The Ray-z- Buffer – An Approach for Ray Tracing Arbitrarily Large Scenes

Bernd Lamparter Heinrich Müller Jörg Winckler

March 7, 1996

Abstract

The ray buffer algorithm presented here is part of an alternative strategy of ray tracing called breadth first ray tracing. Breadth first ray tracing differs from usual pixel-wise rendering in tracing whole generations of rays together. This means that first all rays of view are treated, next the set of reflecting and refracting rays, then the reflecting and refracting rays caused by them, and so on. The calculation of intersections for a generation of rays uses the ray-z-buffer algorithm. The ray-z-buffer algorithm generalizes the well-known depth buffer algorithm. Like the depth buffer algorithm the ray-z-buffer algorithm allows to render scenes with an unbounded number of primitives. An implementation based on nested adaptive quadtrees is presented. Analysis and measuring show the rendering time increasing sublinearly in the resolution of the image, and increasing linearly in the number of primitives. The main memory required only depends on the resolution of the image, in about linear manner.

1 Breadth-first ray tracing

The main task of realistic image synthesis by ray tracing is the detection of the intersection point of a ray with a given spatial scene, closest to the origin of the ray. A spatial scene usually consists of a number of primitives, like triangles, spheres, and patches.

Many algorithms were developed for speeding up this calculation [5]. Nearly all of them first preprocess the scene into a data structure before rendering. The data structure reduces the number of primitives to be tested against a ray. A well-known approach uses regular grids. The min-max bounding box of the scene is subdivided into congruent cells. For each cell a list is held collecting all its intersecting primitives. Once these lists are established in the preprocessing phase, the algorithm detects for each ray those grid cells the ray is passing through. The ray is intersected with the primitives referred to in the traversed cells until the closest intersection point is found, or the bounding box is left. Thus the number of intersection tests is reduced on the number of primitives along the ray. The rays are treated pixel-wise, including all reflecting and refracting rays induced by the pixel, according to a *depth-first strategy*. The storage requirements for rays are neglectable, since only one ray is treated once a time.

A disadvantage of depth-first ray tracing is the dependency of the memory required by the data structure on the number of primitives in the scene. This dependency is one difference between todays ray tracing and the z-buffer algorithm. The main data structure of the z-buffer algorithm is the z-buffer holding for each pixel the distance of the primitive closest to the observer. The primitives are processed sequentially. Coming up with a new primitive, its distance from the observer at each relevant pixel is calculated and compared against the value in the z-buffer. If the primitive is closer, the z-buffer is updated with the new distance and the primitive is registered visible at this pixel.

Evidently, the memory requirements of the z-buffer algorithm depend on the size of the z-buffer which in turn depends on the resolution of the image. The scene may be stored on secondary memory. The possibility to handle a practically unlimited number of primitives within an predictable bound of memory makes the z-buffer algorithm particularly suitable for implementation in hardware.

In this paper, this property of the z-buffer algorithm is transferred to ray tracing. The main idea is to replace depth-first ray tracing by *breadth-first ray tracing*. Breadth-first ray tracing opens by first treating all rays of view. Due to their special orientation, they can be handled by the z-buffer algorithm. Once their first intersection points are determined, calculation continues with the rays of shadow starting at the intersection points and ending at the point light sources in the scene. The rays of shadows may be treated by the z-buffer algorithm too. The result of the algorithm up to now is the portion of color contributed by the the first generation of rays, i.e. the rays of view, to the image. Beyond the color the z-buffer algorithm is required to calculate the rays of reflection and refraction for the first generation of rays. They make up the second generation of rays. For the second generation, again the first intersections have to be found out. Their calculation (and that for all following generations) is done by the *ray-z-buffer algorithm*.

In section 2, the ray-z-buffer approach is outlined in a general manner. In section 3, it is specialized to scenes of rectangles in parallel to the coordinate axes. This is an important special case since other primitives may be easily approximated by them, for example using their min-max bounding boxes or better approximating rectangular polytopes. Rays intersecting the bounding box are then additionally to be tested against the enveloped primitive. In the special case, the ray-z-buffer is implemented by nested adaptive quadtrees. The algorithms for constructing and searching them are described in section 4. In section 5 an implementation of the the ray-z-buffer algorithm and its performance are discussed.

2 The ray-z-buffer algorithm

The ray-z-buffer algorithm solves the following problem:

Ray-Scene-Intersection.

Input. A set R of rays in 3-space.

Output. Those rays in R intersected by any primitive in the scene, as well as the first point of intersection.

The algorithm is split in two phases, the preprocessing and the query phase. In the *preprocessing phase*, the set of rays R is preprocessed into a data structure called *ray-z-buffer*. The ray-z-buffer generalizes the z-buffer. However, it is more complex since it has to take into account the origin and the direction of the rays.

A ray not orthogonal to the z-axis may be described by two equations,

$$x = a \cdot z + b,$$

$$y = c \cdot z + d,$$

and two z-values,

$$z_o, \ z_e$$

for the origin respectively the current end point of the ray. The ray-z-buffer contains tuples $(a, b, c, d, z_o, z_e, prim)$ describing the set of rays R currently stored and information $prim$ about the currently visible primitive

In the *query phase*, the ray-z-buffer is used to answer queries for those rays intersecting a given primitive. The task of this phase is similar to the main phase of the z-buffer algorithm in which the pixels covered by a projected polygon are found out.

Suppose the primitives of the scene are given implicitely by

$$F_0(\mathbf{c}_0, x, y, z) = 0,$$

$$F_i(\mathbf{c}_i, x, y, z) \geq 0,\ i = 1, \ldots, p$$

The first equation defines a surface in 3-space from which a part is cut off by the additional inequalities. The $\mathbf{c}_i \in \mathbb{R}^d$, $d \geq 1$, are parameters controlling the shape of the surface. For example, rectangles perpendicular to the z-axis, in the following called *iso-oriented rectangles*, can be described by

$$z = c_0,$$

$$c_{11} \leq x \leq c_{12},$$

$$c_{21} \leq y \leq c_{22},\ c_0, c_{ij} \in \mathbb{R},\ i, j = 1 \ldots, 4.$$

A ray intersects a primitive if a solution z of

$$F_0(\mathbf{c}_0, a \cdot z + b, c \cdot z + d, z) = 0$$

exits which satisfies the constraints

$$F_i(\mathbf{c}_i, a \cdot z + b, c \cdot z + d, z) \geq 0,\ i = 1, \ldots, p.$$

Among the solutions, only that one between z_o and z_e closest to z_o is of interest. It defines the new z_e of the ray (a, b, c, d, z_0, z_e).

If $z = z(\mathbf{c}_0, a, b, c, d)$ can be isolated in the equation it can be put into the constraints leading to z-free inequalities

$$\overline{F_i}(\mathbf{c}_i, \mathbf{c}_0, a, b, c, d) \geq 0,\ i = 1, \ldots, p,$$

for the rays intersecting the primitive. For iso-oriented rectangles, the inequalities are

$$c_{11} \leq a \cdot c_0 + b \leq c_{12},$$

$$c_{21} \leq c \cdot c_0 + d \leq c_{22},$$

$$c_0, c_{ij} \in I\!R,\ i,j = 1\ldots,4.$$

The ray-z-buffer has to be organized into a data structure so that the rays $(a, b, c, d, z_0, z_e, prim)$ satisfying the inequalities can be found quickly for arbitrary parameters $\mathbf{c}_i$.

The difference in computational complexity between the classical z-buffer algorithm and the ray-z-buffer can now be quantified. While the search space in the first case is two-dimensional (the indices of the pixels covered by the primitive have to be found) here it is the four-dimensional a-b-c-d-space. The space and time required depends on the shape of the query range given by the constraints. In [11], [12] it was shown that for constraints given by polynomials, n rays can be preprocessed into a data structure of size $O(n)$ so that a query can answered in time $O(n^\beta)$, with a constant $0 < \beta < 1$. If the query range consists of only one bilinear inequality, $\beta = (\log 15)/4$ was obtained which is very close to 1. Better query times are achieved by investing more than linaer storage. However, the amount of storage required for a significant speed up in query time is considerable [2].

These worst-case complexity results are not very promising. However, like for the usual ray approaches of ray tracing, data structures with a good practical behavior, though bad in the worst case, may still exist. In the remainder the special case of iso-oriented rectangles is investigated in details.

3 The ray-z-buffer for iso-oriented rectangles

Let us now re-consider the inequalities of intersection for iso-oriented rectangles from the previous section. Somewhat rearranged they are

$$-a \cdot c_0 + c_{11} \leq b \leq -a \cdot c_0 + c_{12},$$

$$-c \cdot c_0 + c_{21} \leq d \leq -c \cdot c_0 + c_{22},$$

$$c_0, c_{ij} \in I\!R,\ i,j = 1 \ldots, 4.$$

An important observation is that the first line only concerns a and b, and the second line only c and d. This means that the four dimensional query problem is reduced to two two-dimensional coupled query-problems, one for the a-b-projection and one for the c-d-projection of the given rays. For both problems, the query ranges are stripes. The projected rays are points. Those of them falling in the stripe have to be found.

In this situation, the solution of the planar stripe query problem by an *hierarchical subdivision* shows favorable. The given set of points is subdivided in a constant number of disjoint parts which are in turn divided recursively. A typical approach is rectangular subdivision. Figure 2 shows a recursive rectangular subdivision.

The coupling between both projections is now carried out as follows. To each member C in the hierarchy of cells in the a-b-projection, a secondary hierarchical subdivision in the c-d-projection is assigned. The set of points over which the secondary subdivision is constructed is the c-d-projection of those rays whose a-b-projections fall in C.

Hence the preprocessing phase of the ray-z-buffer algorithm works as follows.

Algorithm Ray-Z-Buffer, Prepocessing Phase.

Input. A set of rays $(a_i, b_i, c_i, d_i, z_{o,i}, z_{e,i}, prim)$, $i = 1, ..., m$, $z_{e,i}$ and $prim$ initialized on "background".

Output. A tree structure `RZB` of type `cell` with

```
type CELL = record
              RAYS: ^RAY;
              SUBCELL: ARRAY[1..k] OF ^CELL;
              SECONDARY:^CELL;
            end;
type RAY =  record
              A,B,C,D,Z_O,Z_E: real
              NEXT: ^RAY
            end;
```

Statements.

```
subalgorithm PARTITION(
   RAYS: ^RAY; var PART:  array[1:k] of ^RAY);
      (* partitions the set RAYS into
      parts PART[1], ..., PART[k] *)

subalgorithm SUBDIVIDE(
   PROJECTION: integer, var: CURRENT_CELL: CELL);
begin
   if CURRENT_CELL.RAYS contains more than one ray
   then
     if PROJECTION = 1 then
     begin
       RAYS: = the a-b-projection of the list
                  of rays CURRENT_CELL.RAYS;
       PARTITION(RAYS,PART);
       for i:=1 to PART[k] do
       begin
         new(CURRENT_CELL.SUBCELL[i]);
         CURRENT_CELL.SUBCELL[i]^.RAYS := PART[i];
         SUBDIVIDE(1,CURRENT_CELL.SUBCELL[i]);
       end;
```

```
         SUBDIVIDE(2,CURRENT_CELL);
        end else
        begin  (* PROJECTION =2 *)
          RAYS: = the c-d-projection of the list
                   of rays CURRENT_CELL.RAYS;
          PARTITION(RAYS,PART);
          for i:=1 to k do
          begin
            new(CURRENT_CELL.SUBCELL[i]);
            CURRENT_CELL.SUBCELL[i]^.RAYS := PART[i];
            SUBDIVIDE(2,CURRENT_CELL.SUBCELL[i]^);
          end
        end
  end;

begin  (* main algorithm *)
   subdivide(1,RZB);
end.
```

A query is answered by first searching for the maximal cells of the a-b-projection contained in the a-b-stripe. For each cell found, the secondary data structure is searched for cells with the same property, but now for the c-d-projection and the c-d-stripe. The lists of rays belonging to these cells are concatenated and reported as solution. Evidently, they satisfy the inequalities in both projections simultanuously, thus intersecting the query rectangle.

Algorithm Ray-Z-Buffer, Query Phase.

Input.	The ray-z-buffer `RZB` calculated in the preprocessing phase, a set Q of query rectangles.
Output.	The ray-z-buffer updated by the rectangles in Q .
Statements.	

```
subalgorithm STRIPE_QUERY(
  CURRENT_STRIPE: STRIPE; CURRENT_CELL: CELL,
  var CELLS: set of [0..k]);
 (* reports CELLS={0} if CURRENT_CELL lies
    completely in CURRENT_STRIPE or intersecting
    CURRENT_STRIPE and containing one point at most.
    Otherwise, the set of indices  of subcells
    intersected by CURRENT_STRIPE is returned in cells *)

subalgorithm RECTANGLE_QUERY(
   PROJECTION: integer; ABSTRIPE, CDSTRIPE: STRIPE;
   CURRENT_CELL: CELL);
   begin
     if PROJECTION = 1 then
        CURRENT_STRIPE := ABSTRIPE
     else
        CURRENT_STRIPE := CDSTRIPE;
     STRIPE_QUERY(CURRENT_STRIPE,CURRENT_CELL,CELLS)
     if CELLS = {0} and
         ((|CURRENT_CELL.RAYS| <= 1 and the
    probably existing point is in ABSTRIPE)
           or |CURRENT_CELL.RAY|  <> 1)
     then
     begin
        if PROJECTION = 1 then
          RECTANGLE_QUERY(
   2,CDSTRIPE,CURRENT_CELL^.SECONDARY)
        else
          update Z_E and PRIM of the rays in
          CURRENT_CELL.RAYS
     end else
     if |CELLS| > 0 and CELLS <> {0}
```

```
        then
           for i in CELLS do
             RECTANGLE_QUERY(PROJECTION,ABSTRIPE,
                CDSTRIPE,CURRENT_CELL^SUBCELL[i|);
    end;

begin  (* main algorithm *)
   for q in Q do
    begin
       ABSTRIPE := a-b-stripe for rectangle q;
       CDSTRIPE := c-d-stripe for rectangle q;
       RECTANGLE_QUERY(1, ABSTRIPE, CDSTRIPE, RZB);
    end
end.
```

4 The Adaptive Quadtree

There are several strategies of partioning which may be applied in the algorithm of the previous section. Examples are the *polygon trees* [10], the *conjugation trees* or *ham sandwich tree* [4], [3] the *ϵ-nets* [6], and the *partition trees* [9]. All of them were developed to minimize the aymptotic worst case complexity in time and space. For example, the ham sandwich tree requires $O(n)$ space, $O(n \log n)$ preprocessing time, and $O(n^\beta)$ query time for n points, $\beta = \log 0.695$. The query times of the later structures are better, the preprocessing times however seem considerably worse up to now.

The structures mentioned work by hierarchical subdivision. In each step, they partition cells in a balanced manner by lines or line segments. This requires are considerable amount of arithmetic operations, in particular multiplications and divisions. Further, since the line of division can be arbitraray, some amount of memory is necessary to store the subdivision. For these reasons we do not use these structures in the following. Instead, the pattern of subdivision already mentioned in figure **??** is used which is based on lines of subdivision in parallel to the axes of coordinates. The resulting structure is the well known *quad tree* citesamet84.

4.1 Constructing Adaptive Quadtrees

A quadtree carries out a subdvision of a rectangular region in the plane into rectangles by recursively dividing each region by a pair of horizontal and vertical line segments into four subregions called *quads*. The result is a hierachical partition of the plane. Coming up with a stripe, the query for the points within the stripe can be solved by descending down the tree along the quads intersected by the stripe. The number of inspected quads is small if the quadtree is balanced, or, in other words, if the four quads of a subdivision contain nearly the same number of points.

Different from the data structures mentioned in the previous section, a partition of this sort is not always possible. For example, it is not if all the points lie on a straight line. However, the following algorithm guarantees a partition of a set of n points by a pair of a horizontal and a vertical line so that

- every quad contains $\frac{n}{3}$ points at most,
- the maximum of the number of points in each of the four subareas is a local minimum.

The latter implies that a vertical respectively horizontal translation of the subdividing lines to the next possible point does not diminish the maximum.

Algorithm Adaptive Partition

Input.	A set of points (a_i, b_i), $i = 1, ..., n$, in the plane, with different a- resp. b-coordinates, and sorted in increasing order with respect to a and b.
Output.	A point (a_0, b_0) of subdivision dividing the set of points into four quads, $A = \{a_i \geq a_0, b_i \leq b_0\}$, $B = \{a_i \geq a_0, b_i \geq b_0\}$, $C = \{a_i \leq a_0, b_i \leq b_0\}$, $D = \{a_i \leq a_0, b_i \geq b_0\}$.

Statements.

```
begin
  sort the points in ascending order with
  respect to the a- respectively the b-coordinate;

  i:=n/3; j:=0;
  (* i,j is the present vertical respectively
         horizontal division.*)

  calculate the cardinalities |A|,|B|,|C|,|D|
  of the present quads;

  translate the horizontal line in b-direction
  until ||B|-|D|| <=1 and update j accordingly;

  opt:=max(|A|,|B|,|C|,|D|);

  repeat
    i:=i+1; (* one step to the right *)
    update the cardinalities of A,B,C,D;
```

```
    translate the horizontal line in
    direction of the greatest quad until
    |max(|A|,|B|)-max(|C|,|D|)|  <= 1
    and update j accordingly;

    if max(|A|,|B|,|C|,|D|)< opt
    then  opt:=max(|A|,|B|,|C|,|D|)
  until i >= 2n/3;
  return a_0 := a_i, b_0 := b_j
end.
```

A quadtree constructed by first sorting the points and then recursively calling this algorithm is called *adaptive quadtree* (also *pseudo quadtree* in [7]). The recursion ends if the number of points in a rectangle is at most 1.

4.2 Stripe queries in adaptive quadtrees

A stripe query asks for all points lying in a stripe defined by two parallel lines ($g_1 : b = sa + t_1$ and $g_2 : b = sa + t_2$). Using a quadtree it is answered by visisting recursively the maximal quads intersecting the stripe, as follows (figure 1). The slope of the stripe is supposed non-negative, the other cases work analogously.

Algorithm Stripe Query.

Input.	A quad with boundaries $a_{min}, b_{min}, a_{max}, b_{max}$, a point of partition (a_0, b_0), if any.
Output.	Information about how the quad is intersected by the stripe.
Statements.	

```
begin
  if the quad has at most one point
```

```
      and the point lies in the stripe
  then report ''quad within the stripe'';
  else
  begin
    if g_1(a_0)<b_0 then
      (* the stripe lies below
 the mid point (a_0, b_0) *)
       report ''quad D is intersected'';

    if  g_1(a_0) > b_{min} then
       report ''quad C is intersected'';

      if g_1(a_{max}) > b_0 then
      report ''quad  B is intersected''
    else
    if g_2(a_0) > b_0 then
    (* stripe lies above mid point (a_0, b_0) *)
    ...
    (* This case is handled like case 2. *)
    ...
    else
    (* the stripe encloses mid point  (a_0, b_0) *)
    if  g_1(a_{min}) >= b_{max}
and g_2(a_{max}) >= b_{min}
    then
    (* the whole quad lies in the stripe *)
    report ''quad within the stripe''
  else
    report ''quads A, B, C, and D are intersected''.
end.
```

An implementation of this algorithm should minimize the arithmetic operations, especially the number of multiplications. Multiplications can be saved by the observation that the boundary values $g_i(a_j)$, $i \in \{1,2\}$, $j \in \{\min, \max\}$, usually were already calculated on a previous level of recursion in the form $g_i(a_0)$. This reduces the necessary calculations on a comparison and a passing of value to a next level of recusion.

Altogether, the boundary lines of the stripe are intersected with each vertical partitioning line of the quads crossed by the stripe. The number of intersections, increased by two (for the border), is the number of multiplications required with this implementation (see figure 2).

5 The Implementation

The ray buffer algorithm was implemented as an independent process in a multi-process image generation environment. The following processes run in this environment:

- a **parser** which translates the textual input files into binary data, and which computes the initial rays of view.
- a **shading process** which computes the pixel colors and the next generation rays from the intersection results of the current rays and the shading parameters at the intersection point. This process also handles the image matrix.
- one or several **ray-z-buffer processes** which answer the queries for intersections between geometric primitives and rays according to the ray-z-buffer algorithm.

Figure 4 shows the communication channels between these processes. The parser reads an textual input stream and converts it into a binary representation. The input stream contains information about

- projection, image size, number of iterations of raytracing, etc.

- shading parameters defining so-called materials, light sources, etc.

- geometric primitives, query supporting primitives like polyhedrons with iso-oriented faces, etc.

The parser calculates the rays of view out of the projection and sends the rays to a ray-z-buffer process. The ray-z-buffer process constructs the nested quadtree. Then it reads the primitives of the scene and carries out the respective queries. The answer consisting of the intersected rays with information about the intersection point, the normal vector, interior coordinates, and the material is sent to the shading process. The shading process first allocates memory for the image matrix. Then it reads the answers of the ray buffer processes, determines the appropriate color value by evaluating the formula of illumination with the actual parameters and normal vectors, and inserts the value into the image matrix. Finally, the shading process calculates the new rays of shadow, reflection, and refraction, and sends them to a ray buffer process. These new generation rays are handled like the preceeding ones.

The actual implementation runs in an UNIX environment. The inter-process-communication is by named pipes. Alternatively, the remote procedure call mechanism of NCS (network computing system, [1]) which allows the distribution of processes on several machines in a network is applied.

5.1 Implementation of the Ray-Z-Buffer

The ray buffer algorithm was implemented in the programming language $C{+}{+}$ [8]. The basic data structure, the nested adaptive quadtree for rays, is built up by three classes: the *rays*, the *leaf nodes* and the *inner nodes*. The three classes have the following representation (storage requirements in brackets).

- **Rays.** A ray consists of its origin (3 reals = 24 bytes), its end point (3 reals = 24 bytes), the normal vector (3 reals = 24 bytes) at the intersection point of the intersected object (this is required for calculating the rays of reflection and refraction, local coordinates (2 shortreals = 8 bytes) of the object (this is required e.g. for texture mapping, and a classification???? (4 bytes). In total, a ray rquires 84 bytes. The rays are stored in an array. Rays belonging to the same leaf are stored consecutively in the array.

- **Tree nodes.** A tree node has a pointers to its predecessor and to its successor(8 bytes), and a bounding box (2 x 3 shortreals = 24 bytes). The bounding box describes the size of the quad. Totally, a tree node requires 32 bytes.

- **Leaf nodes.** A leaf node represents an undivided quad. Besides the data of a normal tree node it contains a pointer to the array of rays (4 bytes) and the number of rays (1 integer = 2 bytes). The number says how many rays are corresponding to the node. In total, a leaf node requires 38 bytes.

- **Inner nodes.** An inner node of a primary or secondary tree requires the same information like a leaf node, as well as a pointer to a secondary tree (4 bytes) and the crossing point (a_0, b_0) of the dividing perpendicular lines (2 shortreals = 8 Bytes). In total, an inner node requires 44 bytes.

Let m be the number of leaf nodes. m may be different from the number n of rays if the subdivision is not carried out down to one-element quads. The reason for allowing bigger quads is memory reduction. If k is the minimal size of a quad, then $m = n/k$ is the number of points which can lie in a leaf. The degree of branching of an inner node of an adaptive quadtree is 3 or 4. Hence the number of inner nodes lies between $\frac{4}{3}m$ and $\frac{3}{2}m$. This results in

- **storage requirements for ray data:** $60n$ Bytes for n rays,

- **storage for a primary tree:** between $\frac{4}{3} \cdot 38m = 52m$ bytes and $\frac{3}{2} \cdot 38 = 57m$ bytes for inner nodes. The total storage for leaf nodes is $38m$ bytes.

Hence the storage required for the primary tree lies between $90m$ and $95m$ bytes.

The storage requirements of the secondary trees can be estimated using the height of the primary tree. The height is bounded by $\log_3 m$ since the maximal number of points in a node reduce by at least one third from level to level. In the worst case, n points are distributed on each level of a primary tree. The memory required for each level is about the same like for the primary tree. In fact, it is slightly more since the resulting forest of secondary trees has more inner nodes than a single tree. The number of inner nodes increases by the number of secondary trees, i.e. by the size of the primary tree. Altogehter, the primary and secondary trees reuqire between $90 \cdot m \cdot (2 + \log_3 m)$ and $95 \cdot m \cdot (2 + \log_3 m)$ bytes of memory.

It should be noted that the adaptive quadtree is needed three times, for rectangles perpendicular to the x-, y-, and z-axis, respectively.

The strategy of attaching a secondary tree to all nodes of the primary tree is space-consuming. The implementation reduces the space requirements by the following modifications,

- only every k-th level (for example $k = 2$ or 3) has a secondary tree.

- "construction by demand". The secondary tree is constructed only if a quad with a lot of points is intersected by a stripe. This can be combined with a mechanism for deleting secondary trees when they become obsolete. A tree becomes obsolete if it will not be required for future queries.

The latter modification supposes a certain locality of queries. This can be assumed if the queries are arranged according to the neighbourhood relationship of the primitives of the scene.

The reduction of storage must be payed with more time consuming stripe queries.

The time complexity of a stripe query in a simple quadtree can be estimated by an expression of the form $O(f(n) + I)$. I is the number of points lying in the stripe, $f(n)$ is the overhead for tree traversal. The worst case overhead is $f(n) = n$, This overhead can be enforced by constructing for a given line a quadtree so that all its dividing lines are intersected by the given line (cf. figure 3).

The construction of the worst case shows a strong displacement of the dividing lines in the different quads. The resulting partition appears to be very irregular. It may be of interest to inspect the other extreme. In this case the quadtree induces a regular grid (figure 3). The overhead is proportional to the number of intersected quads. This number in turn is proportional to the number of intersected dividing lines (horizontal and vertical). If the tree is not balanced (the height of the tree is $\log_3 n$) this number is equal to $O(2^{\log_3 n}) = O(n^\beta)$, $\beta = log_3 2 = 0.63$. If the tree is balanced (height $\log_4 n$) the overhead is $O(2^{log_4 n}) = O(n^{0.5})$. If the tree is balanced but the dividing lines are irregular then the overhead $f(n)$ satisfies the recurrence

$$f(n) = O(1) + 3 \cdot f(\frac{n}{4}).$$

The recurrence is derived from the fact that a straight line cannot intersect more than three of the four quads. An intersections required constant time ($O(1)$). Solving the recurrence results in an overhead of $O(n^\beta)$, $\beta = \log_4 3 = 0.790$, for this case.

For nested quadtrees, the time complexity increases maximally by a logarithmic factor. In favorable situations the asymptotic behavior even will be the same (with a larger constant). This is demonstrated in the following only for the case of balanced trees. The other cases are analogous. The recurrence from above has to be changed to

$$f(n) = O(n^\beta) + 3 \cdot f(\frac{n}{4}).$$

The first term on the right side estimates the costs for searching in a secondary tree. Under the assumption that the seconday trees are of the same type, β is equal to $\log_4 3$. The solution is of the reccurence is an overhead of of $O(n^\beta \log n)$, $\beta = \log_4 3$.

Summarizing, it can be expected that the use of nested quadtrees reduces the time requirements of a query considerably, except in some rare unfavorable cases.

5.2 Measurements

Figure 5 shows the distribution of the rays reflection and refraction corresponding to figure 6.

The calculation was carried with a resolution of 128×128 pixels. Table 1 contains the exact values of this calculation. The behavior concerning time and memory consumption is shown by two test series. For the examination of the exponent of query time which must lie between 0.5 and 1, as pointed out in section 5.1, an image was computed with different resolutions. The resolution varied from $60 * 60$ pixels to $256 * 256$ pixels. The image with the resolution of $256 * 256$ had been calculated in several passes because the number of rays was too high for one pass. The results are summarized in table 5.2. The values of tables 5.2 and 5.2 are presented graphically in figure 5.2. It is a double logarithmic representation. All points lie on a straight line with slope 0.54. This means that with the time increases sublinear ($O(n^{0.54})$) in the size of the image. This confirms the analysis of section 5.1.

In the second test series, image of the same fixed size were

picture : sphere of crystal
scene : 2 triangles, 2 spheres
resolution: 128×128
statistics of rays.

	viewing	shadow	reflection refraction
generation 1	0	6532	8303
generation 2	0	973	1563

statistics of time:

type	number	reading	preprocessing	intersection tests	writing

generation 1: successful intersection tests: 3431

rays of shadow	5738	-	10.5	-	-
rays of reflection	7509	-	3×15	-	-
$\sum$ 13247	38	55.5	95	29	

generation 1: successful intersection tests: 134

rays of shadow	794	-	1.3	-	-
rays of reflection	794	-	3×1.3	-	-
$\sum$ 1588	4.6	4.2	8.8	3.5	

generation 2: successful intersection tests: 925

rays of shadow	937	-	1.7	-	-
rays of reflection	1563	-	3×2.6	-	-
$\sum$	2536	7.3	9.5	19	5.5

Table 1: The statistics of rays for the sphere of crystal.

Gen.	256	x	256	128	x	128
	r.	i.	t.	r.	i.	t.
0:	65536	1.36	32	16384	1.36	11
1:	46622	0.10	435	11627	0.09	196
2:	2967	0.62	91	746	0.65	44
3:	1382	0.90	62	356	0.90	28
4:	901	0.99	52	234	0.94	24
5:	616	1.15	43	160	1.16	17

Table 2: Statistics for increasing resolutions on a Sun 3/60, 8 MByte – I

Gen.	100	x	100	80	x	80	60	x	60
	r.	i.	t.	r.	i.	t.	r.	i.	t.
0:	10000	1.26	10	6400	1.35	9	3600	1.35	8
1:	7084	0.09	151	4521	0.09	119	2540	0.10	86
2:	444	0.64	33	285	0.68	24	166	0.60	20
3:	211	0.90	21	128	1.01	13	73	0.98	11
4:	138	0.94	17	95	0.91	14	53	1.07	9
5:	89	1.26	12	62	1.20	10	33	1.36	6

Table 3: Statistics for increasing resolutions – II

Gen.	256	x	256
	r.	i.	t.
0:	29899	1.10	11
	29899	1.71	15
	5738	0.56	6
1:	17809	0.03	131
	17809	0.19	203
	11004	0.04	101
7:	17804	0.71	253
	1881	0.41	33

Table 4: The passes for resolution 256 x 256

calculated from scenes with increasing numbers of primitives. The plot in figure 8 shows a linear baehavior of the execution time to the number of primitives, as expected.

The relative behavior of the ray-z-buffer looks quite promising. The absolute times are poor when compared with efficient implementations of usual depth first ray tracing strategies. Partly, inefficiencies of the implementation are responsible for that. Examples are superfluous procedure calls in the most busy loops due to the strict application of the class concept of C++. On the other hand there are still too many multiplications for the stripe queries. Probably, regular quadtrees dividing strictly in the middle may be better, although balancing is worse.

The memory requirements seem high. This point however is not very severe due to the advances in RAM technolgy and predictable bounds of memory requirements only dependent on the resolution of the image.

References

[1] Apollo Computer Inc., Chelmsford, MA 01824. *Network Computing System (NCS) Reference*, 1987.

[2] B. Chazelle. Fast searching in a real algebraic manifold with application to geometric complexity. In *Proceedings CAAP'85*, pages 145–156, 1985.

[3] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Spriner-Verlag, 1987.

[4] H. Edelsbrunner and E. Welzl. Halfplanar range search in linear space and $o(n^{0.695})$ query time. In *Information Processing Letters 23*, pages 289–293, 1986.

[5] A. Glassner, editor. *An Introduction to Ray Tracing.* Academic Press, New York, 1989.

[6] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. In *Proceedings 2th ACM Symp. on Computational Geometry*, 1986.

[7] M. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, 1983.

[8] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[9] E. Welzl. Partition trees for triangle counting and other range search problems. In *Proceedings 4th ACM Symp. on Computational Geometry*, pages 23–33, 1988.

[10] D. Willard. Polygon retrieval. In *SIAM J. on Comput. 11*, pages 149–165, 1982.

[11] A. Yao. Space-time-tradeoff for answering range queries. In *ACM STOC 1982*, pages 128–135, 1982.

[12] A. Yao and F. Yao. A general approach to d-dimensional geometric queries. In *Proceedings ACM STOC 1985*, pages 163–168, 1985.

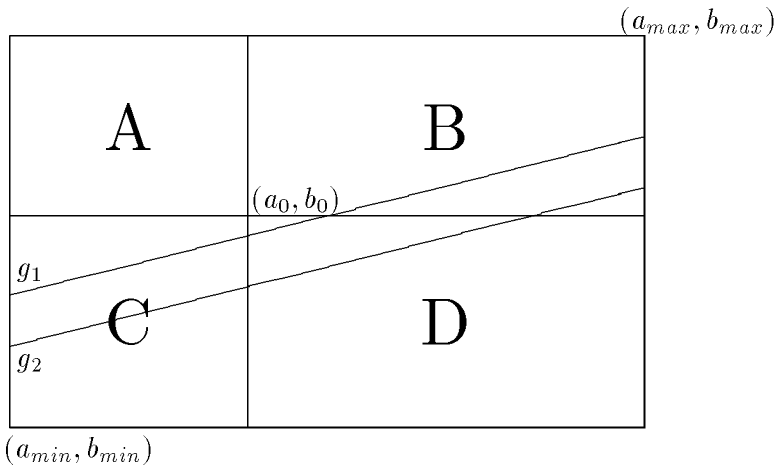

Figure 1: Stripe query in a rectangular subdvision.

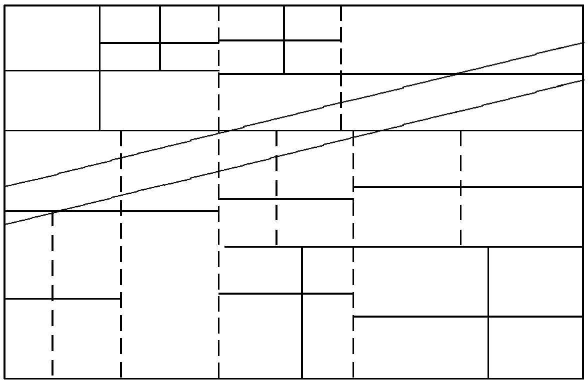

Figure 2: Stripe query in a quadtree

Figure 3: Quadtree structures

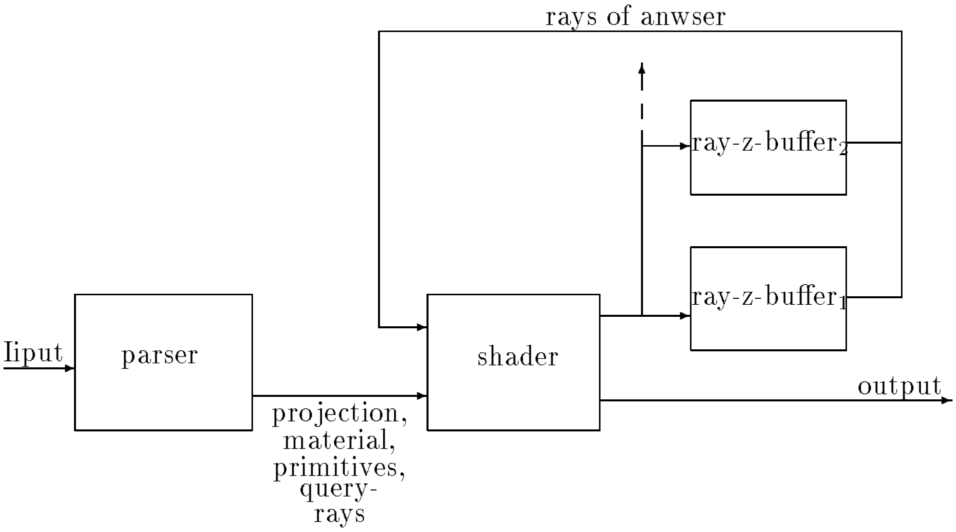

Figure 4: The processes in the image generation system and their channels of communication.

Figure 5: Rays of reflection and refraction

Figure 6: A sphere of crystal

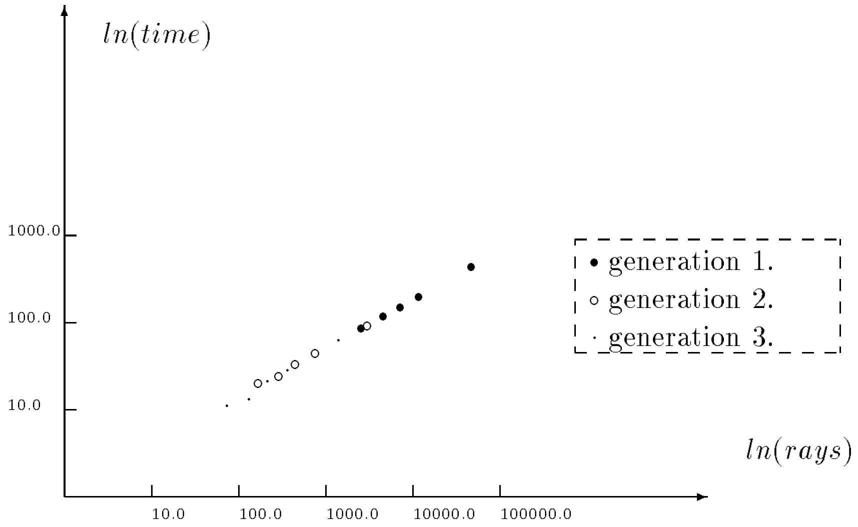

Figure 7: rays versus time in logarithmic presentation.

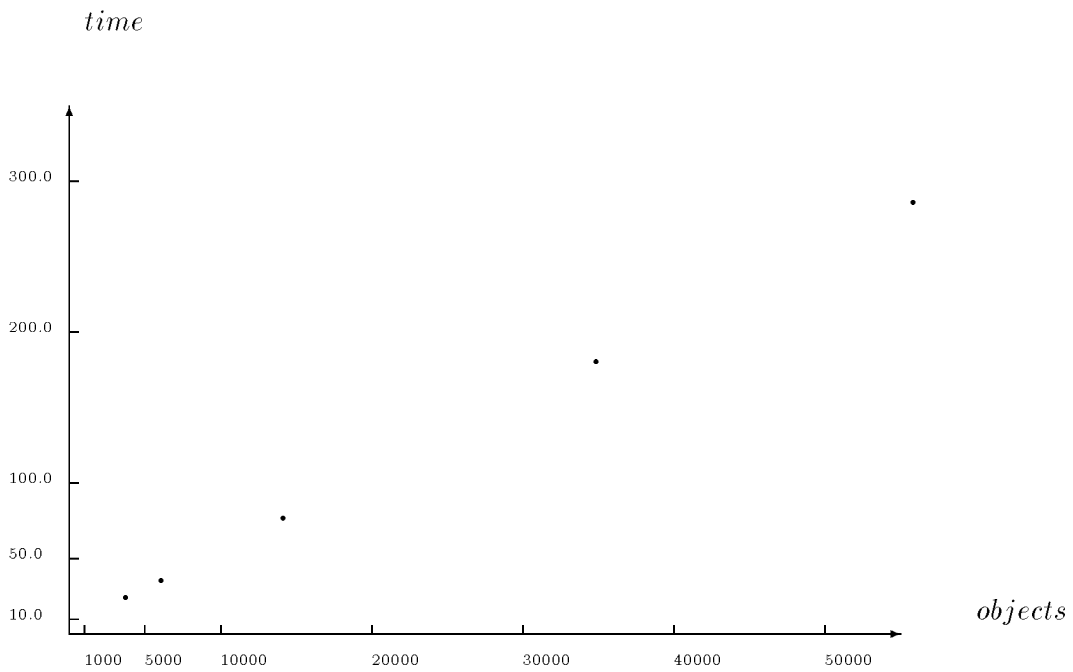

Figure 8: Objects versus time.