Ray Tracing Rational B-Spline Patches in VLSI

Bengt-Olaf Schneider

University of Tübingen
Wilhelm-Schickard-Institut für Informatik
Graphisch-Interaktive Systeme
Auf der Morgenstelle 10, C9
7400 Tübingen, FRG

Rational B-spline surfaces make it possible to merge the concepts of freeform surfaces and that of surfaces described by rational polynomials especially conic sections. For ray tracing it is crucial to determine the intersection between ray and object. Therefore an algorithm is developed that is suitable for a VLSI implementation. Some alternatives for the implementation of this algorithm are presented and discussed. The paper concludes with a discussion of some problems and possible further developments.

1. Introduction

During the last years ray tracing has proved to be an excellent way for generating high quality images [8, 9, 11, 14]. Ray tracing facilitates the modeling of effects like reflection, refraction and cast shadows. The drawback of this method is the huge amount of computations necessary. Although many efforts have been made to speed up ray tracing it remains relatively slow.

This becomes even more true, if not only the objects of the CSG world are allowed to be in the scene [5]. Usually freeform surfaces are a better way to get an appropriate description of real objects. Today these are mostly Bézier or B-spline surfaces. These methods approximate or interpolate given control points to achieve a closed surface with certain continuity at all points of the surface. It has turned out, however, that certain, oftenly used solids cannot be described exactly with the means of B-spline surfaces. (Bézier surfaces are a special case of B-spline surfaces.) Especially conic sections can only be approximated by B-splines. Surfaces that can be described by rational polynomials do not show this restriction [2, 4, 6, 7, 12, 13]. Therefore it would be desirable to be able to process such surfaces with ray tracing.

Since ray tracing freeform surfaces costs much computing time it seems to be reasonable to provide hardware support for at least some crucial parts of the ray tracing process. Because a ray tracer spends most of the time seeking the intersection of a ray with an object it seems to be most promising to implement this step in hardware.

2. Rational B-splines

This section will give briefly the definition of a rational B-spline surface. Then some of its properties will be stated and discussed.

A rational B-spline surface is obtained, if instead of 3D coordinates (P^x,P^y,P^z) for a point P its 4D homogeneous coordinates (p^x,p^y,p^z,p^w) are used. The transformations from 3D coordinates into homogeneous coordinates and vice versa are well known :

$$p = (p^x,p^y,p^z,p^w) = p^w(P^x,P^y,P^z,1) = p^w(P,1)$$

where

$$P^x = \frac{p^x}{p^w}; \quad P^y = \frac{p^y}{p^w}; \quad P^z = \frac{p^z}{p^w}$$

A rational B-Spline surface $q(u,v)$ (in homogeneous coordinates) is given by the equation (in vector notation !) :

$$q(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} p_{i,j}\cdot N_{i,k}(u)\cdot N_{j,l}(v) \tag{1}$$

with $p_{i,j}$ being the control points in homogeneous space and $N_{i,k}$ $(N_{j,l})$ the basic B-splines of degree k (l) as defined in [1].

The 3D-coordinates of this surface are obtained by division by the w-component of q:

$$\begin{aligned} Q(u,v) &= \frac{q(u,v)}{q^w(u,v)} \\ &= \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} p_{i,j}\cdot N_{i,k}(u)\cdot N_{j,l}(v)}{\sum_{i=0}^{m}\sum_{j=0}^{n} p^w_{i,j}\cdot N_{i,k}(u)\cdot N_{j,l}(v)} \\ &= \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} p^w_{i,j}\cdot (P_{i,j},1)\cdot N_{i,k}(u)\cdot N_{j,l}(v)}{\sum_{i=0}^{m}\sum_{j=0}^{n} p^w_{i,j}\cdot N_{i,k}(u)\cdot N_{j,l}(v)} \end{aligned}$$

As can be seen from equation (1) rational B-spline surfaces are defined as 4D B-spline surfaces in homogeneous coordinates. Therefore all properties of 3D B-splines that are defined in terms of single coordinates hold for the rational B-spline scheme too. Some important properties of rational B-spline surfaces are listed below [7, 12, 13]:

(P1) The locality of B-splines is preserved for rational B-splines, i.e. a point on the surface is influenced by no more than $k \cdot l$ control points.

(P2) Rational B-spline surfaces are piecewise defined rational surfaces. They are therefore suited for the representation of all rational surfaces.

(P3) The w-component $p^w_{i,j}$ of a control point $P_{i,j}$ is a direct weight for $P_{i,j}$.

(P4) All points on the surface $q(u,v)$ lie within the convex hull of the control points $p_{i,j}$.

(P5) All methods that operate separately on each coordinate x,y,z of a 3D B-spline surface can be easily extended to rational B-spline surfaces (= 4D B-spline surface).

3. Ray tracing of rational B-spline surfaces

Image generation with ray tracing can be decomposed into two main steps. The first step tests all objects in the scene (or a subset of these objects) against the ray. The second steps performs a choice among all objects hit by the ray. That object which is intersected first by the ray is chosen. The value of that pixel the ray is sent through is determined by the properties of the chosen object (Figure 1). These properties include color, transparency, translucence, reflectance, glossiness and texture.

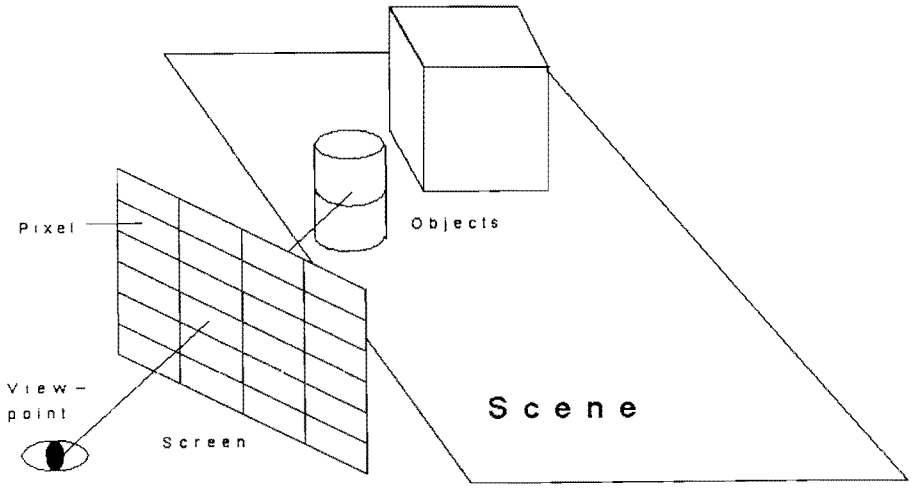

Figure 1: Principle of ray tracing

It has been found that the second step is less important with regard to the computation time [11]. This means that a ray tracer spends most of the time calculating the intersection of the ray with objects. For the class of objects dealt with in this paper, namely freeform surfaces, especially rational B-spline surfaces, there are

two basic approaches to determine the intersection. The first one calculates the exact intersection by solving a system of equations. This method is restricted to a small class of surfaces. Therefore it is beyond the scope of this paper and won't be discussed any further [5].

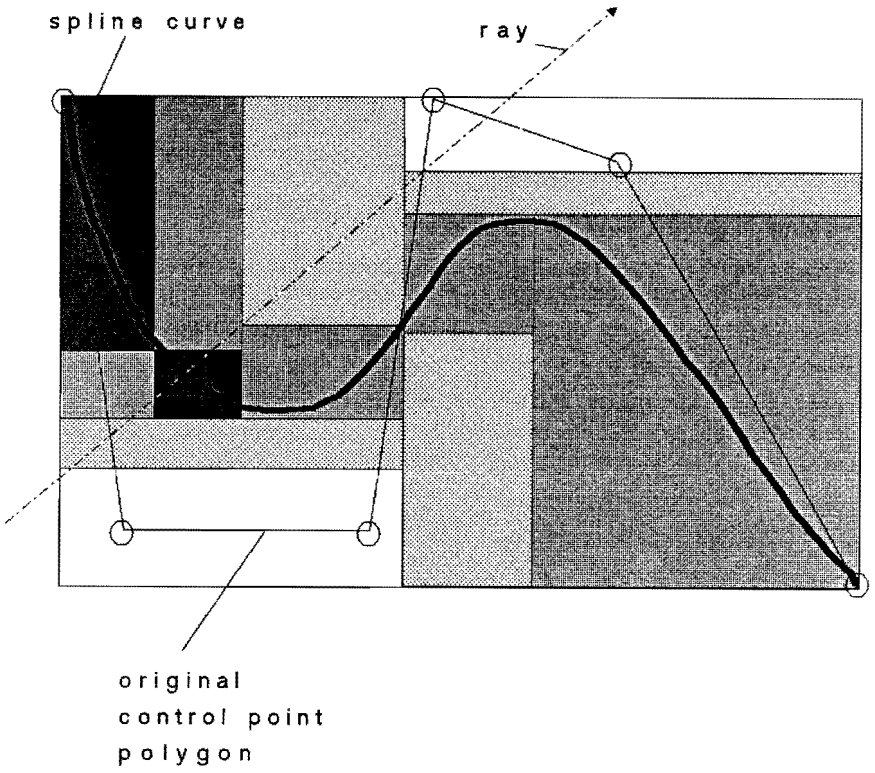

Figure 2: Determining the intersection by recursive subdivision

The second approach tries to get an approximation of the intersection by recursive subdivision of the surface in subpatches [8, 14]. Only those subpatches are treated further that may be intersected by the ray. A possible criterion for further processing a subpatch could be the bounding box of the defining control points. Since the bounding box is a superset of the convex hull of the control point net (see property P4 of the rational B-spline surface), it is guaranteed that every ray that intersects the surface intersects the bounding box too. Figure 2 illustrates this fact with a curve. (To maintain clearness the control points of the subcurves are not drawn.)

```
function get_intersection1 (patch,no_subdivisions) : boolean ;
begin
   ok := false ;
   if no_subdivisions > 0
      then
         begin
            no_subdivisions := no_subdivisions - 1 ;

            subpatch[1..4] := subpatches_of (patch) ;
            for i:=1 to 4 do
               intersection[i] := intersect_with (bounding_box(subpatch[i])) ;

               sort_subpatches ;  { i-th subpatch is the i-th hit by the ray }

               for i:=1 to 4 do
                  begin
                     if intersection[i] then
                     ok := get_intersection1 (subpatch[i],no_subdivisions) ;
                     if ok then goto RETURN ;
                  end ;
         end
      else
         ok := true ;

   RETURN :
   get_intersection1 := ok ;
end ;
```

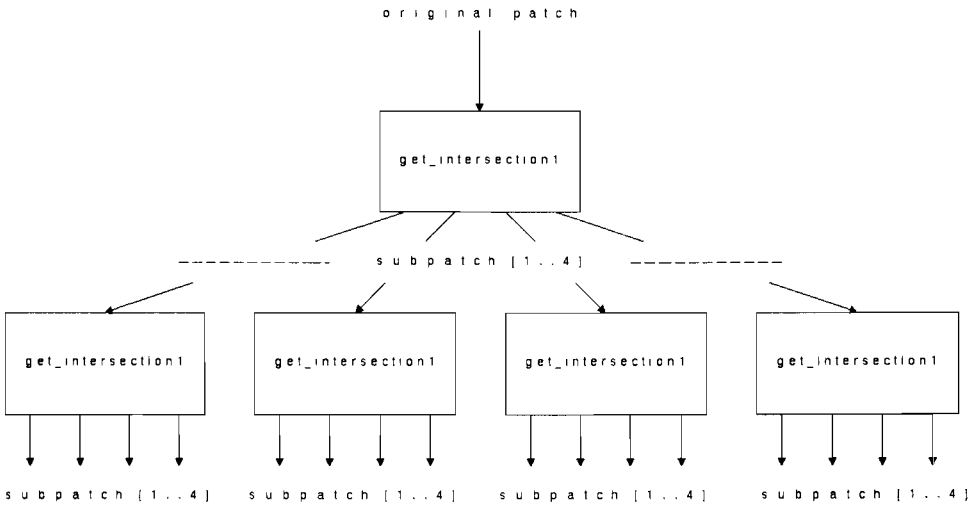

Figure 3: Recursive version of the subdivision algorithm

It can be seen from Figure 2 that it may happen that the ray hits more than one bounding box. If this happens all subpatches concerned have to be subdivided further. If even in the end of the subdivision process there are multiple intersections only the first (nearest) intersection is regarded. This restriction makes sense for the generation of high quality pictures. If only the first intersection is of interest it would be efficient to process the subpatches in the order they are intersected by the ray. This requires sorting the patches.

It is very easy and straight forward to implement the intersection processor in a recursive manner. Such an algorithm can be easily mapped onto an architecture employing a multiprocessor tree. Figure 3 shows the algorithm and the corresponding architecture.

Since a multiprocessor tree approach would require much communication between the nodes this does not seem to be the best choice. Another possibility for the intersection algorithm is an iterative approach. Such an approach gives rise to another problem: How to deal with temporary subpatches that cannot be processed immediately ? They have to be stored on a stack. But pushing a subpatch onto a stack means to push the defining control points. This is a considerable amount of data. A way out of this problem may be the following: Instead of storing the control points of the subpatch, only the way one had obtained the subpatch is stored. This means only the path from the original surface is stored. This data could be stored in a few bytes. Unfortunately this path has to be followed again when the patch is popped from the stack. This is a time consuming step. A combination of both methods is most promising. Instead of storing only either control points or pathes we use both methods. In general only the path that generated a subpatch is stored. But after a certain number of subdivisions the control points are saved. Thus all following subdivisions can be defined relative to these control points. As far as I know this approach has been published first by Pulleyblank and Kapenga [8]. They called it subdivision in stages. Figure 4 shows the algorithm.

The two intersection algorithms discussed are only extremes of a spectrum of solutions that varies from massively parallel to serial alternatives. The iterative, serial solution is a quite simple solution. The recursive and parallel alternative is a bad tradeoff between speed and used resources because rarely all nodes of the tree contribute to the result. This means in most cases many nodes (processing elements) are idle. Therefore further research has to be done to find an architecture that eliminates these drawbacks. I will restrict myself to the iterative approach (subdivision in stages) throughout the rest of the paper.

```
function get_intersection2 (patch,no_subdivisions,path_length) : boolean ;
begin
   if no_subdivisions > 0 then
      begin
         pl := path_length ;
         push_cp (patch) ; { --> control point stack }

         repeat
            pl := pl-1 ;
            if pl=0 then
               begin
                  pl := path_length ;
                  push_cp (patch) ;
               end ;

            subpatch[1..4] := subpatches_of (patch) ;
            for i:=1 to 4 do
               intersection[i] := intersect_with (bounding_box(subpatch[i])) ;

            sort_subpatches ;

            for i:=1 to 4 do
               if intersection[i] then
                  push_path (subpatch[i],no_subdivisions,pl) ;
                                                    { --> path stack }
            ready := true ;
            if not empty(path stack) then
               begin
                  pop_path (patch,no_subdivisions,pl) ;
                  ready := (no_subdivisions=0) ;
                  if ready then get_intersection2 := true ;
               end ;

         until ready ;
      end ;
end ;
```

Figure 4: Algorithm for subdivision-in-stages

4. Subdivision algorithm

Because of the property P5 of the rational B-spline surface, subdivision algorithms developed for normal B-spline surfaces can be used also for rational B-spline surfaces. A well known method for the subdivision of B-spline curves is the OSLO-algorithm of Cohen et. al. [3]. This algorithm starts from an old knot vector, old control points, and a new knot vector that is a superset of the old one. It generates the corresponding new control points. The algorithm is listed in Figure 5.

Given :

$P_i, i = 0 \cdots n$: old control points

$\tau_i, i = 0 \cdots n+k$: old knot vektor

$t_i, i = 0 \cdots q$: new knot vector $(q \geqslant n+k)$

k : order of the B-splines)

It generates the new control points $d_j, j = 0 \cdots q-k$, as follows :

$$d_j = P_{\mu,j}^{[k]}$$

$$P_{i,j}^{[1]} = P_i$$

$$P_{i,j}^{[r+1]} = \frac{(t_{j+k-r} - \tau_i)P_{i,j}^{[r]} + (\tau_{i+k-r} - t_{j+k-r})P_{i-1,j}^{[r]}}{\tau_{i+k-r} - \tau_i}$$

$$\tau_\mu \leqslant t_j < \tau_{\mu+1}$$

Figure 5: The OSLO-algorithm

The OSLO-algorithm is a very powerful method. But it shows some problems with regard to an implementation in VLSI. First, it needs $O(k^2)$ storage elements. Second, the division occuring in the computation of $P_{i,j}^{[r+1]}$ is very unpleasant. Both drawbacks can be overcome by a suitable modification. Since each iteration $P_{i,j}^{[r+1]}$ needs only the previous iteration step r, it is possible to compute and store all $P_{i,j}$ in one field of length k. According to [10] the division step can be eliminated by additional multiplications. The modified OSLO-algorithm is shown in Figure 6. This form does not contain divisions in the inner loops. There is only one final division step $(\star)$. If several subdivision steps have to be carried out in succession, the final division can be postponed until after the last of these subdivisions. This is possible if the f_μ are not initialized to one but to the value of the f_μ from the previous subdivision (as indicated in the figure). Although the additional multiplications seem to slow down the subdivision process, this form is favourable for a VLSI implementation. Since the silicon area for the divider is saved, more area can be consumed to implement more and/or faster multipliers. Hence the modified OSLO-algorithm may be faster than the original one. Another aspect of the modified OSLO-algorithm is that it can be easily parallelized. The R_i and f_i can be computed in parallel. Therefore further speed up is inherent to this algorithm.

for $i := \mu-k+1$ to μ do
  begin
    $R_i := P_i$;
    $f_i := \begin{cases} 1 & \text{if first or single subdivision} \\ f_\mu & \text{from preceding subdivision} \end{cases}$
  end ;

for $r := 1$ to $k-1$ do
  for $i := \mu$ downto $\mu-k+1+r$ do
    begin
      $T1 := (t_{j+k-r}-\tau_i)\cdot f_{i-1}$;
      $T2 := (\tau_{i+k-r}-t_{j+k-r})\cdot f_i$;
      $R_i := T1\cdot R_i + T2\cdot R_{i-1}$;
      $f_i := (\tau_{i+k-r}-\tau_i)\cdot f_{i-1}\cdot f_i$;
    end ;

$d_j := R_\mu / f_\mu$; (*)

Figure 6: The modified OSLO-algorithm. ((*) : see text)

5. A proposal for an architecture

Figure 7 shows the architecture of an intersection processor for ray tracing rational B-spline patches. The registers that can be accessed by the host, store parameters and control points of the patch. Two stacks serve as temporary memory for subpatches that have to be processed further. They contain control points or the path how a patch has been constructed. They are necessary for the subdivision in stages. The heart of the system is the actual intersection unit. This block is composed of a subdivision block, blocks that compute the bounding box of a subpatch, and blocks that determine the intersection between a ray and a box.

The central block of the intersection unit is the subdivision unit. It is composed of three bisecting units (Figure 8).

The bisecting units itself are implementations of the modified OSLO-algorithm. Its implementation gives rise to some problems that stem from the many multiplications. Depending on the restrictions with respect to speed and chip area there are different optimums in degree of parallelism. Two levels where a parallelization may take place have to be distinguished. The first level is the modified OSLO-algorithm as a whole. Here a tradeoff between speed and chip area means to decide how many R_i and f_i are computed simultaneously. Figure 9 shows a completely parallel solution.

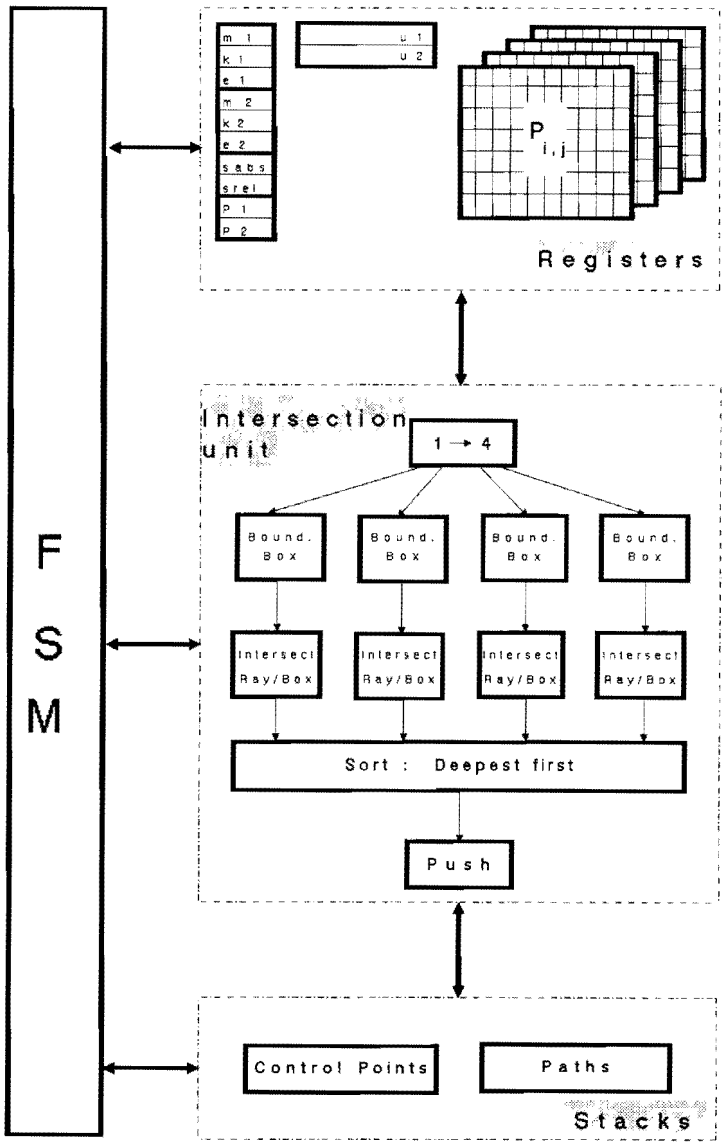

Figure 7: Overview of the intersection processor

Since all R_i and f_i of one iteration are calculated at once in the k computational units C_i, k steps are necessary to compute a new control point. If it is too expensive to provide k computational units it is possible to use fewer C_i. The extreme case is only one computational unit (Figure 10). On average there are $(k+1)/2$ steps necessary to compute the R_i and f_i. Therefore $k(k+1)/2$ steps are required to determine a new control point.

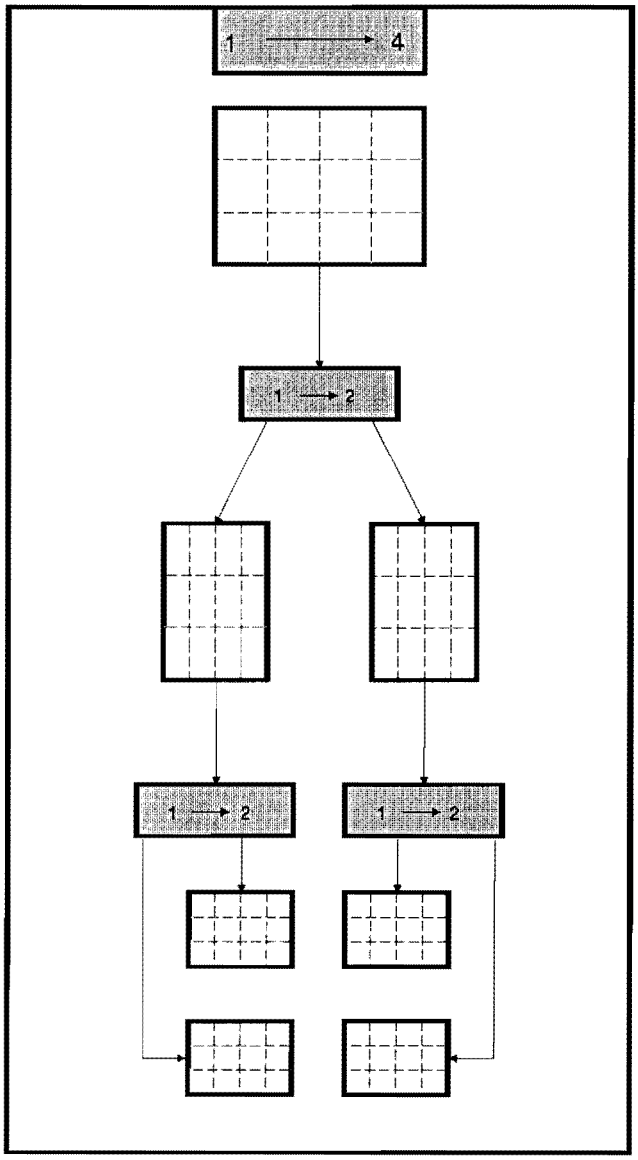

Figure 8: Subdivision unit

Until now it was assumed that the R_i and f_i can be determined in one step. This is true if an architecture like that one in Figure 11 is used. This demands the realization of 6 multipliers and 2 adders/subtractors. The time spent for the calculation of the R_i and f_i equals the propagation time of two multipliers and one adder.

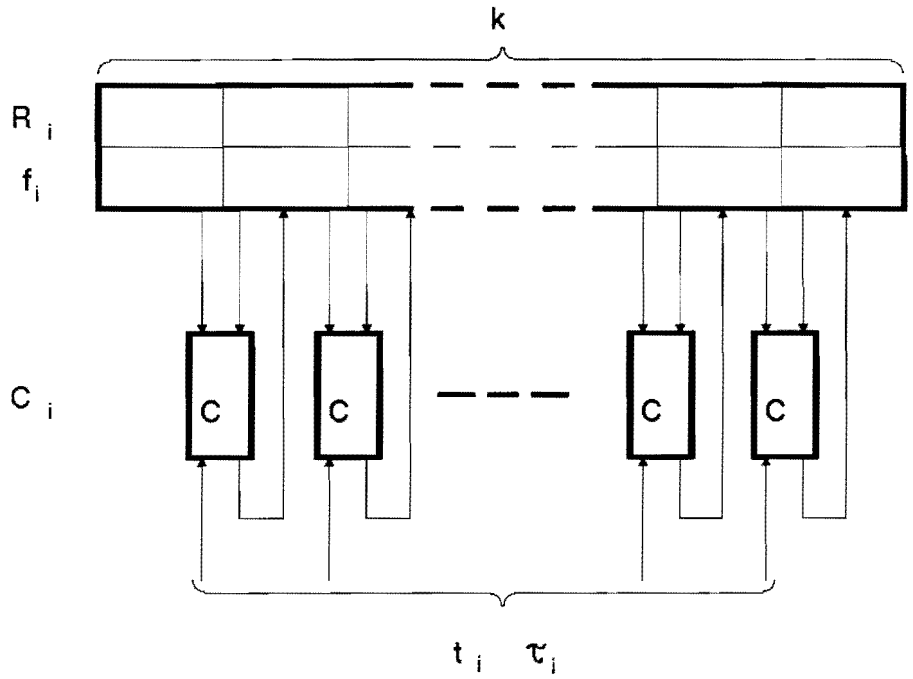

Figure 9: Massively parallel realization of the subdivision algorithm (The blocks C_i implement the computations of the R_i and f_i.)

		Computational unit			
		serial		parallel	
Sub-division-algo-rithm	ser.	M :	2	M :	6
		A :	2	A :	2
		S :	$2k(k+1)$	S :	$1/2k(k+1)$
		M · S :	$4k(k+1)$	M · S :	$3k(k+1)$
	par.	M :	$2k$	M :	$6k$
		A :	$2k$	A :	$2k$
		S :	$4k$	S :	k
		M · S :	$8k^2$	M · S :	$6k^2$

Table 1: Comparison of different approaches for the implementation of the bisecting unit (The four entries in each field are : number of multipliers (M), number of adders/subtractors (A), number of steps for the calculation of a new control point (S), product of M and S (M · S))

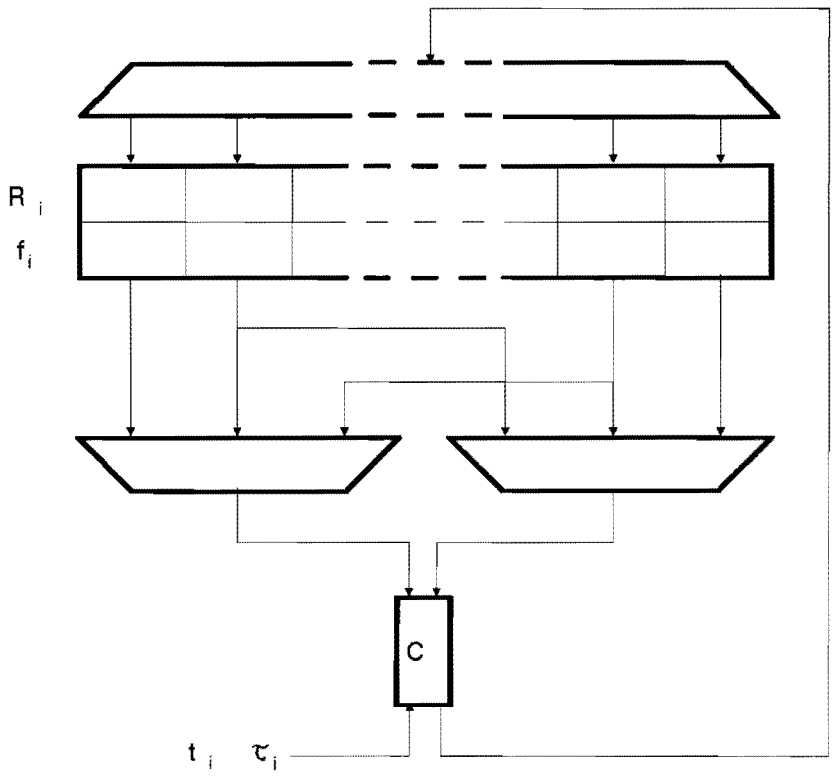

Figure 10: Serial realization of the subdivision algorithm

If such a solution wastes too much chip area, an implementation with less multipliers has to be considerd. In this case slower operation has to be taken into account. Figure 12 demonstrates one possibility.

The computation of the R_i and f_i takes two steps each. Hence in total four steps are necessary.

Table 1 summarizes the different realizations discussed above. If the total costs of the subdivision unit are defined as the product of the multipliers and the required steps one gets an estimation for the quality of each solution. It can be seen easily that the alternative with one parallelly implemented computational unit is best. This results in a bisecting unit that is built like the structure in Figure 10 with computational units from Figure 11.

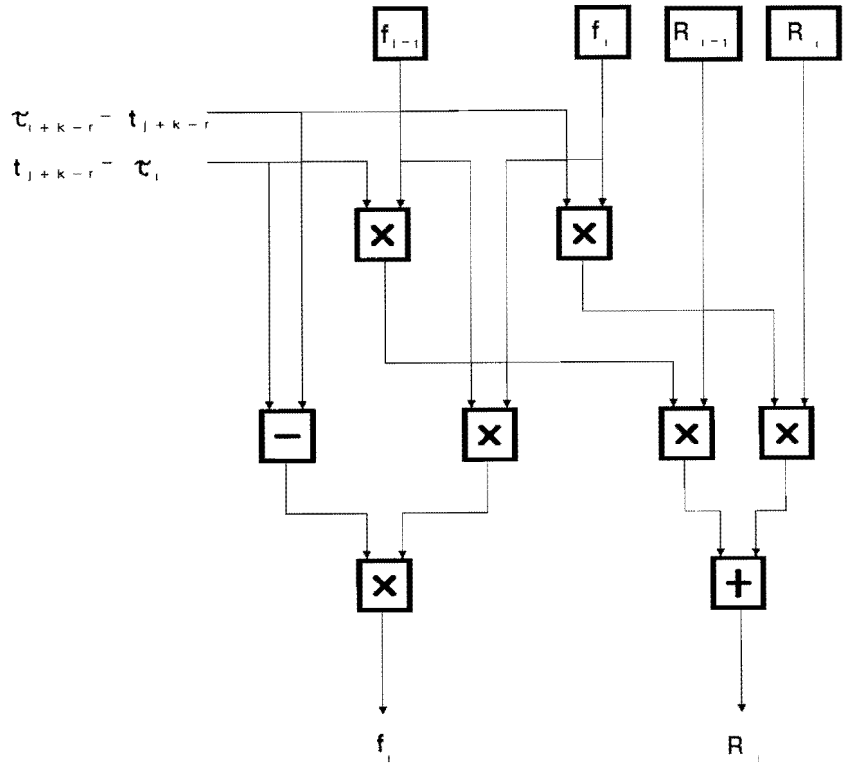

Figure 11: Parallel implementation of the computational units

Table 2 compares the computational units that are required for the OSLO-algorithm with those for the modified OSLO-algorithm. As expected each alternative of the modified OSLO-algorithm shows advantages compared to the original OSLO-algorithm. Either it is only a little bit slower but needs less silicon or it needs more chip area but is faster.

	OSLO-algorithm	modified OSLO-algorithm	
	parallel	serial	parallel
Multipliers	2	2	6
Adders / subtractors	2	2	2
Dividers	1	-	-
Time estimation	1D + 1M	4M	2M

Table 2: Comparison of the computational units for the OSLO-algorithm and its modification

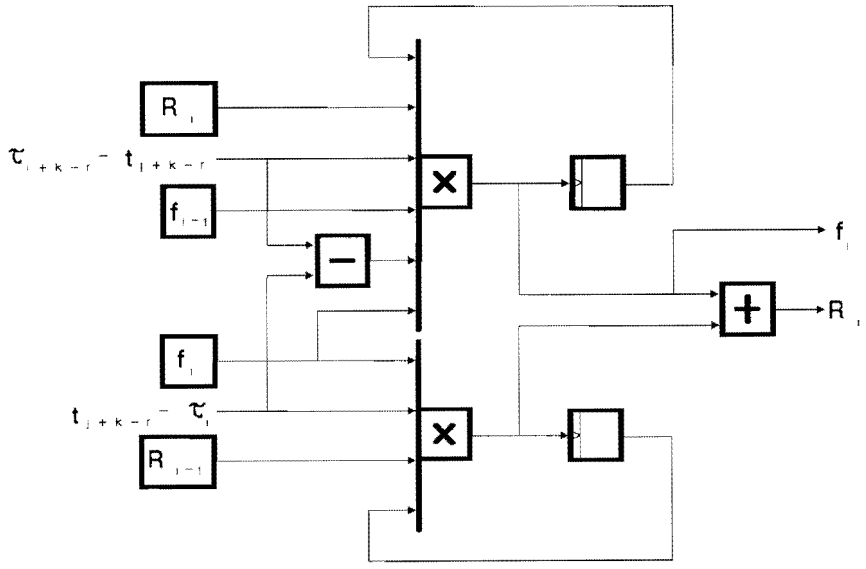

Figure 12: Serial implementation of the computational units

6. Some unsolved problems and a prospect

There are still some questions beyond the scope of this paper that have to be answered before a VLSI implementation may become reality.

First there are some architectural aspects. As already discussed in chapter 3 it is possible to distribute the intersect-and-subdivide algorithm on multiple processing elements. A structure that pays attention to the fact that usually not all subpatches are hit by the ray could be an attractive aim of further reasearch. To achieve an efficient use of the two stacks that are required by the subdivision in stages it is mandatory to develop good caching schemes to allow fast swapping of subpatches to and from the chip.

Another class of questions deals with limitations of the chip with regard to the order of the B-splines and the number of control points. More detailed investigations have to be done especially in the field of surfaces that can be generated by the rational B-spline scheme. Some results indicate that $k \leqslant 5$ and a control point net of 5 by 5 are sufficient.

If these question have been answered satisfactory such a chip will ease the use of normal and rational B-splines in geometric modelling systems. Since a great class of objects can be represented exactly by rational B-spline surfaces, including conic sections, spheres etc., this could be a way for a general and uniform discription of scenes. Hardware like this that assists rendering these objects will enable the fast generation of high quality pictures.

7. Conclusions

Starting from the desire to be able to use rational B-spline surfaces for the generation of high quality pictures, methods have been developed that allow the integration of rational B-spline surfaces into a ray tracer. The central part of all ray tracing algorithms is a fast algorithm for determining the intersection of object and ray. To achieve this for rational B-splines a subdivide-and-intersect algorithm has been presented. The crucial part of this algorithm is the subdivision algorithm. A modification of the OSLO-algorithm given in [10] has been employed. These algorithms have been discussed with regard to a VLSI implementation and various alternatives with different degrees of parallelism have been suggested. Using a quality criterion that contains the number of multipliers used and the computation time, one alternative has been chosen. In the end some unsolved problems have been mentioned for further research.

References

1. C. de Boor, "On Calculating with B-Splines.," *Journal of Approximation Theory* **6**(1) (July 1972).
2. W. Böhm, G. Farin, and J. Kahmann, "A Survey of Curve and Surface Methods in CAGD," *Computer Aided Geometric Design* **1**, North Holland (1984).
3. E. Cohen, T. Lyche, and R. Riesenfeld, "Discrete B-Splines and Subdivision Techniques in Computer-Aided Geometric Design and Computer Graphics," *Computer Graphics and Image Processing* **14** (1980).
4. T. Dokken, *A Method for the Display of Conic Sections and Parametric Polynomial Spline Curves*, Sentralinstitutt for Industriell Forskning, Oslo, 1983.
5. J.T. Kajiya, "Ray Tracing Parametric Patches," *Computer Graphics* **16**(3) (July 1982).
6. K. Klement, "Parametric Rational Curves of Second Degree and Conic Sections Leading to Rational Bèzier-Curves of Second Degree," FB Informatik, Inst. f. Graphisch Interaktive Systeme, TH Darmstadt.
7. L. Piegl, "A Geometric Investigation of the Rational Bézier Scheme of Computer Aided Design," *Computers in Industry* **7** (1986).
8. R.W. Pulleyblank and J. Kapenga, "A VLSI Chip for Ray Tracing Bicubic Patches," in *Advances in Graphics Hardware I*, ed. W. Strasser, Springer, 1987.
9. S.D. Roth, "Ray Casting for Modelling Solids," *Computer Graphics and Image Processing* **18** (1982).
10. H.-P. Seidel and B.-O. Schneider, *Towards a Hardware Implementation of B-spline Algorithms*, To appear 1987.
11. M.A.J. Sweeney, "Ray Tracing Free-Form B-Spline Surfaces," *IEEE Computer Graphics and Applications* (February 1986).
12. W. Tiller, "Rational B-Splines for Curve and Surface Representation," *IEEE Computer Graphics and Applications* (September 1983).
13. K.J. Versprille, "Computer-aided design and applications of the rational B-spline approximation form," PhD. Thesis, Syracuse University (1975).
14. T. Whitted, "An Improved Illumination Model for Shaded Displays," *CACM* **23**(6) (June 1980).