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

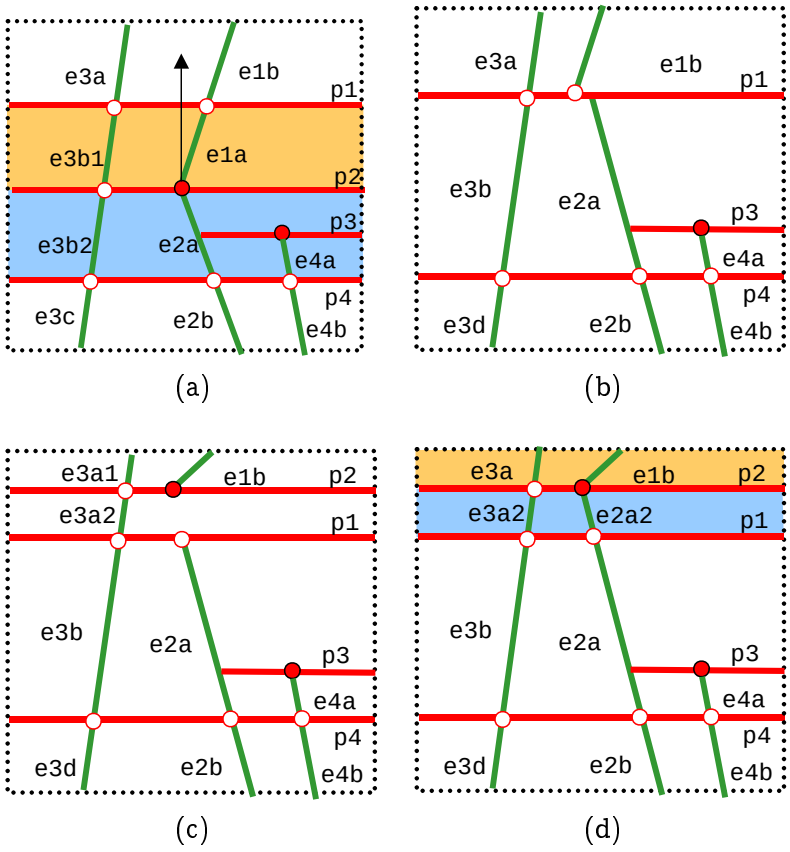


Figure 7.1: Sample example describing updates in a *VV*-event. (a) Initial configuration. (b) Configuration after removing point node. (c) Insertion of point node in its new location. (d) Insertion of incident nodes in new location.

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

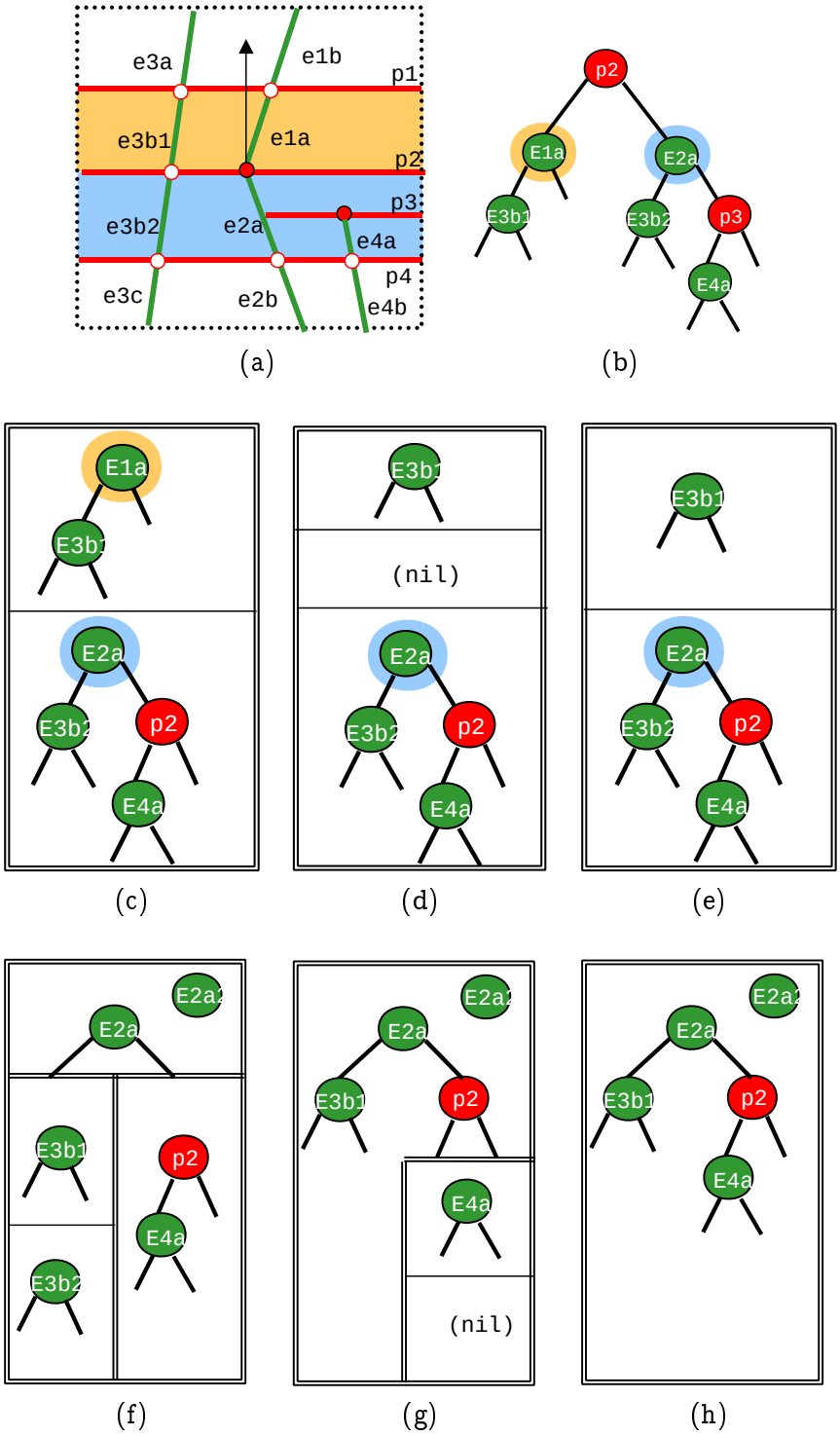


Figure 7.4: Dragging trees example.

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 $left(p_l)$ and $right(p_l)$ 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

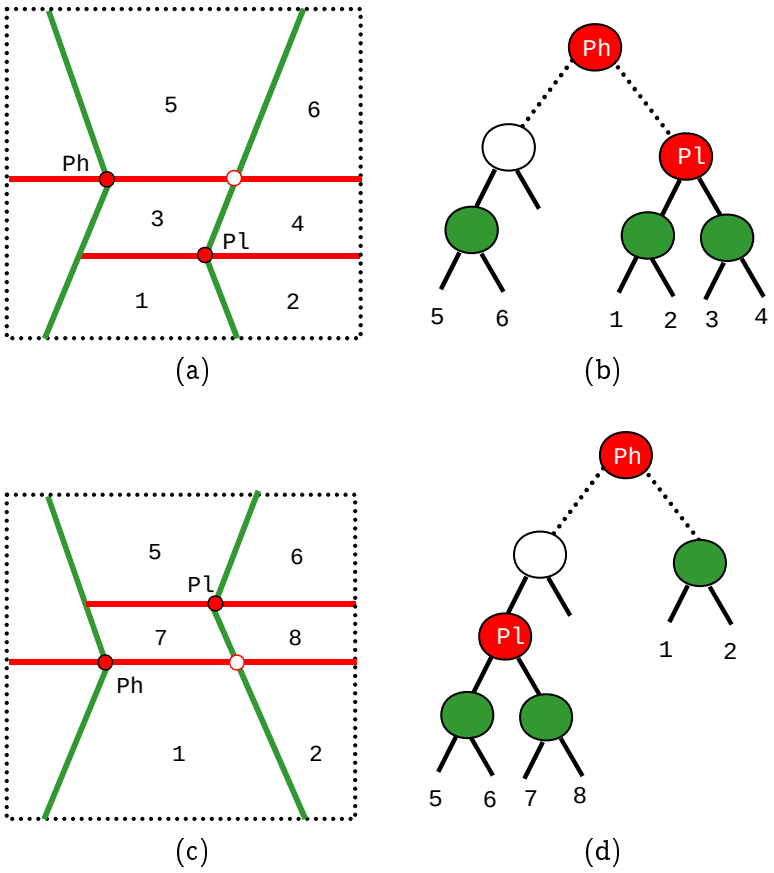


Figure 7.6: *VV* Update Example.

- 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

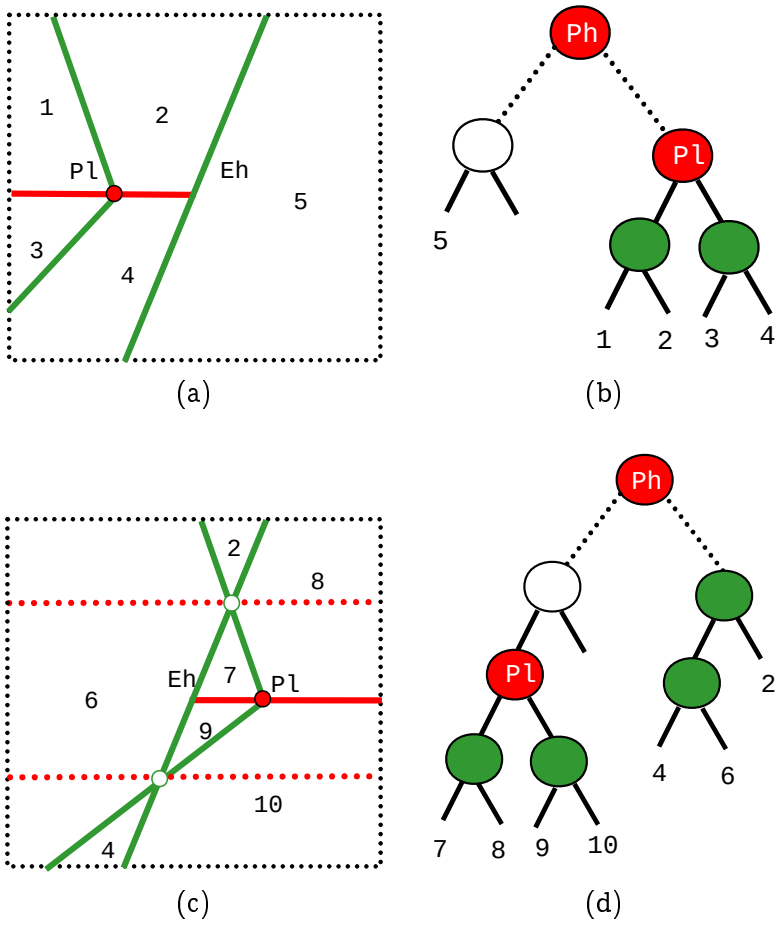


Figure 7.7: VE Update Example.

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.

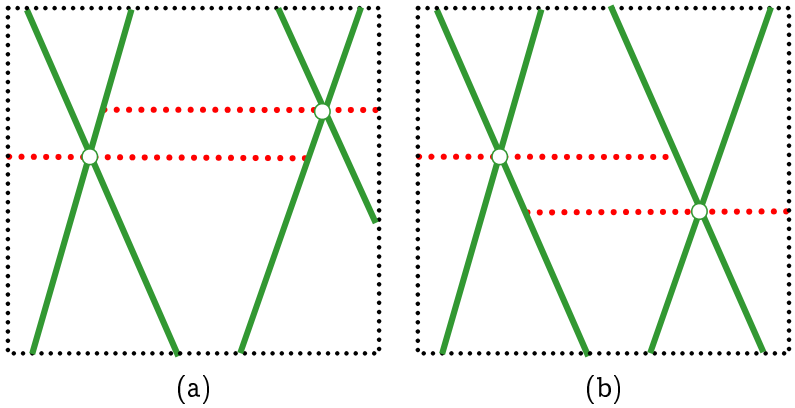


Figure 7.9: Intersection event that does not have an associated edge event.

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

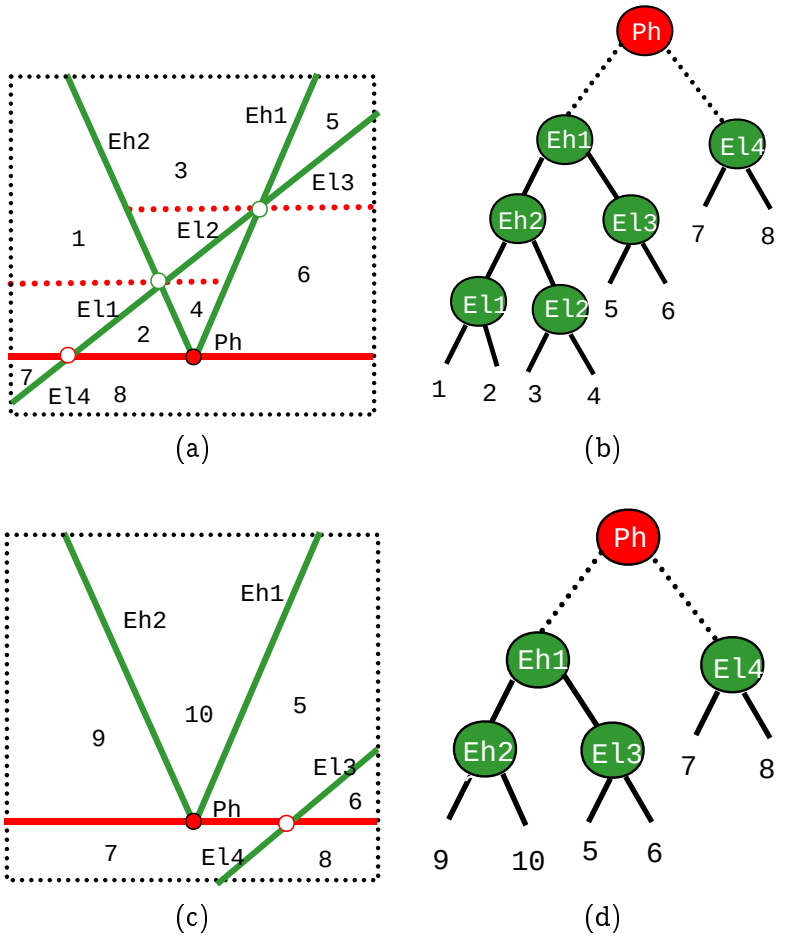


Figure 7.10: Edge events detected by VI and II certificates.

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

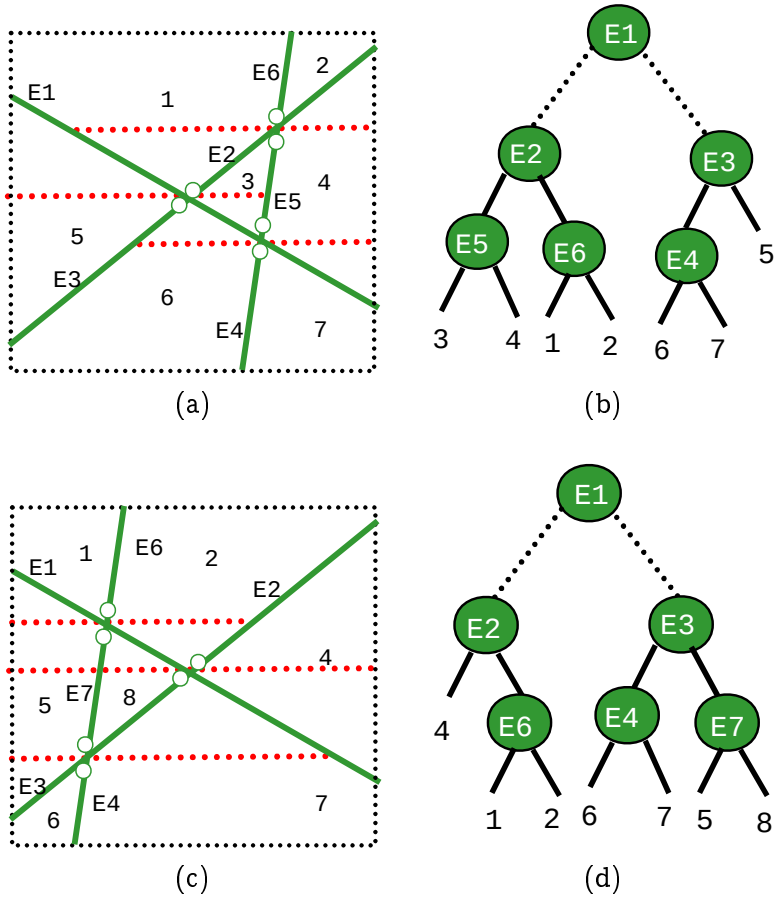


Figure 7.11: Edge events detected by II certificates.

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.

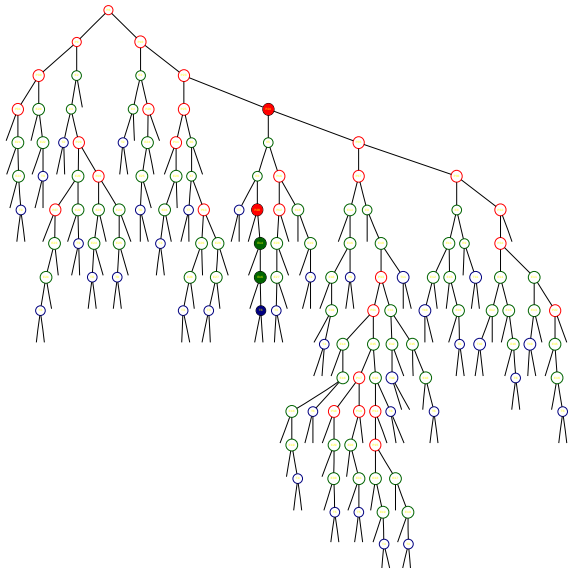
Because objects are assumed to have rigid motions, every node that is created from certain object needs to have its certificates updated. Instead of locating all of these nodes in the tree, only the point nodes created from these objects are marked with invalid certificates. This can be accomplished 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);
}
```

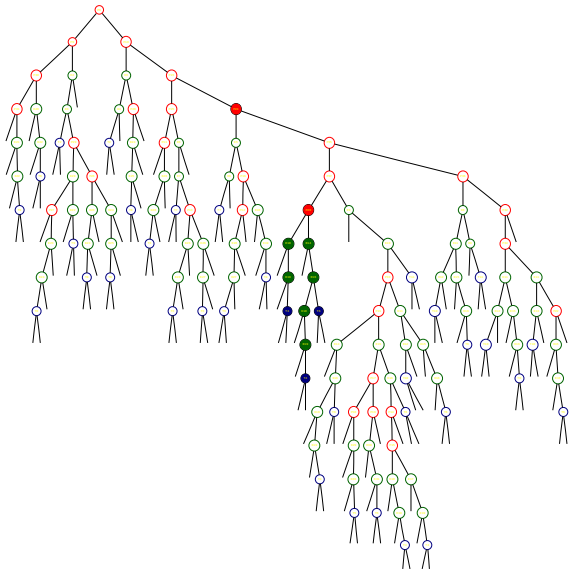
Figure 7.13: Algorithm x-Collide

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.

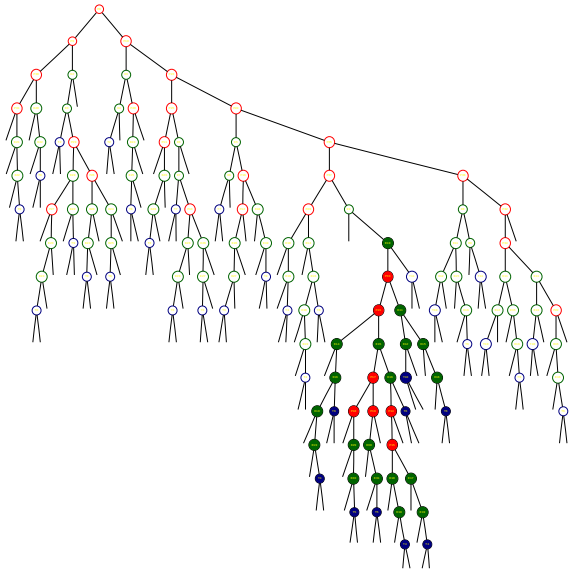


(a)

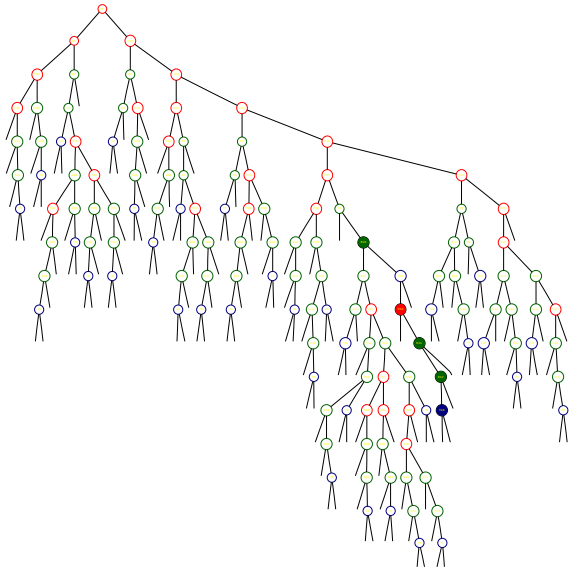


(b)

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.

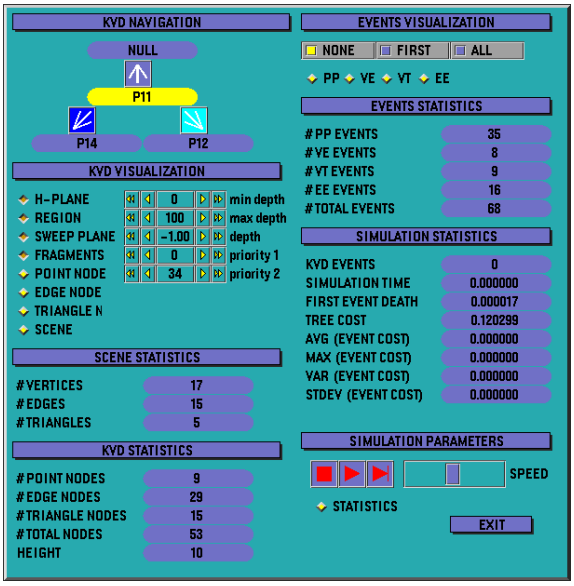
There are two variants of the *movingWorlds* program. First, a non-graphical version is just concerned with running the simulation, without producing any visual illustration about the KVD. This program is very useful once the implementation is complete, and therefore when our primary concern is to verify 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.

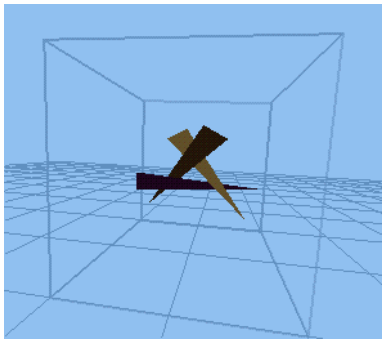
The interaction window is divided into several areas, either used to input user selections, or to display properties present in the KVD. Each of these areas is described in detail below.

Visualization Section

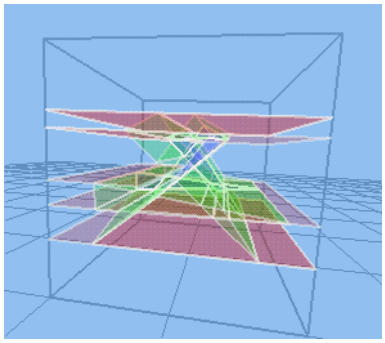
This section of the interaction window contains flags that control several aspects of the visualization of the KVD. There are three mechanisms used to select nodes in the tree: interactive navigation, procedural selection, and sweeping plane. In the interactive navigation approach, one node is always 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



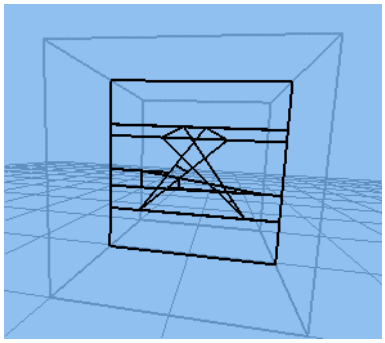
(a) Interaction window.



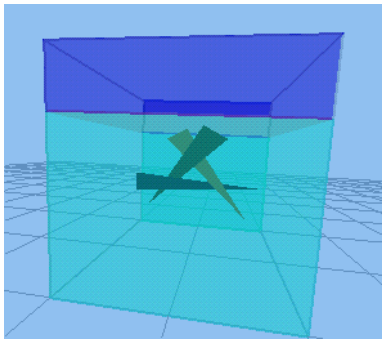
(b) Sample scene



(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.

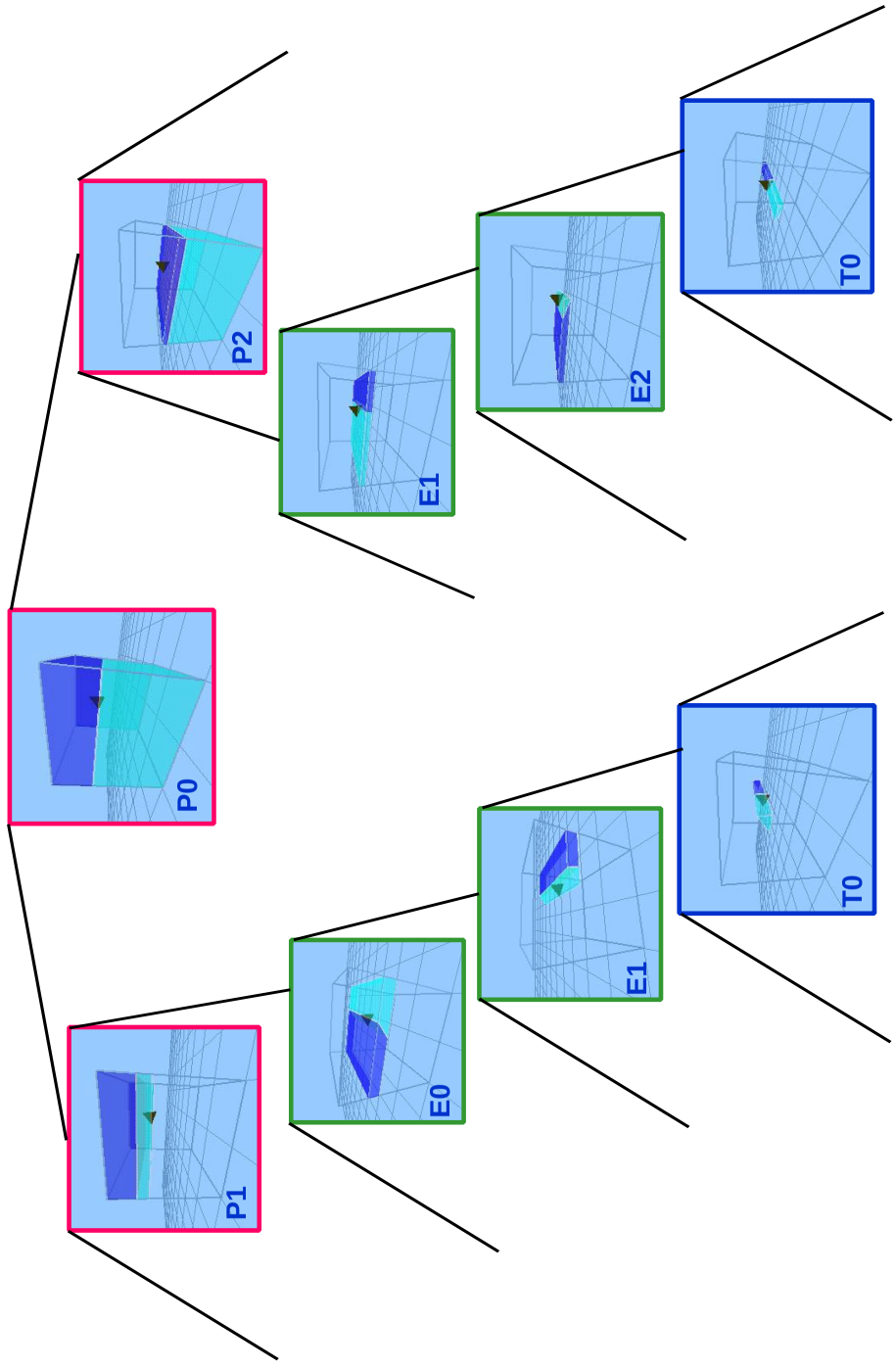


Figure 8.2: KVD tree structure combined with the visualization of the regions of each node.

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.

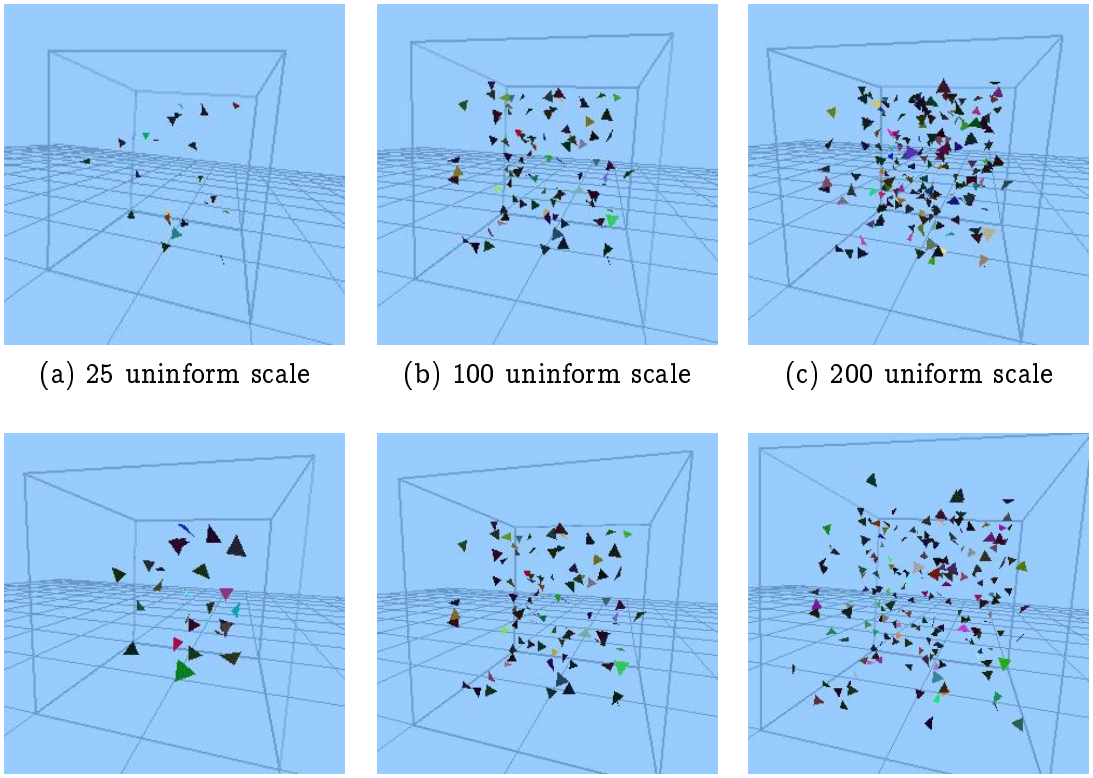
8.2 Kinetic Simulations

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

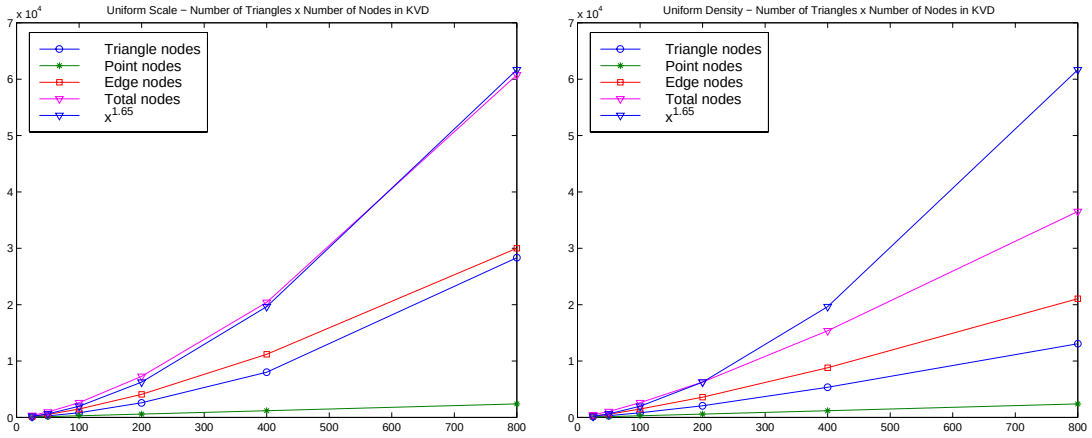
(a) 25 uninform density (b) 100 uninform density (c) 200 uniform density

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.



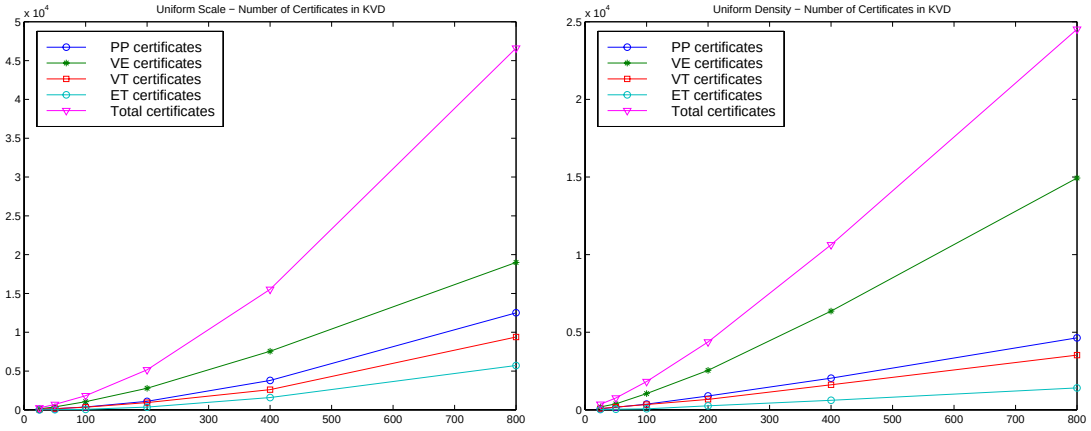
KVD-Tree construction statistics - scenes with triangles - uniform scale						
	25 △s	50 △s	100 △s	200 △s	400 △s	800 △s
Vertices	75	150	300	600	1200	2400
Edges	75	150	300	600	1200	2400
Triangles	25	50	100	200	400	800
KVD P-nodes	75	150	300	600	1200	2400
KVD E-nodes	191	531	1489	4086	11195	30013
KVD T-nodes	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

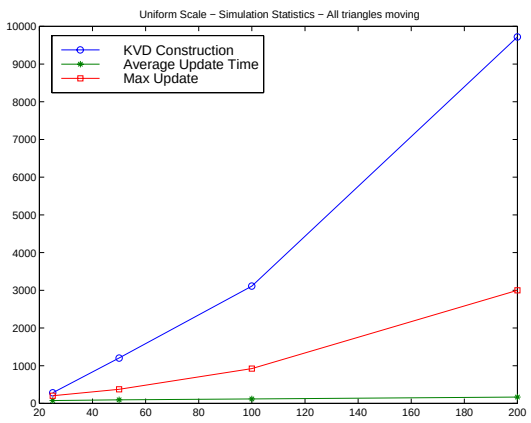
Figure 8.5: KVD Certificate Statistics.

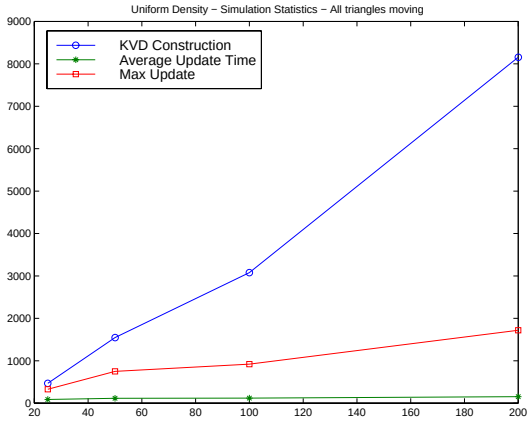
The PP certificates include all certificates of types VV, VI and II. The type of certificate VE contains the largest number of certificates. One reason to have a greater number of VE certificates than PP certificates is because they are always defined twice for six- and five-sided cells, while VV events 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.

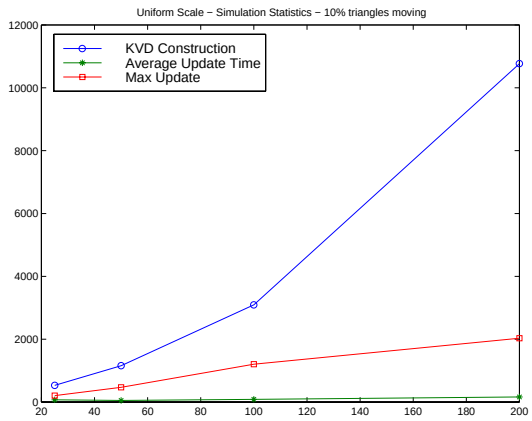


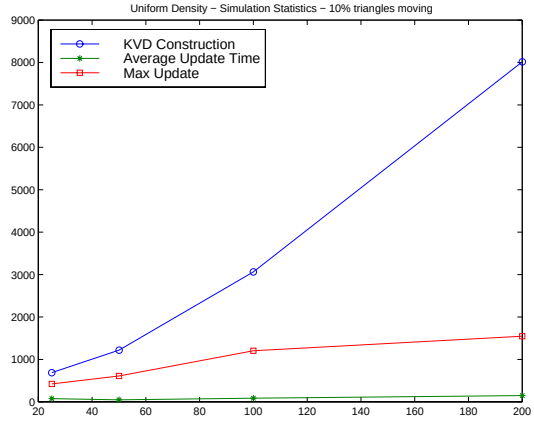


KVD-Tree simulation - 100% moving - uniform scale				
	25 △s	50 △s	100△s	200 △s
KVD Construction	281	1204	3109	9719
Average update	75	94	119	168
Max update	204	375	922	3000
Standard Deviation	24	38	69	177
Tree update / Total Update	0.79	0.75	0.72	0.68
Certificates update / Total Update	0.21	0.25	0.28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

$$kinetic\_gain = s * construction\_cost - e * average\_update\_cost \qquad (8.1)$$

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.