Interactive Ray Tracing of Free-Form Surfaces

Carsten Benthin
Saarland University, Germany
benthin@cs.uni-sb.de

Ingo Wald
MPII Saarbrücken, Germany
wald@mpi-sb.mpg.de

Philipp Slusallek
Saarland University, Germany
slusallek@cs.uni-sb.de

ABSTRACT

Even though the speed of software ray tracing has recently been increased to interactive performance even on standard PCs, these systems usually only supported triangles as geometric primitives. Directly handling free-form surfaces such as spline or subdivision surfaces instead of first tessellating them offers many advantages such as higher precision results, reduced memory requirements, and faster preprocessing due to less primitives. However, existing algorithms for ray tracing free-form surfaces are much too slow for interactive use.

In this paper we present a simple and generic approach for ray tracing free-form surfaces together with specific implementations for cubic Bézier and Loop subdivision surfaces. We show that our approach allows to increase the performance by more than an order of magnitude, requires only constant memory, and is largely independent on the total number of free-form primitives in a scene. Examples demonstrate that even scene with over one hundred thousand free-form surfaces can be rendered interactively on a single processor at video resolution.

Categories and Subject Descriptors

I.3.3 [**Computer Graphics**]: Ray Tracing, Free-Form Surfaces

1. INTRODUCTION

Free-form surface representations such as splines, NURBS, or subdivision surfaces are an essential and powerful tool for describing 3D shapes within a computer. Together with triangle meshes these free-form representations form the basis for modeling geometry in almost all graphics applications. While CAD systems are still mostly based on splines, the popularity of subdivision approaches increases particularly in the entertainment industry due to its better support for modeling organic shape such as human body. As a consequence it becomes more and more important to directly render these shapes fast and in high quality.

Direct rendering of free-form surfaces reduces the memory requirements and preprocessing cost by not having to generate and store the many tessellated triangles. It also allows for improving geometric precision and image quality by eliminating artifacts due to insufficient tessellation. Finally, free-form surfaces often come with trimming curves that cut out irrelevant parts of the surface in the parametric domain. Robust tessellation with trimming curves suffers from the handling of complex special cases and cannot be performed quite in realtime yet [2].

To our knowledge all commercial rendering systems tessellate free-form surfaces before rendering because it is more efficient to optimize the code for a single type of primitive. Recently, graphics chips were extended with hardware support for some limited form of free-form surfaces (e.g. TrueForm from ATI [11]). However, these hardware features simply tessellate free-form surfaces very early in the graphics pipeline and still render only triangles.

1.1 Ray Tracing Free-Form Surfaces

Instead of tesselating the free-form surface for rasterization purposes, another alternative is to directly *ray trace* the surface. One significant advantage of ray tracing is that it can directly handle a wide range of geometric primitives including free-form primitives such as splines and subdivision surfaces. However, all existing algorithms are far from interactive performance and the high complexity of their implementation does not suggest that performance could be improved significantly.

In this paper we present a simple and generic approach for interactively ray tracing free-form surfaces based on recursive subdivision and efficient pruning of irrelevant subtrees of the recursion tree. In particular we present implementations of this approach for two surface types: cubic Bézier splines, and Loop subdivision surfaces. We also discuss support for B-splines and higher order splines surfaces.

While we do not introduce any new algorithm we demonstrate that a careful selection of suitable algorithms together with optimized implementations allows for drastically increased performance. With this increase in efficiency and performance we are now able to achieve interactive ray tracing performance even for complex scenes with many curved surfaces. The main advantage of the new approach is the simple and robust strategy that enables a streamlined and fast implementation.

We also present a detailed analysis of the approach and its performance on current processors. We show that using a fixed subdivision depth is more robust, requires only a small and constant amount of memory, avoids complex crack prevention algorithms, and is usually faster than applying tests for adaptive subdivision. An optimized kd-tree limits the number of primitives that have to be considered per ray, which allows for handling highly complex models with more than a hundred thousand free-form primitives. Finally the approach integrates elegantly into existing ray tracing frameworks. Even on a single PC processor we achieve interactive frame rates at video resolution even for non-trivial scenes.

Afrigraph 2004 Stellenbosch, ZA

2. PREVIOUS APPROACHES

In the last two decades researches have proposed many approaches for direct ray-tracing of free-form surfaces on CPUs and custom hardware. In the following we briefly discuss the most common approaches.

Nishita et al. [13] describes an iterative algorithm called Bézier Clipping to compute intersections between a ray and a Bézier patch by identifying and cutting away regions of the patch that are known not to intersect the ray. More general spline surfaces, e.g. NURBS surfaces, are first converted into Bézier patches. Later Campagna et al. [1] improved and optimized the approach.

Wang et al. [18] combined Newton Iteration and Bézier-clipping and used the coherence of neighboring rays to speed up Nishita's algorithm by roughly a factor of three.

Martin et al. [10] presented a framework for integrating ray-tracing of trimmed NURBS into existing rendering architectures. Paying attention to building a robust and general framework their approach did not target interactivity.

Kobbelt et al. [7] suggested the necessary theory for implementing robust and adaptive ray tracing of subdivision surfaces based on bounding envelopes. It requires the costly computation of minimum and maximum values of the basis function for each vertex influencing the patch. The algorithm is easy to implement but is slow and becomes more complicated when dealing with edges or creases due to a high number of special cases.

An adaptive technique for ray tracing subdivision surfaces was discussed by Müller et al. [12]. She processes the ray in parametric space after first projecting it onto a base triangle. This approach must deal with many special casees at patch boundaries due to adaptive refinement and thus is still rather slow.

In addition to pure software-based approaches, researchers have also proposed dedicated hardware designs for ray tracing free-form surfaces. Lewis et al. [8] presented a design of a pipelined architecture for ray Bézier patch intersection. However, no ray tracing hardware for splines is available yet.

Parker et al. [14] implemented realtime ray tracing using a massively parallel approach that also supported free-form primitives. They used Newton iteration as intersection algorithm combined with a bounding volume hierarchy. However, the best performance is still achieved through tessellation of free-form surfaces.

2.1 Discussion

Tessellation of free-form surfaces for rendering has been tolerable because ray tracing performs well even with the large numbers of triangles generated due to its $O(\log n)$ complexity with respect to scene size. However, tessellation still requires large amounts of memory for storing all generated triangles, which increases preprocessing cost and reduces cache performance.

The good performance of Wald et al. [16, 17] was achieved through simplifying and streamlining the basic algorithms, which allowed for an optimized implementation that exploits the performance features of today's processors. To achieve similar results for free-form surfaces we have to identify and eliminate the bottlenecks of previous approaches.

Existing direct ray-tracing methods can be classified into two different categories: the first category [10, 13, 18] computes the exact hit point by executing analytical or iterative algorithms whereas the second category [7, 8, 12] adaptively or uniformly refines the free-form surfaces on-the-fly during the ray-tracing process itself and eventually intersects the ray with a linear approximation to the refined surfaces.

The limitations of the first category lie in the required algorithmic complexity of computing and isolating the correct root of a high-order polynomial. This is prone to numerical problems and requires careful handling of many special cases. This approach offers little instruction level parallelism while the complex control flow makes it unsuitable for modern processors in general and prohibits the use of data level parallelism in particular.

The second category of algorithms typically uses a recursive refinement algorithm: If the ray might intersect the current primitive, refine it into a set of new primitives and recurse for each of them. Because these operations are usually considered very costly, unnecessary recursion is avoided through an adaptive termination criteria looking at the error between the primitive and a (typically) linear approximation [12, 13].

Unfortunately, the test whether to perform adaptive refinement is also rather costly. In particular it must ensure that cracks between neighboring primitives at different refinement levels are avoided or filled with additional surfaces. In particular, at each refinement step the approach must either conditionally move vertices or insert new primitives. This implies a complex control flow for handling these special cases, additional memory, and costly updates of non-trivial book-keeping data structures [12].

On today's processor architectures such code is likely to cause costly cache misses, miss-predicted branches, and might even require system calls for memory allocation. All of this limits performance significantly.

3. OUR APPROACH

In our approach we explicitly avoid any complex algorithms and take a similar approach as in Wald et al. [**?**, 16, 17]: We simplify and streamline the code as much as possible in order to allow for better optimizing the resulting implementation. While such a "brute force" approach might seem less elegant than clever adaptive refinement tests to avoid computation in the first place, it can dramatically outperform them on today's CPU architectures.

For our technique we adopt the well-known, simple, and generic divide and conquer approach consisting of the following four *core operations* that must then be optimized.

Refinement This operation refines a given primitive into a set of child primitives. The number of child primitives depends on the refinement algorithm used (usually 2 or 4). Depending on the primitive, additional data from its neighborhood must also be included.

Bounding Box Computation For the following pruning test we need to quickly compute the extent of a primitive.

Pruning Test This test must quickly discard any primitive that cannot intersects the ray.

Final Intersection Test After the reaching the fixed predefined refinement depth, a final ray/primitive intersection test is applied to an approximation of the free-from surface.

Note that this generic approach is not limited to a certain primitive representation. In the remainder of the paper we discuss optimized implementations for two important types of free-form surfaces as examples. However, the approach can easily be extended to other surface representations as long as efficient implementations of the above operations are available.

Due to the fixed number of refinement steps cracks cannot be introduced and no special handling is required. Also note that the fixed depth is only required for connected components of free-form surfaces but can still be adjusted for each of them to achieve a LOD

effect, e.g. for objects in the distance. The fixed number of refinement steps has the important advantage of requiring only a fixed amount of (preallocated) memory.

Because the final ray/primitive intersection tests are performed only after many pruning and refinement operations we first focus on efficient algorithms for the latter. We explore ways to take advantage of parallel SIMD (single instruction multiple data) processing using the SSE [5] instruction set as an example. These extensions can perform operations on four single precision floating point values in parallel by executing a single instruction. The use of SIMD has important consequences for data layout and execution flow [4].

4. INTERSECTION COMPUTATION FOR CUBIC BÉZIER PATCHES

We start by looking at Bézier patches as they have a fast and simple refinement algorithm (deCasteljau). They are also used as a common denominator for other types of spline surfaces such as B-splines. Because of the properties of SSE mentioned above we concentrate on bi-cubic splines with 4×4 control points first and discuss extensions and generalizations afterwards.

4.1 Data Layout

The data layout for the 16 points is rearranged to be most effective for the pruning and refinement operations. In particular, instead of storing control points sequentially as an array of structures we group each row by coordinates as a structure of arrays (see Figure 1). This allows for manipulating all four x, y, or z coordinates of one row via a single instruction. This data layout is optimal for algorithms operating on rows (i.e. in *v* direction) but slightly complicates operations on columns.

With this approach each Bézier patch has a compact representation that requires only $16 * 3 * sizeof(float) = 192$ bytes per patch. As a result even a non-trivial scene of one thousand free-form patches such as the head and stingray models in Figure 3 requires only 200-250KB of memory.

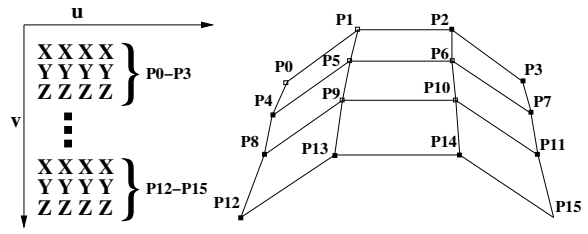

Figure 1: Data layout: For each row we group the data of four control point by coordinates in a structure-of-array format, to ensure maximum performance for pruning and refinement operations using SSE.

4.2 Pruning Test

For pruning we use an idea originally proposed by Woodward [19]. We represent the ray as the intersection of two arbitrary orthogonal planes. The distance of the control points to these planes define a 2D projection along the ray. A patch can be pruned if it completely lies in one half space of the two planes, which can be easily tested by looking at the signs of the distances (see Figure 2).

With the given data layout we can compute the necessary dot products and signs in parallel using only very few SSE instructions. For each plane we need only 12 parallel multiplications and 12 additions plus 1 instruction for obtaining the sign bits. We temporarily store the computed distances and sign bits for later use by the final intersection computation (see below).

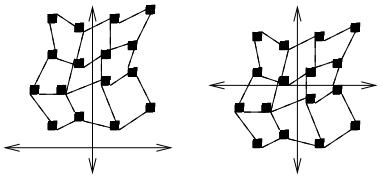

Figure 2: Pruning test: Each ray is represented as an intersection of two orthogonal planes. If all control points are located on the same side of either plane (as in the left example) the ray cannot intersect the Bézier patch (left).

4.3 Refinement

We refine Bézier patches by splitting them in half along a parametric direction using the deCasteljau algorithm. This algorithm performs simple affine combinations with fixed weights of neighboring control points, making it well suited for parallel SIMD computations. We simply alternate between the parametric directions at each refinement level.

Subdivision in the *v* direction can be done in as few as 18 parallel additions and 18 multiplications. Subdivision in the *u* direction is more complicated due to SSE limitations on horizontal operations within one SIMD register. Additional "swizzling" operations have to be inserted which moderate impact performance (see Table 1). Note that the next processor generation will offer improved support for horizontal operations [6].

Step	CPU Cycles
Pruning	86
Refinement(u)	244
Refinement(v)	168
Final Intersection	294-366
Normal	360

Table 1: Number of CPU cycles for each of the core operations. Note however that the pruning and refinement steps are executed much more often than final intersection and normal calculation, and therefore cost most of the total compute time.

4.4 Final Intersection Test

After reaching the maximum refinement depth the control polygon of the patch is considered as a simple triangular mesh by implicitly splitting every quad along its diagonal. Because executing 18 full ray-triangle intersections would be quite costly we reduce this number by two simple and fast culling tests based on the temporarily stored distances and signs.

We first compute the orientation of a quad with respect to each plane using simple bit operations: We first compute the signed distances of the vertices to the plane, and determine their signs. If all the four signs are equal the quad lies entirely in one half space and cannot intersect the ray. Otherwise the second test performs a simple 2D point in triangle test using the stored distances to compute the barycentric coordinates. In the event of a hit we simply have to add an offset to the barycentric coordinates according to the location of the triangle on the patch to obtain the parametric location of the intersection point.

For shading computations we also need the surface normal at the intersection point. To this end we simply perform the normal evaluation at this point using a variation of the above fast deCasteljau

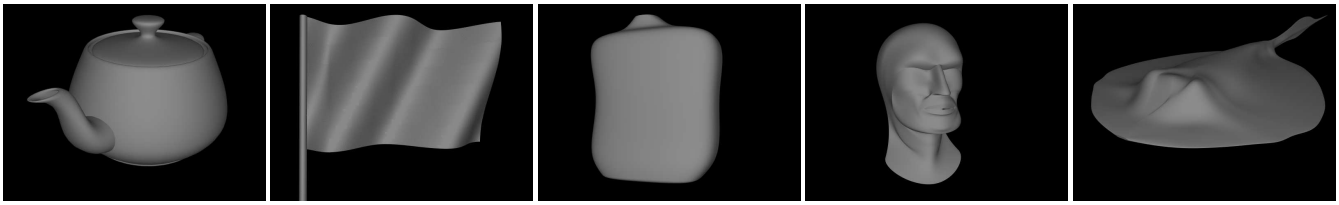

Figure 3: A set of NURBS test scenes that have been converted into bi-cubic Bézier patches. The number of Bézier patches is 32, 54, 110, 915, and 1160 (from left to right). See Figure 4 for performance results.

algorithm. Note, that we do not bother to delay the normal computation until we finally know that the intersection is indeed the right one (there might still be a closer one along the ray). First of all, our efficient pruning leads to very few intersection tests per ray anyway (1.1-2.6 on average, see below). Second, the data is already in the processor caches resulting in a low overhead for computing the normal.

4.5 Performance of Core Operations

Table 1 shows the average number of CPU cycles for each of the core operations including normal computation. All timings were measured on an Intel Xeon 2.2 GHz processor.

The cycles in the final intersection step vary due to different numbers of final ray triangle intersection tests that need to be performed. The two entries for refinement correspond to the two possible subdivision operations, related to the *u* or *v* parametric direction of Figure 1.

The culling tests before the final ray triangle intersection computations are very cheap and highly effective as shown in Table 2, reducing the number of intersection computations by up to 98% for complex scenes.

Note that the recursive sequence of pruning and refinement can easily be implemented using an iterative approach, thus avoiding function call and other overhead. The implementation used for the above measurements is carefully implemented but still leaves room for significant further optimizations. For instance, we currently do not even reuse refinements already computed for previous rays. We estimate that performance can still be increased by about a factor of two.

Scene	Teapot	Flag	Bottle	Head	Stingray
Tests/ray	1.66	1.13	1.23	2.22	2.63
PT1 (%)	11.18	13.46	11.54	11.19	11.36
PT1+2 (%)	4.27	2.88	2.10	2.51	1.82

Table 2: The two culling steps (PT1 and PT2) before the triangle intersection code very effectively reduce the number of intersection computations to 2-5 % of the original number of triangle intersections.

4.6 Trimming Curves

CAD applications usually do not use complete patches for modeling non-trivial geometry as the topological constraints would be too limiting. Instead the patches are *trimmed* by cutting off parts of the surface. To this end one or more *trimming curves* in the parametric domain define the valid parts of the surface. For proper rendering we need to take these trimming curves into account once an intersection has been found.

For each trimming curve we need to compute a 2D point-in-curve test. It turns out that a slightly modified versions of the above refine and prune operations also works for trimming curves. We translate the trimming curves in the parametric domain such that the intersection point is at the origin. Then we recursively refine all curve segments up to a maximum depth unless we prune them using a modified half space argument. In particular we keep all segments that overlap the positive x axis. The final result is obtained by simply counting the number of intersections with the x-axis.

Because this test is only performed once an intersection has been found, the total overhead for handling trimming curves is only 20-30%.

4.7 Support for Larger Scenes

So far we concentrated on speeding up a single ray Bézier patch intersection. For larger scenes containing many free-form patches (see Figure 3) we need to build a spatial index structure in order to reduce the number of patches processed per ray.

By choosing a kd-tree we can directly reuse the efficient building and traversal algorithms known from realtime ray tracing of triangle meshes [17]. Compared to a kd-tree traversal step the ray patch processing is considerably more expensive. As a result it is important to optimize the kd-tree index such that we access as few patches as possible per ray.

The algorithm for building the kd-tree uses only the axis-aligned bounding box of a patch and never looks at the patch shape itself. While this is less accurate it is sufficient and both simple and fast. Thus building the kd-tree for 100 patches takes less than 40 ms, and for 1000 patches approximately $200-300$ ms. For a moderate number of patches (< 100) this even allows for supporting animations and interaction by dynamically updating the patch data and rebuilding the kd-tree.

For building the kd-tree we use the surface area heuristics (SAH) [3, **?**], which places splitting planes according to an area-based cost functions. This approach usually performs best for triangle based scenes. However, because adjacent control polygons share control points the corresponding bounding boxes cannot completely separate them, leading to some overhead at patch boundaries.

Scene	Teapot	Flag	Bottle	Head	Stingray
Patches	32	54	110	915	1160
NO-SAH	3.21	1.14	3.68	29.91	32.18
SAH	2.12	1.14	1.82	4.87	5.01
Reduction %	33.95	0	50.54	83.71	84.43
SAH + MB	1.66	1.13	1.23	2.22	2.63
Reduction %	48.28	0.87	66.57	92.57	91.56

Table 3: Number of input patches and average number of accessed patches per ray with (SAH) and without (NO-SAH) surface area heuristics for kd-tree construction. Combined with mail-boxing (MB) the average number of patches accessed per ray is reduced by up to 92% for complex scenes.

The SAH-based kd-tree is able to reduce the average number of accessed patches per ray by 33-84% (see Table 3). In addition we use efficient mail-boxing to avoid multiple intersection computations for the same ray and patch combinations. This increases the culling rate to almost 93% for complex scenes, resulting in only 1 to 3 ray-patch intersections on average (see Table 3).

Due to the good culling performance of the kd-tree, the speed of the core operations translates almost directly to total rendering performance as shown in Figure 4.

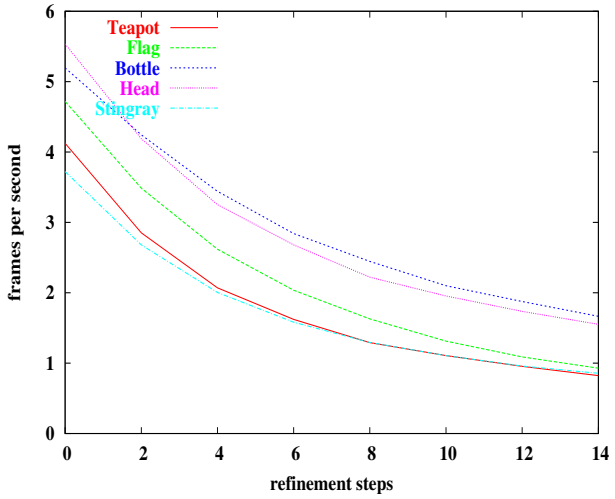

Figure 4: Performance for Bézier scenes as a function of the number of refinement steps. All experiments are performed at a resolution of 640x480 pixels with correct normal computation and simple diffuse shading on a single Intel Xeon 2.2 GHz processor. Even with a very high refinement depth we still achieve interactive performance.

Note that all experiments were performed at a resolution of 640×480 pixels with simple shading and correct normal computation on a *single* Intel Xeon 2.2 GHz processor. Most computations are related to ray tracing as deactivating shading and normal computation reduces the performance only by 5-15% and 5%, respectively, or 10-20% in total.

Obviously, total performance depends on the number of patches $N_{patches}$ and on the chosen number of refinement steps $N_{refsteps}$. The corresponding number of triangles that would have to be created by an equivalent tessellation would be $N_{patches} \times (18 \times 2^{N_{refsteps}})$. Note that we still achieve more than 1 frame per second for the head scene with a refinement depth of 14, even though this corresponds to a tessellated model with 269 *million* triangles.

For increased subdivision depth the performance starts to scale linearly with depth, because the small parts are mostly flat and only one of each refined patch remains.

Compared directly to the highly optimized OpenRT ray tracing engine [16, 17], performance for triangle-based ray-tracing is only 50-70% faster than our approach (see Table 4), with respect the same number of potential triangles. Note that due to memory requirements is was not possible to render scenes with triangle-based ray tracing when using more than 8 refinement steps, corresponding to more than 16 Mtriangles.

Note that the overall performance of our approach is largely independent of total number of Bézier patches in the scene due to the effect of the kd-tree, early ray termination, and the good pruning performance during refinement (see Figure 5).

Steps	Tris	Tri(fps)	Bézier(fps)	ratio(%)
0	16K	8.16	5.19	57.2
2	65K	7.17	4.24	69.1
4	263K	5.94	3.44	72.6
6	1.05M	4.69	2.84	65.1
8	4.21M	3.67	2.44	50.4
10	16.8M	–	2.1	–
12	67.4M	–	1.87	–
14	269.8M	–	1.65	–

Table 4: Performance comparison (in fps) between triangle-based ray tracing (OpenRT) and direct ray tracing of Bézier patches for different subdivision depths, measured in the Head scene with 915 initial patches, 640x480 pixels, and simple diffuse shading on a single Xeon 2.2 GHz processor. While the (not yet fully optimized) Bézier code is somewhat slower than the high-performance triangle code, its performance penalty on average is only 30-50%. For that tolerable price, it offers a much more compact representation that allows for rendering much more complex models, supporting higher accuracy due to finer tesselation, and animation of the patches without a need for rebuilding kd-trees for the tesselated triangles.

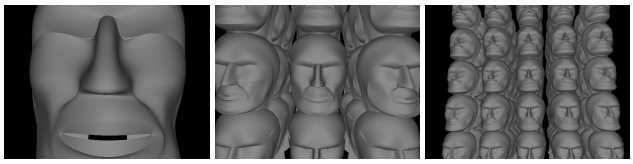

Figure 5: The overall performance is largely independent of the actual number of patches and varies only by roughly 20% while zooming out from a single head model to a view where more than 125 head models are visible. Each head model consists of 915 subdivision patches and is rendered with four refinement steps at 1-2 fps on a single processor.

5. SUBDIVISION SURFACES

As a second example we apply the generic ray tracing algorithm to Loop subdivision surfaces. The very different structure of this implementation suggests that the general approach can also be applied to other types of curved surfaces.

Subdivision surfaces offer a number of advantages over splines, a major one being the more general topological structure including continuity control between adjacent subdivision meshes. Subdivision surfaces have become particularly popular in the animation industry due to the ability to model organic-like shapes.

The Loop construction is simple and often used in practical applications [7, 9, 15]. It is an approximating triangular scheme where the final converged surface in general does not interpolate the original control points (see Figure 7). Most notable the Loop surfaces lies in the convex hull of its control mesh, which simplifies our pruning test.

Subdivision schemes are more difficult to handle than plain Bézier splines for mainly two reasons: Irregular data structures and overlapping control meshes. Irregular data structures are caused by base meshes that contain extraordinary vertices that can have an arbitrary valence — different than the regular value of six. This varying size of the data structure complicates the data layout as well as the algorithms dealing with them.

The overlapping control meshes are more problematic and are the main reason for the reduced performance compared to the Bézier surfaces. Adjacent Loop patches share a significant part of their

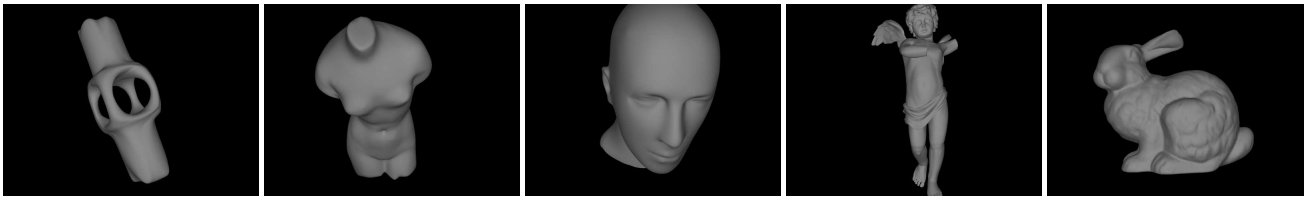

Figure 6: A set of test scenes made up entirely of subdivision surfaces. The number of base triangles is 1,432, 5,672, 5,680, 53,624, and 277,804 (from left to right). See Figure 9 for performance results.

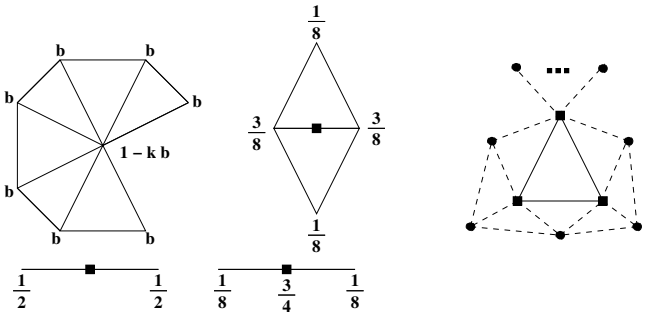

Figure 7: Left: Subdivision rules according to the Loop scheme for edge, interior, and boundary vertices, Right: having only one irregular vertex (top) allows for a simplified data layout for the 1-neighborhood

control mesh such that their bounding boxes overlap considerably. As a result we can prune patches only much later during refinement, increasing the computational cost. We currently compute a conservative but non-optimal bounding box of all control points for a patch. This could be improved by known approaches to quickly compute tighter bounding boxes.

5.1 Data Layout for Subdivision Surface

As just mentioned the data layout is complicated by the fact that any control points in the base mesh can have irregular topology. Handling the four different cases of patches with zero to three irregular control points and appropriate data structures would be too complex and slow.

Instead we distinguish only between fully regular patches and those with a single irregular control point (see Figure 7). More complex patches are simply subdivided once, yielding four subpatches with a maximum of one irregular vertex each. For each base triangle we store its three vertices together with its 1-neighborhood. This results in 12 vertices for regular triangles and $6+val(irreg.vertex)$ for irregular triangles.

Because of the different numbers of control points as well as the non-uniform weighting during refinement, we currently still store the mesh as an array of structures. While this is less elegant and less efficient for SIMD instructions (using only 3 of the 4 slots in a register) we can still operate on an entire control point at once.

5.2 Core Operations

Similar to the pruning test for Bézier patches we need to compute the signed distance of all control points from each of the two planes. For each vertex 3 parallel multiplications, 3 additions, and 1 special instruction for getting the sign have to be performed. This results in $2*12*3*2=72$ operations for a regular triangle and $(6+val(irregularvertex))*3*2*2$ operations for an irregular triangle.

Refinement of Loop patches generates four instead of only two new patches. While this should help in converging to the intersection point more quickly this advantage is offset by the mentioned large overlap between the control meshes of adjacent patches.

Similar to the Bézier case for each ray we read only the control points of a few patches from memory. All other data is computed in the tight inner loop of the refinement and pruning algorithm. As a result all this data stays in the first level cache and the approach is completely compute bound.

For regular triangles, each refinement operation must compute 3 new interior vertices and $3*6=18$ new vertices in the 1-neighborhood of the sub-patches. Each new interior vertex requires 7 multiplications and 6 additions while the others require 2 additions and 2 multiplications finally resulting in $7*3+18*2=57$ multiplications and $6*3+18*2=54$ additions. For irregular triangles the number increases according to the valence of the irregular vertex.

Once the maximum recursion depth is reached we essentially perform the same culling and fast ray triangle intersection tests as for Bézier patches.

We compute the normal vector at each triangle vertex by evaluating a weighted sum over adjacent vertices. The final normal itself is a linear combination of the three vertex normals with respect to the barycentric hit point coordinates.

5.3 Core Performance for Subdivision Patches

Table 5 lists the measured cost of each of the basic operations for direct ray tracing of subdivision surfaces.

Step	Cycles(reg)	Cycles(irreg)
Pruning	172	222
Refinement	405	600
Final Intersection	169	169
Normal	450	550

Table 5: Runtime of the basic operations for Loop subdivision surfaces in CPU cycles.

Compared to the cost of Bézier patches (see Table 1) subdivision surfaces are significantly more expensive. However, each refinement step for subdivision surfaces corresponds to *two* steps for Bézier patches – one in each parametric direction.

5.4 KD-Trees for Larger Scenes

For larger scenes we again need to build a spatial index in order to reduce the number of subdivision surfaces that need to be accessed. We currently use the bounding box of the entire control mesh of a patch for creating the kd-tree similar to the Bézier case. However, due to the increased overlap between adjacent patches, the kd-tree cannot separate them as well, resulting in an increased number of subdivision patches enumerated per ray.

The higher cost is mainly caused by the increased number of instructions required by the non-optimal SIMD usage. Note also

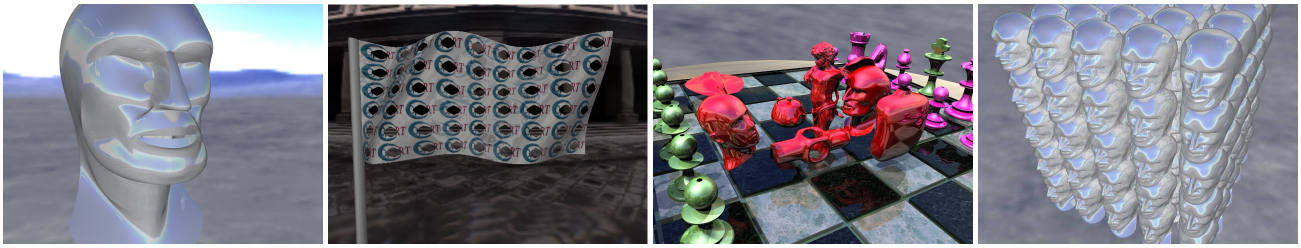

Figure 8: Free-form surface scenes interactively ray-traced: Bi-cubic Bézier spline surfaces, spline surface with trimming curves, and Loop subdivision surfaces. Even complex scenes with several hundred thousands of patches can be rendered efficiently and with minimal memory requirements.

Scene	Mech	Venus	Head	Cupid	Bunny
Triangles	1.43k	5.67k	5.68k	53.6k	277k
NO-SAH	69.58	46.55	33.58	21.14	52.11
SAH	43.59	34.79	26.23	17.27	45.92
Reduction %	37.35	25.26	21.88	18.30	11.87
With MB	9.18	8.18	5.78	7.22	19.02
Reduction %	78.94	76.48	77.96	58.19	64.51

Table 6: The number of base triangles for each subdivision scene and the average number of patches accessed per ray. While the surface area heuristics (SAH) for building the kd-tree combined with mail-boxing (MB) improves the culling by up to 78%, we still have to process more patches per ray than in the Bézier case.

that supporting arbitrary valences results in a cost increase by up to 50% (e.g. for refinement).

The surface area heuristics alone can only reduce the average number of patches per ray by a moderate 18–37% (see Table 6). However, many of the false positives can be caught by mail-boxing raising the rejection rate to 58–78%. Due to the overlap a significantly larger number of patches (4-5x) must be handled per ray if compared to the Bézier case. The performance is reduced proportionally suggesting that better bounding algorithms can speed up the approach further.

The overall performance for subdivision surfaces as a function of subdivision depth essentially follows that for Bézier patches (see Figure 9). It still yields interactive performance for a moderate number of refinement steps. These measurements suggest that future work should focus on computing tighter bounding boxes in order to significantly reduce the number of patches accessed per ray.

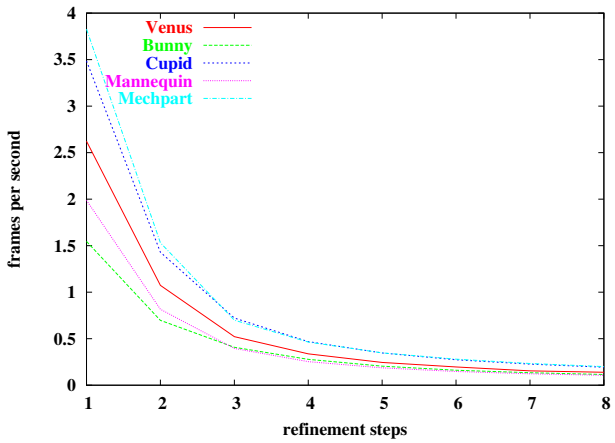

Figure 9: Performance of direct ray tracing of subdivision surface as a function of refinement depth for the scenes shown in Figure 6. Note that one step here corresponds to two steps in the Bézier case. The computations include normal evaluations and diffuse shading and are performed at a resolution of 640x480 on a single Intel Xeon 2.2 GHz processor.

6. DISCUSSION AND FUTURE WORK

In this paper we presented a simple generic approach for ray tracing of free-form surfaces. Its main difference to previous techniques is that we focused on simple and robust algorithms, avoiding complex control flow for handling of special cases, and efficient mapping of the code onto today's processor architectures. For bi-cubic Bézier patches as well as Loop subdivision surfaces we are able to significantly improve performance and achieve interactive performance of a few frames per second even on a single processor.

The performance for Bézier splines is close to that of triangle meshes, while subdivision surfaces currently still suffer from sub-optimal kd-trees. This lead to significant overhead during intersection computations as we have not yet optimized the bounding box computation for these types of surfaces. Note also, that the current implementation still leaves significant room for speedup.

The approach selected here reduces the size of patches by half in each step. In contrast Bézier clipping can converge faster once the refined patch becomes flatter. Unfortunately, we have not been able to include a direct comparison with a similarly level of optimization. However, first experiments with implementing Bézier clipping strongly suggests that its performance will remain significantly below our approach even after full optimization. In contrast to the simple and robust subdivision strategy chosen here, Bézier clipping requires that in each refinement step all control points are transformed from Euclidean space to a 1D distance space and that a convex hull and its intersections with a line are computed there. In addition it must carefully handle several special cases in the inner loop [1]. Also each subdivision step is twice as expensive as the surface needs to be trimmed on both sides of the parametric direction.

Bézier clipping must refine the surface to a finer level on average because it directly computes the intersection point instead of approximating with triangles at the end. This means that subdivision can only terminate once the size of the patch is small enough in parametric space. Finally, errors are significantly more difficult to control due to computations in the intermediate coordinate spaces, in contrast to the simple geometric error control used in our technique.

The presented approach can easily be integrated into existing ray

tracing systems. This allows to completely reuse all layers above the core ray-tracing, e.g. the shading framework, distributed processing, and others. The algorithm also requires only a constant and tiny amount of memory in addition to the original surface representations for keeping a stack of recursively subdivided patches. The size of this stack is strictly limited by the maximum recursion depth.

The use of the kd-tree leads to an algorithm that essentially uses constant time even for complex scenes (see Figure 8). While the traversal of the kd-structure requires only $O(\log n)$ time it is still a major part of ray tracing scenes with large numbers of triangles because of the large depth of the tree and the many cells that need to be traversed. In this case usually less than 30 % of the time are spent on intersection computations.

In the case of free-form surfaces we have significantly shifted the computation from traversal to intersection: Our kd-trees are shallow but intersection computations are costly. Thus even for highly complex scenes the increased kd-tree traversal cost is hardly noticeable. Each traversal yields a small number of patches for intersection testing. This number only depends on the amount of overlap between visible patches. Also, the cost per intersection computation varies little. Because the refinement depth is limited the cost depends only on the orientation of the surface with respect to the ray.

The use of free-form surfaces reduces the size of the data to represent geometry compared triangle meshes. For dynamic scenes this reduces the amount of processing to update the geometry (see Figure 8) and reduces the network bandwidth for the case of distributed rendering.

A major issue of the approach is the fact that we do not perform adaptive refinement, which at first sight seems beneficial for avoiding unnecessary refinements. However, it turns out that the cost of testing refinement is more expensive as performing the refinement. We avoid this cost together with the increase of code complexity to fix potential cracks in areas of different refinement depth.

6.1 Future Work

The missing adaptivity is a price we pay for this decision. We can, however, still adjust the refinement depth for each object separately (e.g. depending on its distance from the camera). However, for highly irregular objects with large flat and highly curved regions this might be insufficient and could be improved in future work.

CAD systems and animation packages offer many different types of spline surfaces including B-Splines, NURBS, or other higher order spline model that we do not support directly. In general this offers two options: Approximating the spline by supported primitives (i.e. bi-cubic splines) or extending the primitive support. For this paper we have taken the first approach but have to accept some approximation error. It is unclear yet what would be good techniques for handling rational or higher order basis functions. Similarly we have not yet investigated other subdivision schemes.

Finally, it would be useful to investigate the use of packet ray tracing also for free-form surfaces to better use the significant coherence between adjacent rays also during refinement and pruning.

7. REFERENCES

[1] S. Campagna, P. Slusallek, and H. P. Seidel. Ray Tracing of Parametric Surfaces. *The Visual Computer*, 13(6):265–282, 1997.

[2] M. Guthe and R. Klein. Efficient nurbs rendering using view-dependent lod and normal maps. In *Journal of WSCG*, volume 11. February 2003.

[3] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[4] Intel Corp. *IA-32 Intel Architecture Optimization – Reference Manual*, 2001.

[5] Intel Corp. Intel Pentium III Streaming SIMD Extensions. http://developer.intel.com/vtune/cbts/simd.htm, 2002.

[6] Intel Corp. *Prescott New Instructions*, 2003. http://www.intel.com/cd/ids/developer/asmona/eng/microprocessors/ia32/pentium4/resources/.

[7] L. Kobbelt, K. Daubert, and H. Seidel. Ray Tracing of Subdivision Surfaces. *9th Eurographics Workshop on Rendering proceedings*, pages 69–80, 1998.

[8] R. Lewis, R. Wang, and D. Hung. Design of a Pipelined Architecture for Ray/Bezier Patch Intersection Computation. *Canadian Journal of Electrical and Computer Engineering*, 28(1), 2002. Available at http://www.idt.mdh.se/ ejn04/referenser/.

[9] C. Loop. Smooth subdivision surfaces based on triangles. 1987. Master Thesis, University of Utah, 1987.

[10] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical Ray Tracing of Trimmed NURBS Surfaces. *Journal of Graphics Tools: JGT*, 5:27–52, 2000.

[11] A. Medvedev. http://www.digit-life.com/articles/atitruform/.

[12] K. Mueller, T. Techmann, and D. Fellner. Adaptive Ray Tracing of Subdivision Surfaces. *Computer Graphics Forum (Proceedings of Eurographics '03)*, pages 553–562, 2003.

[13] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray Tracing Trimmed Rational Surface Patches. *Computer Graphics (Proceedings of SIGGRAPH '90)*, pages 337–345, 1990.

[14] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, pages 119–126, 1999.

[15] J. Stam. Evaluation of Loop Subdivision Surfaces. *SIGGRAPH 99 Course Notes*, 1999.

[16] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.

[17] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[18] S. Wang, Z. Shih, and R. Chang. An Efficient and Stable Ray Tracing Algorithm for Parametric Surfaces. *18th Journal of Information Science and Engineering*, pages 541–561, 2001.

[19] C. Woodward. Ray Tracing of Parametric Surfaces by Subdivision in Viewing Plane. *Theorie and Practice of Geometric Modeling*, 1989.