Ray Tracing Algorithms – Theory and Practice

Chapter · January 1988

DOI: 10.1007/978-3-642-83539-1_42

Wolfgang Leister
Norwegian Computing Center
129 PUBLICATIONS **531** CITATIONS
SEE PROFILE

Heinrich Müller
Technische Universität Dortmund
244 PUBLICATIONS **2,090** CITATIONS
SEE PROFILE

Ray Tracing Algorithms – Theory and Practice*

Alfred Schmitt Heinrich Müller Wolfgang Leister

Institut für Betriebs- und Dialogsysteme
Universität Karlsruhe
West Germany

Abstract. This paper gives a survey on recent advances in making ray tracing a practical technique for realistic image synthesis from spatial scenes. First, the structure of the Karlsruhe ray tracing software VERA is described as an example of practical ray tracing. Then a comprehensive analysis of worst case time bounds of the ray tracing procedure and of some related algorithms, e.g. the visible surface reporting algorithm, is carried out.
Keywords. Computer graphics, computational geometry, realistic image synthesis, ray tracing, visible surface reporting, ray query problem.

1 Photographic Realism in Computer Graphics

One of the goals of todays work in computer graphics is to generate more realistic and more detailed images. Correct light simulation and realistic surface texture adds a lot more of information to a picture. Today, some of the possible applications of computer graphics strongly depend on the attainable degree of realism. Fields of applications include CAD, animation and visualization, robotics, architecture and inside decoration, advertising, reconstruction for medical and other purposes.
What is a realistic image? The correct simulation of various lighting effects is the most important part of true photographic realism. In order to explain this in some detail, we present a series of pictures showing always the same chess still life with some of the chesspieces in unusual positions, cf. figures 1a to 1e.
Perfectly looking realistic images can only be generated on powerful computers using sophisticated software. This holds in particular for *ray tracing*. Ray tracing was brought to a broad knowledge by Whitted [Wh80]. In its simplest form, for each pixel of the image a ray is traced from the viewpoint into the 3d scene to calculate its first intersection with an object, cf. figure 1f. If the object is reflecting or refracting, an appropriate ray is determined by the law of reflection or refraction. These new rays are treated analogously. In order to calculate shadows, we must find out if the ray between the intersection point and the light source is intersected by another object. If there is an object intersecting

*Published in: R.A. Earnshaw (ed): Theoretical Foundations of Computer Graphics and CAD, NATO ASI Series, Series F: Computer and Systems Sciences, Vol. 40, Springer-Verlag, 1988, pp. 997–1029.

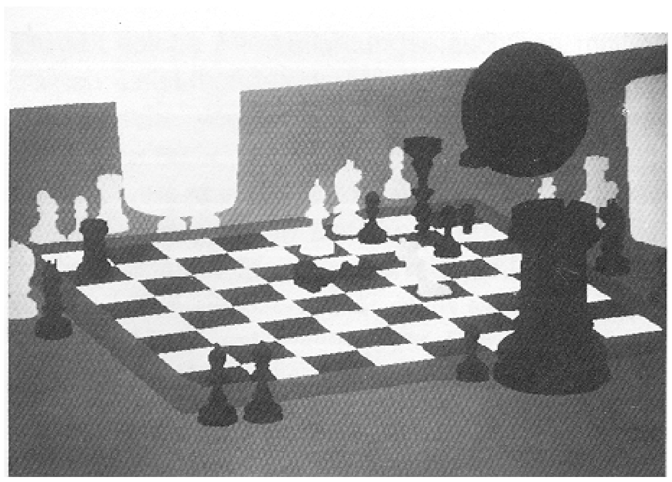

Figure 1a. This picture was generated without any lighting effects based on viewing angles and without a depression of pixel stair cases. The objects look flat and some of the geometric details are not at all visible, e.g. the correct shape of the chess board.

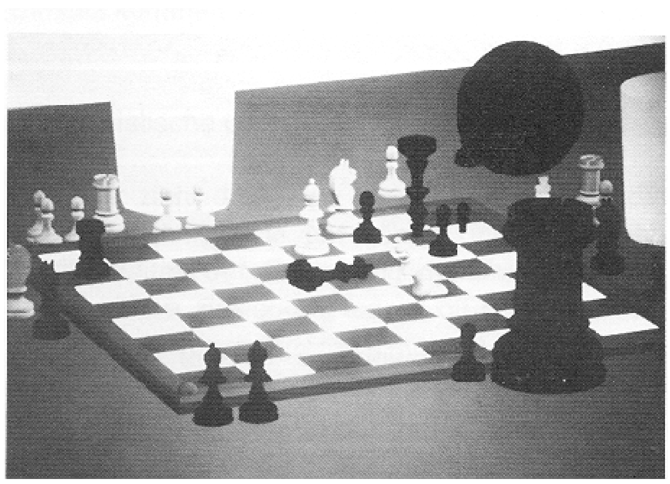

Figure 1b. If anti-aliasing is switched on, the stair cases (jaggies) disappear. The anti-aliasing procedure is based on a controlled oversampling.

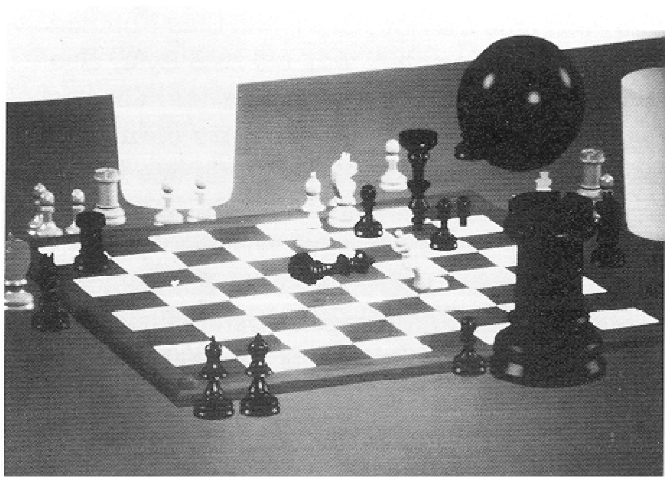

Figure 1c. Here the diffuse angle-dependent part of the light as well as anti-aliasing was switched on. The rendering of the left silver beaker as well as the up till now black glass sphere is far from being realistic since shadows, mirror and glass effects are still switched off.

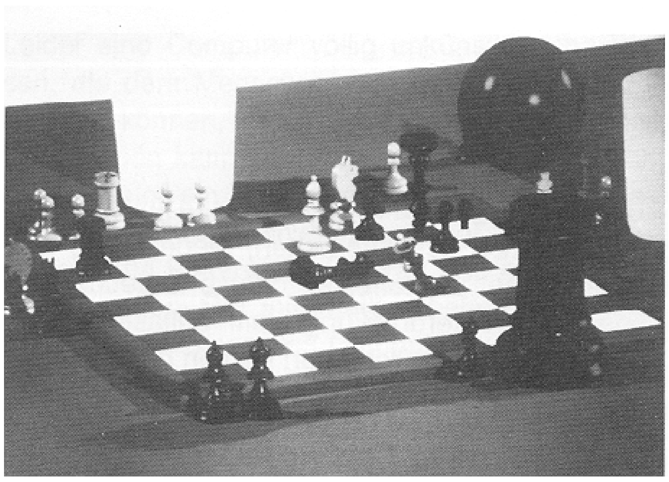

Figure 1d. Three light sources as well as hightlight effects are added. Almost all of todays computer animation sequences are generated using exactly this lighting model. The ubiquitous and simple depth buffer strategy is sufficient to generate this kind of pictures.

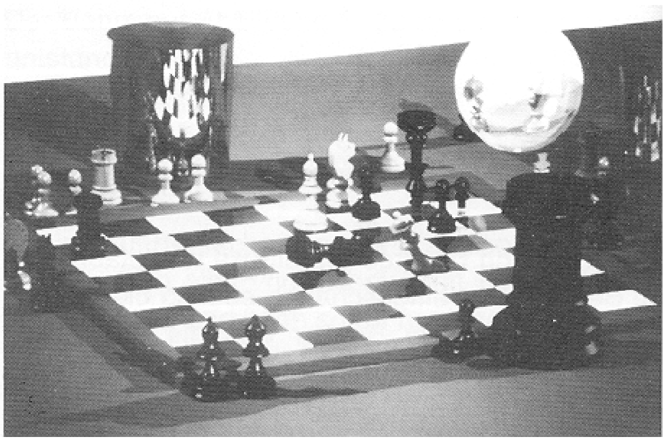

Figure 1e. One of the strong points of the general ray tracing procedure is its ability to visualize mirror and glass objects in a natural way. But there are some more effects that improve realism, e.g. surface texture, secondary light effects, lens distortion, and motion blur.

this ray, the intersection point lies in the shadow of this light source, and its intensity is not taken into account for intensity calculations. The calculation of intensity is carried out according to a lighting formula incorporating parameters of material attached to the geometric objects of the scene.

2 The VERA Ray Tracing Software - Structure of an Implementation

VERA (**V**ery **E**fficient **R**aytracing **A**lgorithm) is able to generate highly realistic raster images of scenes consisting of a great number of objects. The program, written in standard PASCAL, is portable to almost any computer. It consists of three main parts. In a first pass, the scene is read in, while the second pass transforms the scene into a suitable internal data structure for the third pass, where raytracing is done. The output is a compressed raster image.

2.1 Specifying Realistic Spatial Scenes

The description of a scene is done textually, according to the following general syntax:

```
Scene := Projection {LightSource} Geometry {Material} {Subscene}
  Projection   := PROJECTION ViewPoint ImagePlane
  LightSource  := LI Point Intensity Color
  Geometry     := {Sphere | Triangle | RTPatch |
                 RotationalShape | SubSceneCall | Attribute}
    Sphere     := SP Center Radius
```

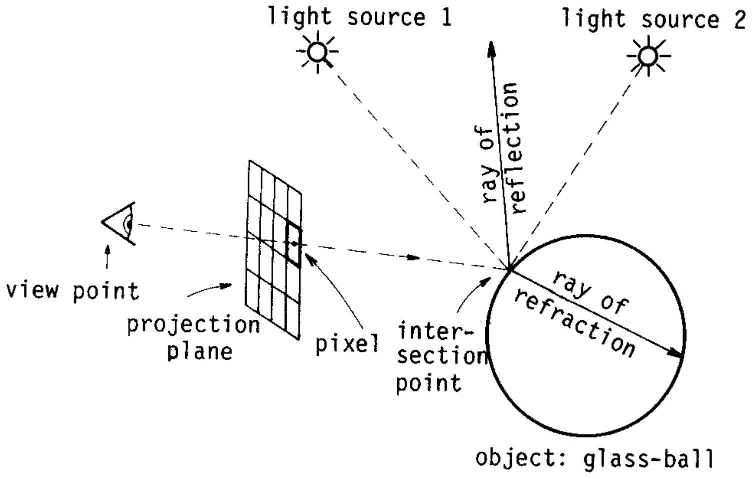

Figure 1f. The principle of ray tracing.

```
    Triangle   := P3 Point1 Point2 Point3
    RotationalShape := RS Axis Contour
    RTPatch    :=  PA Point1 Point2 Point3 Normal1 Normal2 Normal3
    SubSceneCall := SSC SceneIdentifier Transformation
        Transformation :=  MX 3x4-matrix | RT angles | TR vector | SC factor
    Attribute  := CM MatIdentifier | IP point
  Material     := MA MatIdentifier Diffuse SelfLuminosity Reflection Refraction
    Diffuse    := DRF Color ARF Color
    SelfLuminosity := SDI Color SAN Color
    Reflection := RAB Color SPC Aberration
    Refraction := TAB Color
  SubScene     := SCENE SceneIdentifier Geometry
```

VERA accepts a set of primitive objects, which are used to build up a scene. These primitives are triangles (P3), spheres (SP), and rotational shapes (RS). The attribute “IP” (inner point) lies on the inner side of a boundary face of a solid. This makes it unnecessary to specify solids explicitly. The design of more complex scenes is greatly simplified by the concept of hierarchy. It is possible to specify subscenes (SCENE) that can be used several times (SSC) under different transformations (rotational, translational, scaling). This technique is very similar to the procedure- or macro-concept of programming languages. The description of the scene belonging to figure 2a is

```
PROJECTION       54.0    75.0    200.0    (* view point              *)
                 27.0     0.0      0.0    (* lower left              *)
                148.0     0.0      0.0    (* lower right             *)
                 27.0   121.0      0.0    (* upper left of the image *)
LI 110.0 100.0  -25.0  13000.0  1.0 1.0 1.0  (* light source 8000    *)

(* geometry *)
CM darkwhite
```

```
IP 0 -10 0                                           (* floor *)
P4 -200 0 100   800 0 100   800 0 -200   -200 0 -200
CM brightgrey
IP 80 20 -28
P4 90 0 -15     155 0 -15   155 40 -15   90 40 -15
P4 155 0 -15    155 0 -55   155 40 -55   155 40 -15
P4 90 0 -55     155 0 -55   155 40 -55   90 40 -55  (* parallelepiped *)
P4 90 0 -15     90 0 -55    90 40 -55    90 40 -15
P4 90 40 -15    155 40 -15  155 40 -55   90 40 -55
(* is open at the bottom *)
CM refwhite
SP 55 25 0 25                 (* sphere *)
CM mirror
IP 0 40 -100
P4   0 4 -85    160 4 -85    160 120 -85      0 120 -85    (* mirror *)
CM grey
P4  -4 0 -85    164 0 -85    164 4 -85       -4 4 -85
P4 160 4 -85    164 4 -85    164 120 -85     160 120 -85    (* frame *)
P4  -4 120 -85  164 120 -85  164 124 -85     -4 124 -85
P4  -4 4 -85    0 4 -85      0 120 -85       -4 120 -85
CM whiteluminant
SP 110 120 -25 4                (* lamp *)
CM darkgrey
RK 110 120 -25    P    110 160 -25         (* wire of lamp *)
AN 0 0   GE 1 0   GE 1 50   GE 0 50

(* materials *)
MA grey          DRF 0.7 0.7 0.7
MA whiteluminant DRF 0.9 0.9 0.9  SDI 1.0 1.0 1.0
MA darkgrey      DRF 0.4 0.4 0.4
MA mirror        SPC 1.0  RAB 0.7 0.7 0.7 REF 1.0
MA refwhite      DRF 0.2 0.2 0.2  ARF 0.5 0.5 0.5  SAN 0.5 0.5 0.5
MA brightgrey    ARF 0.8 0.8 0.8  DRF 0.2 0.2 0.2
MA darkwhite     DRF 0.8 0.8 0.8
```

A file like this is either specified by a text editor, or results by conversion of the output of an interactive graphics editor or CAD system. The textual format allows easily to transfer files to other computers. This is important in an environment where computational power must be gained from several machines.

2.2 Preprocessing

The specific power of VERA is to process large scenes in relatively moderate time. This is made possible by decreasing the number of intersection tests to be performed between rays and primitives by preprocessing the scene into a suitable data structure. The quality of this data structure is crucial for the practicability of ray tracing. Straightforward ray tracing would require to test every ray against every object. For a number of rays up to several millions and a number of objects up to several 10000 or 100000, the time

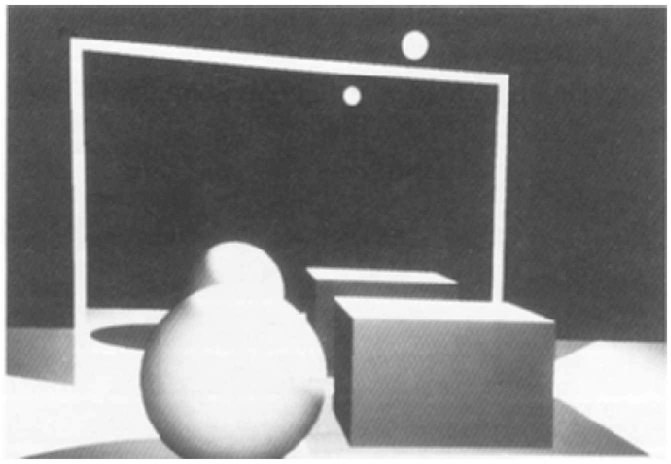

Figure 2a. A black and white still-life.

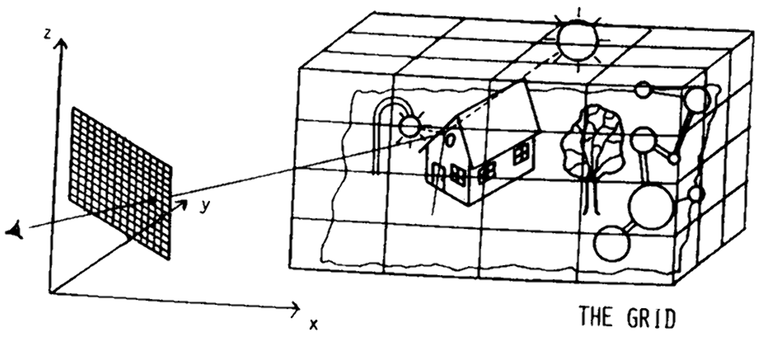

Figure 2b. The Grid Structure.

requirements are days or even weeks for one image.
In the preprocessing phase, VERA partitions the bounding box of a scene by a regular grid into congruent cells, like depicted in figure 2b. For each cell, a list of objects is made up, containing all those objects that have a nonempty intersection with the cell. In fact, only pointers onto the detailed description of the objects are stored in the list, thus keeping the space requirements for this data structure relatively moderate. Subscenes are treated analogously to primitves, using their bounding box for assigning them to cells. The resulting data structure is a tree of grids, which usually shows a good adaption to the distribution of the primitives of the scene.

2.3 Tracing the Rays

Ray tracing is carried out by a recursive procedure *RayTrace*. This procedure successively reports all cells of the grid structure of a scene that are intersected by the ray. The key

feature of the grid approach is that reporting the cells intersected by a query ray can be implemented in a highly efficient manner by using a 3d-vector generator. There are well known efficient algorithms to draw vectors on a 2d raster display as the DDA or Bresenham-algorithm, that can immediately be generalized to 3d space. These methods only require incremental additions and subtractions. The consequence of the 3d vector generator technique is, that the overhead for grid traversal is neglectable, while the restriction of search space by the grid is fully available.
For the objects in the list belonging to a reported cell, an intersection test is carried out by a procedure *IntersectionTest*. This is straightforward for the primitives. For subscenes, the ray is transformed into the grid of the subscene and is processed analogously. If a cell behind the currently closest intersection point is entered by the ray, a closest intersection point is found. This implies that reporting cells is terminated, and a recursive call of *RayTrace* is performed for mirroring or transparent objects. Rays to the light sources are treated analougously.

Illumination

If an intersection point is found, its contribution to the total ntensity of the pixel under consideration must be calculated. This is done according to the following lightness formula. For each of the basic colors red, green and blue we calculate the color-intensity

$$Int_c = Amb_c + Dif_c + Spk_c + Slm_c + Rfl_c + Rfr_c$$

where Amb_c is the ambient light, Dif_c the diffuse light, Spk_c the specular light, Slm_c the self-luminosity, Rfl_c the reflecting light from mirrors, and Rfr_c the refracted light from glass. Let $\vec{N}$ be the normal vector to the object, $\vec{R}$ the ray that intersects the object, $\vec{S}$ the vector of reflection, $\vec{F}$ the vector of refraction and $\vec{L}_i$ the vector from the intersection point to the light source i. Then

$$Amb_c := (\vec{R} \cdot \vec{N} \cdot DRF_c + ARF_c) \cdot AmbLight_c$$

$$Dif_c := \sum_i (\vec{L}_i \cdot \vec{N} \cdot DRF_c + \vec{L}_i \cdot \vec{R} \cdot ARF_c) \cdot Intens_{i,c}.$$

The intensities of the light sources depend on the color, basic intensity and distance.

$$\text{If } \vec{S}_i \cdot \vec{L}_i < SPC \text{ Then } Spk_c := 0$$

$$\text{Else}$$

$$Spk_c := \sum_i \left(\frac{(\vec{L}_i \cdot \vec{S} - SPC) \cdot (\vec{L}_i \cdot \vec{S} + SPC - 2.0)}{distance \cdot (1.0 - SPC)^2} \cdot Intens_{i,c} \right)^2$$

$$Slm_c := \vec{R} \cdot \vec{N} \cdot SAN_c + SDI_c$$

$$Rfl_c := RAB_c \cdot (colour\ of\ recursive\ call\ for\ reflection)_c$$

$$Rfr_c := TAB_c \cdot (colour\ of\ recursive\ call\ for\ refraction)_c.$$

The parameters used here are specified in the scene description, cf. sect. 2.1. Besides the well known coefficients for reflection we also use a coefficient for self-luminosity, both angle-dependent and diffuse. It has the advantage that an object can be seen without the specification of a light source, e.g., glowing metal. So we can design objects having no shadow on their surfaces, even if there are many shadows in the environment.

Displaying the Image

The image is calculated with 24 bits for each pixel (one byte for each basic color red, green and blue). At a resolution of 512×512 or more the resulting amount of data is considerable. VERA reports the image in a compressed manner, using a combination of run-length coding and interpolation. Based on this representation, reduction of colors according to more limited displays using color tables is possible, using dithering techniques like that of Floyd and Steinberg [FS75].

Advanced Features

With the set of primitives used by VERA, large scenes of many distinct subjects can be built up. On the other hand, it is possible to approximate more complex shapes, like smooth or fractal surfaces, by these primitives. For smooth surfaces it is of advantage to use RT-patches instead of triangles. Then VERA interpolates the surface normal similar to the shading method of Phong.
A very important feature to obtain realistic looking images is the firmament-function, where a heaven-earth model is built in. This allows to design the shape of the horizon at the bottom of the scene and over there a sky with a continuous change in colours. Another command allows some automatic exposure measurement. Before the ray tracing of the whole image, the colors of some representative rays are calculated. This allows one to detect whether the image will be too dark or too bright. In such cases we can switch on an option to embrighten or darken the light sources automatically. Further, if desired, VERA chooses the parameters of the image plane so that the whole scene is mapped into the image. Finally, the shape of pixels can be chosen according to the aspect ratio of the monitor. These informations are collected in a control file which is read by VERA in addition to the scene file. A typical control file is shown below.

```
RASTER 4096 2732 (* horizontal/vertical resolution *)
RTDEPTH 2 (* depth of ray tracing *)
EXPMES 0.0 (* exposure measurement *)
AMBIENT 0.5 0.5 0.5 (* switch on ambient light source *)
SKY 0.2 0.2 0.2 (* firmament function *)
HORIZON 0.1 0.1 0.2
EARTH 0.3 0.3 0.3
SHADOWS (* switch on shadows *)
ANTI-ALIAS (* switch on anti-aliasing *)
```

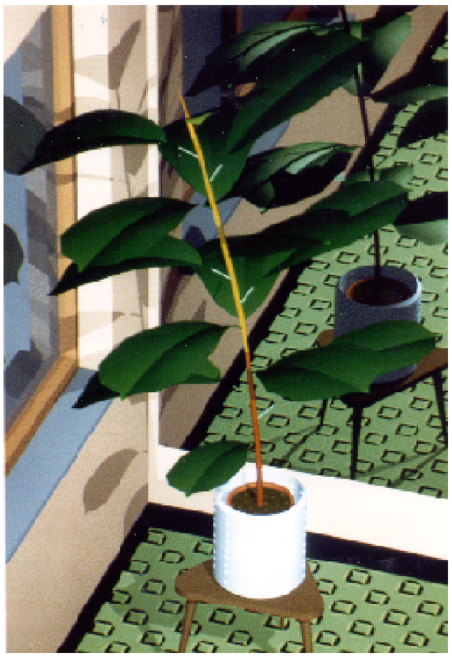

Figure 2c. The ficus plant (rubber tree) shown here was generated by software simulating the growing process of this species. The leaves are modelled by thousands of small triangles. Interior decoration is a possible field of application of photographic realism. Design: C. Cabart, ENS, Paris, France, and G.W. Bieberich, Karlsruhe.

2.4 Examples of Ray Traced Pictures

The examples of pictures in Figure 2c to 2f were all generated by the VERA ray tracing system.

3 Worst Case Time Analysis of Ray Tracing

3.1 The Ray Query Problem

Crucial for the efficiency of ray tracing is to find quickly that geometric object out of a given scene delivering the intersection point closest to a ray's initial point. This central problem of raytracing can be formulated as follows.
Ray Query Problem.
Input. n primitives (triangles, spheres, patches, etc.), a ray.
Output. A geometric object intersected by the ray, closest to its origin.
The aim of the algorithms in this section is to reduce the number of ray-primitive intersection tests by preprocessing the primitives of the given spatial scene into a data structure. In practice, the number of scene primitives can be expected large, while the spatial extension of each primitive is relatively small. This observation explains the

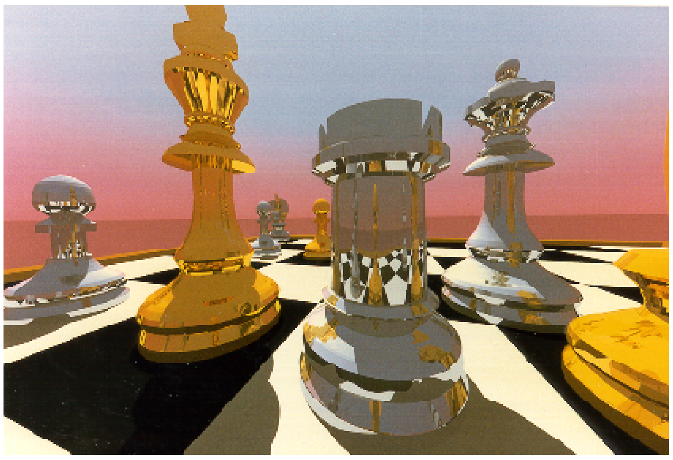

Figure 2d. An artist's view of a chess board. The chess pieces are almost completely modelled by a number of conic sections. Resolution of the slide is 4096 × 2732 pixels. Design: G.W. Bieberich, Karlsruhe.

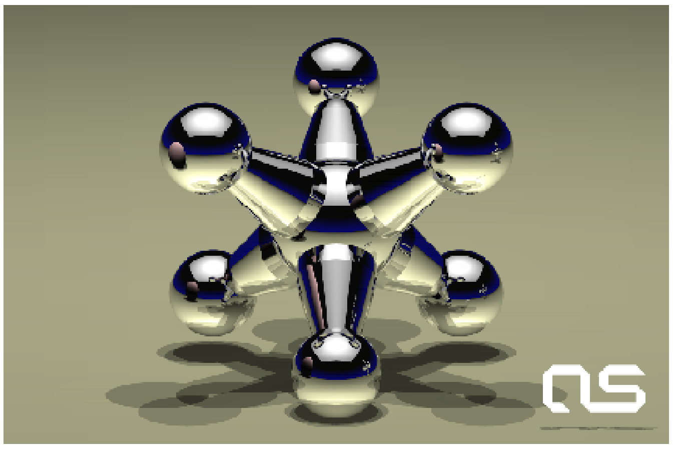

Figure 2e. The lighting model of the VERA raytracer is well suited to render even difficult materials. Perfect rendering of such parts is only possible if the scene defines not only a body but a complete environment. Design: A. Stößer, Karlsruhe.

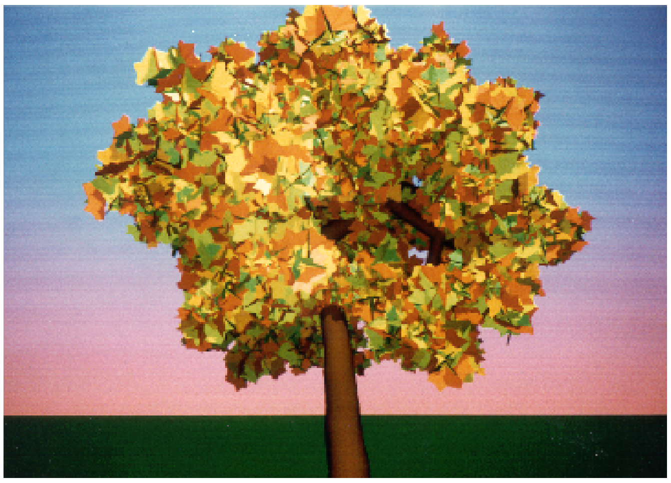

Figure 2f. This tree was generated by a recursive procedure that uses only some basic laws describing the relations between branches and leaves. Such methods are used to design complex graphical objects. Quite similar approaches help to generate so-called fractals. Design: T. Maus, Karlsruhe.

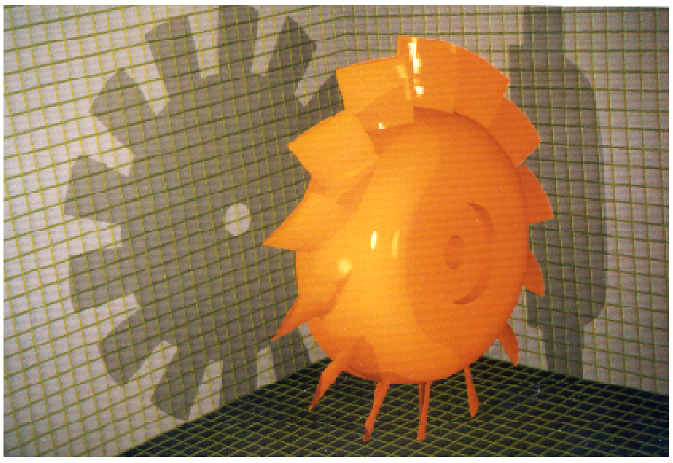

Figure 2g. This cooling rotor is an optimized design. It was developed by a mechanical engineering institute of the University of Karlsruhe. The realistic raytraced picture was used to check the design. An automatically milled model was available only several months later. Design: Doneit, Bieberich, Karlsruhe.

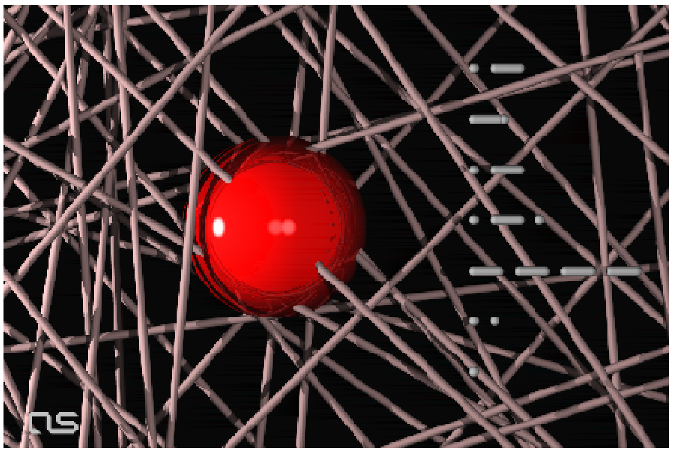

Figure 2h. Still-life demonstrating photographic realism. Design: A. Stößer, Karlsruhe.

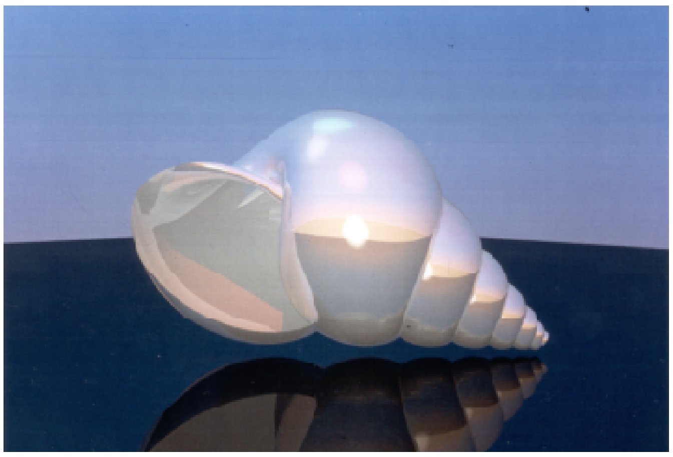

Figure 2i. This snake shell is simply a raytraced (u, v)-parametric surface. The formula is

$$X(u, v) := 1.66^{\frac{v}{2\pi}} \cdot (0.1 \cdot \cos v + 0.125 \cdot \cos v \cdot \sin u),$$

$$Y(u, v) := 1.66^{\frac{v}{2\pi}} \cdot (0.1 \cdot \sin v + 0.125 \cdot \sin v \cdot \sin u),$$

$$Z(u, v) := 1.66^{\frac{v}{2\pi}} \cdot (0.6 + 0.25 \cdot \cos u).$$

The surface is approximated by a sufficiently large number of raytracing patches. This approximation is carried out in a preprocessing step. Design: V. Vlassopoulos, Athens, Greece.

usefulness of *spatial subdivision* [TK84,Gl84,FT85,M86a] as it was applied in the VERA system.
The other approach favored by some implementors is *hierarchy of hulls* [RH82,KK86,Gv86]. Two or more spatially related primitives are clustered into a subscene, subscenes are again clustered together, until a final composite scene is reached. To every node of this tree of scenes, a bounding box of the subscene's primitives is assigned. On this preprocessed tree, ray tracing is performed starting up at the root, and traversing the tree according to the intersected nodes. A survey on clustering methods is given in [DE83]. For tree-like specified scenes, i.e. hierarchies and constructive solid geometry [M86a,RH82], clustering is straightforward by taking subtrees as clusters.
Space decomposition and hierarchy of hulls work well in practice, but theoretically they may behave badly in worst case. The remainder of sect. 3 is devoted to worst case efficient solutions of the ray-primitive intersection test. We present worst case efficient solutions involving classical data structures of computational geometry covered by textbooks like e.g. those of Preparata and Shamos [PS85] and Mehlhorn [Mh84]. The reader is supposed to be familiar with the basics of these data structures. Our more theoretical investigations help to grade ray tracing as relatively complex among other problems of computational geometry. However, they show new directions that may be of practical interest too.

3.2 Ray Tracing without Mirrors and Glass: Simple Ray Tracing

If the 3d scene does not contain light reflecting or refracting materials, the ray tracing procedure has to answer only so-called centrally distributed ray queries. A ray starts at the view point (primary rays) or it starts at one of the L light sources. In this case and if the 3d scene consists of plane polygons that do not penetrate each other, there is a nice solution to show, that a single ray query can be answerd in time of $O(\log n)$, where n is the total number of polygon vertices of the scene.
The basic idea is to transform a central ray query problem to an equivalent plane point location problem. The details of this construction are as follows:

1. Choose for example the view point A as the central point of our ray query problem. Thus we have to solve the ray query problem only for those rays having A as starting point.

2. Wrap up the view point by a box consisting of six regularly oriented square faces, see figure 3a for details.

3. For each of these square faces solve the visible reporting problem with the view point as the central point of the projection and the square face as its screen plane. Details of an applicable algorithm are given in the appendix. There it is shown, that it is possible to perform visible surface reporting in time of $O((n + k) \log n)$, where k depends on the special nature of the 3D scene.

4. For each of these six sets of visible parts of polygons generate a data structure in order to solve the following point location problem at the surfaces of the square

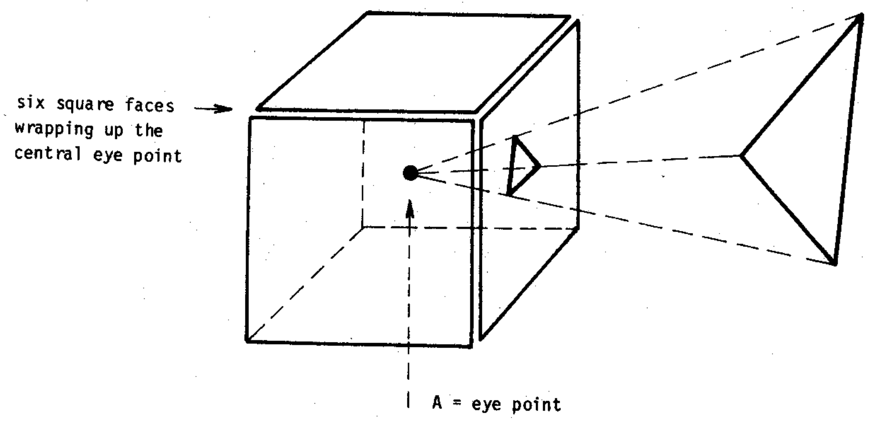

Figure 3a. The centrally distributed ray query problem can be solved by six independent 2d point location problems where the point regions correspond exactly to those parts of the surfaces of the 3d scene which are visible from the view point.

faces: The regions of the point location problem correspont exactly to the visible parts of polygons when projected to the square faces. It was shown by Kirkpatrick [Ki83] that the point location problem based on m triangular regions in the plane needs preprocessing time of $O(m \log m)$ and answers a single point query in time of $O(\log m)$. In our case, m is of the order of $O(n + k)$ in the worst case. Since k is at most of $O(n^2)$, we have a total preprocessing time of $O((n + k) \log(n + k)) = O((n + k) \log n)$ and a single query time of $O(\log n)$.

5. A centrally distributed ray query is now reduced to a single point location query: Determine the square face penetrated by the ray (starting at the view point A) and calculate the penetration point. Use this point in order to solve the point location problem attached to the penetrated square face.

As a result of this construction we can state: Single (central) ray query problems of the specified structure can be answerd in time of $O(\log n)$ and the preprocessing can be done in time of $O((n + k) \log n)$.
In order to derive specific results on the time complexity of simple ray tracing, we first note, that the calculation of a single pixel generates exactly $L + 1$ centrally distributed ray query problems, where L is the number of light sources given for the scene. This amounts to a time bound of $O((L + 1) \log n)$. The preprocessing needs total time of

$$O((L + 1)(n + k_{max}) \log n)$$

and the total time to generate a complete picture with m_x horizontal and m_y vertical pixels is bounded by

$$O((L + 1)(n + k_{max}) \log n + (L + 1) * m_x * m_y * \log n).$$

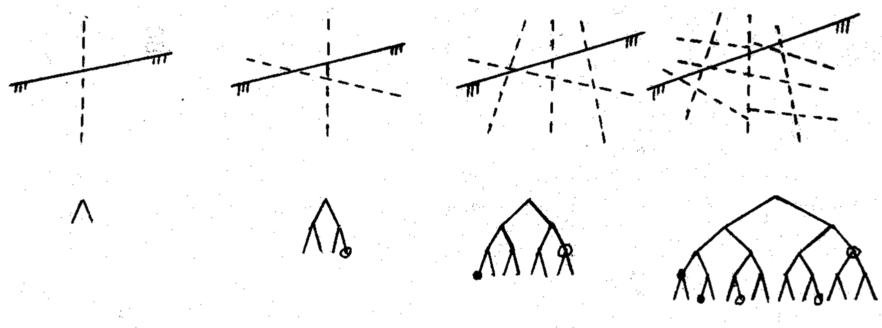
nodes where search terminates: • region outside / o region inside the query plane

Figure 3b. The conjugation tree, and a query half plane. The dashed lines define the partition. Every one of them halves two previous regions w.r.t. the number of points. The given set of points is omitted.

It should be noted that simple ray tracing is more of academic than practical relevance. The method was included here in order to show, that the centrally distributed ray query problem has an efficient solution, at least in theory. Consequent application of visible surface reporting allows an even more direct procedure to generate shaded pictures of the same quality. For details see section 3.4.

3.3 The General Ray-Primitive Intersection Problem

3.3.1 The Query Approach Revisited

In this section we introduce two data structures for the general ray query problem with a nontrivial time bound. To do this, let us restrict the geometric primitives to iso-oriented rectangles orthogonal to the z-axis. Scenes of this type arise from the faces of the rectangular bounding boxes of arbitrary primitives, according to the standard technique of computer graphics to carry out computations as far as possible with these simple approximations. Further, we assume the rays replaced by lines. We are looking for an intersection point of a line with a rectangle, having a minimum z-coordinate. The first data structure is based on nested conjugation trees. The conjugation tree developed by Edelsbrunner and Welzl [EW86] defines a hierarchical decomposition of a finite set of points by ham-sandwich-cutting, cf. figure 3b. The conjugation tree efficiently supports halfplane queries, i.e. to find out those points on, say, the right side of a line. A ray intersection corresponds to four simultaneous halfplane queries, i.e.

$$x^o \geq a * z_o + b,\ x_o \leq a * z_o + b,\ \text{projection on x} - \text{z} - \text{plane}$$

$$y^o \geq c * z_o + d,\ y_o \leq c * z_o + d,\ \text{projection on y} - \text{z} - \text{plane.}$$

(x_o, y_o, z_o) and (x^o, y^o, z_o) are the lower left resp. upper right vertices of some rectangle. These nested queries can be solved by four nested conjugation trees, according to a more general technique developed by Dobkin and Edelsbrunner [DE84]. The computational complexity is summarized by

Proposition 1. It is possible to preprocess n iso-oriented rectangles within $P(n) = O(n \text{ polylog } n)$ time (assuming ham-sandwich cutting done according to Meggido [Me85]) into a data structure of size $O(n \text{ polylog } n)$ s.t. an arbitrary ray query for a closest rectangle can be answered within $Q(n) = O(n^{\alpha}\text{polylog } n)$ time, $\alpha = \log \frac{1+\sqrt{5}}{2} \leq 0.695$ (polylog stands for $\log^c$ for some $c \geq 0$).

A slightly better α can be obtained by applying the ϵ-nets due to Haussler and Welzl [HW86]. However, the question of efficent calculation of ϵ-nets seems unanswered.

A totally different approach is based on point location. The idea is to restrict the query space consisting of all lines to a restricted set of lines with the same answer, as follows. The edges of the rectangles are segments of lines. The query line is moved in parallel to the x-axis to the first line perpendicular to the x-z-plane, and then in parallel to the y-axis to a first intersecting line perpendicular to the y-z-plane, cf. figure 3b. This means that only those query lines must be considered further that connect two such orthogonal lines. This problem is equivalent to the inverse range query problem for rectangles in the plane, i.e. to find the rectangles containing the query point. The perpendicular lines onto which a query line is to be moved is found by point location, as shown in figure 3c and in detail described by Chazelle [Ch83]. The complexitiy of the resulting solution is as follows.

Proposition 2. It is possible to preprocess n iso-oriented rectangles within $P(n) = O(n^3\text{polylog } n)$ time into a data structure of size $O(n^3\text{polylog } n)$, s.t. an arbitrary ray query for a closest rectangle can be answered within $Q(n) = O(\log^3 n)$ time.

3.3.2 The Primitive Query Approach

Image generation by raytracing is usually done pixel by pixel, i.e. all rays originated by a pixel are traced before work continues at the next pixel. This is a very space efficient approach, as long as the number of primitives is small compared with the number of rays, since no rays must be stored. However, the possibility to make use of ray coherence is rather limited [SD85]. An alternative strategy curing this disadvantage is to treat the rays of view simultanously. Their closest intersection points and the rays of reflection and refraction, if any, starting at these points, are calculated. The rays of reflection and refraction make up a new generation of rays, that is treated analogously. In addition, the rays from the intersection points to the light sources make up another generation of rays to be handled simultanously.

The central task in this *generation-wise ray tracing* algorithm is to find for a 'query primitive' its intersecting rays out of the current generation of rays. Let us discuss this problem for iso-oriented rectangles and lines like in the previous section. The intersection condition from there can be reformulated into two strip range queries on points in the plane, i.e.

$$x_o - a * z_o \leq b \leq x^o - a * z_o, \text{ projection on x} - \text{z} - \text{plane}$$

$$y_o - c * z_o \leq d \leq x^o - c * z_o, \text{ projection on y} - \text{z} - \text{plane.}$$

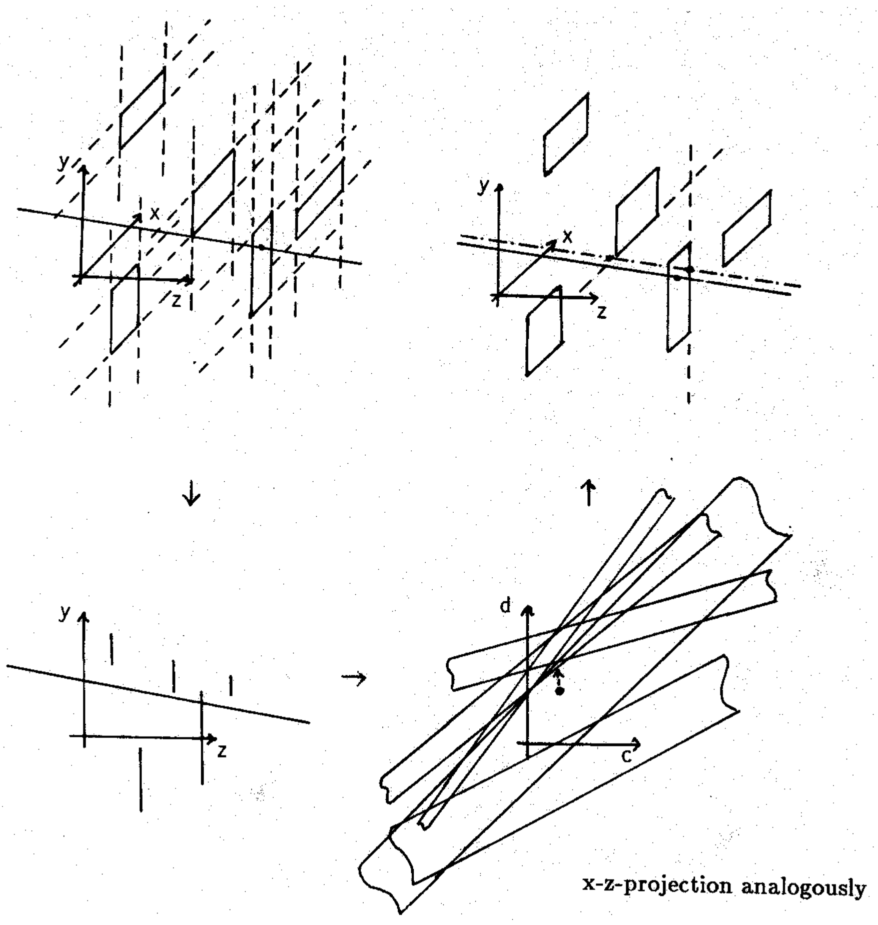

Figure 3c. Time-efficient answering of ray queries. The query ray is moved to a special location, using the dual space of its coefficients.

The strips are initialized by the query rectangle, while the points stem from the lines. These simultanous strip queries can be solved by preprocessing the lines into a two-fold nested conjugation tree, again following Dobkin and Edelsbrunner [DE84]. The computational complexity is stated here:

Proposition 3. It is possible to preprocess m rays within $P(m) = O(m \text{ polylog } m)$ time into a data structure of size $S(m) = O(m \text{ polylog } m)$ s.t. the set of all rays intersected by an arbitrary query rectangle (fixed orientation) can be found within $Q(m) = O(I + m^{\alpha}\text{polylog } m)$, $\alpha = \log \frac{1+\sqrt{5}}{2}$, I the number of intersections reported.

The next data structure answers a primitive query in polylog m time, at the expense of $O(m^2\text{polylog } m)$ storage. The lines of a generation are projected onto the x-z- and y-z-planes. Their intersection points divide the lines into line segments. These line segments are organized into a segment tree, cf. figure 3d. For the line segments stored with a node of the segment tree, a 2-d range tree is built up. These line segments intersect a plane perpendicular to the z-axis with intersection points in the same relative x- and

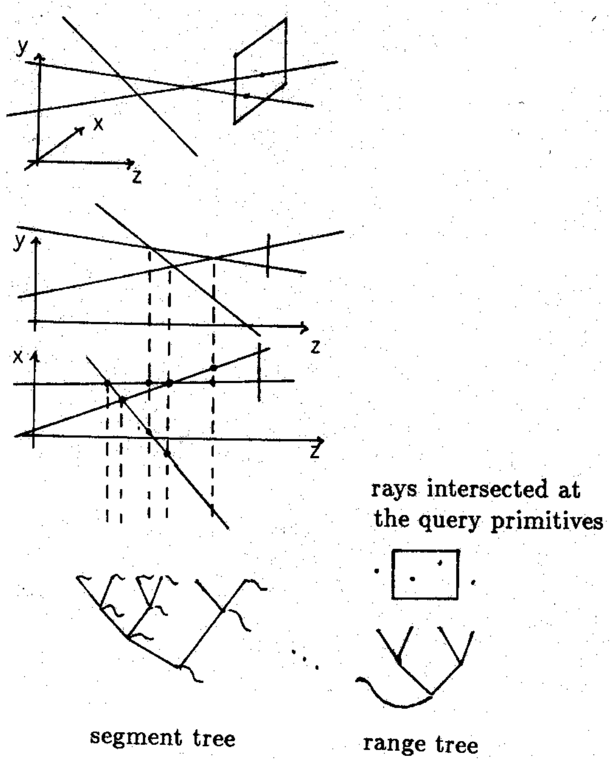

Figure 3d. Answering a query of a primitive. The primary tree is a segment tree, the secondary trees at its nodes are 2d range trees for the respective line segments.

y-order within the range of the segment tree node. The range tree is built up on these intersection points. It allows efficiently to answer rectangular range queries. For a query rectangle, the nodes of the segment tree relevant for the query rectangle are determined, and a range query with this rectangle is carried out in the range trees of these nodes.
Proposition 4. It is possible to preprocess m rays within $P(m) = O(m^2\text{polylog } m)$ time into a data structure of size $S(m) = O(m^2\text{polylog } m)$ s.t. the set of all rays intersected by an arbitrary query rectangle (fixed orientation) can be found within $Q(m) = O(I + \log^4 m)$ time, I the number of intersections reported.
These arbitrary range queries solve a task analogous to determining the pixels covered by a polygon for the depth buffer visible surface algorithm. Comparison of depth in the depth buffer algorithm can be saved, if the priority of the primitives w.r.t. the viewpoint is unique. Then the primitives can be sorted according to depth. For our iso-oriented rectangles, sorting according to increasing z-coordinates has shown to be quite useful. The way we treat these rectangles is space-sweep. A plane perpendicular to the z-axis is moved through the scene, starting at the smallest z-coordinate of the scene. The intersection points of the rays with the sweeping plane are arranged into a semidynamical range tree R. If a rectangle occurs during the sweep, R allows quickly to find those intersection points in its interior, and hence its intersecting rays. The continuous sweep

is directly simulated by processing a sequence of discrete z-events of interest. Clearly, one z-event of interest is the occurence of a rectangle. In this case, the intersection points are reported, and all z-events initiated by the intersected lines are deleted from the list of z-events. Other events of interest are those where the intersection points with the sweeping plane change their x- or y-order. These events require an update of R. Note that these events happen just at the intersection points of the lines projected onto the x-z- and y-z-plane. They can be found by carrying out simultanously the plane sweep algorithm of Bentley and Ottmann [PS86,Mh85] in these two planes. The time necessary to calculate the proposed intersection points is $O((m+k)\log m)$, k the number of intersection points of the projected lines. The amortized time for updating R at these intersection points resp. deleting rays intersected by a rectangle is bounded by $O((m+k)\log^2 m)$. Finally, the time for calculating the intersection points with rectangles by range search is of order $O(n\log^2 m + I)$, I the total number of intersections that are found. The space is bounded by $O(m\log m + n)$. For more details, the reader may consult [M86a].
The crucial point of this algorithm is k, which can be of order $O(m^2)$. A way out is to decompose the given set of lines in, say, p disjoint sets of rays, each of about $\frac{m}{p}$ rays, and apply the algorithm on each of these sets in separate. Particularily suited are the subproblems obtained by cutting a twofold nested conjugation tree from above at some level i. At the leaves of the primary tree, the spacesweep algorithm on the corresponding sets of nodes is carried out. At the leaves of the secondary tree, analogous planesweep algorithm is applied which finds for the iso-oriented line segments induced by the y-z-projection of the rectangles, the intersected projected lines. For $i = \frac{\log m^2 - \log n}{1+\alpha}$, we obtain
Proposition 5. Given m rays and n iso-oriented rectangles in space, $m^{1-\alpha} < n < m^2$, the closest intersection point of every ray with the rectangles can be found in time $O(m^{\frac{2\alpha}{1+\alpha}} n^{\frac{1}{1+\alpha}} \text{polylog } m) = O(m^{0.820} n^{0.590} \text{polylog } m)$, $\alpha = \log\frac{1+\sqrt{5}}{2}$, using $O(m \text{ polylog } m + n)$ space.
In interesting scenes, the number of geometric primitives can be of the same order like the number of rays. Under this assumption, the proposition implies an average query time of $O(n^{0.41})$ per ray. There remains the question whether the first intersection point of a ray can be reported within a time per ray polylogarithmic in m und n, using $O((m+n)\text{polylog}(m+n))$ space at most. For special distributions of rays, like that discussed in sect. 3.2, time bounds of this quality are available.

3.4 A Comparison of Raytracing to Conventional Shading Procedures

An algorithm in wide use is the depth buffer algorithm. It can be seen as an ancestor of the primitive query approach for ray tracing sketched above. A drawback is the fact that shadows and most of the more sophisticated illumination effects like mirror, glass etc. cannot be handled. In the case where the 3d scene consists of a set of n triangular polygons $p_i \in P$, the asymptotic time bound for a typical depth buffer algorithm is $O(n + \sum_i \text{polypixel}(p_i) + L * m_x * m_y)$, 'polypixel($p$)' being the number of pixels covered by polygon p_i and L the number of light sources. If the triangles are replaced by more complex primitives, time analysis becomes more complicated.

A second approach to generate shaded pictures is based on a direct application of a visible surface algorithm. If P is the set of n given triangular polygons and A is the view point, e.g. the central point of a perspective projection, let Vissurf(P, A) denote the set of all parts of P visible from point A (details on the visible surface problem can be found in the appendix). Assume now a light source at point Q. The exact parts of polygons illuminated by the light source at point Q can be determined by first constructing a larger set of polygons P' by $P' := P \cup$ Vissurf(P,A). Then the visible surface reporting problem Vissurf(P',Q) is solved, reporting only those visible parts of polygons contained in Vissurf(P,A). The set of visible and illuminated parts may be P''. The polygons in P'' are projected onto the image plane and scan-converted into the raster image, evaluating the formula of illumination for the light source in question. If there are several light sources, this procedure is repeated as often as necessary and the respective intensities summed into the final shaded picture.
The total time bound in the case of L light sources is of $O((n + k)\log n + L * (n' + k')\log n + L * m_x * m_y)$. The parameters n' and k' are dependent on the scene. n' is the maximum number of polygons in any of the sets P', and k' is the maximum number of line intersections for the respective visible surface problem.
The table (see next page) compares the worst case time bounds of these algrithms with those of raytracing.

4 Final Remarks

Raytracing is a technique of image generation of increasing importance. This is demonstrated by the length of the list of references at the end of this paper. A large portion of these references is devoted to efficient solutions of the ray-primitive intersection test which was excluded here [Br86,Bv85,JB86,Kj83,Kj84,SA84,SB86,To85,Wk84]. Besides well-known approaches to speed up raytracing by restricting the search space, we have presented the new strategy of primitive object query. This approach to consider coherence seems to be of advantage compared with methods usually refered to as "beam-tracing" [Am84], [DK85],[HH84], which certainly may be useful for special types of scenes, but lacks the generality and flexibility of ray-tracing. Finally, new models of illumination were developed, further improving the optical quality of images. Closely related to ray tracing is distributed ray tracing [CP84],[LR86]. Distributed raytracing allows to generate depth of field and motion blur almost at the same cost of computation as usual anti-aliasing. A very different technique is the radiosity approach. The radiosity approach allows diffuse interreflection and refraction of light from primary light sources. Further aspects being excluded here are parallel algorithms and hardware to speed up raytracing. To our experience, raytracing is well suited for pipelined processing, concerning vectorial supercomputers [MC86] as well as systolic processing [M86d]. Other contributions to the discussion on parallel raytracing are [Br84,CW86,DS84].

	depth buffer strategy	visible surface strategy	ray tracing algorithm
shading (incl. highlights)	$\sum_i Polypixel(p_i)$ $+n + L \cdot m_x \cdot m_y$	$n + k)logn$ $+L \cdot m_x \cdot m_y$	$(n + k)logn$ $+(L + logn) \cdot m_x \cdot m_y$
plus correct generation of shadows		$(n + k) \cdot logn$ $+L((n' + k') \cdot logn'$ $+m_x \cdot m_y)$	$(L + 1) \cdot logn$ $\cdot((n + k') + m_x \cdot m_y)$
plus correct modelling of mirrors, glass			$P(n) + m_x \cdot m_y$ $\cdot(2^D - 1) \cdot RI(n)$

m_x:	number of pixels in x direction
m_y:	number of pixels in y direction
n:	total number of polygon edges, polygon are triangles
n':	scene dependent max number of polygons
k, k':	scene dependent maximum number of intersections in 2D space, cf. detail visible surface reporting
L:	number of point light sources
$Polypixel(p_i)$:	number of Pixels intersected or covered by polygon p_i in screen space
$P(n)$:	preprocessing time for general ray tracing
D:	depth of ray tracing
$RI(n)$:	time bound for ray query problem, general case

Table. Worst case time bounds for shaded image generation.

Appendix. The complexity of the visible surface reporting problem

Algorithms for hidden line and hidden surface elimination have a long tradition in computer graphics. They are nowaday a basic function of 3d graphics systems. But there is always a problem with the time of computation, especially when complex 3d scenes composed of thousands of polygons are to be processed without any hardware support. In the following, an $O((n+k)\log n)$ algorithm for visible surface reporting is presented. This algorithm was first published in [Sch81a,Sch81b]. At this time, it seemed to be the first algorithm, for which a worst case time bound of $O((n+k)\log n)$ and a space bound of $O(n + k)$ was effectively proved.
In this note, only exact visible surface algorithms are considered. Such an algorithm is called exact, if its output, i.e. the list of all visible parts of the given polygons, is mathematically correct, if the computer would perform exact real number computations. The 3d scene consists of a set of plane, simple and nonpenetrating polygons with a total of n vertices. It is mapped into the 2d screen space, while retaining all 3d coordinate information. The mapped edges of the polygons are called line segments.

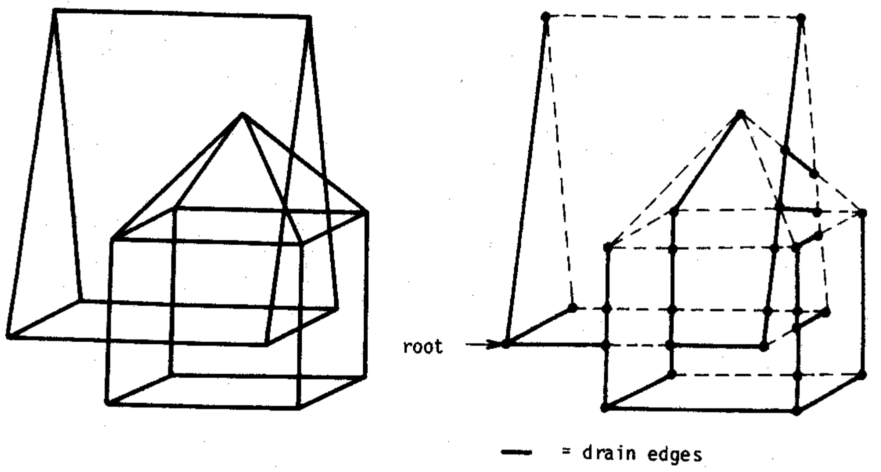

Figure A1. A simple 3d scene consisting of 14 polygons and a total of 50 polygon edges and, left of it, its corresponding connection graph. It may be mentioned that drain lines are necessary in order to connect all parts of the graph in such a way, that there is always a path to the root vertex in straight left direction.

Pass 1: This part of the algorithm concentrates on the mapped 2d scene and generates the so-called *connection graph.* The set of vertices of the connection graph is composed of all points in the plane that are start or end points of line segments, all intersection points of line segments, and all drain points. Two vertices are connected by an edge (of the connection graph), if the two vertices are neighbours on a specific line segment or if a vertex needs a drain connection with another line segment (drain edge). Figure A1 shows a simple 3d scene and its corresponding connection graph. The connection graph is connected, planar and geometric since vertices are discrete points of the plane.
The data structure to store the connection graph is relatively complex. Each vertex which is left endpoint of a line segment has a natural drain line or an artificial one. There is no need for an artificial drain line if the vertex has a line segment edge with an end point left of it. In general we choose that line segment with maximal slope as the natural drain line of a vertex. Note that vertical line segments can be avoided by a small rotation of the whole scene.
A suitable extension of the algorithm of Bentley and Ottmann [Mh84,PS85] for line segment intersection reporting can generate connection graphs in time of $O((n+k)\log n)$. Artificial drain lines and drain vertices are introduced by a simple inspection of the actual line segment ordering generated by this algorithm during the sweep. If m line sements intersect at a point V, the generation of the corresponding vertex data structure for the connection graph is possible in time of $O(m \log m)$ by straightforward methods even if the line segment ordering information of the plane sweep is not used. But this term is absorbed since m intersecting line segments produce $\frac{m(m-1)}{2}$ pairwise intersections and the plane sweep needs time of $O(k \log n)$ to process k intersections.
The space to store the connection graph is of $O(n+k)$, where k is the total number of

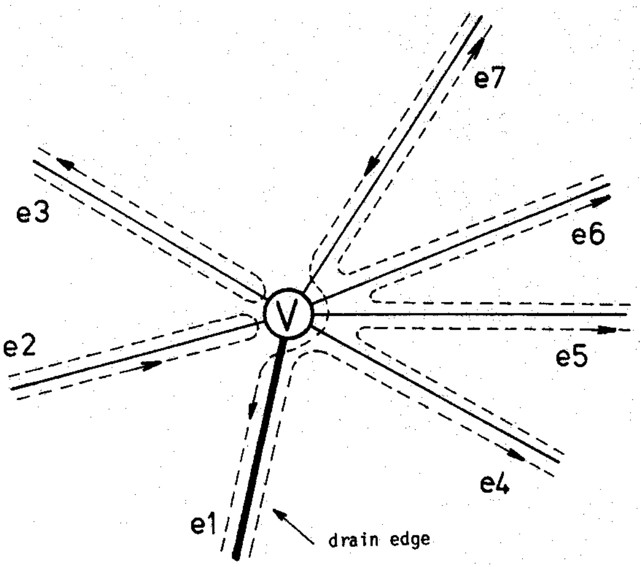

Figure A2. If vertex V is reached by e_2 or e_3 the traversal returns immediately, no updating of the actual inclusion status is necessary. If V is reached by its drain edge e_1, all edges to the right of V are traversed consecutively and recursively.

pairwise intersections of line segments. The additional artificial drain lines are without charge since their number is always less than n and since they are free of intersections.
Pass 2: Now the connection graph is traversed in order to assign to all edges the corresponding visibility status. Consider an edge e of the connection graph. The inclusion status of e is defined as
$I_-(e)$:= ordered set of polygons (visibility order) covering the region below of e,
$I_+(e)$:= ordered set of polygons covering the region above of e.
Note that this definition is unequivocal and sound only if the 3d scene is free of penetrations of polygons. We use balanced trees to store the polygons of an inclusion state in visibility order. Since there are at most n polygons, the space is of $O(n)$ and inclusion/deletion operations can be performed in $O(\log n)$.
There are several different strategies to travers the connection graph in such a way, that each single edge is processed exactly twice. One possible strategy starts at the root vertex and ends there. Each edge is traversed once from left to right and once backwards. If a vertex is reached (from left to right) by an edge, which is not the (natural or artificial) drain edge of that vertex, movement goes back to the left. If a vertex is reached from left by the drain edge, all edges to the right of that vertex are traversed one after the other in natural order, and then a return to the drain edge takes place. See figure A2 for an illustration of this strategy. In order to combine this traversal with a continuous update of the inclusion status for the actually reached edge, we need only to know, what edge belongs to what polygon.
In the case of a transfer to a new edge which is incident to the actual one, updating of the inclusion status is done as follows.

Consider a transfer from an edge e to an edge e' above e.

- If e is an upper border of polygon P then $I_-(e') := I_-(e) - \{P\}$
- If e is an lower border of polygon P, then $I_-(e') := I_-(e) \cup \{P\}$.
- If e' is an upper border of polygon P', then $I_+(e') := I_+(e) - \{P'\}$.
- If e' is an lower border of polgon P', then $I_+(e') := I_+(e) \cup \{P'\}$.

During the traversal the edges are labeled as early as possible with their visibility information. If the top polygons of $I_-(e)$ and $I_+(e)$ are identical, then e is covered by this polygon. If these top polygons are different, the edge is visible and is part of one of these polygons. These two polygons are also recorded. Thus the 'most visible' polygons in the neighbourhood of a visible edge are always known and their determination is possible in time $O(\log n)$.
Since the number of edges is at most of $O(n + k)$, the total time for the traversal (pass 2) is of $O((n + k) \log n)$ and does not increase the overall time complexity.
Pass 3: Now the visible parts of polygons are collected and reported. Due to pass 2, the question for visible edges can be answered immediately. Invisible edges are treated as if not being present. But the subgraph consisting of visible edges and their corresponding vertices may not be connected any more. Therefore we treat some of these invisible drain edges as pseudo-visible in order to connect everything again.
This graph - which is still geometric - partitions the relevant parts of the plane into simply connected regions. These regions represent all visible parts of the given plane surfaces. There is no problem to report all these parts and to specify the polygon of which the visible surface is a part. All this clearly can be done is time of $O(n + k)$.
As a result of this algorithm and of its time analysis we can state:
Theorem. The algorithm described above reports all visible parts of a given set of plane, simple and nonpenetrating polygons of a given 3D scene in time $O((n + k) \log n)$ and space $O(n + k)$, where n = total number of edges of polygons, and k = number of intersections of edges (=line segments) after transformation of the scene into 2d screen space.
It should be mentioned that an implementation of this algorithm is not at all a trivial task since various complicated data structures must be used in order to reach the asymptotic time bounds. For practical implementations an approach based on Franklins cell raster technique [Fr80, MSA85] should be used. It was shown that this technique allows solutions of the visible surface reporting problem for almost all practical scenes in expected linear time. For more details see [MSA 85].
It should be noted that hidden line elimination is always simpler than visible surface reporting. But up to now it is not possible to verify this by presenting different time bounds for both problems. Only recently, Devai [D86] showed that any visible line reporting problem can be solved in time $O(n^2)$, which is an improvement to the $O((n+k) \log n)$ bound in cases where $k = \Omega(n^2)$. In the meantime, McKenna [M86] proved this same time bound for the visible surface problem too. The drawback of these algorithms is that they require $O(n^2)$ space.

References

[Am84] Amanatides, J.: Ray-Tracing with Cones, Computer Graphics 18 (1984) 129-139

[Br86] Barr, A.H.: Ray tracing deformed surfaces, CG 20 (1986) 287-296

[Bv85] Bouville, C.: Bounding ellipsoids for ray-fractal intersection, CG 19 (1985) 45-52

[Bv85] Bouville, C., Brusq, R., Dubois, J.L., Marchal, I.: Generating high quality pictures by ray tracing. Computer Graphics Forum 4 (1985) 87-99

[Br84] Brusq, R.: Synthese d'image par lancer de rayon (ray tracing): la machine Cristal - resultats et perspectives, Deuxieme colloque image, Nice, 1984, 404-410

[Ch83] Chazelle, B.: Filtering Search: A new approach to query answering, 24 IEEE FOCS, 1983, 122-132

[CG83] Chazelle, B., Guibas, L.J., Lee, D.T.: The power of geometric duality, IEEE FOCS, 1983, 217

[CP84] Cook, R. L., Porter, T., Carpenter, L.: Distributed Ray Tracing, Computer Graphics 18 (1984) 137-144

[CW86] Cleary, J.G., Wyvill; B.M., Birtwistle, G.M., Vatti, R.: Multiprocessor ray tracing, Computer Graphics Forum 5 (1986) 3-12

[DK85] Dadoun, N., Kirkpatrick, D.G., Walsh, J.P.: The geometry of beam tracing. 1. Symposium on Computational Geometry, Baltimore, 1985, 55-61

[DE83] Day, W.H.E., Edelsbrunner, H.: Efficient algorithms for agglomerative hierarchical clustering methods, IFIP Graz, Report F121, July, 1983

[D86] Devai. F.: Quadratic bounds for hidden line elimination. Proc. 2^{nd} Annual ACM Symposium on Computational Geometry, Yorktown Heights, New York, June 86, pp. 269-275.

[DS84] Dippé, M., Swensen, J.: An adaptive subdivision algorithm and parallel architectures for realistic image synthesis, Computer Graphics 18 (1984) 149

[DE84] Dobkin, P.D., Edelsbrunner, H. Space searching for intersecting objects, IEEE FOCS, 1984, 387-391

[EW86] Edelsbrunner, H., Welzl, E.: Halfplanar range search in linear space and $O(n^{0.695})$ query time, Information Processing Letters 23, 1986, 289-293

[FS75] Floyd, R.W., Steinberg, L.: An Adaptive Algorithm for Spatial Grey Scale, SID'75, Int. Symp. Dig. tech. Papers, 1975,36

[Fr80] Franklin, W.R.: A linear time exact hidden surface algorithm, Computer Graphics 14 (1980) 117-123

[FI85] Fujimoto, A., Iwata, K.: Accelerated ray tracing. Proceedings of Computer Graphics Tokyo'85, T.L. Kunii, ed., Springer-Verlag, Tokyo, 1985

[Gv86] Gervautz, M.: Kugel- und Quaderumgebungen zur Optimierung des Ray-Tracing-Verfahrens für CSG-Bäume, Proc. AUSTROGRAPHICS'86, Oldenbourg-Verlag, München, Wien, 1986

[Gl84] Glassner, A.S.: Space subdivision for fast ray tracing, IEEE CG & Appl. 4, October 1984, 15-22

[HW86] Haussler, D., Welzl, E. Epsilon-nets and simplex range queries, 2. ACM Symp. on Comput. Geometry, 1986

[HH84] Heckbert, P.S., Hanrahan, P.: Beam Tracing Polygonal Objects. Computer Graphics 18 (1984) 119-127

[JB86] Joy, K.I., Bhetanabhotla, M.,N.: Ray tracing parametric surface patches utilizing numerical techniques and ray coherence, CG 20 (1986) 279-285

[KG79] Kay, D.S., Greenberg, D.: Transparency for computer synthesized images. CG 13 (1979) 158-164

[KK86] Kay, T.L., Kajiya, J.T.: Ray tracing complex scenes, CG 20 (1986) 269-278

[Kj84] Kajiya, J.T., von Herzen, B.P.: Ray tracing Volume Densities. Computer Graphics 18 (1984) 165-174

[Kj83] Kajiya, J.T.: New Techniques for Ray Tracing Procedurally Defined Objects. ACM Trans. on Graphics 2 (1983) 161-181

[LR86] Lee, M.E., Redner, R.A., Uselton, S.P.: Statistically optimized sampling for distributed ray tracing, CG 19 (1986) 61-68

[Me85] Meggido, N.: Partitioning with two lines in the plane, J. of Algorithms 6 (1985) 430-433

[Mh84] Mehlhorn, K.: Data structures and algorithms, III, Springer-Verlag, Berlin, 1984

[McK86] McKenna,M: Worst Case Optimal Hidden-Surface Removal, JHU/EECS-86/05, The John Hopkins University, Baltimore, June 1986

[MHS85] Müller, H., Schmitt, A., Abramowski, S.: Visible Surface Calculation for Complex Unstructured Scenes, Computing 35 (1985) 231-246

[M86a] Müller, H: Realistic image synthesis from complex scenes by raytracing (in German), Angewandte Informatik 4/86, 151-155

[M86b] Müller, H., Image generation by space sweep, Computer Graphics Forum 5 (1986) 189-195

[MC86] Müller, H., Christmann, A.: Realistic image synthesis on vector supercomputers (in German), it-Informationstechnik 5, 1986, 275-280

[M87] Müller, H.: Parallel computers for realistic image synthesis, IEEE Proceedings CompEuro'87, 1987

[PS85] Preparata, F., Shamos, M.I.: Computational geometry: an introduction, Springer-Verlag, New York, 1985

[Rh82] Roth, S.D.: Ray casting for modeling solids, Computer Graphics and Image Processing 18 (1982) 109-144

[SA84] Sederberg, T.W., Anderson, D.C: Ray tracing of Steiner patches, CG 18 (1984) 159-164

[SD85] Speer, L.R., DeRose, T.D., Barsky, B.A.: A theoretical and empirical analysis of coherent ray tracing, Springer-Verlag, Tokyo, 1985

[Sa84] Samet, H.: The quadtree and related hierarchical data structures, ACM Computing Surveys 16 (1984) 187

[Sch81a] Schmitt, A.: Time and space bounds for hidden line and visible surface algorithms, Proceedings Eurographics'81, North-Holland Publ. Comp., 1981, 43-56

[Sch81b] Schmitt, A.: Reporting geometric inclusions with an application to the hidden line problem, in: Proceedings 7th Conference on Graphtheoretic concepts in Computer Science (WG81), Hanser-Verlag, 1981, 135-143

[SB86] Sweeney, M.A.J., Bartels, R.H.: Ray tracing free-form B-Spline surfaces, IEEE CG&Appl., Feb. 1986, 41-49

[TK84] Tamminen, M., Karonen, O., Mäntylä, M.: Ray-casting and block model conversion using a spatial index, CAD 16(4) (1984) 203-208

[To85] Toth, D.L.: On Ray Tracing Parametric Surfaces. CG 19 (1985) 171-179

[WH84] Weghorst, H., Hooper, G., Greenberg, D.P.: Improved computational methods for raytracing, ACM Trans. on Graphics 3 (1984) 52-69

[Wh80] Whitted, T.: An Improved Illumination Model for Shaded Display. CACM 23 (1980) 343 - 349

[Wk84] Wijk, J.J.: Ray Tracing Objects Defined by Sweeping Planar Cubic Splines. ACM Transactions on Graphics 3 (1984) 223-237

Alfred Schmitt is a full professor at the Karlsruhe (technical) University since 1972 and is the leader of an academic teaching and research group working in the field of man-machine dialogue and computer graphics. His present research areas are fast hidden line and visible surface algorithms, ray tracing and its implementation and designing the man machine interface. He received a degree in mathematics and physics from the University of Saarbrücken and a doctoral degree from the Technical University of Hannover. He is a member of the ACM since 1965 and of Eurographics, GI, German Chapter ACM and EATCS.
Heinrich Müller is a lecturer at the Institut für Informatik I, Universität Karlsruhe, West Germany. He received diplomas in mathematics and computer science in 1978, and a Ph.D. in computer science in 1981 from the Universität Stuttgart. Research interests are the development of algorithms, data structures, and systems for geometric problems (in particular in computer graphics, CAD, and robotics) and parallelism. He is a member of the Gesellschaft für Informatik (GI).
Wolfgang Leister is a research associate at the Institut für Informatik I, Universität Karlsruhe, West Germany. interests include computer animation. He received a diploma in computer science from the Universität Karlsruhe in 1986. In his diploma thesis he worked on generating highly realistic images by ray tracing especially for application in computer animation. He is a member of the Gesellschaft für Informatik (GI).