A survey of algorithms for construction of optimal Heterogeneous Bounding Volume Hierarchies

Kasper Amstrup Andersen and Christian Bay

Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
spreak@spreak.dk, taz@nakke.dk

Abstract

In this paper we describe algorithms for automatic object hierarchy construction. We also describe a set of properties that hierarchies must possess to be considered optimal. Based on knowledge from existing algorithms we develop and evaluate a Branch and Bound construction algorithm that searches for good hierarchies, but due to computation time, not the globally optimal hierarchy. The algorithm is designed to construct hierarchies used for collision detection. This is controlled by a heuristic- and cost function. The hierarchy data structure built by the algorithm is the Approximating Hybrid Bounding Volume Hierarchy. It has nodes with varying branching factors, multiple bounding geometry types and approximating geometries in the leaf nodes.

Keywords: Hierarchy construction, bounding volume hierarchies, collision detection.

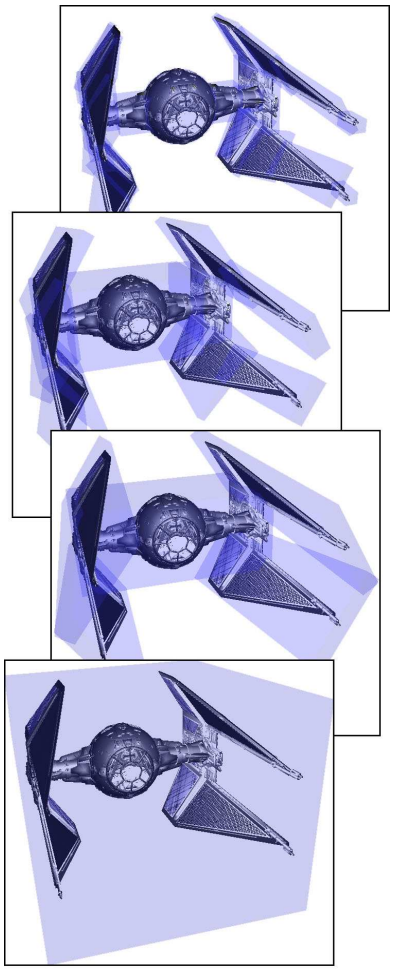


Figure 1: A hierarchy constructed with the Branch and Bound algorithm presented in this paper.

1 Introduction

In graphics there are two fundamentally different types of hierarchies: Space hierarchies and object hierarchies [Haber et al., 1996]. Space hierarchies are used for partitioning space into subregions. Typical examples are Octrees and BSP trees [Foley et al., 1997]. Object hierarchies are used to partition objects into smaller parts. Typical examples are Bounding Volume Hierarchies [Gottschalk, 2000, Zachmann, 2003, Erleben et al., 2005] and OBB trees [Gottschalk et al., 1996]. In this paper we consider only object hierarchies. Object hierarchies cover geometric objects better than space hierarchies [Erleben, 2002] making object hierarchies desirable for the purpose we consider: Real-time collision detection, that is determining if and where interpenetration between objects occur. Collision detection have applications in simulation, animation, virtual reality, and robotics [Erleben, 2002]. We consider only collision detection for static objects and not for e.g. animated or deformable objects. Thus, it is a reasonable assumption that hierarchy construction can be done off-line, and one-time-only. For this reason it is acceptable that hierarchy construction algorithms spends as much time to build a hierarchy that a trained human artist would spend by doing it by hand.

To make real-time collision detection possible for large, complex objects, different levels of detail are needed. This is exactly what object hierarchies can be used for. Collision detection can then be performed on coarser levels before advancing to levels with finer detail.

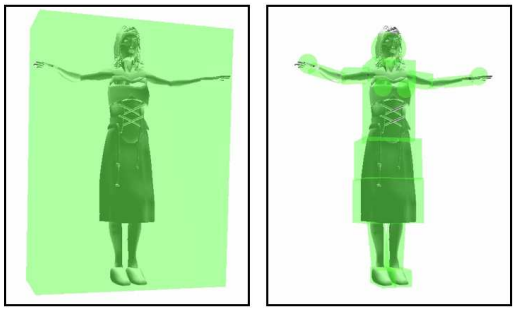


Figure 2: Two-Level method. The coarse level has a single axis aligned box. The fine level has multiple volumes. The fine level is typically made by a human artist while the coarse level is built automatically.

A typical example is the Two-Level method commonly used in the gaming industry. The coarsest level is an axis aligned box, and then there is one finer level of multiple bounding objects made by a human artist. This is shown in Figure 2. The Two-Level method is used e.g. by the Valve Source Engine [Source, 2006]. By using hierarchies with more levels than just two, it becomes difficult and time consuming for artists to design all levels of detail by hand. Tools that can automatically create levels are important and will be a great help to the artist. That is, tools that requires no tuning and tweaking and do not change the work flow for the artist so he only has to create the finest level of detail.

1.1 The Bounding Volume Hierarchy

We consider a specific object hierarchy data structure: The Bounding Volume Hierarchy (BVH). An example of a Bounding Volume Hierarchy is shown in Figure 3, and in Figures 14 and 15. A Bounding Volume (BV) for a set of geometric objects is a closed, typically convex, volume that completely covers the set. A Bounding Volume Hierarchy is a tree data structure containing nodes, storing Bounding Volumes, and edges connecting nodes. Like [Haber et al., 1996] we use the term *primitive* for a leaf node, and *group* for an inner node. Groups always store BVs while primitives can store either BVs or actual geometry.

Formally, a Bounding Volume Hierarchy can be defined as [Zachmann, 2003]:

Definition 1.1. Let $P = \{p_1, p_2, ..., p_n\}$ be a set of n primitives and let e be an integer, $e > 0$ (the branching factor). A Bounding Volume Hierarchy for P is then defined as:

1. If $|P| \leq e$ then we have a single group g with P as its child nodes. The branching factor of g is n.

2. If $|P| > e$ then we have a group g with a set of groups $C = \{c_1, c_2, ..., c_k\}$ as its child nodes. Each child c_i is a group covering a subset $P_i \subset P$, such that the union of all subsets is P.

Definition 1.2. The primitives covered by a node, n, in a BVH, $coverage(n)$, are the leaf nodes that can be reached by traversing all nodes of the sub-hierarchy rooted at n.

A BV of a node in a BVH needs to cover only the BVs of $coverage(n)$. That is, only the BVs of the primitives must be covered. In Figure 3 $coverage(root)$ is

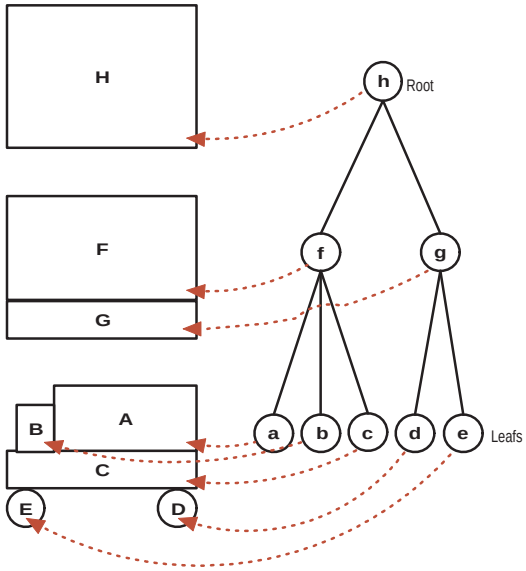


Figure 3: An example of a Bounding Volume Hierarchy with primitives containing Bounding Volumes.

the set of all primitives while $coverage(f)$ is the primitives a, b, and c.

It is not possible to give a single definition of what an optimal hierarchy for collision detection is. Instead, we list some properties that a hierarchy should posses to be considered optimal:

Balancing. Collision detection should be equally fast no matter where on an object an intersection occur. This requires hierarchies to be balanced with respect to nodes. Alternatively a hierarchy can be balanced with respect to volume [Erleben et al., 2005]. In [Somchaipeng, 2005] it is questioned whether balanced hierarchies are always the best choice, and it is suggested that hierarchies should be balanced with respect to density instead. For collision detection, however, balanced hierarchies are desired since collision detection should be equally fast everywhere on objects.

BVs are simple geometric types. It is important that BVs are simple geometric types. It is cheaper to perform collision queries on simple types than on more complex ones making simple types the best choice for real time purposes [Gottschalk, 2000]. It is feasible to use:

- Axis Aligned Bounding Boxes (AABB). These are typically represented as two points min and max in world coordinate space.
- Oriented Bounding Boxes (OBB). These are typically represented as a position and an orientation in world coordinate space, and an extent vector storing the extents in the x, y, and z directions. The extent vector is given in the coordinate frame of the OBB.
- Spheres. These are typically represented as a position in world coordinate space, and a radius.
- Capsules. These are typically represented as a position and a direction vector in world coordinate space, and a radius.

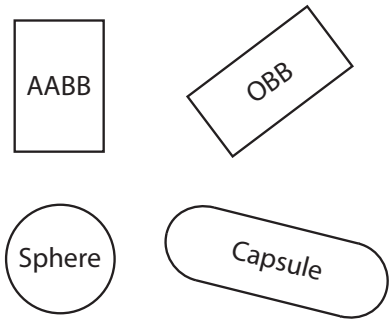


Figure 4: AABB, OBB, sphere and capsule.

Cylinders are not practical for collision detection since intersection tests are too expensive [Eberly, 2000].

Tight fitting Bounding Volumes. Tightness can be measured in many ways. We use a commonly used measure: Tightness by volume [Erleben et al., 2005].

Definition 1.3. Let $C(B)$ be the set of children of a node, B, and let $volume(B)$ be the volume of the BV of B. The tightness, τ can be defined as:

$$\tau = \frac{volume(B)}{\sum_{c \in C(B)} volume(c)} \quad (1)$$

When BVs are tight fitting it becomes possible to reject collisions at an early stage during hierarchy traversal [Erleben et al., 2005] giving hierarchies good pruning capabilities. This is true since a smaller volume implies a smaller chance for collision.

Minimal overlap between BVs of sibling nodes. Covering the same space multiple times in a hierarchy can result in collision queries traversing down two paths simultaneously in the hierarchy. This is not desirable. Reducing overlaps can result in looser fitting hierarchies and thus this property conflicts with having tight fitting BVs [Erleben et al., 2005].

It is also a desirable for hierarchies to be *small and fat*. That is, the height of the hierarchy should be kept low [Klosowski, 1998]. This ensures that traversing the hierarchy from root to leafs can be done in few steps. However, each time a collision is detected for a group, many child nodes will need to be traversed since small, fat hierarchies tend to have nodes with higher branching factors than tall and thin hierarchies.

To determine quality of a hierarchy we use a quality measuring cost function. Such a function is described in [Trifonov et al., 2003] and it is defined as the probability of a collision, volume, multiplied by the number of child nodes.

Definition 1.4. Let H be a BVH, and let $C(n)$ be the set of children of a node, n. Then a quality measure function, *cost*, can be defined as:

$$cost(H) = \sum_{n \in H} (C(n) \cdot volume(n)) \qquad (2)$$

This cost can be kept small by minimizing volume while balancing the hierarchy. We can show this by an example. Assume we have 10 child nodes and there are two possible parent nodes:

- A single node with a BV of volume 50.
- Two nodes with BVs of volumes 30 each.

The cost of using the single node is $10 \cdot 50 = 500$ while the cost of using two nodes are $5 \cdot 30 + 5 \cdot 30 = 300$. Thus, the best cost is achieved by using two nodes and avoid nodes with too many children.

Equation 2 is simple and it encapsulates two of the important properties; tightness and balancing. The function is very cheap to compute since it requires only information local to each node; the volume of the BV and the number of children. Including overlapping in the equation would require non-local information and the equation would be more expensive to compute.

2 Existing algorithms

There exists algorithms for automatic construction of Bounding Volume Hierarchies. They have some important differences:

Construction strategy. Top-down construction algorithms typically work by fitting a BV around a set of primitives, and this set is then split into two or more subsets using a splitting plane. The process is then repeated for subsets until a stopping criterion is reached. This could be when subsets contain only a single primitive.

Bottom-up construction algorithms typically work by merging a set of primitives into groups. The merging is performed again until there is only one group, which becomes the root of the hierarchy. Merging nodes is done by creating a new group, g, making it parent of the merging nodes, and fitting a BV around $coverage(g)$.

Incremental insertion algorithms inserts primitives one at a time into a hierarchy. New primitives are placed at the *cheapest* position in the hierarchy, according to a cost function.

Branching factor of nodes. Most construction algorithms use a branching factor of 2 for all groups, which makes a binary hierarchy. In some situations it may be better to build hierarchies with nodes having varying branching factors. An example shown in Figure 5, where a binary hierarchy introduce an extra level resulting in a higher cost compared to a hierarchy with varying branching factors.

Bounding Volume types. Most algorithms use only a single type of BV. An example is the OBB tree where only the Oriented Bounding Box type is used [Gottschalk et al., 1996]. Tightness can be gained by choosing from a set of BVs instead of using a single type in the entire hierarchy. BVs should be kept simple, though, so fast collision tests are possible.

We adopt the term Approximating Hybrid Bounding Volume Hierarchy (HBVH) from [Erleben, 2002] for a hierarchy having varying branching factors, multiple geometric types, and approximating BVs instead of actual geometry in leaf nodes. In the following sections we describe algorithms that constructs, or could easily be made to construct, Approximating Bounding Volume Hierarchies. Running times

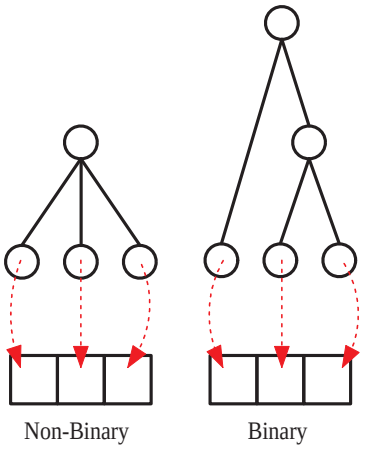


Figure 5: Example showing that binary hierarchies are not always the best choice. It is assumed that leaf BVs have volume equal to $1/3$, and that the cost function from Equation 2 is used. The left hierarchy, with varying branching factor, has two levels; a single node with all primitives as child nodes. The right hierarchy, with fixed branching factor 2, has three levels; two nodes with two child nodes. The cost of the left hierarchy is $1 \cdot 3 = 3$, while the cost of the left hierarchy is $2 \cdot 1 + \frac{2}{3} \cdot 2 \simeq 3.3$. Thus, the right hierarchy has the highest cost.

for the different types of algorithm are deduced in [Erleben et al., 2005]. The expected running time for the algorithms is $O(n \lg^2 n)$ where n is the number of primitives. This is true when the simple geometric types described in Section 1.1 are used, and only points on the convex hulls are used to fit BVs.

2.1 Incremental insertion

An incremental insertion algorithm is presented in [Goldsmith et al., 1987]. It does not explicitly mention the use of a BVH, but we describe it assuming that it is in fact a BVH that is being constructed. A cost function is used to control the insertion of primitives into the hierarchy. It estimates the cost of inserting primitives at different places in the hierarchy, and the algorithm then picks the cheapest place.

When a primitive p is inserted into a partially created hierarchy, the algorithm uses 3 rules:

1. p can be made a child of a group g.
2. p can be combined with another primitive p' of a group g to create a new group g' which becomes a child of g.
3. p can be inserted into a child group g' of a group g recursively. That is, the partially created hierarchy is traversed until reaching case 1 or 2.

The 3 rules are shown in Figure 6.

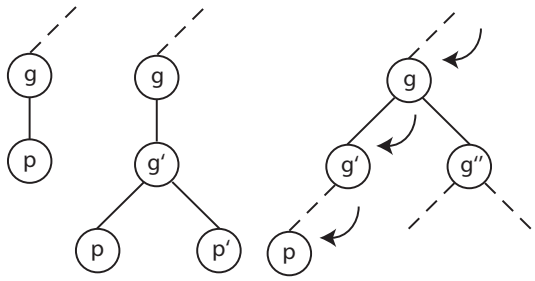


Figure 6: The 3 rules. Rule 1 (left) is making a primitive p a child of a group g. Rule 2 (middle) is combining primitives to create a new group. Rule 3 (right) is recursive insertion into groups.

This algorithm can be used to construct Approximating Hybrid Bounding Volume Hierarchies but it has some disadvantages. The constructed hierarchy is dependent on the insertion order of nodes. This is undesirable. It is not intuitive to human artists that insertion order of primitives affects how the hierarchy turns out. Also, the same set of primitives can become everything from a well balanced hierarchy, to a totally unbalanced hierarchy in the worst case. Another disadvantage is that new groups always cover only two primitives (case 2). As discussed in Section 2, this it not always the best choice.

The disadvantages are addressed in the improved incremental insertion algorithm presented in [Haber et al., 1996]. Specifically, more complex insertion rules are used, and a global optimization step is performed when the hierarchy has been constructed. The global optimization step performs regrouping of nodes in the hierarchy. Two different approaches are used. The first one, *Successive Re-Insertion*, removes nodes from hierarchy that have been placed unfortunately at insertion time, and re-inserts them into the hierarchy. The second one, *Elimination of Ill-formed Groups*, searches for ill formed groups and attempts to split them. Such groups are searched for by using a set of *badness criteria*. Details can be found in [Haber et al., 1996].

The new rules and the global optimization step should reduce the dependency of insertion order, at the cost of extra computation time. However, only the con-

stant factors get higher.

2.2 Splitting

A splitting algorithm is presented in [Müller et al., 1999]. The algorithm recursively partitions a set of primitives into two disjoint subsets until a stopping criterion is reached. Entire objects are assigned to subsets. No BVs are cut into smaller pieces. The algorithm proceeds as follows: The root of the hierarchy is created by sorting all primitives along all major axes using the center of each primitive as key. Then, a cost function is used to consider all possible partition points, and the one with the lowest cost is used. This resulting in two disjoint subsets containing all primitives. The algorithm then proceeds by recursively sorting and splitting subsets until subsets containing only single primitives are reached. This is shown in Figure 7.

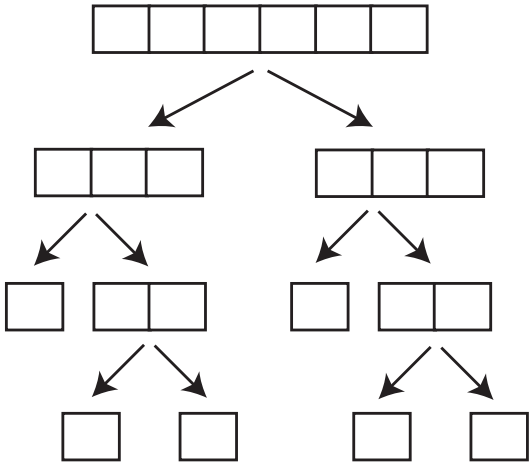


Figure 7: Building a hierarchy by partitioning objects along one of the three axes at a minimum cost point. Primitives belong entirely to one of the two subsets occurring from partitioning. No cutting is performed.

The major disadvantage of this algorithm is that it builds only binary hierarchies which makes it unable to bouild Approximating Hybrid Bounding Volume Hierarchies. This could, however, be changed by considering multiple splittings on each level. Whether the hierarchy is balanced or not depends on the cost function used.

A related algorithm is described in [Gottschalk et al., 1996]. It constructs an OBB-Tree (a tree of Oriented Bounding Boxes). Here, a bounding box is fitted around all primitives, and then a splitting plane orthogonal to the longest axis of the OBB is determined. A splitting point is needed, and here the point along the longest axis, which is the mean point of all vertices, is used. Objects are partitioned according to which side of the splitting plane their center point lies, and again primitives are never cut. An example of splitting along the longest axis of an OBB is shown in Figure 8.

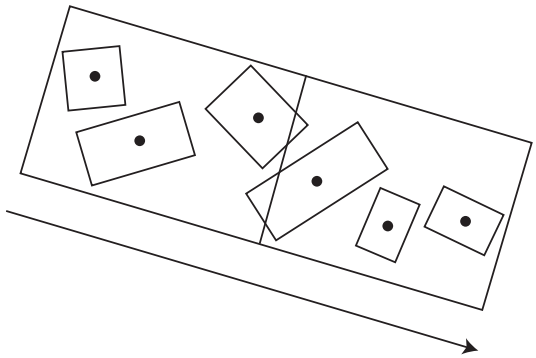


Figure 8: Splting along the longest axis of a fitted OBB using center points.

This algorithm shares disadvantages with the former splitting algorithm. However, it can not be used to build Approximating Hybrid Bounding Volume Hierarchies since it uses only Oriented Bounding Boxes. These are needed to find the splitting axis and thus the splitting plane.

2.3 Merging

A merging algorithm is described in [Erleben et al., 2005] and an implementation of this algorithm can be found in [OpenTissue, 2006]. The algorithm initially builds graph data structure. Nodes[1] in the graph correspond to primitives and edges correspond to neighboring relations. An edge in the graph means that two BVH-nodes are good candidates for being merged. This is determined by a heuristic function which enlarges primitive BVs and notices collisions: A collision means that an edge between two colliding graph-nodes must be added to the graph.

When the graph has been built, it is used to build a hierarchy. This is done by repeatedly collapsing edges until only a single graph-node exists. An edge collapse operation is shown in Figure 9.

[1]To avoid confusion in this section, we denote nodes in the graph *graph-nodes*, and nodes in a BVH *BVH-nodes*.

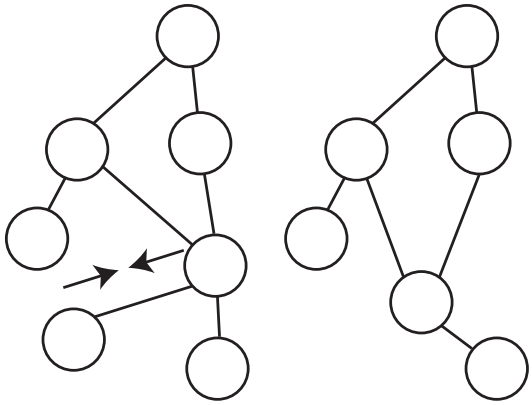


Figure 9: Edge collapsing. An edge is collapsed into a single node.

After an edge collapse in the graph, BVH-nodes are merged into a new group in the BVH only if one of two requirements are fulfilled:

- The graph-node already covers a higher number of graph-nodes than a fixed branching factor.
- There are fewer edges left in the graph than a fixed branching factor.

If none of the requirements are fulfilled the merge is postponed, and the new graph-node is set to cover the graph-nodes of the collapsed edge. The graph-node is considered when it is involved in a collapse operation again.

In order to build a good hierarchy, edges in the graph must be assigned a *priority* such that the edge with the best (smallest) priority is collapsed first. This priority can be assigned by a cost function. The current implementation uses the function shown in Definition 2.1.

Definition 2.1. Let E be an edge and $N(E)$ be the two nodes of the edge. Also, let $volumes(E)$ be the volume of the BV fitted to cover the BVs of $N(E)$, and $volume(n)$ be the volume of the BV of a node n. The priority of an edge can then be defined as:

$$priority(E) = volumes(E) - \sum_{n \in N(E)} volume(n) \quad (3)$$

This algorithm can be used to construct Approximating Bounding Volume Hierarchies. The major disadvantage of the algorithm is that hierarchies become tall and thin since nodes are not required to merge with other nodes on each level. This also causes hierarchies to become quite unbalanced, since some primitives can be connected directly to the root node while others reside deeper in the hierarchy. Finally the fixed enlargement, 25%, of primitive BVs when building the graph does not always ensure that enough edges are created for the algorithm to finish. This could be fixed by repeating enlargement until each primitive collides with at least one other primitive.

3 A Branch and Bound algorithm

We attempt to solve hierarchy construction as an optimization problem. A naive approach is to enumerate all possible hierarchies, use a cost function, and search for the best. This would give the globally optimal hierarchy according to the cost function. It is, however, not practically possible to use this method. As shown in [Trifonov et al., 2003] the number of hierarchies grow exponentially in the number of primitives used. Thus, for a very small number of primitives, the number of hierarchies becomes too great for the problem to be solvable. Instead of solving the problem of finding the globally optimal hierarchy we simplify and solve multiple, local optimization problems; one problem per level in the hierarchy. Thus, the globally optimal hierarchy is not necessarily found, but it should be possible find a hierarchy having many of the desirable properties.

Since we assume that algorithms work on primitives created by artists we find it natural to think of hierarchy construction as a bottom-up action. Thus, we start with a set of primitives in the lowest level in the hierarchy, and merge these into groups.

An optimal *brute-force* merging approach considers all possible sets of nodes merging into groups, we denote these sets *merging candidates*, and picks the optimal combination of merging candidates covering each merging node exactly once. This, however, is not possible in practice since there are too many merging candidates[2]. Instead of considering all possible merging candidates we use a heuristic function to find *good* ones.

We have designed two different heuristic functions. The first one begins with a single node and then re-

[2]The number of ways to partition n elements into nonempty subsets is called the Bell Number [MathWorld, 2006]. The growth of this number starting from 1 is: $1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, ...$

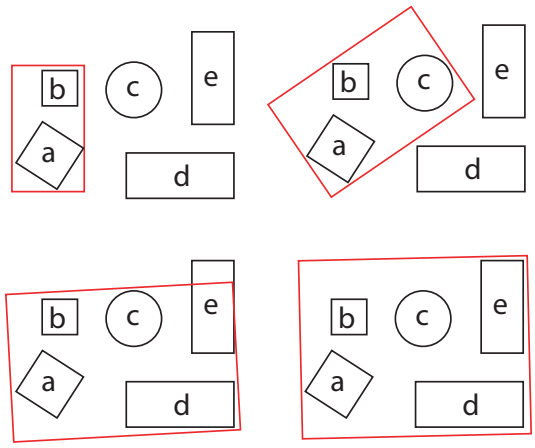


Figure 10: The first heuristic function used to select merging candidates. Groups having only single child nodes are not accepted, so a node is required to merge with at least one other node. The tightest volume can be fitted around a and b so the merging candidate $\{a, b\}$ is saved. Then, the tightest volume can be fitted around a, b, and c so the merging candidate $\{a, b, c\}$ is saved. The heuristic function continues and saves merging candidates $\{a, b, c, d\}$ and $\{a, b, c, d, e\}$. Then this process is repeated for all other nodes.

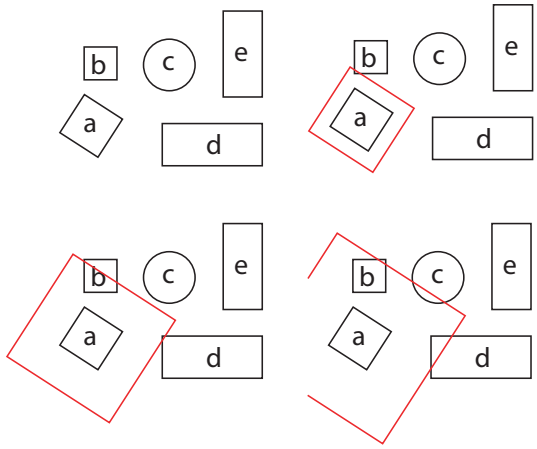


Figure 11: The second heuristic function used to select merging candidates. Groups having only single child nodes are not accepted, so a node is required to merge with at least one other node. The BV of node a is repeatedly enlarged until it has a given number of collisions (in this case 3). Then all possible merging candidates between a and the colliding nodes are generated. Initially, a does not have 3 collisions so it is enlarged until it has a collision with b. It needs more collisions so it is enlarged again until 3 collisions occur. This happens when a collides with b, c, and d.

peatedly searches for sibling nodes that it can merge with. To find the best sibling to merge with, a BV is fitted around the BVs of the nodes, and the sibling that gives the best (smallest) tightness is used. The tightness measure defined in Equation 1 is used. The merging candidate is saved, and then the process is repeated. Figure 10 shows how the heuristic function works. We require nodes to merge with at least one other node on each level to ensure that the algorithm builds small, fat hierarchies. Thus, the function finds at most $n^2/2 - n$ merging candidates.

The second heuristic function is quite similar to the one used by the OpenTissue algorithm described in Section 2.3. That is, it enlarges primitive BVs, but unlike the OpenTissue algorithm BVs are enlarged repeatedly with 5%, one at a time, until a minimum number of collisions are detected. This number is set to 1 since we require each node to merge with at least one other node. To keep the number of merging candidates small enough for the construction problem to be solvable, the heuristic function has a limit on the number of nodes allowed to be covered by a merging candidate. This is currently $\lg(n)$ where n is the number of nodes on the level being merged. Figure 10 shows how the heuristic function works on an BV where enlargement continues until 3 collisions are detected. When the required number of collision has been detected, all possible combinations of merging candidates for that particular collision are saved. The function finds at most $\sum_{i=2}^{lg(n)} \binom{n}{i}$ merging candidates, where n is the number of nodes on the level being merged.

The heuristic functions create a set, A, of n merging candidates for m nodes on a merging level. We denote a combination of merging candidates from A that cover each of the m merging nodes exactly once a *valid solution*. The valid solution with the lowest cost is the *best valid solution*. By representing merging candidates as rows in a table, and using 1 to denote that a node is involved in a merge, and 0 to denote that it is not (see example in Table 1) the problem of finding this best valid solution can be written as a *crew scheduling* or *set partition* problem [Pardalos, 1998]:

$$\text{minimize} \quad \sum_{i=1}^{n} c_i x_i \tag{4}$$

$$\text{subject to} \quad \sum_{i=1}^{n} a_{ij} x_i = 1 \text{ for } j = 1, ..., m \tag{5}$$

$$x_i \in \{0, 1\} \tag{6}$$

where c is given by a cost function, $c_i = \text{cost}(a_{ij})$,

and a_i is a merging candidate. Equation 4 can be solved by searching through all combinations of merging candidates and evaluating cost for valid solutions. This approach is to slow for even small problem sizes. Instead, we use a Branch and Bound algorithm [Pisinger, 2004]. It branches through all combinations merging candidates, found with the heuristic function, and uses a bounding function to determine when branching is unnecessary. We denote the Branch and Bound algorithm that uses the first heuristic function B&B-Tightest, and the Branch and Bound algorithm that uses the second heuristic function B&B-Enlarge. The best valid solution found by the Branch and Bound algorithm becomes groups on a new level, and the algorithm can be repeated:

- Use a heuristic function to find merging candidates.
- Use the Branch and Bound algorithm to find the best valid solution.

The algorithm stops when a level with only one node, the root node, is reached

The cost function used is similar to the one defined in Equation 2. That is, the cost of selecting a merging candidate is the volume of the BV fitted around the BVs of the merging nodes multiplied by the number of merging nodes. This should give tight fitting BVs in the hierarchy while trying to balance each level.

The bounding function used estimates the cost of including all remaining nodes, not covered by the combination of merging candidates currently being considered. The estimate must be a lower bound. The tightest BV that can be fitted around a set of BVs has a volume equal to the sum of the volumes of the BVs. Further, since each node is required to merge with at least one other node, the smallest number of child nodes a group can have is 2. Thus a lower bound, ℓ, on including the remaining nodes to get a valid solution is:

$$\ell = 2 \cdot \sum_{n \in N} volume(n) \tag{7}$$

where N is the set of non-covered nodes, and $volume(n)$ is the volume of the BV of node n.

It can be shown how the complete algorithm works by giving an example. Table 1 shows a few merging candidates[3] found with a heuristic function, and the costs of selecting each candidate. In Table 1 $c_i = min(c_{obb,i}, c_{sphere,i})$ since there are two BV types. How the algorithm proceeds to find the best valid solution is shown in Figure 12.

Merging candidate	OBB cost	Sphere cost
a_{ij}	$c_{OBB,i}$	$c_{sphere,i}$
0 0 1	9	10
0 1 0	4	6
1 0 1	15	13
1 1 0	16	21

Table 1: A table with merging candidates for 3 nodes. The left column shows 4 merging candidates represented as binary numbers; 110 means that the second and third nodes are to be merged. The middle column shows the cost of merging into an OBB (fitting an OBB around the merging nodes) and the right column shows costs of merging into a sphere. The costs are determined by the cost function shown in Equation 2.

The Branch and Bound algorithm is independent of node order, so for a set of primitives the same hierarchy will be created. Also, the algorithm gives tight fitting Bounding Volumes while trying to avoid large volumes with too many child nodes. Branching factors of nodes varies in the hierarchies and the algorithm supports multiple BV types.

The major drawback of the algorithm is the worst case run-time which is exponential in the number of primitives. Also, overlapping Bounding Volumes are not penalized. The same space is paid for twice but it would probably be a good idea to penalize overlapping even more. A last drawback is that the algorithm is heavily dependent on the heuristic function used to select merging candidates for the Branch and Bound algorithm. If this function only finds poor merging candidates, the Branch and Bound algorithm can do nothing to improve the result. Also, if the heuristic function finds only merging candidates that are not disjoint, the Branch and Bound algorithm is not capable of building a hierarchy.

The complete algorithm described above is shown in Algorithm 1, 2, and 3 in Appendix A.

[3]Notice that the table contains merging candidates containing single nodes which is not allowed by our heuristic functions. This is done to keep the example using the table small.

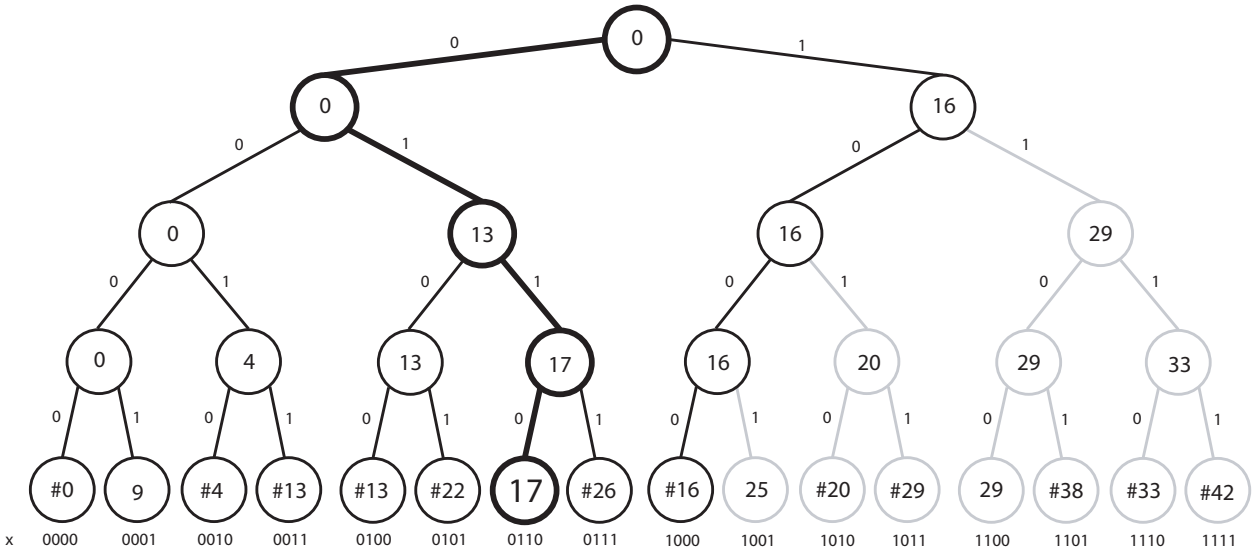


Figure 12: A complete Branch and Bound search tree using the merging candidates from Table 1. The costs of the chosen merging candidates are shown as numbers inside nodes. Solutions marked with # are invalid since they do not cover all merging nodes. The binary selector variable, x, is used to choose merging candidates. The number $x = 0110$ is a valid solution using merging candidates 2 and 3 with cost 17 (the optimal solution). The number $x = 0111$ is an invalid solution using merging candidates 1, 2, and 3 with cost 26. The right part of the hierarchy is grayed out since it is not necessarily searched, depending on the bounding function.

4 Overview of the implementation

In order to experiment with the Branch and Bound algorithm we have implemented the Heterogeneous Bounding Volume Hierarchy data structure. To ease exchange of hierarchies between different applications, we have developed a hierarchy serialization format in XML that is easy to write parsers for, and easy to read and understand for human artists. Our entire implementation is done in C++. We have used OpenTisue [OpenTissue, 2006] as a framework and tinyXML [tinyXML, 2006] for hierarchy serialization.

4.1 Approximating Hybrid Bounding Volume Hierarchy data structure

The Approximating Hybrid Bounding Volume Hierarchy data structure is implemented as a class storing a root node. Nodes in the hierarchy contain a parent pointer, a BV pointer and a list with child nodes. The root node has an empty parent pointer. Using lists with child nodes it becomes easy to visualize entire levels, one at a time, by traversing the hierarchy starting from the root node, and display all child lists at a certain depth/level.

4.2 Construction algorithm

The Branch and Bound algorithm is implemented almost exactly as shown in the Pseudocode in Appendix A. The *binary selector variable*, x, is stored as a 32bit integer, so the current implementation supports up to 32 primitives. This could be changed by using a variable size bit-vector instead.

To fit BVs the algorithm uses functions available from OpenTissue. The current implementation supports only AABBs, OBBs and spheres. AABBs are actually represented as special-case OBBs with orientation aligned with the world coordinate system. AABBs and spheres are straight forward to fit and OBB-fitting is performed by the covariance method [Gottschalk et al., 1996, Erleben et al., 2005].

4.3 Serialization of hierarchies

It is important that a serialization format is easy to read and understand such that it is possible for human artists to create hierarchies by hand for debugging purposes. Also, it should be easy to write parsers that read and write from and to the format. There is an overhead in using a textual format, but XML is is a widespread and commonly known format, and

there exists many tools that can aid in the work of writing parsers. We found this to be more important than keeping files storing hierarchies small.

A hierarchy is serialized with a `hierarchy` tag.

```
<hierarchy> ... </hierarchy>
```

Each volume is serialized with:

```
<volume type="" position="" {type specific arguments} />
```

The specific type argument can be any volume type supported. For now it should be `obb`[4] or `sphere`. Arguments can contain more than one component like position, which contains an x,y and z component. These are separated by a blank character. The position then becomes `position="x y z"`.

If a volume has any children, these are contained inside the volume tag of its parent:

```
<volume type="" position="" {type specific arguments} >
  <volume type="" position="" {type specific arguments} />
  <volume type="" position="" {type specific arguments} />
</volume />
```

This makes it easy to read hierarchies for human designers. Alternatively, indices could have been used and then relations could be shown, by referring to these, but it would make the format harder to read.

The XML serialization format is used as both input to and output from the construction algorithm. We have developed a small exporter for Maya 7.0 that writes BVs to the XML format. The exporter is written entirely in MEL[5] and works by letting the artist select which volumes he wants to export - with the mouse or with the console interpreter - and these volumes will then be written as a set of primitives when the exporter is run. Thus, the artist can keep both actual geometry and bounding volumes in the same scene and easily export exactly what he wants.

A full example of a serialized Approximating Hybrid Bounding Volume Hierarchy is shown in Appendix B.

5 Evaluation of the Branch and Bound algorithm

In this section we evaluate both versions of the Branch and Bound algorithm, B&B-Tightest and B&B-Enlarge, and compare them to the merging algorithm from OpenTissue described in Section 2.3 and to the Two-Level method described in Section 1.

To evaluate hierarchies, we have used four different objects: Cannon, chair, helicopter, and Tie-Interceptor. They are chosen to show how the algorithms works in different situations:

- All objects contain both boxes and spheres.
- Cannon and helicopter contain many axis aligned primitives.
- Tie-Interceptor and chair are symmetric.

The objects are shown in Figure 13.

We begin this section by briefly going through constructed hierarchies for two of the objects; chair and helicopter. They show some interesting results.

The chair hierarchy constructed by the OpenTissue algorithm, is shown in appdendix C.4. It is slightly unbalanced and contains overlaps, some large ones on the third level. The chair hierarchy constructed by the B&B-Tightest algorithm is shown in Appdendix C.5. It is well balanced, but has minor overlaps. The chair hierarchy constructed with the B&B-Enlarge algorithm is shown in Appdendix C.6. This hierarchy is also well balanced, and it contains hardly any overlaps. Here, it is obvious that only primitives are covered by Bounding Volumes on all levels. This can be seen on levels 2 and 3.

The helicopter hierarchy created by the OpenTissue algorithm is shown in Appendix C.1. The hierarchy is quite unbalanced, and the hierarchy is tall since only a few merges are performed at each level. The hierarchy has noticeable overlaps. The helicopter hierarchy constructed by the B&B-Tightest algorithm is shown in Appendix C.2. It is very unbalanced and really shows that the heuristic function used to find merging candidates does not always find good ones. Almost the entire helicopter - the cockpit, the body, and the tail - is merged into a single group. This is not acceptable. The helicopter hierarchy constructed by the B&B-Enlarge algorithm is shown in Appendix C.3. This is much more well balanced than the two other helicopter hierarchies. A disadvantage of forcing nodes to merge with at least one other node on each level is clearly visible in this hierarchy. The merge that the rotor blade performs from the second to the third level is neither obvious nor optimal.

[4] AABBs are OBBs with no orientation

[5] Mel is the script language used by Maya.

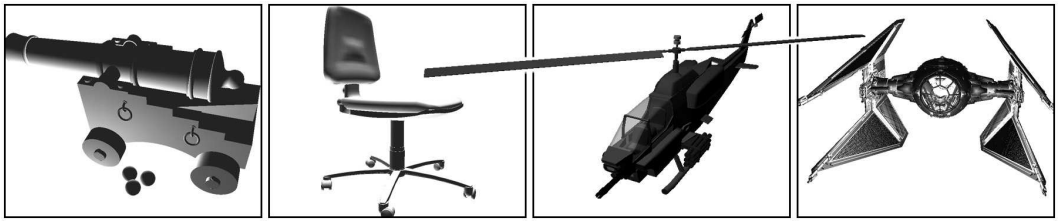


Figure 13: The four objects used to evaluate the hierarchy construction algorithms.

The two remaining objects, Tie-Interceptor and Cannon, are shown in Appendix C.7, C.8, C.9, C.10, C.11, and C.12.

5.1 Construction performance

A Pentium 4 2.8 Ghz (HT) with 1 GB RAM was used to measure construction performance. Times for the Branch and Bound and the OpenTissue algorithms are shown in Table 2.

The most remarkable item to notice in Table 2 is Branch and Bound construction of a hierarchy for the Tie-Interceptor object. This takes more than 7 minutes for B&B-Tightest and more than 3 hours for B&B-Enlarge. The time used by B&B-Tightest is acceptable while the time used by B&B-Enlarge is not, since a trained artist can create a good hierarchy in less than 3 hours. As expected, the OpenTissue algorithm is much faster than both Branch and Bound algorithms.

It may seem surprising that the construction time of the Branch and Bound algorithm does not necessarily grow in the number of primitives. In the worst case it does, and the growth is exponential. However, construction time depends on the merging candidates found by one of the heuristic functions and on how early a good solution is found by the Branch and Bound algorithm. When a good solution is found, many branches can be avoided.

5.2 Hierarchy quality

Hierarchy quality has been measured using the cost function defined in Equation 2. Results are shown in Table 3.

The hierarchies constructed with the Branch and Bound algorithms have a significantly lower cost compared to hierarchies for the same object constructed with the OpenTissue algorithm. The B&B-Enlarge algorithm constructs the hierarchies with the lowest cost, but as shown in Section 5.1 at much higher construction times.

5.3 Collision detection performance

To determine whether it is really the right choice to use hierarchies for collision detection we have compared collision detection performance of the Branch and Bound algorithms to the Two-Level method. Performance of the hierarchies constructed with the OpenTissue have been left out since there is no exporter for these hierarchies to the XML serialization format.

Performance has been evaluated by using a small, simple simulator that does a simple velocity vector projection in case of collisions. The simulator was run on a Barton 2500+ with 1.5 GB RAM. The test scenes used had two of the same objects colliding with each other, and average time spent on the first 1000 collisions was measured. Only tests that showed collisions for the root Bounding Volumes were measured to make sure that actual hierarchy traversal was performed. The traversal method used was the Tandem Traversal which is described in [Erleben et al., 2005]. For intersection tests between geometric types methods from [Gomez, 1999] were used. Results are shown in Table 4.

The hierarchies constructed with the Branch and Bound algorithms perform better than the Two-Level method even though hierarchies do not have too many nodes. Each time the root level Bounding Volumes intersect, all primitives must be tested against each other. For an object with k primitives and another object with n primitives, this is $k \cdot n$ tests. This changes in some cases when objects are totally colliding. When using hierarchies, all groups must be tested before advancing to primitive testing. The Two-Level method does primitive testing immediately after testing the root volumes, so if testing an object with k primitives against an object with n primitives, there

Object	BVs	B&B-Tightest Time	MC	B&B-Enlarge Time	MC	OpenTissue Time
Chair	13	3.7 sec	57	4.4 sec	121	0.79 sec
Helicopter	15	1.8 sec	84	14.5 sec	104	0.90 sec
Cannon	17	4.6 sec	77	2.5 sec	77	2.61 sec
Tie-Interceptor	27	7.3 min	133	3.4 hours	150	2.51 sec

Table 2: Hierarchy construction times for Branch and Bound algorithms, the OpenTissue algorithm, and the Two-Level method. Average over 1000 samples with two of the same objects colliding with each other. MC is the number of merging candidates.

Figure	B&B-Tightest cost	B&B-Enlarge cost	OpenTissue cost	Two-Level cost
Chair	3'466	1'650	3'796	7'150
Helicopter	119'804	73'360	132'448	3'668'010
Cannon	6'739'000	1'973'310	7'361'000	9'866'565
Tie-Interceptor	6'293'130	2'935'520	11'256'200	26'419'662

Table 3: Quality of hierarchies constructed with the Branch and Bound algorithms, the OpenTissue algorithm, and the Two-Level method. Quality is determined by using the cost function shown in Equation 2.

Object	B&B-Enlarge	B&B-Tightest	Two-Level	Two-Level (total)
Cannon	0.769347	0.818668	2.76107	0.857550
Chair	0.815947	0.842466	1.07503	0.537450
Helicopter	0.720436	0.720436	3.79308	0.646449
Interceptor	1.518710	1.498680	6.78322	1.656150

Table 4: Collision detection times (in milliseconds) for Branch and Bound algorithms, the OpenTissue algorithm, and the Two-Level method. Average over 1000 samples with two of the same objects colliding with each other. The rightmost column shows times for the Two-Level method when objects are totally colliding (placed on each other).

are at most n tests before the first collision is found. Collision detection times for totally intersection objects are shown in the rightmost column of Table 4.

5.4 Discussion of the Branch and Bound algorithms

The hierarchies constructed by the B&B-Enlarge algorithm are the best ones. The hierarchies are a little taller than the hierarchies constructed by the B&B-Tightest algorithm, but the total cost is smallest and collision detection fastest. Also, it is apparent that is a good idea to use hierarchies for collision detection, even when objects are not covered by too many Bounding Volumes.

6 Ideas for future algorithms

The Branch and Bound algorithms construct good hierarchies, however, there is still room for many improvements. Also, during our work with designing the Branch and Bound algorithms we have considered alternative approaches to hierarchy construction.

6.1 Improving the Branch and Bound algorithm

The Branch and Bound algorithm could be improved by introducing a heuristic function giving an initial cost guess. By using a tight initial guess, the algorithm can reject solutions that are not optimal much earlier. To reduce run-time even further we might be able to use *delayed column generation* which is a method for solving large linear optimization problems. It has already been applied to *crew scheduling* problems [Borndörfer, 2006], so it might be possible to use this method for hierarchy construction as well.

To improve the constructed hierarchies the algorithm could consider complexity of the BV types used. For time-critical applications simpler BVs with faster collision tests could be preferable over more complex types with more expensive tests, so in this situation the cost could be scaled. Another improvement could be to have BV types with different complexity at different levels in the hierarchy. One could argue that more tight fitting, complex BVs should be allowed on higher levels where there are not too many nodes, to give the hierarchy better pruning capabilities. But it could also be the case that more complex types were required for primitives, at the lowest level in the hierarchy, so artists could create real tight fitting BVs. These extension could be made without changing the Branch and Bound algorithm; only new fitting code and new geometric types would need to be added.

6.2 Alternative construction approaches

A problem related to Bounding Volume Hierarchy construction is construction of Phylogenetic Trees [Hall, 2004]. Here, child nodes share characteristics inherited from a common ancestor. This is quite similar to object hierarchies where primitives are "inherited" down through the hierarchy to the leaf nodes. It is possible that algorithms for Phylogenetic Tree construction could be used for object hierarchy construction or could provide new insight.

Another idea might be to view hierarchy construction as a Steiner tree problem [Promel et al., 2002]. A Steiner tree problem is a geometric minimization problem; given a set of vertices, the shortest tree connecting all vertices must be found. In contrast to the problem of finding the minimum spanning tree, additional points may by inserted to minimize the length of the tree in the Steiner tree problem.

Using primitives as initial vertices, the new, inserted vertices are groups, that is, internal nodes in the corresponding Bounding Volume Hierarchy. The metric, however, is by no means as simple as for the Steiner tree problem since a decision to connect to vertices by an edge cannot be made only from the position of the vertices. For Bounding Volume Hierarchies, it is rarely the best choice for two sibling nodes to merge into a group based on say only the center position of their BVs. What we have is a more general Metric Steiner tree problem. Thus, in order to solve the Bounding Volume Hierarchy problem precise metrics must be developed that encapsulates the desired properties.

7 Conclusion

In this paper we have described algorithms for the construction of optimal Bounding Volume Hierarchy. There are some basic properties that a hierarchy must posses to be considered optimal:

1. It must be balanced.
2. It must have simple geometric types.
3. It must have tight fitting Bounding Volumes.
4. It should seek to minimize overlaps between Bounding Volumes of sibling nodes.

It is a very hard problem to find optimal or near-optimal hierarchies so all of the algorithms we have described use some heuristic to make hierarchy construction computationally possible.

One of the main goals was to develop an algorithm that constructs good hierarchies without changing work flow for the artists. Today, artists typically use a Two-Level method: They create the finest level, the primitives, in the hierarchy, and then a single BV covering this level is automatically created. We have proposed a, to our best knowledge, new approach to hierarchy construction. It works bottom up by merging a level of primitives into multiple, coarser levels. We have developed two different heuristic function that find merging candidates, and thus we have two versions of the algorithm, B&B-Tightest and B&B-Enlarge. Using the algorithm, artists still only have to create the fines level in hierarchies, but they get full hierarchies. Merging is done by using a heuristic function to find good merging candidates and then an optimization problem is solved to find the best selection of merging candidates covering each merging node exactly once.

The Branch and Bound algorithms attempt to create hierarcies having the first three of the four desirable properties, and evaluations show that they are successful in most cases. Evaluations also show that the B&B-Enlarge algorithm constructs the best hierarchies and that it is a good idea to use full hierarchies instead of the simple Two-Level method for collision detection, at least unless objects are totally colliding (placed on each other). Also, both B&B-Tightest and B&B-Enlarge constructs better hierarchies than the algorithm available in OpenTissue, but at much higher construction times.

Before the Branch and Bound algorithms can be used, e.g. in the gaming industry, more research is needed. The hierarchies constructed are acceptable, however the worst case construction times are not. Possible improvements are to provide a good initial guess for the Branch and Bound algorithm or to use different, more advanced techniques to solve the optimization problems.

8 Acknowledgements

First of all, we want to thank Assistant Professor Kenny Erleben. His ideas and his work in the area [Erleben, 2002, Erleben et al., 2005] inspired us to solve hierarchy construction as an optimization problem. Also, we want to thank Professor David Pisinger for taking time, listening to our ideas and involving us in how to solve optimization problems.

8.1 3D Models

We have used models from Simply Maya (http://www.simplymaya.com/, Star Wars - The 3D Model Alliance (http://www.surfthe.net/swma/), and 3D Cafe (http://www.3dcafe.com/). We owe great thanks to these artists. We used the models solely for educational purposes. All rights belongs to the original creators of the models and they may not be used commercially without expressed permission.

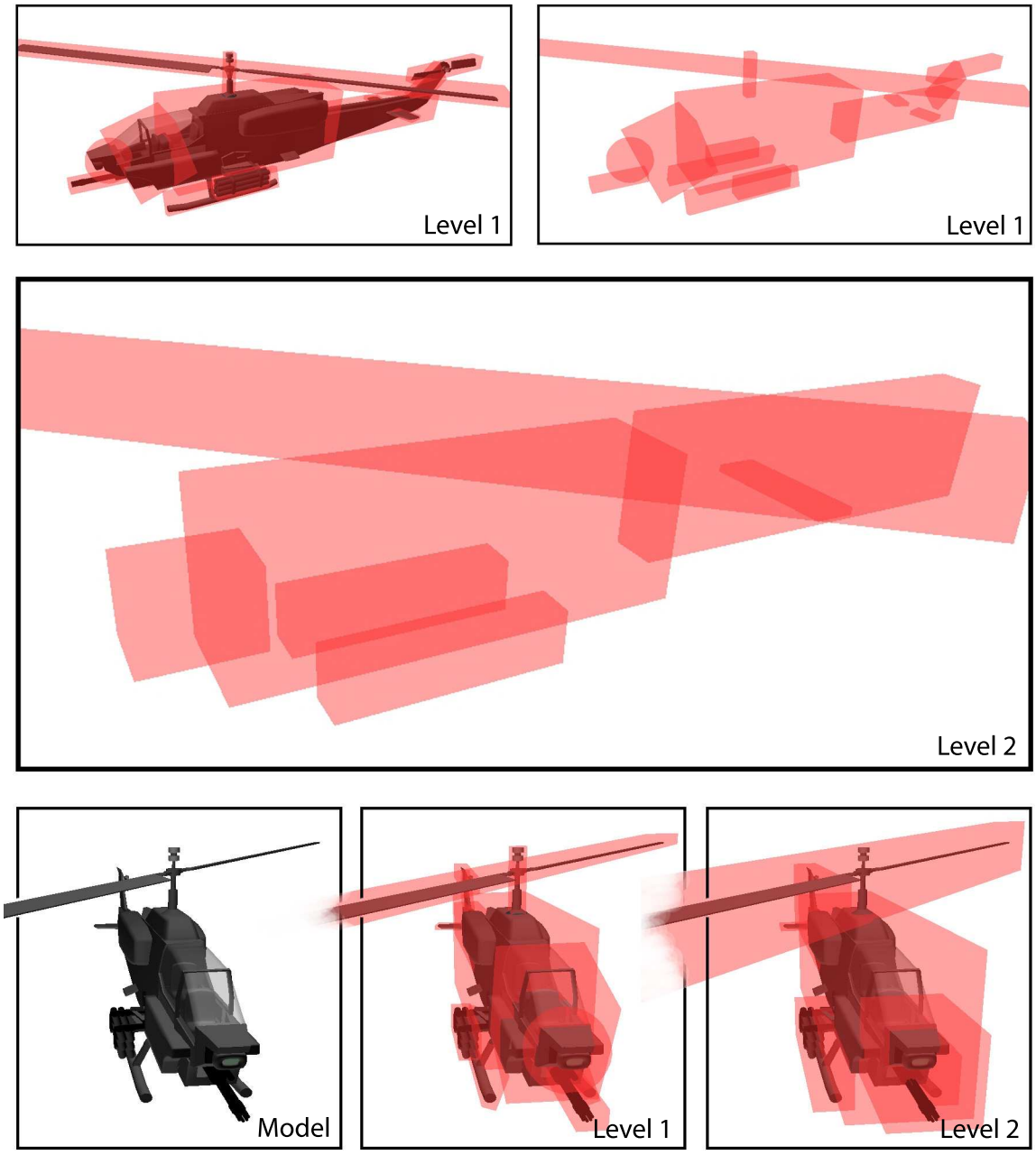


Figure 14: A helicopter, shown on different levels, with Bounding Volumes.

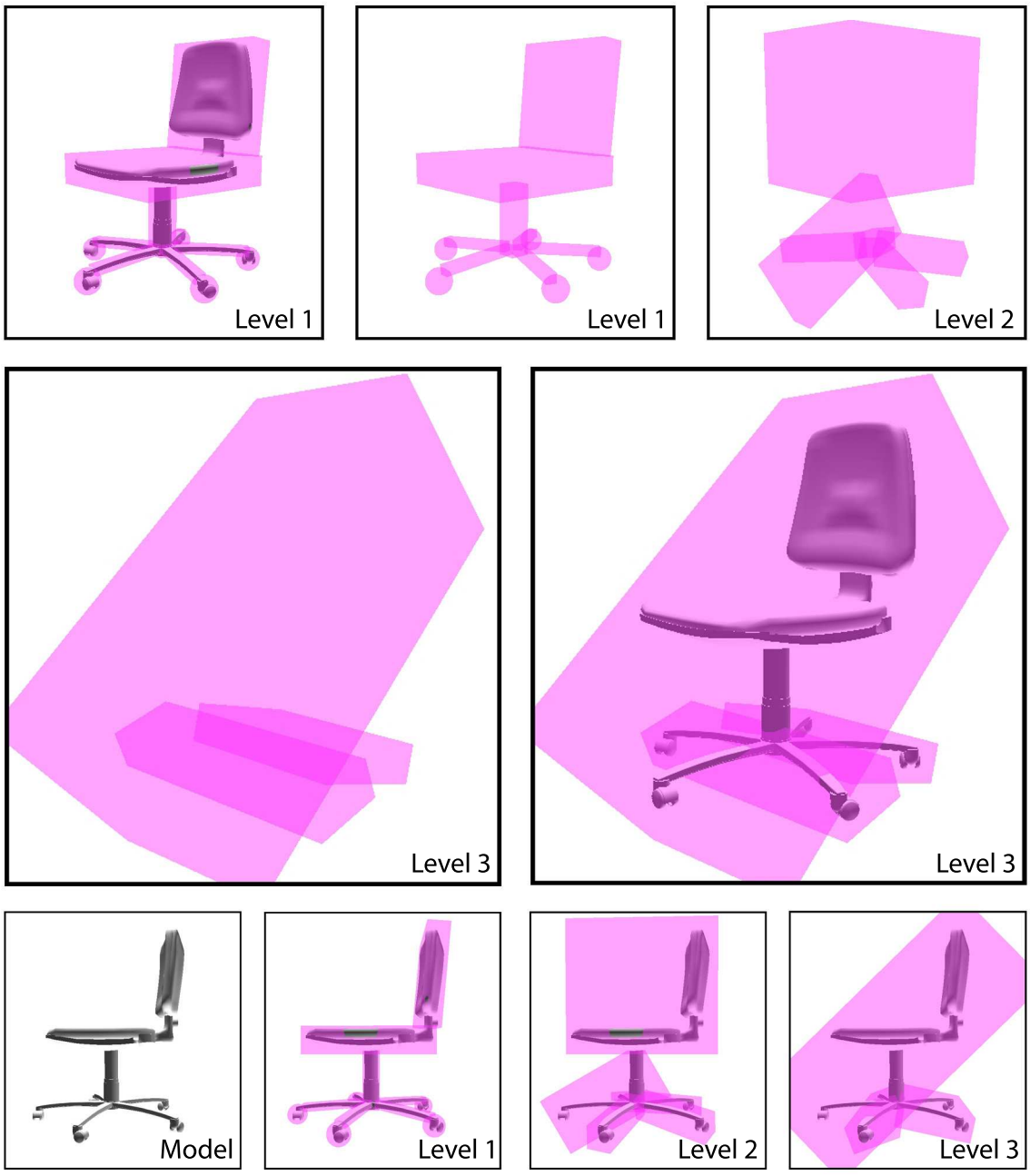


Figure 15: A chair, shown on different levels, with Bounding Volumes.

References

[Borndörfer, 2006] R. Borndörfer, U. Schelten, T. Schlechte, and S. Weider. *A Column Generation Approach to Airline Crew Scheduling*. Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2006.

[Eberly, 2000] D. Eberly. *Intersection of Cylinders*. Geometric Tools, Inc., 2000.

[Erleben, 2002] K. Erleben. *An Introduction to Approximating Heterogeneous Bounding Volume Hierarchies*. Technical Report DIKU-TR-02/04, DIKU, 2002.

[Erleben et al., 2005] K. Erleben, J. Sporring, K. Henriksen, and H. Dohlmann. *Physics-Based Animation*. Charles River Media, 2005.

[Foley et al., 1997] J. D. Foley, A. Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics Principles and Practice - Second Edition in C*. Corrected version from 1997, 18th Printing, Addison Wesley, 1997.

[Goldsmith et al., 1987] J. Goldsmith and J. Salmon. *Automatic Creation of Object Hierarchies for Ray Tracing*. IEEE CGA, 1987.

[Gomez, 1999] M. Gomez. *Simple Intersection Tests for Games*. Gamasutra, 1999.

[Gottschalk et al., 1996] S. Gottschalk, M. C. Lin, and D. Manocha. *OBBTree: A Hierarchical Structure for Rapid Interference Detection*. Computer Graphics (SIGGRAPH'96 Proceedings), 1996.

[Gottschalk, 2000] S. Gottschalk. *Collision Queries using Oriented Bounding Boxes*. PhD thesis, Department of Computer Science, University of North Carolina, 2000.

[Haber et al., 1996] J. Haber, M. Staminger, and H. Seidel. *Enhanced Automatic Creation of Multi-Purpose Object Hierarchies*. IEEE CGA, 2000.

[Hall, 2004] B. G. Hall. *Phylogenetic Trees Made Easy: A How-To Manual, Second Edition*. Sinauer Associates Inc., 2004.

[Klosowski, 1998] J. T. Klosowski. *Efficient collision detection for interactive 3D graphics and virtual environments*. State University of New York, 1998.

[MathWorld, 2006] Wolfram MathWorld - the web's most extensive mathematics resource. `http://mathworld.wolfram.com/BellNumber.html`.

[Müller et al., 1999] G. Müller, S. Schafer, and D. W. Fellner. *Automatic Creation of Object Hierarchies for Radiosity Clustering*. Technical Report TUBS-CG-1999-06, TU Braunschweig, 1999.

[Ogre3D, 2006] Opensource Project, Scene-Oriented, Flexible 3D Engine. `http://www.ogre3d.org`.

[OpenTissue, 2006] Opentissue: Opensource Project, Physics-Based Animation and Surgery Simulation. `http://www.opentissue.org`.

[Pardalos, 1998] P. M. Pardalos and D. Z. Du. *Handbook of Combinatorial Optimization*. Kluwer Academic Publishers, 1998.

[Pisinger, 2004] D. Pisinger. *Branch & Bound Algorithms*. HCØ Tryk, 2004.

[Promel et al., 2002] H. Promel, A. Steger. *The Steiner Tree Problem: A Tour Through Graphs, Algorithms, and Complexity*. Friedrick Vieweg & Son, 2002.

[Somchaipeng, 2005] K. Somchaipeng, K. Erleben, and J. Sporring. *A Multi-Scale Singularity Bounding Volume Hierarchy*. Proceedings of WSCG, pages 179-186, 2005.

[Source, 2006] Valve Source Engine: Commercial game engine, Valve Software, 2006.

[tinyXML, 2006] Opensource Project, XML parser. `http://www.grinninglizard.com/tinyxml`.

[Trifonov et al., 2003] B. Trifonov and K. Ng. *Automatic Bounding Volume Hierarchy Generation Using Stochastic Search Methods*. Mini-Workshop "Stochastic Search Algorithms", University of British Columbia, 2003.'

[Zachmann, 2003] G. Zachmann. *Geometric Data Structures for Computer Graphics*. University of Bonn, 2003.

A Pseudocode

```
    Function : BuildHierarchy(primitives)
1.1 mCand :=
    FindMergeCandidates(primitives);
1.2 groups := OptimalVolumes(mCand);
1.3 while Size(groups)>1 do
1.4 |   mCand :=
    |   FindMergeCandidates(groups);
1.5 |   groups :=
    |   OptimalVolumes(groups,mCand);
1.6 |   return ;
1.7 end
```

Algorithm 1: Pseudocode for the function building an entire hierarchy. A set of primitives is given as input to the function and the function works by repeatedly finding merging candidates and searching for optimal valid solutions with a Branch and Bound algorithm.

```
    Function : OptimalVolumes(groups,mCand)
2.1 stack = new Stack;
2.2 RecB&B(stack, groups, mCand, 0, 0);
2.3 RecB&B(stack, groups, mCand, 0, 1);
2.4 return BestSolution();
```

Algorithm 2: Pseudocode for the function setting up the Branch and Bound function. It works by constructing an empty stack and starting the first two branches of the search tree.

```
    Function : RecB&B(stack, groups, mCand,
                      current, binSelect)
3.1 if SetCover (stack) then
3.2 |   if thisSolution < smallestSolution
    |   then
3.3 |   |   RegisterBestSolution(groups, stack);
3.4 |   |   smallestSolution := thisSolution;
3.5 |   end
3.6 end
3.7 if current > Size(mCand)-1 then
3.8 |   return ;
3.9 end
3.10 bound := MakeBound(groups, stack);
3.11 if smallestSolution < bound then
3.12 |   return ;
3.13 end
3.14 if binSelect=0 then
3.15 |   RecB&B(stack, groups, mCand,
     |   current+1, 0);
3.16 |   RecB&B(stack, groups, mCand,
     |   current+1, 1);
3.17 end
3.18 else
3.19 |   if Conflicts (mCand[current],stack)
     |   then
3.20 |   |   return ;
3.21 |   end
3.22 |   else
3.23 |   |   Push(mCand,stack);
3.24 |   |   RecB&B(stack, groups, mCand,
     |   |   current+1, 0);
3.25 |   |   RecB&B(stack, groups, mCand,
     |   |   current+1, 1);
3.26 |   |   Pop(stack);
3.27 |   end
3.28 end
```

Algorithm 3: Pseudocode for the Branch and Bound function. It uses a stack to build solutions on from a set of merging candidates, and works be searching through all necessary paths in a search tree recursively.

B XML serialization example

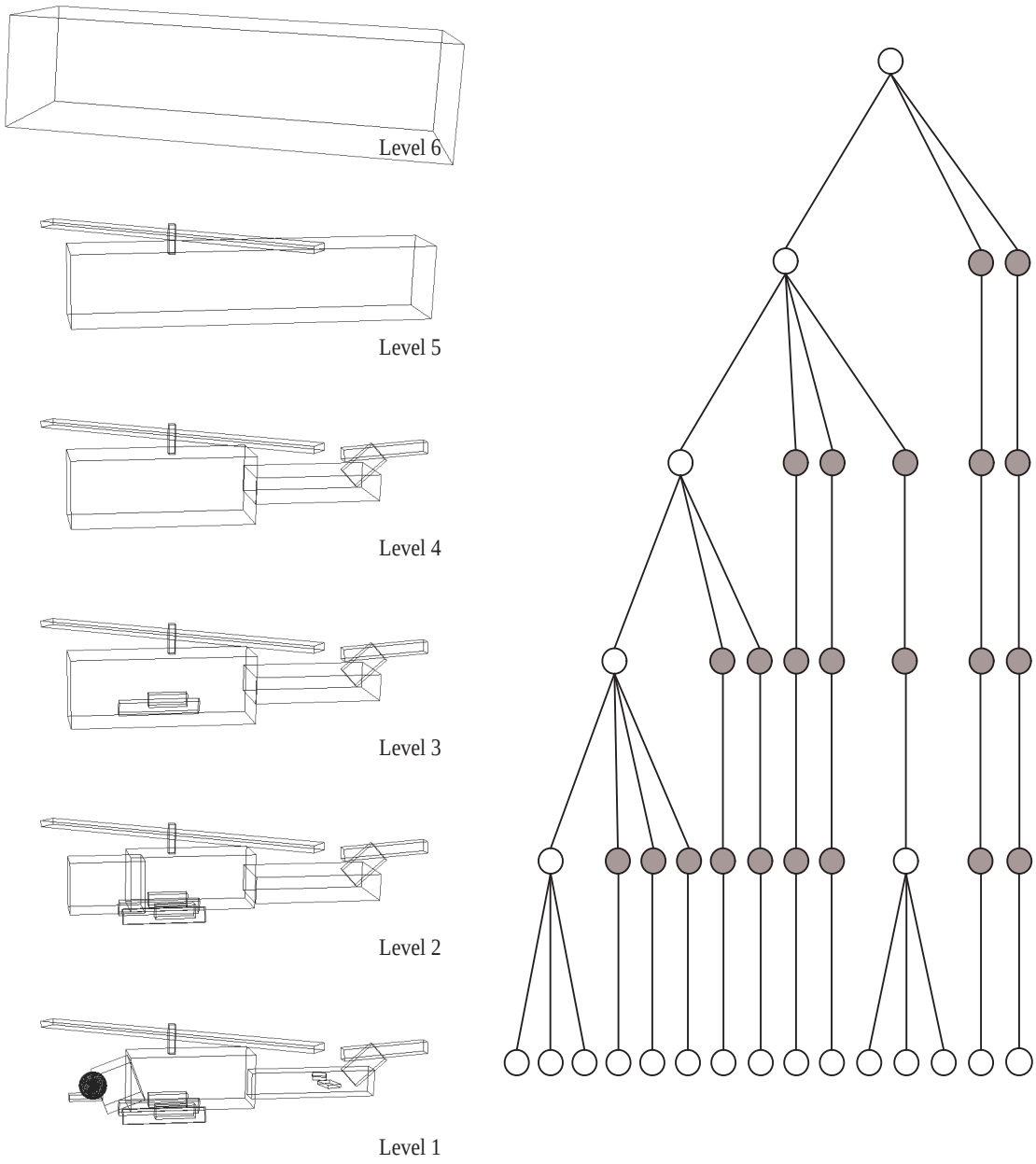
This section contains a full example of a serialized Hybrid Bounding Volume Hierarchy. The root node is the first `volume` element and it encapsulates its children BV elements, which again contain their children until the primitive volumes are reached.

```
<?xml version="1.0" ?>
<hierarchy>
  <volume type="obb" pos="-0.9 4.6 0.1" width="7.7" depth="9.3" height="7.7">
    <volume type="obb" pos="-0.9 0.7 0.5" width="7.7" depth="1.9" height="1.1"
           orientation="-1.0 0.0 0.0 0.0 0.4 -0.9 0.0 -0.9 -0.4">
      <volume type="obb" pos="-2.9 0.8 0.4" width="3.7" depth="1.0" height="1.0"
             orientation="-0.9 -0.3 0.2 -0.2 0.0 -1.0 0.3 -0.9 0.0">
        <volume type="sphere" radius="0.5" pos="-4.2 0.5 0.9" />
        <volume type="obb" pos="-2.7 1.0 0.4" width="0.4" depth="0.4" height="3.1"
                orientation="0.3 -0.1 -0.9 0.0 1.0 -0.1 1.0 0.0 0.3" />
      </volume>
      <volume type="obb" pos="1.2 0.7 0.4" width="3.8" depth="1.0" height="1.0"
             orientation="-0.9 -0.3 0.2 0.2 0.1 1.0 -0.3 1.0 0.0">
        <volume type="sphere" radius="0.5" pos="2.5 0.5 0.8" />
        <volume type="obb" pos="0.9 1.0 0.4" width="0.4" depth="0.4" height="3.1"
                orientation="0.3 0.1 0.9 0.0 1.0 -0.1 -1.0 0.0 0.3" />
      </volume>
    </volume>
    <volume type="obb" pos="-0.9 0.8 -2.0" width="5.1" depth="3.3" height="1.1"
           orientation="-1.0 0.0 0.0 0.0 0.2 -1.0 0.0 -1.0 0.2">
      <volume type="obb" pos="0.4 0.8 -2.0" width="1.0" depth="1.1" height="3.7"
             orientation="-0.7 0.4 0.5 0.4 0.9 -0.2 -0.6 0.1 -0.8">
        <volume type="sphere" radius="0.5" pos="1.1 0.5 -3.1" />
        <volume type="obb" pos="0.3 1.0 -1.8" width="0.4" depth="0.4" height="3.1"
                orientation="0.8 0.04 -0.6 0.0 1.0 0.1 0.6 -0.1 0.8" />
      </volume>
      <volume type="obb" pos="-2.2 0.8 -1.9" width="1.0" depth="1.0" height="3.7"
             orientation="-0.8 0.2 0.6 -0.1 -1.0 0.2 0.6 0.1 0.8">
        <volume type="sphere" radius="0.5" pos="-3.0 0.5 -3.0" />
        <volume type="obb" pos="-2.0 1.0 -1.8" width="0.4" depth="0.4" height="3.1"
                orientation="0.8 0.0 0.6 0.0 1.0 0.1 -0.6 -0.1 0.8" />
      </volume>
    </volume>
    <volume type="obb" pos="-1.0 4.2 -0.5" width="5.4" depth="11.8" height="5.1"
           orientation="-1.0 0.0 0.0 0.0 -0.7 0.7 0.0 0.7 0.7">
      <volume type="obb" pos="-0.9 1.3 1.2" width="1.0" depth="2.9" height="5.6"
             orientation="-1.0 0.0 0.1 0.0 -0.9 -0.5 0.1 -0.5 0.9">
        <volume type="sphere" radius="0.5" pos="-0.8 0.5 3.4" />
        <volume type="obb" pos="-0.9 1.0 1.6" width="0.4" depth="0.4" height="3.1"
                orientation="1.0 0.0 0.0 0.0 1.0 -0.1 0.0 0.1 1.0" />
        <volume type="obb" pos="-1.0 2.2 -0.25" width="0.8" depth="2.9" height="0.8" />
      </volume>
      <volume type="obb" pos="-1.0 6.4 -0.6" width="5.2" depth="5.9" height="6.3">
        <volume type="obb" pos="-1 4.2 -0.3" width="5.2" depth="1.2" height="5.7"/>
        <volume type="obb" pos="-1 7 -3.1" width="4.5" depth="4.5" height="1.0"
                orientation="1.0 0.0 0.0 0.0 1.0 0.1 0.0 -0.1 1.0" />
      </volume>
    </volume>
  </volume>
</hierarchy>
```

C Full hierarchies

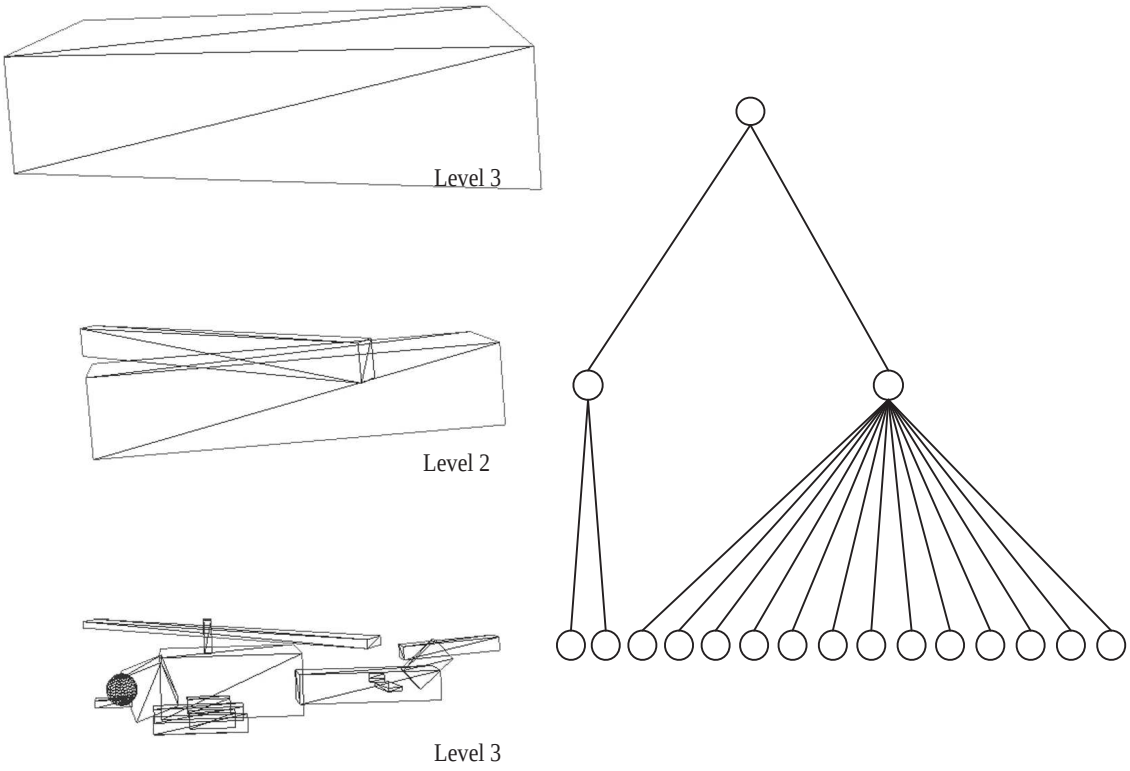
This section contains figures showing full hierarchies for the four objects used to evaluate construction algorithms. The left are of the figures shows hierarchies the OpenTissue fugues are rendered with OpenTissue [OpenTissue, 2006] while the Branch and Bound figures are rendered with Ogre [Ogre3D, 2006]. The right area shows the hierarchy as a tree structure. Some nodes are gray. These are nodes that are *transferred* directly from the level below. Thus, they do not count in the cost of collision testing more than once; only on the lowest level they occur on. They are not included in the total cost of the hierarchy.

C.1 Helicopter hierarchy using OpenTissue



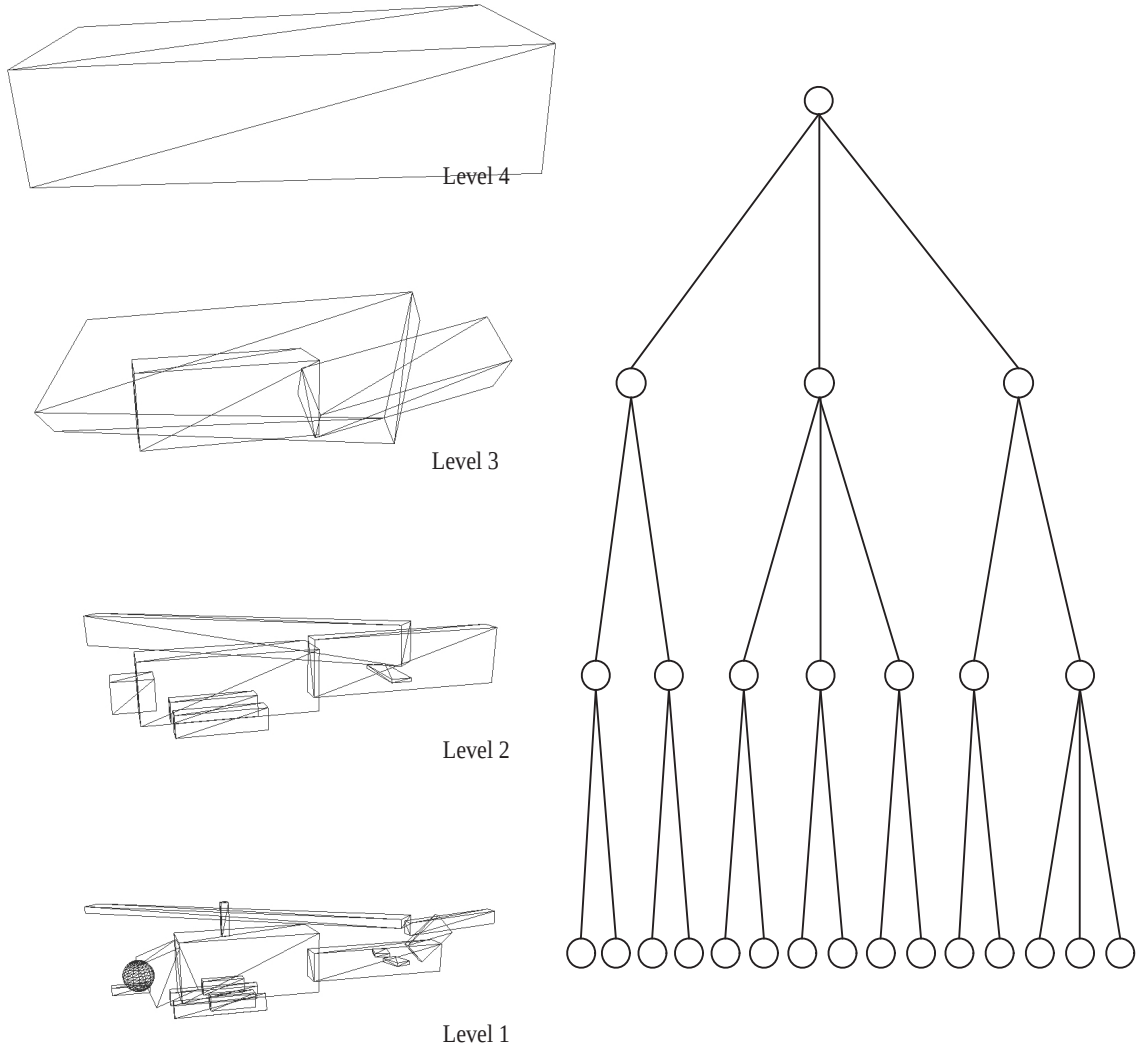
The helicopter HBVH constructed by the OpenTissue algorithm. The HBVH is quite unbalanced and contains major overlaps.

C.2 Helicopter hierarchy using B&B-Tightest



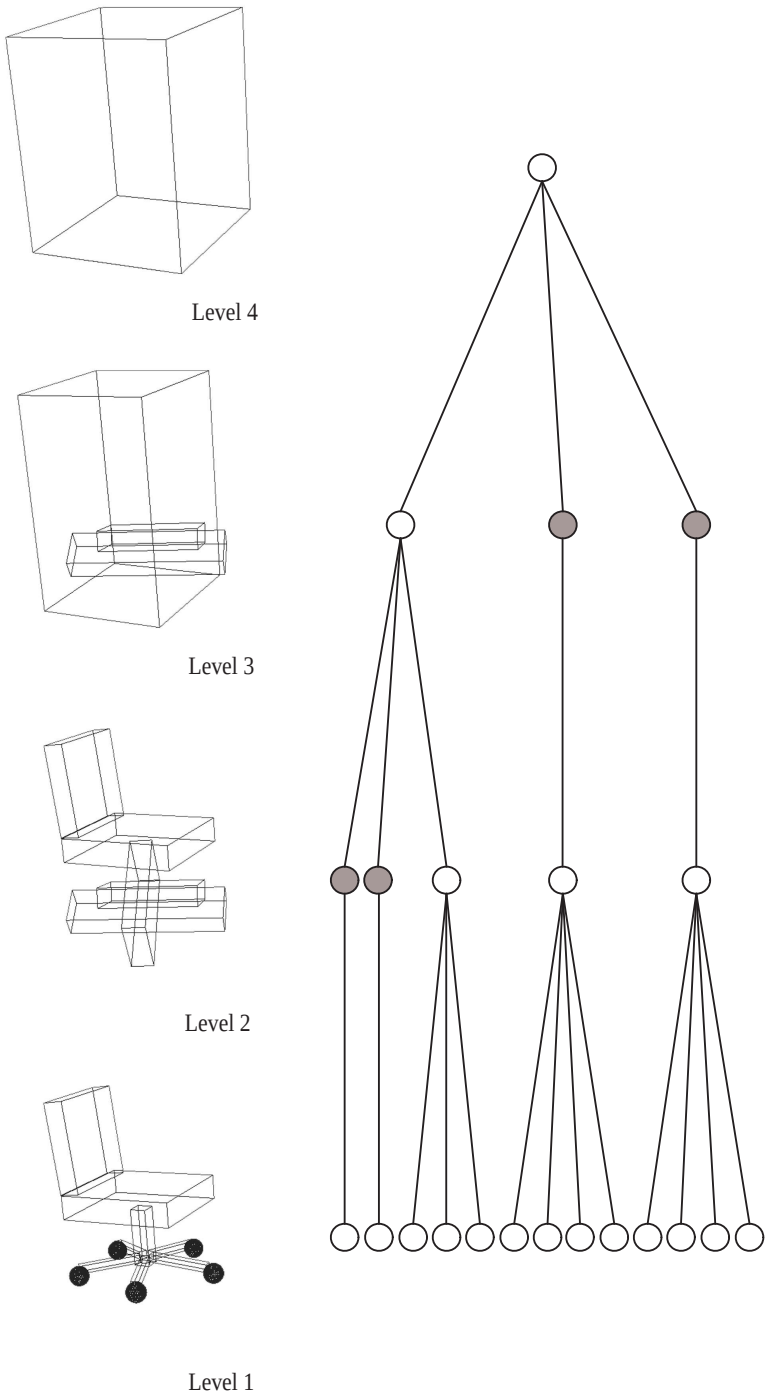
The helicopter HBVH constructed by the B&B-Tightest algorithm. The HBVH is very unbalanced, but it has only minor overlaps.

C.3 Helicopter hierarchy using B&B-Enlarge



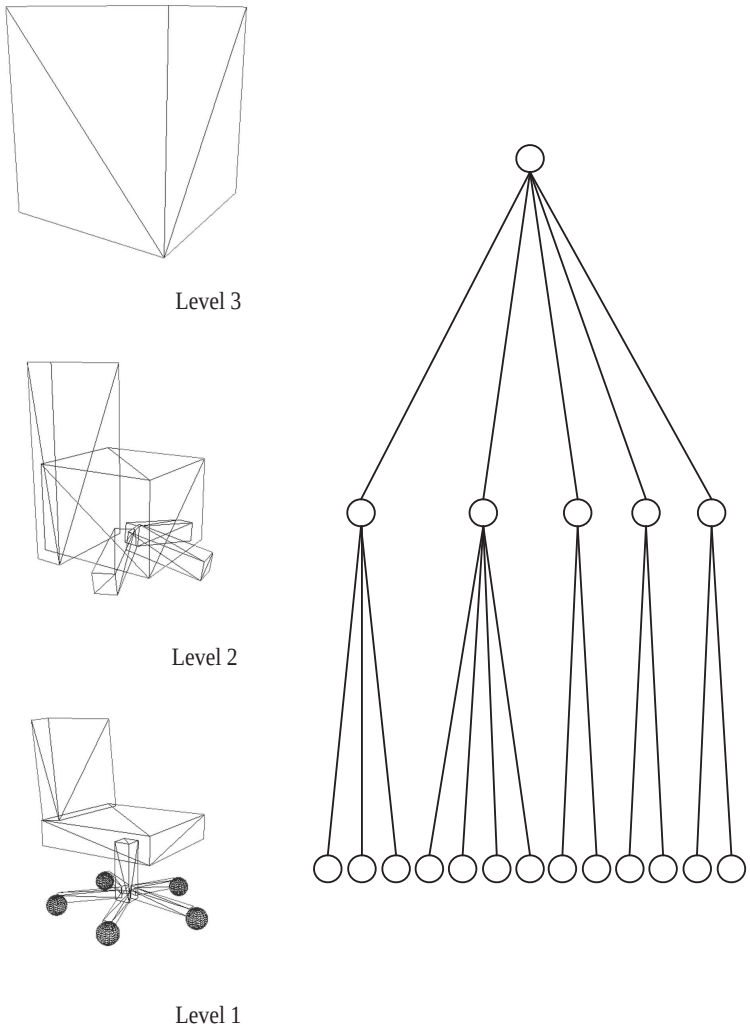
The helicopter HBVH constructed by the B&B-Enlarge algorithm. The HBVH is well balanced, but it has some overlaps.

C.4 Chair hierarchy using OpenTissue



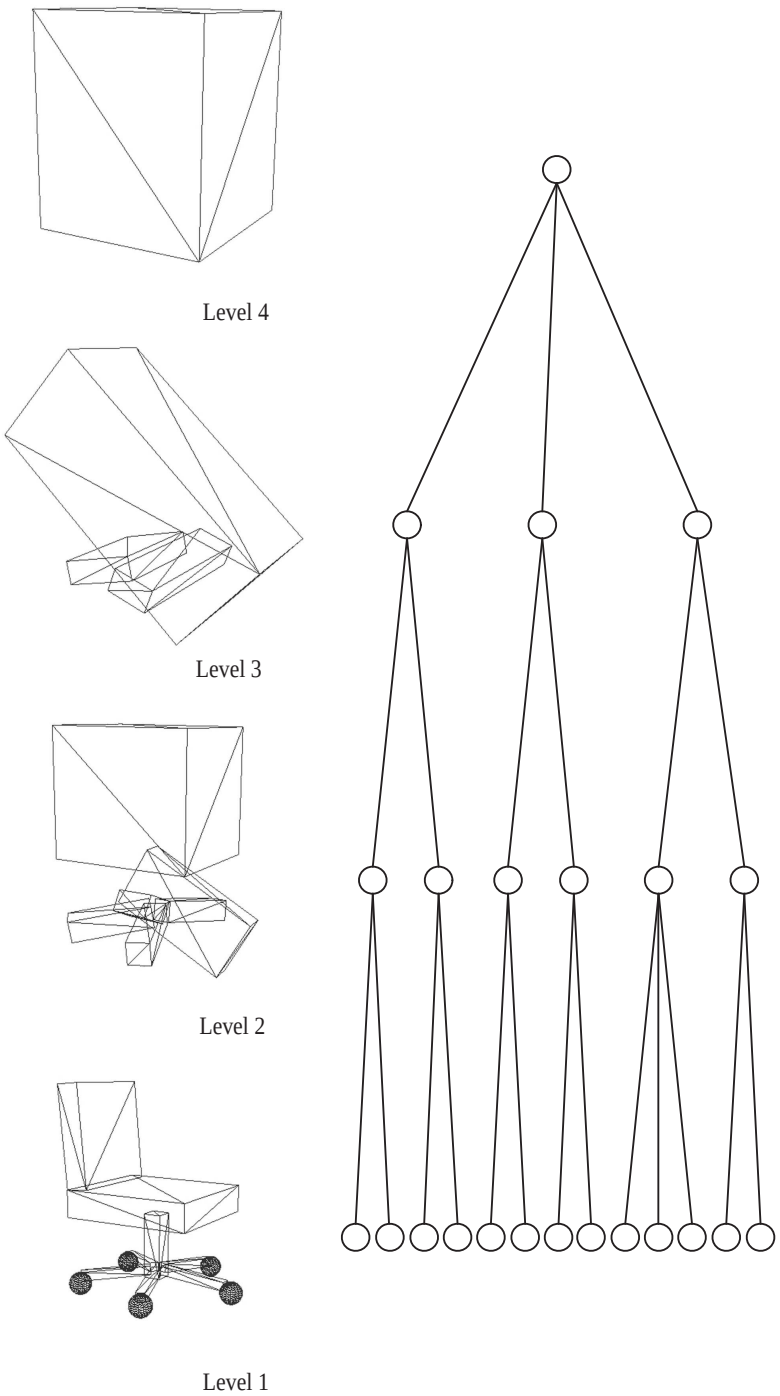
The chair HBVH constructed by the OpenTissue algorithm. The HBVH is quite unbalanced and major overlaps occur on level 3.

C.5 Chair hierarchy using B&B-Tightest



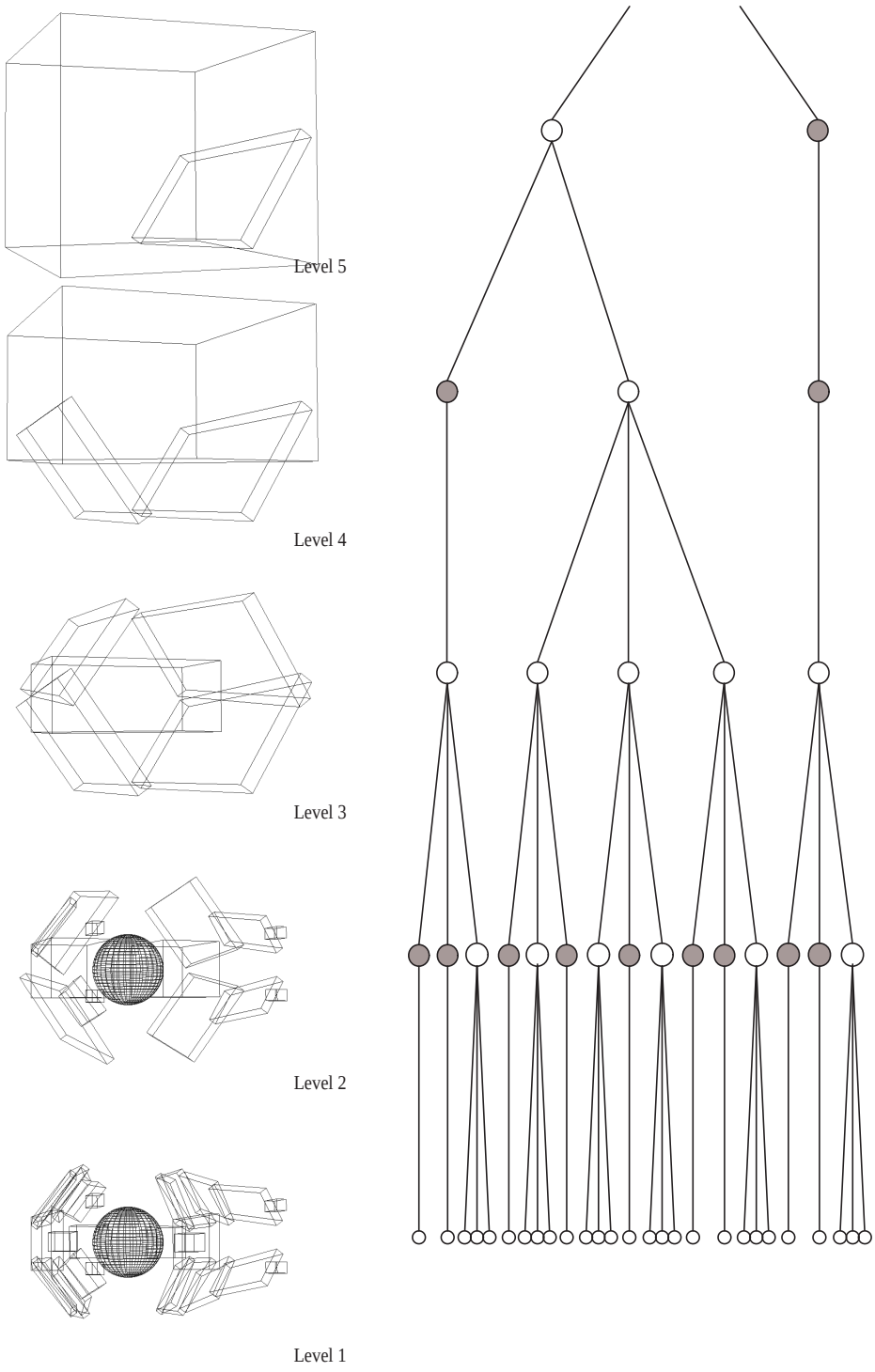
The chair HBVH constructed by the B&B-Tightest algorithm. The HBVH is well balanced, and it has only minor overlaps.

C.6 Chair hierarchy using B&B-Enlarge



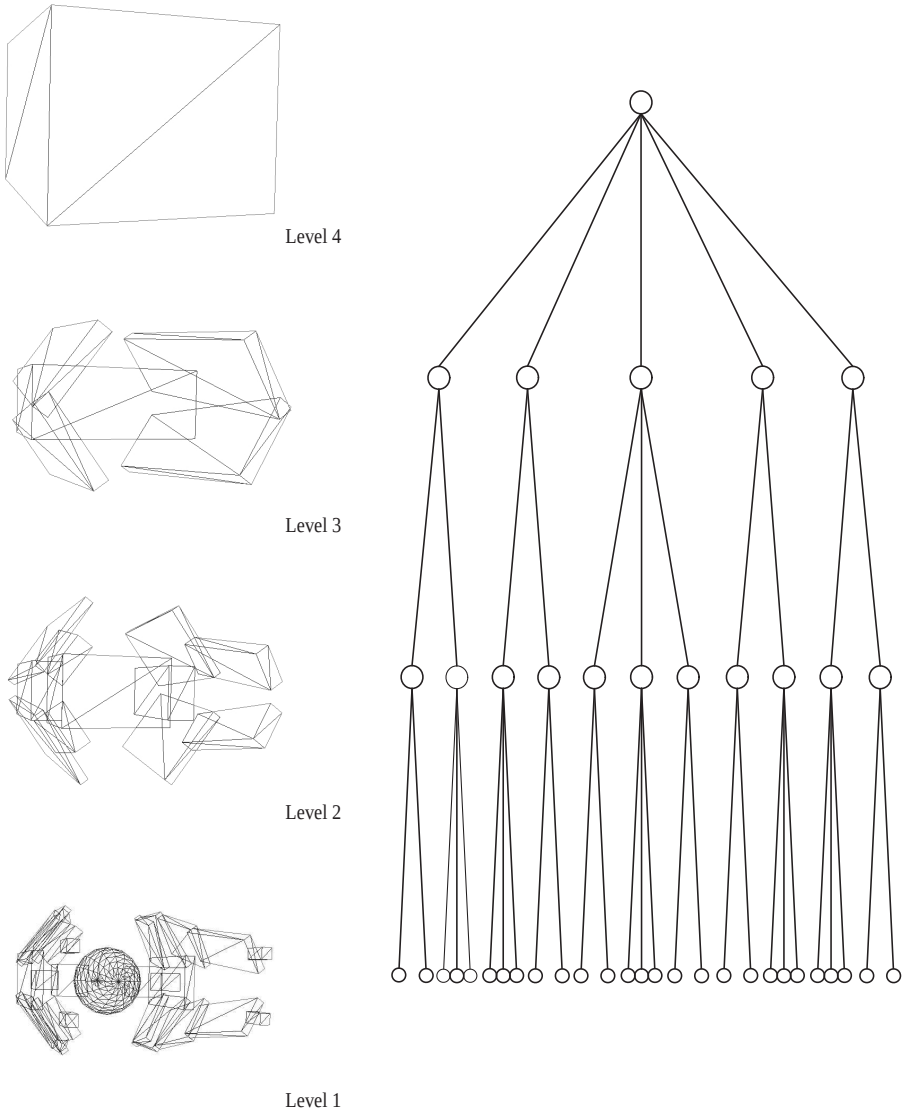
The chair HBVH constructed by the B&B-Enlarge algorithm. The HBVH is well balanced, but it has some overlaps.

C.7 Tie-Inerceptor hierarchy using OpenTissue



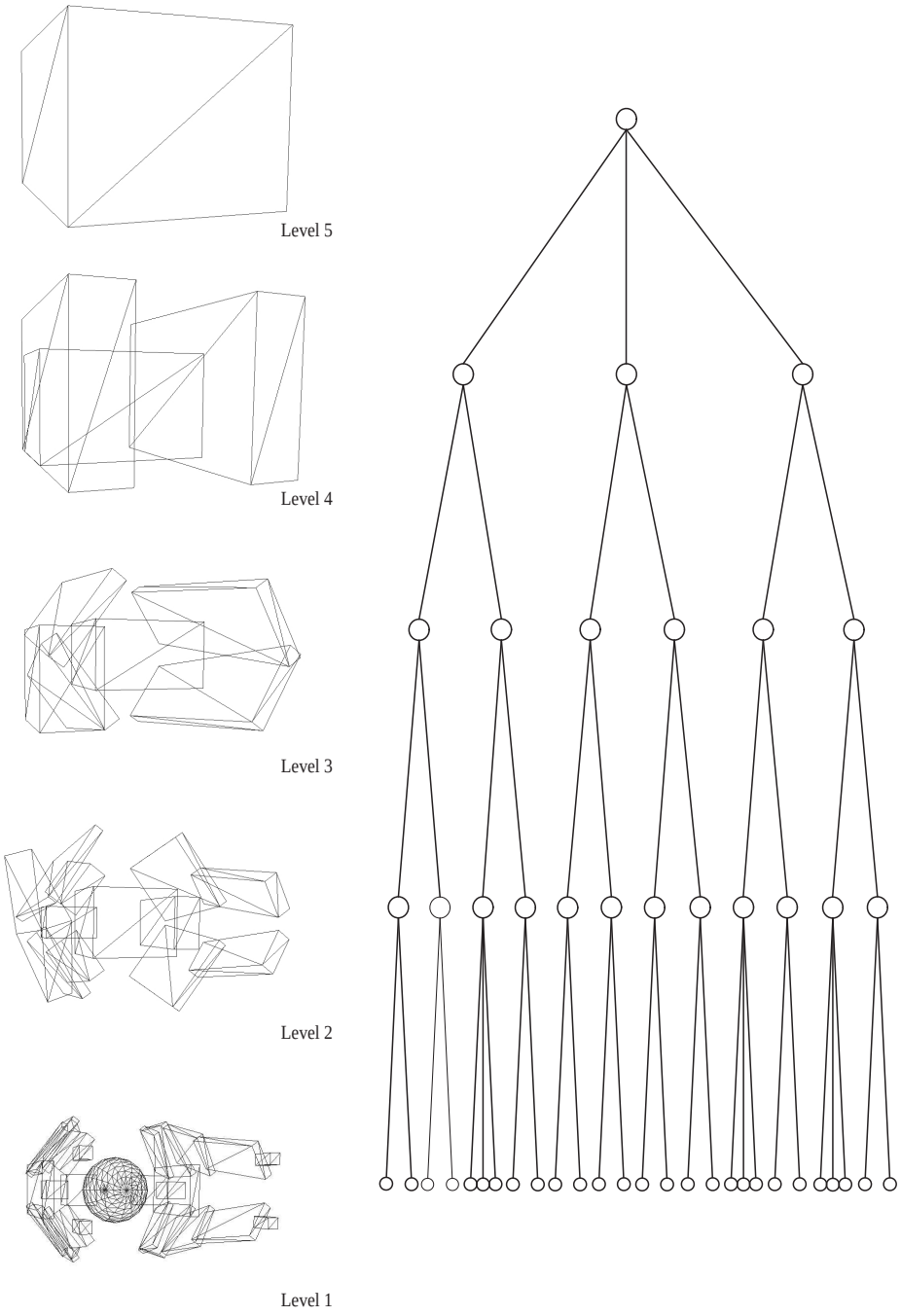
The Tie-Interceptor HBVH constructed by the OpenTissue algorithm. The HBVH is quite balanced, but it has major overlaps.

C.8 Tie-Inerceptor hierarchy using B&B-Tightest



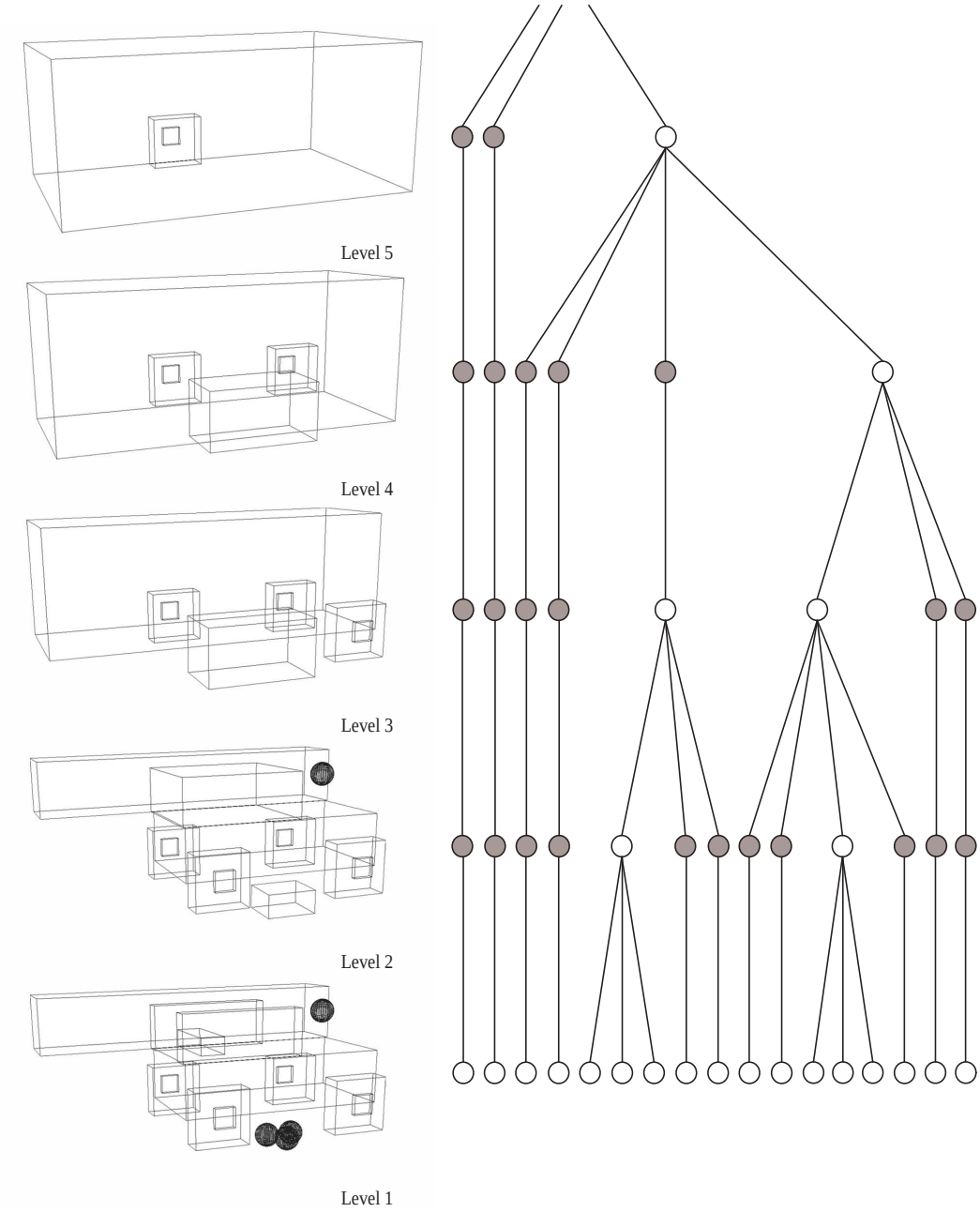
The Tie-Interceptor HBVH constructed by the B&B-Tightest algorithm. The HBVH is well balanced, and it has only minor overlaps.

C.9 Tie-Inerceptor hierarchy using B&B-Enlarge



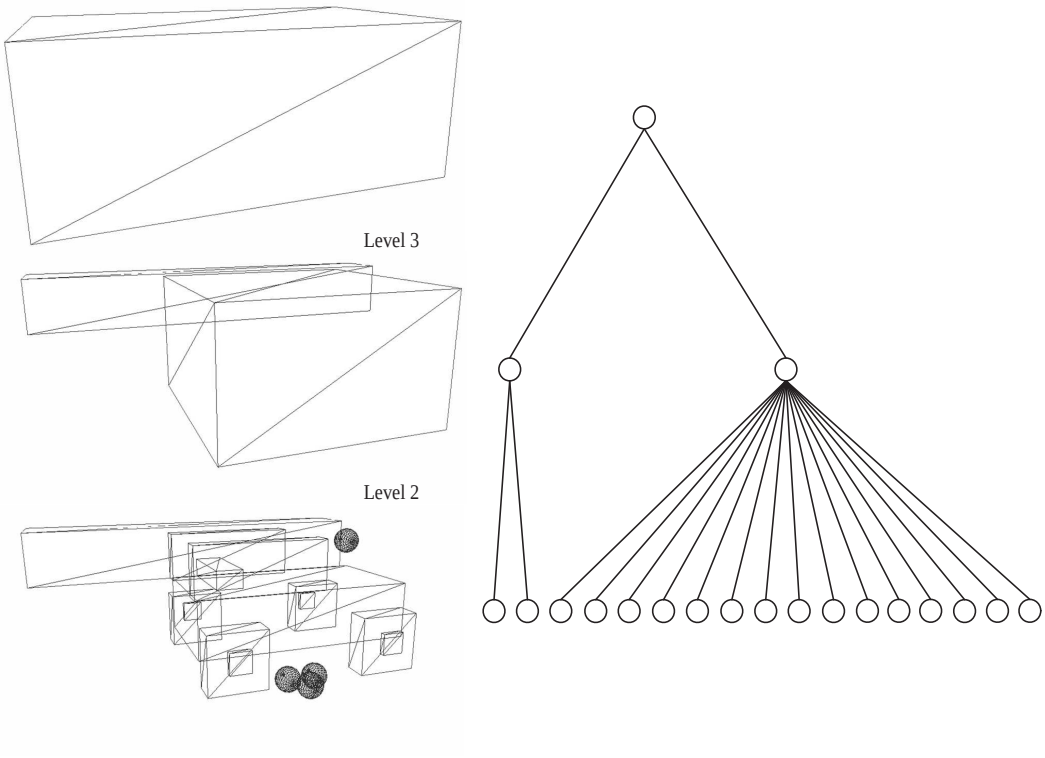
The Tie-Interceptor HBVH constructed by the B&B-Enlarge algorithm. The HBVH is well balanced, but it has some overlaps.

C.10 Cannon hierarchy using OpenTissue



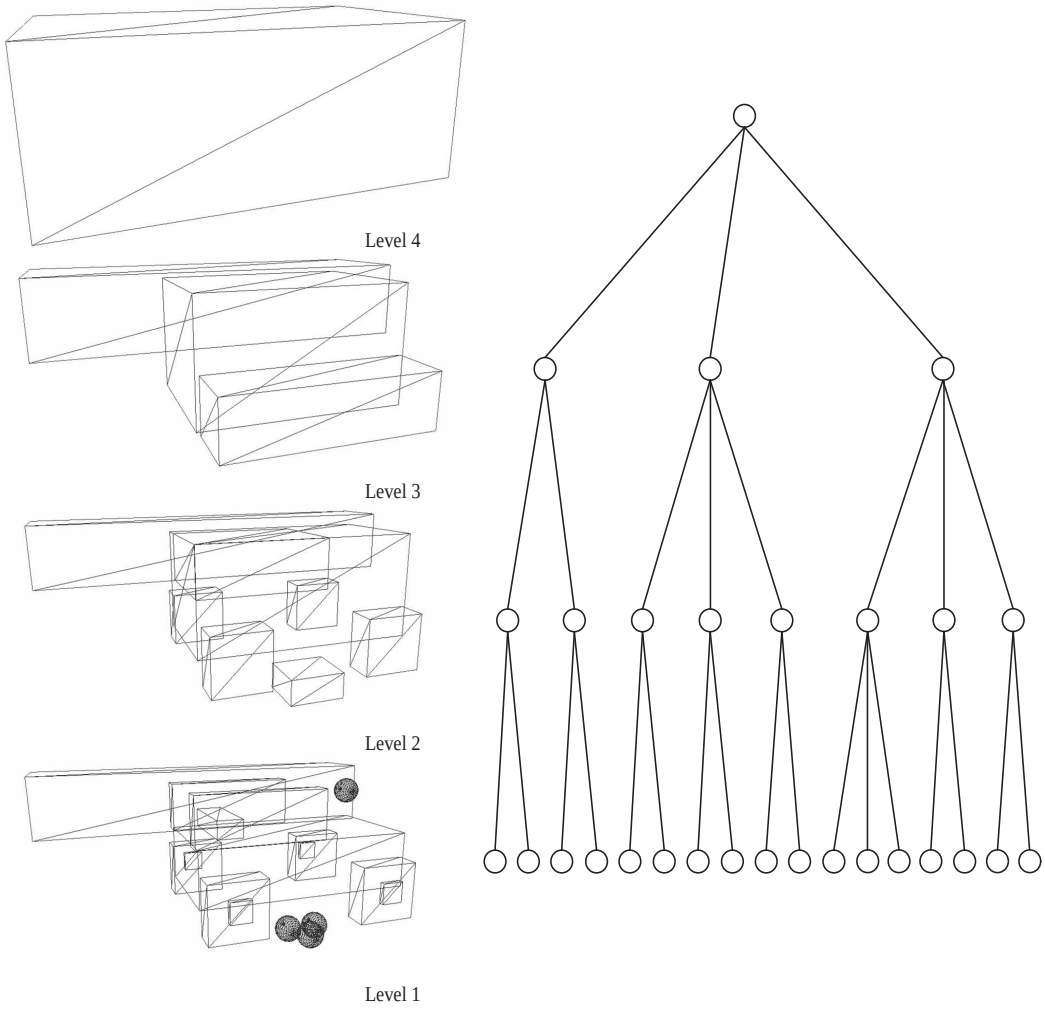
The cannon HBVH constructed by the OpenTissue algorithm. The HBVH is unbalanced, and it has major overlaps.

C.11 Cannon hierarchy using B&B-Tightest



The cannon HBVH constructed by the B&B-Tightest algorithm. The HBVH is unbalanced, and it has some overlaps.

C.12 Cannon hierarchy using B&B-Enlarge



The cannon HBVH constructed by the B&B-Enlarge algorithm. The HBVH is well balanced, but it has some overlaps.