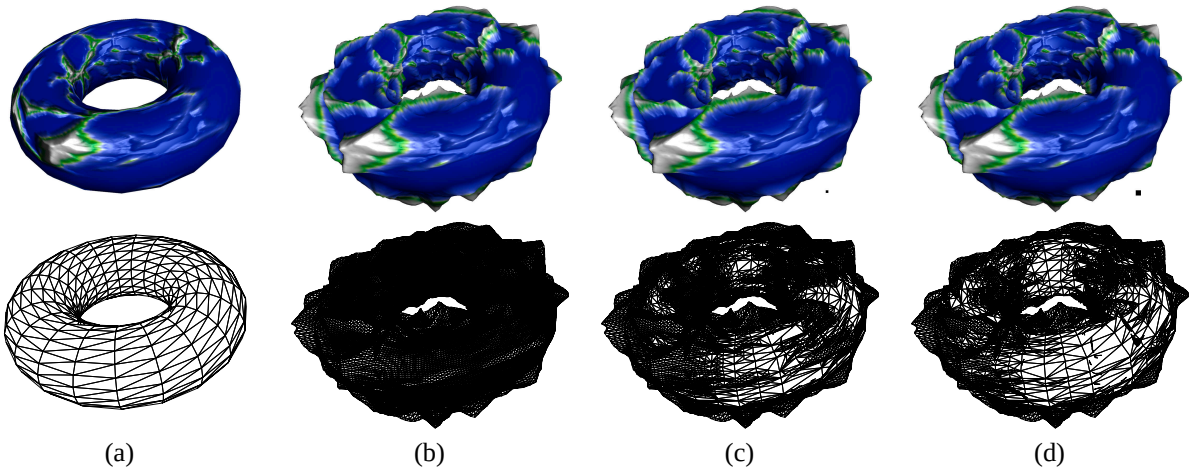
Efficient Bounded Adaptive Tessellation of Displacement Maps

Kevin Moule Michael D. McCool

Computer Graphics Lab
Department of Computer Science
University of Waterloo



(a) (b) (c) (d)

Figure 1: (a) Bump-mapping only (882 triangles), (b) Uniform tessellation (56448 triangles), (c) Adaptive tessellation with a tolerance of 5 pixels (34377 triangles), (d) Adaptive tessellation with a tolerance of 10 pixels (21521 triangles).

Abstract

Displacement mapping is a technique for applying fine geometric detail to a simpler base surface. The displacement is often specified as a scalar function which makes it relatively easy to increase visual complexity without the difficulties inherent in more general modeling techniques. We would like to use displacement mapping in real-time applications. Ideally, a graphics accelerator should create a polygonal tessellation of the displaced surface on the fly to avoid storage and host bandwidth overheads.

We present an online, adaptive, crack-free tessellation scheme for real-time displacement mapping that uses only local information for each triangle to perform a view-dependent tessellation. The tessellation works in homogeneous coordinates and avoids retransformation of displaced points, making it suitable for high-performance hardware implementation. The use of interval analysis produces meshes with good error bounds that converge quickly to the true surface.

Key words: Hardware acceleration. Bump mapping. Displacement mapping. Adaptive tessellation.

1 Introduction

Displacement mapping is a technique for adding surface detail to a base surface. The base surface is moved along its normal at each point on the surface. The distances moved, called the displacements, are generated either by a function defined procedurally or are interpolated from a 2D array. Rendering displacement maps has traditionally been either space-inefficient or computationally expensive (or both). Many techniques neither map onto hardware implementations nor fit within the standard hardware pipeline.

We propose a technique for rendering displacement maps that is targeted towards real time rendering using hardware acceleration. Our technique is not targeted at current accelerators; instead, it is an algorithm *suitable* for implementation in future hardware. However, the technique is simple and efficient enough that a high-performance implementation can be obtained using a software implementation along with the use of a standard accelerator as a back end.

Several restrictions are imposed in the context of hard-

ware accelerated rendering. Firstly, the nature of the graphics pipeline is such that geometric primitives are processed one at a time without any knowledge of the primitive that came before or will come after. Secondly, memory systems in graphics accelerators are specialized for access to coherent data structures. Storing information in general linked structures is usually avoided since variability in access time would adversely effect the rest of the pipeline. This implies that we cannot use or manipulate a complete representation of the geometry, but should design a technique that operates locally.

Under these restrictions we will present a technique that can render displacement mapped surfaces in real-time on current graphics hardware but that is also suitable for direct implementation in future hardware. The technique performs an online adaptive tessellation which tries to minimize the complexity of the resulting mesh. A brute-force technique could also be used. We chose to pursue an adaptive technique so that displacement mapping would produce fewer primitives on average, would access less memory on average, and also to support a simple form of adaptive geometry suitable for terrains, characters, etc.

Our technique uses a hierarchical array structured data representation based on interval analysis as a representation of the displacement function. The extra hierarchical bounding information is used to guide a recursive tessellation process. In hardware, a stack would be used to implement the recursion. At each stage of the recursion, an oracle is used to determine if a given triangle is a sufficient approximation of the displaced surface using an interval error metric. We use a view-dependent, division-free error metric that bounds error in screen space. We also combine displacement with bump-mapped per-pixel lighting. This hides changes in tessellation that would be objectionable under Gouraud shading.

In Section 2, we will define displacement mapping more precisely. Section 3 describes the recursive adaptive tessellation scheme we use. Then, in Section 4 we describe our hierarchical displacement map representation, which provides an efficient interval extension of texture mapping. Section 5 defines an oracle which can efficiently and accurately determine the need for additional mesh subdivision. We evaluate the quality and performance of our algorithm in Section 6. Finally, we summarize our results in Section 7.

2 Displacement Mapping

Displacement mapping is a general technique for applying fine geometric detail to a simpler base surface and can be formulated in many ways. A common approach uses scalar displacement values and the normals at each base surface point to generate the displaced surface (Figure 2). Given a surface $\mathbf{s}(u, v)$ parameterized by u and v the displaced surface $\mathbf{s}_d(u, v)$ can be defined as

$$\mathbf{s}_d(u, v) \quad = \quad \mathbf{s}(u, v) + d(u, v)\hat{\mathbf{n}}(u, v) \qquad (1)$$

where $\hat{\mathbf{n}}(u, v)$ is the unit base surface normal and $d(u, v)$ is the scalar displacement function. The displacement function can be defined procedurally or interpolated from samples stored in an array (a displacement texture).

In the context of hardware based triangle rasterization the displaced surface can be approximated by displacing the vertices of a sufficiently dense subdivision of the base surface. Image-based warping techniques can also be used, but these provide fewer opportunities for backward-compatible deployment on older accelerators.

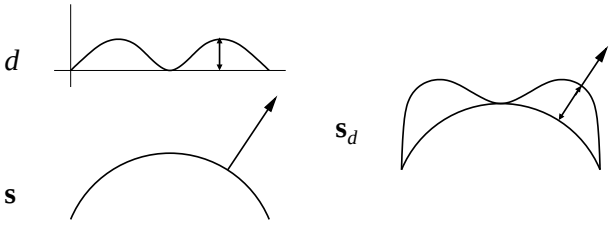


Figure 2: Displacement Mapping.

Displacement mapping was first introduced by Cook in the context of Shade Trees [2] and later in the REYES architecture [3]. The REYES architecture uses sub-pixel micro-polygons for rendering. These micro-polygons are simply displaced at their vertices and this provides a very dense tessellation which gives sufficient accuracy. The REYES architecture is not targeted towards real-time rendering and this micro-tessellation scheme is overkill for real-time application.

Displacement mapping has also been applied in ray-tracing. Early approaches used an inverse technique [10, 11] to warp the base surface to flatten it; the warp curves the ray. This curved ray is then intersected with the displacement map as if it were a height field. Recent techniques have explored direct ray tracing using techniques such as affine arithmetic [7], sophisticated caching schemes [13] and grid base intersections [15].

Alternatively, both image warping [14] and volume rendering [8] techniques have been explored. However, warping techniques require special hardware support (and so would not be backward compatible with existing accelerators), while volumetric techniques require a large texture bandwidth and fill rate.

Recently a few hardware-oriented algorithms for tessellation-based displacement mapping have been proposed. Gumhold and Hüttner [6] proposed a technique

that produces a uniform tessellation based on the screen space projection of each base triangle. The technique also tessellates in the z-direction generating triangles where bump mapping would suffice. Doggett and Hirche [5] proposed a technique similar to ours that performs an adaptive tessellation. Their technique uses a combination of two heuristics to guide the tessellation, a point sampled normal variation test and an average height test based on summed area tables. Independently, neither test is sufficient but the combination of the two produces good results. However, their heuristic test is not guaranteed. In particular, the use of average displacement rather than maximum displace can result in the omission of small features with large displacement. The view dependent element of their approach is limited to edge length in screen space, the nature of the displacement on the untessellated interior of the triangles is not considered. Doggett, Kugler and Strasser [4] proposed a multiple pass technique that first renders the geometry on a coarse screen space displacing the coarse pixels. The coarse pixels are then re-meshed into new triangles which are used in the final rendering. The technique is not adaptive and the memory requirements would be comparable to the stack space needed for our approach.

There are also indications that industry support is growing for real-time displacement mapping. A recent paper from NVIDIA proposes displacement mapped subdivision surfaces as a general modeling technique [9]. In a recent presentation by Microsoft on DX9 displacement mapping was mentioned, although in a simple form: a data stream synchronized to the vertex stream can be provided to vertex shaders. This approach is limited to the displacement of the original vertices.

3 Adaptive Tessellation

Displacement mapping can always be implemented by displacing a uniform subdivision of the base surface. However, adaptive tessellation will produce fewer triangles than brute force tessellation and, especially if view-dependent, will make more efficient use of the rasterization units.

If the base surface is approximated using a triangular mesh with vertex normals, adaptive tessellation can be implemented on a triangle by triangle basis. As an input triangle is processed, a local decision can be made as to whether the triangle needs to be split into several triangles to approximate the displaced surface to sufficient accuracy. If the triangle is split, the resultant sub-triangles are then handled in a recursive fashion [1, 5].

Our tessellator is based on this concept. An input triangle has three edges, E_1, E_2 and E_3. Each of these edges is tested independently using an *oracle*. The oracle is a function which takes an edge and information attached to the vertices at the ends of that edge as input, and returns 0 if the edge does not need to be split and 1 otherwise. Triangle edges are always split at their midpoints, introducing new vertices whose properties are set by interpolating the properties of the endpoints of the edge. Given the results from oracle evaluations for all three triangle edges a three bit code can be constructed. This code is used to select one of eight tessellation patterns (Figure 3) to generate a new set of sub-triangles combining the old and new vertices. The resultant sub-triangles are evaluated and possibly further tessellated recursively. Three of the tessellation patterns produce a quadrilateral that needs to be split into two triangles. We have chosen to split the quadrilateral in an arbitrary but consistent fashion. A more intelligent choice could be made by using the displacement data to drive the split but this would incur extra computation and in our tests did not have a huge impact on the resultant tessellation.

Since the tessellation is based on edge splitting, a crack free tessellation can be guaranteed provided the oracle is deterministic and vertex symmetric, that is, it returns exactly the same result given exactly the same edge. This permits input triangles that share edges to be specified and processed in any order. The information provided to the oracle *must* be local to the edge (such as position and texture coordinates) and should not include any information unique to the triangle (such as the triangle normal). Under these conditions two triangles that share an edge will provide the same information to the oracle and the algorithm will generate the same tessellation on either side of the edge. This scheme is not guaranteed to generate an optimal tessellation. It is, however, online, local, robust (with a suitable oracle) and simple enough for hardware implementation.

A naive implementation of this approach requires redundant computation. As the recursion proceeds from one level to the next any edges generated in the interior of the input triangle will produce two sub-triangles that share a common edge. In the naive approach the oracle for this edge would need to be evaluated for both sub-triangles that share it even though the result will be identical. Splitting this common edge will also require the vertex position and any associated parameters to be interpolated twice. As an example, if a triangle is split into four at every step and the recursion depth is four then only 150 unique vertices are generated but 255 oracle evaluations and edge splits are performed.

To reduce the number of redundant calculations we can restructure the recursion such that the oracle evaluations and interpolations are done one step ahead of time. The evaluation work needed at level i is done at level $i-1$ and

is shared across all triangles at level i. Likewise, level i pre-calculates the evaluation for level $i+1$ and passes this information on. This technique requires a larger stack since up to nine extra oracle evaluations and interpolated vertices need to be stored. However, oracle results take only a small amount of space: one bit each. Also, we would want to allocate memory in advance for the worst case stack size anyway. Interpolation computations for edge splits are computed regardless of the results of the oracle, but are put into effect only if the oracle evaluations return true. Our results have shown that this precalculation technique greatly improves the performance of a software implementation despite the "wasted" oracle and interpolation computations. In a hardware context, we might want to employ multiple arithmetic units for evaluating the worst-case number of oracles and interpolations in parallel. In the case of multiple evaluation units, avoiding a single oracle evaluation or interpolation would not save any time. However, we might want to *only* precompute the oracles, to save space on the stack.

Similar schemes to the one we present here have been used previously for tessellating parametric surfaces [1] and displacement maps [5]. On the one hand, hardware supporting the algorithm proposed here could be extended to the tessellation of procedurally generated parametric surfaces in a future accelerator (supporting generative modeling [16], for instance). This would require only programmable vertex-position functions and interval evaluation of those functions, which is not completely out of the question. However, to keep things simple, we have focussed on displacement mapping. The differences between our technique and the prior use of a similar scheme for adaptive tessellation of displacement maps [5] lie in our oracle and our representation of the displacement map. Our approach is both more robust (it will not miss small features) and operates in homogeneous coordinates. The latter property means we can operate in device space and so do not have to re-transform the vertices generated by the algorithm.

4 Displacement Map Representation

In addition to evaluation of the displacement at a point our oracle requires an upper and lower bound on the range of the displacement function given an area of its domain. In other words, our algorithm is based on interval analysis and we need an interval extension of the displacement function.

More formally, given a displacement function $d(u,v)$ we define its interval extension $D(U,V)$ as follows:

$$\begin{aligned} U &= [u_L, u_H], \\ V &= [v_L, v_H], \end{aligned}$$

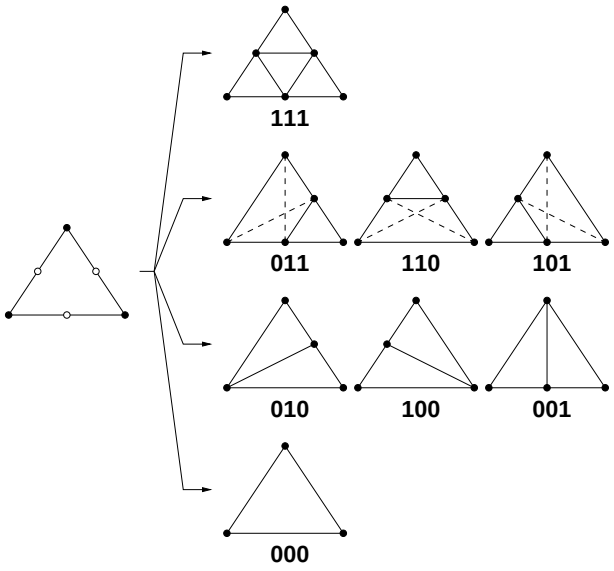


Figure 3: Tessellation patterns. Each pattern has a corresponding three bit code.

$$\begin{aligned} d_L(U,V) &= \min\{d(u,v) | u \in U, v \in V\}, \\ d_H(U,V) &= \max\{d(u,v) | u \in U, v \in V\}, \\ D(U,V) &= [d_L(U,V), d_H(U,V)]. \end{aligned}$$

If the displacement function is defined procedurally then interval arithmetic techniques can be used [16]. This approach was used by Heidrich [7] (using affine interval arithmetic) for ray tracing procedural displacement shaders. If the displacement function is tabulated in the form of a 2D array (as a texture map), or if a texture is used as part of a procedural displacement shader, then an interval extension of the texture lookup operator is needed.

4.1 Bounding a Tabulated Displacement Function

Upper and lower bounds could easily be calculated from a tabulated displacement function by iterating over the desired area, $[u_L, u_H] \times [v_L, v_H]$, to find d_L and d_H. This provides the tightest bound but unfortunately the computational cost is far too high. Instead we use a data structure, similar to MIP-maps, that stores precomputed bounds over predetermined areas.

For our representation, a hierarchy of arrays is built containing $\lg(r)$ levels where r is the resolution of the initial array. Usually, as in MIP-maps, we would start with a square array whose dimensions are a power of two. Each level is a quarter of the size of the previous level, obtained by reducing each dimension by a factor of two. However, unlike MIP-maps, the coarser levels are populated with the *interval* that bounds the correspond-

ing area in the next finer level. Given a particular entry (i, j) at level ℓ, the entry will be populated with the interval over the area $[2^\ell i, 2^\ell(i+1)] \times [2^\ell j, 2^\ell(j+1)]$. The entries can be calculated using the initial tabulated data. Alternatively the entire hierarchy can be constructed in a recursive fashion where entries for level i are calculated using four entries from level $i-1$. A one-dimensional example hierarchy is shown in Figure 4.

The storage requirements for this representation are similar to MIP-maps. The quarter reduction in size of each subsequent level leads to a storage overhead of one third the original size. The interval bound requires an upper and lower value, doubling the size of the hierarchy which results in two thirds the original size.

In our experience, 8-bit displacements are too coarse and at least 16-bit precision is required. However, 8-bit precision can be used for the upper and lower bounds in the internal nodes of the interval hierarchy if outward (conservative) rounding is used. If this is done, then each such interval would take as much space as one original 16-bit sample. This means that a displacement map would take exactly as much space as a single-channel 16-bit MIP-map.

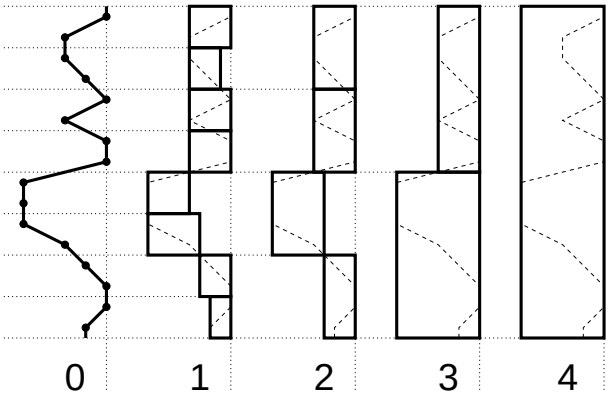


Figure 4: Example interval hierarchy.

The naive approach to acquire an interval bound using the hierarchy is to simply use the entry that completely encompasses the desired area $[u_L, u_H] \times [v_L, v_H]$. This can be accomplished by finding the level ℓ_u that encompasses $[u_L, u_H]$, and the level ℓ_v that encompasses $[v_L, v_H]$, and use $\ell = \max(\ell_u, \ell_v)$. Define the integer representations of u as $i = \lfloor Nu + 0.5 \rfloor$ and v as $j = \lfloor Nv + 0.5 \rfloor$, where N is the number of samples in the finest level of the hierarchy. The level ℓ_u is the number of low-order bits that must be removed from i before $(i_L \gg \ell_u) = (i_H \gg \ell_u)$, and likewise for ℓ_v (where $\gg$ is the right bit shift operator). The appropriate entry in level ℓ can then be found by indexing a 2D array using $(i_L \gg \ell)$ and $(j_L \gg \ell)$.

This approach tends to produce very loose bounds compared to the optimal (exhaustive search) method because the entry selected usually corresponds to an area much larger than the desired one. In particular, if one of the dimensions of the desired area has an integer representation of its interval bounds that are identical in only the topmost bits then the resultant level may be arbitrarily large. This leads to poor convergence and unnecessary tessellation concentrated near the power of two areas in the displacement map. This can be seen in Figure 6(c).

A better approach is to take the interval union of several entries from the hierarchy to construct a tighter bound. We use up to four entries. The possible configurations of the entries is shown in Figure 5. Other configurations and methods for obtaining tighter bounds are possible but in our experiments this set provides sufficiently tight bounds while still being straightforward to calculate.

First, for each dimension an appropriate level is found. The initial level estimate for a given dimension is found using *only* the width of the interval in that dimension, rounded up to the nearest power of two. Without loss of generality we will consider only u, in which case the level estimate would be

$$w_u = \lceil \lg(i_H - i_L + 1) \rceil.$$

Define

$$\begin{aligned} a_u &= (i_L \gg (w_u - 1)), \\ b_u &= (i_H \gg (w_u - 1)). \end{aligned}$$

If $b_u - a_u > 1$, then $\ell_u = w_u$. Otherwise, $\ell_u = w_u - 1$. The initial estimate w_u is sufficient to encompass i_L and i_H. The values a_u and b_u are the entry indices if $w_u - 1$ were used. If b_u and a_u are consecutive entries then $w_u - 1$ would suffice, otherwise w_u is necessary. After finding ℓ_v in the same way, we compute $\ell = \max(\ell_u, \ell_v)$.

Second, once the level ℓ is found each of i_L, i_H, j_L and j_H are right shifted ℓ bits. Due the way ℓ is constructed there are four cases into which the shifted values fall:

1. If $(i_L \gg \ell) = (i_H \gg \ell)$ and $(j_L \gg \ell) = (j_H \gg \ell)$, then only one entry is needed (Figure 5(a)).

2. If $(i_L \gg \ell) = (i_H \gg \ell)$ and $(j_L \gg \ell) < (j_H \gg \ell)$, then two vertical entries are needed (Figure 5(b)).

3. If $(i_L \gg \ell) < (i_H \gg \ell)$ and $(j_L \gg \ell) = (j_H \gg \ell)$, then two horizontal entries are needed (Figure 5(c)).

4. If $(i_L \gg \ell) < (i_H \gg \ell)$ and $(j_L \gg \ell) < (j_H \gg \ell)$, then four entries are needed (Figure 5(d)).

The computational and bandwidth cost of this approach is higher than the naive approach, especially when four entries are combined, which is in fact the predominant case. However, our results have shown that although the cost of the multi-sampling oracle is higher the tighter bounds and better convergence leads to an overall performance that is significantly higher than the naive approach. The lookup cost of the multi-sampling oracle is comparable to bilinear interpolation in ordinary 2D texture mapping (which also requires four samples).

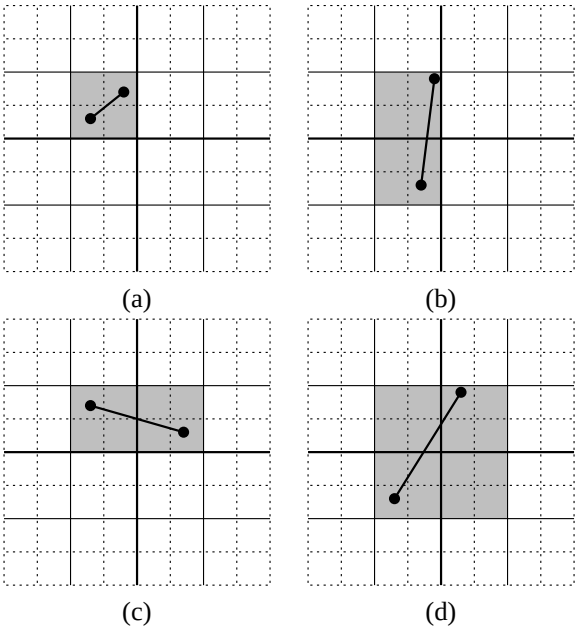


Figure 5: Examples of bounding an edge by combining several intervals.

A comparison of the optimal technique, the naive approach, and our approach are shown in Figure 6. Our approach is significantly closer to the optimal approach, focusing on the areas that require tessellation without excessive over-tessellation in unimportant areas.

5 Oracle

Our oracle function attempts to bound the maximum error within an *area* enclosing an edge. Using the texture coordinates of the two world space coordinate endpoints, $\mathbf{p}_1$ and $\mathbf{p}_2$, we can define an area in texture space (Figure 7). Over this area an interval, D, is found which bounds the displacement using the method described in Section 4. The union of the areas for the three edges that define an input triangle will be guaranteed to cover the interior of the triangle. This guarantees that even though the oracle is evaluated over an edge any displacement details contained on the interior of the triangle will be incorporated

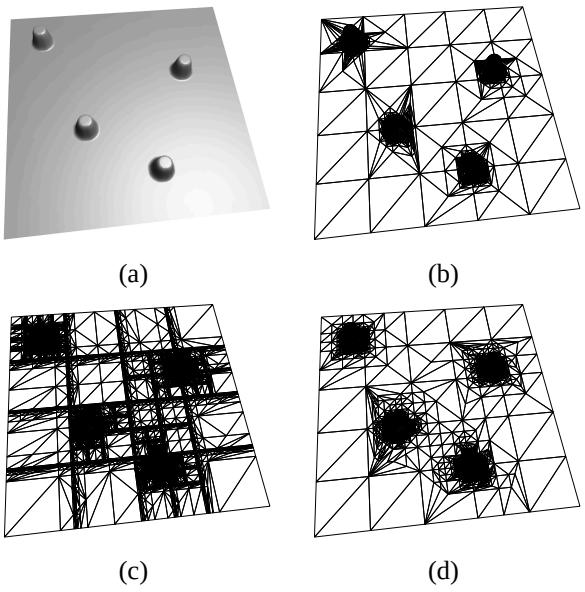


Figure 6: Comparison of interval lookup techniques. (a) Reference rendering, (b) Optimal (exhaustive search), (c) Naive (enclosing interval), (d) Proposed technique (four interval samples).

into one of the three oracle evaluations.

Using the interval bound D and an edge defined between points $\mathbf{p}_1$ and $\mathbf{p}_2$ with normals $\hat{\mathbf{n}}_1$ and $\hat{\mathbf{n}}_2$, we can find the position of the maximum displacement along the edge as follows:

$$\begin{aligned} \mathbf{p}(t) &= \mathbf{p}_1 + t\,(\mathbf{p}_2 - \mathbf{p}_1), \\ \vec{\mathbf{n}}(t) &= \hat{\mathbf{n}}_1 + t\,(\hat{\mathbf{n}}_2 - \hat{\mathbf{n}}_1), \\ \mathbf{p}_L(t) &= \mathbf{p}(t) + d_L\vec{\mathbf{n}}(t) \\ \mathbf{p}_H(t) &= \mathbf{p}(t) + d_H\vec{\mathbf{n}}(t) \\ t^* &= \arg\max_{t\in[0,1]}(||\mathbf{p}_L(t) - \mathbf{p}_H(t)||) \end{aligned} \quad (2)$$

Given t^*, the location of the maximum width, we decide if the edge should be split by comparing the width of this interval point against a user-defined threshold, ϵ.

$$||\mathbf{p}_L(t^*) - \mathbf{p}_H(t^*)|| < \epsilon. \quad (3)$$

In world space coordinates the left hand side of Equation 3 reduces to $||D||$, the width of the interval. This can be used to obtain a fixed world space tessellation.

However, in screen space with the distance measured in x and y only a view dependent tessellation can be performed. In screen space the maximum width will occur at one of the end points, whichever is closer to the eye. To determine if an edge needs to be split we can evaluate

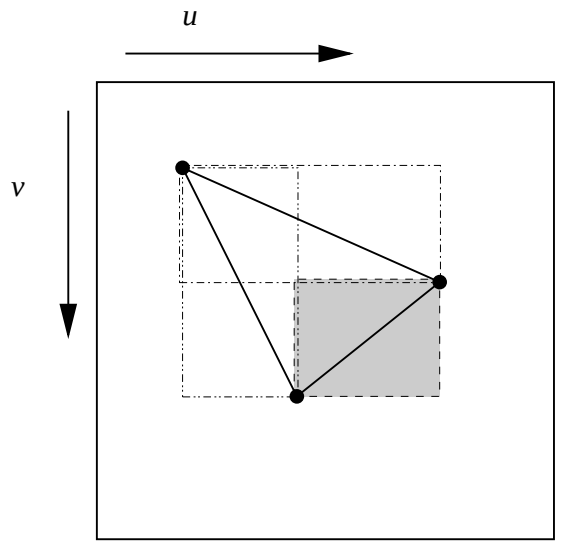


Figure 7: Areas defined by edges in texture space. The displacement is bounded over each of these areas when evaluating the oracle for the corresponding edge.

Equation 3 at $t = 0$ and $t = 1$. If both values are smaller than ϵ the edge need not be split and the oracle returns 0; otherwise the oracle returns 1 and the edge is split. Problems do arise when the edge crosses the eye plane, the maximum displacement will occur at the crossover point. This condition can be easily detected by inspecting the signs of the homogeneous w coordinates of $\mathbf{p}_1$ and $\mathbf{p}_2$. If the signs are different then the edge cross the eye plane and a split is automatic. In the following consider only the case where the signs of the w values are equal.

Using the homogeneous components of $\mathbf{p}_L(t)$ and $\mathbf{p}_H(t)$,

$$\begin{aligned} \mathbf{p}_L(t) &= (x_L, y_L, z_L, w_L), \\ \mathbf{p}_H(t) &= (x_H, y_H, z_H, w_H). \end{aligned}$$

Equation 3 reduces to

$$\sqrt{\left(\frac{x_H}{w_H} - \frac{x_L}{w_L}\right)^2 + \left(\frac{y_H}{w_H} - \frac{y_L}{w_L}\right)^2} \quad < \quad \epsilon.$$

After some further manipulation, this reduces to

$$\begin{aligned} (w_L x_H - w_H x_L)^2 \quad &+ \\ (w_L y_H - w_H y_L)^2 \quad &< \quad (w_H w_L \epsilon)^2. \end{aligned} \tag{4}$$

This form of the oracle will not generate subdivisions when the displacement is directly towards the eye (since bump-mapping alone suffices in that case). It might be interesting to include the $\frac{z}{w}$ term, for example if bump-mapping is not used or the geometry for a shadow map is needed. The nonlinear encoding of z would need to be considered, in particular the interpretation of the ϵ threshold would need to be reconsidered.

6 Results

This algorithm has been implemented as a C library. The library sits on top of OpenGL and exposes an OpenGL-like API extended with displacement mapping functionality. An 866 Mhz Pentium III machine with an NVIDIA GeForce3 running Linux was used to render the images and collect timing information. Bump mapping was implemented in all cases using the NVIDIA texture shader extensions.

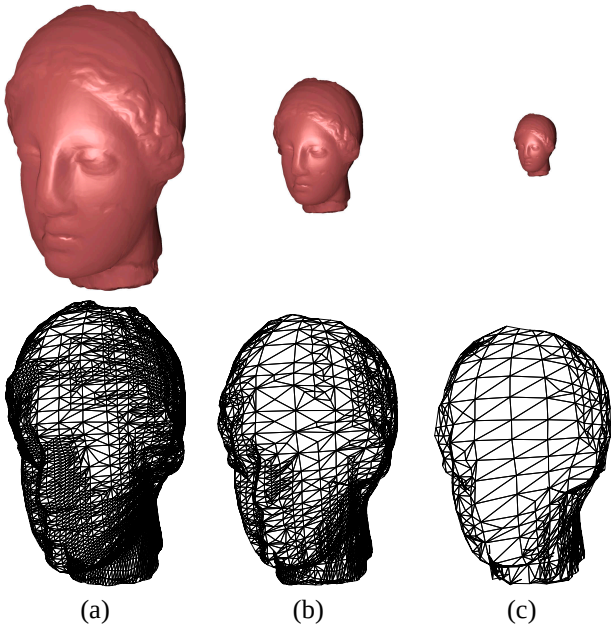


Figure 8: Venus example. Rendering at various distances from the eye with a magnified wire-frame: (a) 21959 triangles at 20fps, (b) 9269 triangles at 30fps and (c) 2516 triangles at 85fps.

Figure 8 shows the venus bust modeled as a displacement mapped sphere. The displacement map was generated using a ray-tracing method similar to the one described by Lee *et al* [9]. In Figure 8 the model is rendered at various distances from the eye. The adjacent wire-frame shows the adaptive tessellation in close-up. As the model moves farther away the oracle decides that more and more edges do not need to be split. This results in significantly reduced tessellation while preserving visual quality. The wire-frame images also demonstrate that the face itself requires less tessellation since it is fairly smooth and relatively perpendicular to the viewing direction. However, the silhouette and the neck require more

tessellation to maintain visual quality and henceforth a higher tessellation level in these regions is apparent at all distances.

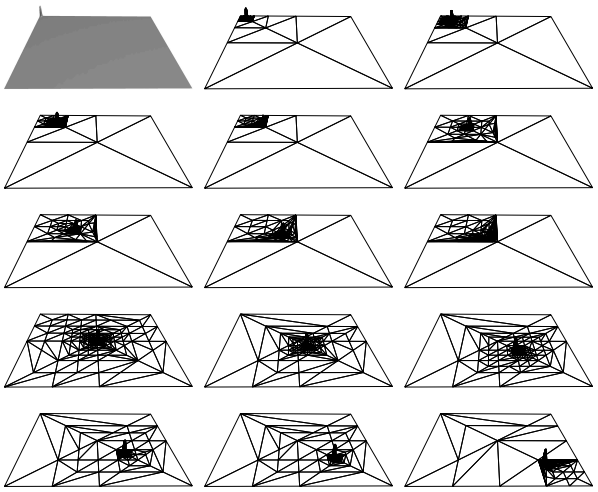


Figure 9: Spike example: average of 700 triangles, frame rate of 1000fps+.

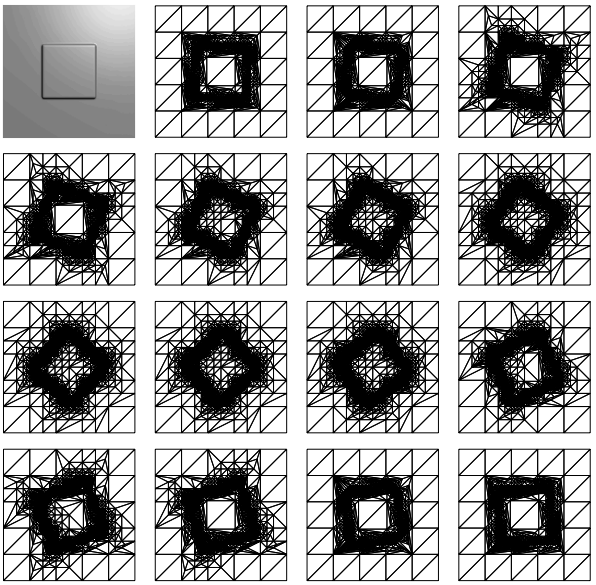


Figure 10: Spin example: average of 2100 triangles, frame rate of 380fps.

Figure 9 demonstrates that the algorithm does not miss details. The displacement map corresponds to a small spike with a large displacement and no displacement elsewhere. The initial model used is a plane consisting of two large triangles. Several displacement maps are used with the spike positioned at different locations along the diagonal. Although the amount of tessellation differs depending on the location of the spike, the spike is never missed. The variation seen in the tessellation is due to interaction between the spike position, the interval hierarchy, and the initial tessellation. In particular, the interval hierarchy is sensitive to power of two boundaries.

Since our tessellation splits triangle edges only at their midpoints, the convergence to specific features in the displacement map may be slower than that of more flexible schemes. This is particularly noticeable when a displacement map has discontinuities. Figure 10 shows several displaced rectangles each rotated at a different angle (the oracle used here uses produces a fixed, world space tessellation). In the first few images the displacement map has a sampling pattern that coincides with the underlying model and the tessellation produces good visual results. As the rectangle is rotated further, more tessellation is required to capture the discontinuity. The fixed recursion depth in this example causes small "braiding" artifacts to appear when the underlying geometry and the displacement map discontinuities conflict.

We point out these limitations of our algorithm only for comparison purposes. It is important, however, to remember the specialized constraints under which our algorithm operates. Also, it should be noted that our algorithm does *not* fail in the presence of these difficult cases; but does generate additional tessellation.

Name	Oracle	# Tris	FPS
Torus (Figure 1(b))	True	56448	14
Torus (Figure 1(c)	Adaptive	34377	21
Venus (not shown)	True	51200	12
Venus (Figure 8(a))	Adaptive	21959	20
Venus (Figure 8(b))	Adaptive	9269	30
Venus (Figure 8(c))	Adaptive	2516	85
Earth (Figure 11(c))	True	204800	4
Earth (Figure 11(b))	Adaptive	31374	20

Table 1: Run-time statistics. The "True" in the Oracle column refers to an oracle that always returns 1, giving a uniform tessellation to the maximum depth. The "Adaptive" oracle is the one described in Section 5 using the screen space distance measured in x and y.

7 Conclusions

Implementation of displacement mapping in hardware would lower bandwidth requirements for complex models and would also permit simple adaptive tessellation of complex models to be performed without software sup-

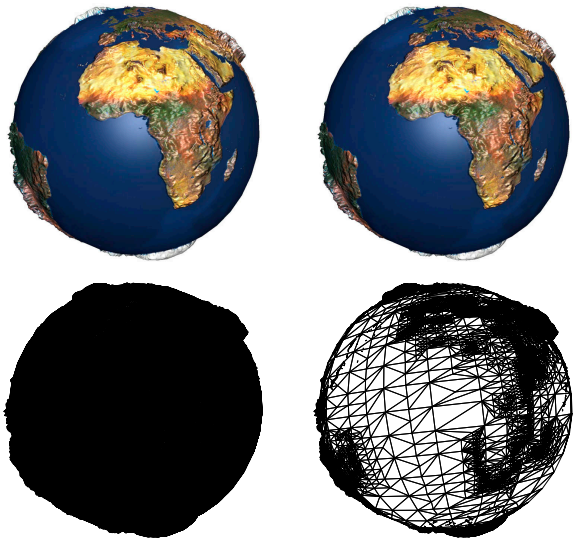


Figure 11: Earth example: (a) uniform tessellation and (b) adaptive tessellation.

port. We have presented an algorithm for implementing displacement mapping, using adaptive tessellation and hierarchical interval bounding of the displacement function. Our interval-based subdivision oracle is more robust than previous approaches that did not consider variation of the displacement over the interior of the input triangles.

Our approach is simple enough to run in real time in software, but would also be suitable for hardware implementation. Since the software implementation runs in real time, it could be added to a driver. Adoption of such an architecture would permit eventual adoption of a hardware displacement unit, while permitting partial hardware acceleration even on current architectures. A conceptual model of the pipeline including the displacement unit is shown in Figure 12. The vertex shading unit would need to be placed ahead of the displacement unit. The displacement unit generates a potentially different set of output vertices each frame. Shading these continuously changing output vertices could possibly produce frame-to-frame shading artifacts.

Figure 12: Location of displacement unit in pipeline.

Our approach could be extended in various ways. We only consider splitting edges at their midpoints. This leads to a simple interpolator and a symmetric edge split, but splitting at other than the midpoint might lead to higher convergence rates. It might be interesting, for instance, to perform edge detection on the displacement map and perform edge splits near locations of known discontinuities. Also, as stated in the introduction, it should be possible to extend our scheme to the adaptive tessellation of arbitrary procedural parametric geometry [16], by simply adding a unit to perform programmable interval analysis of "geometry shader" functions. This should probably be performed using affine arithmetic, which would require compiler support for efficient implementation, but would not require a radically new shader processing architecture. It might also be interesting to extend the representation of tabulated displacement functions to a form suitable for affine arithmetic, by storing slope information as well as bounds. Finally, if information could be attached to edges rather than just vertices it might be possible to come up with better oracles that use, for instance, the dihedral angle between adjacent faces.

Acknowledgements

This research was sponsored by research grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), Communications and Information Technology Ontario (CITO), Canadian Foundation for Innovation (CFI), Ontario Innovation Trust (OIT) and Bell University Labs (BUL). We also gratefully acknowledge NVIDIA's hardware donations and the ever helpful members of the Computer Graphics Lab.

References

[1] A. J. Chung and A. J. Field. A simple recursive tessellator for adaptive surface triangulation. *Journal of Graphics Tools*, 5(3):1–9, 2000.

[2] Robert L. Cook. Shade tress. *Proceedings of SIGGRAPH 84*, pages 223–231, 1984.

[3] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *Proceedings of SIGGRAPH 87*, pages 95–102, 1987.

[4] M. Doggett, A. Kugler, and W. Strasser. Displacement mapping using scan conversion hardware architectures. *Computer Graphics Forum*, 20(1):13–26, 2001.

[5] Michael Doggett and Johannes Hirche. Adaptive view dependent tessellation of displacement maps. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 59–66, 2000.

[6] Stefan Gumhold and Tobias Hüttner. Multiresolution rendering with displacement mapping. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 55–66, 1999.

[7] Wolfgang Heidrich and Hans-Peter Seidel. Ray-tracing procedural displacement shaders. *Graphics Interface '98*, pages 8–16, 1998.

[8] Jan Kautz and Hans-Peter Seidel. Hardware accelerated displacement mapping for image based rendering. *Graphics Interface 2001*, pages 61–70, 2001.

[9] Aaron Lee, Henry Moreton, and Hugues Hoppe. Displaced subdivision surfaces. *Proceedings of SIGGRAPH 2000*, pages 85–94, 2000.

[10] J. R. Logie and J. W. Patterson. Inverse displacement mapping in the general case. *Computer Graphics Forum*, 14(5):261–273, 1995.

[11] J. W. Patterson, S. G. Hoggar, and J. R. Logie. Inverse displacement mapping. *Computer Graphics Forum*, 10(2):129–139, 1991.

[12] Hans Køhling Pedersen. Displacement mapping using flow fields. *Proceedings of SIGGRAPH 94*, pages 279–286, 1994.

[13] Matt Pharr and Pat Hanrahan. Geometry caching for ray-tracing displacement maps. *Eurographics Rendering Workshop 1996*, pages 31–40, 1996.

[14] Gernot Schaufler and Markus Priglinger. Efficient displacement mapping by image warping. *Eurographics Rendering Workshop 1999*, pages 175–186, 1999.

[15] Brian Smits, Peter Shirley, and Michael M. Stark. Direct ray tracing of displacement mapped triangles. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 307–318, 2000.

[16] John M. Snyder. *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design Using Interval Analysis*. Academic Press, 1992.

[17] Xiaochuan Corina Wang, Jérome Maillot, Eugene Fiume, Victor Ng-Thow-Hing, Andrew Woo, and Sanjay Bakshi. Feature-based displacement mapping. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 257–268, 2000.