Max-Planck-Institut für Informatik
Computer Graphics Group
Saarbrücken, Germany

Efficient Ray Tracing of Trimmed NURBS Surfaces

Master Thesis in Computer Science

Computer Science Department
University of Saarland

Alexander Efremov

Supervisors: Dr. Vlastimil Havran
Prof. Dr. Hans-Peter Seidel

Max-Planck-Institut für Informatik
Computer Graphics Group
Saarbrücken, Germany

Begin: September 01, 2004
End: December 31, 2004

Statement under Oath

Hereby I declare on oath that I have drafted the following thesis independently, that I have used the stated sources and means only, and that I have not presented this work at a different examination office previously.

Saarbrücken, January 19, 2010 Alexander Efremov

Abstract

The goal of this master thesis is to study, compare, and improve existing approaches for direct ray tracing trimmed NURBS surfaces. In order to be able to study the topic efficiently, a library for NURBS evaluation has been implemented. The library is supposed to be easily integrated in any ray tracing application to make it support direct ray tracing trimmed NURBS surfaces.

Acknowledgement

Its my pleasure to thank Prof. Hans-Peter Seidel for providing me an opportunity to work in this interesting filed of research. I thank him for the excellent working conditions at the MPI in computer graphics group.

I would like also to thank Prof. Wolfgang Paul who invited me to study computer science at the University of Saarland.

This work would not have been possible, had I not got valuable and timely contributions from my advisor, Vlastimil Havran. I thank him for giving me total freedom to think and at the same time always chipped in with useful suggestions and critics.

A special thanks to DaimlerChrysler AG for providing the real-world data used in car production.

Last but not the least, I thank my parents for making it all happen.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Ray tracing has become a popular method for generating realistic images. In order to produce a realistic image, a rendering program must simulate the physical system that is involved in making a direct photograph of a real scene. Many realistic effects such as penumbrae, motion blur, fog, gloss, translucency, and depth of filed can be obtained using standard recursive ray tracing. Even more realistic effects, such as caustics or indirect illumination, can be simulated using global illumination algorithms for ray tracing.

NURBS surfaces are common way to represent surface meshes in computer aided design. NURBS surfaces have useful geometric properties, which make modeling easy and effective. Trimming curves are a common method for overcoming the topologically rectangular limitations of NURBS surfaces. Trimming curves are defined in the parameter domain of NURBS surfaces and specify the regions of the surface which must be cut away. Almost all possible surface shapes can be modeled using trimmed NURBS surface representation.

Although ray tracing applications generate realistic images, and trimmed NURBS surface representation is common for computer aided design, they are not usually combined together. NURBS surfaces are not used as basic objects in ray tracing applications, because finding ray-NURBS surface intersection is complex problem. Therefore, each NURBS surface is tessellated into triangles before starting the rendering process. Unfortunately, during the tessellation process some information about the surface shape may be lost even with the best tessellation techniques, especially in the case of surfaces with high curvature variance over the surface. Another drawback is the tremendous number of triangles which must be stored in the case of complex scenes. The direct ray tracing NURBS surfaces may have worser time performance (because the intersection routine of a ray with a NURBS surface is much more complicated than the one with a triangle), but the quality of the result image overcomes the quality of the image obtained after ray tracing of tessellated scene models.

1.2 Overview

The Master Thesis consists of six chapters, including the introduction and the conclusion parts. Chapter 1 gives an introducation of direct ray tracing trimmed NURBS surfaces and presents an overview of the Master Thesis.

In Chapter 2 general remarks on curves and surfaces are done. The place of NURBS and Bézier surfaces besides all other subsets of surfaces are determined, and many usefull mathematical and geometrical properties are discussed.

In Chapter 3 the basics of ray tracing techniques are briefly explained. It is also shown how different acceleration data structures can improve the performance of ray tracing.

In Chapter 4 the history of ray tracing parametric surfaces is described. Two common methods for direct ray tracing Bézier surfaces are explained, and some improvements of these methods are proposed.

In Chapter 5 practical aspects of ray tracing trimmed NURBS surfaces are discussed in more details. The way of representing a NURBS surface by the number of Bézier patches is explained. An efficient acceleration data structure for Bézier patches is described, and an efficient scheme of combining it together with methods of ray tracing Bézier patches is given. An efficient way of performing trimming test and some suggestions about numerical robustness are proposed. The chapter is concluded by the section of practically achieved results.

Chapter 6 completes the Master Thesis by conclusions about the done work with the improvements and modifications of the existing techniques.

Appendix A shows the compatibility of the implemented library for NURBS evaluation with VRML97 and X3D specifications. Appendix B presents the framework of integration the library into a ray tracing system.

All algorithm frameworks presented in the Master Thesis are in the *ANSI C++* [39] standard.

Chapter 2

Introduction to Curves and Surfaces

2.1 Introduction

One of the main goals of geometric modeling is to represent objects of real world by mathematical equations in order to operate with these objects in computer. Mathematical representation of an object is not unique, and different approaches are used in order to represent the same object. We want to have the representation, which does not require a lot of memory to store information about the object, and gives us efficient and fast algorithms for different object related computations. The next sections explain how curves and surfaces can be described mathematically in order to meet these requirements.

2.2 Fundamental Curves and Surfaces Representation

2.2.1 Introduction

Curves are often defined as *the locus of a point moving with one degree of freedom* [28]. There is another definition which describes a curve as *a one-dimensional connected point set in a two-dimensional plane or in three-dimensional space*. Surfaces are defined as a *locus of a point moving with two degrees of freedom* [28]. Another definition describes a surface as *a two-dimensional point set in three-dimensional space*. There are three ways to describe curves and surfaces for geometric modeling. These are *explicit equation*, *implicit equation*, and *parametric equation*. Each method has it's advantages and disadvantages depending on the goal of representation.

2.2.2 Explicit Equation

The explicit equation of a curve lying in the xy plane has the general form

$$y = f(x) \tag{2.1}$$

where just one value of y corresponds to each value of x. This representation is useful in many applications, but it cannot adequately represent multiple-valued and closed curves, and cannot be used where a constraint involves an infinite derivative. Moreover, this form of the curve representation is axis dependent. Notice that explicit equation **cannot** describe a curve in 3D space.

The explicit equation of a surface takes the general form

$$z = f(x, y) \tag{2.2}$$

where just one value of z corresponds to each couple of values (x, y). The explicit surface representation has all disadvantages of explicit curve representation. Hence, explicit equations in general are little used in computer graphics or computer aided design.

2.2.3 Implicit Equation

The implicit equation of a curve lying in the xy plane has the general form

$$f(x, y) = 0 \tag{2.3}$$

and describes an implicit relationship between the x and y coordinates of the points lying on the curve. Using implicit representation of a curve it is easy to determine if a given point is on the curve ($f(x, y) = 0$), or on one side of it ($f(x, y) < 0$), or the other ($f(x, y) > 0$). Notice that implicit equation as well as explicit one **cannot** describe a curve in 3D space. An example of implicitly represented curve is the circle of a unit radius centered at the origin, specified by the equation $f(x, y) = x^2 + y^2 - 1 = 0$.

The implicit equation of a surface takes the general form

$$f(x, y, z) = 0 \tag{2.4}$$

By analogy to curves, using implicit representation of a surface it is easy to determine if a given point is on the surface ($f(x, y, z) = 0$), or on one side of it ($f(x, y, z) < 0$), or the other ($f(x, y, z) > 0$). An example of implicitly represented surface is the sphere of a unit radius centered at the origin, specified by the equation $f(x, y, z) = x^2+y^2+z^2-1 = 0$.

Implicit equation is capable of representing multiple-valued functions but is still axis dependent. However, it has a variety of uses in computer graphics and computer aided design. More information about implicit representation can be found in [8].

2.2.4 Parametric Equation

Most modeling applications require that the choice of a coordinate system should not affect the shape of a curve. For this reason the preferred way to represent shapes in geometric modeling is with parametric equations. In parametric form, each of the coordinates of a point on the curve is represented separately as an explicit function of an independent parameter. The parametric equation of a curve lying in the xy plane has the general form

$$x = x(t); \;\; y = y(t) \qquad a \leq t \leq b \tag{2.5}$$

It is instructive to think of parametrically defined function $C(t) = (x(t), y(t))$ as the path traced out by a particle as a function of time; t in this case is the time variable from the time interval $[a, b]$.

Parametric equations avoid many of the problems associated with non parametric functions. They are axis independent, easily represent multiple-valued functions and infinite derivatives, and have additional degrees of freedom compared to either explicit or implicit formulations. The first quadrant of circle of a unit radius centered at the origin can be defined by the parametric functions

$$\begin{aligned} x(t) &= \cos t \\ y(t) &= \sin t \qquad 0 \leq t \leq \pi/2 \end{aligned} \tag{2.6}$$

The parametric representation of a curve **is not unique**. For instance, setting $u = tan(t/2)$, one can derive alternate representation of the same quadrant of the circle, expressed as follows:

$$\begin{aligned} x(u) &= \frac{1-u^2}{1+u^2} \\ y(u) &= \frac{2u}{1+u^2} \qquad 0 \leq u \leq 1 \end{aligned} \tag{2.7}$$

The first and second derivatives of $C(t)$ are the *velocity* and *acceleration* of the particle, respectively. If the magnitude of the velocity vector, $C'(t)$ is a constant on the interval $[a, b]$ then it is referred to as a *uniform parameterization.* It means that the direction of the particle is changing with time, but its speed is constant.

Opposite to explicit and implicit representations, parametric equations are **capable** to represent a curve in 3D using the general form

$$x = x(t); \;\; y = y(t); \;\; z = z(t) \qquad a \leq t \leq b \tag{2.8}$$

Specifying a surface parametrically requires two parameters. So, the parametric representation of a surface has the general form

$$x = x(u, v); \;\; y = y(u, v); \;\; z = z(u, v) \qquad a \leq u \leq b, \;\; c \leq v \leq d \tag{2.9}$$

Sphere of unit radius centered at the origin can be defined by parametric equation (not unique) $S(u, v) = (x(u, v), y(u, v), z(u, v))$, where

$$\begin{aligned} x(u, v) &= \sin u \cos v \\ y(u, v) &= \sin u \sin v \\ z(u, v) &= \cos u \qquad 0 \leq u \leq \pi, \;\; 0 \leq v \leq 2\pi \end{aligned} \tag{2.10}$$

If one of the parameter values is held constant while the other is varied, an isoparametric curve is formed on the surface. In the case of sphere, holding u fixed and varying v generates the latitudinal lines; holding v fixed and varying u generates the longitudinal lines.

The normal at a point on a surface is given by the cross product of the partial derivatives at the point. Denote the partial derivatives of $S(u, v)$ by $S_u(u, v) = (x_u(u, v), y_u(u, v), z_u(u, v))$ and $S_v(u, v) = (x_v(u, v), y_v(u, v), z_v(u, v))$. Notice, that these vectors are the velocities along latitudinal and longitudinal lines. If the vector cross product $S_u \times S_v$ does not vanish, the unit normal vector N is given by the following equation:

$$N = \frac{S_u \times S_v}{|S_u \times S_v|} \tag{2.11}$$

The normal vector is a property of the surface independent of the actual form of the parameterization of the surface. Under parameterization of a sphere given by Equation (2.10), S_v vanishes at the north and south poles of the sphere, i.e., $S_v(0, v) = S_v(\pi, v) = 0$ for all v, $0 \leq v \leq 2\pi$. Clearly, normal vectors do exist at the poles, but under this particular parameterization they cannot be computed by Equation (2.11).

2.3 Polynomial Curves

2.3.1 Introduction

In Section 2.2 it has been shown that representing of curves parametrically has a number of advantages with respect to explicit and implicit representations. Choosing coordinate functions $x(t)$, $y(t)$, and $z(t)$ arbitrary one can obtain a great variety of parametrically defined curves. But for geometric modeling purposes not all of these functions are in the area of interest. The best choice would be to restrict ourselves to a class of functions which are, first of all, easily, efficiently and accurately processed in a computer. More precisely, we need functions which do not require a lot of memory for storage while giving us a possibility for efficient computation of points and derivatives on the curves. A widely used class of functions with these properties is the polynomials.

A d-dimensional *polynomial curve* of degree n is a parameterized curve $u \mapsto P(u)$ of the form

$$P(u) = \sum_{i=0}^{n} a_i \cdot u^i = a_n u^n + \cdots + a_1 u, \qquad a_n, a_{n-1}, \ldots, a_1, a_0 \in R^d \tag{2.12}$$

Denote P as the set of all polynomials and P_n as the set of all polynomials of degree less or equal to n: $P_n = \{p \in P : degree(p) \leq n\}$. The set of all d-dimensional polynomial curves of degree n is denoted by P_n^d. Functions u^i in Equation (2.12) form a basis for polynomial curves which is called *monomial basis* (or *power basis*). As it was mentioned in Section 2.2.4, the parametric representation of a curve is not unique. The general form of the parametric representation of a curve is

$$P(u) = \sum_{i=0}^{n} p_i \cdot X_{i,n}(u), \qquad p_i \in R^d \tag{2.13}$$

Functions $X_{i,n}(u)$ in Equation (2.13) are called *basis functions*. Coefficients $p_i \in R^d$ are used to control the shape of the curve. Special case $X_{i,n}(u) = u^i$ means that *monomial basis* is used to represent a curve. In this case curve is called *power basis curve*. Efficient point evaluation on a power basis curve is possible using *Horner's method*.

Representation of a curve in the power-basis form has the number of disadvantages. First of all, this representation is not convenient for interactive shape design, because coefficients $\{a_i\}$ convey very little geometric insight about the shape of the curve. Moreover, Horner's method is prone to round-off error if the coefficients vary greatly in magnitude. Therefore, the power-basis representation is rarely used in computer aided design. The number of basis functions have been invented in order to achieve better properties. In the next sections *Lagrange polynomials*, *Bernstein polynomials*, and *B-spline basis functions* are discussed.

2.3.2 Lagrange Polynomials

For a given $n{+}1$ parameter values $t_0, \ldots, t_n \in R$ with $t_0 < t_1 < \cdots < t_n$, the polynomial

$$L_i^n(u) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{u - t_j}{t_i - t_j} = \frac{(u - t_0)(u - t_1)\cdots(u - t_n)}{(t_i - t_0)(t_i - t_1)\cdots(t_i - t_n)}, \qquad i = 0, \ldots, n \tag{2.14}$$

is called the i^{th} Lagrange[1] polynomial of degree n. The Lagrange polynomials $L_0^n, L_1^n, \ldots, L_n^n$ form a basis of P_n. Curves in the Lagrange polynomial basis can be defined as follows:

$$P(u) = \sum_{i=0}^{n} L_i^n(u) \cdot p_i, \qquad p_i \in R^d \tag{2.15}$$

The nice property of this representation is *interpolation* of points p_i at parameter values t_i. This property comes from the formula:

$$L_i^n(t_j) = \begin{cases} 1 & \text{for } i = j, \\ 0 & \text{for } i \neq j. \end{cases} \Rightarrow P(t_j) = p_j, \qquad j = 0, \cdots, n \tag{2.16}$$

Notice, that there is only one polynomial of n^{th} degree, that interpolates these $n+1$ control points, therefore the Lagrange representation is unique.

[1] Joseph Louis de Lagrange (1736-1813)

Figure 2.1 shows the quadratic Lagrange polynomials with respect to parameter values $t_0 = 0,\ t_1 = \frac{1}{2}$ and $t_2 = 1$, which have the following equations:

$$\begin{aligned} L_0^2(u) &= \frac{(u-\frac{1}{2})(u-1)}{(0-\frac{1}{2})(0-1)} = 2(u-\frac{1}{2})(u-1) \\ L_1^2(u) &= \frac{(u-0)(u-1)}{(\frac{1}{2}-0)(\frac{1}{2}-1)} = -4u(u-1) \\ L_2^2(u) &= \frac{(u-0)(u-\frac{1}{2})}{(1-0)(1-\frac{1}{2})} = 2u(u-\frac{1}{2}) \end{aligned} \tag{2.17}$$

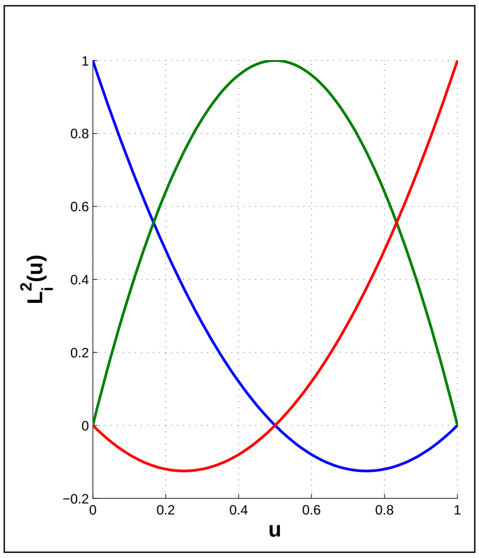


Figure 2.1: Quadratic Lagrange polynomials with respect to parameter values $t_0 = 0,\ t_1 = \frac{1}{2}$ and $t_2 = 1$.

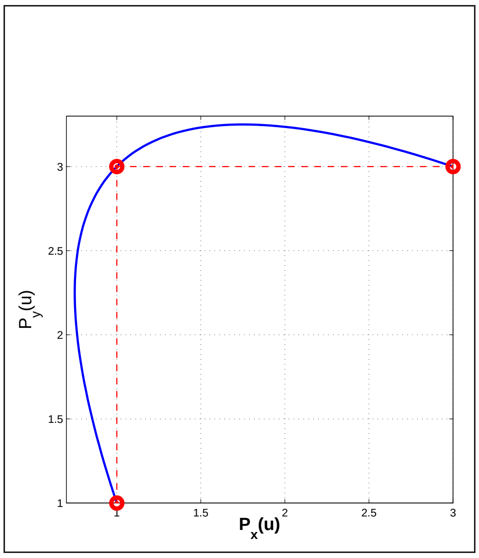


Figure 2.2: Curve defined in quadratic Lagrange polynomial basis using control points $p_1 = (1,1),\ p_2 = (1,3),\ p_3 = (3,3)$.

Figure 2.2 shows a curve defined in quadratic Lagrange polynomials basis using control points $p_1 = (1,1),\ p_2 = (1,3),\ p_3 = (3,3)$ and parameter values $t_0 = 0,\ t_1 = \frac{1}{2}$, and $t_2 = 1$. One can easily see the interpolation property of such curve - it goes through all its control points.

Unfortunately, the interpolation property is the only one useful property of curves defined in Lagrange polynomial basis. Neither control points nor parameter values give any other useful information about the shape of the curve. Moreover, point evaluation and derivative computation on such curves is very time consuming. Therefore, Lagrange polynomials are rarely used in computer aided design for curves and surfaces representation.

2.3.3 Bernstein Polynomials & Bézier Curves

Bernstein polynomials[2] (or *Bézier basis functions*) are the set of polynomials defined as follows:

$$B_i^n(u) = \binom{n}{i}\left(\frac{u-a}{b-a}\right)^i\left(\frac{b-u}{b-a}\right)^{n-i} = \frac{n!}{i!(n-i)!}\left(\frac{u-a}{b-a}\right)^i\left(\frac{b-u}{b-a}\right)^{n-i}, \qquad u \in [a,b] \tag{2.18}$$

Bernstein polynomials $B_0^n, B_1^n, \ldots, B_n^n$ form a basis of P_n. Curves represented in the Bernstein polynomials basis are called *Bézier curves* and can be defined as follows:

$$P(u) = \sum_{i=0}^{n} B_i^n(u) \cdot b_i, \qquad b_i \in R^d \tag{2.19}$$

Bézier basis functions have the number of nice properties:

1. nonnegativity: $B_i^n \geq 0$ for all i, n and $a \leq u \leq b$;
2. partition of unity: $\sum_{i=0}^{n} B_i^n(u) = 1$ for all $a \leq u \leq b$;
3. $B_0^n(a) = B_n^n(b) = 1$;
4. $B_i^n(u)$ attains exactly one maximum on the interval $[a,b]$, that is, at $u = a + \frac{i}{n}(b-a)$;
5. symmetry: for any n, the set of polynomials $\{B_i^n(u)\}$ is symmetric with respect to $u = a + \frac{1}{2}(b-a)$;
6. recursive definition: $B_i^n(u) = \left(\frac{b-u}{b-a}\right)B_i^{n-1}(u) + \left(\frac{u-a}{b-a}\right)B_{i-1}^{n-1}(u)$; (with $B_i^n(u) \equiv 0$ if $i < 0$ or $i > n$);
7. derivatives: $B_i^{n\prime}(u) = \frac{dB_i^n(u)}{du} = n(B_{i-1}^{n-1}(u) - B_i^{n-1}(u))$; (with $B_{-1}^{n-1}(u) \equiv B_n^{n-1}(u) \equiv 0$);

Later in this section for simplicity we assume that $a = 0,\ \ b = 1$, i.e., $u \in [0,1]$. Figure 2.3 shows the quadratic Bernstein polynomials, which have the following equations:

$$\begin{aligned} B_0^2(u) &= \binom{2}{0} u^0 (1-u)^{2-0} = (1-u)^2 \\ B_1^2(u) &= \binom{2}{1} u^1 (1-u)2-1 = 2u(1-u) \\ B_2^2(u) &= \binom{2}{2} u^2 (1-u)^{2-2} = u^2 \end{aligned} \tag{2.20}$$

Figure 2.4 shows a curve defined in quadratic Bernstein polynomial basis using control points $p_1 = (1,1),\ \ p_2 = (1,3),\ \ p_3 = (3,3)$. Notice, that there is no general

[2] Sergeî N. Bernstein (1880-1968)

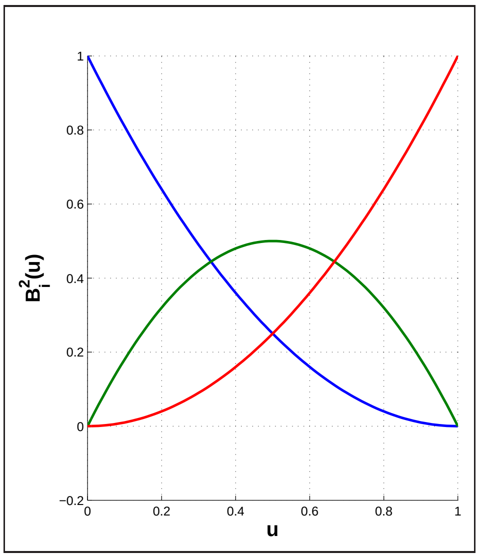


Figure 2.3: Quadratic Bernstein polynomials.

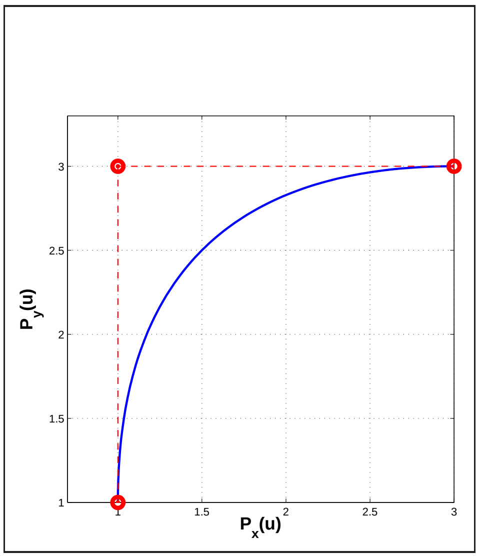


Figure 2.4: Curve defined in quadratic Bernstein polynomials basis using control points $p_1 = (1, 1)$, $p_2 = (1, 3)$, $p_3 = (3, 3)$.

interpolation property for Bézier curves - they go through the first and the last control points **only**.

Recursive definition of the Bernstein polynomial yields an efficient algorithm for computing values of the Bernstein polynomials at fixed values of u:

Algorithm 2.1 (Computing values of Bernstein polynomials)

```
ALL_BERNSTEIN(n, u, B)
{
   /*  Compute all n-th degree Bernstein polynomials.  */
   /*
     Input: @n - polynomial degree; @u - parameter value;
     Output: @B - array of Bernstein polynomials (B[0],...,B[n]).
   */

   B[0] = 1.0;
   u1 = 1.0 - u;
   for (j = 1; j <= n; j++)
   {
      saved = 0.0;
      for (k = 0; k < j; k++)
      {
         temp = B[k];
         B[k] = saved + u1 * temp;
         saved = u * temp;
      }
      B[j] = saved;
```

```
    }
}
```

Point evaluation procedure is straightforward and can be implemented using the following algorithm:

ALGORITHM 2.2 (POINT EVALUATION ON A BÉZIER CURVE)

```
POINT_ON_BEZIER_CURVE(P, n, u, C)
{
   /*  Compute point on Bezier curve.  */
   /*
     Input: @P - array of control points;
       @n - polynomial degree; @u - parameter value;
     Output: @C - evaluated point.
   */

   ALL_BERNSTEIN(n, u, B);
   C = 0.0;
   for (k = 0; k <= n; k++)
     C = C + B[k] * P[k];
}
```

Both algorithms presented here are taken from [31]. Bézier curves have the number of very useful properties for computer graphics and computer aided design:

1. convex hull property: the curves are contained in the convex hulls of their defining control points $\{b_i\}$;
2. affine invariance: rotations, translations, and scalings are applied to the curve by applying them to the control points;
3. variation diminishing property: no straight line (plane) intersects a curve more times than it intersects the curve's control polygon, i.e., a Bézier curve follows its control polygon rather closely and does not wiggle more than its control polygon;
4. endpoints interpolation: $P(0) = b_0$ and $P(1) = b_n$;
5. the k^{th} derivative at $u = 0$ $(u = 1)$ depends on the first (last) $k+1$ control points; in particular, $P'(0)$ and $P'(1)$ are parallel to $b_1 - b_0$ and $b_n - b_{n-1}$, respectively.

Bézier curve $P(u)$ of degree n has one more very useful property - it is obtained as the linear interpolation of two $(n-1)$ degree curves. This is demonstrated by a simple example of a quadratic Bézier curve evaluation at parameter value u:

$$\begin{aligned} P(u) = \sum_{i=0}^{2} B_i^2(u) b_i &= (1-u)^2 b_0 + 2u(1-u) b_1 + u^2 b_2 \\ &= (1-u)(\underbrace{(1-u)b_0 + ub_1}_{\text{linear}}) + u(\underbrace{(1-u)b_1 + ub_2}_{\text{linear}}) \end{aligned} \tag{2.21}$$

Thus, $P(u)$ is obtained as the linear interpolation of two first-degree Bézier curves; in particular, any point on $P(u)$ of degree two is obtained by three linear interpolations.

Fixing $u = u_0$ and denoting b_i by $b_{0,i}$ we get a recursive algorithm for computing the point $P(u_0) = b_{n,0}(u_0)$ on a n^{th} degree Bézier curve:

$$b_{k,i}(u_0) = (1 - u_0)b_{k-1,i}(u_0) + u_0 b_{k-1,i+1}(u_0), \qquad \text{for} \begin{cases} k = 1, \dots, n; \\ i = 0, \dots, n-k. \end{cases} \tag{2.22}$$

Equation (2.22) is called the *de Casteljau Algorithm*. It is a corner cutting process which can be implemented using the following algorithm:

Algorithm 2.3 (The de Casteljau Algorithm)

```
DE_CASTELJAU(P, n, u, C)
{
   /*  Compute point on Bezier curve.  */
   /*
     Input: @P - array of control points;
       @n - polynomial degree; @u - parameter value;
     Output: @C - evaluated point.
   */

   for (i = 0; i <= n; i++)
     Q[i] = P[i];
   for (k = 1; k <= n; k++)
     for (i = 0; i <= (n - k); i++)
       Q[i] = (1.0 - u) * Q[i] + u * Q[i + 1];
   C = Q[0];
}
```

The algorithm presented here is taken from [31]. This algorithm can be thought as a subdivision technique by geometric construction to find a point on a curve corresponding to a given value of the parameter u_0. First, we find the point on each edge of the control polygon that subdivides it proportionally according the value of u_0 and connect these points to form a set of line segments. We subdivide each of these new line segments proportionally according to u_0 and repeat the subdivision and line segment construction until we can construct only one line segment. The point that proportionally subdivides this line segment is the point on the curve corresponding to u_0. In general there are n cycles of subdivision.

Actually, this algorithm not only gives the point on the curve but also subdivides the curve at the parameter value of u_0 and constructs two control polygons which correspond to each new curve segment. Using notation of Equation (2.22), we obtain two control polygons (one for each part of the curve subdivided at the parameter value u_0) by the following equation:

$$\begin{aligned} B_1 &= \{b_{i,0}(u_0)\}; \\ B_2 &= \{b_{n-i,i}(u_0)\}, \qquad \text{for } i = 0, \dots, n; \end{aligned} \tag{2.23}$$

If the initial curve is defined on the parameter interval $[0, 1]$ then two new curves obtained after subdivision at parameter value $u_0 \in [0, 1]$ are defined on the parameter intervals $[0, u_0]$ and $[u_0, 1]$ correspondingly. An example of using de Casteljau algorithm for curve subdivision and point evaluation is shown in Figure 2.5.

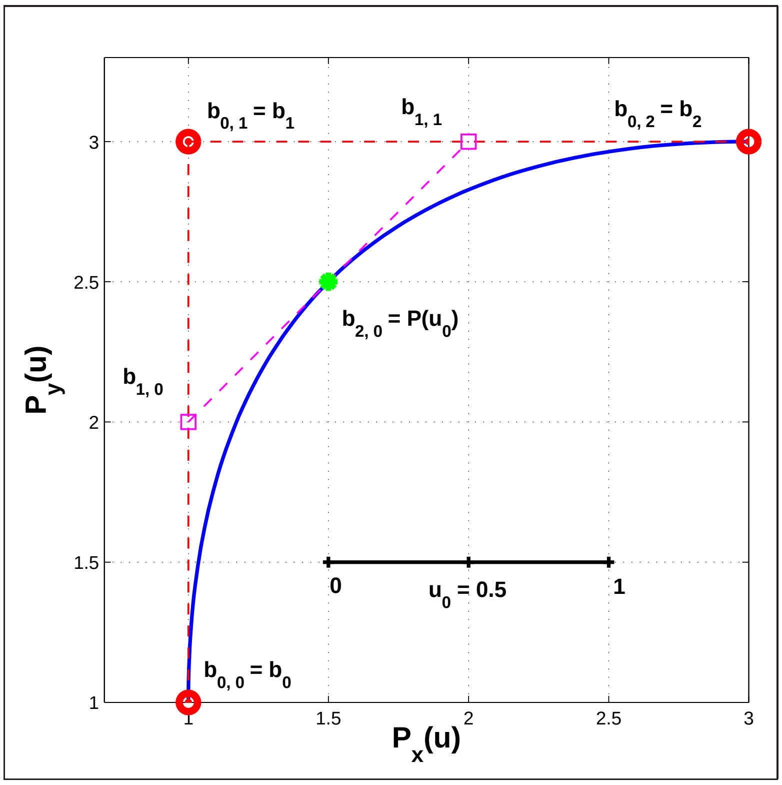


Figure 2.5: Curve subdivision and point evaluation using de Casteljau algorithm.

Notice, that using de Casteljau algorithm it is easy to obtain first derivative at the given point. As the consequence, the calculation of the normal vector at the given point is straightforward. Using notation of Equation (2.22), derivative vector at the given parameter value u_0 is computed by the following equation:

$$\vec{P}'(u_0) = n \cdot (\vec{b}_{n-1,1}(u_0) - \vec{b}_{n,0}(u_0)) = n \cdot (\vec{b}_{n,0}(u_0) - \vec{b}_{n-1,0}(u_0)) \qquad (2.24)$$

Normal vector at the given parameter value can be obtained by normalization of rotated by $\frac{\pi}{2}$ derivative vector. For 2D curves this normal vector is obtained as follows:

$$\begin{aligned} \vec{N}(u_0) &= (-1 \cdot P'_y(u_0),\ \ P'_x(u_0)) && \text{for counterclockwise direction;} \\ \vec{N}(u_0) &= (P'_y(u_0),\ \ -1 \cdot P'_x(u_0)) && \text{for clockwise direction;} \qquad (2.25) \end{aligned}$$

Choosing the direction of rotation (clockwise or counterclockwise) depends on the particular realization. For 3D curves the normal vector is not unique. In particular, each vector on the *normal plane* (plane which is perpendicular to the derivative vector) can be chosen.

2.3.4 B-Spline Basis Functions & NUBS Curves

Curves consisting of just one polynomial segment are often inadequate, because a high degree is required in order to represent curve through large number of control points. For instance, $(n-1)$ degrees are needed in order to represent a curve through n control points. For large n this significantly increases the computation time for evaluation of such curves. The solution would be to represent these curves by *piecewise polynomial* curves, i.e., to represent curves by the number of joint segments with appropriate level of continuity at the break points. This approach has one drawback - the level of continuity depends on the coordinates of control points.

If we have two segments of quadratic Bézier curves with control points $\{b_0^1, b_1^1, b_2^1\}$ and $\{b_0^2, b_1^2, b_2^2\}$ and we want to have the $C^{(1)}$ continuity at the break points b_2^1 and b_0^2, then the coordinates of two last control point of the first segment depend on coordinates of two first control point of the second segment. It is not convenient to manipulate with such segments during computer modeling.

We want to represent a curve using *piecewise polynomial functions* which have *local support* property - i.e., each of piecewise polynomial function is nonzero only on a limited number of subintervals, not the entire domain. We also need curves defined in the basis of these functions to have all nice properties of Bézier curves (see Section 2.3.3). Such functions exist and are called *B-spline basis functions*.

Let $T = \{t_0, \ldots, t_m\}$ be a nondecreasing sequence of real numbers, i.e., $t_i \leq t_{i+1}$, $i = 0, \ldots, m-1$. The t_i are called *knots*, and T is the *knot vector*. The i^{th} B-spline basis function of p-degree (*order* $p+1$), denoted by $N_i^p(u)$, is defined as

$$N_i^0(u) = \begin{cases} 1 & \text{if } t_i \leq u < t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$
$$N_i^p(u) = \frac{u - t_i}{t_{i+p} - t_i} N_i^{p-1}(u) + \frac{t_{i+p+1} - u}{t_{i+p+1} - t_{i+1}} N_{i+1}^{p-1}(u) \tag{2.26}$$

Notice, that the half-open interval $[t_i, t_{i+1})$ is called the i^{th} *knot span*, and can have zero length, since knots need not be distinct.

Figure 2.6 shows quadratic B-spline basis functions defined on the knot vector $T = \{0, 0, 0, 1, 2, 2, 3, 4, 5, 5, 5\}$.

B-spline basis functions have the number of important properties. For degree p and a knot vector $T = \{t_0, \ldots, t_m\}$ they show:

1. local support property: $N_i^p(u) = 0$ if u is outside the interval $[t_i, t_{i+p+1})$;
2. in any given knot span, $[t_j, t_{j+1})$, at most $p+1$ of the $N_i^p(u)$ are nonzero, namely the functions $N_{j-p}^p(u), \ldots, N_j^p(u)$;
3. nonnegativity: $N_i^p(u) \geq 0$ for all i, p and u;

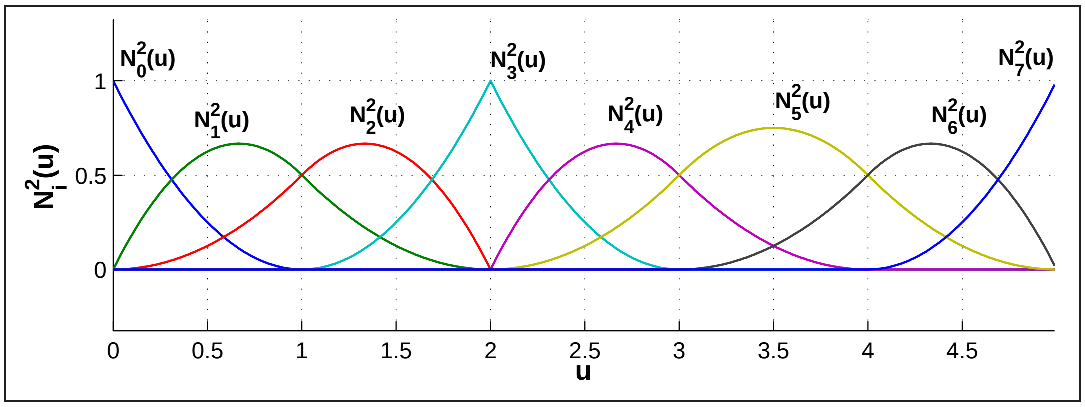


Figure 2.6: The quadratic B-spline basis functions, $T = \{0, 0, 0, 1, 2, 2, 3, 4, 5, 5, 5\}$.

4. partition of unity: for an arbitrary knot span, $[t_i, t_{i+1})$, $\sum_{j=i-p}^{i} N_j^p(u) = 1$ for all $u \in [t_i, t_{i+1})$;

5. at a knot $N_i^p(u)$ is $p - k$ times continuously differentiable, where k is the multiplicity of the knot. Hence, increasing degree increases continuity, and increasing knot multiplicity decreases continuity;

6. $N_i^p(u)$ attains exactly one maximum value, except for the case $p = 0$.

7. derivatives: denoting $N_{i,p}^{(k)}(u)$ as the k^{th} derivative of $N_i^p(u)$ we have

$$N_{i,p}^{(k)}(u) = p \left(\frac{N_{i,p-1}^{(k-1)}(u)}{u_{i+p} - u_i} - \frac{N_{i+1,p-1}^{(k-1)}(u)}{u_{i+p+1} - u_{i+1}} \right). \tag{2.27}$$

Once the degree is fixed the knot vector T completely determines the functions $N_i^p(u)$. There are several types of knot vectors which determine the shape of the curve. The most used in geometric modeling type is called *nonperiodic* (or *open*) knot vector, which has the form

$$T = \{\underbrace{a, \ldots, a}_{p+1}, t_{p+1}, \ldots, t_{m-p-1}, \underbrace{b, \ldots, b}_{p+1}\} \tag{2.28}$$

that is, the first and the last knots have multiplicity $p+1$. For nonperiodic knot vectors we have additional very important property of the basis functions:

- a knot vector of the form $T = \{\underbrace{0, \ldots, 0}_{p+1}, \underbrace{1, \ldots, 1}_{p+1}\}$ yields the Bernstein polynomials of degree p. Hence, $B_i^p(u) = N_i^p(u)$ for all $i \in [0, p]$, where p is degree.

The procedure of computing B-spline basis functions of degree p for a given $u \in [t_i, t_{i+1})$ consists of two steps. First of all, we have to find the knot span index i for a given u. It can be done using binary search procedure.

Algorithm 2.4 (Determining the knot span)

```
int FIND_SPAN(m, p, u, T)
{
   /*  Determine the knot span index.  */
   /*
     Input: @m - index of last knot; @p - degree;
       @u - parameter value; @T - knot vector;
     Return: the knot span index.
   */

   if (u == T[m - p])
     return (m - p - 1);
   int low = p;
   int hight = m - p;
   int mid = (low + hight) / 2;
   while ((u < T[mid]) || (u >= T[mid + 1]))
   {
      if (u < T[mid])
        hight = mid;
      else
        low = mid;
      mid = (low + hight) / 2;
   }
   return mid;
}
```

On the second step we have to compute only basis functions $N^p_{i-p}(u), \ldots, N^p_i(u)$, because all other basis functions are zero.

Algorithm 2.5 (Computing nonzero B-spline basis functions)

```
B_SPLINES(i, u, p, T, N)
{
   /*  Compute the nonvanishing B-spline basis functions.  */
   /*
     Input: @i - index basis function; @u - parameter value;
       @p - degree; @T - knot vector;
     Output: @N - nonvanishing B-Spline basis functions of degree p;
   */

   N[0] = 1.0;
   for (j = 1; j <= p; j++)
   {
      left[j] = u - T[i + 1 - j];
      right[j] = T[i + j] - u;
      saved = 0.0;
      for (r = 0; r < j; r++)
      {
         temp = N[r] / (right[r + 1] + left[j - r]);
         N[r] = saved + right[r + 1] * temp;
         temp = left[j - r] * temp;
      }
      N[j]=saved;
   }
}
```

Curves defined in the basis of the B-spline basis functions are called *nonuniform B-spline curves* (*NUBS curves*). A p^{th}-degree NUBS curve is represented as follows:

$$P(u) = \sum_{i=0}^{n} N_i^p(u) p_i \qquad a \le u \le b \tag{2.29}$$

where the $\{p_i\}$ are the *control points*, and the $\{N_i^p(u)\}$ are the p^{th}-degree B-spline functions defined on the nonperiodic knot vector $T = \{\underbrace{a, \ldots, a}_{p+1}, u_{p+1}, \ldots, u_{m-p-1}, \underbrace{b, \ldots, b}_{p+1}\}$.
In order to compute a point on a B-spline curve at a fixed u value one needs to multiply the values of nonzero basis functions with the corresponding control points.

Algorithm 2.6 (Point evaluation on a B-spline curve)

```
BSPLINE_CURVE_POINT(m, p, T, P, u, C)
{
   /*  Compute point on NUBS curve.  */
   /*
     Input: @m - index of last knot; @p - degree;
       @T - knot vector; @P - point vector;
       @u - parameter value;
     Output: @C - computed point on B-spline curve;
   */

   span = FIND_SPAN(m, p, u, T);
   B_SPLINES(span, u, p, T, N);
   C = 0.0;
   for (i = 0; i <= p; i++)
     C = C + N[i] * P[span - p + i];
}
```

All algorithms presented here are taken from [31]. Figure 2.7 shows a quadratic nonuniform B-spline curve defined on the knot vector $T = \{0, 0, 0, 1, 2, 2, 3, 4, 5, 5, 5\}$ using control points $p_0 = (1, 3)$, $p_1 = (2, 1)$, $p_2 = (3, 3)$, $p_3 = (4, 1)$, $p_4 = (5, 3)$, $p_5 = (6, 1)$, $p_6 = (7, 3)$, $p_7 = (8, 1)$.

Nonrational B-spline curves have the number of usefull properties:

1. endpoint interpolation: $P(a) = p_0$ and $P(b) = p_n$;
2. affine invariance: an affine transformation is applied to the curve by applying it to the control points;
3. strong convex hull property: if $u \in [u_i, u_{i+1})$, $p \le i \le m - p - 1$, then $P(u)$ is in the convex hull of the control points $p_{i-p}, \ldots, p_i$; as the consequence, the whole curve is contained in the convex hull of its control polygon;
4. the degree, p, number of control points, $n + 1$, and number of knots, $m + 1$, are related by $m = n + p + 1$;

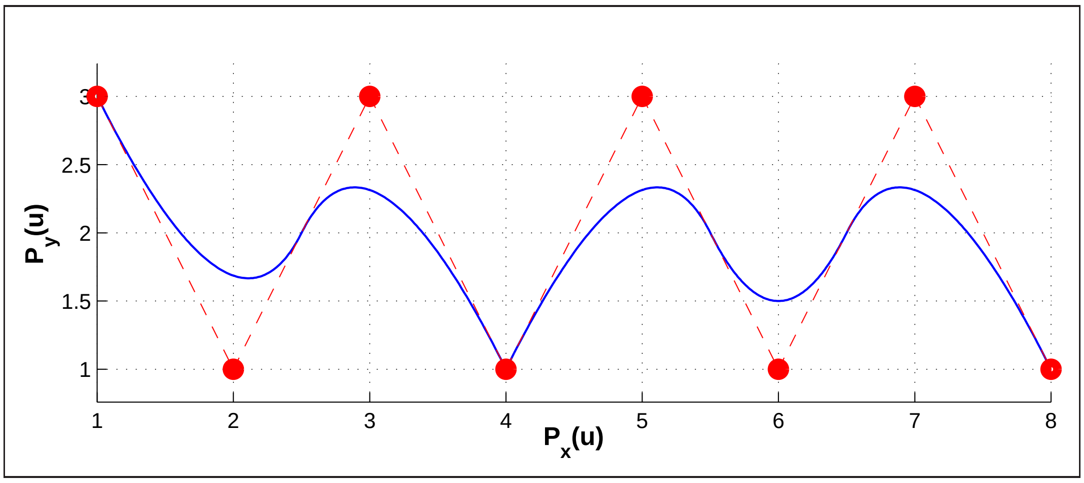


Figure 2.7: Curve defined in quadratic B-spline polynomial basis, $T = \{0, 0, 0, 1, 2, 2, 3, 4, 5, 5, 5\}$.

5. local modification scheme: moving p_i changes $P(u)$ only in the interval $[u_i, u_{i+p+1})$;
6. the control polygon represents a piecewise linear approximation to the curve;
7. variation diminishing property: the curve does not oscillate about any line (plane) more often than its control polygon oscillates about the line (plane);
8. if $n = p$ and T=$\{0, \ldots, 0, 1, \ldots, 1\}$, then $P(u)$ is a Bézier curve.

2.4 Rational Curves

2.4.1 Introduction

As was mentioned in Section 2.3, polynomial curves have the number of very usefull for geometric modeling purposes properties. But unfortunately not every curve can be represented using linear combination of polynomials. Even such important for geometric modeling curves as circles, ellipses, hyperbolas can not be represented using any polynomials. It is known from classical mathematics that all the conic sections can be represented using *rational functions*, which are defined as the ratio of two polynomials

$$x(u) = \frac{X(u)}{W(u)} \qquad y(u) = \frac{Y(u)}{W(u)} \qquad z(u) = \frac{Z(u)}{W(u)} \tag{2.30}$$

where $X(u)$, $Y(u)$, $Z(u)$, and $W(u)$ are polynomials, that is, each of the coordinate functions has the same denominator. Bézier curves and NUBS curves have the corresponding rational form which is discussed in the next sections.

2.4.2 Rational Bézier Curves

Problem of representing conic sections can be solved if we adjust so called weight values w_i to each control point b_i of the control polygon of Bézier curve. This leads us to the definition of *rational Bézier curve.* Rational Bézier curve of n^{th} degree is defined as follows:

$$P(u) = \frac{\sum_{i=0}^{n} B_i^n(u) w_i b_i}{\sum_{i=0}^{n} B_i^n(u) w_i} \qquad i \in [a, b] \tag{2.31}$$

The b_i are the control points which form the control polygon, $B_i^n(u)$ is the Bernstein polynomials as defined in Section 2.3.3, and w_i are weights. If we denote $R_i^n(u)$ by the following equation:

$$R_i^n(u) = \frac{B_i^n(u) w_i}{\sum_{i=0}^{n} B_i^n(u) w_i} \tag{2.32}$$

then we get another representation of rational Bézier curve

$$P(u) = \sum_{i=0}^{n} R_i^n(u) b_i \qquad i \in [a, b] \tag{2.33}$$

$R_i^n(u)$ in this notation are the *rational Bézier basis functions.* These functions have all properties of Bernstein polynomials which were defined in Section 2.3.3. Notice, that if $w_i = 1$ for all i, then $R_i^n(u) = B_i^n(u)$ for all i, i.e., the $B_i^n(u)$ are the special case of the $R_i^n(u)$. Rational Bézier curves also have all properties of polynomial Bézier curves which were defined in Section 2.3.3. The only one drawback of rational Bézier curves is the time consuming procedure of curve evaluation. The simple de Casteljau algorithm can not be applied in order to evaluate such curves. Therefore, rational Bézier curves in d-dimensional space are often represented as a polynomial Bézier curves in $(d + 1)$-dimensional space using *homogeneous coordinates.* Lets consider a rational Bézier curve in $3D$. Each point of such curve have coordinates $P = (x, y, z)$ in Cartesian space. P can be written as $P^w = (wx, wy, wz, w) = (X, Y, Z, W)$ in $4D$ space, $w \neq 0$. Now P is obtained from P^w by dividing all coordinates by the fourth coordinate, W. This process is called mapping P^w from the origin to the hyperplane $W = 1$. This mapping, denoted by H, is a perspective map with center at the origin

$$P = H\{P^w\} = H\{(X, Y, Z, W)\} = \begin{cases} (\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W}) & \text{if } W \neq 0 \\ \text{direction } (X, Y, Z) & \text{if } W = 0 \end{cases}$$

If for a given set of control points b_i and weights w_i we construct the weighted set of control points $b_i^w = (w_i x_i, w_i y_i, w_i z_i, w_i)$, then we can define the polynomial Bézier curve in $4D$ space

$$P^w(u) = \sum_{i=0}^{n} B_i^n(u) b_i^w \tag{2.34}$$

Notice, that this curve can be easily evaluated using de Casteljau algorithm for subdivision. If we apply the perspective map H to $P^w(u)$, then we get the corresponding rational Bézier curve in 3D

$$H\{P^w(u)\} = P(u) \tag{2.35}$$

2.4.3 NURBS Curves

A p^{th}-degree *nonuniform rational B-spline curve* (*NURBS curve*) is defined as follows:

$$P(u) = \frac{\sum_{i=0}^{n} N_i^p(u) w_i p_i}{\sum_{i=0}^{n} N_i^p(u) w_i} \qquad a \leq u \leq b \tag{2.36}$$

where the $\{p_i\}$ are the *control points* which form a *control polygon*, the $\{w_i\}$ are the *weights*, and the $\{N_i^p(u)\}$ are the p^{th}-degree B-spline basis functions defined on the nonperiodic knot vector $T = \{\underbrace{a, \ldots, a}_{p+1}, u_{p+1}, \ldots, u_{m-p-1}, \underbrace{b, \ldots, b}_{p+1}\}$.

If we denote $R_i^p(u)$ by the following equation:

$$R_i^p(u) = \frac{N_i^p(u) w_i}{\sum_{i=0}^{n} N_i^p(u) w_i} \tag{2.37}$$

then we get another representation of NURBS curve

$$P(u) = \sum_{i=0}^{n} R_i^n(u) p_i \qquad a \leq u \leq b \tag{2.38}$$

$R_i^p(u)$ are the *rational B-spline basis functions* which have all properties of nonrational B-spline basis functions defined in Section 2.3.4. Notice, that if $w_i = 1$ for all i, then $R_i^p(u) = N_i^p(u)$ for all i, i.e., the $N_i^p(u)$ are the special case of the $R_i^p(u)$. NURBS curves also have all properties of NUBS curves which were defined in Section 2.3.4. Notice, that conic sections can be easily represented by NURBS curves.

As it was mentioned in Section 2.4.2, rational form of curves representation is not convenient. Therefore, *homogeneous coordinates* are often used in order to represent the NURBS curve in d-dimensional space as a NUBS curve in $(d+1)$-dimensional space. By this way all the algorithms for NUBS curve evaluation can be applied in order to evaluate NURBS curve. If for a given set of control points p_i and weights w_i we construct the weighted set of control points $p_i^w = (w_i x_i, w_i y_i, w_i z_i, w_i)$, then we can define the NUBS curve in $4D$ space as follows:

$$P^w(u) = \sum_{i=0}^{n} N_i^p(u) p_i^w \tag{2.39}$$

If we apply the perspective map H to $P^w(u)$, we get the corresponding NURBS curve in 3D:

$$H\{P^w(u)\} = P(u) \tag{2.40}$$

2.5 Rational Surfaces

2.5.1 Introduction

A surface is a vector-valued function of two parameters, u and v, and represents a mapping of a region, R, of the uv plane into Euclidian three-dimensional space [31].

Many schemes of surface representation have been proposed (see [19]). They differ in the coordinate functions used and the type of region R. One of the simplest and most widely used method in geometric modeling applications is the *tensor product* scheme.

The tensor product method is a bidirectional curve scheme. It uses basis functions and geometric coefficients. The basis functions are bivariate functions of u and v, which are constructed as products of univariate basis functions. The geometric coefficients are geometrically arranged in a bidirectional, $n \times m$ net. Tensor product surfaces have the general form of representation given by the following equation:

$$S(u,v) = (x(u,v), y(u,v), z(u,v)) = \sum_{i=0}^{n}\sum_{j=0}^{m} f_i(u) g_j(v) p_{i,j} \tag{2.41}$$

where $p_{i,j} = (x_{i,j}, y_{i,j}, z_{i,j})$, $a \le u \le b$, and $c \le v \le d$.

Two commonly used in geometric modeling tensor product surfaces are *Bézier surfaces* and *NUBS surfaces* which have excellent geometric properties, but unfortunately can represent only small subset of polynomial surfaces. Therefore, the more general rational form if often used in computer aided design. *Rational Bézier surfaces* and *NURBS surfaces* are not tensor product surfaces, but they can be represented by the central projection of a four dimensional tensor product hypersurfaces into three-dimensional space. Rational Bézier surfaces and NURBS surfaces are discussed in the next sections.

2.5.2 Rational Bézier Surfaces

Points on a *rational Bézier surface* are given by the following tensor product:

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u) B_j^m(v) w_{i,j} b_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u) B_j^m(v) w_{i,j}} = \sum_{i=0}^{n}\sum_{j=0}^{m} R_{i,j}(u,v) b_{i,j}$$

$$\text{where} \qquad R_{i,j}(u,v) = \frac{B_i^n(u) B_j^m(v) w_{i,j}}{\sum_{r=0}^{n}\sum_{s=0}^{m} B_i^n(u) B_j^m(v) w_{i,j}}, \quad a \le u \le b, \ c \le v \le d \tag{2.42}$$

where $B_i^n(u)$ and $B_j^m(v)$ are the Bernstein basis functions in the u and v parametric directions and $b_{i,j}$ are the control points in $3D$ which form the *control mesh* of the surface. In order to simplify definition, rational Bézier surfaces are often represented by the perspective projection of a four-dimensional polynomial Bézier surfaces (see [30]).

$$S(u,v) = H\{S^w(u,v)\} \qquad \text{where } S^w(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u) B_j^m(v) b_{i,j}^w \tag{2.43}$$

Figure 2.8 shows a special case of rational biquadratic ($n = m = 2$) Bézier surface ($w_{i,j} = 1$ for all i, j) together with its control mesh of control points

$$\begin{aligned} p_{0,0} &= (1,1,1), & p_{0,1} &= (1,1,3), & p_{0,2} &= (3,1,3), \\ p_{1,0} &= (1,2,1), & p_{1,1} &= (1,2,3), & p_{1,2} &= (3,2,3), \\ p_{2,0} &= (1,3,1), & p_{2,1} &= (1,3,3), & p_{2,2} &= (3,3,3). \end{aligned}$$

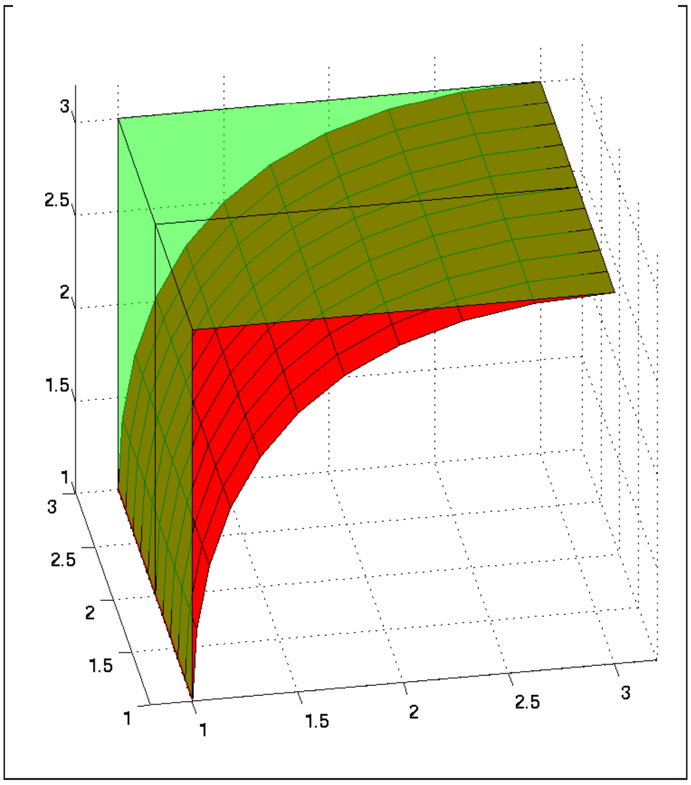


Figure 2.8: Special case of biquadratic rational Bézier surface (red) ($w_{i,j} = 1$ for all i, j) together with its control mesh (green).

The concept of homogeneous coordinates and perspective mapping was discussed in Section 2.4.2. Rational Bézier surfaces have the number of excellent properties which are mostly similar to the corresponding properties of rational Bézier curves:

1. $S(u, v)$ is contained in the convex hull of its control mesh;
2. the surface is invariant under an affine transformation;
3. the surface interpolates the four corner control points;
4. the degree of the surface in each parametric direction is one less than the number of control mesh vertices in that direction;

However, there is no known variation diminishing property for Bézier surfaces (see [32]). Notice also, that $R_{ij}(u, v) \geq 0$ for all i, j, u, v and $\sum_{i=0}^{n} \sum_{j=0}^{m} R_{i,j}(u, v) = 1$ for all u and v.

The deCasteljau algorithm can be easily extended to compute points and derivatives on a Bézier surface.

2.5.3 NURBS Surfaces

Rational B-spline surfaces (*NURBS surfaces*) are the standard for surface modeling in much of computer graphics and computer aided design. Many of the typical surface forms used in computer graphics and computer aided design, such as cylinders, spheres, ellipsoids of revolution, as well as more complex fully sculptured surfaces, are easily and accurately represented by NURBS surfaces. Technically, a NURBS surface is a special case of a general rational B-spline surface that uses a particular form of knot vector. For a NURBS surface, the knot vector has multiplicity of duplicate knot values at the ends equal to the order of the corresponding basis function [34].

A Cartesian product B-spline surface in four-dimensional homogeneous coordinate space of degree p in the u direction and degree q in the v direction is given by

$$S^w(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u)N_j^q(v)p_{i,j}^w \qquad a \le u \le b,\ \ c \le v \le d \tag{2.44}$$

where $\{p_{i,j}^w\}$ are the four-dimensional homogeneous control vertices, and $N_i^p(u)$ and $N_j^q(v)$ are the nonrational B-spline basis functions (see Section 2.3.4) defined on the knot vectors

$$\begin{aligned} U &= \{\underbrace{a,\dots,a}_{p+1}, u_{p+1},\dots,u_{r-p-1},\underbrace{b,\dots,b}_{p+1}\} \\ V &= \{\underbrace{c,\dots,c}_{q+1}, v_{q+1},\dots,v_{s-q-1},\underbrace{d,\dots,d}_{q+1}\} \end{aligned} \tag{2.45}$$

where $r = n+p+1$ and $s = m+q+1$.

Projecting back into three-dimensional space by dividing through by the homogeneous coordinate gives the NURBS surface

$$\begin{aligned} S(u,v) = H\{S^w(u,v)\} &= \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^n(u)N_j^m(v)w_{i,j}p_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^n(u)N_j^m(v)w_{i,j}} \\ &= \sum_{i=0}^{n}\sum_{j=0}^{m} R_{i,j}(u,v)p_{i,j} \end{aligned}$$

$$\text{where} \qquad R_{i,j}(u,v) = \frac{N_i^n(u)N_j^m(v)w_{i,j}}{\sum_{r=0}^{n}\sum_{s=0}^{m} N_i^n(u)N_j^m(v)w_{i,j}},\ \ a \le u \le b,\ \ c \le v \le d \tag{2.46}$$

Figure 2.9 shows a special case of biquadratic ($p = q = 2$) NURBS surface ($w_{i,j} = 1$ for all i, j) together with its control mesh defined on the knot vectors $U = \{0,0,0,1,2,2,3,4,5,5,5\}$ and $V = \{0,0,0,1,1,1\}$ using control points

$$\begin{aligned} &p_{0,0} = (1,1,3),\ \ p_{0,1} = (2,1,1),\ \ p_{0,2} = (3,1,3),\ \ p_{0,3} = (4,1,1), \\ &p_{0,4} = (5,1,3),\ \ p_{0,5} = (6,1,1),\ \ p_{0,6} = (7,1,3),\ \ p_{0,7} = (8,1,1), \\ &p_{1,0} = (1,2,3),\ \ p_{1,1} = (2,2,1),\ \ p_{1,2} = (3,2,3),\ \ p_{1,3} = (4,2,1), \\ &p_{1,4} = (5,2,3),\ \ p_{1,5} = (6,2,1),\ \ p_{1,6} = (7,2,3),\ \ p_{1,7} = (8,2,1), \end{aligned}$$

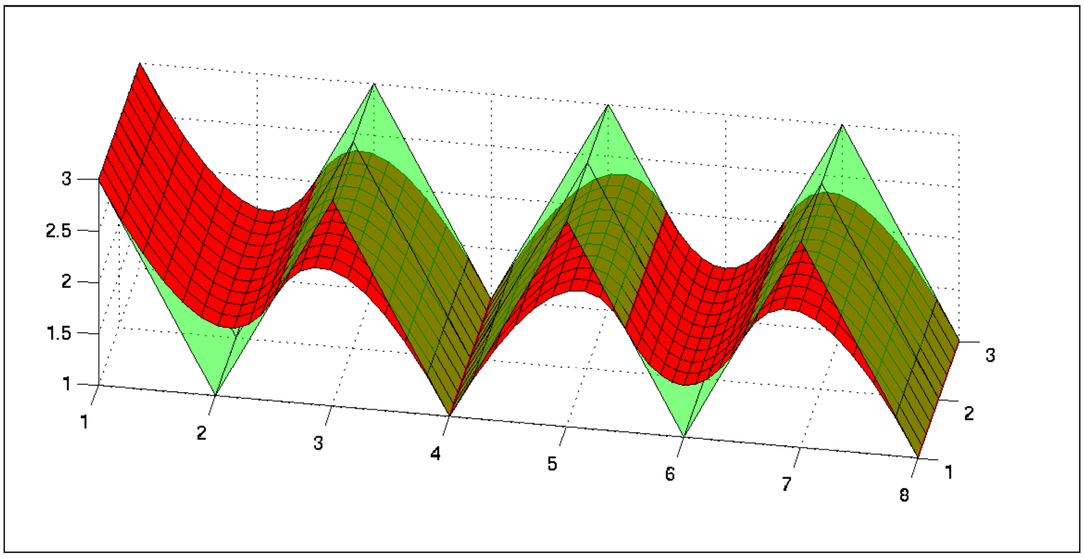


Figure 2.9: Special case of biquadratic NURBS surface (red) ($w_{i,j} = 1$ for all i, j) together with its control mesh (green) defined on the knot vectors $U = \{0, 0, 0, 1, 2, 2, 3, 4, 5, 5, 5\}$, $V = \{0, 0, 0, 1, 1, 1\}$.

$$p_{2,0} = (1, 3, 3),\ \ p_{2,1} = (2, 3, 1),\ \ p_{2,2} = (3, 3, 3),\ \ p_{2,3} = (4, 3, 1),$$
$$p_{2,4} = (5, 3, 3),\ \ p_{2,5} = (6, 3, 1),\ \ p_{2,6} = (7, 3, 3),\ \ p_{2,7} = (8, 3, 1).$$

NURBS surfaces have the following important geometric properties which are mostly similar to the corresponding properties of rational B-spline curves:

1. corner point interpolation: $S(a, c) = p_{0,0},\ S(b, c) = p_{n,0},\ S(a, d) = p_{0,m},\ S(b, d) = p_{n,m}$;

2. affine invariance: an affine transformation is applied to the surface by applying it to the control points;

3. strong convex hull property: assume $w_{i,j} > 0$ for all i, j. If $(u, v) \in [u_{i_0}, u_{i_0+1}) \times [v_{j_0}, v_{j_0+1})$, then $S(u, v)$ is in the convex hull of the control points $p_{i,j}$, $i_0 - p \leq i \leq i_0$ and $j_0 - q \leq j \leq j_0$; as the consequence, the whole surface is contained in the convex hull of its control mesh;

4. local modification: if $p_{i,j}$ is moved, or $w_{i,j}$ is changed, it affects the surface shape only in the rectangle $[u_i, u_{i+p+1}) \times [v_j, v_{j+q+1})$;

5. differentiability: $S(u, v)$ is $p - k$ times differentiable with respect to u at a u knot of multiplicity k, and $q - k$ times differentiable with respect to v at a v knot of multiplicity k;

6. nonrational Bézier surfaces, rational Bézier surfaces, and nonrational B-spline surfaces are special cases of NURBS surfaces.

Notice also, that $R_{ij}(u, v) \geq 0$ for all i, j, u, v and $\sum_{i=0}^{n} \sum_{j=0}^{m} R_{i,j}(u, v) = 1$ for all $u \in [a, b]$ and $v \in [c, d]$.

Chapter 3

Basics of Ray Tracing

3.1 Introduction

Ray tracing is a technique for *image synthesis*: creating a $2D$ picture of a $3D$ world [16]. A common goal of ray tracing is to give the viewer the impression of looking at a photograph (or movie) of some $3D$ scene. In order to understand the principles of ray tracing we need to understand how a camera records a physical scene onto film, since this is the action we want to simulate. We also need to know how light can be simulated and how intersection of the light with different scene objects can be computed. Since scene can consist of millions of objects we also need to understand how one can create different accelerated data structures in order to improve speed performance of the ray tracing. The next sections gives the explanations of these basic ideas behind ray tracing.

3.2 Light Simulation

The models of light used in simulations try to capture the different behaviors of light that arise from its dual nature (the light is a wave and a stream of particles at the same time). There are three different models of lights used in simulations:

- quantum optics;
- wave model;
- geometric optics.

Quantum optics can explain behavior of light at the submicroscopic level. However, this model is generally considered to be too detailed for the purposes of image generation for typical computer graphics scenes and is not commonly used.

Wave model captures effects such as diffraction, interference, and polarization, that arise when light interacts with objects of size comparable to the wavelength of light. However, for purposes of the image generation in computer graphics, the wave nature of light is also typically ignored.

The simplest and most commonly used model of light in computer graphics is the *geometric optics* model. In this model it is assumed, that the wavelength of light is much smaller than the scale of the objects that the light interacts with. Such effects as gravity, or magnetic fields are not taken into account.

The geometric optics assumes that light travels instantaneously through a medium in rays originated at the light emitters (*light sources*). When light interacts with objects it can be reflected, transmitted or absorbed depending on the physical properties of the object at the hit point.

3.3 Virtual Camera and Virtual Screen

The simplest camera model which can be simulated is the *pinhole camera*, illustrated in Figure 3.1. A flat piece of photographic film is placed at the back of a light-proof box.

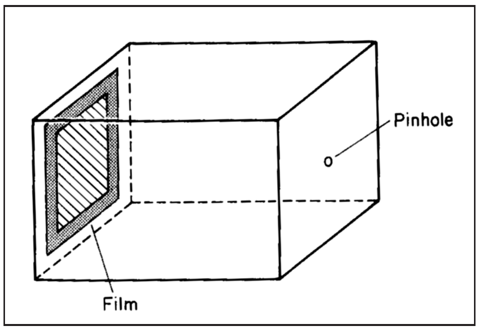


Figure 3.1: The pinhole camera model (adapted from [16]).

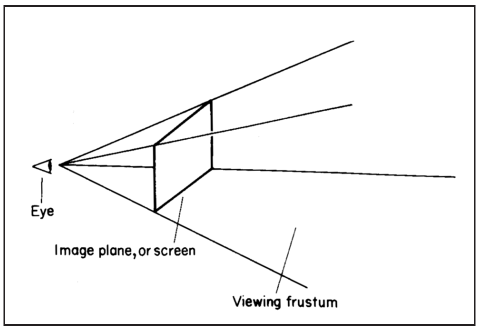


Figure 3.2: The modified pinhole camera model as commonly used in computer graphics (adapted from [16]).

A pin is used to pierce a single hole in the front of the box, which is then covered with a piece of opaque tape. When you wish to make a picture, you hold the camera steady and remove the tape for a while. Light enters the pinhole and strike the film, causing a chemical change in the emulsion. When you are done with the exposure you replace the tape over the hole. Despite its simplicity, this pinhole camera is quite practical for simulation in rendering software [16].

The classical computer graphics version of the pinhole camera moves the plane of the film out in *front* of the pinhole, and renames the pinhole as the *eye*, as shown in Figure 3.2. It is done for the sake of simulation convenience. The $3D$ volume that is visible to the eye, and may thus show up on the screen, is called the *viewing frustum*. The walls that form the frustum are called *clipping planes*. The plane of the screen is called the *image plane*. The location of the eye itself is referred to as the *eye position*.

3.4 Two Common Approaches in Ray Tracing

In Section 3.2 it has been argued that light is suppose to travel in rays during simulation process. Therefore, in order to obtain the picture on the image plane we need to consider all rays originated from light sources which hit the image plane (probably after some interactions with scene objects). There are two common approaches for simulating such process: *forward ray tracing* and *backward ray tracing*.

Forward ray tracing approach simulates light behavior in the natural way. All rays are originated at the light source and interact with scene surfaces. Some of them can intersect the image plane and make contribution to the pixel color at the point of intersection. This process is shown in Figure 3.3. But there is a problem with such a

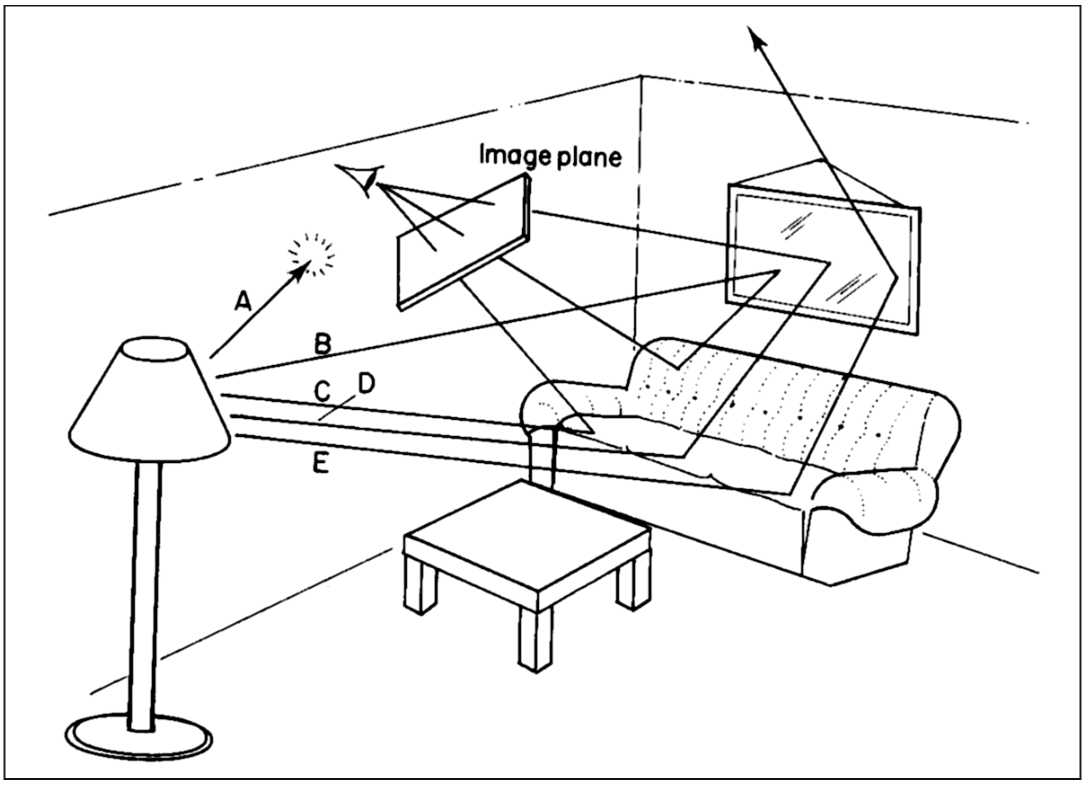


Figure 3.3: Forward ray tracing process (adapted from [16]).

direct simulation, and that is the amount of time it would take to produce an image. Each light source in a scene can generate millions of rays (photons) every second. Many of these rays hit objects that you would never see at all, even indirectly. Other rays just pass right out of the scene. Moreover, if the image plane size is small with respect to the size of the scene, the probability of hitting this image plane by a ray is small. This process might take years just to make one dim picture.

The key insight for computational efficiency is to reverse the problem, by following the rays backwards insted of forwards. This leads to concept of backwards ray tracing. In this approach we are following rays not forward, from the light source to objects to the eye, but backward, from the eye to objects to the light source. This is critical observation because it allows us to restrict our attention to rays that we know are useful to our image - the ones that enter our eye.

Different techniques have been invented in order to compute realistic images using backward ray tracing (*path tracing*). Some techniques use combinations of both approaches (*bidirectional pathtracing*, *photon mapping*). Their description is beyond the scope of this Master Thesis. The interested reader can refer to [12, 38, 24, 11].

3.5 Ray-Object Intersections

As it has been shown in Section 3.4, rays are used in order to solve the problem of image generation. The main problem arising here is how to compute the ray-object intersection. Solution of this problem depends on the mathematical objects representation in the ray tracing application. There are basically three kinds of objects commonly used in ray tracing applications, which have fast algorithms to compute intersection with a ray. These are spheres (see [38]), axis aligned bounding boxes (see [47]) and triangles (see [42]).

Spheres and axis aligned bounding boxes are usually used for constructing acceleration data structure, whose meaning is explained in the next section. Triangles are used as basic construction units of scene objects. Usually every object created in modeling applications tessellated into triangles before passing to the ray tracing application. During such tessellation some information about the shape of the object can be destroyed and the result image can be wrong.

Modeling applications often operate with NURBS surfaces which were discussed in Chapter 2. Even though much research has been done in the field of finding good enough curvature dependent and view dependent tessellation of NURBS surfaces (see [23, 14, 37]), performing direct ray-NURBS surface intersection test would give better result. There are two problems arising here. First problem is the absence of deterministic solutions of finding ray-NURBS surface intersection point - some numerical methods or tricks should be applied in order to solve this problem. As the consequence, the second problem is the time-consuming procedure of direct ray-NURBS surface intersection test. The goal of this Master Thesis is to compare existing approaches of finding the ray-NURBS surface intersection points, and improve them.

3.6 Acceleration Data Structures for Ray Tracing

Complex scenes consist of thousands of objects. In order to find the nearest ray-object intersection the brute and force approach would be to test the ray against each object in the scene and choose the nearest intersection from the found ones. Such approach is time consuming and not practical. For complex shape objects applying ray-object's *bounding box* intersection test first can slightly increase performance. But for a big number of objects in the scene the total computation time is still slow.

In order to improve speed performance of ray tracing applications different *acceleration spatial data structures* are used. The commonly used techniques are *uniform grids*, *octrees*, *bounding volume hierarchy* (*BVH*) and *KD-trees*. The reader interested in acceleration spatial data structures can refer to [17, 10, 4, 40, 18]. Figures 3.4–3.7 show visualization of different acceleration spatial data structures.

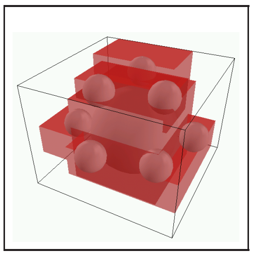


Figure 3.4: The visualization of BVH (adapted from [18]).

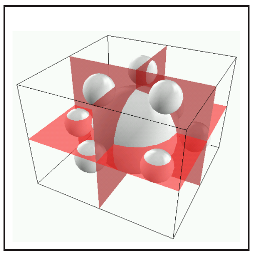


Figure 3.5: The visualization of Octree (adapted from [18]).

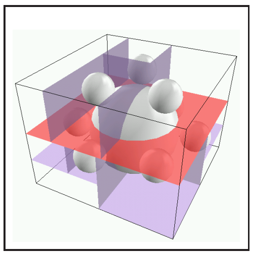


Figure 3.6: The visualization of KD-Tree (adapted from [18]).

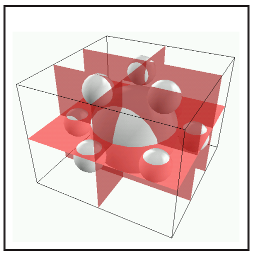


Figure 3.7: The visualization of Uniform Grid (adapted from [18]).

The general idea of every acceleration spatial data structure is to avoid as many unnecessary ray-object intersection tests as possible, by subdividing the whole scene either hierarchically or uniformly into cells of appropriate form (mostly axis aligned bounding boxes) and giving the simple and fast routine for data structure ray traversing. Each cell of the spatial data structure maintains information about the contained objects. If the given ray intersects a cell which contains objects, the ray-object intersection test is applied to each object inside this cell. The ideal case would be to test the ray with the nearest intersected object only, which is almost impossible in practice.

For complex shape objects like NURBS surfaces (or Bézier surfaces) it is possible to create *acceleration object data structure*. The idea is almost the same as for acceleration spatial data structure and mostly used in order to find the parametric regions of interest or the initial guess for numerical solvers. This idea is explained in Section 5.3 in more details.

Chapter 4

Ray Tracing Parametric Surfaces

4.1 Introduction

Ray tracing is one of the commonly used techniques in realistic image synthesis. In the heart of each ray tracer lies the ray-environment intersection routine which is able to test ray against every kind of object in the scene. Intersecting a ray with objects can be time consuming, and ray tracing is generally considered expensive compared with simpler methods.

The parametric method of surface representation is convenient for approximation and design of curved surfaces. In particular, Bézier surfaces and NURBS surfaces are extensively used in computer graphics and computer aided design. Unfortunately, most of the algorithms for intersecting rays with parametric surfaces are expensive or have problems in some special cases. Therefore, most of todays ray tracing applications tessellate parametric surfaces into triangles during the preprocessing step of image generation. Such approach significantly increases computation speed, but can compute wrong images (if tessellation was not good enough) and requires additional memory for storage of generated triangles. Therefore, the problem of finding fast and robust algorithms for ray tracing parametric surfaces is still opened research issue.

In Section 4.2 the history of solving ray-parametric surfaces intersection problem is discussed. In Sections 4.3 and 4.4 two commonly used approaches (*Bézier clipping approach* and *Newton's iteration approach*) which lie in the heart of the most practical methods, are explained in more details.

In the remaining of this chapter we assume for simplicity that rational Bézier curves are defined on the interval $[0, 1]$ and rational Bézier surfaces are defined on the domain $[0, 1] \times [0, 1]$. Extension of presented algorithms for general intervals and domains is straightforward.

4.2 A Bit of History

In far 1980 Whitted [46] described a method for finding ray-parametric surface intersection point using recursive surface subdivision. If the boundary volume of a patch is pierced by the ray, the patch is subdivided and bounding volumes are produced for each subpatch. The subdivision process is repeated until either no bounding volumes are intersected (the ray is assumed to not intersect the patch), or the intersected bounding volume is smaller then a predetermined minimum (the ray is assumed to intersect the bounded subpatch). Whitted used spheres as bounding volumes. This approach is not practical because it requires much memory for storing bounding volumes and is time consuming because many boundary volumes have to be tested against each ray.

In the same year Rubbin and Whitted [36] used basically the same method with bounding boxes instead of spheres. Some object space coherence proposed to be utilized, but even this modification did not significantly speed up the computation speed.

In 1982 Kajiya [21] used ideas from algebraic geometry to obtain a numerical procedure for intersecting a ray with a bicubic surface patch. His method is robust, not requiring preliminary subdivisions to satisfy some a priori approximation. It proceeds more quickly for patches of lower degree. The algorithm is simply structured and does not require memory overhead. But unfortunately the algorithm has many disadvantages. It does not significantly utilize coherence. The algorithm computes all intersections of the given ray with a surface patch, even if just closest intersection need to be found. And finally the algorithm performs enormous amounts of floating point operations. Kajiya estimates that 6000 floating point operations may have to be performed in order to find all of the intersections between one ray and one bicubic patch. In the modern ray tracing applications global illumination algorithms are commonly used, and million of rays can be tested against one parametric patch. It makes the proposed algorithm unpractical.

In 1985 Toth [43] proposed an algorithm for finding ray-surface intersections using multivariate Newton iteration. Toth proposed a method for identifying regions of parameter space in which the Newton iteration is guaranteed to converge to a unique solution by utilizing results from interval analysis. Identification of such regions also provides a good initial guess, and thus Newton iteration converges quickly. The method can be applied to any kind of surface for which routines are provided that compute bounds for the surface and partial derivatives over arbitrary regions of parameter space. The proposed method is robust and can deal correctly with all possible cases. It does not require any preprocessing of the surface and can favor the intersection closest to the origin. Less effort is spent on simpler surfaces, than on the more complicated ones. The efficiency of the method depends strongly upon the efficiency of the computations of bounds for both the surface and its partial derivatives. These computations may be performed several times during a single ray-surface intersection calculation. As the consequence Toth's method also consumes considerable amounts of computing time.

In 1986 Joy and Bhetanabholta [20] proposed to use the quasi-Newton methods for finding local minima of a function representing the squared distance of a ray from points on a parametric surface. Proposed algorithm can find intersections of a ray with arbitrary parametric surfaces and also gives a routine for calculation of first derivatives at each point on the surface. The algorithm utilizes the ray coherence in order to speed up the convergence of the quasi-Newton method. Values calculated on the final iteration for the last ray to hit the surface are stored in order to be reused for the next ray potentially intersecting this surface as the initial guess for quasi-Newton iteration. However, the proposed algorithm has problems concerning coherence utilization. Naive approach may cause convergence to incorrect solutions. Object space subdivision and certain classifications have to be applied in order to avoid this problem. But fine subdivision leads to excessive memory requirements.

In the same year Sweeney and Bartels [41] described a method for ray tracing general B-spline surfaces using refinement of the control mesh for each surface. On top of each refined mesh, a tree of tightly fitting, nested bounding boxes is constructed from the bottom up. The procedural method of Kajiya [22] for ray tracing fractals is then applied. This locates points of intersection near enough and gives the initial guess for Newton's method. Despite the appealing simplicity of this method there are several disadvantages, which make it unusable in real-life rendering systems. The method uses some global parameters whose incorrect settings can significantly increase execution time. The memory requirements of this method are large. Finally, the mathematical validity of proposed method is not proved.

Two improvements on the method reported by Sweeney and Bartels [41] have been proposed in 1987 by Levner [25] and Yang [49]. Levner [25] proposed to create a mesh of points which lie on the surface instead of refining the control vertex mesh. The method was originally developed for ray tracing B-spline surfaces, but has been applied successfully to bicubic surfaces in general. The advantage of this approach is that point evaluations on the surface allows treatment of general parametric surfaces.

Yang [49] proposed to create an individual octree for each surface by subdividing its bounding box. Thus, the tree of bounding boxes is constructed top-down rather than bottom-up. Since surface points are used rather than control points, bounding is tighter.

In 1990 Lischinski and Gonczarowski [26] proposed the algorithm which combines numerical techniques described by Toth [43] and subdivision techniques described by Rubbin and Whitted [36]. Their algorithm allows the utilization of ray coherence and reduces the average ray-parametric surface intersection time compared to both base methods. Uniform spatial subdivision is used in order to reduce the number of objects to be tested against each ray. The *Krawczyk's* operator is used in order to ensure the convergence of Newton method to the correct solution. Information computed while intersecting each ray is cached, and can later be reused for other rays. To prevent cache overflow, the LRU (least recently used) replacement strategy is used. Some techniques to handle reflected, refracted, and shadow rays in a more efficient manner is described.

In the same year Nishita [29] described a method for solving ray-rational Bézier surface intersection problem which he called *Bézier clipping*. This method can be categorized as partly a subdivision based algorithm and partly a numerical method. After representing a ray as the intersection of two planes the problem of finding intersection can be projected from $4D$ to $2D$ space. This reduces the number of arithmetic operations necessary to perform de Casteljau subdivision by 50% on each subdivision iteration. Some hacks to prevent numerical round-off have been proposed. Nishita [29] noted that the idea of Bézier clipping could be successfully applied in order to solve the problem of determining trimming regions, which are specified by the rational Bézier curve equations in parameter space. Later in 1996 Campanga and Slusallek [9] described problems of Bézier clipping algorithm and proposed efficient schemes for avoiding them. In 2000 Wang, Shih, and Chang [45] suggested a modification to the original Bézier clipping algorithm. They showed how one can exploit ray coherence in order to find the nearest intersection in the case of multiple intersections more efficiently.

One more method for ray tracing parametric surfaces was proposed in 1990 by Biard [7]. The algorithm he described is based on algebraic implicitization and inversion. Each surface is associated with numerical matrices in such a way that operations to be done on surfaces can be translated into operations on the corresponding matrices and treated by numerical matrix techniques. These matrices are an implicit version of a given parametric surface and contain all algebraic and topological information about it. This method is robust but too slow and not practical.

In 1992 Enger [13] proposed a new technique for finding the intersection of a ray with a parametric surface. He assumed that computer-generated pictures often contain relatively large uniformly colored sections so that it is unsatisfactory to perform similar calculations for all pixels on the screen eventhough many adjacent pictures could be eventually colored together. A method for obtaining these common regions with only a few arithmetic operations was proposed. The method extends real numbers to real intervals. Interval analysis methods are used to calculate the set of intersection points of ray clusters with objects in the scene. Unfortunately the assumption of the fact that *"adjacent pictures could be eventually colored together"* is not hold for high dynamic range realistic images which were obtained using global illumination algorithms. Moreover, the proposed algorithm works rather for ray casting then for ray tracing, and therefore is not practical.

In 1993 Barth and Stürzlinger [6] proposed an algorithm for efficient ray tracing of Bézier and B-spline surfaces. The described method proposes to construct a hierarchical data structure (binary tree) for each surface. Parallelepipeds are chosen as enclosures (elements of the binary tree), their orientation and the angles between their edges are chosen in such a way that they enclose the respective part of the surface as tightly as possible. The leaves of the tree contain small, almost plane parts of the surface. To intersect a ray with the surface, one needs to 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 a parallelepiped of appropriate subtree, the whole subtree is pruned from father consideration. Finally the search reaches all leaves whose enclosures are hit by the ray. Each leaf contains the approximating parallelogram, which is used to calculate the starting point (initial guess) for the Newton's iteration algorithm. Some problems which can be caused by wrong initial guess values have been noted with proposals how one can avoid them.

In 1994 Fournier and Buchanan [15] demonstrated how one can use *Chebyshev polynomials* to speed up the computation of the intersections between rays and parametric curves and surfaces. They exploit the *minmax property* of Chebyshev polynomials in order to obtain tight bounding boxes for enclosed surface patches. Conversion matrices from power basis and Bézier basis into Chebyshev basis as well as subdivision matrices were derived for cubic curves and surfaces. Later in 1996 Campanga and Slusallek [9] proposed some extensions of Chebyshev boxing in order to improve the general performance. Extending Chebyshev boxing to arbitrary degree is straightforward, but the basis conversion and subdivisions are more time consuming for higher degree. Chebyshev representation strategy has one drawback - there is no way for computing tight bounding volumes based on Chebyshev representation of a rational patch.

In the same year Brath, Lieger, and Schindler [5] showed how the general parametric surface can be adaptively subdivided into small parts in order to construct a binary tree of bounding volumes which are calculated using interval arithmetic for each of these surface patches. From linear approximations and intervals for the partial derivatives it is possible to construct parallelepipeds that adapt the origination and shape of the surface parts well and form tight enclosures. The proposed algorithm is similar to that used with Bézier and B-spline surfaces (see [6]), where the bounding volumes are derived from the convex hull property.

In 1996 Qin, Gong, and Tong [33] used polynomial extrapolation in order to accelerate the convergence of the Newton's method. Ray is defined to be the intersection of two planes. Parallelepipeds are used to enclose the respective patches as tightly as possible.

In the same year Campanga and Slusallek [9] compared Bézier clipping and Chebyshev boxing methods for solving ray-Bézier surface intersection problem. Some modifications of the original methods were proposed. One also was shown how these methods can be combined in order to achieve the correct result more quickly than using each method separately.

In 2000 Martin, Cohen, Fish, and Shirley [27] proposed a method for ray tracing trimmed NURBS surfaces. The used refinement of the knot vectors to generate the bounding volume hierarchy, which results in a lower tree depth than other subdivision-based methods. The idea was to refine the knot vectors in such a way, that after the transformation of a NURBS surfaces into Bézier patches, these patches are flat enough and yield tighter bounding boxes. Bézier patches are not stored in memory and used only for bounding box hierarchy construction. Two schemes for knot vectors refinement have been proposed: an *adaptive subdivision* and a *curvature-based refinement*. The advantage of this approach is that the flat patches yield a good initial guess for Newton's method. Schemes for NURBS surface evaluation based on the knot vectors refinement have been proposed. Unfortunately the NURBS surfaces have the slower algorithms for surface evaluation than Bézier surfaces. Newton's method need to evaluate the point and two partail derivatives on each iteration in order to have quadratic convergence. It is always better to subdivide the initial NURBS surface into Bézier patches during the preprocessing step. This requires the additional amount of memory for storing all patches separately, but it can significantly speed up the computations.

In the same year Wand, Chung, and Chang [45] proposed an efficient algorithm for enhancing the performance of both numerical and subdivision methods. In order to improve the Barth and Stürzlinger's algorithm (see [6]), they proposed to use regular grids instead of binary trees in order to find initial guess for Newton's method. The proposed modification also allows to utilize the ray coherence in an efficient way. One was also suggested how the ray coherence can be utilized in order to obtain the nearest intersection using Bézier clipping algorithm (see [29]) efficiently if multiple intersections exist.

In 2001 Wand, Shin, and Chang [44] proposed an algorithm which combines Bézier clipping algorithm and Newton's iteration algorithm in an efficient way which utilizes ray coherence. First intersection point of the given ray with a Bézier surface is found using Bézier clipping algorithm. All subsequent intersection points along the same scanline are found using Newton's iteration algorithm. Last found intersection point is used as the initial guess for the subsequent one. The *obstruction detecting technique* is then used to verify whether an intersection point found by using Newton's method is the closest one.When Newton's method fails to achieve convergence, Bézier clipping is used as the substitution to find the intersection points.

4.3 Bézier Clipping Approach

4.3.1 Introduction

Bézier clipping approach was first introduced by Nishita [29]. Proposed algorithm was referred to as *new algorithm for computing the points at which a ray intersects a rational Bézier surface patch, and also an algorithm for determining if an intersection point lies within a region trimmed by piecewise Bézier curves.*

Although Nishita's algorithm was proposed to solve problems of finding ray-rational Bézier patch intersection and determining trimming regions in parametric space of the rational Bézier surfaces, it can be also applied to solve general ray-rational Bézier curve in $2D$ intersection problem.

In order to give good explanation of Bézier clipping approach, the more simple ray-rational Bézier curve in $2D$ intersection is explained first in Section 4.3.2. In Section 4.3.3 Bézier clipping approach is applied in order to solve more complex ray-rational Bézier surface intersection problem. In Section 4.3.4 detected problems of Bézier clipping algorithm is discussed with the proposals how to solve them. Some improvement ideas which are completely new is described in Section 4.3.5.

4.3.2 Ray-rational Bézier Curve in $2D$ Intersection Problem

Figure 4.1 shows an example of rational Bézier curve $P(u)$ and the ray $R(t)$ which intersects the curve. Rational Bézier curve is defined by its parametric equation (see Section 2.4.2):

$$P(u) = \frac{\sum_{i=0}^{n} p_i w_i B_i^n(u)}{\sum_{i=0}^{n} w_i B_i^n(u)}, \quad u \in [0,1] \tag{4.1}$$

Ray is also defined by its parametric equation using ray origin $\vec{o}$ and direction $\vec{d}$:

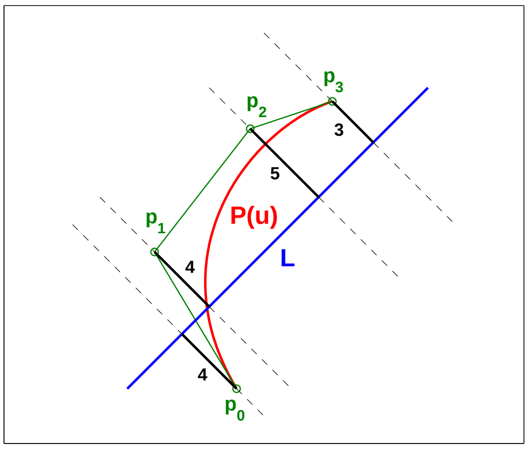


Figure 4.1: Bézier curve-ray intersection (adapted from [29]).

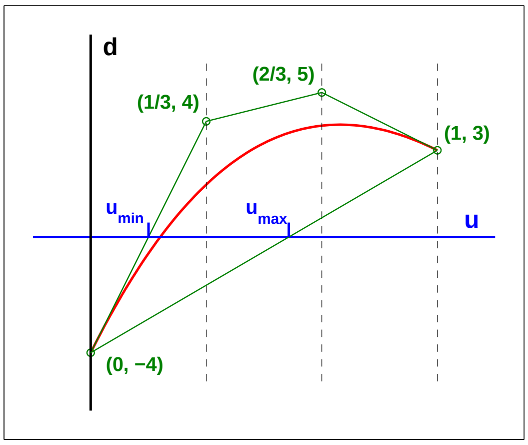


Figure 4.2: Explicit Bézier curve (adapted from [29]).

$$R(t) = \vec{o} + t\vec{d} \tag{4.2}$$

First we define line L which contains the ray R by the implicit equation via given point $\vec{o}$ on the line and given normal vector $\vec{n} = (a, b)$:

$$ax + by + \underbrace{(-ao_x - bo_y)}_{c} = 0 \tag{4.3}$$

Normal vector $\vec{n}$ of the line L can be obtained by the rotation of the ray direction vector $\vec{d}$ by $\frac{\pi}{2}$ in the counterclockwise direction:

$$(a, b) = (-d_y, d_x) \tag{4.4}$$

Using Equation (4.4) coefficient c of the Equation (4.3) can be computed

$$c = d_y o_x - d_x o_y \tag{4.5}$$

The intersection of the line L and the rational Bézier curve $P(u)$ can be found by substituting Equation (4.1) into Equation (4.3):

$$d(u) = \sum_{i=0}^{n} d_i B_i^n(u) = 0, \qquad d_i = w_i(ap_{x_i} + bp_{y_i} + c) \tag{4.6}$$

Note that $d(u) = 0$ for all values of u at which $P(u)$ intersects L. Also, d_i is the weighted distance from corresponding control point p_i to L as shown in Figure 4.1.

Since $\sum_{i=0}^{n} \frac{i}{n} B_i^n(u) \equiv u[(1-u)+u]^n \equiv u$ and the function d(u) in Equation (4.6) is a polynomial in Bernstein form, it can be represented as an "explicit" Bézier curve as follows:

$$D(u) = \begin{pmatrix} u \\ d(u) \end{pmatrix} = \sum_{i=0}^{n} \begin{pmatrix} i/n \\ d_i \end{pmatrix} B_i^n(u) \tag{4.7}$$

Notice that horizontal coordinate of any point $D(u)$ is in fact equal to the parameter value u. Figure 4.2 shows the "explicit" Bézier curve $D(u)$ which corresponds to the Bézier curve $P(u)$ in Figure 4.1.

Since $D(u)$ crosses the u-axis at the same u values at which $P(u)$ intersects L, we can apply the convex hull property of Bézier curves (see Section 2.3.3) to identify ranges of u for which $P(u)$ does **not** intersect L.

If convex hull of the $D(u)$ does not intersect the u axis then initial rational Bézier curve is not intersected by the line L (by the ray $R(t)$). Referring again to Figure 4.2, the convex hull of the $D(u)$ intersects the u axis at the points $u = u_{min}$ and $u = u_{max}$. This means that the initial Bézier curve does not intersect the line L in the parameter ranges $0 \leq u \leq u_{min} \bigcup u_{max} \leq u \leq 1$. Bézier clipping is completed by subdividing $P(u)$ into three segments using de Casteljau algorithm (see Section 2.3.3). Segment 1 is defined over $0 \leq u \leq u_{min}$, segment 2 over $u_{min} \leq u \leq u_{max}$, and segment 3 over $u_{max} \leq u \leq 1$.

From this point we consider the second segment as the initial rational Bézier patch defined over the interval $[u_{min}, v_{min}]$ and repeat the algorithm until either of two cases occurs:

- convex hull of the $D(u)$ does not intersect the u axis, and no intersection reported;
- parameter interval $[u_{min}, u_{max}]$ is small enough (related to ε) and intersection is assumed to exist at $u_{int} = \frac{u_{min}+u_{max}}{2}$;

The only problem occurs if the initial ray $R(t)$ intersects the given rational Bézier curve more than once. In this case the proposed scheme can go into the infinite loop. In order to prevent it, we subdivide the "explicit" Bézier curve in half and repeat the proposed algorithm recursively if on the current iteration Bézier clip fails to reduce the parameter interval width by at least 20%, i.e., if $(u_{max} - u_{min}) > 0.8$. So, the Bézier clipping approach can find multiple intersections if they exist. One can store intersections in the sorted list to be able to obtain the nearest one quickly. Notice that this approach gives just approximate value of intersection which is represented by the value u_{int} in parameter u domain of the initial rational Bézier curve. In ray tracing applications we are often interested in the corresponding value t of the ray in Equation (4.2). Notice that this value is the distance from the ray origin to the point of intersection on the curve. We can obtain this value by projection of vector from ray origin to the evaluated intersection point on the curve to the ray direction vector:

$$t = \vec{d} \cdot (\vec{P}(u_{int}) - \vec{o}) \tag{4.8}$$

Notice that the value of t can be less than 0. In this case ray is assumed to not intersect the curve (the curve in behind the ray).

4.3.3 Ray-rational Bézier Surface Intersection Problem

Bézier clipping algorithm for finding ray-rational Bézier curve in $2D$ intersection can be successfully applied in order to solve more general problem of finding ray-rational Bézier surface intersection. Bézier clipping concept is used to iteratively clip away regions of the surface which do not intersect the ray.

The most costly single operation in any subdivision based ray-Bézier surface intersection algorithm is de Casteljau subdivision. Typically, subdivision is performed in R^3 for non-rational surfaces, and in R^4 for rational surfaces. Woodward [48] shows how the problem can be projected to R^2. This means that the number of arithmetic operations to subdivide a surface is reduced by 33% and 50% for nonrational and rational surfaces respectively.

Suppose that we have a rational Bézier surface $S(u, v)$ (see Section 2.5.2) given by equation

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)w_{i,j}p_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)w_{i,j}}, \quad 0 \le u, v \le 1 \tag{4.9}$$

and a ray $R(t)$ defined by its parametric equation

$$R(t) = \vec{o} + t\vec{d} \tag{4.10}$$

where $\vec{o}$ is the ray origin and $\vec{d}$ is the ray direction. In order to find intersection of the given surface with the given ray, we have to change ray representation. We want to represent the ray as an intersection of two planes as shown in Figure 4.3. Although we

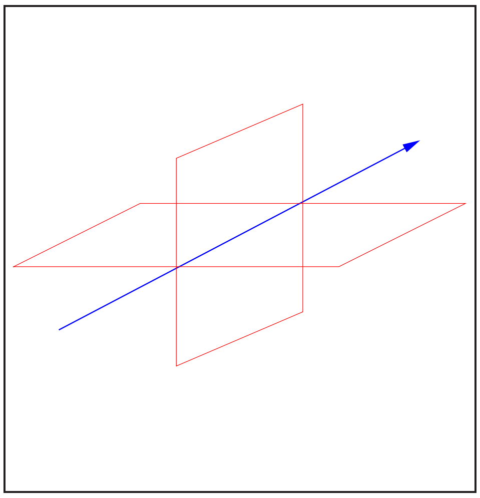


Figure 4.3: Bézier clipping: ray as an intersection of two planes.

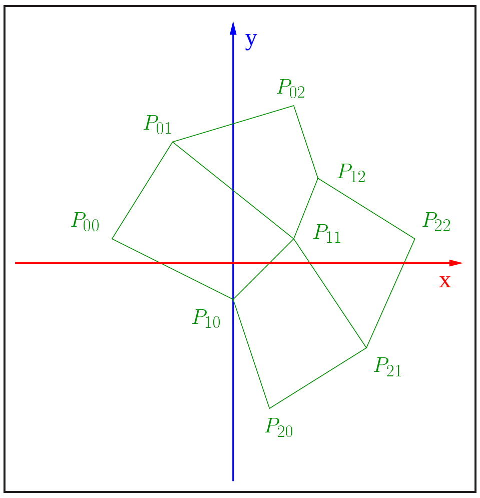


Figure 4.4: Bézier clipping: projected surface $P(u, v)$.

can choose the planes arbitrary it is better to consider two orthogonal planes further denoted by P_1 and P_2. In order to define these planes, we need to define their normal vectors first. Suppose, $\vec{n}_1 = (a_1, b_1, c_1)$ and $\vec{n}_2 = (a_2, b_2, c_2)$ are normal vectors of the first and the second planes respectively. Then we can choose normal vectors that are perpendicular to ray direction $\vec{q}$:

$$\begin{aligned}\vec{n}_1 &= (a_1, b_1, c_1) = (-d_y, d_x, 0)\\ \vec{n}_2 &= (a_2, b_2, c_2) = (0, -d_z, d_y)\end{aligned} \tag{4.11}$$

Now we can represent ray as an intersection of two planes given by their implicit equations

$$\begin{aligned}P_1 &: \ a_1x + b_1y + c_1z + \underbrace{(-a_1o_x - b_1o_y - c_1o_z)}_{d_1} = 0\\ P_2 &: \ a_2x + b_2y + c_2z + \underbrace{(-a_2o_x - b_2o_y - c_2o_z)}_{d_2} = 0\end{aligned} \tag{4.12}$$

The intersection of plane k and the surface can be represented by substituting Equation (4.9) into Equation (4.12) and clearing the denominator

$$\begin{aligned}d^k(u,v) &= \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)d_{ij}^k = 0\\ \text{where } \ d_{ij}^k &= w_{ij}(a_kp_{x_{ij}} + b_kp_{y_{ij}} + c_kp_{z_{ij}} + d_k), \ \ k = [1,2]\end{aligned} \tag{4.13}$$

Note that $S(u, v)$ lies on the plane k iff $d^k(u, v) = 0$. Note also that d_{ij}^k is the weighted distance from control point p_{ij} to plane k. We can now project the surface to a two dimensional (x, y) coordinate system by taking the projected point coordinates to be

$$P_{ij} = (x_{ij}, y_{ij}) = (d_{ij}^1, d_{ij}^2) \tag{4.14}$$

The projected surface is defined by equation

$$P(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)P_{ij} \tag{4.15}$$

In this projection, plane P_1 becomes the y axis, plane P_2 becomes the x axis, and the ray projects to the coordinate system origin. Figure 4.4 shows an example of projected surface $P(u, v)$. The ray-patch intersection problem now becomes one of finding

$$\{(u,v)|P(u,v) = 0; \ \ 0 \le u, v \le 1\} \tag{4.16}$$

This means that de Casteljau algorithm for surface subdivision can be performed now in $2D$ for projected Bézier surface $P(u, v)$ rather than in $4D$ for rational Bézier surface $S(u, v)$. Bézier clipping algorithm now can be easily extended in order to find ray-rational Bézier surface intersection.

First we need to determine two vectors L_u and L_v in the (x, y) space. This two vectors are alternatively used on each iteration of Bézier clipping algorithm. The best choice for these vectors is

$$\vec{L}_u = \frac{1}{2}\left[(\vec{P}_{20} - \vec{P}_{00}) + (\vec{P}_{22} - \vec{P}_{02})\right]$$
$$\vec{L}_v = \frac{1}{2}\left[(\vec{P}_{02} - \vec{P}_{00}) + (\vec{P}_{22} - \vec{P}_{20})\right] \tag{4.17}$$

i.e., the direction of L_u should be approximately perpendicular to u direction, and the direction of L_v should be approximately perpendicular to v direction as shown in Figure 4.5. In this case the Bézier clipping algorithm converges to the right solution more quickly.

If some corner points of the projected surface are coincide then Equation (4.17) can compute zero length vectors. We can avoid this problem in the following way:

- if just one of two vectors is of zero length, it is recomputed as rotated by 90 degrees second vector;
- if both vectors are of zero length, the bounding box of the projected surface patch is used in order to compute these vectors (we can use either diagonals or edges for this purpose);

Now, in order to solve equation $P(u, v) = 0$, we need to alternate the directions L_u and L_v and iterate on the following steps:

Step 1. We need to determine the signed distances d_{ij} of the projected control points (x_{ij}, y_{ij}) to L_u (L_v). This gives a Bézier representation of the "distance-to L_u" ("distance-to L_v") function that can be used to determine the distance of an arbitrary point to L_u (L_v)

$$d(u, v) = \sum_{i=0}^{n}\sum_{j=0}^{m} d_{ij} B_i^n(u) B_j^m(v) \tag{4.18}$$

The function $d(u, v)$ can be represented, in an (u, v, d) coordinate system, as an *explicit* surface, whose control points $D_{ij} = (u_{ij}, v_{ij}, d(u_{ij}, v_{ij}))$ are evenly spaced in u and v: $u_{ij} = \frac{i}{n}$, $v_{ij} = \frac{j}{m}$. A point on such surface has coordinates

$$D(u, v) = \sum_{i=0}^{n}\sum_{j=0}^{m} D_{ij} B_i^n(u) B_j^m(v) \tag{4.19}$$

Step 2. On the L_u-iteration we plot the values $\left(\frac{i}{n}, d_{ij}\right)$ in a $2D$ (u, d) diagram. On the L_v-iteration we plot the values $\left(\frac{j}{m}, d_{ij}\right)$ in a $2D$ (v, d) diagram. This is shown in Figure 4.6.

Step 3. Now we determine the convex hull of the points in this diagram and intersect it with the u-axis (v-axis). This gives an interval $[u_{min}, u_{max}]$ ($[v_{min}, v_{max}]$). We conclude that $d(u, v) \neq 0$, and therefore $P(u, v) \neq 0$, for $u < u_{min}$ and $u > u_{max}$ ($v < v_{min}$ and $v > v_{max}$).

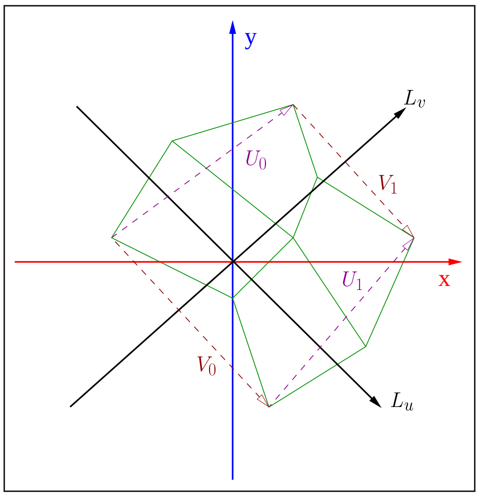


Figure 4.5: Bézier clipping: determining L_u and L_v.

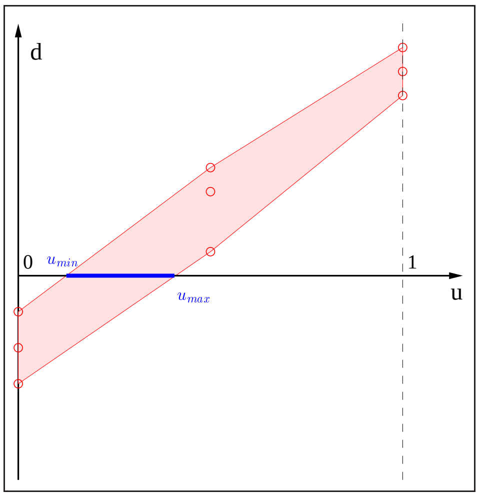


Figure 4.6: Bézier clipping: step in u-dimension.

Step 4. The de Casteljau subdivision algorithm is applied to clip away those regions, leaving the $2D$ patch which is corresponded to the interval $[u_{min}, u_{max}]$ ($[v_{min}, v_{max}]$). This is referred to as the *Bézier clipping* in u (v).

Nishita [29] proposed to precompute the lines L_u and L_v just once before iteration process. But for some surfaces (especially for ones of complex shape) recomputation of L_u and L_v on each iteration using the obtaining clipped patch can give the better speed performance. Moreover, if L_u and L_v are computed just once before iterations and not recomputed on each iteration, Bézier clipping algorithm can experience convergence problems. Therefore, it is recommended to always recompute the lines L_u and L_v on each iteration of Bézier clipping algorithm.

Note that $[u_{min}, u_{max}]$ and $[v_{min}, v_{max}]$ are the regions of interest in the domain $[0, 1]$ in u and v directions respectively, i.e., are normalized. In order to calculate the corresponding regions in the domain of the patch, we need to apply the following transformation

$$\begin{aligned} u_{min,d} &= u_{min,o} + (u_{max,o} - u_{min,o}) \cdot u_{min} \\ u_{max,d} &= u_{min,o} + (u_{max,o} - u_{min,o}) \cdot u_{max} \end{aligned} \tag{4.20}$$

where $[u_{min,o}, u_{max,o}]$ is the parameter domain of the patch in u direction and $[u_{min,d}, u_{max,d}]$ is the region of interest in the domain of the patch. The v parameter direction is handled analogously.

Now we need to switch directions ($L_u \to L_v \to L_u \to \cdots$) and repeat steps **1** to **4**. The parameter values of the intersection point (if intersection exists) must be within the box $[u_{min,d}, u_{max,d}] \times [v_{min,d}, v_{max,d}]$. Figure 4.7 shows the projected surface $P(u, v)$ after two clipping iterations. The algorithm terminates in two cases:

1. the convex hull on the corresponding iteration does **not** intersect with the axis – this indicates that there is **no** intersection of $S(u, v)$ with the ray $R(t)$;

2. the size of the box in each dimension is smaller than a threshold value ε – in this case the intersection $[u_{int}, v_{int}]$ is assumed to exist in the center of the box:

$$\begin{aligned} u_{int} &= \frac{1}{2}\left(u_{min,d} + u_{max,d}\right) \\ v_{int} &= \frac{1}{2}\left(v_{min,d} + v_{max,d}\right) \end{aligned} \tag{4.21}$$

   Note that if the size of the box in one dimension is smaller than ε, this dimension is not considered any more, i.e., only the second direction is proceeded iteratively without alternation. Note also that in this case it is not absolutely necessary to apply Bézier clipping in this direction, and the second direction may be considered either with the same patch segment or with the clipped one.

In order to obtain the distance t from the ray origin to the point of intersection, one needs to apply the following projection

$$t = \vec{d} \cdot (\vec{S}(u_{int}, v_{int}) - \vec{o}) \tag{4.22}$$

If the value of t is less than 0, the intersection is assumed to exist behind the ray, i.e., the ray is assumed to not intersect the surface.

If there are multiple intersections, Bézier clipping does not converge to a single value. Therefore, if a Bézier clip fails to reduce the parameter interval width by at least 20%, we have to split the projected surface $P(u, v)$ in half in the dimension of current iteration. The case of multiple intersections is illustrated in Figure 4.8. First, Bézier clipping in u discards regions 1. In attempting to clip in v, it turns out that $v_{max} - v_{min} > 0.8$. Therefore, the remaining domain is subdivided in half at $v = 0.5$. Two subpatches can be proceed iteratively. The regions 2 of the first subpatch are discarded after one more iteration in u domain. Without further subdivision we can compute the intersection which lies between regions 3 within tolerance. Two more iterations are needed to proceed the second subpatch. After clipping away the regions 4 and 5 one more intersection which lies between regions 6 within tolerance can be computed without further subdivision.

4.3.4 Detected Problems and Proposed Modifications

Sometimes the original Bézier clipping algorithm can converge to wrong intersections. Campanga and Slusallek [9] reported this problem and proposed necessary modifications of the original algorithm in order to make it robust. Figure 4.9 shows an example of such wrong intersections, which are visible as dots near the object. Figure 4.10 shows a closeup view of the wrong intersections. These wrong intersections are not caused by numerical problems, but are a principal problem of the original algorithm, which can be demonstrated by a simple example. Suppose that after projecting the initial surface

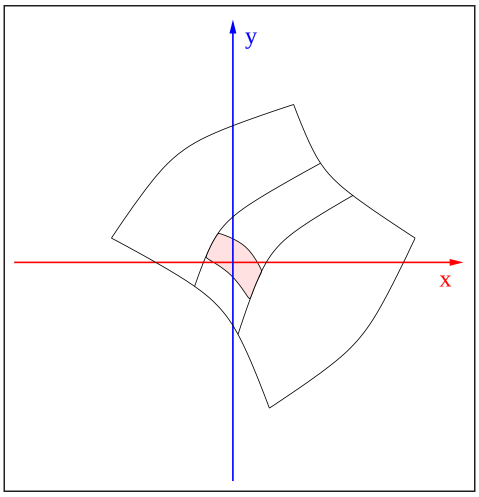


Figure 4.7: Bézier clipping: after two clipping iterations.

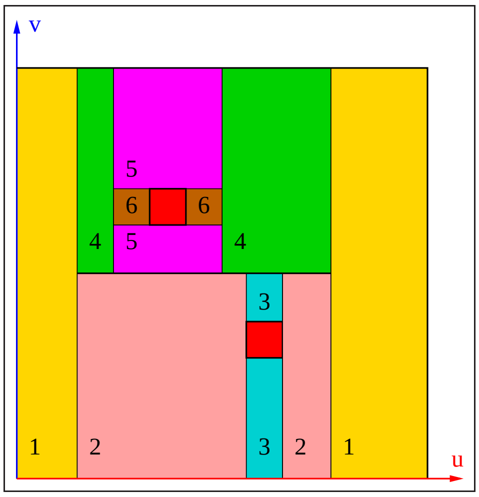


Figure 4.8: Bézier clipping: multiple intersections example.

control mesh in $2D$, we obtain the projected control mesh which is shown in Figure 4.11. One of the control points lies slightly above the line L_u and the others, below.

On the first iteration we consider the u parameter domain. Side view of the distances of the control points to the line L_u together with their convex hull and the region of interest (indicated by two closely spaced dotted lines) is shown in Figure 4.12. Note that the region of interest is within tolerance ε. This means that the value u_{int} of probable intersection is computed, and we have to consider iteratively the v parameter domain only. This can be done in two ways:

1. **with** preliminary Bézier clipping in u parameter direction;
2. **without** preliminary Bézier clipping in u parameter direction.

The first case is shown in Figure 4.13. The corresponding side view of the distances of the control points to the line L_v together with their convex hull and region of interest is shown in Figure 4.14. The second case is shown in Figure 4.15. The corresponding side view of the distances of the control points to the line L_v together with their convex hull and region of interest is shown in Figure 4.16. In both cases, the region of interest for v parameter direction is within tolerance ε, too, and the value of intersection v_{int} can be computed in v parameter direction. Thus, the intersection point is assumed to have coordinates (u_{int}, v_{int}) in parameter domain of the surface. However, neither the original surface nor its control polyhedron contain the origin and thus are **not** intersected by the ray. Therefore, a wrong intersection is reported.

The reason for the wrong intersections is in the fact, that both lines L_u and L_v may intersect the projected surface itself, but these intersections need not contain the

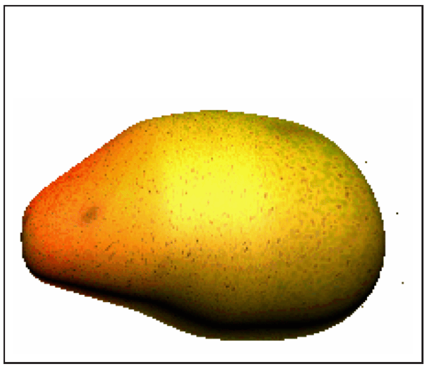


Figure 4.9: Mango with wrongly reported intersection (adapted from [9]).

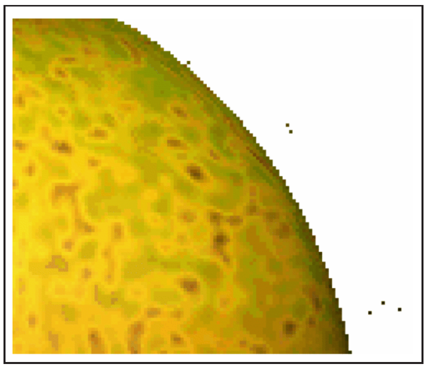


Figure 4.10: Closeup view of the wrongly reported interection (adapted from [9]).

origin, and thus there is no intersection of the ray with the surface.

Campanga and Slusallek [9] proposed to solve the problem of wrong intersections by requiring in addition to the stopping criterion ε, that the line L_u separates the two boundary curves in v direction of the patch. This can be checked by testing that all distances $d_{00}, \ldots, d_{0m}$ (see Section 4.3.3) have the same sign s_0, all distances $d_{n0}, \ldots, d_{nm}$ have the same sign s_n, and that $s_0 \neq s_n$. This ensures that the current patch has an intersection with L_u for every valid v parameter. In an obviously similar manner, we can apply this modification for v parameter direction. As the result, only real intersections of the ray with the surface are reported.

Another solution of the problem is to appropriately enlarge the region of interest, which may force additional iterations. Nishita [29] proposes the following correction of the region of interest before transforming it into the parameter domain of the patch, in order to avoid numerical round-off:

$$\begin{aligned} u'_{min} &= 0.99 \cdot u_{min} \\ u'_{max} &= 0.99 \cdot u_{max} + 0.01 \end{aligned} \tag{4.23}$$

with $[u_{min}, u_{max}]$ being the region of interest in the domain $[0, 1]$. The v parameter direction is handled analogously.

Campanga and Slusallek [9] mentioned that enlarging the region of interest is a sensitive point of the algorithm affecting the performance. They propose the following subtle modification of the region of interest already in the domain of the patch instead of the Nishita's [29] modification:

$$\begin{aligned} u'_{min,d} &= u_{min,d} - (u_{min,d} - u_{min,o}) \cdot f \cdot \varepsilon \\ u'_{max,d} &= u_{max,d} + (u_{max,o} - u_{max,d}) \cdot f \cdot \varepsilon \end{aligned} \tag{4.24}$$

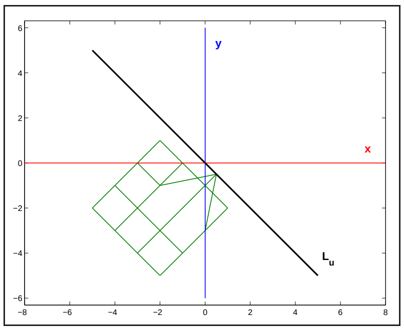


Figure 4.11: The control mesh of the surface after reduction (adapted from [9]).

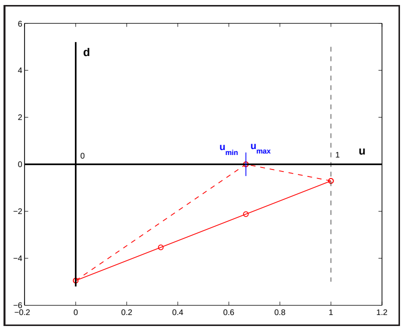


Figure 4.12: Side view of the distances to the line L_u (adapted from [9]).

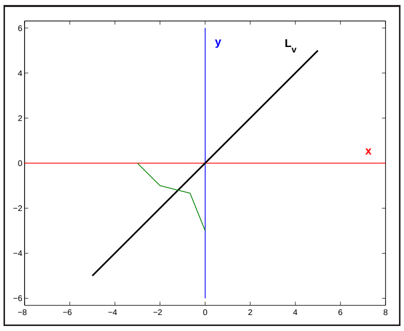


Figure 4.13: Case 1: the control mesh of the patch after the first iteration (adapted from [9]).

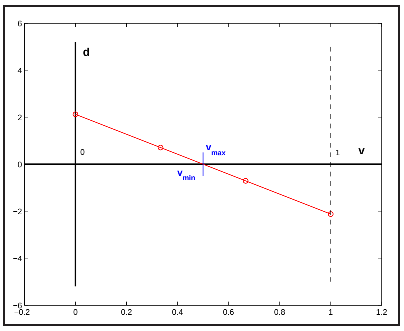


Figure 4.14: Case 1: side view of the distances to the line L_v (adapted from [9]).

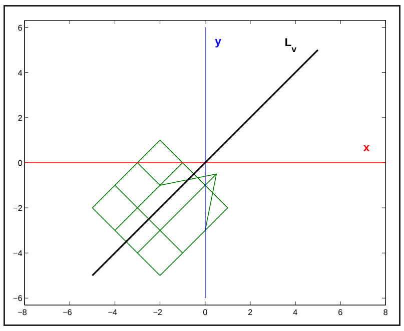


Figure 4.15: Case 2: the control mesh of the patch after the first iteration (adapted from [9]).

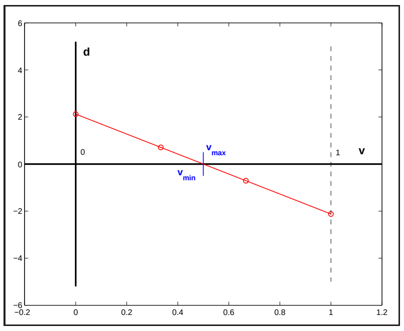


Figure 4.16: Case 2: side view of the distances to the line L_v (adapted from [9])

where $[u_{min,o}, u_{max,o}]$ is the parameter domain for the u direction of the patch segment just considered, and $[u_{min,d}, u_{max,d}]$ is the region of interest in this domain. This modification makes the enlarging of the region of interest depending on the size of the parameter domain and ε value. This leads to smaller enlargements, and it significantly reduces the number of iterations. Modifications for the v parameter direction are analogous.

4.3.5 Multiple Equivalent Intersections Problem

The Bézier clipping algorithm has a problem which was not detected by any of researchers so far. It is a problem of *multiple the same intersections*: sometimes the algorithm can result in thousands of intersections which are in fact equivalent. Let us consider an example in Figure 4.17.

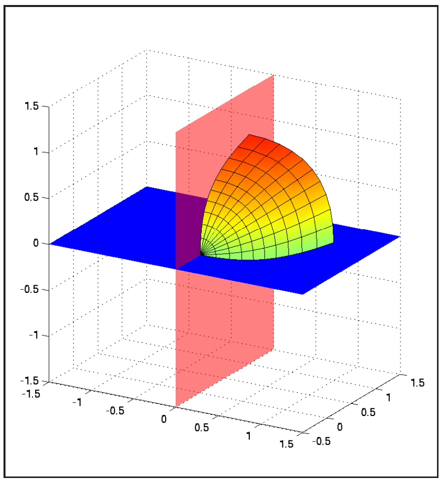


Figure 4.17: Ray-sphere patch intersection.

Suppose that a ray is being tested against a sphere patch which is represented by a rational Bézier patch and has the following mesh of control points and weights:

$$p_{00} = (0,0,0),\ p_{01} = (0,0,1),\ p_{02} = (0,1,1);\ w_{00} = 1,\ w_{01} = \frac{\sqrt{2}}{2},\ w_{02} = 1;$$
$$p_{10} = (0,0,0),\ p_{11} = (1,0,1),\ p_{12} = (1,1,1);\ w_{10} = 1,\ w_{11} = \frac{\sqrt{2}}{2},\ w_{12} = 1;$$
$$p_{20} = (0,0,0),\ p_{21} = (1,0,0),\ p_{22} = (1,1,0);\ w_{20} = 1,\ w_{21} = \frac{\sqrt{2}}{2},\ w_{22} = 1; \quad (4.25)$$

The ray is represented as an intersection of two planes which is also shown in Figure 4.17. After projecting this problem in $2D$ we obtain a projection which is shown in Figure 4.18. Suppose that the initial rational Bézier surface was defined on the

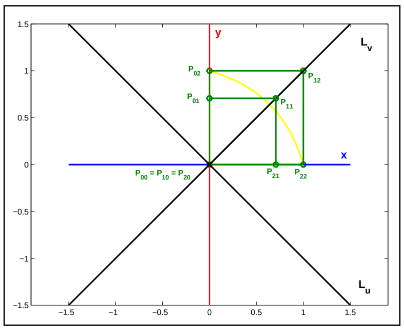


Figure 4.18: The control mesh of the sphere patch after reduction.

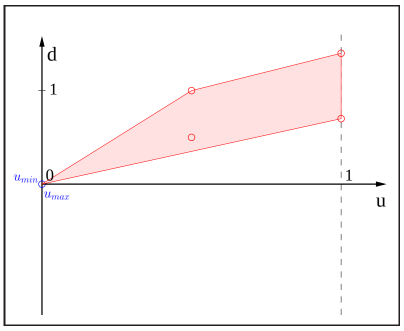


Figure 4.19: Side view of the distances to the line L_u.

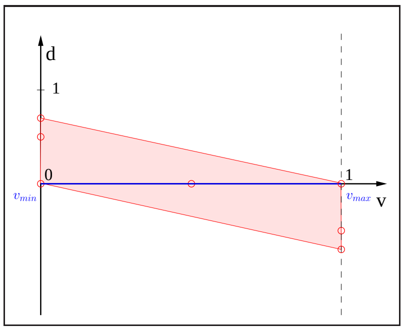


Figure 4.20: Side view of the distances to the line L_v.

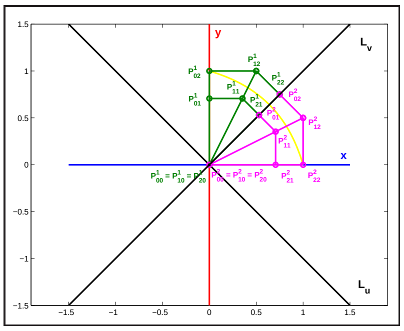


Figure 4.21: Subdivision of the reduced sphere patch by half in v direction.

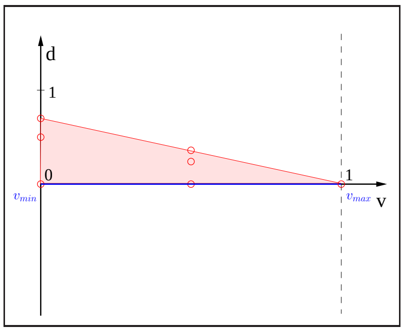


Figure 4.22: Side view of the distances to the line L_v for the first subpatch.

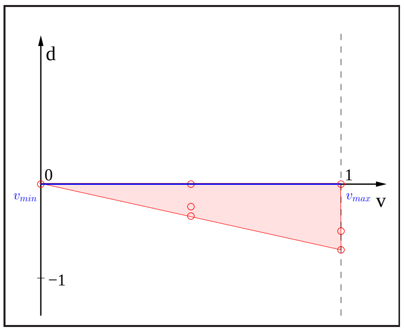


Figure 4.23: Side view of the distances to the line L_v for the second subpatch.

interval $[0,1] \times [0,1]$. Then after determining distances from control points to line L_u we obtain the convex hull which is shown in Figure 4.19. Apparently the area of interest (intersection with the u-axis) is less than ε. This means that we stop to consider the u direction and continue to consider iteratively the v direction only. If we now subdivide the initial Bézier patch in u direction at $u_{int} = \varepsilon$, we obtain a patch that is too small (actually it is almost one point). So, it would be good idea to stop the algorithm at this point and report the intersection because the obtained patch is already negligible with respect to the scene $3D$ space. But the algorithm does not have any stopping criteria like this and the next iteration is executed.

Notice that next iterations causes the multiple the same intersections problem. Suppose for simplicity that after obtaining $u_{int} = \varepsilon$ we do not subdivide the initial Bézier patch and consider the same one in the v direction. If we subdivide the patch, the the scale of figures is much smaller, but the problem is not resolved. After constructing the convex hull in v direction (see Figure 4.20) we see, that the size of the v axis span is almost equal to 1. So, the multiple intersections are supposed to exist and the initial patch is subdivided by half in v direction as shown in Figure 4.21. One subpatch is now defined on the v interval $\left[0, \frac{1}{2}\right]$ and the another one is defined on the v interval $\left[\frac{1}{2}, 1\right]$

After the subdivision each half of the patch is considered separately in v direction only. But Figures 4.22 and 4.23 show that after the convex hulls construction the problem remains. So, the both halves of the patch are assumed to have multiple intersections too and are subdivided once more by half in v direction. This gives us four subpatches to be considered which are defined on v intervals $\left[0, \frac{1}{4}\right]$, $\left[\frac{1}{4}, \frac{1}{2}\right]$, $\left[\frac{1}{2}, \frac{3}{4}\right]$, and $\left[\frac{3}{4}, 1\right]$.

This process of subdivision continues many times until the size of the parametric v direction of each subpatch is too small (less than ε). As the result many intersections are reported which are actually the same intersection.

This shows that after termination of the Bézier algorithm one needs to **filter** the obtained intersection list and delete those intersections which duplicate another ones. In order to do it efficiently, we can store the intersections in a sorted list (which is sorted by distances to the ray origin). When a new intersection is found, we locate a position in the sorted list where it has to be inserted and compare the new intersection with two (at most) neighbor intersections. If the difference in distances is less than ε value, the new intersection is rejected. If the difference in distances is greater then ε value, the new intersection is inserted in the sorted list. This approach can be implemented using the following framework:

Algorithm 4.1 (Processing a new intersection)

```
void PROCESS_INTS(ints_list, ints)
{
   /*  Process new intersection  */
   /*
     Input: @ints_list - the list of all found intersections;
       @ints - a new intersection to be processed;
     Output: modified (is necessary) @ints_list;
   */
```

```
  pos = ints_list.Locate(ints);
  if (pos > 0)
    if (abs(ints - ints_list[pos - 1]) < EPSILON)
      return;
  if (pos < ints_list.size())
    if (abs(ints - ints_list[pos]) < EPSILON)
      return;
  ints_list.Insert(pos, ints);
}
```

We have found out that the problem of multiple equivalent intersections occurs often in practice and has been experienced on many different scene models. Notice that depending on ε value one may compute thousands of unnecessary intersections. It can significantly increase the computation time.

Although the problem of unnecessary multiple intersections can not be avoided in general, using termination criteria which depends on $3D$ coordinates of a patch can highly decrease the number of such intersections.

4.3.6 Efficient Choice for Termination Criteria ε

Using $3D$ space related ε value for termination criteria can optimize the performance of the algorithm and speed up the computations. This idea is explained briefly by a simple example in Figure 4.24. Suppose we are given two objects: *object A* and *object*

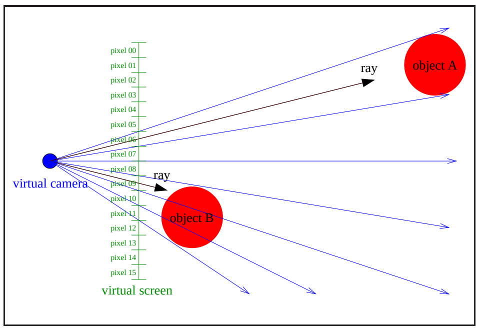


Figure 4.24: Difference in necessary precision of computations.

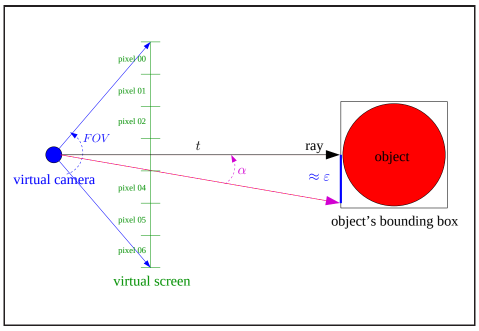


Figure 4.25: An efficient ε computation.

B of the same shape (represented by Bézier patches). It is easy to see that the *object A* covers just one pixel on the screen (namely *pixel 06*), and *object B* covers three pixels on the screen (namely *pixel 09 – pixel 11*). It is obviously that the intersection of a ray with the *object A* can be calculated with three times less precision than the same intersection with the *object B*. Concerning the *object A* we can take any intersection because all of them are projected on the same pixel of the virtual screen.

This shows that in order to optimize the performance of the Bézier clipping algorithm, the termination criteria ε should be dependent on the $3D$ coordinates of objects, position of camera, and the size of screen pixels. The termination of the algorithm dependent on threshold value in parametric domain can overestimate or underestimate the precision.

An efficient strategy of determining $3D$ space related ε value as the termination criteria is explained in more details. Figure 4.25 shows the idea of the efficient ε computation. Suppose t is the distance from the ray origin to the point of intersection with the object's bounding box, FOV is the camera *field of view* parameter, and x and y are the image width and height respectively. Let us compute approximate angle between the ray which goes through the center of a pixel and the ray which goes through the pixel's border by equation

$$\alpha = FOV/(2 \cdot max\{x, y\}) \tag{4.26}$$

Then the ε value can be computed by equation

$$\varepsilon = k \cdot t \cdot \sin \alpha \tag{4.27}$$

Notice that the coefficient $\sin \alpha$ can be precomputed on the preprocessing step. The coefficient k is used to control the exact accuracy and can be set to value from the interval $(0, 1]$. When antialiasing techniques are not involved then this value can be set to 0.5. If antialiasing techniques are involved, the value of k should be set relatively to the number of rays which are shooted through one pixel. One can use the following equation:

$$k = 0.5/n, \tag{4.28}$$

where n is the number of rays which are shooted through one pixel.

This approach works well if we consider non rational Bézier patches (ones which have all control points weights equal to 1). Termination criteria ε for Bézier clipping algorithm must correspond to projected patch where all distances are weighted by control points weight values. If we do not take these values into account, we can again oversestimate or underestimate the precision. To avoid this problem we must multiply the epsilon value calculated by Equation (4.27) by the factor of the minimal weight value of the control mesh of a patch:

$$\varepsilon = k \cdot t \cdot \sin \alpha \cdot min\{w_{ij}\} \tag{4.29}$$

The proposed approach make the ε value more intelligent and optimizes the termination criteria of Bézier clipping algorithm. The problem of multiple the same intersections still exists but the number of such intersections is significantly reduced.

The value of ε can be also chosen efficiently when solving a ray-rational Bézier curve in $2D$ intersection problem. Actually we are not interested in general form of this problem within the contex of this Master Thesis. We are interested in the intersections with curves which determine trimming regions in parametric space of NURBS surfaces because the determination of the trimming regions is based on the ray-curve intersection tests.

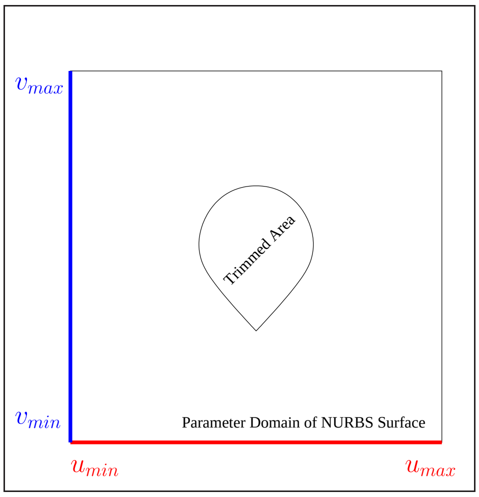


Figure 4.26: Trimming region in parameter space of NURBS surface.

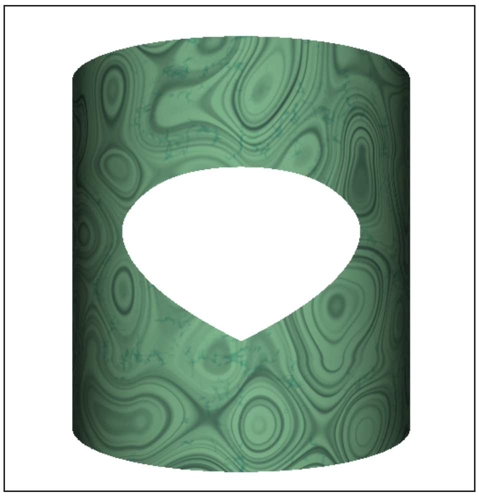


Figure 4.27: Trimmed NURBS surface.

Figure 4.26 shows the parametric domain of a NURBS surface with a trimming region, which is defined by a rational Bézier curve. Figure 4.27 shows the corresponding trimmed NURBS surface. If Bézier clipping is applied in order to solve ray-rational Bézier curve intersection problem, the threshold value ε can be calculated using the following equation

$$\varepsilon = 10^{-k} \cdot min\{(u_{max} - u_{min}), (v_{max} - v_{min})\} \cdot min\{w_i\}, \tag{4.30}$$

where k is the precision parameter. During experiments we have found out that $k = 4$ is a good choice. Minimum weight value of the control polygon points of a curve is taken into account to avoid problems of wrong precision which were mentioned above. Problem of determining trimmed regions in parametric space of NURBS surfaces are explained in more details in Section 5.5.

The original Bézier clipping algorithm must be improved in order to work properly with the efficient $3D$ space oriented ε value. In order to improve the original Bézier clipping algorithm, we have to make the following modifications:

- on each iteration after constructing a convex hull we determine minimum and maximum distances d_{min} and d_{max} from the convex hull points to lines L_u or L_v depending on the current iteration as shown in Figure 4.28.
- we stop to consider the corresponding direction if the size of the distance span in this direction is less than ε value:

$$(d_{max} - d_{min}) < \varepsilon \tag{4.31}$$

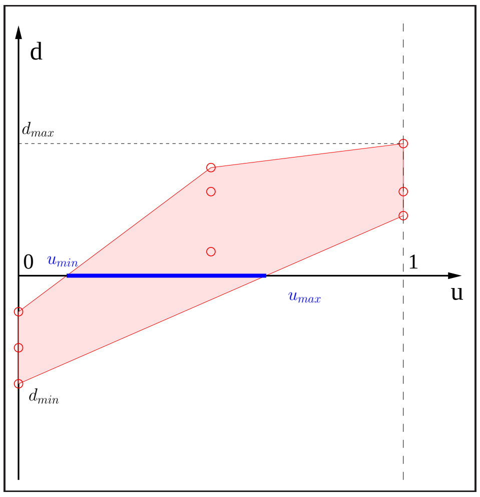


Figure 4.28: Determining the distance span.

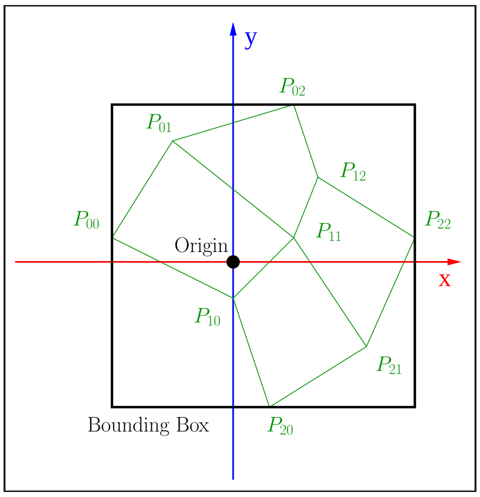


Figure 4.29: Final test.

In this case we do not perform the Bézier clipping and consider the second direction with the same subpatch (or report the intersection if the second direction consideration has already been stopped). Otherwise we perform the Bézier clipping and continue the algorithm execution.

- notice that problems detected by Campanga and Slusallek [9] which were described in Section 4.3.5 can not happen with the modified version of Bézier clipping algorithm, because the algorithm is terminated depending on the size of the distance span for both lines L_u and L_v. Notice also that these lines are not orthogonal and can be of any directions. For some patches they can even coincide. It may cause reporting of wrong intersections because obtained subpatch can lie far away from the point of origin. In order to prevent this problem, we need to execute a *final test*: if ε criteria is held for both u and v directions, we construct the bounding box of the current subpatch and test whether the point of origin $(0, 0)$ is inside this bounding box. If it is not inside, we report no intersections. This idea is shown in Figure 4.29. It can also happen that the origin point is inside the bounding box, but the initial ray misses the enclosed patch. Even in this case we report intersection because the size of the box is negligible (with respect to ε value). Notice that the final test can be performed without any arithmetic operations and is therefore fast.

Analogous modifications can be easily applied in order to solve ray-rational Bézier curve in $2D$ intersection problem. Notice that the termination criteria for curves must be slightly different. One can not just compare the distance span and ε because it might give wrong intersections when long curve has too small distance span with respect to the ray. One can use the diagonal of the bounding box of the current curve segment instead of the distance span. If the diagonal is smaller that ε, the termination criteria holds.

All described improvements make the Bézier clipping algorithm one of the most practical, fast, and robust algorithms for solving ray-rational Bézier surface intersection problem. Interested reader can refer for more details to [29, 9].

4.4 Newton's Iteration Approach

4.4.1 Introduction

If the exact value of the root of a function can not be computed deterministically then Newton's iteration method is one of the best choices to find the approximate value of the root. More general application of this method is solving a system of equations. Newton's iteration method has *quadratic convergence* and can be easily apply to the problem of finding ray-surface intersection.

In Section 4.4.2 basic ideas behind the method are explained. In Section 4.4.3 Newton's iteration method is applied in order to solve ray-rational Bézier curve in $2D$ intersection problem. Section 4.4.4 shows how one can apply Newton's iteration method in order to solve more general ray-rational Bézier surface intersection problem.

4.4.2 Basics Behind

Suppose we have explicitly given function $y = f(x)$ and we are looking for the roots of such function, i.e., we are solving the equation $f(x) = 0$. If it is a quadratic polynomial function $y = ax^2 + bx + c$, roots are easy to determine using the quadratic formula. In fact, it turns out that there are formulas similar to the quadratic formula for finding roots of third and fourth degree polynomials, however, these formulas are rarely used in practice because they are quite complicated. For fifth and higher degree polynomials a general formula for finding roots of equations does not exist. Some methods of approximating roots of equations have to be applied to such functions.

Let us consider a real-valued function $f(x)$ of a single variable whose roots can not be found deterministically. Let us assume for simplicity that this function has only one root x^* and we have some guess about the interval where this root can be. *Parallel-chord method* consists of taking so called guess value x_0 from the guess interval and replacing $f(x)$ at this value by some linear function:

$$l(x) = \alpha(x - x_0) + f(x_0) \tag{4.32}$$

Notice that $l(x)$ is a line which goes through the point $(x_0, f(x_0))$ and has the slope $\alpha \neq 0$. Now, if we solve the equation $l(x) = 0$, we obtain a value x_1 at which the line intersects the x-axis:

$$x_1 = x_0 - \frac{1}{\alpha} f(x_0) \tag{4.33}$$

If the slope α is chosen appropriately, the obtained value x_1 is **closer** to the root than the initial x_0 value. It turns out that the optimal choice for the slope α is the first derivative of $f(x)$, i.e., $f'(x)$. Actually this means that $l(x)$ is the *tangent-line* of $f(x)$ at the point x_0. The value x_1 in this case is equal to

$$x_1 = x_0 - f'(x_0)^{-1} f(x_0) \tag{4.34}$$

If we now repeat this process iteratively, then we obtain the general equation

$$x_i = x_{i-1} - f'(x_0)^{-1} f(x_{i-1}) \tag{4.35}$$

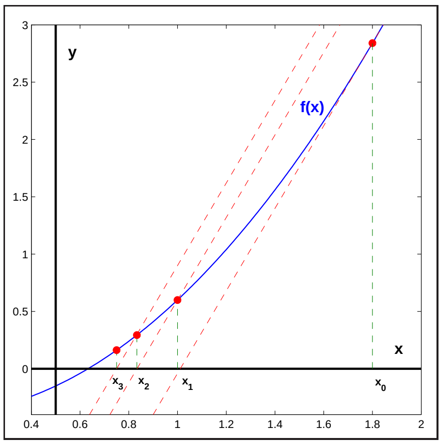


Figure 4.30: Parallel-chord method.

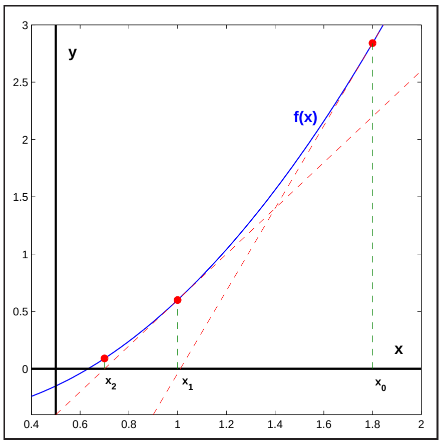


Figure 4.31: Newton's iteration method.

The idea of this method is shown in Figure 4.30. If each new x_i is closer to the root than the value on the previous iteration x_{i-1} (if $|f(x_i)| < |f(x_{i-1}|)$, this process leads to the approximate value of the root x_k on the k^{th} iteration, which lies near the real root x^* within tolerance ε.

$$|f(x_k)| < \varepsilon < |f(x_{k-1})| < \cdots < |f(x_1)| < |f(x_0)| \tag{4.36}$$

Notice that this process has *linear convergence* and guaranties to find the approximate value of the root only if the initial guess x_0 was *good enough*.

If the slope of the line $l(x)$ is changed accordingly to the value of x_i on each iteration, the above process has *quadratic convergence* and the general equation takes the form

$$x_i = x_{i-1} - f'(x_{i-1})^{-1} f(x_{i-1}) \tag{4.37}$$

This process is shown in Figure 4.31 and is known as *Newton's iteration method.* Newton's iteration method is easy to extend to n-dimensional function F: $R^n \to R^n$.

Suppose we are given a set $Y = F(\vec{x})$ of n equations in n variables $x_1, \ldots, x_n$, written explicitly as

$$Y \equiv \begin{bmatrix} f_1(\vec{x}) \\ f_2(\vec{x}) \\ \vdots \\ f_n(\vec{x}) \end{bmatrix} \tag{4.38}$$

or more explicitly as

$$\begin{cases} y_1 = f_1(x_1, \ldots, x_n) \\ \vdots \\ y_n = f_n(x_1, \ldots, x_n) \end{cases} \tag{4.39}$$

Then the solution of this system of equations can be approximately computed using Newton's iteration method

$$\vec{x}_i = \vec{x}_{i-1} - J(\vec{x}_{i-1})^{-1} F(\vec{x}_{i-1}) \tag{4.40}$$

where J is the *Jacobian matrix* which is defined by

$$J(\vec{x}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_n} \end{bmatrix} \tag{4.41}$$

This process also has *quadratic convergence* and guaranties to find the approximate solution to the system of equations only if the initial guess vector $\vec{x}_0$ is *good enough*. Notice that on each iteration of the Equation (4.40) we need to find and invert the Jacobian matrix. Matrix inversion is relatively costly operation. If *linear convergence* is enough for solving the problem, the Equation (4.40) can be rewritten by analogy with real-valued functions:

$$\vec{x}_i = \vec{x}_{i-1} - J(\vec{x}_0)^{-1} F(\vec{x}_{i-1}) \tag{4.42}$$

Now the Jacobian matrix has to be computed and inverted only on the first iteration. On all other iterations it is treated as constant.

4.4.3 Ray-rational Bézier Curve in $2D$ Intersection Problem

The basic idea of solving ray-rational Bézier curve intersection problem is the same as was discussed in Section 4.3.2. Rational Bézier curve is defined by equation

$$P(u) = \frac{\sum_{i=0}^{n} p_i w_i B_i^n(u)}{\sum_{i=0}^{n} w_i B_i^n(u)}, \quad u \in [0, 1] \tag{4.43}$$

The ray $R(t)$ is represented by its origin $\vec{o}$ and direction $\vec{d}$

$$R(t) = \vec{o} + t\vec{d} \tag{4.44}$$

We define a line which contains the ray by its implicit equation.

$$ax + by + c = 0 \tag{4.45}$$

Coefficients a, b, and c can be easily obtained from the ray origin and direction (see Section 4.3.2). In order to find intersection of the given ray $R(t)$ with the given rational Bézier curve $P(u)$, one needs to substitute Equation (4.43) into Equation (4.45). After the substitution our problem of finding ray-curve intersection becomes one to solve the following equation

$$d(u) = \sum_{i=0}^{n} d_i B_i^n(u) = 0, \qquad d_i = w_i(ap_{x_i} + bp_{y_i} + c) \tag{4.46}$$

Assume that we have some initial guess value u_0 which is somewhere in the vicinity of the root. Then we can easily apply Newton's iteration method in order to find approximate value of the root

$$u_i = u_{i-1} - d'(u_{i-1})^{-1} d(u_{i-1}) \tag{4.47}$$

Note that we assume that Equation (4.46) has just one root, i.e., the ray $R(t)$ has just one intersection with the given curve $P(u)$.

First derivative $d'(u)$ can be computed using de Casteljau algorithm for subdivision (see Section 2.3.3). Although $d(u)$ in Equation (4.46) is not a curve (because control coefficients d_i are real number and not vectors), de Casteljau algorithm for subdivision can be easily applied for evaluation of this function. After performing de Casteljau subdivision at the parameter value u_{i-1} we obtain two sets of control coefficients $\{d_0^1, d_1^1, d_2^1, \ldots, d_{n-1}^1, d_n^1 =\}$ and $\{d_0^2, d_1^2, d_2^2, \ldots, d_{n-1}^2, d_n^2 =\}$ where $d_n^1 = d_0^2$. The first derivative can be then computed using equation

$$d'(u_{i-1}) = n \cdot (d_n^1 - d_{n-1}^1) = n \cdot (d_1^2 - d_0^2) \tag{4.48}$$

Equation (4.47) has the *quadratic convergence*. This means that only few iterations are already enough in order to find the approximate value of the root whose deviation from the real root is within tolerance ε.

In order to find intersection of the given ray $R(t)$ with the given rational Bézier curve $P(u)$, we apply Equation (4.47) iteratively starting with the initial guess value u_0 until one of four cases occurs:

1. $|d(u_i)| > |d(u_{i-1})|$, i.e., the new estimate u_i takes us farther from the root than the previous one – in this case no intersection reported (square distances can be compared in order to avoid costly square root operations);

2. $u_i \notin [0, 1]$, i.e., current iteration takes us outside the parameter interval of the curve – notes are given below;

3. $i > max$, where max is the maximum number of iterations allowed – in this case no intersection reported;

4. $|d(u_i)| < \varepsilon$ – in this case intersection is assumed to exist at $u_{int} = u_i$;

Martin, Cohen, Fish and Shirley [27] propose to report no intersection if the iteration takes us outside the parametric domain (in the 2nd case). But although we suppose the curve to be defined on the interval $[0, 1]$, mathematically it also exists outside this interval. It can happen that on the ith iteration the value of u_i is outside the interval, but intersection exists and is found in few more iterations. Therefore, if the 1st condition does not hold and $u_i \notin [0, 1]$, we **must** just execute the next iteration. Notice that obtained intersection can lie outside the parameter interval $[0, 1]$, i.e., so called *virtual intersection* can be reported. So, one needs always check whether the obtained intersection lies inside the parameter interval of the Bézier curve.

Maximum number of iterations max can be set around 7, but the average number of iterations needed to produce convergence is 2 or 3 in practice.

During the iteration process it can happen that we get $d'(u_{i-1}) < \varepsilon$ on the i^{th} iteration. It can occur if a segment of the curve is parallel to the ray or if the curve is not regular. In this case we perform a jittered perturbation of the parametric evaluation point[1] (see [27])

$$u_i = u_{i-1} + 0.1 \cdot (drand48() \cdot (u_0 - u_{i-1})) \tag{4.49}$$

and initiate the next iteration. Notice that if $d'(u_0) < \varepsilon$, we go into an infinite loop, so one needs to take the initial guess value u_0 carefully.

In order to obtain the distance t from the ray origin to the point of intersection, we apply the following projection

$$t = \vec{d} \cdot (\vec{P}(u_{int}) - \vec{o}) \tag{4.50}$$

If the value of t is less than 0, the intersection is assumed to exist behind the ray, i.e., the ray is assumed to not intersect the curve.

The proposed algorithm uses ε value which is related to $2D$ coordinates of a curve as the termination criteria. In Section 4.3.5 it is shown how ε can be chosen efficiently when solving a problem of determining trimming regions in parametric space of NURBS surfaces.

Note that our solution of the ray-rational Bézier curve intersection problem is based on two assumptions. The first assumption is that there are no multiple intersections of the ray with the curve. The second assumption is that the initial guess value u_0 is taken carefully from the vicinity of the real root. In practice these two assumptions are not hold and one have to involve a technique to handle multiple intersections and to obtain *good enough* initial guess value u_0.

If multiple intersections exist then Newton's iteration method may converge to any of them depending on the initial guess value u_0. It can also happen that it does not converge. In practical ray tracing applications we are mostly interested in the nearest intersection point, because all other intersection points can be computed by moving the origin of the ray to the obtained nearest intersection point and repeating nearest intersection search again. This means that everything what we need is the initial guess

[1]C++ function drand48() returns uniformly distributed random real number from the interval $[0, 1)$

value u_0 which guarantees the convergence of the Newton's iteration method to the nearest intersection point. The algorithm for determining such initial guess value is proposed in Section 5.4.3 for more complex ray-rational Bézier surface intersection problem. Its application in $2D$ is straightforward.

4.4.4 Ray-rational Bézier Surface Intersection Problem

The basic idea of solving ray-rational Bézier surface intersection problem is the same as was discussed in Section 4.4.3. Rational Bézier surface is defined by equation

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)w_{i,j}p_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)w_{i,j}}, \quad 0 \le u,v \le 1 \tag{4.51}$$

and a ray $R(t)$ defined by its parametric equation

$$R(t) = \vec{o} + t\vec{d} \tag{4.52}$$

where $\vec{o}$ is the ray origin and $\vec{d}$ is the ray direction. We change the representation of the ray and represent it as an intersection of two planes given by equations

$$\begin{aligned} a_1x + b_1y + c_1z + d_1 &= 0 \\ a_2x + b_2y + c_2z + d_2 &= 0 \end{aligned} \tag{4.53}$$

where coefficients $a_1, b_1, c_1, d_1, a_2, b_2, c_2$, and d_2 can be computed using the ray origin $\vec{o}$ and the ray direction $\vec{d}$ vectors. After substituting Equation (4.51) into Equation (4.53) the intersection of the planes with the surface can be represented by equation

$$\begin{aligned} d^k(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)d_{ij}^k = 0 \\ \text{where } d_{ij}^k = w_{ij}(a_kp_{x_{ij}} + b_kp_{y_{ij}} + c_kp_{z_{ij}}), \quad k = [1,2] \end{aligned} \tag{4.54}$$

Now the problem of finding the intersection of the ray with the surface becomes one of finding the roots of the function $D(u,v)$: $R^2 \to R^2$:

$$D(u,v) = \begin{bmatrix} d^1(u,v) \\ d^2(u,v) \end{bmatrix} \tag{4.55}$$

Assume that we have some initial guess vector (u_0, v_0) which is somewhere in the vicinity of the root. Multidimensional version of Newton's iteration method can be then applied in order to find approximate solution of this problem:

$$\begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} u_{i-1} \\ v_{i-1} \end{pmatrix} - J(u_{i-1}, v_{i-1})^{-1} D(u_{i-1}, v_{i-1}) \tag{4.56}$$

where Jacobian matrix J is defined by

$$J = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix} = \begin{bmatrix} \frac{\partial d^1}{\partial u} & \frac{\partial d^1}{\partial v} \\ \frac{\partial d^2}{\partial u} & \frac{\partial d^2}{\partial v} \end{bmatrix} \tag{4.57}$$

The inverse of the Jacobian matrix is calculated using a result from linear algebra

$$J^{-1} = \frac{adj(J)}{det(J)} \tag{4.58}$$

The adjoint $adj(J)$ is equal to the transpose of the cofactor matrix

$$adj(J) = C^T = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}^T \tag{4.59}$$

where $C_{ij} = (-1)^{i+j} \cdot det(\hat{J}_{ij})$ and $\hat{J}_{ij}$ is the submatrix of J which remains when the ith row and jth column are removed. Finally we have

$$adj(J) = \begin{bmatrix} J_{22} & -J_{12} \\ -J_{21} & J_{11} \end{bmatrix}, \qquad det(J) = J_{11}J_{22} - J_{21}J_{12} \tag{4.60}$$

Notice that point $D(u,v)$ and two partial derivatives $D_u(u,v)$ and $D_v(u,v)$ are evaluated simultaneously using de Casteljau algorithm for subdivision. Notice also that inversion of 2×2 Jacobian matrix can be done by perturbation of its elements with only few additional multiplications and additions operations. It makes a single Newton's iteration fast.

In order to find intersection of the given ray $R(t)$ with the given rational Bézier surface $S(u,v)$, we apply Equation (4.56) iteratively starting with the initial guess vector (u_0, v_0) until one of the four cases occurs:

1. $\|D(u_i, v_i)\| > \|D(u_{i-1}, v_{i-1})\|$, i.e., the new estimate (u_i, v_i) takes us farther from the root than the previous one – in this case no intersection reported (square distances can be compared in order to avoid costly square root operations);

2. $u_i \notin [0,1]$ or $v_i \notin [0,1]$, i.e., current iteration takes us outside the parametric domain of the surface – notes are given below;

3. $i > max$, where max is the maximum number of iterations allowed – in this case no intersection reported;

4. $\|D(u_i, v_i)\| < \varepsilon$ – in this case intersection is assumed to exist at $\binom{u_{int}}{v_{int}} = \binom{u_i}{v_i}$;

Martin, Cohen, Fish, and Shirley [27] propose to report no intersection if $u_i \notin [0,1]$ or $v_i \notin [0,1]$ (in the 2nd case). But although we suppose the surface to be defined on the domain $[0,1] \times [0,1]$, mathematically it exists outside this domain too. It can happen that on the ith iteration the value of u_i or v_i is outside the domain but intersection exists and may be found in few more iterations. Therefore, if the first condition does not hold and $u_i \notin [0,1]$ or $v_i \notin [0,1]$, we **have to** execute the next iteration. Notice that obtained intersection can lie outside the parameter domain interval $[0,1] \times [0,1]$, i.e., so called *virtual intersection* can be reported. So, one needs always check whether the obtained intersection lies inside the parametric domain of the rational Bézier surface.

The problem which arises here is in the following fact. Sometimes virtual intersection (which lies outside the parameter domain of the surface) can be reported, although the real intersection is not virtual (lies inside the parameter domain). It happens often near the borders of the surface and can cause cracks along the borders of adjacent surface patches. This problem happens because the termination criteria ε can terminate the algorithm execution earlier than it converges to the real root tightly. This problem can be solved as follows. If the obtained intersection lies outside the parameter interval of the surface, we mirror it in order to place it inside the parameter domain:

$$u_{mir} = \begin{cases} -u_i & \text{if } u_i < 0 \\ 2 - u_i & \text{if } u_i > 1 \end{cases}$$

v direction is handled analogously. Now, if $\|D(u_{mir}, v_{mir})\| < \varepsilon$, we report (u_{mir}, v_{mir}) as the found intersection. Otherwise the obtained intersection is really the virtual one.

In order to avoid costly square root operation when determining $\|D(u_i, v_i)\| = \sqrt{D_x^2(u_i, v_i) + D_y^2(u_i, v_i)}$, we can compute approximate length instead of exact one using equation:

$$\|D(u_i, v_i)\| \approx |D_x(u_i, v_i)| + |D_y(u_i, v_i)| \tag{4.61}$$

Computed in such a way approximate length is always greater than the exact one, therefore the precision of calculation is not affected.

Maximum number of iterations max can be set around 7, but the average number of iterations needed to produce convergence is 2 or 3 in practice.

During the iteration process it can happen that we get $|det(J)| < \varepsilon$ on the ith iteration. It can occur if either the surface is not regular ($S_u(u_{i-1}, v_{i-1}) \times S_v(u_{i-1}, v_{i-1}) = 0$) or the ray is parallel to a silhouette ray at the point $S(u_{i-1}, v_{i-1})$. In this case we perform a jittered perturbation of the parametric evaluation point[2] (see [27])

$$\begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} u_{i-1} \\ v_{i-1} \end{pmatrix} - 0.1 \cdot \begin{pmatrix} drand48() \cdot (u_0 - u_{i-1}) \\ drand48() \cdot (v_0 - v_{i-1}) \end{pmatrix} \tag{4.62}$$

and initiate the next iteration. Notice that if $|det(J)| < \varepsilon$ on the first iteration, we go into an infinite loop, so one needs to take the initial guess vector (u_0, v_0) carefully.

In order to obtain the distance t from the ray origin to the point of intersection, we apply the following projection

$$t = \vec{d} \cdot (\vec{S}(u_{int}, v_{int}) - \vec{o}) \tag{4.63}$$

If the value of t if less than 0, the intersection is assumed to exist behind the ray, i.e., the ray is assumed to not intersect the surface.

The proposed algorithm uses ε value which is related to a surface $3D$ coordinates as the termination criteria. In Section 4.3.5 it is shown how ε can be chosen efficiently depending on the $3D$ coordinates of objects, camera position, and the size of virtual screen.

As in the case of ray-rational Bézier curve intersection problem our solution of the ray-surface intersection is based on two assumptions: there are no multiple intersections

[2]C++ function drand48() returns uniformly distributed random real number from the interval $[0, 1)$

and the initial guess vector is *good enough*. The problem of determining *good enough* initial guess vector (which guarantees the convergence of Newton's iteration method to the nearest intersection) is opened. The algorithm of determining such initial guess vector is given in Section 5.4.3.

Chapter 5

Practical Ray Tracing Trimmed NURBS Surfaces

5.1 Introduction

Useful geometric properties and a compact representation have made NURBS surfaces widely used in Computer Aided Design. Though the NURBS surface representation is convenient for modeling, a direct ray tracing of NURBS surfaces is time consuming because of the slow NURBS evaluation procedures. It is known, that any NURBS surface can be represented by the number of rational Bézier patches without loosing of accuracy. Section 5.2 outlines basic ideas behind the NURBS to rational Bézier convertion.

The performance of both algorithms for ray tracing Bézier patches presented in Chapter 4 can be improved by associating an object oriented acceleration data structure with each Bézier patch. Section 5.3 describes an acceleration data structure which can be utilized for this purpose.

Section 5.4 shows how the acceleration data structure and algorithms for ray tracing Bézier patches can be combined together in order to yield better performance.

In order to achieve more flexibility in modeling, NURBS surfaces are often assigned with a set of trimming contours which lie in the parameter space of the parent surface and determine regions on the surface which must be cut away. As far as the trimming contours are often defined by NURBS curves and the number of such curves may reach thousands, trimming contours must be organized in a special hierarchical data structure in order to yield better performance. The ideas behind the trimming hierarchy and an efficient trimming test are described in Section 5.5.

Section 5.6 shows how adaptive construction of the acceleration data structure, described in Section 5.3.3 must be updated to deal with trimmed surfaces.

Section 5.7 gives suggestions about the numerical robustness for the algorithms presented in the Master Thesis.

Section 5.8 presents the comparison of approaches for the ray tracing NURBS surfaces described throughout the Master Thesis and summarizes the achieved results.

In Section 5.9 some images rendered with high resolution are presented.

5.2 From NURBS to Rational Bézier Representation

NURBS curves are used to specify trimming regions in parametric space of NURBS surfaces. This approach has become standard in computer aided design because it allows to create surfaces of complex shape. When ray-NURBS surface intersection point is found and the corresponding coordinates in parameter uv space of the NURBS surface are obtained one needs to test whether the obtained point is inside or outside trimming regions. This problem is described in Section 5.5 in more details. What is important now – it is the fact that determining of trimming regions is based on ray-NURBS curve intersection problem.

In Chapter 4 two algorithms for ray tracing[1] rational Bézier curves and surfaces were described in details, namely *Bézier clipping method* and *Newton's iteration method*. Bézier clipping method is based on subdivision and the convex hull property, and Newton's iteration method is based on point and derivatives evaluation. Although NURBS curves (surfaces) also have the convex hull property and can be subdivided using *de Boor algorithm* or *knot insertion*, these algorithms work much slower than simple *de Casteljau algorithm* for Bézier curves (surfaces) subdivision. Point and derivatives evaluation on NURBS curves (surfaces) can be done using different methods (de Boor method, knot insertion, or direct evaluation) but all these methods are slow with respect to the methods of Bézier curves (surfaces) evaluation. Therefore, neither Bézier clipping method nor Newton's iteration method is applied directly to solve ray-NURBS curve (surface) intersection problem.

NURBS representation is generalization of Bézier representation and each NURBS curve (surface) can be represented by a number of joint rational Bézier splines (patches). So, in order to improve performance of ray tracing, it is better to subdivide each NURBS curve (surface) into rational Bézier splines (patches) during preprocessing step and operate with rational Bézier splines (patches) as basic units.

Each NURBS curve (surface) can be subdivided into rational Bézier splines (patches) using multiple knot insertion (*knot refinement*) procedure. Let us consider transformation of NURBS curves into Bézier splines first as simpler example with respect to surfaces transformation. Suppose we have a NURBS curve of degree p in three dimensional homogeneous coordinate space given by equation

$$P^w(u) = \sum_{i=0}^{n} N_i^p(u) p_i^w \qquad a \le u \le b \tag{5.1}$$

The curve is defined on a knot vector

$$U = \{\underbrace{a, \ldots, a}_{p+1}, u_{p+1}, \ldots, u_{r-p-1}, \underbrace{b, \ldots, b}_{p+1}\} \tag{5.2}$$

[1]Although the ray tracing term is related to surfaces only it is also used within this Master Thesis in the context of $2D$ curves.

where $r = n + p + 1$. Notice that the control polygon is formed by $n + 1$ control points. Let us refine knot vector in such a way that each knot has multiplicity $p + 1$:

$$U = \{\underbrace{a, \dots, a}_{p+1}, \underbrace{u_{p+1} = \dots = u_{2p+1}}_{p+1}, \dots, \underbrace{u_{r'-2p-1} = \dots = u_{r'-p-1}}_{p+1}, \underbrace{b, \dots, b}_{p+1}\}$$

where $r' = n' + p + 1$. Notice that after the knot refinement the number of control points increases proportionally to the number of inserted knots. Suppose that $(n' + 1)$ is the number of control points after the knot refinement and $\hat{p}_i^w$ is the new control polygon. Then each interval span $[u_k, u_{k+1}]$ of not coinciding subsequent knots u_k and u_{k+1} represents a rational Bézier spline of the form

$$P^w(u) = \sum_{i=0}^{p} B_i^p(u) b_i$$
$$\text{where } b_i = \hat{p}^w_{k-(p-i)}, \quad u \in [u_k, u_{k+1}] \tag{5.3}$$

Figure 5.1 shows an example of a NURBS curve together with its control polygon. Figure 5.2 shows the corresponding rational Bézier splines and their control polygons obtained after the transformation.

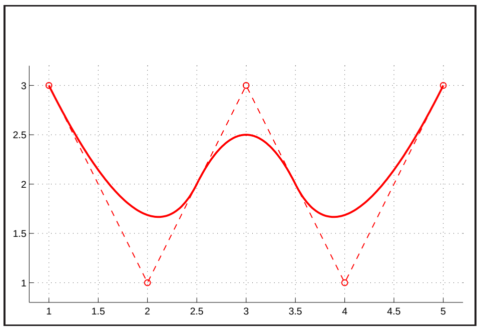


Figure 5.1: Initial NURBS curve.

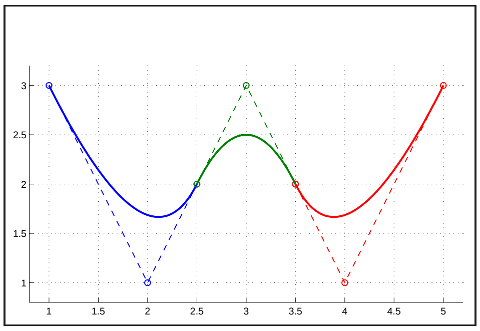


Figure 5.2: Obtained Bézier splines after transformation.

Now let us consider more complex NURBS surface transformation. Suppose we have a NURBS surface of degree p in u direction and degree q in v direction which is represented in four dimensional homogeneous coordinate space by equation

$$S^w(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) p_{i,j}^w \qquad a \le u \le b, \;\; c \le v \le d \tag{5.4}$$

The surface is defined on knot vectors

$$\begin{aligned} U &= \{\underbrace{a,\ldots,a}_{p+1}, u_{p+1},\ldots,u_{r-p-1},\underbrace{b,\ldots,b}_{p+1}\} \\ V &= \{\underbrace{c,\ldots,c}_{q+1}, v_{q+1},\ldots,v_{s-q-1},\underbrace{d,\ldots,d}_{q+1}\} \end{aligned} \tag{5.5}$$

where $r = n + p + 1$ and $s = m + q + 1$. Notice that the control mesh is formed by $(n + 1) \times (m + 1)$ control points.

Let us refine knot vectors in such a way that each knot has multiplicity $p + 1$ in U knot vector and multiplicity $q + 1$ in V knot vector:

$$\begin{aligned} U &= \{\underbrace{a,\ldots,a}_{p+1},\underbrace{u_{p+1}=\cdots=u_{2p+1}}_{p+1},\ldots,\underbrace{u_{r'-2p-1}=\cdots=u_{r'-p-1}}_{p+1},\underbrace{b,\ldots,b}_{p+1}\} \\ V &= \{\underbrace{c,\ldots,c}_{q+1},\underbrace{v_{q+1}=\cdots=v_{2q+1}}_{q+1},\ldots,\underbrace{v_{s'-2q-1}=\cdots=v_{s'-q-1}}_{q+1},\underbrace{d,\ldots,d}_{q+1}\} \end{aligned} \tag{5.6}$$

where $r' = n' + p + 1$ and $s' = m' + q + 1$. Notice that after the knot refinement the number of control points in u and v directions increases proportionally to the number of inserted knots. After the knot refinement $(n'+1)$ is the number of control points in u direction and $(m' + 1)$ is the number of control points in v direction. Denote new mesh of refined control points by $\hat{p}^w_{ij}$. Then each domain span $[u_k, u_{k+1}] \times [v_t, v_{t+1}]$ of not coinciding subsequent knots u_k and u_{k+1} in u direction and v_t and v_{t+1} in v direction represents a rational Bézier patch of the form

$$\begin{aligned} & S^w(u,v) = \sum_{i=0}^{p}\sum_{j=0}^{q} B_i^p(u)B_j^q(v)b_{ij} \\ & \text{where } b_{ij} = \hat{p}^w_{k-(p-i),t-(q-j)}, \quad u \in [u_k, u_{k+1}], \quad v \in [v_t, v_{t+1}] \end{aligned} \tag{5.7}$$

Figure 5.3 shows an example of a NURBS surface. The corresponding rational Bézier patches obtained after the transformation are shown in Figure 5.4.

Notice that each individual rational Bézier spline (patch) is defined on its own parameter interval (domain). It is very important to understand especially when implementing Bézier clipping method and Newton's iteration method. It is obviously that the total number of obtained rational Bézier splines (patches) is equal to the number of individual interval (domain) spans.

Efficient algorithms of knot insertion and knot vector refinement for curves and surfaces can be found in [31].

After subdividing NURBS surfaces into Bézier patches we need to construct an accelerating data structure for every patch. The reason for that and an efficient way how to do that are described in the next section.

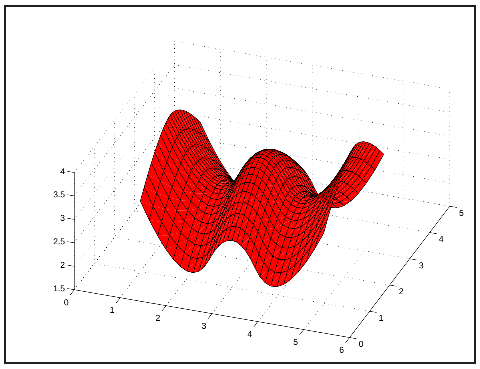


Figure 5.3: Initial NURBS surface.

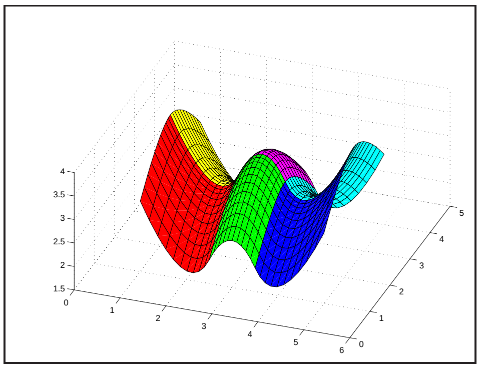


Figure 5.4: Obtained Bézier patches after transformation.

5.3 Acceleration Data Structure (ADS)

5.3.1 Introduction

In order to find the nearest object which is intersected by a ray, the brute force approach would be to test the ray with every object in a scene and select the nearest intersected one. Thousands of rays are shot through the virtual screen in order to compute an image. Each ray can recursively generate shadow, reflected, and refracted rays. If global illumination techniques are involved, the number of rays can reach millions. If the scene consists of many objects, it could take us days for computing the image, even if none of the scene objects is caught by the virtual camera. Therefore, in order to improve the performance of ray tracing, different acceleration spatial data structures are used. Some of them were mentioned in Section 3.6. Details of these techniques are beyond the scope of this Master Thesis (we refer the interested reader to [10, 17]). But the general idea of all these techniques is to avoid as many unnecessary ray-object intersection tests as possible. Unfortunately it is not possible to avoid all of them in practice. Therefore, during ray traversing of accelerating spatial data structure a number of unnecessary intersection tests with scene objects are often executed. For objects those ray intersection test is time consuming it can highly increase the computation time.

Ray-rational Bézier surface intersection test is costly. In order to save the computation time, one needs to use some kind of *object space oriented acceleration data structure*. The simplest solution is to construct for each Bézier patch a bounding volume which completely contains the given Bézier patch and has fast intersection test with a ray. Ray-bounding volume intersection test is then performed first. If the ray does not intersect the bounding volume of a Bézier patch, it supposed to miss the patch itself. This approach can save much unnecessary computation time and its performance depends on the kind of the boundary volume and the shape of a Bézier patch. The more tightly the bounding volume encloses the Bézier patch – the more unnecessary ray-rational Bézier patch intersection tests are avoided. But tighter bounding volumes usually have more complex intersection test with a ray. So, there is a tradeoff between tightness of enclosure and the speed of ray-bounding volume intersection test.

The most commonly used bounding volumes are spheres, axis aligned bounding boxes, oriented bounding boxes, parallelepipeds, trapezoid prisms etc. In the context of this Master Thesis a modification of axis aligned bounding boxes, namely *hierarchical axis aligned bounding boxes* are used. Rules for their construction and an efficient algorithm for intersection test is given in the next sections.

5.3.2 ADS Construction

Construction of hierarchical axis aligned bounding boxes is based on de Casteljau subdivision and the convex hull property of Bézier patches (see Section 2.3.3). If we construct a bounding box for Bézier patch using its control mesh points, we obtain the bounding box which is not tight enough. But if we subdivide the patch by half (in u or v direction) and construct children bounding boxes for each patch separately, the union of these bounding boxes gives us a bounding box for initial patch which is tighter. If we repeat this process recursively until appropriate level of recursion depth (alternating the direction of subdivision on each level), we obtain tight bounding box for initial patch. Additionally we store all intermediate bounding boxes together with pointers to their children bounding boxes. We obtain hierarchical data structure which is called *hierarchical axis aligned bounding boxes*.

All leaf boxes of this hierarchy does not have children and must store information about parametric domain of enclosed subpatch. This information is used in order to obtain *initial clipping* for Bézier clipping method and *initial guess* for Newton's iteration method for finding ray-rational Bézier patch intersection more efficiently. This is explained in more details in Section 5.4.

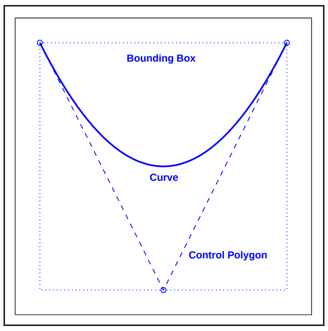


Figure 5.5: Bounding volume before subdivision step.

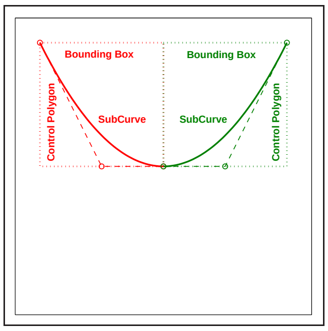


Figure 5.6: Bounding volume after subdivision step.

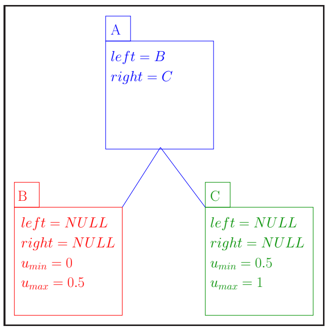


Figure 5.7: Bounding volume hierarchy.

Let us consider an example in Figure 5.5. It shows a Bézier curve defined on interval $[0, 1]$ together with its bounding box which is not tight enough. Figure 5.6 shows the same Bézier curve after de Casteljau subdivision at the midpoint in parameter interval. Each obtained subcurve has its own control polygon and bounding box. Left subcurve is defined on interval $[0, \frac{1}{2}]$ and the right curve is defined on interval $[\frac{1}{2}, 1]$. One can see that the union of children bounding boxes gives us a tighter root bounding box. In this particular example children bounding boxes are not overlapped bat they **can** overlap each other in general. The corresponding hierarchical data structure is shown in Figure 5.7. Root bounding box maintains two pointers to children bounding boxes. Children bounding boxes maintain information about the parameter domain of enclosed part of the curve.

Every hierarchical axis aligned bounding box for a rational Bézier surface requires $12 \cdot 4$ bytes $= 48$ bytes of memory:

- 6 **floats** for coordinates (*min* and *max* points in 3D);
- 2 **integers** for storing two pointers to children;
- 4 **floats** for storing parameter domain (u_{min}, u_{max}, v_{min}, and v_{max}) of enclosed part of the surface (for leaf boxes only).

In order to store data efficiently, we can separate data which are stored for each bounding box from data which are stored for leaf bounding boxes only. It can be done using the following data structures

Algorithm 5.1 (ADS-specific data structures)

```
//Structure for representing Axis Aligned Bounding Box in 3D
struct BBox
{
   Vec3f min; //minimum point of bounding box
   Vec3f max; //maximum point of bounding box

   //methods
   ...
};

//Structure for representing Leaf Specific Data
struct HLeaf
{
   float umin, umax, vmin, vmax; //parameter domain of enclosed patch

   //methods
   ...
};

//Structure for representing Hierarchical ADS
struct HBox
{
   BBox bbox; //bounding box of the current node
   HBox *left, *right; //left and right children of the current node
   HLeaf *leaf; //leaf specific data for leaf nodes only

   //methods
   ...
};
```

This requires to store additionally a pointer to leaf specific data for each bounding box (4 bytes), but the amount of saved memory is much higher, because we store leaf specific data for leaves only. Finally we need:

- **36 bytes** for non-leaf nodes;
- **52 bytes** for leaf nodes;

The amount of memory which is necessary to store l-level ADS can be calculated using the following equation:

$$M = 2^l \cdot 52 + (2^l - 1) \cdot 36 = 2^{l+3} \cdot 11 - 36 \qquad (5.8)$$

An efficient algorithm for construction of hierarchical axis aligned bounding boxes is given below.

Algorithm 5.2 (Efficient algorithm for ADS construction)

```
HBox* HIERARCHY(const BSurface &s, int level, bool plane = true)
{
   /*  Build hierarchical axis aligned bounding box for a given surface  */
   /*
     Input: @s - Bezier surface;
       @level - necessary depth level;
       @plane - auxiliary variable for alternating clipping direction
     Output: hierarchical data structure;
   */

   HBox *hbox = new HBox();

   if (level != 0)
   {
      BSurface a, b;
      if (plane)
        s.SplitU(a, b);
      else
        s.SplitV(a, b);

      hbox->leaf = NULL;
      hbox->left = HIERARCHY(a, (level - 1), !plane);
      hbox->right = HIERARCHY(b, (level - 1), !plane);
      hbox->bbox = hbox->left->bbox + hbox->right->bbox;
      return hbox;
   }

   hbox->bbox = s.GetBoundingBox();
   hbox->left = NULL;
   hbox->right = NULL;
   hbox->leaf = new HLeaf();
   hbox->leaf->umin = s.umin;
   hbox->leaf->umax = s.umax;
   hbox->leaf->vmin = s.vmin;
   hbox->leaf->vmax = s.vmax;
   return hbox;
}
```

Notice that absolutely the same approach can be applied for handling rational Bézier curves. We need fast ray-rational Bézier curve intersection test in order to determine trimming regions in parameter space of NURBS surfaces (see Section 5.5). It can be done efficiently if we use accelerating data structure for curves too. A term bounding box must be replaced with term *bounding rectangle*. Algorithm for construction of hierarchical axis aligned bounding rectangles is analogous. We just do not want to alternate the clipping direction because it does not make sense for the rational Bézier curves in $2D$.

The only one problem is how to choose the depth level of hierarchy efficiently. The good idea would be to set the level of depth in such a way that each leaf box (rectangle) encloses almost flat part of a surface (straight segment of a curve). Even more efficient would be to terminate the recursion depending of the flatness of the enclosed part of the surface (curve), i.e., to create not a balanced tree. In this section each rational Bézier patch is assumed to have a ADS tree of constant level 6 which is enough at least for low degree rational Bézier patches (curves). The next section shows how curvature based subdivision can be utilized in order to create efficient non redundant ADS trees.

5.3.3 Improvements of ADS

Sometimes alternation of the clipping directions one by one during the ADS construction can give inefficient ADS trees. It might happen in the case of stretched surface patches. In this case the alternation of clipping directions one by one results in stretched leaves of ADS tree which overlap each other much. It can significantly slow down the traverse of such tree. In order to improve the performance of ADS traverse, we have to modify the ADS construction algorithm in the following way. On each iteration we have to subdivide the surface patch in that direction *which gives two subpatches those bounding boxes have smaller aggregate area*. This idea is shown in Figures 5.8 and 5.9. Figure 5.8 shows wrongly chosen clipping direction because it results in more stretched boxes than the initial one. Figure 5.9 shows rightly chosen clipping direction. Notice that the area of a bounding box can be calculated as a sum of areas of all its faces.

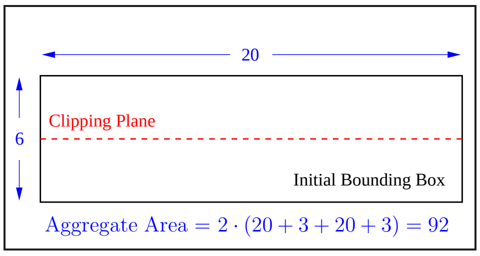


Figure 5.8: Wrongly chosen clipping plane.

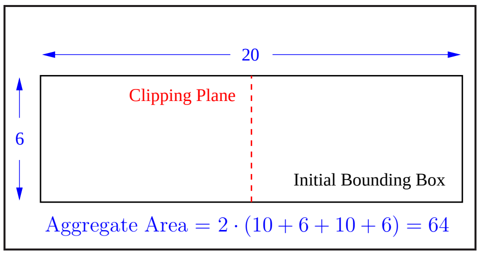


Figure 5.9: Rightly chosen clipping plane.

Even more efficiently would be to determine by this way not only directions of subdivision but also the subdivision coefficient (instead of taking the middle of each parameter direction). Appropriate number of samples can be taken in order to determine the best clipping direction and clipping coefficient. Notice that in this case taking into account just aggregate area is not enough, because samples near the border of parameter domain can give boxes of small aggregate area, but their sizes differ significantly (one of them is be much greater than another one). It is better to have boxes of almost equal size. So, in order to create good ADS trees, we need to minimize the difference of box areas as well. If s_i^l and s_i^r are surface areas of the left and right bounding boxes for the i^{th} sample, we have to choose that sample (see [17]) which gives us

$$min\{s_i^l + s_i^r + |s_i^l - s_i^r|\} \tag{5.9}$$

The algorithm of ADS construction which was proposed in the previous section has to be modified in the following way in order to create good ADS trees.

Algorithm 5.3 (The best clipping plane for the ADS construction)

```
HBox* HIERARCHY(const BSurface &s, int level)
{
   /*  Build hierarchical axis aligned bounding box for a given surface  */
   /*
     Input:  ...
     Output: hierarchical data structure;
   */

   ...
   if (level != 0)
   {
      BSurface a, b, c, d;
      float S1 = 0.0, S2 = 0.0;
      for (int i = 0; i < 2; i++) //loop over parameter directions
        for (int j = 0; j < samples; j++) //loop over samples in chosen direction
        {
           if ((i == 0) && (j == 0)) //for the first time
           {
              //split the surface using the specified parameters
              s.Split(i, j, samples, a, b);
              S1 = a.GetBoundingBox().GetArea() + b.GetBoundingBox().GetArea() +
                fabsf(a.GetBoundingBox().GetArea() - b.GetBoundingBox().GetArea());
              continue;
           }
           //split the surface using the specified parameters
           s.Split(i, j, samples, c, d);
           S2 = c.GetBoundingBox().GetArea() + d.GetBoundingBox().GetArea() +
             fabsf(c.GetBoundingBox().GetArea() - d.GetBoundingBox().GetArea());
           if (S1 > S2) //if better clipping point and direction are found
           {
              a = c;
              b = d;
              S1 = S2;
           }
        }
      ...
   }
   ...
}
```

Constant level trees might be redundant for surfaces of a simple shape. For almost flat patches we do not need deep trees, because probability of double intersection with a ray is low for such patches. Notice that problem of double ray-Bézier patch intersection is critical for Newton's iteration method, which is described in Section 5.4.3. Even for a surface which is partially flat we do not need to enclose its flat regions in leaves of too high depth. This section shows how the termination criteria of the algorithm proposed is Section 5.3.2 can be modified in order to create non redundant trees taking surface curvature into account.

Suppose we construct an ADS tree for a rational Bézier surface. Suppose also that the root bounding box for the surface is constructed and has diameter D. In order to decide whether to subdivide surface and increase the depth level of the ADS tree, we have to consider the maximum surface curvature with respect to the diameter of the root bounding box. Suppose that the maximum curvature is known and is equal to C_{max}. Then the termination criteria for the ADS tree construction is given by the following equation:

$$D \leq \alpha \cdot \frac{1}{C_{max}} \tag{5.10}$$

where α is the coefficient to control flatness. If we approximate a surface in the vicinity of the maximum curvature point by a sphere with the same curvature value, the coefficient $\frac{1}{C_{max}}$ is the radius of this sphere.

Let us consider a $2D$ example in Figure 5.10. One can see that the radius of the approximating sphere is smaller than the diameter of the bounding box. This means that we have to continue the subdivision process recursively.

Another example in Figure 5.11 shows the case when the diameter of the bounding box is smaller than radius of the approximating sphere. In this case we do not have to subdivide the enclosed surface patch and can already terminate the construction of current branch of the ADS tree.

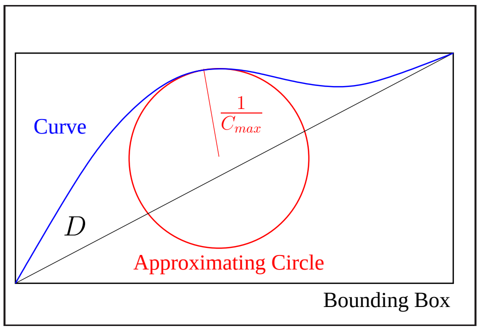


Figure 5.10: Termination criteria fails.

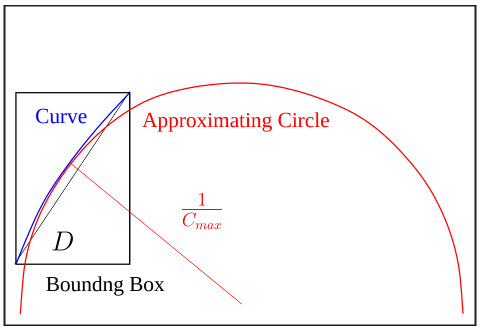


Figure 5.11: Termination criteria holds.

The flatness coefficient α in Equation (5.10) is used to control the flatness of enclosed patches. Notice that if it is too small, the ADS tree might be of too high depth. Therefore, a maximum constant level can still be used as an additional termination criteria for ADS construction. If current branch of ADS tree already has predetermined maximal allowed depth, its construction is terminated.

Axis aligned bounding boxes are not tight enclosures for surface patches and even too flat patch can have not tight axis aligned bounding box. Flatness of enclosed patches is used as termination criteria for building of ADS trees only because Bézier clipping method and Newton's iteration method for finding ray-rational Bézier patch intersection converge more quickly with flat patches. Moreover, flatness of a patch decreases the probability of its double intersection with a ray (which is critical for Newton's iteration method). Both methods for finding ray-rational Bézier patch intersection utilizing ADS structure is described in Section 5.4.

In order to compute maximum surface curvature C_{max}, the constant number of uniformly distributed samples can be used (each sample contains the maximum curvature K_{max} at the specific surface point). Surface curvature K_{max} on a rational Bézier patch at the point $(0, 0)$ is equal to the maximum of absolute values of *principal curvatures* $K_{1,2}$:

$$K_{max} = max\{|K_1|, |K_2|\} \tag{5.11}$$

where

$$K_{1,2} = K_M \pm \sqrt{K_M^2 - K_G} \tag{5.12}$$

Mean curvature K_M can be computed using equation

$$K_M = \frac{1}{2}\frac{LG - 2MF + NE}{EG - F^2} \tag{5.13}$$

Gaussian curvature K_G can be computed using equation

$$K_G = \frac{LN - M^2}{EG - F^2} \tag{5.14}$$

where E, F, and G are the coefficients of the first fundamental form:

$$E = \vec{r}_u \cdot \vec{r}_u, \qquad F = \vec{r}_u \cdot \vec{r}_v, \qquad G = \vec{r}_v \cdot \vec{r}_v \tag{5.15}$$

L, M, and N are the coefficients of the second fundamental form:

$$L = \vec{r}_{uu} \cdot \vec{n}, \qquad M = \vec{r}_{uv} \cdot \vec{n}, \qquad N = \vec{r}_{vv} \cdot \vec{n} \tag{5.16}$$

$\vec{r}_u$ and $\vec{r}_v$ are the first partial derivatives at the point $(0, 0)$, $\vec{r}_{uu}$, $\vec{r}_{uv}$, and $\vec{r}_{vv}$ are the second partial derivatives at the point $(0, 0)$, and $\vec{n}$ is the normal vector at the point $(0, 0)$:

$$\vec{n} = \frac{\vec{r}_u \times \vec{r}_v}{||\vec{r}_u \times \vec{r}_v||} \tag{5.17}$$

Efficient way for computation of the first and the second fundamental form coefficients for rational Bézier patches via control mesh points and related weights can be found in [50].

One of the ways of ADS trees construction is to create trees from the bottom to the top recursively, because backward order guarantees tightness of bounding boxes on each level. Modified version of the ADS construction algorithm (which was presented in Section 5.3.2) shows how the top-bottom curvature based subdivision described in this section can be implemented efficiently in the bottom-top manner.

ALGORITHM 5.4 (ADAPTIVE TERMINATION OF THE ADS CONSTRUCTION)

```
HBox* HIERARCHY(const BSurface &s, int level, const float alpha,
  float &Cmax)
{
   /*  Build hierarchical axis aligned bounding box for a given surface  */
   /*
     Input: ...
       @alpha - coefficient to control the flatness;
       @Cmax - auxiliary variable for propagating maximum curvature value
               between levels of the tree.
     Output: hierarchical data structure;
   */

   ...
   static float D = 0.0; //diagonal of bounding box
   if (level != 0)
   {
      ...
      float Cleft = 0.0, Cright = 0.0;
      hbox->left = HIERARCHY(a, level-1, alpha, Cleft);
      hbox->right = HIERARCHY(b, level-1, alpha, Cright);
      Cmax = (Cleft > Cright) ? Cleft : Cright;
      hbox->bbox = hbox->left->bbox + hbox->right->bbox;
      D = hbox->bbox->GetDiagonalLength();
      if (D > alpha / Cmax)
        return hbox;
      delete hbox->left;
      delete hbox->right;
   }
   else
   {
      hbox->bbox = s.GetBoundingBox();
      Cmax = s.GetMaxCurvature();
   }

   hbox->left = NULL;
   hbox->right = NULL;
   ...
}
```

Figure 5.12 shows a visualization of constant level 9 ADS tree leaf boxes for a surface represented by four rational Bézier patches. Figure 5.13 shows a visualization of maximum level 9 ADS tree leaf boxes for the same surface obtained after reduction. The number of leaf boxes in the reduced ADS tree is much smaller than the number of leaf boxes in the ADS tree of constant level.

So, taking surface curvature into account one can create non redundant ADS trees

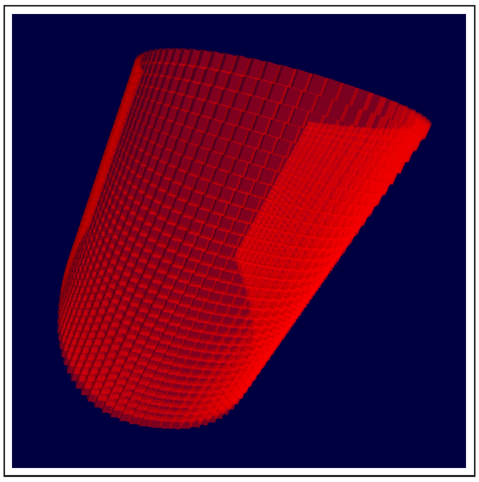


Figure 5.12: Leaves of not reduced ADS tree (complete tree with depth=9).

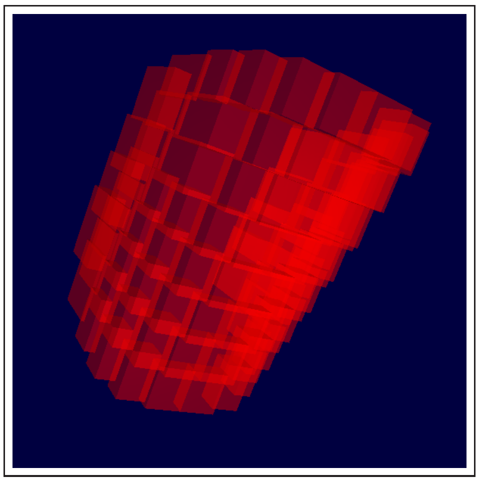


Figure 5.13: Leaves of reduced ADS tree (adaptive curvature-based construction).

which require less amount of memory for storage and still provide good enclosures for surface patches.

Now, in order to minimize the number of unnecessary ray-rational Bézier surface intersection tests, we need to traverse the ray through the hierarchical data structure and find all leaf boxes which are intersected by the ray. If we need the nearest intersection only, the data structure traverse must be implemented efficiently. Notice that bounding boxes can overlap each other. The efficient traverse scheme is given in the next section.

5.3.4 Ray Traversal Through ADS

In order to traverse hierarchical data structure, we need some kind of recursive algorithm. An efficient one is described in this section.

Figure 5.14 shows an example of two-levels hierarchical data structure which was described in Section 5.3.2. Suppose we have a root axis aligned bounding box which has two children. A ray is tested against this structure in order to obtain the nearest intersection with an enclosed rational Bézier patch. In order to do it efficiently, we need to go through the following steps:

1. Determine whether the ray intersects the root bounding box.

2. Determine a distance L_{min} to the entrance point and a distance L_{max} to the exit point of the left child box. If the left box is not intersected by the ray, we set $L_{min} = \infty$ and $L_{max} = -\infty$.

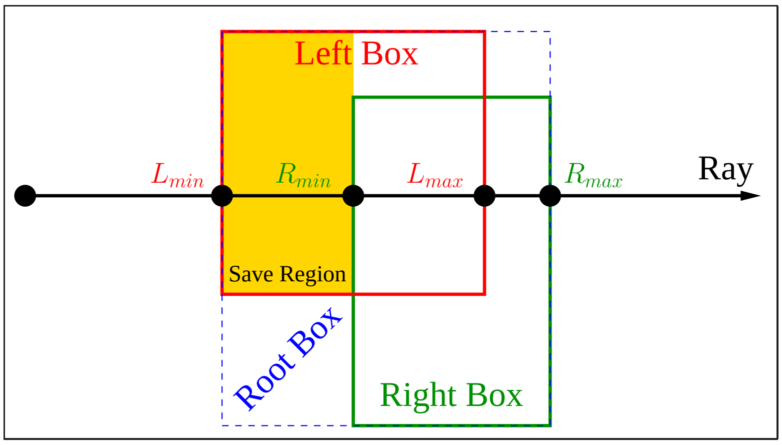


Figure 5.14: Ray traverse of hierarchical axis aligned bounding boxes.

3. Determine a distance R_{min} to the entrance point and a distance R_{max} to the exit point of the right child box. If the right box is not intersected by the ray, we set $R_{min} = \infty$ and $R_{max} = -\infty$.

4. If both children boxes are missed by the ray, we report no intersection.

5. Select the first intersected child box – i.e., the one which has the smallest entrance distance, and try to find an intersection within this box recursively.

6. Perform one of the following steps (depending on conditions):

    (a) If the box with farthest entrance distance (*sibling box*) was not intersected by the ray (steps 2 or 3), we report the result of step 5.

    (b) If the intersection was not found on step 5, we try to find an intersection within the sibling box and report the result.

    (c) If intersection was found on the step 5, we traverse the sibling box **only** if the distance to the found intersection is greater then the distance to the sibling box entrance point, i.e., outside the *save region* (see Figure 5.14). In this case more near intersection may exist within the sibling box.

The key point of this algorithm is: once intersection point was found in child box we check its sibling box **only** if the distance to this intersection point is greater than distance to the sibling box entrance point. Otherwise the found intersection point is within the *save region* (see Figure 5.14) and we can already report the intersection. Notice that once the leaf box is reached (the one which does not have children) then we apply intersection test with enclosed rational Bézier patch. Information about the parameter domain of the patch which is enclosed within the leaf box is used in order to do it efficiently (see Section 5.4). The framework of the proposed algorithm is presented below.

Algorithm 5.5 (Computing the nearest ray-patch intersection)

```
bool NEAREST_INT(const BSurface &s, HitPoint &p, Ray &r, HBox *hbox=NULL)
{
   /*  Obtain the nearest intersection with a Bezier surface
       utilizing ADS  */
   /*
     Input: @s - Bezier surface;
       @p - nearest intersection point information to be found;
       @r - traverse ray;
       @hbox - auxiliary variable for recursive traverse
     Output: true/false if intersection was/was not found;
   */

   if (hbox == NULL)
   {
      hbox = s.hbox;
      if (!hbox->bbox.Intersect(r))
        return false;
   }

   if (hbox->left == NULL)
     return s.FindIntersection(p, r, hbox);

   float Lmin = Infinity, Lmax = -Infinity;
   float Rmin = Infinity, Rmax = -Infinity;
   bool Lint = false, Rint = false;
   if (hbox->left->bbox.Intersect(r, Lmin, Lmax))
     Lint = true;
   if (hbox->right->bbox.Intersect(r, Rmin, Rmax))
     Rint = true;

   if (!(Lint || Rint))
     return false;

   bool result = false;
   if (Lmin < Rmin)
   {
      result = NEAREST_INT(s, p, r, hbox->left);
      if (!result & Rint)
        result = NEAREST_INT(s, point, ray, hbox->right);
      else
        if (result & Rint & (ray.t > Rmin))
          NEAREST_INT(s, point, ray, hbox->right);
   }
   else
   {
      result = NEAREST_INT(s, p, r, hbox->right);
      if (!result & Lint)
        result = NEAREST_INT(s, point, ray, hbox->left);
      else
        if (result & Lint & (ray.t > Lmin))
          NEAREST_INT(s, point, ray, hbox->left);
   }
   return result;
}
```

Notice that almost the same algorithm can be applied in order to solve ray-rational

Bézier patch intersection problem. One just need to change the term *surface* by the term *curve* and the term *bounding box* by the term *bounding rectangle*.

When a ray reaches a leaf box of hierarchical data structure then it is tested against the enclosed rational Bézier patch. Bézier clipping method or Newton's iteration method described in Chapter 4 can be applied in order to solve this problem. The next section describes how it can be done efficiently utilizing the hierarchical axis aligned bounding boxes structure.

5.3.5 Leaves Only Versus Trees

If acceleration spatial data structure (see Chapter 3) is used in order to improve performance of ray tracing system, the leaves of ADS tree can be used as basic objects for that system. The ADS trees are created during the preprocessing step and destroyed after their leaves are loaded in the ray tracing system. Each leaf in this case additionally stores **only** a pointer to the parent surface and not the enclosed patch itself. Therefore, memory is not affected. The total amount of memory is approximately two times smaller because all interior nodes of the trees are destroyed.

The speed performance of such approach depends on the used acceleration data structure and its settings. Section 5.8 shows the comparison of both approaches (with and without ADS trees) for two different methods for ray tracing rational Bézier patches.

5.4 Finding Nearest Ray-Bézier Patch Intersection

5.4.1 Introduction

After a leaf box of the ADS described in Section 5.3 is reached by a ray, it is necessary to test whether the ray intersects an enclosed rational Bézier patch within this leaf box. Methods for finding ray-rational Bézier patch intersection were described in Chapter 4. This section shows how the ADS can improve the performance of these methods. Notice that all modifications which presented in this section can be also applied for solving ray-rational Bézier curve in $2D$ intersection problem in analogous way.

5.4.2 Using Bézier Clipping Method

The Bézier clipping method and its improvements were described in Section 4.3. This method is based on the de Casteljau subdivision. When a ray intersects a leaf bounding box then the parameter domain of enclosed part of the surface is known (see Section 5.3.2). This means that in order to save much computation time we can perform an *initial clipping* and cut out parts of the surface which do not belong to the leaf bounding box. This requires two subdivisions in u direction of the patch and two subdivisions in v direction of the patch. Notice that the initial clipping must be applied to the projected patch and not to the surface itself. In this case we apply de Casteljau subdivision in $2D$ rather than in homogeneous $4D$. This can save up to 50% of computation time.

After applying the initial clipping we obtain *initial patch* which is used as a starting point of the Bézier clipping algorithm. If multiple intersections exist within the leaf box, Bézier clipping algorithm finds them all. We just need to select the nearest one (the first one in the sorted list which was described in Section 4.3.5).

5.4.3 Using Newton's Iteration Method

The Newton's iteration method was described in Section 4.4. This method is based on the initial guess which must lie in the basin of a root. Notice that the initial guess must be specified in the surface parameter space. The problem which arises here is multiple intersections problem. If multiple intersections exist within one leaf box, Newton's iteration method can converge to any of them depending on the initial guess. So, one needs to take such initial guess which forces the Newthon's method to converge to the nearest intersection.

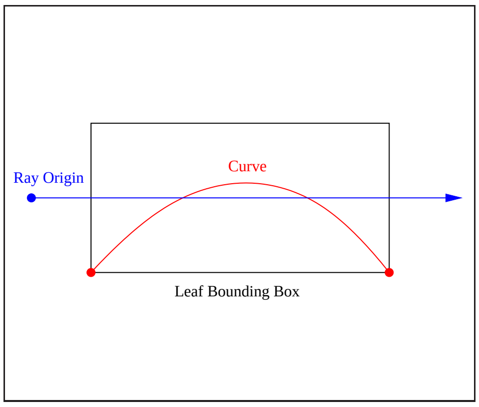


Figure 5.15: Multiple intersections problem.

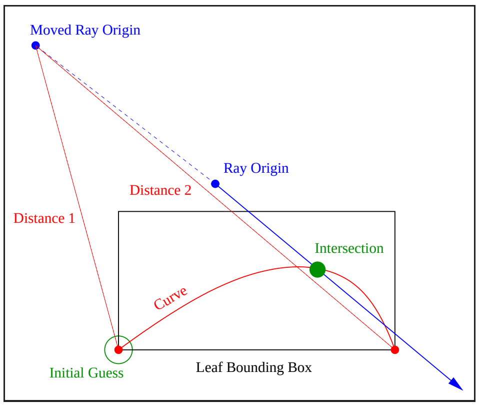


Figure 5.16: Robust approach for finding the nearest intersection.

A leaf bounding box of the ADS described in Section 5.3 can give us information about the parameter domain of the enclosed patch of the surface. We could take the middle of this domain as the initial guess **but** in this case there is no guarantee that the Newton's iteration method converges to the nearest intersection if multiple intersections exist within one leaf box. Even if we suppose that the enclosed patch of the surface within one leaf box is flat enough, we still can have multiple intersections when the ray is nearly tangential to the part of the surface (see Figure 5.15). Barth and Stürzlinger [6] proposed a solution of this problem, for the case when parallelepipeds are used as enclosures of the surface. But unfortunately the proposed solution does not work for the case of axis aligned bounding boxes because the faces of the bounding boxes do not have any relation with the parameter domain of the enclosed surface.

In order to have robust Newton's iteration method, we propose to make the following modifications of the ADS. Each leaf box must additionally store:

- precomputed value of its doubled diagonal (*MD value*);
- four corner points of the enclosed patch of the surface.

From the property of Bézier surfaces (see Section 2.5.2) we know that the corner points of a patch correspond to the corners of the parameter domain of this patch. The modification of the ADS for the surface version of Newton's iteration method requires to store additionally 4 corners $\times$ 3 coordinates $\times$ 4 bytes + 1 diagonal $\times$ 4 bytes = 52 bytes of information for each leaf box. Now each leaf box requires $48 + 52 = 100$ bytes of memory. Notice that we store the corner points in $3D$ space rather than in homogeneous $4D$ space.

The structure of ADS leaf node specific data presented in Section 5.3.2 must be modified in the following way

Algorithm 5.6 (Newton's iteration modification of the ADS data)

```
//Structure for representing Leaf Specific Data
struct HLeaf
{
   ...
   float md; //value of doubled diagonal of bounding box
   Vec3f p00, p01, p10, p11; //four corner points of enclosed patch
   ...
};
```

After modification we need the following amount of memory which is necessary to store ADS nodes:

- **36 bytes** for interior nodes;
- **104 bytes** for leaf nodes;

The amount of memory which is necessary to store l-level ADS can be now calculated using the following equation:

$$M' = 2^l \cdot 104 + (2^l - 1) \cdot 36 = 2^{l+2} \cdot 35 - 36 \tag{5.18}$$

The algorithm of non redundant ADS trees construction presented in Section 5.3.3 must be modified in the following way

Algorithm 5.7 (Newton's iteration modification of the ADS construction)

```
HBox* HIERARCHY(const BSurface &s, int level, const float alpha,
  float &Cmax)
{
   /*  Build hierarchical axis aligned bounding box for a given surface  */
   /*
     Input:  ...
     Output: hierarchical data structure;
   */

   ...
```

```
    if (level != 0)
    {
       ...
       hbox->leaf->md = 2 * D;
    }
    else
    {
       ...
       hbox->leaf->md = 2 * hbox->bbox->GetDiagonalLength()
    }

    ...
    hbox->leaf->p00 = s.GetMeshPoint(0, 0);
    hbox->leaf->p01 = s.GetMeshPoint(s.degree_u, 0);
    hbox->leaf->p10 = s.GetMeshPoint(0, s.degree_v);
    hbox->leaf->p11 = s.GetMeshPoint(s.degree_u, s.degree_v);
}
```

The same modification of the ADS structure can be applied in order to solve ray-rational Bézier curve in $2D$ problem in analogous way. Each leaf bounding rectangle must store additionally two corner points of enclosed curve and doubled precomputed value of its diagonal.

When a ray intersects a leaf bounding box then we move temporary the origin of the ray by the MD value along the opposite ray direction. Then we determine the square distances (to avoid costly square root operations) from new ray origin to each of the corner points of the patch and select the nearest one. The corresponding corner in the parameter space is then taken as the initial guess for Newton's iteration method. After that we move the ray origin back to its initial position. $2D$ example of this idea is shown in Figure 5.16.

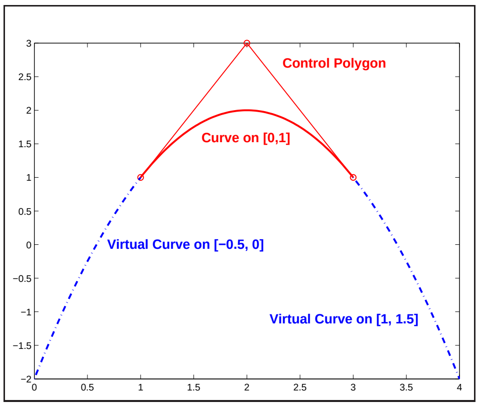


Figure 5.17: Example of virtual curves.

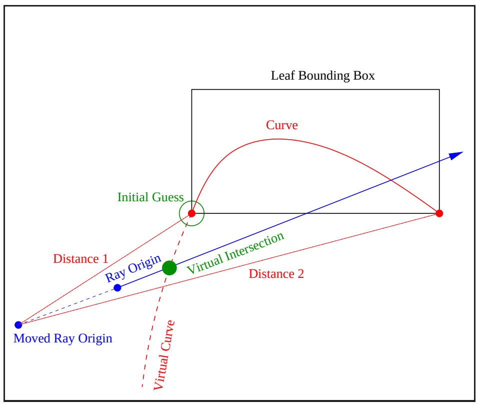


Figure 5.18: Example of virtual intersection.

In order to avoid numerical round off errors during Newton's iterations, one needs to take not exactly the corner of the parameter domain but a point which is near the corner. We propose to do it as follows:

$$\begin{aligned} u_{guess} &= u_{p,min} + (u_{p,max} - u_{p,min}) * k_1 \\ v_{guess} &= v_{p,min} + (v_{p,max} - v_{p,min}) * k_2 \end{aligned} \tag{5.19}$$

where $[u_{p,min}, u_{p,max}] \times [v_{p,min}, v_{p,max}]$ is the parameter domain of enclosed patch and $k1$, and $k2$ are variables which can take values 0.1 or 0.9 depending on the corner. This modification moves the corner points slightly towards the center of the parameter domain.

Moving of the ray origin along the opposite ray direction by the MD value is necessary in order to avoid a report of wrong intersections when the origin of the ray is inside the leaf box or near the faces of the leaf box (in this case a wrong initial guess value may be chosen).

Selected in such a way initial guess value forces the Newton's iteration method to converge to the nearest intersection. But there are two cases which must be handled carefully.

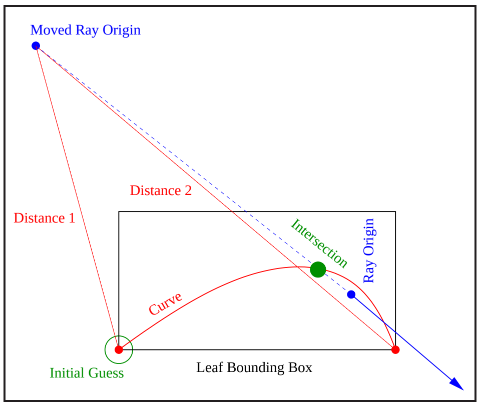


Figure 5.19: Intersection behind a ray.

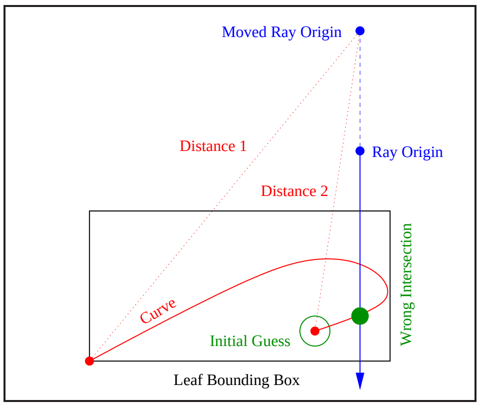


Figure 5.20: Problem with patches with high surface curvature.

The fist special case is based on the following fact: although a Bézier surface is defined on parameter domain $[u_{min}, u_{max}] \times [v_{min}, v_{max}]$, mathematically it exists outside this domain too (see $2D$ example in Figure 5.17). Let us call the patch of the surface which defined outside the parameter domain as *virtual surface*. If ray intersects the virtual surface and initial guess lies in the basin of a *virtual intersection*, Newton's iteration method converges to the virtual intersetion (see Figure 5.18). If it happens, we take the opposite corner of the parameter domain as the initial guess and try to find the nearest intersection once more again.

The second special case occurs when the obtained intersection lies behind the ray origin. It can happen if the origin of the ray is inside the leaf bounding box (see Figure 5.19). If it happens, we take the opposite corner of the parameter domain as the initial guess and try to find the nearest intersection once more again. If we obtain the same intersection or one which is behind the ray too, we report no intersections.

Notice that it can happen that the obtained intersection does not lie inside the considered leaf bounding box. It can lie in the neighbor leaf bounding box as well (and still belong to the same surface). We do not handle this case as the special one, because it does not change the logic of the proposed algorithm.

Notice that the proposed approach works **only** under assumption that the enclosed patch of the rational Bézier surface is *almost flat*. If it is not the case, this approach may give wrong results (see Figure 5.20).

Unfortunately the proposed algorithm doubles the number of memory which is required in order to store on leaf bounding box. Additional time is necessary in order to compute square distances to corner points of enclosed patch of the surface. And finally it works only under the assumption that the enclosed in every leaf bounding box patch of the surface has **at most two** intersections with any ray. This makes the Newton's iteration method not robust in general for ray tracing of NURBS surfaces when using axis aligned bounding boxes as accelerating data structure, because it can not guaranty 100% correct result.

5.5 Performing Trimming Test

5.5.1 Introduction

Trimming curves are a common method for overcoming the topologically rectangular limitations of NURBS surfaces. Trimming curves are defined in the parameter domain of NURBS surfaces and specify the regions of the surface which must be cut away. This fact is convenient for ray tracing. Once the intersection point of a ray with a NURBS surface is obtained, the coordinates of intersection in the parameter space of the NURBS surface are known. One just need to test whether this point lies in the trimmed region or not, i.e., a *trimming test* has to be executed. This section explains an effective algorithm for trimming test.

5.5.2 Trimming contours classification

A *trimming curve* is the curve which lies in the parameter domain of NURBS surface. For our purposes we restrict ourself to two subsets of trimming curves: *polylines* and *NURBS curves*. Each polyline is given by its vertices, and each NURBS curve is given by its knot vector, degree, control points and weights (see Section 2.4.3). In order to speed up the trimming test, NURBS curves have to be transformed into rational Bézier curves on the preprocessing step (see Section 5.2). Each rational Bézier curve is given by its degree, control points and weights (see Section 2.4.2).

Trimming curves form *trimming contours*. Trimming contour must be closed (its first point and last point are coincide) and has its own orientation (clockwise or counter-clockwise). The orientation of a trimming contour determines which region of the

NURBS surface to be kept. Let us use the convention (according to VRML'97 specification) that the part of the surface to be removed is on the **right** side of the curve (as you walk in the direction of its orientation).

Trimming contours are not allowed to cross and have conflict orientations (see Figures 5.21 and 5.22). Trimming contours can contain trimming contours of opposite

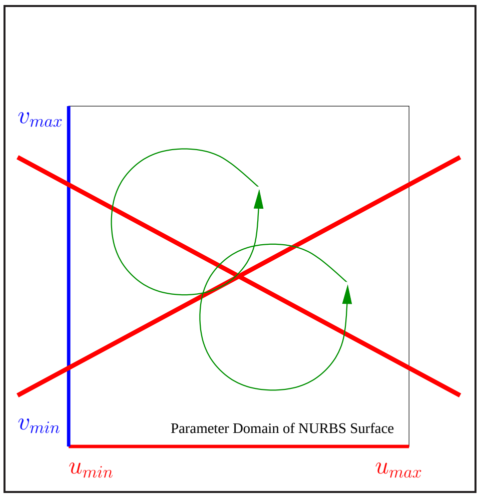


Figure 5.21: Trimming contours are not allowed to cross.

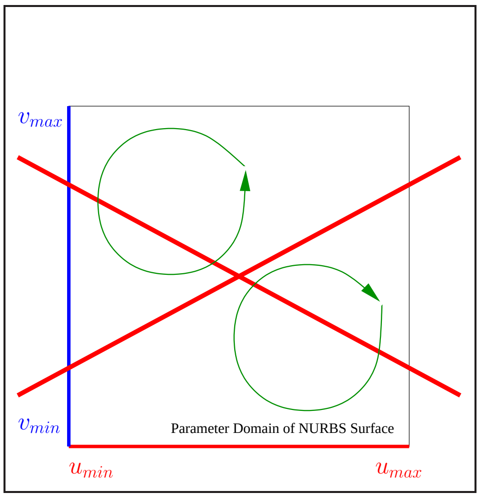


Figure 5.22: Trimming contours are not allowed to have conflict orientations.

orientation (see Figure 5.23) and share vertices and edges (see Figure 5.24).

Areas inscribed by counter-clockwise trimming contours are often termed *regions*, while those inscribed by clockwise contours are termed *holes*. The orientation of a trimming contour can be calculated using the method of Rokne ([35]) for computing the area of a polygon. Given contour points $\{p_i = (u_i, v_i)\}$, $p_0 = p_n$, $i = 0, \ldots, n$, the signed area can be computed by

$$A = \frac{1}{2} \cdot \sum_{i=0}^{n} u_i \cdot v_{(i+1)\ mod\ (n+1)} + u_{(i+1)\ mod\ (n+1)} \cdot v_i \tag{5.20}$$

If A is negative, the contour has a clockwise orientation. Otherwise, the orientation is counter-clockwise.

Notice that trimming contours consist of polylines and rational Bézier curves. In order to determine orientation of trimming contours, vertices of polylines and control points of rational Bézier curves should be used. Using of Bézier curves control points overestimates the area but the orientation is not affected.

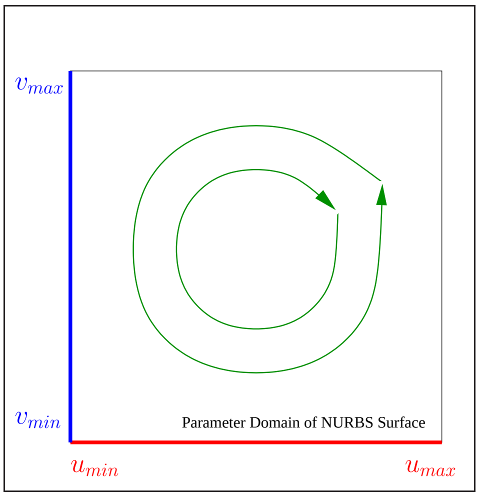


Figure 5.23: Trimming contours can contain trimming contours of opposite orientation.

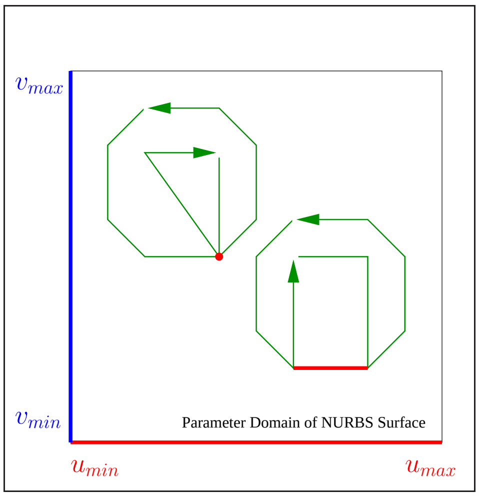


Figure 5.24: Trimming contours can share vertices and edges.

5.5.3 Building Trimming Hierarchy

In order to implement trimming test efficiently, one needs to create a trimming hierarchy (a tree of trimming contours based on containment). Since the contours are not allowed to cross, there are only three possible relationship between two contours a and b:

- a contains b;
- b contains a;
- a and b do not have common regions.

Each node in the hierarchy is represented by a trimming contour and can refer to yet another list of nodes (those contours fall inside of its contour). Each list of nodes preserves the value of its level of depth in the hierarchy (0 for top level trimming list) and precomputed orientation of the parent trimming contour. In order to accelerate trimming test, each node can also store a precomputed axis aligned bounding box of its trimming contour.

Algorithm 5.8 (Trimming hierarchy-specific data structures)

```
//structure for representing a node of the trimming hierarchy
struct Trim
{
   //the closed trimming contour
   TContour contour;
   //precomputed axis aligned bounding box of the trimming contour
```

```
    TBox bbox;
    //children nodes (represented by trimming contours
    //which are contained in the current one)
    TrimmingList children;
};

//structure for representing a list of trimming contours
struct TrimmingList
{
    //the precomputed orientation of the parent trimming contour
    //(not defined for the top level list)
    bool clockwise;
    //the level of depth of the list (0 for top level list)
    int level;
    //elements of the list (might be empty)
    vector<Trim> nodes;
};
```

Figure 5.25 shows an example of trimming contours, and Figure 5.26 shows the corresponding trimming hierarchy.

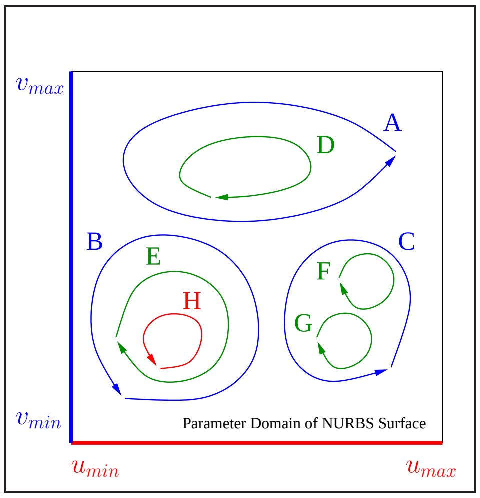


Figure 5.25: Example of trimming contours.

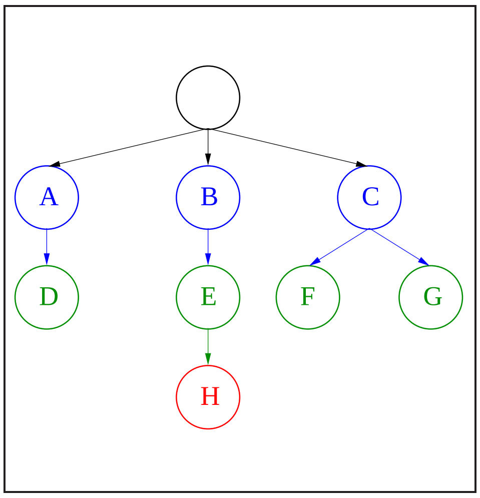


Figure 5.26: The trimming hierarchy.

Building the trimming hierarchy is depicted in the following algorithm

Algorithm 5.9 (Building the trimming hierarchy)

```
void InsertTrim(TrimmingList &tlist, Trim trim)
{
    /*  Inserts a trimming contour in a trimming list  */
    /*
      Input: @tlist - trimming list;
        @trim - trimming contour to be proceed.
    */
```

```
      //loop over trimming contours in the current list
      for (int i = (tlist.Size() - 1); i >= 0 ; i--)
      {
         //Check whether containment is possible at all
         if (trim.bbox.Overlap(tlist[i].bbox))
         {
            if (trim.bbox.GetArea() > tlist[i].bbox.GetArea())
            {
               //trim more probably contains tlist[i]
               if (trim.Contains(tlist[i]))
               {
                  InsertTrim(trim.tlist, tlist[i]);
                  tlist.Remove(tlist[i]);
                  continue;
               }
               if (tlist.Contains(trim))
               {
                  InsertTrim(tlist[i].tlist, trim);
                  return;
               }
            }
            else
            {
               //tlist[i] more probably contains trim
               if (tlist.Contains(trim))
               {
                  InsertTrim(tlist[i].tlist, trim);
                  return;
               }
               if (trim.Contains(tlist[i]))
               {
                  InsertTrim(trim.tlist, tlist[i]);
                  tlist.Remove(tlist[i]);
                  continue;
               }
            }
         }
      }
      //set the level of depth
      trim.tlist.SetDepthLevel(tlist.GetDepthLevel() + 1);
      tlist.Add(trim); //Add trimming contour in the current list
   }
```

Since trimming contours can share edges and vertices, special kind of containment test has to be applied. In order to determine whether contour a contains contour b, one needs to execute contour a inside/outside trimming test (see the next section) on the sample points of segment b. Sample points of contour b can be obtained by taking midpoints of each polyline segment and uniformly spaced sample points on each rational Bézier segment. One can take n sample points on each n-degree rational Bézier segment. **If and only if** one of the sample points of contour b falls inside contour a then contour b is judged to be contained in contour a. The case when contour a is tested on containment in contour b is treated analogous. Figure 5.27 shows an example of the containment test.

In order to avoid numerical round-off error (in the case of sample points on shared edges), the inside/outside test has to be performed with regard to some ε. If either horizontal or vertical scanline, which goes through a sample point p_i of contour b,

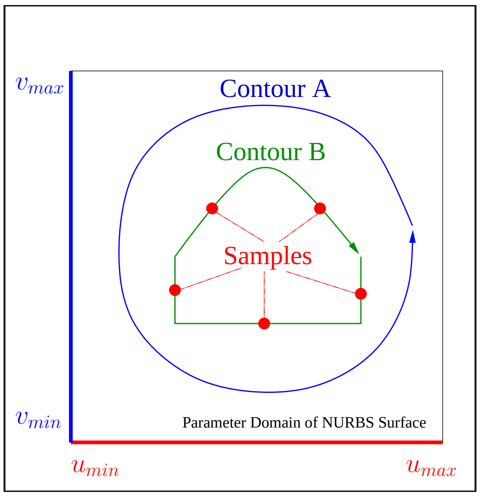


Figure 5.27: The containment test via sample points.

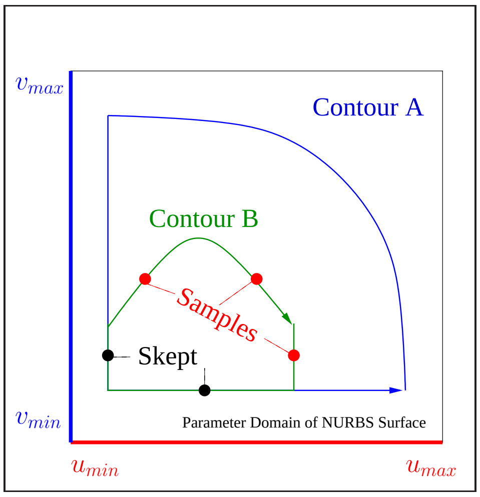


Figure 5.28: Sample points in the case of shared edges.

intersects contour a in the vicinity of the sample point p_i (with regard to ε), this point must be skipped and the next one must be considered (see Figure 5.28).

Notice that containment test is applied **only if** bounding boxes of contours a and b are overlapped, otherwise no containment is possible. As trimming contours consist of polylines as well as rational Bézier curves, it is possible that contours with smaller bounding boxes contains contours with larger bounding boxes (though the probability of such event is low). Therefore, contour b is tested on containment in contour a **first** if its boundary box area is smaller than the area of contour a bounding box. Otherwise contour a is tested on containment in contour b first.

Bounding box for each contour can be computed by union of boundary boxes of its segments: polylines and rational Bézier curves. Notice that for contours which consist of rational Bézier curves tight bounding boxes have to be computed, i.e., the refined (by recursive subdivision) control polygons of rational Bézier curves must be used for this purpose.

5.5.4 Trimming Test

Once the nearest intersection point $S = (u_{int}, v_{int})$ (in parameter uv space) of a ray and a rational Bézier patch is obtained, one needs to execute the *trimming test* in order to decide whether or not the intersection point belongs to any of trimmed regions, i.e., to one defined by a contour of clockwise orientation. If it belongs to any of trimmed regions, the ray is assumed to miss the rational Bézier patch at this point, and other possible intersections are to be computed. If the intersection point does not belong to trimmed regions, it can be reported as the nearest one.

One can see that in order to execute the trimming test, one needs to determine the lowest level trimming contour of the trimming hierarchy (see the previous section) which contains the obtained intersection point S. If the lowest level contour has clockwise orientation, the point lies in the trimmed region, and therefore the intersection of the ray with the Bézier patch does not exist at this point. If the lowest level contour has counter-clockwise orientation, the intersection of the ray with the Bézier patch is assumed to exist at this point.

Recall a corollary of the Jordan curve theorem: if any ray R in $2D$ parameter space (not to be confused with a tracing ray in $3D$) emanating from point S intersects a trimming contour an odd (even) number of times, then S is inside (outside) the contour. One can see that the inside/outside is based on the $2D$ ray-trimming contour intersection. For simplicity we assume that the ray R points in the positive u direction, i.e., its direction vector is $(1, 0)$. In practice, one can choose the ray with any of $\pm u$ or $\pm v$ directions. Recall that each trimming contour consists of segments: polylines and rational Bézier curves. The total number of ray-trimming contour intersections is equal to the number of intersections with its segments. Notice that we are not interested in the exact number of ray-contour segment intersections but only in its parity.

An efficient algorithm for determining the parity of the number of intersections with Bézier curves utilizing the convex hull property (see Section 2.3.3) is proposed by Nishita [29]. Although it was designed for Bézier curves it can be easily applied to polylines too. The algorithm begins by splitting the parameter domain of the rational Bézier patch into quadrants which meet as S as shown in Figure 5.29. To determine if R

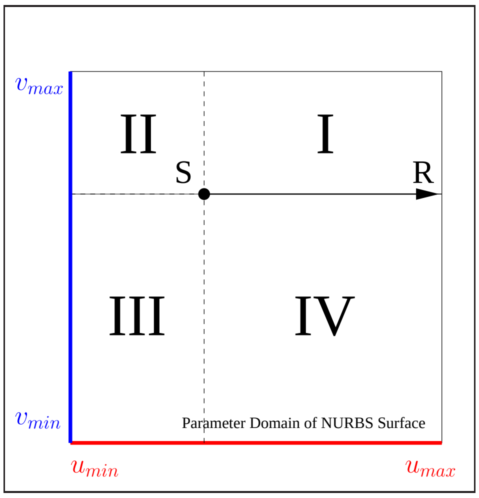


Figure 5.29: Quadrants of parameter domain.

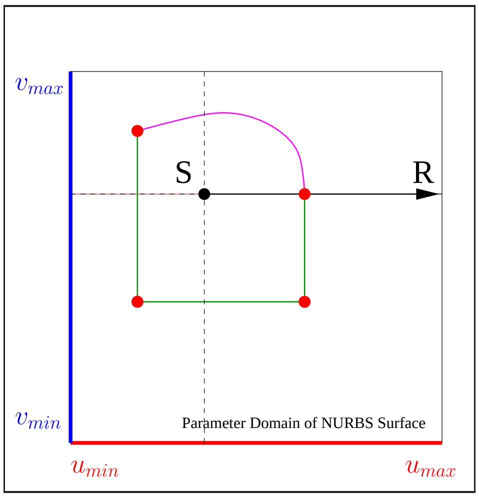


Figure 5.30: Problem of shared vertices.

intersects a given Bézier trimming curve an even or odd number of times, we categorize the curve based on which quadrants its control points occupy:

- **Case** A: All control points lie on the same side of the line containing R (in quadrants I, II, III, IV, I&II, or III&IV) or "behind" R (in quadrants II&III). The convex hull property of Bézier curves guarantees zero intersections with R.
- **Case** B: All control points lie in quadrants I&IV, but not case A. Since the curve is continuous and obeys the convex hull property, if the curve endpoints lie in the same quadrant, the curve crosses R an even number of times. Otherwise, the curve intersects R an odd number of times. Note that tangencies between the ray and trimming-curve tangencies, even those of high order, do not pose a problem.
- **Case** C: All other case.

If a trimming Bézier curve is case A or B no further processing is needed to determine its intersection parity with a ray. For a case C, we subdivide it using the de Casteljau algorithm (see Section 2.3.3) into three rational Bézier segments in such a way that the two end segments are guaranteed *a priori* to be case A or B. This can be accomplished by applying the Bézier clipping technique (see Section 4.3.2) against either the u quadrant axis ($L = u - u_{int} = 0$) or the v quadrant axis ($L = v - v_{int} = 0$) where the ray anchor $S = (u_{int}, v_{int})$. If a case C curve is Bézier clipped against the u quadrant axis, the resulting curve end segments 1 and 3 must be case A and segment 2 could be any case. If a case C curve is Bézier clipped against the v quadrant axis, the resulting curve end segments 1 and 3 must be case A or B and the middle segment 2 could be any case.

We should clip against the axis which results in the smallest segment 2. A good heuristic for this is to measure the distance from the curve endpoints to each of the axis. Generally, the largest the distance from an axis, the larger the clip tends to be. Denote $d_u = |d_0| + |d_n|$ for the case when L is the u quadrant axis, and $d_v = |d_0| + |d_n|$ when L is the v quadrant axis. Thus, if $d_u > d_v$, it is usually best to clip against $L = u - u_{int}$.

The complete point classification algorithm appears as follows:

Algorithm 5.10 (Point classification algorithm)

```
//stack of Bezier curves
typedef stack<TBezierCurve> TCurvesStack;
//the case of the curve (see the algorithm description)
enum {A, B, C};
//the direction of applied clip
enum {Du, Dv};
//determined parity of intersections
enum {ON, ODD, EVEN};

uint Parity(TBezierCurve &curve, const TVec2f &p)
{
   /* Determines whether ray with origin at p and direction (0, 1)
      intersects the given Bezier curve even or odd number of times
      (or its origin is on the Bezier curve)
   */
   /*
```

```
    Input: @curve - a curve to be tested;  @p - a checking point.
    Output: ON if the point @p is on the curve;
      ODD if the ray intersects the curve odd number of times.
      EVEN if the ray intersects the curve even number of times;
  */

  int parity = 0;
  TCurvesStack cs = TCurvesStack(0);
  cs.push(curve);
  while (cs.size() > 0)
  {
     TBezierCurve cr = cs.top();
     cs.pop();
     //determine the case of the curve (see the algorithm description)
     switch (cr.GetPointsCase())
     {
        case A:
          break; //nothing to do
        case B:
          //compare quadrants of the first and the last curve control points
          if (cr.FirstPointQuadrant() != cr.LastPointQuadrant())
            parity++;
          break;
        case C:
          //if the checking point is almost on the curve
          if (cr.LargestBoxDimension() < Epsilon)
            return ON;

          TBezierCurve b1, b2, b3;
          int dim = cr.MakeBestClip(p, b1, b2, b3);
          if (dim == Du) //curve has been Bezier clipped at Lu line
            cs.push(b2);
          else //curve has been Bezier clipped at Lv line
          {
             cs.push(b1);
             cs.push(b2);
             cs.push(b3);
          }
          break;
     }
  }

  //determine parity for the whole curve
  if ((parity % 2) == 0)
    return EVEN;
  return ODD;
}
```

Notice that if the distance from the point to one of the trimming curves is less than a tolerance value ε, the point is declared to be on a trimming curve.

Polylines can be handled analogously. Intersection test with a line segment in the case of axis aligned ray direction is relatively easy and fast. Therefore, if a polyline is of the case C, all its line segments are tested against the ray and the aggregate number of intersections is calculated in order to determine the parity.

Now, in order to determine the parity of ray-trimming contour intersections, we need to execute the parity test for each segment of the contour, i.e., for each polyline and for each rational Bézier curve which form the contour. If the ray intersects contour odd number of times (the parity is odd), its point of origin S is supposed to be inside the contour. If the ray intersects contour even number of times (the parity is even), its point of origin S is supposed to be outside the contour:

ALGORITHM 5.11 (INSIDE/OUTSIDE POINT-TRIMMING CONTOUR TEST)

```
bool Contains(const Trim &trim, const TVec2f &p)
{
   /*  Determines whether the trimming contour contains
       the given point. */
   /*
     Input: @trim - a trimming contour to be tested;  @p - testing point.
   */

   if (!trim.bbox.Contains(p)) //bounding box containment test first
     return false; //even or zero number of intersections with the contour

   uint parity = 0;
   //loop over all segments of the trimming contour
   for (int i = 0; i < trim.segments.Size(); i++)
     parity += Parity(trim.segments[i], p);

   //determine parity for the whole contour
   if ((parity % 2) == 0)
     return false;

   return true;
}
```

A problem can arise when R happens to pass through an end control point shared by two contour segments (or through a vertex shared by two polyline segments), because two intersections are reported when one is often the correct answer (see Figure 5.30). To avoid this problem one needs to perturb S away from R on a sub-pixel distance ε.

Now we ray trace trimmed NURBS (transformed into rational Bézier patches) by first performing ray intersection with the untrimmed surface. If an intersection point $S = (u_{int}, v_{int})$ is found, we look to the trim hierarchy to determine whether it is to be culled or returned as a hit:

ALGORITHM 5.12 (PERFORMING THE TRIMMING TEST)

```
bool IsTrimmed(const TrimmingList &tlist, const TVec2f &p)
{
   /*  Determines whether the point is trimmed (culled) or not.*/
   /*
     Input: @tlist - a list of trimming contours;  @p - a checking point.
     Output: true if the checking point is trimmed and false otherwise.
   */

   for (int i = 0; i < tlist.Size(); i++) //loop over trimming contours in the current list
     if (Contains(tlist[i], p)) //if the current contour contains the given point
       return Inside(tlist[i].tlist, p);

   //top level list must be handled in a special way
   if (tlist.GetDepthLevel() == 0)
   {
      if (tlist.Size() > 0)
        return !tlist[0].tlist.IsClockwise();
      else
```

```
        return false; //there are no trimming contours at all
    }

    return tlist.IsClockwise();
}
```

5.5.5 Removal of fully trimmed ADS nodes

As was explained in the previous sections each NURBS surface is transformed into rational Bézier surfaces and ADS tree is created for each rational Bézier surface. In the case of trimmed NURBS surfaces it can happen that surface patches enclosed in some leaves of the ADS tree (or even the whole rational Bézier surface) are fully trimmed. In this case such nodes are redundant and should be removed from the ADS tree in order to improve the performance of ray tracing. Notice that it can decrease the computation time because **neither** ray-rational Bézier patch intersection test **nor** trimming test are executed.

Parameter domain of each surface patch enclosed in ADS tree leaf is represented by a *domain rectangle*. The patch is fully trimmed **if and only if** the three conditions hold:

1. neither of the domain rectangle edges crosses any of trimming contours;
2. the domain rectangle does not contain any trimming contours;
3. any point of the domain rectangle is inside a trimmed region.

Martin, Cohen, Fish, and Shirley [27] suggested to deal only with the first and the last condition. This approach is not absolutely robust in the case when the parameter domain of the ADS leaf contains some trimming contours. In this case wrong images might be computed. The probability of such case is low with the higher depths of the ADS tree, but in order to guarantee the robustness, we must deal with the second condition as well.

Notice that if only the first two conditions hold, the enclosed surface patch does not have any trimmed regions. In this case we can flag the corresponding ADS leaf as ”not having trimmed regions” and skip trimming test completely for this leaf. This can slightly improve the overall performance.

Figure 5.31 shows an example of trimming contours for a single rational Bézier surface. The constant level 6 ADS tree is built for the surface. Parameter domains of leaf nodes of the ADS tree are represented by dashed rectangular mesh. Those leaves which are fully trimmed (and should therefore be removed from the tree) are marked by blue rectangles. Those leaves which do not have trimmed regions (and must therefore be flagged to accelerate the trimming test) are marked by green rectangles. All other leaves are not marked and contain surface patches which are partially trimmed.

Notice that if the left and the right child of a node of the ADS tree are removed from the tree, the node itself must be also removed.

In order to determine whether edges of the domain rectangle intersect any of trimming contours, we must anchor four rays at the corner points of the domain rectangle

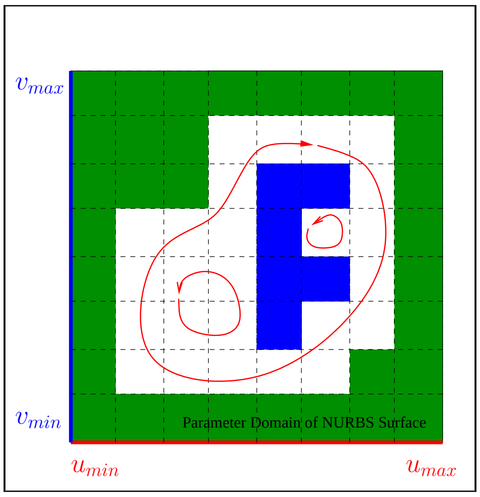


Figure 5.31: Determining fully trimmed ADS leaves.

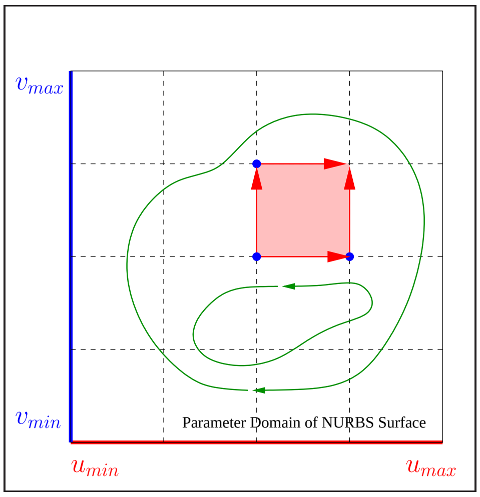


Figure 5.32: Parameter rectangle cross test.

(as shown in Figure 5.32) and shoot the rays in positive u (v) direction in order to obtain the nearest intersection with the trimming contours. If the distance to the nearest intersection point from the ray origin is less than the length of the corresponding edge of the domain rectangle, the rectangle is supposed to intersect the trimming contours and the corresponding leaf of the ADS tree **cannot** be therefore removed or flagged.

Notice that the Bézier clipping algorithm described in Section 4.3 can be used to obtain the nearest ray-rational Bézier curve intersection. Ray-polyline intersection test is relatively easy and is therefore not discussed here.

The algorithm framework for determining whether an axis aligned line segment crosses any of trimming contours is given below.

Algorithm 5.13 (Trimming contour cross test for a line segment)

```
 bool IsCrossed(const TrimmingList &tlist, const TVec2f &p,
   bool u_axis, float length, bool &trimmed)
{
   /* Determines whether an axis aligned line segment crosses
      a trimming contour.*/
   /*
     Input: @tlist - a list of trimming contours;
       @p - anchor (first point) of the line segment;
       @u_axis - determines u/v line segment direction (true/false);
       @length - the length of the line segment
     Output: returns true in the case the line crosses the trimming
       contour and false otherwise; in the case of false the @trimmed
       is set to true if the line segment lies in the trimmed region and
       false otherwise.
   */
```

```
    //loop over trimming contours in the current list
    for (int i = 0; i < tlist.Size(); i++)
    {
       float distance = Infinity; //distance to the nearest intersection
       //check whether the line intersects contour; if it intersects then
       //@distance is set to the distance to the nearest intersection
       if (tlist[i].FindNearestIntersection(p, u_axis, distance)
       {
          if (distance < length)
            return true; //the line segment crosses the contour

          //check children contours
          if (Contains(tlist[i], p))
            return IsCrossed(tlist[i].tlist, p, u_axis, length, trimmed);
       }
    }

    trimmed = tlist.contour.IsClockwise();
    return false; //the line segment does not cross the contour
 }
```

The structure of ADS leaf node specific data presented in Section 5.3.2 and modified in Section 5.4.3 must be again modified as follows:

Algorithm 5.14 (Trimming modification of the ADS data)

```
 //Structure for representing Leaf Specific Data
 struct HLeaf
 {
    ...
    //specifies whether the enclosed surface patch
    //does not have any trimmed regions
    bool solid;
    ...
 };
```

One more byte is now required in order to store one leaf ADS node. The ADS building procedure presented in Section 5.3.2 and modified in Sections 5.3.3 and 5.4.3 must be again modified as follows:

Algorithm 5.15 (Trimming modification of the ADS construction)

```
 HBox* HIERARCHY(const BSurface &s, int level, const float alpha,
   const TrimmingList &tlist, float &Cmax)
 {
    /*  Build hierarchical axis aligned bounding box for a given surface  */
    /*
      Input: ...
        @tlist - a list of trimming contours for the given surface;

      Output: hierarchical data structure or NULL is surface is fully trimmed
        and must be removed from the ray tracing application.
    */

    ...
    if (level != 0)
    {
        ...
        hbox->left = HIERARCHY(a, level-1, alpha, tlist, Cleft);
        hbox->right = HIERARCHY(b, level-1, alpha, tlist, Cright);

        //if both node's children are fully trimmed
        if ((hbox->left == NULL) & (hbox->right == NULL))
        {
           delete hbox;
           return NULL;
        }

        if (hbox->left == NULL) //if left child is fully trimmed
        {
           HBox *tmp = hbox->right;
           hbox = hbox->right; //the right child becomes the current one
           delete tmp;
           Cmax = Cright;
           return hbox;
        }

        if (hbox->right == NULL) //if right child is fully trimmed
        {
           HBox *tmp = hbox->left;
           hbox = hbox->left; //the left child becomes the current one
           delete tmp;
           Cmax = Cleft;
           return hbox;
        }

        ...
     }
     else
     {
        ...
     }

     ...
     //check whether the domain rectangle does not contain and does not
     //cross any of trimming contours; in this case hbox->leaf->solid will
     //be set to true if the surface patch is a region and to false if the surface
     //patch is a hole
     if (IsEmpty(tlist, s.umin, s.umax, s.vmin, s.vmax, hbox->leaf->solid))
     {
        if (!hbox->leaf->solid)
        {
           delete hbox;
           return NULL;
        }
     }
  }
```

Ray-rational Bézier patch intersection test presented in Section 5.3.4 must be modified as follows in order to support trimming:

Algorithm 5.16 (Trimming modification of the intersection routine)

```
  bool NEAREST_INT(const BSurface &s, HitPoint &p, Ray &r,
    const TrimmingList &tlist, HBox *hbox=NULL)
  {
     /*  Obtain the nearest intersection with a Bezier surface
         utilizing ADS  */
    /*
      Input: ...
        @tlist - a list of trimming contours for the given surface;
      Output: true/false if intersection was/was not found;
    */

    ...
    if (hbox->left == NULL)
    {
       //find the nearest intersection
       if (!s.FindIntersection(p, r, hbox))
         return false;

       //if corresponding surface patch  does not have
       //any trimmed regions then report the intersection
       if (hbox->leaf->solid)
         return true;

       //return the result of the trimming test
       return !IsTrimmed(tlist, p)
    }

    //all NEAREST_INT calls below must include @tlist parameter
    ...
}
```

5.6 Adaptive ADS Construction for Trimmed Surfaces

Adaptive curvature based ADS construction described in Section 5.3.3 does not work properly if fully trimmed ADS nodes are removed during the construction. If one of the interior node leaves is fully trimmed and the area of the surface patch enclosed in this leaf is relatively big with respect to the whole surface area, then there is no need to apply curvature based construction for the current tree branch, even if the surface patches enclosed in the branch nodes are flat.

Let us consider a plane which has many holes, such as many ADS nodes are removed as fully trimmed. If we apply adaptive curvature based ADS construction, we get just one root bounding box which does not have any children. In this case ray-surface intersection routine is executed every time the ray hit the bounding box, and many unnecessary intersection points are computed. These unnecessary intersections can be avoided if we take trimming regions into account during adaptive curvature based ADS construction. We should merge two leaves of an interior node into single leaf **only if** trimmed surface area enclosed in these leaves is below some threshold value.

As we do not need the exact value of trimmed surface area, we can use the trimmed parameter domain area instead and measure it approximately using samples. The samples are already used for curvature measurement (see Section 5.3.3), so we can just apply trimming test for each sample point and calculate approximate trimmed parameter domain area by equation

$$A_{trm} = A_n \cdot \frac{N_{trm}}{N} \tag{5.21}$$

where A_n is the area of the parameter rectangle of the considered ADS node, N_{trm} is the number of trimmed samples, and N is the total number of samples.

During the ADS construction we calculate A_{trm} for each leaf and apply curvature based construction only if the following equation holds for an interior node:

$$\frac{A_{l,trm} + A_{r,trm}}{A} < \beta \tag{5.22}$$

where $A_{l,trm}$ and $A_{r,trm}$ are the trimmed parameter domain area of the left and the right child respectively, A is the total parameter domain area of the considered Bézier patch, and β is the control coefficient. During experiments we have found that $\beta = 0.2$ is a good choice for the control coefficient.

The ADS building procedure presented in Section 5.3.2 and modified in Sections 5.3.3, 5.4.3, and 5.5.5 must be again modified as follows (notice that only the part presented in Section 5.3.3 must be modified):

Algorithm 5.17 (Adaptive ADS construction for trimmed patches)

```
static float S; //parameter domain area of the initial Bezier patch.

HBox* HIERARCHY(const BSurface &s, int level, const float alpha,
  const float beta, const TrimmingList &tlist, float &Cmax, float &TA)
{
   /*  Build hierarchical axis aligned bounding box for a given surface  */
   /*
     Input: ...
       @beta - coefficient to control the ADS construction for
               trimmed surfaces;
       @TA    - auxiliary variable for propagating trimmed parameter
               domain area between levels of the tree.
     Output: hierarchical data structure;
   */

   ...
   if (level != 0)
   {
      ...
      float Cleft = 0.0, Cright = 0.0;
      float TAleft = 0.0, TAright = 0.0;
      hbox->left = HIERARCHY(a, level-1, alpha, beta, tlist, Cleft, Aleft);
      hbox->right = HIERARCHY(b, level-1, alpha, beta, tlist, Cright, Aright);
      TA = TAleft + TAright;

      if ((hbox->left == NULL) & (hbox->right == NULL))
      {
         ...
      }
      ...

      Cmax = (Cleft > Cright) ? Cleft : Cright;
      hbox->bbox = hbox->left->bbox + hbox->right->bbox;
      if ((TA / S) > beta)
        return hbox;
      D = hbox->bbox->GetDiagonalLength();
      if (D > (alpha / Cmax))
        return hbox;
      ...
   }
```

```
      else
      {
         ...
         s.GetTrimmedAreaAndMaxCurvature(TA, Cmax);
      }
      ...
   }
```

Notice again that the value of maximum curvature and the approximate value of trimmed parameter domain area can be calculated using the same sample points in parameter space.

In the framework presented above S is the static float variable which must be set to the parameter domain area of the initial Bézier patch before executing the routine for ADS construction for this patch.

In the presented technique we have assumed that surface parameterization is uniform. The better performance can be achieved if we consider surface area in $3D$ instead of taking the area of parameter domain. The surface area in $3D$ can be computed approximately via triangulation, but such approach requires more preprocessing time.

5.7 Numerical Robustness

Bézier surface and Bézier curve evaluation routines can be implemented either using direct formulae or using de Casteljau algorithm for subdivision (see Section 2.3.3). The latter approach is preferable because it is faster than the former one. Moreover, when using de Casteljau algorithm there is a way for evaluation of point on a surface and partial derivatives at this point at the same time (using only two de Casteljau subdivisions: one in u direction and one in v direction). Control points of one of the four obtained subpatches can be used in order to evaluate the initial Bézier surface.

In order to avoid numerical round-off errors when evaluating a surface at the borders of its parameter domain, one always want to take the subpatch which has the largest parameter subdomain as shown in Figure 5.33.

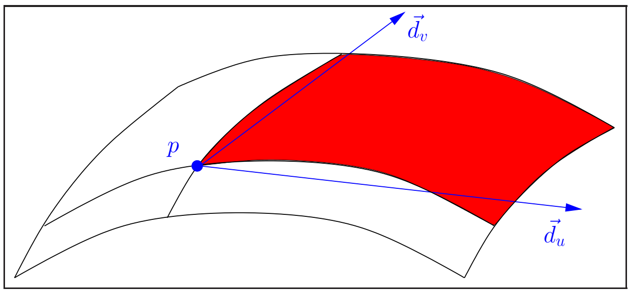


Figure 5.33: Surface evaluation via de Casteljau subdivision.

Another approach would be to use always the same subpatch (let us say one which corresponds to the upper right subdomain), but perturb the subdivision point away from the border of parameter space on ε distance in order to avoid computation of zero length partial derivatives vectors.

Notice that partial derivatives evaluation is critical for Newton's iteration method (see Section 4.4) of finding the nearest ray-rational Bézier patch intersection, because projected rational Bézier patch is evaluated on each iteration in order to converge to the right solution.

Another numerical problem happens in the case of wrongly modeled scenes where two neighbor patches are not "glued" properly and holes between them are visible on the result image. There is one way to avoid these visible cracks - to enlarge each NURBS surface along the surface borders on ε value. It can be done as follows:

1. we move all border points of u direction (along the vector through the point and the neighbor point) in v direction on ε value away from surface;

2. we move all border points of v direcation (along the vector through the point and the neighbor point) in u direction on ε value away from surface.

This idea is shown in Figure 5.35 where black points correspond to the border points of the patch before the enlargement, blue points correspond to the border points after the first step, and red points correspond to the border points after the second step.

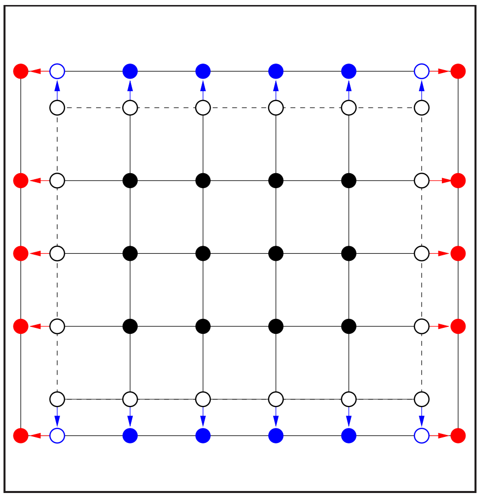


Figure 5.34: Enlarging NURBS surfaces.

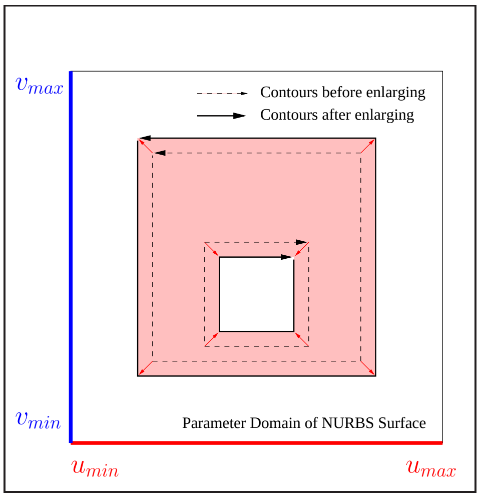


Figure 5.35: Enlarging trimming regions.

Sometimes cracks are caused by wrongly modeled trimming contours. In this case we can slightly enlarge regions and diminish holes in order to prevent visible artifacts on the final image. As kind of a trimming contour (hole or region) is determined by its orientation (or orientation of normals of its segments) everything what we want to do

- it is to move all trimming contour points along the negative direction of normals at this points on ε distance (see Figure 5.35). In the case of polylines we move polyline vertices and in the case of rational Bézier curve we move its control points. Here we suppose the normal at the point p_i to have orientation of the edge $p_i - p_{i+1}$ rotated by 90 degrees in the counter-clockwise direction. Notice that normal at each point is equal to the normalized sum of normals of two adjacent edges.

However, such approach may be inconvenient in the case of non-uniform parameterization when too small area in parameter space of a surface corresponds to relatively large area of the surface. One needs to take care when using the proposed trick.

Even with the proposed tricks the result images may still have artifacts which are caused by precision problem of floating point representation. Such artifacts are often visible as dots of wrong color because intersections at some points were wrongly computed. One way to avoid such problem is to use doubles instead of the floats. But this approach doubles the amount of memory for storing rational Bézier patches and ADS trees. Moreover, arithmetic operations with double numbers are slower than with float numbers.

The mentioned artifacts can be also more or less avoided when using antialiasing techniques (shooting more than one ray per pixel in order to obtain its color). As antialiasing techniques are used at any rate in order to improve quality of result images this approach is more preferable.

5.8 Comparison and Results

In Section 5.2 we described reasons for subdividing NURBS surfaces into rational Bézier patches on the preprocessing step of ray tracing. An efficient data structure for accelerating ray-rational Bézier patch intersection test was described in Section 5.3. Section 5.4 showed how this data structure can be utilized when using Bézier clipping method and Newton's iteration method for finding ray-rational Bézier patch intersection. Both methods were implemented within a library (*myNURBS*) which is supposed to be used in ray tracing applications. In order to test the implementation, the library was integrated into GOLEM [1] ray tracing system. Tests have been performed on Intel(R) Xeon(TM) CPU 3.06GHz (512KB cache) with an image resolution 800 × 800. Ray casting technique has been used in order to generate images. Kd-trees were used as acceleration spatial data structure. Tables 5.1 – 5.10 show the testing results for scene models which do not have trimming regions[2] for different scene models in *VRML'97*

[2]the tables use the following notations:
Preproc. Time, Sec - preprocessing time in seconds for building ADS trees;
Rendering Time, Sec - rendering time in seconds;
A1 - adaptive ADS of maximum level 7 and flatness coefficient 0.5;
A2 - adaptive ADS of maximum level 7 and flatness coefficient 0.25.
MA per HRay - number of mail boxes accesses per hit ray;
BX per HRay - number of bounding boxes tests per hit ray;
BH per HRay - number of bounding hierarchy (ADS) traverses per hit ray;
BZ per HRay - number of Bézier patch tests per hit ray;

format.

Ray tracing rational Bézier patches works in the following way. Each ray has the minimum and the maximum allowed distance from the origin where intersection points are accessed (it is done for efficient acceleration spatial data structure traverse). Let us call this interval *the interval of interest*.

At first, mail box (cache) of a patch is accessed in order to determine whether the intersection test of the current ray with the patch has already been performed (it might happen during acceleration spatial data structure traverse). If the mail box test successes, the result (intersection or miss) is obtained from the mail box cache.

If mail box test fails then the ray is tested against the bounding box of the patch. If the bounding box test fails, we have to update the mail box cache (with miss) **only if** the ray misses bounding box completely (not only on the interval of interest), i.e., of interval $(0, \inf)$. In any case the miss is reported.

If the ray intersects the bounding box then the ray traverses the acceleration data structure (ADS) of the patch **not taking** the interval of interest into account, i.e., on the whole interval $(0, \inf)$ (in order to be able to update the mail box cache with the result of the traverse).

If the ray reaches leaf bounding box of the ADS tree then it is tested against the patch, i.e., the ray-rational Bézier patch intersection test is performed. Notice that during the ADS traverse ray-rational Bézier patch intersection test can be performed many times[3].

The result of the ADS traverse (intersection or miss) is written in the mail box cache of the patch **irrespective** of the interval of interest, i.e., even if the obtained intersection is outside this interval. But the intersection is reported **only** if the obtained intersection is inside the interval of interest. Otherwise the miss is reported.

Using the notations of the Tables 5.1 – 5.10 one can write the scheme for ray tracing rational Bézier patches:

$$MA \rightarrow BX \rightarrow BH \circlearrowleft BZ \tag{5.23}$$

Tables 5.11 – 5.20 show the testing results for scene models which have trimming regions[4].

Tests have been performed for different depth levels of the ADS trees. Higher levels of the ADS trees have not been tested for trimmed scenes because of the time consuming prepreccessing step. Adaptively constructed ADS trees (using curvature based subdivision described in Section 5.3.3) have also been tested. 9 samples have been used in order to estimate maximal curvature of the surface patch enclosed in each

Avg Its/Any BZ Call - average number of iterations per one Bézier patch test;
Avg Its/Scs BZ Call - maximum number of iterations per one success Bézier patch test;
Memory, Mb - amount of memory for storing ADS;
Artifacts - visible artifacts on the final image.

[3]even for one leaf the intersection test can be performed twice if Newton's iteration method is used.

[4]the tables use additionally the following notations:
Trimming Tests (TT) - total number of executed trimming tests;
Hit TT/Total TT - ratio of the number of trimming tests which return hit to the total number of trimming tests.

leaf. Both methods for ray tracing rational Bézier patches discussed in this Master Thesis (Bézier clipping method and Newton's iteration method) have been utilized for the testing purposes. Both approaches for ADS construction (with ADS trees and with ADS leaves only) have been tested. Notice that Newton's iteration method gives wrong images in the case of "trimmed" scenes because of the low depth of the ADS trees and is given only for comparison.

Test shows that Newton's iteration method is faster than Bézier clipping method almost for all scenes and levels of ADS. The approach with ADS leaves only is in general faster than approach with ADS hierarchy. The best timings have been achieved with the maximum level of ADS.

Notice that Newton's iteration method gives wrong images for lower levels of ADS. In the case of complex scenes with thousands of NURBS surfaces construction of high level ADS is time consuming, therefore Newton's iteration method is unacceptable for such scenes. Bézier clipping method computes right images for any level of hierarchy and is therefore of more general use.

The adaptively constructed ADS do not give the best timings, but using the adaptively constructed ADS one can achieve the equilibrium between consuming of memory and speed performance.

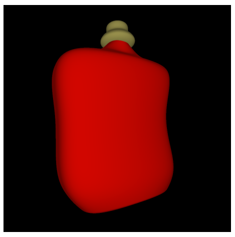
Tests show that the ADS reduction for trimmed scene models significantly improves the rendering performance, because not only unnecessary trimming tests are skipped, but many unnecessary ray-Bézier patch intersections are avoided.

	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.00	0.01	0.02	0.03	0.06	0.13	0.27	0.54	0.79	0.86
Rendering Time, Sec	4.42	4.19	3.65	3.78	3.52	3.21	3.09	**3.02**	3.39	3.10
MA per HRay	9.35	11.10	9.49	9.48	10.78	9.41	9.40	9.39	9.39	9.39
BX per HRay	9.00	10.46	9.20	9.19	10.18	9.12	9.12	9.10	9.10	9.10
BH per HRay	2.66	2.81	2.66	2.66	2.78	2.66	2.66	2.64	2.64	2.64
BZ per HRay	2.66	2.24	1.76	1.64	1.65	1.52	1.49	1.46	1.48	1.46
Avg Its/Any BZ Call	2.71	3.09	3.38	3.31	3.14	2.82	2.56	2.41	3.26	2.73
Avg Its/Scs BZ Call	5.54	5.18	4.92	4.55	4.10	3.62	3.23	3.02	4.14	3.42
ADS Memory, Mb	0.00	0.01	0.02	0.04	0.09	0.19	0.37	0.75	0.23	0.42
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.00	0.01	0.01	0.03	0.08	0.14	0.27	0.52	0.80	0.86
Rendering Time, Sec	4.53	3.84	3.35	3.12	2.78	2.75	2.52	**2.50**	2.80	2.61
MA per HRay	9.59	7.32	7.11	6.47	4.90	5.06	4.85	4.51	4.72	4.42
BX per HRay	9.20	6.94	6.67	6.02	4.47	4.64	4.40	4.10	4.21	3.95
BH per HRay	2.74	2.16	1.72	1.55	1.43	1.39	1.36	1.34	1.38	1.34
BZ per HRay	2.74	2.16	1.72	1.55	1.43	1.39	1.36	1.34	1.38	1.34
Avg Its/Any BZ Call	2.74	3.04	3.34	3.28	3.03	2.75	2.50	2.36	3.23	2.70
Avg Its/Scs BZ Call	5.57	5.18	4.92	4.53	4.01	3.59	3.22	3.02	4.17	3.44
ADS Memory, Mb	0.00	0.01	0.01	0.03	0.06	0.11	0.22	0.44	0.14	0.25
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.01	0.01	0.02	0.03	0.06	0.14	0.26	0.54	0.79	0.87
Rendering Time, Sec	4.00	3.44	2.78	3.45	2.52	2.31	2.26	**2.21**	2.33	2.24
MA per HRay	9.32	11.05	9.44	9.43	10.72	9.36	9.36	9.34	9.34	9.34
BX per HRay	8.97	10.41	9.15	9.14	10.13	9.08	9.08	9.06	9.06	9.06
BH per HRay	2.66	2.80	2.64	2.64	2.76	2.64	2.64	2.62	2.63	2.62
BZ per HRay	3.54	2.86	2.13	1.90	1.82	1.63	1.57	1.50	1.60	1.53
Avg Its/Any BZ Call	2.29	2.11	1.90	1.75	1.57	1.38	1.23	1.11	1.52	1.26
Avg Its/Scs BZ Call	2.54	2.21	1.93	1.75	1.57	1.38	1.23	1.11	1.52	1.26
ADS Memory, Mb	0.01	0.02	0.03	0.07	0.15	0.30	0.60	1.19	0.36	0.67
Visible Artifacts	many	some	some	no	no	no	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.00	0.01	0.02	0.04	0.07	0.09	0.27	0.56	0.79	0.90
Rendering Time, Sec	4.04	3.12	2.49	2.22	2.00	1.90	1.81	**1.77**	1.96	1.85
MA per HRay	9.56	7.28	7.06	6.42	4.86	5.02	4.79	4.45	4.68	4.36
BX per HRay	9.17	6.90	6.63	5.99	4.44	4.61	4.37	4.06	4.18	3.92
BH per HRay	2.74	2.13	1.69	1.54	1.41	1.37	1.34	1.32	1.36	1.32
BZ per HRay	3.65	2.72	2.05	1.78	1.57	1.48	1.41	1.37	1.48	1.40
Avg Its/Any BZ Call	2.29	2.10	1.90	1.74	1.54	1.36	1.22	1.10	1.51	1.26
Avg Its/Scs BZ Call	2.54	2.19	1.92	1.74	1.53	1.36	1.21	1.10	1.51	1.25
ADS Memory, Mb	0.01	0.01	0.03	0.06	0.11	0.22	0.44	0.89	0.27	0.50
Visible Artifacts	many	some	some	no	no	no	no	no	no	no

File Name:	**Couch.wrl**
Number of Patches:	**70**
Number of Control Points:	**1120**
Average Surface Degree:	**3.00**
Screen Coverage:	**13%**

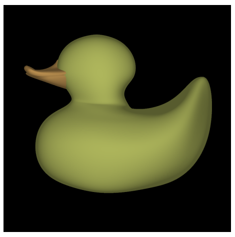
Table 5.1: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Couch.wrl model. *Copyright 1999 Lunatic interactive, Berlin*).

	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.01	0.02	0.03	0.06	0.12	0.26	0.55	1.09	1.55	1.71
Rendering Time, Sec	11.70	9.67	9.14	9.03	8.86	8.70	8.25	**7.89**	8.65	8.54
MA per HRay	23.54	9.84	9.63	9.60	9.47	9.53	9.79	9.52	9.52	9.52
BX per HRay	22.58	9.02	8.84	8.82	8.74	8.77	9.01	8.77	8.76	8.77
BH per HRay	2.34	2.08	2.05	2.03	2.03	2.03	2.03	2.03	2.03	2.03
BZ per HRay	2.34	1.97	1.90	1.89	1.84	1.81	1.77	1.76	1.80	1.78
Avg Its/Any BZ Call	3.31	3.17	2.99	2.87	2.81	2.67	2.31	1.96	2.77	2.62
Avg Its/Scs BZ Call	6.03	5.46	5.12	4.94	4.77	4.49	3.79	3.19	4.60	4.32
ADS Memory, Mb	0.01	0.02	0.04	0.09	0.19	0.38	0.76	1.52	0.46	0.88
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.01	0.02	0.04	0.07	0.10	0.27	0.55	1.10	1.54	1.74
Rendering Time, Sec	12.20	9.43	8.56	8.26	8.08	7.84	7.33	**7.11**	7.87	7.72
MA per HRay	30.57	12.08	9.00	6.64	6.50	6.90	7.16	6.96	6.33	6.56
BX per HRay	28.25	10.64	7.57	5.35	5.25	5.66	5.89	5.74	5.00	5.21
BH per HRay	2.46	1.91	1.80	1.77	1.71	1.69	1.66	1.66	1.68	1.66
BZ per HRay	2.46	1.91	1.80	1.77	1.71	1.69	1.66	1.66	1.68	1.66
Avg Its/Any BZ Call	3.21	3.21	3.08	2.99	2.94	2.77	2.35	2.00	2.90	2.73
Avg Its/Scs BZ Call	6.03	5.45	5.12	4.94	4.76	4.47	3.71	3.15	4.62	4.33
ADS Memory, Mb	0.01	0.01	0.03	0.06	0.11	0.23	0.45	0.90	0.28	0.52
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.01	0.02	0.03	0.06	0.14	0.27	0.56	1.07	1.56	1.72
Rendering Time, Sec	9.30	7.40	7.05	6.81	6.60	6.40	6.19	**5.99**	6.61	6.34
MA per HRay	23.63	9.88	9.66	9.60	9.45	9.49	9.75	9.48	9.48	9.48
BX per HRay	22.66	9.04	8.86	8.81	8.72	8.74	8.98	8.74	8.74	8.74
BH per HRay	2.36	2.09	2.06	2.03	2.03	2.02	2.03	2.02	2.02	2.02
BZ per HRay	2.93	2.37	2.23	2.14	2.04	1.96	1.87	1.82	2.02	1.91
Avg Its/Any BZ Call	2.31	2.10	1.98	1.85	1.73	1.59	1.42	1.24	1.77	1.58
Avg Its/Scs BZ Call	2.55	2.23	2.07	1.91	1.76	1.60	1.40	1.21	1.80	1.59
ADS Memory, Mb	0.01	0.03	0.07	0.15	0.30	0.60	1.21	2.42	0.74	1.40
Visible Artifacts	huge	many	many	some	no	no	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.01	0.02	0.03	0.07	0.14	0.27	0.56	1.14	1.57	1.74
Rendering Time, Sec	9.88	7.15	6.50	6.06	5.66	5.45	5.20	**5.06**	5.68	5.38
MA per HRay	30.63	12.09	9.05	6.63	6.46	6.83	7.07	6.84	6.26	6.46
BX per HRay	28.32	10.65	7.61	5.35	5.23	5.61	5.83	5.67	4.95	5.15
BH per HRay	2.47	1.91	1.80	1.76	1.69	1.67	1.63	1.62	1.65	1.63
BZ per HRay	3.07	2.27	2.07	1.95	1.85	1.78	1.72	1.68	1.84	1.75
Avg Its/Any BZ Call	2.29	2.11	1.99	1.85	1.73	1.57	1.39	1.20	1.76	1.57
Avg Its/Scs BZ Call	2.56	2.22	2.05	1.88	1.73	1.56	1.36	1.17	1.78	1.56
ADS Memory, Mb	0.01	0.03	0.06	0.11	0.23	0.45	0.90	1.80	0.55	1.04
Visible Artifacts	huge	many	many	some	no	no	no	no	no	no



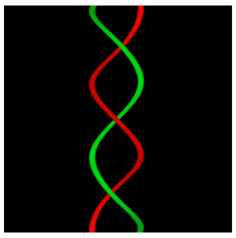
File Name:	**Parfum.wrl**
Number of Patches:	**142**
Number of Control Points:	**2144**
Average Surface Degree:	**2.89**
Screen Coverage:	**34%**

Table 5.2: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Parfum.wrl model. *Blaxxun interactive - Intel NURBS export*).

	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.01	0.02	0.04	0.06	0.15	0.32	0.60	1.28	1.71	1.91
Rendering Time, Sec	12.80	11.50	11.00	10.60	10.40	9.99	9.61	**9.37**	10.10	10.20
MA per HRay	9.01	8.32	8.40	8.43	8.38	8.37	8.37	8.37	8.37	8.37
BX per HRay	7.93	7.21	7.28	7.35	7.27	7.26	7.26	7.26	7.25	7.25
BH per HRay	2.71	2.60	2.61	2.59	2.57	2.57	2.57	2.56	2.56	2.56
BZ per HRay	2.71	2.51	2.45	2.37	2.28	2.23	2.19	2.17	2.25	2.19
Avg Its/Any BZ Call	3.13	2.81	2.59	2.51	2.44	2.25	2.02	1.82	2.31	1.96
Avg Its/Scs BZ Call	6.59	5.74	5.22	4.98	4.73	4.28	3.77	3.32	4.44	3.65
ADS Memory, Mb	0.01	0.02	0.05	0.10	0.21	0.43	0.86	1.72	0.49	0.88
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.01	0.02	0.04	0.09	0.12	0.31	0.65	1.35	1.75	1.91
Rendering Time, Sec	13.40	11.00	10.30	10.10	9.41	8.99	8.72	**8.33**	9.25	8.64
MA per HRay	11.91	13.56	11.53	9.12	9.25	8.88	8.95	8.90	8.59	8.89
BX per HRay	10.17	11.85	9.80	7.46	7.51	7.23	7.27	7.27	6.96	7.21
BH per HRay	2.87	2.42	2.33	2.25	2.17	2.14	2.11	2.08	2.14	2.10
BZ per HRay	2.87	2.42	2.33	2.25	2.17	2.14	2.11	2.08	2.14	2.10
Avg Its/Any BZ Call	2.95	2.84	2.65	2.56	2.48	2.27	2.03	1.84	2.35	1.98
Avg Its/Scs BZ Call	6.58	5.73	5.23	4.98	4.73	4.27	3.76	3.32	4.43	3.64
ADS Memory, Mb	0.01	0.02	0.03	0.06	0.13	0.26	0.51	1.02	0.30	0.52
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.01	0.02	0.03	0.08	0.15	0.31	0.62	1.25	1.75	1.88
Rendering Time, Sec	10.50	9.47	9.19	8.87	8.31	8.07	7.77	**7.53**	8.12	8.31
MA per HRay	9.41	8.41	8.41	8.41	8.36	8.35	8.34	8.34	8.35	8.34
BX per HRay	8.24	7.27	7.29	7.33	7.25	7.24	7.24	7.23	7.24	7.24
BH per HRay	2.83	2.62	2.61	2.59	2.57	2.56	2.56	2.56	2.56	2.56
BZ per HRay	3.52	3.06	2.93	2.75	2.58	2.46	2.35	2.28	2.49	2.35
Avg Its/Any BZ Call	2.37	2.29	2.19	2.08	1.94	1.80	1.64	1.49	1.79	1.60
Avg Its/Scs BZ Call	2.81	2.61	2.42	2.22	2.02	1.84	1.66	1.48	1.84	1.61
ADS Memory, Mb	0.02	0.04	0.08	0.17	0.34	0.68	1.37	2.75	0.79	1.40
Visible Artifacts	huge	huge	many	some	some	some	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.01	0.02	0.04	0.08	0.16	0.30	0.65	1.30	1.71	1.92
Rendering Time, Sec	10.80	9.00	8.38	7.83	7.27	6.98	6.68	**6.38**	7.01	6.64
MA per HRay	12.47	13.70	11.56	9.10	9.21	8.83	8.88	8.80	8.55	8.82
BX per HRay	10.56	11.94	9.82	7.45	7.49	7.20	7.22	7.21	6.93	7.17
BH per HRay	3.01	2.44	2.33	2.23	2.14	2.10	2.07	2.03	2.11	2.06
BZ per HRay	3.73	2.89	2.70	2.52	2.36	2.27	2.20	2.13	2.28	2.18
Avg Its/Any BZ Call	2.35	2.31	2.22	2.09	1.93	1.79	1.63	1.47	1.78	1.59
Avg Its/Scs BZ Call	2.83	2.59	2.39	2.18	1.98	1.80	1.62	1.45	1.80	1.58
ADS Memory, Mb	0.02	0.03	0.06	0.13	0.26	0.51	1.02	2.04	0.59	1.04
Visible Artifacts	huge	huge	many	some	some	some	no	no	no	no
										

File Name:	**Duck.wrl**
Number of Patches:	**161**
Number of Control Points:	**2576**
Average Surface Degree:	**3.00**
Screen Coverage:	**36%**

Table 5.3: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Duck.wrl model. *Blaxxun interactive - Intel NURBS export*).

	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.03	0.05	0.10	0.17	0.36	0.74	1.38	2.84	4.93	4.90
Rendering Time, Sec	12.20	7.81	5.92	4.92	4.51	4.36	4.21	**4.11**	**4.11**	4.16
MA per HRay	63.54	58.10	58.09	58.08	58.06	58.06	58.05	58.01	58.01	58.01
BX per HRay	63.53	58.07	58.07	58.06	58.04	58.04	58.03	57.99	57.99	57.99
BH per HRay	29.03	28.71	28.71	28.69	28.68	28.68	28.66	28.67	28.67	28.67
BZ per HRay	29.03	14.99	8.60	5.50	4.07	3.54	3.50	3.43	3.44	3.43
Avg Its/Any BZ Call	1.11	1.34	1.71	2.15	2.57	2.80	2.35	2.05	2.05	2.05
Avg Its/Scs BZ Call	5.13	5.01	4.99	4.82	4.72	4.65	3.86	3.24	3.24	3.24
ADS Memory, Mb	0.02	0.05	0.11	0.23	0.47	0.95	1.92	3.85	3.82	3.85
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.03	0.06	0.09	0.19	0.39	0.74	1.49	2.83	5.05	5.03
Rendering Time, Sec	12.10	6.69	4.12	2.55	1.98	1.79	1.62	**1.49**	**1.49**	1.52
MA per HRay	83.62	33.93	19.79	11.01	10.96	11.26	7.32	4.99	5.14	4.99
BX per HRay	82.63	32.49	18.85	10.04	10.06	10.39	6.28	3.98	4.10	3.98
BH per HRay	29.12	14.79	7.79	4.08	2.66	2.16	2.04	1.94	1.95	1.94
BZ per HRay	29.12	14.79	7.79	4.08	2.66	2.16	2.04	1.94	1.95	1.94
Avg Its/Any BZ Call	1.10	1.33	1.66	2.07	2.52	2.81	2.31	2.03	2.03	2.03
Avg Its/Scs BZ Call	5.15	5.02	5.00	4.85	4.75	4.69	3.87	3.24	3.24	3.24
ADS Memory, Mb	0.02	0.04	0.07	0.14	0.29	0.57	1.14	2.29	2.27	2.29
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.03	0.06	0.10	0.18	0.36	0.72	1.40	2.91	4.93	4.92
Rendering Time, Sec	11.00	7.32	5.49	4.33	3.74	3.52	3.51	**3.46**	3.47	3.47
MA per HRay	76.43	61.43	57.27	56.77	56.70	56.70	56.68	56.67	56.67	56.67
BX per HRay	76.41	61.40	57.25	56.75	56.68	56.68	56.66	56.65	56.65	56.65
BH per HRay	34.90	30.33	28.30	28.03	28.00	28.00	27.98	27.99	27.99	27.99
BZ per HRay	39.48	18.72	10.60	6.84	4.88	4.05	3.84	3.70	3.71	3.70
Avg Its/Any BZ Call	1.69	1.86	1.84	1.72	1.57	1.49	1.52	1.34	1.35	1.34
Avg Its/Scs BZ Call	2.99	2.50	2.16	1.89	1.66	1.58	1.54	1.33	1.33	1.33
ADS Memory, Mb	0.04	0.08	0.18	0.37	0.76	1.53	3.06	6.14	6.09	6.14
Visible Artifacts	huge	huge	huge	many	some	no	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.03	0.06	0.11	0.19	0.37	0.72	1.46	2.88	4.99	5.04
Rendering Time, Sec	11.10	6.37	3.76	2.21	1.56	1.32	1.21	**1.14**	1.16	1.15
MA per HRay	100.59	35.97	19.50	10.68	10.59	10.82	6.95	4.66	4.81	4.66
BX per HRay	99.38	34.35	18.57	9.76	9.75	10.02	6.01	3.76	3.88	3.76
BH per HRay	35.00	15.64	7.66	3.95	2.56	2.06	1.94	1.83	1.84	1.83
BZ per HRay	39.61	18.41	9.50	4.99	3.10	2.36	2.14	1.98	1.99	1.98
Avg Its/Any BZ Call	1.69	1.85	1.86	1.78	1.65	1.59	1.51	1.33	1.34	1.33
Avg Its/Scs BZ Call	3.00	2.51	2.16	1.86	1.62	1.54	1.44	1.27	1.27	1.27
ADS Memory, Mb	0.04	0.07	0.14	0.29	0.57	1.14	2.29	4.57	4.53	4.57
Visible Artifacts	huge	huge	huge	many	some	no	no	no	no	no
										

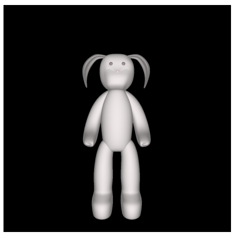
File Name:	**Dna.wrl**
Number of Patches:	**360**
Number of Control Points:	**5760**
Average Surface Degree:	**3.00**
Screen Coverage:	**5%**

Table 5.4: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Dna.wrl model. *Model courtesy of Mr. Phillip Sand Hansel II*).

	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.05	0.08	0.13	0.27	0.49	1.10	2.23	4.49	5.63	6.17
Rendering Time, Sec	7.58	6.71	6.41	6.17	6.02	5.81	5.82	**5.79**	6.23	5.98
MA per HRay	8.37	8.30	8.24	8.22	8.22	8.20	8.21	8.21	8.21	8.21
BX per HRay	7.61	7.57	7.51	7.53	7.52	7.52	7.53	7.52	7.52	7.52
BH per HRay	2.50	2.48	2.47	2.48	2.48	2.47	2.47	2.47	2.47	2.47
BZ per HRay	2.50	2.27	2.17	2.10	2.07	2.03	2.00	1.98	2.10	2.03
Avg Its/Any BZ Call	2.73	2.49	2.36	2.16	1.95	1.76	1.71	1.69	2.22	1.98
Avg Its/Scs BZ Call	5.32	4.89	4.48	4.02	3.54	3.14	3.00	2.98	4.14	3.57
ADS Memory, Mb	0.03	0.08	0.17	0.36	0.74	1.50	3.03	6.07	0.76	1.94
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.05	0.09	0.15	0.30	0.51	1.16	2.23	4.50	5.65	6.12
Rendering Time, Sec	7.93	6.42	6.00	5.70	5.51	5.25	5.20	**5.18**	5.85	5.46
MA per HRay	13.82	8.41	8.04	7.72	7.88	7.74	6.47	6.51	7.59	7.42
BX per HRay	11.99	6.87	6.74	6.40	6.68	6.54	5.28	5.33	6.11	6.10
BH per HRay	2.70	2.21	2.08	2.01	1.99	1.97	1.94	1.94	2.02	1.96
BZ per HRay	2.70	2.21	2.08	2.01	1.99	1.97	1.94	1.94	2.02	1.96
Avg Its/Any BZ Call	2.56	2.53	2.43	2.22	1.99	1.79	1.75	1.71	2.28	2.03
Avg Its/Scs BZ Call	5.32	4.89	4.47	4.00	3.52	3.13	3.00	2.98	4.14	3.57
ADS Memory, Mb	0.03	0.06	0.11	0.22	0.45	0.90	1.80	3.60	0.46	1.16
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.05	0.08	0.16	0.28	0.52	1.13	2.27	4.41	5.63	6.12
Rendering Time, Sec	6.63	5.82	5.44	5.17	4.93	4.73	5.29	**4.50**	5.27	4.90
MA per HRay	8.89	8.56	8.34	8.25	8.18	8.15	8.16	8.16	8.17	8.16
BX per HRay	8.03	7.79	7.60	7.57	7.49	7.48	7.50	7.48	7.49	7.49
BH per HRay	2.67	2.55	2.49	2.48	2.46	2.46	2.46	2.46	2.46	2.46
BZ per HRay	3.52	2.98	2.69	2.50	2.35	2.23	2.13	2.06	2.50	2.29
Avg Its/Any BZ Call	2.14	1.96	1.83	1.68	1.51	1.35	1.19	1.08	1.71	1.49
Avg Its/Scs BZ Call	2.33	2.06	1.87	1.69	1.50	1.33	1.17	1.06	1.71	1.47
ADS Memory, Mb	0.06	0.13	0.28	0.59	1.19	2.40	4.83	9.67	1.23	3.09
Visible Artifacts	huge	huge	many	some	no	no	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.05	0.09	0.15	0.26	0.57	1.08	2.29	4.52	5.67	6.14
Rendering Time, Sec	7.02	5.47	4.94	4.55	4.31	4.04	4.10	**3.79**	4.69	4.27
MA per HRay	14.58	8.70	8.12	7.74	7.81	7.63	6.36	6.38	7.54	7.32
BX per HRay	12.56	7.11	6.81	6.43	6.64	6.47	5.21	5.25	6.08	6.05
BH per HRay	2.87	2.28	2.09	2.01	1.96	1.93	1.89	1.89	2.00	1.92
BZ per HRay	3.77	2.85	2.51	2.32	2.19	2.09	2.01	1.97	2.34	2.14
Avg Its/Any BZ Call	2.12	1.95	1.82	1.66	1.49	1.32	1.17	1.06	1.70	1.47
Avg Its/Scs BZ Call	2.34	2.04	1.85	1.65	1.46	1.29	1.14	1.04	1.69	1.44
ADS Memory, Mb	0.06	0.11	0.22	0.45	0.90	1.80	3.60	7.20	0.93	2.31
Visible Artifacts	huge	huge	many	some	no	no	no	no	no	no
	File Name: **Head.wrl** **Number of Patches:** **567** **Number of Control Points:** **9072** **Average Surface Degree:** **3.00** **Screen Coverage:** **23%**									

Table 5.5: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Head.wrl model. *Model courtesy of Charles Adams*).

	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.07	0.11	0.17	0.35	0.69	1.38	2.75	5.55	7.59	8.48
Rendering Time, Sec	4.54	3.99	3.86	3.62	3.49	3.39	**3.37**	**3.37**	3.48	3.43
MA per HRay	9.02	8.26	8.39	8.29	8.13	8.05	8.14	8.11	8.10	8.10
BX per HRay	8.27	7.59	7.72	7.65	7.40	7.33	7.41	7.39	7.38	7.38
BH per HRay	2.34	2.26	2.25	2.25	2.20	2.20	2.20	2.20	2.20	2.20
BZ per HRay	2.34	2.04	1.96	1.90	1.82	1.80	1.77	1.75	1.82	1.77
Avg Its/Any BZ Call	3.04	3.03	2.90	2.53	2.25	2.09	2.07	2.08	2.30	2.09
Avg Its/Scs BZ Call	5.18	4.86	4.51	3.82	3.31	3.02	2.98	2.96	3.38	3.00
ADS Memory, Mb	0.04	0.09	0.21	0.45	0.93	1.88	3.78	7.58	1.90	4.07
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.07	0.11	0.21	0.35	0.72	1.39	2.79	5.70	7.62	8.53
Rendering Time, Sec	4.50	3.93	3.44	3.22	2.99	2.90	**2.85**	2.89	3.02	2.88
MA per HRay	13.60	7.08	6.75	6.18	6.20	6.23	5.07	5.12	6.24	5.02
BX per HRay	11.94	5.72	5.44	4.94	5.00	5.03	3.91	3.94	4.96	3.80
BH per HRay	2.40	1.89	1.75	1.68	1.63	1.62	1.58	1.57	1.63	1.57
BZ per HRay	2.40	1.89	1.75	1.68	1.63	1.62	1.58	1.57	1.63	1.57
Avg Its/Any BZ Call	2.95	3.07	2.97	2.58	2.27	2.12	2.11	2.11	2.33	2.13
Avg Its/Scs BZ Call	5.18	4.86	4.50	3.80	3.29	3.02	2.98	2.96	3.36	3.00
ADS Memory, Mb	0.04	0.07	0.14	0.28	0.56	1.12	2.25	4.49	1.14	2.42
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.07	0.11	0.19	0.32	0.67	1.40	2.82	5.75	7.75	8.48
Rendering Time, Sec	3.32	3.02	3.04	2.75	2.59	2.51	2.46	**2.43**	2.67	2.52
MA per HRay	9.25	8.20	8.31	8.20	8.01	7.93	8.01	7.99	8.00	7.98
BX per HRay	8.44	7.54	7.66	7.57	7.31	7.24	7.31	7.30	7.30	7.29
BH per HRay	2.38	2.24	2.23	2.22	2.17	2.17	2.17	2.17	2.17	2.17
BZ per HRay	2.87	2.40	2.23	2.10	1.94	1.85	1.77	1.72	1.93	1.79
Avg Its/Any BZ Call	1.96	1.87	1.73	1.55	1.36	1.18	1.06	0.99	1.36	1.12
Avg Its/Scs BZ Call	2.12	1.92	1.74	1.53	1.33	1.15	1.04	0.97	1.32	1.09
ADS Memory, Mb	0.07	0.16	0.35	0.73	1.49	3.00	6.03	12.08	3.03	6.49
Visible Artifacts	huge	huge	many	some	some	some	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.07	0.11	0.21	0.39	0.76	1.38	2.88	5.70	7.63	8.55
Rendering Time, Sec	3.40	2.72	2.46	2.31	2.20	2.11	1.99	**1.98**	2.19	2.04
MA per HRay	13.86	7.02	6.65	6.05	6.00	5.98	4.80	4.84	6.05	4.77
BX per HRay	12.12	5.68	5.37	4.86	4.87	4.87	3.77	3.79	4.85	3.66
BH per HRay	2.45	1.87	1.72	1.65	1.58	1.56	1.51	1.51	1.58	1.51
BZ per HRay	2.95	2.18	1.93	1.81	1.69	1.63	1.56	1.54	1.68	1.56
Avg Its/Any BZ Call	1.95	1.87	1.72	1.53	1.34	1.15	1.04	0.98	1.33	1.09
Avg Its/Scs BZ Call	2.11	1.89	1.70	1.49	1.29	1.11	1.01	0.96	1.28	1.05
ADS Memory, Mb	0.07	0.14	0.28	0.56	1.12	2.25	4.49	8.99	2.27	4.84
Visible Artifacts	huge	huge	many	some	some	some	no	no	no	no



File Name:	**Bunny.wrl**
Number of Patches:	**708**
Number of Control Points:	**11328**
Average Surface Degree:	**3.00**
Screen Coverage:	**14%**

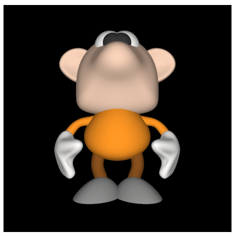
Table 5.6: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Bunny.wrl model. *Model courtesy of Charles Adams*).

	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.07	0.12	0.18	0.34	0.70	1.53	3.02	6.02	8.09	8.36
Rendering Time, Sec	8.05	7.58	7.48	7.01	6.73	**6.68**	6.73	6.75	7.13	6.86
MA per HRay	9.24	9.23	9.22	9.78	9.87	9.88	9.87	9.90	9.90	9.90
BX per HRay	8.28	8.30	8.25	8.84	8.90	8.90	8.90	8.92	8.92	8.92
BH per HRay	1.96	1.95	1.94	1.95	1.95	1.95	1.95	1.95	1.95	1.95
BZ per HRay	1.96	1.88	1.83	1.80	1.78	1.76	1.74	1.72	1.84	1.79
Avg Its/Any BZ Call	2.98	2.77	2.51	2.24	2.02	1.93	1.93	1.94	2.34	2.06
Avg Its/Scs BZ Call	5.13	4.67	4.13	3.62	3.20	3.02	3.00	2.99	3.84	3.29
ADS Memory, Mb	0.04	0.10	0.23	0.49	1.00	2.03	4.09	8.21	0.47	1.28
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.07	0.12	0.21	0.36	0.78	1.51	3.03	6.04	8.09	8.39
Rendering Time, Sec	8.28	7.43	6.83	6.50	6.23	6.12	**6.08**	6.09	6.75	6.33
MA per HRay	12.96	7.95	7.44	7.68	7.62	7.87	6.24	6.26	8.13	7.70
BX per HRay	11.24	6.50	6.05	6.27	6.25	6.46	4.93	4.95	6.68	6.23
BH per HRay	2.04	1.85	1.79	1.75	1.73	1.72	1.69	1.68	1.80	1.74
BZ per HRay	2.04	1.85	1.79	1.75	1.73	1.72	1.69	1.68	1.80	1.74
Avg Its/Any BZ Call	2.90	2.79	2.54	2.27	2.05	1.96	1.97	1.97	2.36	2.09
Avg Its/Scs BZ Call	5.13	4.66	4.12	3.60	3.19	3.02	3.00	2.99	3.83	3.27
ADS Memory, Mb	0.04	0.08	0.15	0.30	0.61	1.22	2.43	4.87	0.29	0.77
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.08	0.12	0.21	0.40	0.77	1.48	3.03	5.98	8.05	8.35
Rendering Time, Sec	6.32	5.95	5.81	5.45	5.24	5.09	5.01	**4.97**	5.66	5.40
MA per HRay	9.20	9.19	9.18	9.73	9.82	9.82	9.82	9.85	9.85	9.85
BX per HRay	8.25	8.27	8.22	8.81	8.86	8.86	8.86	8.89	8.89	8.89
BH per HRay	1.96	1.94	1.93	1.94	1.94	1.94	1.94	1.94	1.94	1.94
BZ per HRay	2.44	2.25	2.12	2.03	1.95	1.87	1.80	1.75	2.13	1.98
Avg Its/Any BZ Call	1.96	1.83	1.68	1.53	1.36	1.21	1.12	1.05	1.61	1.38
Avg Its/Scs BZ Call	2.00	1.83	1.67	1.50	1.33	1.17	1.08	1.02	1.58	1.33
ADS Memory, Mb	0.08	0.18	0.38	0.79	1.61	3.25	6.53	13.08	0.77	2.05
Visible Artifacts	some	some	no	no	no	no	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.08	0.12	0.22	0.40	0.77	1.55	3.11	6.10	8.03	8.40
Rendering Time, Sec	6.48	5.68	5.20	4.91	4.70	4.47	4.37	**4.29**	5.27	4.75
MA per HRay	12.91	7.90	7.38	7.60	7.52	7.73	6.10	6.11	8.07	7.61
BX per HRay	11.21	6.47	6.01	6.22	6.19	6.39	4.85	4.86	6.64	6.17
BH per HRay	2.03	1.84	1.77	1.73	1.70	1.68	1.65	1.64	1.78	1.71
BZ per HRay	2.55	2.18	2.02	1.91	1.84	1.78	1.71	1.68	2.03	1.86
Avg Its/Any BZ Call	1.95	1.83	1.68	1.52	1.34	1.19	1.10	1.04	1.60	1.35
Avg Its/Scs BZ Call	2.00	1.82	1.65	1.47	1.29	1.14	1.05	1.00	1.56	1.29
ADS Memory, Mb	0.08	0.15	0.30	0.61	1.22	2.43	4.87	9.74	0.59	1.55
Visible Artifacts	some	some	no	no	no	no	no	no	no	no

File Name:	**Monster.wrl**
Number of Patches:	**767**
Number of Control Points:	**12272**
Average Surface Degree:	**3.00**
Screen Coverage:	**30%**

Table 5.7: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Monster.wrl model. *Blaxxun interactive - Intel NURBS export*).

	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.09	0.16	0.24	0.46	0.96	1.89	3.76	7.55	9.64	10.44
Rendering Time, Sec	10.20	8.89	8.34	7.93	7.62	7.62	7.27	**7.20**	7.88	7.41
MA per HRay	10.82	10.61	10.59	10.49	10.53	10.50	10.50	10.53	10.53	10.53
BX per HRay	9.86	9.67	9.67	9.58	9.61	9.58	9.56	9.61	9.61	9.61
BH per HRay	2.78	2.74	2.72	2.70	2.70	2.70	2.70	2.70	2.70	2.70
BZ per HRay	2.78	2.53	2.42	2.35	2.30	2.27	2.22	2.21	2.31	2.25
Avg Its/Any BZ Call	3.08	2.83	2.62	2.40	2.21	2.04	1.88	1.79	2.22	1.98
Avg Its/Scs BZ Call	6.21	5.50	4.90	4.41	4.01	3.65	3.29	3.09	4.04	3.52
ADS Memory, Mb	0.05	0.13	0.29	0.62	1.28	2.59	5.21	10.45	1.38	3.31
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.09	0.15	0.28	0.46	0.94	1.99	3.85	7.82	9.69	10.48
Rendering Time, Sec	10.40	8.25	7.43	6.96	6.63	6.48	6.09	**6.01**	6.71	6.25
MA per HRay	14.92	8.98	8.64	8.45	8.65	8.30	6.49	6.45	9.07	6.57
BX per HRay	12.82	7.25	6.87	6.83	7.08	6.81	5.16	5.22	7.39	5.17
BH per HRay	3.04	2.43	2.24	2.16	2.11	2.08	2.00	1.99	2.12	2.03
BZ per HRay	3.04	2.43	2.24	2.16	2.11	2.08	2.00	1.99	2.12	2.03
Avg Its/Any BZ Call	2.89	2.80	2.65	2.43	2.24	2.06	1.91	1.81	2.23	2.01
Avg Its/Scs BZ Call	6.20	5.47	4.90	4.42	4.01	3.66	3.30	3.09	4.02	3.52
ADS Memory, Mb	0.05	0.10	0.19	0.39	0.77	1.55	3.10	6.20	0.83	1.98
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.09	0.16	0.25	0.46	0.95	1.94	3.83	7.65	9.60	10.49
Rendering Time, Sec	8.02	7.21	6.72	6.38	6.06	5.83	11.60	**5.52**	6.57	5.78
MA per HRay	11.71	10.97	10.79	10.48	10.49	10.42	10.41	10.44	10.49	10.45
BX per HRay	10.54	9.97	9.84	9.58	9.57	9.52	9.49	9.54	9.58	9.55
BH per HRay	3.02	2.82	2.73	2.69	2.69	2.68	2.68	2.68	2.68	2.68
BZ per HRay	3.75	3.19	2.86	2.67	2.53	2.42	2.30	2.23	2.55	2.37
Avg Its/Any BZ Call	2.17	2.08	1.96	1.83	1.67	1.52	1.36	1.22	1.66	1.43
Avg Its/Scs BZ Call	2.52	2.27	2.06	1.88	1.69	1.52	1.34	1.20	1.68	1.42
ADS Memory, Mb	0.10	0.23	0.49	1.01	2.05	4.14	8.31	16.65	2.21	5.29
Visible Artifacts	huge	huge	many	many	some	some	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.09	0.15	0.28	0.52	0.97	1.97	3.96	7.78	9.65	10.50
Rendering Time, Sec	8.51	6.61	5.85	5.41	5.11	4.90	4.55	**4.40**	5.19	4.68
MA per HRay	16.26	9.35	8.69	8.38	8.54	8.15	6.31	6.24	8.95	6.40
BX per HRay	13.80	7.51	6.91	6.79	7.00	6.71	5.05	5.08	7.31	5.07
BH per HRay	3.27	2.50	2.24	2.13	2.06	2.03	1.94	1.92	2.08	1.97
BZ per HRay	4.07	3.01	2.61	2.39	2.26	2.17	2.04	1.99	2.29	2.09
Avg Its/Any BZ Call	2.16	2.09	1.98	1.85	1.68	1.52	1.35	1.21	1.66	1.42
Avg Its/Scs BZ Call	2.53	2.26	2.06	1.87	1.68	1.50	1.33	1.19	1.66	1.39
ADS Memory, Mb	0.10	0.19	0.39	0.77	1.55	3.10	6.20	12.39	1.67	3.95
Visible Artifacts	huge	huge	many	many	some	some	no	no	no	no



File Name:	**Gnom.wrl**
Number of Patches:	**976**
Number of Control Points:	**15616**
Average Surface Degree:	**3.00**
Screen Coverage:	**27%**

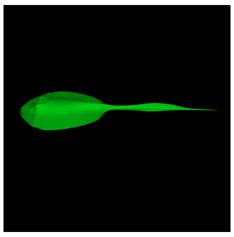
Table 5.8: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Gnom.wrl model. *Blaxxun interactive - Intel NURBS export*).

	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.11	0.18	0.32	0.53	1.12	2.27	4.59	9.08	13.98	14.92
Rendering Time, Sec	6.03	5.28	4.70	4.40	4.24	4.12	4.14	4.13	4.22	**4.10**
MA per HRay	8.63	9.95	9.93	10.10	10.10	10.09	10.24	10.20	10.19	10.20
BX per HRay	8.05	9.37	9.36	9.51	9.53	9.52	9.64	9.60	9.59	9.60
BH per HRay	3.85	3.92	3.94	3.90	3.89	3.88	3.88	3.88	3.88	3.88
BZ per HRay	3.85	2.98	2.48	2.21	2.06	1.99	1.95	1.93	1.96	1.95
Avg Its/Any BZ Call	2.11	2.34	2.45	2.41	2.31	2.22	2.18	2.16	2.42	2.21
Avg Its/Scs BZ Call	5.22	4.85	4.27	3.72	3.32	3.08	2.97	2.91	3.34	3.00
ADS Memory, Mb	0.06	0.15	0.35	0.73	1.51	3.05	6.15	12.34	5.41	8.23
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.11	0.16	0.32	0.55	1.09	2.31	4.68	9.24	13.89	15.21
Rendering Time, Sec	6.66	4.93	4.08	3.54	3.16	2.92	**2.85**	2.90	3.15	2.95
MA per HRay	14.43	10.00	8.21	6.40	5.22	3.82	3.56	3.51	4.54	3.91
BX per HRay	12.50	8.56	7.09	5.47	4.43	3.07	2.86	2.83	3.65	3.18
BH per HRay	4.58	3.01	2.26	1.88	1.66	1.53	1.48	1.47	1.60	1.50
BZ per HRay	4.58	3.01	2.26	1.88	1.66	1.53	1.48	1.47	1.60	1.50
Avg Its/Any BZ Call	1.88	2.25	2.44	2.42	2.30	2.21	2.18	2.15	2.42	2.19
Avg Its/Scs BZ Call	5.21	4.84	4.26	3.71	3.30	3.06	2.97	2.90	3.32	2.98
ADS Memory, Mb	0.06	0.11	0.23	0.46	0.91	1.83	3.66	7.31	3.22	4.88
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.10	0.18	0.30	0.54	1.07	2.26	4.52	9.12	13.83	15.14
Rendering Time, Sec	4.99	4.22	3.71	3.37	3.16	3.04	2.97	**2.95**	3.08	2.99
MA per HRay	8.71	9.81	9.76	9.93	9.94	9.93	10.07	10.04	10.03	10.03
BX per HRay	8.11	9.26	9.22	9.37	9.39	9.38	9.51	9.47	9.46	9.47
BH per HRay	3.88	3.86	3.87	3.83	3.82	3.82	3.82	3.82	3.82	3.82
BZ per HRay	4.66	3.52	2.85	2.44	2.19	2.04	1.95	1.89	2.02	1.94
Avg Its/Any BZ Call	1.90	1.79	1.65	1.48	1.33	1.18	1.05	0.98	1.28	1.10
Avg Its/Scs BZ Call	2.19	1.91	1.67	1.47	1.31	1.16	1.03	0.96	1.26	1.09
ADS Memory, Mb	0.11	0.27	0.58	1.19	2.42	4.88	9.80	19.65	8.62	13.11
Visible Artifacts	huge	many	some	some	no	no	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.11	0.17	0.32	0.57	1.13	2.33	4.63	9.23	13.97	15.14
Rendering Time, Sec	5.62	3.99	3.13	2.60	2.38	2.09	2.00	**1.99**	2.25	2.05
MA per HRay	14.54	9.83	8.01	6.19	5.02	3.62	3.35	3.28	4.31	3.67
BX per HRay	12.58	8.44	6.94	5.33	4.30	2.96	2.74	2.70	3.51	3.04
BH per HRay	4.60	2.96	2.19	1.81	1.59	1.47	1.41	1.40	1.53	1.43
BZ per HRay	5.47	3.52	2.55	2.02	1.71	1.54	1.45	1.42	1.62	1.48
Avg Its/Any BZ Call	1.84	1.79	1.66	1.49	1.33	1.17	1.04	0.96	1.29	1.09
Avg Its/Scs BZ Call	2.21	1.91	1.67	1.47	1.29	1.14	1.01	0.95	1.26	1.07
ADS Memory, Mb	0.11	0.23	0.46	0.91	1.83	3.66	7.31	14.62	6.44	9.77
Visible Artifacts	huge	many	some	some	no	no	no	no	no	no

File Name:	**Lamp.wrl**
Number of Patches:	**1152**
Number of Control Points:	**18216**
Average Surface Degree:	**2.98**
Screen Coverage:	**14%**

Table 5.9: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Lamp.wrl model. *Model courtesy of Charles Adams*).

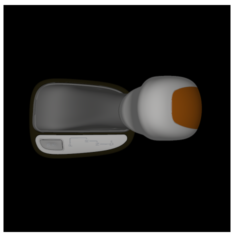
	ADS Level									
	0	1	2	3	4	5	6	7	A1	A2
Bezier Clipping Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.12	0.18	0.30	0.56	1.07	2.24	4.52	9.03	13.08	13.69
Rendering Time, Sec	2.28	1.97	1.86	1.81	1.81	**1.79**	1.82	1.80	1.85	1.83
MA per HRay	10.42	10.27	10.25	10.24	10.19	10.29	10.30	10.30	10.29	10.29
BX per HRay	9.73	9.60	9.59	9.59	9.52	9.63	9.64	9.64	9.63	9.64
BH per HRay	3.39	3.37	3.37	3.37	3.37	3.37	3.37	3.37	3.37	3.37
BZ per HRay	3.39	2.71	2.41	2.21	2.16	2.10	2.10	2.06	2.19	2.14
Avg Its/Any BZ Call	2.04	2.04	2.01	2.07	2.04	2.08	2.03	2.00	2.31	2.19
Avg Its/Scs BZ Call	4.26	3.65	3.26	3.10	3.01	2.98	2.92	2.81	3.43	3.19
ADS Memory, Mb	0.06	0.15	0.35	0.74	1.52	3.08	6.19	12.42	3.47	5.19
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Bezier Clipping Method (ADS Leaves Only)										
Preproc. Time, Sec	0.12	0.19	0.33	0.53	1.13	2.32	4.57	9.22	13.03	13.78
Rendering Time, Sec	2.39	1.85	1.63	1.49	1.43	1.34	1.36	**1.31**	1.62	1.51
MA per HRay	14.39	11.69	10.72	9.45	8.76	6.32	5.93	5.82	6.76	6.98
BX per HRay	12.28	9.84	9.00	7.90	7.35	5.08	4.76	4.72	4.90	5.37
BH per HRay	3.62	2.59	2.16	1.89	1.74	1.60	1.55	1.50	1.99	1.80
BZ per HRay	3.62	2.59	2.16	1.89	1.74	1.60	1.55	1.50	1.99	1.80
Avg Its/Any BZ Call	1.99	2.01	2.01	2.07	2.04	2.06	2.02	1.96	2.33	2.19
Avg Its/Scs BZ Call	4.26	3.61	3.25	3.11	3.01	2.97	2.90	2.76	3.44	3.16
ADS Memory, Mb	0.06	0.12	0.23	0.46	0.92	1.84	3.68	7.36	2.07	3.09
Visible Artifacts	no	no	no	no	no	no	no	no	no	no
Newton's Method (with ADS Hierarchy)										
Preproc. Time, Sec	0.12	0.18	0.30	0.58	1.12	2.28	4.52	9.03	12.96	13.67
Rendering Time, Sec	2.03	1.67	1.51	1.40	1.37	1.33	1.34	**1.32**	1.47	1.39
MA per HRay	10.36	10.17	10.14	10.15	10.11	10.22	10.23	10.23	10.23	10.23
BX per HRay	9.67	9.52	9.50	9.51	9.46	9.57	9.58	9.59	9.59	9.59
BH per HRay	3.36	3.32	3.33	3.33	3.33	3.34	3.35	3.35	3.34	3.34
BZ per HRay	4.41	3.35	2.83	2.48	2.33	2.19	2.14	2.05	2.60	2.40
Avg Its/Any BZ Call	1.66	1.44	1.26	1.14	1.04	0.98	0.92	0.86	1.30	1.15
Avg Its/Scs BZ Call	1.66	1.42	1.25	1.13	1.03	0.97	0.92	0.86	1.29	1.14
ADS Memory, Mb	0.12	0.27	0.58	1.20	2.44	4.92	9.87	19.78	5.54	8.27
Visible Artifacts	many	some	some	some	no	no	no	no	no	no
Newton's Method (ADS Leaves Only)										
Preproc. Time, Sec	0.12	0.18	0.31	0.54	1.16	2.34	4.63	9.25	13.06	13.80
Rendering Time, Sec	2.12	1.53	1.30	1.15	1.05	0.98	0.97	**0.95**	1.27	1.15
MA per HRay	14.32	11.55	10.58	9.31	8.63	6.20	5.77	5.55	6.64	6.82
BX per HRay	12.21	9.73	8.90	7.81	7.26	5.01	4.67	4.56	4.83	5.26
BH per HRay	3.59	2.54	2.11	1.84	1.70	1.57	1.52	1.46	1.96	1.77
BZ per HRay	4.72	3.16	2.49	2.05	1.83	1.64	1.56	1.48	2.35	2.00
Avg Its/Any BZ Call	1.67	1.45	1.27	1.14	1.04	0.98	0.91	0.84	1.32	1.16
Avg Its/Scs BZ Call	1.67	1.42	1.25	1.13	1.03	0.97	0.91	0.84	1.31	1.16
ADS Memory, Mb	0.12	0.23	0.46	0.92	1.84	3.68	7.36	14.73	4.15	6.18
Visible Artifacts	many	some	some	some	no	no	no	no	no	no



File Name:	**Stingray.wrl**
Number of Patches:	**1160**
Number of Control Points:	**18560**
Average Surface Degree:	**3.00**
Screen Coverage:	**6%**

Table 5.10: Comparison of ray tracing untrimmed rational Bézier surfaces methods (Stingray.wrl model. *Blaxxun interactive - Intel NURBS export*).

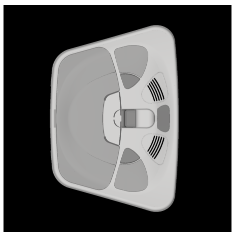
	ADS Level								
	w/o ADS reduction					with ADS reduction			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	3.47	3.64	3.95	4.02		3.45	3.69	4.02	4.04
Rendering Time, Sec	51.40	47.40	48.60	48.30		34.00	26.70	25.10	**25.00**
MA per HRay	24.35	25.08	25.08	25.08		15.28	12.20	12.20	12.20
BX per HRay	19.50	20.12	20.12	20.13		11.80	9.96	9.96	9.96
BH per HRay	9.17	9.11	9.11	9.11		5.62	4.55	4.55	4.55
BZ per HRay	9.18	9.15	9.22	9.20		5.90	4.55	4.59	4.58
Avg Its/Any BZ Call	3.52	3.22	3.27	3.25		3.61	3.21	3.25	3.24
Avg Its/Scs BZ Call	5.29	4.89	5.01	4.97		5.55	5.13	5.21	5.19
ADS Memory, Mb	0.19	0.42	0.29	0.31		0.10	0.20	0.12	0.13
Trimming Tests (TT)	773486	773200	773290	773201		475951	357794	359714	358733
Hit TT/Total TT	0.20	0.20	0.20	0.20		0.33	0.43	0.43	0.43
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	3.52	3.66	3.96	3.99		3.44	3.75	4.09	4.08
Rendering Time, Sec	49.70	45.50	54.20	48.20		34.60	**24.10**	24.60	25.70
MA per HRay	34.45	32.11	32.93	31.97		23.61	15.95	17.01	16.50
BX per HRay	23.24	22.65	23.23	22.41		15.95	10.74	11.83	11.49
BH per HRay	9.23	8.64	8.80	8.72		6.05	4.38	4.49	4.46
BZ per HRay	9.23	8.64	8.80	8.72		6.05	4.38	4.49	4.46
Avg Its/Any BZ Call	3.47	3.29	3.32	3.31		3.54	3.26	3.27	3.26
Avg Its/Scs BZ Call	5.27	4.89	4.98	4.95		5.54	5.14	5.21	5.19
ADS Memory, Mb	0.14	0.28	0.20	0.21		0.07	0.12	0.08	0.08
Trimming Tests (TT)	767125	751160	756109	753014		476225	349756	352966	350730
Hit TT/Total TT	0.20	0.20	0.20	0.20		0.33	0.43	0.43	0.43
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	3.48	3.61	3.91	4.05		3.52	3.71	4.17	4.00
Rendering Time, Sec	53.30	52.20	53.90	53.20		34.70	**26.10**	26.20	29.30
MA per HRay	28.76	27.39	27.41	27.41		18.11	13.38	13.39	13.39
BX per HRay	22.89	21.85	21.87	21.86		13.86	10.85	10.86	10.86
BH per HRay	10.79	9.92	9.92	9.92		6.59	4.95	4.95	4.95
BZ per HRay	18.21	16.98	17.13	17.09		10.99	7.46	7.55	7.52
Avg Its/Any BZ Call	1.92	1.81	1.86	1.84		2.00	1.88	1.91	1.90
Avg Its/Scs BZ Call	1.96	1.84	1.89	1.86		2.07	1.93	1.97	1.96
ADS Memory, Mb	0.32	0.70	0.49	0.53		0.17	0.32	0.20	0.21
Trimming Tests (TT)	1465172	1594394	1552428	1572104		837253	619713	613024	616016
Hit TT/Total TT	0.10	0.10	0.10	0.10		0.17	0.25	0.25	0.25
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	3.42	3.73	4.05	4.06		3.55	3.71	4.16	4.12
Rendering Time, Sec	53.30	52.00	51.70	51.10		35.90	**24.30**	25.70	25.70
MA per HRay	41.76	35.48	36.34	35.36		28.62	17.68	18.83	18.25
BX per HRay	27.93	24.91	25.54	24.69		19.11	11.82	13.04	12.67
BH per HRay	11.07	9.47	9.65	9.58		7.23	4.79	4.91	4.87
BZ per HRay	18.49	16.07	16.41	16.27		11.40	7.18	7.36	7.30
Avg Its/Any BZ Call	1.92	1.81	1.86	1.83		2.01	1.89	1.92	1.91
Avg Its/Scs BZ Call	1.96	1.83	1.88	1.85		2.07	1.93	1.97	1.96
ADS Memory, Mb	0.28	0.55	0.40	0.43		0.15	0.24	0.15	0.16
Trimming Tests (TT)	1435155	1532155	1504166	1516877		835396	612047	610094	611599
Hit TT/Total TT	0.11	0.11	0.11	0.11		0.18	0.28	0.28	0.28



File Name:	**Gear_shift.wrl**
Number of Patches:	**1391**
Number of Control Points:	**27559**
Average Surface Degree:	**4.04**
Number of Trimming Contours:	**781**
Screen Coverage:	**18%**

Table 5.11: Comparison of ray tracing trimmed rational Bézier surfaces methods (Gear_shift.wrl model. *Model courtesy of DaimlerChrysler AG*).

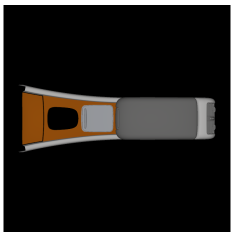
	ADS Level								
	w/o ADS reduction					with ADS reduction			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	3.93	4.18	4.89	4.96		3.95	4.17	4.99	4.93
Rendering Time, Sec	363.00	356.00	379.00	354.00		226.00	246.00	240.00	**181.00**
MA per HRay	80.11	80.09	80.10	80.10		43.53	36.51	36.50	36.50
BX per HRay	63.64	63.63	63.64	63.64		35.29	29.60	29.60	29.60
BH per HRay	34.27	34.26	34.27	34.27		21.01	17.78	17.78	17.78
BZ per HRay	33.25	33.16	32.76	32.77		20.57	14.89	14.92	14.89
Avg Its/Any BZ Call	4.10	4.05	4.14	4.11		4.36	4.40	4.41	4.40
Avg Its/Scs BZ Call	4.96	4.89	4.94	4.91		4.98	4.95	4.96	4.95
ADS Memory, Mb	0.27	0.61	0.47	0.55		0.10	0.17	0.15	0.16
Trimming Tests (TT)	6621326	6621317	6621424	6621490		4366190	3207562	3233542	3213853
Hit TT/Total TT	0.04	0.04	0.04	0.04		0.06	0.08	0.08	0.08
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	3.92	4.17	4.88	4.94		3.90	4.17	4.99	4.93
Rendering Time, Sec	402.00	348.00	354.00	348.00		234.00	202.00	**183.00**	187.00
MA per HRay	119.07	109.84	109.40	109.06		63.93	41.00	40.73	40.92
BX per HRay	74.46	71.50	68.25	67.91		39.17	26.28	26.24	26.14
BH per HRay	33.48	32.31	32.84	32.49		21.11	14.86	14.97	14.88
BZ per HRay	33.48	32.31	32.84	32.49		21.11	14.86	14.97	14.88
Avg Its/Any BZ Call	4.09	4.10	4.11	4.11		4.31	4.38	4.39	4.39
Avg Its/Scs BZ Call	4.96	4.89	4.95	4.92		4.98	4.95	4.96	4.95
ADS Memory, Mb	0.20	0.40	0.32	0.37		0.07	0.10	0.09	0.10
Trimming Tests (TT)	6637839	6537991	6589249	6568710		4422679	3191689	3224798	3201819
Hit TT/Total TT	0.04	0.04	0.04	0.04		0.06	0.08	0.08	0.08
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	3.93	4.16	4.97	4.98		3.94	4.13	4.95	4.88
Rendering Time, Sec	422.00	444.00	416.00	405.00		258.00	206.00	**195.00**	196.00
MA per HRay	83.17	82.80	82.83	82.82		45.38	37.76	37.78	37.77
BX per HRay	65.67	65.42	65.44	65.43		36.46	30.43	30.44	30.44
BH per HRay	35.32	35.18	35.19	35.18		21.66	18.25	18.25	18.25
BZ per HRay	66.37	66.14	65.38	65.36		40.57	28.97	29.11	28.99
Avg Its/Any BZ Call	1.82	1.72	1.76	1.73		1.83	1.72	1.73	1.72
Avg Its/Scs BZ Call	1.83	1.72	1.76	1.74		1.84	1.73	1.74	1.73
ADS Memory, Mb	0.47	1.00	0.79	0.92		0.17	0.28	0.23	0.26
Trimming Tests (TT)	13576579	14206215	13501537	13807982		8698072	6419471	6401545	6412780
Hit TT/Total TT	0.02	0.02	0.02	0.02		0.03	0.04	0.04	0.04
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	3.91	4.31	4.95	4.95		3.90	4.18	4.96	4.99
Rendering Time, Sec	440.00	394.00	412.00	811.00		264.00	**193.00**	194.00	677.00
MA per HRay	124.54	114.20	114.00	113.47		67.19	42.74	42.45	42.63
BX per HRay	77.01	73.56	70.29	69.90		40.54	27.06	27.03	26.93
BH per HRay	34.47	33.11	33.69	33.31		21.75	15.24	15.36	15.26
BZ per HRay	66.80	64.39	65.54	64.78		41.68	28.93	29.23	29.00
Avg Its/Any BZ Call	1.81	1.71	1.75	1.72		1.83	1.72	1.73	1.72
Avg Its/Scs BZ Call	1.82	1.72	1.76	1.73		1.84	1.72	1.74	1.72
ADS Memory, Mb	0.40	0.80	0.64	0.73		0.15	0.21	0.18	0.19
Trimming Tests (TT)	13572113	13866369	13386011	13598510		8819422	6396271	6391138	6397321
Hit TT/Total TT	0.02	0.02	0.02	0.02		0.03	0.04	0.04	0.04



File Name:	**DBE.wrl**
Number of Patches:	**2010**
Number of Control Points:	**56358**
Average Surface Degree:	**7.48**
Number of Trimming Contours:	**665**
Screen Coverage:	**38%**

Table 5.12: Comparison of ray tracing trimmed rational Bézier surfaces methods (DBE.wrl model. *Model courtesy of DaimlerChrysler AG*).

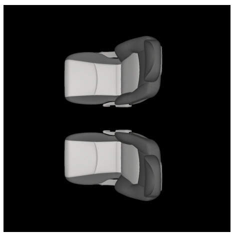
	ADS Level								
	w/o ADS reduction					with ADS reduction			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	6.51	7.03	8.19	9.23		6.58	7.29	8.46	8.44
Rendering Time, Sec	15.00	14.90	16.30	16.10		13.40	**10.80**	11.90	11.70
MA per HRay	12.82	12.90	12.90	12.90		8.51	7.17	7.17	7.17
BX per HRay	10.20	10.30	10.29	10.30		6.62	5.57	5.57	5.57
BH per HRay	4.02	4.02	4.02	4.02		3.13	2.77	2.77	2.77
BZ per HRay	3.96	3.89	3.98	3.99		3.06	2.59	2.67	2.66
Avg Its/Any BZ Call	2.95	2.72	3.04	3.01		2.94	2.70	3.08	3.02
Avg Its/Scs BZ Call	4.41	4.00	4.56	4.53		4.51	4.04	4.56	4.51
ADS Memory, Mb	0.42	0.95	0.76	0.79		0.29	0.60	0.50	0.52
Trimming Tests (TT)	256262	256170	256129	256159		178944	150012	167759	165455
Hit TT/Total TT	0.44	0.44	0.44	0.44		0.55	0.61	0.62	0.63
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	6.58	6.97	8.16	8.10		6.64	7.37	8.48	8.55
Rendering Time, Sec	14.80	13.10	14.80	14.70		12.00	**9.61**	11.10	11.00
MA per HRay	14.94	13.17	16.04	15.90		10.60	8.39	10.56	10.61
BX per HRay	10.44	9.27	10.49	10.42		7.68	6.37	7.36	7.41
BH per HRay	3.83	3.52	4.02	4.02		3.03	2.45	2.74	2.71
BZ per HRay	3.83	3.52	4.02	4.02		3.03	2.45	2.74	2.71
Avg Its/Any BZ Call	2.97	2.77	3.00	2.98		2.96	2.77	3.05	2.99
Avg Its/Scs BZ Call	4.42	4.00	4.57	4.54		4.51	4.03	4.57	4.52
ADS Memory, Mb	0.31	0.63	0.51	0.53		0.21	0.38	0.32	0.34
Trimming Tests (TT)	249638	237608	255258	255253		178296	145303	170098	167181
Hit TT/Total TT	0.44	0.45	0.44	0.44		0.54	0.62	0.62	0.63
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	6.62	7.00	8.28	8.98		6.62	7.19	8.45	8.44
Rendering Time, Sec	15.50	15.50	16.40	16.00		14.80	**10.60**	11.60	11.50
MA per HRay	12.57	12.63	12.63	12.63		8.35	7.01	7.01	7.01
BX per HRay	10.05	10.13	10.13	10.13		6.53	5.46	5.46	5.46
BH per HRay	3.96	3.95	3.95	3.95		3.08	2.72	2.72	2.72
BZ per HRay	6.24	6.14	6.28	6.29		4.59	3.72	3.88	3.85
Avg Its/Any BZ Call	1.51	1.42	1.53	1.49		1.59	1.49	1.62	1.60
Avg Its/Scs BZ Call	1.50	1.40	1.53	1.49		1.58	1.48	1.62	1.60
ADS Memory, Mb	0.74	1.58	1.27	1.32		0.50	0.98	0.82	0.85
Trimming Tests (TT)	420289	451757	415624	418349		275450	236797	239694	236875
Hit TT/Total TT	0.28	0.26	0.28	0.28		0.37	0.41	0.45	0.46
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	6.55	7.00	8.23	8.37		6.62	7.30	8.46	8.49
Rendering Time, Sec	16.10	13.40	17.90	15.70		12.10	**9.10**	11.10	10.40
MA per HRay	14.60	12.80	15.66	15.51		10.37	8.17	10.32	10.36
BX per HRay	10.25	9.06	10.30	10.22		7.55	6.25	7.23	7.27
BH per HRay	3.75	3.43	3.95	3.93		2.97	2.39	2.68	2.66
BZ per HRay	5.99	5.44	6.35	6.33		4.51	3.44	3.98	3.93
Avg Its/Any BZ Call	1.51	1.41	1.54	1.51		1.58	1.48	1.62	1.60
Avg Its/Scs BZ Call	1.49	1.39	1.54	1.50		1.57	1.45	1.62	1.60
ADS Memory, Mb	0.63	1.25	1.03	1.06		0.42	0.76	0.65	0.67
Trimming Tests (TT)	407969	412008	414436	416956		274809	229040	245178	241363
Hit TT/Total TT	0.29	0.29	0.29	0.29		0.38	0.43	0.46	0.47



File Name:	**Middle_console.wrl**
Number of Patches:	**3159**
Number of Control Points:	**97520**
Average Surface Degree:	**5.49**
Number of Trimming Contours:	**1265**
Screen Coverage:	**16%**

Table 5.13: Comparison of ray tracing trimmed rational Bézier surfaces methods (Middle_console.wrl model. *Model courtesy of DaimlerChrysler AG*).

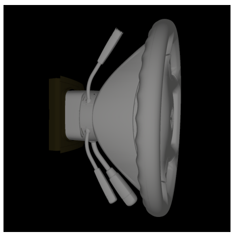
	ADS Level								
	w/o ADS reduction					with ADS reduction			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	8.06	9.21	12.12	12.39		8.49	9.83	12.93	13.16
Rendering Time, Sec	21.80	21.10	21.20	20.00		20.70	**14.50**	14.80	15.10
MA per HRay	9.15	9.11	9.12	9.11		7.64	6.55	6.55	6.55
BX per HRay	7.11	7.06	7.07	7.06		5.89	5.17	5.17	5.17
BH per HRay	3.67	3.68	3.68	3.68		3.16	2.65	2.65	2.65
BZ per HRay	3.71	3.77	3.73	3.74		3.16	2.64	2.61	2.62
Avg Its/Any BZ Call	3.19	2.78	2.91	2.85		3.34	2.93	3.06	3.00
Avg Its/Scs BZ Call	4.98	4.33	4.50	4.43		5.03	4.36	4.50	4.44
ADS Memory, Mb	0.90	2.02	1.54	1.72		0.72	1.52	1.19	1.31
Trimming Tests (TT)	302141	300429	300475	300425		241317	192598	199488	196000
Hit TT/Total TT	0.48	0.48	0.48	0.48		0.48	0.59	0.59	0.59
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	8.13	9.22	12.84	12.38		8.43	9.88	13.02	13.04
Rendering Time, Sec	20.70	20.50	19.90	20.70		18.90	**14.10**	14.40	16.50
MA per HRay	14.01	13.81	14.85	13.92		11.60	9.58	9.46	9.44
BX per HRay	9.35	9.68	9.52	9.23		7.80	6.89	6.43	6.61
BH per HRay	3.90	3.64	3.84	3.69		3.33	2.61	2.63	2.62
BZ per HRay	3.90	3.64	3.84	3.69		3.33	2.61	2.63	2.62
Avg Its/Any BZ Call	3.03	2.76	2.75	2.78		3.17	2.89	2.99	2.95
Avg Its/Scs BZ Call	4.98	4.31	4.49	4.41		5.02	4.35	4.49	4.43
ADS Memory, Mb	0.66	1.33	1.05	1.15		0.53	0.98	0.79	0.86
Trimming Tests (TT)	299340	288056	290540	289408		239881	187630	196202	191978
Hit TT/Total TT	0.48	0.49	0.48	0.49		0.48	0.59	0.60	0.59
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	8.09	9.09	12.05	12.21		8.39	9.91	13.21	13.02
Rendering Time, Sec	23.70	21.00	21.70	21.30		20.30	**13.70**	15.20	14.70
MA per HRay	9.03	8.94	8.94	8.94		7.55	6.42	6.42	6.42
BX per HRay	7.04	6.96	6.96	6.96		5.84	5.09	5.09	5.09
BH per HRay	3.64	3.62	3.62	3.62		3.13	2.61	2.61	2.61
BZ per HRay	5.57	5.63	5.57	5.57		4.60	3.60	3.56	3.58
Avg Its/Any BZ Call	1.74	1.61	1.65	1.63		1.75	1.62	1.66	1.63
Avg Its/Scs BZ Call	1.76	1.62	1.66	1.63		1.76	1.61	1.66	1.62
ADS Memory, Mb	1.56	3.35	2.59	2.87		1.25	2.50	1.97	2.17
Trimming Tests (TT)	460779	513985	487265	497439		358425	300991	295926	299356
Hit TT/Total TT	0.33	0.31	0.32	0.32		0.34	0.40	0.43	0.41
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	8.08	9.18	12.40	12.32		8.44	9.79	13.29	13.16
Rendering Time, Sec	24.10	20.50	23.50	21.40		20.00	13.90	15.60	**13.70**
MA per HRay	13.82	13.47	14.51	13.59		11.45	9.34	9.22	9.21
BX per HRay	9.24	9.49	9.36	9.06		7.72	6.76	6.32	6.49
BH per HRay	3.85	3.56	3.76	3.62		3.30	2.56	2.58	2.56
BZ per HRay	5.78	5.34	5.64	5.41		4.82	3.52	3.57	3.54
Avg Its/Any BZ Call	1.76	1.61	1.66	1.62		1.77	1.62	1.67	1.63
Avg Its/Scs BZ Call	1.77	1.60	1.67	1.62		1.77	1.61	1.66	1.62
ADS Memory, Mb	1.33	2.66	2.10	2.30		1.07	1.97	1.57	1.72
Trimming Tests (TT)	454498	490172	466777	475523		357916	297631	294984	297032
Hit TT/Total TT	0.35	0.34	0.35	0.35		0.35	0.44	0.46	0.44



File Name:	**Seats.wrl**
Number of Patches:	**6704**
Number of Control Points:	**243606**
Average Surface Degree:	**5.51**
Number of Trimming Contours:	**1562**
Screen Coverage:	**21%**

Table 5.14: Comparison of ray tracing trimmed rational Bézier surfaces methods (Seats.wrl model. *Model courtesy of DaimlerChrysler AG*).

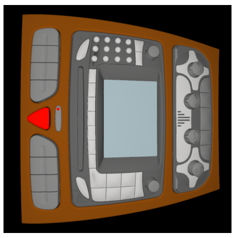
	ADS Level								
	w/o ADS reduction					with ADS reduction			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	12.60	15.12	20.88	22.19		16.94	24.70	31.17	31.21
Rendering Time, Sec	138.00	106.00	106.00	112.00		134.00	92.00	**91.70**	93.50
MA per HRay	43.64	43.94	43.93	43.93		40.07	36.16	36.15	36.15
BX per HRay	35.11	35.38	35.37	35.37		32.71	28.77	28.77	28.77
BH per HRay	22.12	22.24	22.24	22.24		20.75	19.03	19.03	19.02
BZ per HRay	21.89	17.14	17.11	17.10		20.47	14.38	14.43	14.40
Avg Its/Any BZ Call	0.95	0.96	0.99	0.98		0.89	0.85	0.88	0.87
Avg Its/Scs BZ Call	4.93	4.44	4.59	4.55		4.92	4.42	4.58	4.52
ADS Memory, Mb	2.46	5.55	4.07	4.79		2.11	4.67	3.44	4.04
Trimming Tests (TT)	538214	538239	538300	538286		397715	280210	324495	300329
Hit TT/Total TT	0.47	0.47	0.47	0.47		0.53	0.58	0.63	0.60
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	12.69	15.09	21.15	21.52		17.10	24.68	32.19	31.24
Rendering Time, Sec	160.00	90.40	93.60	93.70		152.00	**79.70**	83.00	85.60
MA per HRay	79.64	46.77	48.90	48.99		76.10	39.97	39.97	39.79
BX per HRay	51.25	31.18	33.15	32.86		49.82	26.65	26.96	26.76
BH per HRay	26.73	14.70	15.08	15.06		24.98	12.85	13.03	12.98
BZ per HRay	26.73	14.70	15.08	15.06		24.98	12.85	13.03	12.98
Avg Its/Any BZ Call	0.83	1.02	1.06	1.06		0.80	0.90	0.94	0.93
Avg Its/Scs BZ Call	4.93	4.47	4.61	4.56		4.95	4.42	4.59	4.52
ADS Memory, Mb	1.83	3.66	2.78	3.20		1.57	3.05	2.32	2.68
Trimming Tests (TT)	526031	482381	504442	509046		396123	265050	314838	291752
Hit TT/Total TT	0.49	0.50	0.48	0.48		0.54	0.58	0.62	0.59
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	12.65	15.27	23.85	22.70		17.14	24.65	30.81	31.38
Rendering Time, Sec	153.00	132.00	122.00	123.00		144.00	**102.00**	104.00	109.00
MA per HRay	44.49	43.79	43.80	43.80		40.89	36.06	36.07	36.07
BX per HRay	35.37	35.21	35.22	35.21		32.98	28.65	28.66	28.66
BH per HRay	22.28	22.15	22.15	22.15		20.94	18.97	18.98	18.97
BZ per HRay	26.49	21.23	21.24	21.18		23.94	16.70	16.87	16.78
Avg Its/Any BZ Call	1.40	1.41	1.43	1.42		1.36	1.37	1.38	1.38
Avg Its/Scs BZ Call	1.95	1.80	1.84	1.82		1.98	1.89	1.91	1.89
ADS Memory, Mb	4.29	9.21	6.86	7.99		3.68	7.71	5.76	6.71
Trimming Tests (TT)	834593	961032	901626	915906		574912	443770	477816	456370
Hit TT/Total TT	0.32	0.30	0.31	0.31		0.38	0.40	0.46	0.43
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	12.62	15.14	21.19	21.47		17.15	24.78	31.69	31.47
Rendering Time, Sec	190.00	103.00	104.00	104.00		178.00	**88.20**	92.70	91.00
MA per HRay	81.70	46.56	48.77	48.81		78.33	39.88	39.92	39.69
BX per HRay	51.87	31.03	33.03	32.72		50.59	26.57	26.91	26.67
BH per HRay	27.02	14.59	14.98	14.95		25.27	12.77	12.98	12.91
BZ per HRay	31.33	17.87	18.68	18.58		28.51	14.79	15.24	15.10
Avg Its/Any BZ Call	1.36	1.43	1.43	1.42		1.33	1.38	1.39	1.38
Avg Its/Scs BZ Call	2.00	1.83	1.81	1.79		2.04	1.89	1.88	1.85
ADS Memory, Mb	3.66	7.31	5.56	6.40		3.14	6.10	4.65	5.35
Trimming Tests (TT)	798466	815785	821077	842629		567113	419355	468256	449629
Hit TT/Total TT	0.35	0.35	0.34	0.34		0.41	0.43	0.47	0.44



File Name:	**Steering_wheel.wrl**
Number of Patches:	**18432**
Number of Control Points:	**547470**
Average Surface Degree:	**4.62**
Number of Trimming Contours:	**1414**
Screen Coverage:	**31%**

Table 5.15: Comparison of ray tracing trimmed rational Bézier surfaces methods (Steering_wheel.wrl model. *Model courtesy of DaimlerChrysler AG*).

	ADS Level								
	w/o ADS reduction					with ADS reduction			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	64.33	73.78	86.27	86.19		68.57	84.36	99.73	99.43
Rendering Time, Sec	2180.00	1940.00	2020.00	2330.00		593.00	**407.00**	409.00	414.00
MA per HRay	240.38	240.52	240.46	240.46		54.60	46.27	46.26	46.26
BX per HRay	139.06	139.24	139.19	139.20		30.52	23.95	23.95	23.95
BH per HRay	70.53	70.53	70.52	70.52		14.03	12.70	12.70	12.70
BZ per HRay	73.90	75.00	74.52	74.36		14.93	13.21	13.40	13.39
Avg Its/Any BZ Call	3.43	2.80	3.15	3.11		3.32	2.42	2.46	2.44
Avg Its/Scs BZ Call	5.42	4.71	5.29	5.21		6.50	5.27	5.39	5.34
ADS Memory, Mb	8.63	19.49	8.99	10.80		3.26	7.02	4.75	5.25
Trimming Tests (TT)	18793113	18795516	18790137	18790342		2925325	2347735	2428541	2398656
Hit TT/Total TT	0.04	0.04	0.04	0.04		0.23	0.28	0.30	0.29
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	66.34	73.40	85.05	88.20		69.64	83.28	101.17	99.27
Rendering Time, Sec	2300.00	1760.00	2250.00	2020.00		673.00	**425.00**	469.00	471.00
MA per HRay	437.02	408.76	416.25	424.19		92.41	77.75	90.64	90.56
BX per HRay	179.30	157.05	175.17	173.07		31.11	23.66	26.44	26.76
BH per HRay	80.42	71.10	84.90	83.17		16.42	13.61	15.85	15.80
BZ per HRay	80.42	71.10	84.90	83.17		16.42	13.61	15.85	15.80
Avg Its/Any BZ Call	3.14	2.60	3.00	2.96		3.03	2.29	2.47	2.45
Avg Its/Scs BZ Call	5.26	4.65	5.14	5.07		6.26	5.34	5.33	5.30
ADS Memory, Mb	6.41	12.83	6.62	7.69		2.42	4.55	3.21	3.50
Trimming Tests (TT)	18690302	16573523	20909020	20455658		2983193	2256364	2946645	2908823
Hit TT/Total TT	0.04	0.04	0.05	0.05		0.24	0.27	0.31	0.30
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	64.86	72.00	85.02	85.37		68.93	84.09	99.73	99.13
Rendering Time, Sec	2180.00	2530.00	2610.00	2600.00		853.00	**798.00**	803.00	802.00
MA per HRay	241.20	239.84	239.96	239.95		54.84	46.12	46.13	46.13
BX per HRay	139.41	138.78	138.87	138.86		30.58	23.84	23.84	23.84
BH per HRay	70.60	70.28	70.32	70.32		14.10	12.68	12.68	12.68
BZ per HRay	123.17	126.41	125.63	125.28		22.73	19.53	19.98	19.94
Avg Its/Any BZ Call	1.45	1.34	1.56	1.53		1.55	1.46	1.50	1.50
Avg Its/Scs BZ Call	1.56	1.42	1.69	1.66		1.82	1.68	1.74	1.73
ADS Memory, Mb	15.05	32.31	15.61	18.49		5.68	11.57	7.96	8.75
Trimming Tests (TT)	36731487	39090075	35737779	36067152		4721752	4107273	4108018	4085130
Hit TT/Total TT	0.02	0.02	0.02	0.02		0.15	0.17	0.18	0.18
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	64.83	72.08	84.54	87.54		68.57	84.61	100.39	100.09
Rendering Time, Sec	2500.00	1990.00	2300.00	2270.00		881.00	**517.00**	553.00	551.00
MA per HRay	439.58	407.94	415.75	423.60		93.21	77.48	90.44	90.36
BX per HRay	179.58	156.18	174.67	172.55		31.17	23.48	26.31	26.61
BH per HRay	80.32	70.64	84.61	82.82		16.45	13.54	15.79	15.73
BZ per HRay	129.06	115.20	142.42	138.89		23.79	18.95	22.92	22.80
Avg Its/Any BZ Call	1.43	1.31	1.51	1.49		1.48	1.39	1.40	1.40
Avg Its/Scs BZ Call	1.55	1.41	1.64	1.61		1.73	1.66	1.61	1.60
ADS Memory, Mb	12.83	25.65	13.25	15.38		4.85	9.10	6.42	7.00
Trimming Tests (TT)	36197929	33521075	40161529	39410876		4614313	3561631	4581232	4541580
Hit TT/Total TT	0.02	0.02	0.03	0.03		0.16	0.18	0.21	0.21



File Name:	**Comand.wrl**
Number of Patches:	**64659**
Number of Control Points:	**1261136**
Average Surface Degree:	**7.58**
Number of Trimming Contours:	**8651**
Screen Coverage:	**71%**

Table 5.16: Comparison of ray tracing trimmed rational Bézier surfaces methods (Comand.wrl model. *Model courtesy of DaimlerChrysler AG*).

	ADS Level								
	w/o ADS reduction					with ADS reduction			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	240.66	269.62	324.51	332.16		263.93	320.59	382.15	384.46
Rendering Time, Sec	1230.00	1020.00	1040.00	1030.00		353.00	**249.00**	252.00	**249.00**
MA per HRay	87.79	87.68	87.68	87.69		30.27	26.23	26.22	26.23
BX per HRay	46.25	46.39	46.39	46.40		18.31	15.70	15.70	15.70
BH per HRay	25.65	25.58	25.58	25.58		10.72	10.09	10.09	10.09
BZ per HRay	27.45	27.33	27.32	27.26		10.17	8.66	8.69	8.68
Avg Its/Any BZ Call	2.80	2.18	2.32	2.31		2.45	1.79	1.87	1.86
Avg Its/Scs BZ Call	5.59	4.81	5.18	5.14		5.56	4.52	4.71	4.67
ADS Memory, Mb	30.97	69.89	44.71	51.38		20.30	44.01	31.79	35.94
Trimming Tests (TT)	6444241	6445569	6446165	6446814		1878214	1452515	1572237	1539296
Hit TT/Total TT	0.12	0.12	0.12	0.12		0.34	0.41	0.44	0.43
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	241.31	270.03	326.48	328.00		265.01	323.72	385.24	387.46
Rendering Time, Sec	1360.00	859.00	1010.00	946.00		470.00	**255.00**	267.00	266.00
MA per HRay	194.10	150.02	150.59	150.89		69.91	39.70	42.45	42.13
BX per HRay	66.78	44.59	45.98	45.73		27.18	15.39	18.10	18.10
BH per HRay	34.28	25.11	27.38	27.01		14.72	9.01	9.63	9.53
BZ per HRay	34.28	25.11	27.38	27.01		14.72	9.01	9.63	9.53
Avg Its/Any BZ Call	2.32	1.98	2.15	2.14		1.94	1.70	1.80	1.80
Avg Its/Scs BZ Call	5.64	4.71	4.98	4.94		5.67	4.56	4.73	4.70
ADS Memory, Mb	23.00	46.01	31.12	35.07		15.08	28.61	21.39	23.84
Trimming Tests (TT)	6032569	5300839	6069612	5990560		1942617	1393122	1673635	1628957
Hit TT/Total TT	0.13	0.14	0.13	0.13		0.33	0.40	0.43	0.42
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	241.55	271.37	327.58	326.15		267.38	320.67	381.34	384.33
Rendering Time, Sec	1150.00	1710.00	1650.00	1690.00		**407.00**	682.00	688.00	679.00
MA per HRay	89.39	87.86	87.87	87.86		31.11	26.31	26.32	26.32
BX per HRay	47.04	46.45	46.45	46.44		18.78	15.73	15.74	15.74
BH per HRay	26.00	25.63	25.63	25.63		10.96	10.14	10.14	10.14
BZ per HRay	38.10	39.53	39.52	39.41		13.88	11.40	11.53	11.47
Avg Its/Any BZ Call	1.30	1.32	1.40	1.38		1.44	1.41	1.46	1.45
Avg Its/Scs BZ Call	1.47	1.48	1.61	1.58		1.73	1.63	1.71	1.68
ADS Memory, Mb	53.97	115.90	75.83	86.44		35.37	72.61	53.18	59.77
Trimming Tests (TT)	10036552	11750633	10867757	11010552		3046684	2559047	2570713	2557536
Hit TT/Total TT	0.08	0.07	0.08	0.08		0.22	0.25	0.28	0.28
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	241.48	270.06	327.06	332.07		277.66	322.46	385.28	392.61
Rendering Time, Sec	1260.00	1030.00	1160.00	1100.00		521.00	**341.00**	363.00	353.00
MA per HRay	197.67	150.31	151.00	151.15		71.97	39.93	42.74	42.37
BX per HRay	67.63	44.58	46.08	45.80		27.69	15.41	18.16	18.14
BH per HRay	34.61	25.15	27.46	27.08		14.96	9.06	9.69	9.59
BZ per HRay	45.03	35.68	40.01	39.31		18.86	11.55	12.69	12.50
Avg Its/Any BZ Call	1.28	1.29	1.36	1.35		1.38	1.37	1.41	1.40
Avg Its/Scs BZ Call	1.50	1.47	1.56	1.53		1.77	1.65	1.68	1.66
ADS Memory, Mb	46.01	92.01	62.25	70.13		30.15	57.21	42.78	47.68
Trimming Tests (TT)	9705452	10017606	10947502	10903042		3087225	2371021	2688445	2657886
Hit TT/Total TT	0.09	0.09	0.08	0.08		0.23	0.27	0.29	0.29

File Name:	**W203_Interieur_1.wrl**
Number of Patches:	**231926**
Number of Control Points:	**5119047**
Average Surface Degree:	**4.94**
Number of Trimming Contours:	**38318**
Screen Coverage:	**97%**

Table 5.17: Comparison of ray tracing trimmed rational Bézier surfaces methods (W203_Interieur_1.wrl model. *Model courtesy of DaimlerChrysler AG*).

	ADS Level								
	w/o ADS reduction					**with ADS reduction**			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	242.60	268.70	336.08	325.24		263.33	320.96	398.38	383.55
Rendering Time, Sec	1000.00	670.00	681.00	672.00		322.00	228.00	227.00	**225.00**
MA per HRay	46.94	46.66	46.66	46.66		19.78	16.38	16.38	16.38
BX per HRay	27.74	27.62	27.63	27.63		12.22	10.11	10.11	10.11
BH per HRay	15.48	15.44	15.45	15.45		6.96	6.16	6.16	6.16
BZ per HRay	16.40	17.15	16.78	16.97		7.22	6.41	6.20	6.37
Avg Its/Any BZ Call	2.97	2.24	2.32	2.29		3.03	2.25	2.38	2.32
Avg Its/Scs BZ Call	5.45	4.81	4.91	4.89		5.96	4.86	5.00	4.97
ADS Memory, Mb	30.97	69.89	44.71	51.38		20.30	44.01	31.79	35.94
Trimming Tests (TT)	4011288	4006899	4008056	4007850		1779103	1384461	1457716	1434802
Hit TT/Total TT	0.17	0.17	0.17	0.17		0.34	0.41	0.43	0.42
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	240.70	270.65	324.88	330.22		266.49	322.84	393.17	392.38
Rendering Time, Sec	827.00	590.00	658.00	657.00		359.00	**226.00**	234.00	237.00
MA per HRay	94.19	75.91	78.01	76.38		40.27	25.27	26.29	26.26
BX per HRay	38.28	27.63	29.34	28.85		17.24	10.64	12.16	12.36
BH per HRay	18.97	15.70	17.64	17.47		8.65	6.32	6.54	6.57
BZ per HRay	18.97	15.70	17.64	17.47		8.65	6.32	6.54	6.57
Avg Its/Any BZ Call	2.65	2.17	2.60	2.60		2.66	2.23	2.31	2.29
Avg Its/Scs BZ Call	5.47	4.76	5.34	5.34		5.93	4.88	5.00	4.96
ADS Memory, Mb	23.00	46.01	31.12	35.07		15.08	28.61	21.39	23.84
Trimming Tests (TT)	3842163	3484472	4491695	4450763		1775454	1340873	1491241	1465699
Hit TT/Total TT	0.18	0.20	0.16	0.16		0.34	0.41	0.43	0.42
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	240.47	271.86	329.76	332.95		265.10	320.78	397.80	385.31
Rendering Time, Sec	959.00	1750.00	1430.00	1430.00		**490.00**	761.00	773.00	762.00
MA per HRay	46.96	46.26	46.34	46.26		19.83	16.25	16.29	16.25
BX per HRay	27.74	27.41	27.44	27.42		12.24	10.04	10.06	10.04
BH per HRay	15.49	15.36	15.37	15.36		6.98	6.13	6.14	6.13
BZ per HRay	23.35	25.20	24.49	24.86		10.55	9.15	8.78	9.12
Avg Its/Any BZ Call	1.49	1.47	1.51	1.51		1.75	1.69	1.71	1.71
Avg Its/Scs BZ Call	1.68	1.64	1.71	1.70		1.96	1.83	1.87	1.86
ADS Memory, Mb	53.97	115.90	75.83	86.44		35.37	72.61	53.18	59.77
Trimming Tests (TT)	6490497	7750110	7028542	7266674		2868700	2654907	2468495	2620743
Hit TT/Total TT	0.12	0.10	0.11	0.11		0.23	0.24	0.28	0.25
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	242.77	273.50	332.50	333.08		271.66	324.55	392.87	387.57
Rendering Time, Sec	1230.00	823.00	925.00	939.00		555.00	**354.00**	355.00	355.00
MA per HRay	94.49	75.29	77.57	75.74		40.48	25.03	26.13	26.05
BX per HRay	38.31	27.41	29.20	28.66		17.27	10.54	12.09	12.27
BH per HRay	19.00	15.60	17.57	17.38		8.68	6.28	6.51	6.53
BZ per HRay	25.98	22.65	26.37	26.12		12.18	8.75	9.09	9.21
Avg Its/Any BZ Call	1.48	1.46	1.58	1.57		1.72	1.63	1.65	1.64
Avg Its/Scs BZ Call	1.73	1.65	1.80	1.78		1.99	1.82	1.85	1.83
ADS Memory, Mb	46.01	92.01	62.25	70.13		30.15	57.21	42.78	47.68
Trimming Tests (TT)	6238403	6727341	6944073	7062697		2804298	2463830	2461661	2585595
Hit TT/Total TT	0.13	0.13	0.12	0.12		0.25	0.28	0.30	0.29

File Name:	**W203_Interieur_2.wrl**
Number of Patches:	**231926**
Number of Control Points:	**5119047**
Average Surface Degree:	**4.94**
Number of Trimming Contours:	**38318**
Screen Coverage:	**100%**

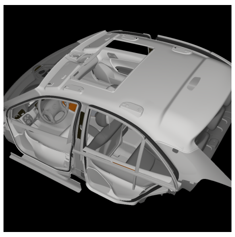
Table 5.18: Comparison of ray tracing trimmed rational Bézier surfaces methods (W203_Interieur_2.wrl model. *Model courtesy of DaimlerChrysler AG*).

	ADS Level								
	w/o ADS reduction					with ADS reduction			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	315.70	345.41	397.82	403.78		342.11	407.55	464.68	464.29
Rendering Time, Sec	1300.00	964.00	1680.00	977.00		325.00	**274.00**	294.00	316.00
MA per HRay	98.65	98.87	98.87	98.87		34.57	24.04	24.04	24.04
BX per HRay	31.17	31.33	31.33	31.33		13.05	11.83	11.83	11.83
BH per HRay	22.90	22.85	22.85	22.84		9.14	8.34	8.34	8.34
BZ per HRay	25.30	25.57	25.52	25.54		9.43	8.28	8.30	8.31
Avg Its/Any BZ Call	1.92	1.46	1.51	1.50		1.95	1.66	1.75	1.73
Avg Its/Scs BZ Call	6.88	5.80	6.02	6.00		5.07	4.83	5.01	4.98
ADS Memory, Mb	30.97	69.89	44.71	51.38		20.30	44.01	31.79	35.94
Trimming Tests (TT)	3435472	3432088	3431944	3431908		1760838	1468778	1560658	1533789
Hit TT/Total TT	0.25	0.25	0.25	0.25		0.40	0.48	0.49	0.48
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	315.82	347.78	409.09	482.24		339.93	401.68	463.27	467.03
Rendering Time, Sec	1180.00	995.00	917.00	900.00		396.00	**275.00**	330.00	713.00
MA per HRay	178.20	138.98	139.17	145.28		51.58	36.89	37.90	37.53
BX per HRay	44.01	34.06	36.38	35.32		18.76	12.44	15.17	14.64
BH per HRay	27.53	24.14	25.55	25.07		11.01	8.51	9.19	8.98
BZ per HRay	27.53	24.14	25.55	25.07		11.01	8.51	9.19	8.98
Avg Its/Any BZ Call	1.84	1.33	1.44	1.42		1.82	1.49	1.63	1.59
Avg Its/Scs BZ Call	6.79	5.69	5.78	5.80		4.97	4.78	4.94	4.91
ADS Memory, Mb	23.00	46.01	31.12	35.07		15.08	28.61	21.39	23.84
Trimming Tests (TT)	3442876	2930974	3358710	3213833		1821823	1345474	1620231	1521348
Hit TT/Total TT	0.25	0.26	0.26	0.26		0.41	0.49	0.50	0.49
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	313.77	343.63	399.29	401.07		339.56	400.53	464.66	463.93
Rendering Time, Sec	6160.00	891.00	907.00	889.00		1640.00	**253.00**	255.00	260.00
MA per HRay	99.44	99.35	99.35	99.34		34.91	24.13	24.13	24.13
BX per HRay	31.47	31.51	31.51	31.51		13.21	11.89	11.89	11.89
BH per HRay	23.15	23.01	23.01	23.01		9.27	8.38	8.39	8.39
BZ per HRay	30.35	30.53	30.49	30.50		12.13	10.23	10.34	10.33
Avg Its/Any BZ Call	1.56	1.29	1.33	1.32		1.52	1.34	1.39	1.38
Avg Its/Scs BZ Call	1.91	1.68	1.81	1.79		1.70	1.60	1.70	1.67
ADS Memory, Mb	53.97	115.90	75.83	86.44		35.37	72.61	53.18	59.77
Trimming Tests (TT)	5015115	5246715	4908896	4997860		2780718	2364666	2369697	2376691
Hit TT/Total TT	0.18	0.18	0.19	0.19		0.27	0.32	0.34	0.33
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	314.92	346.24	402.39	410.61		341.95	401.45	463.20	471.61
Rendering Time, Sec	6160.00	809.00	870.00	846.00		1670.00	**256.00**	273.00	575.00
MA per HRay	179.82	139.58	139.85	145.96		52.10	37.00	38.05	37.65
BX per HRay	44.45	34.25	36.61	35.54		18.95	12.48	15.24	14.69
BH per HRay	27.78	24.30	25.72	25.23		11.10	8.53	9.22	9.00
BZ per HRay	32.91	28.45	31.02	30.14		13.99	10.24	11.43	11.08
Avg Its/Any BZ Call	1.54	1.26	1.34	1.32		1.50	1.31	1.39	1.36
Avg Its/Scs BZ Call	1.91	1.67	1.79	1.76		1.72	1.60	1.68	1.66
ADS Memory, Mb	46.01	92.01	62.25	70.13		30.15	57.21	42.78	47.68
Trimming Tests (TT)	5061270	4574255	5081917	4907368		2890674	2161713	2482365	2388389
Hit TT/Total TT	0.18	0.20	0.20	0.19		0.28	0.35	0.36	0.35

File Name:	**W203_Interieur_3.wrl**
Number of Patches:	**231926**
Number of Control Points:	**5119047**
Average Surface Degree:	**4.94**
Number of Trimming Contours:	**38318**
Screen Coverage:	**99%**

Table 5.19: Comparison of ray tracing trimmed rational Bézier surfaces methods (W203_Interieur_3.wrl model. *Model courtesy of DaimlerChrysler AG*).

	ADS Level								
	w/o ADS reduction					with ADS reduction			
	0	1	A1	A2		0	1	A1	A2
Bezier Clipping Method (with ADS Hierarchy)									
Preproc. Time, Sec	244.54	269.26	329.65	326.71		262.80	326.56	399.96	383.67
Rendering Time, Sec	438.00	317.00	312.00	317.00		153.00	**95.70**	103.00	98.70
MA per HRay	49.22	49.22	49.18	49.18		24.23	21.40	21.39	21.39
BX per HRay	30.96	31.05	31.03	31.03		16.60	16.02	16.01	16.01
BH per HRay	20.06	20.04	20.04	20.04		10.37	9.31	9.31	9.31
BZ per HRay	20.95	17.01	17.02	17.02		9.44	6.87	7.09	7.04
Avg Its/Any BZ Call	1.94	1.74	1.80	1.77		2.02	1.78	1.91	1.87
Avg Its/Scs BZ Call	4.93	4.44	4.63	4.56		4.61	3.95	4.18	4.11
ADS Memory, Mb	30.97	69.89	44.71	51.38		20.30	44.01	31.79	35.94
Trimming Tests (TT)	1889237	1887179	1886255	1886211		1028748	754490	917189	885536
Hit TT/Total TT	0.27	0.27	0.27	0.27		0.35	0.42	0.46	0.46
Bezier Clipping Method (ADS Leaves Only)									
Preproc. Time, Sec	242.61	271.82	328.11	329.38		264.48	324.75	386.97	384.47
Rendering Time, Sec	464.00	295.00	309.00	311.00		168.00	**95.10**	101.00	99.70
MA per HRay	107.69	71.17	75.09	74.27		44.58	24.10	29.05	27.95
BX per HRay	41.62	27.71	28.05	27.72		22.23	12.14	15.55	15.18
BH per HRay	22.91	16.42	17.43	17.16		10.79	6.47	7.35	7.13
BZ per HRay	22.91	16.42	17.43	17.16		10.79	6.47	7.35	7.13
Avg Its/Any BZ Call	1.88	1.74	1.74	1.74		1.84	1.81	1.85	1.84
Avg Its/Scs BZ Call	5.06	4.49	4.66	4.59		4.62	3.98	4.20	4.14
ADS Memory, Mb	23.00	46.01	31.12	35.07		15.08	28.61	21.39	23.84
Trimming Tests (TT)	1906322	1802352	1854050	1844168		1041324	728420	923448	885575
Hit TT/Total TT	0.27	0.27	0.27	0.27		0.35	0.42	0.45	0.45
Newton's Method (with ADS Hierarchy)									
Preproc. Time, Sec	240.74	268.91	324.51	328.15		263.25	322.65	395.92	386.51
Rendering Time, Sec	884.00	384.00	383.00	383.00		311.00	**147.00**	155.00	155.00
MA per HRay	49.00	48.54	48.58	48.55		23.98	21.00	21.02	21.01
BX per HRay	30.87	30.66	30.68	30.66		16.45	15.76	15.77	15.76
BH per HRay	20.05	19.81	19.82	19.81		10.27	9.14	9.14	9.14
BZ per HRay	26.99	22.84	22.92	22.91		12.99	9.23	9.74	9.62
Avg Its/Any BZ Call	1.38	1.33	1.40	1.38		1.49	1.37	1.46	1.44
Avg Its/Scs BZ Call	1.62	1.49	1.61	1.57		1.63	1.44	1.57	1.53
ADS Memory, Mb	53.97	115.90	75.83	86.44		35.37	72.61	53.18	59.77
Trimming Tests (TT)	3113435	3498231	3202690	3296160		1752581	1382510	1495675	1488853
Hit TT/Total TT	0.19	0.18	0.18	0.18		0.23	0.27	0.31	0.30
Newton's Method (ADS Leaves Only)									
Preproc. Time, Sec	245.23	272.17	328.77	330.07		265.69	325.10	388.90	389.54
Rendering Time, Sec	913.00	488.00	380.00	382.00		326.00	**146.00**	155.00	152.00
MA per HRay	107.39	70.10	74.12	73.25		44.22	23.51	28.46	27.35
BX per HRay	41.46	27.26	27.66	27.31		22.03	11.85	15.25	14.86
BH per HRay	22.90	16.20	17.24	16.95		10.69	6.31	7.20	6.97
BZ per HRay	29.29	21.81	23.75	23.23		14.61	8.62	10.30	9.87
Avg Its/Any BZ Call	1.39	1.33	1.42	1.39		1.48	1.37	1.50	1.46
Avg Its/Scs BZ Call	1.65	1.50	1.64	1.59		1.66	1.45	1.61	1.56
ADS Memory, Mb	46.01	92.01	62.25	70.13		30.15	57.21	42.78	47.68
Trimming Tests (TT)	3105048	3281970	3132853	3196134		1763820	1325053	1513675	1491786
Hit TT/Total TT	0.19	0.18	0.18	0.18		0.23	0.28	0.31	0.31



File Name:	**W203_Interieur_4.wrl**
Number of Patches:	**231926**
Number of Control Points:	**5119047**
Average Surface Degree:	**4.94**
Number of Trimming Contours:	**38318**
Screen Coverage:	**59%**

Table 5.20: Comparison of ray tracing trimmed rational Bézier surfaces methods (W203_Interieur_4.wrl model. *Model courtesy of DaimlerChrysler AG*).

5.9 Image Gallery

This section contains examples of images which are rendered using GOLEM [1] ray tracing system. The implemented NURBS evaluation library *myNURBS* has been integrated into the system in order to make it support direct ray tracing trimmed NURBS surfaces. Ray casting technique has been used in order to generate images. Four rays per pixel have been shot to prevent aliasing. All image scenes contain NURBS surfaces only.

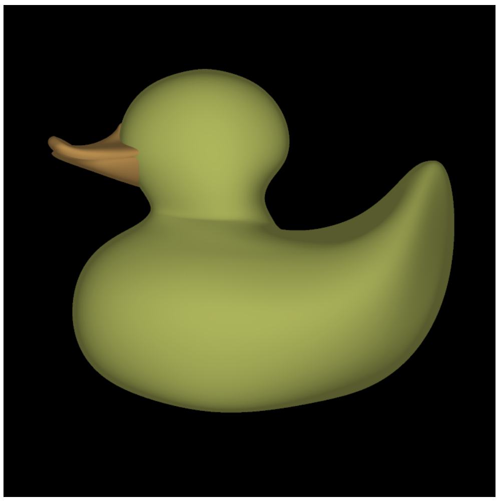


Figure 5.36: Duck.wrl model rendered with 800x800 image resolution. *Blaxxun interactive - Intel NURBS export.*

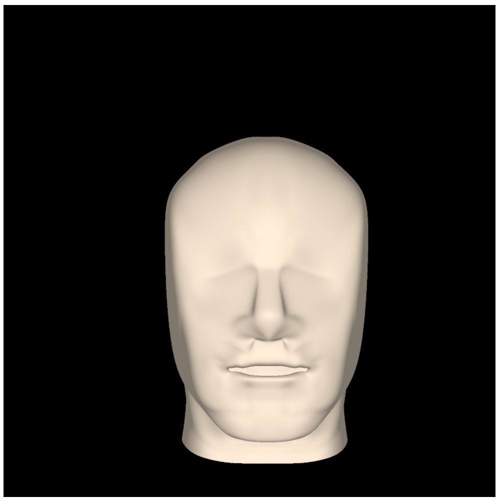


Figure 5.37: Head.wrl model rendered with 800x800 image resolution. *Model courtesy of Charles Adams.*

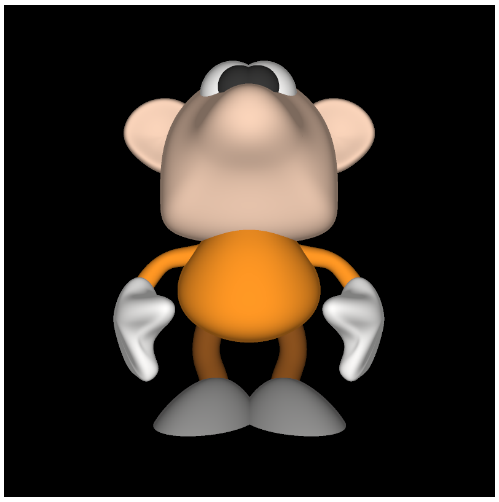


Figure 5.38: Gnom.wrl model rendered with 800x800 image resolution. *Blaxxun interactive - Intel NURBS export.*

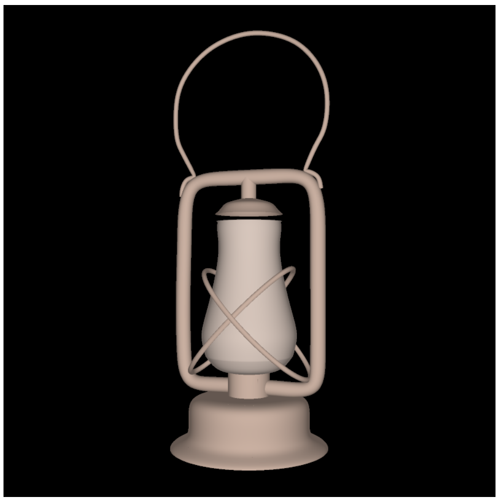


Figure 5.39: Lamp.wrl model rendered with 800x800 image resolution. *Model courtesy of Charles Adams.*

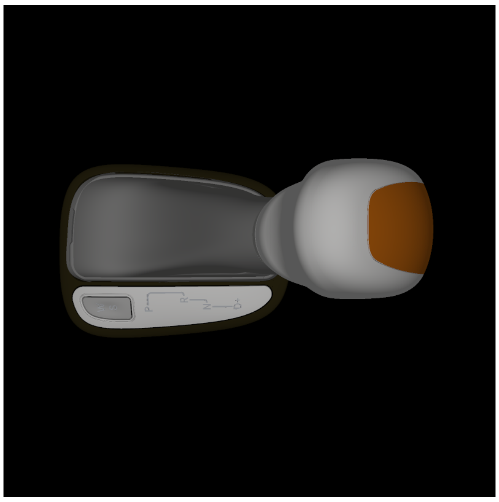


Figure 5.40: Gear_shift.wrl model rendered with 800x800 image resolution. *Model courtesy of DaimlerChrysler AG.*

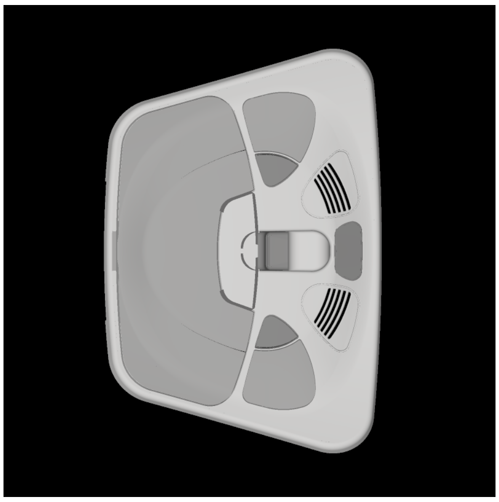


Figure 5.41: DBE.wrl model rendered with 800x800 image resolution. *Model courtesy of DaimlerChrysler AG.*

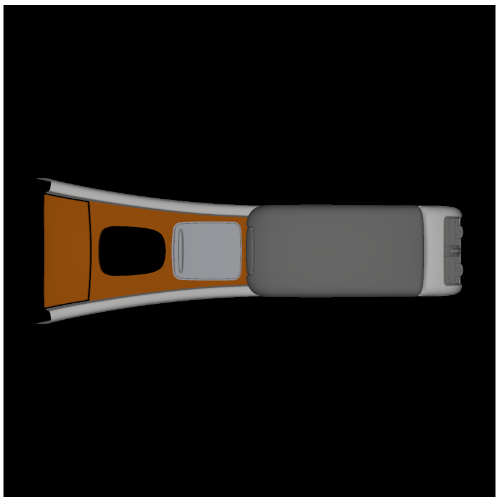


Figure 5.42: Middle_console.wrl model rendered with 800x800 image resolution. *Model courtesy of DaimlerChrysler AG.*

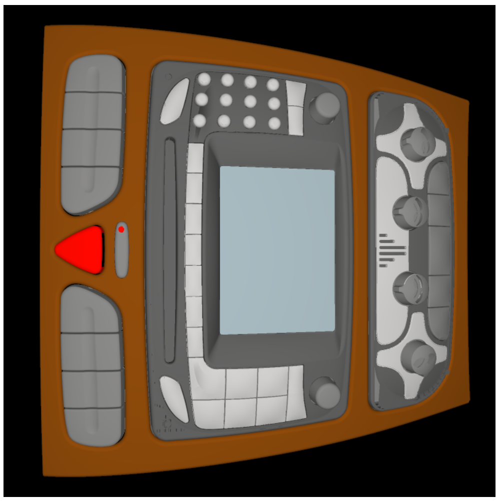


Figure 5.43: Comand.wrl model rendered with 800x800 image resolution. *Model courtesy of DaimlerChrysler AG.*

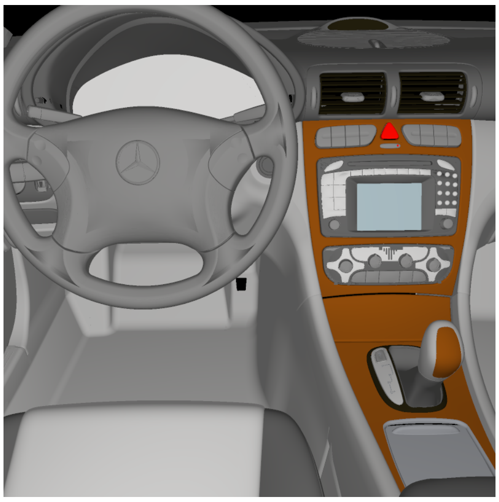


Figure 5.44: W203_Interieur_1.wrl model rendered with 800x800 image resolution. *Model courtesy of DaimlerChrysler AG.*

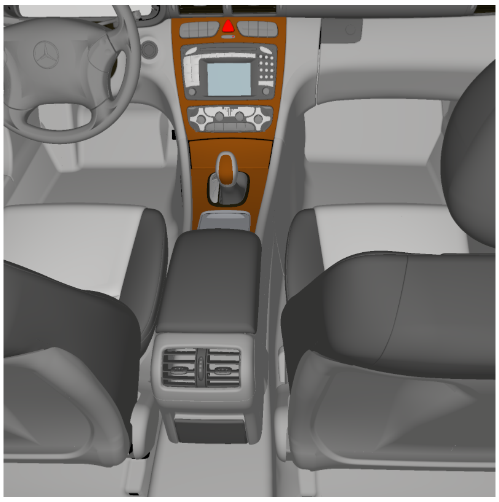


Figure 5.45: W203_Interieur_2.wrl model rendered with 800x800 image resolution. *Model courtesy of DaimlerChrysler AG.*

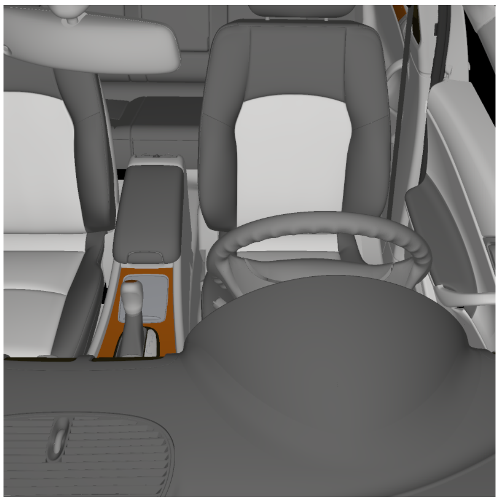


Figure 5.46: W203_Interieur_3.wrl model rendered with 800x800 image resolution. *Model courtesy of DaimlerChrysler AG.*

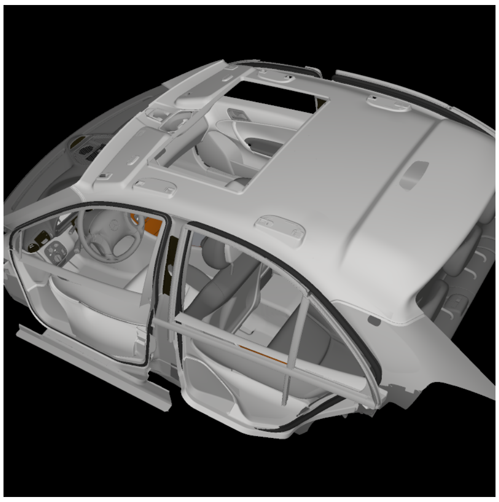


Figure 5.47: W203_Interieur_4.wrl model rendered with 800x800 image resolution. *Model courtesy of DaimlerChrysler AG.*

Chapter 6

Conclusion

The problem of direct ray tracing of trimmed NURBS surfaces has been thoroughly discussed throughout the Master Thesis. The first chapters of the Master Thesis introduce a reader to curves, surfaces, and basics of ray tracing techniques. The rest of the Master Thesis is dedicated to the problem of finding ray-trimmed NURBS surface intersection.

Different approaches for ray tracing trimmed NURBS surfaces have been described in the Master Thesis. Two most practical methods for ray tracing NURBS surfaces have been described in more details, namely Bézier clipping method and Newton's iteration method. It has been shown that it is preferable to transform NURBS surfaces into rational Bézier patches on the preprocessing stage of the ray tracing, because they have faster evaluation routines. The importance of utilizing acceleration data structure (ADS) for Bézier patches has been underlined and an efficient trimming test has been proposed.

Some reported problems as well as some suggestions for improvement of existing techniques have not been noticed by any researches so far. Below we summarize our novel achievements.

In Section 4.3.3 performance of Bézier clipping method has been improved by recomputing clipping lines L_u and L_v on each iteration.

Multiple equivalent intersections problem of Bézier clipping method has been reported in Section 4.3.5 with suggestions of handling of such intersections.

Termination criteria of Bézier clipping method has been modified by considering $3D$ object space rather than $2D$ parameter space. An efficient computation of epsilon value as termination criteria has been proposed in Section 4.3.6.

Termination criteria of Newton's iteration method has also been modified in Section 4.4.4 for handling virtual intersections correctly.

Efficient acceleration data structure (ADS) for Bézier patches has been proposed in Section 5.3 in order to accelerate the ray tracing process. Efficient choice of clipping plane for building ADS trees and ideas of adaptive curvature based ADS construction have been explained in details in Section 5.3.3. In Sections 5.5.5 and 5.6 it has been shown how leaves of ADS which enclose fully trimmed or fully untrimmed surface patches can be handled efficiently. In Section 5.3.5 it has also been shown that approach with ADS leaves only can have better performance when utilizing acceleration spatial data structure for ray tracing (e.g., kd-trees).

The efficient scheme of combining the ADS with two most practical methods for ray tracing Bézier patches (Bézier clipping method and Newton's iteration method) has been proposed in Section 5.4. In Section 5.4.2 it has been shown how the initial clipping can improve the performance of Bézier clipping method. In Section 5.4.3 it has been shown how the initial guess can be computed carefully for Newton's iteration method in order to decrease probability of convergence to wrong intersections.

Some experienced numerical problems have been described in Section 5.7 together with suggestions how they can be eliminated or alleviated.

Most practical techniques for ray tracing trimmed NURBS surfaces have been implemented within a library for NURBS evaluation (*myNURBS*). The library can be integrated into ray tracing applications in order to make them support direct ray tracing of trimmed NURBS surfaces.

For testing purposes the implemented library has been integrated into GOLEM [1] ray tracing system. By this way the presented techniques have been tested with different settings of parameters. Section 5.8 shows the comparison of presented methods and achieved results.

A possible future work can be to extend the library with adaptive trimmed NURBS triangulation routines and compare two different approaches for ray tracing trimmed NURBS surfaces: direct one and one via triangulation.

Appendix A

Library Compatibility with VRML97 and X3D

The implemented for testing purposes library for NURBS evaluation (*myNURBS*) is compatible with some NURBS nodes of VRML97 [2] and X3D [3] specifications, i.e., behavior of such nodes can be simulated using the library classes and methods. Table A.1 shows the library compatibility with VRML97 [2] specification. Table A.2 shows the library compatibility with X3D [3] specification.

VRML97 NURBS Node	**Compatibility**
Contour2D	yes
CoordinateDeformer	yes
NurbsCurve	yes
NurbsCurve2D	yes
NurbsGroup	no
NurbsPositionInterpolator	yes
NurbsSurface	yes
NurbsTextureSurface	yes
Polyline2D	yes
TrimmedSurface	yes

Table A.1: The library compatibility with VRML97 [2] specification.

X3D NURBS Node	**Compatibility**
Contour2D	yes
ContourPolyline2D	yes
CoordinateDouble	no
NurbsCurve	yes
NurbsCurve2D	yes
NurbsOrientationInterpolator	yes
NurbsPatchSurface	yes
NurbsPositionInterpolator	yes
NurbsSet	no
NurbsSurfaceInterpolator	yes
NurbsSweptSurface	no
NurbsSwungSurface	no
NurbsTextureCoordinate	yes
NurbsTrimmedSurface	yes

Table A.2: The library compatibility with X3D [3] specification.

Appendix B

Integration the Library into a Ray Tracing System

The library provides a set of classes and methods for maintaining trimmed NURBS surfaces in a ray tracing application. Evaluation procedures of NURBS surfaces are much slower than evaluation procedures of rational Bézier patches. Therefore, NURBS surfaces are supposed to be transformed into rational Bézier patches during the preprocessing step of ray tracing, i.e., the rational Bézier patches must be used as basic objects in a ray tracing application.

We can distinguish two stages of operations with trimmed NURBS surfaces in a ray tracing application:

- operations during the preprocessing step;
- rendering operations.

During the preprocessing stage a hierarchy of trimming contours must be created, a texture mapping must be specified, NURBS surfaces must be transformed into rational Bézier patches. The trimming hierarchy, the texture mapping, and the Bézier patches specify interface objects for a ray tracing application, which are used during the rendering process. Notice that the trimming hierarchy and the texture mapping are optional and may be set unspecified. A framework of the preprocessing stage is given below.

Algorithm B.1 (The framework of the preprocessing stage)

```
//---------------------------------------------------------------------//
//FRAMEWORK FOR CREATING THE TRIMMING HIERARCHY FOR A NURBS SURFACE
//---------------------------------------------------------------------//

//prepare a trimming hierarchy
myTrimmingContour *trim = new myTrimmingContour();

//loop over all contours to be processed
for (;;)
{
  //prepare a contour for future processing
  myContour2D* contour = new myContour2D();

  //create a nurbs curve
  myNurbsCurve nurbs_curve = myNurbsCurve(degree, knots, points, weights);

  //represent the nurbs curve by the number of bezier splines
  vector<myBezierCurve> bezier_splines;
  nurbs_curve.nurbsGetBezierSplines(bezier_splines);

  //loop over obtained bezier splines
  for (int i = 0; i < (int)bezier_splines.size(); i++)
  {
    //create a pointer to the allocated in memory bezier spline
    myBezierCurve *spline = new myBezierCurve(bezier_splines[i]);
    //add the bezier spline to the contour
    contour->contourAdd(spline);
  }

  //create a pointer to an allocated in memory polyline
  myPolyline2D *polyline = new myPolyline2D(vertices);
  //add the polyline to the contour
  contour->contourAdd(polyline);

  //--------------------------------------------------------------------//
  //process all other nurbs curves and polylines which compose the
  //contour. notice, that the contour must be closed, i.e. the first
  //point of the first contour segment must be the equal to the last
  //point of the last contour segment.
  //--------------------------------------------------------------------//

  //add the contour to the trimming hierarchy
  trim->trimmingAddContour(contour);
}


//--------------------------------------------------------------------//
//FRAMEWORK FOR CREATING THE NURBS TEXTURE MAPPING FOR A NURBS SURFACE
//--------------------------------------------------------------------//

//create the nurbs texture mapping
myNurbsTextureSurface texture = myNurbsTextureSurface(degree_u, degree_v,
  dimension_u, dimension_v, knots_u, knots_v, points);

//represent the nurbs texture by the number of bezier textures
myTBezierTexturePatches textures;
texture.nurbsGetBezierPatches(textures);


//--------------------------------------------------------------------//
//FRAMEWORK FOR PROCESSING A NURBS SURFACE
//--------------------------------------------------------------------//

//set up the building parameters for all processing nurbs surfaces.
//the routine has a number of parameters, which are described in
//details in myBezierRT.h header file.
myBezierSurfaceRT::SetBuildingParameters(...);

//set up the epsilon value for 2D parameter space and 3D object space
myNURBS::Epsilon2D = eps2D;
myNURBS::Epsilon3D = eps3D;
```

```
//create a nurbs surface
myNurbsSurface surface = myNurbsSurface(degree_u, degree_v,
  dimension_u, dimension_v, knot_u, knot_v, points);

//apply the current matrix transformation (if necessary)
surface.nurbsApplyTransformation(matrix);

//represent the nurbs surface by the number of bezier patches
myTBezierSurfacePatches patches;
surface.nurbsGetBezierPatches(patches);

//loop over obtained bezier patches
for (int i = 0; i < (int)patches.size(); i++)
{
  for (int j = 0; j < (int)patches[i].size(); j++)
  {
    //create an interface object for the bezier patch
    myBezierSurfaceRT *object = new myBezierSurfaceRT(patches[i][j],
      trim, new myBezierTextureSurface(textures[i][j]));

    //if the bezier patch is fully trimmed then skeep the future
    //processing, i.e. delete and skeep the current patch.
    if (myBezierSurfaceRT::IsTrimmed())
    {
      delete object;
      continue;
    }

    //use the interface object in order to create an object of
    //a ray tracing system which utilizes the library
    CBEZIER *bezier = new CBEZIER(object);
  }
}
```

The rendering operations are computing the nearest intersection, normal and partial derivatives evaluation at the intersection point, obtaining the texture coordinates for the intersection point, and other. A framework of the simplest rendering scheme is given below. More information about supported routines can be found in well documented header files of the library.

Algorithm B.2 (The framework of the rendering stage)

```
//----------------------------------------------------------------------//
//FRAMEWORK FOR FINDING THE NEAREST INTERSECTION
//----------------------------------------------------------------------//

//prepare a ray
myRay ray = myRay(origin, direction);

//set up the epsilon value for finding the nearest ray-object
//intersection. instead of myNURBS::Epsilon3D one can use the
//view dependently calculated epsilon value or other values.
object->SetEpsilon(myNURBS::Epsilon3D);

//set up the maximum allowed distance to the nearest intersection
ray._t = Infinity;

//prepare two auxiliary variables
float min = 0.0, max = 0.0;
```

```
bool result = false;

//if the ray misses the object's bounding box
if (!object->_voxel->_bbox.bboxIntersect(ray, min, max))
  return;

//find the nearest intersection using Bezier Clipping Method
result = object->bezierFindNearestIntersectionA(point, ray);

//find the nearest intersection using Newton's Method
result = object->bezierFindNearestIntersectionB(point, ray);

//--------------------------------------------------------------------//
//FRAMEWORK FOR OBTAINING THE NORMAL AND PARTIAL DERIVATIVES
//AT THE INTERSECTION POINT
//--------------------------------------------------------------------//

//normalize parameter coordinates of intersection to the
//parameter domain [0..1]x[0..1] in the context of bezier patch
float nu = point._u, nv = point._v;
object->BezierToNormalizedBezier(nu, nv);

//obtain the normal at the intersection point
normal = object->bezierCalculateNormal(nu, nv);

//obtain the partial derivatives at the intersection point
float du = 0.0, dv = 0.0;
object->bezierCalculateDuDv(nu, nv, du, dv);

//--------------------------------------------------------------------//
//FRAMEWORK FOR OBTAINING THE TEXTURE COORDINATES
//AT THE INTERSECTION POINT
//--------------------------------------------------------------------//

//normalize parameter coordinates of intersection to the
//parameter domain [0..1]x[0..1] in the context of nurbs surface
float tu = point._u, tv = point._v;
object->BezierToNormalizedBezier(tu, tv);

//obtain texture coordinates at the intersection point
object->GetTextureCoordinates(tu, tv);
```

Bibliography

[1] GOLEM Ray Tracing System. http://www.cgg.cvut.cz/GOLEM/.

[2] VRML97, The Virtual Reality Modeling Language. http://www.web3d.org/x3d/specifications/vrml/ISO_IEC_14772-All/.

[3] X3D, Extensible 3D. http://www.web3d.org/x3d/specifications/ISO-IEC-19775-FDIS-X3dAbstractSpecification/.

[4] J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics Conference*, 1987.

[5] W. Barth, R. Lieger, and M. Schindler. Ray tracing general parametric surfaces using interval arithmetic. *Visual Computer*, 10(7):363–371, 1994.

[6] W. Barth and W. Sturzlinger. Efficient Ray Tracing for Bezier and B-Spline Surfaces. *Computers & Graphics*, 17(4):423–430, 1993.

[7] L. Biard. Parametric Surfaces and Ray Tracing. In *Proceedings Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 31–51, 1990.

[8] J. Bloomenthal. *Introduction to Implicit Surfaces*. Morgan Kaufmann, Hardcover edition, 1997.

[9] S. Campanga and P. Slusallek. Improving Bézier Clipping and Chebyshev Boxing for Ray Tracing Parametric Surfaces. Technical report, University of Erlangen, Computer Graphics Group, 1996.

[10] A.Y. Chang. A Survey of Geometric Data Structures for Ray Tracing. Technical report, Department of Computer and Information Science, 2001.

[11] P. Dutré. Global Illumination Compendium. Technical report, Computer Graphics, Department of Computer Science, Katholieke Universiteit Leuven, 2003.

[12] P. Dutré, P. Bekaert, and K. Bala. *Advanced Global Illumination.* AK Peters Ltd, 2003.

[13] W. Enger. Interval Ray Tracing – a divide and conquer strategy for realistic computer graphics. *The Visual Computer*, 8(9):91–104, 1992. ISSN 0178-2789.

[14] F.J. Espinio, M. Bóo, and M. Amor. Adaptive Tessellation of NURBS Surfaces. *Journal of WSCG*, 2003.

[15] A. Fournier and J. Buchanan. Chebyshev Polynomials for Boxing and Intersections of Parametric Curves and Surfaces. *Computer Graphics Forum*, 13(3):C/127–C/142, 1994.

[16] A.S. Glasner. *An Introduction to Ray Tracing.* Morgan Kaufmann, 1989.

[17] V. Havran. *Heuristic Ray Shooting Algorithms.* PhD thesis, Czech Technical University, Faculty of Electrical Engineering, Department of Computer Science and Engineering, 2000.

[18] V. Havran and L. Dachs. VIS-RT: A Visualization System for RT Spatial Data Structures. *WSCG*, 2001.

[19] J. Hoschek and D. Lasser. *Fundamentals of Computer Aided Geometric Design.* AK Peters Ltd, Hardcover edition, 1993.

[20] K.I. Joy. Ray Tracing Parametric Patches Utilizing Numerical Techniques and Ray Coherence. *Computer Graphics*, 20(4), 1986.

[21] J.T. Kajiya. Ray Tracing Parametric Patches. *Computer Graphics*, 16(3), 1982.

[22] J.T. Kajiya. New Techniques For Ray Tracing Procedurally Defined Objects. *Computer Graphics*, 17(3):91–102, 1983.

[23] S. Kumar and D. Manocha. Efficient rendering of trimmed NURBS surfaces. *Computer-Aided Design*, 1995.

[24] S.K. László. Monte-Carlo Mathods in Global Illumination. Technical report, Institute of Computer Graphics, Vienna University of Technology, 1999.

[25] G. Levner, P. Tassinari, and D. Marini. A simple, general method for ray tracing bicubic surfaces. *Computer Graphics*, pages 285–302, 1987.

[26] D. Lischinski and J. Gonczarowski. Improved Techniques for Ray Tracing Parametric Surfaces. Technical report, Department of Computer Science, The Hebrew Uiversity of Jerusalem, 1990.

[27] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical Ray Tracing of Trimmed NURBS Surfaces. *Journal of Graphics Tools: JGT*, 5(1):27–52, 2000.

[28] M.E. Mortenson. *Geometric Modeling.* Willey, 2nd edition, 1997.

[29] T. Nishita and T.W. Sederberg. Ray Tracing Trimmed Rational Surface Patches. *Computer Graphics*, 24(4), 1990.

[30] L. Piegl. A geometric investigation of the rational bezier scheme of computer aided design. In *Computers in Industry*, volume 7, pages 401–410, 1986.

[31] L. Piegl and W. Tiller. *The NURBS Book.* Springer, 2nd edition, 1997.

[32] H. Prautzsch and T. Gallagher. Is there a geometric variation diminishing property for B-spline or Bezier surfaces? In *Computer Aided Geometric Design*, volume 9, pages 119 – 124, 1992.

[33] K. Qin and M. Gong. Fast Ray Tracing NURBS Surfaces. Technical report, Department of Computer Science, Tsinghua University, 1996.

[34] D.F. Rogers. *An Introduction to NURBS: with historical perspective.* Morgan Kaufmann, 1st edition, 2000.

[35] J. Ronke. The area of a simple polygon. In James R. Avro, editor, *Graphics Gems II.* Academic Press, 1991.

[36] S.M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, 1980.

[37] H. Sanchez. Evaluation of NURBS surfaces: an overview based on runtime efficiency. Technical report, University of Extremadura, Department of Computer Science, 2004.

[38] P. Shirley. *Realistic Ray Tracing.* AK Peters Ltd, 2000.

[39] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Professional, 3rd edition, 1997.

[40] K. Sung and P. Shirley. Ray tracing with the bsp tree. In David Kirk, editor, *Graphics Gems III*, volume 6, pages 271–274. Morgan Kaufmann, 1994.

[41] M. Sweeney and R.H. Bartels. Ray tracing free-form B-spline surfaces. *IEEE Computer Graphics & Applications*, 6(2), 1986.

[42] T. Möller and B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of graphics tools*, 1997.

[43] D.L. Toth. On Ray Tracing Parametric Surfaces. *Computer Graphics*, 19(3), 1985.

[44] S.-W. Wang and Z.-C. Shih. An Efficient and Stable Ray Tracing Algorithm for Parametric Surfaces. *Journal of Information Science and Engineering*, (18), 2001.

[45] S.-W. Wang, Z.-C. Shih, and R.-C. Chang. An improved rendering technique for ray tracing Bézier and B-spline surfaces. *The Journal of Visualization and Computer Animation*, 11(4):209–219, 2000.

[46] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[47] A. Williams and P. Shirley. An Efficient and Robust Ray-Box Intersection Algorithm. *Journal of graphics tools*, 2003.

[48] C. Woodward. Ray tracing parametric surfaces by subdivision in viewing plane. *Theory and practice of geometric modeling*, pages 273–287, 1989.

[49] C.-G. Yang. On speeding up ray tracing of B-spline surfaces. *Computer Aided Design*, 19(3), 1987.

[50] J. Zheng and T.W. Sederberg. Gaussian and mean curvatures of rational Bézier patches. *Computer Aided Geometric Design*, 2003.