KINETIC VERTICAL DECOMPOSITION TREES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

João Luiz Dihl Comba
September 1999

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Leonidas J. Guibas (Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Patrick Hanrahan

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jean-Claude Latombe

Approved for the University Committee on Graduate Studies:

To my wife, Ângela, my son Alexandre, my mother Gilda, and my brothers Pedro and Luiz Antônio.

Abstract

This thesis presents a new structure called the Kinetic Vertical Decomposition Tree (KVD-tree), used for the dynamic maintenance of visibility information for a set of moving objects in space. Visibility information is important in many applications, including graphical rendering, animation, motion planning, etc. Yet visibility is expensive to compute and difficult to update as the objects in the environment move and occlude each other and the observer(s).

The KVD-tree is a single structure that not only (1) allows dynamic maintenance of visibility, but also (2) represents a vertical decomposition of the space, (3) allows collision detection among moving objects, and (4) it is kinetically maintained based on the kinetic data structures framework.

In terms of structure, the KVD-tree is a special type of Binary Space Partition tree (BSP-tree), a hierarchical data structure commonly used in solid modeling and computer graphics for feature classification and visibility determination. For a scene composed of polygonal objects, the BSP-tree corresponds to a hierarchy of binary partitions of space using the hyperplanes that support the faces of the polygons as partitioners. In the KVD-tree, additional cuts are introduced from edges and vertices, so that a vertical decomposition induced by the polygonal model is formed. The bounded complexity of the cells in this decomposition allows the creation of certificates that indicate times when the movement of objects causes a change in the decomposition. These certificates are used within the framework of kinetic data structures to identify when the structure of the KVD-tree changes. The update of the KVD-tree involves a series of local changes in the tree, accomplished by special update algorithms. The certificates can be used to detect collisions of objects in the scene, which can then be avoided by providing appropriate actions to the update algorithms.

Acknowledgements

Leo Guibas was my advisor here at Stanford. In many ways Leo was able to contribute to my formation. His deep understanding of geometric algorithms, computer graphics and related areas were shared with me in many situations. As an advisor, his guidance was fundamental throughout many crucial points along the way. I am deeply thankful for all that.

Jorge Stolfi was one of the main reasons I came to Stanford. Jorge invited me to spend a summer at Digital SRC, where I met Leo and later came to Stanford. Jorge played a big role in convincing me to come to the United States and later assisted me in many occasions during my stay at Stanford.

Jean-Claude Latombe and Pat Hanrahan served in the reading committee of my thesis. Jean-Claude shared his enthusiasm, and offered many useful suggestions on how to improve this work. Pat was always a receptive person, willing to discuss new problems, and his graphics experience was extremely helpful. I also would like to thank Chris Bregler and Stephen Boyd, who participated in my thesis committee. I thank agency CNPq from Brazil for support of part of this work through process 200789/92.

I was fortunate to participate in many geometry lunches we had throughout the years. The geometry lunch was designed by Leo to involve his group in the study and discussion of selected papers during a school quarter. For me, the geometry lunch was a way to deeply study selected topics in areas concerning geometric algorithms and graphics. The experience of preparing and presenting talks in a regular basis was extremely useful to me. On the other hand, the graphics lunches (meetings of the graphics group) helped me to keep me up-to-date with the exciting research being developed in the graphics lab, as well as the research of many invited speakers.

I had many friends during my stay at Stanford, and I separate them in two categories: friends from Brazil, and not from Brazil. Friends from Brazil include many people I met here at Stanford or in the US. Claudionor and Carla were good friends, and helped us start life here at Stanford. Alexandre Santoro was a great friend, and together we had espresso cups enough to fill a swimming pool. Ricardo Tarabini and Marcela shared their *carioca* style. Paulo Cruz and family, Heitor and family were truly good friends we had. Luiz Franca and Jorge Campello shared with me their love for the game of soccer and great friendship. Marcus Vinicius was a close friend in his short stay at Stanford. His positive attitude towards life was a great example I learned from him. Claudio Silva was a good friend I met here in the US. Although Claudio lives in the east coast, we were able to maintain a good friendship while only seeing each other at Siggraph conferences. My friends not from Brazil were more present into my life in my final years at Stanford. Scott Cohen was a very helpful and supportive friend. From him, I learned several lessons on how to give a good seminar. Olaf Hall-Holt was a great office mate and friend. He helped reviewing this text, with many helpful suggestions on how I could improve the final result. Murali shared his knowledge of BSPs with me. Everybody in Leo's group was kind and very supportive. The people in other groups, graphics and robotics, created a nice atmosphere to work. I specially thank Jutta McCormick for her always efficient management of administrative questions and friendship.

I thank the Brazil soccer team for the world cup 94, which I was able to be a part of it going to all games played at Stanford. My soccer team Gremio was always followed by me and I thank the development of the internet for the chance of listening in real time soccer games from Brazil. My work here involved long hours of debugging code, which involved in some cases several levels of recursive code. Music made me company during these tasks, specially songs from Sarah Mclachlan, Heather Nova and Beth Orton (among many others).

I would not be able to finish this dissertation if it were not for the support of my family. My mother and brothers gave much support through some hard times. Above all, my wife Ângela and my son Alexandre were fundamental during my stay here at Stanford. They loved me so much during this time, showed support and encouragement that I have to consider myself very fortunate, and I deeply thank them from the bottom of my heart.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Visibility is a central notion to a number of disciplines dealing with the computer modeling of the physical world. In the field of Computer Graphics, for instance, visibility information is used to compute parts of objects that are visible from a viewpoint (into an image plane). Historically, computer graphics was the first branch of computer science to have faced the visibility problem, which gave rise to the famous *hidden-surface elimination* problem [24].

The existence of many different solutions to the hidden-surface elimination problem illustrates the fact that there are many ways to approach it, with many tradeoffs to be considered, depending on the application. For real-time visualization, for instance, high frame-rates may be achieved by reducing the amount of realism in each picture generated. In other situations, like a computer generated movie picture, the need for high realism drastically slows down the generation of images. Both examples illustrate the tradeoff between quality (realism) and speed necessary to generate each image. In the design of visibility algorithms, these tradeoffs need to be taken into consideration.

There are important properties that are common to many of the visibility algorithms. *Sorting*, for instance, is an example of such a common notion. The possibility of having one object occlude another (with respect to a viewer), illustrates the fact that the distance to the viewer can be used to define a sorting order. Intuitively, such an ordering (also called a *visibility ordering* or *depth ordering*) can be used to render objects in a far-to-near

approach, where objects far from the viewer are drawn before near objects. This process is also called the *painter's algorithm*[20], because it simulates to some extent the order of actions performed by a painter when creating a picture. The efficiency of the visibility computation depends on how fast the visibility ordering can be obtained.

A second important notion applicable to visibility problems is *coherence*, which represents a tendency for some properties of a given problem to be locally constant. *Image coherence*, for instance, describes the fact that neighbor pixels in the image space are likely to have similar colors. The notion of *object coherence* pertains to the likelihood of close points in object space to belong to the same object. In figure 1.1 the different types of coherence are illustrated. Visibility algorithms exploit these types of coherence, either separately or together, to reduce the computational effort. Algorithms that exploit only image space coherence are classified as image-space algorithms, while algorithms that use object space coherence are called object-space algorithms.

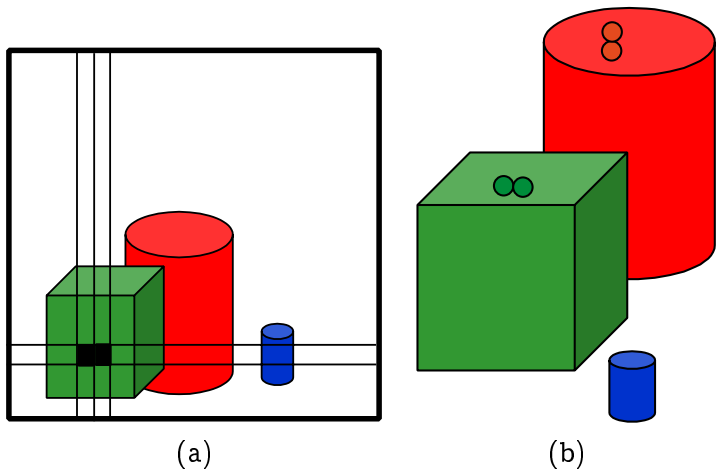

Figure 1.1: Coherence information. (a) Image-space. (b) Object-Space.

The simple formulation of the visibility problem allows only static objects and a single static viewpoint position. Generalizing to a moving viewpoint is the easiest extension that we might consider. In this case, the representation of the world remains static, but the algorithms that extract visibility information need to take into account the movement of the viewpoint. The fully general problem happens when both objects and the viewpoint

move. In this case, the representation of objects needs to be updated every time the objects move, which becomes a challenging task when combined with the extraction of visibility information in an efficient manner.

Motion problems typically possess another type of coherence called *temporal coherence*. In a dynamic scene, objects usually move along continuous paths and the visibility ordering computed from a previous frame is likely to stay the same in a succeeding frame, with a small expected number of changes. In figure 1.2 we illustrate a small example with moving objects, to illustrate some changes in visibility relationships caused by objects occluding or being occluded by other objects.

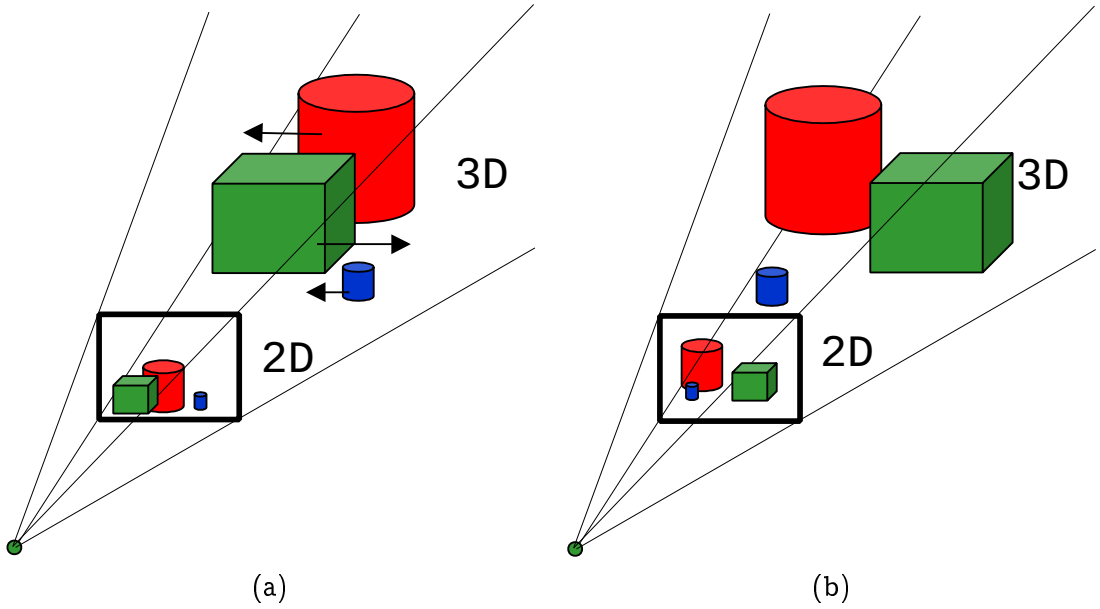

Figure 1.2: Dynamic Visibility Problem. (a) sample scene (b) changes in scene due to the movement of objects.

The challenge here is to design a data structure that can quickly extract visibility information, while using as much temporal coherence as possible. In this work, we solve this problem by first considering a data structure that can be used to extract visibility information for static scenes, and propose techniques to update this structure. We use as much as temporal coherence as possible. In order to do this, we change the problem of maintaining dynamic visibility information into a similar problem of maintaining a data

structure that can be used to extract visibility information in a dynamic scenario.

This new data structure is called the *Kinetic Vertical Decomposition Tree* (KVD-tree or simply KVD). The KVD is a special type of binary space partition tree (BSP tree or just BSP). The BSP is a data structure commonly used in solid modeling and computer graphics for feature classification and visibility determination. The BSP is in essence a static structure, although some approaches to create dynamic BSPs have been explored in the literature ([17][26][7]).

In this work we choose to use the framework for designing *kinetic data structures (KDS)* proposed in [4]. KDS's are specifically designed for continuously moving objects, and we apply that framework to the problem of dynamic maintenance of a BSP. Similar approaches have been considered in the literature [2] [1], mainly from a theoretical point of view. Our approach here is grounded in practice, in other words, the KVD corresponds to a fully implemented 3D kinetic BSP.

This work is motivated by the observation that many changes in the BSP are local, and therefore require updates in just a few places of the tree. The design and implementation of a kinetic BSP structure that has local update algorithms and has a set of certificates that identify combinatorial changes in the tree is the major contribution of this thesis.

In this chapter we review the fundamental concepts involving BSPs and Kinetic Data Structures framework. An informal presentation of the KVD tree is given next. We conclude the chapter with a summary of the contributions of this thesis and the organization of the text.

1.2 Binary Space Partitioning Trees (BSPs)

Binary Space Partitioning trees (BSP-Trees) are spatial search structures used in many different aspects of Computer Graphics and Geometric Modeling. Applications in solid modeling [14] [16] [25] [27], visibility orderings [10] [11] [26] and image representations [18] , among others, can be found in the literature. In order to describe the concepts of BSP-Trees, it is always nice to first explain the relation it has with Binary Search Trees.

Binary Search Trees have been used in Computer Science in many ways, but mostly as a data structure to accelerate search queries based in symbolic values. A geometric

interpretation of this data structure is as a hierarchy of binary partitions of the real line (see figure 1.3), where the partitioner is a point and each partition obtained represents an interval.

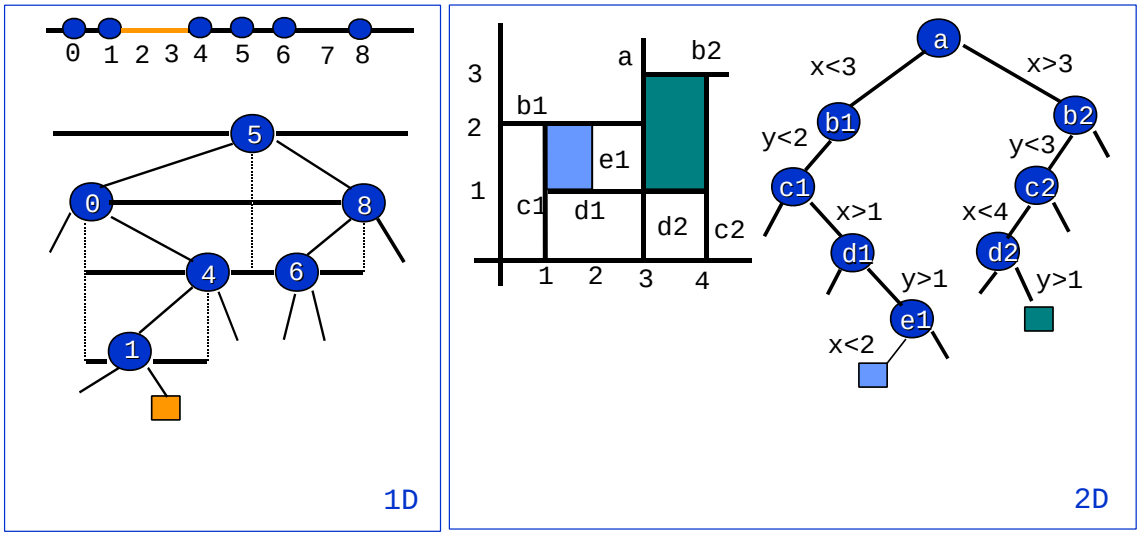

Figure 1.3: BSPs in 1D and 2D

The problem with this interpretation is that it does not appear to directly generalize to higher dimensions, as points do not partition higher dimensional spaces. A useful generalization can be obtained when the partitioner is in fact a hyperplane, rather than a point. In general, for a d-dimensional space, the partitioner corresponds to a $d-1$ hyperplane, and the partition has the same dimension of the underlying space. For example, in two dimensions the partitioner is a line, and in three dimensions it is a plane. BSP-Trees and Partition Trees [12] use this analogy to extend the concepts of Binary Search Trees to higher dimensional spaces. One of the great advantages of BSPs is its natural ability to combine a search structure with a data structure that can represent a polygonal object. This property motivates the use of BSPs in solid modeling applications.

Visibility information can be extracted from the BSP in a very simple way, and we illustrate this with an example in 2D (figure 1.4).

In this case, the input set consists of line segments, which are used to create the partitioner or cuts in the BSP. At the nodes of the BSP we store the equation of the line segment that defines the corresponding cut. We also store the intersection of the defining segment

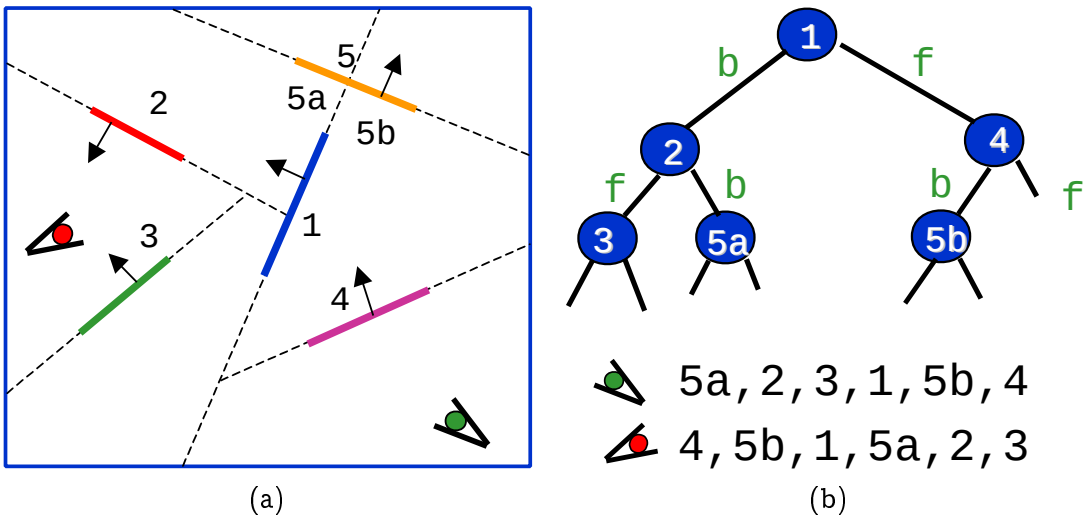

Figure 1.4: Extracting visibility from the BSP (a) BSP subdivision (b) BSP tree and visibility ordering for two viewpoints.

with the current partition of the BSP, which we call the segment fragment. In the figure 1.4, segment 5 generates two nodes in the BSP, each with the proper segment fragment.

For any given viewpoint, a visibility ordering can be obtained by performing a back-to-front traversal of the BSP, as follows. We traverse the BSP tree one node at a time, beginning at the root. For every node, the subtree that does not contain the viewpoint is visited first, followed by the node itself, finally the subtree that contains the viewpoint is visited. The resulting visibility order for two example viewpoints is illustrated in the figure 1.4. The complexity of this operation is clearly proportional to the number of nodes in the tree.

1.3 Kinetic Data Structures

Data structures are fundamental to the study of algorithms. Often they involve a delicate compromise between different operations on data, each of which may be easy to implement in isolation, while their union generates conflicting requirements. Usually this conflict arises between operations which produce a desired attribute of the data, say the largest value among a set of numbers, and those that update the data, say by adding or deleting

numbers from the set. Data structures supporting insertions and deletions of objects are referred to as *dynamic*. In this work we focus instead on *kinetic* data structures (KDS for short), in which we want to maintain the attribute of interest under continuous changes in the data. This framework for data structure design was proposed in [3], where the attributes to be maintained are named *configuration functions*. In the context of this work, the attribute we maintain is a representation of the combinatorial structure of a BSP, which is used to extract the visibility information among the objects.

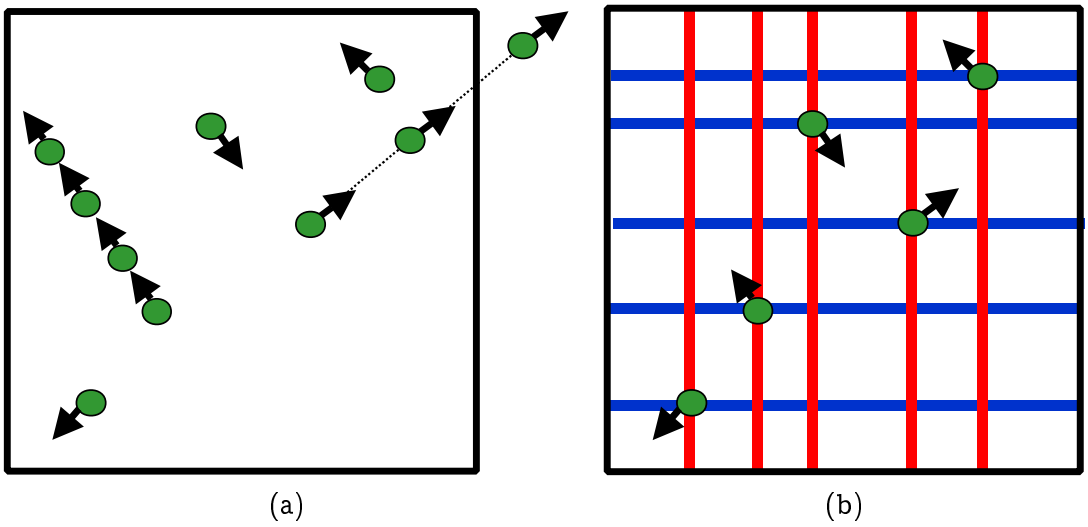

Figure 1.5: Moving balls inside a rectangle. (a) Fine sampling may lead to unnecessary work, while coarse sampling may miss some events. (b) Certificates that serve as a proof that the balls stay inside the rectangle

Here we briefly review some of the key issues regarding kinetic data structures. Consider the example in figure 1.5, where we want to maintain a set of moving balls inside a rectangle at all times, while bouncing the balls from the walls whenever they try to escape. Suppose that each ball has a posted *flight plan* that gives full or partial information about its current motion. A simple approach to animating this scenes would be to test at frequent intervals of time whether a ball leaves the rectangle. This may lead to unnecessary work, because most of the times the balls are inside the rectangle and no flight plan update is necessary. On the other hand, if we sample the movement the balls too coarsely we may miss critical times when balls leave the rectangle.

A better approach for our purposes is as follows. We maintain for each ball the vertical and horizontal distances to each one of the walls. Because we have the flight plan for each ball, we are able to precisely calculate the event times the balls hit the rectangle. The enumeration of a set of conditions that can be used to prove that a combinatorial structure correctly describes the current situation, combined with a mechanism that computes event times when the structure changes, form the foundation of the kinetic data structure framework.

For general problems, a flight plan *update* can occur because of interactions between an object and other moving objects, the environment, etc. For example, a collision between two moving objects will in general result in updates to the flight plans of both objects. The interface between the kinetic data structures and the object motions is through a global event queue. A key aspect is that we have special definition for motion. What we mean by this is that the kinds of events we have in the event queue correspond to possible combinatorial changes involving a constant (and typically very small) number of objects each. For example, in the case of visibility maintenance, one type of event we might use is "the points A and B become occluded by edge e of opaque triangle T". Indeed, it will turn out that the correctness of whatever configuration function we maintain can be guaranteed by a conjunction of similar low-degree algebraic sign conditions, each involving a bounded number of objects each, We call these conjunctions the *certificates* of the KDS.

At any one time, the event queue will contain several KDS events corresponding to future times when certificates might change sign. The times for these events are calculated using the posted flight plans of the objects involved. If, because of other events, the flight plan of an object is updated, then all certificates involving that object must be located and have their 'sign change' time recalculated according to the new plan. In this way the event queue adapts to the evolving motions of the objects. In general the approach taken is that each moving object needs to be aware of all the events in the event queue that involve it and the validity assumptions about its motion on which these events are based. If the motion of the object changes so that any of these assumptions is no longer valid, then it is the responsibility of the object to take the steps necessary to have these events rescheduled at the times appropriate for its new motion.

To summarize, kinetic data structures are different from classical dynamic data structures: though we can (and often want to) accommodate insertions and deletions, our focus is on continuous motions, rather than arbritrary modifications. Furthermore, the structures are on-line and can be used to implement correct simulations even when the object flight plans change because of interactions between the objects themselves or the objects and their environment, or even when only partial information about the motions is available.

1.4 Kinetic Vertical Decomposition Trees

The advantage of the kinetic data structure framework to this problem is that it focuses our attention to those spatial relationships between the data in the scene which are crucial in forming the BSP. Thus, as long as the certificates defining the KDS do not change, the current BSP stays valid combinatorially — even though objects may have moved in the meantime. When one of the certificates does fail, the continuity of the motion which caused it to fail makes it likely that the required change in the BSP will be a local one. These local structural changes to the BSP are operations akin to rotations in classical binary search trees [9], though of course more complex: BSPs represent multidimensional data, so that each node may point to lower-dimensional structures, etc.

From the point of view of kinetization, it seems advantageous to form a BSP by using cuts which are as uniform as possible — for example, for a set of non-intersecting lines in two dimensions, an approach where all cuts are parallel to a given axis (except cuts through lines) has been suggested [19]. The decomposition created in this case corresponds to the vertical decomposition of the set of lines. The advantage of such uniform cuts is that the combinatorial changes in the structure of the tree happen at specific times when the parallel cuts cross each other. The extension to three dimensions – for example, for a set of non-intersecting triangles, can be done in a similar way using the three-dimensional notion of vertical decompositions. As with several other problems, the extension to a higher dimension creates many more complex cases, but it is still possible to define a fixed number of critical events.

The structure we propose is called the *Kinetic Vertical Decomposition Tree* (KVD). It is a special type of BSP that represents the vertical decomposition for a set of triangles

in $\mathbb{R}^3$. In the KVD, additional cuts are introduced from vertices and edges along specified directions. In some cases, the viewpoint or light source is used in the specification of these directions, allowing recovery of view-dependent (or light-source) visibility. The update of the KVD involves a series of local changes in the tree, accomplished by special update algorithms. The certificates of the KVD can be used to detect collisions of objects in the scene. These collisions can be prevented by assigning appropriate actions to the update algorithms.

In summary, the KVD is a single structure that has the following characteristics:

1. Hierarchical representation of a dynamic vertical decomposition.
2. Dynamic maintenance of visibility ordering.
3. Viewpoint or light source dependent visibility information available.
4. Certificates detect combinatorial changes.
5. Collision detection among moving objects built-in in certificate structure.
6. Local update algorithms.

1.5 Organization of dissertation

The text is organized as follows. In chapter 2 we review the main properties of BSPs, and discuss some new techniques to help in its visualization. In chapter 3 we review previous work on dynamic and kinetic BSPs. In chapter 4 we present a general introduction to Kinetic Vertical Decomposition Trees, and discuss the main characteristics of this structure. The next three chapters describe in detail important parts of the KVD. In chapter 5 we discuss issues regarding its representation and construction. In chapter 6, we discuss the set of events used to detect combinatorial changes. Chapter 7 is devoted to the algorithms used to perform structural updates in the tree. We evaluate the performance of the KVD and discuss some of these results in chapter 8. We conclude with chapter 9, where we discuss the main contributions of the thesis, and finish with a presentation of future directions and work.

Chapter 2

BSP Concepts

2.1 Definitions

BSP algorithms involve many concepts and definitions, which for clarity and terminology are presented in detail in this section. Let us consider a scene in d-dimensional space. The partition of this space is done by means of a *hyperplane*, described by the following equation:

$$h \equiv \{(x_1, ..., x_d) \mid a_1x_1 + ... + a_dx_d + a_{d+1} = 0\} \tag{2.1}$$

The hyperplane separates the space in two *halfspaces*, the positive and the negative halfspace. As a convention, the positive halfspace is also called the *IN halfspace*, while the negative halfspace is called the *OUT halfspace*. The *normal* of the hyperplane is defined by the vector $(a_1, a_2, ..., a_d)$.

A *BSP node* contains data describing the binary partition being performed on the space. It consists of a partitioning hyperplane, node-specific information and left and right children that point to BSP representations of the positive and negative halfspaces. The hyperplane that defines the node n is denoted by $h(n)$. A *BSP* leaf contains attributes associated with a given region. Many attributes may be stored at the leaves, like whether the region is part of a solid, its color, density, etc. We call the special case of BSPs with solidity attributes in the leaves of the tree by a *solid BSP*.

A *region path* of a node corresponds to the path through the tree that leads up to the root of the tree. This path consists of all ancestors of the node in the tree and is represented by an ordered list of nodes L. A *region* of a given node represents the geometric interpretation of the partition defined by the region path *RP(n)*. It corresponds to the intersection of all halfspaces in the region path.

In polygonal models, the planes that support the faces of the model are used to define cuts in the tree. During the construction of the BSP, the insertion of a cut corresponding to a face is done by locating the leaves in the BSP that contain the face. For each leaf node reached, a new node is created, containing the *fragment* of the face that survived to that particular location. Together, all nodes created from a given face contain fragments of the original face, and their union reconstructs the face.

The visibility ordering extracted from the BSP is given by the enumeration, in back-to-front ordering, of the fragments stored in the nodes of the tree. In figure 2.1 we illustrate some of the concepts we have seen so far.

The *auto-partition BSP* is a special type of BSP where the choice of cutting hyperplanes used during the construction of the BSP is limited to planes that support the faces of the input model, as in the example above. The main advantage of this limitation on cuts is that the set of partitioner planes is minimal. For convex objects, however, the use of auto-partition cuts results in a tree of linear depth, which follows directly from the fact that no splitting of fragments happens. In this case, operations that have running time proportional to the depth of the tree may become expensive. One solution to reduce the depth of the tree is to use external cuts to extend the set of auto-partition cuts. The price to be paid in this case is related to the appearance of splitting operations that partition the nodes and increase the overall number of nodes in the tree. We usually use the term BSP for the type of BSP that has no restrictions on the types of cuts, allowing use of external cuts if necessary.

2.2 BSP Operations

BSP algorithms range from numerical procedures (classifications, partition of fragments) to tree specific procedures (construction, merging and partition of trees). In this section we

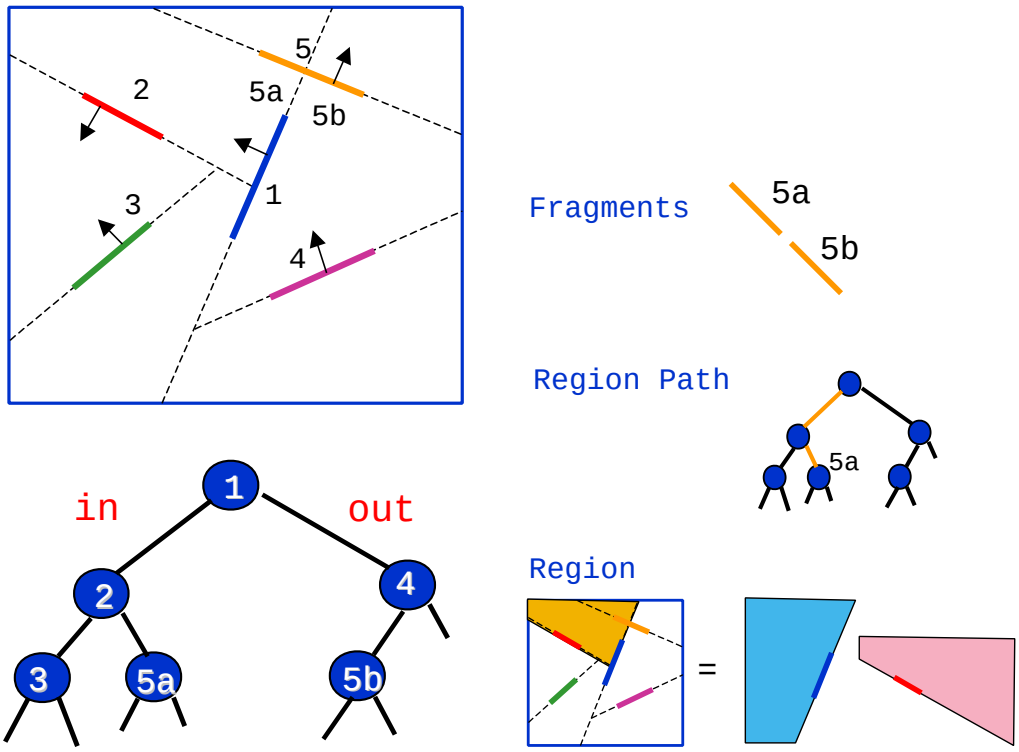

Figure 2.1: BSP Concepts. Sample subdivision, the corresponding tree representation, fragments, a region path and the corresponding region (in yellow).

explain in detail all these operations and provide several examples.

2.2.1 Classification

Classification methods compute the spatial relationship between fragments and hyperplanes. In the construction algorithm, for instance, the insertion of an element in the tree uses this operation to find which halfspaces of the root node are occupied by the the element. If the element crosses the node hyperplane, a partition of the element into fragments is done and the process continues with the fragments in the subtrees of the node. On the other hand, if an element belongs entirely to one of the halfspaces no partition is required and the insertion continues in just one of the subtrees. The classification operation returns a code that represents this relationship between the node in the tree and the inserted

fragment.

Let n_h and n_s represent two BSP nodes. The classification of n_s against n_h ($cl(n_s, n_h)$) returns one of six results:

- **CL_IN**: If $frag(n_s)$ is contained in $h^+(n_h)$
- **CL_OUT**: If $frag(n_s)$ is contained in $h^-(n_h)$
- **CL_ON_IN**: If $frag(n_s)$ is contained in $h(n_h)$, with normals facing IN halfspace.
- **CL_ON_OUT**: If $frag(n_s)$ is contained in $h(n_h)$ with normals facing OUT halfspace.
- **CL_ON**: If $frag(n_s)$ is contained in $h(n_h)$ (when a normal is not defined for $frag(n_s)$).
- **CL_CROSS**: If $frag(n_s)$ belongs both to $h^+(n_h)$ and $h^-(n_h)$.

In order to implement this operation we compute the dot product of each point in the fragment $frag(n_s)$ against the hyperplane $h(n_h)$, as the resulting sign represents the halfspace in which the point is located. The classification types are illustrated in figure 2.2.

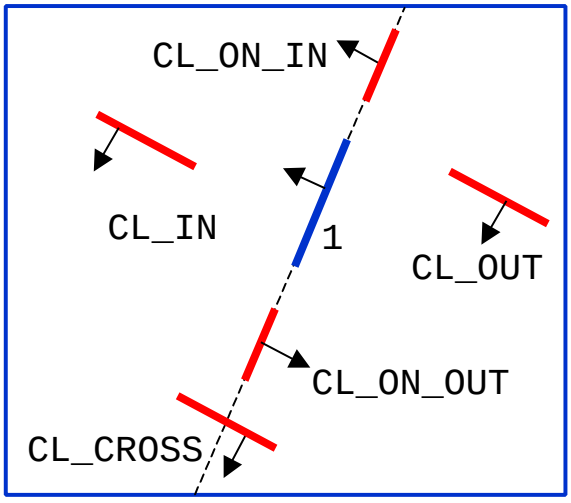

Figure 2.2: Possible results for the classification operation.

2.2.2 Partition of a fragment

The fragments are used to control the insertion of cuts in the BSP. For simplicity, fragments are stored as an ordered list of vertices, and the edges of the fragment are defined by consecutive vertices in this list. In order to decide which halfspaces are occupied by a given edge we use the classification operation described above. If any edge fragment crosses the hyperplane, a partition is required to create the corresponding fragments in each one of the halfspaces. In figure 2.3 we illustrate some cases of fragment partitions.

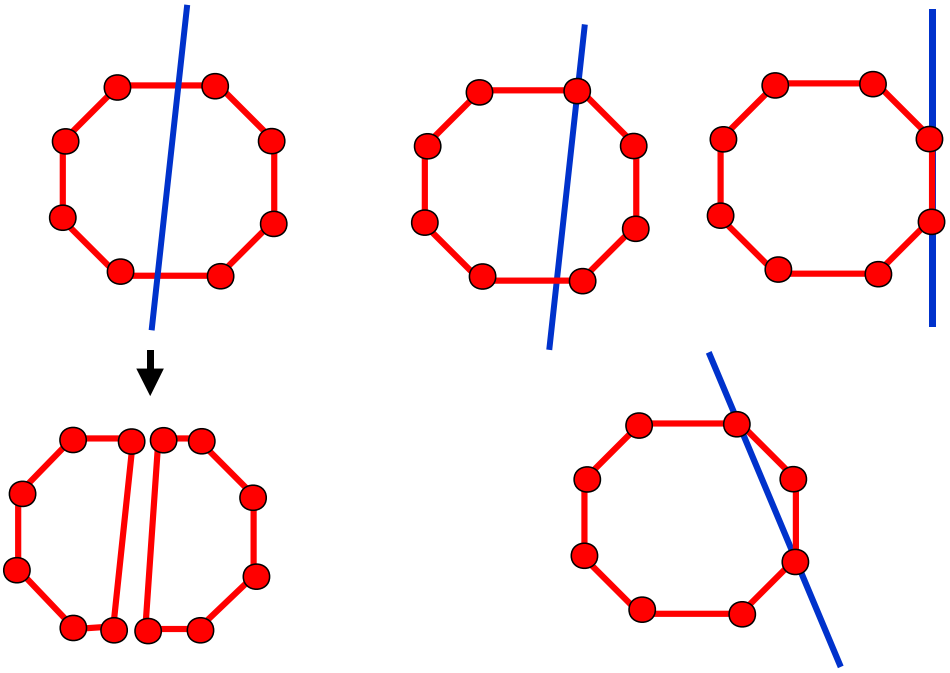

Figure 2.3: Partition of a fragment. Points of the fragment are classified in order against the partitioner. Special cases where a point lies in the partitioner may create special situations in the partition algorithm.

The algorithm to partition a fragment by a hyperplane performs a classification operation for every vertex of the fragment. The results of the classification operations are evaluated in order to decide whether an edge of the fragment crossed the hyperplane. Two lists of vertices, containing points in IN and OUT halfspaces are created in this process. The fragment list is an implicit representation of the edges, and every time two consecutive

points have different classification results an action must be taken. In the simple case, where the points are in both halfspaces of the cut (one classification is CL_IN and the other CL_OUT - or vice-versa), an intersection vertex is created and inserted in both IN and OUT lists.

The possibility of a point lying exactly on the partitioner creates an additional difficulty. Assuming the numerical precision of the classification operation for this case is sufficient, we wait until another point is found whose classification is different than CL_ON. This new classification result is compared against the last classification result obtained before the CL_ON was received. If the classification results are different, we insert the CL_ON vertex in both lists. Otherwise, we insert the point in only one of the lists, the one indicated by the last classification result.

2.2.3 Partition of a BSP

The partition operation can be extended to objects represented by BSPs as well. The result of this operation creates two new trees, corresponding to the BSPs in the positive and negative halfspaces of the partitioner. In figure 2.4 we illustrate a simple example of this partition operation.

The classification operation is used to decide in which of the output trees each node of the original tree belongs. The fragments associated with each node are used in this process. For the case that a fragment is completely inside one of the halfspaces of the partitioner, the node is inserted in only the the output tree of the corresponding halfspace. For the case of a fragment that crosses the partitioner, a partition of the fragment is done using the partition operation described before, and two nodes are inserted with the proper fragments in the output trees.

The algorithm that creates the partition performs a traversal in the tree, computing the classification of the fragment of each node. Depending on the result, nodes are inserted in the proper trees. In the example of figure 2.4 we have a simple case of a BSP composed of three cuts. The first node to be tested, node a, is contained in one of the halfspaces and only inserted in one of the subtrees. The traversal continues in node b, which is fragment in two nodes, b_1 and b_2, each inserted in one of the output trees. Similar behavior is observed for the last node in the tree.

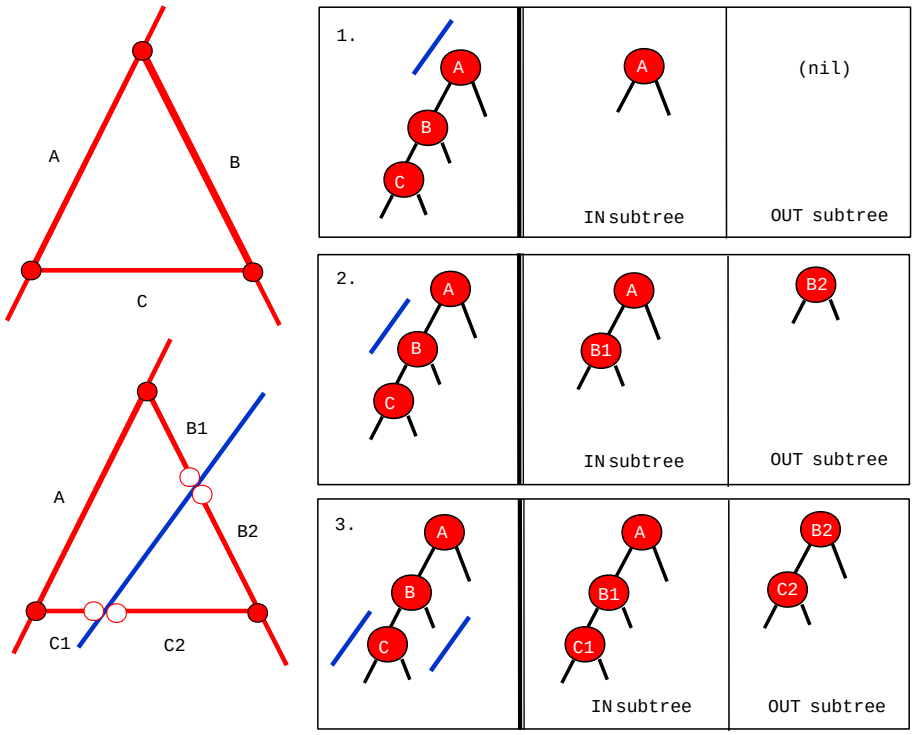

Figure 2.4: Partition of a BSP.

2.2.4 Merging Operation

The merging of two BSPs is a very powerful operation used to combine two BSP representations. If we consider each BSP as a representation scheme, it becomes natural to define an algebra that allows us to combine different BSP representation. In solid modeling, for instance, the merging of BSPs is used to create a new BSP that results of a boolean combination of the BSPs (union, intersection or difference). In figure 2.5 we illustrate the merging of two BSPs corresponding to their union.

The simple way to combine BSPs is to keep one of the BSPs static, while inserting the second tree into it. Because the cells of the BSP may contain attributes, such as solidity or color, it is important to maintain the structure of the tree to be inserted, rather than inserting one node at time. The merging operation is accomplished by a recursive algorithm that performs a traversal of the static tree. For each visited node in the static tree, we use

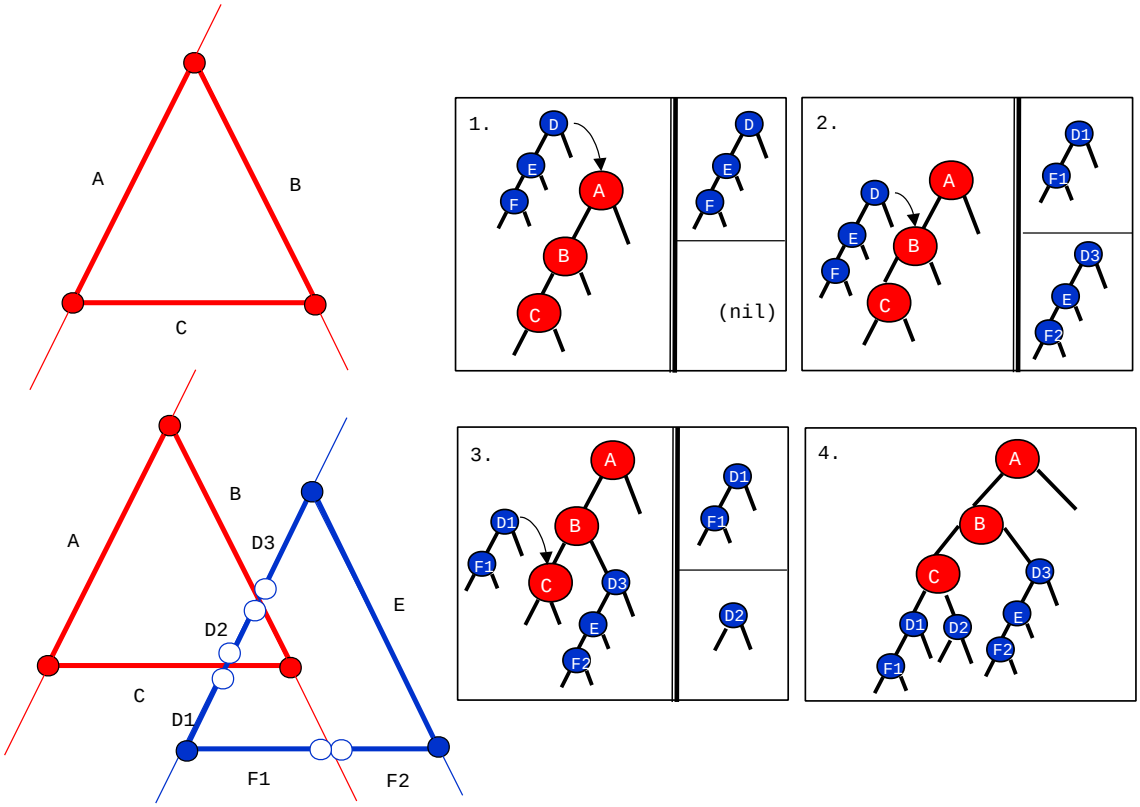

Figure 2.5: Tree Merging.

the partitioner of the node to cut the inserted tree, using the operation described before to partition a BSP. Each one of the trees we obtain in this process is passed along to the subtrees of the node, and the process is repeated. We keep repeating this process until a leaf is reached, which is replaced by the current filtered tree.

For the example in figure 2.5, the partition of the inserted tree does not create an additional tree when compared against node a, but it creates two trees when compared against node b. The result of this process is passed along the subtrees of b, and the tree that faces node c is again partitioned. The moment the tree reaches a leaf node, the filtered tree replaces the previous leaf node. In the case that a solidity attribute is present, this final stage needs to combine the solidity information of the leaf with all nodes of the filtered tree.

2.3 Good BSPs

The process of choosing the set of cutting partitioners in the BSP leads to the question of constructing optimal BSPs. The definition of an optimal BSP is not well established, since several independent properties can contribute to improved performance. In some situations, the depth of the tree is the overriding consideration, but obtaining lower depth trees may require increased splitting. If the number of nodes in the tree is important, then splitting may be reduced at the expense of increasing the height of the tree. Another property that affects the quality of a BSP is the amount of geometry per region. In figure 2.6 we see the example discussed in [15] that compares a balanced cut with a cut that allocates as much geometry as possible in small volumes of space. In cases where geometry is not uniformly distributed the balanced cut is not optimal because the subtrees that are obtained contain lines that have a greater probability of cutting other lines in the same subtree.

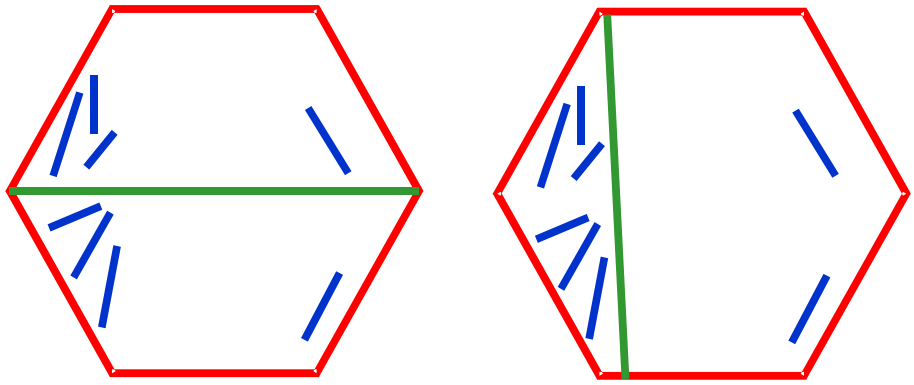

Figure 2.6: A Balanced BSP versus a BSP that prefers allocation of large number of geometry in small regions.

Besides the fact that we have conflicting properties, the evaluation of all possible combinations of BSPs requires exponential work, and approximate solutions become the right approach to achieve good BSPs with reasonable computation times. Some efforts in the literature discuss near-optimal solutions. In [15], the use of expected models that take into account balancing, splitting and the geometry distribution of objects is discussed. The use of evolutionary techniques such as genetic programming is discussed in [5]. The basic idea

is to formulate the BSP construction as an optimization problem, and use similar expected models as before to drive a genetic simulation that evolves a BSP with changes until a good BSP is obtained. A rather simpler approach is to randomly select a partitioner cut from the set of partitioner candidates. The resulting BSP, called a *randomized BSP*, is discussed extensively in [13], where expected bounds on the height and size of the BSP are showed to be optimal.

2.4 Visualizing the structure of a 3D BSP

In this section we illustrate how visualization can be helpful for understanding the structure of 3D BSPs. We have created a framework for visualizing BSPs that focuses on important properties of the tree, combined with a selection mechanism that narrows the set of nodes used in the visualization. Some applications that can benefit from this type of visualization tool include: design of heuristic strategies for BSP construction, manual scene analysis, performance evaluation of BSPs and debugging of dynamic BSPs.

Many applications in Computer Graphics require the manipulation of complex spatial data, and the representation of this information in a way that is efficient and economical in terms of storage is a major design challenge. Hierarchical data structures, like quadtrees, octrees and binary space partition trees (BSPs), allow the representation of complex models in a hierarchical fashion, using the divide-and-conquer strategy. Many algorithms in the literature describe different ways to construct hierarchical structures for a great variety of models[22][21].

Among these hierarchical models, the BSP is the one that allows the greater flexibility in the creation process. Unlike octrees and quatrees, that constrain cuts to be aligned with fixed directions, the BSP allows cuts of the space using hyperplanes of arbitrary orientation. This flexibility increases the number of different BSPs that can be constructed, and raises questions on how to evaluate and compare different trees. As the definition of an optimal BSP is not clearly defined, heuristic strategies have been proposed to produce near-optimal solutions [5][15]. The design of such heuristics is based on the evaluation of qualitative and quantitative properties a BSP should have in order to perform a given set of operations efficiently.

Similarly to binary search trees, balanced BSPs are usually desirable. As pointed out by Naylor[15], the notion of balancing in the BSP not only involves the structure of the tree, but also how geometry is partitioned by the tree. This observation motivates the use of heuristics that also take into account geometric information. Therefore, a tool to visualize the resulting BSP, not only by its structure, but also by the geometric decomposition it creates, is extremely useful in the performance evaluation of such heuristics.

Even if good heuristics have been designed, it may be the case that an automatic BSP generator is not a suitable choice, and a user-controlled partition may be necessary. For instance, in the creation of large environments the user may call an automatic BSP generator to create an initial BSP representation of the scene, and perform a manual fine-tuning process to fix or improve the initial solution by interactively rearranging nodes in the BSP. This manual process can only be accomplished with the visualization of the geometric information of the BSP.

Our visualization tool has been particularly helpful during the debugging and evaluation of the KVD implementation. The KVD changes at critical times, resulting from changes in the structure and the geometric information stored in the tree. As the large combinatorial complexity of the KVD may distract the user from finding the possible source of problems in the code, a visual tool can substantially increase the chances of locating such problems. Also, if the KVD is used as a search structure, a visualization of frequently visited nodes or costly search operations may be done by a visual inspection of the nodes involved. In some cases, a visual inspection may lead to the conclusion that some modification in the structure of the tree is necessary to improve performance.

In this section we discuss issues in the visualization of BSPs. The visualization process combines the enumeration of desirable properties of the BSP, the binding of these properties with display methods, and the selection process that helps identify subsets of the structure of the tree. We illustrate the results with many examples at the end.

2.4.1 Display techniques

Properties

The nodes of the BSP contain many different sources of information helpful to understand the structure of the tree. We separate the information associated with a tree into two categories: geometric and statistic. In the geometric category we include geometric data, like the hyperplane, fragments, normal and regions associated with nodes in the tree. The visualization of each of these properties is done by displaying the geometric information they encode using specified material properties (often color and transparency).

The statistic category contains scalar data associated with nodes in the tree. Examples of statistical information are: number of subtree nodes, maximum subtree depth, volume or area of a node, etc. Each piece of statistical information is represented as a different function, and we call the set of all statistical functions the universe of statistical functions.

The statistical properties do not have an obvious geometric realization, and we visualize them by creating such realizations. We establish a correspondence between statistical data and geometric data, and use material properties to change the visual aspect of the geometric data. After the binding of statistical to geometric data is done, we specify a mapping from scalar values to material properties (color or transparency). For example, we can draw the region associated with a node with a mapped color, corresponding to a certain statistical property of the node. Statistical data is also used in the selection process to define the subset of a nodes you want to display information.

Selection

Once we defined properties about nodes in the tree, the question arises as to which nodes we will select to display such information. As the BSP is a hierarchical structure, it is natural to narrow the selected set of nodes. In this section we discuss the selection process, and discuss three different ways to select nodes in the tree: procedural selection, interactive navigation and intersection plane.

Procedural Selection

The universe of statistical functions define properties associated with nodes in the BSP. In the procedural selection scheme the user defines a logical expression combining functions from the universe of statistical functions. For instance, if $\mathrm{depth(n)}$ specifies the depth in the tree of a given node, the expression $(\mathrm{depth(n)} < 5)$ defines a valid range of depth values. The selection occurs by evaluating the expression, and only nodes that satisfy this expression are selected and displayed.

In general, the selection expression can be any logical expression using the universe of statistical functions. This type of selection function is useful when a global property of the tree needs to be evaluated.

Interactive Navigation

In this selection method the user is interested in understanding the topological structure of the tree. Unlike the procedural selection, where a set of nodes may be used for visualization, here the emphasis is in the evaluation of a single node at time.

In this case, a node in the tree is defined as the current selected node. The structure of the tree gives possible choices to move from the current to a new selection, and typical choices involve moving to adjacent nodes in the tree. In the BSP, these pointers correspond to parent, left and right pointers associated with a node.

More complex movements can be defined in terms of different ways to traverse the tree. For example, from the current node, we can find the next or previous node in a given traversal procedure of the tree (e.g. find next node in a pre-order visit of the tree). The ability to navigate interactively through the nodes of the tree can be very helpful for understanding the topological structure of the tree.

Intersection Plane

One of the great difficulties in the visualization of the geometric information in the BSP is the fact that the visualization may contain occlusion. In the intersection plane approach, a plane is intersected against the structure of the tree, and the resulting cuts of the BSP are displayed over the plane, reducing the dimension of the displayed information.

The selection of nodes using this approach is different than the previous two approaches, as the selected nodes are obtained by intersecting the tree against the intersection plane. This selection method helps to understand portions of the tree that are cluttered with a lot of regions.

2.4.2 Results

In figure 2.7 we illustrate some results that show the application of the techniques described above. The first six examples discuss different visualization techniques for a sample scene composed of blocks (figure 2.7.a). In figure 2.7.b we show the visualization of all hyperplanes in the corresponding BSP for the blocks model. The use of transparency in this example is fundamental because it allows the visualization of the structure of the tree even though many hyperplanes occlude each other. It is important to mention that the compositing of alpha values to simulate transparency is done correctly using the visibility ordering provided by the BSP.

The visualization of all hyperplanes in the tree is important because it helps to understand the global structure of the tree. If a local visualization is also important, one solution is to use the intersection plane selection. In figure 2.7.c we give an example that shows a cut in the BSP by an intersection plane. In this case, the user benefits most by this technique tool if the intersection plane can be moved through the BSP while showing the resulting cuts. For example, a sweep along a specified direction of the intersection plane can very helpful to understand scenes with many orthogonal cuts. This will usually be the case in, for example, architectural models.

The interactive navigation of the tree is illustrated in the next three figures. In a simple navigation, the user is located at a selected node and has the possibility to move up (parent node) or down the tree (left or right nodes). In order to help the user to choose between the left and right nodes, the regions associated with them are displayed. In figure 2.7.d, the current selected node is the root of the tree and the regions of the left and right nodes are displayed with different colors (blue for the left node, green for the right node). In figures 1.e and 1.f we have the same kind of visualization for the left and right nodes of the root. The display of regions in the interactive navigation is very important because it gives the user geometric information about nodes in the tree. This geometric information can then

be used to locate critical parts of the tree.

In Figure 2.7.g we visualize all hyperplanes for a simple scene illustrating a cycle in the visibility graph. In Figure 2.7.h, we show a more complex BSP called Shadow Volume BSP[6], that has additional cuts defined by edges of the model and a particular point (a light source or a viewpoint). Structures like this one are used in applications to accelerate shading calculations or to perform occlusion culling. The greatest benefit occurs when a front-to-back traversal of the BSP discards the traversal of large parts of the BSP because it contain objects that are occluded by already visited objects.

The last example shows how statistical properties of the tree can be used to help the visualization of complex objects. In this case, the number of hyperplanes can grow to be very large, cluttering the visualization. It is possible to understand part of the structure of the model by limiting the set of nodes using the procedural selection. In figure 2.7.i the depth of the tree was used to limit the nodes of the tree, with only the the first five levels in the BSP used in the visualization. The ability to limit the set of nodes in the visualization of BSPs is very important, because it not only reduces the complexity of the tree but also allows the user to focus on important properties of the data.

Finally, the use of this visualization tool is illustrated along this dissertation by the figures used to explain concepts of the KVD. As mentioned before, this tool was crucial to debug the implementation of a complex structure such as the KVD.

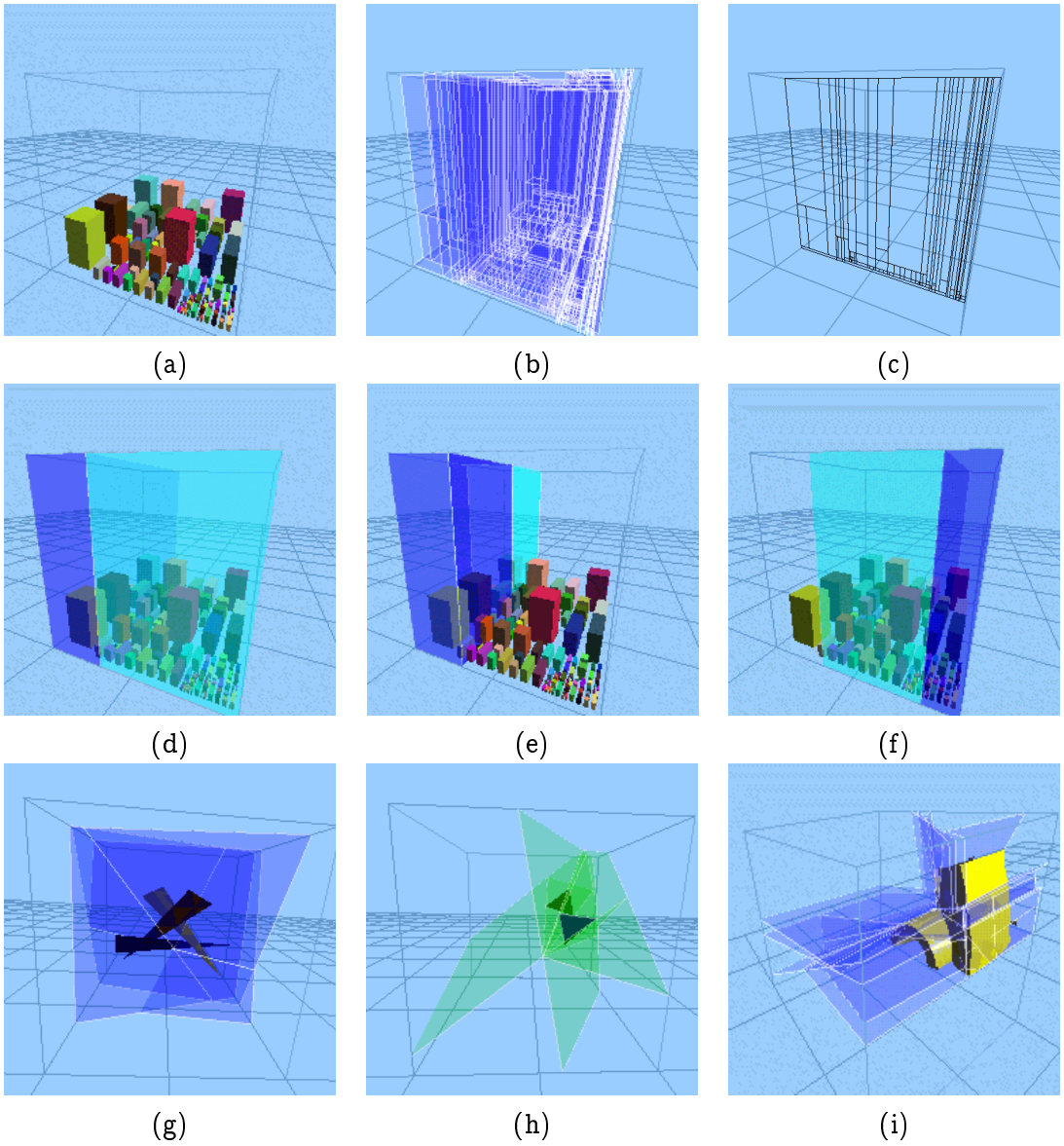

Figure 2.7: Examples of the visualization of BSPs. (a) Scene with blocks, (b) All hyperplanes in the BSP, (c) Intersection Plane Selection, (d) Root hyperplane of the tree and the positive and negative regions, (e)(f) Same information for the left(right) subtrees of the root, (g) All hyperplanes of scene illustrating cycle in the visibility graph, (h) Shadow Volume BSP for a scene composed of triangles and (i) First 5 levels of hyperplanes in a tree for a complex object.

Chapter 3

BSPs for moving geometry

One of the greatest advantages of BSPs for visibility computations is that no re-computation of its structure is necessary when the viewpoint moves. The extraction of visibility information from the BSP is accomplished with a traversal of the tree in a specific order determined by the position of the viewpoint. Although the traversal may not be the same for different viewpoints, the BSP remains the same.

The same can not be said if the objects that define the BSP move. In this case, the movement of objects has a direct affect on the BSP that is constructed from them. This direct relation between objects and the structure of the tree may create an inconsistent situation if the partitioning defined by the BSP is violated, as will happen, for instance, when a polygon moves from one halfspace to another. If no change in the BSP is performed, the resulting visibility ordering may be incorrect.

The static nature of the BSP motivated several people to work on extensions of BSPs to represent moving geometry. In this section we review previous work in two categories. The first category, called *Dynamic* BSPs, is concerned with the update of BSPs as quickly as possible, using a brute-force approach such rebuilding the BSP. The solutions presented in this category emphasize on the implementation aspects of the problem. The second category, called *Kinetic BSPs*, discusses solutions that try to exploit temporal coherence as much as possible to avoid rebuilding the BSP every time an object moves. The identification of specific times that require a change in the BSP, combined with local updates in the BSP, are the main points of this approach. Unlike the first category, this work is focused

in theoretical aspects of the problem, providing high-level descriptions of algorithms and analyses of the complexity characteristics of algorithms.

The work we present in this dissertation is based on algorithms of the second category, with two major differences. First, we propose a structure that is a *fully 3D BSP*. As we will see later, the kinetic solutions we review do not provide an event mechanism that works in the structure of the BSP in 3D, but rather control events in the projection of the BSP onto a 2D plane. Second, we present a *fully working implementation* of kinetic BSPs, with a description of the details of the algorithm, rather than focusing in the complexity analysis. For this problem, there is a big gap between the description of the algorithm at a high-level and its actual implementation. The fact that many cases can be described by simple reflections of a small number of base cases simplifies the description of the algorithm. In the implementation, however, the need to have a correct solution for every single case makes the problem much harder. Obviously, similar behavior in different cases needs to be exploited, and allows the problem to be treated in a reasonable manner. As in similar situations, many problems not discussed in higher-level descriptions of the algorithms need to be addressed.

3.1 Dynamic BSPs

In [26] a new structure called a *Dynamic BSP* is proposed. The motivation is to insert additional planes in the BSP in order to reduce the number of situations that require changes in the BSP. In practice, this approach localizes the updates needed for deletion and reinsertion of moving objects in a BSP. This approach does try to exploit, by introducing additional planes, the spatial coherence of the dynamic changes in the tree. The proposed structure contains four new types of planes. The first type is called a *first range separating plane*, and corresponds to a plane that linearly separates objects in two regions. For cases where a linear separation is not possible, a *second range separating plane* is introduced to define which object will stay the same, and which one is going to be partitioned. To avoid extra splitting, *wrapping planes* are used around objects to serve as a bounding volume. Finally, user-introduced planes are called *divisor planes*. In figure 3.1 we illustrate an example of this structure.

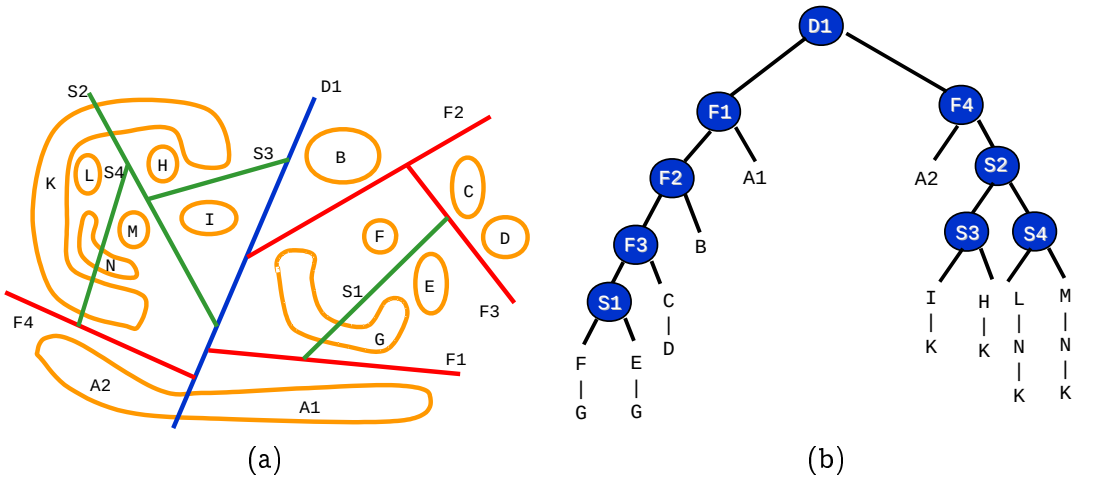

Figure 3.1: Dynamic BSP using separating, wrapping and user-defined cuts. (a) Divisor cuts (D*, blue), first range separating planes (F*, red) and second range separating planes (S*, green) are added before the objects. (b) Resulting BSP.

Once the structure is created with the addition of external planes, the update due to the movement of an object is done with a brute-force procedure that checks affected parts by the object and rebuilds the tree if necessary. In some situations, where the moving object remains isolated by a valid separating plane, no reconstruction is necessary.

In [17] a method to implement dynamic changes in a BSP-tree is described, where the static objects are represented by a balanced BSP tree (computed in a pre-processing stage), and then the moving objects are inserted at each time step into the static tree. Only insertions are required, as a copy of the static world is used, and no constraints on the path of the inserted objects are defined.

In [7] a more general approach is proposed (but only for BSPs in two dimensions), which does not make any distinction between static and moving objects. By keeping additional information about topological adjacencies in the tree, the algorithm performs insertions and deletions of any node in a more localized way. The augmentation by topological pointers of BSP trees in higher dimensions is also discussed in [8], but not in the context of dynamic changes. All these prior efforts boil down to deleting moving objects from their earlier positions and reinserting them in their current positions after some time interval has elapsed. Such approaches suffer from the fundamental problem that it is very difficult to know how

to choose the correct time interval size: if the interval is too small, then the BSP does not in fact change combinatorially, and the deletion/re-insertion is just wasted computation; if it is too big, then important intermediate events can be missed which affect visibility.

The use of *parallel BSPs* is described in [28]. The application here is related with the real time visualization of ultrasound volumes, and BSPs are responsible for providing a visibility ordering, used in the accumulation of opacity values. The input data consist of a set of slices of the data set, and a fixed number of slices are combined to generate a final image. A great deal of coherence is present, because one slice is discarded and one inserted at every step. In order to avoid the problem of deleting and rebuilding the BSP every time a new slice is processed, deleted slices are marked as invalid in the BSP. In order to speed-up the process of marking invalid slices, a second copy BSP (called *replacement*) corresponding to a different phase in time is maintained. The visualization uses one of the BSPs (called *active*) to render images. After a certain number of frames is processed, the replacement BSP starts to be formed until the number minimum of slices is reached, when the role active and replacement BSPs is interchanged, and the old active BSP is reinitialized. This approach is claimed to produce a near constant frame rate, important in the real time visualization.

3.2 Kinetic BSPs

The problem of maintaining visibility when the geometric entities of the scene start to move requires fast updates in the BSP-tree. These updates are traditionally handled by deleting the moving element from its old location in the tree, and re-inserting it again in its actual location.

One disadvantage of such an approach is the fact that the structure of the tree (which represents the topology of the subdivision) may not change at every movement, leading to useless computation. Also, the changes that need to be performed in the tree are local to some parts of the tree. The traditional process of deletion/insertion does not take this property into account, instead performing a global update in the tree.

The second category of BSP algorithms for moving geometry is based on the use of the framework of kinetic data structures. The movement of objects does not always require the

update of the BSP, and kinetic BSPs exploit this fact for the case of continuously moving geometry. More specifically, the idea is to establish certificates that serve as proof that the BSP remains combinatorially valid (i.e., does not require an update). These proofs are used to establish the critical times when the BSP requires changes. A BSP that is maintained following this approach is called a *Kinetic BSP*.

One of the major difficulties of designing kinetic BSPs (and kinetic data structures in general) is the design of certificates. We discuss this issue in much more detail in the next chapter, and briefly just state here that one of the major difficulties is related to the fact that the BSP represents an arrangement that contains cells of unbounded combinatorial complexity. The solution to this problem, used in previous work and also used in this thesis, is to include external cuts from the objects along pre-defined directions. These cuts are called *cylindrical cuts*, because they transform a general arrangement into an array of cells with bounded complexity (trapezoids in 2D or rectangular prisms in 3D - called *cylindrical cells*).

In [2] an algorithm is presented to maintain a kinetic BSP for n non-intersecting segments moving continuously in the plane. Two types of cuts are present in this BSP: edge cuts (along segments) and point cuts (along a specific direction and passing through an endpoint of a segment). Events that change this kinetic BSP happen when two endpoints switch x-order (or y-order, depending on the convention for the point cuts). When an event occur, a trapezoid of the subdivision disappear. In figure 3.2 we illustrate the BSP created for a set of line segments, and shade with different colors the trapezoids that will disappear when one particular line segment moves.

The detection of event times that change the BSP allows the algorithm to avoid rebuilding the BSP every time an object moves. This is not the only benefit of the kinetic approach to BSPs. Because the events have information about the specific nodes that require changes in the BSP, a local reconstruction of the affected parts of the BSP can be done, instead of a complete reconstruction of the BSP. The nature of the local algorithms involves movement (deletion and insertion) of nodes in the tree, and a finite set of possible cases can be identified. Using local updates, the paper claims that is possible to update the BSP in $O(\log n)$ expected time. The construction time is expected $O(n \log n)$, and the expected height of the tree is $O(\log n)$.

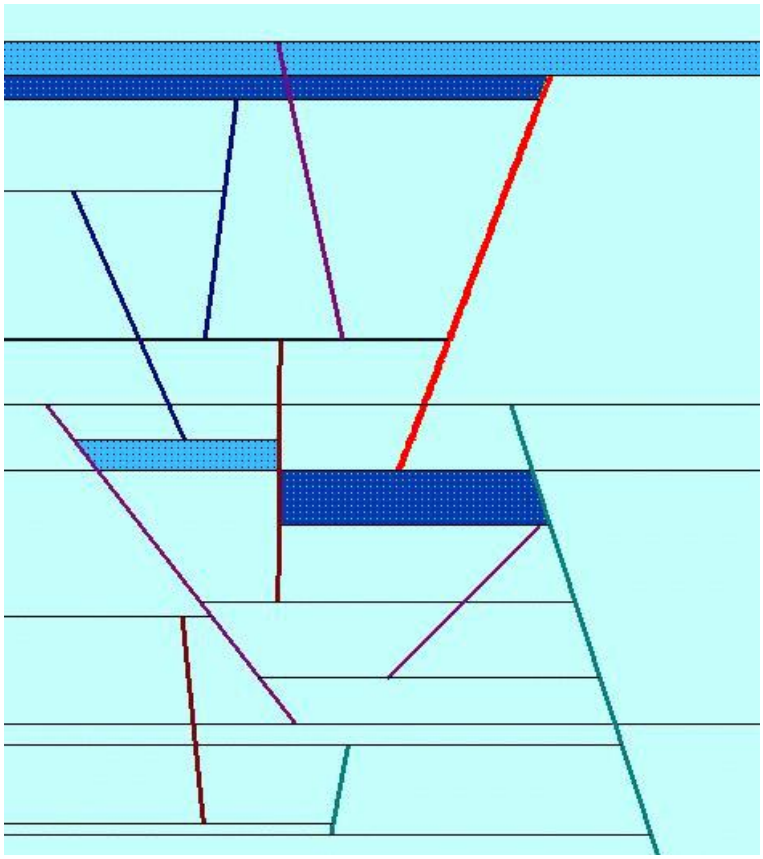

Figure 3.2: Kinetic BSPs in 2D

The extension to three-dimensions of this work is presented in [1], where a kinetic BSP is proposed for continuosly moving non-intersecting triangles in 3D. The approach to extending the algorithm to 3D requires first extending the previous algorithm in 2D to the case of intersecting line segments. The possibility of having intersecting lines creates many more cases that can change the BSP, affecting both the set of certificates and the local update algorithms. One difficulty comes from the fact that additional cuts need to be inserted at the intersection point of two line segments.

Once this new algorithm is in place, the 3D algorithm can be explained. A 3D BSP is constructed from a set of non-intersecting triangles with cuts from vertices (point cuts), edges (edge cuts), intersections of edge cuts (intersection cuts) and triangles (triangle cuts). If all cuts are projected into a single plane, the BSP induced in the plane by the 3D BSP is called shadow BSP. In figure 3.3 we illustrate a sample scene with two triangles and the

corresponding shadow BSP projected onto one of the walls of the bounding box.

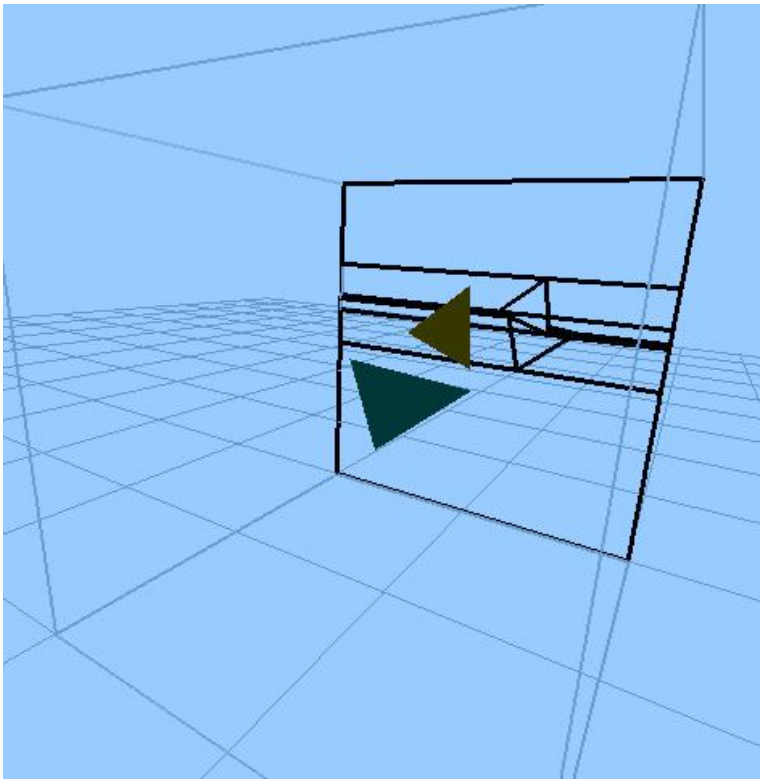

Figure 3.3: Kinetic BSPs in 3D

Instead of defining certificates in the 3D BSP, the authors of [1] propose maintaining certificates in the shadow BSP. Because there are no intersecting triangles, changes in the 3D BSP correspond to changes in the shadow BSP. The opposite is not true, as events may occur in the shadow BSP but not in the 3D BSP. The shadow BSP corresponds to a set of intersecting line segments in the plane, which can be maintained by the algorithm discussed previously. Although the resulting BSP works in 3D, events are defined in a 2D BSP. The complexity bounds obtained are (all expected times): depth ($O(\log n)$), size ($O(n\log^2 n+k')$, where k' is the number of intersections between pairs of edges in the projection plane), construction time ($O(n\log^3 n+k'\log n)$ and update time ($O(\log^2 n)$).

Chapter 4

Kinetic Vertical Decomposition Trees

4.1 Motivation

The kinetization of a BSP is another problem where the choice of cutting planes involves tradeoffs. In this case, topological changes in the structure of the tree happen when there is a change in the topology of one of the cells of the subdivision. In auto-partition BSPs, for example, cells may have unbounded complexity, which makes the maintenance of the topological structure much more difficult. The example in figure 4.1 illustrates two situations where a BSP constructed with auto-partition cuts is defined using a moving line segment (red) and several static segments (blue).

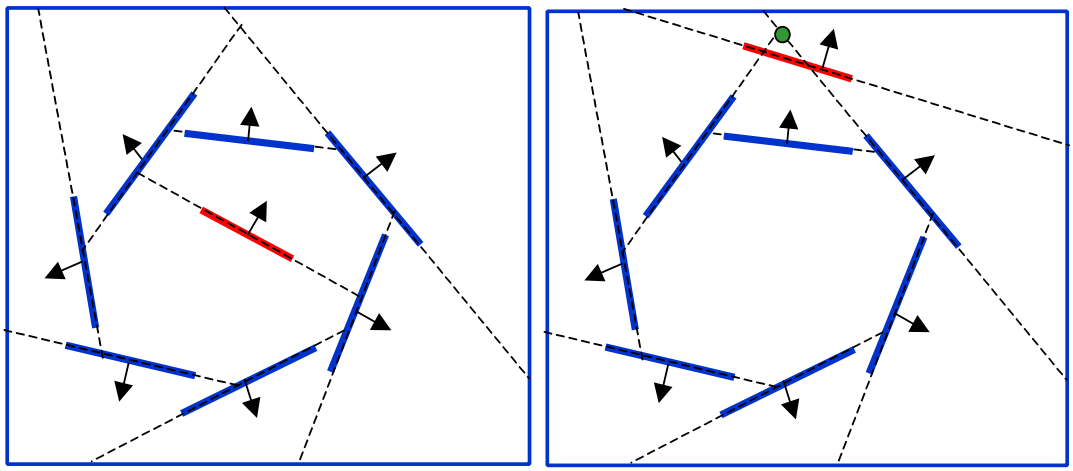

Figure 4.1: Example of events in the traditional BSP

In both situations, events that change the BSP correspond to a point passing through a line (or vice-versa), or two segments passing through each other. This last case is not a problem because we assume in this discussion that the segments are non-intersecting. In order to discuss the principal case, a line passing through a point (or vice-versa), it is first necessary to discuss the possible types of points and lines in the subdivision. The lines are of only one type, corresponding to the lines that support the input line segments. We have two types of points, defined by the endpoints of the line segments or by the intersection of two lines. Note that this last type of point does not belong to the input set of objects but rather is defined by the cuts of the auto-partition BSP.

For the first situation, the movement of the segment inside the region will require a change in the BSP when one of its endpoints passes through one of the seven lines that define the region that encloses the segment. In order to detect such an event, it would be necessary to maintain the topological structure of the cell that contains the line segment. In the second situation, the segment was split into three parts. In this case, for the middle part of the segment, an event happens when it passes through the green vertex, which corresponds to one of the points of the region that encloses the segment.

From these two examples we conclude that events occur when a moving feature (a segment) passes through the lines or points that define the enclosing region. This requires the representation of the topological structure of the arrangement, which in itself is a challenging problem if the features of the arrangement move. The fact that some events involve points that do not belong to the input objects is a major reason why this representation is necessary. Moreover, additional complexity arises because regions in the arrangement are organized in a hierarchical fashion, corresponding to the intersection of halfspaces along paths in the BSP (see chapter 2 for a discussion of BSP regions).

In our approach we explore how the addition of auxiliary planes can limit the topology of cells, and as a consequence, simplify the kinetization process. More specifically, we include additional planes during the construction of the BSP in such a way that the subdivision created by the BSP corresponds to a hierarchical vertical decomposition of the input objects. The decomposition that we obtain for the example before is illustrated in figure 4.2.

Compared to the auto-partition approach, the BSP with additional planes has several advantages. The most important difference is that we do not need to explicitly maintain

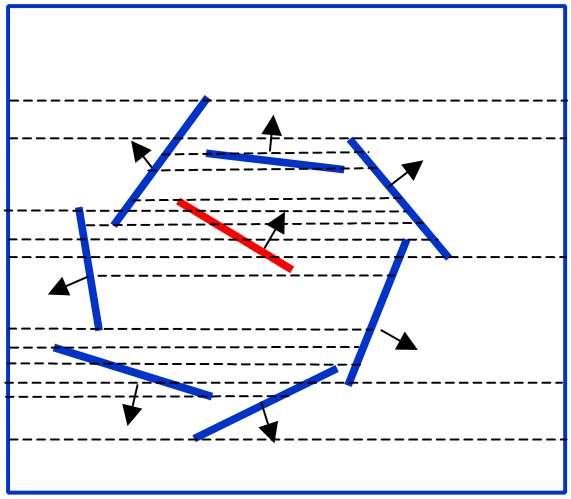

Figure 4.2: Example of events in BSP with additional cuts to form a vertical decomposition

the combinatorial structure of the arrangement. This is only possible because the vertical decomposition limits the special types of points that involve events to points of the input objects. Also, the fact that cells have bounded complexity simplifies the identification of regions that enclose features, and the structure of the tree can be used to extract them without need for an explicit representation.

The goal in this chapter is to give an overview of the main ideas concerning the structure that combines a BSP-tree representation of a vertical decomposition with a kinetic structure to allow the detection of changes in the topology of the cells. We call this special type of BSP the *Kinetic Vertical Decomposition tree* (KVD-tree or simply KVD).

We start the presentation with reviews of important basic concepts, such as *Plücker* coordinates and vertical decompositions (in 2D and 3D). Next we review the main issues of the kinetic data structures framework, and present the KVD structure, discussing briefly its main components, which will be fully explained in the following chapters.

4.2 Plücker Coordinates

Homogeneous coordinates are very important in many applications in Computer Graphics, because they allow a simple and unified representation of projective spaces. The generalization of homogeneous coordinates to higher dimensions is also called *Plücker* (or Grassmann)

coordinates (see [23]). The use of Plücker coordinates in this work is motivated by the fact that vertical decompositions require the computation of walls that are defined by a combination of features of the model (vertices and edges) and directions. Replacing directions with points in Plücker coordinates turns out to be an elegant, robust and generic way to generate the requisite plane equations.

In this discussion we restrict ourselves to the representation of points, lines and planes in three dimensions, which are described as follows:

- A point p is represented in Plucker coordinates in 3D as the array $[a_0, a_1, a_2, a_3]$, which corresponds to the euclidian point $(a_1/a_0, a_2/a_0, a_3/a_0)$.

- A line l is represented in Plucker coordinates in 3D as the array $[l_0, l_1, l_2, l_3, l_4, l_5]$.

- A plane m is represented in Plucker coordinates in 3D as the array $[m_0, m_1, m_2, m_3]$. It corresponds to the plane $m_0 + m_1x + m_2y + m_3z = 0$.

The manipulation of Plücker coordinates requires a special set of formulas (a complete set of formulas can be found for up to four dimensions in [23]). In particular, two formulas are used here to compute plane equations, corresponding to the formula that computes a line representation given two points, and the one that computes a plane given a line and a point. Both of these formulas are given in table 4.1.

line $l \leftarrow$ point $p \bigvee$ point q
$l_0 \leftarrow p_0q_1 - p_1q_0$
$l_1 \leftarrow p_0q_2 - p_2q_0$
$l_2 \leftarrow p_1q_2 - p_2q_1$
$l_3 \leftarrow p_0q_3 - p_3q_0$
$l_4 \leftarrow p_1q_3 - p_3q_1$
$l_5 \leftarrow p_2q_3 - p_3q_2$

plane $m \leftarrow$ line $l \bigvee$ point p
$m_0 \leftarrow -l_2p_3 + l_4p_2 - l_5p_1$
$m_1 \leftarrow l_1p_3 - l_3p_2 + l_5p_0$
$m_2 \leftarrow -l_0p_3 + l_3p_1 - l_4p_0$
$m_3 \leftarrow l_0p_2 - l_1p_1 + l_2p_0$

Table 4.1: Useful Plücker Formulas in 3D

4.3 Vertical Decompositions

4.3.1 2D Vertical Decompositions

Let $S = \{s_1, ..., s_n\}$ be a collection of n non-intersecting line segments in $\mathbb{R}^2$. We call the vertical decomposition $V_{\hat{x}}(S)$ of S the subdivision formed by the segments of S and walls (lines in 2D) erected from the endpoints of each segment in S. The resulting cells of this decomposition are called *trapezoidal* cells, with bounded complexity of at most four sides. An example of a vertical decomposition is shown in figure 4.2. In traditional vertical decompositions, walls are extended along the specified direction until a segment is reached. In another approach, that we call a *hierarchical vertical decomposition*, segments are inserted in order, together with the walls they create. Unlike the traditional type, walls are extended until a segment that was already inserted in the vertical decomposition is reached. This type of vertical decomposition is important because it allows its representation by a hierarchical structure such as the BSP. In the discussion to follow, the notion of a vertical decomposition corresponds to the hierarchical variant.

There are no constraints on the direction $\hat{x}$. In most figures in the plane we use the direction of x-axis, for the simple reason of personal preference. In the literature, however, the y-axis is the preferred choice, which also justifies the name of vertical decomposition. The use of a different direction also emphasizes the fact that a general direction can be specified.

Vertical decompositions are very useful because they are composed of cells of bounded complexity. For general arrangements, the fact that cells can have complex topologies make the task of performing operations and representation much more difficult. In terms of complexity, the vertical decomposition is optimal in two dimensions, which means that it has the complexity of the underlying arrangement.

4.3.2 3D Vertical Decompositions

Let $T = \{t_1, ..., t_n\}$ be a collection of n non-intersecting triangles in $\mathbb{R}^3$. We call the vertical decomposition $V_{\hat{x},\hat{z}}(T)$ of T the subdivision formed by the three types of walls (planes in 3D), erected from the vertices, edges and triangles in T. The first type of wall is called a P-wall, and corresponds to the wall extended from a vertex v of a given triangle along

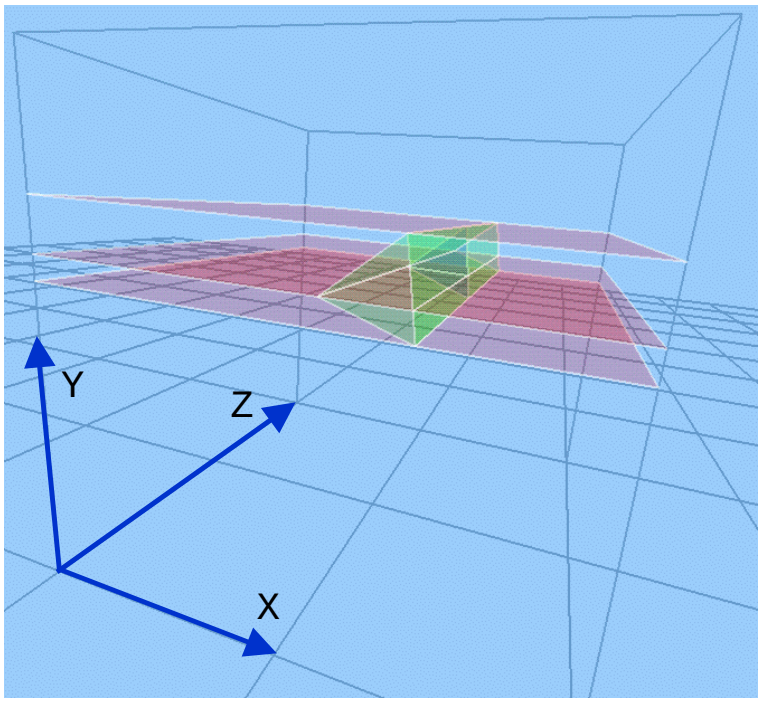

Figure 4.3: Example of vertical decomposition in 3D

the plane defined by ν and the two directions $\hat{x}$ and $\hat{z}$. The second type of wall is called an E-wall, and corresponds to the wall extended from an edge e of a given triangle along the plane defined by e and the direction $\hat{z}$. The final type of wall is called a T-wall, and corresponds to the wall defined by the plane of a triangle.

The resulting cells of this decomposition are called *cylindrical* cells, with bounded complexity of at most six sides. The direction orthogonal to $\hat{x}$ and $\hat{z}$ is called $\hat{y}$. As in the 2D version, the directions that define the vertical decomposition may be given by any two non collinear vectors. In addition, we represent directions using Plücker coordinates, which allow the representation of directions as points at infinity. Here, the representation of directions as points in Plücker coordinates gives us the flexibility to replace directions by euclidian points (points that do not have a zero homogeneous coordinate). Therefore, instead of having walls of the same type parallel to each other, we have walls that converge from one element of the triangle (either a vertex or edge) into a single point. This is useful if this single point corresponds to the viewer position or a light source in a scene. In the first case, the vertical decomposition will be directly related to the visibility information

associated with the viewer, which can be very useful during the computation of the visibility ordering. If this point is a light source, then the vertical decomposition encodes information that can be useful to compute shadow information.

In terms of complexity, the vertical decomposition of a set of n triangles in three-dimensional space is near-optimal, and is defined by $O(n^{2+\epsilon}+K)$, where K is the complexity of the arrangement of the triangles.

4.4 Kinetic Vertical Decomposition Trees

4.4.1 Representation and Construction

A KVD is a special type of BSP, with the addition of cuts to form a vertical decomposition of the space. For the case of triangles in $\mathbb{R}^3$, the KVD contains three different types of cuts. A *point cut* (P-cut) is defined by a plane that passes through the triangle vertex, and is parallel to directions $\hat{x}$ and $\hat{z}$. An *edge cut* (E-cut) is defined by a plane defined by two vertices of an edge and parallel to $\hat{z}$. A *triangle cut* (T-cut) is a cut along the supporting plane of the triangle. In figure 4.4 we illustrate the different types of cuts for a single triangle.

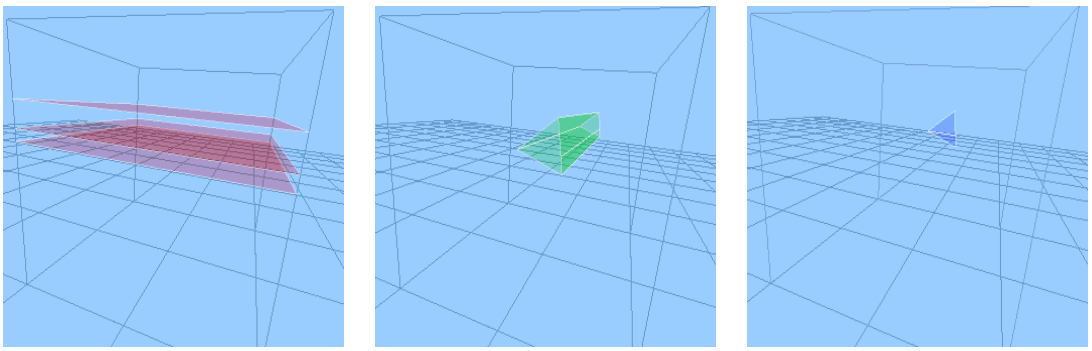

Figure 4.4: KVD-Tree Cuts. (a) Point Cuts, (b) Edge Cuts, (c) Triangle Cuts

There are many issues to be discussed regarding the representation and the construction of the KVD (see chapter 5). The representation of dynamic information is always a

challenging task, especially in the case of geometric algorithms, which combine both geometry and topological data. In our case, the geometry is associated with the representation of the input model (vertices, edges and triangles) and cuts in the tree (hyperplanes and fragments). Similarly, topological data comes from the model (adjacency relations between vertices, edges and triangles) and the structure of the tree. We will see that it is convenient to replace geometric information stored explicitly in the tree by external indexes to a geometric structure.

The construction of the KVD corresponds to an incremental insertion of triangles and all associated cuts. We will establish a fixed random order for this insertion, called the *priority order*. The use of a random order can be proved to provide efficient results regarding depth and size of the KVD. Moreover, it also provides a way to check the correctness of the updates in the KVD.

Other issues regarding the implementation will also be discussed. The representation of binary trees involving different types of nodes but with many similar procedures can be exploited nicely in the framework of object-oriented languages.

4.4.2 Events and Certificates

The KVD-tree represents under the kinetic framework a vertical decomposition for a moving set of triangles. The movement of triangles affects the cylindrical cells of the vertical decomposition, and the identification of specific times when the topology of the cylindrical cells changes is one of the core tasks in the kinetic simulation. These changes are also important because they represent changes in the combinatorial structure of the tree. The example given in figure 4.5 describes the case where the movement of two point cuts may cross each other, destroying the cell between them. For better understanding of the case, a line is used to connect the vertices that define both cuts.

There is a limited number of events that create changes in the tree. Based on the enumeration of all these cases, it is possible to define a set of certificates that encode the combinatorial structure of the tree. As long as this set remains invariant, the structure of the tree stays the same. Every time a certificate fails, an algorithm to perform a local change in the KVD is invoked.

The set of certificates is affected by changes in the tree, with new certificates to be

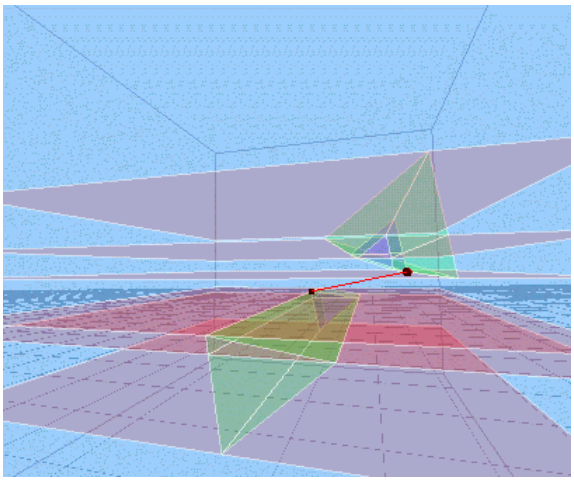

Figure 4.5: A KVD event corresponding to two point cuts passing through each other.

added, and others to be deleted. The efficient update of the set of certificates relies on the fact that certificates are associated with nodes in the tree. Because of this, the certificates to be updated correspond to nodes that are affected by the changes in the structure of the tree. Therefore, there is a big connection between the algorithms that update the tree and the update of the certificates. This observation motivates the storage of certificates at nodes in the tree, rather than in a external structure, which simplifies the updates in the certificates by the algorithms that update the tree.

In addition, the storage of certificates at nodes in the tree allows the use of the structure of the tree to represent a priority queue containing certificates ordered by event times. This is explained in more details in 6, but the basic idea is that each node contains, besides its certificates, the next certificates to happen in time. As a result, the root of the tree always contains the next certificates to fail in the kinetic simulation. This ability to incorporate the priority queue into the data structure being kinetized is a novel extension to the kinetic data structures framework.

4.4.3 Update Algorithms

A topological change in the cells of the KVD-tree requires an update of the structure of the tree. Based in the events and certificates of the KVD, it is possible to describe the actions

that need to be performed in the tree for each specific situation. Different behaviors are described in separate update algorithms, and each one of them is described in chapter 7. The existence of many update cases makes it difficult to implement the update algorithms. In order to describe all possible cases into a smaller number of algorithms, more complex BSP operations are defined.

A KVD has a more complex update procedure than traditional BSPs because additional cuts are created for every triangle. The incidence relations defined in the topology of the input model creates an additional relationship between all nodes derived from a single triangle. This relation is not directly stored in the tree, where nodes that have such relationships may be stored at different locations. If one of these nodes is affected by a change, the remaining nodes connected by the topology of the model are also affected. Therefore, the topological relations need to be taken into account when updating the tree.

The use of a new operation, called a *tree dragging*, is defined to move nodes in the tree with the above behavior. Usually, changes in the KVD correspond to one node (called moving node) crossing a hyperplane defined by another node (called crossing node) in the tree. This requires the deletion of the moving node from the subtree it is located (corresponding to the old halfspace), and insertion into the other subtree of the crossing node (the new halfspace). The deletion of the moving node requires the merging of its two subtrees. Also, because of connectivity relations, some nodes that are incident to the moving node may also need to move into the new subtree. The dragging operation is a more complex operation that performs both actions described above. It consists of a merging algorithm that checks the incidence relation of the subtrees against the moving node. If a node incident to the moving node is found, the behavior of the movement of the incident node is computed, and the appropriate actions to the incident node are taken.

The use of a fixed priority order in the construction of the tree creates the need to insert nodes into new locations in the tree, which may contain lower priority nodes. This may result in the insertion of a node in a position different than a leaf of the tree, and may require the splitting of subtrees. This behavior is different than traditional BSPs, and new insertion algorithms and merging procedures are described to take into account the priority of the nodes in the tree. These operations are described in detail in chapter 7, before the presentation of all update algorithms.