Efficient Bounding of Displaced Bézier Patches

Jacob Munkberg[†1] Jon Hasselgren[1] Robert Toth[1] Tomas Akenine-Möller[1,2]

[1]Intel Corporation [2]Lund University

Abstract

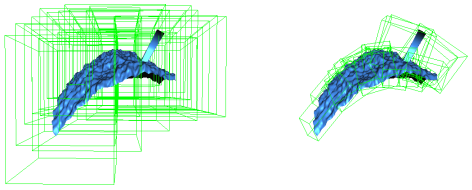
In this paper, we present a new approach to conservative bounding of displaced Bézier patches. These surfaces are expected to be a common use case for tessellation in interactive and real-time rendering. Our algorithm combines efficient normal bounding techniques, min-max mipmap hierarchies and oriented bounding boxes. This results in substantially faster convergence for the bounding volumes of displaced surfaces, prior to tessellation and displacement shading. Our work can be used for different types of culling, ray tracing, and to sort higher order primitives in tiling architectures. For our hull shader implementation, we report performance benefits even for moderate tessellation rates.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Hidden line/surface removal

1. Introduction

Modern graphics processors contain dedicated hardware for tessellating base patches into many small triangles. The Direct3D 11 API adds three new stages to the graphics pipeline to support tessellation: the *hull shader*, which is executed once per patch and once per control point, typically to compute tessellation factors and change control point bases. The fixed-function *tessellator*, which generates a large set of vertex positions in the domain of the input primitive. The *domain shader*, which is executed once per generated vertex position and outputs a displaced point in clip space. We expect high pressure on these shader stages, due to significant geometry amplification. It is therefore of utmost importance to reduce the number of domain shader evaluations. This can be done by culling patches that do not contribute to the final image. To make this efficient, an algorithm for computing tight bounds of displaced surfaces is needed.

In tile-based rendering architectures [FPE*89, LDE*08], bounds for input primitives are needed for efficient sorting into tiles. Since the domain shader is programmable, it is hard to give conservative and tight bounds of the output positions. Thus, the generated small triangles have to be sorted into tiles individually. This increases the memory requirements for the tile queues and prevents efficient occlusion culling on a patch level.



CBOX TPATCH

Figure 1: *CBOX, which represent previous work, bounds displaced Bézier surfaces by its control points and a user-provided displacement bound. Our approach, TPATCH, uses oriented bounding boxes, a min/max hierarchy of the displacement map and an efficient normal bounding algorithm, that combined bound the patches significantly tighter.*

Related Work In some REYES/RenderMan [CCC87, AG00] implementations, the user can provide an explicit `displacementbound` parameter, so that the primitive can be bounded and possibly culled during the split-dice step

† jacob.munkberg@intel.com

of the pipeline. However, this places the burden on the user, who has to estimate the maximum displacement radius. In addition, this value does not decrease during the split-dice loop, so the convergence is rather poor, as can be seen in the left side of Figure 1. Our approach is to compute these bounds based on the domain shader only (i.e., no need for any user specified parameter), and to adaptively refine the bounds as the primitive is split into smaller sub-patches.

Previous work on pre-tessellation culling [HMAM09] has shown that bounding displaced surfaces can give performance benefits for sub-pixel sized polygons. In contrast to that work, we focus on a particular use case (displaced Bézier patches). In addition, we approach the problem *hierarchically* in order to improve the total performance.

Several algorithms for normal vector bounding of Bézier surfaces exist [SM88, SAE93, Yam97, LE09]. We extend these approaches so that they fit in our framework of bounding *displaced* patches. This is a harder problem than bounding the Bézier normal vector in isolation.

Displacement map lookups can be bounded by using min-max mipmap hierarchies [MM02, HAM07], storing the minimum and maximum displacement values for each texture footprint and miplevel. We use this technique for conservative texture bounds.

The main contribution of this paper is a complete algorithm for conservative and tight bounding of *displaced* Bézier patches, using efficient normal bounding, oriented bounding boxes and min-max mipmap hierarchies of the displacement texture. The algorithm is applicable in DX11 GPUs and for hierarchical bounding in offline rendering.

2. Bounding Displaced Bézier Patches

Collections of bi-cubic Bézier patches are popular rendering primitives in production pipelines and CAGD [NCP*09]. Commonly, displacements from high resolution textures are added in the patch's normal direction to increase the surface detail. Furthermore, recent work [LS08, LSNC09, NYM*08, MNP08] has shown that Catmull-Clark subdivision surfaces can be approximated by collections of Bézier patches. This implies that the Bézier patch with displacement could be a prime use case for domain shaders in DX11. The Bézier patch is compactly represented by its control points, and this parametric surface representation can be efficiently evaluated in parallel (unlike recursive subdivision surfaces).

A Bézier patch, $\mathbf{p}(u,v)$, is a surface defined over two parametric coordinates, u and v. A *displaced* Bézier patch,

$$\mathbf{d}(u,v) = \mathbf{p}(u,v) + \hat{\mathbf{n}}(u,v)t(u,v), \tag{1}$$

contains the base patch position, $\mathbf{p}(u,v)$, and a displacement value, $t(u,v)$, acting along the normalized surface normal $\hat{\mathbf{n}}(u,v)$. Typically $t(u,v)$ is taken from a texture. The clip space position, $\mathbf{q}$, in homogeneous coordinates, is obtained by multiplying the displaced point with the model view projection matrix, $\mathbf{M}$:

$$\mathbf{q}(u,v) = \mathbf{M}\,\mathbf{d}(u,v) = \mathbf{M}(\mathbf{p}(u,v) + \hat{\mathbf{n}}(u,v)t(u,v)). \tag{2}$$

This equation constitutes the domain shader we want to bound. The task at hand is finding conservative bounds of $\mathbf{q}(u,v)$ over a parametric domain, $(u,v) \in [a,b] \times [c,d]$.

3. Algorithm

This section describes how we bound each term in Equation 2.

3.1. Bounding Bézier Patches

Following standard notation for tensor product Bézier surfaces [Far96], a Bézier patch $\mathbf{p}(u,v) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ is defined by:

$$\mathbf{p}^{m,n}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{c}_{i,j} B_i^m(u) B_j^n(v), \tag{3}$$

where $\mathbf{c}_{i,j}$ are the control points, m and n are the degrees of the patch in the parametric coordinates, u and v, respectively, and the $B(\cdot)$'s are Bernstein polynomials. In the following, we will use the term *base patch* to denote the Bézier patch which has not (yet) been displaced. This is to distinguish it from the final displaced surface. Bézier patches have the convex hull property [Far96], and they can easily be bounded by their control points. Finding an axis-aligned bounding box (AABB) for a Bézier patch accounts for 3 `min` and 3 `max` operations per control point.

3.1.1. Coordinate Frame from Control Points

We have devised a simple method for finding a coordinate frame which more tightly encloses the base patch. For a Bézier curve, the vector between the first and last control point often forms a good, first axis for a two-dimensional OBB. For a Bézier patch, we simply average the vectors from the corner control points (Figure 2), to get two axes. Given a patch with $m \times n$ control points, we denote the four corner control points $\mathbf{c}_{0,0}$, $\mathbf{c}_{m,0}$, $\mathbf{c}_{0,n}$ and $\mathbf{c}_{m,n}$, and form the two vectors:

$$\mathbf{t} = \mathbf{c}_{m,0} - \mathbf{c}_{0,0} + \mathbf{c}_{m,n} - \mathbf{c}_{0,n}, \tag{4}$$

$$\mathbf{b} = \mathbf{c}_{0,n} - \mathbf{c}_{0,0} + \mathbf{c}_{m,n} - \mathbf{c}_{m,0}. \tag{5}$$

$\mathbf{t}$ and $\mathbf{b}$ can be seen as approximate average gradients in the u and v parametric directions respectively. Their cross product gives a third axis $\mathbf{n} = \mathbf{t} \times \mathbf{b}$, and to form an orthonormal coordinate system, we set $\mathbf{x} = \mathbf{t}$, $\mathbf{y} = \mathbf{n} \times \mathbf{t}$, and $\mathbf{z} = \mathbf{n}$ and normalize each vector. The final coordinate system is: $(\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}})$. More elaborate OBB fitting schemes based on the control point cage could be derived, but in practice, the simple approach above produces axes for OBBs that bound the surface tightly. The difference in quality between bounding with AABBs and OBBs is highlighted in Figure 3 for curves

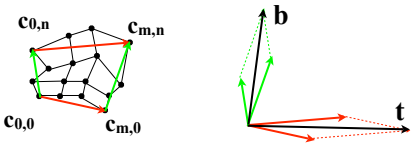


Figure 2: *By forming vectors between the corners of the patch, the OBB axes can be derived.*

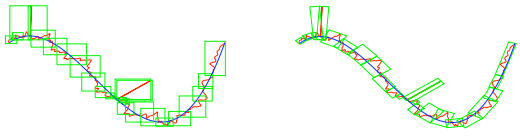


Figure 3: *A cubic Bézier curve with high frequency displacement is bounded. The left image use AABBs, and the right image use OBBs, whose axes are determined by the control points of the Bézier curve.*

and in Figure 1 for a displaced Bézier patch. As we will show below, the derived OBB axes are reused in the normal bounding algorithms.

3.2. Bounding the Normal

Bounding the patch normal, $\hat{\mathbf{n}}(u,v)$, over a domain is considerably more difficult than bounding the base position. The normal direction is computed as the cross product of two parametric derivatives of the base patch, $\mathbf{p}(u,v)$. The partial derivatives of a Bézier patch (Equation 3) can be written as:

$$\frac{\partial \mathbf{p}}{\partial u}(u,v) = \sum_{i=0}^{m-1}\sum_{j=0}^{n} \mathbf{a}_{i,j} B_i^{m-1}(u) B_j^n(v), \tag{6}$$

$$\frac{\partial \mathbf{p}}{\partial v}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n-1} \mathbf{b}_{i,j} B_i^{m}(u) B_j^{n-1}(v), \tag{7}$$

where:

$$\mathbf{a}_{i,j} = m(\mathbf{c}_{i+1,j} - \mathbf{c}_{i,j}),\ \mathbf{b}_{i,j} = n(\mathbf{c}_{i,j+1} - \mathbf{c}_{i,j}). \tag{8}$$

Note that $\mathbf{a}_{i,j}$ and $\mathbf{b}_{i,j}$ are (scaled) differences of the control points of the base patch, and therefore vectors. If the *bidegree* of $\mathbf{p}(u,v)$ is (m,n) in the parametric coordinates (u,v), the first order parametric derivatives have degrees $(m-1,n)$ and $(m,n-1)$, which can be seen in Equations 6 and 7. As shown below, the bidegree of the patch after taking the cross product of the patches is $(m+n-1,m+n-1)$. A patch representing the normal vector of a bi-cubic Bézier patch thus needs bidegree (5,5) to be represented exactly.

3.2.1. Normal Bounds from the Normal Vector Patch

Here, we describe a normal bounding algorithm, inspired by *Bézier cone* techniques [SM88, SAE93]. In summary, Bézier patches for the parametric derivatives are computed, and used to calculate a normal vector Bézier patch [Yam97]. Its control vectors are normalized, and the solid angle of this patch on the unit sphere is bounded in an OBB coordinate frame, resulting in conservative bounds of the normalized normal.

The Bézier patch's normal direction is defined by:

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

$$\sum_{j=0}^{n}\sum_{i=0}^{m-1} \mathbf{a}_{i,j} B_i^{m-1}(u) B_j^n(v) \times \sum_{k=0}^{m}\sum_{l=0}^{n-1} \mathbf{b}_{k,l} B_k^m(u) B_l^{n-1}(v). \tag{9}$$

Using the formula for products of Bernstein polynomials [Far96],

$$B_i^m(u) B_j^n(u) = \frac{\binom{m}{i}\binom{n}{j}}{\binom{m+n}{i+j}} B_{i+j}^{m+n}(u), \tag{10}$$

Equation 9 is written as:

$$\sum_{i,j,k,l} \mathbf{a}_{i,j} \times \mathbf{b}_{k,l} \frac{\binom{m-1}{i}\binom{m}{k}\binom{n}{j}\binom{n-1}{l}}{\binom{m+n-1}{i+k}\binom{m+n-1}{j+l}} B_{i+k}^{m+n-1}(u) B_{j+l}^{m+n-1}(v). \tag{11}$$

This is a Bézier patch of bi-degree $(m+n-1,m+n-1)$ with control vectors, $\mathbf{v}_{p,q}$, given by:

$$\mathbf{v}_{p,q} = \sum_{\substack{i+k=p \\ j+l=q}} \mathbf{a}_{i,j} \times \mathbf{b}_{k,l} \frac{\binom{m-1}{i}\binom{m}{k}\binom{n}{j}\binom{n-1}{l}}{\binom{m+n-1}{i+k}\binom{m+n-1}{j+l}}. \tag{12}$$

To conservatively bound the normal over the patch, we follow the approach by Sederberg and Myers [SM88]. The control vectors, $\mathbf{v}_{p,q}$, are normalized and bounded by a cone on the unit sphere, as shown in Figure 4. For efficiency, we reuse the $\hat{\mathbf{z}}$-axis from the OBB coordinate frame derived for the base patch (Section 3.1.1) as cone axis, which is an approximation of the patch's average normal. The minimal scalar product between this axis and any normalized control vector gives the cosine of the half-angle, θ, of a cone $N : \{\hat{\mathbf{n}}, \theta\}$, where $\hat{\mathbf{n}} = \hat{\mathbf{z}}$. The cone N will enclose all the normals. As shown in Figure 5A, the bounds for the normal expressed in the OBB coordinate frame are:

$$([-\sin\theta, \sin\theta], [-\sin\theta, \sin\theta], [\cos\theta, 1]). \tag{13}$$

In our experience, this approach gives very tight bounds, and as the patch is subdivided, the normal bounds converges quickly. The normal vectors could be bounded using a more elaborate algorithm for finding a bounding volume on the spherical surface. However, the cone approach combined with our OBB coordinate frame is efficient and facilitates the enclosure of the bounds from the base patch and the displacement along the normal vector. The main disadvantage is the cost of deriving the normal vector patch. For a bi-cubic Bézier patch, the computation of $\mathbf{v}_{p,q}$ includes 144 cross products and 36 normalization operations. The binomial coefficients, though, can be pre-computed in a small lookup table of 36 entries.

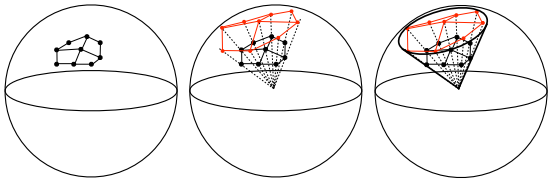


Figure 4: *Bounding control vector patches (e.g. normal or tangents). The leftmost image shows a control vector patch. In the middle image, each control vector is normalized, so that they map to points on the unit sphere (marked in red). Finally, in the rightmost image, points on the unit sphere are bounded by a cone.*

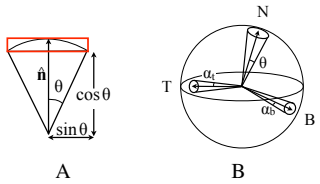


Figure 5: ***A.*** *In an OBB coordinate frame with one axis aligned with the cone's axis, the bounds of the cone on the unit sphere are easily derived using the cone half angle* θ. ***B.*** *Given bounding cones for the the two parametric derivatives (denoted T and B), a cone that bounds the cross product of any vector inside T and any vector inside B can be derived, here denoted N.*

3.2.2. Normal Bounds From Tangent Cones

As shown by Sederberg and Myers [SM88], coarser bounds can be obtained more quickly by forming two *tangent cones* from the control vectors of the first order parametric derivative patches, $\partial\mathbf{p}/\partial u$ and $\partial\mathbf{p}/\partial v$ (see Equations 6 and 7). The control vectors of the two derivative patches are normalized and bounded on the unit sphere (as shown in Figure 4), forming two cones $T:\{\hat{\mathbf{t}},\alpha_t\}$ and $B:\{\hat{\mathbf{b}},\alpha_b\}$. We use the $\hat{\mathbf{t}}$ and $\hat{\mathbf{b}}$ axes derived in Section 3.1.1 as axes for the cones T and B. Note that these are not necessarily orthogonal. As discussed in Section 3.2.1, the cosine of the cone angle α_t is the minimum scalar product of any normalized control vector from the tangent patch $\partial\mathbf{p}/\partial u$ with the $\hat{\mathbf{t}}$ axis. The half angle α_b is derived analogously. If the cones T and B do not overlap, a cone N that bounds all possible cross products of two vectors, one from each of T and B, can be constructed (Figure 5B). Its axis is in the direction $\mathbf{t}\times\mathbf{b}$ and its half-angle is given by [SM88]:

$$\sin\theta = \frac{\sqrt{\sin^2\alpha_t + 2\sin\alpha_t\sin\alpha_b\cos\beta + \sin^2\alpha_b}}{\sin\beta}, \qquad (14)$$

where β is the smallest of the two angles between the axes in the $\hat{\mathbf{t}}$ and $\hat{\mathbf{b}}$ directions. The cone, $N:\{\widehat{\mathbf{t}\times\mathbf{b}},\theta\}$, conservatively bounds the patch's normalized normal. Given θ and our choice of tangent cone axes, the normal cone axis is aligned with the OBB $\hat{\mathbf{z}}$-axis, and we can again use Equation 13 to obtain normal vector bounds in the base patch's OBB coordinate frame.

If the tangent cones overlap ($\alpha_t+\alpha_b>\beta$), we bound the normal using the unit box in the OBB coordinate frame. The tangent cone approach results in coarser bounds than the full normal vector patch approach, but is considerably less expensive. Furthermore, if the input patch is subdivided, the bounds converge quickly.

3.3. Bounded Texture Lookups

Techniques for bounding texture lookups are covered in previous work [MM02, HAM07]. The idea is to keep two extra mipmap hierarchies. The first stores maximum displacement values for each texture footprint and level and the second stores the corresponding minimum displacement values. In general, when the parametric domain decreases (e.g. the patch is subdivided), so do the texture bounds, which is a desirable characteristic.

The final bounds of the displacement vector, $\mathbf{o}=\hat{\mathbf{n}}t$, is the product (on interval arithmetic form) of the interval from the texture lookup $[t_{min},t_{max}]$ times the intervals of the normalized normal vector along each axis. Using the notation $[\underline{a},\bar{a}]$ to define an interval, where $\underline{a}$ is the lower limit and $\bar{a}$ is the upper limit, multiplication of two intervals is defined by [Moo66]:

$$[\underline{a},\bar{a}]\otimes[\underline{b},\bar{b}] = [\min(\underline{ab},\underline{a}\bar{b},\bar{a}\underline{b},\bar{a}\bar{b}),\max(\underline{ab},\underline{a}\bar{b},\bar{a}\underline{b},\bar{a}\bar{b})]. \qquad (15)$$

Hence, the interval version of the x-component of $\mathbf{o}$, is simply: $[\underline{o_x},\overline{o_x}] = [\underline{t},\bar{t}]\otimes[\underline{\hat{n}_x},\overline{\hat{n}_x}]$, and the other components are derived similarly.

3.4. Matrix Transformation

The last step in Equation 2 is the matrix transformation to clip space, so the remaining part in obtaining bounds for $\mathbf{q}$ is the model view projection matrix, which does not depend on the parametric domain, and can be seen as a constant. This constant matrix is multiplied with the eight corners of the OBB obtained for the displaced patch $\mathbf{d}$, resulting in clip space bounds for $\mathbf{q}$.

3.5. Hierarchical Refinement

To obtain tighter bounds, the patch can be subdivided in its parametric domain. In each subdivision step, a patch is split in two pieces, p_A and p_B. The normal bounds are recomputed for each subpatch and the min/max displacement maps are queried on the smaller footprints. The de Casteljau steps needed to generate the control points for p_A will generate the control points for p_B as a side product. The control point cage for the base patch converges quickly. The normal

bounds and texture lookups generally become more accurate in each subdivision steps, resulting in a convergent hierarchical bounding algorithm. Re-evaluating the normal bounds for each subdivision step is costly, so in some scenarios, we can keep the normal bounds from a coarse level, and rely on inexpensive base patch subdivision and bounded texture lookups in the remaining steps. Also, for position bounding in surface regions without displacement (regions where $t(u,v)$ is zero), no normal bounding is needed and can be bypassed.

For adaptive refinement, such as in a REYES-like *bound & split* loop, we can maintain a priority queue of the bounding boxes of the subdomains and in each subdivision step, take the top element of the queue, split it, and insert the child boxes back into the queue. The exact sorting criteria is application dependent, and may include the screen-space extents of the bounding box, the depth values, or to prioritize boxes intersecting a frustum plane for view frustum culling.

4. Applications

As mentioned in Section 1, the obtained bounds can be used in a wide array of rendering techniques and optimizations. In this section, we present a few applications areas and suitable subdivision metrics for each.

Culling View frustum culling is performed by testing the OBB corners against the frustum planes. We can prioritize sub-patches straddling the camera frustum planes, so that geometry outside the frustum planes is culled. The culling results of the patch can also be used to avoid clip-testing the generated triangles when the patch is completely inside the view frustum.

Given a coarse depth buffer, a subpatch can be occlusion culled if its bounding box is entirely occluded by already drawn primitives [GKM93]. We can adapt the subdivision criterion so that sub-patches closer to the camera are processed and rasterized first, therefore increasing the likelihood of z-culling.

Backface culling is the hardest type of culling, due to the difficulty of efficiently bounding the geometric normal after displacement. However, given the tessellation rate, the normal bounds and a tight interval of the displacement, bounds for the displaced surface normal can be derived [HMAM09]. Furthermore, the subdivision criterion can be adapted to split patches with high normal variation [LE09].

Tile-Sorting from Bounds A bounded representation of the displaced Bézier patch can be used to sort patches into tiles before tessellation. Tile-overlap can be reduced by hierarchical subdivision of the largest screen-space bounding box.

Ray Tracing & Collision Detection In a ray tracing environment, we want to reduce the total surface area of each bounding box. Using the algorithms from Section 3, we can build a tight bounding hierarchy for the displaced patches offline, where each split is carefully chosen to minimize the surface area of the child boxes. This bounding hierarchy can then be used at runtime for efficient hierarchical intersection testing. Alternatively, the hierarchy can be built on the fly and be cached for coherent ray paths [PKGH97, HS98]. In collision detection, the splits should be chosen to minimize the OBB volumes in world space.

	#instr	ATI HD5870	CPU
Domain Shader	1	1	1
CBOX	1.5	1.6	1.5
OBBTEX	2.7	2.7	2.4
TPATCH	4.5	3.8	4.5
NPATCH	11	83	11

Table 1: *Cost comparison of bounding algorithms. The presented cost is relative to the cost of executing a single domain shader. The domain shader evaluates a cubic Bézier patch, including texture based displacement in the normal direction and model view projection. For reference, we report CPU scores with texture lookups removed (as texture sampling is considerably more costly on CPUs).*

5. Results

In this section, we denote the bounding algorithms as follows: CBOX refers to bounding the patch by its control points by finding the minimum and maximum value along the Cartesian axes. A constant displacement bound (the min-max value of the displacement texture) is added in all directions. In OBBTEX, the control points are projected on OBB axes, and the displacement value is bounded by min-max mipmap textures. No normal bounding is applied. NPATCH extends OBBTEX with the normal patch bounding algorithm from Section 3.2.1. TPATCH extends OBBTEX with the tangent cone normal bounding approach from Section 3.2.2. Finally, TAYLOR is Taylor model domain shader bounding [HMAM09] of bi-degree 5 (so that the normal direction of a cubic patch can be represented exactly), using an OBB for the bounds computations.

5.1. Cost Analysis

We first look at the case of a displaced bi-cubic patch and compare the execution cost of the bounding shader with the cost of the domain shader (evaluating Equation 2). We measure the relative performance running the shaders on an Intel Core i7 3.2 GHz CPU (on one thread) and an ATI Radeon HD5870 graphics card. We also count the number of scalar shader assembly instructions for reference. As seen in Table 1, the algorithms scale as expected from the instruction count, with the exception of the NPATCH algorithm which exhausts the hardware resources (temporary registers) of the

ATI card, making it perform very poorly. TAYLOR is considerably more expensive than the other bounding approaches, due to the normalization operation, which is very costly when implemented using Taylor models. When measured on the CPU *without* normalization, TAYLOR has approximately the same cost as NPATCH, but with the normalization operation included, the cost increases to about 25× the cost of NPATCH, which makes it non-competitive from a cost perspective.

It should be noted that although the bounding shaders are more expensive than the corresponding domain shader, we only need to execute the bounding shader *once* per patch, while the domain shader may be executed thousands of times per patch due to tessellation. Therefore, the total cost of executing the bounding shaders is typically considerably lower than the total cost of executing the domain shaders. For example, if we assume that we tessellate only down to the control point level (16 vertices / patch), the cost of the TPATCH bounding algorithm will only be approximately 25% of the total domain shader cost. However, it is reasonable that the tessellation level is higher than the number of control points, since it would otherwise be better to simply send the vertices and avoid tessellation and Bézier evaluations altogether. The tessellation factors are often known at the time the culling shader is applied, which implies that the bounding shader can be dynamically enabled only in areas of high tessellation.

5.2. Quality Analysis

Our test scenes consist of the three subdivision meshes shown in Figure 8, as well as the *Spikelog* mesh shown in Figure 10, which is a difficult stress case for the OBBTEX algorithm. The SubD11 mesh comes from a February 2010 DX11 SDK sample, and the *Killeroo* and *Monsterfrog* meshes are popular test cases for subdivision surfaces. For all our test scenes, the Catmull-Clark subdivision mesh is converted to bi-cubic Bézier patches with corresponding tangent patches, using Loop & Schaefer's ACC algorithm [LS08]. The conversion gives us 3753 Bézier patches for the SubD11 mesh, 2728 patches for Killeroo, 1292 patches for Monsterfrog, and 96 patches for Spikelog. It should be noted that all meshes except SubD11 use displacement maps to add surface detail. For the SubD11 mesh, a constant displacement is added in the normal direction, replicating the SDK sample. We use a displacement magnitude of 1.0 for the SubD11 mesh unless explicitly specified.

Figure 6 presents volume and projected screen space area relative to to a near-optimal reference. The reference is computed by evaluating the domain shader at 32×32 domain points per patch and bounding the generated vertices in the OBB coordinate frame described in Section 3.1.1. Thereafter we apply our bounding algorithms and compare the resulting bounds with the reference bounds. We use the relative total volume (the total volume for an algorithm divided by the total reference volume) and relative projected screen space area as accuracy metrics. The volume metric is intended to represent quality for volume based algorithms, such as collision detection, and the projected screen space area is an efficiency metric for tile-based rendering. Both metrics are also indicators for view frustum and occlusion culling potential.

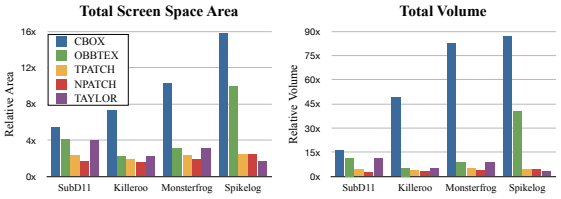


Figure 6: *Quality comparison of the bounding methods. The left chart shows the total screen space bounding box area obtained by the different methods, relative to reference screen space bounding boxes. Similarly, the right chart shows the total volume of the generated bounding boxes, relative to reference bounds.*

We observe that OBBTEX is significantly tighter than CBOX for all four scenes. This indicates that the OBB coordinate frame and min-map displacement lookups do make the bounds tighter. Also note that for Killeroo and Monsterfrog, OBBTEX is close in quality to TPATCH and NPATCH despite the lack of normal bounding. This is due to the low displacement magnitudes relative to the patch sizes in these scenes. Figure 7 shows the patch bounding boxes visually.

The Spikelog scene contains large displacement amplitudes. This is a difficult case for the OBBTEX algorithm, where the bounding boxes are expanded in all directions rather than just around the surface normal. As can be seen in Figure 6 and Figure 10, the TPATCH algorithm gives tighter bounds. Also note that the TPATCH bounds converge quickly as the patches are subdivided.

In a tile-based architecture, higher order primitives may be sorted into tile-specific queues based on their screen space extents before they are tessellated into small triangles. Depending on the rendering architecture, each tile may tessellate and domain shade its overlapping primitives independently, instead of caching and reusing processed geometry. This is especially true in highly parallel tiling architectures where the communication between processing units often should be kept at a minimum. It is therefore important to reduce the *tile overlap* so that primitives are not added to more tile-queues than necessary. However, this requires tight screen space bounds. With accurate bounds, the tile overlap can be significantly reduced for displaced patches. This is shown in Figure 8, where the screen-space overlap has been encoded as a heat map.

Figure 9 shows the bounding quality as function of displacement amplitude and subdivision level for the SubD11

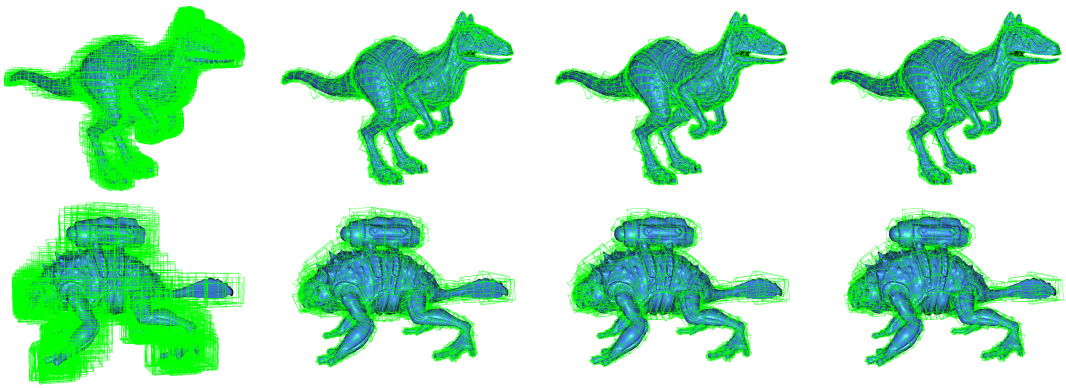


Figure 7: *Object space volumes for the Killeroo and Monsterfrog models.* OBBTEX *bounds are smaller than* CBOX *thanks to the use of OBBs and the min-max texture hierarchy. The low displacement amplitudes make the benefit of accurate normal bounds small for these models.*

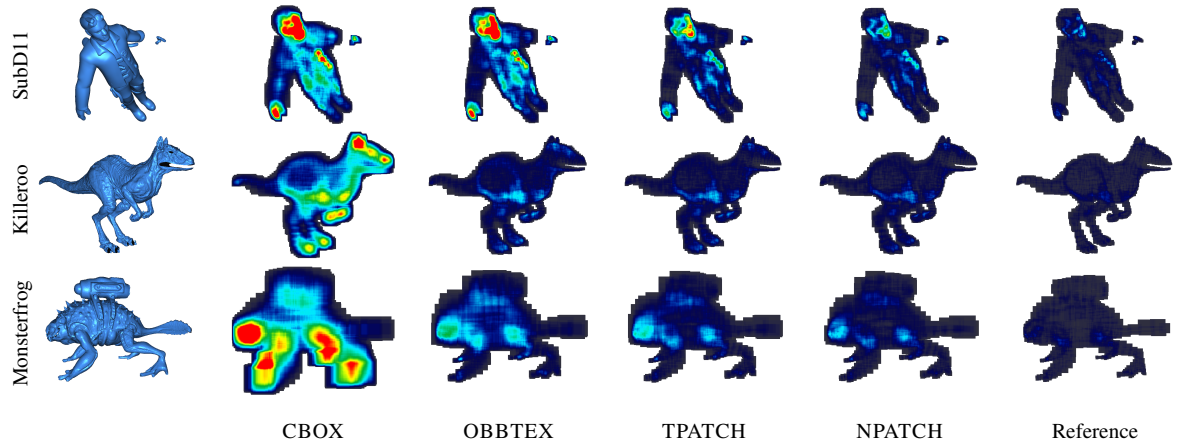


Figure 8: *False color images that show the bounding box overlap in screen space. Red means 128 or more overlapping bounding boxes. For the SubD11 mesh, a constant displacement is added to the base mesh in the base patch's normal direction. For the Killeroo and Monsterfrog meshes, the original displacement maps are used.*

mesh. When the displacement amplitude increases, TPATCH and NPATCH provide significantly tighter bounds, since they bound the normal more accurately. When the patch is subdivided, the convergence rate compared to CBOX and OBBTEX is even more significant. As the displacement is a constant offset in this test, the min-max textures do not help, and the only quality difference between CBOX and OBBTEX is due to the use of the OBB coordinate frame.

TAYLOR bounds the base patch very tightly, but as soon as displacement is added, the bounds are similar in quality to OBBTEX, as the Taylor model algorithm struggles to bound the normal efficiently. This is largely due to the high polynomial degrees involved in the Taylor model normalization operation. As seen in the rightmost chart in Figure 9, as the patches are subdivided and their normal vectors become more coherent, TAYLOR converges, but it is far from the quality of TPATCH and NPATCH. For very high subdivision levels (> 64), TAYLOR, TPATCH and NPATCH are very similar in quality, but TAYLOR is considerably more expensive. The Spikelog scene is an exception, where TAYLOR performs very well. The reason for this is that the curvatures of the base patches are relatively low, which means that the normalization operation can be accurately represented. When this happens, the higher polynomial degree of the Taylor model gives an additional improvement.

5.3. GPU Based Culling

As a stress test case for our bounding algorithms, we implemented culling in the shaders of the SubD11 sample. Due

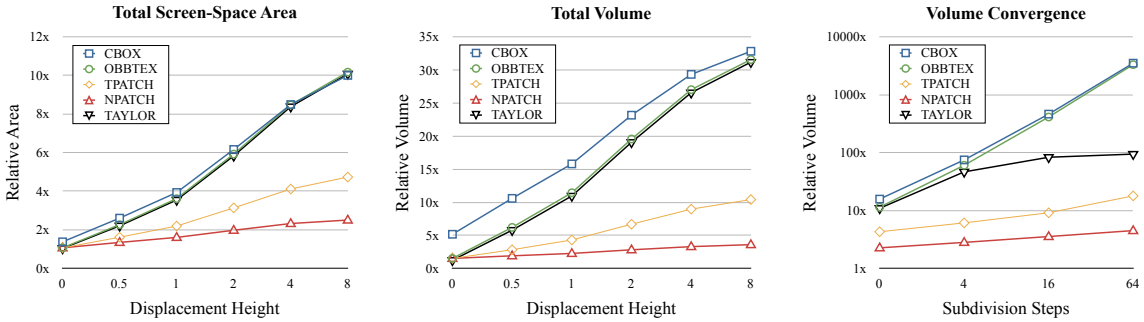


Figure 9: *Measurements of bounding quality of all patches from the SubD11 sample. The total volume/area for each algorithm is divided by a reference total volume/area, and we report this ratio for each algorithm. The leftmost chart shows the screen space area as a function of the displacement height. The middle chart shows the total volume (before transformation into clip space). Finally, the rightmost chart shows the total volume as a function of the number of subdivisions applied to each patch. In this chart, the displacement value is set to 1.0. As can be seen, normal bounding is critical for convergence. Note that the rightmost chart uses a logarithmic scale on the y-axis.*

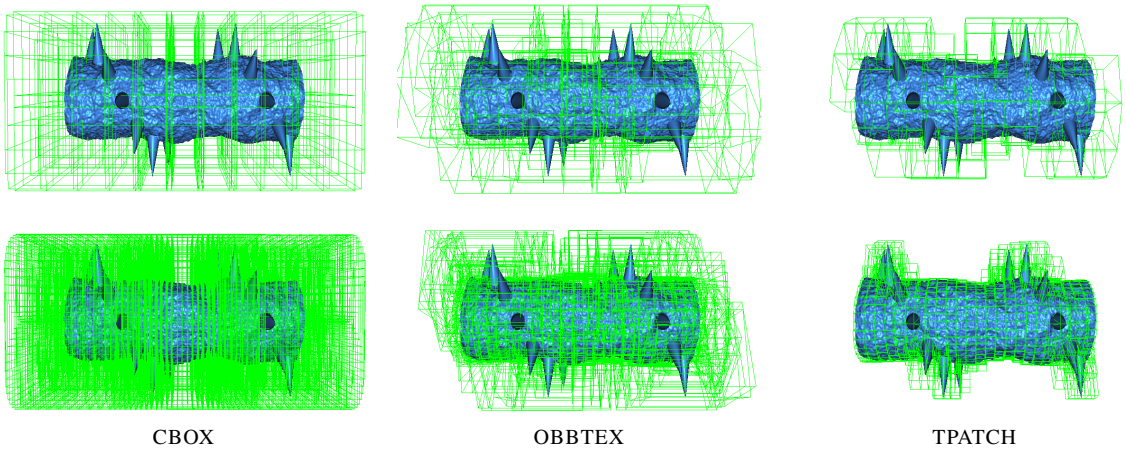


Figure 10: *The Spikelog scene contains high amplitude displacement compared to the size of of the base patches. The upper row shows the bounding volumes around the base patches, and in the lower row, each patch has been divided into 16 subpatches. This is a difficult case for the* CBOX *algorithm as it can never refine the texture bounds. Similarly, the* OBBTEX *algorithm gives poor bounds, as the displacement is added in all directions. In contrast, the* TPATCH *algorithm only applies the displacement around the normal direction. This gives tighter bounds, that converge towards the underlying surface when the base patches are subdivided.*

to the poor GPU scaling of the NPATCH algorithm that we observed in the cost analysis, we chose not to use that algorithm for this application.

We implement our bounding algorithms and culling tests in the *patch-constant* hull shader. This part of the hull shader may read the Bézier control cage generated in the *control point* hull shader, and we use this control cage in our bounding algorithms. We then perform simple view frustum and backface culling tests and output a zero tessellation factor if the patch can be culled. Passing zero as tessellation factor will cause the tessellation hardware to discard the patch early in the pipeline. Due to graphics API limitations, we do not subdivide the patches hierarchically. The application supports displacement, but in the current version, all displacement maps contain a constant value that the user can scale by a slider. Therefore, we can implement backface culling using the normal bounds computed in the TPATCH algorithm, by creating a cone that bounding both the geometric normal of the patch and the normal given by the ACC tangent patches. It should be noted that backface culling can be done even for general displacement maps [HMAM09], but in this case the culling rate is expected to be significantly lower.

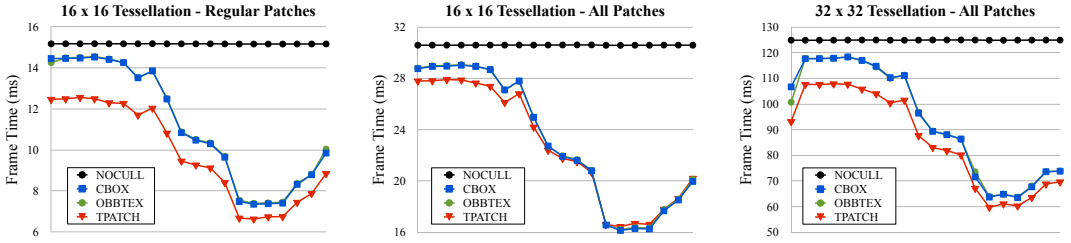


Figure 11: *Each chart shows the frame time during the SubD11 animation measured on an ATI HD5870 GPU. In the second part of the animation, the camera zooms in on the character, and there is more view frustum culling potential.* NOCULL *represent the original demo without culling. As can be seen, for high tessellation levels, and for the regular patches,* TPATCH *has a performance edge, but for lower tessellation levels, the naive bounding approaches are faster. Note that* TPATCH *reduces the longest frame time in all three charts, which is the most important to accelerate for real-time rendering.*

	Tessellation:	4×4	8×8	16×16	32×32
Regular	No Culling	2.39	3.59	15.2	61.3
	CBOX	2.42	2.93	11.2	45.0
	OBBTEX	2.50	2.93	11.2	45.0
	TPATCH	2.48	2.69	9.82	39.1
All Patches	No Culling	2.75	7.01	30.6	125
	CBOX	3.14	5.76	23.1	93.5
	OBBTEX	3.27	5.83	23.1	93.6
	TPATCH	3.89	6.92	22.7	86.0

Table 2: *Average frame time (ms) for the SubD11 animation at different tessellation levels. In the upper four rows, the sample is modified to render only the regular patches. The lower four rows is the original sample, including both regular and irregular patches.*

For regular patches, there is an exact Bézier surface representation of the Catmull-Clark surface. However, for irregular patches, the Catmull-Clark surface and its normal needs to be approximated by separate Bézier patches for the position and tangent vectors [LS08]. Unfortunately, this approximation is relatively complex and needs to be done in the hull shader. When we add our bounding algorithms, it is very easy to reach the hardware resource limits mentioned in Section 5.1, which causes hull shader performance to scale very poorly. Since this is a limitation of the particular hardware architecture, we ran two benchmarks, which gave the results shown in Table 2. In the first benchmark, we modified the SubD11 sample to render only regular patches, which we believe represents approximately how the culling will scale on future hardware with sufficient registers or efficient support for register spilling. In the second benchmark, we perform culling on all patches. As can be seen in Table 2, this approach can still be beneficial for high-quality GPU accelerated rendering applications where the tessellation factors are expected to be very high. Figure 11 shows the frame time variation over the animation for 16×16 and 32×32 tessellation.

For the irregular patches, the pressure on the hull shader is significant, and high tessellation rates are needed to maintain a consistent performance benefit from the TPATCH algorithm. For the (cheaper) regular patches, there is a clear performance benefit even for lower tessellation rates.

6. Conclusions and Future Work

We have presented algorithms for efficient bounding of displaced Bézier patches, which accelerates early culling of geometry, binning of higher order primitives and construction of high quality bounding volume hierarchies. In many cases, the OBBTEX algorithm performs very well, and we expect that this algorithm will be the best short time alternative for GPU-based culling. However, for high quality tile-based renderers, larger displacements need to be handled robustly and subdivision convergence rate is important. For these cases, we believe that the TPATCH algorithm provides a better tradeoff between performance and bounding box tightness. With hardware/pipeline modifications such as support for coarse occlusion culling based on hull shader bounds, min-max texture filtering and better register management, we believe this technique can be even faster. As future work, we want to apply a variant of the TPATCH algorithm for efficient culling of displaced Gregory patches [LSNC09].

Acknowledgements

We thanks the anonymous reviewers for their valuable feedback. Tomas 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. The original SubD11 code sample and mesh is a part of Microsoft's DirectX11 SDK. The Killeroo subdivision model is courtesy of Headus (metamorphosis) Pty Ltd (available at `www.headus.com.au`). The Monsterfrog model is courtesy of Bay Raitt, Valve Software.

References

[AG00] APODACA A. A., GRITZ L.: *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000. 1

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)* (1987), pp. 96–102. 1

[Far96] FARIN G.: *Curves and Surfaces for GAGD - A Practical Guide*. Academic Press, 1996. 2, 3

[FPE*89] FUCHS H., POULTON J., EYLES J., GREER T., GOLDFEATHER J., ELLSWORTH D., MOLNAR S., TURK G., TEBBS B., ISRAEL L.: Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System using Processor-Enhanced Memories. In *Computer Graphics (Proceedings of ACM SIGGRAPH 89)* (1989), vol. 23, pp. 79–88. 1

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-Buffer Visibility. In *Proceedings of ACM SIGGRAPH 93* (August 1993), pp. 231–238. 5

[HAM07] HASSELGREN J., AKENINE-MÖLLER T.: PCU: The Programmable Culling Unit. *ACM Transactions on Graphics, 26*, 3 (2007), 92.1–92.10. 2, 4

[HMAM09] HASSELGREN J., MUNKBERG J., AKENINE-MÖLLER T.: Automatic Pre-Tessellation Culling. *ACM Transactions on Graphics, 28*, 2 (2009), 1–10. 2, 5, 8

[HS98] HEIDRICH W., SEIDEL H.-P.: Raytracing procedural displacement shaders. In *Proceedings of Graphics Interface 1998* (1998), pp. 8–16. 5

[LDE*08] LARRY S., DOUG C., ERIC S., TOM F., MICHAEL A., PRADEEP D., STEPHEN J., ADAM L., JEREMY S., ROBERT C., ROGER E., ED G., TONI J., PAT H.: Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3 (2008), 1–15. 1

[LE09] LOOP C., EISENACHER C.: *Real-Time Patch-Based Sort-Middle Rendering on Massively Parallel Hardware*. Tech. Rep. MSR-TR-2009-83, Microsoft Research, 2009. 2, 5

[LS08] LOOP C., SCHAEFER S.: Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Transactions on Graphics, 27*, 1 (2008), 1–11. 2, 6, 9

[LSNC09] LOOP C., SCHAEFER S., NI T., CASTAÑO I.: Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. *ACM Transactions on Graphics, 28*, 5 (2009), 1–9. 2, 9

[MM02] MOULE K., MCCOOL M. D.: Efficient Bounded Adaptive Tessellation of Displacement Maps. In *Proceedings of Graphics Interface* (2002), pp. 171–180. 2, 4

[MNP08] MYLES A., NI T., PETERS J.: Fast Parallel Construction of Smooth Surfaces from Meshes with Tri/Quad/Pent Facets. *Computer Graphics Forum, 27*, 5 (2008), 1365–1372. 2

[Moo66] MOORE R. E.: *Interval Analysis*. Prentice-Hall, 1966. 4

[NCP*09] NI T., CASTAÑO I., PETERS J., MITCHELL J., SCHNEIDER P., VERMA V.: Efficient Substitutes for Subdivision Surfaces. In *ACM SIGGRAPH 2009 Courses* (2009), pp. 1–107. 2

[NYM*08] NI T., YEO Y. I., MYLES A., GOEL V., PETERS J.: GPU Smoothing of Quad Meshes. *IEEE International Conference on Shape Modeling and Applications 27*, 1 (2008), 1–11. 2

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of ACM SIGGRAPH 1997* (1997), pp. 101–108. 5

[SAE93] SHIRMAN L. A., ABI-EZZI S. S.: The Cone of Normals Technique for Fast Processing of Curved Patches. *Computer Graphics Forum, 12*, 3 (1993), 261–272. 2, 3

[SM88] SEDERBERG T. W., MEYERS R. J.: Loop Detection in Surface Patch Intersections. *Computer Aided Geometric Design, 5*, 2 (1988), 161–171. 2, 3, 4

[Yam97] YAMAGUCHI Y.: Bézier Normal Vector Surface and Its Applications. In *Proceedings of the 1997 International Conference on Shape Modeling and Applications* (1997), IEEE Computer Society, p. 26. 2, 3