Automatic Pre-Tessellation Culling

JON HASSELGREN, JACOB MUNKBERG and TOMAS AKENINE-MÖLLER
Lund University and Intel Corporation

Graphics processing units supporting tessellation of curved surfaces with displacement mapping exist today. Still, to our knowledge, culling only occurs *after* tessellation, that is, after the base primitives have been tessellated into triangles. We introduce an algorithm for *automatically* computing tight positional and normal bounds on the fly for a base primitive. These bounds are derived from an arbitrary vertex shader program, which may include a curved surface evaluation and different types of displacements, for example. The obtained bounds are used for backface, view frustum, and occlusion culling *before* tessellation. For highly tessellated scenes, we show that up to 80% of the vertex shader instructions can be avoided, which implies an "instruction speedup" of 5×. Our technique can also be used for offline software rendering.

Categories and Subject Descriptors: I.3.1 [**Computer Graphics**]: Hardware Architecture—*Graphics processors*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Hidden line/surface removal*

General Terms: Algorithms

Additional Key Words and Phrases: Rasterization, tessellation, culling, hardware

ACM Reference Format:

Hasselgren, J., Munkberg, J., and Akenine-Möller, T. 2009. Automatic pre-tessellation culling. ACM Trans. Graph. 28, 2, Article 19 (April 2009), 10 pages. DOI = 10.1145/1516522.1516530 http://doi.acm.org/10.1145/1516522.1516530

1. INTRODUCTION

To provide rich surface representations for real-time rendering, it is expected that most graphics hardware in the near future will have support for tessellation of curved surfaces with displacement mapping. The Xbox 360 [Doggett 2005] and the ATI Radeon HD 2000 series [Tatarchuk et al. 2007] already have support for this. A primitive with a triangular or square domain is tessellated, and barycentric coordinates are forwarded to the vertex shader, which may compute an arbitrary position based on these coordinates, and more. To the best of our knowledge, these systems only perform culling *after* tessellation using the conventional graphics pipeline. Clearly, it would be advantageous to cull *before* tessellation occurs as illustrated in Figure 1.

Over the years, culling techniques have seen many uses in both real-time graphics and offline rendering. In general, RenderMan implementations [Apodaca and Gritz 2000; Cook et al. 1987] use culling on many different levels. However, the details may vary for different implementations. View frustum and occlusion culling are performed, often prior to tessellation, and splitting of primitives may also occur. Backface culling is usually done after tessellation. Wexler et al. [2005] describe a GPU-optimized implementation, where (among other things) occlusion queries are used to accelerate rendering. However, to bound a displaced surface in RenderMan, the user either has to provide the renderer with a conservative upper bound, or the displacement shader is executed on micropolygons, and exact bounds computed from these [Apodaca and Gritz 2000]. In this latter case, no culling occurs before tessellation.

Shirman and Abi-Ezzi [1993] use cones to bound a set of normals on a patch, and can thus perform efficient backface culling. Kumar and Manocha [1996] derive a different method for backface culling of curved surfaces, and use a conservative technique to bound the normals and then test for culling. However, neither of these techniques can handle arbitrary surface evaluations automatically on the fly. Han et al. [2005] describe an alternative GPU implementation, where the part of the vertex shader that computes the position of a vertex is executed first. After that follows backface culling. If the triangle is culled then unnecessary lighting calculations are avoided. Our goal is similar, but we want to perform culling before tessellation even occurs.

There is a wealth of literature on adaptive on-the-fly tessellation, and as our work can be combined with such techniques, we only list some of them. Doggett and Hirche [2000] use a summed-area table of the displacement map and a normal test to guide the tessellation. A similar approach is to use interval arithmetic and interval textures to focus the tessellation efforts [Moule and McCool 2002]. To provide a continuous level of detail, Moreton [2001] introduces *fractional tessellation* where tessellation factors are specified as floating-point numbers per triangle edge. This allows for adaptive tessellation across a mesh, and similar techniques are used in modern GPUs [Tatarchuk et al. 2007].

T. Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation. In addition, we acknowledge support from the Swedish Foundation for Strategic Research.

Authors' address: J. Hasselgren, J. Munkberg, and T. Akenine-Möller, Lund University, Box 117, S-221 00 Lund, Sweden; email: {jon,jacob,tam}@cs.lth.se

DOI 10.1145/1516522.1516530 http://doi.acm.org/10.1145/1516522.1516530

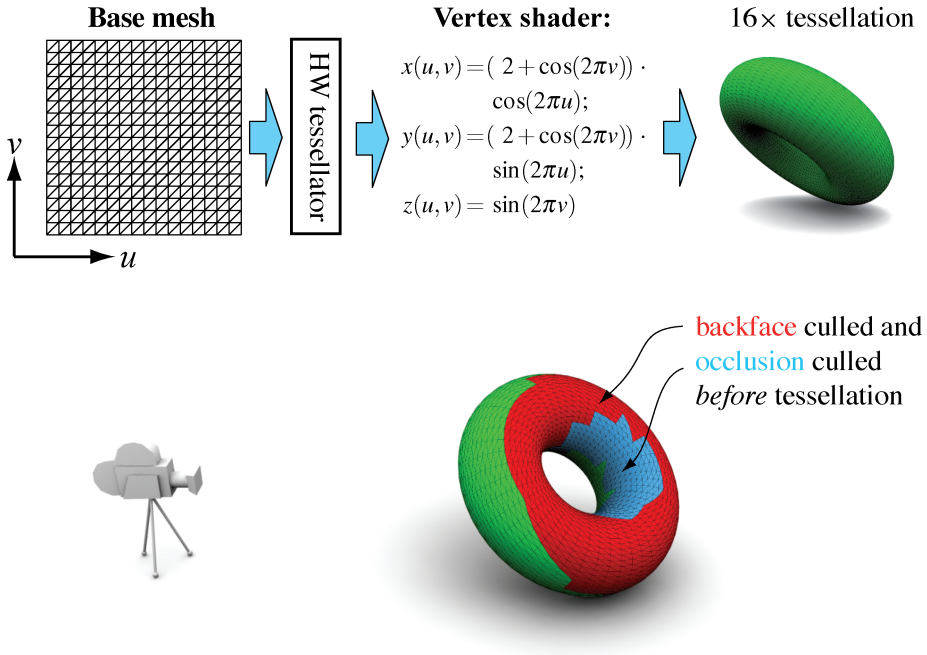


Fig. 1. GPUs with tessellation hardware are given a base mesh over a parameter space, (u, v), as input. In this case, the tessellator increases the number of triangles by a factor of 16, and a vertex shader evaluates a point on a torus surface. In the lower part, we visualize the base triangles that our culling algorithm *automatically* can avoid to tessellate, and where vertex shader evaluations can be avoided. We are able to cull 56% of the triangles prior to tessellation.

In contrast to the previous work described before, we focus on presenting a single automatic solution. Our article contributes with a novel pre-tessellation backface, view frustum, and occlusion culling method which is:

—fully automatic, based only on arbitrary vertex shader code which can include, for example, deformations, curved surfaces, and displacement mapping;
—implemented with tightly bounded arithmetic on triangular domains; and
—suitable for implementing in both hardware and software rendering systems.

Next, we describe our algorithm in detail.

2. TESSELLATION CULLING

The goal of our work is to efficiently avoid tessellating the majority of surfaces which do not contribute to the final image. This occurs when a surface is backfacing, outside the view frustum, or occluded by previously rendered surfaces. Furthermore, we believe it is of utmost importance that fully *arbitrary* vertex displacement shaders can be handled in a completely *automatic* way. In this section, we present a novel algorithm for this. Without loss of generality, we restrict ourselves to triangular domains and tessellation.

2.1 Overview

We extend the current GPU tessellation pipeline [Tatarchuk et al. 2007] with our new culling unit as illustrated in Figure 2. Note that this type of pipeline is also rather similar to offline rasterization pipelines. Without our culling unit, *base triangles* are first injected into the pipeline, and these can be tessellated to a desired number of triangles by the hardware. For each created vertex, the tessellator forwards its barycentric coordinates, (u, v), down the pipeline. The vertex shader then computes the position, $\mathbf{p}(u, v)$, of each vertex as a function of its barycentric coordinates. This may include, for example, the evaluation of a Bézier triangle with texture displacement, procedural noise, and transform matrices. Each term can also depend on a time parameter in order to animate a water surface, for example.

Our culling algorithm works as outlined in Figure 3. First, we analyze the vertex shader program and isolate all instructions that are used to compute the vertex position. We then compute geometric bounds for this position over an entire base triangle, and use these bounds to perform the culling.

Recently, it has been shown [Hasselgren and Akenine-Möller 2007] that pixel shaders can be executed, bounded, and culled over a block of pixels using interval arithmetic [Moore 1966]. In this case, the programs used for culling are often short (terminated by a `KIL` instruction). However, in our context, the shader programs are significantly more complex, and therefore we use Taylor models [Berz and Hoffstätter 1998] to approximate the shader function over the triangle domain. We then use Bernstein expansion [Hungerbühler and Garloff 1998] to compute tight bounding boxes for the Taylor models, and use these bounding boxes for culling.

In the following, we first present some background on Taylor models in Section 2.2. Then follows an algorithm for computing tight polynomial bounds in Section 2.3, and our program analysis and generation in Section 2.4. In Section 2.5, we describe how selective execution of our culling can be done, and finally, the culling algorithms are described in Section 2.6.

2.2 Taylor Arithmetic

Taylor arithmetic has seen little use in computer graphics research, but there is a recent exception in collision detection [Zhang et al. 2007]. Interval arithmetic [Moore 1966], on the other hand, has been

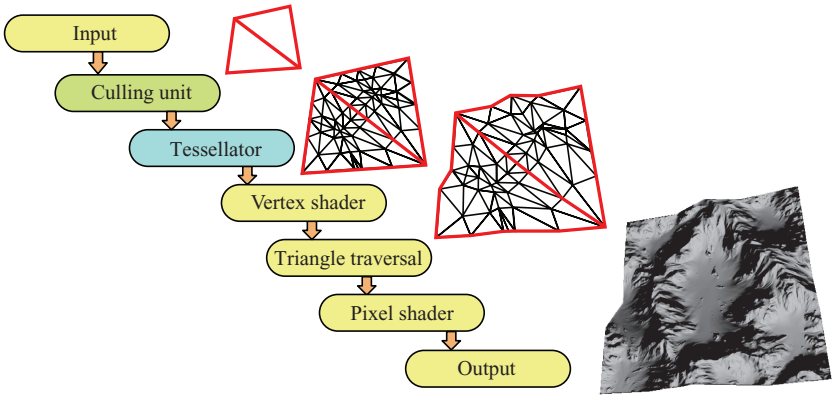
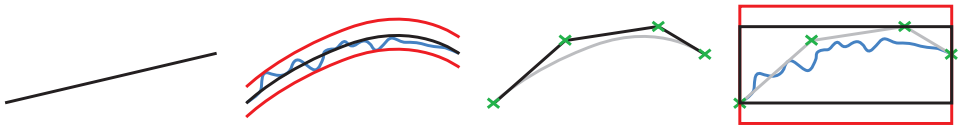


Fig. 2. To support tessellation in the GPU pipeline, a tessellation unit has recently been added. We propose to add the culling unit, which automatically determines whether tessellation of a base primitive can be avoided.



(a) base triangle (b) Taylor model (c) Bernstein expansion (d) final bounds

Fig. 3. Algorithm overview: (a) a base triangle (seen from the side) with precomputed tessellation factors is sent to the tessellation unit; (b) by expressing the vertex program in Taylor form (polynomial + interval remainder), a conservative estimate of the surface is obtained; (c) the Taylor polynomial is expanded in Bernstein form for efficient range bounding (using the convex hull property); (d) finally, by adding the interval remainder term from the Taylor model to the Bernstein bounds, conservative surface bounds (red) are obtained.

used extensively in graphics. Intervals are used in Taylor models, and the following notation is used for an interval $\hat{a}$.

$$\hat{a} = [\underline{a}, \overline{a}] = \{x \mid \underline{a} \leq x \leq \overline{a}\} \tag{1}$$

Given an $n+1$ times differentiable function, $f(u)$, where $u \in [u_0, u_1]$, the Taylor model of f is composed of a Taylor polynomial, T_f, and an interval remainder term, $\hat{r}_f$ [Berz and Hoffstätter 1998]. An nth-order Taylor model, here denoted $\tilde{f}$, over the domain $u \in [u_0, u_1]$ is then

$$\tilde{f}(u) = \underbrace{\sum_{k=0}^{n} \frac{f^{(k)}(u_0)}{k!} \cdot (u - u_0)^k}_{T_f} + \underbrace{[\underline{r_f}, \overline{r_f}]}_{\hat{r}_f} = \sum_{k=0}^{n} c_k u^k + \hat{r}_f. \tag{2}$$

This representation is called a Taylor model, and is a conservative enclosure of the function, f over the domain $u \in [u_0, u_1]$.

Similarly to interval arithmetic, it is also possible to define arithmetic operators on Taylor models, where the result is a conservative enclosure (another Taylor model) as well [Berz and Hoffstätter 1998]. Addition is defined as follows: Assume that $f + g$ shall be computed and these functions are represented as Taylor models, $\tilde{f} = T_f + \hat{r}_f$ and $\tilde{g} = T_g, \hat{r}_g$. The Taylor model of the sum is then

$$\widetilde{f + g} = (T_f + T_g) + (\hat{r}_f + \hat{r}_g). \tag{3}$$

Note here that $T_f + T_g$ is an addition of two polynomials.

Similarly, for multiplication of a Taylor model, $\tilde{f}$, by a scalar value, λ, we get that

$$\widetilde{\lambda \cdot f} = (\lambda \cdot T_f) + (\lambda \cdot \hat{r}_f). \tag{4}$$

Multiplication between two Taylor models is more complicated. Assume again that we want to compute $f \cdot g$ where f and g are represented by Taylor models. The Taylor model of the product is then

$$\widetilde{f \cdot g} = \underbrace{\underline{T_f \cdot T_g}}_{T_{f \cdot g}} + \underbrace{B\left(\overline{T_f \cdot T_g}\right) + B\left(T_f\right) \cdot \hat{r}_g + B\left(T_g\right) \cdot \hat{r}_f + \hat{r}_f \cdot \hat{r}_g}_{\hat{r}_{f \cdot g}}. \tag{5}$$

The polynomial part of this equation, $T_{f \cdot g}$ is simply the multiplication of the polynomials T_f and T_g, but clamped (denoted $\underline{T_f \cdot T_g}$) so that all terms of higher order than the Taylor model have been removed.

The remainder has several contributing terms. First, we have the part of the polynomial multiplication that overflows and has terms only of higher order than the Taylor model ($\overline{T_f \cdot T_g} = T_f \cdot T_g - \underline{T_f \cdot T_g}$). Note that we want the remainder term on interval form, and therefore we must bound the overflow of the polynomial multiplication over the domain (this is indicated by the bounding operator, $B()$). To compute the bounds, we directly evaluate overflowing terms using interval arithmetic and accumulate them to the remainder. More complex bounding computations, such as the one presented in Section 2.3, are possible, but since multiplication is such a frequent operation, we must ensure that it is fast to compute its bounds. The other terms found in the remainder involve computing the bounds of T_f and T_g and are treated similarly to the overflow from the polynomial multiplication. It should be noted that one or more of the terms in the remainder often are zero. For instance, if $\hat{r}_f$ or $\hat{r}_g$ is zero, then the corresponding terms will be zero as well. As an optimization, we detect these cases and avoid the computations.

By using Taylor expansion and the addition and multiplication operations presented previously we can derive more complex

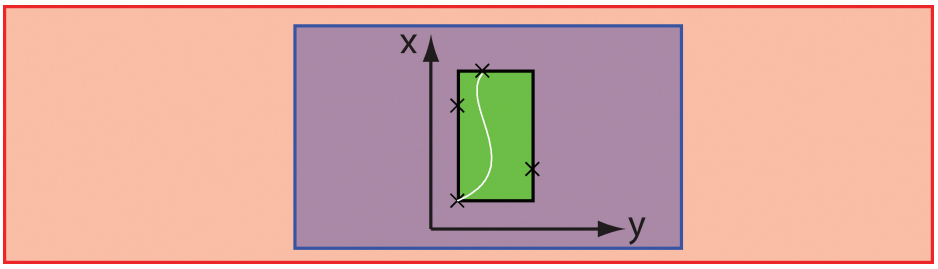


Fig. 4. A comparison of the bounds for a parametric curve $(p_x(t), p_y(t))$ of degree 3 in t for interval aritmethic (red), affine arithmetic (blue), and Taylor models with Bernstein bounds (green).

arithmetic operators, like sine, log, exp, reciprocal, and so on. We refer to the work of Berz and Hoffstätter [1998] and Makino and Berz [2003] for more details.

Motivation. The motivation for us to use Taylor models is that curved surfaces and subdivision schemes are often based on polynomials. Polynomial computations can be represented exactly by Taylor models (provided they are of high enough order) which leads to very tight bounds. Previous work on shader analysis [Greene and Kass 1994; Heidrich et al. 1998; Hasselgren and Akenine-Möller 2007] have successfully used interval and affine arithmetic, which are computationally less expensive than Taylor models. However, note that they subdivide the domain into small tiles before evaluating the bounded shader. In contrast, we must bound the shader over the entire domain (the base triangle) in a *single* evaluation, and consequently we need much tighter bounds. A side-by-side comparison between the tightness of interval arithmetic, affine arithmetic, and Taylor models can be found in the example in Section 2.3.

Taylor models also provide a flexible framework since it is essentially a superset of interval and affine arithmetics. It allows us to tweak interval sharpness versus computational overhead by changing the order of the Taylor model. Orders zero and one correspond to interval arithmetics, and generalized interval arithmetics [Hansen 1975], which is similar to affine arithmetics.

2.3 Tight Polynomial Bounds

Our approach to tessellation culling is to evaluate the vertex shader using Taylor arithmetic as described earlier. We execute the part of the shader that affects the position attribute using Taylor arithmetic. This results in a Taylor model for each of the components in the position attribute: (x, y, z, w). To find a geometrical bounding box, one could then find local minima and maxima for each of these. However, this requires numerical, iterative methods for polynomials of degree $n > 4$, and also quickly becomes impractical due to the dependence on the two parametric coordinates, (u, v).

Instead, we use a faster, conservative approach which still produces tight bounds. The resulting Taylor polynomials are in power form, and the core idea is to convert these to Bernstein form. The convex hull property of the Bernstein basis guarantees that the actual surface or curve of the polynomial lies inside the convex hull of the control points. Thus, we compute a bounding box by finding the minimum and maximum control point value in each dimension.

In practice, we obtain bivariate polynomials from the vertex shader evaluation using Taylor arithmetic, and for a single component (e.g., x) this can be expressed in the power basis as follows (where we have omitted the remainder term, $\hat{r}_f$, for clarity).

$$p(u, v) = \sum_{i+j \le n} c_{ij} u^i v^j \tag{6}$$

We want to transform Eq. (6) into the Bernstein basis

$$p(u, v) = \sum_{i+j \le n} p_{ij} B^n_{ij}(u, v), \tag{7}$$

where $B^n_{ij}(u, v) = \binom{n}{i}\binom{n-i}{j} u^i v^j (1-u-v)^{n-i-j}$ are the Bernstein polynomials in the bivariate case over a triangular domain. We can convert a polynomial in the power basis form into the Bernstein form using the following formula [Hungerbühler and Garloff 1998].

$$p_{ij} = \sum_{l=0}^{i} \sum_{m=0}^{j} \frac{\binom{i}{l}\binom{j}{m}}{\binom{n}{l}\binom{n-l}{m}} c_{lm} \tag{8}$$

To compute a bounding box, we simply compute the minimum and the maximum value over all p_{ij} for each dimension, x, y, z, and w. This gives us a bounding box, $\hat{\mathbf{b}} = (\hat{b}_x, \hat{b}_y, \hat{b}_z, \hat{b}_w)$, in clip space. Next, we will give an example of the effectiveness of this technique when compared to interval and affine arithmetic.

Example. Assume we have the following parametric curve, $\mathbf{p}(t) = (p_x(t), p_y(t))$, where $t \in [0, 1]$, $p_x(t) = 1 + 3t + 3t^2 - 2t^3$, and $p_y(t) = 1 + 9t - 18t^2 + 10t^3$. We will illustrate how interval and affine arithmetic compare to our tight polynomial bounds when computing a two-dimensional axis-aligned bounding box of this curve over the domain, $t \in [0, 1]$. The resulting bounds are visualized in Figure 4. Using standard interval arithmetic, we obtain $\hat{p}_x = [1, 1] + [0, 3] + [0, 3] + [-2, 0] = [-1, 7]$ and $\hat{p}_y = [1, 1] + [0, 9] + [-18, 0] + [0, 10] = [-17, 20]$, and these two intervals represent a box with an area of 296. Similarly, applying affine arithmetic [Comba and Stolfi 1993] on the same example gives us $p_x = 3 + 9/4\varepsilon_1 + 1/2\varepsilon_2 - 3/4\varepsilon_3$ and $p_y = 9/4 - 3/4\varepsilon_1 - 13/4\varepsilon_2 + 15/4\varepsilon_3$, where $\varepsilon_i \in [-1, 1]$ are noise symbols. The bounding box becomes $\hat{p}_x = [-0.5, 6.5]$, $\hat{p}_y = [-5.5, 10]$, which represents a box with an area of 108.5. To apply our tight polynomial bounds, we first observe that the polynomials for p_x and p_y are essentially in Taylor form already. Our strategy is therefore to rewrite these on Bernstein form: $p_x(t) = \mathbf{1} \cdot (1-t)^3 + \mathbf{2} \cdot 3(1-t)^2 t + \mathbf{4} \cdot 3(1-t)t^2 + \mathbf{5} \cdot t^3$, and $p_y(t) = \mathbf{1} \cdot (1-t)^3 + \mathbf{4} \cdot 3(1-t)^2 t + \mathbf{1} \cdot 3(1-t)t^2 + \mathbf{2} \cdot t^3$, where the control points have been typeset in boldface. The bounding box is then found as the minimum and maximum of the control points in x and y. This gives us $\hat{p}_x = [1, 5]$ and $\hat{p}_y = [1, 4]$, which has an area of 12. The tightest fit axis-aligned box has $\hat{p}_x = [1, 5]$ and $\hat{p}_y = [1, 2.37]$, with an area of 5.48.

2.4 Program Analysis and Generation

In a graphics pipeline with a tessellation unit, the vertex shader receives barycentric coordinates and the associated base triangle information, and then outputs a vertex position in clip space. In the simplest vertex shader, the vertex position is computed by interpolating the base triangle vertices, using the barycentric coordinates, and transforming this position into clip space by a matrix multiplication. In the general case, the vertex position is displaced using an arbitrary function (of the barycentric coordinates) before the clip space transform.

We want to bound this position over the entire barycentric domain, and must therefore evaluate the vertex shader output for every possible barycentric coordinate, since this is the only input that varies over the base triangle. To accomplish this, we reformulate the vertex shader using Taylor models.

We represent each Taylor model as a coefficient list. Each coefficient has a scalar value and an id i, indicating that it is the coefficient of the x^i term. For example, the polynomial $4 + 3x + 0.5x^2$ over the domain $x \in [0, 1]$ would be represented, as a Taylor model of order 2, by the list $[\{4, 0\}, \{3, 1\}, \{0.5, 2\}, \hat{r} = 0]$. It could also be represented as a Taylor model of order one as $[\{4, 0\}, \{3, 1\}, \hat{r} = [0, 0.5]]$.

Our only varying input, the barycentric coordinates, are expressed as two-dimensional Taylor models. Generalizing the list representation from before to two dimensions so that a polynomial term $\alpha x^i y^j$ is represented by a coefficient $\{\alpha, i, j\}$, we can write the barycentric coordinates as two-dimensional Taylor models.

$$\begin{aligned} u &= 0 + 1 \cdot u + 0 \cdot v = [\{1, 1, 0\}] \\ v &= 0 + 0 \cdot u + 1 \cdot v = [\{1, 0, 1\}] \\ w &= 1 - 1 \cdot u - 1 \cdot v = [\{1, 0, 0\}, \{-1, 1, 0\}, \{-1, 0, 1\}] \end{aligned}$$

These are Taylor models of order 1 ($i, j \leq 1$) over the domain $u \in [0, 1]$, $v \in [0, 1]$. Note that no remainder is needed.

We then proceed by evaluating all instructions using Taylor models. We will briefly exemplify the implementation of addition and multiplication of Taylor models, as more complex operations will be expressed in these in the end.

Addition. Addition is done by adding the polynomial part of each Taylor model. Our internal representation of the polynomial part is a list of nonzero coefficients. Thus, the polynomial addition essentially becomes a sparse vector addition at runtime. Here is an example.

$$\begin{aligned} u + w &= [\{1, 0, 0\}, \{\mathbf{1} - \mathbf{1}, 1, 0\}, \{-1, 0, 1\}] \\ &= [\{1, 0, 0\}, \{-1, 0, 1\}] = 1 - v. \end{aligned} \tag{9}$$

Note that we only need to perform additions for nonzero terms existing in both u and w, as the other terms can be handled using variable renaming. The remainder term, if nonzero, is handled using normal interval arithmetic. A more realistic shader would include linear interpolation between two, at compile-time unknown, positions. This requires us to work with variables rather than constants. Thus the example becomes

$$\begin{aligned} p_1 u + p_0 w &= [\{p_0, 0, 0\}, \{\mathbf{p_1} - \mathbf{p_0}, 1, 0\}, \{-p_0, 0, 1\}] \\ &= p_0 + (p_1 - p_0)u - p_0 v. \end{aligned} \tag{10}$$

Multiplication. Here, we loop over the nonzero components in one Taylor model and multiply it by all nonzero components in the other. Thus, the runtime complexity is roughly $O(a \cdot b)$ multiplications, where a and b are the number of nonzero coefficients in each of the two polynomials being multiplied. We bound the remainder terms using interval arithmetics. This can be optimized by exploiting that our domain is $(u, v) \in [0, 1]$, as all multiplications by zero can be omitted. For multiplication, the order of the Taylor model will increase, so we have the choice to bound the higher-order terms and add to the remainder, or increase the order of the model. A higher-order Taylor model has more precision (polynomials up to the order of the model can be represented exactly), but is also more costly computationally. With the sparse list representation given before, we can use a fixed order and models of lower orders will not have any computational overhead, as only nonzero terms are stored and used in the arithmetic operations.

Polynomial displacement shaders (Bézier surfaces) are simply a sequence of Taylor multiplications and additions, and elementary functions can also be bounded by Taylor models. Like standard Taylor expansions, a higher-order representation leads to tighter bounds. Once all arithmetic operations have been converted to Taylor form, we express them using regular vertex shader code. Therefore, we do not need to introduce any new specialized instruction set for our bounding shader. However, the bounding shaders will be significantly longer than the corresponding vertex shader.

Finally, our program analysis gives us a polynomial approximation of the vertex position attribute. We then compute its bounds using the algorithm in Section 2.3. Once again, we generate the necessary vertex shader code for this operation.

Discussion. Program analysis is done in the exact same way as a standard implementation [Berz and Hoffstätter 1998] of Taylor models, with the exception that we need to treat symbolic constants (variables) rather than values, and we need to emit code rather than executing the operations.

It should be noted that the Taylor models for the barycentric coordinates are the same for all base triangles, and thus we can treat them as constants rather than varying input. This means that the order for all Taylor instructions can be computed statically at compile time. Furthermore, we can do most standard optimizations (for example, exploiting $c \cdot 0 = 0$, and $c + 0 = c$), as well as all control flow that is internally needed in the Taylor model computations, at compile time. This greatly increases the runtime shader performance.

In conclusion, the complexity of each Taylor operation is highly dependent on the "order" of the vertex shader. For instance, for a program with only interpolation and a matrix multiplication, the Taylor models will have no nonzero coefficients over order one. In contrast, cubic Bézier triangle evaluation uses polynomials of degree 3, and consequently the Taylor models will have more higher-order coefficients. The instruction ratio between the culling program and vertex shader grows for more complex shaders (see Section 4). Note that we can determine the number of instructions during compile time. Thus we can compile the program, see how expensive it gets, and only trigger culling if there is potential for performance gain.

2.4.1 *Texture Mapping.*

Shaders using texture map lookups are problematic, as the texture map may contain an arbitrarily complex function. However, texture mapping is an important feature, as displacement mapping is a prime use-case of a tessellation unit.

We implement texture mapping using interval-based texture lookups [Moule and McCool 2002; Hasselgren and Akenine-Möller 2007], which computes a bounding interval for the texture in a given region. If, for example, we want to displace a surface in the direction of an interpolated normal, then the texture interval will be used in subsequent arithmetic computations. Therefore, we must convert the interval to Taylor form.

A naive way of doing this is to treat the texture lookup in the interval remainder term, $\hat{r}_f$, of the Taylor model. However, we found

this approach unsatisfactory, as the remainder term in Taylor models is treated using standard interval arithmetic, which causes the bounds to grow rapidly. Instead, we treat every texture lookup as a functional parameter. Specifically, instead of treating the shader as a two-dimensional Taylor model

$$\tilde{f}(u,v) = \sum_{i+j\leq n} c_{ij} u^i v^j + \hat{r}_f, \tag{11}$$

we treat it as a three-dimensional Taylor model

$$\tilde{f}(u,v,a(u,v)) = \sum_{i+j+k\leq n} c_{ijk} u^i v^j a(u,v)^k + \hat{r}_f, \tag{12}$$

where $a(u,v)$ is an unknown (texture map) function defined over the interval domain (which we computed in the interval texture lookup). By increasing the dimensionality of the Taylor models, we can track correlations for arithmetic operations which depend on texture lookups. In effect we defer the interval evaluations to the last part of the shader, which consists of the bounds computations. To support an arbitrary number of texture lookups, all Taylor arithmetic, as well as the tight bounding computations of Section 2.3, can be generalized to n-dimensional domains. For details, we refer to the work by Berz and Hoffstätter [1998] and Lin and Rokne [1996].

2.4.2 *Branching and Looping.* We can easily support branching and looping when the conditional expression is a value (or equivalently, a zero:th order Taylor model with no remainder). In this case, it is uniquely determined which branch we should take, or how many iterations of a loop we should perform. A typical example would be looping over an, at compile time unknown, number of fractal noise octaves.

We can also handle branches with Taylor models for conditional expressions. In such cases we compute quick bounds for the Taylor model based on interval arithmetic (see multiplication in Section 2.2). If the bounds of the condition are ambiguous, we must execute both branches. Furthermore, if a variable is assigned a value in both branches, we must assign it the union of those values. A union of two Taylor models could be computed by computing the average of their polynomial parts, and growing the rest term accordingly to enclose both polynomials.

A construct that we cannot handle is loops with a Taylor model as the conditional expression. One example is iterative computation on the barycentric coordinates that loops until the result has converged. As previously explained by Hasselgren and Akenine-Möller [2007], such computations are not guaranteed to converge when bounded arithmetics are used, and we may get an infinite loop. Fortunately, we can easily detect these cases and simply disable our culling.

2.5 Selective Execution

We have observed that the bounding shaders are roughly 3–15× more expensive than the corresponding vertex shader in terms of instructions. Since this cost is rather significant, it makes sense to execute the bounding shader only in regions where we are likely to improve overall performance. A statistical analysis shows that it is beneficial to execute the bounding shader if the following holds:

$$\frac{c(cull)}{c(vertex)} \leq p(cull)\cdot n, \tag{13}$$

where $\frac{c(cull)}{c(vertex)}$ is the cost ratio between the cull and the vertex program, $p(cull)$ is the probability that a base triangle is culled, and n is the number of vertices that will be generated during tessellation.

2.6 Culling

In this section, we will describe how the actual culling is performed. We want to emphasize that the culling algorithm per se is not a novel contribution. However, some details are given here for the sake of completeness. Recall that the output from the bounding shader program are geometrical bounds: $\tilde{\mathbf{p}}(u,v) = (\tilde{p}_x, \tilde{p}_y, \tilde{p}_z, \tilde{p}_w)$, namely, four Taylor models. As described earlier, we use the convex hull property of the Bernstein form to obtain a bounding box from these Taylor models. This box is denoted $\hat{\mathbf{b}} = (\hat{b}_x, \hat{b}_y, \hat{b}_z, \hat{b}_w)$, where each element is an interval, for example, $\hat{b}_x = [\underline{b_x}, \overline{b_x}]$.

2.6.1 *View Frustum Culling.* For view frustum culling, we simply need to test the geometrical bounds against the planes of the frustum. Since we have the bounding box, $\hat{\mathbf{b}}$, in homogeneous clip space, we can perform the test in this space as well. We use the standard optimization for plane-box tests [Haines and Wallace 1994], where only a single corner of the box is used to evaluate the plane equation. Each plane test then amounts to an addition and a comparison. For example, testing if the box is outside the left plane is done with: $\overline{b_x} + \overline{b_w} < 0$. Since these tests are inexpensive, our culling always starts with the view frustum test.

2.6.2 *Backface Culling.* After the vertex shader has been executed, the vertex $\mathbf{p}$ is in homogeneous clip space (before division by w). This means that the model-view transform has been applied, so the camera position is at the origin. Now, given a point, $\mathbf{p}(u,v)$, on a surface, backface culling is in general computed as

$$c = \mathbf{p}(u,v)\cdot\mathbf{n}(u,v), \tag{14}$$

where $\mathbf{n}(u,v)$ is the normal vector at (u,v). If $c > 0$, then $\mathbf{p}(u,v)$ is backfacing for that particular value of (u,v). For a parameterized surface, the unnormalized normal, $\mathbf{n}$, can be computed as

$$\mathbf{n}(u,v) = \frac{\partial\mathbf{p}(u,v)}{\partial u} \times \frac{\partial\mathbf{p}(u,v)}{\partial v}. \tag{15}$$

After our bounding shader has been executed, we have Taylor models, $\tilde{\mathbf{p}}(u,v)$, for the position. As part of the bounding shader program, these are differentiated as well, resulting in $\partial\tilde{\mathbf{p}}(u,v)/\partial u$ and $\partial\tilde{\mathbf{p}}(u,v)/\partial v$. Finally, the Taylor model of the normal, $\tilde{\mathbf{n}}(u,v)$, is computed using these.

There are two issues with this technique which we need to solve. The first problem arises if $\tilde{\mathbf{p}}(u,v)$ contains a nonzero remainder term, $\hat{r}_p$, since this must be accounted for when computing the partial derivatives. We solve this by using knowledge about the tessellation frequency of the base primitive. Assume that a worst-case sawtooth tessellation pattern is generated by the remainder term, as shown in Figure 5(a). The maximum slope for such a configuration is $(f(x+\Delta x) - f(x) + w)/\Delta x$, where Δx is the shortest edge generated during tessellation and w is the width of the interval remainder term. This expression is bounded by $f'(x) + w/\Delta x$ according to the mean value theorem. Similar reasoning holds for the minimum slope. Thus $\partial\tilde{\mathbf{p}}(u,v)/\partial u$ is bounded by $\partial T_p/\partial u \pm (\overline{r_p} - \underline{r_p})/\Delta x$.

It should be noted that fractional tessellation may introduce edges that are arbitrarily short, since new vertices may be inserted at the positions of old ones. This makes it very hard to bound the derivatives of Taylor models with remainder terms, as we must assume

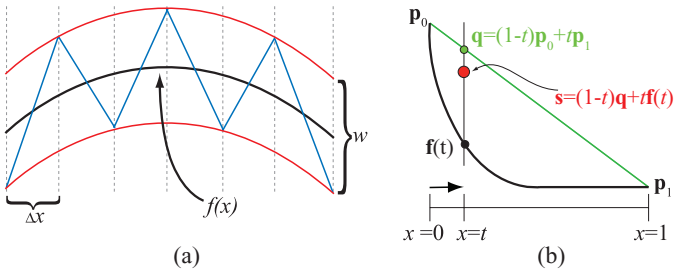


Fig. 5. We must take special care of the interval remainder term when performing backface culling. Figure (a) shows a worst-case derivative of a Taylor model with a polynomial $f(x)$ an interval remainder term with width w. The worst-case derivative that can be introduced by the remainder term is given by the blue sawtooth pattern, which has a period of $2\Delta x$ where Δx is the length of the shortest edge created during tessellation. Figure (b) shows how we alter the original fractional tessellation algorithm to avoid problems that would arise in Figure (a) if Δx is very small.

that $\Delta x = 0$. We propose to modify the fractional tessellation algorithm so that new vertices are inserted in a bilinearly interpolated fashion. As shown in Figure 5(b), we find the point $\mathbf{q}(t)$ by linearly interpolating between the two neighbors $\mathbf{p_0}$ and $\mathbf{p_1}$. Then we interpolate again between the actual position, $\mathbf{f}(t)$, and $\mathbf{q}(t)$. Given this modification, it is possible to show that the derivative from the previous section will behave as if the minimum edge length is half of the edge length in a corresponding uniform tessellator. This means that we can now bound the slope.

The second issue concerns treatment of texture maps. As can be seen in Eq. (12), a Taylor model with texture lookups will contain terms which depend on some unknown texture function $a(u, v)$. When such a term is differentiated, we will obtain partial derivatives $\partial a(u, v)/\partial u$ and $\partial a(u, v)/\partial v$. Our solution is to evaluate these terms using textures of precomputed differentials. These differential textures are treated just like the regular textures described in Section 2.4.1, and increase the dimension of the Taylor models. It should be noted that this increase in dimension is not computationally costly, as we rarely get more than linear dependencies of a texture.

2.6.3 *Occlusion Culling.* Our occlusion culling technique is similar to hierarchical depth buffering [Greene et al. 1993], except that we use only a single extra level (8×8 pixel tiles) in the depth buffer. The maximum depth value, z^{tile}_{max}, is stored in each tile. This is a standard technique in GPUs [Morein 2000] used when rasterizing triangles. We project our clip-space bounding box, $\hat{\mathbf{b}}$, and visit all tiles overlapping this axis-aligned box. At each tile, we perform the classic occlusion culling test: $z^{box}_{min} \geq z^{tile}_{max}$, which indicates that the box is occluded at the current tile if the comparison is fulfilled. The minimum depth of the box, z^{box}_{min} is obtained from our clip-space bounding box, and the maximum depth of the tile, z^{tile}_{max}, from the hierarchical depth buffer (which already exists in a contemporary GPU). Note that we can terminate the testing as soon as a tile is found to be nonoccluded, and that it is straightforward to add more levels to the hierarchical depth buffer. Our occlusion culling test can be seen as a very inexpensive prerasterizer of the bounding box of the triangle to be tessellated. Since it operates on a tile basis, it is less expensive than an occlusion query.

3. IMPLEMENTATION

We have implemented our automatic culling unit in a C++ software framework simulating the GPU pipeline. We execute the bounding shader program before tessellating each base primitive. We noted that both view frustum and backface culling may be realized in the bounding shader, and our implementation generates code for this. The output of our bounding shader is therefore a single Boolean indicating if the base triangle should be culled or not, and a positional bounding box. The bounding box is required for the occlusion culling, which cannot be implemented in vertex shader code as it includes (coarse level) rasterization operations. Occlusion culling is implemented further down the pipeline as a quick rasterization algorithm.

We use fourth-order Taylor models in our program analysis. This gives us an exact representation of the position and normal for cubic polynomial surfaces, which are frequently used. Some examples are curved PN-triangles [Vlachos et al. 2001] and bicubic patches, such as Loop and Shaefer's Catmull-Clark approximation [2007]. Higher-order terms will be handled by the remainder term in the Taylor model.

We believe that our automatic tessellation culling could be implemented in a graphics hardware system at a moderate cost. For a full implementation, we need additional hardware that enables us to do the following:

—execute a bounding shader once per base primitive. The instruction set and program inputs are identical to the vertex shader. With unified shader architectures, this should be fairly straightforward to add.

—perform the occlusion culling described in Section 2.6.3.

—remove a base triangle before tessellation based on a Boolean culling flag.

The remaining tasks can be done either in the bounding shader code or in a preprocessing step in a driver.

A partial implementation of our automatic culling algorithm could be realized on current hardware in two passes. First, we would execute the bounding shader program and use it to compute tessellation factors for the subsequent rendering pass. The tessellation factor can then be set to zero for all culled triangles.

4. RESULTS

Our test setup and results will be presented in this section. We use the software GPU simulator described in the previous section, and render all images in 1920×1280 resolution. Since, to the best of our knowledge, no system exists that can *automatically* perform culling based on vertex shader analysis, cull shader generation, and on-the-fly execution, we decided to compare our system against an "optimal" culling unit. This unit can, for example, backface cull a base triangle only if all tessellated triangles are backfacing. In

practice, such optimal culling uses too many resources and so will not provide much (if any) speedup. However, from a scientific point of view, it is interesting to find out how close to an optimal culling unit our algorithm performs.

To investigate the performance of our algorithm, we use four test scenes, two of which have recently been used in GPU tessellation contexts. These are *Ninja*, *Terrain*, *Figurines*, and *Spike Balls*, as can be seen in Table I. In addition, we decided to use four tessellation rates, giving approximate triangle areas of 8, 4, 2, and 0.5 pixels. We motivate these rates by the fact that GPUs were balanced for eight-pixel triangles already two years ago [Pharr 2006], and the introduction of tessellation units indicates an intention to further decrease the size. Our highest tessellation rate (0.5 pixels) is inspired by production rasterization pipelines [Apodaca and Gritz 2000], which is another possible application of our culling unit.

The Terrain scene is a common usage area for tessellation. A coarse mesh is finely tessellated and displaced. The camera moves over the landscape, and so a fair amount of view frustum culling should be possible. This scene has the most inexpensive bounding shader, which is only 2.8× as expensive as the vertex shader. The Ninja scene uses displacement mapping along an interpolated normal. The model is always inside the view frustum, and so only backface and occlusion culling can occur. Furthermore, the base mesh is highly tesselated, which makes it a rather hard case for our algorithm. A highly tesselated base mesh will not generate as many tesselated vertices, and hence, there is not as much to be gained by the culling. The Figurines scene consists of a set of models using PN-triangles [Vlachos et al. 2001], namely, cubic Bézier triangles. We included this scene to demonstrate that render-time mesh smoothing can be handled efficiently by our culling algorithm. The scene shows a grid of meshes seen from the front and tests all three types of culling. It has the highest number of base primitives, but also has many more separate objects and the most complex geometry. The final scene, Spike Balls, shows PN-triangulated spheres with displacement mapping. This scene has the most expensive bounding shader program, approximately 2400 instructions long. Since everything is inside the view frustum, this scene only uses backface and occlusion culling.

We present our performance figures in Table I. The culling rates show how our culling unit compares to the optimal culling unit described before. Note that our culling unit in some cases performs better than the optimal unit at occlusion culling. This only occurs because the occluded triangles were removed by the backface culling test in the optimal culling unit. For the culling rate figures, we execute our bounding shader for every base triangle in order to make a fair comparison to the optimal culling unit. For the performance figures (instruction speedup), we instead execute the bounding shader based on Eq. (13), where we chose $p(cull) = 0.5$.

It should be noted that the instruction counts for bounding and vertex shaders presented in Table I are the number of *scalar* instructions used, and not vector instructions. The motivation for this is that modern graphics hardware architectures use scalar instructions internally, and achieve parallelism by operating on multiple vertices or pixels instead. Note also that we counted multiplications and additions separately for simplicity. It is, however, likely that the bounding shader programs can be significantly shortened using multiply-add.

It should also be noted that our performance numbers do not include the actual tessellation (i.e., generation of connected vertices) nor execution of instructions in the vertex shader not dealing with computing vertex position (e.g., vertex lighting, tangent space transforms, etc.). In addition, our simple occlusion culling is not included either since it has to be implemented in custom hardware, but given its simple nature it should be very efficient. In summary, we believe that our performance would be even better if these factors were taken into account.

Discussion. Given that our culling is automatically derived from a vertex shader program, we consider our culling rates very high, compared to the optimal culling rates. Note that we have intentionally avoided very simple test scenes where, for example, a detailed, tessellated character is behind a wall. In such cases, our occlusion culling would cull the entire character given that the wall was rendered first. One thing we noted in particular is that backface culling of displacement mapped surfaces is a very hard task (although our algorithm handles the Ninja and Spike Balls scenes fairly well).

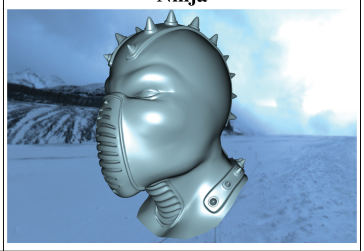
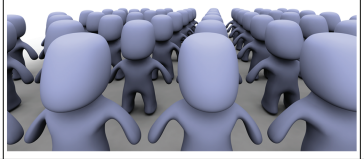
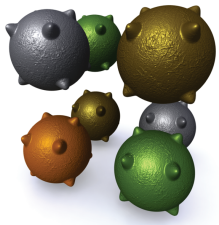
We also compared our culling results with generalized interval arithmetic (first-order Taylor models), and noted that the results directly dropped to 0% culling for the scenes with Bézier surfaces, namely, Figurines and Spike Balls. This clearly motivates our choice of higher-order Taylor models as a suitable arithmetic for bounding shaders. For the remaining test scenes, Terrain and Ninja, we get the exact same behavior for generalized interval arithmetic and higher-order Taylor models. This is to be expected, since our Taylor model implementation never uses higher order than necessary. Thus, the culling performance, and the instruction ratio between the bounding and vertex shader, are identical for these scenes.

Our PN-triangle scenes (Figurines and Spike Balls) use third-order surfaces, similar to the popular Catmull-Clark subdivision schemes. We also performed initial experiments with Loop and Schaefer's [2007] implementation of Catmull-Clark, for the Figurines scene. As the surfaces are bicubic, they contain more high-order terms than corresponding Bézier triangles, and consequently the bounding shader becomes more expensive (6536 instructions bounding shader, and 159 instructions vertex shader, as compared to 1612/126 instructions with PN-triangles). However, we only need to execute the bounding shader once for every quad, in this case. The culling rate was within 2% of that of the PN-triangle version. It should be noted that shaders as long as 6536 instructions may not fit in current instruction caches, which may harm performance. However, we believe that future hardware will be able to handle longer shaders.

As can be seen in Table I, the performance is very high for scenes with view frustum culling (the Terrain and Figurines scenes). In all scenes, we use fractional tessellation and projected edge lengths to determine the tessellation factors for each edge of the base triangles. A fundamental problem with this approach is that we cannot conservatively determine if the edge will be visible or not without tessellating it. Therefore, we chose tessellation factors based on projected edge lengths, without clipping the edges by the view frustum. This leads to highly tessellated base triangles close to the (infinite) near clipping plane, and consequently we get a substantial speedup if we can cull these. This is a general problem in tessellation, and not bound to our application. In fact, using our culling unit makes it much simpler to design a tessellation heuristic, since culling is handled automatically.

Our tessellation heuristic also includes a maximum tessellation factor to avoid generating base triangles being too highly tessellated. This limit is reached when the Terrain scene is rendered at high tessellation rates. Consequently, the vertex rate of the base triangles close to the camera (many of which we can cull) goes down, and this explains why the performance gain (instruction speedup) for this scene decreases when we increase the tessellation rate.

Table I. Performance Evaluation for Our Four Test Scenes

Scene	**Terrain**				**Ninja**			
								
# Base tris	2048				8884			
# Instructions (BS / VS)	140 / 50				825 / 69			
Cull rate (VF/BF/OC)	**31.8%** (26.8 / 0 / 5.0)				**39.6%** (0 / 13.5 / 26.1)			
Opt. cull rate	38.7% (27.8 / 5.3 / 5.6)				53.0% (0 / 40.7 / 12.3)			
Avg. tri area	8.0	4.0	2.0	0.5	8.0	4.0	2.0	0.5
Instruction speedup	**3.39**×	**3.53**×	**3.46**×	**3.23**×	**0.96**×	**0.99**×	**1.08**×	**1.26**×
Scene	**Figurines**				**Spike Balls**			
								
# Base tris	42784 (764 per object)				4480 (560 per object)			
# Instructions (BS / VS)	1612 / 126				2400 / 149			
Cull rate (VF/BF/OC)	**71.0%** (18.0 / 29.6 / 23.4)				**34.9%** (0 / 23.6 / 13.3)			
Opt. cull rate	74.1% (18.2 / 33.3 / 22.6)				59.5% (0 / 48.8 / 10.7)			
Avg. tri area	8.0	4.0	2.0	0.5	8.0	4.0	2.0	0.5
Instruction speedup	**2.03**×	**2.95**×	**3.81**×	**5.55**×	**0.92**×	**1.12**×	**1.24**×	**1.37**×

The instructions row shows the number of scalar instructions for the vertex shader (VS), and the bounding shader (BS). The cull rate row shows how many base primitives our algorithm can automatically cull. The bold figure is the total culling rate, and the numbers in the parentheses are for view frustum (VF), backface (BF), and occlusion (OC) culling. The optimal cull rate row shows the best possible culling. For each scene, we then show instruction speedup for four different tessellation rates, so that the average tessellated triangle area is 8, 4, 2, and 0.5. These figures were computed by dividing the number of instructions to compute the vertex position of every tessellated triangle by the sum of the instructions used by our bounding shader program and the instructions used for the non-culled vertices.

It should be noted that our culling technique is not limited to polynomial surfaces. Figure 1 shows an example of a vertex shader with sines and cosines, wrapping a planar surface to a torus. Still, we can cull 56% (60% optimal) of the triangles before tessellation.

5. CONCLUSION AND FUTURE WORK

The trend in GPU rendering is steadily continuing to close in on the quality of rasterization-based production pipelines. Using hardware to obtain highly tessellated objects is another step in this direction. We are therefore excited about the recent developments in hardware tessellation and, hopefully, our work can be used in future implementations of GPUs to accelerate rendering further. As we have shown, this would give significantly better performance, and since our technique is fully automatic, we believe the application developers would find more motivation to use hardware tessellation if the culling is done for them by the system. For future work, we would like to investigate hierarchical tessellation, so that parts of a base primitive can be culled, or even several base primitives in a single cull operation. In addition, we have realized that backface culling is the most difficult type of culling when it comes to handling arbitrary vertex shaders. Therefore, we would like to do research on novel techniques to further increase the backface cull rate at a low cost. Furthermore, our work can be used in a software rendering pipeline as well, and it would be interesting to evaluate exactly what kind of performance can be obtained in such contexts.

ACKNOWLEDGMENTS

Thanks to Natalya Tatarchuck for giving us access to the ninja model.

REFERENCES

APODACA, A. A. AND GRITZ, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

BERZ, M. AND HOFFSTÄTTER, G. 1998. Computation and application of taylor polynomials with interval remainder bounds. *Reliable Comput. 4,* 1, 83–97.

COMBA, J. L. D. AND STOLFI, J. 1993. Affine arithmetic and its applications to computer graphics. In *Proceedings of the VII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'93)*, 9–18.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *Proceedings of the ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques*, 96–102.

DOGGETT, M. 2005. Xenos: XBOX 360 GPU. Eurographics presentation.

DOGGETT, M. AND HIRCHE, J. 2000. Adaptive view dependent tessellation of displacement maps. *Graph. Hardw.* 59–66.

GREENE, N. AND KASS, M. 1994. Error-Bounded antialiased rendering of complex environments. In *Proceedings of the ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques*, 59–66.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical z-buffer visibility. In *Proceedings of the ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques*, 231–238.

HAINES, E. AND WALLACE, J. 1994. Shaft culling for efficient ray-traced radiosity. In *Proceedings of the 2nd Eurographics Workshop on Rendering*, 122–138.

HAN, C.-Y., IM, Y.-H., AND KIM, L.-S. 2005. Geometry engine architecture with early backface culling hardware. *Comput. Graph. 29,* 5, 415–425.

HANSEN, E. R. 1975. A Generalized Interval Arithmetic. In *Proceedings of the International Symposium on Interval Mathemantics*, 7–18.

HASSELGREN, J. AND AKENINE-MÖLLER, T. 2007. PCU: The programmable culling unit. *ACM Trans. Graph. 26,* 3, 92.1–92.10.

HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. 1998. Sampling procedural shaders using affine arithmetic. *ACM Trans. Graph. 17,* 3, 158–176.

HUNGERBÜHLER, R. AND GARLOFF, J. 1998. Bounds for the range of a bivariate polynomial over a triangle. *Reliable Comput. 4,* 1, 3–13.

KUMAR, S. AND MANOCHA, D. 1996. Hierarchical visibility culling for spline models. In *Graphics Interface*, 142–150.

LIN, Q. AND ROKNE, J. 1996. Interval approximation of higher order to the ranges of functions. *Comput. Math. Appl. 31,* 7, 101–109.

LOOP, C. AND SCHAEFER, S. 2007. Approximating Catmull-Clark subdivision surfaces with bicubic patches. Tech. rep., MSR-TR-2007-44, Microsoft Research.

MAKINO, K. AND BERZ, M. 2003. Taylor models and other validated functional inclusion methods. *Int. J. Pure Appl. Math. 4,* 4, 379–456.

MOORE, R. E. 1966. *Interval Analysis*. Prentice-Hall.

MOREIN, S. 2000. ATI Radeon HyperZ technology. In *Proceedings of the Hot 3D Workshop on Graphics Hardware*. ACM Press.

MORETON, H. 2001. Watertight tessellation using forward differencing. In *Graphics Hardware*, 25–32.

MOULE, K. AND MCCOOL, M. D. 2002. Efficient bounded adaptive tessellation of displacement maps. In *Graphics Interface*, 171–180.

PHARR, M. 2006. Interactive rendering in the post-GPU era. Keynote in *Graphics Hardware*.

SHIRMAN, L. A. AND ABI-EZZI, S. S. 1993. The cone of normals technique for fast processing of curved patches. *Comput. Graph. Forum 12,* 3, 261–272.

TATARCHUK, N., OAT, C., MITCHELL, J. L., GREEN, C., ANDERSSON, J., MITTRING, M., DRONE, S., AND GALOPPO, N. 2007. Advanced real-time rendering in 3D graphics and games. SIGGRAPH course.

VLACHOS, A., PETERS, J., BOYD, C., AND MITCHELL, J. L. 2001. Curved PN triangles. In *Proceedings of the Symposium on Interactive 3D Graphics*, 159–166.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. GPU-Accelerated high-quality hidden surface removal. *Graph. Hardw.* 7–14.

ZHANG, X., REDON, S., LEE, M., AND KIM, Y. 2007. Continuous collision detection for articulated models using Taylor models and temporal culling. *ACM Trans. Graph. 26,* 3, 15.1–15.10.

Received April 2008; accepted January 2009