these are precomputed at vertex level and stored in memory which adds to the footprint. Furthermore, when deformation of the surface is applied the first order derivatives must be reevaluated every frame. For a deformation which is not based on linear transformations the reevaluation requires a fully deformed version of the surface in memory which means deformation cannot be done in a vertex shader. These problems were solved in [Sch06] by computing first order derivatives in the fragment shader. However, this solution still uses normal maps and requires 2D texture coordinates.

A height map represents a single value per texel and thus has the potential to compress better than the equivalent normal map. A normal map may be thought of as height *derivative map* which requires two values. Furthermore, additional effects such as parallax/relief texture mapping [KTI*01], [OBM00] use a height map.

The requirement that 2D texture coordinates exist means the method does not apply to certain forms of proceduralism. For instance, rendering in the film industry has enjoyed the ability to distribute heights across a surface from compositions of scalar functions/textures defined on domains of arbitrary dimension since the introduction of the Reyes architecture [CCC87]. Until now this ability has not been available on the GPU. We present an efficient method, for the GPU, which overcomes these limitations and does not rely on any predetermined parametrization.

2 The Formulation of Bump Mapping

Let $S \subset \mathbb{R}^3$ represent some manifold surface and let β represent the height function such that a scalar value is associated with each point on S. In Jim Blinn's formulation the surface S has a known surface parametrization and we denote this $\sigma : (s,t) \rightarrow S$ where $(s,t) \in \mathbb{R}^2$. Furthermore, the height map is assumed to be a function of the same parameter $\beta : (s,t) \rightarrow \mathbb{R}$. Given the unit length surface normal $\vec{n}$ the displaced surface is defined as

$$\tau = \sigma + \beta \cdot \vec{n}$$

The surface normal of τ represents the perturbed normal $\vec{n}'$ used in bump mapping. To simplify the evaluation Blinn uses approximations for the first order derivatives in which the last term is neglected.

$$\begin{aligned} \tau_s &\simeq \sigma_s + \beta_s \cdot \vec{n} \\ \tau_t &\simeq \sigma_t + \beta_t \cdot \vec{n} \end{aligned}$$

Note that the normal $\vec{n}$ represents the visible side of the initial surface S. It is not taken into account in [Bli78] that the parametrization defined by σ is not necessarily orientation preserving. A derivation, of the perturbed normal $\vec{n}'$, which takes this into account is given in [Mik08] and the unnormalized result is

$$\begin{aligned} N &= \left[\begin{array}{c|c|c} \sigma_s & \sigma_t & \vec{n} \end{array} \right] \\ \vec{n}' &= (N^{-1})^T \cdot \begin{bmatrix} -\beta_s \\ -\beta_t \\ 1 \end{bmatrix} \\ &= \vec{n} + \frac{(\vec{n} \times \sigma_t)\beta_s + (\sigma_s \times \vec{n})\beta_t}{\vec{n} \bullet (\sigma_s \times \sigma_t)} \qquad (1) \end{aligned}$$

where the symbol $\bullet$ denotes the dot product. Furthermore, the denominator in the second term of equation (1) is equal to

$$\begin{aligned} \det(N) &= \vec{n} \bullet (\sigma_s \times \sigma_t) \\ &= \pm\|\sigma_s \times \sigma_t\| \end{aligned}$$

and is positive when the parametrization is orientation preserving. The formulation given here is almost identical to that of Jim Blinn except that it is not an assumption here that $\det(N)$ is positive.

In the following section we will give a formulation of bump mapping which does not require a user–defined parametrization of S or β.

3 The Surface Gradient Based Formulation

The height function β represents a scalar field on the surface S. In vector calculus, the gradient of a scalar field is a vector field which points in the direction of the greatest rate of increase of the scalar field, and whose magnitude is the greatest rate of change.

The surface gradient ∇_S of a scalar field is tangent to S. For any point p, on S, if there exists some local extension of β which is a smooth function on some open subset of $\mathbb{R}^3$, containing p, then the surface gradient at p is equal to

$$\nabla_S \beta = \nabla \beta - \vec{n} \cdot (\vec{n} \bullet \nabla \beta) \qquad (2)$$

which is the projection of the regular gradient onto the tangent plane. For a known chart σ, containing p, the surface gradient is equal to

$$\begin{aligned} \nabla_S \beta &= \begin{bmatrix} \beta_s & \beta_t & 0 \end{bmatrix} \cdot N^{-1} \\ &= \frac{(\sigma_t \times \vec{n})\beta_s + (\vec{n} \times \sigma_s)\beta_t}{\vec{n} \bullet (\sigma_s \times \sigma_t)} \qquad (3) \end{aligned}$$

The surface gradient $\nabla_S \beta$ has the two properties that it is a vector in the tangent plane and that the dot product between it and some given unit length tangent vector $\vec{v}$ gives the rate of change in the direction of $\vec{v}$. Equation (3) clearly obeys the former and the latter is true since $\nabla_S \beta \bullet \vec{v} = a \cdot \beta_s + b \cdot \beta_t$ for $\vec{v} = a\sigma_s + b\sigma_t$ where $a, b \in \mathbb{R}$.

From equations (1) and (3) it follows that the perturbed normal can be expressed as

$$\vec{n}' = \vec{n} - \nabla_S \beta \qquad (4)$$

Intuitively this means the normal is pulled away from the tangent direction of the greatest rate of increase in β. Interestingly the result is identical to Blinn's perturbed normal, for a known surface parametrization, as indicated by equation (3).

The form given by equation (4) indicates that the perturbed normal does not depend on a specific parametrization. It depends on the shape of the surface and the distribution of height values. Thus we can evaluate the surface gradient $\nabla_S \beta$ using some, per pixel chosen, local parametrization. For our solution we choose inverse projection from the screen domain $M \subset \mathbb{R}^2$ to the surface S. Inverse projection does not represent a valid chart for the entire surface since a coordinate $(s_0, t_0) \in M$ can map to multiple points on S. However, the map represents a valid local parametrization σ at each intersection point. In other words the intersection point is contained in some small neighborhood on S which does not overlap itself if projected onto the screen. It is then a local diffeomorphism. For further details the reader is referred to the definition of a manifold and also the inverse function theorem [Pre01].

In the next section we will discuss how this math, equation (3) in particular, translates to a shader for the GPU.

4 The Implementation

A surface is a two dimensional manifold which means on a small scale a neighborhood, containing $p \in S$, can be perceived as planar. In a sense we inherit the differential structure associated with $\mathbb{R}^2$. On the GPU a surface is rendered as some, arbitrarily dense, triangular representation which complements this concept. A triangle does not overlap itself after projection unless the projection is a line segment or a point in which case it is rejected by the GPU. Thus inverse projection from the screen to the triangle is a valid parametrization. We denote this $\sigma : (s, t) \to p$ where the coordinate of the pixel center is (s, t) and the corresponding point on the triangle is p. This function is resolved simply by passing the vertex position to the interpolator.

To compute the surface gradient by equation (3) we must produce the derivatives σ_s, σ_t, β_s and β_t. The screen–space derivative calculation functions provided by the GPU are known as `ddx_fine` and `ddy_fine`. On the current generation these produce the same result for pixel pairs in blocks of 2×1 and 1×2 respectively. Better results would be achieved by using central differencing but in order to keep the solution simple and effective we rely on the built–in calculation. The implementation is shown in listing 1. For those primarily seeking to use this technique with traditional bump maps a better quality is achieved by replacing the height derivative evaluation in listing 1 with, for instance, forward differencing as is done in listing 2. Three independent taps are used to achieve this and cache coherence is not an issue given the locality of these taps. Similarly, central differencing may be achieved using four taps. It is also worth noting that the derivative operator is linear so it is possible to use manual differencing for texture maps and not for procedural functions but still perturb the normal only once.

A triangular mesh is piecewise flat. However, it is common in computer graphics to use averaged normals at the vertices which are, subsequently, inter-

```
float3 PerturbNormal(float3 surf_pos,
    float3 surf_norm, float height)
{
  float3 vSigmaS = ddx( surf_pos );
  float3 vSigmaT = ddy( surf_pos );
  float3 vN = surf_norm;     // normalized

  float3 vR1 = cross(vSigmaT,vN);
  float3 vR2 = cross(vN,vSigmaS);

  float fDet = dot(vSigmaS, vR1);

  float dBs = ddx_fine( height );
  float dBt = ddy_fine( height );

  float3 vSurfGrad =
    sign(fDet) * ( dBs * vR1 + dBt * vR2 );
  return normalize(abs(fDet)*vN-vSurfGrad);
}
```

Listing 1: Perturbed normal calculation without the use of texture coordinates.

```
  float2 TexDx = ddx(In.texST);
  float2 TexDy = ddy(In.texST);
  float2 STll = In.texST;
  float2 STlr = In.texST + TexDx;
  float2 STul = In.texST + TexDy;
  float Hll = bmap.Sample(sampler, STll).x;
  float Hlr = bmap.Sample(sampler, STlr).x;
  float Hul = bmap.Sample(sampler, STul).x;

  float dBs = Hlr-Hll;
  float dBt = Hul-Hll;
```

Listing 2: Screen–space height derivative evaluation by forward differencing.

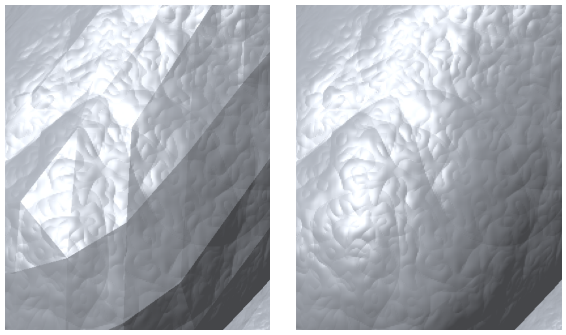

(a) flat (b) smooth

Figure 1: In figure 1(a) we see hard edges as a result of using $\tau_s \times \tau_t$ to produce the normal. Using the method in listing 1 perturbs the interpolated vertex normal as shown in figure 1(b).

polated across the triangle to produce a look which is more soft. We use the same normalized interpolation of the vertex normals as the input parameter `surf_norm`. If the vertex normals are not known then the normal of the triangle can be obtained using `cross(vSigmaS, vSigmaT)` instead.

For completeness we point out that there exists an even simpler approach to perturb the normal. This is the cross product between the first order derivatives of the displaced surface position $\tau_s \times \tau_t$. However, this will perturb the face normal and not the interpolated vertex normal. A comparison between this approach and our solution is shown in figure 1.

The method which we have discussed until now relies on the availability of the shader functions `ddx_fine` and `ddy_fine`. This issue will be discussed in the next section.

5 Pre and Post–Process Bump Mapping

The shader function given by listing 1 can only be used in a pixel shader during the in–process fixed function rasterization of triangles. This excludes bump mapping in a pre–process such as a vertex shader or any post–processing solution such as deferred rendering. Though the method in listing 1 supports spatial height maps we can, for such a map, achieve the same normal perturbation without it.

```
float3 PerturbNormal(float3 surf_norm,
    float3 vGrad)
{
  float3 vSurfGrad =
    vGrad - surf_norm*dot(surf_norm, vGrad);
  return normalize( surf_norm - vSurfGrad);
}
```

Listing 3: This is the surface gradient based normal perturbation using a spatial height map.

It was proposed by Perlin [Per85] that a normal perturbing effect can be achieved by adding the gradient, of a spatial texture, to the unit normal $\vec{n}$. However, this does not produce the normal associated with the displaced surface. In this paper we have proved that Jim Blinn's formulation of the perturbed normal is equivalent to equation (4). Our implementation is shown in listing 3. The function takes as input the gradient of the spatial texture $\nabla\beta$. For some functions the gradient can be determined analytically which is the case for Perlin noise. When this is not the case then numerical approximation is

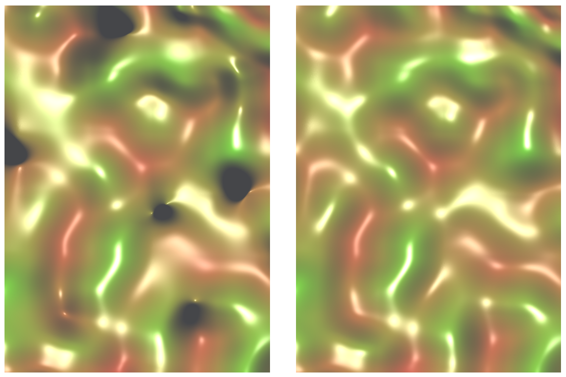
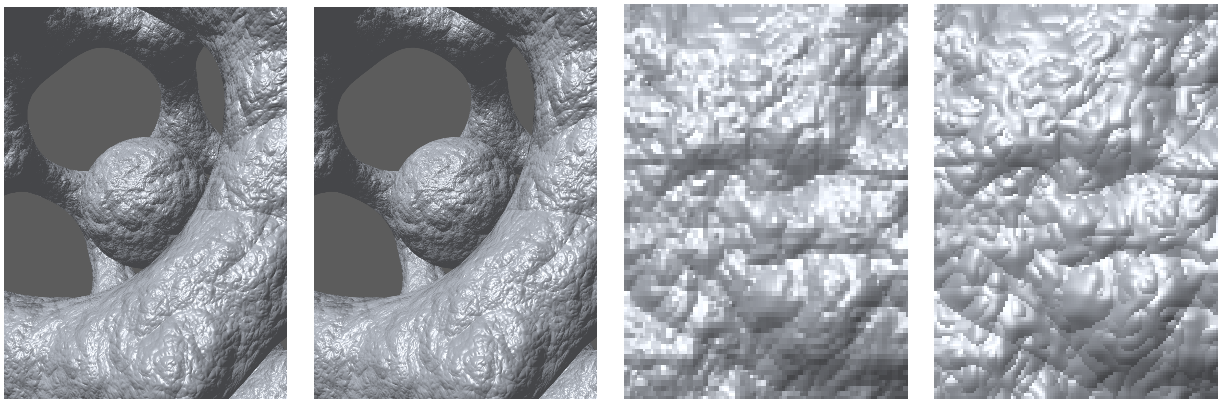

(a) Perlin's perturbation method (b) Surface gradient based

Figure 2: The result of Perlin based normal perturbation is shown in figure 2(a). There are visual artifacts which are gone using our method shown in figure 2(b).



(a) method in listing 1 (b) method in listing 3 (c) image cut out 1 (d) image cut out 3

Figure 3: Figures 3(a) and 3(b) confirm that the same results are produced by the surface gradient method and listing 1 which is our **ddx_fine** and **ddy_fine** based implementation. However, looking closer at the images as shown in figures 3(c) and 3(d) shows that the quality of derivative calculation using the function in listing 1 depends on pixel density.

also a possibility. For instance forward differencing is used by Perlin in [Per04].

The difference between Perlin's perturbation method and equation (4) is visually significant. If the gradient is above the tangent plane the perturbation impact is reduced and results are understated. If the gradient is below the tangent plane the perturbation impact is increased. This increase eventually results in the perturbed normal pointing inward. A visual is shown in figure 2(a) based on a low frequency Perlin noise for the height map. The negated gradient was used to produce this picture since adding it effectively inverts the height map. Using the function in listing 3 gives the artifact free result shown in figure 2(b).

6 Results

The results presented in this section were produced using an implementation in Direct3D 11 using the hlsl shading language. Our first scene is a triangulated isosurface with a single point light. We assign a height to each surface point using the spatial function `turbulence()` by Ken Perlin [Per85]. Six octaves are used and the Perlin noise function is done with the implementation by Simon Green in [Gre05]. Note that the gradient of Perlin noise has an analytical solution `dnoise()` which can easily be implemented in a shader. This implies that the resulting perturbed normal has an analytical solution using equations (2) and (4). However, the function `turbulence()` accumulates numerical values of noise so we use central differencing to approximate the gradient. The result is shown in figure 3(b) and the corresponding result in figure 3(a) using the function in listing 1. The results are the same which confirms the analysis in section 3. However, at closer inspection there are differences in terms of quality since the derivative calculation using listing 1 depends on the pixel density across the surface relative to the frequency level of the height function. As an example a small part of the sphere in figures 3(a)–3(b) is shown in figures 3(c)–3(d) respectively. Note that these correspond to zooming in on the images. In contrast approaching the sphere increases pixel density and gives similar results using these methods. Increasing the resolution or super–sampling will also achieve this.

The following test compares results between using the function in listing 1 and normal mapping using interpolation of per vertex tangent spaces. The model used is a treasure item from the game "Uncharted 2: Among Thieves". Again results are the same as shown in figures 4(a) and 4(b). However, there is a difference since derivatives are computed in screen–space from a reconstruction of the height function. This reconstruction is done using the built–in texture sampler which produces a signal of relatively low accuracy. For instance when the texture map is magnified due to undersampling the reconstruction of heights is reduced to a piecewise linear function. The derivative function of such a signal is a piecewise constant function. The result is illustrated as a close up of the model in figure 4(e).

A possible solution is of course to use higher order filtering but this makes the solution significantly more expensive. We propose either accepting the limitation of the method or hiding the problem using a well known solution which is detail maps. Figure 4(f) shows the result of using `turbulence()` for detail.

7 Conclusion

In this paper we have shown how bump mapping can be done on the GPU using the following inputs only in the perturbation function: surface position, surface normal and the height value.

The solution works without precomputed tangent vectors and in fact does not even require that the height value is produced from a texture coordinate. The height function can be either spatial or defined on the 2D domain. It can be analytical/procedural or represented by a texture map. Thus our method presents a uniform solution to determine the perturbed normal.

A limitation exists due to our use of the screen–space derivative instructions `ddx_fine` and `ddy_fine`. These are only available when using the built–in rasterization pipeline of the GPU which is restricted to triangles. This excludes certain use cases such as perturbation of the normal during a deferred process. However, in this paper we have shown that Jim Blinn's formulation of the perturbed normal is equivalent to subtracting the surface gradient of the height function from the unit normal. Thus if the height function is spatial, such as Perlin noise, the normal can be perturbed without the use of `ddx_fine` and `ddy_fine`. This is particularly efficient in the case of Perlin noise because it has an analytical gradient function. When this is not the case the gradient can be approximated numerically.

The primary limitation is that bump mapping in the general case is only supported during in–process rendering of triangles. For future work it would be interesting to see if we can achieve support for two dimensional height maps in the vertex shader and/or achieve a uniform solution in post process.

Acknowledgments. This author would like to thank Bryan Mcnett for helpful discussions, Manchor Ko, Pal–Kristian Engstad, and Swaminathan Narayanan for constructive comments and proof reading. Finally, thank you to Naughty Dog for allowing me to use the treasure object for this paper.

References

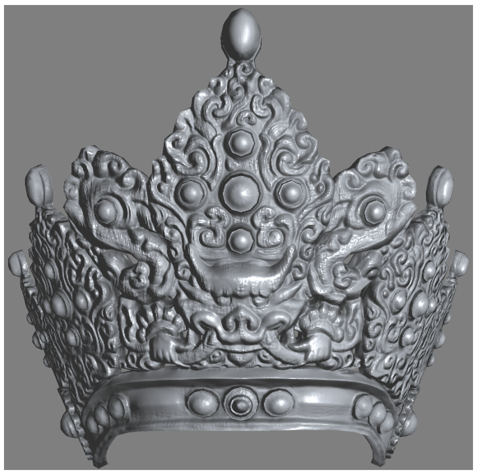
[Bli78] BLINN J.: Simulation of wrinkled surfaces. In *ACM Computer Graphics (SIGGRAPH '78)* (1978), pp. 286–292.

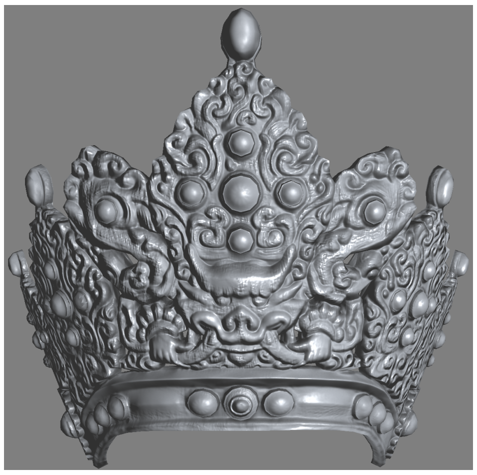
[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The reyes image rendering architecture. In *ACM Computer Graphics (SIGGRAPH '87)* (1987), ACM, pp. 95–102.

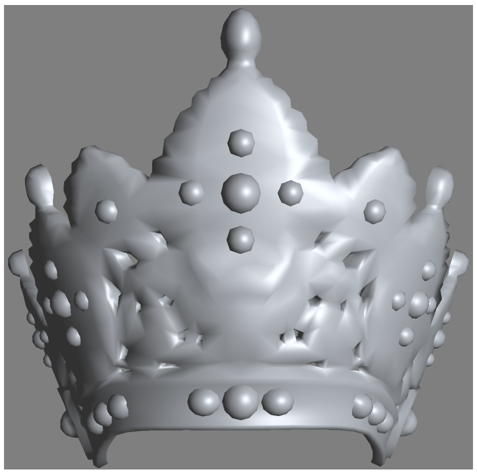
[Gre05] GREEN S.: *GPU Gems 2 – Programming Techniques for High–Performance Graphics and General–Purpose Computation"*. Addison–Wesley Publishing, 2005, ch. 26, pp. 409–416.

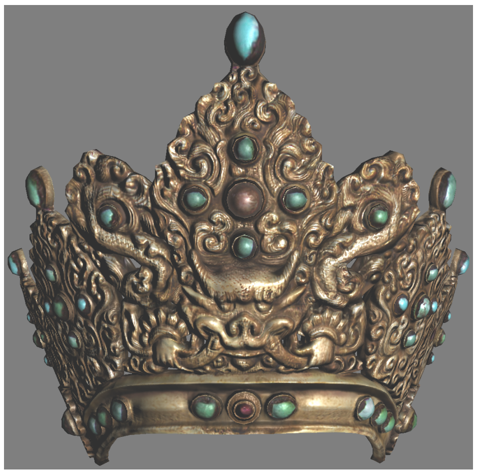
[KTI*01] KANEKO T., TAKAHEI T., INAMI M., KAWAKAMI N., YANAGIDA Y., MAEDA T., TACHI S.: Detailed shape representation with parallax mapping. In *Proceedings of the ICAT 2001* (2001), pp. 205–208.

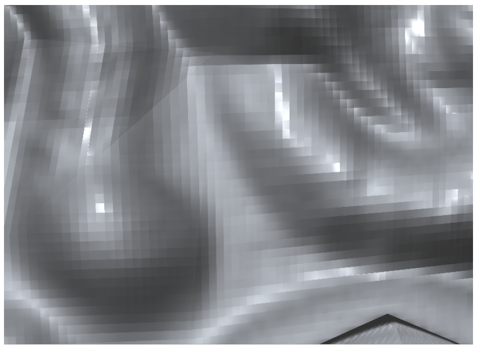
[Mik08] MIKKELSEN M. S.: *Simulation of Wrinkled Surfaces Revisited*. Master's thesis, Department of Computer Science at the University of Copenhagen, 2008.


(a) method in listing 1


(b) normal mapping


(c) no perturbation


(d) method in listing 1 with albedo


(e) undersampling

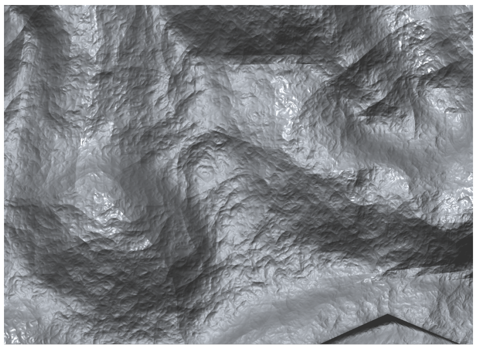

(f) undersampling with detail

Figure 4: Our listing 1 method gives results which are very similar to that of normal mapping as shown in figures 4(a)-4(b). The model is shown without perturbation of normals, in figure 4(c), to illustrate which elements of the detail are in the height map. The final result is shown in figure 4(d) with the albedo map applied. When the height texture map is magnified the surface becomes piecewise flat as shown in figure 4(e). The shortcoming can be hidden by using a detail map which is shown in figure 4(f).

[OBM00] Oliveira M. M., Bishop G., McAllister D.: Relief texture mapping. In *ACM Computer Graphics (SIGGRAPH '00)* (2000), Addison–Wesley Publishing, pp. 359–368.

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

[Per04] Perlin K.: *GPU Gems – Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison–Wesley Publishing, 2004, ch. 5, pp. 409–416.

[Pre01] Pressley A.: *Elementary Differential Geometry*. Springer, 2001.

[Sch06] Schuler C.: Normal mapping without precomputed tangents. In *ShaderX5: Advanced Rendering Techniques*, Engel W., (Ed.). Charles River Media, 2006, pp. 131–140.