Interactive Ray Tracing of Trimmed Bicubic Bézier Surfaces without Triangulation

Markus Geimer Oliver Abert
Institute for Computational Visualistics
University of Koblenz-Landau
Universitätsstraße 1
D-56070 Koblenz, Germany
mgm@uni-koblenz.de abert@uni-koblenz.de

ABSTRACT

By carefully exploiting the resources of today's computer hardware, interactive ray tracing recently became reality even on a single commodity PC. In most of these implementations triangles are used as the only geometric primitive. However, direct rendering of free-form surfaces would be advantageous for a large number of applications, since robust tessellation of complex scenes into triangles is a very time-consuming process. Additionally, scenes consisting of free-form surfaces require less memory and provide a much higher precision resulting in less rendering artifacts.

In this paper, we present our implementation of an efficient and robust algorithm for rapidly finding intersections between rays and trimmed bicubic Bézier surfaces. Using SIMD instructions provided by many of today's CPUs, we perform the intersection test of a packet of four rays with a single Bézier surface in parallel. An optimized bounding volume hierarchy provides good initial guesses needed for fast convergence of the Newton iteration, which forms the core of our intersection algorithm. As a result, we demonstrate that it is feasible to render complex scenes of several thousand Bézier surfaces at video resolution with interactive frame rates on a single PC.

Keywords

Interactive Ray Tracing, Bézier Surfaces, Trimming

1. INTRODUCTION

Free-form surface representations such as splines, NURBS, or subdivision surfaces provide a simple but still powerful way of describing three-dimensional geometrical objects for use in computer graphics applications. Unlike triangle meshes, which are the second commonly used way of defining 3D shapes, they are able to describe curved surfaces exactly. Therefore, free-form surfaces form the foundation of most CAD systems used in the industry today.

WSCG 2005 conference proceedings ISBN 80-903100-7-9
WSCG'2005, January 31-February 4, 2005
Plzen, Czech Republic.

If the models should be displayed in an interactive setting, however, free-form surfaces are currently tessellated into triangles as well, since this is the only primitive that can be handled by today's rasterization hardware. For this kind of applications it would therefore be desirable to render free-form surfaces directly.

Direct rendering of free-form surfaces instead of triangle meshes has a number of advantages. Obviously, the time-consuming overhead of triangulating the surfaces can be avoided. Secondly, due to the smaller number of primitives the costs for additional preprocessing needed for most rendering algorithms, e.g. building up acceleration data structures, are also reduced. Moreover, representing objects as free-form surfaces requires less memory, which can be a limiting factor for complex scenes. In addition, rendering free-form surfaces directly provides a much higher precision resulting in less rendering artifacts. For example, cracks between adjacent surfaces due to different tessellation parameters can be completely avoided.

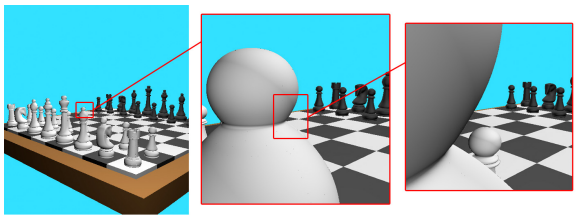

Figure 1. The Chessboard from different view points. Note that due to the direct rendering of free-form surfaces, the objects remain curved, even from the shortest viewing distance.

Finally, free-form surfaces are often used in conjunction with trimming curves that cut out parts of the surface in the parametric domain. Robust tessellation of free-form surfaces with trimming curves is a non-trivial task with a high computational cost.

In contrast to current rasterization hardware, ray tracing is capable of rendering every primitive for which the intersection between a ray and the surface can be calculated [Whi80a]. Nevertheless, it has the reputation of being a very time-consuming rendering algorithm. However, recently it has been shown that it is possible to achieve interactive frame rates even on a single commodity PC by carefully exploiting the resources of today's CPUs [Wal01a].

In this paper, we present the details of our implementation of the intersection test between a ray and a trimmed bicubic Bézier surface in the context of an interactive ray tracing system. While we do not introduce any new intersection algorithm, we rather show that a significant speed-up can be achieved by carefully optimizing a well-known intersection technique. As a result, we demonstrate that it is feasible to render complex scenes of several thousand Bézier surfaces at video resolution with interactive frame rates on a single PC.

Although we currently restrict ourselves to bicubic Bézier surfaces, our approach may be extended to support other parametric surfaces as well. In addition, more general spline surfaces such as NURBS can be converted into Bézier patches [Roc89a].

2. PREVIOUS WORK

Ray tracing at interactive frame rates has been first presented by [Muu95a]. Later, [Par99a] described a full-featured interactive ray tracing system running on a shared-memory supercomputer that is able to handle arbitrary geometry, including parametric surfaces (e.g. NURBS). However, they had to use expensive high-end hardware to achieve this goal.

By contrast, [Wal01a, Wal01b] presented a highly optimized ray tracer running on a cluster of commodity PCs, paying careful attention to data layout, coherence, and caching issues. In addition, their system extensively uses SIMD instructions provided by most of today's CPUs to trace packets of rays in parallel, thereby achieving a speed-up of more than an order of magnitude compared to other well-known ray tracers. Meanwhile, the employed algorithms have been further improved [Wal04a]. However, this ray tracer is restricted to triangles as the only geometric primitive.

In the last 25 years, a variety of algorithms has been proposed to calculate the intersections between a ray and parametric surfaces of different kinds. One common approach is to use a multivariate Newton iteration. This method has the advantage of being general enough to handle any parametric surface, but requires a good initial value to ensure correctness and fast convergence.

For example, [Swe86a] refine the control meshes of B-spline surfaces until they closely approximate them. Then, the intersection of the ray and the control mesh is used as the initial value of the following Newton iteration. By contrast, [Mar00a] employ a hierarchy of bounding volumes enclosing disjoint regions of NURBS surfaces to yield a suitable initial value.

Another numerical approach for Bézier surfaces is called Bézier Clipping [Nis90a]. This algorithm tries to iteratively identify regions of the patch that are known not to be intersected by the ray, thereby restricting the parameter domain where intersections can occur. This approach is also used by [Wan01a], who combine Bézier Clipping with Newton iteration. Additionally, they exploited the coherence of neighboring rays to speed up the calculation. Nevertheless, all approaches mentioned above were far from interactive.

Recently, [Ben04a] presented their implementation of a subdivision method, which refines the control meshes of the surfaces on-the-fly and calculates an approximate intersection point using a triangle mesh generated from the control points. Depending on the model and the number of refinement steps, they achieve up to 5.5 fps at video resolution on a single PC. Nevertheless, the number of refinement steps has to be the same for all surfaces to avoid cracks between adjacent patches.

3. SYSTEM OVERVIEW

Before going into the details of our implementation of the intersection test between a ray and a bicubic Bézier surface, we present a brief overview of the underlying interactive ray tracing system.

Similar to [Wal01a], we use SIMD instructions found in many of today's CPUs to trace packets of

four rays in parallel. This applies to both the traversal of an acceleration data structure as well as the actual intersection calculations. However, instead of targeting only at a single CPU architecture, we have implemented our ray tracing system on top of a SIMD abstraction layer that allows us to write platform-independent SIMD code (see [Gei03a] for details). Currently, Intel's SSE [Int04a] and Motorola's AltiVec [Mot99a] instruction sets are supported, as well as a special mode that uses the FPU to emulate the specified functionality.

In order to achieve a good rendering performance, it is crucial to reduce the total number of intersection calculations to a minimum. Therefore, we employ a hierarchy of axis-aligned bounding boxes that is iteratively traversed in depth-first order [Smi98a]. Although other acceleration data structures are usually considered to be faster in the context of ray tracing, we have found that this approach is well suited for a SIMD implementation and also adapts well to our intersection algorithm presented below.

Currently, our ray tracing system is restricted to static scenes, allowing only interactive walk-throughs. However, it could be easily extended to support dynamic scenes with hierarchical movements as well using the ideas presented in [Lex01a] and [Wal03a].

Because ray tracing naturally lends itself to a parallel implementation, our system is no exception. At present, we support rendering with multiple threads, which allows us to take advantage of multiprocessor PCs as well as Intel's HyperThreading technology [Int04b].

4. OUR APPROACH

Instead of using complex algorithms, we rather take a "brute force" approach. While this doesn't seem to be very clever at first sight, it has been shown that a carefully optimized implementation can easily out-perform more complex algorithms on current CPU architectures (e.g. [Wal01a] or [Ben04a]).

Our intersection algorithm combines many known techniques to achieve a fast computation of the intersection point of a ray and bicubic Bézier patches as well as the corresponding surface normals. Similar to [Mar00a], we employ a hierarchy of axis-aligned bounding boxes to find a suitable initial guess needed for the Newton iteration that is used to calculate the intersection point. In the following subsections, we will describe the core components of our approach.

Preprocessing

As a first step, the scene description file is read (currently, the IGES format is partially supported) and converted into a binary data file that can then be processed by the actual rendering application.

During this preprocessing, we convert the trimming curves originally represented as B-splines into piecewise cubic Bézier curves for use with our trimming algorithm and build up the bounding volume hierarchy that is used to speed up the intersection calculation.

Compared to other bounding volume hierarchies, on the lowest level our bounding boxes do not surround entire scene objects (i.e. Bézier surfaces) or even lists of objects. Instead, we create them for small, disjoint regions of the individual surfaces. On the one hand, this leads to tighter bounds, thereby reducing the number of unnecessary intersection calculations. On the other hand, we are able to get better initial guesses for the Newton iteration used to calculate the actual intersection point.

Therefore, we recursively subdivide each Bézier surface into a larger number of subpatches and calculate their corresponding control points using de Casteljau's algorithm. Due to the convex hull property of Bézier surfaces, we can use these control points to determine an axis-aligned bounding box for each subpatch.

At present, we alternately subdivide the surfaces resp. subpatches at the mean of the parameter domain in u and v direction, until the generated patches are reasonably flat or a predefined maximum subdivision depth has been reached. These two parameters can be easily controlled by the user. Unfortunately, they are scene dependent and must be chosen with care. For our test scenes, we have found that a maximum subdivision depth of 4-6 is already sufficient.

Since our Bézier surfaces are subdivided into small, disjoint regions by the bounding volume hierarchy, the individual subpatches can be classified during preprocessing to improve the rendering performance of trimmed surfaces. For each region that is known to be enclosed by a trimming curve (i.e. considered not to be part of the patch), we can immediately discard the corresponding subpatch as well as its associated bounding box, thereby avoiding any intersection calculation. By contrast, for regions that are known to lie completely inside the patch, the trimming test can be skipped during rendering. Otherwise, at least one trimming curve cuts out a part of the region and we have to perform the entire trimming calculation.

As we use the original Bézier surface for the actual intersection test, the control points of the subpatches are only needed for creating the bounding boxes and the aforementioned classification, so they can be discarded immediately afterwards.

Finally, the bounding volume hierarchy is created from the bounding boxes of the remaining subpatches using the top-down approach presented by [Gol87a].

Patch Representation and Data Layout

A bicubic Bézier surface is given by its 16 control points and can be represented in matrix form through

$$S(u,v) = [U][N][CP][N][V]^T .$$

Here, the 4x4 matrix $[CP]$ stores the control points, $[N]$ contains the Bernstein polynomial coefficients, and $[U] = [u^3\ u^2\ u\ 1]$ resp. $[V] = [v^3\ v^2\ v\ 1]$. Obviously, the inner product $[P] = [N][CP][N]$ can be computed in advance.

We store the 16 vector components of $[P]$ as SIMD data in an array (see Figure 2). The first three elements of each SIMD variable store the X, Y, and Z component of the corresponding matrix element, whereas the last component remains unused. With this approach, each Bézier surface can be represented by exactly 16 * 4 * *sizeof*(float) = 256 bytes.

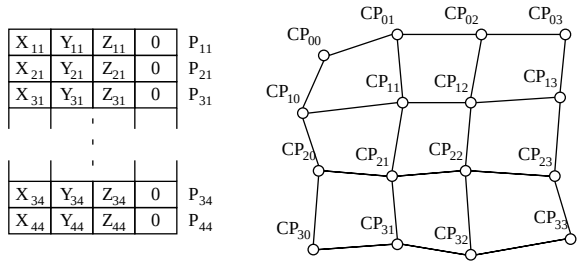

Figure 2. Patch representation: The matrix $[P]$ computed from the control points CP_{ij} is stored column-major in an array of SIMD variables, leaving one component of each element unused.

Storing the patch data as a structure-of-arrays, i.e. storing the same components of four control points together in a single SIMD variable, would reduce the memory requirements by 64 bytes per surface, but during our experiments we have found that this data layout results in a significantly slower evaluation algorithm due to the necessary shuffling of data. Moreover, the fourth component of each element in our SIMD data array can be used to store additional patch information, e.g. a pointer to the associated list of trimming curves.

As already mentioned before, the subpatch data structure does not contain any control point information. Here we only store the parameter domain of the patch, the classification flag indicating whether it has to be trimmed or not, and the pointer to the original Bézier surface. For alignment reasons, we pad this data structure to a total size of 32 bytes.

For each bounding box, we store the minimum and maximum coordinate value along each axis, a pointer to the associated geometry (i.e. a subpatch), and the skip pointer used during the traversal of the hierarchy. This data structure also occupies 32 bytes.

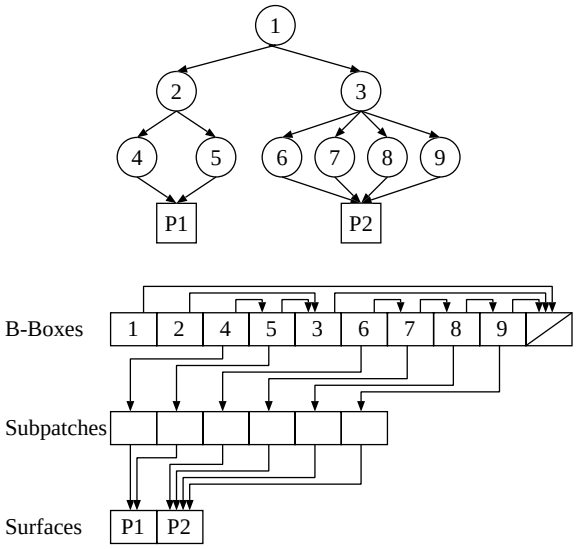

Figure 3. Data layout: During preprocessing, the example hierarchy shown at the top is converted into the internal representation (a set of arrays) shown at the bottom.

Rendering Core

As already stated in section 3, our rendering core traces packets of four rays in parallel using SIMD instructions. First, each ray packet generated by the camera iteratively traverses the bounding volume hierarchy to restrict the number of intersection candidates. Here, if any ray of the packet hits a bounding box, the entire packet has to continue the traversal of its children.

Since the leaf nodes of our hierarchy correspond to small surface regions and not to an entire Bézier patch, this approach does not only restrict the number of intersection candidates, but also the parametric domain where intersections may occur. Provided that a proper hierarchy has been created, the center of the enclosed parametric domain can then be used as an initial guess for a Newton iteration that calculates the actual intersection point between the ray and the surface. Of course, this can also be done for four rays in parallel using SIMD instructions.

4.3.1 Intersection Test

The core of the intersection test is similar to the approach presented by [Mar00a] who solved the problem for NURBS surfaces but without targeting interactivity.

We represent each ray by two orthogonal planes $P_1 = (N_1, d_1)$ and $P_2 = (N_2, d_2)$ where the N_i are orthogonal vectors of unit length, perpendicular to the ray direction D. The d_i are given by $d_i = -N_i \circ O$. Here, O denotes the origin of the ray. To find the intersection point between the ray

and a parametric surface $S(u,v)$, we have to solve for the roots of

$$R(u,v) = \begin{bmatrix} N_1 \cdot S(u,v) + d_1 \\ N_2 \cdot S(u,v) + d_2 \end{bmatrix}.$$

Several numerical methods exist that could be used to target this problem. We have chosen the classical approach of Newton iteration for several reasons: First, it converges quadratically if the initial guess is close to the actual root, which can be assured by our bounding volume hierarchy. Secondly, the surface derivatives exist and are very easy to compute. And last but not least, this algorithm is well suited for an implementation using SIMD instructions.

Basically, Newton's method is a Taylor series which is truncated after the first derivative. As we are solving a two-dimensional problem, the Newton step is defined as

$$\begin{bmatrix} u_{n+1} \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} u_n \\ v_n \end{bmatrix} - J^{-1} \cdot R(u_n, v_n),$$

where J is the Jacobian matrix of R which is given by

$$J = \begin{bmatrix} N_1 \cdot S_u(u,v) & N_1 \cdot S_v(u,v) \\ N_2 \cdot S_u(u,v) & N_2 \cdot S_v(u,v) \end{bmatrix}.$$

Here, S_u and S_v denote the partial derivative in the corresponding parametric direction. The inverse of the Jacobian can be efficiently computed using the submatrices J_{ij} of J that remain when the ith row and the j th column are removed:

$$J^{-1} = \frac{1}{\det(J)} \cdot \begin{bmatrix} J_{22} & -J_{12} \\ -J_{21} & J_{11} \end{bmatrix}.$$

We continue the iteration until one of three criteria is met: An intersection between the ray and the surface is found, if and only if we are closer to the root than some user defined threshold ε

$$\left| R(u_n, v_n) \right| < \varepsilon.$$

Otherwise, the iteration continues until either this threshold criterion is met, the next iteration takes us further away from the root, i.e.

$$\left| R(u_{n+1}, v_{v+1}) \right| > \left| R(u_n, v_v) \right|,$$

or a maximum number of iterations has been performed.

Unfortunately, since we employ SIMD instructions to calculate four intersections at once, the iteration can be stopped only if all rays of a packet meet any of these criteria. However, this is still more than three times faster than computing the intersections sequentially.

4.3.2 Evaluation

The Newton iteration often needs to evaluate surface points as well as partial derivatives for given parameter values (u,v). In this section we present a way how these can be computed efficiently.

Basically, we have to compute the product of three matrices. That is

$$S(u,v) = [u^3 \; u^2 \; u \; 1][P][v^3 \; v^2 \; v \; 1]^T.$$

A naive implementation would simply compute the two matrix products, which consists of 60 multiplications and 45 additions ($[P]$ contains three-dimensional vectors!), ignoring the operations needed to compute of $[U]$ and $[V]$. However, as the surface evaluation is used very often, it is necessary to optimize it as much as possible.

First of all, the equation above can be rewritten as

$$S(u,v) = \sum_{i=1}^{4} [U][P_i][V_i],$$

where $[P_i]$ denotes the ith column of $[P]$ and $[V_i]$ the ith row of $[V]$. Note that $[V_i]$ represents only a single floating-point.

Given the parameters (u,v), we can then evaluate a point $S(u,v)$ on the surface using the following C-like pseudo code fragment:

```
u2 = u * u
u3 = u * u2
f  = 1
s  = 0
i  = 15
while (i >= 0) {
   t = p[i--]
   t = t + u  * p[i--]
   t = t + u2 * p[i--]
   t = t + u3 * p[i--]
   s = s + t  * f
   f = f * v
}
return s
```

This code is pretty straightforward to implement using SIMD instructions, thereby calculating four surface evaluations in parallel. Moreover, we can easily take advantage of the combined multiply-add instructions provided by some CPU architectures (e.g. PowerPC). Furthermore, by performing loop-unrolling, additional unnecessary operations can be eliminated (for example the last two statements in the loop for the first iteration).

Scene	# Patches	# Trims	#B-Boxes	Memory consumption	Preprocessing time (min)	Average Framerate
Teapot	32	-	2736	150 kB	0:01	6.4 fps
Cessna	1555	-	53216	3.2 MB	0:03	4.2 fps
Chessboard	16182	-	227794	15.6 MB	0:13	6.7 fps
VW Polo	11576	38556	448500	28.3 MB	5:26	5.1 / 3.4 fps*

Table 1. Statistics for our test scenes (*=trimming disabled/enabled)

The computation of both partial derivatives can be executed even faster. For example, the derivative in u direction is

$$S_u(u,v) = [3u^2 \;\; 2u \;\; 1 \;\; 0][P][V]^T .$$

If we apply the same optimizations as presented above, the calculation of both partial derivatives simplifies to 33 multiplications and 33 additions each.

4.3.3 Trimming

For surface areas that have been classified to need trimming during the preprocessing step, an additional 2D point-in-curve test is performed after an intersection has been found. Currently, this is done sequentially for the (up to) four intersection points, as we do not employ any SIMD instructions for trimming yet.

In addition, we currently perform this point-in-curve test for all trimming curves associated to the original Bézier surface and not only for those parts that are relevant for the examined region (i.e. the subpatch containing the intersection point).

In our implementation, we use the point classification approach presented by [Nis90a]. However, instead of using Bézier Clipping to subdivide the trimming curves into three segments if they cannot be clearly classified, we again take a brute force approach by splitting the curves at $t = 0.5$ and recursively test both segments.

4.3.4 Shading

Finally, all valid intersection points are shaded for display. At present, we support the simple Phong shading model [Bui75a]. This is also done using SIMD instructions, calculating the color of up to four intersection points in parallel. Here, additional rays for calculating shadows, reflections or refraction may be generated.

5. RESULTS

In this section we present some timings of our ray tracing system for a couple of test scenes of varying complexity (see Table 1 for the numbers and Figure 4 for renderings), measured on a dual processor PowerMac G5 running at 2 GHz using only a single rendering thread. All images are generated at a fixed screen resolution of 512x512 pixels using simple Phong shading. Note that all scenes are lit by a single point light source, except for the Chessboard which is lit by three point lights.

Our ray tracing system based on the algorithms presented above is able to render the Utah Teapot consisting of 32 Bézier patches at an average frame rate of 6.4 fps, whereas the more complex Cessna and Chessboard scenes can be rendered at 4.2 frames/s and 6.7 frames/s respectively.

To compare our results to the approach presented by [Ben04a], we additionally rendered the Teapot model at a screen resolution of 640x480. Here, the frame rate drops to 5.5 fps due to the larger number of pixels. This frame rate is comparable to their result for a subdivision depth of 0 (i.e. directly rendering the control mesh), already taking into account that we use a different test platform that is approx. 35 percent faster. Nevertheless, as the number of refinement steps usually has to be higher in order to obtain a good rendering quality, our method easily outperforms the subdivision approach for this model. For example, when using only four refinement steps, our implementation is already twice as fast.

In contrast to the other test scenes, the VW Polo is the only model that contains trimming curves. Here, the large number of trims comes from the conversion of the original B-splines into piecewise cubic Bézier curves. It should also be noted that most of the preprocessing time is spent for reading the IGES file (~15%) and for the classification of the generated subpatches (~78%). However, the preprocessing code is fairly unoptimized at the moment.

As can be seen from Table 2, the classification of the patches works quite well for the Polo model. Trimming calculations need to be performed only for a small fraction of the generated patches during rendering, as most subpatches can be categorized in advance.

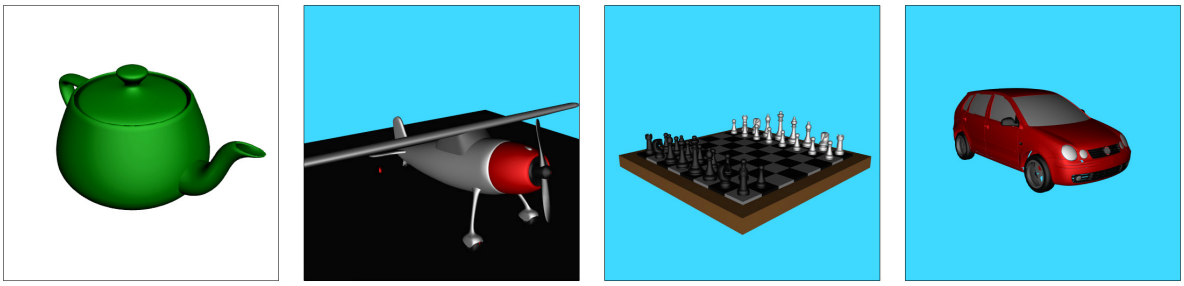

Figure 4. Renderings of our test scenes: Utah Teapot, Cessna, Chessboard, and VW Polo (with trimming).

With trimming disabled, the Polo can be displayed at an average frame rate of 5.1 frames per second. After trimming is enabled, the average frame rate drops to 3.4 fps. On the one hand, this can be explained by the fact that trimming is currently performed sequentially for the intersections found. On the other hand, we still perform a lot of unnecessary trimming calculations, as the point-in-curve test is executed for all trimming curves associated to the original Bézier patch and not only for those parts that are relevant to the examined subpatch.

# Subpatches (total)	496827	100.0%
Without trims	70523	14.2%
With trims	426304	85.8%
Totally in	241480	48.6%
Need trimming	33987	6.8%
Discarded	150837	30.4%

Table 2. Number of subpatches and their classification for the VW Polo model.

For comparison, we also rendered a tessellated model of the VW Polo consisting of 326159 triangles using a modified version of our ray tracing system, achieving an average frame rate of 4.8 fps. Although this is still faster than ray tracing the Bézier-based model, one should keep in mind that comparable models used for design reviews in the industry today usually consist of several million triangles that easily occupy gigabytes of memory.

As memory access has been shown to be a limiting factor for ray tracing [Wal01a], direct rendering of bicubic Bézier surfaces can already be a valuable alternative to triangle-based approaches, especially for large models.

6. FUTURE WORK

Since our interactive ray tracing system for trimmed bicubic Bézier surfaces is still at an early stage of development, there is much room for improvements left, both in terms of rendering performance and image quality.

Currently, we are working on a distributed version of our rendering system, running on a cluster of heterogeneous PCs. First results indicate that the performance scales almost linearly with the number of processors, as could be expected.

In addition, the trimming code can be improved in different ways. As already stated before, the trimming curves can be split up in such a way that for each subpatch only the relevant parts have to be tested. Secondly, it would be interesting to examine if the usage of SIMD calculations can also speed up the trimming calculations. And finally, efficient handling of the special case of line segments may further improve performance.

Moreover, the creation of our bounding volume hierarchy could be improved by using a more sophisticated heuristic to guide the subdivision of the Bézier patches (e.g. taking the curvature of the surface into account) instead of using the simple flatness criterion.

As already stated in the introduction, our system is currently restricted to bicubic Bézier surfaces. Extending the presented approach to Bézier patches of arbitrary degree is relatively easy to implement. Thinking even further, it would be interesting to investigate more complex surface descriptions such as B-splines or NURBS.

Finally, the next step in terms of image quality will be to add support for the typical ray tracing effects like shadows, reflections, and refractions.

7. CONCLUSION

Recently, it has been shown that it is possible to ray trace complex scenes at interactive frame rates even on a single commodity PC. Currently, however, almost all of these implementations use triangles as the only geometric primitive.

In this paper, we have presented the details of our ray tracing system that is capable of directly rendering trimmed bicubic Bézier surfaces. We have shown that by carefully optimizing the implementation of a well-known intersection technique using SIMD instructions provided by many of today's CPUs, it is feasible to render this kind of free-form surfaces at interactive frame rates as well.

Our results indicate that direct ray tracing of Bézier surfaces is already a valuable alternative to triangle-based approaches, especially for complex models. Moreover, we also suggested a number of possible improvements to further increase the achievable frame rate, which will make this method even more competitive.

8. ACKNOWLEDGEMENTS

We would like to thank all the people that have contributed to this paper, in particular Matthias Biedermann and Thorsten Grosch for many helpful discussions and their comments on preliminary versions, as well as Arne Claus for modeling the Cessna and Chessboard scenes. In addition, we would like to thank the reviewers for their suggestions to improve this paper. Special thanks also go to the Volkswagen AG for providing the data of the Polo model and granting permission to use it in this publication.

9. REFERENCES

[Ben04a] Benthin, C., Wald, I., and Slusallek, P. Interactive Ray Tracing of Free-Form Surfaces. ACM Afrigraph, pp.99-106, 2004.

[Bui75a] Bui-Tuong, P. Illumination for Computer Generated Pictures. Com. of ACM 18, No.6, pp.311-317, 1975.

[Gei03a] Geimer, M., and Müller, S. A Cross-Platform Framework for Interactive Ray Tracing, Proc. of GI Graphiktag, pp.25-34, 2003.

[Gol87a] Goldsmith, J., and Salmon, J. Automatic Creation of Object Hierarchies for Ray Tracing. IEEE CG&A 7, No.5, pp.14-20, 1987.

[Int04a] Intel Corp. IA-32 Architecture Software Developer's Manual. 2004.

[Int04b] Intel Corp. Hyper-Threading Technology. http://www.intel.com/technology/hyperthread/

[Lex01a] Lext, J., and Akenine-Möller, T. Towards Rapid Reconstruction for Animated Ray Tracing. EUROGRAPHICS Short Presentations, pp.311-318, 2001.

[Mar00a] Martin, W., Cohen, E., Fish, R., and Shirley, P. Practical Ray Tracing of Trimmed NURBS Surfaces. JGT 5, No.1, pp.27-52, 2000.

[Mot99a] Motorola, Inc. AltiVec Technology Programming Interface Manual. 1999

[Muu95a] Muuss, M. J. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. Proc. BRL-CAD Symposium, 1995.

[Nis90a] Nishita, T., Sederberg, T.W., and Kakimoto,M. Ray Tracing Trimmed Rational Surface Patches. Computer Graphics 24, No.4, pp.337-345, 1990.

[Par99a] Parker, S., Martin, W., Sloan, P.-P. J., Shirley, P., Smits, B., and Hansen, C. Interactive Ray Tracing. Sym. Interactive 3D Graphics, pp.119-126, 1999.

[Roc89a] Rockwood, A., Heaton, K., and Davis, T. Real-Time Rendering of Trimmed Surfaces. Computer Graphics 23, No.3, pp.107-116, 1989.

[Smi98a] Smits, B. Efficiency Issues for Ray Tracing. JGT 3, No. 2, pp.1-14, 1998.

[Swe86a] Sweeney, M., and Bartels, R. Ray Tracing Free-Form B-Spline Surfaces. IEEE CG&A 6, No.3, pp.41-49, 1986.

[Wal01a] Wald, I., Slusallek, P., Benthin, C., and Wagner, M. Interactive Rendering with Coherent Ray Tracing. Computer Graphics Forum 20, No. 3, pp.153-164, 2001.

[Wal01b] Wald, I., Slusallek, P., and Benthin, C. Interactive Distributed Ray Tracing of Highly Complex Models. Rendering Techniques 2001, pp. 274-285, Springer, 2001.

[Wal03a] Wald, I., Benthin, C., and Slusallek, P. Distributed Interactive Ray Tracing of Dynamic Scenes. IEEE Sym. on Parallel and Large-Data Visualization and Graphics, pp.77-86, 2003.

[Wal04a] Wald, I. Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[Wan01a] Wang, S., Shih, Z, and Chang, R. An Efficient and Stable Ray Tracing Algorithm for Parametric Surfaces. Journal of Information Science and Engineering 18, pp.541-561, 2001.

[Whi80a] Whitted, T. An Improved Illumination Model for Shaded Display. Com.of ACM 23, No.6, pp.343-349, 1980