Vision, Modeling, and Visualization (2011)
Peter Eisert, Konrad Polthier, and Joachim Hornegger (Eds.)

Effective Back-Patch Culling for Hardware Tessellation

C. Loop[†1] and M. Nießner[‡2] and C. Eisenacher[§2]

[1]Microsoft Research
[2]University of Erlangen-Nuremberg

Abstract

When rendering objects with hardware tessellation, back-facing patches *should be culled as early as possible to avoid unnecessary surface evaluations, and setup costs for the tessellator and rasterizer. For dynamic objects the popular cone-of-normals approach is usually approximated using tangent and bitangent cones. This is faster to compute, but less effective. We present a novel approach using the Bézier convex hull of the* parametric tangent plane*. It is much more accurate, and by operating in clip space we are able to reduce the computational cost significantly. As our algorithm vectorizes well, we observe comparable test times with increased cull-rates.*

1. Introduction

Hardware tessellation of patch primitives, possibly animated or generated on the fly [LS08], is now a part of the real-time rendering pipeline and supported by commodity graphics hardware. A compact geometric patch description is held in fast on-chip cache, and repeatedly evaluated in parallel to generate a dense triangulation of the patch, ready for immediate consumption by the rasterization stage.

On current hardware back-facing *triangles* can be culled to avoid unnecessary rasterization and pixel shading. However, if the plane normals of all generated triangles for a given patch point away from the viewer, considerable amounts of computation are wasted for surface evaluation and triangle setup. In this paper we explore the feasibility and performance of back-*patch* culling. That is, we perform a culling test on entire patch primitives earlier in the pipeline, to avoid computations that happen before rasterization.

This is not a new idea, but to date only low order approximations have appeared [SAE93, KML95]. This is likely due to the limited compute resources of previous generation graphics processors. As pointed out by Kumar and Manocha [KM96], patch culling is a trade-off between *efficiency*: how much computational effort is needed to reach a culling decision, and *effectiveness*: how many patches are actually culled.

We present a novel approach, based on the *parametric tangent plane* of a patch to accurately partition the space of eyepoint positions into *front-facing*, *back-facing*, or *silhouette* regions. As shown in Figure 1, this is more effective than previous methods.

Operating in clip space simplifies our test considerably, and all steps vectorize very well. Our computation times are comparable to existing approaches, but the increased effectiveness gives us a performance advantage as patch tessellation density increases.

2. Previous Work

A viewer cannot possibly see back-facing polygons of a closed surface, and *back-face* culling is a standard technique used by GPUs to quickly identify and remove them. To avoid calculating the dot-product between plane normal and viewing direction for each polygon, hierarchical approaches cluster polygons by normal and cull entire polygon groups [KMGL96]. This concept can be transferred to parametric surfaces, where *back-patch* culling removes an entire patch before it is tessellated into polygons.

NCone: In a preprocess, Shirman and Abi-Ezzi [SAE93] determine the *normal patch*, $N(u,v) = \partial B(u,v)/\partial u \times \partial B(u,v)/\partial v$, for a given Bézier patch and compute its bounding *cone-of-normals* $(apex : \mathbf{l},\ axis : \mathbf{a},\ angle : \alpha)$. During runtime, for eyepoint $\mathbf{e}$ the vector $\mathbf{v} = (\mathbf{e}-\mathbf{l})/\|\mathbf{e}-\mathbf{l}\|$

† e-mail: Charles.Loop@microsoft.com
‡ e-mail: Matthias.Niessner@cs.fau.de
§ e-mail: Christian.Eisenacher@cs.fau.de

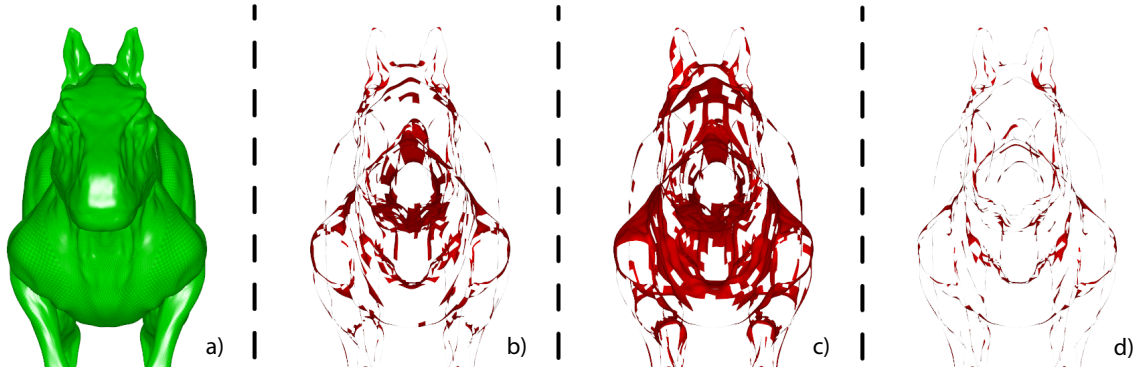

Figure 1: *The killeroo (a, 11532 Bézier patches), is rendered with different strategies for back-patch culling. For each algorithm we visualize wasted computations: areas processed by the tessellator but back-facing and hence not visible; less area is better. The cone-of-normals (b, 3697 patches culled) is effective, but costly for dynamic scenes. Its approximation from tangent and bitangent cones is faster to compute, but less precise (c, only 2621 patches culled). Our approach is faster than the cone-of-normals, and more effective (d, 4604 patches culled).*

is used in the simple test $\mathbf{v}\cdot\mathbf{a} \leq \sin(\alpha)$ to determine if the patch can be culled safely. The bound is comparably tight and the runtime test fast, but the calculation of the normal patch is expensive. This is not a problem for static models, but a draw back for patches that are animated or generated on the fly.

TCone: In a different context Sederberg and Meyers [SM88] construct a similar cone-of-normals by combining the cones derived from tangent and a bitangent patches. This results in coarser bounds but is faster to compute. Later Munkberg et al. [MHTAM10] approximate the construction of tangent and bitangent cones in order to fit the algorithm onto a modern GPU with hardware tessellation. They perform the calculations in the constant hull shader and set the tessellation factors of back-facing patches to zero in order to cull them. While they describe a general strategy to bound displaced Bézier patches, it is important to note that their GPU implementation only considers constant displacements. Therefore, the back-patch culling part of their algorithm is essentially a faster approximation of Sederberg and Meyers [SM88].

The work of Kumar et al. [KML95] focuses on NURBS models. For each surface patch, they compute a bounding box for the normalized control vectors of the normal patch. At runtime, the vertices of this bounding box are tested against the viewing direction to see if all surface normals point away from the viewer. If so, then the patch is back-facing and culled. This is similar to the cone-of-normals approach, but does not take into account that rays from the eye to points on the patch may differ from the view direction.

3. Parametric Tangent Plane

In this section, we develop the key geometric concepts behind our algorithm and introduce the *parametric tangent plane*. We work with homogeneous vectors in $\mathbb{R}^4$, and maintain a distinction between *points*, represented by row vectors, and *planes* represented by column vectors. We note the distinct transformation rules

$$\mathbf{q} = \mathbf{p}\cdot\mathbf{P} \qquad \text{and} \qquad \mathbf{s} = \mathbf{P}^{-1}\cdot\mathbf{t}$$

for points $\mathbf{p}$ and $\mathbf{q}$, and planes $\mathbf{s}$ and $\mathbf{t}$, given the 4×4 transformation matrix $\mathbf{P}$.

We focus on widely used bicubic Bézier patches, but extending our ideas to other polynomial patch types could be done in a similar fashion. A (rational) bicubic Bézier patch is defined by

$$B(u,v) = \mathbf{B}^3(u)\cdot\begin{bmatrix} \mathbf{b}_0 & \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 \\ \mathbf{b}_4 & \mathbf{b}_5 & \mathbf{b}_6 & \mathbf{b}_7 \\ \mathbf{b}_8 & \mathbf{b}_9 & \mathbf{b}_{10} & \mathbf{b}_{11} \\ \mathbf{b}_{12} & \mathbf{b}_{13} & \mathbf{b}_{14} & \mathbf{b}_{15} \end{bmatrix}\cdot\mathbf{B}^3(v),$$

where $u,v\in[0,1]^2$, $\mathbf{B}_i^d(t) = \binom{d}{i}(1-t)^{d-i}t^i$ are the degree d Bernstein basis functions, and $\mathbf{b}_j\in\mathbb{R}^4$ are homogeneous 3D control points. The parametric tangent plane $T(u,v)$ of $B(u,v)$ satisfies

$$\begin{bmatrix} B(u,v) \\ \frac{\partial}{\partial u}B(u,v) \\ \frac{\partial}{\partial v}B(u,v) \end{bmatrix}\cdot T(u,v) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

We can compute $T(u,v)$ directly as

$$T(u,v) = \text{cross4}\left(B(u,v), \tfrac{\partial}{\partial u}B(u,v), \tfrac{\partial}{\partial v}B(u,v)\right), \qquad (1)$$

where cross4() is the generalized cross product of 3 vectors in $\mathbb{R}^4$, see Appendix A. For bicubic $B(u,v)$, the parametric

tangent plane is a polynomial of bidegree 7 and can be written in Bézier form as

$$T(u,v) \;=\; \mathbf{B}^7(u)\cdot \begin{bmatrix} \mathbf{t}_0 & \mathbf{t}_1 & \cdots & \mathbf{t}_6 & \mathbf{t}_7 \\ \mathbf{t}_8 & \mathbf{t}_9 & \cdots & \mathbf{t}_{14} & \mathbf{t}_{15} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{t}_{48} & \mathbf{t}_{49} & \cdots & \mathbf{t}_{54} & \mathbf{t}_{55} \\ \mathbf{t}_{56} & \mathbf{t}_{57} & \cdots & \mathbf{t}_{62} & \mathbf{t}_{63} \end{bmatrix} \cdot \mathbf{B}^7(v),$$

where the $\mathbf{t}_i$ form an 8×8 array of *control planes*. Each $\mathbf{t}_i$ results from a weighted sum of cross4() products among the control points of $B(u,v)$.

Note that $T(u,v)$ being of bidegree 7 is less by one in both parametric directions than expected from adding the polynomial degrees of inputs to equation (1). This is easily verified with symbolic algebra software, but can be traditionally proven using properties of DeCastlejau's algorithm [FH00]. We have not seen the object that we are calling the parametric tangent plane formally defined or used in previous work. We acknowledge the strong probability that the parametric tangent plane has a classical definition, but we have not found one.

4. Visibility Classification

We use the generic term *visibility* here to mean that a point on an oriented surface can be *seen* from a given eyepoint. We do not consider the effects of occlusion, self or otherwise. Our goal is to classify entire surface patches, with respect to a given eyepoint, as front-facing, back-facing, or silhouette. We first note an optimization that will significantly reduce the computational cost of our algorithm; we do this in the context of familiar triangle culling.

4.1. Triangle Culling

Given a triangle defined by points $\mathbf{v}_0, \mathbf{v}_1$, and $\mathbf{v}_2$, its oriented spanning plane is $\mathbf{t} = \text{cross4}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)$. We say that triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ cannot be *seen* from eyepoint $\mathbf{e}$, if $\mathbf{e}$ lies in the negative half-space defined by $\mathbf{t}$. We express this as a dot product, if $\mathbf{e}\cdot\mathbf{t} < 0$ then triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ is *back-facing*. Conversely, if $\mathbf{e}\cdot\mathbf{t} > 0$ we say that triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ is *front-facing*. Otherwise if $\mathbf{e}\cdot\mathbf{t} = 0$ then triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ appears edge-on; we classify such triangles as *silhouette*.

Note that this visibility classification does not depend on a coordinate system. Given the composite *world*, *viewing*, and *perspective* transform $\mathbf{P}$ that maps world space to *clip space*, we can write

$$\mathbf{e}\cdot\mathbf{t} \;=\; \mathbf{e}\cdot\mathbf{I}\cdot\mathbf{t} \;=\; (\mathbf{e}\cdot\mathbf{P})\cdot\left(\mathbf{P}^{-1}\cdot\mathbf{t}\right) \;=\; \mathbf{f}\cdot\mathbf{s},$$

where $\mathbf{f}$ and $\mathbf{s}$ represent the transformations of eyepoint $\mathbf{e}$ and plane $\mathbf{t}$ to clip-space, respectively. By convention $\mathbf{f} = \begin{bmatrix} 0 & 0 & \alpha & 0 \end{bmatrix}$ in clip-space, so that $\mathbf{f}\cdot\mathbf{s} = \alpha s_z$, where s_z is the z component of $\mathbf{s}$, and α is of known sign. This means that in clip-space, visibility classification can be done by simply checking the sign of s_z. So instead of computing the plane containing triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ in world space and dotting the result with $\mathbf{e}$, we only need to compute the z component of the plane containing the transformed vertices in clip space. Similarly for patch culling, we only need to compute the clip-space z component of the parametric tangent plane.

4.2. Patch Culling

We classify the visibility for a patch $B(u,v)$ using its parametric tangent plane $T(u,v)$, $u,v \in [0,1]^2$, with respect to homogeneous eyepoint $\mathbf{e}$ using the Continuous Visibility function:

$$\text{CVis}(B,\mathbf{e}) = \begin{cases} \text{back-facing}, & \text{if } (\mathbf{e}\cdot T(u,v) \;<\; 0)\,,\ \forall\, u,v \in [0,1]^2, \\ \text{front-facing}, & \text{if } (\mathbf{e}\cdot T(u,v) \;>\; 0)\,,\ \forall\, u,v \in [0,1]^2, \\ \text{silhouette}, & \text{otherwise.} \end{cases}$$

A similar viewing space *back-patch condition* appears in [KM96]. Though equivalent, our classification is more general in that it is invariant to projective transformation. Computing $\text{CVis}(B,\mathbf{e})$ precisely will require costly iterative techniques to determine the roots of a bivariate polynomial. Instead, we compute a more practical discrete variant, based on the Bézier convex hull of the scalar valued patch

$$\mathbf{e}\cdot T(u,v) = \mathbf{B}^7(u)\cdot \begin{bmatrix} \mathbf{e}\cdot\mathbf{t}_0 & \mathbf{e}\cdot\mathbf{t}_1 & \cdots & \mathbf{e}\cdot\mathbf{t}_6 & \mathbf{e}\cdot\mathbf{t}_7 \\ \mathbf{e}\cdot\mathbf{t}_8 & \mathbf{e}\cdot\mathbf{t}_9 & \cdots & \mathbf{e}\cdot\mathbf{t}_{14} & \mathbf{e}\cdot\mathbf{t}_{15} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{e}\cdot\mathbf{t}_{48} & \mathbf{e}\cdot\mathbf{t}_{49} & \cdots & \mathbf{e}\cdot\mathbf{t}_{54} & \mathbf{e}\cdot\mathbf{t}_{55} \\ \mathbf{e}\cdot\mathbf{t}_{56} & \mathbf{e}\cdot\mathbf{t}_{57} & \cdots & \mathbf{e}\cdot\mathbf{t}_{62} & \mathbf{e}\cdot\mathbf{t}_{63} \end{bmatrix} \cdot \mathbf{B}^7(v).$$

Patch visibility classification reduces to counting the number of negative values, *Ncnt*, produced by taking the 64 dot products $\mathbf{e}\cdot\mathbf{t}_i$ using the Discrete Visibility function:

$$\text{DVis}(B,\mathbf{e}) = \begin{cases} \text{back-facing}, & \text{if } (\mathit{Ncnt} = 64)\,, \\ \text{front-facing}, & \text{if } (\mathit{Ncnt} = 0)\,, \\ \text{silhouette}, & \text{otherwise.} \end{cases}$$

It is important to note that the classification produced by $\text{DVis}(B,\mathbf{e})$ is a conservative approximation of $\text{CVis}(B,\mathbf{e})$: Sign differences among the Bézier coefficients are a necessary, but not sufficient condition for determining the presence of a root. Therefore, it is possible for $\text{DVis}(B,\mathbf{e})$ to classify a front or back facing patch as silhouette in error. While we can construct such cases, they seem to be rare in practice, and as demonstrated in Section 7, we are able to cull significantly more patches than previous techniques.

5. Serial Algorithm

Using symbolic algebra software, we expand equation (1) for the parametric tangent plane and find its Bézier representation. Each Bézier coefficient $\mathbf{t}_i$ is the result of a weighted sum of cross4() products among the control points of the bicubic patch $\mathbf{B}(u,v)$

$$\mathbf{t}_i \;=\; \cdots + \mathit{wgt}\cdot\text{cross4}(\mathbf{b}_j,\mathbf{b}_k,\mathbf{b}_l) + \cdots$$

For each of these cross4() products, we extract a destination index i, and source indices j,k, and l, as well as the corresponding scalar weight wgt. These values will be the same for all bicubic Bézier patches, and we place them in a header file with format

```
uint idx[4][] = {{i1, j1, k1, l1},
                 ...
                {im, jm, km, lm}};
float wgt[]   = {w1, ..., wm};
```

For the bicubic Bézier case, we require 516 cross4() products. We hence omit listing the values here, but include the defining header file as supplemental material and will make it available online.

As noted earlier, we operate in clip space, and thus only need the z component of the parametric tangent plane. Using the predetermined indices and weights, the serial algorithm to compute this is

```
for (uint k = 0; k < 516; k++)
    t[idx[k][0]] += wgt[k]
                  * cross4Z(b[idx[k][1]],
                            b[idx[k][2]],
                            b[idx[k][3]]);
```

where `float4 b[16]` contains the values of the patch control points *after* transformation to clip space, `cross4Z()`, computes just the z component of cross4(), see Appendix A, and `float t[64]` will contain the z components of the control planes, all initialized to zero.

6. Parallel Algorithm

Creating a parallel version of the parametric tangent plane algorithm is relatively straightforward, though a little care is needed to avoid write hazards. The first observation is, that each weighted cross4() product is independent of every other. So our strategy will be to have each thread in a group compute such a weighted cross4() product, and add its result to a target location in shared memory.

There are 516 weighted cross4 products, but only 64 target locations. Allocating more than 64 threads per patch will guarantee a write hazard, since a single target location will simultaneously be written to by more than one thread. Allocating exactly 64 threads would not be efficient, since the distribution of weighted cross4() products is non-uniform over the 64 target locations, so many threads will be idle after only a few computations. This distribution is illustrated in the matrix below:

$$\left[\begin{array}{cccc|cccc} 1 & 2 & 4 & 5 & 5 & 4 & 2 & 1 \\ 2 & 4 & 8 & 10 & 10 & 8 & 4 & 2 \\ 4 & 8 & 11 & 17 & 17 & 11 & 8 & 4 \\ 5 & 10 & 17 & 21 & 21 & 17 & 10 & 5 \\ \hline 5 & 10 & 17 & 21 & 21 & 17 & 10 & 5 \\ 4 & 8 & 11 & 17 & 17 & 11 & 8 & 4 \\ 2 & 4 & 8 & 10 & 10 & 8 & 4 & 2 \\ 1 & 2 & 4 & 5 & 5 & 4 & 2 & 1 \end{array}\right].$$

Each entry of this matrix shows the number of times the corresponding target location is accessed by the 516 weighted cross4() products. By partitioning the target locations into 4×4 blocks as shown above, and summing the blocks we get the following much more uniform distribution

$$\left[\begin{array}{cccc} 32 & 33 & 33 & 32 \\ 33 & 31 & 31 & 33 \\ 33 & 31 & 31 & 33 \\ 32 & 33 & 33 & 32 \end{array}\right].$$

This suggests a strategy where we allocate 16 threads per patch, and each thread is responsible for the 4 corresponding target locations of the 8×8 target array. Each thread will need to loop 33 times, compute a weighted cross4() product, and add the result to a target shared memory location. After reordering the array elements within `idx` and `wgt` according to this load distribution, the parallel code to compute the parametric tangent plane looks like

```
// ceil(516 / 16) = 33 iterations max.
for (uint k = threadIdx; k < 516; k += 16)
    t[idx[k][0]] += wgt[k]
                  * cross4Z(b[idx[k][1]],
                            b[idx[k][2]],
                            b[idx[k][3]]);
```

The final step is counting the signs of `t[64]` using a simple parallel reduction strategy. We include this code as supplemental material.

7. Results and Discussion

To evaluate our approach we extend the SimpleBezier example from the DX11 SDK. As real world applications will spend additional resources to determine tessellation factors, or construct tangent patches and evaluate those, this serves as a lower bound for the performance gains expected due to the better cull precision. We are mainly interested in dynamic surfaces, and hence only use the Bézier control points as input for the cull tests for each frame.

In contrast to Munkberg et al. [MHTAM10], we implement our cull tests using DX 11 compute shaders, and feed the decision into the constant hull shader using a small texture. This gives us more flexibility and is considerably faster, as the constant hull shader seems to execute only a single thread per patch and multiprocessor. Also, the performance difference between TCONE and NCONE is much less dramatic than reported earlier; we attribute this difference to a combination of implementation details and more recent hardware. For TCONE and NCONE we use 1 thread per patch and 128 patches per block. For OUR test we use 16 threads per patch and 8 patches per block. Those settings were determined empirically to give the best performance.

The effectiveness of back-patch culling strongly depends on the used model and viewpoint. Our test culls more patches than the previous methods for any view. To quantify the improvement, we determine the number of culled patches for 10 k random views, and list the average cull-rates for three popular models in Table 1.

	BigGuy (3570)	MonsterFrog (5168)	Killeroo (11532)
TCONE	1260 (35%)	1911 (37%)	3790 (33%)
NCONE	1601 (45%)	2286 (44%)	4685 (40%)
OURS	1729 (**48%**)	2478 (**48%**)	5206 (**45%**)

Table 1: *Average cull-rate for 10 k random views. Our method consistently performs best, and culls close to 50% of the patches.*

For a particularly challenging view of the killeroo, shown in Figure 1, we measure the total time per frame for different tessellation factors, and graph it in Figure 2. We need 0.76 ms per frame to cull 4604 patches. This is faster than NCONE, which needs 0.86 ms to cull 3697 patches. For tessellation factors larger than 8 the additional cull precision pays off, and our time per frame is lower than with TCONE, which needs 0.36 ms, but only culls 2621 patches.

These timings seem counterintuitive, as the arithmetic cost of our algorithm is roughly 10 to 20 times higher than that of NCONE and TCONE. However, both algorithms require many registers, limiting the number of active blocks per multiprocessor. In our method each individual thread needs few registers, and shared memory is naturally used very efficiently. As result we have much more active threads, but a comparable number of patches per multiprocessor, and hence similar computation times.

Aside from performance and precision advantages, our classification technique is projectively invariant, and we support *rational* bicubic patches directly. Further, by counting *positive* values ($Pcnt$) among the 64 dot products in $\mathrm{DVis}(B, \mathbf{e})$ and changing the back-facing condition to: if$(Pcnt = 0)$, we can correctly classify degenerate patches with collapsed edges, e.g., the top of the Utah teapot.

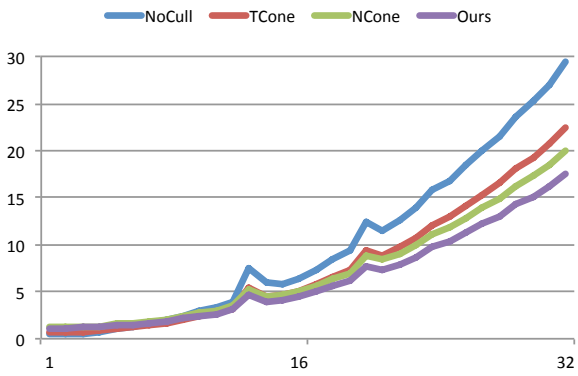

Figure 2: *Time per frame for Figure 1 using different tessellation factors. The rendering times for the other models and views behave similar. Time in ms on an Nvidia GTX 480.*

8. Conclusion and Future Work

We presented a novel strategy to cull back-facing patches, to avoid unnecessary work in the hardware tessellator. We demonstrated its feasibility for bicubic Bézier patches, but we are not limited to this patch type. The calculations vectorize very well, and compared to the popular cone-of-normals approach it is both more effective and more efficient on current hardware. Compared to the fast approximation using tangent and bitangent cones it is about 2x slower, but the better cull-rate pays off quickly as tessellation density increases.

In addition to back-patch culling, we feel that our precise visibility classification technique could be useful for other applications as well. One area we plan to explore is better handling of adaptive tessellation for silhouette patches, as these are generally the areas where most over-tessellation occurs.

Culling parametric surfaces with displacement mapping is a hard problem, and we did not address it in this paper. However, we believe that merging our ideas with the Taylor approximations of Hasselgren and Akenine-Möller [HMAM09] will be interesting.

Acknowledgments

We thank Bay Raitt of Valve Software for the BigGuy and MonsterFrog models. The Killeroo model is courtesy of Headus (metamorphosis) Pty Ltd (available at `http://www.headus.com`).

References

[FH00] FARIN G., HANSFORD D.: *The Essentials of CAGD*. A K Peters, Ltd, 2000. 3

[HAM07] HASSELGREN J., AKENINE-MÖLLER T.: PCU: the programmable culling unit. In *ACM SIGGRAPH 2007 papers* (2007), ACM, pp. 92–es.

[HMAM09] HASSELGREN J., MUNKBERG J., AKENINE-MÖLLER T.: Automatic pre-tessellation culling. *ACM Transactions on Graphics (TOG) 28*, 2 (2009), 19. 5

[KM94] KRISHNAN S., MANOCHA D.: Global visibility and hidden surface removal algorithms for free form surfaces. Technical Report TR94-063, Department of Computer Science, University of North Carolina, 1994.

[KM96] KUMAR S., MANOCHA D.: Hierarchical visibility culling for spline models. In *In Proceedings of Graphics Interface 96* (1996), pp. 142–150. 1, 3

[KMGL96] KUMAR S., MANOCHA D., GARRETT B., LIN M.: Hierarchical back-face culling. In *In 7th Eurographics Workshop on Rendering* (1996), pp. 231–240. 1

[KML95] KUMAR S., MANOCHA D., LASTRA A.: Interactive display of large-scale nurbs models. In *In 1995 Symposium on Interactive 3D Graphics (1995), ACM SIGGRAPH* (1995), pp. 51–58. 1, 2

[LS08] LOOP C., SCHAEFER S.: Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics (TOG) 27*, 1 (Mar. 2008), 1–11. 1

[LSNC09] LOOP C., SCHAEFER S., NI T., CASTANO I.: Approximating subdivision surfaces with gregory patches for hardware tessellation. *ACM Transactions on Graphics (TOG) 28*, 5 (2009), 1–9.

[MHTAM10] MUNKBERG J., HASSELGREN J., TOTH R., AKENINE-MÖLLER T.: Efficient bounding of displaced Bézier patches. In *Proceedings of the Conference on High Performance Graphics* (2010), Eurographics Association, pp. 153–162. 2, 4

[Mic09] MICROSOFT CORPORATION: Direct3D 11 Features, 2009. http://msdn.microsoft.com/en-us/library/ff476342(VS.85).aspx.

[NCnP*09] NI T., CASTAÑO I., PETERS J., MITCHELL J., SCHNEIDER P., VERMA V.: Efficient substitutes for subdivision surfaces. *ACM SIGGRAPH 2009 Courses* (2009), 1–107.

[SAE93] SHIRMAN L. A., ABI-EZZI S. S.: The cone of normals technique for fast processing of curved patches. *Computer Graphics Forum 12* (1993), 261–272. 1

[SM88] SEDERBERG T., MEYERS R.: Loop detection in surface patch intersections. *Computer Aided Geometric Design 5*, 2 (1988), 161–171. 2

[ZH97] ZHANG H., HOFF K. E.: Fast backface culling using normal masks. In *ACM Symposium on interactive 3D graphics* (1997).

Appendix A: The 4D Cross Product

We define the function $\text{cross4}(\mathbf{a},\mathbf{b},\mathbf{c}) = \begin{bmatrix} x & y & z & w \end{bmatrix}^T$ as:

$$x = \det\begin{bmatrix} a_y & b_y & c_y \\ a_z & b_z & c_z \\ a_w & b_w & c_w \end{bmatrix}, \quad y = -\det\begin{bmatrix} a_x & b_x & c_x \\ a_z & b_z & c_z \\ a_w & b_w & c_w \end{bmatrix},$$

$$z = \det\begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_w & b_w & c_w \end{bmatrix}, \quad w = -\det\begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{bmatrix}.$$

Geometrically, $\begin{bmatrix} x & y & z & w \end{bmatrix}^T$ is the oriented plane that spans homogeneous points $\mathbf{a},\mathbf{b},\mathbf{c} \in \mathbb{R}^4$.