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

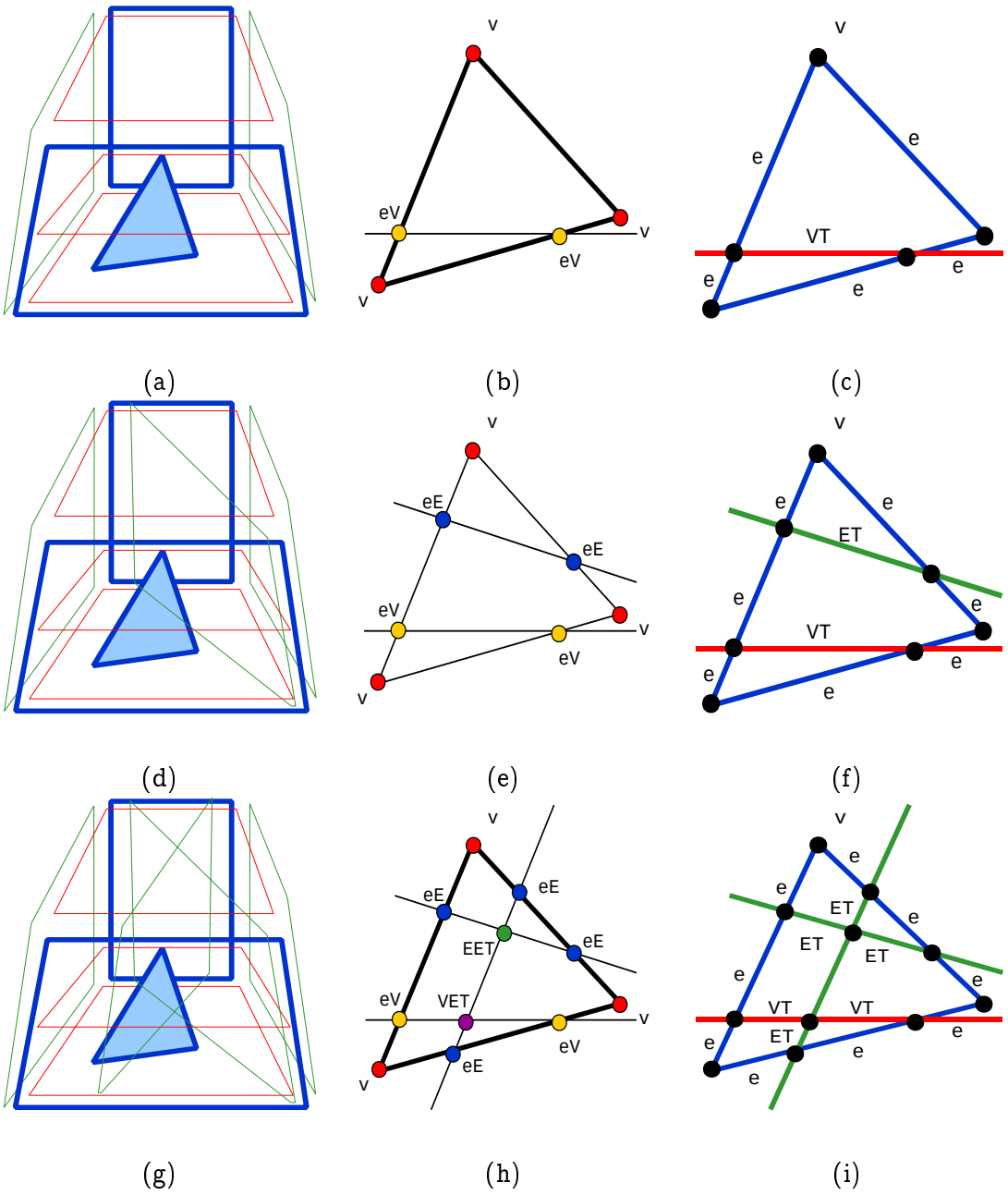

Figure 5.1: Possible types of vertices and edges that can appear in edge and triangle fragments. (a) Sample scene with one point cut. Types of vertices (b) and edges (c) corresponding to scene in (a). (d)-(f) Scene, vertices and edges for one point and edge cut. (g)-(i) Scene, vertices and edges for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

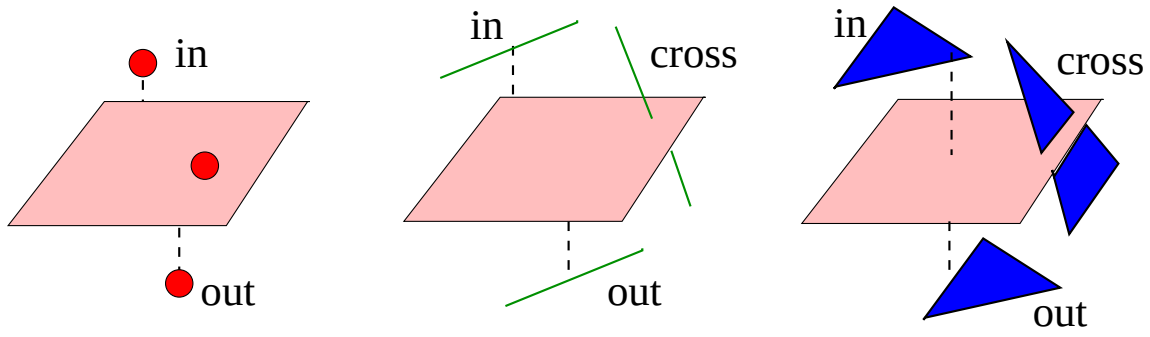

Figure 5.2: Possible Types Of Classification Results

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.

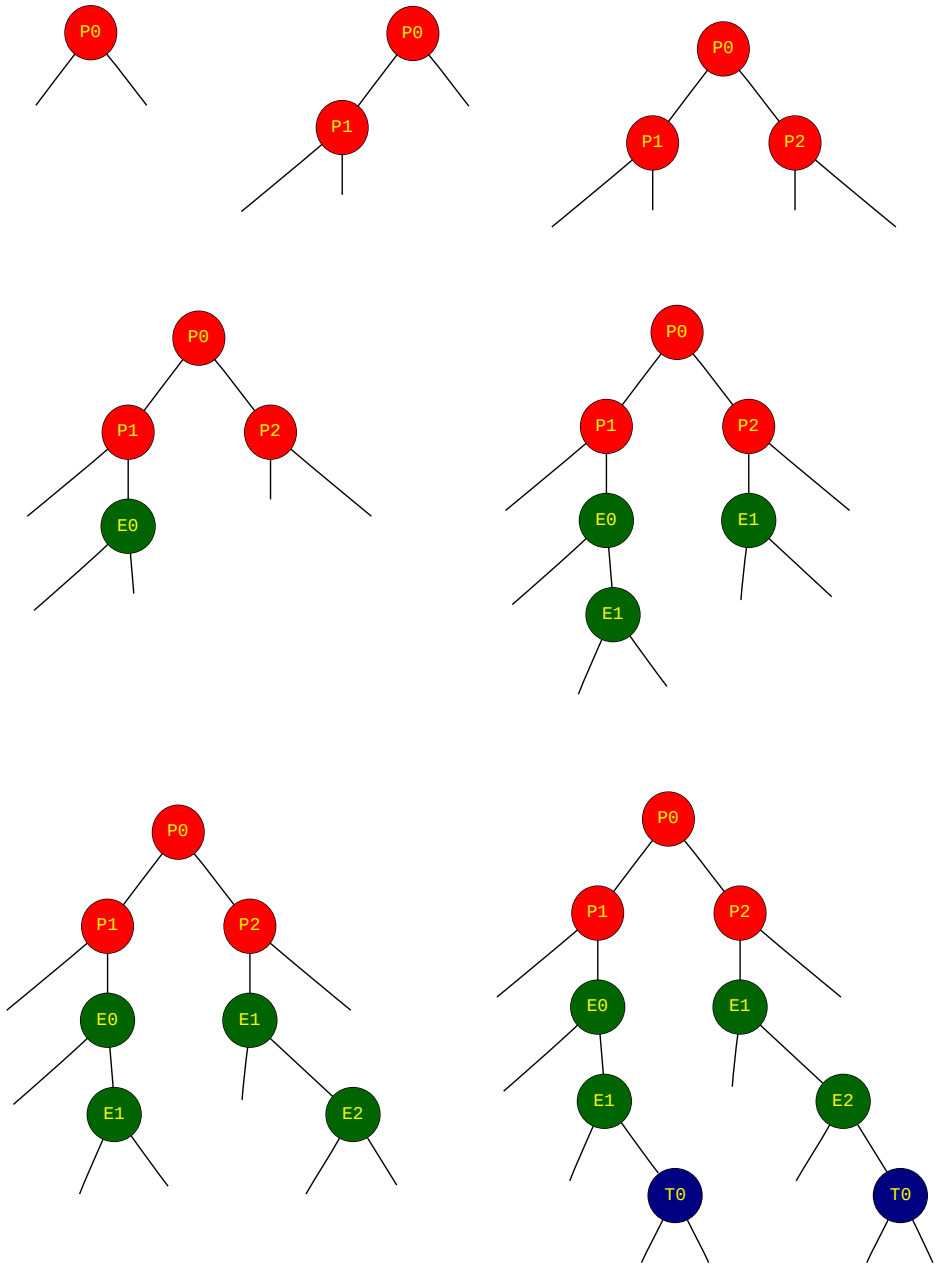

Figure 5.3: Incremental construction of the KVD for a single triangle. Point nodes are the first ones inserted (red nodes), followed by edge nodes (green) and triangle nodes (blue).

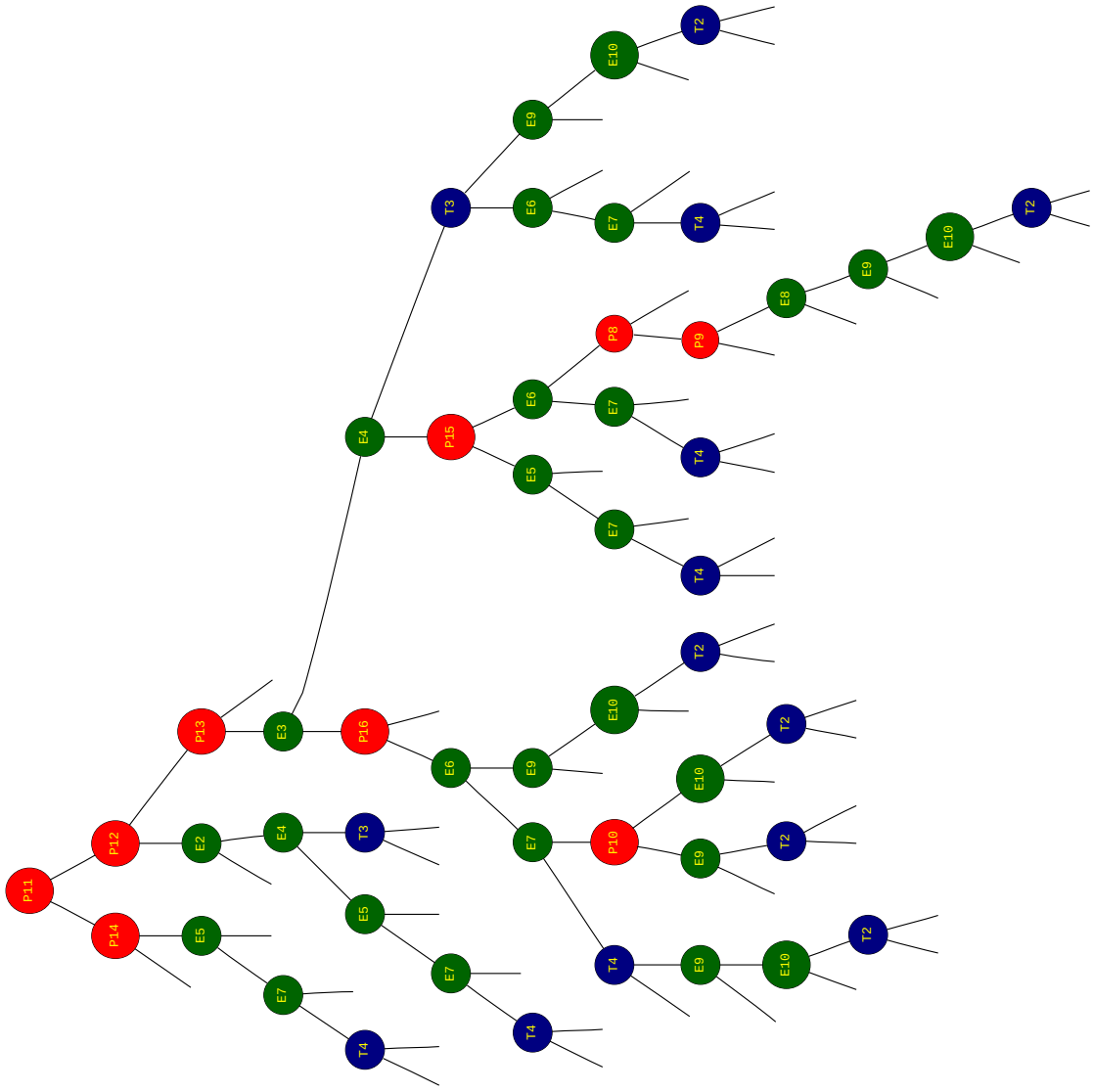

Figure 5.4: KVD tree for a scene with three triangles that cause a cycle in the visibility graph.

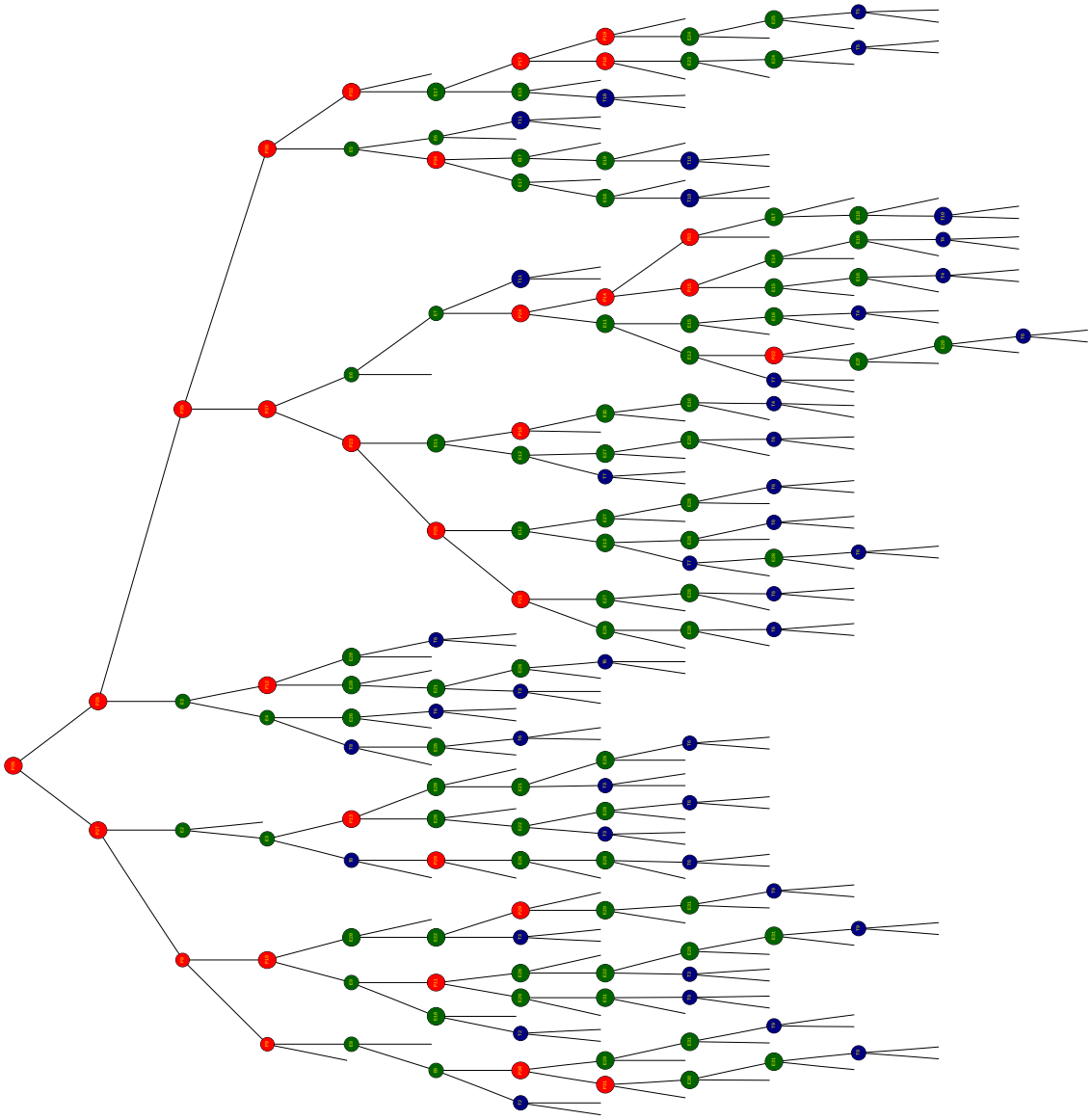

Figure 5.5: KVD for a scene with 10 triangles.

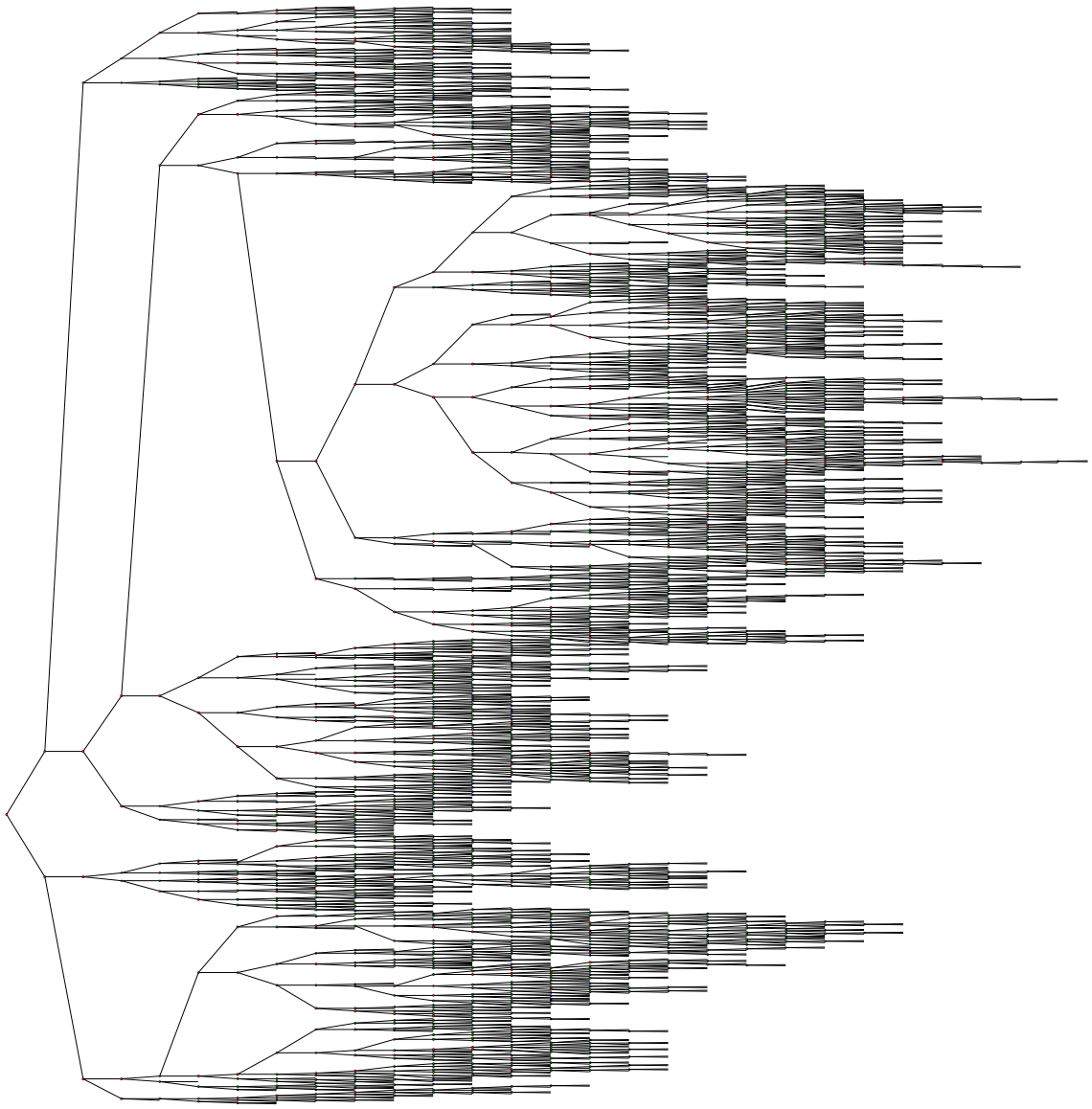

Figure 5.6: KVD for a scene with 100 triangles.

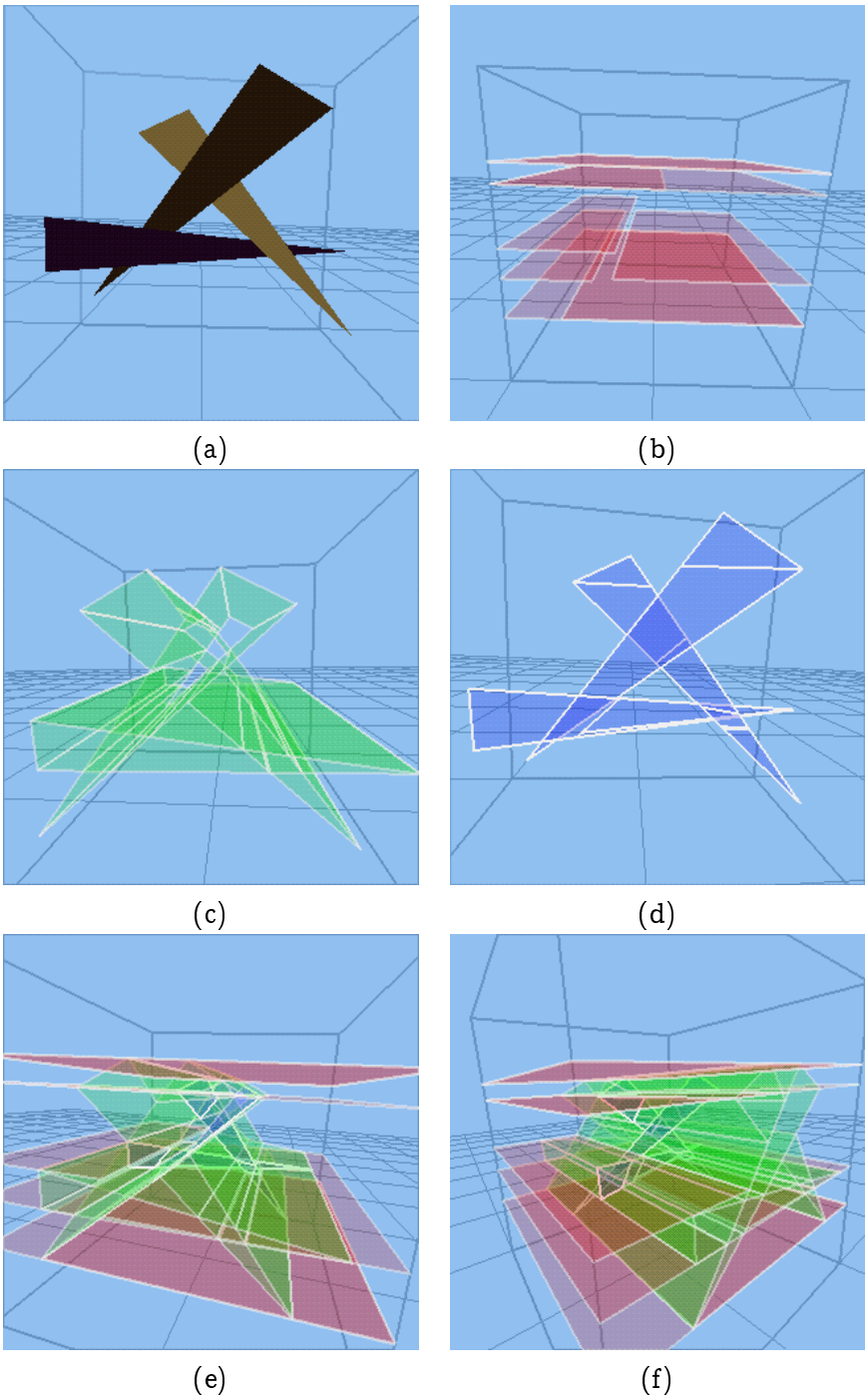

Figure 5.7: KVD decomposition for a scene with three triangles that causes a cycle in the visibility graph.

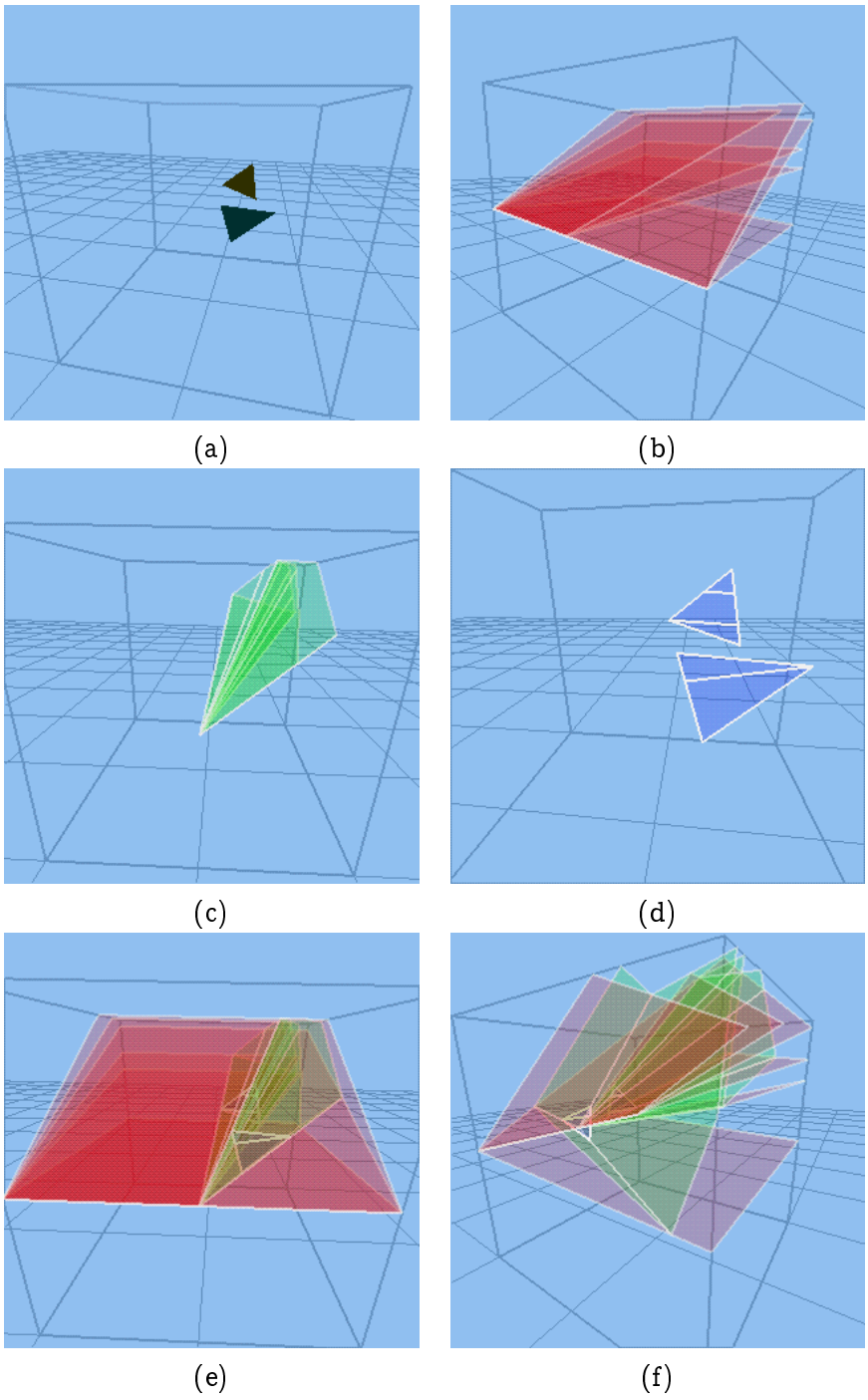

Figure 5.8: KVD decomposition with for a scene with two triangles with one direction of the vertical decomposition given by a euclidian point.

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 $\mathrm{cl}(n, ancestor(n))$, 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: $\mathrm{cl}(\mathrm{P},\mathrm{P})$, $\mathrm{cl}(\mathrm{P},\mathrm{E})$, $\mathrm{cl}(\mathrm{P},\mathrm{T})$, $\mathrm{cl}(\mathrm{E},\mathrm{P})$, $\mathrm{cl}(\mathrm{E},\mathrm{E})$, $\mathrm{cl}(\mathrm{E},\mathrm{T})$, $\mathrm{cl}(\mathrm{T},\mathrm{P})$, $\mathrm{cl}(\mathrm{T},\mathrm{E})$ and $\mathrm{cl}(\mathrm{T},\mathrm{T})$. 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

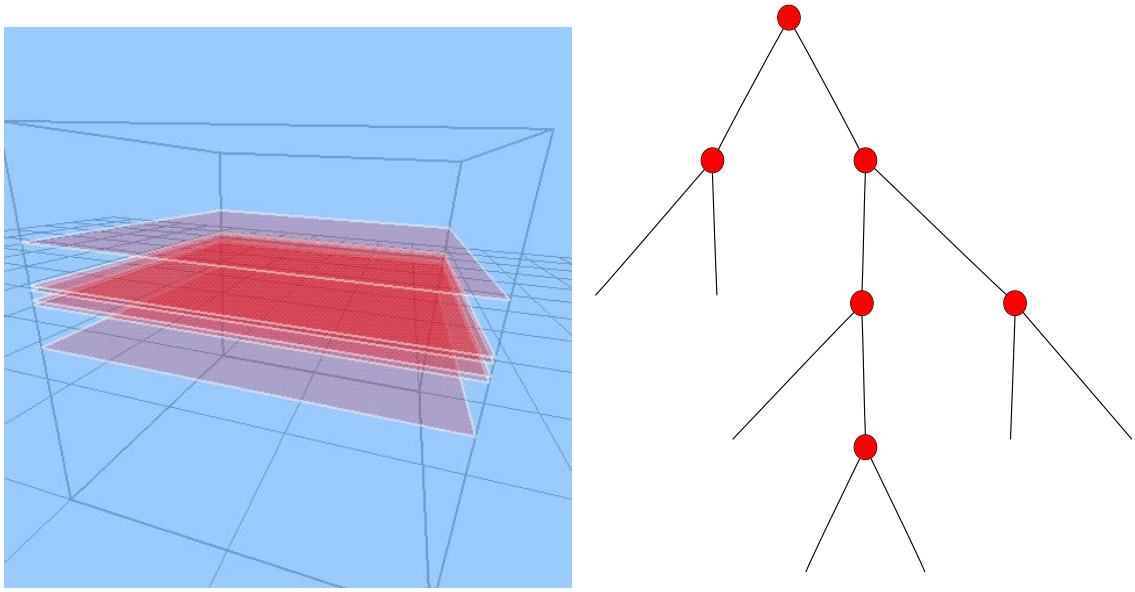

Figure 6.1: KVD containing only point cuts. (a) vertical decomposition, (b) tree structure.

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)

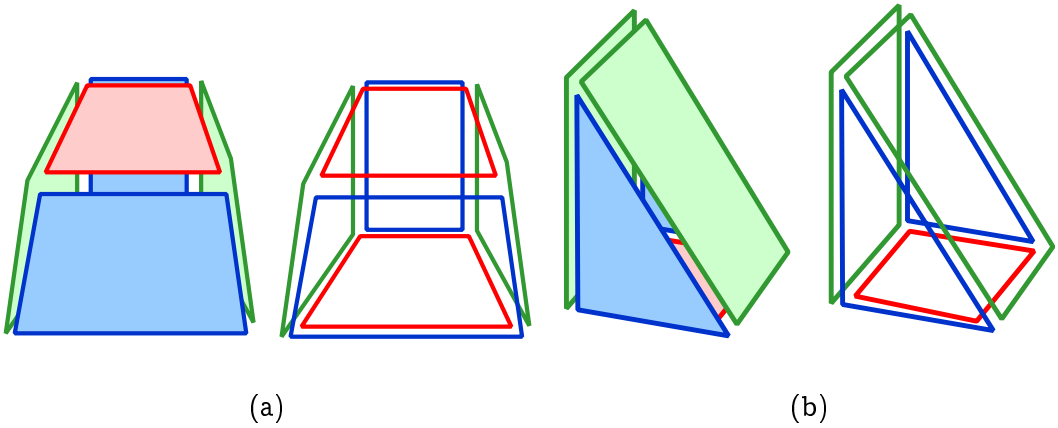

Figure 6.2: KVD cylindrical cells. (a) six-sided cells, (b) five-sided cells

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

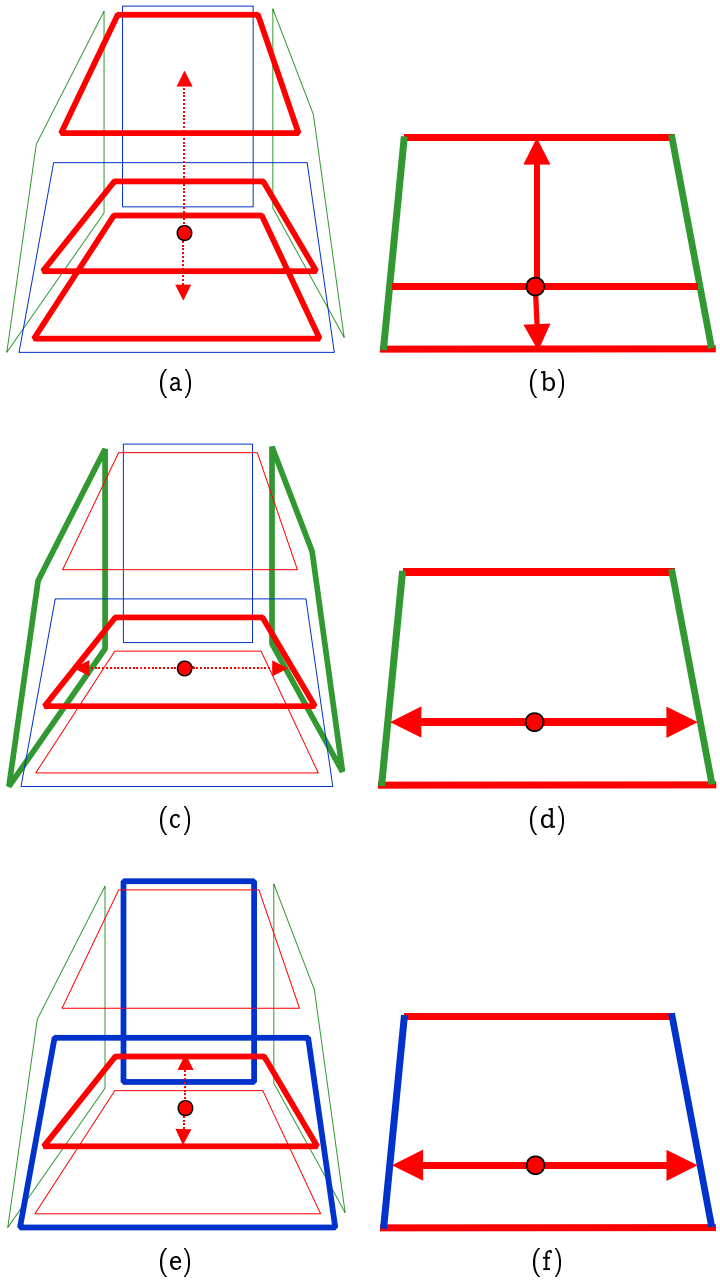

Figure 6.3: Vertex events. (a) VV event in 3D and (b) corresponding 2D view. (c) VE event in 3D and (d) corresponding 2D view. (e) VT event in 3D and (f) corresponding 2D view.

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* ($\mathrm{LFC}(\mathrm{f})$) 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* ($\mathrm{PFC}(\mathrm{cf}_1, \mathrm{cf}_2)$) 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

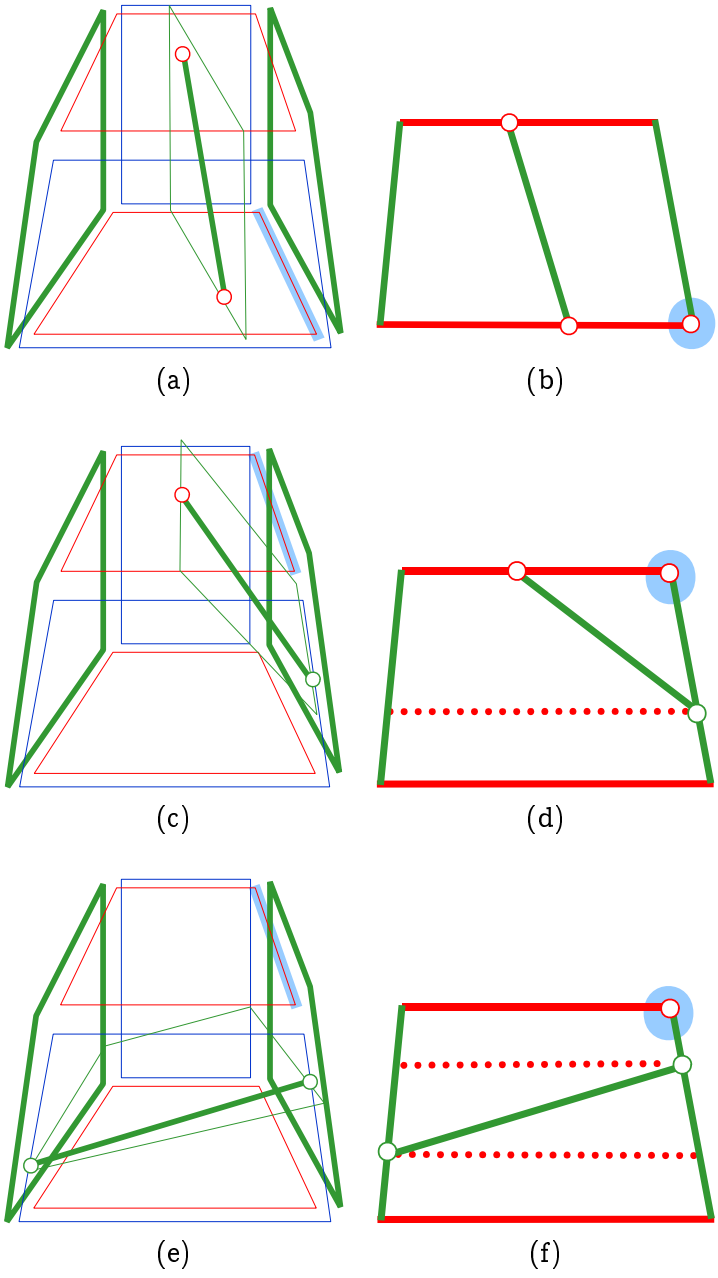

Figure 6.4: Edge events in six-sided cylindrical cells. The edge of the cyindrical cell that is crossed by the exiting edge is highlighted. (a) PFC(P,P) case in 3D and (b) corresponding 2D view. (c) PFC(P,E) case in 3D and (d) corresponding 2D view. (e) PFC(E,E) and corresponding 2D view.

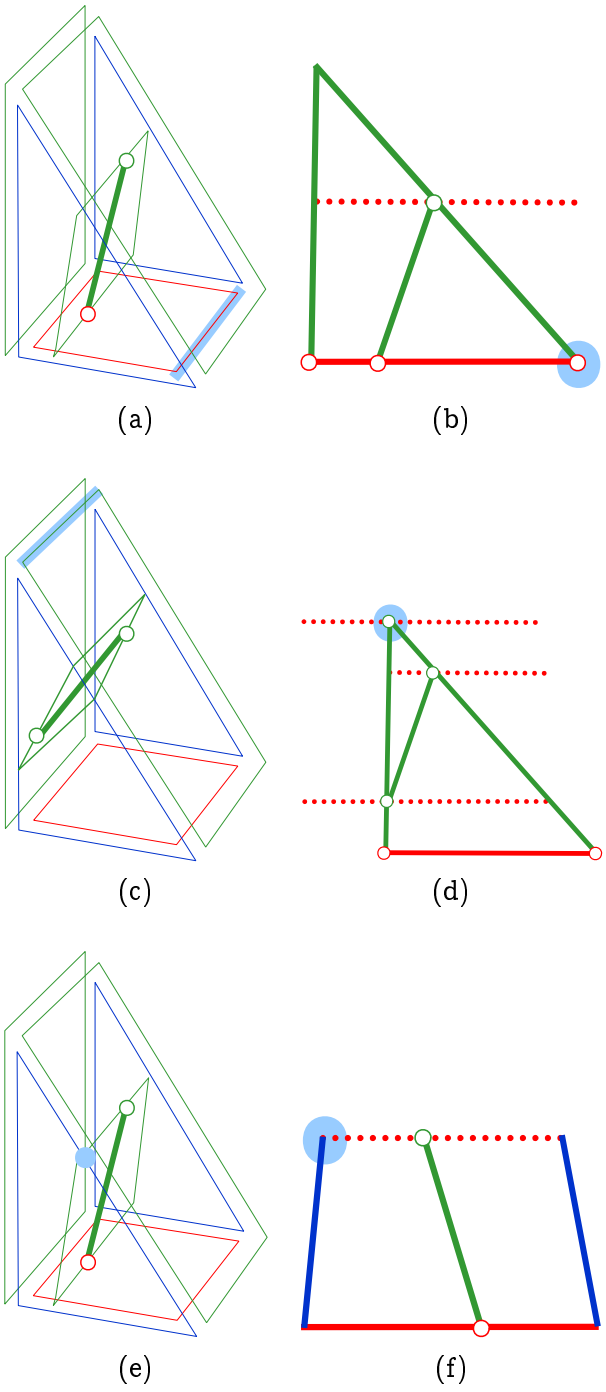

Figure 6.5: Edge events in five-sided cylindrical cells. (a) PFC(P,E) case in 3D and (b) corresponding 2D view. (c) PFC(E,E) case in 3D and (d) corresponding 2D view. (e) PFC(P,P) and corresponding 2D view.

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.

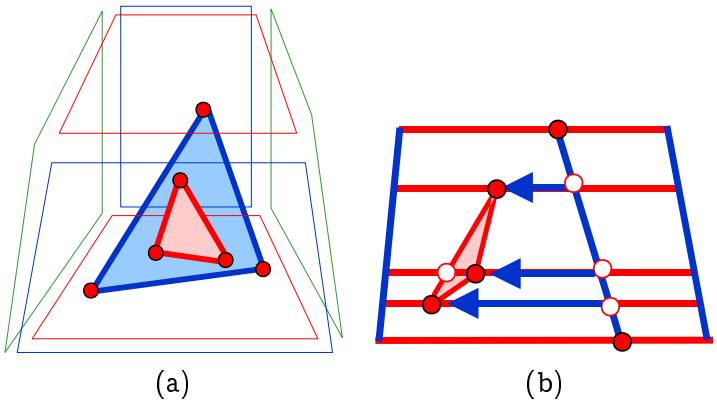

Figure 6.6: TT event cause by a triangle-vertex collision. (a) 3D view and corresponding 2D view.

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.

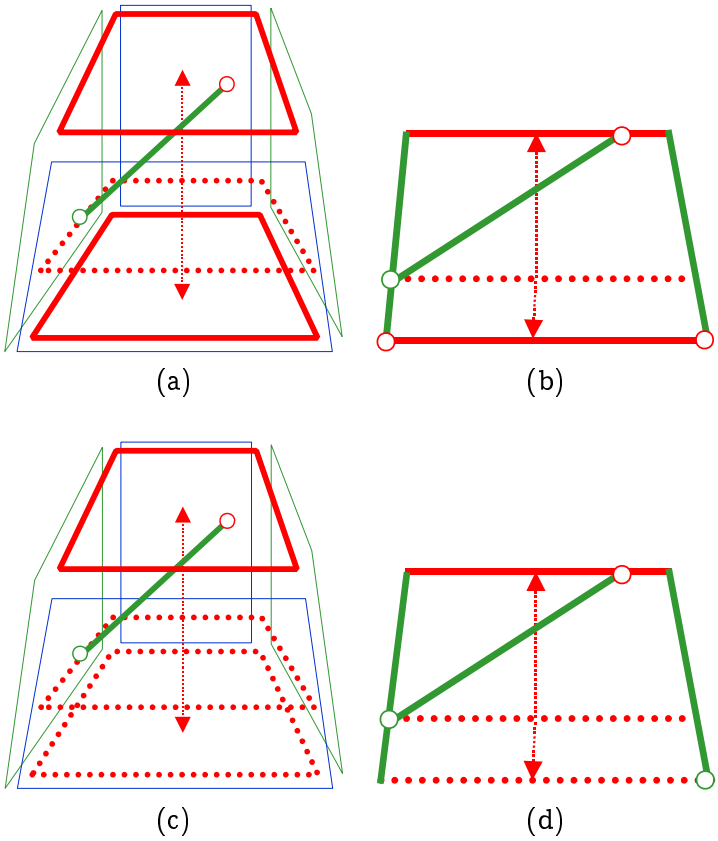

Figure 6.7: Intersection events. (a) 3D view and corresponding 2D view of a IV event. (b) 3D view and corresponding 2D view of an II event.

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:

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

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:

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

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

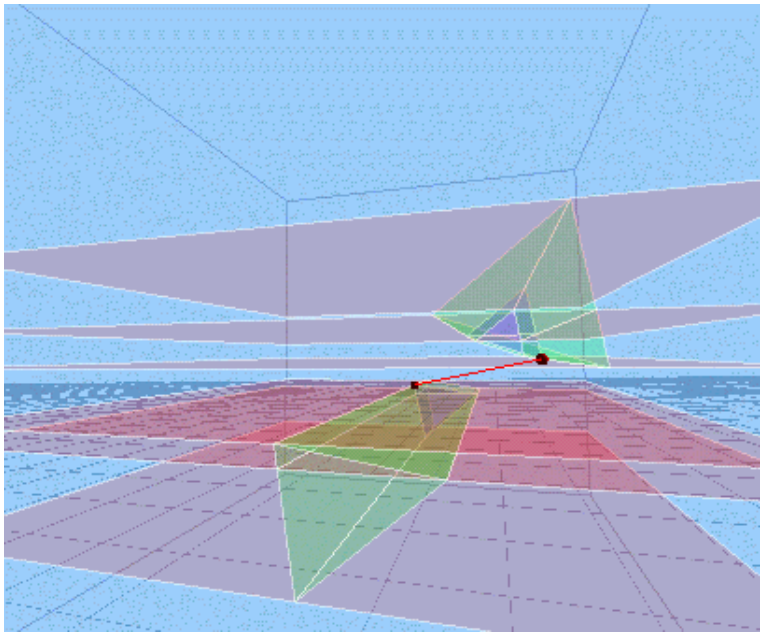

Figure 6.8: VV certificate.

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.

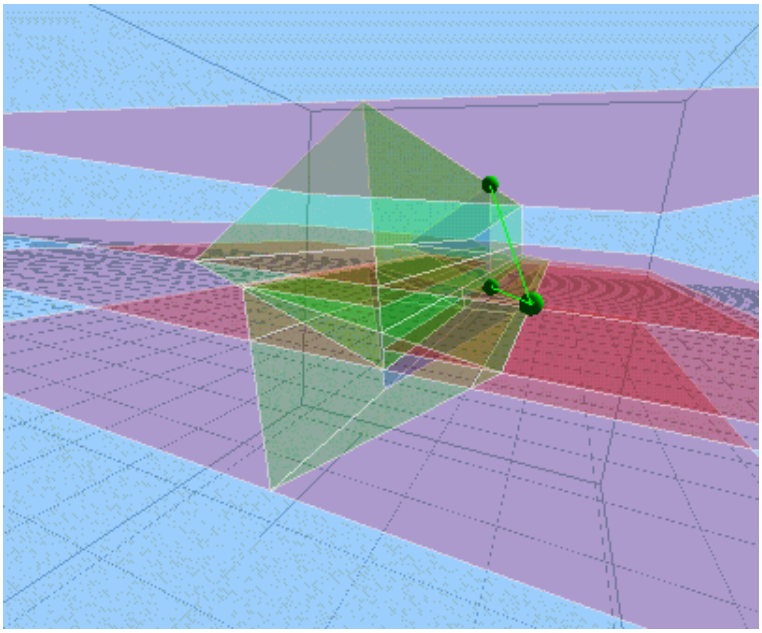

Figure 6.9: VE certificate.

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:

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

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.

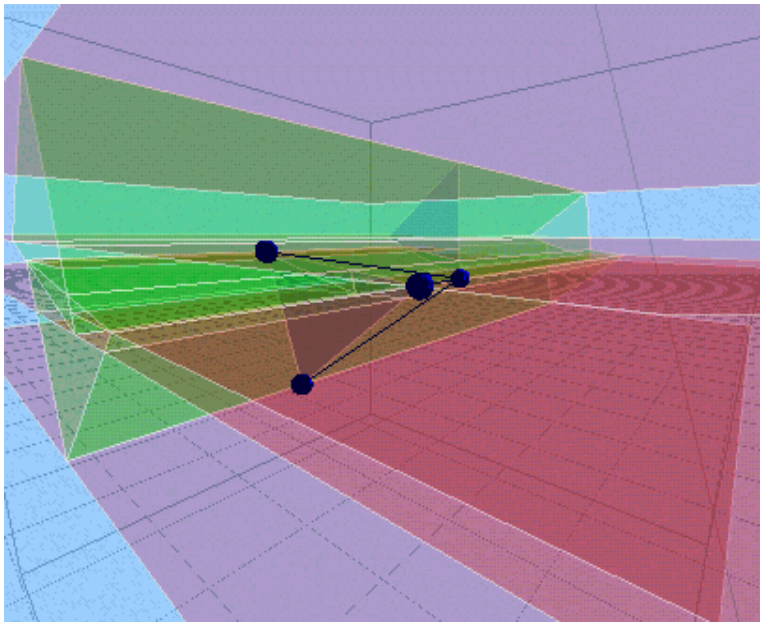

Figure 6.10: VT certificate.

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.

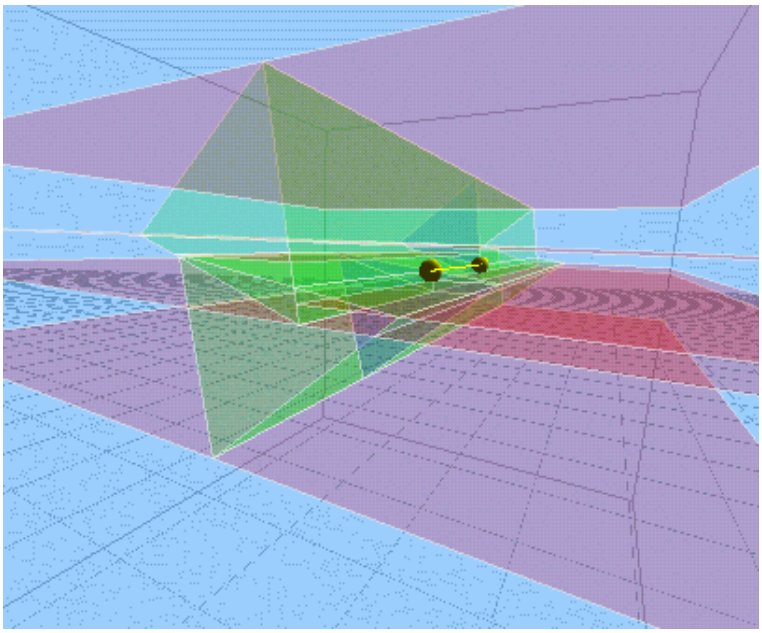

Figure 6.11: ET certificate.

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: $\mathrm{ET}(p_i, e)$
- Thread endpoint: $\mathrm{VE}(p_i, e)$

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.

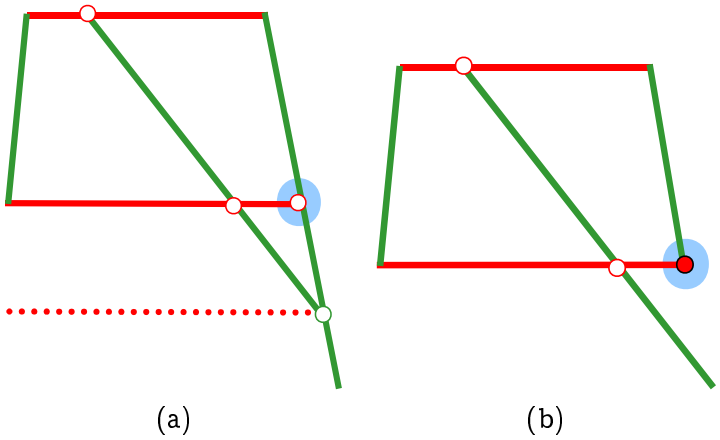

Figure 6.12: Cases that can happen for the PPC(P,P) case and the edge passing through an edge wall.

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:

- $VT(t, p_u)$
- $VT(t, p_d)$

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:

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

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:

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

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

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.