Efficient Ray Tracing for Bezier and B-Spline Surfaces

by

W. Barth
W. Stürzlinger

Technical University of Vienna
Institut für Computergraphik
Karlsplatz 13/1861
A–1040 Vienna
Austria

Tel.: +43(1)58801-4075
Fax : +43(1)5874932
e-mail: barth@eichow.tuwien.ac.at

Efficient Ray Tracing for Bezier and B-Spline Surfaces

W. Barth, W. Stürzlinger

Institut für Computergraphik, Technical University of Vienna, Austria

Abstract

Generating realistic pictures by raytracing requires intersecting the objects with many rays (1 million or more). With Bezier or B–Spline surfaces as objects the intersections must be calculated by an iterative method. This paper describes an algorithm which performs these calculations efficiently. In a preprocessing step the surface is subdivided adaptively into parts and a tight enclosure is calculated for each part. We selected parallelepipeds (first order approximations) as enclosures, their orientation and the angles between their edges are chosen in such a way that they enclose the respective part as tightly as possible, they are not rectangular in general. A binary tree built with these enclosures allows to test very fast which parts of the surface may be hit by a given ray. The leaves of the tree contain small, almost plane parts of the surface. For each part a linear approximation is calculated, this is a parallelogram, in general not rectangular. For each ray which hits the enclosure the intersection with this approximation is calculated first, yielding an accurate starting point for the following iteration.

Efficient Ray Tracing for Bezier and B-Spline Surfaces

W. Barth, W. Stürzlinger

Institut für Computergraphik, Technical University of Vienna, Austria

1 Introduction

The generation of realistic pictures by raytracing is an expensive method. It is necessary to "shoot" at least one ray for each pixel of the picture. Each "shooting" operation involves intersecting the ray and the displayed scene. Reflections and refractions create additional rays. To avoid aliasing–effects normally more than one ray per pixel has to be "shot". Thus raytracing a picture involves a lot of ray–object–intersections (1 million or more), and efficient routines for their computation are needed.

Because of the large number of rays it is advantageous to use a preprocessing step, which leads to a speed–up in computing the intersections. Even a very time consuming preparation which yields only a tiny saving per intersection may reduce the total time significantly, because preprocessing is executed only once.

Bezier or B–Spline surfaces are polynomials in the parameters u and v. The intersection point of a ray and such a surface has to be calculated iteratively. The Newton–Iteration is well suited if the starting point is sufficiently accurate. Therefore we will adaptively subdivide the surface into parts during the preprocessing step. This subdivision is done by halving the surface, or a part of the surface, parallel to the u– or v–axis. Subdivision will be repeated until all final parts of the surface can be approximated accurately by a plane parallelogram. This linear approximation in general yields non rectangular parallelograms. By intersecting the rays with these approximations we get very good starting points for the following Newton–Iteration.

Many rays totally miss the surface, and these should be detected with minimal computational effort. Similarly the fact that a ray will not hit a specific part has to be discovered very fast. We exploit a property of the Bezier and B–Spline surfaces, namely that such a surface lies completely in the inner of the convex hull of its control points, to construct a tight and simple enclosure for each part of the surface. Then for a lot of parts we can very quickly detect that they cannot be hit because the ray misses their enclosures. The tighter the enclosures, the more parts will be excluded from further computations, the simpler they are, the faster the algorithm will work. We construct the enclosures by expanding the approximating parallelograms in the third "dimension" until all control points are inside the enclosure. This results in parallelepipeds which adapt well to the shape and orientation of the surface parts. For small parts over–estimation is caused only by second order terms.

The algorithm that selects the parts of the surface possibly hit by a ray, i.e. parts whose enclosures are hit, should also be as efficient as possible. When subdividing a surface, in each step we subdivide into two parts by splitting it along a line parallel to the u– or v–axis. All parts are arranged in a binary tree, the nodes of which contain the parallelepipeds enclosing the respective parts. The root encloses the whole surface and the leaves are very small, almost plane parts of the surface.

Picture 1 shows the enlosing parallelepipeds for a goblet [4]. You can see that they adapt well to the surface and that the degree of subdivision depends on the shape of the surface. The enclosures are very thin, consequently they approximate the parts of the surface well. Therefore they are a well suited base for our iteration method which produced Picture 2. Picture. 3 shows the large enclosing parallelepipeds corresponding to a very rough subdivision of the surface from Picture 4. It is easy to see that the angles of the parallelepipeds are chosen in such a way that they approximate the surface part very tightly. In general they are not orthogonal.

To intersect a ray with the surface, we now test whether the ray hits the parallelepiped of the root. If it does, we test both subtrees and so on until we reach the leaves. If the ray misses the enclosure of a subtree we know that it misses all parts of the surface enclosed by it, and we can exclude the subtree from further consideration. With high probability each of the reached leaves selects a part of the surface hit by the ray. Now the intersection can be calculated iteratively.

The entire algorithm consists of two parts:

a) Preprocessing
The surface is divided along a line parallel to the u– or v–axis, the parts are divided again, and so on, until each part can be approximated accurately by a plane parallelogram. For these subdivisions we use the algorithm of de Casteljau respectively de Boor [5], [9] which uses the control points of the surface(–part) to calculate those for the two parts. From the control points of the parts we calculate the enclosing parallelepipeds. A binary tree is built, in which each node contains only data about the parallelepiped, while the leaves contain the u– and v–interval and the approximating parallelogram too.

b) Intersection
Beginning with the root of the tree, i.e. the whole surface, we test whether the ray intersects the enclosing parallelepiped. Then this test is done for succeeding nodes of the tree. If it misses a parallelepiped the appropriate subtree is pruned from further consideration. Finally the search will reach all leaves whose enclosures are hit by the ray. Each leaf contains the approximating parallelogram which is used to calculate the starting point for the iteration depending on the ray. The following iteration is done on the whole surface using its control points; therefore the control points of the surface parts need not be stored.

1.1 Related work

Müller and Hagen [8] describe an algorithm for speeding up raytracing for Bezier and B–Spline surfaces by dividing them into small parts. Some other papers e.g. [3] present an approach preventing failure of the iteration by using interval arithmetic, but this is very time consuming. Hierarchical structures for the parts of the Bezier or B–Spline surface have been used by other authors, e.g. [11], [13]. Sweeney and Bartels [11] use rectangular bounding boxes as enclosures. Through fine subdivision they get tight enclosures and good starting points. Yen et al. [13] construct better boxes by using "oriented slabs", i.e. the enclosures adapt to the orientation of the surface parts, but they are rectangular, too. Other authors, e.g. [6], [12], use bounding boxes as enclosures for parts of the scene in raytracing, but don't refer to Bezier or B–Spline surfaces.

Using accurate approximations for the surface parts and non rectangular parallelepipeds as well fitting enclosures has been described first by Barth [1]. The method uses the parallelogram, that is the first order approximation of the surface part, for calculating a starting point and expands the parallelogram by second order terms in the third dimension to get a simple and tight enclosure. This yields much better starting points for the iteration and very good exclusion criteria for the parts not hit by a ray without increasing the computational effort considerably. Giger–Hofmann uses the method of [1]; in this thesis, however, iteration is based on a "down–hill" method; additionally, coherence is exploited for speeding up the calculations. The main ideas of [1] are repeated here. Additionally we report on new experiences and from these we develop some new variants of the algorithm.

2 Bezier and B–Spline Surfaces

Bezier surfaces as treated in this paper are defined by their control points $P_{i,j}$. The formula for this definition is

$$B(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} b_{j,m}(v) b_{i,n}(u) \quad \text{for } 0 \le u, v \le 1 \tag{2.1}$$

where the blending functions are the Bernsteinpolynomials

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{2.2}$$

Additionally we apply our method to B–Spline surfaces defined by

$$BS_k(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} N_{j,k}(v) N_{i,k}(u) \qquad \begin{array}{l} \text{for } 0 \le u \le n-k+2 \\ \text{and } 0 \le v \le m-k+2 \end{array} \tag{2.3}$$

with the normal blending functions. All details for calculating points of these surfaces, normal vectors and etc. may be looked up in a textbook or monograph, e.g. [5] or [9]. Especially we require methods for dividing a (part of a) surface into two parts by cutting it along a u– or v–line, which is normally performed by the algorithm of de Casteljau respective de Boor.

3 The preprocessing step

In section 1 we mentioned the necessity of a tight enclosure of a part of the surface. Furthermore this enclosure should be a convex solid that is formed as simple as possible. These properties aid in testing quickly whether a ray hits a part of the surface. If we expect a hit, a good initial value for the iteration process should be easy to find too.

3.1 The approximating parallelogram

Let a given Bezier or B–Spline surface be cut into parts that are all nearly plane. Each part belongs to two parameter intervals $\underline{u} < u < \overline{u}$ and $\underline{v} < v < \overline{v}$ (see Fig. 3.1). Its four cornerpoints and the control points are known. The plane parallelogram defined by the two vectors $\vec{v}_1$ and $\vec{v}_2$ is a good approximation (the first term of the Taylor series) for the part of the surface. This holds only if the part is small enough, and contains no singularities, especially no edges.

This method will fail if the vectors $\vec{v}_1$ and $\vec{v}_2$ are linearly dependent or at least one of them is zero, in which case other vectors $\vec{v}_1$,$\vec{v}_2$ have to be used, for instance the axes of the coordinate–system.

This Taylor approximation favours one of the four vertices of the surface part. It is better to avoid the unsymmetry and to use the mean value of the two vectors from $E1$ to $E2$ and from $E3$ to $E4$ instead of $\vec{v}_1$ and to calculate a similar replacement for $\vec{v}_2$. By moving the parallelogram spanned by $\vec{v}_1$ and $\vec{v}_2$ in an orthogonal direction so that the "highest" and "lowest" control points of the surface part have the same distance from it – on different sides – a very good approximation is constructed. We found that the last method saves almost one iteration step per ray.

The intersection of the ray with the parallelogram gives an appropriate starting point for the iteration, the values of u and v are calculated by linear interpolation (see section 4.3).

3.2 The enclosing parallelepiped

After calculating the parallelogram according to the method of the previous section we expand it to an oblique angled parallelepiped. A third vector $\vec{v}_3$ is used to generate a parallelepiped spanned by $\vec{v}_1$, $\vec{v}_2$ and $\vec{v}_3$. Then the three major edges are elongated until all control points are enclosed (see Picture 5). The parallelepiped is defined by

$$E + \beta_1 \vec{v}_1 + \beta_2 \vec{v}_2 + \beta_3 \vec{v}_3 \tag{3.1}$$

where E is one vertex of the approximating parallelogram. The coefficients β_1,β_2,β_3 are intervals. For all control points we calculate the coefficients of a representation according to equation (3.1). If all these coefficients are contained in β_1,β_2,β_3 then all control points are enclosed by the parallelepiped. We start the calculation of the intervals with $[0,\,1]$ for β_1 and β_2 and with $[0,0]$ for β_3 – this corresponds to the approximating parallelogram – calculate the coefficients for the remaining control points and extend the β–intervals if necessary. This provides an expansion of the parallelepiped in the plane of the parallelogram as well as in the third direction. Picture 5 shows the control points for a part of the surface, the part itself, the approximating parallelogram and the enclosing parallelepiped.

3.3 The tree of the surface parts

When subdividing the given Bezier or B–Spline surface, we cut it into two parts of approximately equal size in each step. These parts are arranged in a binary tree. In each node we store the information about the corresponding part, which will be used later for the intersection test or for calculation of a starting point for the iteration. The essential information stored in a node is

1. Internal node: The enclosing parallelepiped.
2. Leaf: The enclosing parallelepiped, the parameter domain $\underline{u}$, $\overline{u}$ and $\underline{v}$, $\overline{v}$, the approximating parallelogram

It is not necessary to store the control points of the part of the surface, because they are not required for the calculation of the ray–surface–intersections performed later: The intersection test and the calculation of a starting point are performed with the parallelogram and the oblique angled parallelepiped. The iteration process described in the next section uses the control points of the original surface only. Because it yields the same results regardless whether it is performed on the whole surface or on a part, the iteration is always executed on the whole.

3.4 Subdivision strategy

We subdivide the current part of the surface by halving either in the u– or in the v–direction. The algorithms of de Casteljau and de Boor allow us to calculate the control points of the two new parts from those of the larger part. The new control points are required for further subdivision during our preprocessing step. But they are not stored in the tree as mentioned in section 3.3.

Subdivision stops when the resulting part is approximated well enough by a plane parallelogram of section 3.1. This can be seen from its control points. Both of the following conditions must hold for each part that will not be divided further.

1. all control points are close to the plane of the approximating parallelogram, i.e. the enclosing parallelepiped is thin.
2. the four corner points E_1, E_2, E_3 and E_4 approximately form a parallelogram, i.e. the lengths of the opposite sides do not differ much.

The decision whether to halve along the u– or the v– axis is based on the curvature of the u– and v–lines. The control points for a given parameter direction show how much these lines deviate from a straight line. To get an estimate for the curvature in the v–direction we take three points $P_{i,j}$ with fixed i and ascending j, calculate the chord of the two outer points and the distance of the third point from it. Then we take the maximum of these chord–distances for all triples. The same is done for the u–direction. Now we divide along a v–line if the measure for the curvature in u–direction is the bigger one.

Also the subdivision along a v–line should be favoured if the distance between E_1 and E_3 differs much from the distance of E_2 and E_4 (see Fig. 3.1), and this difference is bigger than that for the other two sides of the parallelogram. This causes a split of a ring–shaped surface into two sectors, and this split is carried out even if the surface is almost plane.

4 Calculation of the ray–surface–intersections

After performing the preparations described in the previous sections the raytracing process is started and the following steps will be performed for each ray: The tree of the surface parts is searched for parts which may be hit by the ray, a starting point is calculated for each part hit, and the iteration is started. Because of the preparations the calculations can be performed very quickly.

4.1 Searching the tree of surface parts

First, we search for all parts of the surface which may contain an intersection with the ray. We start at the root of the tree (the whole surface) and test whether the ray hits the enclosing parallelepiped. If it does we continue with both of the successors and carry out the same test. We go on until we reach those leaves of the tree whose parallelepipeds are intersected by the ray. These leaves correspond to small, almost plane parts of the surface. Because the parallelepipeds enclose the surface tightly, we often find a node which is not hit by the ray, and therefore we can prune the corresponding subtree from further consideration (all direct and indirect succesors of this node belong to parts of the surface that are enclosed by this parallelepiped).

As we use enclosures we are sure that for each ray all parts of the surface which may contain a hit will be found. For these parts the iterative process which will be discussed in section 4.4 is started.

4.2 Intersection with the parallelepiped

The test whether a ray hits an epiped is simple. The ray is given by

$$A + t\vec{r} \tag{4.1}$$

where A is the origin and r the direction vector. We define $\vec{p}$ as the vector from A to E_1 and $\vec{w}_k$ as the normal vector on the faces of the epiped, then we use (4.1) and (3.2), and after some transformations we get

$$t_k = \frac{(\vec{p} \cdot \vec{w}_k) + \beta_k (\vec{v_k} \cdot \vec{w}_k)}{(\vec{r} \cdot \vec{w}_k)} \quad \text{with} \quad k = 1, 2, 3 \tag{4.2}$$

Taking the boundaries of the interval β_k calculated in the preparation step we obtain from (4.2) the boundaries of an interval for t_k, which determine the intersection of the ray with the two planes containing opposite faces of the parallelepiped in the k–th dimension. Therefore the interval t_k gives the "time" the ray "spends" between these two planes. The intersection of the three intervals t_k

$$[t] = [t_1] \cap [t_2] \cap [t_3] \tag{4.3}$$

states the "time"–interval for the ray being inside the parallelepiped. Especially if $[t]$ is empty the ray misses the epiped completely. If the denominator in (4.2) is zero the ray is parallel to the two corresponding faces. It lies either between the two planes or it is outside. Instead of developing (complicated) rules for distinguishing between these two cases we calculate the intersection of a nearly parallel ray by taking a very small value for the denominator. Then we get a very large intervall t_k containing the interesting domain of t completely if the ray is between the two planes, in the other case t_k is far away and (4.3) will yield an empty interval.

In raytracing all primary rays come from the same origin A, the eye point, but they have different directions $\vec{r}$. All these rays have to be tested for intersection with the same tree of parallelepipeds and therefore very often the same epiped is tested against many rays with the same $\vec{p}$. In (4.2) only the denominator is dependent on $\vec{r}$. All other values can be precalculated for fixed $\vec{p}$ and they can be stored in the epiped tree. During raytracing, for each primary ray only 3 dotproducts for the denominator, 6 divisions for the t_k–intervals and the comparisons for (4.3) are required to calculate the intersection completely. Only for rays with a different origin the first dotproduct in the numerator of (4.2) has to be calculated additionally.

These intersection tests should be optimized very carefully, because they heavily affect the efficiency of the method, as they have to be done for each pixel (when oversampling even more often) and for each node of the tree which is reached. In most cases they yield an empty interval: For all rays far from the surface, in all branches of the tree ending at an intermediate node. Therefore intersection tests are executed much more frequently than the calculation of a starting point and the iteration.

4.3 Starting point for the iteration

As an initial point for the iteration we use the intersection point between the parallelogram from section 3.1 and the ray. The t parameter for this point is found by setting $k = 3$ and $\beta_3 = 0$ in (4.2)

$$t_0 = \frac{(\vec{p} \cdot \vec{w}_3)}{(\vec{r} \cdot \vec{w}_3)}. \tag{4.4}$$

In section 4.2 $\vec{w}_3$ has been defined as the normal vector to the approximating parallelogram, therefore it is an approximation for the normal to the surface. The point itself is found by

$$D_0 = A + t_0 \cdot \vec{r}. \tag{4.5}$$

To get the first approximate values u_0, v_0 for the intersection point we calculate the coefficients β_1 and β_2 for D_0 and get

$$u_0 = \underline{u} + \beta_1 \cdot (\overline{u} - \underline{u}) \tag{4.6}$$

$$\text{and} \quad v_0 = \underline{v} + \beta_2 \cdot (\overline{v} - \underline{v}) \tag{4.7}$$

Some problems arise if the denominator of (4.4) is zero or very small, i.e. the ray $\vec{r}$ is almost orthogonal to $\vec{w}_3$. The geometric interpretation is that the ray is nearly tangential to the part of the surface (see Fig 4.1).

This critical case can be recognized, because it appears only when the ray passes through a very thin box approximately parallel to the larger face, we take as a criterion

$$\tan \alpha > \frac{b}{h} \tag{4.8}$$

Then it is very likely that more than one intersection point exists. This situation causes convergence to be very slow. But there is yet another problem, we may find the wrong intersection. For raytracing this proves to be fatal! The algorithm computes the color as if the ray came from the wrong side of the surface. And this color is usually completely different, as the one side is usually in shadow if the other one is lighted. We avoid such errors by using the entrance point of the ray into the u,v–domain as the initial value. Now the iteration is more likely to converge to the first intersection. But because of efficiency this additional iteration is only done when (4.8) holds, therefore in all normal cases we do not affect the speed of convergence.

4.4 The iteration

The calculation of an accurate value for the intersection point is performed by the Newton method, which generally converges very quickly.

In section 4.3 we calculated the initial values u_0, v_0 from the parallelogram which approximates a part of the surface. The iteration process itself works on the whole surface and with the original control points. We calculate the corresponding point P_0 with the partial derivatives P_{0u} and P_{0v} and the normal n_0. The ray is then intersected with the plane tangential to the point P_0 corresponding to u_0,v_0 and we get a new approximation u_1,v_1.

We stop the iteration process if two succeeding points differ less than a given ϵ. This ϵ can be calculated either as the maximum of the differences between the u and v values of two approximations or from the x,y and z values.

Finally we have to cope with another difficulty: termination of the iteration without success. We start the iteration for each ray which hits the enclosure of a part. But some of these rays miss the corresponding part of the surface. In these cases the iteration cannot converge. Consequently, if the convergence criteria are not met after a preset maximum number of iteration steps (e.g. 3) the algorithm assumes that the ray misses the part of the surface and returns the result "no hit". Another indication is that the iteration leaves the u,v–domain of the surface part. Then the intersection is probably inside the domain of a neighbour part. Because the iteration is likely to approach the solution from one side we go back to the border of the domain after leaving it for the first time. If the next

iteration step again leads to a point outside the domain we return the result "no hit". These two heuristic rules identify almost all rays which have no intersection point inside the actual part. It is also very unlikely that they eliminate (by mistake) a ray which does intersect.

In order to avoid losing an intersection we start a second iteration process after an unsuccessfull iteration, using the entrance point into the epiped as the initial value. This happens only for a small fraction of all rays and therefore does not significantly slow down the whole algorithm. If a third iteration is carried out after an unsuccessful second one, using the exit point of the ray as initial value, it is almost impossible to miss an intersection.

5 Implementation and results

The algorithm described in this paper has been implemented, and it has been integrated in the RISS–System [2] which is a raytracing system developed at the Institute for Computer Graphics at the Technical University of Vienna.

Tests showed the predicted behaviour, namely quick convergence of the iteration and a very small number of failures if the given surface has been subdivided into sufficiently small parts. We even observed some effects which were counterintuitive at first glance – e.g. the total computation time may decrease as the number of subdivisions increases. This surprising behaviour results from the fact that finer subdivision yields more accurate starting values for the "costly" iteration and therefore less iterations are required. These savings are a multiple of the increased effort for search in the epiped tree. As expected the additional time for the preprocessing step is minimal. Picture 6 [7] shows the tea-pot as generated by RISS [2]. For anti–aliasing we used the method of "Adaptive Stochastic Sampling" [10]. The surface has been divided into 1397 parts. Approximately half of the rays required only one iteration, this is always neccesary to check the accuracy. For nearly all other rays a second iteration was sufficient to get an accurate intersection. The preprocessing step took 22 seconds, while the actual rendering for 1000 by 750 pixels took about 4 hours on a VAXStation 3100/38.

Even more important is the fact that the number of failures drops with the number of subdivisions. If the parts of the surface are too big, the following will happen at the outline of the surface: Based on slow convergence, further computation is considered useless and the iteration is – falsely – stopped. The algorithm erroneously assumes the surface is not hit, although a parallelepiped has been hit. It consequently assigns the color of the background to the pixel. Therefore the outline may have some notches. Sometimes we even found such "gaps" in areas of the surface where the curvature changed its sign. The iteration also may converge to the rear of the two intersection points, as was discussed in section 4. As an example, if picture 6 is done without anti–aliasing only four wrong pixels are calculated. But they are not easily noticeable, and in the anti–aliased picture all wrong values are mixed with correct ones, and therefore invisible.

As we can get only an approximate value for the intersection point (and not the exact one), we also have to consider the situation where the approximation point lies "behind" the surface. Then the test whether a light source contributes to the illumination shows that the surface shades itself. This problem can be avoided by placing the intersection point a little closer to the origin of the ray, the magnitude of the correction has to be larger than the error of the last iterated point.

Picture 7 shows some of the failures that can occur when no attention is paid to the directives discussed earlier: For this picture we used a too rough subdivision (Picture 3), therefore the algorithm failed to converge after the preset number of steps at parts of the surface where the parallelograms don't approximate the surface well. Additionally the test for almost tangential rays (section 4.3) was disabled. Consequently artifacts appear near the outline of the surface. The correct picture generated with a finer subdivision and careful iteration is shown in Picture 4.

Literatur

[1] W. Barth: *Effizientes Ray–Tracing für Bezier– und B–Spline Flächen.* In Encarnacao, Hoschek, Rix: Geometrische Verfahren der Graphischen Datenverarbeitung, ZGDV Beiträge zur Graphischen Datenverarbeitung, Springer–Verlag 1990

[2] M. Gervautz, W. Purgathofer: *RISS — Ein Entwicklungs–System zur Generierung realistischer Bilder.* In W. Barth (Hrsg.): Visualisierungstechniken und Algorithmen, Informatik–Fachberichte Nr. 182, Springer–Verlag, 1988

[3] C. Giger: *Ray Tracing Polynomial Tensor Product Surfaces.* In Hansmann, Hopgood, Strasser (Hrsg.): Proc. Eurographics 1989, North–Holland, S. 125 – 136

[4] R. Groß: *Ray–Tracing für Bezier–Flächen.* Diplomarbeit, Techn. Univ. Wien, 1991

[5] J. Hoschek, D. Lasser: *Grundlagen der geometrischen Datenverarbeitung.* Teubner Stuttgart, 1989

[6] T. Kay, J. Kajiya: *Ray Tracing Complex Scenes*, Computer Graphics, Proceedings SIGGRAPH August 1986.

[7] W. Kiendl: *Ray–Tracing für Bezier–Flächen.* Diplomarbeit, Techn. Univ. Wien, 1991

[8] H. Müller, H. Hagen: *Beschleunigung der Bilderzeugung für Freiformflächen durch Speichereinsatz.* In Clauer, Purgathofer (Hrsg.): Proc. Austrographics 1986, Oldenbourg–Verlag, Wien, München, 1986

[9] T. Pavlidis: *Algorithms for Graphics and Image Processing.* Springer–Verlag, Heidelberg, 1982

[10] W. Purgathofer: *A Statistical Method for Adaptive Stochastic Sampling.* In Requicha (Hrsg.): Proc. Eurographics 1986, S. 145 – 152, und Computers and Graphics, Vol. 11, S. 157 –162, 1987

[11] M. Sweeney, R. Bartels: *Ray Tracing Free–Form B–Spline Surfaces*, IEEE Computer Graphics and Applications, February 1986.

[12] H. Weghorst, G. Hooper, D. Greenberg: *Improved Computational Methods for Ray Tracing*, ACM Transactions on Graphics, January 1984.

[13] J. Yen, S. Spach, M. Smith, R. Pulleyblank: *Parallel Boxing in B–Spline Intersection*, IEEE Computer Graphics and Applications, January 1991.

[14] C. Giger–Hofmann: *Ein Ray–Tracing–Verfahren zur Visualisierung polynomialer Tensorproduktflächen*, Dissertation, Technische Hochschule Darmstadt (Germany), June 1992.