Efficient Ray Tracing of Subdivision Surfaces using Tessellation Caching

Carsten Benthin[1] Sven Woop[1] Matthias Nießner[3] Kai Selgard[2] Ingo Wald[1]

[1]Intel Corporation [2]University of Erlangen-Nuremberg [3]Stanford University

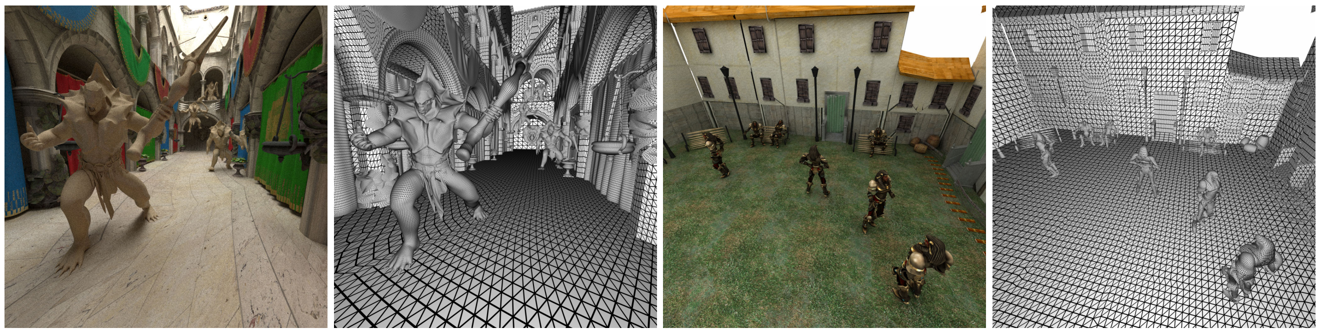

Figure 1: *Example subdivision surface scenes rendered with diffuse path tracing (up to 8 bounces,* $7-12$ *secondary rays/primary ray). Right: the* Courtyard *scene (66K patches after* feature-adaptive subdivision*) is adaptively-tessellated into 1.4M triangles from scratch per frame, and ray traced with over 90M rays per second (including shading) on a high-end Intel ®Xeon ® processor system using our efficient lazy-build caching scheme. Left: four* Barbarians *embedded in the Sponza Atrium scene (426K patches) and adaptively-tessellated into 11M triangles are ray traced with 40M rays per second. A 60MB lazy-build cache allows for rendering this scene with over* 91% *of the performance of an unbounded memory cache. Compared to ray tracing a pre-tessellated version, the memory consumption is reduced by* $6-7\times$.

Abstract

A common way to ray trace subdivision surfaces is by constructing and traversing spatial hierarchies on top of tessellated input primitives. Unfortunately, tessellating surfaces requires a substantial amount of memory storage, and involves significant construction and memory I/O costs. In this paper, we propose a lazy-build caching scheme to efficiently handle these problems while also exploiting the capabilities of today's many-core architectures. To this end, we lazily tessellate patches only when necessary, and utilize adaptive subdivision to efficiently evaluate the underlying surface representation. The core idea of our approach is a shared lazy evaluation cache, which triggers and maintains the surface tessellation. We combine our caching scheme with SIMD-optimized subdivision primitive evaluation and fast hierarchy construction over the tessellated surface. This allows us to achieve high ray tracing performance in complex scenes, outperforming the state of the art while requiring only a fraction of the memory. In addition, our method stays within a fixed memory budget regardless of the tessellation level, which is essential for many applications such as movie production rendering. Beyond the results of this paper, we have integrated our method into Embree, an open source ray tracing framework, thus making interactive ray tracing of subdivision surfaces publicly available.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: ray tracing, subdivision surfaces, caching

1 Introduction

Subdivision surfaces [Catmull and Clark 1978] have been the standard modeling primitive in the movie industry for many years [DeRose et al. 1998], and are gaining more and more traction in the context of real-time computer graphics [Pixar 2012]. Many rendering systems choose to follow the REYES approach [Cook et al. 1987] and render subdivision surfaces through dense tessellation. This is a memory-efficient solution, as the finely-tessellated geometry can be discarded immediately after use. For ray tracing, however, any portion of the scene may be required by any ray. Unfortunately, a complete, dense tessellation requires a significant amount of memory in order to store all tessellated data during ray traversal. As this data can easily exceed the available memory, multi-resolution geometry caching (MRGC) [Christensen et al. 2003] has been used to impose fixed bounds on memory usage. Despite the large number of polygons required to render a subdivision surface with adequate smoothness, the initial coarse mesh is relatively compact. This suggests that even somewhat incoherent ray paths access roughly the same patches, making caching of tessellated patches a viable option for exploiting spatial and temporal ray coherence (i.e., avoiding redundant tessellation).

However, managing a (shared) tessellation cache is challenging, especially on today's many-core architectures with dozens of hardware threads per CPU. Efficient synchronization among all threads is required to ensure that no thread removes cached data still being accessed by other threads. This continuous locking and unlocking of memory regions significantly reduces performance on many-core architectures. Distributed caches [Djeu et al. 2007], where each thread manages its own cache, do not require sophisticated synchronization when removing data. Unfortunately, they suffer from data replication, as different threads accessing the same patch will replicate the corresponding tessellation data in their own local tessellation caches. This replication severely limits effective cache sizes and causes significant computational overhead, especially on architectures with dozens of threads.

In this paper, we tackle the aforementioned problems by introducing a novel lazy-build evaluation cache specifically designed for ray tracing subdivision surfaces on many-core architectures. To this end, we dynamically construct and cache local tessellation hierarchies on top of input patch primitives. We efficiently obtain patch tessellations by combining adaptive subdivision (where needed) and direct surface evaluation (where possible) using a SIMD-optimized evaluation. The core of our approach is the fixed sized lazy-build tessellation cache, which maintains the dynamically-generated surface points and spatial hierarchies. The cache is shared among all threads and designed for efficiently managing memory allocation and deallocation of irregular-sized data, while maintaining scalability with respect to thread count.

Overall, we achieve interactive frame rates even for complex scenes, as shown in Figure 1. Compared to previous work, our approach is over 2.4$\times$ faster, while requiring only a fraction of the memory, as all threads share the lazy-build cache. We believe that scaling tessellation caches to many-core architectures is a crucial step towards real-time ray tracing of subdivision surfaces. This becomes even more important when models are fully animated.

2 Related Work

For parametric surfaces, various numerical approaches based on Bézier Clipping [Sederberg and Nishita 1990; Campagna et al. 1997; Efremov et al. 2005; Tejima et al. 2015], Newton Iteration [Toth 1985; Geimer and Abert 2005; Benthin 2006; Abert et al. 2006] and bounding envelopes [Kobbelt et al. 1998] exist for computing ray-surface intersections. These direct intersection approaches require very little or no additional data but are very compute intensive. In addition, special care needs to be taken to handle numerical instabilities (e.g., tangential rays) and corner cases. A major drawback of these approaches is that they cannot directly support displacement mapping, which is a common requirement in the industry. To this end, parametric patches are typically tessellated into polygonal representations, which can be efficiently intersected using a spatial hierarchy. For ray tracing subdivision surfaces, the basic idea of a multi-resolution geometry cache (MRGC) was first introduced by Christensen et al. [2003]. They combined adaptive multi-resolution techniques (using ray differentials to determine subdivision levels) and caching to avoid redundant subdivision and displacement operations. Unlike our solution, Christensen et al. explicitly targeted offline rendering, thus never considering efficiency issues when extending the tessellation cache to dozens of threads in the context of an interactive application.

In the *Razor* system, Djeu et al. [2007] also use a multi-resolution geometry cache, and, like our solution, target interactive rendering. Due to the high synchronization cost in coordinating shared cache access between different threads, Djeu et al. were the first to propose that each CPU thread maintains its own cache independently from all other cores. Unfortunately, a tessellation cache per thread suffers from redundant computation and data replication, thus leading to excessive memory consumption on architectures with large thread counts. A more detailed performance comparison to the *Razor* architecture is given in Section 6.3.

Rather than caching, Benthin et al. [2007] rely exclusively on on-the-fly subdivision and packet amortizations. The use of large packets amortizes both culling tests and subdivision operations across all rays in the packet, thus achieving high performance for coherent rays and large enough packets. Relying solely on amortization, however, creates a conflict between efficiency and coherence: larger packets produce better amortization, but only work for coherent rays; smaller packets work better for less-coherent rays, but then require frequent re-tessellation, leading to bad performance. For offline rendering, Hanika et al. [2010] proposed a two-level approach, where rays potentially intersecting the same patch are queued. When enough rays are queued, the patch is diced into micropolygons, a hierarchy is built on top, and all queued rays continue traversal through this hierarchy. The costs of dicing and hierarchy build are amortized over all queued rays. Hou et al. [2010] propose an algorithm for efficiently ray tracing micropolygons which targets high-quality defocus and motion blur. A key component of their method is a BVH based on 4D hyper-trapezoids which project into 3D oriented bounding boxes.

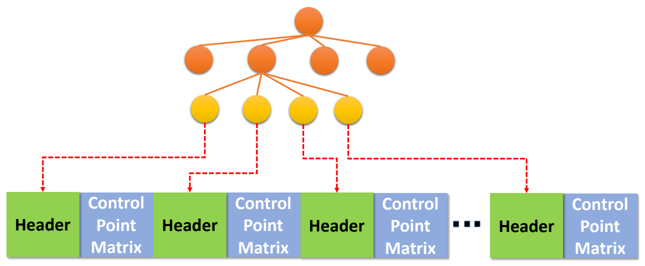

Figure 2: *Each patch consists of a header (64 bytes) and patch control point data (256 bytes), which are either 16 control points of a bi-cubic B-Spline patch or 20 control points of a Gregory patch. All patches are stored in a linear array in memory irrespective of whether they are based on the coarse input primitives or generated by feature-adaptive subdivision. Based on the bounds of each patch, a 4-wide BVH (BVH4) is build over all patches. Each BVH4 leaf points to a single patch.*

3 Patch Generation

For ray tracing subdivision surfaces, we dynamically tessellate Catmull-Clark patch primitives. The tessellation is then used to determine ray-surface intersections (see Section 4). In this section, we describe how we efficiently evaluate Catmull-Clark patches; caching the resulting tessellation is explained in Section 5. For simplicity, we assume that all models are Catmull-Clark subdivision surfaces [Catmull and Clark 1978]; however, our patch evaluation is also compatible with other schemes such as Loop subdivision [Loop 1987]. In order to efficiently evaluate subdivision patches, we follow a similar data flow to the feature-adaptive subdivision algorithm in OpenSubdiv [Pixar 2012; Nießner et al. 2012a; Nießner et al. 2012b], where the input data is given as a coarse mesh over polygons (typically quad-dominant) with additional data arrays defining features like tessellation level, edge creases, vertex creases, and holes.

Based on the input edge data, a half-edge structure is built to efficiently support adjacency queries. These queries are used to first collect the 1-ring per vertex and thus all 1-rings per initial input primitive. All 1-rings per input primitive are finally converted to a set of bi-cubic B-Spline and Gregory patches [Gregory 1974; Loop et al. 2009] using feature-adaptive subdivision [Nießner et al. 2012a]. To achieve as few patches as possible, we adaptively subdivide the patch only as long as it is not a quad or crease features are present. Finally, B-Spline patches are used to efficiently cover the typical case of regular faces, and Gregory patches are used to approximate patches with irregular vertices. If accuracy of the surface is a concern, feature-adaptive subdivision could easily be continued to reduce the size of Gregory patches without any modification to the rest of our approach.

For each edge of the input mesh, a tessellation level has been set by the application. Note that our approach allows for arbitrary integer tessellation levels, similar to Djeu et al. [2007]. However, our implementation could be easily extended to fractional tessellation following Moreton [2001]. The edge tessellation levels are modified through the adaptive subdivision process, and the final levels for the B-Spline/Gregory patches are set accordingly. All edge levels are set before rendering a frame, and remain constant during the rendering of the current frame.

The set of bi-cubic B-Spline and Gregory patches are the base primitives over which we build a top-level hierarchy. While we build the top-level hierarchy at the beginning of every frame (as geometry is potentially animated), each patch also maintains its own local BVH. A local BVH is traversed once a ray reaches a leaf of the top-level BVH. In contrast to the top-level BVH, all local hierarchies are built on-demand during ray traversal and their data is stored and managed by the tessellation cache (see Sections 4 and 5). In the following, we will discuss the memory layout of both original and adaptively-subdivided patches and how to construct the top-level bounding volume hierarchy (BVH).

3.1 Patch Data Layout

A single patch (see Figure 2) requires a total of 320 bytes of storage, corresponding to five 64 byte cache lines [Int 2001]. The first 64 bytes store header information like patch type (bi-cubic B-Spline/Gregory), the four edge tessellation levels, the total size of the patch's vertex grid (when tessellated) plus the size of the local BVH data, the 2D u, v range, and a *cache pointer*, which will later be used to point to the tessellation data in our cache. The next four cache lines (256 bytes) hold the patch's 16 control points (4×4 layout) in AOS (array-of-structure) format (xyzw). The additional control point data for Gregory patches (4 extra control points) is stored in the 4th component of the 16 control points. All patches are stored in a continuous region in memory, irrespective of whether the patch is based on the coarse input primitives or generated by feature-adaptive subdivision.

3.2 Patch Bounds and Top-Level Hierarchy

Intersecting a ray with a bi-cubic B-Spline or Gregory patch requires multiple (costly) steps (see Section 4). A tight bounding box per patch lowers the probability of performing these tests in the first place, thus significantly impacting performance. To this end, we temporarily tessellate each patch (based on the edge tessellation levels) and compute tight bounds over the resulting grid of vertices. This is computationally inexpensive (see Section 4.3) and straightforward to parallelize across all threads, and provides tighter bounds than the convex hull approximation given by the control points of the underlying patch representation. Note that we only store the spatial bounds and discard the temporarily-generated vertex positions. Once all patch bounding boxes are computed, a high-quality SAH-based BVH is build over them. Each BVH leaf contains a reference to a single patch. In the following, we will refer to this hierarchy as the *top-level BVH*.

As rays will not only traverse the top-level BVH, but also a local per patch BVH (see Section 4), BVH traversal performance is crucial. To this end, we exploit the Embree [Wald et al. 2014] ray tracing kernels which support SIMD-optimized wide-BVH traversal. In our current implementation, we restrict the width of the BVH to 4 (*BVH4*); however, an extension to wider BVH widths (8 or 16) is straightforward.

4 Ray Patch Intersection

In order to intersect a ray with a subdivision model, we first traverse the ray through the top-level hierarchy (see Section 3.2). If a ray reaches a leaf, we need to compute the intersection between the ray and the associated patch. To this end, we evaluate the patch at a uniformly-spaced set of 2D domain locations, thus obtaining a reg-

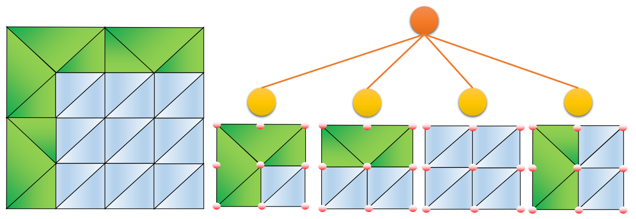

Figure 3: *Left: evaluated 5×5 vertex grid of a patch (edge tessellation factor of 4). Cracks are fixed by edge stitching; i.e., some triangles are reduced to lines and ignored during the ray-intersection test. Right: the 5×5 grid is subdivided into four 3×3 sub-grids. The local patch BVH4 is build over all 3×3 sub-grids, and each BVH4 leaf points to a single sub-grid. When a ray reaches a leaf, a SIMD-intersection test processes 8 triangles in parallel.*

ular grid of vertices. If displacements are defined, we apply them to grid vertices (see Section 4.1). We then construct a local BVH4 over the corresponding tessellation (see Figure 3), enabling fast ray traversal. Finally, when a ray reaches a leaf of a local BVH4, triangle intersection tests are performed. Note that a leaf refers to multiple triangles, allowing us to exploit SIMD parallelism for triangle intersection tests.

4.1 Fast Patch Evaluation and Tessellation

Based on the four edge tessellation levels of a patch, we first determine the sampling rate in the u, v domain of a patch. Thus, we obtain the grid resolution by taking the maximum level n in the u-direction and the maximum level m in the v-direction of the parameter domain. Next, an $n \times m$ grid of u, v coordinates in single precision floating point format is initialized, corresponding to the u, v positions at which the patch will be evaluated. As the $n \times m$ grid resolution might differ from the four edge tessellation levels, a crack-fixing step needs to be applied (see Figure 3). To this end, we apply an edge stitching pass on the u, v positions following Moreton et al. [2001].

In the final step, we iterate over the grid of u, v positions with SIMD-width granularity and evaluate the patch in parallel using SIMD instructions to obtain the vertex grid. The actual vertex grid is stored as four arrays, where the first three arrays contain the vertex positions in each dimension and the fourth the u, v coordinates (with respect to the original input primitive) discretized to pairs of 16-bit integer values. Note that a coarse input primitive can be subdivided into multiple patches, each having their own u, v range. Each grid vertex therefore needs u, v coordinates with respect to the original input primitive. A grid vertex requires a total of 16 bytes (3×32 bits for x, y, z and 2×16 bits of u, v). The local patch u, v positions used for evaluation will be discarded after the vertex grid has been evaluated.

Supporting displacement mapping in our approach is straightforward. After evaluating vertex positions, we pass grid vertices, u, v coordinates, and the evaluated patch normals to a function callback which can arbitrarily modify the vertices. Figure 4 shows an example scene with displacement mapping and texturing. For computing patch bounds, we either extend the vertex grid bounds by displacement bounds (if the application has provided them), or simply apply the displacement shader to the vertex grid before computing the bounds. Alternatively, one could use a variety of approximate bounding approaches for displaced bi-cubic patches [Munkberg et al. 2010; Nießner and Loop 2012].

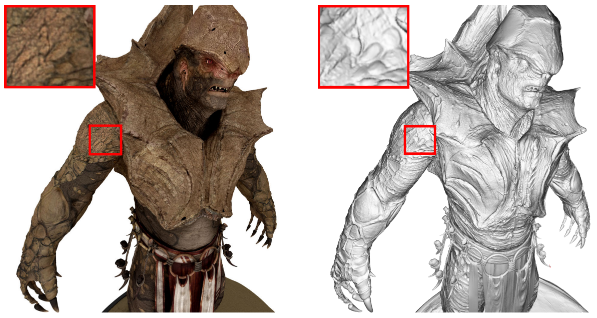

Figure 4: *The Barbarian model (tessellated into 6.6M triangles) with texturing and displacements, rendered with diffuse path tracing (left), and flat shaded with displacements (right).*

4.2 Local Patch BVH4

After the $n \times m$ vertex grid of a patch has been generated, we subdivide the grid into sub-grids (see Figure 3) and build a BVH4 over these sub-grids using a simple recursive split-in-the-middle approach. A BVH4 leaf refers to a single sub-grid. A sub-grid size of 3×3 is preferable for CPUs with 4 or 8-wide SIMD as it allows for testing the 8 triangles for intersection in parallel. A sub-grid size of 5×3, corresponding to 16 triangles, is more suited for 16-wide SIMD architectures as supported by the Xeon Phi ™ (hardware specifications are given in Section 6). Once a ray reaches a leaf of the local patch hierarchy, we load the 3 rows of data out of the $n \times m$ grid, apply shuffle instructions to convert the data to 8 triangles in SOA (structure-of-array) layout, and then perform all ray-triangle intersection tests in parallel using SIMD instructions.

4.3 Patch Evaluation and BVH4 Build Performance

Throughput performance of vertex grid evaluation and BVH4 construction for varying grids sizes are shown in Table 1 (grid resolution = edge tessellation level $+1$). The vertex grid and BVH4 data require less than 7 KB of data for a resolution smaller than 17×17, which typically covers the majority of patch grid resolutions during rendering. For high grid resolutions like 33×33 and 65×65, even a single Xeon ® processor core is able to achieve a throughput of $10-47$ patches per millisecond.

Grid resolution	3×3	5×5	9×9	17×17	33×33	65×65
Xeon patches/ms	1351	1048	350	116	47	10
XeonPhi patches/ms	1000	666	222	62.5	18	4
mem/patch (KB)	0.2	0.38	1.6	6.7	27.2	109
bytes/triangles	96	21.1	16.3	14.8	14.1	13.7

Table 1: *Throughput for evaluating a patch vertex grid (x,y,z,u,v), crack fixing at patch borders, and building a BVH4 over all sub-grids. Throughput performance is given in ms for a single Xeon ®/Xeon Phi ™ core. Combined memory consumption for the vertex grid and BVH4 data ranges from less than 0.2 KB to 109 KB, depending on the tessellation level which approaches approximately 14 bytes per triangle at high grid resolutions.*

4.4 Conservative Traversal and Triangle Intersection

Visual artifacts caused by rays shooting through shared edges of neighboring triangles are typically caused by numerical imprecision during ray traversal and ray-triangle intersection. The probability of these artifacts increases with more densely-tessellated patches. We prevent these artifacts by employing a robust triangle intersection test, which performs 3 edge tests [Davidovič et al. 2012], and a conservative BVH traversal [Ize 2013]. Note that higher traversal and intersection costs affect ray tracing performance by $10-15\%$. More sophisticated (and costly) approaches [Woop et al. 2013] to guarantee water-tightness were not necessary.

5 Shared Lazy-Build Cache

Even though patch tessellation and local BVH4 construction is fast (see Section 4.3), performing them every time a ray intersects a patch will severely impact performance, as the associated construction cost is much higher than that of traversing a ray through the local BVH4 afterwards. As ray distributions typically exhibit spatial and temporal coherence, caching previously generated tessellation and BVH4 data is therefore an efficient approach to save redundant operations. Additionally, the large on-chip hardware caches on today's CPUs make geometry caching even more effective for temporal coherent data accesses.

To this end, we propose a new lazy-build evaluation cache specifically designed for many-core architectures. Our cache operates as a globally-shared segmented FIFO (first-in first-out) cache [Cho and Moakar 2009] that can efficiently store irregular-sized tessellation and BVH4 data under heavy multi-threaded conditions. As long as the cache is filling up, the scheme behaves exactly like a lazy hierarchy build [Hunt et al. 2007] with unbounded memory storage. However, we are able to stay within a fixed memory budget – which is essential for many applications –, and we provide an efficient solution for scaling up to many compute cores.

5.1 Cache Lookup

Every time a ray reaches a leaf of the top-level BVH4, we check whether the lazy-build cache already contains the patch tessellation and the local BVH4 data. Cache lookups can be performed with hardly any computational overhead, as only a cache pointer in the patch header needs to be dereferenced. If the corresponding cache pointer is valid, it refers directly to the lazy-build cache where the local BVH4 and vertex grid of the patch are stored. If the pointer is invalid, the patch has not been accessed before and the tessellation and hierarchy generation needs to be triggered (see Section 4). In addition to the cache pointer, the patch header stores a time stamp, indicating when the patch data was generated and cached. A cache pointer is invalid if the time stamp is too old (see Section 5.2).

On a successful cache lookup, the cache pointer is used to extract the root node of the patch's BVH4 and traversal continues. On a cache miss, the cache pointer is (atomically) set to a special invalid state indicating that the patch tessellation and hierarchy build process is in progress by the given thread. This causes subsequent threads accessing the same patch to wait until the cache pointer becomes valid again. The thread performing the tessellation and hierarchy build now allocates a new memory block from the cache (to store BVH4/vertex grid data), and starts the tessellation and hierarchy build process. When finished, it sets the cache pointer to the root of the local BVH4 and the time stamp to the global time stamp. At this point, the cache pointer has become valid and all waiting threads are allowed to continue.

5.2 Memory Allocation and Deallocation

Relying on standard system calls for memory allocation and deallocation (i.e., malloc/free) is suboptimal, as they are costly, have limited thread scalability, and introduce heap fragmentation. Supporting a hard upper bound in total memory allocation is also difficult to realize, which is essential for many applications such as movie production rendering. Our lazy-build cache thus avoids sys-

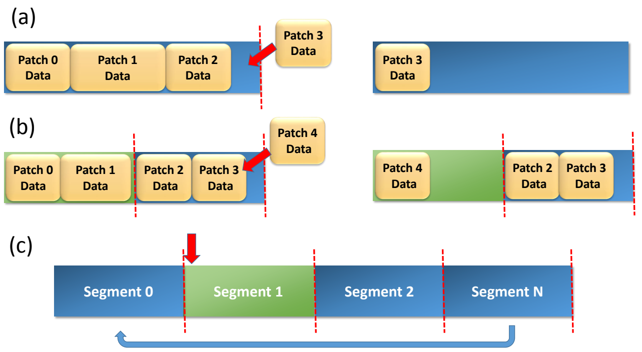

Figure 5: *(a) The lazy cache is filled with varying-sized data (vertex grid and patch BVH4). The data of the last patch cannot be inserted as the cache is not large enough. After making sure that no thread is currently accessing the cache, it is invalidated and the new patch data is put at the beginning. (b) and (c): the cache is divided into n segments ($n = 2$ for (b)), forming a circular buffer. Instead of invalidating the entire cache, only a single segment is invalidated at a time to free up allocation space. All previous segments are kept valid. At the transition point between segments, all threads are temporarily synchronized and blocked from the cache. Invalidating a single segment corresponds to invalidating $1/n$ of the cache.*

tem calls but uses a fixed-size scratchpad memory. This storage is pre-allocated at application start-up time, and remains constant during rendering. In contrast to Djeu et al. [2007], the lazy-build cache is shared among all CPU threads and uses a simple atomic counter as a parallel memory allocator. Each thread simply increments the counter by the number of cache-lines (64 bytes each) required. Note that requiring only a single atomic increment operation allows efficient scaling to a large number of threads without introducing notable synchronization overhead.

When our system runs out of allocation space in the scratchpatch storage (i.e., the cache is full), we need to deallocate memory in order to free up space. Our lazy-build cache does not deallocate individual cache entries, but invalidates entire segments of the cache (see Figure 5). Instead of invalidating the entire cache and erasing all cached data, the scratchpad storage is partitioned into n segments and only a single segment is invalidated.

These segments form a circular buffer, and at any time, only a single segment is used for allocating memory. If no allocation space is left in the current segment, all threads are synchronized (see Section 5.2.1) and therefore blocked from accessing cached data. The next segment is then marked invalid, the atomic allocation counter is reset, and threads are unblocked (see Figure 5). The segment invalidation process continues in a round-robin fashion. When transitioning from segment m to segment $m+1$, the latter is automatically invalidated by incrementing the global time stamp counter g by 1. Invalidation here means that tessellation and local BVH4 data stored in this segment has become invalid. Thus, the cache pointer for the corresponding patches is also invalidated. While performing a cache lookup, the loaded cache pointer is only valid if the patch time stamp s fulfills $s + (n-1) \geq g$. Note that all segments in the cache could be invalidated by adding the number of segments n to the global time stamp counter g.

Based on the absolute size of the cache, the total number of CPU threads, and their instantaneous working set during rendering, we found that dividing the cache into 8 segments on Xeon ® (4 on Xeon Phi ™) provided the best performance. If the absolute size of the cache is too small (e.g., a few MBs), a larger number of segments will reduce the size of a single segment to the point where it cannot serve the allocation requests from all threads at a given time. In this case, some threads would stall and wait until allocation space becomes available in the next segment. Comparing the allocation/deallocation performance against a standard system call-based approach shows that our scheme achieves a $5-10\times$ higher throughput.

5.2.1 Efficient Thread Synchronization

Before a segment is invalidated and its content is overwritten, we first need to make sure that no other thread is currently accessing data stored in the segment. To this end, each thread maintains a local counter which is initially set to 0. Before traversing a local patch BVH4, the thread atomically increments its local counter, and checks whether the original value was 0. If the value was 0, the local BVH4 traversal starts; if not, the thread enters a blocking state until the counter is reset (by another thread) to 0 again. Once local BVH4 traversal is finished, the counter is atomically decremented.

This strategy allows a master thread to efficiently block all other threads from accessing invalid data by first incrementing the other threads' local counters and waiting until the counter values become 1. At this point, it is guaranteed that all other threads have finished local BVH4 traversal or are in a blocking state before local BVH4 traversal. Now, the master thread can safely reset the allocation counter to the next segment, increment the global time stamp counter, and finally invalidate the next segment in order to free cache storage. Afterwards, it will reset the other threads' local counters to zero, and thus release them from their blocking state. The cost of synchronizing all threads using this scheme is very low (about $0.01-0.02$ ms on a high-end Xeon ® CPU and $0.09-0.12$ ms on a Xeon Phi ™) and happens only in the transitioning phase between segments. Note that we need to ensure that only a single master thread is responsible for blocking the others threads at a certain point in time; otherwise a dead-lock situation could occur.

The main advantage of our synchronization scheme is that there is hardly any synchronization overhead across threads in the common cases of reading from the lazy-build cache and performing allocation operations. This turns out to perform much faster than a standard *multiple-reader-single-writer* locking scheme per patch, where each request for read access causes an atomic update operation. For instance, if multiple threads access the same patch, the lock/unlock-related atomic update would cause cache lines to bounce between involved cores, as write operations require cache lines to be exclusively in the core's L1 cache. The cost of bouncing cache lines can affect the overall performance by $5-10\times$ on many-core architectures.

6 Results

We evaluate our approach using two different systems with many threads per CPU to stress our lazy-build cache: 1) a dual-socket Intel ®Xeon ®-E5-2600 v3, where each CPU has 18 cores (2 threads per core, 8-wide SIMD), with a total amount of 72 threads; 2) a 7120 Intel ®Xeon Phi ™ co-processor with 61 cores and a total of 244 threads (4 threads per core, 16-wide SIMD).

We have integrated our patch generation and evaluation, as well as lazy-build cache, into the Embree ray tracing framework [Wald et al. 2014]. All performance numbers include patch normal evaluation, sampling, and shading. For diffuse path tracing, the maximum recursion depth for the path tracer has been set to 8, which results on average in $7-12$ secondary rays per primary ray. In the following, we evaluate the efficiency of our lazy-build cache by comparing the performance and memory consumption against an unbounded and pre-tessellated version using varying cache sizes.

cache size %	100	80	60	40	20
cache size (MB)	150	120	90	60	30
	total cache invalidation				
# cache misses	282K	654K	1284K	2546K	5346K
cache hit rate %	99.2	98.2	96.6	93.6	87.4
rel. perf %	100	98.2	88	79	57
# patch rebuilds	1.0	2.3	4.8	9.1	18.3
	8 segments, 1 segment gets invalidated				
# cache misses	282K	468K	908K	1860K	4003K
cache hit rate %	99.2	98.6	97.4	94.6	89.4
rel. perf %	100	98.9	96.5	91	72
# patch rebuilds	1.0	1.7	2.9	6.4	14.0
ratio # cache misses	1.0×	0.74×	0.70×	0.73×	0.74×
ratio # patch rebuilds	1.0×	0.73×	0.60×	0.70×	0.76×
	multi-segment vs. per-thread cache				
rel. perf	7.1×	10.5×	16.1×	26.4×	32.3×

Table 2: *Lazy-build cache efficiency statistics for diffuse path tracing in the complex* Sponza-Barbarians *scene. We report numbers for the baseline cache size, and relative performance using smaller-sized caches. The baseline of* 150*MB is set such that the cache is sufficiently large to cache all dynamically-generated patch data without needing to deallocate memory blocks. Reducing the cache size to* 40% *of the initial size: a complete cache invalidation scheme achieves only* 79% *of the original performance; in contrast, our multi-segment-based invalidation scheme is able to maintain* 91% *of the original performance. Overall, the multi-segment cache reduces the number of misses by* $0.70-0.74\times$*, and the number of rebuilds per accessed patch by* $0.60-0.76\times$*. Compared to a per-thread cache, our multi-segment cache achieves* $7-32\times$ *higher performance, depending on the cache size.*

6.1 Caching Efficiency and Absolute Performance

As cache misses invoke patch tessellation and BVH4 rebuilding, the ray tracing performance directly correlates with cache hits (see Section 4). Our cache hit rate for diffuse path tracing (see Table 2) is over $\geq 99\%$ when the size of the lazy cache is set to cover the entire working set during rendering. In this case, no cache capacity misses occur and no deallocation needs to be performed; in other words, the patch tessellation and BVH4s for each accessed patch need only be generated once.

By reducing the cache to 40% of the initial size and invalidating the entire cache when no space is left for allocation, the relative performance drops to 79%. In this case, the patch tessellation and BVH4 rebuilds already account for over 35% of the total compute time. In contrast, our multi-segment invalidation scheme is able to reduce the number of misses by $0.70-0.74\times$, thus maintaining 91% of the performance relative to the full-sized cache. At this cache-hit rate, patch tessellation and BVH4 builds account for only 10% of the total ray tracing cost, compared to 60% traversal/intersection and 30% shading/sampling/rest costs. Hence, our caching scheme shifts the bottleneck to traversal/intersection computations.

We have also tried using more than eight segments, and reducing the size of the lazy-build cache further. However, this turns out to be counter-productive in relation to the absolute cache size. In this case, the size of a single segment is not enough to cover all instantaneous thread allocation requests. Thus, threads run idle while waiting until the next segment becomes available for allocation.

Table 2 also shows the performance of our multi-segment cache in comparison to a per-thread caching approach (with full cache invalidation). The size of a per-thread cache is set to the size of the multi-segment cache divided by the number of threads. The small caches cannot efficiently handle the large working set, resulting in frequent cache invalidations (only $50-67\%$ average cache hit rate) which leads to a significantly lower performance.

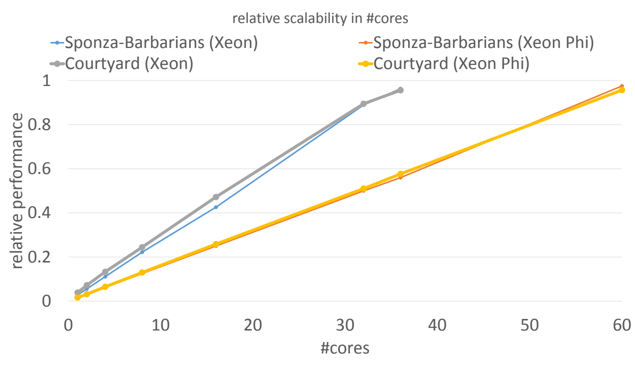

Figure 6: *Relative scalability in the number of CPU cores for the* Courtyard *and the* Sponza-Barbarians *scene with diffuse path tracing and a* 16/60*MB lazy-build cache. Our multi-segment-based lazy-build cache with fast memory allocation/deallocation allows for reaching almost linear scalability.*

Figure 6 shows an evaluation of the scalability of our method in the number of cores. On both processors, the Xeon ® and Xeon Phi ™, the performance scales almost linearly with the used core count. This suggests that there is hardly any overhead of the synchronization phases of our multi-segment invalidation scheme.

Table 3 shows the absolute rendering performance for our lazy-build cache compared to a static and highly-optimized pre-tessellated version – we provide numbers for both 100% and 40% of the initial cache size. Even though the pre-tessellated variant is $1.4-1.7\times$ faster for the Xeon ® and $1.8-2.0\times$ for Xeon Phi ™, it requires significantly more memory ($6-7\times$). The lower rendering performance is caused by patch tessellation and BVH4 build costs, but most importantly by the BVH4 quality: while a high-quality SAH-BVH4 is built over all triangles, the local BVH4s are built by a simple split-in-the-middle heuristic.

	unbounded		bounded		pre-tessellation	
	Xeon	XeonPhi	Xeon	XeonPhi	Xeon	XeonPhi
	Sponza-Barbarians					
cache size (MB)	150	150	60	60	-	-
tri/patch data (MB)	128	128	128	128	1100	1100
perf (M rays/s)	40	20	36	19	67	34
	Courtyard					
cache size (MB)	40	40	16	16	-	-
tri/patch data (MB)	20	20	20	20	216	216
perf (M rays/s)	90	56	89	53	123	85

Table 3: *Absolute path tracing performance (million rays/s) and memory consumption (MB) for our lazy-cache versus both a lazy-cache with unbounded memory and a reference high-quality BVH4 over pre-tessellated geometry. The static pre-tessellated version is* $1.4-1.7\times$ *faster on the Xeon* ® *(*$1.5-2.0\times$ *on Xeon Phi* ™*) but also requires* $6-7\times$ *more memory than the combined size of patch data and lazy-build cache storage.*

6.2 Adaptive Tessellation Per Frame

Adaptive tessellation requires the recalculation of all edge tessellation levels every frame. Thus, half-edge structures, patch headers, and patch bounds need to be updated and reevaluated at the beginning of each frame. Then, the top-level BVH4 over all patches needs to be rebuilt and the lazy cache is completely invalidated. For medium complex scenes like the *Courtyard* (66K patches) scene or the *Barbarian* model (50K patches), this takes less than 10 ms on

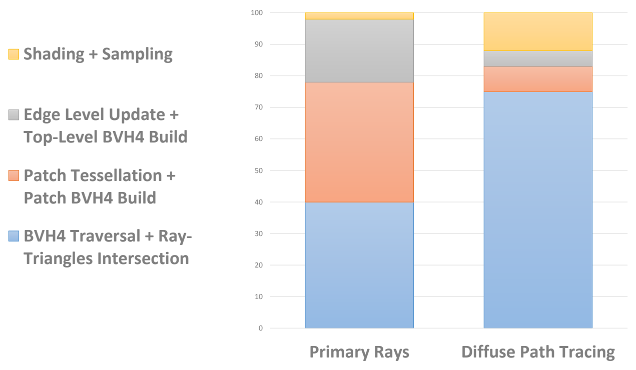

Figure 7: *The graph shows a breakdown of the cost for ray tracing the adaptively-tessellated* Barbarian *model. Per frame costs include: recalculating edge tessellation levels, top-level BVH4 rebuild, local patch tessellation and BVH4 build, BVH4 traversal, and ray-triangle intersection tests. The* Barbarian *model (50K coarse input primitives) is ray traced with fully-adaptive tessellation per frame* (1024 × 1024) *at over 80M rays/s (primary rays only, Xeon ®). For diffused path tracing, we achieve over 75M rays/s.*

the Xeon ®/ Xeon Phi ™. Figure 7 shows a cost breakdown of the different stages per frame. For primary rays only, recalculating the edge levels plus top-level BVH4 rebuild takes 20% of the total frame time, while the local patch tessellation/BVH4 build takes 40%; everything is rebuilt from scratch per frame. For diffuse path tracing, more patch data can be reused from the lazy-build cache. This increases the relative traversal and triangle intersection costs from 40% to 75% out of the total compute time.

6.3 Comparison to Previous Approaches

The closest method to our approach is the *Razor* architecture by Djeu et al. [2007]. They use a per-thread tessellation caching scheme and invalidate the entire cache when no more allocation space per thread is available. However, the Razor architecture followed different design goals (per-ray tessellation level, coarser geometry for secondary rays, decoupled shading etc.), thus making a direct comparison quite challenging. In order to provide the fairest possible comparison, we rendered the *Courtyard* scene with closely-matching view, lighting, tessellation, and rays-per-pixel settings running on an older Xeon ® system similar to that used by Djeu et al. (8 cores, 4-wide SIMD only). For these settings, Djeu et al. report a ray tracing performance of 7.5M rays/s on eight cores, which translates to 0.93M rays/s per core. Our approach achieves more than 2.4M rays/s per core (including top-level BVH build time), uses only 16 MB for our lazy-build cache, and roughly 20 MB for all patch data (after feature-adaptive subdivision). Even including all texture and original input scene data, this adds up to only a fraction of the 710 MB reported by Djeu et al..

6.4 Limitations

One of the limitations of our approach is the choice of the absolute size of a cache segment. On one hand, it must not be too small with respect to the number of CPU threads to provide for a good cache hit ratio. On the other hand, a small-sized segment will not be able to cover all instantaneous thread requests, causing threads to run idle until new allocation space is available. Luckily, it is relatively easy to adjust the cache size depending on the ray distributions and tessellation levels in a scene environment. In offline production rendering, it is also common to restrict the tessellation cache size to a fraction of the total system memory (e.g., 1GB) regardless of scene complexity or tessellation levels. We currently follow this approach and only allocate address space for the lazy-build cache. In the case of underutilization, no physical memory is wasted.

Currently, if multiple threads access the same patch at the same time, only a single thread will perform the tessellation and BVH4 build process. This somewhat limits scalability for high tessellation levels and simultaneous thread accesses in scenes with a small number of patches. One solution could be to allow waiting threads to join the tessellation/build process. Additionally, in the case of low tessellation levels, the size of the patch array with 320 bytes/patch will consume most of the memory, thus resulting in an inferior memory consumption compared to pre-tessellation. By caching the patch's 4x4 control point matrix (256 bytes) in addition to the patch's tessellation data instead of storing the control point matrix for all patches, the up-front memory consumption would be reduced from 320 to 64 bytes per patch.

Auxiliary vertex data (e.g., texture coordinates) sharing the same connectivity as vertex position data can simply be evaluated and cached in the same fashion as tessellation data, with only a moderate increase in cache footprint. However, supporting different connectivity for vertex data is more complicated, as the adaptive subdivision process would potentially generate a very different set of patches for the auxiliary data. The mapping from the first to the second set of patches could be realized by binary search based on the u,v coordinates.

Currently, when we generate patches, we resolve all semi-sharp crease tags by subdivision (see Section 3). However, a better option would be to directly evaluate regular patches with creases [Nießner et al. 2012b]. We will include this optimization in the future.

7 Conclusion and Future Work

We have presented a method to efficiently evaluate and cache patch data for ray tracing (displaced) subdivision surfaces on modern many-core architectures. As our cache has a fixed size and is shared among all threads, it is more memory- and time-efficient than approaches based on per-thread tessellation caches. Due to our fast allocation and deallocation operations for irregular-sized tessellation and hierarchy data, our method scales well to a high number of threads without introducing any significant synchronization costs. The SIMD-optimized patch evaluation and the local BVH4 rebuild provides interactive adaptive tessellation per frame, thus allowing for fully dynamic and animated scene content.

Our method can also easily be combined with other ray tracing techniques. We have implemented our method in Embree[1], an open source ray tracing framework, and made it publicly available. While our method is implemented in a CPU-based ray tracing framework, the concepts apply equally well to GPUs. In fact, since GPUs are typically composed of more cores than CPUs, many-core scalability becomes even more relevant. Hence, we could very well see our approach being combined with GPU ray tracing techniques such as the method proposed by Aila and Laine [2009].

Acknowledgments

We would like to thank James Jeffers and Eric Tabellion for their valuable feedback and guidance. The Barbarian model is courtesy of Autodesk (Jesse Sandifer is the original artist), and the Atrium Sponza Palace scene is courtesy of Crytek. The character models in the Courtyard scene are copyright by Digital Extreme. We also thank Angela Dai for the video voice over.

[1]https://embree.github.io/

References

ABERT, O., GEIMER, M., AND MÜLLER, S. 2006. Direct and Fast Ray Tracing of NURBS Surfaces. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 161–168.

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, ACM, 145–149.

BENTHIN, C., BOULOS, S., LACEWELL, J. D., AND WALD, I. 2007. Packet-based Ray Tracing of Catmull-Clark Subdivision Surfaces. Tech. Rep. UUSCI-2007-011.

BENTHIN, C. 2006. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University.

CAMPAGNA, S., SLUSALLEK, P., AND SEIDEL, H.-P. 1997. Ray Tracing of Parametric Surfaces. *The Visual Computer 13*, 6, 265–282.

CATMULL, E., AND CLARK, J. 1978. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-aided design 10*, 6, 350–355.

CHO, S., AND MOAKAR, L. A. 2009. Augmented fifo cache replacement policies for low-power embedded processors. *Journal of Circuits, Systems and Computers 18*, 06, 1081–1092.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. In *Computer Graphics Forum (Eurographics 2003 Conference Proceedings)*, Blackwell Publishers, 543–552.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The REYES Image Rendering Architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 1987)*, 95–102.

DAVIDOVIČ, T., ENGELHARDT, T., GEORGIEV, I., SLUSALLEK, P., AND DACHSBACHER, C. 2012. 3d rasterization: A bridge between rasterization and ray casting. In *Proceedings of the 2012 Graphics Interace Conference*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, GI '12, 201–208.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision surfaces in character animation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, 85–94.

DJEU, P., HUNT, W., WANG, R., ELHASSAN, I., STOLL, G., AND MARK, W. R. 2007. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Tech. Rep. UTCS TR-07-52, University of Texas at Austin Dept. of Comp. Sciences, Jan. Conditionally accepted to ACM Transactions on Graphics.

EFREMOV, A., HAVRAN, V., AND SEIDEL, H.-P. 2005. Robust and Numerically Stable Bézier Clipping Method for Ray Tracing NURBS Surfaces. In *SCCG'05 Proceedings*.

GEIMER, M., AND ABERT, O. 2005. Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *WSCG (Full Papers)*, 71–78.

GREGORY, J. A. 1974. Smooth interpolation without twist constraints. *Brunel University Mathematics Technical Papers collection;*.

HANIKA, J., KELLER, A., AND LENSCH, H. 2010. Two-level Ray Tracing with Reordering for Highly Complex Scenes. In *Proceedings of Graphics Interface 2010*, 145–152.

HOU, Q., QIN, H., LI, W., GUO, B., AND ZHOU, K. 2010. Micropolygon ray tracing with defocus and motion blur. *ACM Trans. Graph. 29*, 4 (July), 64:1–64:10.

HUNT, W., MARK, W. R., AND FUSSELL, D. 2007. Fast and lazy build of acceleration structures from scene hierarchies. In *IEEE/EG Symposium on Interactive Ray Tracing 2007*, IEEE/EG, 47–54.

INTEL CORP. 2001. *IA-32 Intel Architecture Optimization – Reference Manual*.

IZE, T. 2013. Robust BVH ray traversal. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (July), 12–27.

KOBBELT, L., DAUBERT, K., AND SEIDEL, H.-P. 1998. Ray Tracing of Subdivision Surfaces. *Proceedings of the 9th Eurographics Workshop on Rendering*, 69–80.

LOOP, C., SCHAEFER, S., NI, T., AND CASTAÑO, I. 2009. Approximating subdivision surfaces with gregory patches for hardware tessellation. ACM, vol. 28.

LOOP, C. 1987. *Smooth Subdivision Surfaces Based On Triangles*. Master's thesis, University of Utah.

MORETON, H. 2001. Watertight tessellation using forward differencing. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ACM, New York, NY, USA, 25–32.

MUNKBERG, J., HASSELGREN, J., TOTH, R., AND AKENINE-MÖLLER, T. 2010. Efficient bounding of displaced bézier patches. In *Proceedings of the Conference on High Performance Graphics*, Eurographics Association, 153–162.

NIESSNER, M., AND LOOP, C. 2012. Patch-based occlusion culling for hardware tessellation. In *Computer Graphics International*, vol. 2.

NIESSNER, M., LOOP, C., MEYER, M., AND DEROSE, T. 2012. Feature-adaptive gpu rendering of catmull-clark subdivision surfaces. *ACM Transactions on Graphics (TOG) 31*, 1, 6.

NIESSNER, M., LOOP, C. T., AND GREINER, G. 2012. Efficient evaluation of semi-smooth creases in catmull-clark subdivision surfaces. In *Eurographics (Short Papers)*, 41–44.

PIXAR, 2012. OpenSubdiv. http://graphics.pixar.com/opensubdiv/.

SEDERBERG, T. W., AND NISHITA, T. 1990. Curve Intersection using Bezier Clipping. *Computer-Aided Design 22*, 9, 538–549.

TEJIMA, T., FUJITA, M., AND MATSUOKA, T. 2015. Direct ray tracing of full-featured subdivision surfaces with bezier clipping. *Journal of Computer Graphics Techniques (JCGT) 4*, 1 (March), 69–83.

TOTH, D. L. 1985. On Ray Tracing Parametric Surfaces. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 171–179.

WALD, I., WOOP, S., BENTHIN, C., JOHNSON, G. S., AND ERNST, M. 2014. Embree: a kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG) 33*, 4, 143.

WOOP, S., BENTHIN, C., AND WALD, I. 2013. Watertight ray/-triangle intersection. *Journal of Computer Graphics Techniques (JCGT) 2*, 65–82.