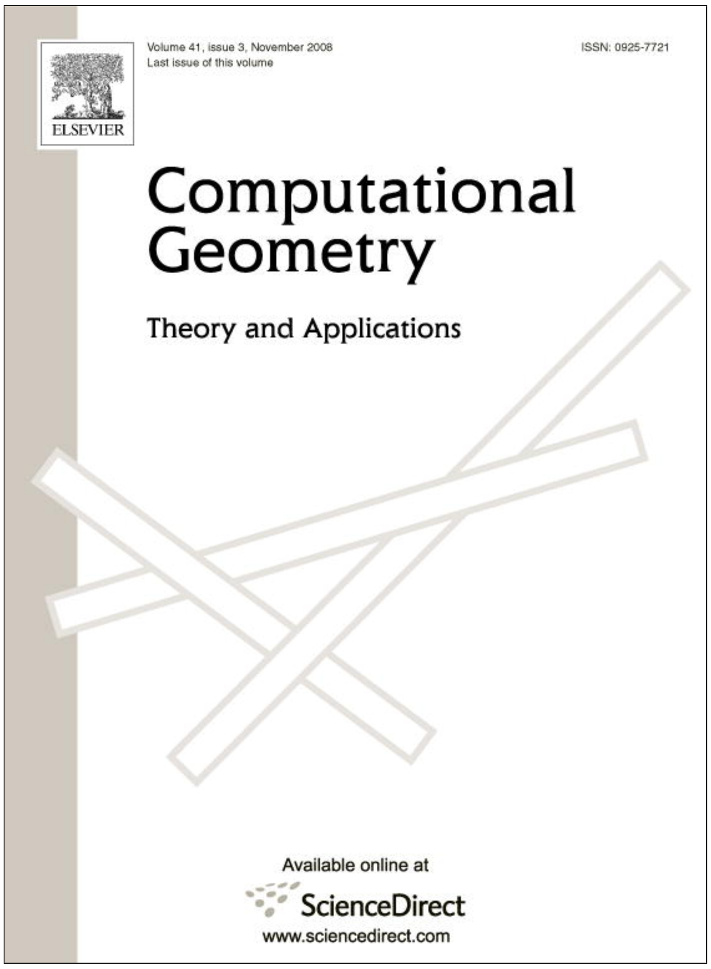
Volume 41, issue 3, November 2008
Last issue of this volume
ISSN: 0925-7721
ELSEVIER
Computational Geometry
Theory and Applications
Available online at
ScienceDirect
www.sciencedirect.com

Available online at www.sciencedirect.com

ScienceDirect

Computational Geometry 41 (2008) 167–187

Computational Geometry
Theory and Applications

www.elsevier.com/locate/comgeo

The BOXEL framework for 2.5D data with applications to virtual drivethroughs and ray tracing

Nir Goldschmidt, Dan Gordon [*]

Department of Computer Science, University of Haifa, Haifa 31905, Israel

Received 5 January 2007; accepted 18 September 2007

Available online 21 November 2007

Communicated by T. Asano

Abstract

The framework of *boxels* is developed to represent 2.5D datasets, such as urban environments. Boxels are axis-aligned non-intersecting boxes which can be used to directly represent objects in the scene or as bounding volumes. Guibas and Yao have shown that axis-aligned disjoint rectangles in the plane can be ordered into four total orders so that any ray meets them in one of the four orders. This is also applicable to boxels, and it is shown that there exist four different partitionings of the boxels into ordered sequences of disjoint sets, called *antichains*, so that boxels in one antichain can act as occluders of the boxels in subsequent antichains. The expected runtime for the antichain partitioning is $O(n \log n)$, where n is the number of boxels. This partitioning can be used for the efficient implementation of virtual drivethroughs and ray tracing. Boxels can also be easily organized into hierarchies to speed up the rendering. For drivethroughs, the antichains are processed in front-to-back order together with a run-length encoding of the boxel horizon, yielding real-time rendering of scenes with up to 300,000 buildings. For ray tracing, a ray intersects at most one boxel in an antichain, and the time to determine that boxel is $O(1)$ for most "natural" scenes, and at worst, logarithmic in the size of the antichain. Objects which are not axis-aligned can also be handled by a simple modification. Boxel rendering can also be parallelized for multi-core machines.
© 2007 Published by Elsevier B.V.

Keywords: 2.5D data; Antichains; Axis-aligned; Boxels; Drivethrough; Front-to-back rendering; Horizon; Partial order; Ray-tracing; Rectangles; Urban scene; Virtual reality; Walkthrough

1. Introduction

An active research area in occlusion techniques consists of methods of accelerating the display of urban scenes for virtual walk/drivethroughs. For a recent survey of results in this area see Cohen-Or et al. [6]. Most current methods employ the notion of *occluders*, which are real or virtual objects which can occlude large portions of a scene potentially visible from a certain region. A recent example of such results is Downs et al. [7].

On the other hand, there exists a large body of results on space partitioning techniques and bounding volumes whose purpose is to accelerate ray tracing or projection techniques. Regular spatial partitionings, such as voxels [16], can also

[*] Corresponding author.
E-mail addresses: goldschmidt_nir@emc.com (N. Goldschmidt), gordon@cs.haifa.ac.il (D. Gordon).

0925-7721/$ – see front matter © 2007 Published by Elsevier B.V.
doi:10.1016/j.comgeo.2007.09.003

serve as representations of the basic objects. Some of the partitioning schemes, e.g., octrees and their variants [22–24] are also hierarchical in nature, while others, such as simple spatial subdivisions [10] are not.

A basic property common to many of the above-mentioned schemes is that of *axis-order*, which can be described as follows: There exist eight different traversal orders so that a ray in any direction encounters the sub-volumes in one of these orders. This property is useful both for ray tracing and projection methods. Projection methods using this property can be in either *back-to-front* order [9,19], or in *front-to-back* order [18,21].

The useful property of axis-order is offset by several disadvantages. One of them is that the partitioning planes cut through the entire volume or sub-volume. This means that some objects may be cut arbitrarily, increasing the complexity of the partitioning and the rendering algorithms. In spatial subdivisions, both uniform and non-uniform, *all* the partitioning planes run throughout the entire volume. Another disadvantage of some of these schemes (octrees and uniform subdivisions) is that partitions can only occur at predetermined positions, again incurring a penalty by not being adaptive to the objects in the scene. In the example of [7], a quadtree is used to partition the scene.

A more generally useful scheme would be arbitrary, axis-aligned (disjoint) boxes in 3D space. Such a scheme does not have the property of axis-order, because cyclic overlap is possible. A natural question is what happens when one considers 2.5-dimensional data, i.e., can such data be organized so as to have the property of axis-order? 2.5D data is usually defined as follows: If a point belongs to an object, then all points below it (and above the xy-plane) also belong to the same object. Geographical data in general and urban scenes in particular are usually assumed to be 2.5D in nature.

As a positive answer to the 2.5D problem, we introduce the notion of *boxels*. The term “boxel” is derived from *box-element* or *box-cell*. Boxels have the following properties:

- Axis-order: Four different orders are sufficient for back-to-front or front-to-back traversal.
- There are no boundaries that extend beyond a boxel.
- Boxels can be hierarchical or not, depending on the application or on considerations of efficiency. The hierarchy option is detailed in the rendering application.

Boxels are axis-aligned boxes with disjoint interiors, satisfying the 2.5D property. The boxels can be used as bounding volumes of various types of objects, or they can serve as the basic building blocks of the actual data, just like voxels. The above properties of boxels follow from the fact that disjoint, axis-aligned rectangles in the plane can be organized so as to have the property of axis-order, as shown by Guibas and Yao [15], who studied these objects with the purpose of translating sets of rectangles in the plane without intersections. The notion of boxels and their basic properties were (independently) proposed by Gordon in [12,13] as potentially useful for rendering 2.5D and some 3D scenes. Axis-aligned rectangles were also studied by Bellantoni et al. [4], and by Asano et al. [2].

In this paper we develop further theoretical and practical properties of boxels which make them extremely useful for very rapid rendering of urban scenes. We show that there exist four different partitionings of the boxels into ordered sequences of disjoint sets, called *antichains*, so that any ray interacts with one of the sequences as follows:

1. For every antichain in the sequence, the ray intersects at most one boxel in the antichain.
2. Given an antichain in the sequence, it takes at most logarithmic time to determine which boxel (if any) in the antichain is intersected by the ray. For most “natural” scenes, this can even be done in O(1) time.
3. If the ray hits a boxel in a certain antichain, then it is obscured from all the boxels in subsequent antichains.
4. The partitioning into antichains takes an expected time of $O(n \log n)$, where n is the number of boxels.

A result of these properties is that from any viewpoint, a scene can be rendered in front-to-back order so that the visible boxels act as occluders, resulting in output-sensitive display. Non-axis aligned objects can also be handled, at a very slight increase in overhead. A hierarchy of boxels, together with a run-length encoding (RLE) of the boxel horizon enables extremely fast rendering: datasets of 250,000 boxels can be rendered in real time on a standard PC with standard graphics hardware. Our rendering method lends itself to very simple, image-driven parallelization, thus it can take advantage of current multi-core processors. The antichain partitioning is also very useful for the acceleration of ray tracing in 2.5D scenes. However, this topic has not yet been tested in practice.

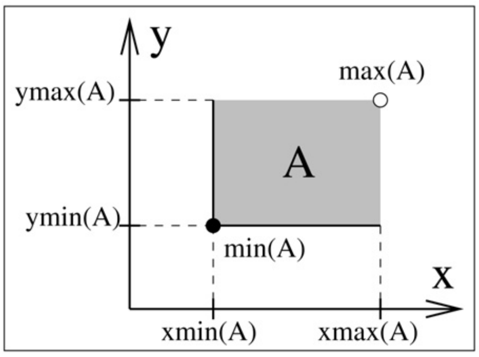

Fig. 1. Axis-aligned rectangle A defined by its opposite corners.

The rest of this paper is organized as follows: Section 2 presents the basic theory of boxels and Section 3 presents the antichain partitionings of boxels. Section 4 explains the front-to-back rendering of boxels with a horizon mechanism, while Section 5 presents the runtime results. We conclude in Section 6 with some further research directions.

2. Previous work

The results in this section were obtained by Guibas and Yao [15], but we present them in some detail since they are basic to our extensions and applications. They also appear in [13].

2.1. *Definitions and basic properties*

In the following, we assume that all rectangles in the xy-plane are axis-aligned, i.e., their sides are parallel to the x and y coordinate axes. An axis-aligned rectangle A is defined by its lower-left corner (x_1, y_1) and its upper-right corner (x_2, y_2). We assume throughout the following that such a rectangle satisfies $x_1 < x_2$ and $y_1 < y_2$, i.e., the rectangles are not degenerate. Given a rectangle A as above, we denote $\text{xmin}(A) = x_1$, $\text{ymin}(A) = y_1$, $\text{xmax}(A) = x_2$ and $\text{ymax}(A) = y_2$. We also denote its minimal and maximal corners as $\min(A) = (x_1, y_1)$ and $\max(A) = (x_2, y_2)$. Fig. 1 illustrates the above definitions.

In order to avoid ambiguous containment relations between rectangles, we henceforth assume that the lower left corner together with its two incident edges are part of A, and the other corners and edges are not. This is shown in Fig. 1, in which the edges forming part of A are enhanced. Formally, we have: $A = \{(x, y) \mid \text{xmin}(A) \leqslant x < \text{xmax}(A),\ \text{ymin}(A) \leqslant y < \text{ymax}(A)\}$.

The concept of *dominance* is well-known in computational geometry [20], though here we use it with strict inequalities:

Definition 1. Let $p_1 = (x_1, y_1),\ p_2 = (x_2, y_2)$ be two points in the plane. We say that p_1 *is dominated by* p_2, and denote it by $p_1 \prec p_2$, if $x_1 < x_2$ and $y_1 < y_2$.

We now use the notion of dominance to define a binary relation $\ll$ between rectangles as follows:

Definition 2. Let A and B be two axis-aligned rectangles with *disjoint interiors* in the plane. We say that A *precedes* B, and denote it by $A \ll B$, if $\min(A) \prec \max(B)$.

Fig. 2 shows an example of the $\ll$ relation among various rectangles. The region dominated by the maximal corners is delimited by the broken lines and the arrows are in the direction of the relation $\ll$. Clearly, if $A \ll B$, then any ray in direction of non-decreasing x and y, if it meets both A and B, must meet A before B.

However, the relation $\ll$ is not suitable for our purposes, since it is not total, and not even transitive, as shown in Fig. 2: $A \ll B \ll C$, but A and C are incomparable with respect to $\ll$. We wish to extend the relation $\ll$ to a total order, but this cannot be done globally for all possible rectangles for a simple reason: Consider the incomparable rectangles A and C of Fig. 2. If our set of rectangles includes a separating rectangle B as in the figure, then in any

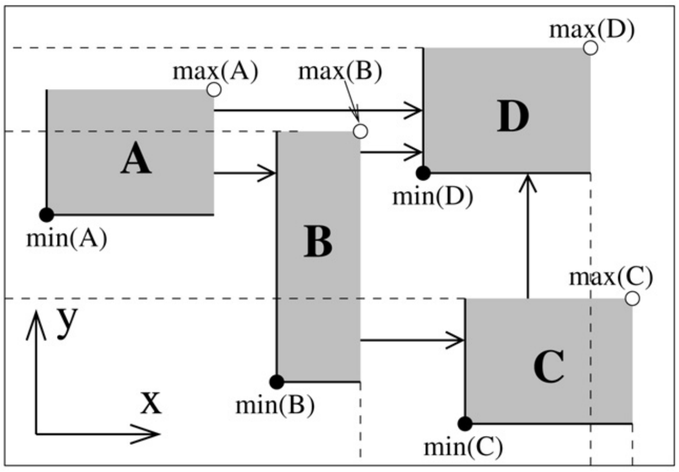

Fig. 2. 4 rectangles, with the relation $\ll$, shown by the arrows, derived from the dominance relation. $A \ll B \ll C$, but A and C are incomparable with respect to $\ll$.

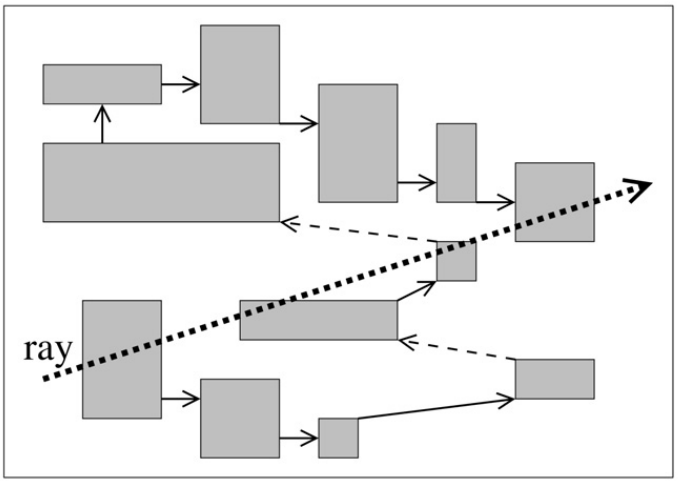

Fig. 3. A set of rectangles with the total order $<_+^+$ shown by the arrows. The diagonal (dashed) arrows are between elements which are incomparable with respect to $\ll$. A ray meets the rectangles in the order $<_+^+$.

total order extending $\ll$, A must precede C. However, if A and C were separated by a long thin *horizontal* rectangle B', then we would have $C \ll B' \ll A$, so in any total order extending $\ll$, C would have to precede A.

We therefore restrict our goal to that of extending $\ll$ to a total order when the set of rectangles is fixed. Let V be a set of axis-aligned rectangles with disjoint interiors. Assume that the relation $\ll$ on the elements of V has been extended to its transitive closure. We can define a digraph (directed graph) $G = (V, E)$ whose vertices are the rectangles, and the arcs (directed edges) are defined as follows: $E = \{(A, B) \mid A, B \in V \text{ and } A \ll B\}$ The extension of $\ll$ to a total order on V follows from the following theorem, which is proved in [13,15].

Theorem 1. *Let $G = (V, E)$ be a digraph as defined above. Then G is acyclic.*

Since we have a directed acyclic graph, we can extend $\ll$ to a total order by doing a topological sort [1,17]. Note that the total order obtained is not necessarily unique. The problem of computing such a total order, which we call the *boxel ordering problem*, is addressed in Section 2.2.

Given a set of rectangles V, we find some total order $<_+^+$ extending $\ll$, and we denote by $<_-^-$ the reverse total order. Similarly, we find some total order $<_+^-$ extending the partial order of non-increasing x and non-decreasing y, with $<_-^+$ denoting its reverse order. Fig. 3 shows a set of rectangles with the total order $<_+^+$ shown by the arrows. The diagonal arrows join rectangles which are incomparable with respect to $\ll$.

2.2. Topological sorting of boxels

We now consider the following problem: Given a set V of axis-aligned rectangles (with disjoint interiors), find a total order extending $\ll$. If we tried a naive implementation of the general topological sorting algorithm, we would first need to compute E from V, and this would take $\Omega(n^2)$ time, where $n = |V|$. To see this, consider a set of rectangles stacked on top of each other—such a set is totally ordered by $\ll$, so $|E| = n(n-1)/2$, from which it follows that just producing E from V takes $\Omega(n^2)$ time. Note that finding E is a special case of the *dominance merge problem* [20] in two dimensions. Another problem with the naive approach is that even if E were available, we would still need $\Omega(n^2)$ time (and space) because the optimal time for topological sorting is $\mathrm{O}(|V| + |E|)$ [1,17].

For our purposes, we do not really need an explicit representation of E. It is sufficient to obtain the sequential order in some linear data structure that can be traversed in both directions, e.g., a doubly-linked list. The following result was shown in [15]; it also appears in [13].

Theorem 2. *Let $G = (V, E)$ be a digraph as defined above. The boxel-ordering problem can be solved in time of $\mathrm{O}(n \log n)$ and optimal space $\mathrm{O}(n)$. The time is optimal in the decision-tree model of computation.*

The $\mathrm{O}(n \log n)$ optimality follows from the fact that sorting is linear-time reducible to the boxel-ordering problem. The boxel-ordering algorithm is in the form of a plane-sweep, as detailed below.

Algorithm 1 *(Boxel-ordering).*

```
Input: A set $V$ of $n$ axis-aligned rectangles.
Output: A doubly-linked list of the rectangles whose order is an extension of the relation $\ll$.
Data Structures:
  TREE: A balanced binary search tree containing the active rectangles, ordered by xmin.
  LIST: A doubly-linked list, whose elements are always in some total order extending $\ll$.
begin
  Let $Y = \{\mathrm{ymin}(A) \mid A \in V\} \cup \{\mathrm{ymax}(A) \mid A \in V\}$.
  ($Y$ is a set and every value appears in it only once.)
  Sort $Y = \{y_1, \ldots, y_k\}$, where $y_1 < \cdots < y_k$ are all the different elements of $Y$ in sorted order.
  for (every $y_j \in Y$) maintain 2 separate lists, one for rectangles starting at $y_j$ and one for those terminating at $y_j$.
  Initialize TREE and LIST to $\emptyset$.
  for ($j = 1$ to $k$) (sweep stage)
    Delete from TREE all rectangles whose ymax is $\leqslant y_j$.
    for (every $A \in V$ with $\mathrm{ymin}(A) = y_j$)
      Insert $A$ into TREE.
      Let $B = A$'s successor in TREE (if $A$ is last, $B =$ NIL).
      Insert $A$ into LIST before $B$ (at end if $B =$ NIL).
    endfor
  endfor
  Output LIST.
end
```

The arrows in Fig. 3 show the total order that is obtained by applying the boxel-ordering algorithm to the given set of boxels. Arrows connecting incomparable adjacent elements on LIST are shown as dashed. Two implementation details are given below.

An element's successor on a binary search tree is the last element on the search path from the root to the element's position, from which the *left* branch was taken during the search—see [11]. Thus, we can easily find a new element's successor (as required by algorithm 1) by initializing a pointer to *NIL* and changing its value at every node from which the left branch is taken. At the end of the search, the pointer points at the inserted rectangle's successor on TREE. If the pointer remains *NIL*, then the inserted element is last on TREE.

Another implementation detail is the following: Algorithm 1 produces a list of the elements in an order $<_+^+$ extending $\ll$, and the same list also gives us $<_-^-$. We can obtain the orders $<_+^-$ and $<_-^+$ at a minimal additional cost as follows: We use the same sorted set Y and the same TREE, but in addition to LIST, we maintain another list, LIST1, for the order $<_+^-$. During the insertion of a new rectangle A, in addition to finding A's successor on TREE, we also find A's predecessor on TREE and insert A into LIST1 immediately before that predecessor. If A is first on TREE, we insert A at the end of LIST1. At the end, LIST1 holds the rectangles in a total order $<_+^-$ extending the relation of non-increasing x and non-decreasing y; the reverse order on LIST1 is $<_-^+$.

3. The antichain partitioning

This section extends the previous boxel definitions and structures for the purposes of fast rendering. Given a set with a non-reflexive binary relation, a subset is called an *antichain* if any two elements in it are pairwise incomparable w.r.t. the relation. Consider a set of boxels V with the relation $\ll$: We want to partition it into subsets so that every subset is an antichain. There may be several different such partitionings of V, but, for our purposes, we want the antichains to form an ordered *sequence* so that the boxels in one antichain can act as occluders of boxels in subsequent antichains. This is shown in the following two subsections.

3.1. Definitions and basic properties

Definition 3. Given a set of boxels V, we define

$$\min(V, \ll) = \{B \in V \mid \neg(\exists A \in V)(A \ll B)\}.$$

$\min(V, \ll)$ is the set of all elements of V which are minimal w.r.t. $\ll$, i.e., they have no predecessors. The antichain partitioning of V is given by:

Definition 4. Given a set of boxels V, the antichain partitioning of V w.r.t. $\ll$ is the sequence $V_1, \ldots, V_m$ of disjoint subsets of V, defined by

$$V_1 = \min(V, \ll);$$

$$\text{For } i > 1, \quad V_i = \min\left(V - \left(\bigcup_{j=1}^{i-1} V_j\right), \ll\right).$$

The number of elements, m, in the sequence is the number of non-empty sets obtained by the above rule.

Clearly, each V_i is an antichain w.r.t. $\ll$. Similar definitions to 3 and 4 also apply to the other three relations on V. Fig. 4 shows an example of the antichain partitioning. The main properties of the partitioning are summarized in the following theorem.

Theorem 3. *Let V be a set of boxels, $V_1, \ldots, V_m$ the antichain partitioning of V w.r.t. $\ll$, and R a ray in direction of non-decreasing x and y. Then*:

1. *For $1 \leqslant i \leqslant m$, any two boxels in V_i are incomparable with respect to $\ll$.*
2. *For $1 \leqslant i \leqslant m$, R meets at most one boxel of V_i.*
3. *If $1 \leqslant i < j \leqslant m$ and R meets some boxel of V_i, then R is obscured from all boxels of V_j.*
4. *For $1 \leqslant i \leqslant m$, the time to determine which boxel in V_i (if any) intersects R is logarithmic in the size of V_i.*

The proof of items 1–3 of Theorem 3 follow immediately from the basic properties of the relation $\ll$ and the antichain definition. Item 4: Consider the minimal corners of the boxels of V_i. Clearly, they are strictly ordered in order of increasing x (and also in order of increasing y). Hence, a binary search on these boxels will determine in logarithmic time which boxel, if any, is intersected by the ray.

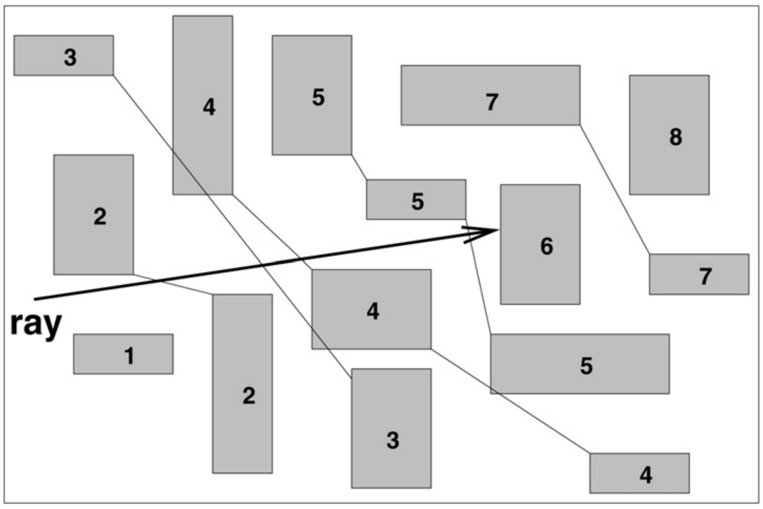

Fig. 4. The antichain partitioning $V_1, \ldots, V_8$ of a set of boxels, w.r.t. to $\ll$. The boxels of V_i are marked with i.

3.2. Construction of the antichain partitioning

Clearly, the first boxel on LIST (denote it B) is a minimal element w.r.t. $\ll$ (i.e., it has no predecessors), so it belongs to the first antichain V_1. The other elements of V_1 (if any) will be some elements of LIST which have no predecessors w.r.t. $\ll$ on LIST. A naive search of LIST using this approach would require a total search time of $O(n^2)$. Instead of proceeding in this manner, we utilize the information available to us from the boxel-ordering algorithm, together with some additional data structures.

The total order $<_+^+$ is an extension of the partial order $\ll$, so LIST can be broken up into t *sublists* of boxels, $B_1^1, \ldots, B_{k_1}^1,\ B_1^2, \ldots, B_{k_2}^2, \ldots, B_1^t, \ldots, B_{k_t}^t$, such that:

- In every sublist, successive elements are strictly ordered by the relation $\ll$.
- The last element of every sublist is incomparable with the first element of the next sublist, i.e., for every $1 \leqslant i < t$, $B_{k_i}^i$ and B_1^{i+1} are incomparable w.r.t. $\ll$.

We call the first elements of the sublists the *headers* of the sublists. Clearly, the elements of V_1 can only be sublist headers. As already noted, $B_1^1 \in V_1$. Whenever a header is added to an antichain, then next element on its sublist (if any) becomes the sublist's header.

It might seem as if the next element in V_1 (after B_1^1) is simply the first sublist header on LIST which is incomparable with B_1^1, but this is not so: on the first sublist, $B_1^1, B_2^1, \ldots, B_{k_1}^1$, there might be elements which extend further to the left than B_1^1. This means that even if B_1^1 is incomparable with B_1^i for some $i > 1$, we could have $B_j^1 \ll B_1^i$ for some $1 < j \leqslant k_1$. An example of this is shown in Fig. 5: the first sublist is $B_1^1, \ldots, B_6^1$, and the next sublist starts with B_1^2. Even though B_1^1 and B_1^2 are incomparable, B_1^2 cannot be on the same antichain as B_1^1 because it is preceded by B_5^1 in the order $\ll$.

In order to resolve this problem, we add the following information to the boxels on LIST: every boxel B on a sublist has a data field containing the minimal xmin of all boxels on the sublist from (and including) B to the end of the sublist. We denote this value by sublist_xmin(B); it is found in linear time by working *backwards* on each sublist. For this purpose, the sublists are organized as two-way linked lists. In the example of Fig. 5,

$$\text{sublist\_xmin}(B_1^1) = \text{xmin}(B_5^1).$$

Since all elements of an antichain are sublist headers, we construct a list of pointers $P = (p_1, \ldots, p_t)$, such that each p_i points at B_1^i. P is organized as a two-way linked list, and it can be easily constructed from LIST in linear time. Note that the sublist headers are in increasing order of xmin, so each antichain will be produced in sorted order by the algorithm below.

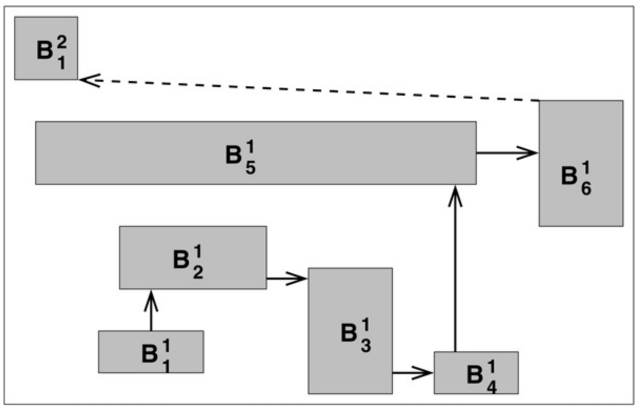

Fig. 5. An example of searching for elements of an antichain.

Algorithm 2 *(Antichain construction).*

```
Input: The $t$ sublists of LIST, each organized as a two-way list, as explained above.
Output: An ordered sequence of antichains $V_1, V_2, \dots,$ each sorted by xmin.
Data Structure:
  A two-way list of pointers $P = (p_1, \dots, p_t)$, with $p_i$ pointing at $B_1^i$.
begin
  for ($i = 1$ to $t$) [loop to set up sublist_xmin.]
    current_xmin $= \infty$
    for ($j = k_i$ downto 1)
      current_xmin $=$ min(current_xmin, xmin($B_j^i$))
      sublist_xmin($B_j^i$) $=$ current_xmin.
    endfor
  endfor
  $k = 0$ [$k$ will index the antichains.]
  while ($t > 1$)
    $k = k + 1$
    Initialize $V_k$ to an empty linked list.
    current_xmin $= \infty$.
    for ($i = 1$ to $t$) [loop on sublist headers.]
      if (xmax($B_1^i$) $\leqslant$ current_xmin)
        Place $B_1^i$ on antichain $V_k$.
        Remove $B_1^i$ from its sublist and update pointers of $P$.
      endif
      current_xmin $=$ min(current_xmin, xmin($B_1^i$)).
    endfor
    for (every empty sublist) $t = t - 1$.
  endwhile
  if ($t = 1$) place every remaining boxel in a new (and separate) antichain.
  Place every sorted antichain into an array.
end
```

Theorem 4. *Let V be a set of boxels, LIST the list of boxels produced by Algorithm* 1*, and assume we are given the sublists of LIST as described above. Then Algorithm* 2 *produces the antichain partitioning $V_1, \dots, V_m$ of V.*

Proof. Let V_1' be the set of boxels produced by Algorithm 2 when its variable k is equal to 1. It is clearly sufficient to show that $V_1' = V_1$. As already noted, the elements of V_1 can only consist of boxels which are sublist headers, therefore

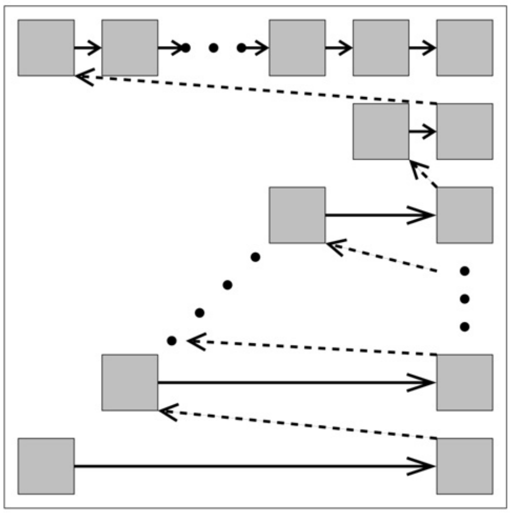

Fig. 6. An example for which the antichain construction takes $O(n^2)$ time.

$V_1 \subseteq V_1'$. Assume that Algorithm 2 places some sublist header B_1^i in V_1', but $B_1^i \notin V_1$. Hence, B_1^i has some predecessor $B \in V$ w.r.t. $\ll$, i.e., $B \ll B_1^i$. B cannot come after B_1^i on LIST, because the order on LIST extends $\ll$, therefore B precedes B_1^i on LIST. From $B \ll B_1^i$, it follows that the point $\max(B_1^i)$ dominates the point $\min(B)$. Therefore, we have

$$\mathrm{xmin}(B) < \mathrm{xmax}(B_1^i). \tag{1}$$

Consider now the variable current_xmin used in Algorithm 2. When it is compared with $\mathrm{xmax}(B_1^i)$, its value is always the minimal xmin of all the boxels preceding B_1^i on LIST (because in every sublist, the header's sublist_xmin is minimal in the sublist, and current_xmin is the minimal sublist_xmin of all the headers preceding B_1^i on LIST). Therefore, when current_xmin was compared with $\mathrm{xmax}(B_1^i)$, the following inequality was true:

$$\mathrm{current\_xmin} \leqslant \mathrm{xmin}(B). \tag{2}$$

When B_1^i was placed by Algorithm 2 in V_1', the following condition necessarily held (see Algorithm 2):

$$\mathrm{xmax}(B_1^i) \leqslant \mathrm{current\_xmin}. \tag{3}$$

Now, from inequalities (2) and (3), we have $\mathrm{xmax}(B_1^i) \leqslant \mathrm{xmin}(B)$, contradicting inequality (1). Therefore, $V_1' = V_1$. □

The worst-case of the above algorithm takes $O(n^2)$, as can be seen from the example shown in Fig. 6. Each sublist, except the topmost, consists of just two elements, and the sublist headers form a diagonal running from bottom-left to top-right. Every element on the diagonal belongs to a separate antichain, but the antichain construction algorithm compares every boxel on the diagonal with all the boxels above it on the diagonal (except the last), so the number of comparisons is $O(n^2)$. Note that exchanging the roles of x and y in the boxel sorting and the antichain algorithms will not change this worst-case. However, if we work in the direction of decreasing x and y, the time for this example will be just $O(n)$.

Unfortunately, a simple example (based on the above) can be constructed such that the time for the antichain construction is $O(n^2)$ in any of the directions: consider a layout of boxels in which the above example is first reflected about the center of the rightmost column, and the result is then reflected about the center of the topmost row. Any chosen direction will now have the same worst-case behavior.

However, in practice, the antichain construction is extremely fast. This can be explained by considering an example of n identical square boxels, arranged in a grid of $m \times m$, where $m = \sqrt{n}$, as shown in Fig. 7. The sublists are the rows, and the antichains are the diagonals. It is easy to see that every diagonal from the lower left up to (and including) the main diagonal, uses exactly $m - 1$ comparisons, and each successive diagonal needs one less than its predecessor. Since we have $2m - 1$ diagonals, the antichain construction (from LIST) takes $O(m^2) = O(n)$ time. It can be argued

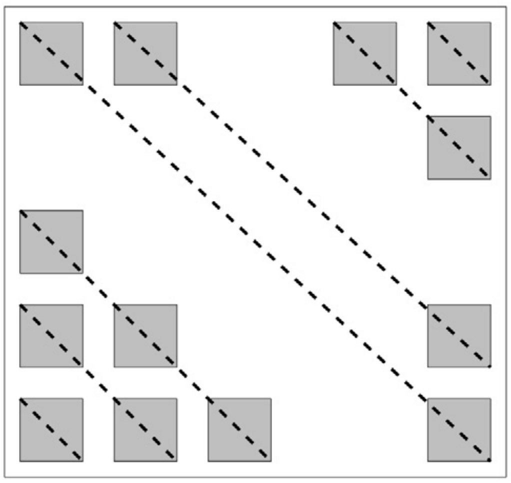

Fig. 7. An example for which the antichain construction takes linear time.

that most "reasonable" examples of city environments are, in some sense, not too different from this example, so the antichain construction is very fast in practice. In the next subsection, we propose a probabilistic model for the distribution of boxels, and show that in this model, the expected time is also linear.

3.3. A probabilistic model for boxel distribution

We consider the following method for setting up boxels: An axis-aligned rectangle of dimensions $L \times W$ is considered as bounding all the boxels. Its lower-left corner is $(0, 0)$ and its upper-right corner is (L, W). Now, for any $n \geqslant 2$, we set up n boxels as follows: The x-coordinate of the boxel's center is chosen randomly according to some distribution in the open interval $(0, L)$, and the y-coordinate is chosen according to some (possibly different) distribution in the open interval $(0, W)$. The choices of the x- and y-coordinates are made independently of each other.

We now choose the x and y extents of the boxel as two numbers a and b, respectively, from some distribution functions, which may depend on n. The exact distribution is immaterial, but we stipulate the following restriction: a may *not* have a lower bound $a^* > L/2$. We select n boxels in this manner, independently of each other. If, after setting up the n boxels, there are any boxel intersections, or if some boxel extends beyond the bounding rectangle, then the entire set of boxels is rejected, and the process is repeated. Let $\{B_1, \dots, B_n\}$ be a set of boxels selected in this manner.

For $n \geqslant 2$, we denote by p_n the probability that for any ordered pair of boxels, (B_i, B_j), with $i \neq j$ and $1 \leqslant i, j \leqslant n$, B_i is completely to the left of B_j (i.e., $\mathrm{xmax}(B_i) \leqslant \mathrm{xmin}(B_j)$). Note that according to our method for selecting the set of n boxels, no order between the boxels is involved, so p_n does not depend on the positions of i and j in the sequence $1, \dots, n$.

We can now state the following result.

Theorem 5. *Assuming the above model for the distribution of boxels, the expected runtime of the antichain partitioning algorithm is linear in the number of boxels n.*

The proof is based on the following two lemmas:

Lemma 1. *Assuming the above model for the distribution of boxels,* $0 < p_2 \leqslant p_n \leqslant 0.5$.

Proof. $p_n \leqslant 0.5$ follows from the definition of p_n by symmetry consideration. $p_2 > 0$ follows from our restriction on a possible lower bound on the choice of a (if we allow a lower bound $a^* > L/2$, then we cannot place two boxels so that one is to the left of the other, meaning that $p_2 = 0$). We now show that p_n is monotonically increasing. Denote by A_n the average area of a boxel in all possible choices of n boxels. Since the boxels do not intersect and they do not extend beyond the bounding rectangle, the sum of the boxel areas is bounded by LW, so A_n is a monotonically decreasing sequence. It follows from this that if we denote by a_n the average x-extent of a boxel, then a_n is also monotonically decreasing. Consider now two boxels B_i, B_j in a choice of n boxels. There are three mutually exclusive possibilities:

1. B_i lies to the left of B_j.
2. B_i lies to the right of B_j.
3. Their projections on the x-axis overlap.

The probabilities of cases 1 and 2 are both p_n, so the probability of case 3 is $1-2p_n$. Since a_n decreases monotonically, $(1-2p_n)$ also decreases monotonically. Therefore, p_n is monotonically increasing. □

Lemma 2. *Assuming the above model for the distribution of boxels, then in the antichain partitioning algorithm, the expected number of comparisons required to place a boxel on an antichain is bounded by $1/p_n$.*

Proof. After placing the first boxel on an antichain, each successive sublist header is placed on the antichain if it lies to the left of a certain boxel in a previous sublist. The probability of this is $1/p_n$. If it is not placed, then the next sublist header is examined, and so on. The number of boxels that are examined before the next boxel is added to an antichain is at most $n-1$. Hence, the expected number of comparisons to place a boxel on an antichain is bounded by:

$$E_n \leqslant 1p_n + 2(1-p_n)p_n + 3(1-p_n)^2 p_n + \cdots + (n-1)(1-p_n)^{n-2} p_n.$$

Note that we may be making all these comparisons without finding any sublist header to place on the antichain. In this case, the antichain is terminated, and we "charge" these comparisons to the first element on the next antichain. We thus get:

$$E_n \leqslant p_n \sum_{i=1}^{n-1} i(1-p_n)^{i-1} < p_n \sum_{i=1}^{\infty} i(1-p_n)^{i-1}. \tag{4}$$

From Lemma 1, $0.5 \leqslant 1-p_n \leqslant 1-p_2 < 1$. The sum on the right in Eq. (4) is easily evaluated by elementary calculus. For $0 < x < 1$, $\sum_{i=0}^{\infty} x^i = 1/(1-x)$, and the sum converges uniformly on the closed interval $[0.5, 1-p_2]$. This allows us to differentiate both sides w.r.t. x, yielding $\sum_{i=1}^{\infty} i x^{i-1} = 1/(1-x)^2$, for $x \in [0.5, 1-p_2]$. Taking $x = 1-p_n$, we get $\sum_{i=1}^{\infty} i(1-p_n)^{i-1} = 1/p_n^2$. Therefore, $E_n < p_n/p_n^2 = 1/p_n$. □

Proof of Theorem 5. From Lemmas 1 and 2, the expected number of comparisons for each boxel placed on an antichain is $E_n < 1/p_n \leqslant 1/p_2$. Since all the other times required by the algorithm are linear in n, the total expected time is also linear in n. □

4. Boxel rendering

4.1. General comments

We will consider projection methods and ray tracing. By "projection methods" we mean rendering methods where the objects are projected onto the screen. To obtain hidden surface removal, several techniques are available, such as the Z-buffer, or rendering in a back-to-front or front-to-back order. Rendering a single boxel depends on the application: If the boxels are the actual data, then we simply project the boxel on the screen. If the boxels are bounding volumes of other objects, then we project the object(s) inside. We assume that we have routines for projecting a single boxel, and this means that these routines can handle anything inside a single boxel.

Assume first that we are dealing with orthogonal projections only, and that the screen is orthogonal to a vector whose projection on the (world) xy-plane is in direction of non-decreasing x and y. Clearly, all we have to do to obtain hidden surface removal is to render the boxels in *back-to-front* order, i.e., in the order $\overset{-}{<}$. The boxels can also be rendered in *front-to-back* order. This requires a data structure for the displayed pixels which will enable the rapid elimination of hidden objects. Meagher [18] used quadtrees for the image, when the 3D objects were octrees. Data structures that have been used for this purpose were quadtrees [18] (when the 3D objects were octrees), and run-length encodings of scanlines for voxels [21] and BSP trees [14].

Perspective projections are slightly more complicated. Consider the projection of the view volume [8] on the world xy-plane. If it does *not* contain a line parallel to the x or y axis, then back-to-front (or front-to-back) techniques will

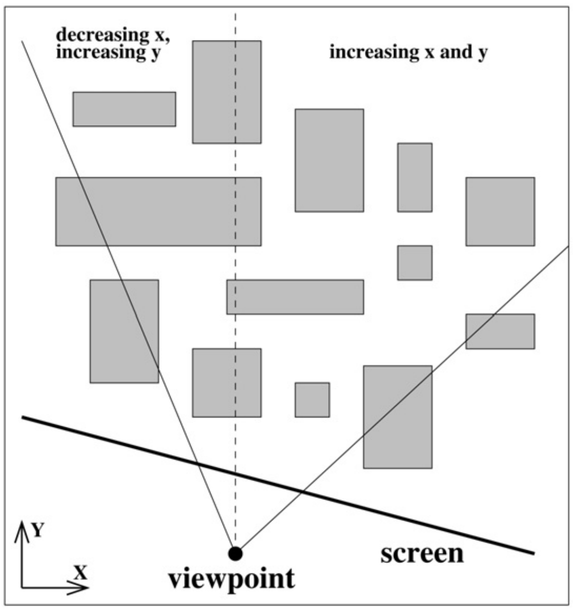

Fig. 8. Perspective projection: The view volume is split in two. Each part of the scene is displayed by projecting according to a different total order.

achieve hidden surface removal. Otherwise, we need to split the view volume into two parts, and display the objects in each part according to the relevant orders. This is demonstrated in Fig. 8. The view volume is split by a plane passing through the viewpoint, perpendicular to the xy-plane, and parallel to the x or y axis (as may be required) of the world coordinates. If the view angle is greater than 90°, we may even need to split the view volume into three. We assume that the rendering routines are capable of handling the display of a boxel which is only partially contained in one part of a view volume.

Recent years have seen the emergence of multi-core processors for workstations, desktop PC's and even notebook computers. Our projection methods can utilize such processors very simply: we divide the view volume into several subvolumes, in a manner similar to that shown in Fig. 8. Each core is assigned to one part of the volume and processes it. Note that the dividing lines do not necessarily have to be parallel to the x- or y-axis.

4.2. Horizon-aided, front-to-back rendering

Our horizon approach to rendering boxels employs 2 main features:

- Front-to-back traversal of the antichains.
- A dynamic structure, maintaining at each stage the currently visible horizon.

Fig. 9 explains our rendering method. With each boxel, we associate two horizontal spans (in image space), determined as follows:

- The *upper* span is at the maximal height of the boxel's projection on the screen. Any object which hides the upper span necessarily hides the entire boxel.
- The *lower* span is at the maximal level such that any object behind the boxel, if it is below the lower span, then it hidden by the boxel.

The horizon is maintained in image space, and it consists of a list of horizontal spans whose projections on the bottom of the screen do not intersect, but they cover the entire bottom line of the screen. The antichain processing progresses on the antichains in front-to-back order, and in each antichain, it processes the boxels in left-to-right order. Each boxel (on the antichain) is either completely below the current horizon (this is determined by the boxel's upper span), or not. In the first case, the boxel is ignored, and in the second case, the boxel's *lower* span is integrated into

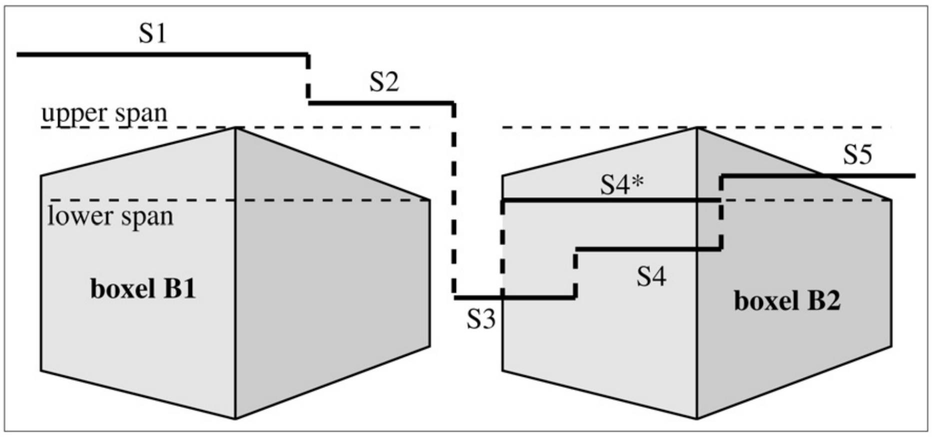

Fig. 9. Two boxels with their upper and lower spans. Current horizon with spans S1–S5 hides B1 because its upper span is beneath S1 and S2. B2 is displayed. Part of S3 and S4 are replaced by S4*, which is part of the lower span of B2.

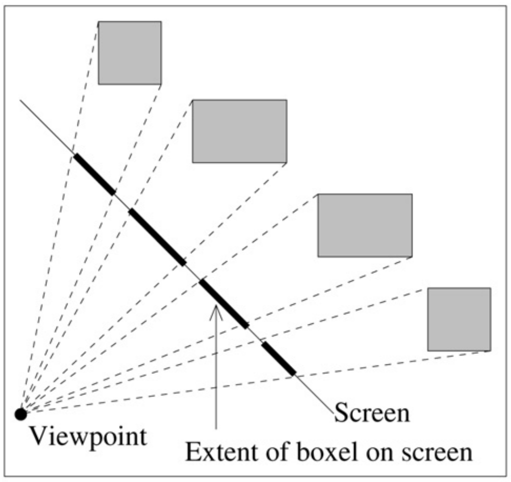

Fig. 10. An antichain of boxels and its corresponding spans on the screen.

the horizon and updates it. Also, in the second case, the boxel is displayed on the screen (or added to a display list to be processed later).

Note that this method of display is approximate—it may happen that some boxel will be determined as "visible" by this method even if it is completely hidden by other boxels. However, the vast majority of the hidden boxels will be eliminated, and the hardware Z-buffer takes care of correct hidden surface removal.

We use run-length encoding (RLE) for maintaining the visible horizon. Downs et al. [7] also use short horizontal spans with conservative estimates, but they use a binary tree for the spans, whereas we use a simple linked list. The advantage of the linked list is that each antichain is also available as a linear structure in left-to-right (or right-to-left) order. This allows us to process the two linear structures in a *merge-type* fashion, similarly to the merging of two sorted sequences. After such processing, we get additional boxels to render and obtain an update of the horizon, as shown in Fig. 9. The complexity of the merge-type operation is linear in the total number of objects in the two lists. Fig. 10 shows (from above) how boxels in an antichain correspond to horizontal spans on the screen. We forgo the technical description of the algorithm since it is essentially fairly straightforward.

To further speed up the antichain processing, we can use some fast search method to find the leftmost (or rightmost) boxel of the antichain that falls in the viewing angle. Such a method can be a binary search on the antichain or the "interpolation" method described in Section 4.5. The merge-type operation terminates either when a boxel falls outside the viewing angle or when the end of the antichain is reached.

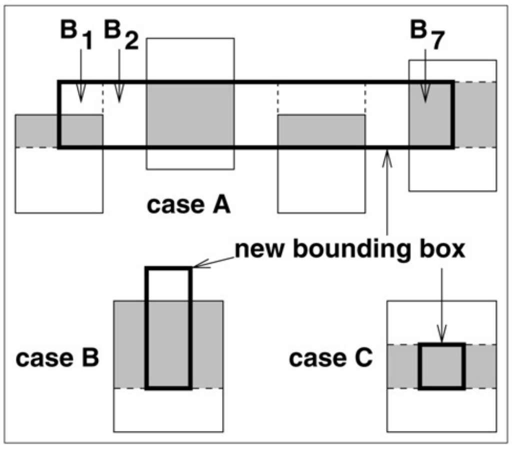

Fig. 11. Bounding boxes added to TREE.

4.3. Handling non-axis-aligned objects

Naturally, real scenes contain objects which are not necessarily axis-aligned. It turns out that this is easier to handle than expected. First, we place a bounding, axis-aligned box around each object. These bounding boxes will usually not form a set of boxels with disjoint interiors, but they can be used to generate such boxels.

This is done by modifying the boxel-sorting algorithm. The data structures TREE and LIST of the boxel-sorting algorithm will contain only boxels (i.e., their interiors will be non-intersecting). Fig. 11 shows what happens when we attempt to insert a new bounding box (outlined with thick lines) to TREE. The figure shows three different cases or situations, A, B and C, which can arise when the box is added. In case A, the new box intersects several boxels which are already on TREE. In cases B and C, xmax and xmin of the new box fall strictly inside a single boxel on TREE. Note that according to our order of insertion, ymin of the new box cannot be smaller than the ymin of any element on TREE. The modifications to the boxel-sorting algorithm can be described as follows:

- Search for the position of the box's xmin and xmax on TREE.
- Create a list of all the boxels on TREE which are intersected by the new bounding box.
- Subdivide the new box into a sequence of boxes $B_1, \dots, B_k$ by *vertical* lines corresponding to the intersections of the box with the vertical edges of the boxels on TREE which it intersects. In case A of Fig. 11, the new box is subdivided into seven boxes.
- For every boxel on TREE intersected by some B_i, subdivide it into two or three parts (as the case may be) by *horizontal* lines corresponding to the horizontal edges of the new bounding box. Note the horizontal dividing lines in Fig. 11. If ymin of the new box is equal to the ymin of some boxel, then there will be no lower part.
- For every subdivided boxel, output the *lower* part to LIST.
- If a subdivided boxel was divided into three parts, add the *upper* part to the list of boxels waiting to be inserted (second and fourth boxels of case A and case C of Fig. 11).
- Consider the remaining boxels on TREE which have a non-empty intersection with some b_i—see the shaded boxels in Fig. 11. We have already cut away the bottom part and the top part (if it existed). The remaining parts are now of two types:
  - Type 1: Boxels contained entirely in some B_i (the second and third boxel of case A of Fig. 11).
  - Type 2: Boxels which are divided into two (or three) parts by one (or both) vertical edge(s) of some B_i (the first and fourth boxel of case A, and cases B and C of Fig. 11).

  Boxels of type 2 are now divided by one (or two) vertical line(s) corresponding to the vertical edge(s) of the B_i that intersects them.
- Every B_i intersecting an existing boxel of the type 2 may need to be subdivided into two parts by a horizontal line corresponding to the top edge of the boxel it intersects (B_1 of case A and case B of Fig. 11). The top part of this B_i is added to the list of boxels waiting to be inserted into TREE.
- The structure of TREE is modified to reflect the boxels that were split.

- Those B_i's which did not intersect any boxel are now inserted into TREE in the regular manner.
- Any boxel formed from an intersection of a bounding box and a boxel now contains links to all the objects of the boxes.

4.4. Hierarchical structure

In order to further speed up the display, we have also implemented a hierarchy on the boxels. Consider the boxels as being at the bottom of the hierarchy, and consider a regular subdivision of the scene into a grid of $n \times m$ axis-aligned squares. With each square, we associate a height which is equal to the tallest object intersected by the square. The boxels in each grid square are subdivided into antichains *independently* of the boxels in other grid squares. The rationale for this hierarchy is based on the fact that if the height of a grid square is lower than the spans of the current horizon which cover the square, then the entire square can be eliminated from further processing. Furthermore, with perspective projections, boxels that are closer to the viewpoint will have a greater height than further ones, so the rate at which we eliminate far squares will (on average) increase as the horizon progresses from front to back.

The grid squares have a natural partitioning into antichains; these are simply successive diagonals on the grid. Rendering with the grid proceeds by accessing the grid antichains in front-to-back order, as follows:

Algorithm 3 *(Hierarchical boxel rendering).*

```
begin
  Set the current horizon to zero.
  for (each grid antichain $A$ in front-to-back order)
    for (each grid square $S \in A$ in left-to-right order)
      if ($S$ is not hidden by the current horizon)
        Update the current horizon with the boxels of $S$ by processing them in the usual manner.
      endif
    endfor
  endfor
end
```

The above algorithm is specified just for a single level above the basic boxels, but extending it to a hierarchy of any level is straightforward. The hierarchy is governed by two parameters which can be fine-tuned to maximize performance: The height of the hierarchy tree, and the grid size. Note that a quadtree partitioning of the scene is actually a special case of a hierarchical partitioning in which the grid size at every level is 2.

4.5. Ray tracing

Since the antichains act as occluders, they provide a simple and natural means for speeding up ray tracing of boxels. Any ray is always in one of the primary directions. Assume that $V_1, \dots, V_k$ is the sequence of antichains (in the antichain partitioning) containing a boxel which may be hit by a ray. Starting with V_1, perform a search to find if the ray hits a boxel in V_1. If the ray passes between two boxels of V_1, we continue with V_2, and so on. Also, it is possible that the projection of a ray on the ground intersects the projection of a boxel on the ground, but it misses the object(s) in the boxel. In that case, the search also continues with V_2.

Recall that the boxels of an antichain are stored in an array. Therefore, one can perform a binary search on the antichain. However, this method can be improved, and there are other efficient methods which we describe below.

Bounded binary search

This method is demonstrated by Fig. 12. It is based on the observation that when a ray passes between two successive boxels on an antichain, its direction is limited by two lines emanating from the edges of the two boxels, as shown in Fig. 12. We can preprocess the antichains so that each edge of a boxel has "bounding links" pointing at the relevant positions in the next antichain, as shown in the figure. These positions act as bounds on the binary search to be performed on the next antichain. Note that if a ray's projection meets a boxel's projection but it does not meet any

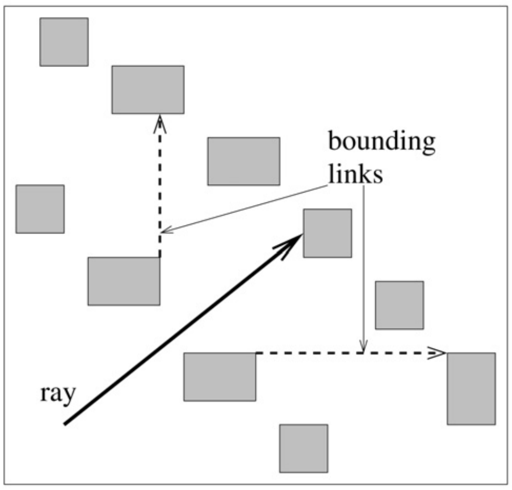

Fig. 12. Ray passing between boxels in antichain. Links point at positions in next antichain limiting the next binary search.

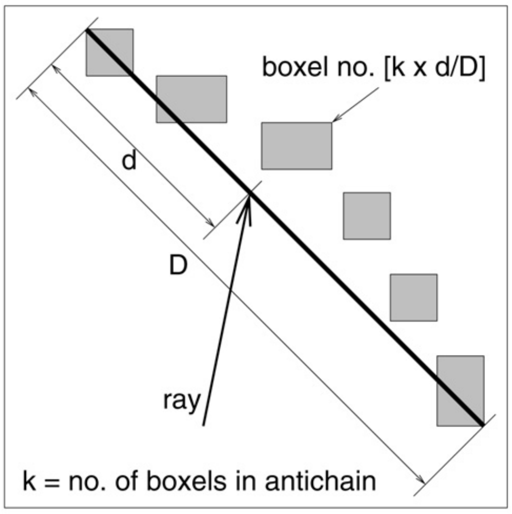

Fig. 13. Searching for a boxel in an antichain using interpolation.

object bounded by the boxel, then similar links can also be used as bounds on the binary search to be performed on the next antichain. When we consider all four directions in the plane, we can see that every boxel needs eight such links—two from each of its four sides.

Interpolation search

This method is demonstrated in Fig. 13. It is based on performing an interpolation-based search on an antichain. Let D be the distance between the extreme corners of the two extremal boxels of the antichain. Assume that the ray meets the line segment joining these extreme corners at a distance d from the left (or right) end of the segment. Our search starts by first examining boxel number $\lfloor k \times d/D \rfloor$ of the antichain, where k is the number of boxels in the antichain (we consider the boxels to be numbered consecutively from 0 to $k-1$). If it misses the boxel, the search continues linearly along the antichain. This method is simpler than the link method and it does not require any preprocessing. However, it can benefit somewhat from a preliminary calculation of the segments, but this preprocessing requires only O(1) time and space per antichain. Clearly, this method is most efficient when the variation in the boxels' sizes is small and the boxels are evenly distributed. Note that the bounded binary search and the interpolation method can be combined by restricting the interpolation search to the subsequence of boxels determined by the bounding links.

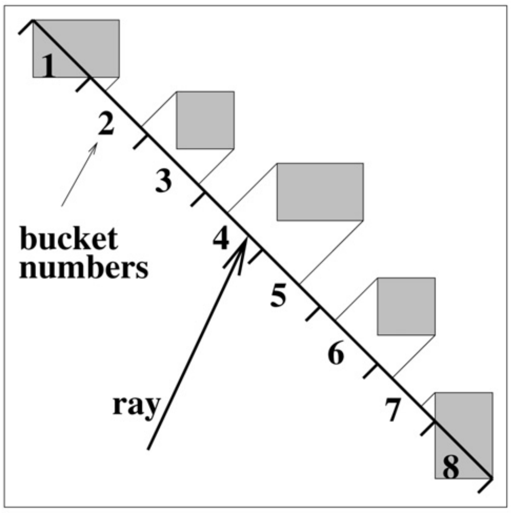

Fig. 14. The bucket method for determining a ray's intersection with an antichain.

Bucket search

Our third method is based on the well-known fact that bucketing techniques are known to work very well in practice for some geometric search problems—see for example [3,5]. Similarly to the interpolation method, we consider the line segment joining the extremal corners of the boxels in the antichain, as shown in Fig. 14. We divide the line segment into a number of equal-length segments which form the buckets. The boxels of the antichain are projected on the line segment, and each bucket is linked to the boxel(s) whose projections intersect it (if there are none, we just indicate the two nearest boxels). Given a ray, we can find the appropriate bucket and start our search from the boxel(s) associated with the bucket. If we take the size of a bucket to be the shortest projection of a boxel, then each bucket will be linked to no more than two boxels.

Two other issues need to be specified. The first one is the determination of the first object to be hit by a primary ray (emanating from a pixel). The most efficient way to do this is to perform a hidden surface removal of the scene (using the horizon method), and to maintain, for every pixel, the identity of the object that was displayed on the pixel. The next issue is handling the reflected rays. Note that every reflected ray emanates from some surface in some boxel in one of the four main directions (increasing x and y, etc.). The boxel belongs to two separate antichains, corresponding to two different antichain partitionings. Also, each antichain containing the boxel will (generally) have two neighboring antichains, so there is a total of four potential antichains to search. Clearly, the ray's direction determines which of these four antichain should be searched.

5. Experimental results

To test the performance of our method we have created synthetic scenes of cities with up to 400,000 buildings of different sizes. Each building consists of about 120 texture-mapped polygons. The algorithm was implemented in C++ with OpenGL. Occlusion testing was done with the antichain-horizon method, in which we implemented the RLE method for coding the horizon. In addition, we implemented the RLE horizon with both axis-aligned buildings and non-axis-aligned buildings. The differences in timings between the two modes was very small. We also implemented various hierarchical partitionings superimposed on the basic scene, and this produced a marked improvement. All the following methods were tested on non-axis-aligned buildings (except where noted otherwise):

1. Hierarchical boxel method, with RLE horizon.
2. Boxel method (no hierarchy), with RLE horizon, on axis-aligned buildings.
3. Quadtree method with binary tree coding for the horizon, (similar to [7]).
4. Boxel method (no hierarchy), with RLE horizon.
5. Boxel method (no hierarchy), with binary tree coding for the horizon.
6. Boxel method (no hierarchy), with column coding for the horizon.

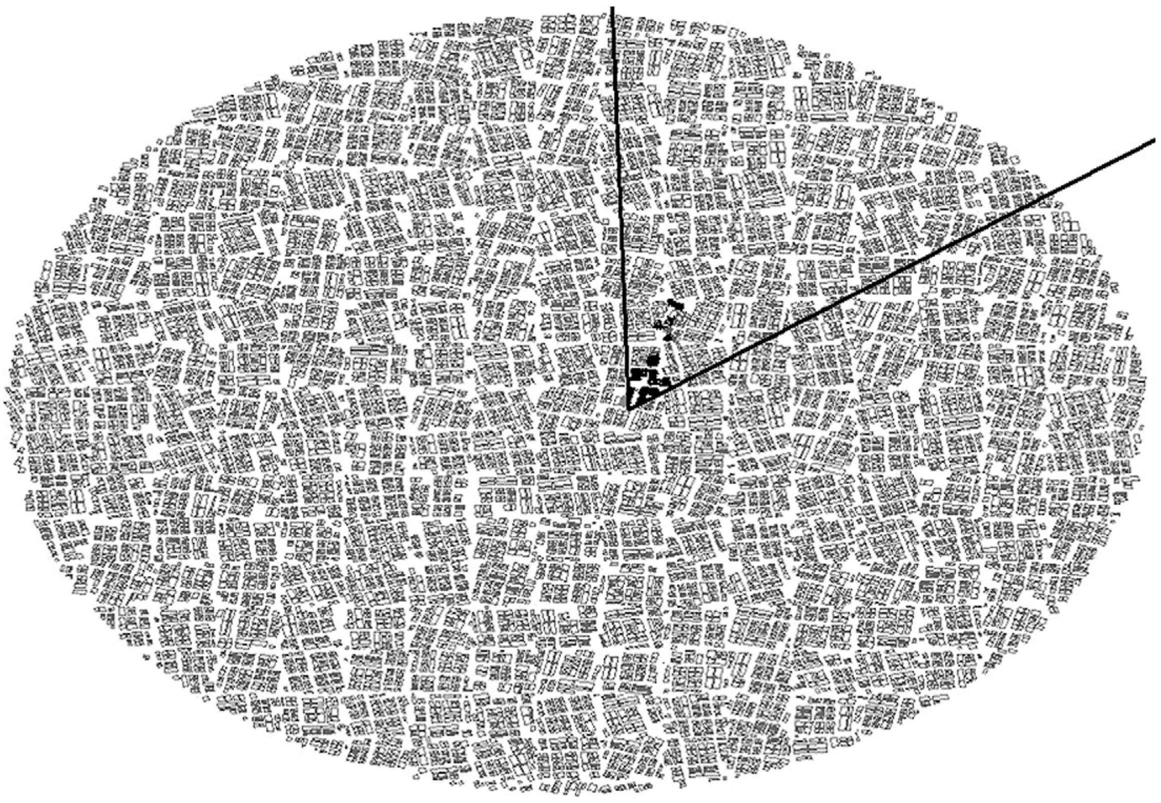

Fig. 15. Overview of a synthetic city consisting of non-axis-aligned buildings. The black lines show the field-of-view, and the black buildings are the only ones that are actually displayed by the boxel method.

In column coding, we maintain an array with one entry for every screen column. A value in the array represents, at every stage, the highest pixel that was rendered in the corresponding column. Note that with this method, occlusion is exact, so there is no need to use the hardware depth buffer. However, it turns out that it is more efficient to use conservative occlusion testing together with the hardware depth buffer. The binary tree horizon encoding was used by [7]. Instead of maintaining exact values for every column, it maintains short horizontal spans, similarly to our RLE method. However, instead of a list of spans (as in our RLE method), this method uses a binary tree for the spans.

The performance results of the various methods are based on the average number of FPS (frames per second) during a 30 second drivethrough, including several turnings. The same path was used for all the algorithms. The image quality and resolution (800) were the same for all tests. The algorithms were run on a 64-bit AMD Athlon 3500+ processor with 1 GB memory, running at 2 GHz, with an Nvidia FX-5200 graphics card with 256 MB memory. Boxel ordering and antichain partitioning on 300,000 buildings (non-axis-aligned) took about 10 seconds.

Fig. 15 shows an overview of a synthetic city, with non-axis-aligned buildings, marked in grey. The view volume is delineated by the black lines, and the rendered buildings are marked in black. It can be seen that only a very small proportion of the buildings are sent to the Z-buffer. Figs. 16 and 17 show typical scenes with axis-aligned and non-axis-aligned buildings.

Table 1 shows the average rendering rates of all the tested methods. The last two lines show the exact hierarchy at which the best times for the hierarchical boxel method with RLE horizon were obtained. Fig. 18 plots the rendering rate of the various methods in frames-per-second, for data with up to 300,000 buildings (the largest dataset with which we could achieve real-time). It can be seen that the hierarchical boxel method, combined with the RLE horizon, is significantly better than all the other methods.

6. Conclusions and future research

Boxels are a new framework for 2.5D environments, extending the concepts of octrees and their variants, and other known space partitioning methods. In common with those methods, boxels require just four different traversal orders

Table 1
Rendering rates of the various methods in frames per second

No. of buildings ×1000:	10	50	100	150	200	250	300	350	400
Occlusion method:									
Boxel hierarchy + RLE horizon	261	208.1	142.5	92.8	61.1	43.4	28.1	14.9	7.8
Boxel + RLE horizon (aligned)	244.4	74.8	40.7	25.3	16.4	15.7	9.5	7	4.5
Quadtree + tree horizon	235	6436.8	21.7	16.1	14.5	8.5	6.5	2.9	
Boxel + RLE horizon	216.9	63.3	36.7	21.4	14.4	13.4	7.8	7.3	3
Boxel + tree horizon	138.2	42.4	2212.6	9.7	7.2	2.8	1.5	1.4	
Boxel + column horizon	120.4	3618.6	11.9	6.9	4.3	1.4	1.5	1.3	
Best results: no. of levels:	1	1	2	2	3	3	2	2	2
subdivisions in each level:	2X2	3X3	2X2	2X2	2X2	2X2	3X3	3X3	3X3

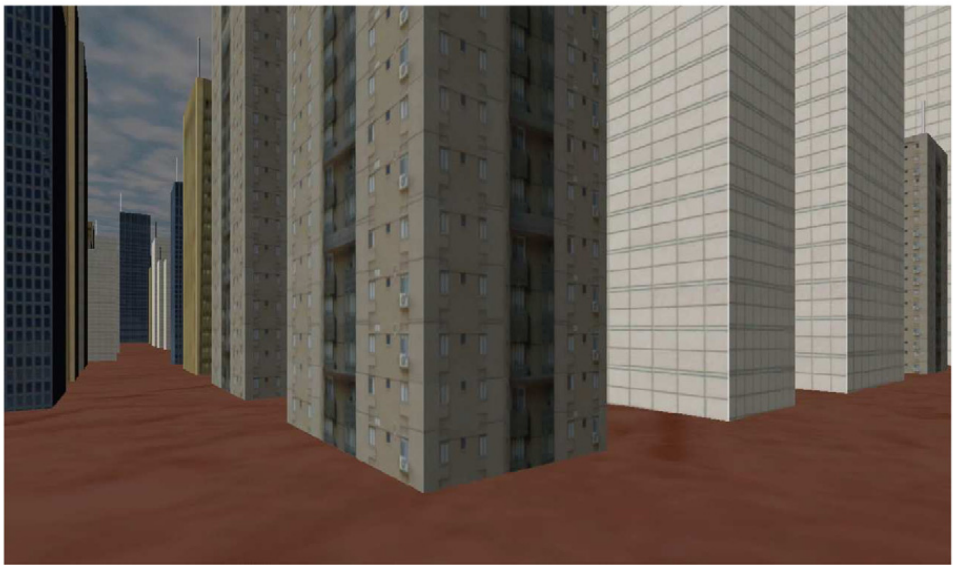

Fig. 16. Urban scene (axis-aligned).

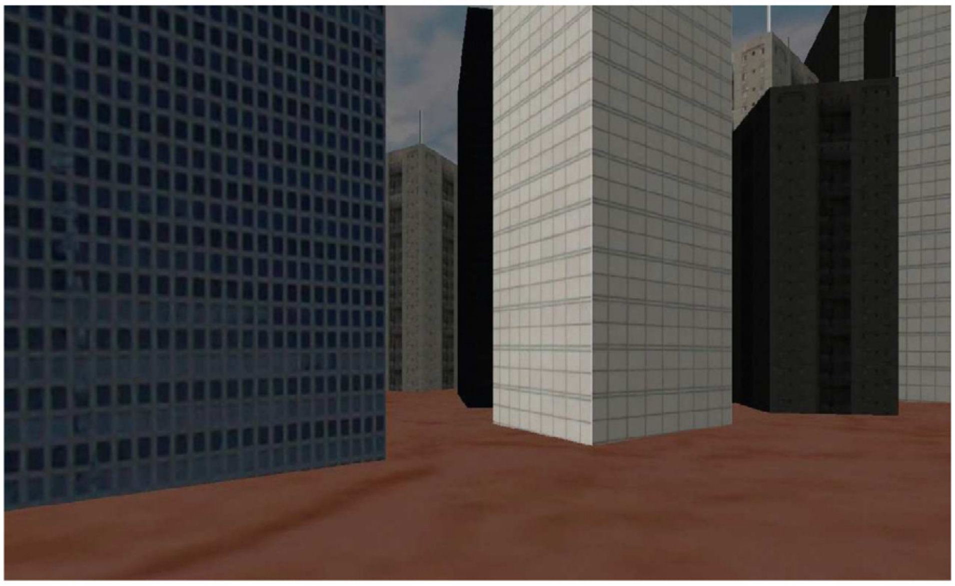

Fig. 17. Urban scene (non-axis-aligned).

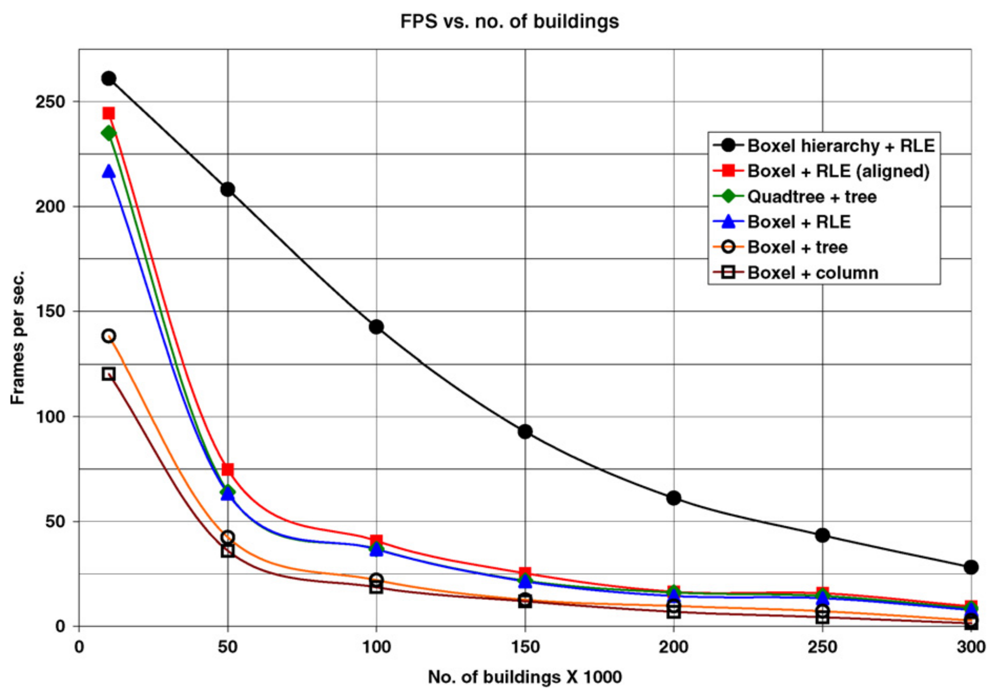

Fig. 18. Rendering rates of the various methods in frames per second.

for ray tracing and for rendering in back-to-front or front-to-back order. The advantages of boxels over other methods are that partitions are not determined a priori and boundaries do not extend throughout an entire volume or subvolume. Boxels can be bounding volumes of other objects, or they can be the basic objects of the scene.

In particular, we have shown that boxels present many advantages for the rapid rendering of urban scenes, as may be desired for virtual reality and game applications. The RLE horizon representation was shown to be the most suitable method for working with the antichain partitioning of boxels. Non-axis-aligned objects can be handled, and a hierarchical boxel structure produces marked improvements. Note that a small number of additional objects, such as avatars in a virtual environment, can be easily handled by the Z-buffer.

Future work on boxels will concentrate on further applications, enhancements and generalizations, such as:

- Image-space parallelism on a multi-core computer. This can be easily done by allowing each processor to handle a sector of the view angle.
- Implementation and testing of ray tracing using the acceleration methods described in this paper.
- Design and test algorithms for dynamic insertions and deletions of boxels. Occasional deletions are harmless, but insertions of new boxels may alter the antichain partitioning.
- Apply and test the boxel framework on a real urban scene.
- Extend the boxel framework to 3D, by dividing 3D space into "slabs", so that in each slab, the data will be 2.5D. This extension could be useful for architectural scenes, in which the partitioning into 2.5D slabs should be simple.
- Extend the lighting model to include shadows. Note that a shadow cast by a boxel in an antichain can be considered exactly like a view angle, so the antichain construction can be utilized.
- Application to radiosity: given a boxel face, the boxel data structures allow us to determine which faces of other boxels are hidden from it, and which ones face it (and are not hidden). This information can be used for the efficient computation of the form factors.

Acknowledgements

This research is based on the first author's MSc thesis in Computer Science, carried out under the second author's supervision at the University of Haifa. Thanks are due to the Caesarea Rothschild Institute for its generous support. This research was also supported in part by Grant No. 01-01-01509 from the Israel Ministry of Science and Technology. The authors wish to thank Ilan Newman for helpful discussions concerning the probabilistic model. Thanks are also due to the anonymous referees for their helpful comments.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.
[2] T. Asano, M. de Berg, O. Cheong, H. Everett, H. Haverkort, N. Katoh, A. Wolff, Optimal spanners for axis-aligned rectangles, Computational Geometry: Theory and Applications 30 (1) (2005) 59–77.
[3] T. Asano, M. Edahiro, H. Imai, M. Iri, K. Murota, Practical use of bucketing techniques in computational geometry, in: G.T. Toussaint (Ed.), in: Computational Geometry, Machine Intelligence and Pattern Recognition, vol. 2, Elsevier, Amsterdam, 1985, pp. 153–195.
[4] S. Bellantoni, I. Ben-Arroyo Hartman, T. Przytycka, S. Whitesides, Grid intersection graphs and boxicity, Discrete Mathematics 114 (1993) 41–49.
[5] F. Cazals, P. Puech, Bucket-like partitioning data structures with applications to ray-tracing, in: Proceedings 13th Ann. Symp. on Computational Geometry, Annual Conference, Nice, France, ACM, 1997, pp. 11–20.
[6] D. Cohen-Or, Y. Chrysanthou, C. Silva, F. Durand, Survey of visibility for walkthrough applications, IEEE Trans. on Visualization and Computer Graphics 9 (3) (2003) 412–431.
[7] L. Downs, T. Moller, C.H. Sequin, Occlusion horizons for driving through urban scenery, in: Proc. ACM Symposium on Interactive 3D Graphics, Berkeley, CA, 2001, pp. 121–124.
[8] J.D. Foley, A. van Dam, S.K. Feiner, J. Hughes, Computer Graphics: Principles and Practice, second ed., Addison-Wesley, Reading, MA, 1990.
[9] G. Frieder, D. Gordon, R.A. Reynolds, Back-to-front display of voxel-based objects, IEEE Computer Graphics & Applications 5 (1) (1985) 52–60.
[10] A.S. Glassner (Ed.), An Introduction to Ray Tracing, Academic Press, London, 1989.
[11] D. Gordon, Eliminating the flag in threaded binary search trees, Information Processing Letters 23 (1986) 209–214.
[12] D. Gordon, Boxels: A new framework for 2.5D and 3D objects, in: Proc. Israel–Korea Conf. on New Themes in Computerized Geometrical Modeling, Tel-Aviv, Israel, Feb. 1998, pp. 223–226.
[13] D. Gordon, Theoretical foundations of boxels—a new framework for 2.5D and 3D objects, Technical report, Dept. of Computer Science, The Technion, December 1998.
[14] D. Gordon, S. Chen, Front-to-back display of BSP trees, IEEE Computer Graphics & Applications 11 (5) (1991) 79–85.
[15] L.J. Guibas, F.F. Yao, On translating a set of rectangles, Advances in Computing Research 1 (1983) 61–77; Also in: 12th ACM STOC, 1980, pp. 154–160.
[16] A. Kaufman, Volume Visualization, IEEE Computer Society Press, Los Alamitos, CA, 1991.
[17] U. Manber, Introduction to Algorithms, Addison-Wesley, Reading, MA, 1989.
[18] D.J. Meagher, Efficient synthetic image generation of arbitrary 3D objects, in: Proceedings IEEE Computer Society Conference on Pattern Recognition and Image Processing, Annual Conference, IEEE Computer Society Press, June, 1982, pp. 473–478.
[19] D.J. Meagher, Geometric modelling using octree encoding, Computer Graphics & Image Processing 19 (1982) 129–147.
[20] F.P. Preparata, M.I. Shamos, Computational Geometry, Springer-Verlag, New York, 1985.
[21] R.A. Reynolds, D. Gordon, L.-S. Chen, A dynamic screen technique for shaded graphics display of slice-represented objects, Computer Vision, Graphics & Image Processing 38 (3) (1987) 275–298.
[22] H. Samet, Applications of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.
[23] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.
[24] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan-Kaufmann, San Francisco, CA, 2006.