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.

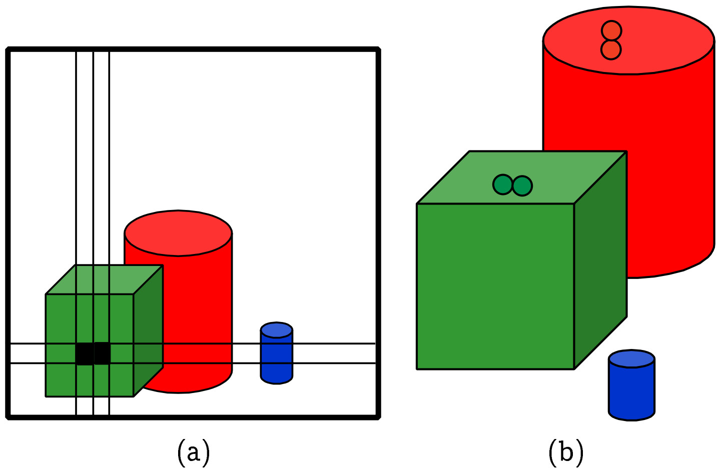


Figure 1.1: Coherence information. (a) Image-space. (b) Object-Space.

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.

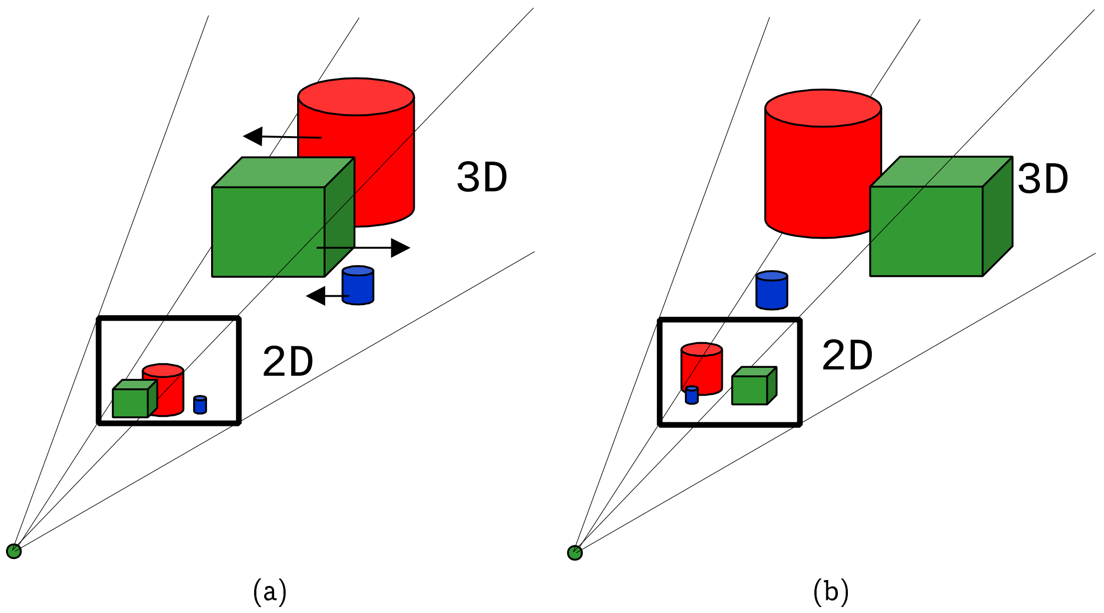


Figure 1.2: Dynamic Visibility Problem. (a) sample scene (b) changes in scene due to the movement of objects.

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.

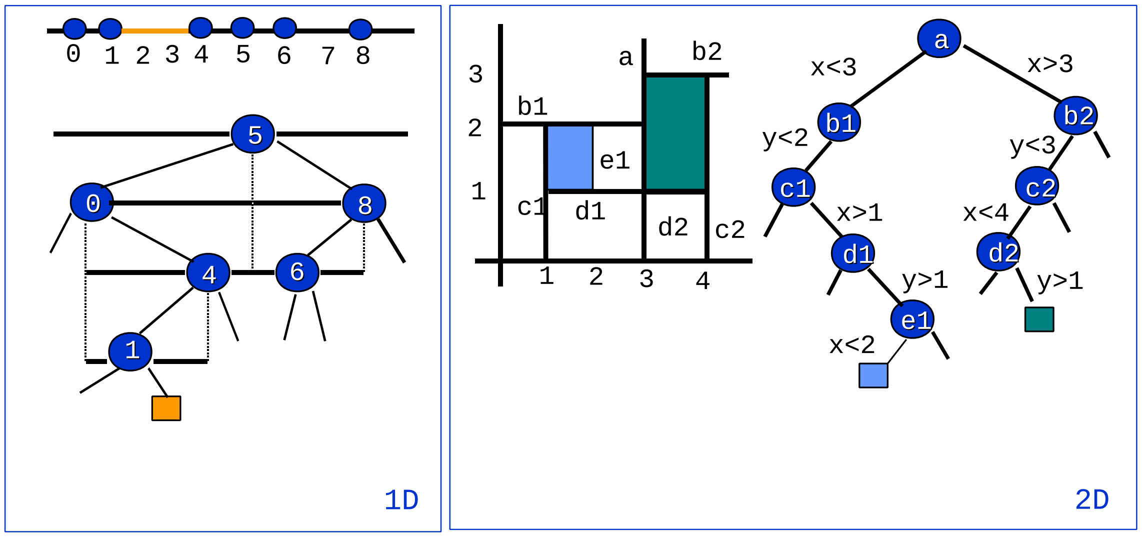


Figure 1.3: BSPs in 1D and 2D

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

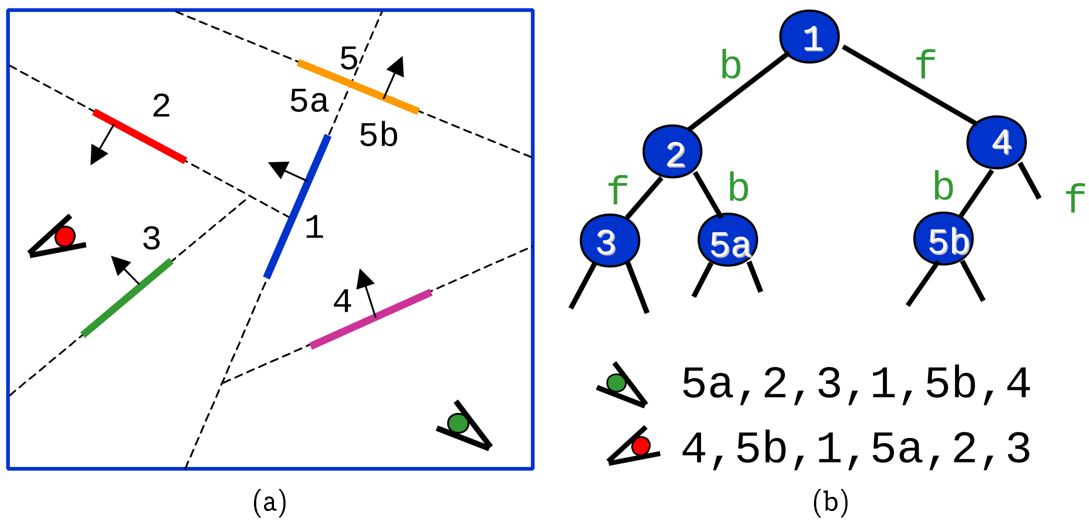


Figure 1.4: Extracting visibility from the BSP (a) BSP subdivision (b) BSP tree and visibility ordering for two viewpoints.

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.

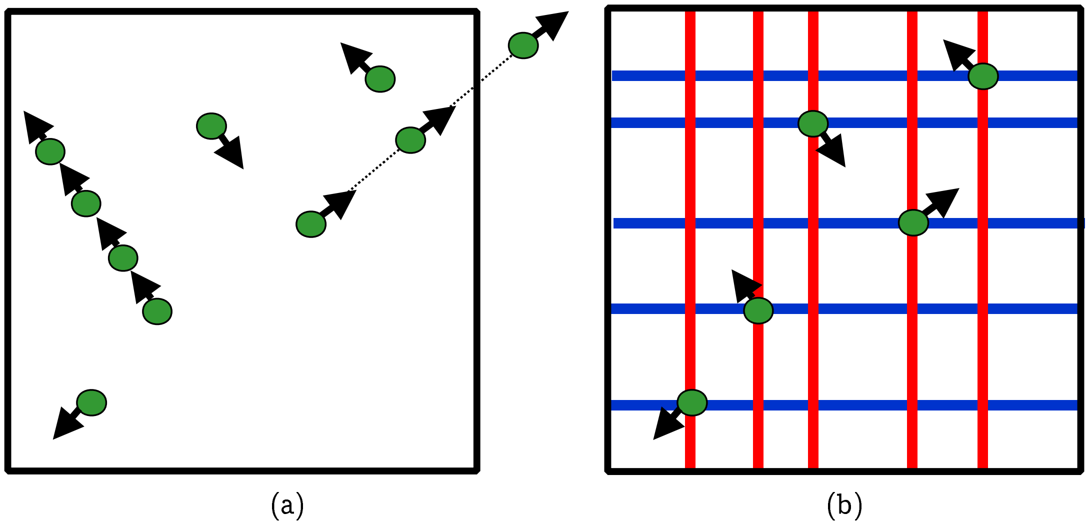


Figure 1.5: Moving balls inside a rectangle. (a) Fine sampling may lead to unnecessary work, while coarse sampling may miss some events. (b) Certificates that serve as a proof that the balls stay inside the rectangle

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:

$$h \equiv \{(x_1, \ldots, x_d) \mid a_1 x_1 + \ldots + a_d x_d + a_{d+1} = 0\} \tag{2.1}$$

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 $(a_1, a_2, \ldots, a_d)$.

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

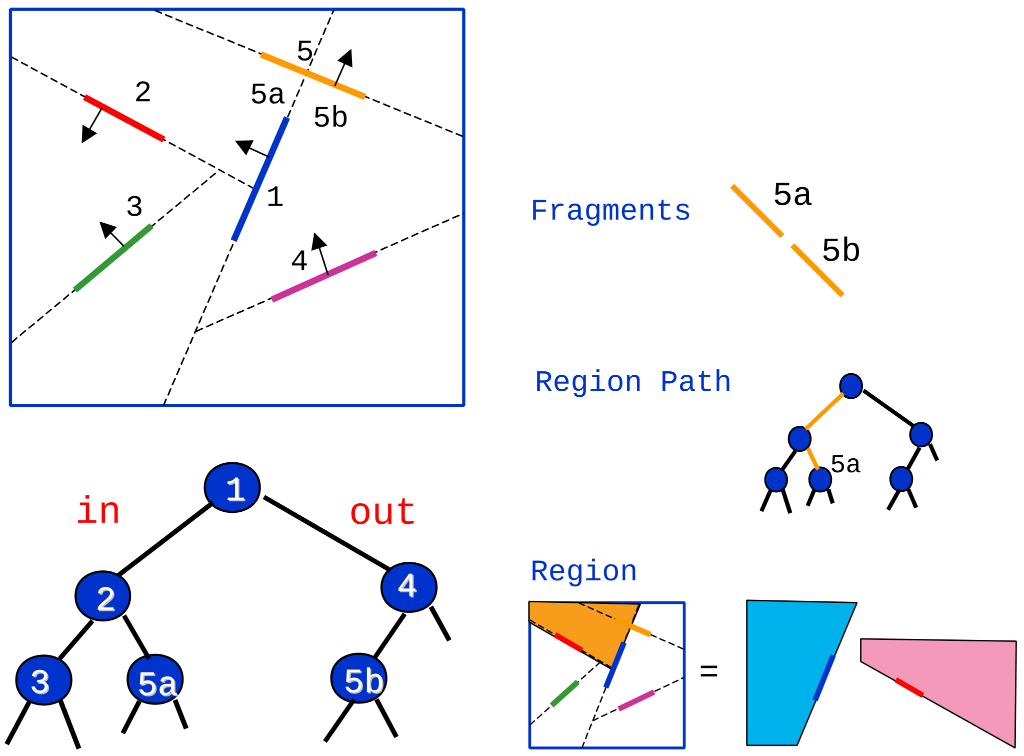


Figure 2.1: BSP Concepts. Sample subdivision, the corresponding tree representation, fragments, a region path and the corresponding region (in yellow).

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.

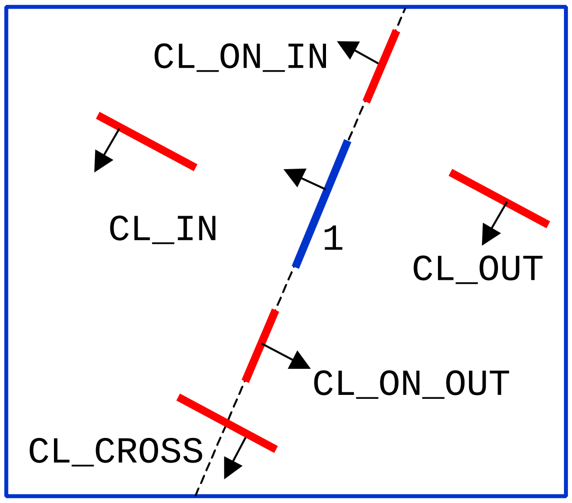


Figure 2.2: Possible results for the classification operation.

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.

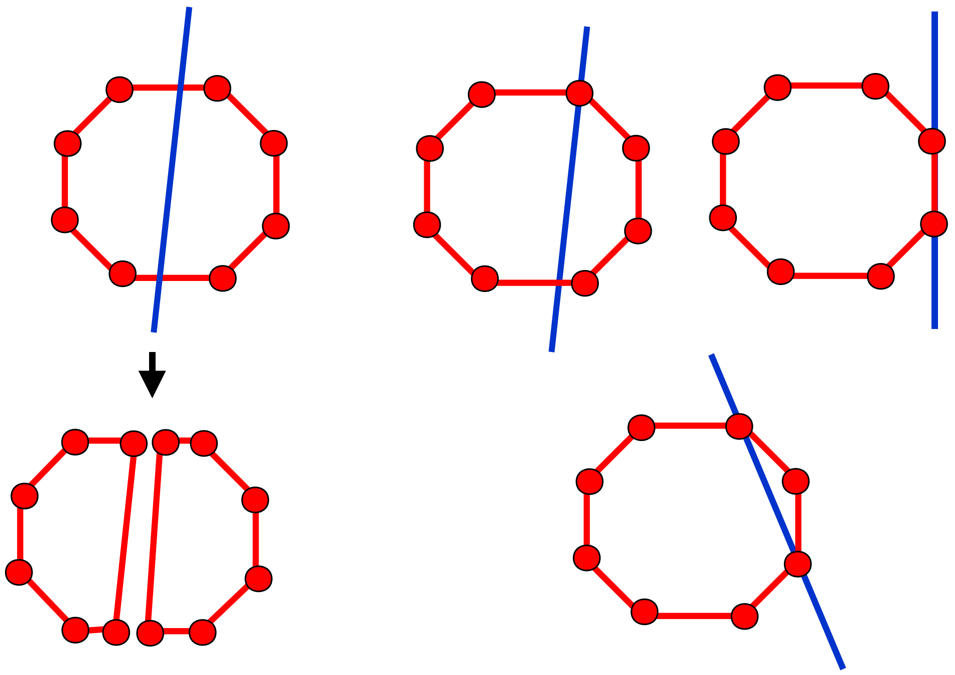


Figure 2.3: Partition of a fragment. Points of the fragment are classified in order against the partitioner. Special cases where a point lies in the partitioner may create special situations in the partition algorithm.

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.

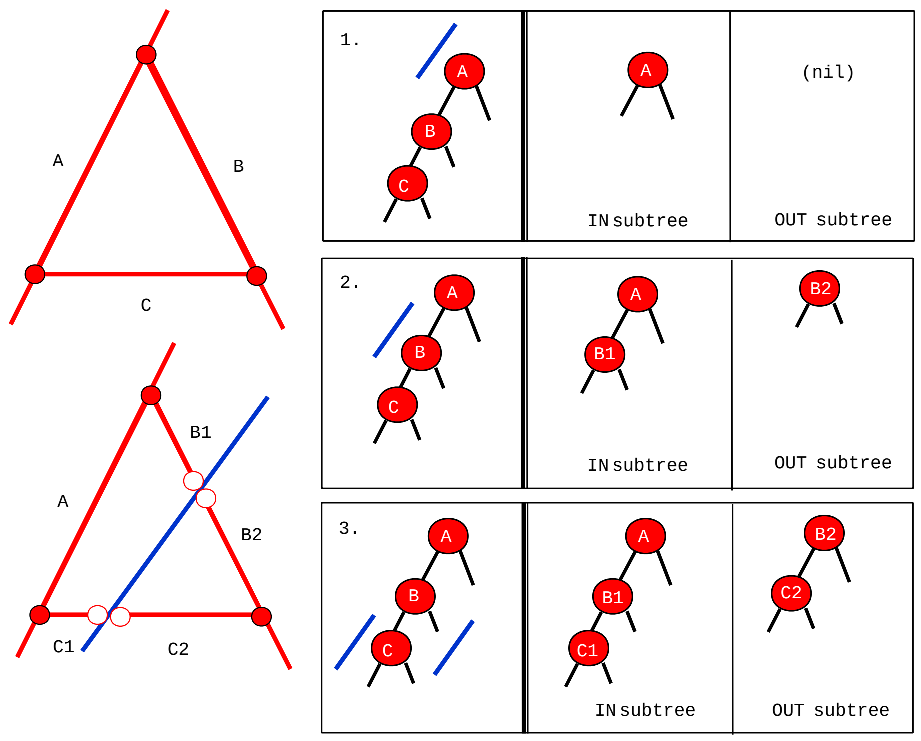


Figure 2.4: Partition of a BSP.

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

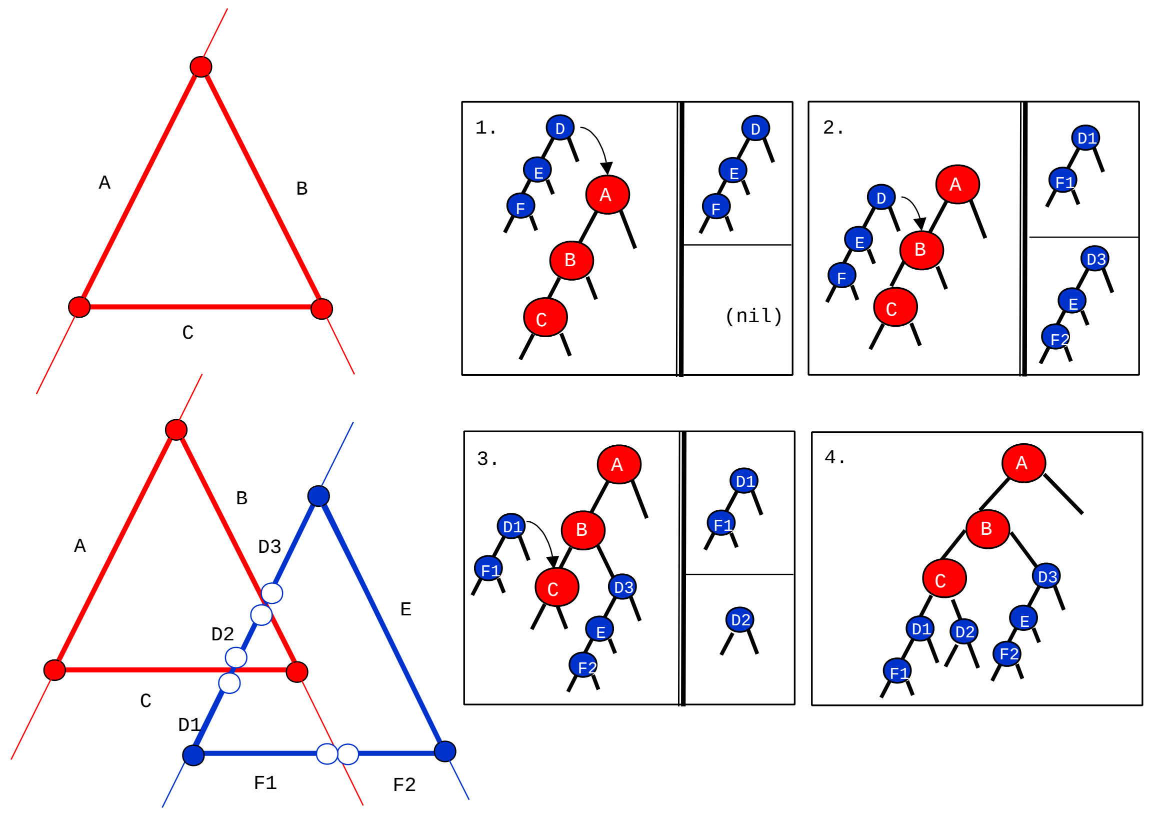


Figure 2.5: Tree Merging.

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.

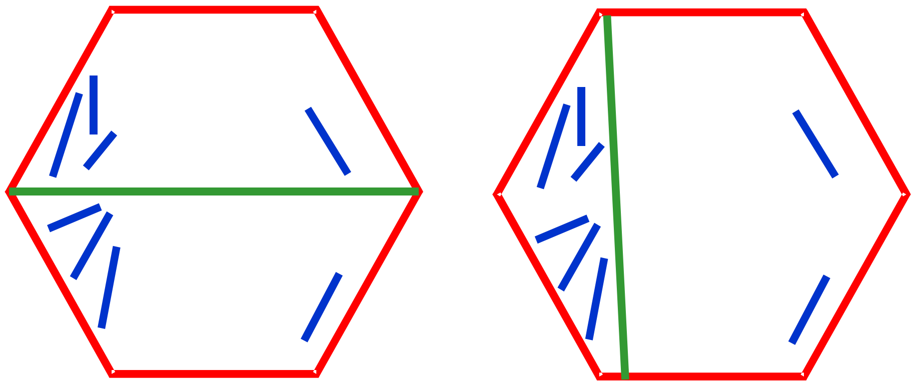


Figure 2.6: A Balanced BSP versus a BSP that prefers allocation of large number of geometry in small regions.

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.

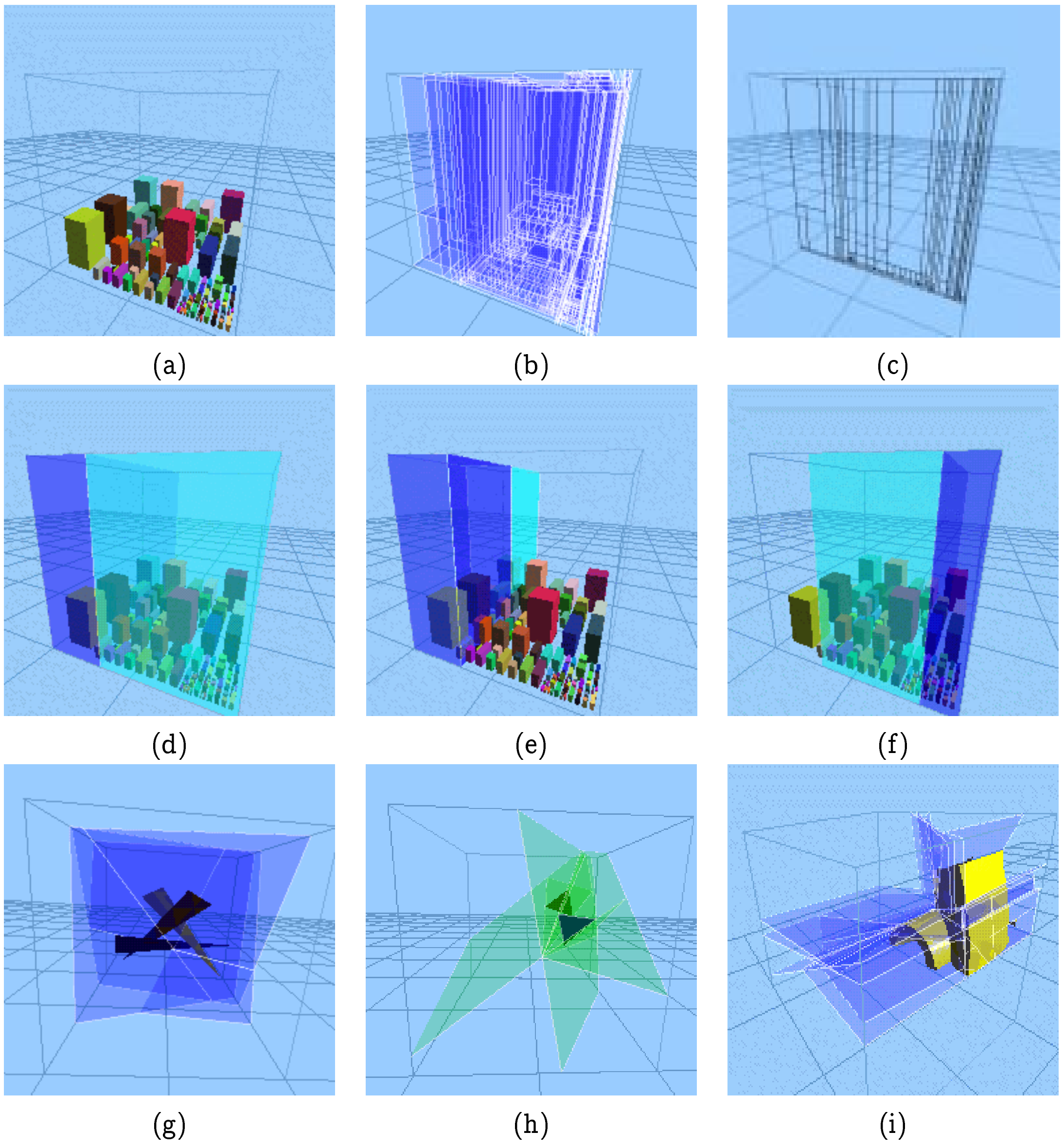


Figure 2.7: Examples of the visualization of BSPs. (a) Scene with blocks, (b) All hyperplanes in the BSP, (c) Intersection Plane Selection, (d) Root hyperplane of the tree and the positive and negative regions, (e)(f) Same information for the left(right) subtrees of the root, (g) All hyperplanes of scene 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.

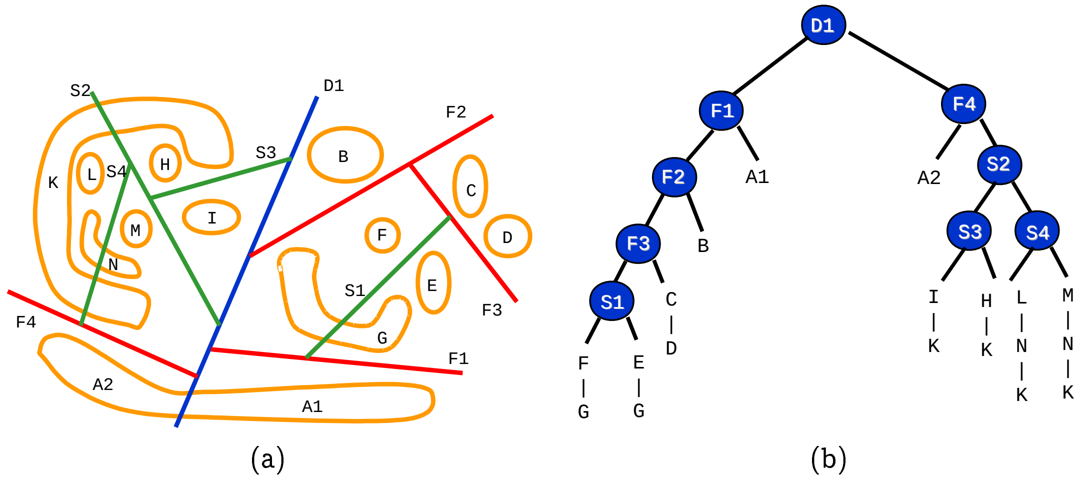


Figure 3.1: Dynamic BSP using separating, wrapping and user-defined cuts. (a) Divisor cuts (D*, blue), first range separating planes (F*, red) and second range separating planes (S*, green) are added before the objects. (b) Resulting BSP.

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 $O(\log n)$.

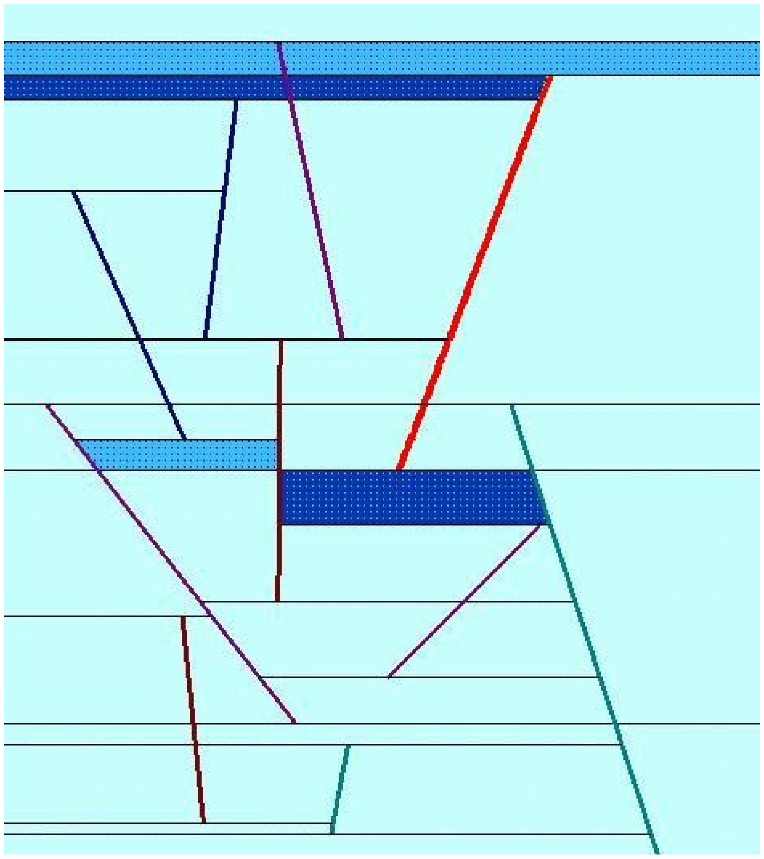


Figure 3.2: Kinetic BSPs in 2D

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.

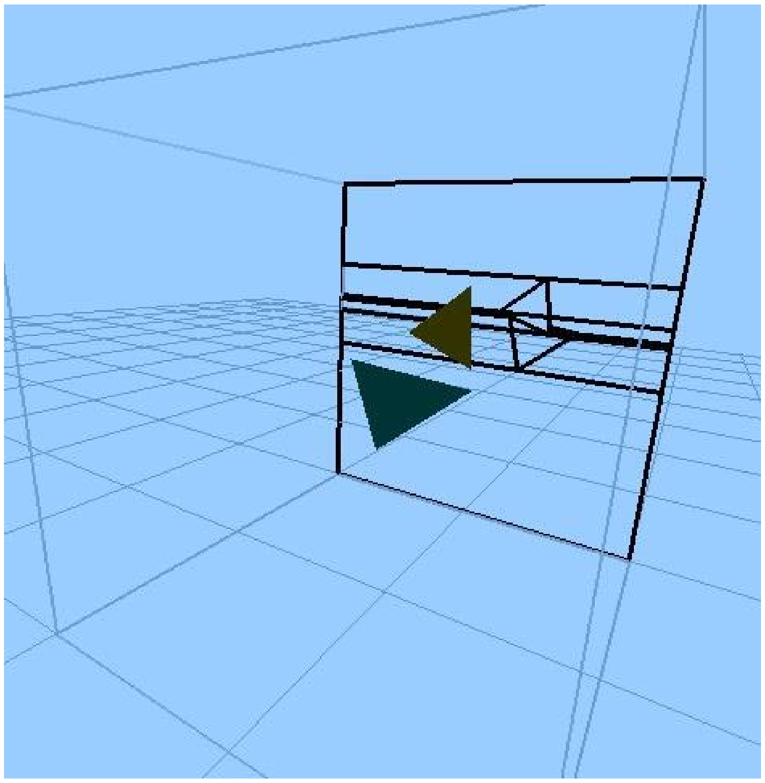


Figure 3.3: Kinetic BSPs in 3D

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).

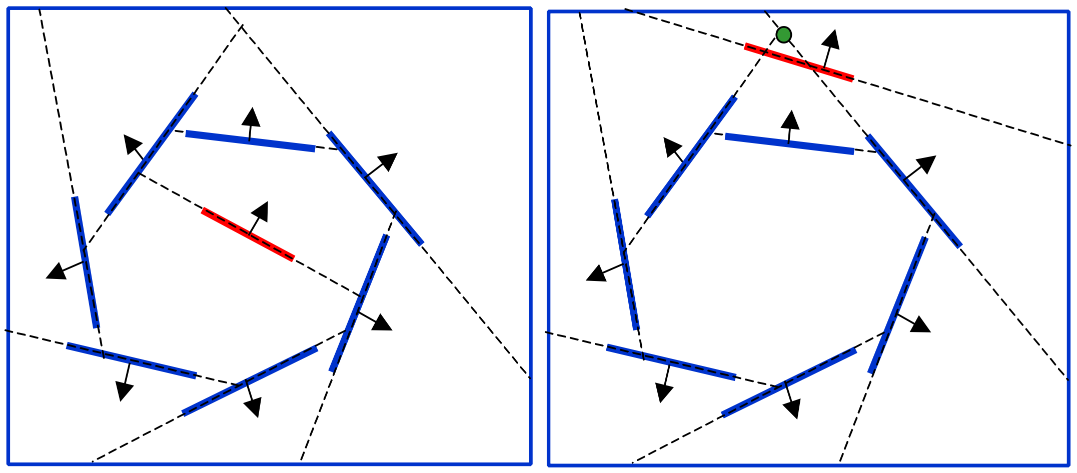


Figure 4.1: Example of events in the traditional BSP

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

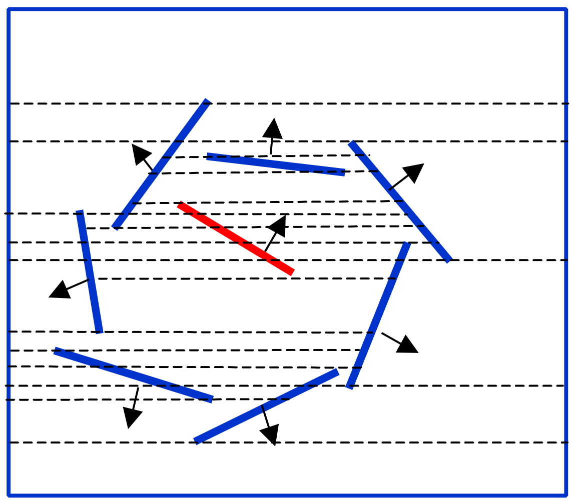


Figure 4.2: Example of events in BSP with additional cuts to form a vertical decomposition

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

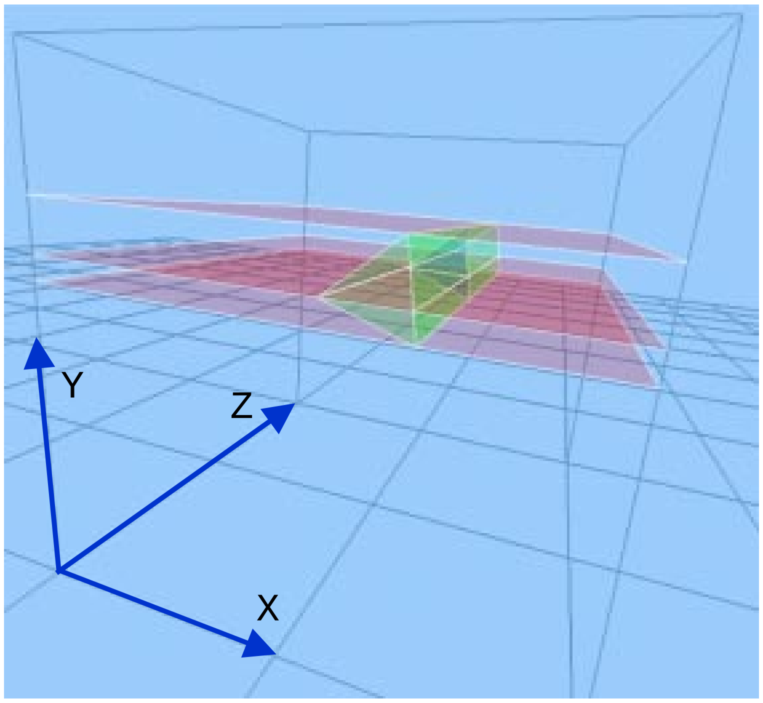


Figure 4.3: Example of vertical decomposition in 3D

the plane defined by v 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.

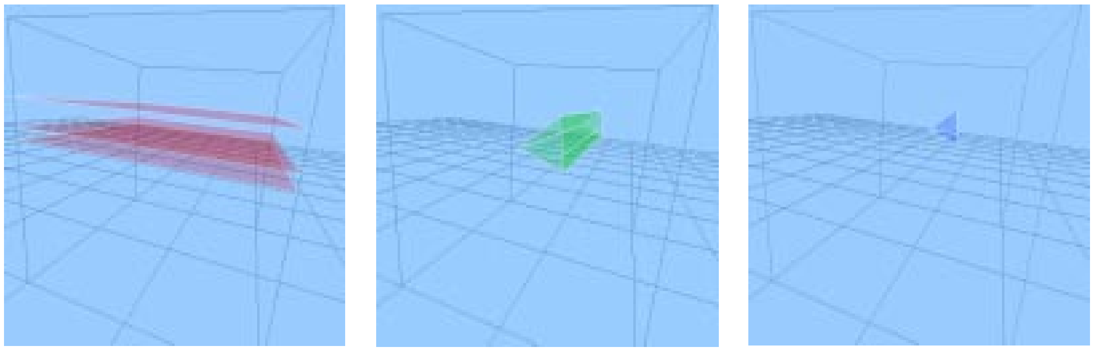

(a) (b) (c)

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

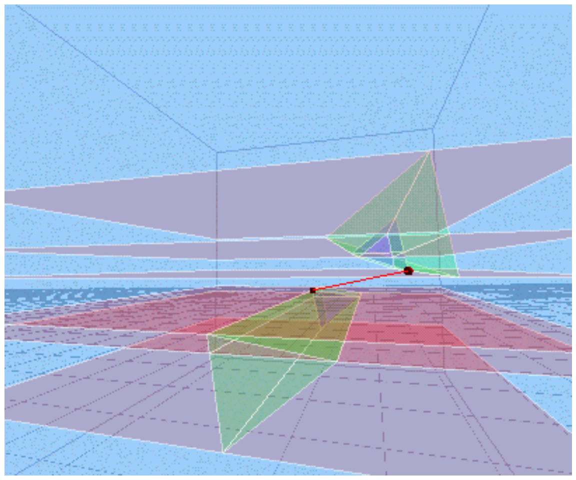


Figure 4.5: A KVD event corresponding to two point cuts passing through each other.

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.

Chapter 5

KVD Representation and Construction

5.1 Symbolic Representation of Geometry

It is common practice in the implementation of static BSPs to store the geometric information about hyperplanes and fragments explicitly in the tree. This can be important during the rendering of the BSP, where quick access to fragments in visibility ordering is necessary. In the scenario of moving geometry, however, hyperplanes and fragments change frequently, and the cost of updating their representation in the tree overcomes the advantages of explicit storage. For the simple case of a moving face, the updates required in the tree include all nodes that the face generates in the tree, each containing fragments of the original face. Because these nodes are stored in different places in the tree, it becomes harder to recover and update all of them. In addition, the structure of the tree itself may change with moving geometry, but not as frequently as the fragments and hyperplanes. These observations suggest removing geometric information from the tree.

Indirection is a key concept to be exploited in this situation. Explicit storage of hyperplanes and fragments is replaced by a new representation that does not involve geometric coordinates, but instead indexes to an external structure. This structure, called the *scene structure*, contains geometric and topological information about all objects, such as the vertices, edges and faces of the model and their adjacency relations. Unlike the explicit storage approach, moving geometry may not require changes in the tree, and be handled entirelly with updates in the scene structure. The situation where the tree changes requires special treatment, but as it happens with less frequency, the index approach still can be very useful because it optimizes the most frequently ocurring cases.

The indexing scheme will be responsible for representing all possible types of hyperplanes and fragments in the tree. The KVD has three different types of nodes: point, edge and triangle nodes. They store information about three different types of cuts: P-cuts, E-cuts and T-cuts. The hyperplane that defines a triangle node, for instance, corresponds to the plane equation that supports the triangle used to define the cut. In the new approach, the index of this triangle is stored in the node instead of the hyperplane, and the hyperplane equation can be easily recovered by accessing the scene structure with the stored index. Similarly, point and edge nodes contain the indices of vertices and edges. In both of those latter cases, the computation of the hyperplane equation is not a simple lookup process in the scene structure, because the main directions of the vertical decomposition need to be taken into account. The computation of the hyperplane equation is therefore enclosed in the scene structure, and the hyperplanes are represented for the different types of nodes by a single index (vertex, edge or triangle).

The representation of the fragment is more subtle, because it depends on the location of the node in the tree. For a triangle node defined by a triangle t, for instance, a triangle fragment corresponds to the intersection of t with the halfspaces of all ancestor nodes. The triangle fragment is obtained by repeated partition operations that are performed when the node is inserted in the tree. For edge nodes, the edge fragment corresponds to the partition created over the edge that defines the cut (a subset of the original edge). Point nodes are special because no partition happens over vertices. In this case the fragment is simply the vertex that defines the cut. Therefore, the only types of fragments that require a special representation are edge and triangle fragments. It is important to observe that edge fragments are only defined over edges of the input model, and not over edges obtained in the partitioning process. The internal edges are only represented in triangle fragments.

An edge fragment is represented by two vertex and one edge descriptors, while a triangle fragment is composed of a collection of vertices and edge descriptors. Because the KVD has only three different types of planes, it is possible to create a symbolic representation to encode all possible types of vertices and edges that can appear in edge and triangle fragments. The possible types are shown in figure 5.1. We illustrate the new types of vertices and edges that arise every time a new cut is introduced in the KVD.

Based on the enumeration of all possible cases, an index scheme is proposed, which relies on the creation of a symbolic representation of the vertices, edges and planes. The notions of a symbolic plane (SP), edge (SE) and vertex (SV) are defined to be used in the representation of hyperplanes and fragments. In the discussion to follow we use P, E and T to represent the cuts created respectively by the vertex v, edge e and triangle f of the scene structure.

5.1.1 Symbolic Plane

A symbolic plane is used in the representation of hyperplanes at each node of the tree. The representation is straightforward, defined as a function of the types of cuts used in the KVD, defined as follows.

- $SP(V) = (v)$, the index of the vertex used to define the P-cut.
- $SP(E) = (e)$, the index of the vertex used to define the E-cut.
- $SP(F) = (f)$, the index of the triangle used to defined the T-cut.

The hyperplane that defines a T-cut can easily be obtained by accessing the scene structure with the symbolic plane index. In this case the index refers to a triangle index, and the corresponding plane equation of the triangle is used to define the hyperplane for the node.

For P-cuts and E-cuts there is no direct correspondence between an index in the node and the hyperplane equation. For the P-node, the hyperplane is defined by computing the plane that passes through the vertex that defines the node, and the two directions of the vertical decomposition. Similarly, the E-cut is defined by the primary direction of the vertical decomposition and the two vertices that define the edge used in the node.

5.1.2 Symbolic Edge

The symbolic edge represents all possible partitions that can occur over an edge of the input model. This representation does not include all types of edges that can occur in the

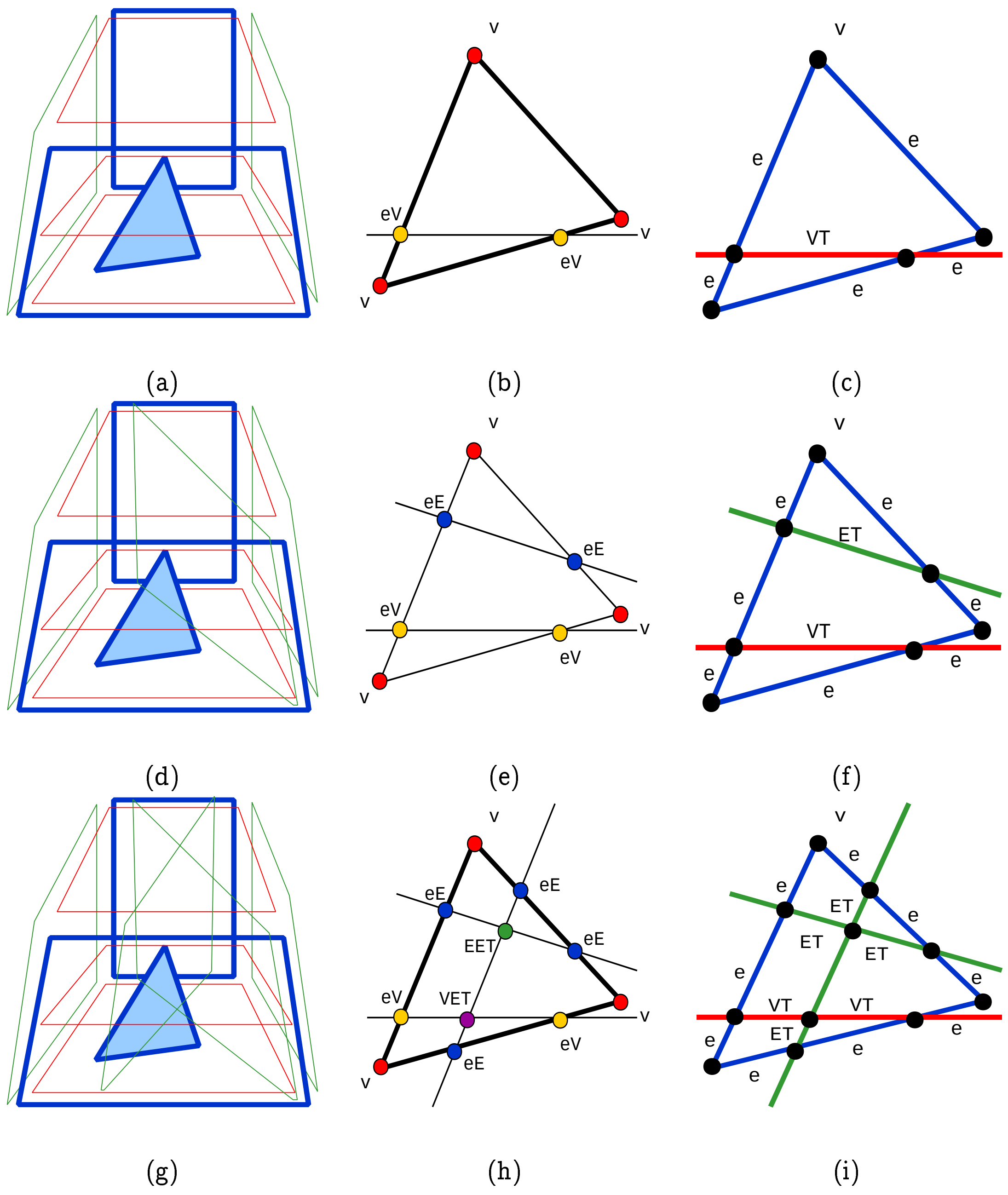


Figure 5.1: Possible types of vertices and edges that can appear in edge and triangle fragments. (a) Sample scene with one point cut. Types of vertices (b) and edges (c) corresponding to scene in (a). (d)-(f) Scene, vertices and edges for one point and edge cut. (g)-(i) Scene, vertices and edges for one point and two edge cuts.

cylindrical cells of the vertex decomposition. A general representation would require the additional of several other cases. Because the only subset of edges to be represented are over triangles of the input model, the following code is simplified to represent only the necessary cases, described as follows (see Figure 5.1):

- $SE(e) = (e)$, the index of the edge used to defined the E-cut.
- $SE(V,T) = (SP(V), SP(T))$, the edge obtained by the intersection of a P-plane and a T-plane.
- $SE(E,F) = (SP(E), SP(F))$, the edge obtained by the intersection of an E-plane and a T-plane.

5.1.3 Symbolic Vertex

The simplest representation for a symbolic point would be the indexes of the three planes that intersect to create the vertex. Some of the planes of the vertical decomposition do not have an index, as they are defined by the combination of vertices and edges with specified directions, and therefore another representation is necessary. The symbolic vertex representation has five different types, described as follows(see Figure 5.1):

- $SV(v) = (v)$, a vertex index of the input model. We refer to this vertex as an *endpoint* vertex.
- $SV(e,E) = (e,SP(E))$, the intersection of an edge of the input model with the plane that defines an E-cut. We refer to this vertex as an *intersection* vertex.
- $SV(e,V)= (e,SP(V))$, the intersection of an edge of the input model with the plane that defines an P-cut. We refer to this vertex as a *thread* vertex.
- $SV(E_1,E_2,T) = (SP(E_1),SP(E_2),SP(T))$, the intersection of a triangle of the input model with the planes that defines two E-cuts. We refer to this vertex as an *internal intersection* vertex.
- $SV(V,E,T) = (SP(V),SP(E),SP(T))$, corresponding to the intersection of a triangle of the input model with the planes of a P-cut and an E-cut. We refer to this vertex as an *internal thread* vertex.

The representation of fragments is more involved because it represents a convex region over the hyperplane. For edge fragments a single tuple of the form (SV_1, SV_2, SE) is used. For a triangle fragment with n vertices, we use n tuples of the format (SV, SE) to represent every vertex and edge in this region oriented along a pre-defined orientation (clockwise or counter-clockwise).

5.2 Data Structures

The current implementation of the KVD uses C++. The use of a programming language like C++ that allows object-oriented data types was essential in simplifing the design and implementation of a data structure like the KVD. The implementation is composed of three major data structures: the scene structure, the KVD tree and the KVD global.

The scene structure stores information about the various objects that compose the scene. For each object it maintains a list of vertices, edges and triangles and the corresponding incidence information. Additional information, like bounding volume information and directions of the vertical decomposition are stored in the scene structure as well. The description of this structure as a C++ class follows:

```
class Scene {
private:
  gmVector4 _xhat, _zhat;  // vertical decomposition directions
  gmVector3 _minBox, _maxBox;  // bounding box
  Object *_objects;  // static and dynamic objects information
  Priority *_priority;  // priority order
  Vertex *_vertices;  // vertex information
  Edge *_edges;  // edge information
  Triangle *_triangles;  // triangle information
public:
  // Query methods: access information regarding vertices, edges, hyperplanes, etc
  // Update Methods: update private data
  // Display Methods: print or draw in 3D representations of private data
  // Classification Methods: classify fragments with respect to hyperplanes
}
```

The KVD is the binary tree structure that represents the partition of the space. Each one of the different cuts in the KVD has its own characteristics, and therefore is represented using different node types (point, edge and triangle nodes). Although the representation using different nodes seems natural, some common structure exists between them. For example, many methods are common to all node types, especially tree traversal methods. The inheritance mechanism available in C++ is used to remove this redundancy of representation. The class *KVDTree* is defined to contain methods and common field structures that are shared by the different types of nodes, and all node classes are derived from this base class. This class is described as follows:

```
class KVDTree {
private:
  KVDTree* _left, _right, _parent; // tree pointers
  KVDGlobal* _global; // global information structure
  SymbolicPlane _hyperplane;
// hyperplane used to partition the space;
  ...
public:
  // Query methods: access information about private data. hyperplanes, etc
  // Update Methods: update private data
  // Display Methods: print or draw in 3D representations of private data
  // Classification Methods: classify node fragments against hyperplanes
  // Events and certificate operations.
  // Basic tree operations: merge, partition, etc.
  // Local update algorithms
  ...
}
```

There are three classes that are derived from the *KVDTree* to contain specific information about each type of node: *KVDPointNode*, *KVDEdgeNode* and *KVDTriangleNode*. Specific information defined for each node includes: fragment representation, certificates and events, methods that perform updates in the KVD, display information, etc. The three different types of nodes in the KVD are described as follows:

```
class KVDPointNode: public KVDTree {
private:
  // Point node certificate information
public:
  ...
}


class KVDEdgeNode: public KVDTree {
private:
  // Edge node certificates
  EdgeFragment _eFragment;
public:
  ...
}


class KVDTriangleNode: public KVDTree {
private:
  TriangleFragment _tFragment;
  // triangle node certificates
public:
  ...
}
```

Many of the node methods require access to global information that is unique and common to all nodes in the tree. In order to avoid redundancy, we store this information in a common structure, called *KVDGlobal* . We include a reference to this structure in the base class *KVDTree*. Every time a given method requires access to global information, the pointer to the KVD global is accessed to return the desired information. This solution scales really well, as every new common property can be easily added to the KVD global abstract node class without having to make any changes in the abstract types of the nodes. Some examples of common properties stored as KVD global information are: display flags (control the display information), and pointers to other general structures (KVD, scene structure, kinetic simulation structure). The class definition follows:

```
class KVDGlobal
{
private:
  KVDTree *_root; // Root of the KVD
  KVDPPointNode *_pointNode; // Point nodes in the tree
  Scene *_scene; // Scene data structure
  ...
  // Viewpoint and light source information
  // Display properties
  // Simulation information
  // Statistics information
}
```

5.3 KVD Construction

The KVD is constructed from a scene composed of non-intersecting triangles in $\mathbb{R}^3$. A small random perturbation is initially applied to the coordinates of all points in the scene, so that degenerate cases where points have the same x, y or z coordinates are removed. A universe bounding box is defined to enclose the entire input scene.

The construction of the KVD proceeds by incrementally inserting of cuts in the tree. Before describing the construction algorithm, we revisit the classification operation used in the construction of BSPs and discuss how the symbolic representation can make it more robust. Next we discuss one of the more important aspects of the structure of the KVD, the priority order of insertion of cuts in the tree. This is very important, because it will provide a way to check the correctness of the KVD. We conclude this chapter with a presentation of the construction algorithm and give examples of KVDs obtained for sample scenes.

5.3.1 Classification Operation

The *classification* methods compute the spatial relationship between fragments and hyperplanes, used in many of the KVD algorithms. The existence of different types of fragments for each of the KVD nodes result in new types of classification operations, described in figure 5.2.

The essential computation of the classification operation is the dot product between a point and the normal of a plane equation. Because of numerical imprecision, points that lie in a plane usually return values close but not equal to zero. This imprecision may lead to wrong classification results. In order to fix this problem, classification algorithms use

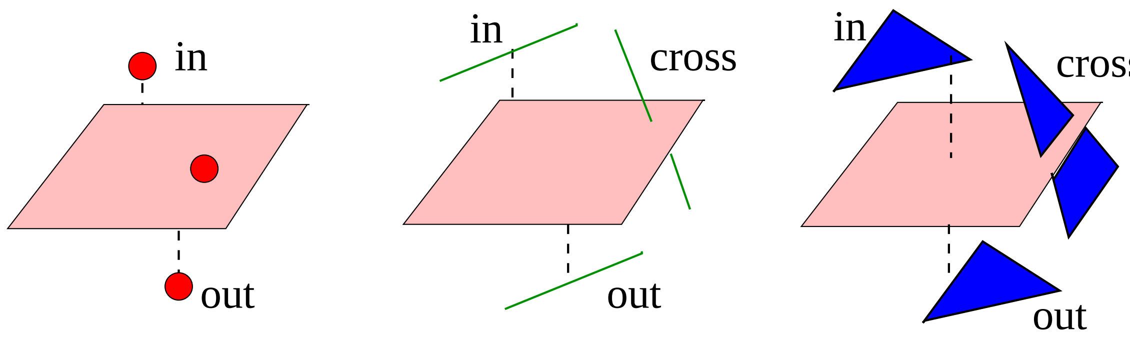


Figure 5.2: Possible Types Of Classification Results

epsilon intervals to evaluate the result of a dot product. A dot product within an epsilon distance to zero are classified as lying on the plane. This solution works reasonably well in practice but requires the specification of an epsilon value, which may need to change depending on the scale of the geometry coordinates of the model. Also, for large models with objects of different scales, more than one epsilon may be required.

In the KVD classification procedures, the fact that we have a symbolic representation of fragments is used to make the classification operation more robust. Because most cuts are defined by points, edges and triangles of the input model, many situations arise where a point is classified against a plane that already contains the point. If the symbolic representation of the point and the plane are compared, cases where a point lies over the plane can be detected without computing the dot product.

5.3.2 Priority Order

Techniques to construct *good* BSPs were briefly discussed in chapter 2. Approximation techniques based on the evaluation of cost models are the preferred choice for selecting cuts to be used in the tree. This solution works well for static BSPs, but for the case of moving geometry it becomes unclear how to allow the cost models to evolve with time and still produce reasonable results.

The randomized approach is another technique used to build BSPs. 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. The construction of randomized BSPs is done by an incremental insertion of triangles following a random order (called priority order). In the case of moving geometry, changing the priority order seems to be the right choice to maintain the expected bounds. The use of a static priority order, however, is showed to produce near optimal results if objects move along pseudo-algebraic trajectories ([2][1]). This observation motivates the use of a randomized approach with static priorities in the KVD.

The use of a static priority order has the additional property of providing a way to verify the correctness of the KVD. Changes in the structure of the KVD are local most of the times, which can be exploited by the algorithms that update the KVD. The use of local updates is always better than a global reconstruction of the tree. However, using local updates creates the need to check that no inconsistencies are created in the tree. An inconsitency may lead to wrong visibility ordering information. Because a static priority order is used, one way to check the correctness of the local updates is to compare the locally modified KVD with a KVD built from scratch for the same geometry. The existence of a mechanism to verify correctness is especially important during the implementation of the KVD.

The static priority order in the KVD is defined for each cut to be inserted in the tree. There are two levels of ordering, one among the triangles, and another among the cuts generated from a triangle. The *triangle priority order (TPO)* is defined for the set of all the input triangles by a random permutation of all triangle indexes. Because scenes can be composed of static and dynamic triangles, static triangles are assigned higher priority values than any of the dynamic triangles. The separation of static and dynamic triangles is important because static triangles will create nodes that will not change, unlike dynamic triangles that will create nodes that cause changes in the tree. We prefer to have static nodes higher in the tree (closer to the root), and dynamic nodes close to the leaves, since movement of nodes almost always require deletions, which are easily performed at the leaves.

The second level of ordering is called the cut priority order. For every triangle seven cuts are introduced in the KVD (three from vertices, three from edges and one from the triangle). The cuts from vertices have the highest priority, and are inserted first in the KVD. Among vertex cuts, the vertex with smallest index has the highest priority. The next cuts in the cut priority order correspond to cuts from edges, and similarly smaller edge indexes have higher priority. Finally, the triangle cut has least priority.

In summary, to retrieve the priority order of a single cut of a triangle the following functions are used:

- Vertex cuts: given vertex index $i (i = 0..2)$ of a triangle t, the priority(t,vertex(t,i)) is equal to 7 * TPO(t) + i.
- Edge cuts: given edge index $i (i = 0..2)$ of a triangle t, the priority(t, edge(t,i)) is equal to 7 * TPO(t) + 3 + i.
- Triangle cuts: given a triangle t, the priority(t) is equal to 7 * TPO(t) + 6.

5.3.3 Construction Algorithm

The construction algorithm performs an incremental insertion of cuts in the tree, based in the triangle and cut priority order. For every cut to be inserted in the KVD, a node is created with all information concerning the cut. The symbolic representation of vertices, edges and planes is used in the representation of hyperplane and fragments of the node. For a point node, both hyperplane and fragments are defined by the index of the vertex used to created the node. For edge cuts, the hyperplane is described by the edge index, while the fragments contain the two vertex endpoints and the edge index. For triangle cuts, the hyperplane is defined by the triangle index, and the fragment is defined by the indexes of the vertices and edges of the triangle. Once all relevant information is stored in a node, the cut is inserted into the tree by filtering its descriptor node in the tree, partitioning the node into additional nodes if necessary, until the leaves of the tree are reached.

For a point node, the process of filtering a node in the tree involves a classification operation of the point that defines the cut against the hyperplanes of nodes in the tree. The result of this operation indicates the subtree where the process will continue, until leaf of the tree is found. Note that in this case no partition happens, and this operation becomes very similar to a *point location* procedure that finds the region of a leaf node that contains the query point.

The insertion of an edge cut has similar behavior, however the fragment now corresponds to a line segment. Unlike point nodes, partitioning may happen because the edge fragment may be cut by hyperplanes of point and edge nodes. The scene is assumed to always have non-intersecting triangles, therefore an edge fragment can not be partitioned by a triangle node. The other cases may still happen because they do not represent an intersection of triangles, but an intersection of the external planes defined by points and edges with the input model.

The partition of an edge fragment by a point node creates two new edge nodes, which contain the representation of the node in each of the halfspaces of the partitioner. The fragment of the original edge node is also partitioned in two, with the creation of an additional vertex (a *thread vertex*).

The case where the partition of the edge fragment is done by an edge node requires special attention. The new vertex to be created in the fragment is an *intersection vertex*. In order to maintain cells with bounded complexity in the vertical decomposition, it is necessary to add point cuts for each intersection vertex created. In this case, two point nodes need to be inserted, corresponding to the same intersection vertex used in the two fragments created by the partition of the edge fragment. Instead of adding these cuts at the time they are created, we postpone their addition until after all types of cuts have been inserted in the tree.

The cut defined by the triangle is inserted after all point and edge nodes. This insertion also may generate partitioning, and new types of vertex may arise. The partition of a fragment is similar to the algorithm described for traditional BSPs in chapter 2, with the difference that the creation of new vertices uses the symbolic representation of vertices and edges.

After cuts are inserted for all triangles, we need to handle the intersection cuts. The fact that intersection cuts are handled at the end is a major difference in the order of insertion of cuts if compared to the approach described in [1], where intersection cuts of a triangle are added after the insertion of the edge cuts of the triangle. The insertion of intersection cuts after all other cuts at the end has the important property of not creating any partitioning in any nodes in the tree. This is an interesting result, because the number of partition operations directly affects the size of the tree. Although the addition of intersection cuts is required by the vertical decomposition, we use the fact that all intersection cuts are stored in the leaves of the tree to not insert them at all in the tree, but to treat them instead as if they were inserted for the events that control the structure of the tree. This implicit representation of intersection cuts is done through the edge nodes, which contain the information in the fragments about all intersection cuts in the tree. The only need to have such nodes in the tree happens when the set of certificates that control the combinatorial structure of the tree is designed. Using the information stored in edge nodes, it is possible to add all certificates that involve intersection cuts without explicitly storing them in the tree.

The construction algorithm can then be summarized as follows:

1. For every triangle in the scene in triangle priority order
    - Insert P-cuts in cut priority order from all vertices of the triangle.
    - Insert E-cuts in cut priority order from all edges of the triangle.
    - Insert T-cut corresponding to the triangle.

5.3.4 KVD structure examples

In this section we present some examples of KVD constructions. In figure 5.3 a step-by-step construction of a KVD for a scene composed of a single triangle is illustrated. The insertion of cuts for the triangle follows the cut priority order, first with point nodes, followed by edge nodes and the triangle node. In figure 5.4 we have a KVD for a scene with three triangles that causes a cycle in the visibility graph. Figures 5.5 and 5.6 illustrate the structure of the KVD for scenes composed of a greater number of triangles.

In figure 5.7 we illustrate the vertical decomposition created for a scene with three triangles that causes a cycle in the visibility graph. In figure 5.8 we give an example that uses a Euclidean point instead of the traditional directions in the vertical decomposition, which creates a decomposition similar to shadow-volume BSP.

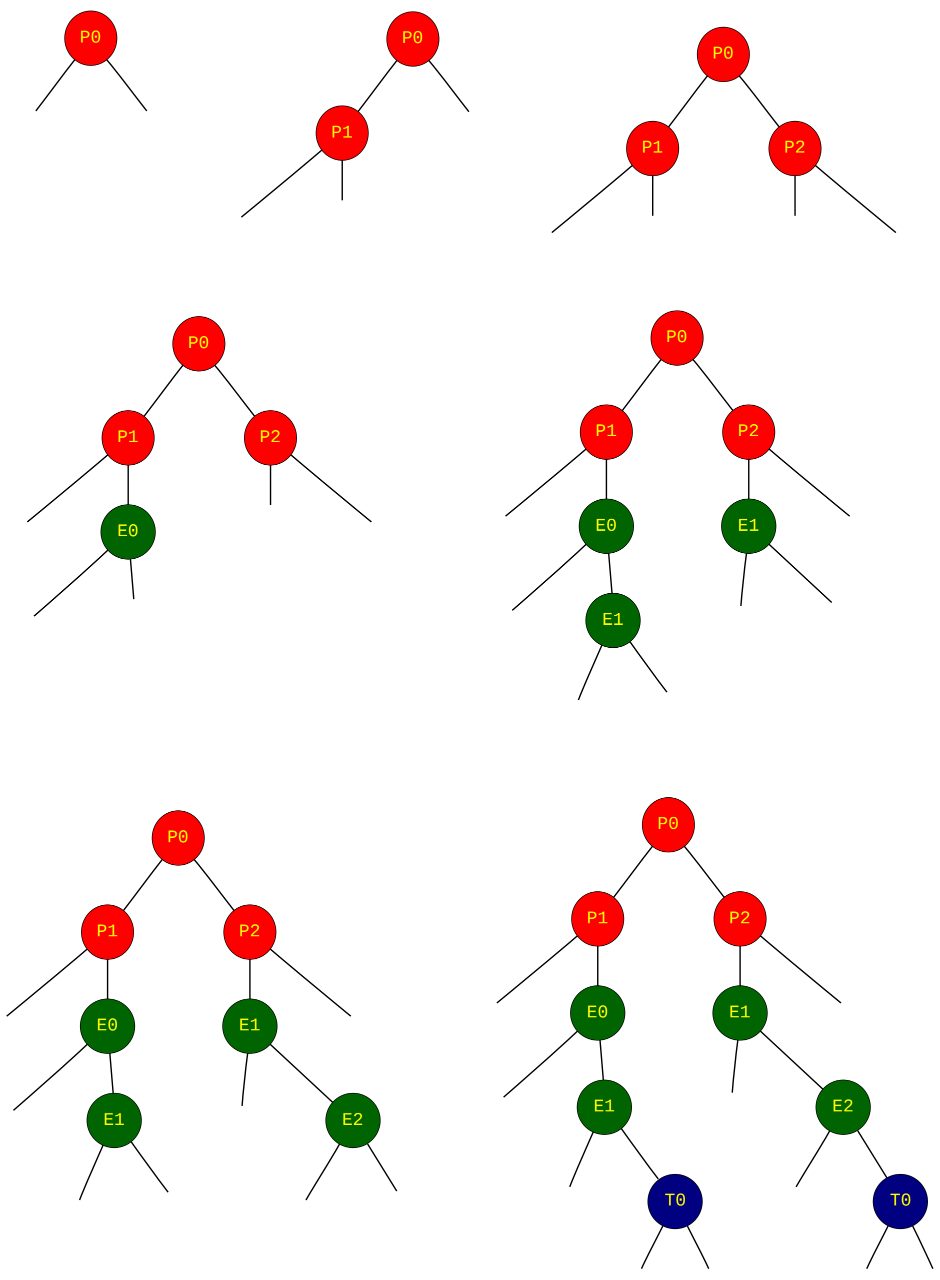


Figure 5.3: Incremental construction of the KVD for a single triangle. Point nodes are the first ones inserted (red nodes), followed by edge nodes (green) and triangle nodes (blue).

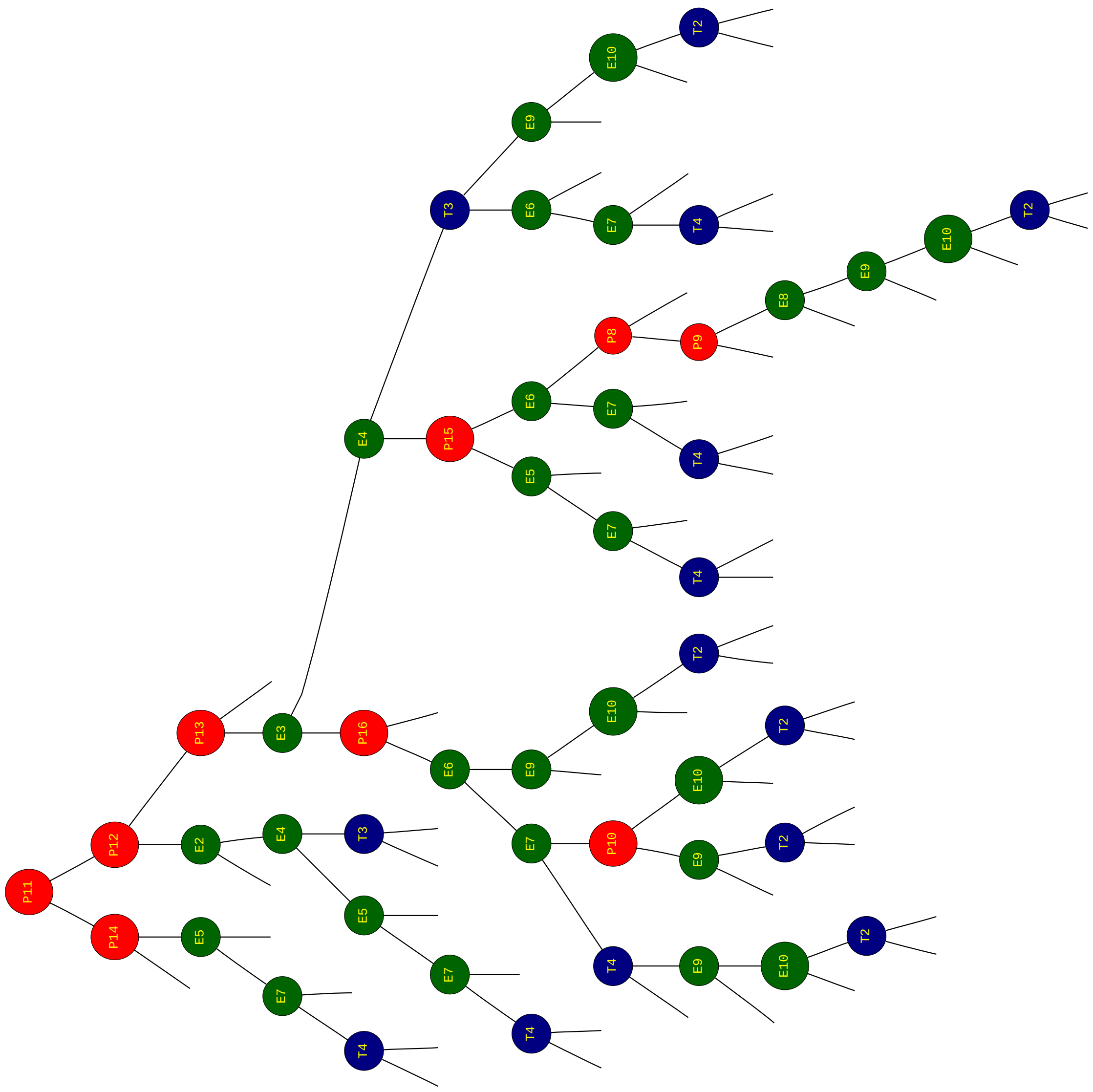


Figure 5.4: KVD tree for a scene with three triangles that cause a cycle in the visibility graph.

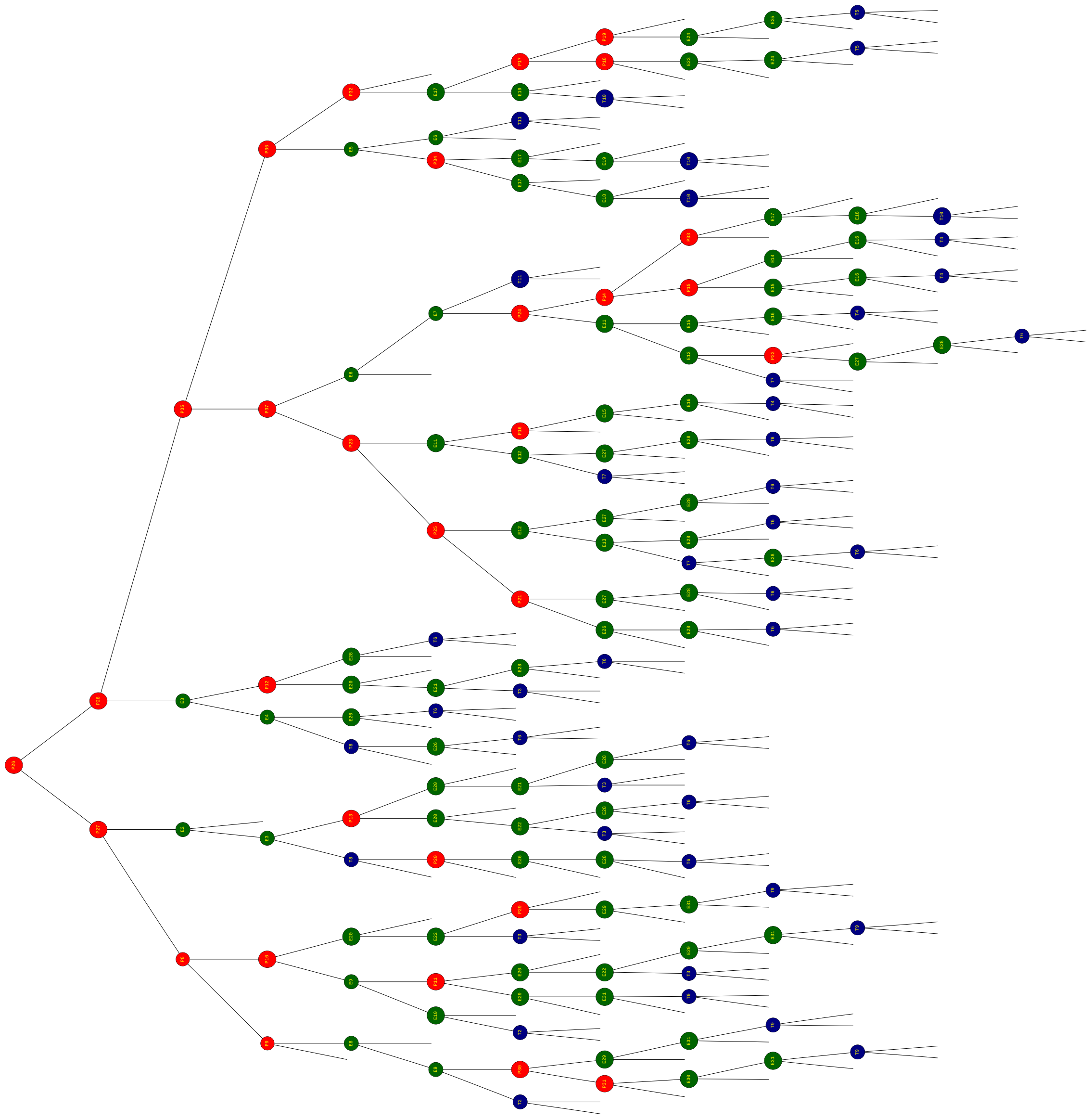


Figure 5.5: KVD for a scene with 10 triangles.

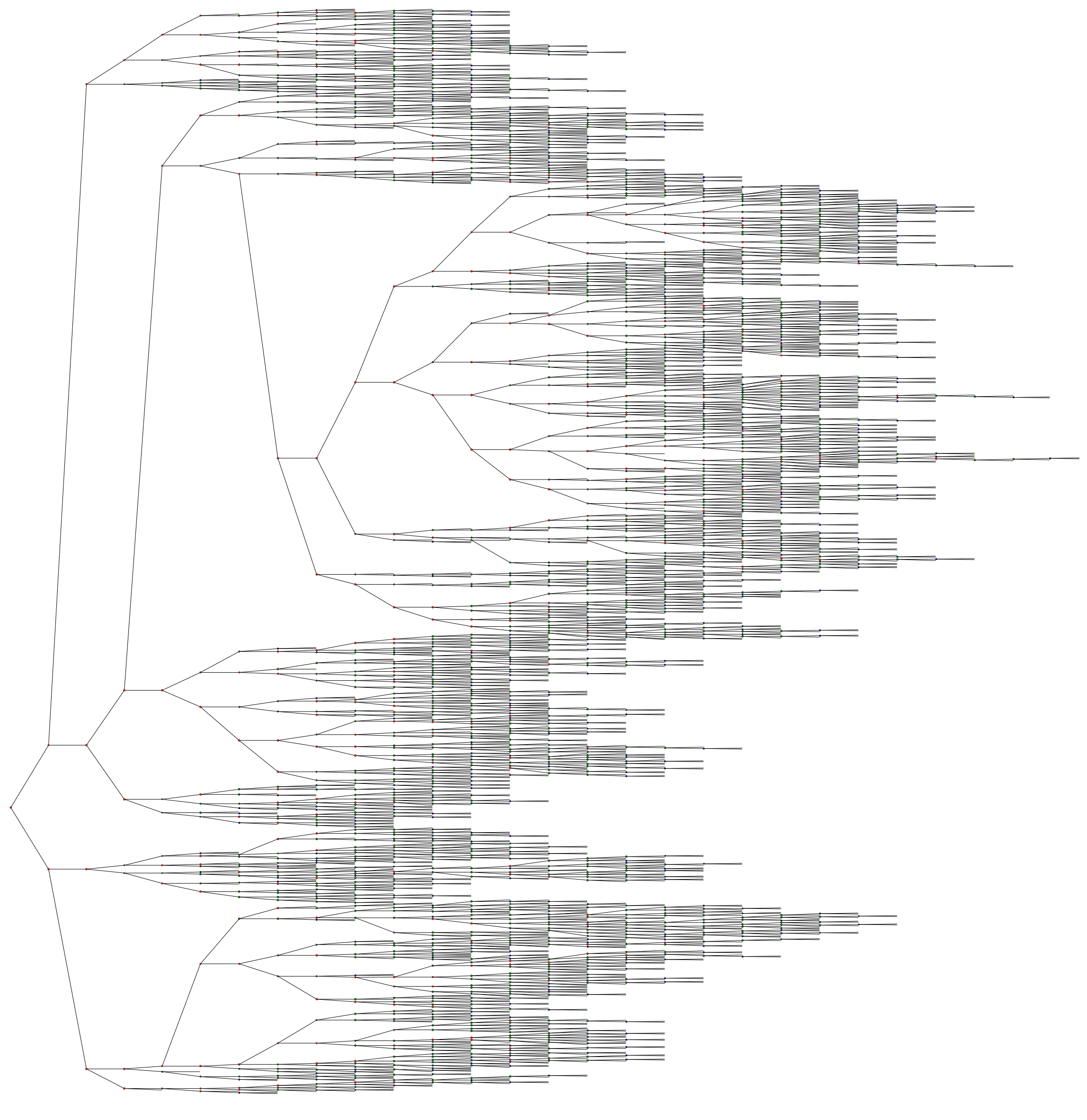
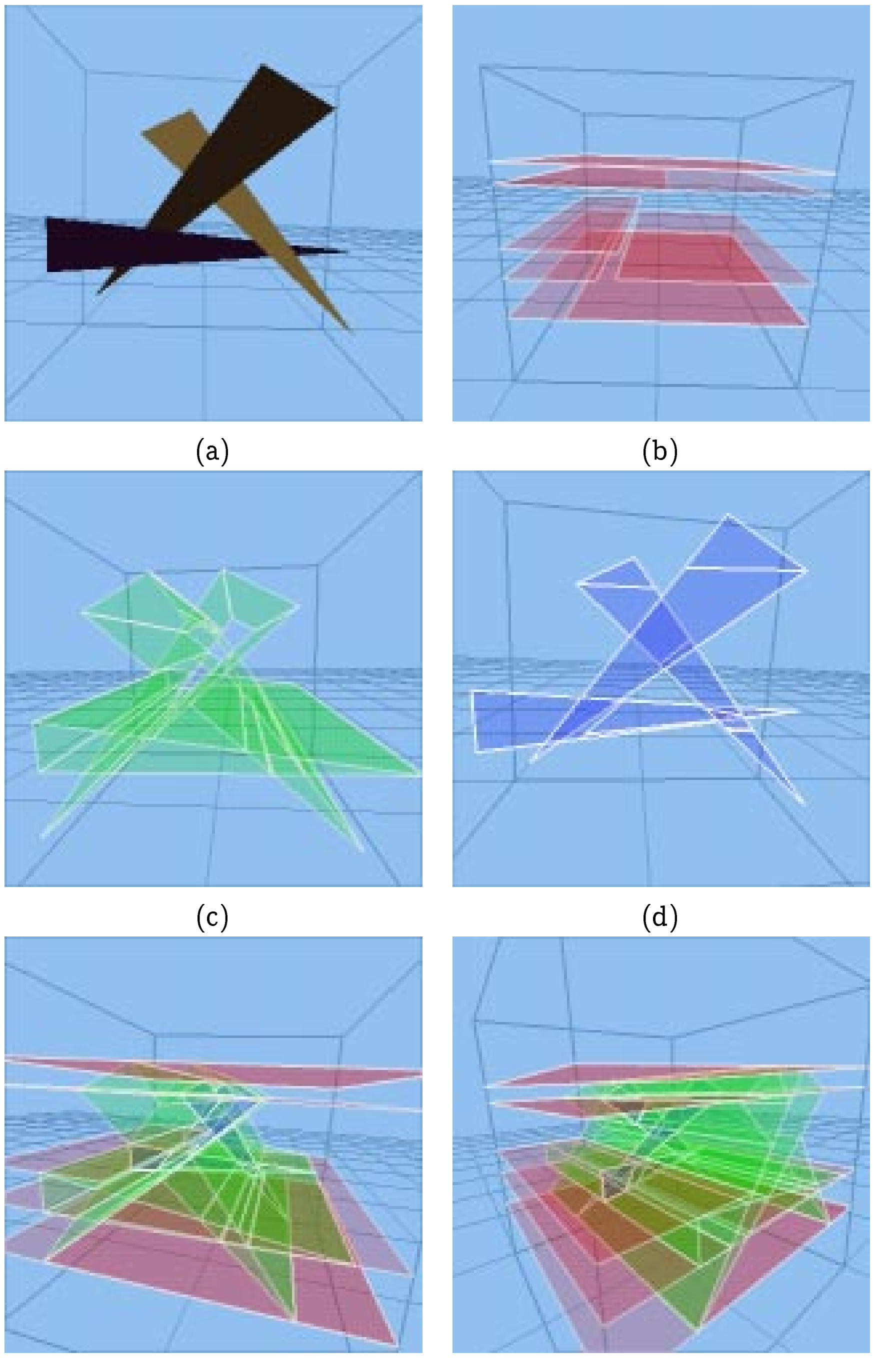


Figure 5.6: KVD for a scene with 100 triangles.

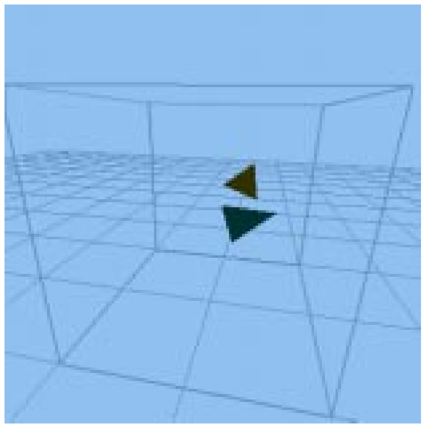


(a) (b)

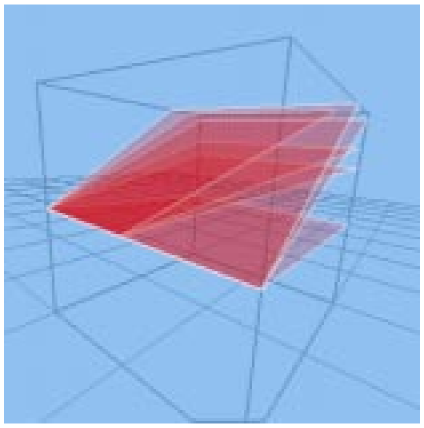
(c) (d)

(e) (f)

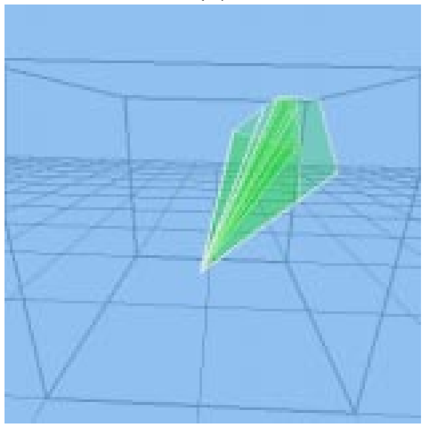
Figure 5.7: KVD decomposition for a scene with three triangles that causes a cycle in the visibility graph.



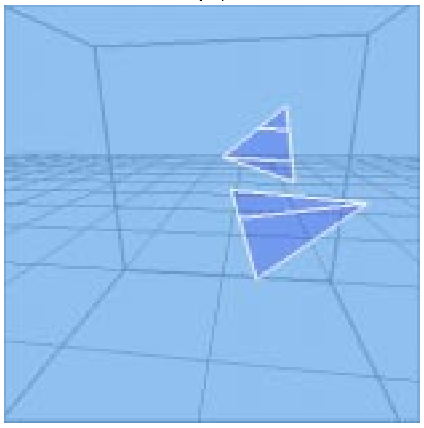
(a)



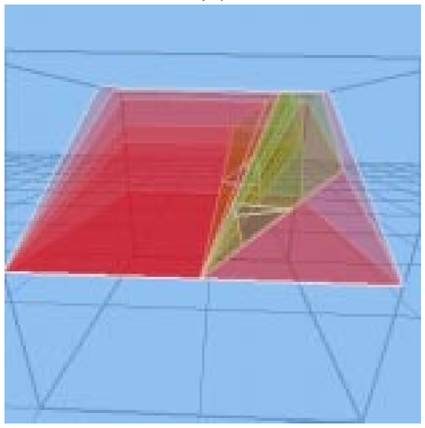
(b)



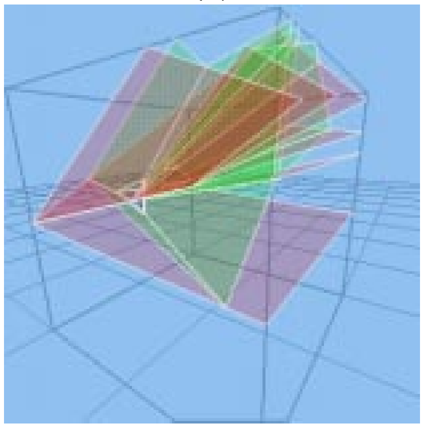
(c)



(d)



(e)



(f)

Figure 5.8: KVD decomposition with for a scene with two triangles with one direction of the vertical decomposition given by a euclidian point.

Chapter 6

KVD Events and Certificates

The *certificates* are one of the most crucial aspects of a kinetic data structure. For the simple case of maintaining a set of moving balls inside a rectangle at all times, the certificates represent a set of conditions that guarantee that the balls stay inside the rectangle. The violation of one of these certificates creates an *event*. This event may require an update of the underlying kinetic data structure, combined with the reconstruction of the set of certificates. In this simple example, changes in the movement of the violating point combined with the update of its certificates suffices. In general, however, events require complex updates in both the data structure and the set of certificates.

The KVD is a special type of BSP that represents a vertical decomposition for a moving set T of triangles in $\mathbb{R}^3$. Every time a triangle moves, the geometry of the cylindrical cells induced by the triangle in the vertical decomposition changes. The topology of these cells, however, may stay the same. The specific time when the topology of the cylindrical cells changes produces an event, which requires an update in the combinatorial structure of the tree.

In this chapter we discuss the kinds of events that require reconstruction of the KVD, and present a set of certificates that are used to compute the time that such events occur. We also discuss practical issues regarding the maintenance of certificates in our current implementation.

6.1 Topological changes in the structure of the KVD

The KVD is a combinatorial structure represented by a binary tree. The construction of the tree involves an incremental insertion of cuts defined by the triangles in the scene. An insertion uses a classification operation, that compares the fragment of the inserted node with hyperplanes of nodes in the tree. The result of every classification operation represents the halfspace(s) occupied by the inserted element. Depending on the classification result, a partition operation that divides the fragment in two may occur. The insertion continues recursively in each subtree of the node with the appropriate fragments obtained from the partitioning step. Because the classification results determine the location of a node in the tree, we conclude that there is a direct correspondence between these classification results and the structure of the tree. We explore this correspondence to define certificates for combinatorial changes in the tree.

Let CL represent the set of all classification operations performed during the construction of the tree. This set is composed by tuples of the format $cl(n, ancestor(n))$, as every node inserted in the tree is classified against one of its ancestors. The KVD has three different types of nodes: P-nodes, E-nodes and T-nodes, which leads to nine different types of tuples: $cl(P,P)$, $cl(P,E)$, $cl(P,T)$, $cl(E,P)$, $cl(E,E)$, $cl(E,T)$, $cl(T,P)$, $cl(T,E)$ and $cl(T,T)$. Because we assume that the priority order of insertion is maintained at all times, the only way that the set CL may change is when one of the classification results is modified. The invariance of the set CL represents a proof that the combinatorial structure of the tree stays the same, which is used as basis for the creation of the certificates.

The enumeration of all classification comparisons used during construction creates a set of certificates that represents the combinatorial structure of the tree. The disadvantage of this approach is the large number of classification results (at most $O(n\log^2 n)$) for a tree of height $O(\log(n))$. We reduce the size of this set using the fact that the structure of the vertical decomposition limits how classification results change.

We illustrate this observation with an example. Suppose we build a KVD using only point cuts. In figure 6.1, we have a simple case with five point cuts, with illustrations of both the decomposition created by the point cuts and the tree structure they create. In this particular situation the tree will only change its combinatorial structure when two P-cuts

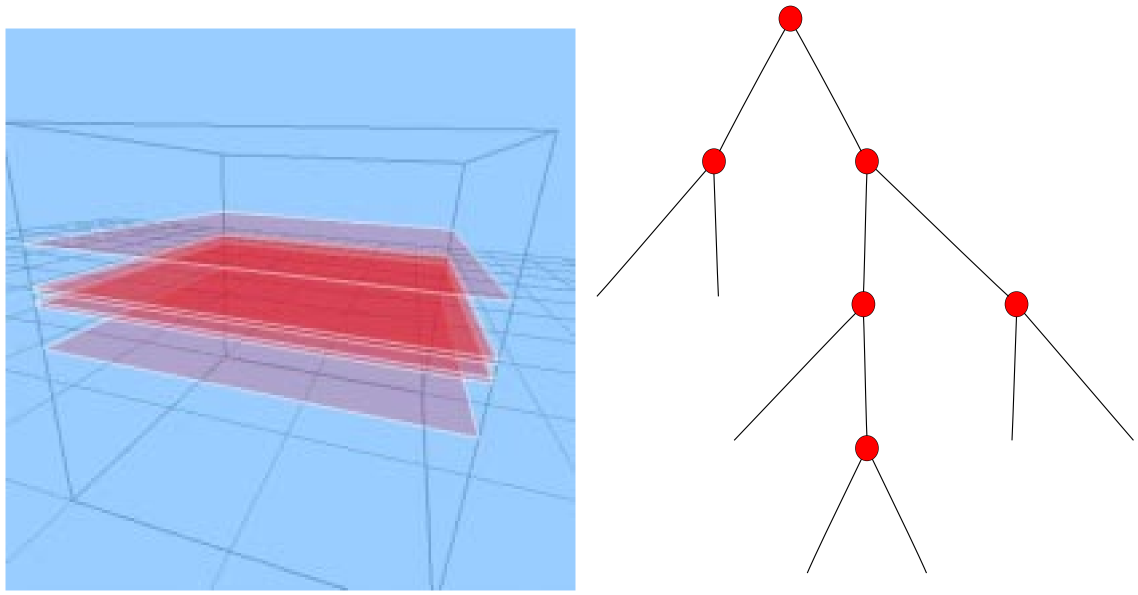


Figure 6.1: KVD containing only point cuts. (a) vertical decomposition, (b) tree structure.

pass through each other. If we enumerate all possible types of classification results computed during the construction of the tree, it would require more results than are actually necessary. For instance, the deepest node in the tree can produce three classification results, while it is easy to see that only two certificates are necessary for each point, containing P-cuts of ancestor nodes directly above and below each point. We conclude from this example that it is possible to explore the additional constraints given by the decomposition to substantially reduce the number of certificates.

In the general case of the vertical decomposition described in previous chapters, the cylindrical cells limit the number of planes that can be crossed by moving geometry (points, edges and triangles). In the KVD, every node in the tree has an associated region (a cylindrical cell), obtained by the intersection of the halfspaces of all ancestors of the node. Because cylindrical cells have bounded complexity (five or six sides), the number of ways that classification results may change is also bounded by the cell complexity. In the figure 6.2 we illustrate the two possible types of cylindrical cells.

Besides the fact that the cylindrical cells have bounded complexity, a clear structure in the types of walls can be seen from the example. For six-sided cells, opposing faces have the same type of cut. In addition , all three different types of cuts (P-cut, E-cut and T-cut)

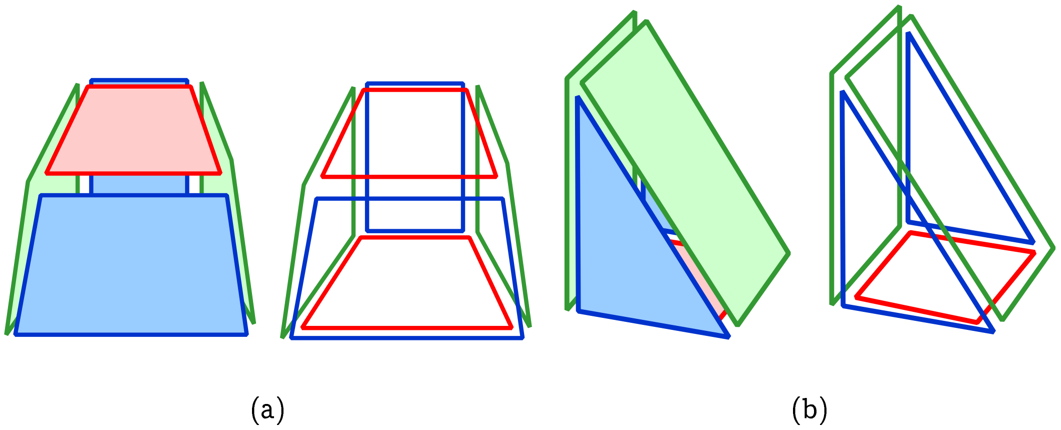


Figure 6.2: KVD cylindrical cells. (a) six-sided cells, (b) five-sided cells

are used, which leads to the existence of at most two faces for each of the possible cuts. For five-sided cells, there is one fewer P-cut, the other faces remain the same as before. This structure in the formation of the cells becomes important because the identification of the closest faces to a given point can be limited to a certain type of cut, as we will see in the definition of certificates.

6.2 KVD Events

In this section we discuss in detail the different types of events caused by changes in classification results. We separate the set of events according to the type of node fragment that is moving: vertex, edge, triangle and intersection events. The intersection events correspond to events that involve intersection points. They are similar to vertex events, but have a different behavior because intersection nodes are not explicitly stored in the tree.

6.2.1 Vertex Events

A vertex event happens when the classification of a point node (defined by a triangle vertex) changes with respect to the hyperplane of an ancestor node. Because the vertical decomposition is composed of cylindrical cells, the ancestors that may cause such vertex event are reduced to the ones that define the walls of the cylindrical cell that encloses the point node.

From this observation we conclude that a vertex event will only happen when the vertex moves through one of the walls of its corresponding cylindrical region. Because the cylindrical cell is composed of only three different types of cuts, the only vertex events that can happen are:

- VV : a vertex crossing a plane defined by a point node. (The point node is defined by a vertex of a triangle).
- VE : a vertex crossing a plane defined by an edge node.
- VT : a vertex crossing a plane define by a triangle node. This is a collision event.

In figure 6.3 we illustrate the types of vertex events. The different cases are described using an illustration in 3D, followed by a 2D illustration that corresponds to the same situation, only projected into one of the walls of the cylindrical region. When drawing certificates, we often use a 2D illustration instead of the 3D counterpart, because these illustrations simplify the discussion while preserving the essential properties of the 3D situation.

6.2.2 Edge events

The edge events correspond to the movement of an edge segment through one of the planes of its cylindrical cell. As with vertex events, the only ancestors that may cause a change in the classification results are the ones defining the enclosing cylindrical cell. The different types of walls of the cylindrical cell define the type of edge events that can happen.

- EV : The edge fragment of an $\mathcal{E}$-node passes through the plane defined by a point node.
- EE : The edge fragment of an $\mathcal{E}$-node passes through the plane defined by an edge node.
- ET : The edge fragment of an $\mathcal{E}$-node passes though the plane of a triangle. This is a collision event.

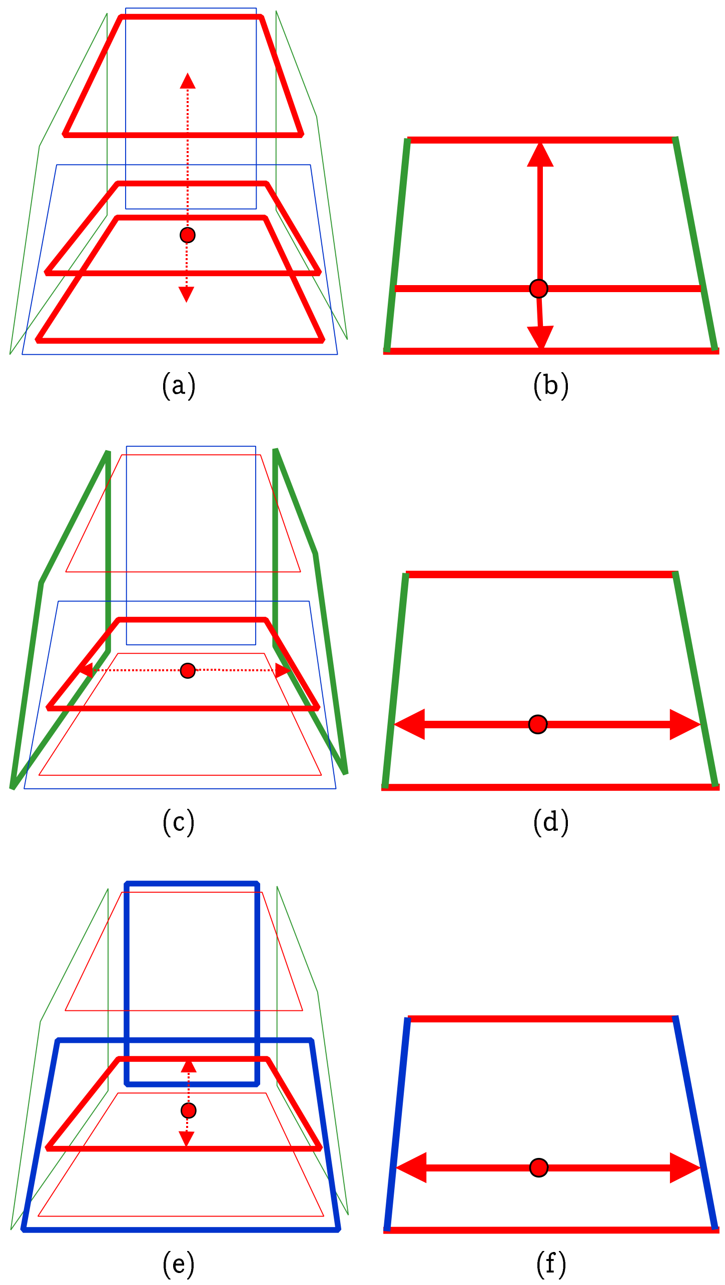


Figure 6.3: Vertex events. (a) VV event in 3D and (b) corresponding 2D view. (c) VE event in 3D and (d) corresponding 2D view. (e) VT event in 3D and (f) corresponding 2D view.

The edge events correspond to the most complex set of events that can change the KVD. Although the classification proposed above express all possible ways that an edge can leave a cylindrical cell, it becomes necessary to detail each of these cases a little further to have a better understanding of all scenarios that can happen. We use an alternate way to classify edge events to discuss more detailed cases, and establish a connection with this previous classification.

A different way to classify the possible ways that an edge can leave a cylindrical cell is to use the last feature of contact with the cell. Let us call the faces, edges and vertices of the enclosing cylindrical cell *c-faces*, *c-edges* and *c-vertices*, respectively. Let the edge partially or completely leaving the cell be called an *exiting edge*. We call the *last feature of contact* ($\mathsf{LFC}(f)$) the feature (face, edge or vertex) of smallest dimension of the cylindrical cell that makes contact with the exiting edge.

Suppose the exiting edge leaves a cylindrical cell by just one c-face. In this case, the LFC is simply a c-face. If the edge leaves by two c-faces and a c-edge, the last feature of contact is called a c-edge, because it leaves the cell by the c-edge that connects the two c-faces. The case of an exiting edge leaving the cylindrical cell by three c-faces has a c-vertex as the last feature of contact, but this case does not happen because of the general position assumption on the trajectories of the objects.

Therefore, only two cases of last feature of contact may happen. The case where the last feature of contact is a c-face creates an event that was already described before in the vertex events. Because the exiting edge leaves by a c-face, the endpoints of the exiting edge also need to cross the exiting face. We assume that this type of edge event is detected by the vertex event that happens at the same time and there is no need to have an additional edge event.

The case where the last feature of contact is a c-edge represents the only new type of event that arises from edge events. Let us call the *previous features of contact* ($\mathsf{PFC}(cf_1, cf_2)$) the two c-faces that are crossed by the exiting edge before it crosses one of the edges of the cylindrical cell. The possible pairs of c-faces in the PFC is limited by the known structure of the walls of the cylindrical cells. For six-sided walls, for instance, no face is adjacent to a wall of the same type, while for five-sided cells only edge walls are adjacent to walls of the same type. Moreover, we assume that no intersections of the triangles themselves happen at any time, therefore a c-face of a T-cut will never be one of the previous regions of contact. The only pairs of previous regions of contact are: PFC(P, P), PFC(P, E) and PFC(E, E).

- PFC(P,P): This corresponds to the EE and ET cases above.
- PFC(P,E): This corresponds to the EV and EE cases above.
- PFC(E,E): This corresponds to the EV and EE cases above.

In figure 6.4 and figure 6.5 we illustrate the possible edge events in cylindrical cell with five and six sides.

6.2.3 Triangle Events

The triangle events that can happen are identified in most situations by the previously described vertex and edge events. Like vertex and edge events, triangle events correspond to all possible changes in classification results of a triangle node against one of its ancestors:

- TV : The fragment of a T-node passes through the plane of a vertex node.
- TE : The fragment of a T-node passes through the plane of an edge node.
- TT : The fragment of a T-node passes through the plane of a triangle node.

The TE and TV events only occur at the same time that one of the vertex or edge events that involve triangle happens. In other words, whenever a triangle passes through the plane of a P-cut or an E-cut, either its vertices or edges will also cross the plane. We assume that the vertex and edge events are responsible for detecting such events.

Because the input is assumed to contain no intersecting triangles, the TT event may only happen after triangles are allowed to intersect. There are two ways that two non-parallel triangles may intersect: a vertex-triangle collision, or an edge-edge collision. We inspect the previous vertex and edge events to check if previously defined events cover all situations of TT events.

The edge-edge collision case is the simplest to evaluate, because the edge events EE and ET include all possible situations that may happen. The vertex-triangle collision is subtle

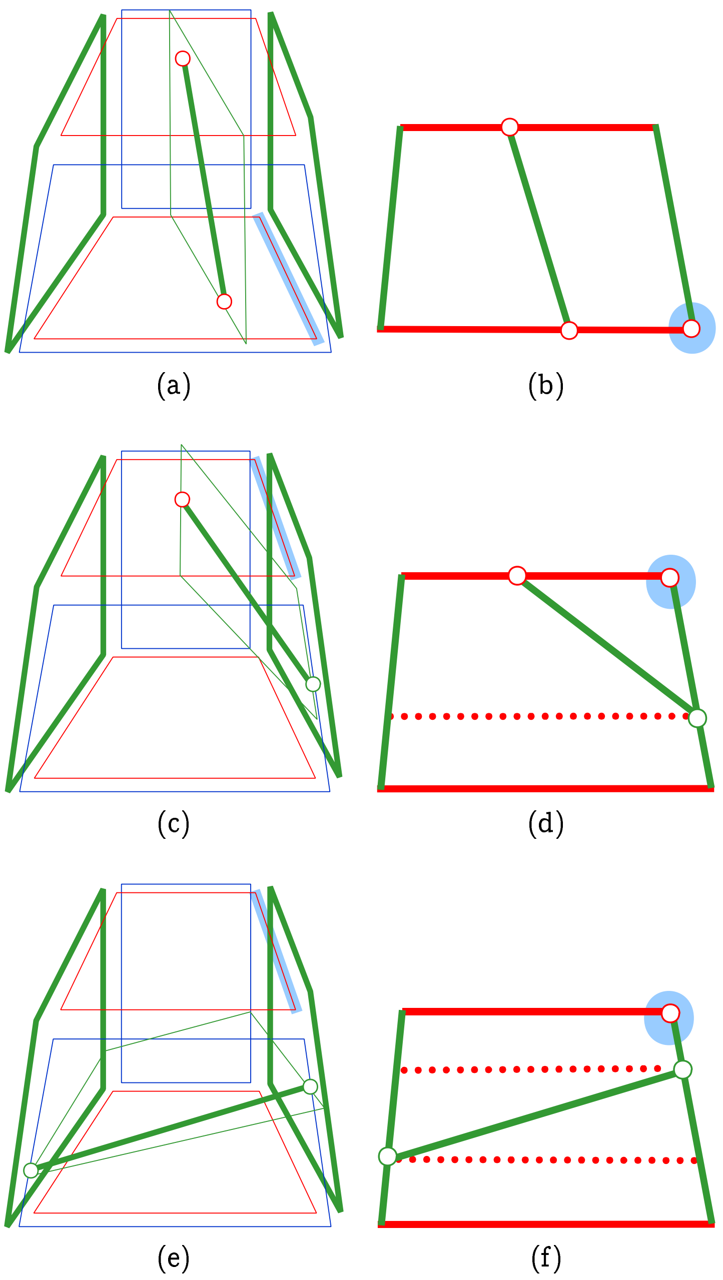


Figure 6.4: Edge events in six-sided cylindrical cells. The edge of the cyindrical cell that is crossed by the exiting edge is highlighted. (a) PFC(P,P) case in 3D and (b) corresponding 2D view. (c) PFC(P,E) case in 3D and (d) corresponding 2D view. (e) PFC(E,E) and corresponding 2D view.

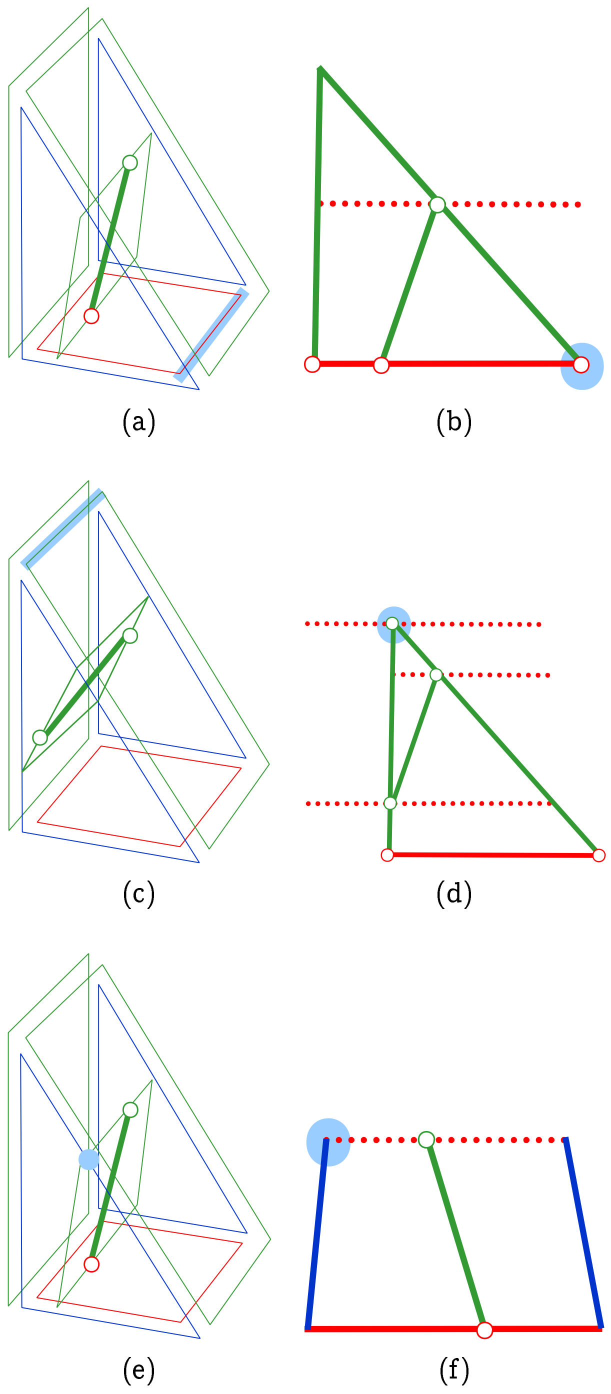


Figure 6.5: Edge events in five-sided cylindrical cells. (a) PFC(P,E) case in 3D and (b) corresponding 2D view. (c) PFC(E,E) case in 3D and (d) corresponding 2D view. (e) PFC(P,P) and corresponding 2D view.

because the vertex event VT only takes care of the case of a lower priority vertex passing through the plane of an ancestor triangle node. For the opposite case of a lower priority triangle passing through a higher priority vertex a new TT event arises. Note that this is not a TV event, because the triangle is not passing through the plane defined by a vertex node, but through the vertex itself. This new situation is the only triangle event that is not covered by any of the events described before, and is described in figure 6.6.

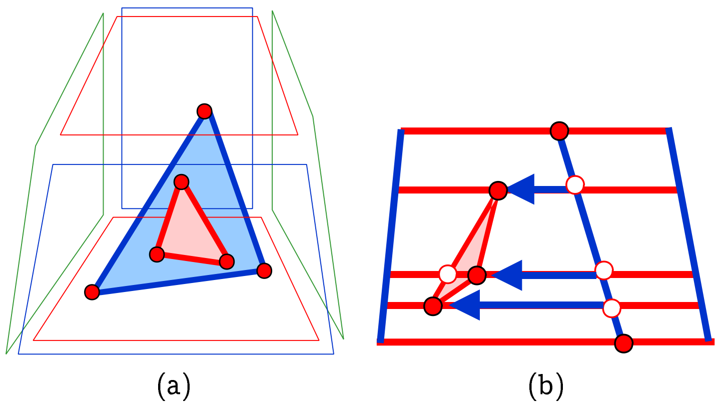


Figure 6.6: TT event cause by a triangle-vertex collision. (a) 3D view and corresponding 2D view.

6.2.4 Intersection Events

In the construction of the vertical decomposition every time an E-cut partitions an edge of the model, additional P-cuts from the intersection point need to be inserted to guarantee that the resulting decomposition has cells of bounded complexity. The cuts defined by the intersection points have the same type of vertex cuts, and are supposed to be inserted in the tree after all other types of cuts. In the current implementation these additional cuts are not stored explicitly in the tree, but it is still necessary to maintain the events they generate (called *intersection* events).

Because intersection cuts can not cut any of the nodes in the tree, the only events that can happen are the ones where an intersection point passes through the cut defined by a vertex or another intersection point. In figure 6.7 we illustrate all possible situations of intersection events, described as follows:

- IV :Intersection point passes through a P-cut defined by another vertex.
- II :Intersection point passes through a P-cut defined by another intersection point.

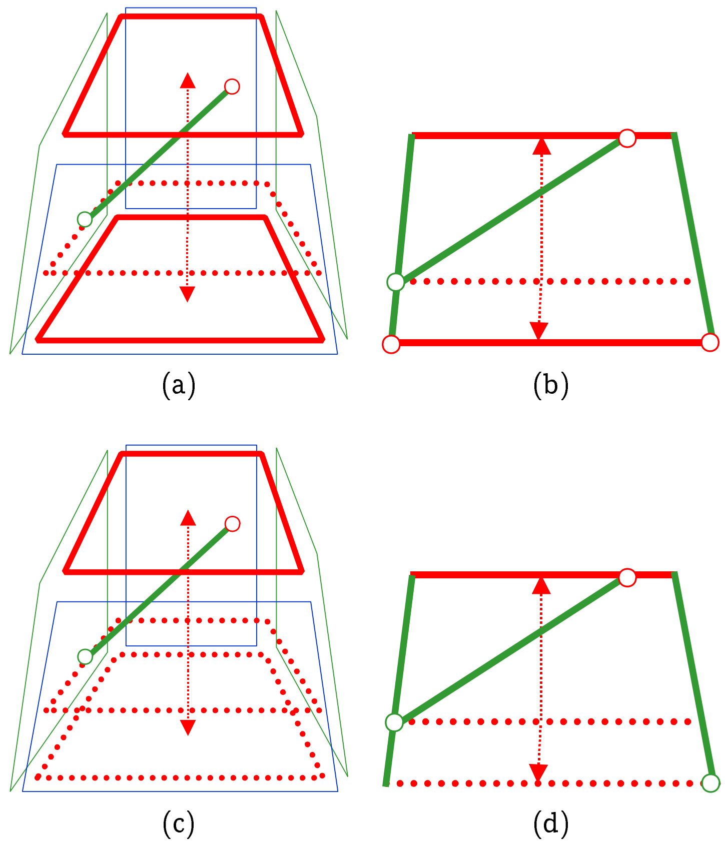


Figure 6.7: Intersection events. (a) 3D view and corresponding 2D view of a IV event. (b) 3D view and corresponding 2D view of an II event.

It is important to note that there is a direct relationship between many of the intersection events and the edge events described before. Unlike the other events, the presentation of intersection events does not rely on the previously defined events. We will explore the connection between intersection and edge events when certificates are defined next in the chapter.

6.3 KVD Certificates

6.3.1 Definitions

The different event types described in the previous section represent all possible situations that can cause a combinatorial change in the structure of the tree. In this section we discuss the creation of certificates that are used to represent all types of events. In addition, the certificates use the equation of motion of the objects to compute critical times when the certificate fails.

The certificates are divided in several categories: PP-certificate, VE-certificate, VT-certificate, ET-certificate, IV-certificate and II-certificate. Each certificate contains information about the nodes in the tree used to define the event they represent. The storage of pointers to nodes in the tree is very important because it allows a local reconstruction of the tree by the update algorithms described in chapter 7. Before describing each of the certificates, we introduce terminology and define useful operations that will be necessary in the presentation.

Let $p(\tau)$ represent a *polynomial equation* of degree n in the variable τ (time) as follows:

$$p(\tau) \equiv a_0 + a_1\tau + a_2\tau^2 + \ldots + a_n\tau^n \tag{6.1}$$

The *motion* of an object in a scene is specified by four polynomial equations, one for each of the i-th coordinates (i=0..3). In the notation used, the x-, y-, z- and w-coordinates corresponds to the 0-, 1-, 2- and 3-rd coordinates. The definition of the motion of an object is expressed as:

$$m_o(\tau) = \{m_o^0(\tau),\ m_o^1(\tau),\ m_o^2(\tau),\ m_o^3(\tau)\} \tag{6.2}$$

Objects are assumed to have rigid motions, and therefore every vertex, edge and triangle of an object n are subject to the same equation of motions of the object where they are defined. Let $v_n^i(\tau)$ represent the polynomial equation of motion of the i-th coordinate (i=0..3) of the n-th vertex. Let $e_n^i(\tau)$ and $t_n^i(\tau)$ represent the polynomial equation of motion of the i-th coordinate of the n-th edge and triangle in a scene. We call $ev(n,k)$ the k-th vertex (k=0..1) of the n-th edge, and $et(n,l)$ the the l-th vertex (l=0..2) of the n-th triangle.

Let $\hat{x}$ and $\hat{z}$ represent the main directions of the vertical decomposition. Because we represent both directions using Plücker coordinates, each of these directions corresponds to a point in $\mathbb{P}^3$. Let $\hat{x}^i(\tau)$ and $\hat{z}^i(\tau)$ represent the i-th polynomial motions of $\hat{x}$ and $\hat{z}$.

In many events it will become important to recover the motion of an intersection point, which corresponds to the point of intersection between the plane defined by an edge cut e_i, and another edge e_j of the scene. Let $s(i,j)$ represent this intersection point. The computation of the equation of motion of $s(i,j)$ needs to take into account the equations of motions of e_i and e_j. We call $s(i,j)^i(\tau)$ the equation of motion of the i-th coordinate of the intersection point.

The fundamental operation used to compute event times is detecting when four moving points become coplanar. One way to compute this time for the moving points v_0, v_1, v_2 and v_3 is to evaluate the following determinant:

$$coplanar(v_1,\ v_2,\ v_3,\ v_4) \;=\; \det \begin{vmatrix} v_1^0(\tau) & v_1^1(\tau) & v_1^2(\tau) & v_1^3(\tau) \\ v_2^0(\tau) & v_2^1(\tau) & v_2^2(\tau) & v_2^3(\tau) \\ v_3^0(\tau) & v_3^1(\tau) & v_3^2(\tau) & v_3^3(\tau) \\ v_4^0(\tau) & v_4^1(\tau) & v_4^2(\tau) & v_4^3(\tau) \end{vmatrix} \tag{6.3}$$

This matrix is composed of elements that are polynomial equations of motion, and its determinant represents a polynomial equation in τ, with the roots corresponding to times where the four points become coplanar. In the discussion of the certificate types, this *coplanar* primitive will be used to compute the death time of each type of certificate.

All certificate classes are derived from a base class that contains shared information among all certificates, described in a C++ class as follows:

```
class KVDcertificate {
public:
  timestamp _deathTime;
private:
  void processDeath();
public:
  // Query methods: access information, like the death time
  // Update methods: update information of private data
  // Death Methods: process actions related with the death of the certificate
  // Display Methods: print or draw in 3D representations of private data
}
```

6.3.2 VV-certificate

The VV-certificate is used to represent the VV event, where a vertex v_1 crosses the plane of an ancestor point node v_2. The time that the event happens correspond exactly to the time that v_1 becomes coplanar with the plane of v_2, which can be formulated as:

$$\triangle_{VV}(v_1, v_2) = \mathrm{coplanar}(v_1, v_2, \hat{x}, \hat{z}) \tag{6.4}$$

The VV-certificate needs to store both point nodes that create the event. It is described by a C++ class as follows:

```
class VVcertificate: public KVDCertificate {
private:
  KVDPointNode *_pLowerPriority, *_pHigherPriority;
public:
  ...
}
```

In figure 6.8 we show a VV-certificate in a simple scene. In order to identify the certificate, a line is drawn over the KVD structure connecting the vertices that define point nodes used in the certificate .

6.3.3 VE-certificate

A VE certificate represents the event of a vertex v_i passing through the plane of an edge node defined by an edge e_j. It can be used to represent the two possible cases when the

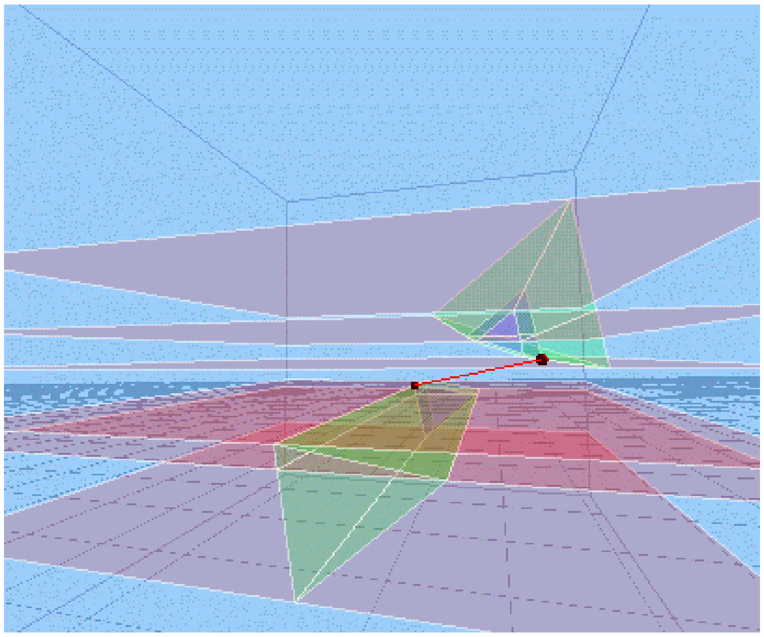


Figure 6.8: VV certificate.

point node has a higher or lower priority than the edge node. The death time of this certificate is given by the following formula:

$$\triangle_{VE}(v_i, e_j) = coplanar(v_i, ev(e_j, 0), ev(e_j, 1), \hat{z}) \tag{6.5}$$

The VE-certificate structure contains both the point and edge nodes that create the event. It is expressed by the following class:

```
class VEcertificate: public KVDCertificate {
private:
  KVDPointNode *_p;
  KVDEdgeNode *_e;
public:
  ...
}
```

In figure 6.9 we show a VE-certificate in a simple scene. In order to identify the certificate, two lines are drawn in the KVD structure connecting the vertex to the endpoints of the edge.

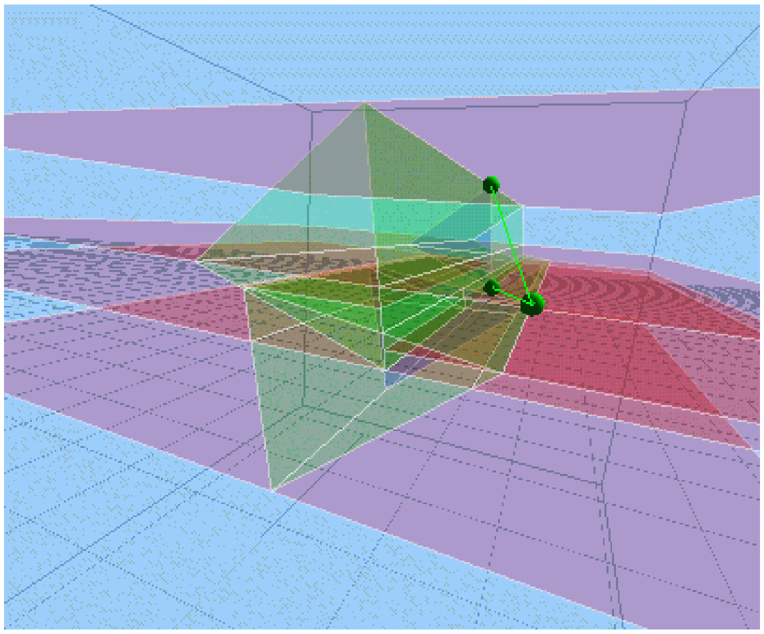


Figure 6.9: VE certificate.

6.3.4 VT-certificate

A VT certificate represents the event of a vertex v_i passing through a triangle plane t_j. It can be used to represent both situations where the vertex has a higher or lower priority than the triangle. The death time of this certificate is given by the following formula:

$$\triangle_{VT}(v_i, t_j) = coplanar(v_i, tv(t_j, 0), tv(t_j, 1), tv(t_j, 2)) \quad (6.6)$$

The VT-certificate contains both the point and triangle nodes. It is expressed by the following class:

```
class VTcertificate: KVDCertificate {
private:
  KVDPointNode *_p;
  KVDTriangleNode *_t;
public:
  ...
}
```

In figure 6.10 we show a VT-certificate in a simple scene. In order to identify the certificate, three lines are drawn in the KVD structure connecting the vertex to the vertices of the triangle.

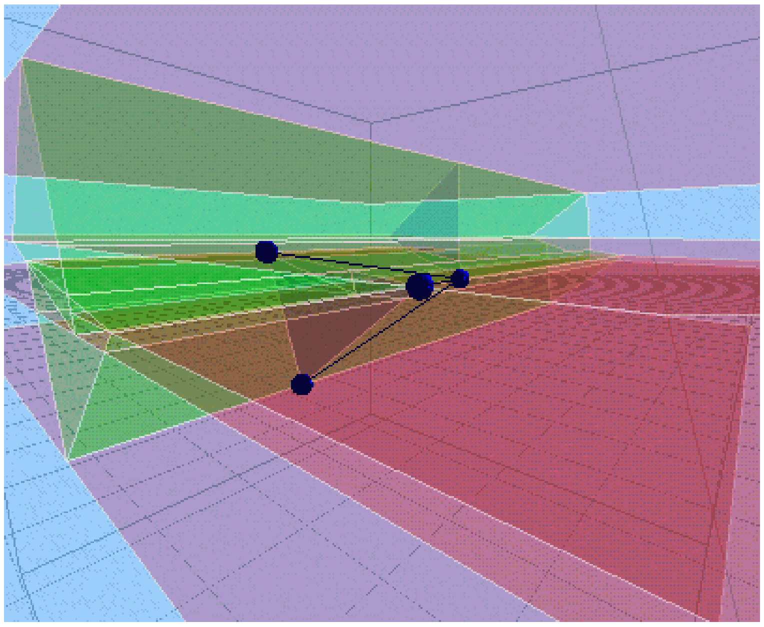


Figure 6.10: VT certificate.

6.3.5 ET-certificate

The ET certificate represents the cases where an edge e_i collides with another triangle by one of its edges e_j. The two edges involved in the collision create an intersection node $s(e_i, e_j)$, that is stored in the edge with lower priority e_i. The death time of this certificate is given by the following formula:

$$\triangle_{ET}(e_i, e_j) \;=\; coplanar(s(e_i, e_j), ev(e_j, 0), ev(e_j, 1),\ \hat{x}) \tag{6.7}$$

The ET-certificate contains both edge nodes (one containing the intersection point node), and is expressed by the following class:

```
class ETcertificate: KVDCertificate {
private:
  KVDEdgeNode *_ei;
  KVDEdgeNode *_ej;
public:
  ...
}
```

In figure 6.11 we show a ET-certificate in a simple scene. In order to identify the certificate, one line connecting the edges that cause the event is drawn in the KVD structure.

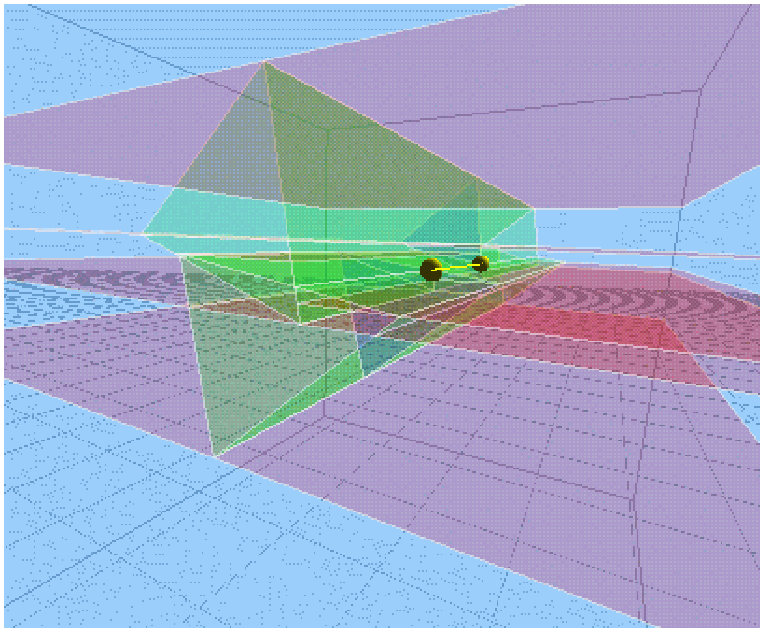


Figure 6.11: ET certificate.

6.3.6 IV-certificate

The IV-certificate is used to represent the case where an intersection point $s(e_u, e_v)$ passes through the plane of a point node v_n defined by a point node. The death time of this certificate is given by the following formula:

$$\triangle_{IV}(e_u, e_v, v_n) \;=\; coplanar(s(e_u, e_v), v_n, \hat{x}, \hat{z}) \tag{6.8}$$

The certificate stores the pointers of edge node that contains the intersection point, and the crossed point node, and is represented by the following class:

```
class IVcertificate {
private:
  KVDPointNode *_p;
  KVDPointNode *_i;
  KVDEdgeNode *_e;
public:
  ...
}
```

6.3.7 II-certificate

The IV-certificate is used to represent the case where an intersection point $s(e_{u1},\ e_{v1})$ passes through the plane of another intersection nodes $s(e_{u2},\ e_{v2})$. The death time of this certificate is given by the following formula:

$$\triangle_{IV}(e_{u1}, e_{v1}, e_{u2}, e_{v2}) \;=\; coplanar(s(e_{u1}, e_{v1}), s(e_{u2}, e_{v2}), \hat{x}, \hat{z})) \tag{6.9}$$

The certificate contain the pointers of both edge nodes where the intersection node are defined, and is represented by a C++ class as follows:

```
class IIcertificate {
private:
  KVDEdgeNode *_eu1;
  KVDEdgeNode *_eu2;
public:
  ...
}
```

6.4 Using certificates to represent events

The certificates are defined to represent all possible events that can happen because of a change in the classification result between a node and of one its ancestors in the tree. In this section we review the events generated for each node in the tree, and explain how the event is detected by one of the certificates described above.

In order to define certificates, the enclosing cylindrical cell of each node needs to be computed. The creation of a certificate involves the location of an ancestor node that defines one of the walls of this enclosing cylindrical cell. The cylindrical cells are not stored explicitly in the tree, therefore it becomes necessary to define a procedure to quickly compute them.

The cylindrical cell has a particular structure that is used to simplify this calculation. The computation of the enclosing region of a node can be done by checking the proximity to ancestor nodes along the six possible directions given by the vertical decomposition: $\hat{x}^+$, $\hat{x}^-$, $\hat{z}^+$, $\hat{z}^-$, $\hat{y}^+$ and $\hat{y}^-$. Because of the way that the walls of the cylindrical cell are defined, point nodes need to be checked for proximity only along the $\hat{y}$ directions, edge nodes along the $\hat{x}$ directions and triangle nodes along the $\hat{z}$ directions.

Based on these observations, a quick computation of the cylindrical cell of a node can be defined. We perform a traversal from the node up in the tree until the root is reached. For each node visited, a distance along a certain direction from the node fragment until the ancestor node fragment is computed. The direction used in this computation is given by the type of the node, as described above. The nodes with minimum distance along the six possible directions are maintained, and at the end of the computation they correspond to the walls of the cylindrical cell.

Let p_u and p_d represent the point nodes that define two of the walls of a six-sided cylindrical cell. For five-sided cells, only one point node appears, and either p_u or p_d is used to represent this wall. For both five- and six-sided cells, let e_l, e_r define the edge nodes and t_f and t_b the triangle nodes that represent the remaining four walls of the cylindrical cell.

For each type of node different events were described, and in the remainder of this section we show how each type of event can be represented using the basic set of certificates.

6.4.1 Representation of Vertex Events

The vertex events of a point node p_n correspond to situations where the vertex that defines the node leaves its cylindrical cell. Three types of events were defined, corresponding to the different types of walls of the cylindrical cell. All of these events can be represented by the following certificates:

- VV events: $VV(p_n, p_u)$ and $VV(p_n, p_d)$
- VE events: $VE(p_n, e_l)$ and $VE(p_n, e_r)$
- VT events: $VT(p_n, t_f)$ and $VT(p_n, t_b)$

6.4.2 Representation of Intersection Events

An intersection node p_i represents information about the intersection of an edge e_1 with the plane defined by another edge node e_2. The intersection node is not inserted as a node in the tree, but instead is stored at the edge node that was used to create it, in this case, p_i is stored at the e_1 node.

Intersection nodes are similar to point nodes defined by vertices, because the planes they define use the same construction. The resulting events that can be created are of only two types: IV and II. For each intersection node, two certificates can be constructed corresponding to the enclosing point node walls.

Because intersection nodes are considered to be implicitly stored at the leaves of the tree, the computation of proximity information is different for intersection nodes. The problem arises because the intersection nodes are stored at edge nodes, and the proximity computation checks the ancestor nodes starting from this edge node. In some cases, the closest wall corresponds to a point node that is not in this path, but in one of the subtrees of the edge node. Note that the proximity computation would not have this problem if the intersection nodes were to be explicitly stored at the leaves of the tree.

The solution to this problem is to change the way that proximity computation is done for point nodes. For every ancestor edge node visited, not only the proximity distance to the edge node is computed, but also the proximity distance for every intersection node stored at the edge node with respect to the point node. Therefore, the right proximity computation for an intersection node is achieved not by the traversal started by the edge node, but by the traversal started by the point node down the tree. This is a global solution, that requires proximity information to be evaluated for every node in a tree or subtree. Because most of the time the certificates need to be computed or updated for entire subtrees, this solution works really well.

Once the closest point nodes along the $\hat{y}$ directions are computed, it remains to construct the certificate. The enclosing point node in one of the directions can be either be defined by a vertex (p_u or p_d), or by another intersection node p_j stored at an edge node e_3, corresponding to intersections with a plane defined by another edge node e_4. For this direction, the certificate is defined based on the type of the point node:

- Vertex point node: $IV(e_1, e_2, p_u)$
- Intersection point node: $II(e_1, e_2, e_3, e_4)$

Events are defined along the other direction in the same way as above.

6.4.3 Representation of Edge Events

The edge events were the most complex types of events described before. We explore the fact that some edge events can be identified by intersection events to substantially reduce the number of edge certificates to be defined.

Let e be an edge node. The edge fragment of e is defined by two endpoints, which can be either a vertex, a thread vertex or an intersection vertex. Let p_i be a point node that we will create depending on the type of the edge endpoint. If the endpoint is a vertex, let p_i represent the point node defined by this vertex. If the endpoint is an intersection vertex, let p_i represent the intersection node stored at e. If the vertex is a thread vertex, let p_i represent the point node used to partition e and create the thread vertex.

One edge certificates is created for each edge endpoint, depending on its type:

- Vertex endpoint: No certificates are defined.
- Intersection endpoint: $\mathsf{ET}(p_i, e)$
- Thread endpoint: $\mathsf{VE}(p_i, e)$

The simplicity of these certificates does not suggest that they can handle all edge events. There are cases that are not detected by these certificates, but because they are detected by certificates defined by intersection nodes, there is no need to create additional certificates.

The possible cases of edge events were described in figures 6.4 and 6.5. For the cases where the previous feature of contact is of types PFC(P,E) or PFC(E,E) (figures 6.4(c)(e), 6.5(a)(c)), an intersection node is involved. In this case, an edge fragment passes through an edge or point wall at the same time that an intersection wall passes through points or intersection walls, and therefore these events are detected by the intersection certificates described before.

The type of event defined by a previous feature of contact PFC(P,P) can also be handled by the previous certificates, but the justification is more involved. If the edge passes through a triangle, either the endpoint of the edge crosses the triangle, or two edges collide. The first case is detected by one of the previous vertex certificates. In the edge-edge collision, it must be the case that an intersection node is defined by the intersection of the edges, and therefore the ET certificate described above for the edge handles this situation.

If the edge e passes through an edge wall defined by another edge node e_2, than this case will only be missed by the intersection events if the two edges do not intersect in an adjacent cell. This situation is illustrated in figure 6.12(b). If they do intersect (figure 6.12(a)), the intersection event in the adjacent cell happens at the same time that the edge leaves the cell. Like before, this intersection event is used to define this edge event.

If the edges do not intersect, then it must be the case that e is passing through one of the endpoints of e_2. Because the endpoint of e_2 cuts the edge fragment of an edge, a thread endpoint is defined and the VE certificate described above for the edge event covers this situation.

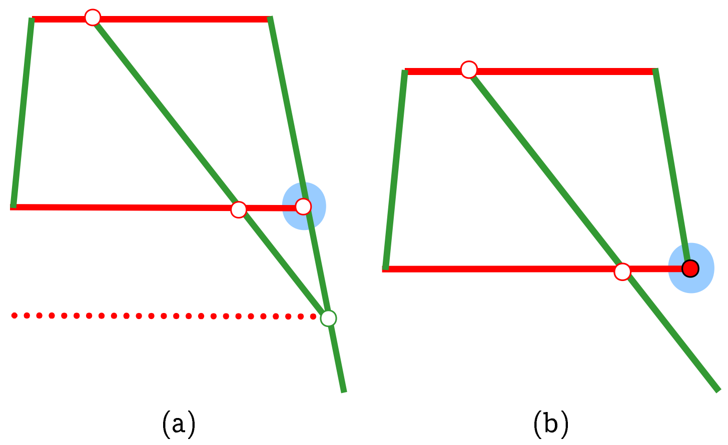


Figure 6.12: Cases that can happen for the PPC(P,P) case and the edge passing through an edge wall.

6.4.4 Representation of Triangle Events

The only triangle event defined (TT) represented the situation where the triangle would pass through a vertex that defines an ancestor point node. It must be the case that the vertex partitions the triangle, and therefore it belongs to the triangle fragment associated with the triangle node. More specifically, the vertex is used to define an edge of the triangle fragment, corresponding to the intersection of the triangle with the plane defined by the vertex node. Because of the structure of the cylindrical cells, such edges can appear only twice in each triangle fragment, corresponding to two vertex nodes.

The triangle certificates of a triangle node t are defined by inspecting its triangle fragment, and creating one certificate for each cutting point node p_u and p_d:

- $\mathsf{VT}(t, p_u)$
- $\mathsf{VT}(t, p_d)$

If the cutting vertex is one of the other vertices of the triangle, then no certificate needs to be created.

6.5 Kinetic Priority Queue

The set of certificates stored at the nodes of the tree gives a way to detect combinatorial changes in the KVD. If the equations of motion of the objects are known a priori, it is possible to establish event times (or death times) where a given certificate will fail. In order to maintain the correctness of the KVD at all times, it is necessary to quickly recover the time that the first certificate will fail, and process the corresponding changes in the tree. Eventually, old certificates will expire and need to be marked as invalid after these updates are performed, which may to lead to the creation of new certificates or deletion of old ones.

The priority queue is an efficient tree structure to recover the minimum value of a set of n elements in $O(\log n)$ time. In the problem of finding the first certificate to fail, a priority queue that contains death times as values can be constructed. The minimum element of this priority queue will correspond to the death time of the first certificate to fail. Because the certificates change in a kinetic way, the priority queue only needs to be updated when certificates fail. The moment a certificate fails requires deletion of expired certificates and insertion of new certificates in the tree. The deletion step is the most difficult, because it requires finding the location of the expired certificate in the priority queue. Locating a certificate in the priority tree can be done either through an ordinary search, or by keeping pointers directly from the certificate structure into the priority queue nodes.

For other kinetic problems this solution works really well. In our problem, however, we explore the fact that the kinetic structure that we maintain is itself a binary tree. We define a new structure, called *Kinetic Priority Queue(KPQ)*, that uses the tree structure of the KVD as the supporting tree structure of the priority queue. The KPQ stores at each node the minimum certificates among all certificates defined at the node and in all nodes of its subtrees. Following this construction, the root of the tree contains the minimum certificates of the entire tree, which correspond to the first certificates to fail.

Because changes to be performed in the tree are local, only the nodes in the tree affected by changes need to have their certificates updated. Once all certificates for affected nodes are computed, the minimum certificates are updated for every node that belongs to a path from an affected node to the root of the tree. We call this process a *kinetic tournament*, because we confront the certificates of a node with the certificates of its parents, and the minimum certificates (winners) are propagated up in the tree.

The integration of the KVD with a priority queue has several advantages. The identification of which certificates need to be deleted is done naturally by the update algorithms. Every time a change happens, the certificate that fails contains pointers to the nodes that caused the changes in the tree. The affected nodes can be identified during this update, and a reconstruction of their certificates is requested. Another advantage is that a single tree structure is maintained, and therefore no additional tree re-balancing is necessary.

In addition to ordinary priority queue operations, the KPQ maintains all elements within an ϵ distance of the minimum. This is important because it is possible to have more than one certificate failure at a given time. More specifically, there is the possibility of many certificate failures at exactly the same time. In order to guarantee that numerical imprecision in the calculation of certificate death times results in processing distinct events as though they are simultaneous events, we maintain not only the first certificate to fail, but a set of certificates within an ϵ range from the first one. The maintenance of a set of first certificates allows a scheduling algorithm to combine multiple events into one.

Before describing the set of minimum certificates stored at each node, we first define the notion of an ϵ-minimum is defined to an ordered set T as follows:

$$\epsilon min(T) \ = \ min(T) \ \cup \ \{t \mid t \in T \ and \ (t - min(T) \ < \ \epsilon)\} \tag{6.10}$$

The ϵ-minimum of an ordered set is used to define the ϵ-minimum of a node in the tree. Let us call $certificates(n)$ the ordered set of all certificates associated with a given node in the tree, computed as described in the previous section. The ordering relation for this set corresponds to the order defined by the certificate death times. The ϵ-minimum of a node in the tree is computed from the set of certificates stored at the node, together with all certificates stored in the subtrees of the node. We use the following recursive definition of an ϵ-minimum of a node n:

- $n = \mathrm{NULL} \rightarrow \quad \epsilon min(n) = \emptyset.$

- $n \neq \mathrm{NULL} \rightarrow \quad \epsilon min(n) = \epsilon min(\ \epsilon min(left(n)) \cup \epsilon min(right(n)) \cup certificates(n))$

The resulting ϵ-minimum set of a node contains the minimum certificates within an ϵ distance of all certificates stored at all nodes in the subtree rooted at the given node. Because we want to maintain these sets for all nodes in the tree, a post-order traversal of the tree accomplishes the task by computing first the ϵ-minimum sets for nodes close to the leaves. The computation for nodes higher in the tree uses the information about ϵ-minimum computed and stored at the subtrees.

Chapter 7

KVD Update Algorithms

The topological change in the cells of the KVD requires an update in the structure of the tree. This update usually involves the movement of nodes in the tree, with insertion of new nodes and removal of old ones. The fact that additional cuts from edges and vertices are introduced in the KVD for every triangle makes the update more complex than traditional BSPs. The complexity arises because all nodes originated from a single triangle must preserve the incidence relations defined in the topology of the input model. For instance, a point node defined by a vertex v of the input model has incident edge nodes defined by all edges e_i incident to v in the scene. Every movement of a node in the tree needs to take into account the incidence relations, which may cause the additional movement of incident nodes.

Another important aspect to consider in updates is the fact that the priority order assigned to triangles remains unchanged during the kinetic simulation. In chapter 5, the priority ordering was used to define the order of insertion of cuts in the KVD. The maintenance of the priority order at all times can be used as a way to check the correctness of the KVD after local updates are processed, which is extremely useful during debugging stages of the implementation. The KVD obtained must be equivalent to one built from scratch using the new geometric information of the scene. Besides the correctness aspect, the use of a fixed priority order can be used to claim several performance bounds of the algorithm, related to the depth of the tree, size and number of events.

The update of the KVD following a fixed priority scheme creates a new behavior in certain tree operations. During the first construction of the KVD, triangles were inserted in priority order and partition operations only occured for elements being inserted in the tree, and not for elements already at the tree. Consequently, inserted elements would always be stored at the leaves of the tree. The subsequent insertions due to the movement of nodes can be more complex because partitions may occur for nodes already in the tree, as a consequence of the priority preservation policy. This new behavior affects operations like the merging of trees, and all insertion operations.

In this chapter we discuss algorithms that perform updates in the structure of the tree. We first review the consequences of the movement of nodes in the tree and describe actions to be performed for each situation. A new insertion operation that takes into account priority orders is described. A new BSP operation, called the *dragging* of a tree, is presented to accomplish the deletion of moving point nodes. Besides the merging of subtrees, this operation also checks the nodes affected by the moving node and perform appropriate actions, which may require additional deletions, or insertion of new nodes in another locations in the tree.

Once the new set of operations is presented, we discuss the update algorithms for each type of certificate presented in chapter 6. Three algorithms are sufficient to perform all types of updates in the tree: V-update, E-update and X-collide. Each presentation discusses simple examples that illustrate the necessary changes both in the topology of the cells and in the structure of the tree, and concludes with the description of the algorithm. For simplicity, we illustrate the changes in the subdivision with figures in the plane, as the extension to three dimensions is straightforward and not necessary for the understanding of the situation.

7.1 Update Effects in the Tree

The update of the KVD is necessary when certificates fail, which corresponds to a node moving across the hyperplane of one of its ancestors. This situation requires the deletion of a node from its current location, and insertion into the other subtree of the parent node that contains the crossed hyperplane. Unlike traditional BSPs, the insertion and deletion operations to be performed have a more complex behaviour.

For the insertion operation, the difficulty arises because a priority order is preserved at all times. As a result, the insertion of nodes are not anymore guaranteed to be at the leaves of the tree. If a node is inserted into a subtree that contains lower priority nodes, the preservation of priority order will require that this node be inserted at a location that no node with lower priority is one of its ancestors. The subtree of lower priority nodes is then replaced by the inserted node, and its subtrees are computed using a tree partitioning operation.

An additional difficulty in these operations arises because the certificates were designed to allow vertex events to also detect some edge and triangle events, and to allow edge events to also detect some triangle events. The updates required in the tree when a vertex event happens are not only accomplished by the updates caused by this event, but also by related edge and triangle events. Therefore, the movement of a single node in a vertex event is not enough to update the tree, but it becomes necessary to look at incident edge and triangle nodes and decide which actions need to be taken. Because point nodes are inserted before other types of nodes, it suffices to check the subtrees of a point node to find its incident edge and triangle nodes. In addition, not only incident nodes are affected by a deletion, but all nodes that are partitioned by the deleted node. For all these nodes, an update of their fragments is necessary to reflect the removal of the partition caused by the deleted node.

In figure 7.1 we have an example of a *VV*-event, which is defined by a point node p_2 crossing the plane defined by an ancestor point node p_1. The initial configuration is described in figure 7.1(a), with the regions corresponding to the two subtrees of p_2 drawn with different colors. The first step in processing the update caused by the movement of p_2 is described in figure 7.1(b), which shows the configuration with the removal of the cut introduced by p_2. Note that the nodes in the orange subtree were all removed, because the corresponding region disappears when p_2 passes through p_1. In figure 7.1(c) we insert p_2 into its new location, which partitions some of the nodes of this subtree (the edge $e3a$ for example is split in two). Finally, the incident edge node that moved together with p_2 is inserted into the configuration (figure 7.1(d)).

The incident nodes play an important role in the updates in the tree, depending on the effect that the movement causes over them. Only two possible effects are identified on

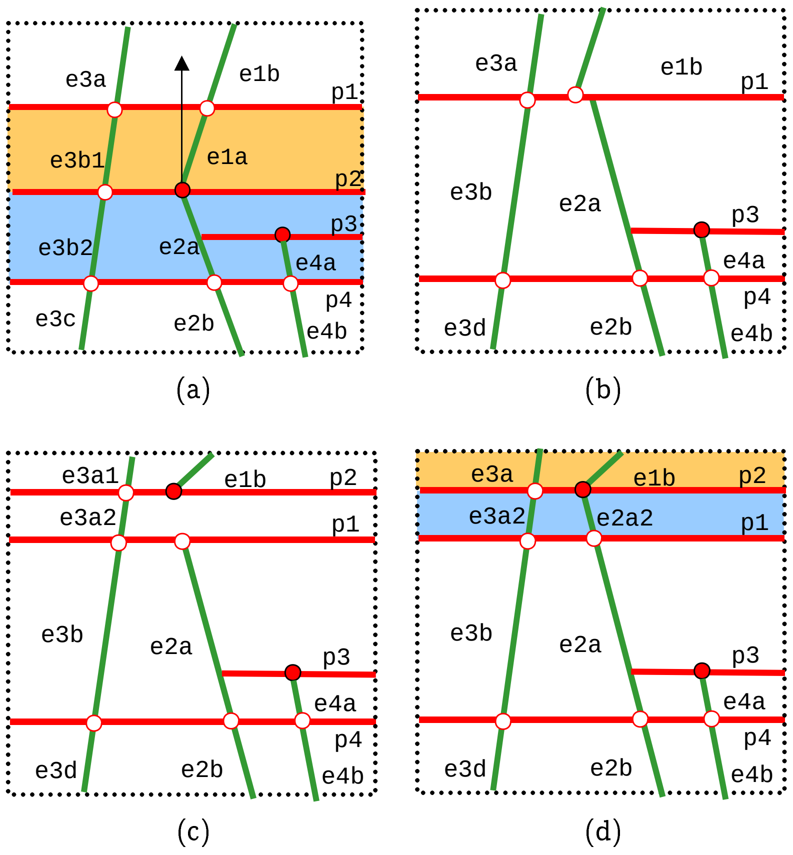


Figure 7.1: Sample example describing updates in a VV-event. (a) Initial configuration. (b) Configuration after removing point node. (c) Insertion of point node in its new location. (d) Insertion of incident nodes in new location.

incident nodes:

- EFFECT_MERGE: The node needs to be deleted from the tree, because it is going to be merged into an adjacent node.
- EFFECT_SPLIT: The node is split in two due to a partition operation. The old node is re-used and stays in the original tree, while the new node is inserted into another subtree (the same that now contains the original moving node).

The behavior of incident nodes can be used to guide the updates in the tree because they encode additional information about nodes, other than the moving node, that need to be deleted or inserted in the tree. The evaluation of incident nodes can be done while processing the changes caused in the tree by the moving node, because all incident nodes are contained in the subtrees of the moving node. One approach to performing this evaluation is to include it inside the merging algorithm that combines the subtrees of the moving node. This more complex operation becomes capable of not only merging two trees, but also deleting the merge nodes, and inserting split nodes into another places in the tree. This new operation is called a *dragging* of a tree and is described in more detail later.

7.2 Extended Tree Operations

7.2.1 Priority-Based Merging of Trees

The movement of nodes in the tree requires the deletion and insertion of nodes. In the deletion case, a node can be easily removed if both of its subtrees are empty, by simply assigning an empty subtree to the parent of the node. Even if one of the subtrees is empty the deletion is trivial, because the node can be removed and replaced by the non-empty subtree. The complex deletion case happens when both subtrees are not empty, which requires the *merging* of the subtrees and the assignment of the resulting merged tree to the parent of the node.

The merging process takes two subtrees t_1 and t_2 and returns a merged subtree t_m. In classic BSPs, the merging of trees is a very useful operation to combine trees using boolean operations, like union, intersection or difference, which can be used in solid modeling applications like Constructive Solid Geometry (CSG). Most merging algorithms described for classic BSPs maintain the structure of t_1 unchanged, while inserting each element of t_2 into t_1. This process is usually referred to as inserting a tree into another tree.

The merging operation has a slightly different behavior when priorities are taken into account. In general, we do not keep one of the subtrees unchanged, but instead we compare the priorities of nodes to decide which node will be used in each step of the merging algorithm. For the nodes n_1 of t_1 and n_2 of t_2, we decide which one has higher priority, and use it as the root of the new merged tree. If n_1 is the node with higher priority, the process will continue to build the new left and right subtrees of n_1. This requires the partitioning (or splitting) of the tree t_2 by the hyperplane defined by n_1. In the case that nodes have the same priority, the nodes correspond to the same element but with different fragments. The solution is to join the nodes into a single one, while combining their fragments.

The code for the priority based merging algorithm is described in figure 7.2. The paremeters to this procedure consits of two pointers to root nodes of the trees to be merged.

```
KVDTree* KVDTree::priorityMerging(KVDTree *t1, KVDTree *t2)
{
  if (t1 == NULL) return t2;
  if (t2 == NULL) return t1;
  switch (t1.comparePriorities(t2)) {
  case PRIORITY_LOWER:
    switch (t1.classifyNode(t2))
    case CL_IN:
      t1.assignLeft(priorityMerging(t1.left(), t2); break;
    case CL_OUT:
      t1.assignT2(priorityMerging(left.right(), t2); break;
    case CL_CROSSING:
      t2.splitTree(t1, newLeft, newRight);
      t1.assignLeft(priorityMerging(t1.left(), newLeft);
      t1.assignRight(priorityMerging(t1.right(), newRight); break;
    }
    return t1;
  case PRIORITY_HIGHER:
    // Similar to previous case, replacing t1 by t2 and vice-versa
  case PRIORITY_EQUAL:
    KVDTree oldT1 = t2.left(), oldRight = t2.right();
    t1.mergeNode(t2);
    t1.assignLeft(mergeTrees(t1.left(), OldLeft));
    t1.assignRight(mergeTrees(t1.right(), OldRight));
    return t1;
  }
}
```

Figure 7.2: Priority-Based Merging

7.2.2 Out-Of-Order Insertion of Nodes

The movement of nodes across subtrees may require the insertion of a node in a subtree that contains nodes of lower priority. In order to preserve the priority order of insertion, it is necessary to re-arrange the lower priority nodes to be descendants of the higher priority node. This was not a problem the first time that the tree was built, because nodes were inserted in priority order, and therefore new nodes would always go to the leaves of the tree.

The insertions of nodes that do not follow the priority order are called *out-of-order* insertions. The insertion of a node n_h under these new circumstances is similar to the traditional insertion method until a node n_l with lower priority is found. Because priority order is maintained at all times in the tree, the subtree rooted at n_l node has only nodes with smaller priorities. The node n_h replaces n_l in the tree, and the two subtrees of n_h are obtained by a splitting operation of the tree rooted at n_l with the hyperplane that defines n_h. The code for a general insertion operation is described in figure 7.3.

```
void KVDTree::outOfOrderInsertion(KVDTree *node)
{
 if (comparePriorities(node) == PRIORITY_LOWER) {
   // Traditional insertion: the node has a lower priority
   switch(classifyNode(node)) {
   case CL_IN:
     if (_left != NULL) _left.outOfOrderInsertion(node);
     else assignLeft(node);
     break;
   case CL_OUT:
     Similar to case above, using the right subtree instead
   case CL_CROSSING:
     KVDTree *aux = splitNode(node);
     if (_left != NULL) _left.outOfOrderInsertion(node);
     else assignLeft(node);
     if (_right != NULL) _right.outOfOrderInsertion(aux);
     else assignRight(aux);
     break;
 }
 else {
   // Insertion is changed to preserve priority
     if (parentSubtree() == CL_IN) parent().assignLeft(node);
     else parent().assignRight(node);
     KVDTree *newLeft, *newRight;
     splitTree(node, newLeft, newRight);
     node.assignLeft(newLeft);
     node.assignRight(newRight);
 }
}
```

Figure 7.3: Out-Of-Order Insertion

7.2.3 Dragging Trees

The *dragging* operation is designed as a special merging operation to combine the subtrees of a node that is moving into another location in the tree. Unlike the previous merging procedure, the dragging operation not only combines trees, but inspects the nodes of the trees for possible effects that the movement may cause.

Let n_l represent a node to be moved across subtrees of an ancestor node n_h. First we delete n_l from its current location, and then insert it into the other subtree of n_h. Because n_l may have non-empty subtrees before the movement, it becomes necessary to merge its subtrees into a single tree. During this merging process, every node is checked for incidence to n_l and the effect that the movement causes in the node is computed. If the incident node has a split behavior, a split of the node is performed, and a new node is inserted in the same subtree that that contains nl. If a node has a merging behavior, the node is deleted from its location, and its subtrees are merged using the same process recursively. Some nodes that were orginally partitioned by n_l may be joined together because the partition is removed. After the additonal actions required by incident nodes are performed, the merging proceeds in a recursive fashion.

In figure 7.4 we illustrate a step-by-step execution of the dragging operation with a simple example, where p_2 moves across the subtrees of p_1.

The geometric configuration is described in figure 7.4(a), with a partial tree corresponding to the subtrees of p_2 described in figure 7.4(a). After p_2 is inserted into the new subtree of p_1, we need to merge its subtrees. The incident nodes e_{1a} and e_{2a} have merging (orange highlight) and splitting (blue highlight) effects due to this movement. The dragging operation needs to merge the subtrees of p_2, while deleting e_{1a}, and splitting e_{2a} in two nodes, one that will stay at the tree (e_{2a}), and another that will be inserted in the other subtree of p_1. The operation starts with the two subtrees as parameters and check the roots of the tree for incidence with the moving node (figure 7.4(c)). Because the first root is incident to p_2, the effect is processed. In this case, the node is deleted and a new dragging operation is called to merge its subtrees (figure 7.4(d)), which in this case is a simple merging procedure. After the effect is processed for the first root, the second root is evaluated and another incident node is discovered. This time the node has a split behaviour, which creates an additional node e_{2a2}, that will be inserted into the other subtree of p_1. Finally, after

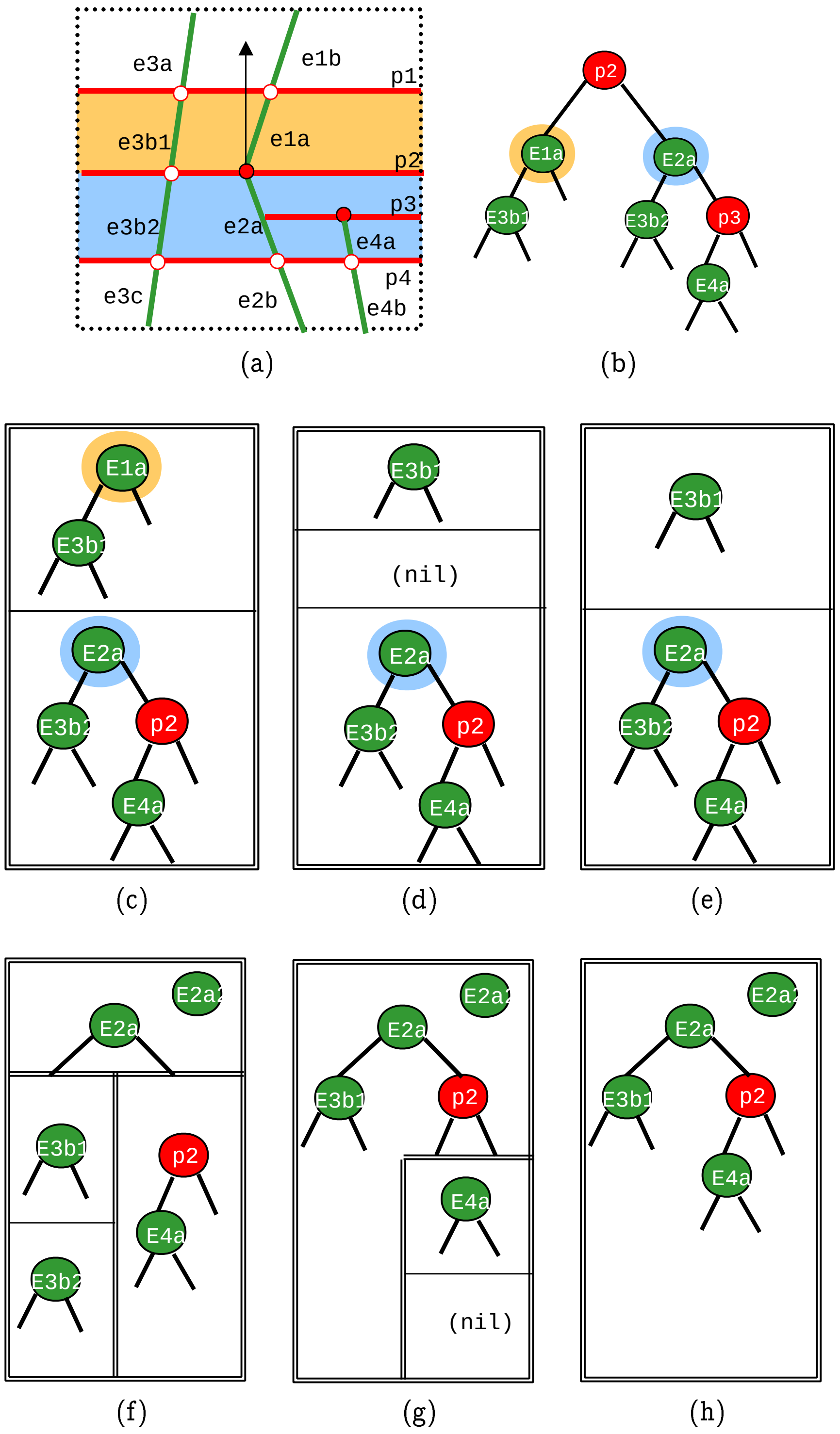


Figure 7.4: Dragging trees example.

both effects are processed, the priority merging is performed, and the node e_{2a} is chosen to be the root of the merged tree because of its higher priority (figure 7.4(e)). The left and right subtrees of e_{2a} are obtained with recursive calls of the dragging operation. Note that when forming the left subtree of e_{2a} two nodes with the same priority are compared, corresponding to different fragments of the same edge $e3$. In this case, a single node $e3b$ replaces both nodes, with a fragment that corresponds to the union of the fragments of the previous nodes. The resulting tree is showed in figure 7.4(h), with the only split node created displayed in the upper-right corner of the figure.

The code for the dragging algorithm is described in figure 7.5. The input for this algorithm corresponds to two pointers to subtrees (t_1 and t_2), and the lower(n_l) and higher(n_h) priority nodes that creates the event that required the dragging operation. The node n_l corresponds to the moving node and is used to check incidence of nodes in t_1 and t_2, and both n_l and n_h are used to detect the effect that the movement causes in incident nodes. A simpler version of the dragging operation with one tree as parameter (*dragOneTree*) is used in cases that the process continues with only one subtree (the code is very similar to the *dragTrees* procedure). The actions that process the effects caused over incident nodes are encoded into the *processEffect* procedure.

7.3 Update Algorithms

7.3.1 Algorithm V-update

The V-update algorithm describes the actions necessary to process two of the three vertex events: $VV-$ and $VE-$events. The remaining vertex event, the VT-event, is handled by the X-collide algorithm described later. In figure 7.6 and 7.7 we show some of the events that are handled by the V-update algorithm.

Let p_l represent a point node, and let n_h represent the ancestor that defines the hyperplane that is crossed by p_l. The node n_l can be of two types: either another point node (representing a VV-event), or an edge node (a VE-event). The update to be performed here consists of the following tasks:

- Save the subtrees $\mathtt{left}(p_l)$ and $\mathtt{right}(p_l)$ for further actions.

```
KVDTree* KVDTree::dragTrees(KVDTree *t1, KVDTree *t2, KVDTree *nLow, KVDTree *nHigh)
{
 if (t2 == NULL && t1 == NULL) return NULL;
 if (t2 == NULL && t1 != NULL) return dragOneTree(t1,nLow,nHigh);
 if (t2 != NULL && t1 == NULL) return dragOneTree(t2,nLow,nHigh);
 while (t1 != NULL && t1.effect(nLow,nHigh) == EFFECT_MERGE) {
   KVDTree *auxLeft = t1.left(), auxRight = t1.right();
   t1.processEffect(EFFECT_MERGE,nLow,nHigh);
   t1 = mergeTrees(auxLeft, auxRight);
 }
 while (t2 != NULL && t2.effect(nLow,nHigh) == EFFECT_MERGE) {
   KVDTree *auxLeft = t2.left(), auxRight = t2.right();
   t2.processEffect(EFFECT_MERGE,nLow,nHigh);
   t2 = mergeTrees(auxLeft,auxRight);
 }
 if (t2 == NULL && t1 == NULL) return NULL;
 if (t2 == NULL && t1 != NULL) return dragOneTree(t1,nLow,nHigh);
 if (t2 != NULL && t1 == NULL) return dragOneTree(t2,nLow,nHigh);
 switch (t1.comparePriorities(t2)) {
 CASE PRIORITY_LOWER:
   t1.processEffect(t1.effect(nLow,nHigh),nLow,nHigh);
   switch(t1.classifyNode(t2)) {
     case CL_IN:
       t1.assignLeft(dragTrees(t1.left(),t2,nLow,nHigh));
       t1.assignRight(dragOneTree(t1.right,nLow,nHigh));
       break;
     case CL_OUT: // Similar to above, changing left and right subtrees of t1
     case CL_CROSSING:
       KVDTree *newLeft, *newRight;
       t2.splitTree(t1,newLeft,newRight);
       t1.assignLeft(dragtTrees(t1.left(),newLeft,nLow,nHigh));
       t1.assignRight(dragtTrees(t1.right(),newRight,nLow,nHigh));
   }
   return t1;
 CASE PRIORITY_HIGHER: // Same as above, interchancing t1 with t2
 CASE PRIORITY_EQUAL:
   KVDTree *auxLeft=t2.left(), *auxRight=t2.right();
   t1.joinNode(t2);
   t1.processEffect(t1.effect(nLow,nHigh),nLow,nHigh);
   t1.assignLeft(dragtTrees(t1.left(),auxLeft,nLow,nHigh));
   t1.assignRight(dragtTrees(t1.right(),auxRight,nLow,nHigh));
   return t1;
 }
}
```

Figure 7.5: Dragging of Trees

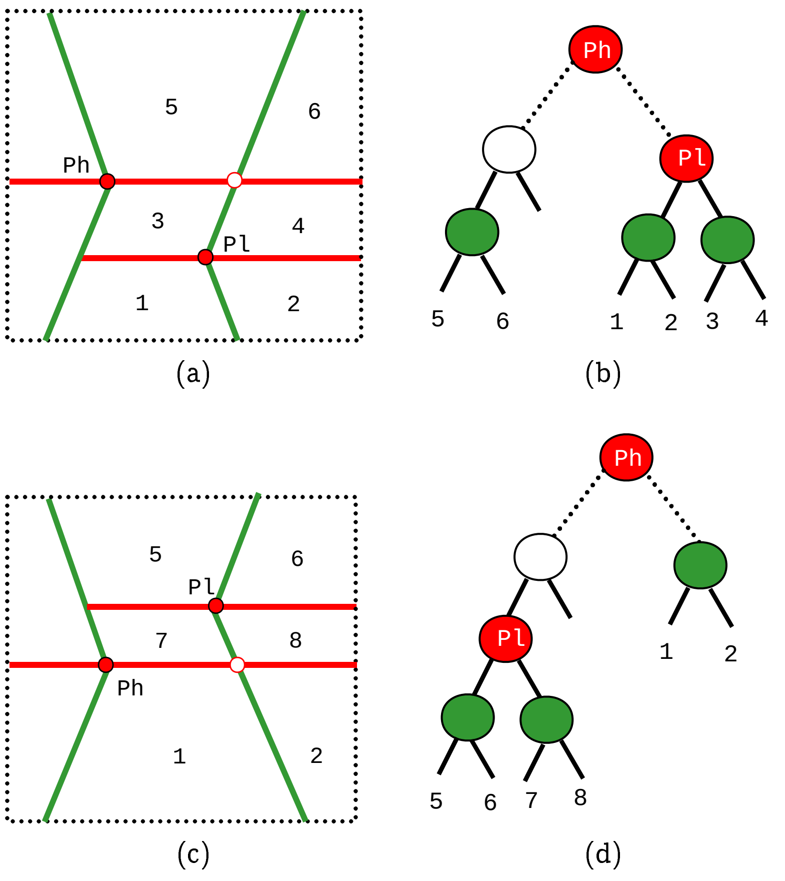


Figure 7.6: VV Update Example.

- Remove p_l from its current location.
- Insert p_l in its new location. This is accomplished by using an out-of-order insertion operation of p_l into the other subtree of n_h.
- Perform a dragging operation with the saved left and right subtrees of p_l. The result of this operation will be a tree, that is assigned to the old location of p_l. Nodes that were incident to p_l that need to be split, are inserted by the dragging operation in the same subtree that p_l was inserted.

The actions described above are explained in more detail in the code for the V-update algorithm described in figure 7.8. The input to the algorithm consists of the moving point

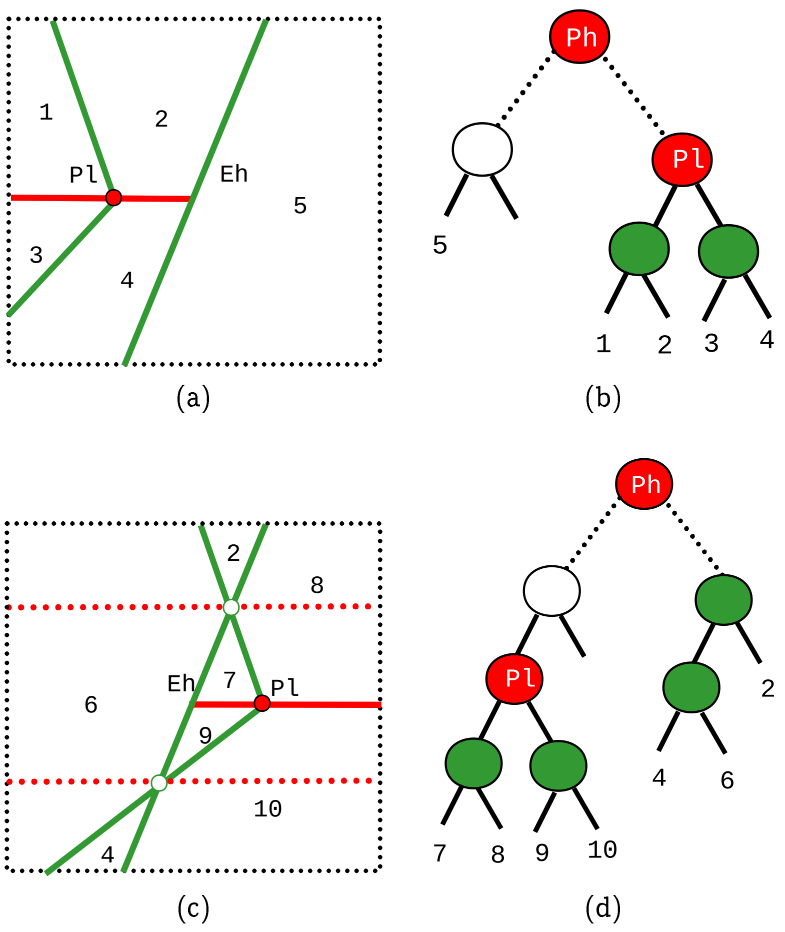


Figure 7.7: VE Update Example.

node and the ancestor node. Note that the nodes that are affected by changes need to have its certificates updated. This is accomplished by marking a certificate flag at these nodes. During the next traversal of the tree performed during rendering, every node that has this flag set causes a recomputation of its certificates.

The importance of the dragging operation in this algorithm can be seen by the simplicity of its description. Although the updates to be performed by these events are complex, the complexity is mostly encoded inside the dragging operation. Another important aspect in the design of this algorithm is that it can be used for both types of ancestor nodes (point node and edge node) that can be crossed by a moving point node. The existence of a single

```
void KVDTree::vUpdate(KBSPPointNode *pointNodeLow, KVDTree *nodeHigh)
{
 // Compute new halfspace occupied by the node
 Classification cl = nodeHigh.classifyNode(pointNodeLow);
 // Save pointer information
 KVDTree *parent = pointNodeLow.parent();
 KVDTree *left = pointNodeLow.left();
 KVDTree *right = pointNodeLow.right();
 // Recover which parent subtree the node was located
 Classification subtree = pointNodeLow.parentSubtree();
 // Reset pointer information
 pointNodeLow.resetPointers();
 // Out of order insertion in the new location
 nodeHigh.insertOutOfOrder(pointNodeLow);
 // Indicate that new certificates need to be computed for the node
 pointNodeLow.updateCertificates(1);
 // Drag the previous subtrees to the new location, and assign
 // the remaining subtree to previous parent node
 if (subtree == CL_IN)
   parent.assignLeft(dragTrees(left,right,pointNodeLow,nodeHigh,cl));
 else
   parent.assignRight(dragTrees(left,right,pointNodeLow,nodeHigh,cl));
 // New certificates need to be computed for the parent node
 parent.updateCertificates(1);
}
```

Figure 7.8: Algorithm V-update

algorithm simplifies the implementation, but one might argue that the VV-event has special properties that could be explored if separate algorithms were designed. For example, the merging of subtrees is extremelly trivial in the VV-event because one of the subtrees of the moving node always disappears, and the merged tree simply corresponds to the other subtree of the moving node. However, the merged subtree would need to be traversed anyway to find incident nodes that have a split behavior. In addition, the nodes that are not incident to the moving node need to be joined into a single node, while updating its fragments. This was the case in the example used during the discussion of the dragging trees procedures (figure 7.4((f)). Therefore, the merged tree would also need to be traversed to check for nodes that require fragment updates. It turns out that the approach using the dragging operation does a better job because all fragment updates are obtained when the merging procedure encounters nodes with the same priority. As a result, we choose to use the dragging operation for both cases.

7.3.2 Algorithm E-update

The E-update algorithm describes the actions necessary to process all edge and intersection events that do not involve collisions, which are handled by the X-collide algorithm described later. Edge and intersection events are closely related and often happen most of the times concurrently. We explored this connection before to avoid creating duplicate certificates to detect these events, and we again explore this connection in the design of the update algorithms. The only intersection event that does not cause an edge event (Figure 7.9) can be handled in a very simple way, with the update of the certificates of the intersection nodes. From now on, the intersection events we discuss happen together with edge events.

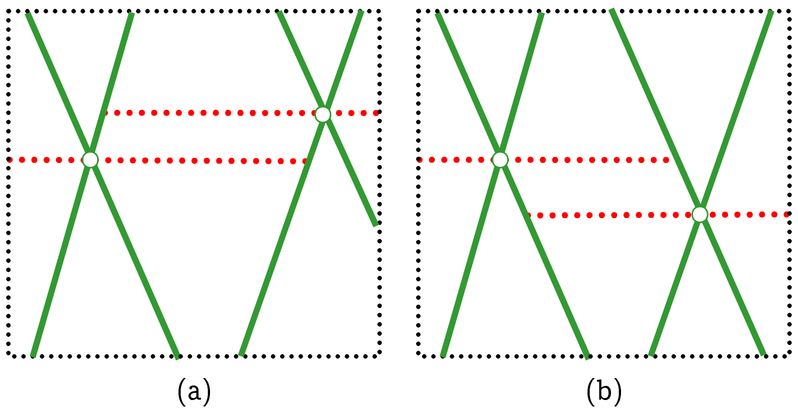


Figure 7.9: Intersection event that does not have an associated edge event.

Intersection events were used to detect edge events that involved intersection points. Because intersection points are not explicitly stored in the tree, intersection events itself do not cause changes in the structure of the tree, which are caused by the movement of the edge nodes where the intersection points are defined. In summary, intersection certificates are used to detect the events, but the update in the tree is done through edge node updates. In figures 7.10 and 7.11 we review some of the cases that are handled by the E-update algorithm. We observe from these examples that many edge events defined over a single edge may happen at the same time. This is a direct consequence of one edge

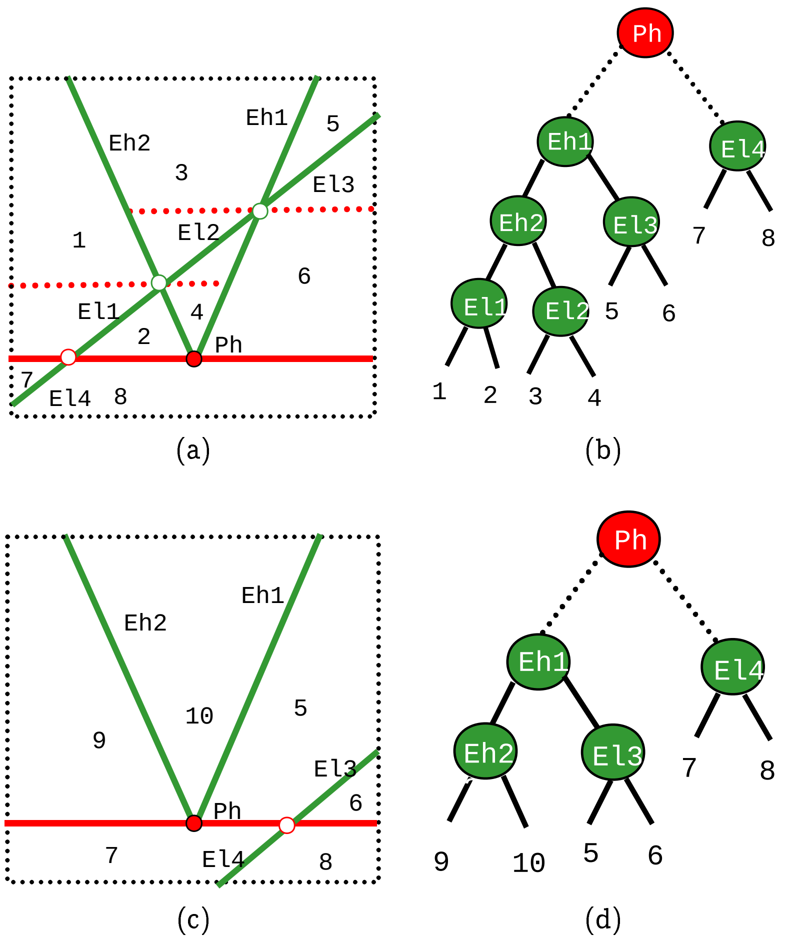


Figure 7.10: Edge events detected by VI and II certificates.

being partitioned in several parts, each corresponding to a different node stored in the tree. The time an edge event happens corresponds to an edge passing through the plane of an ancestor node, and usually adjacent edge nodes defined on the same edge are affected by these events. The occurrence of multiple events in edge events motivates a different approach in the update algorithm than the one used for vertex events.

The types of effects that an edge event may cause on nodes are the same effects that were observed in the discussion of the algorithm for vertex events. A node has a merging effect (EFFECT_MERGE) when it needs to be deleted from its current location, while a

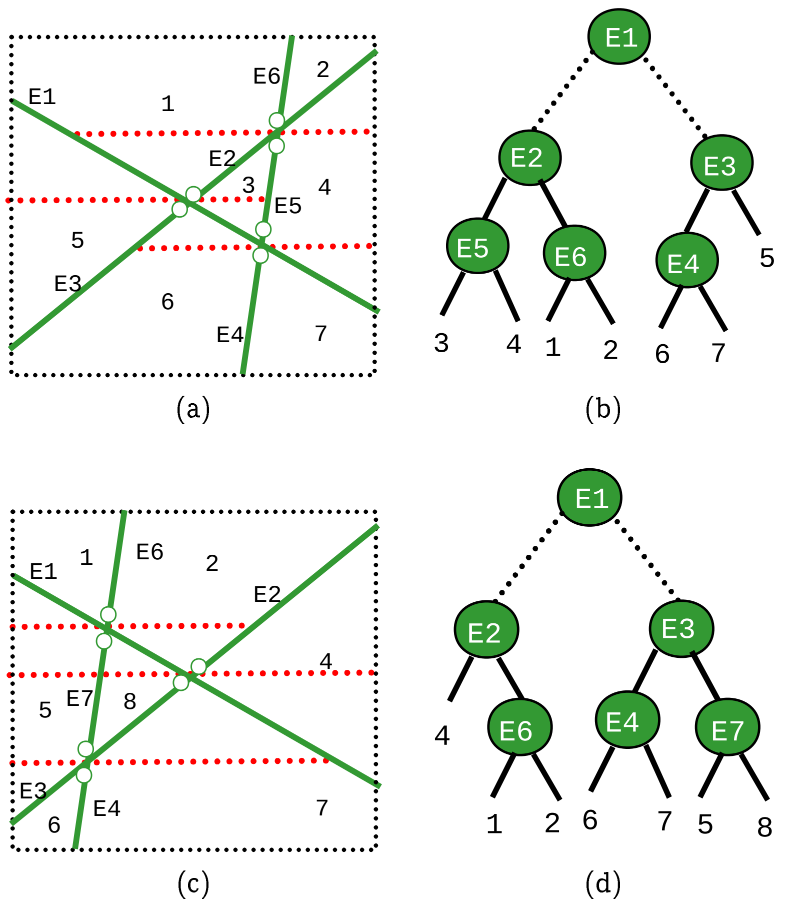


Figure 7.11: Edge events detected by II certificates.

splitting effect (EFFECT_SPLIT) causes a node to be split into additional nodes. In the V-update algorithm these cases were handled during the execution of the dragging operation, with the effects being processed as visited. Here, we follow a different approach, where we separate the nodes in two groups according to the type of effect, and handle the updates in each of these sets separately.

This grouping of events only makes sense because the edges involved in one of these events correspond to nodes of the same edge in a scene. The reason to separate merged edge nodes from splitting edge nodes is directly related to the fact that all edge fragments of merging nodes collapse to a point when the event happens. Suppose we advance time by an infinitesimal amount after the event happened. New fragments may be just created giving rise to several nodes in another subtree, with new fragments that do not have any connection with previous fragments. One way to create all these new nodes is to identify every new location occupied by the edge and incident triangle nodes, and for each location found, insert new edge and triangle nodes. Another way to accomplish the same result is to perform othe insertion of a single edge node and its incident triangle nodes, starting from a node higher in the tree. This higher node, however, needs to have an associated region that is guaranteed to contain all the new nodes. The several nodes that need to be created are naturally obtained in this approach, because as nodes are filtered down the tree, partition operations are applied and nodes are created. The replacement of several insertions of edge nodes by a single insertion of an edge node suggests that we handle merging and splittind nodes in groups.

The E-update algorithm first enumerate all edge nodes that are involved in an intersection certificate failure. This can be easily done because these certificates contain pointers to the edges that define the intersection nodes. We separate these edges into merging and splitting sets depending on the effect that the event has on each edge node. The effects caused by edges in the merging set are processed first. In this merging step, we delete all edge nodes in the merging set from the tree, and replace it with the priority merging of its subtrees. Note that we do not use the dragging operation here because it automatically perform actions for splitting nodes, which we do not want at this point. The update of the fragments of all nodes is done by the priority merging algorithm.

After all merging edge nodes were processed, we continue with updates caused by edge nodes in the splitting set. For all these edge nodes, it is necessary to update its fragments, because one of the endpoints of the edge node changes when the event is processed. In addition, every node incident or cut by a splitting edge node may also require a fragment update. More specifically, all fragments that contain the endpoint of the edge node that changes when the event is processed need to have their fragment updated. After all fragments are updated, it is necessary to perform a single insertion of an edge node into a subtree that is guaranteed to contain all the new edge nodes. In the cases where an edge passes through an ancestor point node, we use one of the subtrees of this point node as the place of insertion. In the case of figure 7.10(a) we would insert this edge node in the right subtree of Ph. For the case that an edge nodes passes through another edge node (figure

7.11(a)), we would insert the edge node in the right subtree of E3. Finally, all triangles that were incident to the moving edge in the scene are used to create triangle nodes, that are inserted in the tree at the same place as above.

The code for the E-update algorithm is described in figure 7.12. The input for the algorithm consists of a set of edge nodes defined over the same edge of the scene, the cardinality of this set, and a node that is guaranteed to contain all edge nodes to be created. This node is used as the place to insert a new edge node and incident triangle nodes.

```
void KVDTree::eUpdate(KBSPEdgeNode *edgeNode[], int nEvents, KVDTree *nodeHigh)
{
  // Merging step
  for (int i=0; i¡nEvents; i++) {
    if (edgeNode[i].effect(nodeHigh) == EFFECT_MERGE) {
      // Process merge. Delete node and merge its subtrees
      edgeNode[i].processEffect(EFFECT_MERGE,nodeHigh);
  }
  // Splitting step
    if (edgeNode[i].effect(nodeHigh[i]) == EFFECT_SPLIT) {
      // Update fragments of the edge node and all other nodes that
      // depend on the endpoint that is changed in the edge fragment
      edgeNode[i].updateFragments();
  }
    // Simple insertions creates all new nodes
  KVDEdgeNode *auxEdgeNode = new KVDEdgeNode(edgeNode[0]);
  // Out of order insertion in the new location
  nodeHigh.insertOutOfOrder(auxEdgeNode);
    Every triangle incident to the edge also need to be inserted
  for (int i=0; i¡auxEdgeNode.incidentTriangles(); i++) {
    KVDTriangleNode *auxTriangleNode =
      new KVDTriangleNode(incidentTriangleNode(auxEdgeNode, i));
    nodeHigh.insertOutOfOrder(auxTriangleNode);
  }
}
```

Figure 7.12: Algorithm E-update

7.3.3 Algorithm X-collide

The algorithm X-collide is used to perform the necessary actions to handle the possible collision events: VT−, TV− and ET− event. The solution to avoid collision includes modifying the equation of motion of the objects that define each of the nodes causing the collision. This is accomplished in our implementation with the reversal of the direction of the equation of motion of the objects.

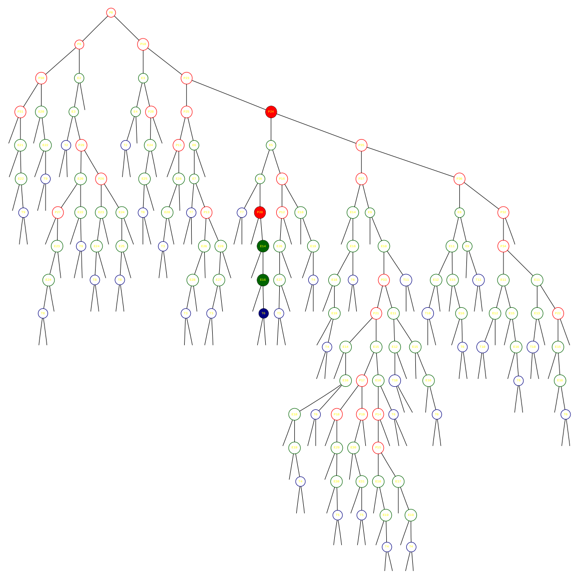
Because objects are assumed to have rigid motions, every node that is created from certain object needs to have its certificates updated. Instead of locating all of these nodes in the tree, only the point nodes created from these objects are marked with invalid certificates. This can be accomplished if we keep an array of pointers to all point nodes in the tree indexed by the vertex index. In addition, because they are inserted before the other types of edge nodes, marking a point node as having invalid certificates will require the update of certificates for every node in the subtrees of these point nodes. The code for this algorithm is described in figure 7.13. The input to the algorithm correspond to the nodes that cause the collision.

```
void KVDTree::xCollide(KVDTree *nodeLow, KVDTree *nodeHigh)
{
 // The collision is avoided by reverting the objects motion
 int nodeLowObject = nodeLow.object();
 int nodeHighObject = nodeHigh.object();
 revertEqMotion(nodeLowObject);
 revertEqMotion(nodeHighObject);
 // Every node create by these objects need to update its certificates.
 // Mark as invalid the certificate flag of all point nodes of the object
 setUpdateCertificatesFlag(nodeLowObject, 1);
 setUpdateCertificatesFlag(nodeHighObject, 1);
}
```

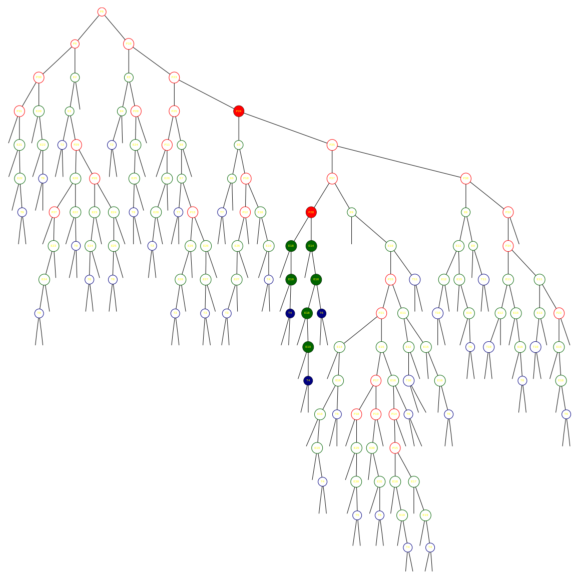
Figure 7.13: Algorithm x-Collide

7.4 Updates examples

The modification that are created by a simple event are better understood if we visualize the structure of the tree before and after updates. In figures 7.14 and 7.15 we have two examples corresponding to two successive tree updates in the tree. The nodes are drawn using the same color scheme as before (red for point node, green for edge nodes and blue for triangle nodes). Nodes have different filling styles, depending on the effect that the tree update has over each node. The filled nodes correspond to the nodes that are affect by the update, with the node higher in the tree corresponding to the ancestor node that defines the event. In figure 7.14 we have an example of a VV update, where a point node passes through the plane of another point node. In figure 7.15 we have an example of a VE update, where a point node passes thorugh the plane of an edge node.

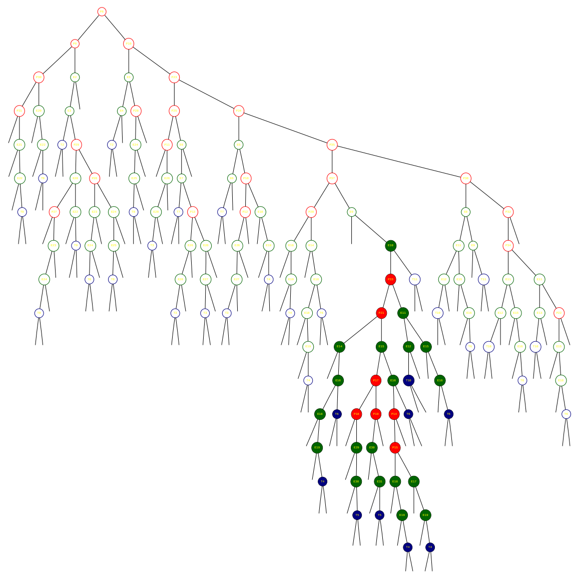


(a)

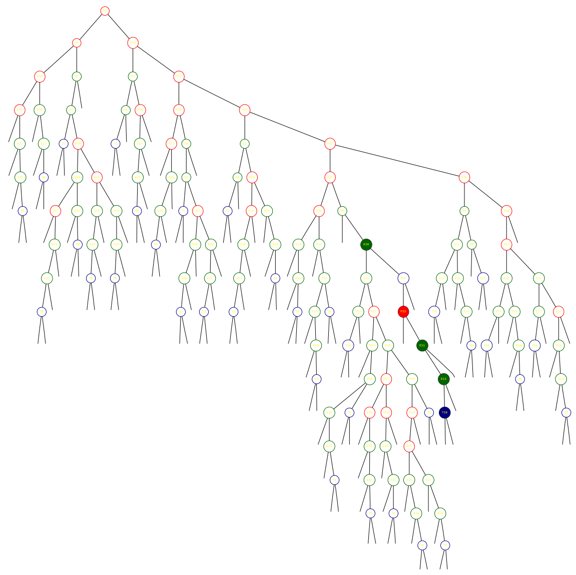


(b)

Figure 7.14: VV update example.



(a)



(b)

Figure 7.15: VE update example.

Chapter 8

Results

In this chapter we evaluate the performance of the KVD. The presentation starts with a description of the implementation, including a discussion on the user interface used to visualize several aspects of the KVD. The KVD is tested by running kinetic simulations through different scenes. For each of these simulations, results are given regarding the size, number of certificates and several statistics concerning the performance of update algorithms.

8.1 Implementation

The implementation of the KVD was done using the C++ programming language, which has been used throughout this text in the description of data structures and algorithms. The different parts of the code are all integrated in a single program *movingWorlds*, that performs the kinetic simulation of moving scenes composed of objects with triangular faces.

The input to the *movingWorlds* program consists of a scene composed of static and dynamic objects with triangular faces. Each object contains the description of vertices, edges and faces of the geometric model. In addition, material properties such as color values are also described in each model. Each dynamic object is assumed to have a rigid motion, therefore a single equation of motion is specified for each object. Other parameters that are provided as input to the program include: the vertical decomposition directions ($\hat{x}$ and $\hat{z}$) and the value of ϵ-min used in the kinetic priority queue.

Given this input, the *movingWorlds* program creates an initial KVD tree using a randomized priority order scheme. For this initial tree, all certificates associated with the tree are computed, and the kinetic priority queue is updated to contain the information about the next certificates to fail. Once the initial KVD is computed, the program is able to start a kinetic simulation. At any moment during the simulation, the KVD can be used to extract a visibility ordering for the scene. The most important tasks that the program needs to accomplish are (1) the correct detection of when events happen, and (2) the correct update of the KVD to conform to new positions of geometry.

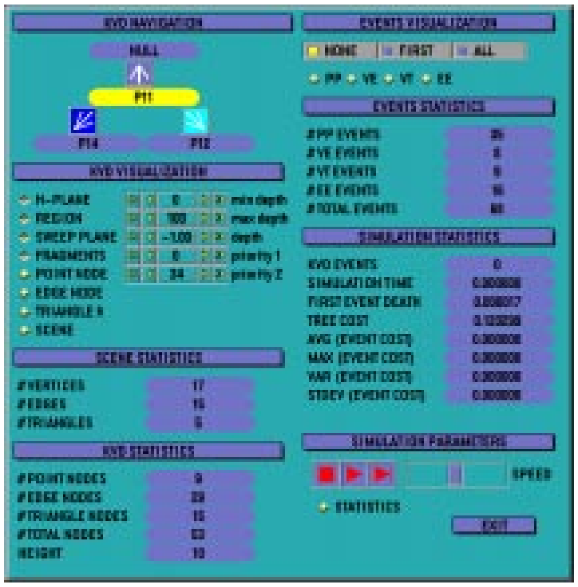
There are two variants of the *movingWorlds* program. First, a non-graphical version is just concerned with running the simulation, without producing any visual illustration about the KVD. This program is very useful once the implementation is complete, and therefore when our primary concern is to verify the correctness of the tree, and to evaluate the performance of the updates in the KVD. A second version contains a graphical interface, that is used to display most of the geometric and combinatorial structure of the KVD. This version was extremely useful during the debugging stages of the implementation.

The ideas behind the graphical version were discussed in chapter 2. The user interface is composed of two windows. The interaction window allows the user to control a series of parameters that are used in the visualization of the three dimensional structure of the KVD. The visualization window contains one view of the structure of the KVD as defined by the interaction window, with a trackball mechanism that allows the user to interactively update the projection used to compute the visualization. In figure 8.1 we show the interaction window, and some examples of the visualization window.

The interaction window is divided into several areas, either used to input user selections, or to display properties present in the KVD. Each of these areas is described in detail below.

Visualization Section

This section of the interaction window contains flags that control several aspects of the visualization of the KVD. There are three mechanisms used to select nodes in the tree: interactive navigation, procedural selection, and sweeping plane. In the interactive navigation approach, one node is always marked as selected, and some properties are described only about this node (e.g. a hyperplane or a region of a node). This selection mechanism



KVD VISUALIZATION

H-PLANE	0	min depth
REGION	100	max depth
SWEEP PLANE	-1.00	depth
FRAGMENTS	0	priority 1
POINT NODE	34	priority 2
EDGE NODE		
TRIANGLE #		
SCENE		

SCENE STATISTICS

#VERTICES	17
#EDGES	15
#TRIANGLES	6

KVD STATISTICS

#POINT NODES	9
#EDGE NODES	29
#TRIANGLE NODES	15
#TOTAL NODES	53
HEIGHT	10

EVENTS STATISTICS

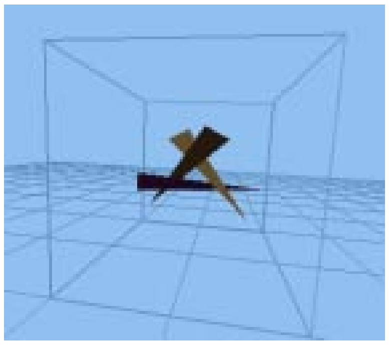
#PP EVENTS	25
#VE EVENTS	8
#VT EVENTS	9
#EE EVENTS	16
#TOTAL EVENTS	60

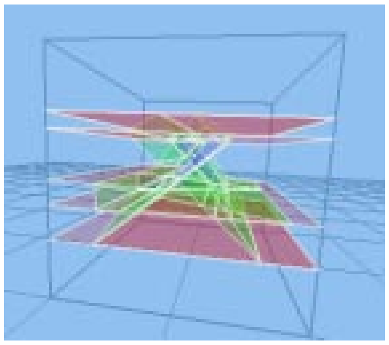
SIMULATION STATISTICS

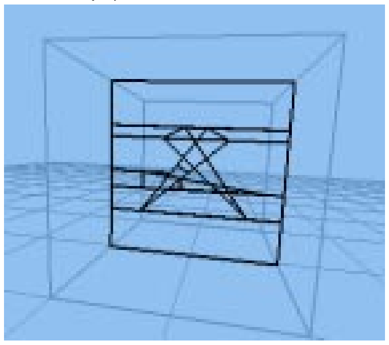
KVD EVENTS	0
SIMULATION TIME	0.000000
FIRST EVENT DEATH	0.000017
TREE COST	0.100000
AVG (EVENT COST)	0.000000
MAX (EVENT COST)	0.000000
VAR (EVENT COST)	0.000000
STDEV (EVENT COST)	0.000000

SIMULATION PARAMETERS

(a) Interaction window.


(b) Sample scene


(c) Fragments


(d) Sweeping plane

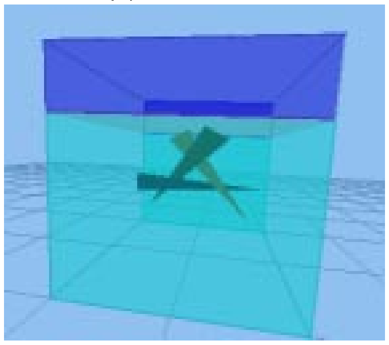

(e) Regions

Figure 8.1: User interface and different visualizations of the KVD

is described in more details in the navigation section below.

In the procedural selection, there are several flags that specify properties of nodes which are to be displayed. For each selected node, the fragments associated with the node are displayed using a specific color for each type of node. The use of the alpha channel in the specification of colors allow the simulation of transparency, which is useful when displaying a complex structure such as the KVD. The visibility ordering provided by the KVD is used to correctly compose the alpha channels of the fragments displayed.

In this selection mechanism, the only nodes selected are the ones that satisfy a set of properties, given by boolean expressions. A simple example of expression is defined by a flag, that indicates if certain property is defined for a node. For instance, a flag that indicates that point nodes may be used to indicate that point nodes are to be included in the set of candidates to selected nodes. Similarly, edge and triangle node flags can be used to indicate that edge and triangle nodes are also candidates. Another example of a boolean expression used to select nodes is related with the specification of an interval in which a certain attribute is allowed to vary. The only nodes that are selected as candidates are the ones that have this attribute within the given range. In the interaction window, we use intervals of depth and priority values.. The ability to change the minimum and maximum depth values allows the selection of different nodes in the tree. The only nodes selected for display are the ones that satisfy all boolean expressions defined in the interaction window.

The sweeping plane technique is used to illustrate a moving cross-section of the KVD. For simplicity, we only use cross-sections perpendicular to a single fixed direction. The user controls a single parameter, the depth of the plane, that allows the movement of a sweeping plane between the front and back face of the universe bounding box. Finally, a flag is defined to switch on or off the display of the scene.

Navigation section

One node is always defined as the selected node from the tree, which is used by the region and hyperplane display methods. The selected node is changed by moving to one of its adjacent nodes in the tree (the parent or either one of its children). The arrows on the display correspond to these alternatives. The region of a node is divided into two sub-regions, each corresponding to the regions of the left and right subtrees. Instead of drawing the region of a node with a single color, we prefer to draw the sub-regions of its subtrees with different colors. Our convention is to always associate blue with the left subtree, and green with the right subtree. This convention helps the user identify in the navigation selection area which of the subtrees correspond with which children are associated to which region.

The interactive navigation, combined with the visualization of the region of the node, is a powerful tool for understanding the structure of the tree. Hierarchical structures are often only evaluated by statistical evaluations, but the geometric structure can complement such results. Figure 8.2 shows the structure of the tree with its accompanying regions, using snapshots from the visualization process.

Events Visualization

The depiction of certificates in the structure of the KVD can be used to understand the behavior of certain events. We apply the technique described in chapter 6, which consists of drawing straight lines connecting points that are involved in the creation of the event. A simple selection mechanism allows the display of no events, the first event to happen, and all events for all nodes.

Statistics

The following statistics are displayed in the interface window:

- Scene statistics: contains information about the total number of vertices, edges and faces contained in the input scene.
- KVD statistics: contains information about the different types of nodes in the tree, and the maximum height of the tree.
- Events statistics: Contains information about the current set of certificates.
- Simulation statistics: Contains several informations about the kinetic simulation: number of events processed, current time of the simulation, next event death time, cost to compute initial KVD, average cost to update the KVD with local algorithms, maximum update cost, variance and standard deviations of the cost updates.

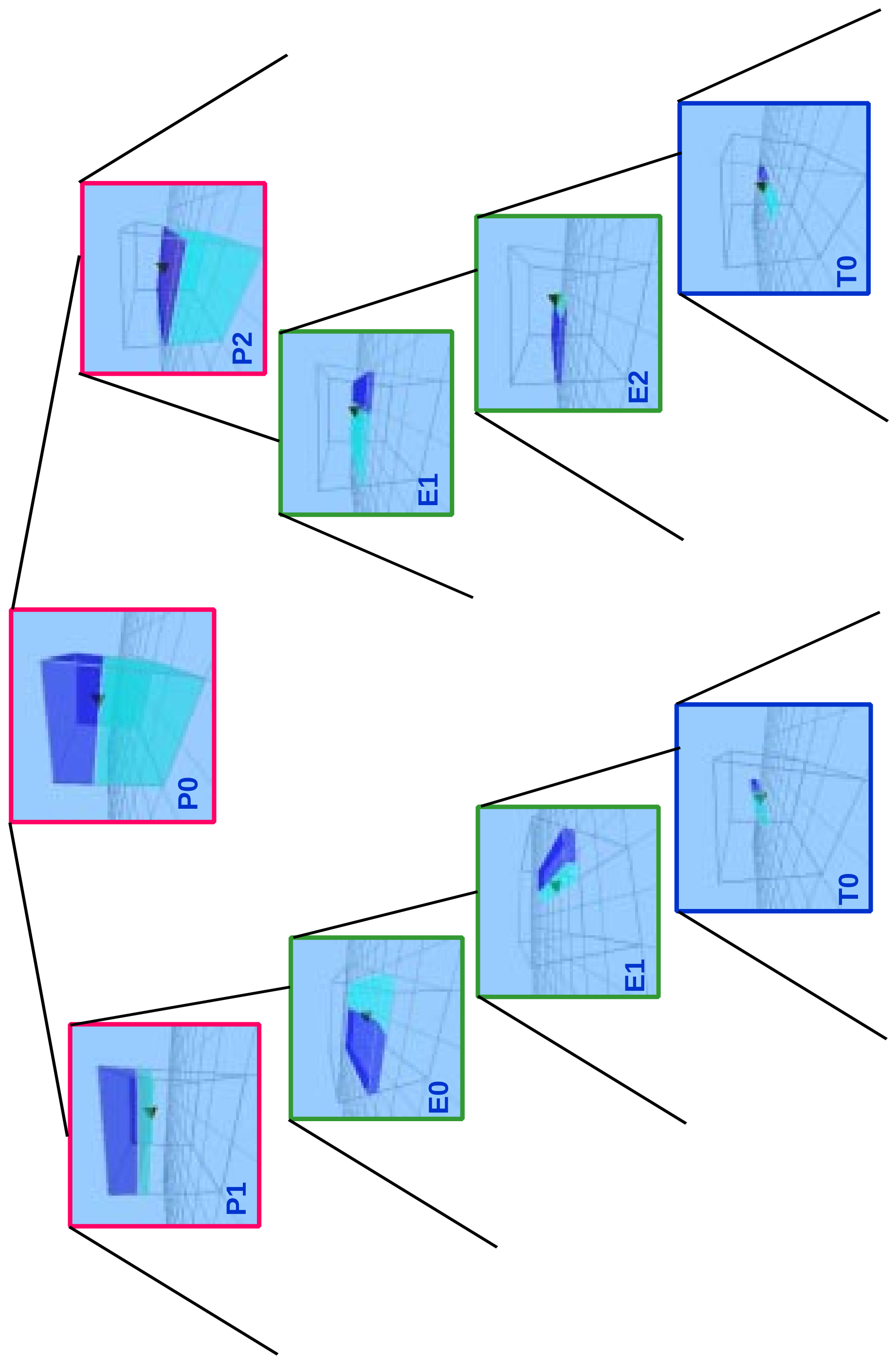


Figure 8.2: KVD tree structure combined with the visualization of the regions of each node.

Simulation Parameters

The simulation is started and stopped through the activation of small buttons in the simulation properties parts of the interface. A *play* button is used to run the simulation until a *stop* button is pressed. A *play-and-pause* button is used to run the simulation only until the first certificate fails, when the simulation is stopped.

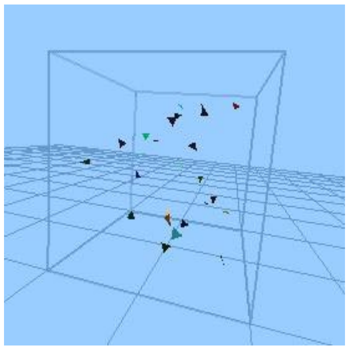
8.2 Kinetic Simulations

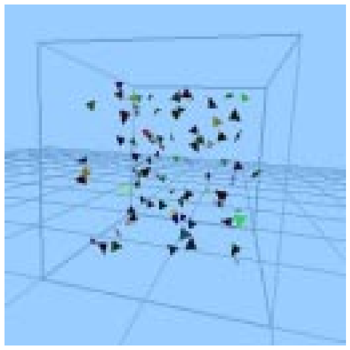
The properties displayed in the interaction window do not include all possible properties that affect the operation of the simulation mechanism. The remaining options are specified by command-line arguments to *movingWorlds*.

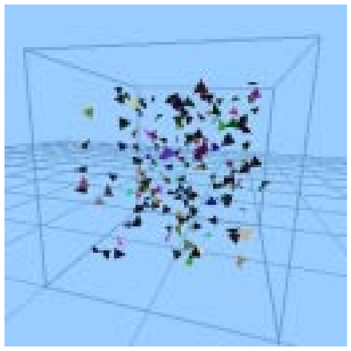
During the debugging stages of the implementation, the result of every local update of the KVD was compared with a KVD built from scratch, based on the new geometric position of the scene. A validation procedure checked whether the structure of both trees was exactly the same by performing simultaneous traversals in both trees, and comparing each node of one tree against the corresponding node of the other. If the nodes had different types, or if the elements used to define the node were different, the validation procedure failed. In addition, even if all these comparisons were valid, but the fragments stored in each node were different, the validation still failed. In cases where errors occurred, the validation procedure reported the type of error encountered, the nodes that caused the error, and the displayed the incorrect trees.

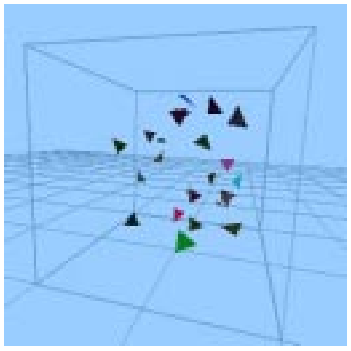
The performance of the KVD was tested with simulations of several scenes composed of moving triangles inside a bounding box. Two different types of data sets were created. The first one, called the *uniform scale*, contains different scenes with a increasing number of triangles (25, 50, 100, 200, 400, 800). The triangles in all these scenes are congruent triangles, and differ only in position and orientation. A second data set, called *uniform density*, contains scenes of triangles that are all congruent within a single scene, but have different sizes across scenes, depending on the number of triangles in the scene. The idea is to create data sets that maintain the same density of occupation obtained in a base scene with 100 triangles. Therefore, scenes with a smaller number of triangles than 100 are composed of bigger triangles, and smaller triangles are used for scenes with more triangles.

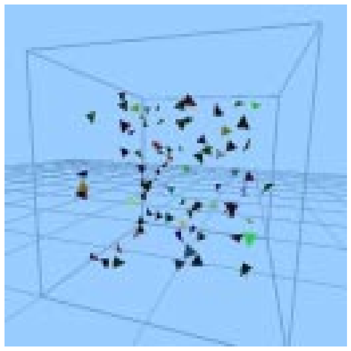
The construction of both sets was done in such way that the scenes with 100 triangles are identical. In figure 8.3 we show sample scenes from each of these sets.


(a) 25 uninform scale


(b) 100 uninform scale


(c) 200 uniform scale


(a) 25 uninform density


(b) 100 uninform density

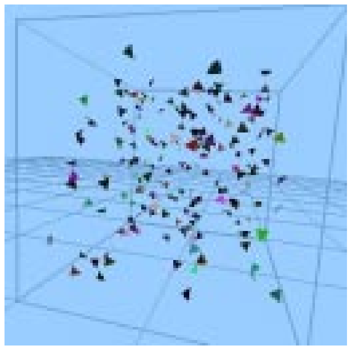
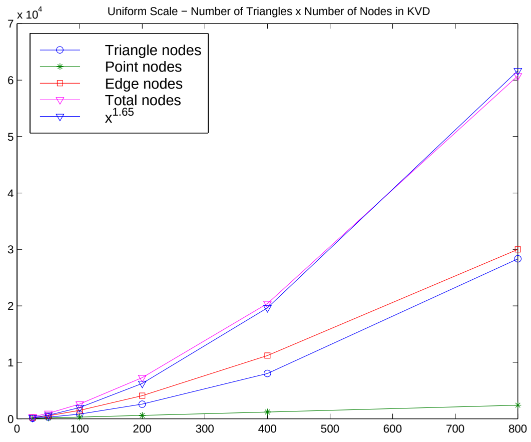
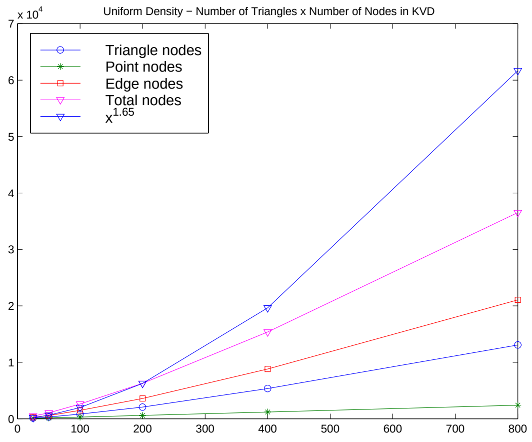

(c) 200 uniform density

Figure 8.3:

8.2.1 KVD-Tree Construction Statistics

The first important statistic about the KVD is the size of the tree. In figure 8.4 we show the results obtained for each of the sets described above. We present these statistics in graphical and tabular format. The edge and triangle nodes are most numerous because they are the only ones that are partitioned by other cuts in the tree. The number of point nodes corresponds exactly to the number of vertices in the input scene. For scenes with 800 triangles, the uniform scale approach creates trees with more nodes, because more partitioning happens due to the fact that the density increases as more triangles of the same size are inserted. On the other hand, scenes with uniform density produce more triangle fragments when fewer triangles are present, because larger triangles are involved.





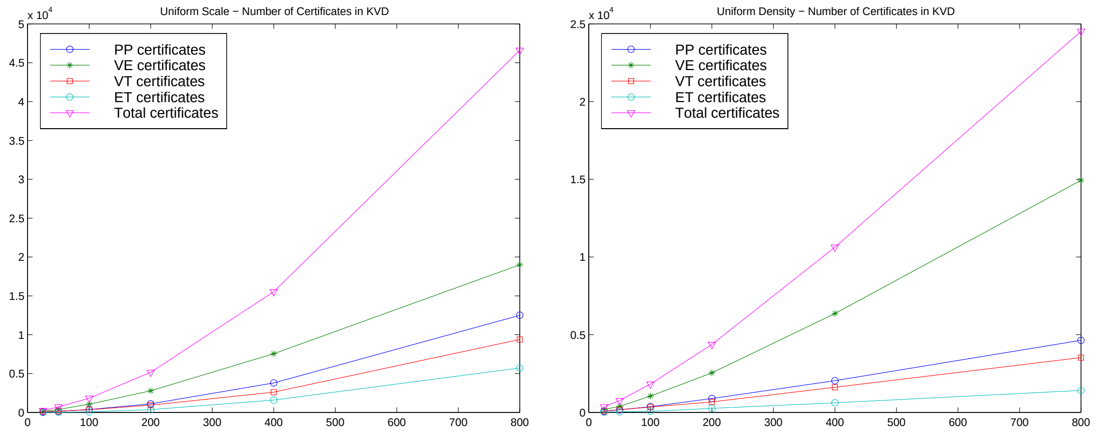
KVD-Tree construction statistics - scenes with triangles - uniform scale						
	25 △s	50 △s	100 △s	200 △s	400 △s	800 △s
Vertices	75	150	300	600	1200	2400
Edges	75	150	300	600	1200	2400
Triangles	25	50	100	200	400	800
KVD P-nodes	75	150	300	600	1200	2400
KVD E-nodes	191	531	1489	4086	11195	30013
KVD T-nodes	96	283	835	2604	8020	28339
KVD total nodes	360	964	2624	7290	20415	60752
KVD Height	17	20	24	37	34	47

KVD-Tree construction statistics - scenes with triangles - uniform density						
	25 △s	50 △s	100 △s	200 △s	400 △s	800 △s
Vertices	75	150	300	600	1200	2400
Edges	75	150	300	600	1200	2400
Triangles	25	50	100	200	400	800
KVD P-nodes	75	150	300	600	1200	2400
KVD E-nodes	261	571	1489	3599	8822	21062
KVD T-nodes	147	320	835	2078	5345	13068
KVD total nodes	483	1041	2624	6277	15367	36529
KVD Height	18	28	24	29	37	42

Figure 8.4: KVD Construction Statistics.

8.2.2 KVD-Tree Certificate Statistics

In figure 8.5 we have statistics about the types of certificates created in each of the uniformly scale and density datasets.



KVD-Tree certificate statistics - scenes with triangles - uniform scale						
	25 △s	50 △s	100 △s	200 △s	400 △s	800 △s
PP Certificates	55	145	364	1098	3782	12496
VE Certificates	125	379	1048	2777	7550	18997
VT Certificates	66	153	344	932	2609	9388
ET Certificates	4	20	66	350	1580	5720
Total Certificates	250	697	1822	5157	15521	46601

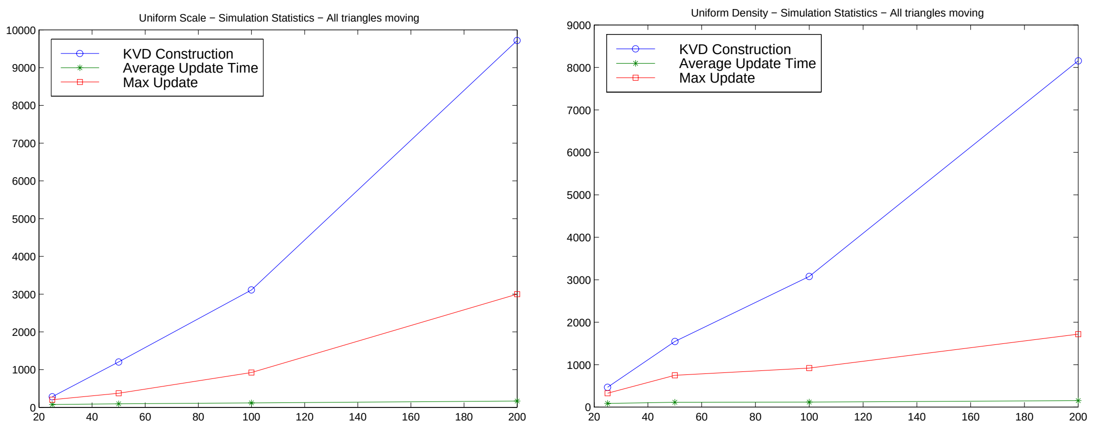
KVD-Tree certificates statistics - scenes with triangles - uniform density						
	25 △s	50 △s	100 △s	200 △s	400 △s	800 △s
PP Certificates	71	164	364	894	2041	4640
VE Certificates	190	382	1048	2547	6361	14936
VT Certificates	82	176	344	673	1615	3526
ET Certificates	20	40	66	260	614	1416
Total Certificates	363	762	1822	4374	10631	24518

Figure 8.5: KVD Certificate Statistics.

The PP certificates include all certificates of types VV, VI and II. The type of certificate VE contains the largest number of certificates. One reason to have a greater number of VE certificates than PP certificates is because they are always defined twice for six- and five-sided cells, while VV events are defined twice for six-sided cells, but only once for five-sided cells. The certificates of type ET are more numerous in higher density scenes (uniform density with less than 100 triangles, uniform scale with more than 100 triangles).

8.2.3 KVD-Tree Simulation Statistics

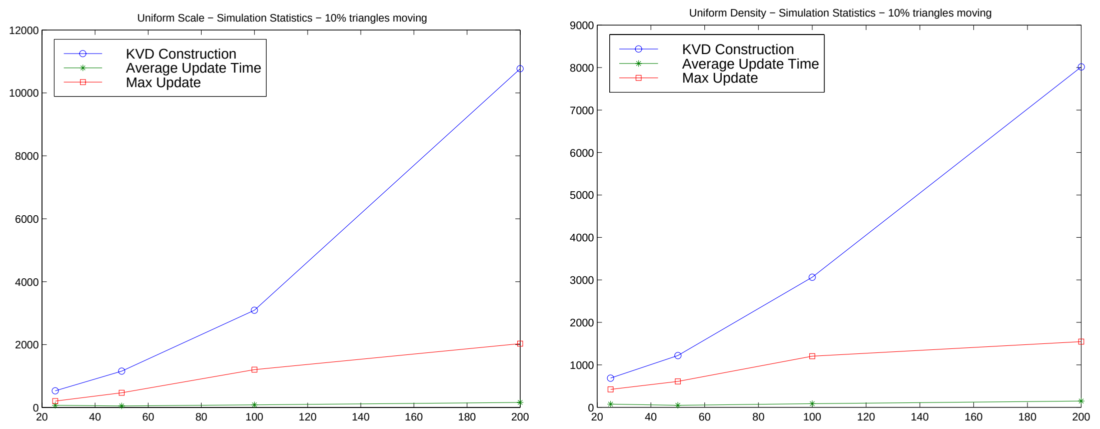
In figures 8.6 and 8.7 we have the results for several kinetic simulations. All reported times are expressed in mili-seconds.



KVD-Tree simulation - 100% moving - uniform scale				
	25 △s	50 △s	100△s	200 △s
KVD Construction	281	1204	3109	9719
Average update	75	94	119	168
Max update	204	375	922	3000
Standard Deviation	24	38	69	177
Tree update / Total Update	0.79	0.75	0.72	0.68
Certificates update / Total Update	0.21	0.25	0.28	0.32

KVD-Tree simulation - 100% moving - uniform density				
	25 △s	50 △s	100△s	200 △s
KVD Construction	469	1547	3079	8156
Average update	87	114	119	153
Max update	329	750	921	1719
Standard Deviation	34	69	69	119
Tree update / Total Update	0.77	0.72	0.72	0.68
Certificates update / Total Update	0.23	0.28	0.28	0.32

Figure 8.6: KVD Simulation Statistics - All triangles moving.



KVD-Tree simulation - 10% moving - uniform scale				
	25 △s	50 △s	100△s	200 △s
KVD Construction	532	1484	3485	10766
Average update	72	113	120	162
Max update	204	188	2656	2032
Standard Deviation	24	9	131	138
Tree update / Total Update	0.59	0.47	0.55	0.67
Certificates update / Total Update	0.41	0.53	0.45	0.33

KVD-Tree simulation - 10% moving - uniform density				
	25 △s	50 △s	100△s	200 △s
KVD Construction	687	1687	3735	8015
Average update	74	91	120	146
Max update	422	469	2672	1546
Standard Deviation	37	21	131	108
Tree update / Total Update	0.79	0.51	0.55	0.71
Certificates update / Total Update	0.21	0.49	0.45	0.29

Figure 8.7: KVD Simulation Statistics - 10% triangles moving.

For each one of this scenes, we ran a kinetic simulation until a fixed number of events was processed, and we reported results based on the performance of the KVD during this simulation. For the results reported below, 1000 events are processed. The motion of the triangles in these scenes was always a linear motion, randomly generated. The simulation statistics are described for two types of situations. The first one corresponds to having all triangles in a scene moving. In the second situation, only 10% of triangles are allowed to move. The separation of static from dynamic triangles is used in the construction of the KVD, with static triangles inserted before any of the dynamic triangles.

The average update time obtained in all simulations is considerably smaller than the time to construct the entire tree. In some situations, the maximum update cost can be high, but the percentage of the total simulation time used by the maximum update decreases as the scene complexity grows. The time used to perform updates in the tree and in the events structure is illustrated as percentages of the total simulation time. The tree update is most of the times the most expensive operation.

In figure 8.8 we compare the costs of locally updating the tree against an approach that reconstructs the tree at given sampling intervals. Two construction times are presented. The first one represents the cost to rebuild the KVD, while the second one builds a standard BSP for the set of triangles, with no additional point and edge cuts. The comparison between the kinetic and the interval sampling during an interval of time t needs to take into account the number of events e processed by the kinetic approach during this interval, and the number of samples s processed during this interval. The kinetic approach is advantageous over the interval approach if the average time of events times the number of events processed is smaller than the construction time times the cost to construct the tree. The difference between these values is defined as the kinetic gain:

$$kinetic\_gain = s * construction\_cost - e * average\_update\_cost \tag{8.1}$$

The kinetic gain is usually bigger when the number of events is smaller, because the time sampling parameter s is usually constant over a time interval. In cases where a lot of events happen, the kinetic gain can become negative, and therefore the interval sampling approach have a better performance. Ideas to combine both strategies are discussed in chapter 8.

8.2.4 KVD-tree for occlusion culling and shadow computation

The KVD can be used to perform occlusion culling and shadow computation. For these applications, the $\hat{z}$ direction of the vertical decomposition is defined by an euclidean point,

KVD-Tree update statistics - uniform scale				
	25 △s	50 △s	100△s	200 △s
KVD Construction	281	1204	3109	9719
BSP Construction	31	203	250	407
Average update	75	94	119	168
Average update / KVD Construction	0.110	0.168	0.081	0.041
Average update / BSP Construction	2.419	0.463	0.476	0.412

KVD-Tree update statistics - uniform density				
	25 △s	50 △s	100△s	200 △s
KVD Construction	469	1547	3079	8156
BSP Construction	46	187	266	344
Average update	87	114	119	153
Average update / KVD Construction	0.185	0.073	0.038	0.018
Average update / BSP Construction	1.89	0.609	0.447	0.444

Figure 8.8: KVD Update Statistics.

rather than as a point at infinity. If the point used is the viewpoint of the scene, the KVD can be used for occlusion culling. If the point used corresponds to the location of one light source, then the KVD can be used to compute shadow information created by this light source.

In the situation where the viewpoint is used, the KVD can perform occlusion culling by simply not traversing one of the subtrees of opaque triangle nodes. Because the additional cuts introduced in the KVD pass through the viewpoint, a triangle node serve as a single occluder to all nodes in the subtree corresponding to the halfspace that does not contains the viewer. The occlusion culling can be incorporated to the algorithm that traverses the KVD to display triangle fragments. This simple modification in the algorithm was tested with the previous datasets and results are reported in figure 8.9. The number of triangles nodes that are culled are reported along the total number of nodes in the occluded subtrees.

For the uniform scale cases, as expected, there is an increase of the percentage of occlusion culling as the number of objects increases. For the uniform density cases, the percentage of occlusion culling varies little with the different scenes. It is important to note that this algorithm only defines a node n_1 as an occluder of a node n_2 if n_2 is in the occluded subtree of n_1. In the opposite situation, where a node is occluded by several

KVD-Tree Occlusion culling - uniform scale						
	25 △s	50 △s	100△s	200 △s	400 △s	800 △s
Triangle Nodes Culled	1	11	42	329	1679	8709
Triangle Nodes Total	78	295	832	2546	8129	26466
% Triangle Nodes Culled	1.28	3.72	5.04	12.92	20.65	32.90
Tree Nodes Culled	5	25	96	732	3388	16523
Tree Nodes Total	308	1008	2604	7285	20259	57094
% Tree Nodes Culled	1.62	2.48	3.68	10.04	16.72	28.93

KVD-Tree Occlusion culling - uniform density						
	25 △s	50 △s	100△s	200 △s	400 △s	800 △s
Triangle Nodes Culled	11	22	42	164	590	1329
Triangle Nodes Total	145	316	832	2105	5568	12926
% Triangle Nodes Culled	7.58	6.96	5.04	7.79	10.59	10.28
Tree Nodes Culled	27	53	96	361	1273	2994
Tree Nodes Total	489	1045	2604	6337	15931	36550
% Tree Nodes Culled	5.52	5.07	3.68	5.69	8.01	8.19

Figure 8.9: Occlusion culling using the KVD.

nodes that belong to one of its subtrees, this algorithm do not detect occlusion culling. In order to this it would be necessary to combine occluders, which may be expensive.

A similar algorithm can be used to detect shadows if the position of a light source is used in the definition of the vertical decomposition. Like the occlusion culling algorithm, one of the subtrees of an opaque triangle node is entirely in shadow with respect to the light source (shadow subtree), while the other subtree may be illuminated by the light source (illuminated subtree). Unlike the occlusion culling algorithm, the shadow computation algorithm requires the detection of the cases where nodes are in front (with respect to the light source) to one of its ancestor nodes. For every triangle node, this corresponds to computing the shadow cast by all triangle nodes in its illuminated subtree. Some solutions for this problem have been presented in the literature [6][7]. One solution to this problem is to project every triangle node against the planes of each ancestor triangle node. The final image of the triangle node is obtained by drawing the entire fragment of the triangle node, and all projected triangles obtained as above. Another solution is to filter down the fragment associated with a triangle node. In this case, only the illuminated subtree is traversed, and the nodes in this tree are used to partition the fragment. Only the fragment parts that are not occluded by other triangle nodes are displayed.

Chapter 9

Conclusions

9.1 Main Contributions

The hidden-surface elimination problem is one of the oldest problems faced by the computer graphics community. It consists of the computation of parts of objects that are visible to a viewer, which are then combined to create an image that represents this information. A challenging variant of this problem corresponds to situations where both the objects and the viewpoint are not static, but are allowed to move. This scenario directly affects the computation of visibility information, which has to be re-computed for every image frame generated.

In this work we describe a new data structure that can be used to extract dynamic visibility information. The Kinetic Vertical Decomposition Tree (KVD) is a special type of BSP, that is used to represent a vertical decomposition of a set of triangles in $\mathbb{R}^3$. Unlike the standard BSP, the KVD introduces additional cuts from vertices and edges along specified directions.

For scenes composed of triangles moving along known trajectories, the KVD can (1) detect when an update in its structure is necessary, and (2) perform updates only in the affected parts, without requiring a complete reconstruction of the structure. The KVD was designed to perform all these tasks in the following way. First, events and certificates were defined to identify when the combinatorial structure of the KVD needs to be updated. This was only possible through an evaluation of all cases that can create changes in the KVD.

For each event, a certificate is defined to serve as proof that the KVD stays combinatorially valid.

For every certificate that fails, the nodes that define the certificate are examined, and the tree structure is locally updated using an appropriate update algorithm.

In summary, the main contributions of this work can be enumerated as follows:

1. *Design and Implementation of a 3D Kinetic BSP*: This work describes the KVD, the first fully 3D Kinetic BSP, and the first implementation of a kinetic BSP. Previous work concentrated on the theoretical analysis of kinetic BSPs. The kinetic maintenance of a BSP is much simpler to describe than it is to implement. This implementation was feasable because common substructures were identified in the many complex cases that can arise for the updates. The design of a small number of certificates and update algorithms illustrate how similar situations were handled in a unified manner.

2. *Visualization of BSPs*: The implementation of a complex structure like the KVD required a visual tool to display properties of the KVD during debugging. The ability to display geometric properties of the nodes, combined with a selection mechanism that reduces the set of nodes to be used in the visualization was extremely helpful, and can potentially be applied to the visualization of other complex spatial partitions.

3. *Algorithmic Aspects*: Several contributions can be highlighted from the current work. The idea of a symbolic representation of geometry for BSPs is extremely useful in dynamic situations. New complex BSP operations were designed to accomplish the updates in the KVD following a static priority scheme: priority merging of trees, out-of-order insertion of nodes. The dragging operation is a novel BSP operation used to merge trees while evaluating the behavior of certain nodes in the trees. This unique operation is used in several of the update algorithms. Finally, the use of the KVD as the supporting structure for a priority queue that detects the first certificates to fail was important to reduce the time spent updating certificates in the tree. This was only possible because the KVD is a binary tree, and therefore this approach may not generalize to other kinetic problems. In any case, it was the first time that such a combination was proposed.

9.2 Future Directions

There are many possible ways to continue the work described in this dissertation. In this section we review a few of these ideas.

9.2.1 Migration of Priorities

The insertion of cuts in the KVD follows a specific priority order, randomly assigned to triangles in a scene. During a kinetic simulation, this order is maintained unchanged at all times, which provides a mechanism to check the correctness of the local updates in the tree. It can be proven that for objects moving along pseudo-algebraic trajectories, the use of a fixed priority order results in trees of reasonable expected depth and size.

The worst situation for a fixed priority order approach happens when the tree updates involve higher nodes in the tree. The fundamental cost of a tree update consists of moving a node from one place to another in the tree. This movement is accomplished by first deleting the node from its previous locations, followed by its insertion into a new location. The deletion step requires an operation that merges the subtrees of the node, which is more expensive for nodes closer to the roots of the tree. Because nodes with higher priority are inserted first in the KVD, the nodes that create costly update operations correspond to high priority nodes. In an ideal situation, the priority ordering would be defined in such a way that the moving nodes are closer to the leaves, where the merging operation is usually cheaper.

One approach to reducing the number of costly updates is to create a mechanism to alter the priority ordering based on the events encountered during a number of updates in the KVD. For a higher priority node that creates several costly updates in the tree, a solution would be to change its priority in such way to reduce the costs of future events involving the node. The change in the priority order, however, needs to be done in such way that a mechanism to check the correctness of the tree can always be defined.

Suppose the nodes associated with a higher priority triangle create several events in the tree that requires costly updates in the KVD. A possible implementation of this idea for the KVD keeps a pointer to the locations of all point nodes in the tree, therefore all point nodes associated with the given triangle can be quickly accessed. When the priority of the triangle is changed, starting at each one of the the point nodes of the triangle, we delete every node defined from the points, edges and face of the triangle. This will completely remove all the nodes created by the triangle in the tree. A new priority can then be assigned to the triangle in such way that it is smaller than any priority present in the tree. Reinsert all cuts originated from the triangle in the tree, which necessarily will go to the leaves of the tree because of the small priority assigned to them.

We call this process a *migration of priorities*. In this solution, the information about the events being processed is used to modify the priorities, in order to minimize the costs of the events. This solution imposes a self-adjusting nature to the structure, which nicely handles the situations that high priority nodes cause costly updates in the tree.

9.2.2 Combination of kinetic and interval sampling

The fundamental event that requires an update in the combinatorial structure of a BSP corresponds to a change in the classification result of a node against one of its ancestors. In an auto-partition BSP, where cuts are defined only through the supporting planes of the input faces, such events correspond to a vertex passing through a plane of an ancestor node. Suppose a complex polygonal object passes through one of the cutting planes of the KVD. In this case, for every vertex of this model that passes through the cutting plane, an event is generated. In these situations, the kinetic approach may be too expensive, because many events happen in time.

One solution for this problem is to use a mixed kinetic and interval sampling approach. The kinetic sampling is extremely useful when no updates are necessary in the kinetic structure for a large period of time. However, for situations such as the one described above, an interval sampling approach, that deletes and inserts and object in specific time increments may have better performance. In other words, instead of processing every event caused when every vertex of the model crosses the plane in a kinetic way , an interval sampling approach is used. After all events are processed, the kinetic mechanism would resume control until new situations like this one happen again.

9.2.3 Topological k-D-tree

The kinetic BSPs described in this and previous work are based on cylindrical decompositions of the space. The reason for this is that these decompositions contain cells of bounded size, which allow for easier detection of events, at the expense of an increase in the size of the tree. On the other hand, it would be much better if we knew how to maintain auto-partition BSPs, but this would require maintaining all the subdivision of space, while dealing with cells of arbitrary complexities.

An intermediate solution can be designed as follows. Suppose the construction of the BSP creates some constraints on the cuts in such a way that the complexity of its cells is always bounded. Let A be a rectangular (or cylindrical) region of the space. We define a *valid* cut in this region if it cuts opposite walls of this region. Note that the cut does not need to be parallel to the walls of the region. This cut creates two new regions and we define valid cuts in these regions in the same way. In other words, cuts are made in space in such way that the region is always defined by four sides (in the plane), and therefore fixed-size cells are always obtained. The set of candidate cuts is composed of all supporting planes of the input model. If all candidate cuts satisfy this property, the resulting structure is an auto-partition BSP with cells of bounded complexity. We call the resulting structure a topological k-D-tree, because a simple transformation in the cuts used can produce a k-D-tree, a tree structure composed by cuts orthogonal to the coordinate planes of a space.

In general, such an auto-partition is hard to obtain, because in some situations no valid cuts can be defined. In these cases, an external valid cut is created and inserted into the tree. In summary, this approach uses auto-partition cuts as much as possible, and only inserts external cuts when necessary. The reduction of the number of external cuts helps improve the performance of the kinetic simulation. On the other hand, kinetic updates become more complex, because new external cuts may need to be inserted, or even old ones deleted when events are processed.

9.2.4 Kinetic k-D-tree

Suppose all objects in an input scene have associated aligned bounding boxes. Assume for simplicity that either the objects are moving along linear trajectories, or that the motion is more complex but the bounding box contains the object for any possible orientation of the object.

We create a kinetic BSP using the planes that support the faces of all bounding boxes of objects. Note that the faces of the objects are not inserted in this tree. Because all the faces are aligned, the resulting subdivision has cells of bounded complexity. In fact, this structure is a particular case of the KVD, and can be easily implemented from the current implementation of the KVD. We call this structure a Kinetic k-d-tree because the resulting tree contains cuts that are orthogonal to coordinate planes (a k-d-tree).

The problem is that because no cuts were introduced by the objects, no fragments are stored in the tree, and this structure can not be used as before to extract visibility ordering. The idea is to use the structure of this tree to extract visibility ordering by a more direct approach, that compares objects directly. The hierarchical structure of this tree can be used to reduce the number of tests to be performed. In between updates in this structure, a previously computed visibility ordering remains valid in a kinetic sense, which means that it will only be violated when an certificate fails.

9.3 Conclusion

We hope that this dissertation motivates new research in an area that contains extremely hard and exciting problems.

Bibliography

[1] Pankaj K. Agarwal, Jeff Erickson, and Leonidas J. Guibas. Kinetic BSPs for intersecting segments and disjoint triangles. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 107–116, 1998.

[2] Pankaj K. Agarwal, Leonidas J. Guibas, T. M. Murali, and Jeffrey Scott Vitter. Cylindrical static and kinetic binary space partitions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 39–48, 1997.

[3] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 7th SIAM Symp. on Discr. Algorithms*, page to appear, 1997.

[4] J. Basch, Leonidas J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 747–756, 1997.

[5] T. Cassen, K.R. Subramanian, and Z. Michalewicz. Near-optimal construction of partitioning trees by evolutionary techniques. In *Proceedings of Graphics Interface '95*, pages 263–270, May 1995.

[6] Norman Chin and Steven Feiner. Near real-time shadow generation using bsp trees. In Richard J. Beach, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 99–106, August 1989.

[7] Yiorgos Chrysanthou. *Shadow Computation for 3D Interaction and Animation.* Ph.D. thesis, Queen Mary and Westfield College, University of London, 1996.

[8] Joao Comba and Bruce Naylor. Conversion of binary space partitioning trees to boundary representation. In *Proceedings of Theory and Practice of Geometric Modeling '96*, October 1996.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[10] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics (SIGGRAPH '80 Proceedings)*, 14(3):124–133, July 1980.

[11] Dan Gordon and Shuhong Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics and Applications*, 11(5):79–85, September 1991.

[12] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[13] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[14] Bruce Naylor. SCULPT an interactive solid modeling tool. In *Proceedings of Graphics Interface '90*, pages 138–148, May 1990.

[15] Bruce Naylor. Constructing good partition trees. In *Proceedings of Graphics Interface '93*, pages 181–191, Toronto, Ontario, Canada, May 1993. Canadian Information Processing Society.

[16] Bruce Naylor, John Amanatides, and William Thibault. Merging BSP trees yields polyhedral set operations. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 115–124, August 1990.

[17] Bruce F. Naylor. Interactive solid geometry via partitioning trees. In *Proceedings of Graphics Interface '92*, pages 11–18, May 1992.

[18] Bruce F. Naylor. Partitioning tree image representation and generation from 3D geometric models. In *Proceedings of Graphics Interface '92*, pages 201–212, May 1992.

[19] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[20] David F. Rogers. *Procedural Elements for Computer Graphics - Second Edition.* WCB/McGraw-Hill, 1998.

[21] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

[22] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[23] J. Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations.* Academic Press, New York, NY, 1991.

[24] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, March 1974.

[25] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 153–162, July 1987.

[26] Enric Torres. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In C. E. Vandoni and D. A. Duce, editors, *Eurographics '90*, pages 507–518. North-Holland, September 1990.

[27] G. Vanecek, Jr. Brep-index: a multidimensional space partitioning tree. *Internat. J. Comput. Geom. Appl.*, 1(3):243–261, 1991.

[28] Mary C. Whitton Willian F. Garret, Henry Fuchs and Andrei State. Real-time incremental visualization of dynamic ultrasound volumes using parallel bsp trees. In Roni Yagel and Gregory M. Nielson, editors, *Visualization '96*, pages 235–240. IEEE Computer Society, September 1996.