Direct Ray Tracing of Smoothed and Displacement Mapped Triangles

Brian Smits Peter Shirley Michael M. Stark
University of Utah
Technical Report UUCS-00-008
www.cs.utah.edu/ bes/papers/height/tech.html

Abstract.
We present an algorithm for ray tracing displacement maps that requires no additional storage over the base model. Displacement maps are rarely used in ray tracing due to the cost associated with storing and intersecting the displaced geometry. This is unfortunate because displacement maps allow the addition of large amounts of geometric complexity into models. Our method works for models composed of triangles with normals at the vertices. In addition, we present a special purpose displacement that creates a smooth surface that interpolates the triangle vertices and normals. Thus two adjacent triangles which share two vertices and normals will be smoothly interpolated. This displacement can be added to the displacement associated with the object. The combination allows relatively coarse models to be displacement mapped and ray traced with much less storage and fewer artifacts due to tessellation.

1 Introduction

Visually rich images are often generated from simpler models by applying *displacement maps* to increase surface detail (Figure 1). Displacement maps are a special type of *offset surface*, and are usually assumed to perturb surface positions a small distance using some function. Images with displacement maps are usually computed using explicit subdivision [3]. The displacement is often a semi-random procedural function that uses Perlin-style noise [11]. Somewhat surprisingly, displacement maps are almost never used in ray tracing. This turns out to be for entirely technical reasons; a straightforward implementation would need to store more micropolygons than would fit in main memory on most computers [4]. For this reason, sophisticated caching strategies have been suggested [12]. Although caching strategies work well for a variety of applications they are problematic for applications that resist reordering such as Metropolis Light Transport [15]. Alternatively, explicit numeric root-finding can be used, provided the displacements can be nicely bounded [5, 8]. A third approach that could work for displacement mapped surfaces is the recursive subdivision scheme used for procedural geometry by Kajiya[6]. This approach requires knowing tight bounds over each subdivided region of the displacement function in order to be efficient. Because most global illumination algorithms require ray tracing, it is desirable to find a simple way to add displacement maps to ray tracing programs. This would allow realism in both global lighting complexity and local geometric complexity.

We introduce a method for ray tracing polygonal models with displacements that avoids complex strategies by restricting the allowable base geometry to triangle meshes with vertex normals. Although this is a narrow class of modeling primitive, almost all other modeling primitives can be converted to triangle meshes in a practical manner.

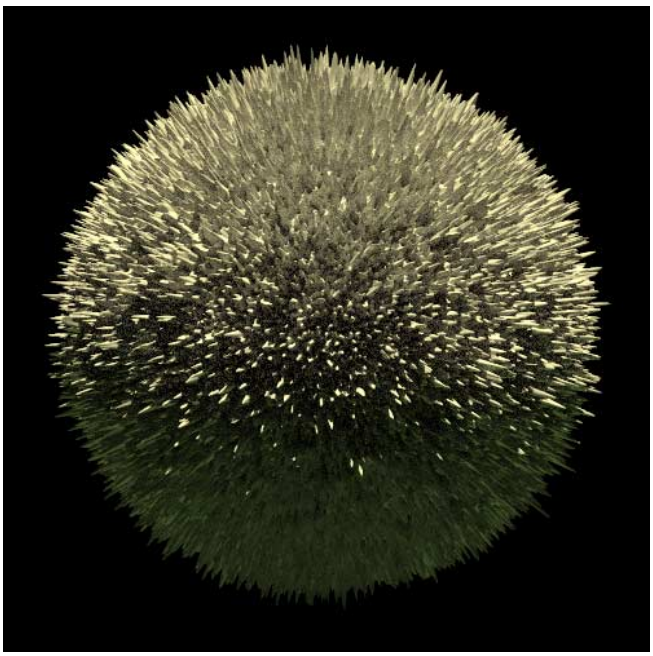

Fig. 1. *An image of a complex object created by displacement mapping an icosahedron. The figure is ray traced with global illumination. Only twenty triangles are stored.*

The key problem with triangle meshes is the well-known faceting artifacts. However, we show how to use a deterministic spline displacement function to smooth tessellated models. While we have restricted how our base models must be represented, we feel the resulting benefits in computation and storage make up for this restriction.

In Section 2 we give an overview of our assumptions on the model and the restrictions we impose for our algorithm. In Section 3 we present the ray intersection algorithm for triangles with displacement functions. The requirements for a displacement function used to smooth triangle meshes is discussed in Section 4. Images resulting from the algorithm are shown in Section 5. Finally, we discuss future directions for the work in Section 6.

2 Overview

The inspiration for our method comes from the *REYES* rendering architecture [3]. That simple architecture has worked well for almost two decades, and relies on three simplifying assumptions related to displacements:

- displacements are bounded in distance,
- base surfaces know how to subdivide themselves,
- subdividing the displaced base surfaces into a net of simple sub-pixel patches provides sufficient accuracy.

We borrow these assumptions directly. By assuming that a finely subdivided model provides sufficient accuracy, we can use micropolygon normals directly, so no deriva-

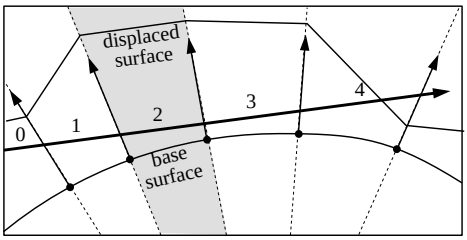

Fig. 2. *A set of points with normals partitions space into cells (one is shaded) which can be traversed in order by a ray. This observation holds in 3D as well. An analogous partition can be added within each cell.*

tive properties of the displacement need be known. We also add the assumption that the displacements are along the direction of the interpolated normal. Although this is more restrictive than the displacement mapping found in the REYES architecture, it is the type of displacement mapping found in Maya[1]. For the intersection method, first imagine a base surface being "carved up" with a set of vertices and normals (Figure 2). Within each partition we could displace a triangle whose vertices lie along projected normal vectors from the base surface. If one considers a given triangle under all possible displacements, it sweeps out a 3D region in space. For reasonably well-behaved surfaces, adjacent triangles have adjacent regions. The shape of the boundaries between these regions depends on how the normal vectors of base geometry behave. If one imagines all the regions swept out by all triangles, each triangle forming a "column" in space, the possibility of a traversal algorithm presents itself. If the base geometry is a plane then all displacements are perpendicular to the plane and the traversal algorithm would be similar to that usually used for ray intersections with height fields [9], except that the traversed cells would have triangular rather than rectangular cross-sections. We would like to choose a base geometry that is general enough to be geometrically expressive, but restrictive enough that such a traversal algorithm is feasible.

Because they are so often used in practice, three obvious choices are NURBS surfaces, subdivision surfaces, and implicit surfaces such as metaballs. Since all three of these primitive types are quite different from each other, it is desirable to find a common representation that they could all be converted into. The only obvious choice for this common representation is a triangulated mesh, to which it is straightforward to convert for NURBS and subdivision surfaces, and at least feasible for implicit surfaces [14]. For this reason we choose triangles as our base geometry. To ensure that the displaced surface is continuous, we use shared vertex normals and displace along normals computed via barycentric interpolation (i.e., Phong normal interpolation [13]). Although more general displacements are useful [10], we leverage this restriction on the direction of displacement to create a simpler algorithm than would be possible otherwise.

We strengthen the restriction of a bound on the displacement to limit the range of possible displacements so that any resulting displaced surface is unable to intersect itself. Each point in the valid region corresponds to exactly one position and displacement value on the base triangle. This restriction means that each region has only one set of neighbors, another requirement for a simple traversal algorithm. It also means that the first intersection found will be the closest intersection to the ray origin.

Our displacement framework assumes there is a point $\mathbf{p}$ on an underlying surface which is displaced in the direction of the normal vector $\mathbf{n}(\mathbf{p})$ by a displacement function $h(\mathbf{p})$ (Figure 3). For a triangle with points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ and corresponding normals

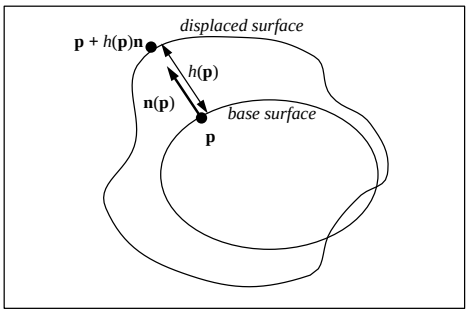

Fig. 3. *A simple displacement by function h in the normal direction creates a new curve in 2D.*

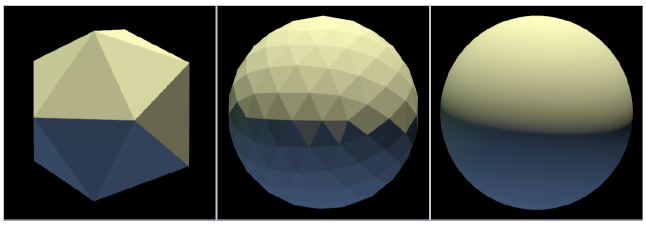

Fig. 4. *Icosahedron with displacement pushing each point to a sphere, $N = 1, 4, 100$.*

$\mathbf{n}_0, \mathbf{n}_1, \mathbf{n}_2$ the bilinearly interpolated points and normals $\mathbf{p}$ and $\mathbf{n}$ are:

$$\begin{aligned} \mathbf{p} &= \alpha\mathbf{p}_0 + \beta\mathbf{p}_1 + \gamma\mathbf{p}_2, \\ \mathbf{n} &= \alpha\mathbf{n}_0 + \beta\mathbf{n}_1 + \gamma\mathbf{n}_2, \end{aligned}$$

where (α, β, γ) are the *barycentric coordinates* on the triangle, so $\alpha + \beta + \gamma = 1$. Our displaced surface $\mathbf{p}_d$ is thus:

$$\mathbf{p}_d = \alpha\mathbf{p}_0 + \beta\mathbf{p}_1 + \gamma\mathbf{p}_2 + h(\alpha\mathbf{p}_0 + \beta\mathbf{p}_1 + \gamma\mathbf{p}_2)\,(\alpha\mathbf{n}_0 + \beta\mathbf{n}_1 + \gamma\mathbf{n}_2)$$

3 Ray Intersection

Our ray intersection test is similar in spirit to intersecting a ray with a height field using a regular grid over the base plane. We will take advantage of an implicit triangular grid formed by the barycentric coordinates. We choose a subdivision amount N (Figure 4) and use dividing lines $\alpha_i = \gamma_i = \beta_i = i/N$ for $i = 0, ..., N$ which creates N^2 grid cells for each triangle. Each grid cell generates one displaced microtriangle, as shown in Figure 5. The grid is regular on the base triangle, but due to the interpolated surface normals, it is irregular throughout space. Although it is irregular, our restrictions limit the range of the displacement function $h()$ to the interval $[-m, +M]$ where a traversal algorithm is possible.

Much like standard grid traversal algorithms, there are two phases to the algorithm. First the start point must be initialized. Next the grid must be traversed, checking each cell for an intersection with the triangle it contains. The traversal algorithm will be described first in order to determine the quantities that need to be initialized.

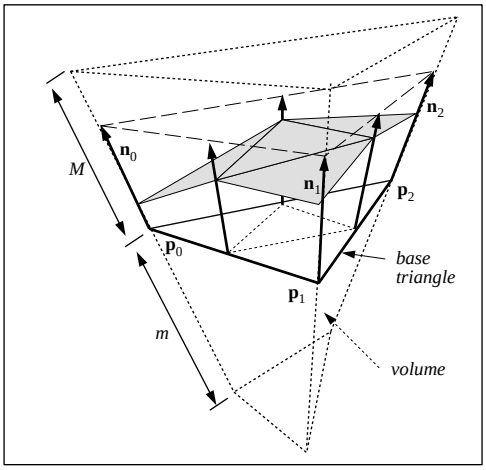

Fig. 5. *The base triangle and four displaced microtriangles generated by setting the subdivision parameter, N, to 2. The volume for the maximum displacement is also shown.*

3.1 Traversal

Assuming we will be able to initialize the traversal algorithm, we focus first on how to do an efficient traversal. This traversal is conceptually simple, however the use of triangles complicates the indexing. For each cell entered, the microtriangle is generated. If it is hit, the traversal is over, if it is missed, the next cell must be determined and a new microtriangle generated. The new triangle will differ from the previous triangle by exactly one vertex. This means that for each step through the grid we need only evaluate the expensive displacement function once.

A position in the grid will be labeled by a triple, (i, j, k), corresponding to the lines of constant barycentric coordinates $\alpha = i/N, \beta = j/N, \gamma = k/N$. The indices sum to either $N - 1$ or $N - 2$ depending upon whether the triangle is a *lower* triangle or an *upper* triangle as shown in Figure 6. The classification into lower and upper determines how the vertices are generated given the indices. For a lower triangle, the barycentric lines corresponding to indices are the edges of the triangle. For an upper triangle, the barycentric lines corresponding to the indices touch the triangle only at the vertices. This is not as neat as other possible numbering schemes, however it means that each triangle differs from its neighbors by one in exactly one index.

Each microtriangle is represented by three displaced points, **a**, **b**, and **c**, with the order chosen such that the ray is assumed to have entered the cell passing through the side corresponding to edge $\mathbf{a}, \mathbf{b}$. The next cell to be tested can be marked based on which index will change and if the index will be incremented or decremented. This flag can be represented as {iplus, jminus, kplus, iminus, jplus, kminus}, and depends upon the orientation of the current triangle and which side of the cell the ray exits through. By knowing how the ray entered the current cell, there are usually only two options for how the ray leaves the cell. The exception for when the ray exits through the face it enters is handled by the initialization code and will be discussed later. These options can be checked by seeing on which side of the line determined by $\mathbf{c} + s\mathbf{n}_c$ (the far point and its normal) the ray passes, as shown in Figure 7. If the above list of choices is viewed as a ring, the next possible choice is either the next flag in the ring, or the

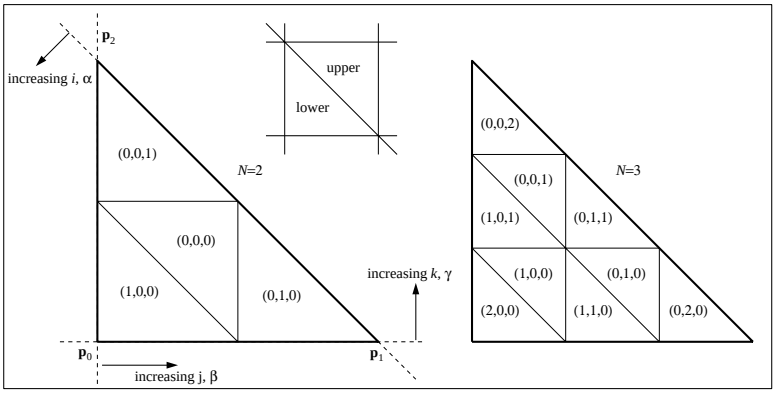

Fig. 6. *Barycentric indexing for $N = 2$ and $N = 3$. When moving between adjacent triangles, exactly one index changes by one. For a given triangle, this change has the same sign for all three edges. The "upper" triangle for a given (j, k) is the one with the smaller i index.*

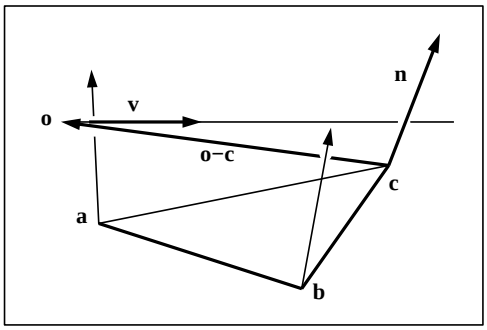

Fig. 7. *The ray $\mathbf{o} + t\mathbf{v}$ passes between the normals at $\mathbf{a}$ and $\mathbf{b}$. It will leave either between the normals at $\mathbf{a}$ and $\mathbf{c}$, or between the normals at $\mathbf{b}$ and $\mathbf{c}$. This can be tested by whether the ray passes left or right of $\mathbf{n}$. It goes to the left of the line $\mathbf{c} + t\mathbf{n}$ if $\mathbf{v} \cdot (\mathbf{n} \times (\mathbf{o} - \mathbf{c}))$ is negative.*

previous flag in the ring.

The traversal can be terminated by checking if the (i, j, k) values are the same as the stop cell (i_e, j_e, k_e) determined by the initialization phase. We also terminate the traversal if the ray exits the volume. The traversal loop can be expressed in pseudocode as follows:

```
Ray ray       // ray, including valid interval for t
Vector3 a,b,c       // microtriangle vertices, ordered
Vector2 uva,uvb,uvc       // (β,γ) for each vertex
Vector3 cNormal       // normal at vertex c
int i, j, k       // indices of current cell
bool rightOfC       // flag used to determine next cell
LastChange change       // where change is one of:
                  // {iplus, jminus, kplus, iminus, jplus,kminus}
float delta = 1 / N
while(true)
  if TriangleIntersect(ray, a, b, c)
    intersectionNormal = (b-a) × (c-a)
    return true
  if EndCell(i,j,k) return false
  rightOfC = ((cNormal × (ray.Origin() - c)) * ray.Direction() > 0)
  if(rightOfC)
    a = c,    uva = uvc
  else
    b = c,    uvb = uvc
      // Take advantage of numbering. 5 = −1 mod 6
  change = AdvanceType((change + (rightOfC ? 1 : 5)) % 6)
  if(change == iminus)
    if (−− i < 0) return false
    uvc = Vector2((j+1)*delta, (k+1)*delta)
  else if(change == iplus)
    if (++ i ≥ N) return false
    uvc = Vector2( j*delta, k*delta)
  else if(change == jminus
    if (−− j < 0) return false
    uvc = Vector2( j*delta, (k+1)*delta)
  else if(change == jplus)
    if (++ j ≥ N) return false
    uvc = Vector2( (j+1)*delta, k*delta)
  else if(change == kminus
    if (−− k < 0) return false
    uvc = Vector2( (j+1)*delta, k*delta)
  else if(change == kplus)
    if (++ k ≥ N) return false
    uvc = Vector2( j*delta, (k+1)*delta)
  (c,cNormal) = GetPoint(uvc)
```

3.2 Initialization

The initialization phase of the algorithm must determine where in the grid the traversal algorithm starts and ends. The volume through which the traversal takes place is shown

in Figure 5. The top and bottom of the space are bounded by triangles, the sides are bounded by bilinear patches.

Before the start and end cells are determined, the subdivision amount N must be found. This can either be fixed for the displacement map, $N = C$, or made adaptive, based on projected screen area. We allow either, and compute the adaptive size based on the area and an estimate of the distance to the camera, with a user defined $N_{\max}$.

The initialization phase must determine the correct index (i, j, k) for starting the traversal. In standard grid traversal algorithms the traversal may start anywhere inside the grid. This clearly makes sense and would be ideal, but determining the index given an arbitrary point is equivalent to determining the barycentric coordinates and displacement (height) for the point. The computation involves solving a cubic equation, and the method seemed to have numeric problems. Our solution is to treat the ray as an infinite line and find the place where that line enters the volume and where it exits. This can require a longer traversal than necessary, however unlike uniform space subdivision in ray tracing, where the grid bounds the environment or a complex object, the displaced triangle tends to occupy a relatively small fraction of the scene, so most rays will pass completely through the volumes of most triangles.

The start and end points are the smallest and largest intersections of the ray with the volume. If the intersection point is on one of the bilinear side patches, one of the barycentric coordinates is zero, and the u parametric value found while intersecting the side can be used directly to determine the other two. If the intersection point is on one of the triangular end caps, the barycentric coordinates of the intersection point are exactly what is needed. The index for the grid cell is then $(\lfloor \alpha * N \rfloor, \lfloor \beta * N \rfloor, \lfloor \gamma * N \rfloor)$.

The last part of the initialization is to determine which face of the cell the ray entered from, so that the traversal algorithm can determine the appropriate next cell. This is given if the intersection is on one of the bilinear sides, however it is not given for the top or bottom boundaries. In this case, the bilinear walls of the cell can be checked. As the ray entered either the top or the bottom, the side hit will be the side the ray leaves from. It is valid to assume the ray entered from either of the other two sides. If the ray does not hit any sides, then this cell is the end cell as well, so the parameter does not matter.

3.3 Complications

There are some complications created in using a traversal algorithm to walk through an irregular volume filled with many small triangles. The first and most significant is that the sides of the volume are not planar and the ray may intersect one twice. This means that the traversal may exit the grid without reaching the correct stop cell. More importantly, the intersection with the surface may lie in the second interval within the volume. The initialization code can be modified so that if the ray hits one of the bilinear sides twice, and no intersection is found in the first interval, then the traversal is called again with a new start cell determined by the second intersection point.

A second complication occurs due to the sides of the grid cells being non-planar. The first way this could cause problems was briefly mentioned while discussing the traversal. Geometrically, the ray can enter a cell briefly, and then quickly return to the first cell. This does not happen in our algorithm because of the way the traversal chooses the next cell; the ray passes on the same side of both point-normal pairs for that side, so the ray never enters the cell. In terms of Figure 7, although the ray could possibly intersect the bilinear patch along edge **bc** twice, our algorithm ignores the double intersection and chooses the cell on the other side of edge **ac**. For certain extreme

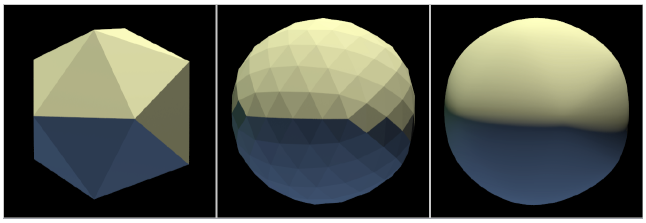

Fig. 8. *An icosahedron with a smoothing displacement that only uses the vertices and vertex normals for the triangle being displaced for* $N = 1, 4, 100$.

configurations, it is possible that the ray may actually intersect the microtriangle in the missed cell. Because the cells in general do not exactly bound the microtriangles, it is possible the ray should have hit the neighbor's triangle even if the ray misses the bilinear wall of the cell. Due to the small size of the microtriangles, and the significantly smaller size of the potentially missed piece, we have not noticed any significant errors caused by this problem. One solution would be to grow the triangle slightly in the triangle intersection test, a solution sometimes used to prevent cracking in simple triangle meshes. We chose not to do this because in our experience, expanding geometry eventually causes it's own set of problems.

A final issue to consider is that this method has the potential to create very small triangles. Some of the standard triangle intersection tests use epsilons that may be not be suitable for the size of the input. This can cause microtriangles to be falsely missed.

The intersection algorithm is implemented entirely using four byte floats. Although there are occasional rays that miss the surface, these problems are about the same frequency as those often found in ray tracers using simple polygonal objects.

4 A Smoothing Displacement Function

Since we have a mechanism to create images with displacements, it is useful to have a displacement that creates a smooth mesh. This would allow rendering smoothed versions of tessellated models with or without additional displacements. To make the problem as local as possible, we assume the smoothing displacement only has knowledge of a given triangle's vertices and vertex normals. Knowledge about neighboring triangles would allow a smoother surface, but we leave that as future work. Our goal is to create a simple smoothing displacement as a proof of concept. Although examining how to smooth triangle meshes has been examined by many researchers (e.g., [7]), our problem is different in that our function must have the algebraic form of a height function in barycentric coordinates with respect to barycentric interpolated normals.

We would like the displacement to interpolate the triangle vertices, and have a smooth tangent plane on the transition between two adjacent triangles. This implies a number of constraints:

- the surface must depend only on the vertices and vertex normals,
- the surface must be smooth over the triangle,
- the surface must interpolate the vertices of the triangle,
- the surface normal at each vertex must match the prescribed vertex normals,
- the tangent plane along each edge of the surface must match that constructed on an adjacent triangle, so that joined patches meet with G^1 continuity.

The final requirement listed above is the one which is the most difficult to satisfy, because Hermite (derivative) interpolation is more difficult to enforce over a line than at single points. We use the Coons patch approach to construct our surface. First, boundary curves and prescribed tangent planes are constructed using ordinary Hermite interpolation. Then we use transfinite interpolation to construct three surfaces which interpolate the boundary curves and tangents along two of the edges. These three surfaces are blended in such a way as to preserve the derivatives and remove the "bad" edges from the final surface. The surface will be constructed in terms of barycentric coordinates. The approach applied to an icosahedron is shown in Figure 8.

Note that in our entire discussion the vertex normals are assumed to be outward facing and unit-length. However, the interpolated normals are not necessarily unit-length, i.e. they are not automatically renormalized.

4.1 Surface Construction

The general problem of determining the boundary curves and the surface curves are special cases of the following interpolation problem: given two points $\mathbf{p}_0$ and $\mathbf{p}_1$, as well as associated normals $\mathbf{n}_0$ and $\mathbf{n}_1$, find a function of the form

$$\mathbf{p}(t) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1 + g(t)\left[(1-t)\mathbf{n}_0 + t\mathbf{n}_1\right].$$

which interpolates surface points $\mathbf{p}_0$ and $\mathbf{p}_1$ with tangent normal to $\mathbf{n}_0$ and $\mathbf{n}_1$ at the endpoints. If $\mathbf{p}(t)$ is to interpolate the endpoints, then $g(0) = g(1) = 0$. We compute the required values for $g'(t)$ at the endpoints and apply Hermite interpolation.

Derivative interpolation requires that the tangents $\mathbf{p}'(0)$ and $\mathbf{p}'(1)$ lie in the plane normal to the $\mathbf{n}_0$ and $\mathbf{n}_1$, respectively, which amounts to requiring $\mathbf{p}'(0) \cdot \mathbf{n}_0 = 0$ and $\mathbf{p}'(1) \cdot \mathbf{n}_1 = 0$. We have

$$\mathbf{p}'(t) = \mathbf{p}_1 - \mathbf{p}_0 + g(t)(\mathbf{n}_1 - \mathbf{n}_0) + g'(t)\left[(1-t)\mathbf{n}_0 + t\mathbf{n}_1\right]$$

and consequently

$$g'(0) = \frac{[\mathbf{p}_0 - \mathbf{p}_1 + g(0)(\mathbf{n}_0 - \mathbf{n}_1)] \cdot \mathbf{n}_0}{\mathbf{n}_0 \cdot \mathbf{n}_0} = [\mathbf{p}_0 - \mathbf{p}_1] \cdot \mathbf{n}_0 \tag{1}$$

$$g'(1) = \frac{[\mathbf{p}_0 - \mathbf{p}_1 + g(1)(\mathbf{n}_0 - \mathbf{n}_1)] \cdot \mathbf{n}_1}{\mathbf{n}_1 \cdot \mathbf{n}_1} = [\mathbf{p}_0 - \mathbf{p}_1] \cdot \mathbf{n}_1. \tag{2}$$

The height function $g(t)$ is then constructed from the endpoint derivative values and the Hermite basis functions:

$$g(t) = g'(0)H_1^3(t) - g'(1)H_2^3(t) \tag{3}$$

where H_i^3 are two of the cubic Hermite basis functions:

$$\begin{aligned} H_1^3(t) &= (1-t)^2 t \\ H_2^3(t) &= (1-t)t^2. \end{aligned}$$

4.2 Boundary Curves

The first step in the construction of the surface is to construct three boundary curves which interpolate the endpoints and derivatives (from the vertex normals). The edges

of the triangle are parameterized in terms of a single barycentric coordinate as follows:

$$\begin{aligned}
\mathbf{p}_0\mathbf{p}_1 &\leftrightarrow (1-\beta,\beta,0)\\
\mathbf{p}_1\mathbf{p}_2 &\leftrightarrow (0,1-\gamma,\gamma)\\
\mathbf{p}_2\mathbf{p}_0 &\leftrightarrow (\alpha,0,1-\alpha).
\end{aligned}$$

Each is therefore a univariate function and the cubic Hermite interpolation described in the previous section may be applied. The height function on the endpoints is necessarily zero, and if we assume the vertex normals are unit vectors, the boundary height functions have the clean formulation

$$\begin{aligned}
h(1-\beta,\beta,0) &= \mathbf{n}_0\cdot(\mathbf{p}_0-\mathbf{p}_1)(1-\beta)^2\beta\\
&- \mathbf{n}_1\cdot(\mathbf{p}_0-\mathbf{p}_1)(1-\beta)\beta^2 \qquad (4)\\
h(0,1-\gamma,\gamma) &= \mathbf{n}_1\cdot(\mathbf{p}_1-\mathbf{p}_2)(1-\gamma)^2\gamma\\
&- \mathbf{n}_2\cdot(\mathbf{p}_1-\mathbf{p}_2)(1-\gamma)\gamma^2 \qquad (5)\\
h(\alpha,0,1-\alpha) &= \mathbf{n}_2\cdot(\mathbf{p}_2-\mathbf{p}_0)(1-\alpha)^2\alpha\\
&- \mathbf{n}_0\cdot(\mathbf{p}_2-\mathbf{p}_0)(1-\alpha)\alpha^2. \qquad (6)
\end{aligned}$$

We have used the same letter h for all three edges because in effect the above equations represent the desired height function restricted to the boundaries. The Coons patch approach "fills in" the rest of the surface function in a meaningful way. The actual boundary curves are computed as

$$\begin{aligned}
\mathbf{p}(1-\beta,\beta,0) &= (1-\beta)\mathbf{p}_0+\beta\mathbf{p}_1\\
&+ h(1-\beta,\beta,0)\left[(1-\beta)\mathbf{n}_0+\beta\mathbf{n}_1\right] \qquad (7)\\
\mathbf{p}(0,1-\gamma,\gamma) &= (1-\gamma)\mathbf{p}_1+\gamma\mathbf{p}_2\\
&+ h(0,1-\gamma,\gamma)\left[(1-\gamma)\mathbf{n}_1+\gamma\mathbf{n}_2\right] \qquad (8)\\
\mathbf{p}(\alpha,0,1-\alpha) &= \alpha\mathbf{p}_0+(1-\alpha)\mathbf{p}_2\\
&+ h(\alpha,0,1-\alpha)\left[\alpha\mathbf{n}_0+(1-\alpha)\mathbf{n}_2\right]. \qquad (9)
\end{aligned}$$

4.3 Edge surface normals

To apply the Coons patch technique, the partial derivatives of the surface at the boundary curves must be specified. We do this by constructing a surface normal along each boundary curve, in a way which is dependent only on the two vertices and vertex normals of the edge so that the surface normal will be compatible with an adjacent triangle sharing the edge and vertex normals.

By definition, the surface normals at the vertices are simply the corresponding vertex normals. The surface normals along the edges, however, must be perpendicular to the tangents of the boundary curves constructed in the previous section, but this still leaves one degree of freedom. There are two obvious candidates for the surface normal: the interpolated normal along the edge, and the boundary curve normal obtained from the second derivative. But neither works. The interpolated normal does not generally match the curve tangent along the edge, while the curve normal fails to match the the vertex normals at the endpoints. So we use a normal constructed as

$$N_S(\alpha,\beta,\gamma) = [\mathbf{T}(\alpha,\beta,\gamma)\times\mathbf{n}(\alpha,\beta,\gamma)]\times\mathbf{T}(\alpha,\beta,\gamma) \qquad (10)$$

where **T** is the tangent vector to the boundary curve, and **n** is the interpolated normal on the edge. The tangent vector is simply the derivative of the boundary curve; in terms of the expressions above, these derivatives are

$$\begin{aligned}
\frac{\partial \mathbf{p}}{\partial \beta}(1-\beta,\beta,0) &= \mathbf{p}_1 - \mathbf{p}_0 + h(1-\beta,\beta,0)(\mathbf{n}_1 - \mathbf{n}_0) \\
&+ \frac{\partial h}{\partial \beta}(1-\beta,\beta,0)\left[(1-\beta)\mathbf{n}_0 + \beta\mathbf{n}_1\right] \\
\frac{\partial \mathbf{p}}{\partial \gamma}(0,1-\gamma,\gamma) &= \mathbf{p}_2 - \mathbf{p}_1 + h(0,1-\gamma,\gamma)(\mathbf{n}_2 - \mathbf{n}_1) \\
&+ \frac{\partial h}{\partial \gamma}(0,1-\gamma,\gamma)\left[(1-\gamma)\mathbf{n}_1 + \gamma\mathbf{n}_2\right] \\
\frac{\partial \mathbf{p}}{\partial \alpha}(\alpha,0,1-\alpha) &= \mathbf{p}_0 - \mathbf{p}_2 + h(\alpha,0,1-\alpha)(\mathbf{n}_0 - \mathbf{n}_2) \\
&+ \frac{\partial h}{\partial \alpha}(\alpha,0,1-\alpha)\left[\alpha\mathbf{n}_0 + (1-\alpha)\mathbf{n}_2\right]
\end{aligned}$$

where

$$\begin{aligned}
\frac{\partial h}{\partial \beta}(1-\beta,\beta,0) &= \mathbf{n}_0 \cdot (\mathbf{p}_0 - \mathbf{p}_1)\left[3(1\text{–}\beta)^2 - 2(1\text{–}\beta)\right] \\
&- \mathbf{n}_1 \cdot (\mathbf{p}_0 - \mathbf{p}_1)\left[2\beta - 3\beta^2\right] \\
\frac{\partial h}{\partial \gamma}(0,1-\gamma,\gamma) &= \mathbf{n}_1 \cdot (\mathbf{p}_1 - \mathbf{p}_2)\left[3(1\text{–}\gamma)^2 - 2(1\text{–}\gamma)\right] \\
&- \mathbf{n}_2 \cdot (\mathbf{p}_1 - \mathbf{p}_2)\left[2\gamma - 3\gamma^2\right] \\
\frac{\partial h}{\partial \alpha}(\alpha,0,1-\alpha) &= \mathbf{n}_2 \cdot (\mathbf{p}_2 - \mathbf{p}_0)\left[3(1\text{–}\alpha)^2 - 2(1\text{–}\alpha)\right] \\
&- \mathbf{n}_0 \cdot (\mathbf{p}_2 - \mathbf{p}_0)\left[2\alpha - 3\alpha^2\right].
\end{aligned}$$

By construction, the boundary curve tangents are perpendicular to the vertex normals at the vertices, so equation (10) matches the direction of the vertex normals. Also by construction, the surface normal is smooth on the edges, and only depends on the two incident vertices and vertex normals, so an adjacent triangle with the same vertex normals will have matching edge normals. Substituting the appropriate derivative for **T** in Equation 10 produces a viable surface normal on each edge. This construction therefore provides a viable edge surface normal, as long as neither the tangent vector nor the interpolated normal is zero.

4.4 Interpolated Surfaces

Given the boundary curves and associated surface normals, we can create a surface over the triangle using the "loft" operator

$$\begin{aligned}
\mathcal{P}_0 h(\alpha,\beta,\gamma) &= H_0^3(t)h(\alpha,1-\alpha,0) \\
&+ H_1^3(t)h_1(\alpha,1-\alpha,0) \\
&+ H_2^3(t)h_1(\alpha,0,1-\alpha) \\
&+ H_3^3(t)h(\alpha,0,1-\alpha)
\end{aligned}$$

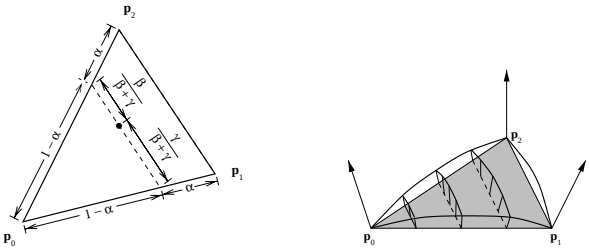

Fig. 9. *Left: Transfinite interpolation along a line of constant α. The parameter is either $\beta/(\beta+\gamma)$ or $\gamma/(\beta+\gamma)$, depending on the direction. Right: The resulting interpolated curves form a surface on the triangle.*

where

$$t = \frac{\gamma}{\beta + \gamma}.$$

The operator takes the boundary function and returns a surface (defined over the entire triangle) formed by Hermite interpolation along lines of constant α, as shown in Figure 9. The values of h_1 are the directional derivatives of h, in the direction of constant α, and correspond to the derivatives g' computed in equations (1) and (2). The normals $\mathbf{n}_a$ and $\mathbf{n}_b$ are the edge surface normals $\mathbf{n}_s(\alpha, 1-\alpha, 0)$ and $\mathbf{n}_S(\alpha, 0, 1-\alpha)$, respectively.

The surface function $\mathcal{P}_0 h$ interpolates both the boundary curve and the surface normal on the two edges $\mathbf{p}_0\mathbf{p}_1$ and $\mathbf{p}_2\mathbf{p}_0$, but only interpolates the curve on the edge $\mathbf{p}_1\mathbf{p}_2$.

Surfaces $\mathcal{P}_1 h$ and $\mathcal{P}_2 h$ are constructed similarly; each $\mathcal{P}_i$ has the proper interpolation on the two edges adjacent to vertex i, but not on the opposite edge.

4.5 Blending

The traditional Coons patch approach is to combine the lofted surfaces $\mathcal{P}_i$ using Boolean sums in a way which enforces the correct interpolation on all three edges. An alternative approach, which we follow, is to blend the three surfaces using Hermite blending functions. That is, the final height function is computed by blending the three surfaces

$$\begin{aligned} h_b(\alpha,\beta,\gamma) &= b_0(\alpha,\beta,\gamma)\mathcal{P}_0 h(\alpha,\beta,\gamma) \\ &+ b_1(\alpha,\beta,\gamma)\mathcal{P}_1 h(\alpha,\beta,\gamma) \\ &+ b_2(\alpha,\beta,\gamma)\mathcal{P}_2 h(\alpha,\beta,\gamma) \end{aligned} \tag{11}$$

using appropriate blending functions b_i.

To see what blending functions are required, consider the edge $\mathbf{p}_0\mathbf{p}_1$ where $\gamma = 0$. The surfaces $\mathcal{P}_0 h$ and $\mathcal{P}_1 h$ have the proper behavior on this edge, but $\mathcal{P}_2 h$ does not. We must therefore have both the blending function b_2, *and its derivatives with respect to* α *and* β, be zero for $\gamma = 0$. Again we are in the realm of Hermite interpolation; we blend based on the function

$$f(t) = 3t^2 - 2t^3$$

which has $f(0) = 0$, $f(1) = 1$, and $f'(0) = f'(1) = 0$. Furthermore, f satisfies the

symmetry relation $f(1-t) = 1 - f(t)$. Our blending functions are

$$\begin{aligned} b_0(\alpha,\beta,\gamma) &= \frac{f(\alpha)}{\sigma(\alpha,\beta,\gamma)} \\ b_1(\alpha,\beta,\gamma) &= \frac{f(\beta)}{\sigma(\alpha,\beta,\gamma)} \\ b_2(\alpha,\beta,\gamma) &= \frac{f(\gamma)}{\sigma(\alpha,\beta,\gamma)} \end{aligned}$$

with $\sigma(\alpha,\beta,\gamma) = f(\alpha)+f(\beta)+f(\gamma)$. (This denominator is included so that the blending functions always sum to 1, and remarkably enough, is necessary for the derivative interpolation to remain valid.)

There is a great deal of repetitive computation in the above exposition. In our implementation, we have a single function which, when passed two vertices and corresponding unit normals, returns the height function of the interpolated curve. A similar function returns the surface normal. Another function effects the $\mathcal{P}_0 h$ surface function by evaluating the boundary curves at $(\alpha, 1-\alpha, 0)$ and $(\alpha, 0, 1-\alpha)$ and applying Hermite interpolation. This function is called three times, each time with the vertices "cycled", and the resulting three height values are blended according to the blending function.

The Hermite interpolation was done with respect to the contrived edge surface normal N_S in this exposition, but that surface normal could be anything, as long as it matches the vertex normals. For example, if one of the edges of the triangle were part of a "crease" in a mesh, the edge surface normal could be different than that of the adjacent triangle.

5 Results

We evaluated our system on models with a large number of displaced triangles. Additionally, we wanted to verify the robustness of the algorithm under fairly extreme displacements. All scenes were rendered in parallel on an SGI O2K with 250 MHZ R1000K processors using a fairly standard Monte Carlo path tracer in order to capture shadows and indirect lighting effects.

The image in Figure 1 shows an icosahedron with high frequency displacements of roughly half the sphere radius. Without a smoothing displacement, the outline of the icosahedron would be visually obvious.

The second example is a piece of pottery containing 4680 initial triangles. The final displaced pottery is shown in Figure 10. For this scene N was fixed at 80. The 4680 initial triangles would have generated 30 million triangles if the geometry had been represented explicitly. Note that instancing would not have helped here. The image shows global illumination and shadowing effects on the grooves that would not have been possible either with bump mapping in a ray tracer, or without a global illumination framework. The 640x480 image was rendered using 256 paths of length 4 per pixel, and took roughly 24 CPU hours to run.

The final example is a small section of terrain data consisting of roughly 55,000 thirty meter cells. The resulting 110,000 triangles have been displacement mapped with an expensive displacement function based on several uses of the turbulence function[11] and is shown in Figure 11. The viewpoint is set near the ground, roughly at eye height for a person. The amount of subdivision was determined adaptively for each triangle.

Fig. 10. *A vase modeled with 4860 triangles showing interreflection effects. Generating all displaced microtriangle would have resulted in 30,000,000 triangles.*

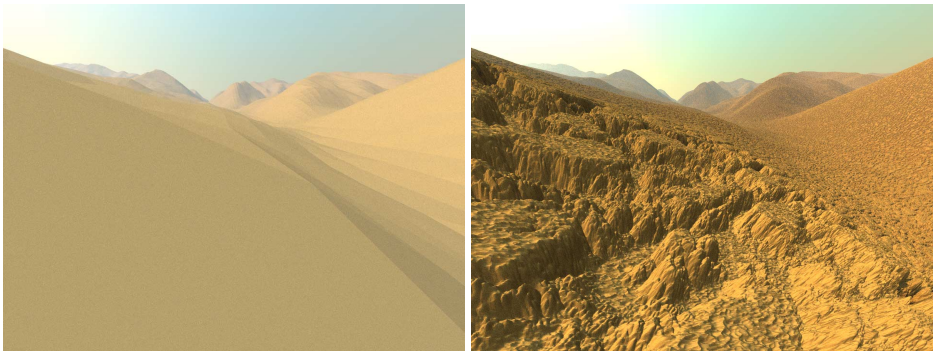

Fig. 11. *A terrain dataset with 110,000 initial polygons shown without displacements on the left. Right, with* $N_{\max} = 3162$ *and a procedural displacement map. Instantiating all the geometry would have resulted in more than 1,000,000,000,000 triangles.*

Because of the view, the foreground must be subdivided a large amount. We set $N_{\max} = 3162$, resulting in ten million potential microtriangles per input triangle (approximately 1cm wide microtriangles). The maximum N is achieved and needed for the left quarter of the image, where some facets can still be seen. Storing all 10^{12} triangles would have required about 100 terabytes. Our implementation requires roughly 10 megabytes for the terrain data. The 1200x900 image was generated with 36 paths of length 2 per pixel. Total CPU time was 43 hours. We believe that optimizing the algorithm and displacement function could reduce this time, as could changing the assumption in the ray tracer that object intersections are cheap, so testing objects multiple times is acceptable.

6 Discussion

The algorithm presented in this paper can produce ray traced images of displacement mapped geometry without resorting to explicitly stored tessellation or numerical root-finding. The goal of our system is to be able to render models with large amounts of displaced geometry. If the resulting displaced geometry is small, explicitly generating all polygons and putting them into a general acceleration scheme should prove faster. Our approach benefits from processor speeds continuing to grow faster than memory speeds and sizes, and provides a viable alternative to geometry caching schemes and numerical root finding.

We view this work as a proof-of-concept. There are potential numeric stability problems with the traversal. There are many areas where efficiency could be improved. Adaptively determining the subdivision amount, N, provides some performance benefits, however, there are two problems that can occur. Changing the level of subdivision for two adjacent pixels may cause some tearing. We are conservative in choosing the subdivision level, and haven't seen any artifacts due to this. A potentially more serious problem occurs when the displacement maps are used to represent surfaces such as brushed or scratched metal. Reducing the subdivision level can result in significant changes in appearance, even if the geometry itself is subpixel. In this case, we would like to carefully replace geometry with BRDF as discussed by Becker and Max [2].

Our current spline-based smoothing displacement function ensures that the base tessellations can be converted to smooth surfaces. The smooth surface is not always desirable, particularly in regions of high curvature or where the triangles have poor aspect ratios. It may take more information about the surface than we allowed in the restrictions from Section 4 to eliminate these problems. A more global interpolation scheme could ensure higher orders of continuity or a more intuitive fit to the data.

Acknowledgments

Thanks to Michael Ashikmin, Mark Bloomenthal, Elaine Cohen and Simon Premoze for helpful discussions. Thanks to Alias|Wavefront for their donation of Maya. This work was supported by NSF grants CDA–96–23614, 97–96136 and 97–31859.

Appendix: Proof of Smoothing Properties

To prove that the blending does create the proper surface function, let h_b be the blended height function defined in equation (11), and suppose h is an arbitrary height function with the desired boundary derivatives (consistent with the surface normal given in equation (10).) We show the derivatives of h_b match those of h. We will restrict attention to the edge with $\gamma = 0$, and the other edges will follow from symmetry. For simplicity, we write h_i for $\mathcal{P}_i h$, and leave off the explicit dependence on the barycentric coordinates in all expressions.

On the edge $\gamma = 0$ we have $\alpha+\beta = 1$ and as $f(\beta) = f(1-\alpha) = 1 - f(\alpha)$ we have $f(\alpha) + f(\beta) = 1$ and $\sigma(\alpha, \beta, 0) = 1$. Thus $b_0 = f(\alpha)$, $b_1 = f(\beta)$ and $b_2 = f(\gamma) = 0$.

Consequently, we have

$$\begin{aligned} h_b &= b_0 h_0 + b_1 h_1 \\ &= f(\alpha) h_0 + f(\beta) h_1 \\ &= [f(\alpha) + f(\beta)]\, h \\ &= h \end{aligned}$$

the last equality coming from the fact both $h_0(\alpha, \beta, 0)$ and $h_1(\alpha, \beta, 0)$ interpolate the boundary height function on the edge.

We prove the derivative interpolation by showing all the directional derivatives of the blended h_b surface match those of the boundary surface h itself. In barycentric coordinates, a directional derivative in the direction $\mathbf{d} = (d_\alpha, d_\beta, d_\gamma)$ (where the components sum to zero) is

$$h_{\mathbf{d}} = d_\alpha \frac{\partial h}{\partial \alpha} + d_\beta \frac{\partial h}{\partial \beta} + d_\gamma \frac{\partial h}{\partial \gamma}$$

It therefore suffices to show the partials of the blended height function h_b are equal to those of the boundary height partials. (The partials of the boundary function h have not been explicitly formulated, only implicitly in terms of the edge surface normal, but their explicit values are not needed to show that they match those of the blended function h_b.)

The partials of the blended surface evaluate to

$$\begin{aligned} \frac{\partial h_b}{\partial \alpha} &= b_0 \frac{\partial h_0}{\partial \alpha} + \frac{\partial b_0}{\partial \alpha} h_0 + b_1 \frac{\partial h_1}{\partial \alpha} + \frac{\partial b_1}{\partial \alpha} h_1 + b_2 \frac{\partial h_2}{\partial \alpha} + \frac{\partial b_1}{\partial \alpha} h_2 \\ \frac{\partial h_b}{\partial \beta} &= b_0 \frac{\partial h_0}{\partial \beta} + \frac{\partial b_0}{\partial \beta} h_0 + b_1 \frac{\partial h_1}{\partial \beta} + \frac{\partial b_1}{\partial \beta} h_1 + b_2 \frac{\partial h_2}{\partial \beta} + \frac{\partial b_1}{\partial \beta} h_2 \\ \frac{\partial h_b}{\partial \gamma} &= b_0 \frac{\partial h_0}{\partial \gamma} + \frac{\partial b_0}{\partial \gamma} h_0 + b_1 \frac{\partial h_1}{\partial \gamma} + \frac{\partial b_1}{\partial \gamma} h_1 + b_2 \frac{\partial h_2}{\partial \gamma} + \frac{\partial b_1}{\partial \gamma} h_2 \end{aligned}$$

The partials of the blending functions on the edge are

$$\begin{array}{lll} \frac{\partial b_0}{\partial \alpha} = f'(\alpha) - f(\alpha) f'(\alpha) & \frac{\partial b_1}{\partial \alpha} = -f(\beta) f'(\alpha) & \frac{\partial b_2}{\partial \alpha} = 0 \\ \frac{\partial b_0}{\partial \beta} = -f(\alpha) f'(\beta) & \frac{\partial b_1}{\partial \beta} = f'(\beta) - f(\beta) f'(\beta) & \frac{\partial b_2}{\partial \beta} = 0 \\ \frac{\partial b_0}{\partial \gamma} = 0 & \frac{\partial b_1}{\partial \gamma} = 0 & \frac{\partial b_2}{\partial \gamma} = 0. \end{array}$$

Substitution, combined with the identities $f(\alpha) + f(\beta) = 1$, and the observation that the relevant partials of h_0 and h_1 are equal to those of h on the edge (by construction) it follows that

$$\frac{\partial h_b}{\partial \alpha} = \frac{\partial h}{\partial \alpha}, \qquad \frac{\partial h_b}{\partial \beta} = \frac{\partial h}{\partial \beta}, \qquad \frac{\partial h_b}{\partial \gamma} = \frac{\partial h}{\partial \gamma}$$

and thus the blending produces the proper surface.

References

1. ALIAS|WAVEFRONT. *Maya v. 1.5*. Toronto, Canada, 1998.

2. BECKER, B. G., AND MAX, N. L. Smooth transitions between bump rendering algorithms. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 183–190.
3. COOK, R. L., CARPENTER, L., AND CATMULL, E. The reyes image rendering architecture. *Computer Graphics (SIGGRAPH '87 Proceedings)* (July 1987), 95–102. Held in Anaheim, California.
4. GRITZ, L., AND HAHN, J. K. BMRT: A global illumination implementation of the renderman standard. *Journal of Graphics Tools 1*, 3 (1996), 29–47. ISSN 1086-7651.
5. HEIDRICH, W., AND SEIDEL, H.-P. Ray-tracing procedural displacement shaders. *Graphics Interface '98* (June 1998), 8–16. ISBN 0-9695338-6-1.
6. KAJIYA, J. T. New techniques for ray tracing procedurally defined objects. In *Computer Graphics (SIGGRAPH '83 Proceedings)* (July 1983), vol. 17, pp. 91–102.
7. KRISHNAMURTHY, V., AND LEVOY, M. Fitting smooth surfaces to dense polygon meshes. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), H. Rushmeier, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 313–324. held in New Orleans, Louisiana, 04-09 August 1996.
8. LOGIE, J. R., AND PATTERSON, J. W. Inverse displacement mapping in the general case. *Computer Graphics Forum 14*, 5 (December 1995), 261–273.
9. MUSGRAVE, F. K. Grid tracing: Fast ray tracing for height fields. Technical Report YALEU/DCS/RR-639, Yale University Dept. of Computer Science Research, 1988.
10. PEDERSON, H. K. Displacement mapping using flow fields. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)* (July 1994), A. Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, pp. 279–286. ISBN 0-89791-667-0.
11. PERLIN, K., AND HOFFERT, E. M. Hypertexture. In *Computer Graphics (SIGGRAPH '89 Proceedings)* (July 1989), J. Lane, Ed., vol. 23, pp. 253–262.
12. PHARR, M., AND HANRAHAN, P. Geometry caching for ray-tracing displacement maps. *Eurographics Rendering Workshop 1996* (June 1996), 31–40. ISBN 3-211-82883-4. Held in Porto, Portugal.
13. PHONG, B.-T. Illumination for computer generated pictures. *Communications of the ACM 18*, 6 (June 1975), 311—317.
14. STANDER, B. T., AND HART, J. C. Guaranteeing the topology of an implicit surface polygonization for interactive modeling. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 279–286. ISBN 0-89791-896-7.
15. VEACH, E., AND GUIBAS, L. J. Metropolis light transport. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 65–76. ISBN 0-89791-896-7.