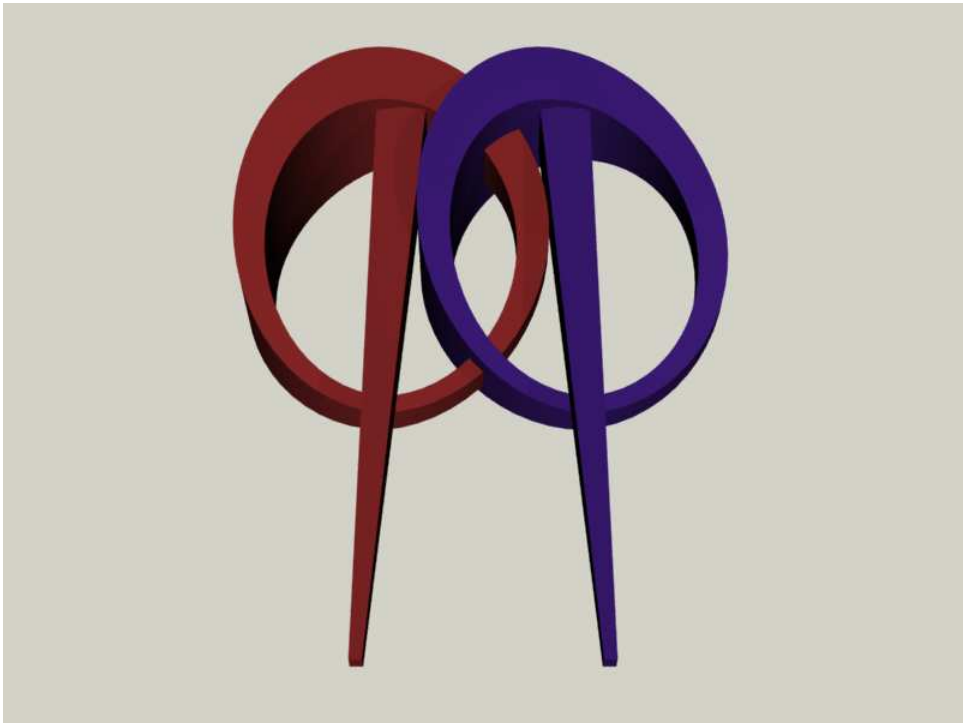
Stackless BVH Collision Detection for Physical Simulation

Jesper Damkjær

Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100, Copenhagen Ø, Denmark
damkjaer@diku.dk

February 28, 2007



Contents

1 Introduction

When doing collision detection in computer games, physical simulations or surgical simulations, different types of collision detection techniques are used. One often used technique is the *bounding volume hierarchy* (BVH). A BVH is a collection of *bounding volumes* (BVs) in a tree structure, where the root of the tree contains a BV that encloses the entire geometry of the object. The leaves of the tree contain BVs that enclose the primitives of the object, e.g. triangles [4].

When doing collision detection between two objects using BVHs, the actual collision detection on the objects is done on the corresponding BVHs. The way this is done is by allocating some area of the computer's memory, and using this as a stack or a queue. In the following a stack will be assumed, but it could as well be a queue.
Two BV-nodes, one from each BVH is added to the stack, starting with the roots of each BVH-tree. When deciding which BV-nodes to compare, two BV-nodes are popped from the top of the stack. When a collision between two BV-nodes it detected, new BV-nodes will be added to the stack, according to some *traversal rule*. A traversal rule is the rule which decides in what manner the BVH-trees are descended. This continues as long as there are elements on the stack [3].

This type of collision detection either needs to know the largest possible size of the stack, and allocate that size of memory before start, or it will have to dynamically expand the size of the stack at runtime. Either way some part of the memory will have to be used to handle the stack, which in worst case may be deleted from the cache and will have to be fetched from memory again, adding overhead to the entire collision detection time.

This paper proposes a new algorithm for doing collision detection between two BVHs without using queue or stack. This is useful for doing e.g. collision detection on a *Graphics Processing Unit* (GPU) where it is not possible, or at least very difficult, to maintain a stack. It is also a hope, that getting rid of the cost for maintaining the stack will generate a speedup.

Throughout this paper it is assumed that the reader is familiar with collision detection and BVHs. Good introductions to collision detection and BVHs can be found in [4] and [3].

The rest of this paper is organized in the following way: In section 2 a breif overview of previous work in the area is presented. In section 3, a stackless algorithm for raytracing is introduced and discussed. In section 4, a common dynamic stack algorithm is presented and compared to the stackless

algorithm from section 3. In section 5, a more specialized and predictable algorithm is presented. This algorithm will be analyzed in section 6 and some properties for it are proved. In section 7, the properties from section 5 are used in combination with the algorithm from section 3 to create a stackless algorithm which traverse the BVHs in the same way as the predictable stack algorithm. In section 8, the stackless algorithm found in section 7 is extended to work on more generalized trees. In section 9, tests of both stackless and stack algorithms are done, and in section 10, the results are compared and discussed. At the end, in section 11, some conclusions will be drawn, and future perspectives outlined.

2 Previous work

Collision detection and bounding volume hierarchies are well researched areas, and there is a lot of litterature on both subjects. In [4] there is an overview of both subjects, where BVHs are used in collision detection. For more indepth detail, see [3].

BVHs have been used in collision detection for a number of years. In ray tracing it is used to find the collision between a ray and an object [5], and in e.g. games it is used to find the collision between two objects in real-time [3]. It is also used to find self intersection in soft objects, e.g. cloth, in physical simulations.

In [7] a stackless technique for doing BVH traversal in raytracing on a GPU is described. The authers introduce an *escape index*, which is a pointer from a node in the tree to another node in the tree.

The escape index is used when an intersection test between the ray and a BV does not result in an intersection. In this case, the escape index points to the next BV-node in the BVH, to test for intersection. The escape index will be explained in more detail in section 3.

To the author's knowledge, no one has yet tried to extend this to handle BVH/BVH collisions. Even though the open source physics engine bullet [1] at present time has an implementation like the above, it only handles an AABB/BVH collision, which is very similar to the ray/BVH collision in [7].

3 The escape index

In [7], the authors describe a BVH traversing technique for ray tracing. The novelty of their algorithm is, that the BVH traversal is done without the use of a stack. This is achieved by creating an index called the escape index for every node in the BVH. The escape index points to another node in the BVH, always from the left to the right. In case there is no intersection between the ray and the current node, the escape index points to the next node in the BVH to test for intersection. In case of no intersections, the traversal will therefore *escape* to another node in the BVH. In figure 1 an example of a BVH with its escape indices is presented.

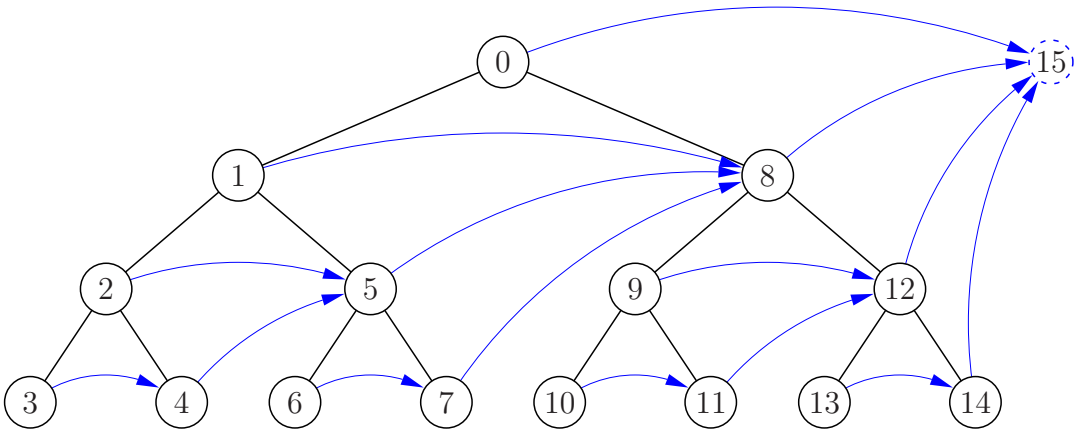


Figure 1: A BVH with its escape indices. In case there is no intersection between a node and the ray, the escape index points to the next node to test in the ray tracing traversal. Note that all the righthand-side nodes point to an imaginary node, labeled 15. If the node 15 is reached, then all possible collisions in the tree have been visited. The last node is labeled with the value of the number of nodes in the BVH.

As can be seen, the escape index is created in such a way, that when the ray and a BV do not intersect, then the escape index points to the next BV to test. This corresponds to the way the nodes will be placed on the stack, if the right child of a node is always pushed onto the stack before the left child. The way the escape index for a node in binary trees is determined, can be expressed in two definitions, depending on whether the node is a left or a right child of its parent.

Definition 3.1. *The escape index for a left child, is the right child of the parent.*

Definition 3.2. *A right child has the same escape index as its parent.*

As can be seen from definition 3.2, and in figure 1, the escape index will be inherited downwards in the tree from a parent to its right child. The escape index for a right child will therefore be the same as the escape index for the first ancestor upwards in the tree that is a left child.

The root node is a special case, and can be defined in the following way:

Definition 3.3. *The root is a left child, and its escape index is the value of the last node in the BVH plus one.*

Originally, the authors of [7] created their technique for ray tracing, where they wanted to find which face of an object a ray hit. Instead of using a ray, we could use a bounding volume of some kind, e.g. a box, a sphere or a cylinder.

Where the authors of [7] were interested in finding the first geometry of the object the ray hit, we are interested in finding all collisions between a given BV, and the BVH.

Based on the above, we can create an algorithm, which finds all collisions between two BVHs without use of the stack. Assume tat all the leaves in the first BVH-tree, hereafter called BVH-tree $\boldsymbol{A}$, are encapsulated in a BV at the lowest level of the BVH. Then we can compare each of these leaf BVs to the other BVH-tree, hereafter called BVH-tree $\boldsymbol{B}$, by just iterating over all the leaf BVs in $\boldsymbol{A}$, and using each of them as the "ray".

This algorithm, called *the leaf algorithm*, is sketched in pseudo-code in algorithm 1.

The leaf algorithm works by comparing all the leaves of BVH-tree $\boldsymbol{A}$, to BVH-tree $\boldsymbol{B}$, one leaf at a time, and using the escape index in $\boldsymbol{B}$ to skip over subtrees in case no collision is found between the leaf and the root of the subtree.

As shown in [4], this scheme results in a lot of redundant testing, and the leaf algorithm does therefore not seem to be very efficient, although a similar technique has been used for cloth collision detection [2]. As an example, take the case where there is little overlap between $\boldsymbol{A}$ and $\boldsymbol{B}$. The algorithm still has to compare every leaf from $\boldsymbol{A}$ to, at least, the root of $\boldsymbol{B}$. Since this is an inefficient way to do collision detection, the next section will try to compare this algorithm to a more common stackbased collision detection algorithm.

Algorithm 1 *The leaf algorithm*: A stackless algorithm for BVH/BVH collision detection. It uses neither a stack or a queue.

```
 1: end-index ← end index for BVH-tree B
 2: for All leaves in BVH-tree A do
 3:   leaf ← next leaf in BVH-tree A
 4:   node ← root of BVH-tree B
 5:   while index of node ≠ end-index do
 6:     if leaf and node are colliding then
 7:       if node is a leaf in BVH-tree B then
 8:         Find intersection between leaf and node
 9:       else
10:         node ← left child of node
11:       end if
12:     else
13:       node ← escape-index of node
14:     end if
15:   end while
16: end for
```

4 The dynamic stack algorithm

The leaf algorithm does not seem to be the best choice for collision detection. A more common approach would be to use a dynamic stack algorithm, as the one presented in algorithm 2.

This algorithm has a traversal rule in line 9, that makes a dynamic choice about which BVH-tree to descend into. The choice is based on the size of the volumes of the BVs, where the BV with the largest volume will be traversed first. The argument is, that by minimizing the largest volume, the chance for a collision is also minimized [3]. This algorithm is often used because its dynamic traversal rule gives it some rather efficient pruning capabilities.

When examining the leaf algorithm and the dynamic stack algorithm, one can see that they both handle three different cases:

1. There is no collision.
2. There is a collision between two leaf nodes.
3. There is a collision between two nodes, which are both non-leaf nodes.

These three cases are what decide what the algorithms must do in the next iteration, and appropriate actions must be taken.

The leaf algorithm must find out which node in BVH-tree $\boldsymbol{B}$ it must use in the next iteration, for comparison with the leaf node from BVH-tree $\boldsymbol{A}$. In

Algorithm 2 *The volume-based stack algorithm*: A dynamic stack-using algorithm for BVH/BVH collision detection.

```
1: Stack S
2: Push roots of BVH-tree A and BVH-tree B onto S
3: while S is not empty do
4:    Pop the top element into the nodes A and B.
5:    if A and B are colliding then
6:       if Both A and B are a leaf then
7:          Find intersection between A and B
8:       else
9:          if volume of A < volume of B then
10:            Push A and right child of B onto S
11:            Push A and left child of B onto S
12:         else
13:            Push right child of A and B onto S
14:            Push left child A and B onto S
15:         end if
16:      end if
17:   end if
18: end while
```

cases 1 and 2, the leaf algorithm will jump to the escape index of the current node in $\boldsymbol{B}$, and in case 3 it will go to the left child of the current node in $\boldsymbol{B}$.

In the first two cases of the stack algorithm, nothing gets pushed onto the stack, and the next iteration will start by popping the top element from the stack. In the third case, two elements containing two nodes each will be pushed onto the stack, and the next iteration will start by popping the top element from the stack, which must be one of the two newly added elements.

This shows that the leaf algorithm is predictable, since it must know which node to use in the next iteration before the end of the current iteration.

Given two nodes to compare in the leaf algorithm, the information about which nodes to compare in the next iteration in any of the three above mentioned cases is already available.

In the case of the dynamic stack algorithm, this would correspond to knowing which element is on the top of the stack, just after an element has been popped.

The example in figure 2 shows, that based on the dynamic nature of the traversal rule of the dynamic stack algorithm, it is not possible to predict which element lies beneath a given element. It is therefore necessary to find a traversal rule which ensures a predictable traversal.

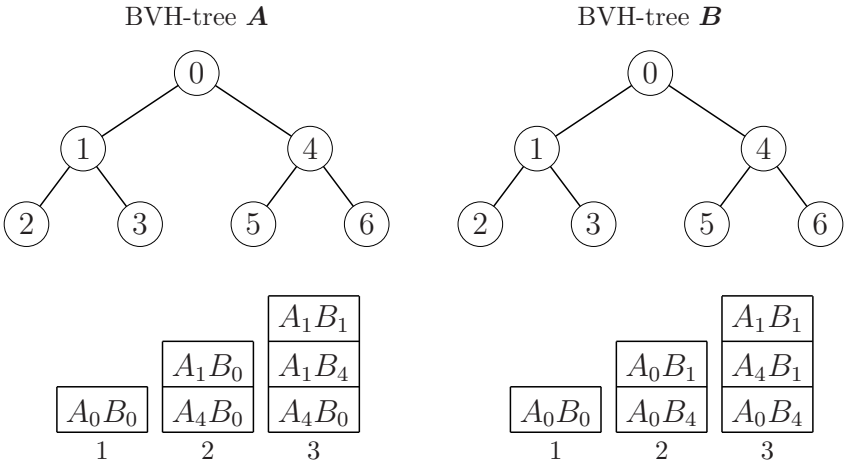


Figure 2: This is an example, that a dynamic traversal rule based on largest volume first, is not predictable. In the left stack example, assume that $A_0 > B_0$ and $A_1 < B_0$. In the right stack example, assume that $A_0 < B_0$ and $A_0 > B_1$. In both examples, when reaching step 3, the elements below the top element (A_1B_1) differ. This is because of the dynamic traversal rule.

5 The predictable stack algorithm

To make it possible to predict the next element on the stack, it would help if elements were pushed onto the stack in an ordered way. That is, the traversal rule must follow a predefined pattern. One traversal rule that does so, is the alternately stepping traversal rule[3]. This rule traverses one step down one of the two BVH-trees in the first iteration. In the second iteration it traverses a step down the other BVH-tree. In the third iteration it traverses another step down the first BVH-tree, and so on. This traversal rule ensures that the elements are always pushed onto the stack in the same, predefined order. An outline of an alternately stepping traversal stack algorithm can be seen in algorithm 3. The algorithm utilizes a function called *Level(node)*. This function returns which *level* a given node has. The level of the node is the node's depth in the BVH-tree, starting at the root. In figure 3 is an example of a tree with its levels.

The alternatly stepping stack algorithm pushes an element containing the roots of the two BVH-trees onto the stack, and starts the while loop. This loop will run as long there are elements on the stack.

In each iteration, the top element is popped from the stack. The two

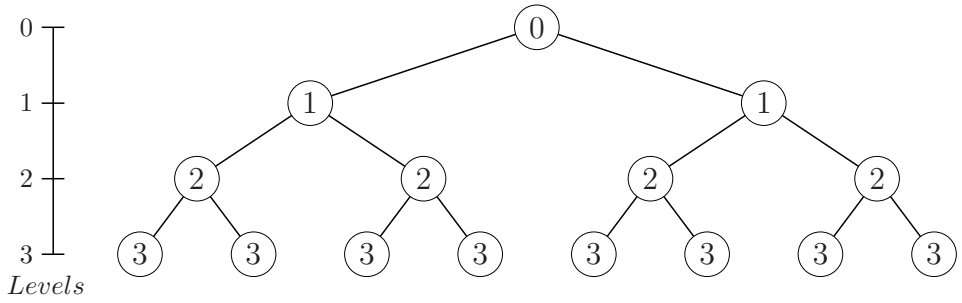


Figure 3: A BVH-tree with depth-levels shown on the left and inside the nodes. The root has level 0. The root's children has level 1. Their children have level 2, and so on. This value is what is returned from the function *Level(Node)*.

Algorithm 3 *The alternately stepping stack algorithm*: Stack-using algorithm for collision detection using BVHs. This algorithm compares the levels of the nodes to decide which BVH-tree to descend into.

```
 1: Stack S
 2: Push roots of BVH-tree A and BVH-tree B onto S
 3: while S is not empty do
 4:    Pop the top element into the nodes A and B.
 5:    if A and B are colliding then
 6:       if both A and B are a leaf then
 7:          Find intersection between A and B
 8:       else
 9:          if level of A = level of B then
10:             Push A and right child of B onto S
11:             Push A and left child of B onto S
12:          else { level of A = level of B − 1 }
13:             Push right child of A and B onto S
14:             Push left child A and B onto S
15:          end if
16:       end if
17:    end if
18: end while
```

nodes in the element will be tested for overlap. If this test is a success, and if the two nodes are not both leaves, new elements will be pushed onto the stack.

The level-check in line 9, guarantees that the elements that are pushed onto the stack, always have two nodes that are on the same level, or that the level of A is one less than the level of B. This is what ensures that the algorithm traverses one single step down through first one of the BVH-trees, and then the other. The alternating traversal combined with the fact, that the elements will always be pushed onto the stack in the same order, is what will result in a predictable algorithm.

The algorithm will only stop to push elements onto the stack when it encounters a leaf in both BVH-trees. This will happen if the leaf nodes lie at the same depth in both of the BVH-trees. To ensure this, and for the ease of demonstration, the two BVH-trees will be limited to fully binary trees of equal height. This limitiation will be dealt with later in section 8. For convinience, definition 3.3, stating that the root nodes are left children, will be used for these BVH-trees as well.

Although this algorithm takes alternating steps, it does not have the same pruning capabilites as the dynamic stack algorithm in the previous section. It is therefore assumed, that this algorithm will not perform as well as the volume-based algorithm in all cases.

In the next section this algorithm will be analysed further.

6 Analysis of the predictable stack algorithm

The purpose of the analysis is to find some predictability in the way elements are pushed onto, and popped from, the stack. That is, during a given iteration, it must be possible to predict which nodes to compare in the next iteration. If such predictability is found, it will be possible to create an algorithm that behaves in the same way as the stack algorithm in algorithm 3, but without the use of a stack.

The stack works in the normal way: Elements are pushed onto the top of the stack, and elements are popped from the top of the stack.

An element on the stack is a collection of two nodes. The first node is from BVH-tree $\boldsymbol{A}$, and the second node is from BVH-tree $\boldsymbol{B}$.

6.1 Notation

Throughout the rest of this paper the following notation will be used. A BVH-tree will be denoted by capital boldfaced italic letters, $\boldsymbol{A}$ and $\boldsymbol{B}$. A node in a BVH-tree will be described by capital italic letters, A and B, where A is a node in $\boldsymbol{A}$, and likewise B is a node in $\boldsymbol{B}$. Left or right children to nodes in the BVH-trees will be descibed as subscripts to the node, where A_l is the left child to the node A and B_r is the right child to the node B. Indices into a node-array will be described using the symbols α and β where α is an index to a node in $\boldsymbol{A}$ and β is an index to a node in $\boldsymbol{B}$. Elements on the stack will be denoted by lowercase boldfaced italic letters, e.g. $\boldsymbol{e}$ and $\boldsymbol{f}$. Sometimes elements will be described by their nodes, $\boldsymbol{e}$=(A, B), or just (A, B). A node in an element will be denoted as a subscript to the element, $\boldsymbol{e}_A$.

6.2 Properties of the predictable stack algorithm

In this section, some properties of the stack-using algorithm will be defined. These properties will be used to describe the predictability of the algorithm.

Property 6.1. *An element $\boldsymbol{e}$ on the stack contains the nodes A and B. The two nodes can either have the same level, or the node A has a level one less than the node B:*

These two possibilities can be described in the following way:

$$level(A) = level(B) \tag{1}$$

$$level(A) = level(B) - 1 \tag{2}$$

Since the trees by definition are balanced, full binary trees, all the leaves in both $\boldsymbol{A}$ and $\boldsymbol{B}$ must have the same level. According to the if statement in line 6 in algorithm 3, when two leaves are found, no new elements are added to the stack.

Proof of property 6.1: By induction; the first element to be pushed onto the stack is, as seen in line 2 of algorithm 3, the roots of the two trees. The roots are trivially on the same level. The property therefore holds for the first element.

Assume that the property holds for the first n elements. Now given the element n from the stack, then either the level of A equals the level of B, or

the level of A is one less than the level of B.

In case the level of A is equal to the level of B, then in lines 10 and 11, two new elements are added to the stack, where the level of A is one less than the level of the children of B (figure 4). These new elements also upholds the property.

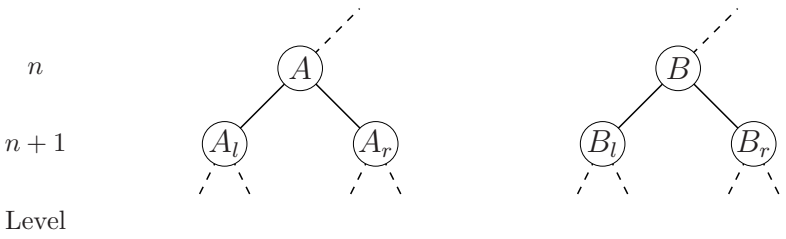


Figure 4: *Property 6.1*: Given a collision between node A and node B, where A and B have the same level, then the two new elements (A, B_r) and (A, B_l) will be pushed onto the stack in that order, where the level of A will be on less than the level of B_r and the level of A will be on less than the level of B_l.

In case the level of A is one less than the level of B, then in lines 13 and 14, two new elements are added to the stack, where the level of the children of A is on the same level as B (figure 5). These new elements also uphold property 6.1. □

Property 6.2. *Two elements are always added to the stack at the same time, and in the same order.*

As can be seen in algorithm 3, whenever elements are added to the stack, there is always added two elements at a time.
In the algorithm it is seen, that there are two ways elements can be added to the stack, which depends on the levels of the nodes.
According to property 6.1, there are two different ways the levels of the nodes can be related. If the levels are related according to equation 1, then the element (A, B_r) is pushed onto the stack just before the element (A, B_l). If the levels are related according to equation 2, then the element (A_r, B) is pushed onto the stack just before the element (A_l, B).

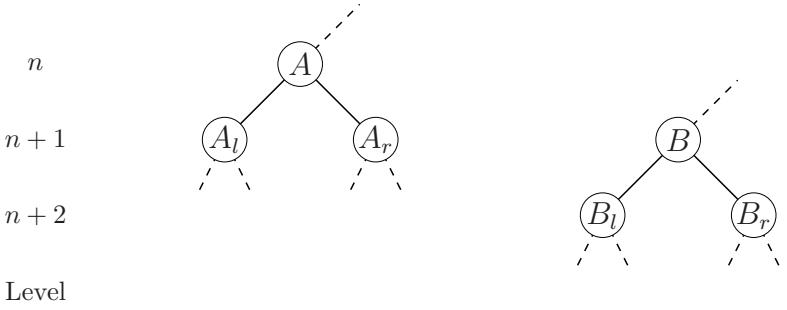


Figure 5: *Property 6.1*: Given a collision between node A and node B, where the level of A has a level one less than B, then the two new elements (A_r, B) and (A_l, B) will be pushed onto the stack in that order, where the nodes in the new elements will have the same level.

Property 6.3. *If the stack is empty at the beginning of an iteration of the while loop, all collisions between BVH-tree* $\boldsymbol{A}$ *and BVH-tree* $\boldsymbol{B}$ *with root nodes* A_0 *and* B_0 *have been found.*

At the beginning of the very first iteration, only the element containing the root nodes of the two BVH-trees is on the stack. This element is popped, leaving the stack empty, and the element's two nodes are tested for collision. In case of no collision between the root nodes, property 6.3 trivially holds. In the case where the root nodes collide, new elements will be pushed onto the stack before the beginning of the next iteration. As long as collisions between nodes occur, new elements will be added to the stack. According to [6], a tandem traversal will find all collisions between two BVHs. Therefore, when the stack is empty, all collisions between the BVH-trees spanned from the root nodes of BVH-tree $\boldsymbol{A}$ and BVH-tree $\boldsymbol{B}$, must have been found.

Property 6.4. *Assume a stack with top element* $\boldsymbol{e}$ *and the element second to the top* $\boldsymbol{f}$*. When the element* $\boldsymbol{e}$ *is popped from the stack, all collisions between* $\boldsymbol{e}_A$ *and* $\boldsymbol{e}_B$ *must have been found before the element* $\boldsymbol{f}$ *can be popped.*

Assume a substack starting from the element $\boldsymbol{e}$. This substack has only one element, $\boldsymbol{e}$, containing the root nodes $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$. According to property 6.3, when this substack is empty, all collisions between the two BVHs with root nodes $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ have been found. This leaves the element $\boldsymbol{f}$ on the top of the original stack.

Property 6.5. *When popping an element (A, B_r) from the stack, where A has a level one less than B_r, the stack will be the same as when the element (A, B) was popped.*

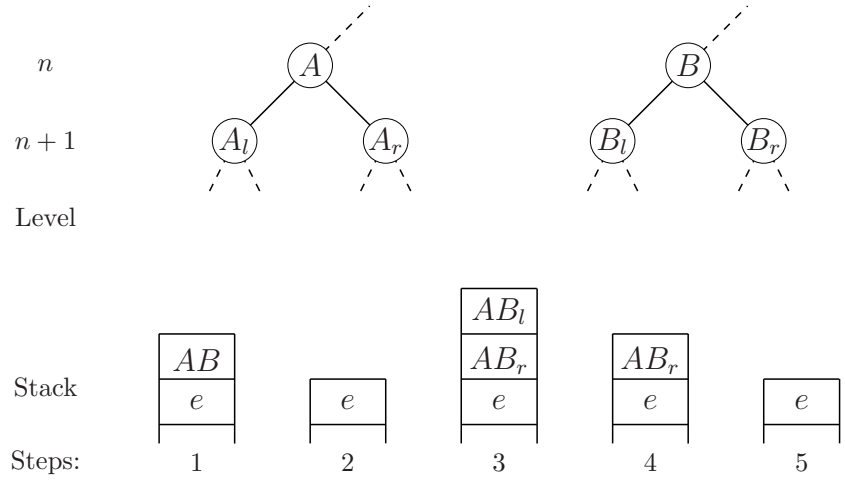


Figure 6: *Property 6.5*: When the element (A, B_r) is popped in step 5, the stack will look exactly as it did when the element (A, B) was popped in step 2.

Assume two nodes, A and B from two different BVH-trees are in an element on the top of the stack. The nodes are on the same level, see figure 6. In step 1 in the figure, the element (A, B) is on the top of the stack. At the beginning of a loop iteration, this element is popped (step 2). Assuming A and B are colliding, two new elements (A, B_r) and (A, B_l) will be pushed onto the stack according to property 6.2 (step 3).
At the beginning of the next iteration of the loop, the top element (A, B_l) will be popped (step 4). According to property 6.4, all collisions between the subtrees with roots A and B_l will then be found. After this in the beginning of a new iteration, the element (A, B_r) will be popped (step 5), leaving the stack in exactly the same way as it was when the element (A, B) was popped in step 2.

Property 6.6. *Popping an element (A_r, B_l) from the stack, where A_r and B_l have the same level, will give the same stack, as when the element (A, B_l) was popped.*

Assume two nodes A and B_l from two different BVH-trees are in an element on the top of the stack. The node A has a level one less than the

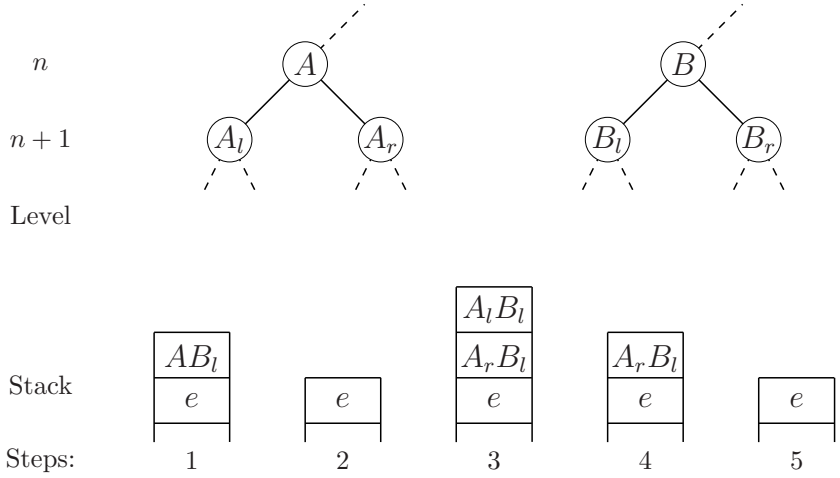


Figure 7: *Property 6.6*: When the element (A_r, B_l) is popped in step 5, the stack will look exactly as it did when the element (A, B_l) was popped in step 2.

node B_l (see figure 7).
In step 1, the element (A, B_l) is on the top of the stack. At the beginning of an iteration of the loop, this element is popped (step 2). Assuming A and B_l are colliding, two new elements, (A_r, B_l) and (A_l, B_l) are pushed onto the stack, according to property 6.2 (step 3).
At the beginning of the next iteration of the loop, the top element (A_l, B_l) will be popped (step 4). According to property 6.4, all collisions between the subtrees with roots A_l and B_l will then be found, leaving the stack as in step 4. After this, at the beginning of a new loop iteration, the element (A_r, B_l) will be popped (step 5), leaving the stack in exactly the same way as it was when the element (A, B_l) was popped in step 2.

Property 6.7. *Popping an element (A_r, B_r) from the stack, where A_r and B_r have the same level, will give the same stack as when the element (A, B) was popped.*

Assume the two nodes A and B from two different BVH-trees are in an element on the top of the stack. The nodes are on the same level.
Step 1 to step 5 are the same as in property 6.5.
Assume there is a collision between A and B_r, then in step 6, two new elements (A_r, B_r) and (A_l, B_r) are pushed onto the top of the stack. At the beginning of the next loop iteration, the top element (A_l, B_r) will be popped from the stack (step 7). According to property 6.4, all collisions between the

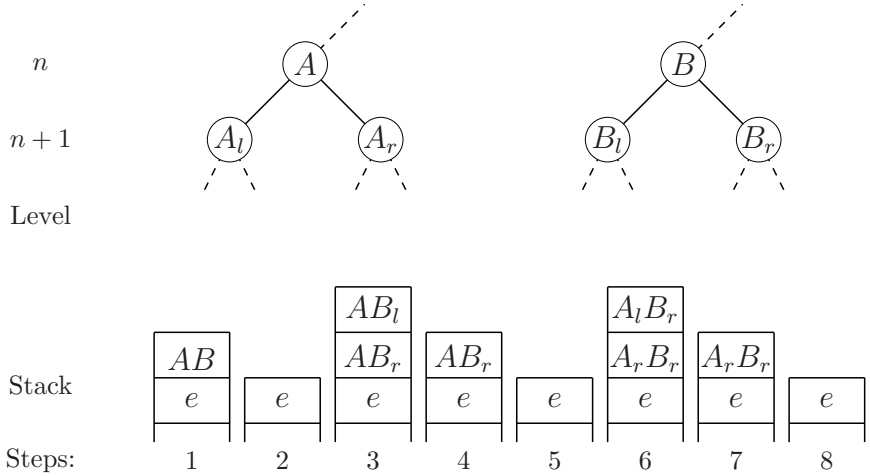


Figure 8: *Property 6.7*: When the element (A_r, B_r) is popped in step 8, the stack will look exactly as it did when the element (A, B_r) was popped in step 5, and according to property 6.5, the stack will be the same as when the element (A, B) was popped in step 2.

subtrees with roots A_l and B_r will then be found. After this at the beginning of a new iteration, the element (A_r, B_r) will be popped (step 8), leaving the stack exactly as it was in step 5, and according to property 6.5, it will be the same as in step 2, when the element (A, B) was popped.

Property 6.8. *The two nodes A and B, in an element $\mathbf{e}$ can be related in eight different ways.*

It is only possible for node A to be either a left child, or a right child. The same goes for B. The level of A must be the same as the level of B, or the level of A must be one less than the level of B. This gives eight different relations, as can be seen in table 1.

6.3 Analysing the predictable stack algorithm

In the previous subsection, some properties for the stack-using algorithm were found. In this section, an analysis of the algorithm will be made using these properties.

The purpose of this analysis is to show that given some element on the stack, it is possible to deduce which element is placed below it on the stack.

As stated in property 6.8, there are eight different possible relations between the two nodes in an element on the stack. These eight cases can be

seen in table 1. In the following, each of the eight cases will be analysed, and for each case it will be shown which element will be on the top of the stack when an element of the given case is popped.

case	levels: $A = B$	levels: $A = (B - 1)$	A_l	A_r	B_l	B_r
1	X		X		X	
2	X		X			X
3		X	X		X	
4		X		X	X	
5	X			X	X	
6		X	X			X
7	X			X		X
8		X		X		X

Table 1: The eight possible relations between two nodes in an element on the stack (property 6.8). Either the A-node can be a left child of its parent (A_l) or the A-node can be a right child of its parent (A_r). The B-node can either be a left child of its parent (B_l) or the B-node can be a right child of its parent (B_r). The level of the A-node is either the same as the level of the B-node ($A = B$) or the level of the A-node is one less than the level of the B-node ($A = (B - 1)$).

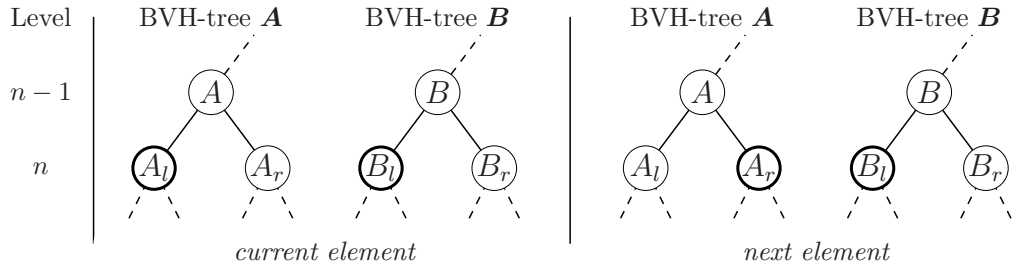


Figure 9: **Case 1**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_l and B_l (thick nodes in current element), and A_l has the same level as B_l, then the next element on the stack - the element below $\boldsymbol{e}$ on the stack - will contain the two nodes A_r and B_l (thick nodes in next element).

Cases 1 and 2:

In case 1, an element $\boldsymbol{e}$ on the stack is given, containing an element from BVH-tree $\boldsymbol{A}$ and an element from BVH-tree $\boldsymbol{B}$. The node from $\boldsymbol{A}$ is a left child, and the node from $\boldsymbol{B}$ is also a left child. The two nodes in the element have the same level.
Let the parent of the node in $\boldsymbol{e}$ from $\boldsymbol{A}$ be the node A, and let the parent of

the node from $\boldsymbol{B}$ be the node B. That is, $\boldsymbol{e}$ contains the nodes A_l and B_l, see figure 9.

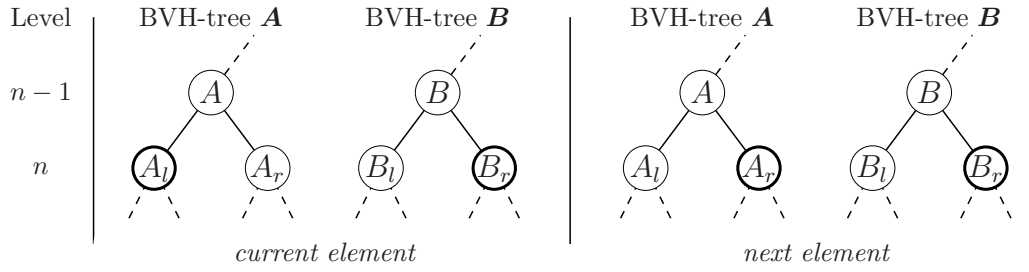


Figure 10: **Case 2**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_l and B_r (thick nodes in current element), and A_l has the same level as B_r, then the next element on the stack - the element below $\boldsymbol{e}$ on the stack - will contain the two nodes A_r and B_r (thick nodes in next element).

From property 6.2, it is seen that if the bounding volumes of the two nodes in an element collide, and the two nodes are on the same level, then two new elements will be added to the stack, where the nodes in each of the elements are not on the same level. If the two colliding nodes in an element are on different levels, then two new elements are added to the stack, containing nodes that are on the same level.
Since the nodes A_l and B_l are on the same level, then according to property 6.2, they must have been pushed onto the stack because there was a collision between the nodes A and B_l. According to property 6.2, the element (A_l, B_l) was pushed onto the stack just after the element (A_r, B_l), which must therefore be the next element on the stack, that is, below the element (A_l, B_l).
The only difference from case 1 to case 2, is that the node from BVH-tree $\boldsymbol{B}$ is a right child. As was seen in case 1, whether $\boldsymbol{e}_B$ was a left or a right child did not have any influence on what the next element was. Therefore in case 2, given an element (A_l, B_r) on the stack, then the element below it on the stack will be (A_r, B_r) (figure 10).
These two cases can be summarized in the following way: Given an element, $\boldsymbol{e}$, on the stack where $\boldsymbol{e}_A$ is a left child, and the level of $\boldsymbol{e}_A$ is the same as the level of $\boldsymbol{e}_B$, then the next element, $\boldsymbol{f}$, on the stack has $\boldsymbol{f}_A$ to be the right child of the parent of $\boldsymbol{e}_A$ and $\boldsymbol{f}_B$ to be the same as $\boldsymbol{e}_B$.

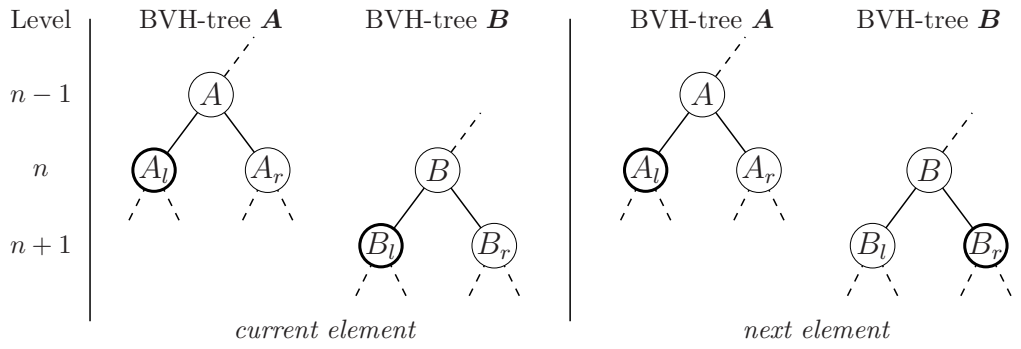


Figure 11: **Case 3**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_l and B_l (thick nodes in current element), and if the level of A_l is one less than the level of B_l, then the next element on the stack - the element below $\boldsymbol{e}$ on the stack - will contain the two nodes A_l and B_r (thick nodes in next element).

Cases 3 and 4:

In case 3, an element $\boldsymbol{e}$ is given, containing a node from BVH-tree $\boldsymbol{A}$ and a node from BVH-tree $\boldsymbol{B}$. The node from $\boldsymbol{A}$ is a left child, and the node from $\boldsymbol{B}$ is also a left child. The node in $\boldsymbol{e}_A$ has a level one less than the node in $\boldsymbol{e}_B$.
Let the parent of the node in $\boldsymbol{e}_A$ be the node A, and let the parent of the node in $\boldsymbol{e}_B$ be the node B. That is, $\boldsymbol{e}$ contains the nodes A_l and B_l, see figure 11.

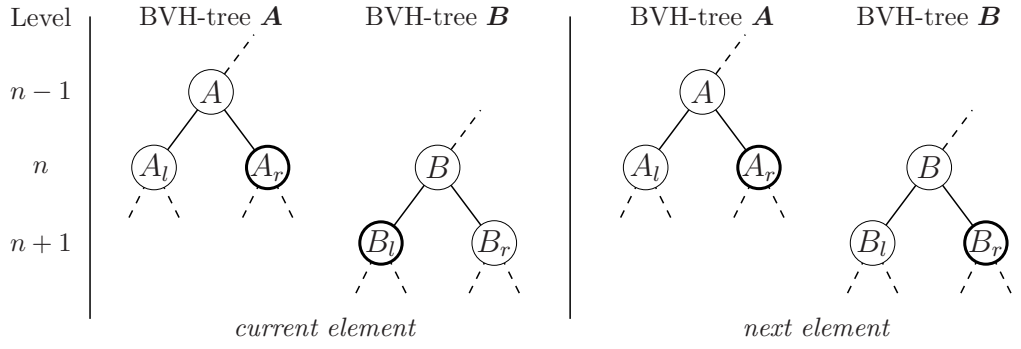


Figure 12: **Case 4**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_r and B_l (thick nodes in current element), and if the level of A_l is one less than the level of B_l, then the next element on the stack - the element below $\boldsymbol{e}$ on the stack - will contain the two nodes A_r and B_r (thick nodes in next element).

Since node A_l has a level one less than node B_l, then according to property 6.2, they must have been pushed onto the stack because there was a collision

between A_l and B. According to property 6.2, the element (A_l, B_l) was pushed onto the stack just after the element (A_l, B_r), which must therefore be the next element on the stack.
The only difference from case 3 to case 4, is that the node from $\boldsymbol{A}$ is a right child. As was seen in case 3, whether $\boldsymbol{e}_A$ was a left or a right child, did not have any influence on what the next element was. Therefore in case 4, given an element (A_r, B_l) on the stack, then the element below it on the stack will be (A_r, B_r) (figure 12).
These two cases can be summarized in the following way: Given an element, $\boldsymbol{e}$, on the stack where $\boldsymbol{e}_B$ is a left child, and the node $\boldsymbol{e}_A$ has a level one less than the node $\boldsymbol{e}_B$, then the next element, $\boldsymbol{f}$, on the stack has $\boldsymbol{f}_A$ to be the same as $\boldsymbol{e}_A$, and $\boldsymbol{f}_B$ to be the right child of the parent of $\boldsymbol{e}_B$.

Case 5:

In this case an element, $\boldsymbol{e}$, is given, containing a node from BVH-tree $\boldsymbol{A}$ and a node from BVH-tree $\boldsymbol{B}$. The node from $\boldsymbol{A}$ is a right child and the node from $\boldsymbol{B}$ is a left child. The nodes have the same level.
Let the parent of the node in $\boldsymbol{e}$ from $\boldsymbol{A}$ be the node A, and let the parent of the node from $\boldsymbol{B}$ be the node B. That is, $\boldsymbol{e}$ contains the nodes A_r and B_l, see figure 13.

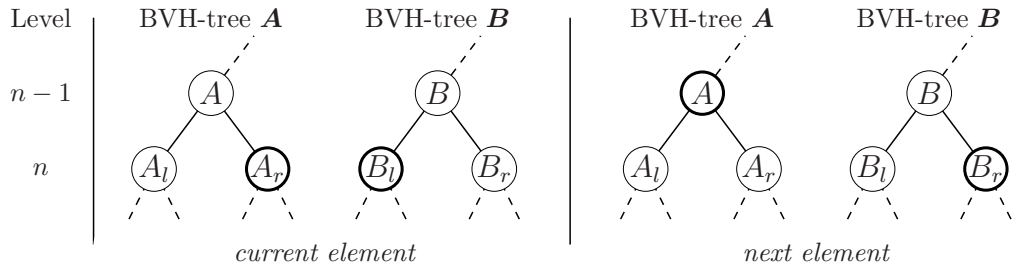


Figure 13: **Case 5**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_r and B_l (thick nodes in current element), and if the level of A_r is the same as the level of B_l, then the next element on the stack - the element below $\boldsymbol{e}$ - will contain the two nodes A and B_r (thick nodes in next element).

According to property 6.6, popping an element (A_r, B_l) from the stack, where the levels of A_r and B_l are equal, will give the same stack as when popping the element (A, B_l). That is, the next element on the stack, following the element (A_r, B_l) will be the same as the next element following (A, B_l). If A is a left child, then (A, B_l) is of case 3, and if A is a right child, then (A, B_l) is of case 4. The next element will therefore be (A, B_r).

In other words: Given an element, $\boldsymbol{e}$, on the stack where $\boldsymbol{e}_A$ is a right child, and $\boldsymbol{e}_B$ is a left child, and $\boldsymbol{e}_A$ has the same level as $\boldsymbol{e}_B$, then the next element on the stack, $\boldsymbol{f}$, has $\boldsymbol{f}_A$ to be the parent of $\boldsymbol{e}_A$ and $\boldsymbol{f}_B$ to be the right child of the parent of $\boldsymbol{e}_B$.

Case 6:

In this case an element $\boldsymbol{e}$ is given, containing a node from BVH-tree $\boldsymbol{A}$ and a node from BVH-tree $\boldsymbol{B}$. The node from $\boldsymbol{A}$ a left child and the node from $\boldsymbol{B}$ is a right child. The level of $\boldsymbol{e}_A$ is one less than the level of $\boldsymbol{e}_B$.
Let the parent of the node in $\boldsymbol{e}$ from $\boldsymbol{A}$ be the node A, and let the parent of the node from $\boldsymbol{B}$ be the node B. That is, $\boldsymbol{e}$ contains the nodes A_l and B_r, see figure 14.

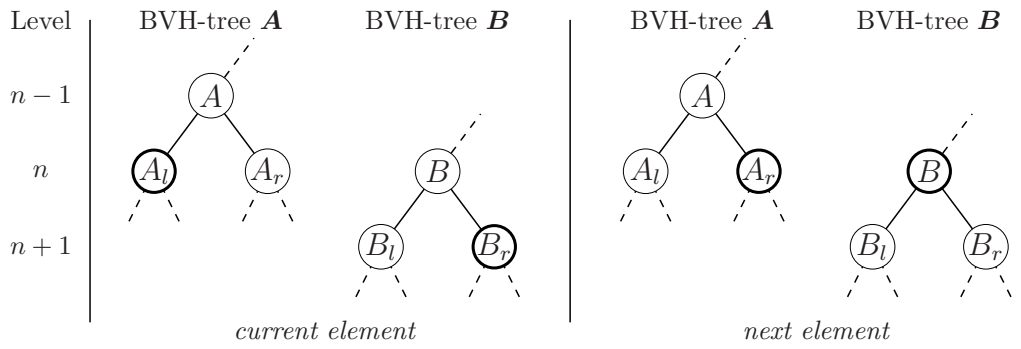


Figure 14: **Case 6**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_l and B_r (thick nodes in current element), and the level of A_l is one less than the level of B_r, then the next element on the stack - the element below $\boldsymbol{e}$ on the stack - will contain the two nodes A_r and B (thick nodes in next element).

According to property 6.5, popping an element (A_l, B_r) where the level of A_l is one less than the level of B_r, will give the same stack as when the element (A_l, B) was popped. That is, the next element on the stack, following the element (A_l, B_r), will be the same as the next element following (A_l, B). This was found in case 1 and case 2 to be (A_l, B).
In other words: Given an element, $\boldsymbol{e}$, on the stack where $\boldsymbol{e}_A$ is a left child, and $\boldsymbol{e}_B$ is a right child, and $\boldsymbol{e}_A$ has a level of one less as the level of $\boldsymbol{e}_B$, then the next element on the stack $\boldsymbol{f}$ has $\boldsymbol{f}_A$ to be the right child of the parent of $\boldsymbol{e}_A$ and $\boldsymbol{f}_B$ to be the parent of $\boldsymbol{e}_B$.

Case 7:

In this case an element $\boldsymbol{e}$ is given, containing a node from BVH-tree $\boldsymbol{A}$ and a node from BVH-tree $\boldsymbol{B}$. The node from $\boldsymbol{A}$ and the node from $\boldsymbol{B}$ are both right children. The level of $\boldsymbol{e}_A$ is the same as the level of $\boldsymbol{e}_B$.
Let the parent of the node in $\boldsymbol{e}$ from $\boldsymbol{A}$ be the node A, and let the parent of the node from $\boldsymbol{B}$ be the node B. That is, $\boldsymbol{e}$ contains the nodes A_r and B_r, see figure 15.

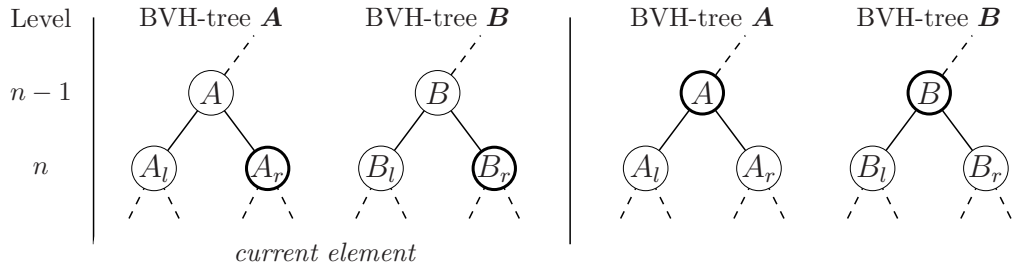


Figure 15: **Case 7**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_r and B_r (thick nodes in current element), then the next element on the stack - the element below $\boldsymbol{e}$ on the stack - will depend on the relations between the two nodes A and B.

According to property 6.7, popping an element (A_r, B_r) where the level of A_r is the same as the level of B_r, will give the same stack as when the element (A, B) was popped. To decide which element was on top of the stack after the element (A, B) was popped, depends on the relation between the A and B. According to property 6.8, there are only eight possible relations between two nodes in an element on the stack. The relation between the nodes A and B must therefore be one of these eight relations. Since A_r and B_r have the same level, A and B must also have the same level. That is, there are only four possible relations between the nodes A and B:

- A and B are both left children.
- A is a left child and B is a right child.
- A is a right child and B is a left child.
- A and B are both right children.

If both A and B are left children, then the relation between A and B must be of case 1 as previously shown. The next element on the stack will in this case be an element containing the right child to the parent of A, and B (see

figure 16, where the x in Level on the left is equal to 1).
If A is a left child, and B is a right child, then the relation between A and B must be of case 2. The next element on the stack will in this case be an element containing the right child to the parent of A, and B.
If A is a right child, and B is a left child, then the relation between A and B must be of case 5. The next element on the stack will in this case be an element containing the parent of A, and the right child to the parent of B.
If both A and B are right children, then the stack will look like it did when the element containing their parents was popped from the stack. As can be seen, this can go on as long as the examined element contain nodes which are both right children. Since the root nodes by definition are left children, this can at most go on until the root nodes are reached.

When two nodes in an element $\boldsymbol{g}$ are reached which are not both right children, the next element on the stack can be found to be the same element as was on the top of the stack when $\boldsymbol{g}$ was popped. Since it is known that the nodes in $\boldsymbol{g}$ have the same level, and the nodes can not both be right children, then the only possible relations between the nodes in $\boldsymbol{g}$ are of case 1, case 2 and case 5.

To summarize this: Given an element $\boldsymbol{e}$, where $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ are both right children, and where $\boldsymbol{e}_A$ has the same level as $\boldsymbol{e}_B$, then the next element, $\boldsymbol{f}$, on the stack will be dependent on the relation between the nodes in $\boldsymbol{g}$, where $\boldsymbol{g}_A$ and $\boldsymbol{g}_B$ are the first ancestors from $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ which have the same level and which are not both right children. See figure 16.

Case 8:

In this case an element, $\boldsymbol{e}$, is given, containing a node from BVH-tree $\boldsymbol{A}$ and a node from BVH-tree $\boldsymbol{B}$. The node from $\boldsymbol{A}$ and the node from $\boldsymbol{B}$ are both right children. The level of $\boldsymbol{e}_A$ is one less than the level of $\boldsymbol{e}_B$.
Let the parent of the node in $\boldsymbol{e}$ from $\boldsymbol{A}$ be the node A, and let the parent of the node from $\boldsymbol{B}$ be the node B. That is, $\boldsymbol{e}$ contains the nodes A_r and B_r, $\boldsymbol{e}$=(A_r, B_r).
According to property 6.5, the next element on the stack - the element below $\boldsymbol{e}$ - will be the same as when the element $\boldsymbol{f}$=(A_r, B) was popped.
If B is a left child, then $\boldsymbol{f}$ is of case 5, and the next element on the stack can be found using that case. If B is a right child, then $\boldsymbol{f}$ is of case 7, and the next element on the stack can be found using that case.

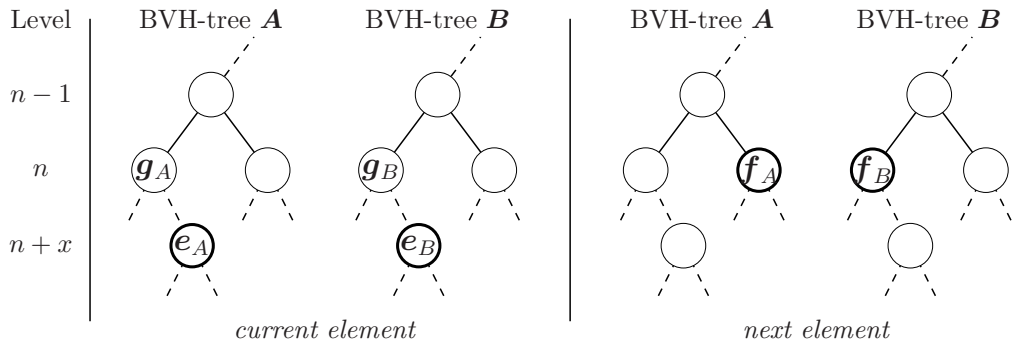


Figure 16: **Case 7.1**: Given an element $\boldsymbol{e}$ on the stack, then the way to find the next element, $\boldsymbol{f}$, on the stack depends on the relation between the nodes in the element $\boldsymbol{g}$. The nodes in $\boldsymbol{g}$ are ancestors of the nodes in $\boldsymbol{e}$, and the paths from $\boldsymbol{g}_A$ to $\boldsymbol{e}_A$ and from $\boldsymbol{g}_B$ to $\boldsymbol{e}_B$ contain $x-1$ nodes each, all of which are right children. In this case where $\boldsymbol{g}_A$ and $\boldsymbol{g}_B$ are both left children, the next element on the stack will be $\boldsymbol{f}_A$ which is the right child to the parent of $\boldsymbol{g}_A$ and $\boldsymbol{f}_B$ which is the same as $\boldsymbol{g}_B$, as previously shown in case 1. Had $\boldsymbol{g}_A$ been a left child and had $\boldsymbol{g}_B$ been a right child, the next element would be found using case 2, and had $\boldsymbol{g}_A$ been a right child and had $\boldsymbol{g}_B$ been a left child, the next element would be found using case 5.

7 The stackless algorithm

In each of the eight cases from the previous section, it was shown that, given an element from the top of the stack, it is possible to deduce what the next element on the stack is. It is therefore possible to jump straight to the next element without the use of a stack. In this section the stackless algorithm will be created, and each of the eight cases from the previous section will get an equal case in the stackless algorithm.
The idea behind the stackless algorithm is that the two BVH-trees will be represented in arrays. This is possible since the BVH is just a collection of nodes. These nodes could just as well be in an array, see figure 17.

Instead of using the stack, indices into the arrays will be used. That is, instead of popping elements containing nodes from the stack as the stack-using algorithm does, the stackless algorithm will use the knowledge about the relations between the current nodes, and use this to find indices to the next element to examine.

This section is structured in the following way: First, the eight cases from the previous section will be rewritten into sub-algorithms using indices, and afterwards the complete stackless algorithm will be presented.

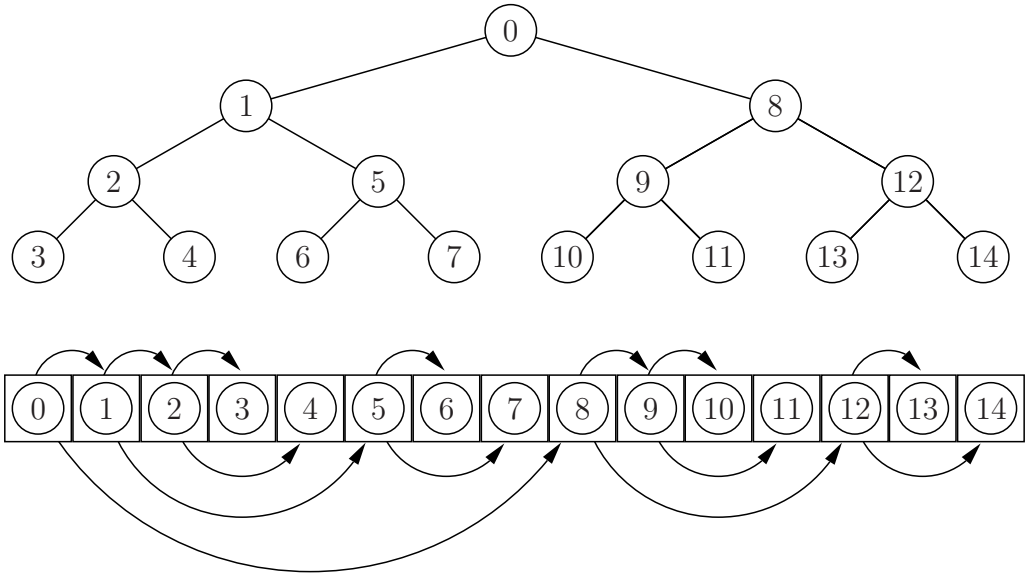


Figure 17: A pre-indexed BVH represented on the top as a tree, and below as an array. The edges in the tree are represented as curved arrows in the array, where the edge from a parent to a left child is the curved arrow above the array, and the edge from a parent to a right child is the curved arrow below the array.

7.1 Rewriting the eight cases

In the remainder of this section some definitions will be used. Since the BVH-trees will be represented in arrays, each BV-node in these arrays will have an index. Some functions will be used:

Level(index): will return the level-value for the node at the given index.

LeftChild(index): will return true if the node at the given index is a left child.

RightChild(index): will return true if the node at the given index is a right child.

Parent(index): will return the index to the parent of the node at the given index.

Escape(index): will return the escape index for the node at the given index. The escape index works as described in section 3.

Case 1 and 2:

Both cases 1 and 2 give the same result, and the only difference between these two cases is whether the node B is a left or a right child.

Therefore given an element, $\boldsymbol{e}$, where the levels of $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ are equal, and $\boldsymbol{e}_A$ is a left child. Then the next element, $\boldsymbol{f}$, on the stack will have $\boldsymbol{f}_A$ to be the right child to the parent of $\boldsymbol{e}_A$, while $\boldsymbol{f}_B$ will remain the same as $\boldsymbol{e}_B$.

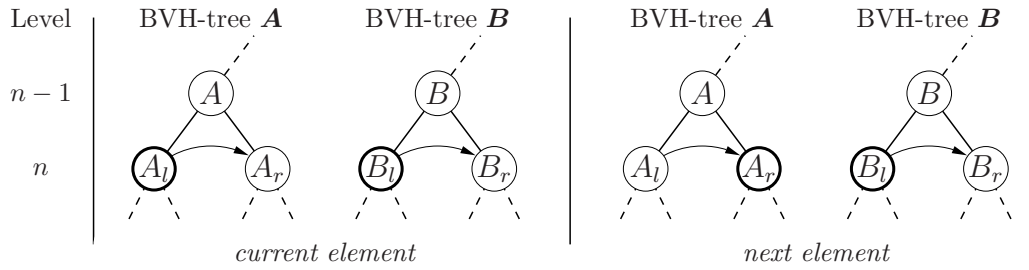


Figure 18: **Case 1**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_l and B_l (thick nodes in current element), and A_l has the same level as B_l, then the next element, $\boldsymbol{f}$, on the stack has $\boldsymbol{f}_A$ to be the node pointed to by the escape index (the curved arrows) for A_l, and $\boldsymbol{f}_B$ to be B_l (thick node in next element). In **Case 2**, the current element would be $\boldsymbol{e}$=(A_l, B_r) and the next element would be $\boldsymbol{f}$=(A_r, B_r).

According to definition 3.1, the escape index of a left child will point to the parent's right child, see figure 18. This is exactly what is needed, and the algorithm for this case will then look like this (notice the use of the indices α and β instead of the nodes A and B):

---

if Level(α) = Level(β) and LeftChild(α) **then**
  $\alpha \leftarrow$ Escape(α)
end if

Cases 3 and 4:

Both case 3 and case 4 give the same result, and the only difference between these two cases is whether the node from BVH-tree $\boldsymbol{A}$ is a left or a right child.
Given an element, $\boldsymbol{e}$, where the level of $\boldsymbol{e}_A$ is one less than the level of $\boldsymbol{e}_B$, and $\boldsymbol{e}_B$ is a left child of its parent, then the next element, $\boldsymbol{f}$, on the stack has $\boldsymbol{f}_B$ to be the right child to $\boldsymbol{e}_B$'s parent, while $\boldsymbol{f}_A$ will remain the same as $\boldsymbol{e}_A$.

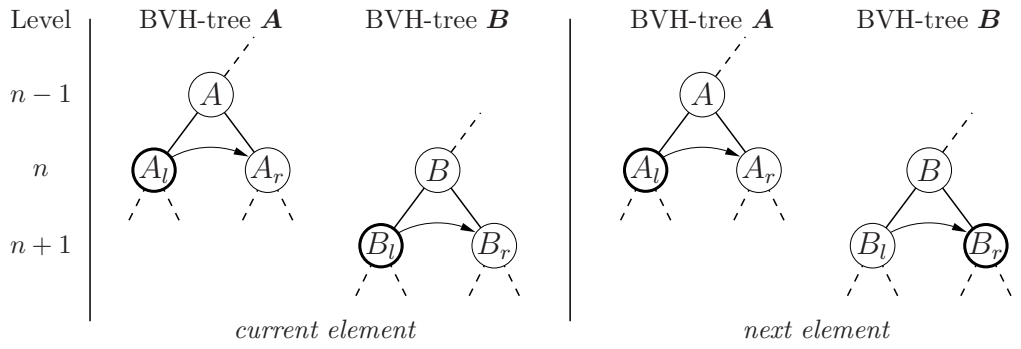


Figure 19: **Case 3**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_l and B_l (thick nodes in current element), and if the level of A_l is one less than the level of B_l, then the next element, $\boldsymbol{f}$, on the stack (thick nodes in next element), has $\boldsymbol{f}_A$ to be A_l, and $\boldsymbol{f}_B$ to be the node pointed to by the escape index (the curved lines) for B_l. In **Case 4**, the current element would be $\boldsymbol{e}= (A_r, B_l)$ and the next element would be $\boldsymbol{f}= (A_r, B_r)$.

Using definition 3.1, the escape index of $\boldsymbol{e}_B$ will point to exactly the right child of the parent of $\boldsymbol{e}_B$, see figure 19. The algorithm for these cases will then look like this:

---

if Level(α) = (Level(β) − 1) and LeftChild(β) **then**
  $\beta \leftarrow$ Escape(β)
end if

Case 5:

Given an element, $\boldsymbol{e}$, where the levels of $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ are equal, and $\boldsymbol{e}_A$ is a right child and $\boldsymbol{e}_B$ is a left child, the next element, $\boldsymbol{f}$, has $\boldsymbol{f}_A$ to be the parent of $\boldsymbol{e}_A$, and $\boldsymbol{f}_B$ to be the right child to the parent of $\boldsymbol{e}_B$, which is what the escape index of $\boldsymbol{e}_B$ points to, see figure 20.

This gives the following algorithm:

---

if Level(α) = Level(β) and RightChild(α) and LeftChild(β) **then**
  $\alpha \leftarrow$ Parent(α)
  $\beta \leftarrow$ Escape(β)
end if

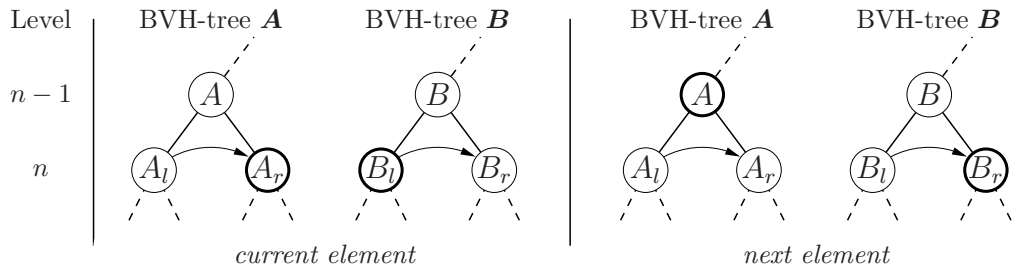


Figure 20: **Case 5**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_r and B_l (thick nodes in current element), where A_r and B_l have the same level, then the next element, $\boldsymbol{f}$, on the stack (thick nodes in next element), has $\boldsymbol{f}_A$ to be the parent of A_r, and $\boldsymbol{f}_B$ to be the node pointed to by the escape index (curved arrows) for B_l.

Case 6:

Given an element, $\boldsymbol{e}$, where the level of $\boldsymbol{e}_A$ is one less than the level of $\boldsymbol{e}_B$, and $\boldsymbol{e}_A$ is a left child, and $\boldsymbol{e}_B$ is a right child, then the next element, $\boldsymbol{f}$, has $\boldsymbol{f}_A$ to be the right child to the parent of $\boldsymbol{e}_A$, which is what the escape index of A points to, and $\boldsymbol{f}_B$ will be the parent of $\boldsymbol{e}_B$, see figure 21. This gives the following algorithm:

---

if Level(α) = Level((β) − 1) and LeftChild(α) and RightChild(β) **then**
  $\alpha \leftarrow$ Escape(α)
  $\beta \leftarrow$ Parent(β)
end if

Case 7:

Assume an element, $\boldsymbol{e}$, where the levels of $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ are equal, and both $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ are right children.
As was shown in case 7 in section 6, we need to look recursively at the ancestors of $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ until we reach a set of ancestors which are not both right children. That is, we will have to follow $\boldsymbol{e}_A$'s and $\boldsymbol{e}_B$'s parents upward in the trees, until a node in either BVH-tree $\boldsymbol{A}$ or BVH-tree $\boldsymbol{B}$ or both is found which is not a right child, and where the nodes have the same level.
For this purpose a new value is added to each node in the tree, a *Right Child Level* (RCL). This value expresses how many straight right children there are upwards from a node, including the node itself. That is, a left node will have RCL-value 0, a right child node with a left child parent will have an RCL-value 1. A tree with RCL-values can be seen in figure 22.

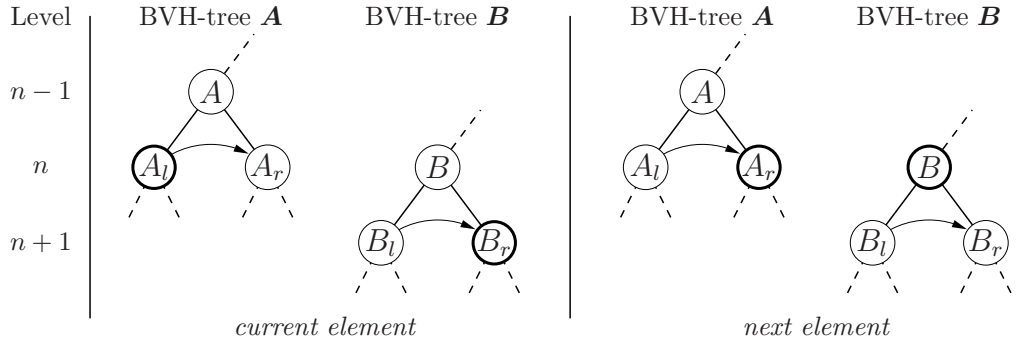


Figure 21: **Case 6**: If a current element, $\boldsymbol{e}$, on the stack contains the two nodes A_l and B_r (thick nodes in current element), and the level of A_l is one less than the level of B_r, then the next element, $\boldsymbol{f}$, on the stack (thick nodes in next element) has $\boldsymbol{f}_A$ to be the node pointed to by the escape index (the curved arrows) for A_l and $\boldsymbol{f}_B$ to be the parent of B_r.

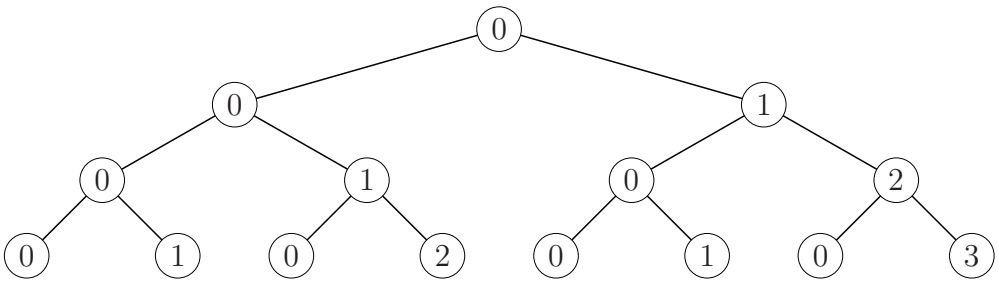


Figure 22: The *Right Child Level*s of nodes in a tree. The RCL-values indicate how many straight right children there are above the node. Note that according to defintion 3.3 the root is considered a left child.

Given the RCL-value, it is known how far upward in the trees from both $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ a left child is found. Then, when deciding wheter $\boldsymbol{e}_A$ or $\boldsymbol{e}_B$ will reach a left child node first, when travelling upwards in the tree, can be done by comparing RCL-values. For this purpose, let us create a function *RCL(index)* which returns the RCL-value for the given index.
Having the RCL-value is not enough. It only tells us what should be done according to the case lying at some length upwards in the tree, and a function *LevelUp(index, R)* must be created.
The *LevelUp(index, R)*-function, given an index to a node and a value R, finds the index to the node lying R nodes upwards in the tree, assuming that all nodes between the current node and the resulting node are all connected right children. Assuming the preindexed tree in 17, a *LevelUp(index, R)*-function could be found for this specific tree/array. But if the tree was indexed in some other way, another function would have to be used. The *LevelUp(index, R)*-function can therefor vary depending on how the BV-nodes are arranged in the array.

Given the RCL and the *LevelUp*-function, the next element, $\boldsymbol{g}$, can be found in the following way:
Let $x_A = RCL(\boldsymbol{e}_A)$, and $x_B = RCL(\boldsymbol{e}_B)$. If x_A is less than or equal to x_B, then a left child node is reached in BVH-tree $\boldsymbol{A}$ before or at the same time as in BVH-tree $\boldsymbol{B}$. In this case, it is known that at x_A levels up in $\boldsymbol{A}$, there will be a node which is a left child. We will therefore have to examine the element $\boldsymbol{f}$, where $\boldsymbol{f}_A$ is found by using the function *LevelUp($\boldsymbol{e}_A$, x_A)*, and where $\boldsymbol{f}_B$ is found by using the function *LevelUp($\boldsymbol{e}_B$, x_A)*. Note that both functions get the nodes placed x_A levels up.
Since the level of $\boldsymbol{e}_A$ is equal to the level of $\boldsymbol{e}_B$, the level of $\boldsymbol{f}_A$ must equal the level of $\boldsymbol{f}_B$. And since $\boldsymbol{f}_A$ is a left child, $\boldsymbol{f}$ must be of either case 1 or 2. $\boldsymbol{g}_A$ will then be the right child to the parent of $\boldsymbol{f}_A$, and according to definition 3.2 this will be what the escape index of $\boldsymbol{e}_A$ points to. $\boldsymbol{g}_B$ will be the same as $\boldsymbol{f}_B$.

If x_A is greater than x_B, then a left child node is reached first in $\boldsymbol{B}$.We will therefore have to examine $\boldsymbol{f}$, where $\boldsymbol{f}_A$ is found by using the function *LevelUp($\boldsymbol{e}_A$, x_B)*, and where $\boldsymbol{f}_B$ is found by using the function *LevelUp($\boldsymbol{e}_B$, x_B)*. $\boldsymbol{f}_A$ will therfore be a right child, $\boldsymbol{f}_B$ will be a left child, and $\boldsymbol{f}_A$ and $\boldsymbol{f}_B$ will have the same level. This is the same as in case 5. $\boldsymbol{g}_A$ will therefore be the parent to $\boldsymbol{f}_A$, and $\boldsymbol{g}_B$ will be the right child to the parent of $\boldsymbol{f}_B$, which is what the escape index of $\boldsymbol{e}_B$ points to. This gives the following algorithm:

```
if Level(α) = Level(β) and RightChild(α) and RightChild(β) then
  if RCL(α) ≤ RCL(β) then
    α ← Escape(α)
    β ← LevelUp(β, RCL(α))
  else
    α ← LevelUp(α, RCL(β)+1)
    β ← Escape(β)
  end if
end if
```

Case 8:

Assume an element $\boldsymbol{e}$, where the level of $\boldsymbol{e}_A$ is one less than the level of $\boldsymbol{e}_B$, and both $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$ are right children.
The only difference between this case and the previous case, is that the level of $\boldsymbol{e}_A$ is one less than the level of $\boldsymbol{e}_B$. Thus by moving $\boldsymbol{e}_B$ up to level with $\boldsymbol{e}_A$, this case will be the same as the above.
Since $\boldsymbol{e}_B$ is a right child, it must have an RCL-value greater than or equal to 1, and the RCL-value of $\boldsymbol{e}_B - 1$ must be equal to the RCL-value of the parent of $\boldsymbol{e}_B$. Then, by using the RCL-value of the parent of $\boldsymbol{e}_B$, the above case can be used.
This gives the following algorithm:

```
if Level(α) = Level(β) − 1 and RightChild(α) and RightChild(β) then
  if RCL(α) ≤ RCL(β)−1 then
    α ← Escape(α)
    β ← LevelUp(β, RCL(α)+1)
  else
    α ← LevelUp(α, RCL(β))
    β ← Escape(β)
  end if
end if
```

7.2 Creating the stackless algorithm

In the previous subsection, all eight cases of possible relations between the nodes in an element on the stack were rewritten into stackless sub-algorithms. Here these will be collected into the complete stackless algorithm.

Pseudocode for the algorithm can be seen in algorithm 4, and will be described in the following:

Algorithm 4 *The stackless algorithm*, for full binary trees of equal size.

```
1: while α and β points to an element in their array do
2:   if Collision(α, β) then
3:     if Leaf(α) and Leaf(β) then
4:       FindIntersection(α, β)
5:     else
6:       if Level(α) < Level(β) then
7:         α ← GetLeftChild(α)
8:       else
9:         β ← GetLeftChild(β)
10:      end if
11:      next
12:    end if
13:  end if
14:  if Level(α) = Level(β) then
15:    if LeftChild(α) then {Case 1 and 2:}
16:      α ← Escape(α)
17:      next
18:    end if
19:    if RightChild(α) and LeftChild(β) then {Case 5:}
20:      α ← Parent(α)
21:      β ← Escape(β)
22:      next
23:    end if
24:    if RightChild(α) and RightChild(β) then {Case 7:}
25:      if RCL(α) ≤ RCL(β) then
26:        α ← Escape(α)
27:        β ← LevelUp(β, RCL(α))
28:      else
29:        α ← LevelUp(α, RCL(β)+1)
30:        β ← Escape(β)
31:      end if
32:      next
33:    end if
34:  else
35:    if LeftChild(β) then {Case 3 and 4:}
36:      β ← Escape(β)
37:      next
38:    end if
39:    if LeftChild(α) then {Case 6:}
40:      α ← Escape(α)
41:      β ← Parent(β)
42:      next
43:    end if
44:    if RightChild(α) and RightChild(β) then {Case 8:}
45:      if RCL(α) ≤ RCL(β)−1 then
46:        α ← Escape(α)
47:        β ← LevelUp(β, RCL(α)+1)
48:      else
49:        α ← LevelUp(α, RCL(β))
50:        β ← Escape(β)
51:      end if
52:      next
53:    end if
54:  end if
55: end while
```

First, in line 2, is the check for collision between the BV's of the nodes pointed to by α and β. If the nodes are leaves, the intersection is found, but the next two nodes to check have to be found, and the algorithm skips to line 14. If none of the nodes are leaves, then either α will be checked against β's left child, or α's left child will be checked against β. This is what happens in lines 6 to 10, and it corresponds to pushing an elemnt on the stack, and at the beginning of the next iteration, immediately popping it again.
Next, in line 14, if there was no collision, or there was collision between two leaves, then the next two nodes to check will have to be found, by using the eight possible cases.

The eight cases are grouped into two major sections. One where the level of the node at α is equal to the level of the node at β, and one where the level of the node at α is one less than the level of the node at β.
When the next two nodes are found, the algorithm skips to the next iteration. This will continue until one or both of the indices has the value of a root's escape index.

8 Extending the stackless algorithm

So far, the BVH-trees have been limited to binary trees, where all parents have two children, and where all the leaves in both trees lie on the same level. Trees like that are rarely experienced in real world collision detection. This section will therefore extend the stackless algorithm from the previous section, in order to make it able to handle collision detection between unbalanced trees of unequal height.

There are two situations to consider when extending the algorithm.

- A tree of depth n must be tested against a tree of depth m where $n \neq m$.
- A parent-node must be able to have from 1 to n children.

These two cases will be discussed in the rest of this section.

8.1 Comparing trees of different sizes

When comparing trees of different hights, or unbalanced trees, situations can occur where the alternately stepping traversal can not traverse down one of the BVH-trees because the algorithm has reached a leaf in one tree, but not in the other. This happens in two different cases:

- A node in BVH-tree $\boldsymbol{A}$ with level n, must be compared to a leaf-node in BVH-tree $\boldsymbol{B}$ with level m, where $n > m$.
- A leaf-node in BVH-tree $\boldsymbol{A}$ with level n, must be compared to a node in BVH-tree $\boldsymbol{B}$ with level m, where $m > n + 1$.

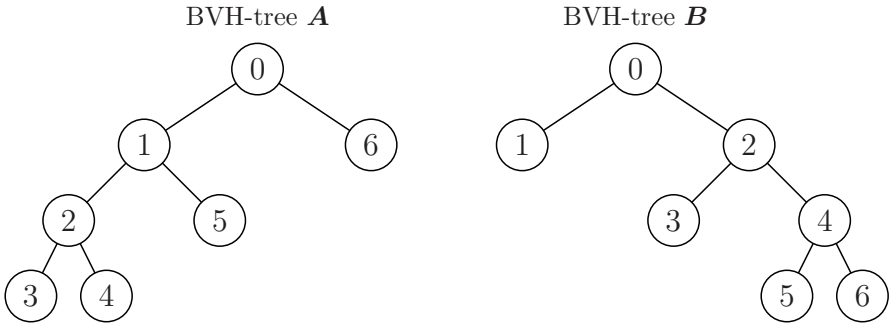


Figure 23: Assume a collision between the nodes A_1 and B_1, or between the nodes A_6 and B_4. Due to the alternately stepping traversal, the stackless algorithm can not reach the next nodes to compare. The stackless algorithm will have to be extendend to handle these kinds of situations. One solution is to extend the algorithm in these cases with the leaf algorithm (algorithm 1) from section 3.

In both cases, the problem can be solved be using the leaf-algorithm from section 3. In the first case, some leaf-node β in $\boldsymbol{B}$ must be compared to the subtree in $\boldsymbol{A}$ with root at some node, α.
In the second case, some leaf-node, α, in $\boldsymbol{A}$, must be compared to the subtree in $\boldsymbol{B}$ with root at some node β. This is exactly what the leaf-algorithm does, and in both these cases, the leaf algorithm can be inserted into the stackless algorithm.

8.2 Parent-nodes with n children

To allow for more or less than two children to a node, two different cases must be examined:

- Just one child.
- More than two children.

First of all, it should be noted that in the algorithm created in section 7, a node, which is not a leaf, only needs to know of three other nodes. Its left child, its parent, and its escape index. A leaf-node only needs to know its parent and its escape-index.

In the case where a node only has a single child, should this child then be a left or a right child? If the child were considered a left child, then definition 3.2, saying that the escape index to a left child is its parents right child, will not be possible to uphold. If the child on the other hand were considered a right child, then definiton 3.2 trivially holds. But the parent still needs to know which child is the left child. In stead of letting the parent have an index to its left child, it could have an index to its *left-most child*. This can still be the right child, which will trivially be the leftmost child since it is a single child.

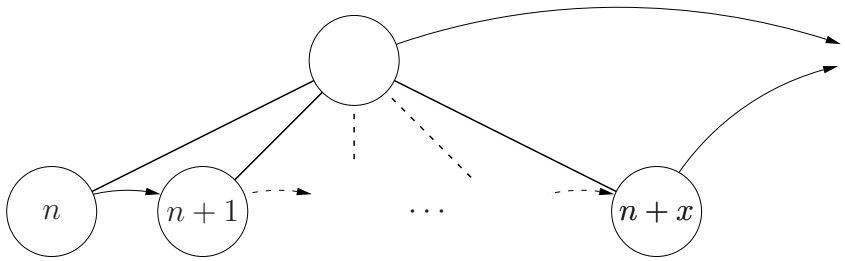


Figure 24: A parent node with x children. The child n is the left-most child to the parent, and the child $n+x$ is the right child. All the children from n to $n+(x-1)$ are considered left children. Each child only needs to know the next child, by the escape index, and the parent only needs to know the left-most child, n.

In the case where a node has more than two children, it will be convinient if the parent still only needs to know its left-most child. Since the escape index for a left child points to the parent's right child, this can be used to add more children to the parent. Instead of letting the escape index of the left child point to the right child of the parent, it could be pointing to the next child of the parent, see figure 24. But should all the intermediate children then become left children, right children or some kind of middle children? If we designated all the intermediate children as left children, then nothing would have to be changed in the algorithm. Only definition 3.1 would have to be changed into the following definition:

Definition 8.1. *The escape index of a left child is the next child of the parent.*

Since all the intermediate children are left children, they will get the RCL-value, 0, which makes sense since this value is only needed in relation to the right child. The algorithm can therefore be used as is with more than two children.

8.3 Post-indexed stackless algorithm

With the previous found extensions, the stackless algorithm is now able to handle collisions between non-binary unbalanced trees of different heights. But in section 7, when the stackless algorithm was created, a function called *LevelUp(N, x)* was introduced. This function found the ancestor node, N, at x levels up in the BVH, given all nodes between the current node and the ancestor was all right-children. Given the added possibilites to handle unbalanced trees, this function has become slightly inefficient, because it has to iterate up through the tree, one node at a time (see figure 25). It is therefore desirable to create a more efficient function.

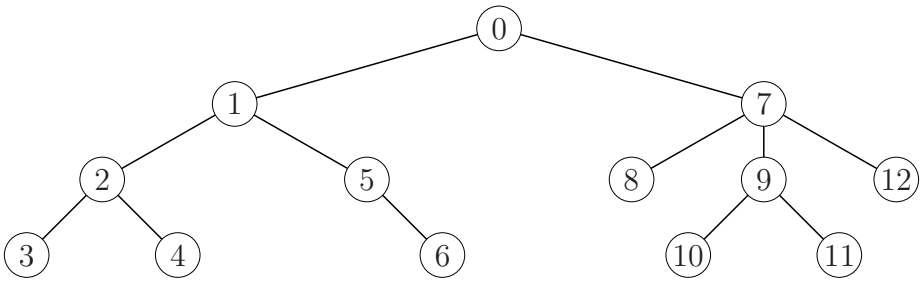


Figure 25: Given a unbalanced preindexed tree, all the left-most children know the index of their parent, since it is the same as their own minus one. The right-most children, on the other hand, has no possible way of deduce what index their parent has, since that depends on the number of nodes in the subtrees spanned by the parents left children.

Given the way the nodes are numbered by pre-indexing, it is not possible to know which index the parent of a given node has if the node is a right child. Figure 25 shows an unbalanced tree with pre-indexed nodes. In the figure it is seen, how all the left-most children know the index of their parent - since it is just the same as their own minus one, and their grand parent is their own index minus two. If the tree was post-indexed instead of pre-indexed, the right-children would be able to find their parents and grand parents in a similar manner, see figure 26.

The *LevelUp(N, x)*-function now only has to subtract x from the current index to get the index of the ancestors node.

There is a drawback to this technique. If the nodes are packed in an array, in a forward manner, then there will be a lot of backwards travelling in the array given the reversed node-indices. In the worst case, this may give cache misses on some architectures, but on architectures where there is no cache, or a very small cache, this ought to be faster than the pre-indexed version, given the faster *LevelUp(N, x)*-function.

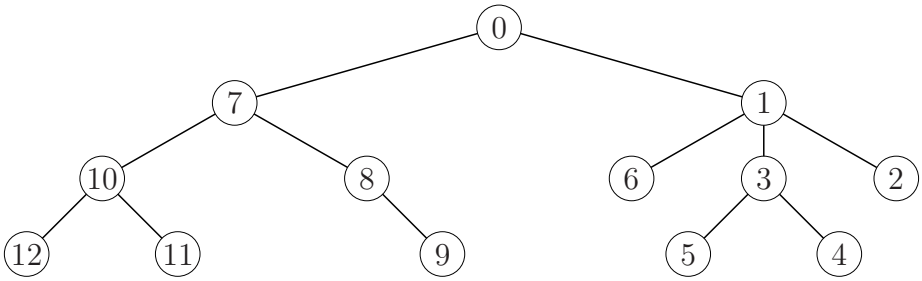


Figure 26: An unbalanced post-indexed tree. All the right children can find the index to their ancestores - as long as all nodes between the right child and the ancestor are right children - just by subtracting the number of levels up to the ancestor from the index of node itself.

9 Test

In order to find out how well the stackless algorithm performs, compared to the dynamic stack algorithm, some tests will be performed.
The following algorithms will be tested:

- The dynamic stack algorithm
- The alternately stepping stack algorithm
- The stackless algorithm
- The post-indexed stackless algorithm
- The leaf algorithm

For all tests, the types of bounding volumes used are Axis-Aligned Bounding Boxes. The BVHs are created in a top-down manner. The object gets a top-BV. As long a BV does not contain a single primitive - in this case triangular faces - the BV is split into two new BVs, each containing half the number of primitives of the parent BV. The number of primitives are divided on the middle along the longest coordinate axis.

All tests were done on a Lenovo 3000 N100 laptop with an Intel Centrino Duo processor running at 2 GHz with 2 MB level 2 cache, and with 1 GB ram. The computer was running Ubuntu Linux 6.04. All programs were compiled using gcc with the -O2 flag. For timing of functions, boost-timer was used. All times presented in the remainder of this section are given in seconds.

It is assumed that in most cases, the dynamic stack algorithm will be the fastest, given its dynamic nature. The alternately stepping stack algorithm is supposedly not as fast as the stackless algorithm, given the cost of handling the stack. The leaf algorithm is used for comparison purposes, and is assumed to be the slowest in almost every case.

There are a number of setups that must be tested to give an indication of how well the two stackless algorithms perform against the dynamic stack algorithm. In each of these cases, the alternately stepping stack algorithm and the leaf algorithm are only used for comparison, and the main focus is on how well the stackless and the dynamic algorithms compare.
The test setups are the following:

- Intersection: How do the algorithms perform when two objects collide?
- Close proximity: How do the algorithms perform when the BVs of the BVHs have many collisions, but the objects do not collide.
- Small proximity: How do the algorithms perform when the BVs of the BVHs have few collisions, and the objects do not collide.
- Separation: How do the algorithms perform when the objects and their BVHs do not collide?
- Robustness: Do the algorithms get the same results?
- Scalability: Do the algorithms perform equally well, regardless of how many primitives are in the objects?
- How do the size of the objects influence on the performance of the algorithms?
- How do the difference in the number of primitives in the objects influence on the performance of the algorithms.

All of these cases will be tested in the remainder of this section. Robustness will be tested as part of the all the other tests. In each of the tables showing the results, there are two columns labeled *leaf overlaps* and *geometry overlaps*. The leaf overlaps states how many overlaps there were found between leaves in the BVHs. Geometry overlap states how many overlaps were found on the primitives. The number of geometry overlaps should always be less than the number of leaf overlaps.

9.1 Intersection, separation, close proximity and small proximity

In each of the intersection, separation, small proximity and close proximity tests, three different setups will be used. The three different setups are used to test the robustness of the algorithms.

In each of the three setups, the five algorithms will be tested on the same setup. Each algorithm will be repeated a number of times, where a timer will run for the duration of the number of repetitions. The number of repetitions is stated for each setup. To make sure that the time is valid, the entire number of repetitions will be done 100 times over, and the minimum, maximum and mean-time together with standard deviation will be found. The number of iterations in each algorithm's loop and the number of overlap test will also be stated.

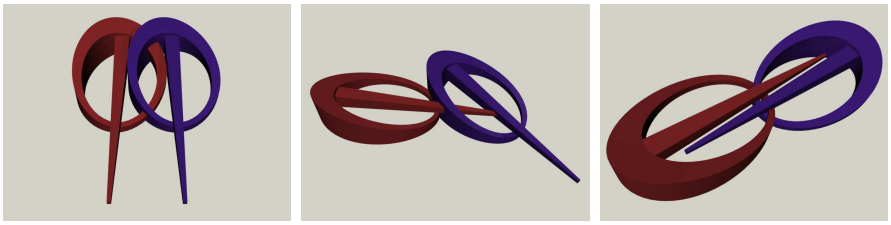


Figure 27: Images for the three different intersection setups. In each of the setups, the two objects intersects at least in one place.

For intersections, the images for the three different setups can be seen in figure 27 and the results are shown in table 2. In the intersection tests, two identical objects are used. The objects have 796 faces each, and the two object are intersecting. The purpose here is to see how well the algorithms perform when intersection occurs.

Algorithm	min.	max.	mean	standard deviation	Iterations	overlaps	leaf overlaps	geometry overlaps
Intersection test 1 (1000 repetitions):								
stack	1.66	1.72	1.67580	0.00839	5029	3382	868	106
alternately	1.97	2	1.98500	0.00557	11807	6771	868	106
stackless	1.96	1.99	1.97740	0.00541	11807	6771	868	106
stackless (post)	1.95	1.97	1.96020	0.00583	11807	6771	868	106
stackless (leaf)	1.85	1.87	1.86080	0.00483	13968	7454	868	106
Intersection test 2 (1000 repetitions):								
stack	5.8	5.86	5.82000	0.00927	10331	8426	3261	160
alternately	5.87	5.9	5.88300	0.00794	12407	9464	3261	160
stackless	5.85	5.88	5.86290	0.00752	12407	9464	3261	160
stackless (post)	5.85	5.89	5.86900	0.00806	12407	9464	3261	160
stackless (leaf)	5.82	5.9	5.83530	0.01044	15724	10725	3261	160
Intersection test 3 (1000 repetitions):								
stack	1.65	1.68	1.66180	0.00590	6209	3913	809	116
alternately	1.88	1.9	1.89020	0.00583	11263	6440	809	116
stackless	1.86	1.88	1.87240	0.00634	11263	6440	809	116
stackless (post)	1.87	1.9	1.88520	0.00624	11263	6440	809	116
stackless (leaf)	1.9	1.93	1.90750	0.00712	17138	8980	809	116

Table 2: Intersection tests. It is seen, that the dynamic stack algorithm is the fastest in all three cases, but most significant in the first and third test where the number of iterations for the dynamic stack algorithm is almost half that of the stackless algorithms. It is of interest to note, that in the first test, the leaf algorithm is faster than the stackless and the post-indexed stackless algorithm even though it has more iterations.

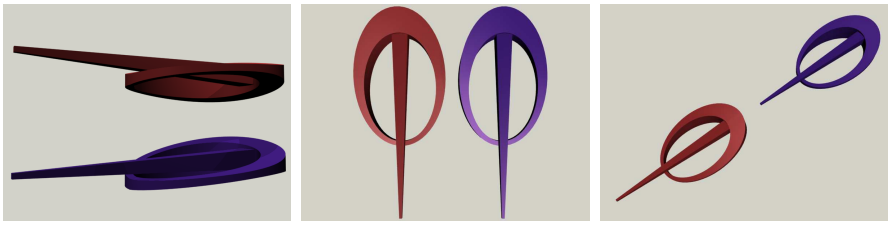


Figure 28: Images for the three different separation setups. In each of the setups, the two objects are seperated by distance insuring their BVHs do not overlap.

For separation, the images for the three different setups can be seen in figure 28 and the results are stated in table 3. In the separation tests, two identical objects are used. The objects have 796 faces each. The purpose here is to see how well the algorithms perform when seperated by a distance making sure that their BVHs do not overlap at all.

Algorithm	min.	max.	mean	standard deviation	Iterations	overlaps	leaf overlaps	geometry overlaps
Separation test 1 (100000 repetitions):								
stack	0.04	0.05	0.04650	0.00477	1	0	0	0
alternately	0.04	0.06	0.04620	0.00506	1	0	0	0
stackless	0	0.02	0.00830	0.00448	1	0	0	0
stackless (post)	0	0.01	0.00710	0.00454	1	0	0	0
stackless (leaf)	3.05	3.09	3.06800	0.00748	796	0	0	0
Separation test 2 (100000 repetitions):								
stack	0.04	0.05	0.04640	0.00480	1	0	0	0
alternately	0.04	0.05	0.04570	0.00495	1	0	0	0
stackless	0	0.01	0.00670	0.00470	1	0	0	0
stackless (post)	0	0.01	0.00670	0.00470	1	0	0	0
stackless (leaf)	2.59	2.62	2.60240	0.00634	796	0	0	0
Separation test 3 (100000 repetitions):								
stack	0.04	0.06	0.04760	0.00492	1	0	0	0
alternately	0.04	0.05	0.04660	0.00474	1	0	0	0
stackless	0	0.01	0.00740	0.00439	1	0	0	0
stackless (post)	0	0.01	0.00750	0.00433	1	0	0	0
stackless (leaf)	3.6	3.62	3.61050	0.00622	796	0	0	0

Table 3: Separation tests. Most noticable in this test is the leaf algorithm. The fact that it has to compare every leaf of one of the trees to the root of the other is what is showing the impact here. Also of interest is that the stackless and post-indexed stackless algorithms perform better than the stack-using algorithms. This is due to the setup time of the stack in each of the 100000 repetitions.

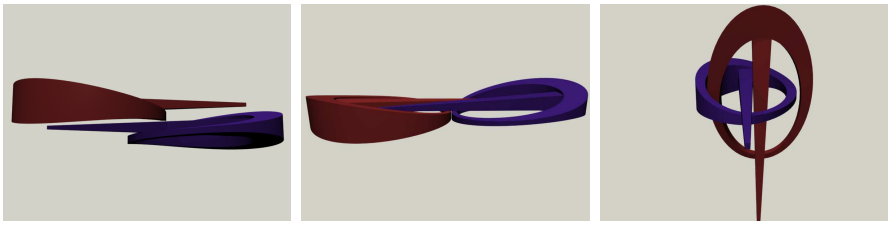


Figure 29: Images for the three different close proximity setups. In each of the setups, the two objects are positioned in way, making sure there is no overlap between the objects, but many overlaps between the BVHs.

For close proximity, the images for the three different setups can be seen in figure 29 and the results are stated in table 4. In the close proximity tests, two identical objects are used. The objects have 796 faces each. The two objects are placed in such a way, that their BVHs have lots of overlap, but without having the objects intersecting.

Algorithm	min.	max.	mean	standard deviation	Iterations	overlaps	leaf overlaps	geometry overlaps
Close proximity test 1 (1000 repetitions):								
stack	0.35	0.37	0.35850	0.00517	1681	1020	180	0
alternately	0.46	0.49	0.47310	0.00560	4135	2247	180	0
stackless	0.45	0.48	0.46150	0.00477	4135	2247	180	0
stackless (post)	0.46	0.49	0.47100	0.00574	4135	2247	180	0
stackless (leaf)	0.6	0.62	0.60500	0.00592	9772	4668	180	0
Close proximity test 2 (1000 repetitions):								
stack	0.98	1.01	0.99430	0.00752	3759	2408	529	0
alternately	1.19	1.21	1.20090	0.00549	8449	4753	529	0
stackless	1.16	1.2	1.17930	0.00667	8449	4753	529	0
stackless (post)	1.19	1.21	1.20030	0.00670	8449	4753	529	0
stackless (leaf)	1.27	1.3	1.28550	0.00669	14734	7498	529	0
Close proximity test 3: (10000 repetitions)								
stack	1.95	1.97	1.95870	0.00462	1851	974	49	0
alternately	1.98	2.01	1.99210	0.00571	2273	1185	49	0
stackless	1.92	1.95	1.93370	0.00611	2273	1185	49	0
stackless (post)	1.94	1.97	1.95680	0.00581	2273	1185	49	0
stackless (leaf)	2.68	2.71	2.68880	0.00725	5428	2365	49	0

Table 4: Close Proximity tests. The dynamic stack algorithm is the fastest in the first two tests, but in the third test it is a little bit slower than the stackless and post-indexed stackless algorithms. Again, it is seen that the pruning capabilities of the dynamic stack algorithm are the best, giving the fewest number of iterations. (please note, that there are 10000 repetitions of the third test and just 1000 repetitions of the first and the second test.)

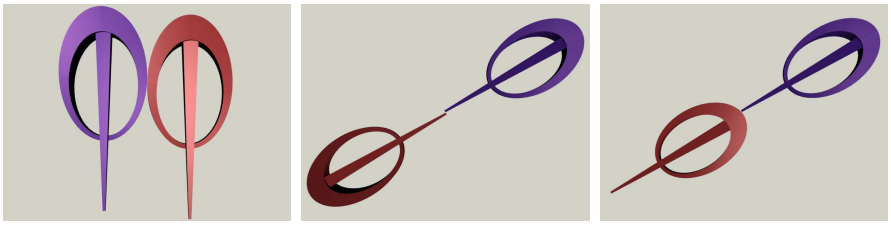


Figure 30: Images for the three different small proximity setups. In each of the setups, the two objects do not intersect, but the objects are positioned in a way, making sure there are a few overlaps between their BVHs.

For small proximity, the images for the three different setups can be seen in figure 30 and the results are stated in table 5. In the small proximity tests, two identical objects are used. The objects have 796 faces each. The two objects are placed, such that their BVHs have a few overlaps, but without having the objects intersecting.

Algorithm	min.	max.	mean	standard deviation	Iterations	overlaps	leaf overlaps	geometry overlaps
Small proximity test 1 (10000 repetitions):								
stack	0.33	0.35	0.33700	0.00574	181	106	16	0
alternately	0.36	0.38	0.36720	0.00618	253	142	16	0
stackless	0.32	0.35	0.33300	0.00500	253	142	16	0
stackless (post)	0.32	0.34	0.33000	0.00469	253	142	16	0
stackless (leaf)	0.55	0.58	0.56160	0.00612	1048	142	16	0
Small proximity test 2 (10000 repetitions):								
stack	3.46	3.49	3.47420	0.00710	917	651	193	0
alternately	3.41	3.44	3.42550	0.00669	933	659	193	0
stackless	3.37	3.4	3.38350	0.00669	933	659	193	0
stackless (post)	3.38	3.41	3.39350	0.00712	933	659	193	0
stackless (leaf)	3.68	3.71	3.69790	0.00752	1966	778	193	0
Small proximity test 3 (10000 repetitions):								
stack	0.76	0.79	0.77300	0.00592	647	351	28	0
alternately	0.64	0.67	0.65240	0.00531	461	258	28	0
stackless	0.61	0.64	0.62140	0.00566	461	258	28	0
stackless (post)	0.62	0.64	0.62430	0.00587	461	258	28	0
stackless (leaf)	0.87	0.89	0.87580	0.00603	1094	177	28	0

Table 5: Small Proximity tests. It is worth noting, that the stackless and post-indexed stackless algorithms are a little bit faster than the dynamic stack algorithm. This is due to the setup time for the stack. In the third test, the better performance is also caused by the fact that the algorithms using alternatly stepping have fewer iterations.

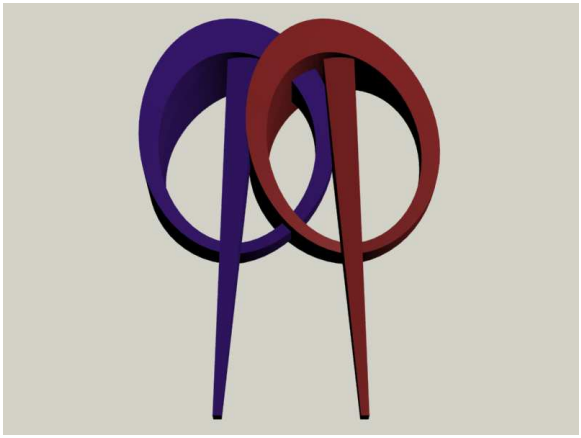


Figure 31: Image for both the scalability test and the test for objects with different number of primitives. The objects are intersecting.

9.2 Scalability

In this test, two identical objects will be placed in a way that ensures they are intersecting. There are five different setups, and in each of the setups, both of the objects will be subdivided a number of times from 0 to 4. An image for the setup can be seen in figure 31. The purpose of this test is to observe how well the algorithms perform when the number of primitives, and hence the number of BVs, are increased The same information will be stated as in the previous tests, and the results are shown in table 6.

9.3 Objects of different sizes

In this test, two identical objects will be tested, where one of the objects will be scaled smaller than the other. The objects will be positioned in such a way, that their BVHs are colliding, but the objects are not. This will test how well the stackless algorithm compares to the dynamic algorithm when the dynamic algorithm can use its volume-test to quickly prune non-colliding BVs based on the volume size.
An image of the setup can be seen in figure 32, and the results are shown in table 7. The same information will be stated as in the previous tests.
In this test, the smaller object is BVH-tree $\boldsymbol{A}$, and the larger object is BVH-tree $\boldsymbol{B}$. The leaf algorithm will therefore compare all its leaves from the smaller object to the BVH of the larger object. This will give the leaf algorithm a disadvantage in this particular case.

Algorithm	min.	max.	mean	standard deviation	Iterations	overlaps	leaf overlaps	geometry overlaps
Sub 0: 796 faces (1000 repetitions):								
stack	2.31	2.34	2.32140	0.00548	6697	4536	1188	201
alternately	2.63	2.66	2.64640	0.00656	13821	8098	1188	201
stackless	2.64	2.66	2.64780	0.00576	13821	8098	1188	201
stackless (post)	2.61	2.63	2.62000	0.00548	13821	8098	1188	201
stackless (leaf)	2.49	2.52	2.49670	0.00633	15940	8760	1188	201
Sub 1: 3184 faces (1000 repetitions):								
stack	6.25	6.27	6.25990	0.00640	26449	16070	2846	417
alternately	9.05	9.08	9.06530	0.00699	86807	46249	2846	417
stackless	8.98	9.01	8.99790	0.00637	86807	46249	2846	417
stackless (post)	8.66	8.69	8.67960	0.00662	86807	46249	2846	417
stackless (leaf)	8.74	8.76	8.74760	0.00634	112326	57417	2846	417
Sub 2: 12736 faces (1000 repetitions):								
stack	14.21	14.25	14.22780	0.00832	72647	42229	5906	843
alternately	33.6	33.62	33.60880	0.00637	488033	249922	5906	843
stackless	31.9	31.94	31.91570	0.00930	488033	249922	5906	843
stackless (post)	31.21	31.25	31.22720	0.00776	488033	249922	5906	843
stackless (leaf)	22.78	22.81	22.79100	0.00768	357508	178292	5906	843
Sub 3: 50944 faces (1000 repetitions):								
stack	26.78	26.84	26.79390	0.00835	89873	57321	12385	1683
alternately	40.64	40.82	40.74190	0.01677	379753	202261	12385	1683
stackless	39.85	40.08	39.94290	0.04043	379753	202261	12385	1683
stackless (post)	39.18	39.54	39.20970	0.04640	379753	202261	12385	1683
stackless (leaf)	41.75	41.89	41.77280	0.01450	549472	261649	12385	1683
Sub 4: 203776 faces (1000 repetitions):								
stack	48.14	48.18	48.16280	0.00873	136451	90763	22538	3357
alternately	60.73	60.86	60.75300	0.01396	403059	224067	22538	3357
stackless	59.61	59.65	59.63190	0.00880	403059	224067	22538	3357
stackless (post)	60.69	60.73	60.71270	0.00835	403059	224067	22538	3357
stackless (leaf)	98.45	98.83	98.65870	0.10559	1559634	700467	22538	3357

Table 6: Scalability test. It is seen that all the algorithms scale equally well, except the leaf algorithm in the second subdivison, where it has fewer iterations than the alternately stepping algorithms. It is also worth noting that except in the fourth subdivision, the post-indexed stackless algorithm is a little faster than the stackless algorithm.

9.4 Objects with different number of primitives

In this test, two objects will be intersecting. One of the objects will have many more primitives than the other. This will result in a situation where

Algorithm	min.	max.	mean	standard deviation	Iterations	overlaps	leaf overlaps	geometry overlaps
stack	0	0.01	0.00490	0.00500	71	35	0	0
alternately	0.28	0.3	0.28550	0.00589	6163	3081	0	0
stackless	0.25	0.27	0.25490	0.00574	6163	3081	0	0
stackless (post)	0.24	0.29	0.27390	0.00691	6163	3081	0	0
stackless (leaf)	0.94	0.97	0.96040	0.00706	30220	14712	0	0

Table 7: Test with different sized objects: The dynamic stack algorithm is very fast compared to the alternately stepping algorithms. This is because the dynamic algorithm always traverses the BVH with the largest volume first. The leaf algorithm is very slow compared to the other algorithms. This is because the leaves of the smallest object are compared to the BVH of the largest object. It would have been faster the other way around.

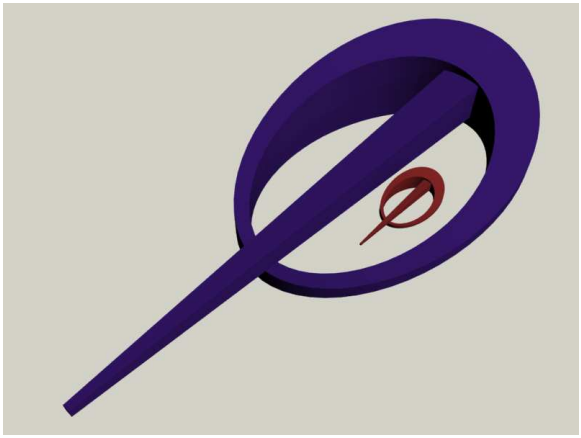


Figure 32: Image for the test of objects with different sizes. In this test the red object is scaled down an placed between the outer and inner part of the blue object. The objects do not intersect, but their BVHs overlap

a leaf in one of the BVHs will have to be compared to a subtree of the other BVH, many times. In the first test, the first object is subdivided four times giving 203776 primitives and a maximum depth level of 18. The second object is not subdivided, giving it 796 primitives and a maximum depth level of 10. In the second test, the objects switch places.

This test will show how well the stackless algorithm performs when the leaf-subalgorithm is used intensively.
An image of the setup can be seen in figure 31, and the results are shown in table 8.

Algorithm	min.	max.	mean	standard deviation	Iterations	overlaps	leaf overlaps	geometry overlaps
Different leveled objects. Largest first:								
stack	50.25	50.55	50.39380	0.07335	85259	71902	29273	1881
alternately	50.36	50.6	50.50860	0.07414	89627	74086	29273	1881
stackless	48.63	48.85	48.75800	0.07126	89627	74086	29273	1881
stackless (post)	49.36	49.6	49.49930	0.07442	89627	74086	29273	1881
stackless (leaf)	139.23	139.6	139.41180	0.10355	2670790	1262780	29273	1881
Different leveled objects. Smallest first:								
stack	86.51	87.18	86.87170	0.14642	113805	105099	48197	1677
alternately	83.55	84.15	83.77760	0.11930	120075	108234	48197	1677
stackless	80.11	80.45	80.31470	0.09204	120075	108234	48197	1677
stackless (post)	81.93	82.28	82.11830	0.10029	120075	108234	48197	1677
stackless (leaf)	82.24	83.16	82.44180	0.12014	118254	106926	48197	1677

Table 8: Test with objects with different number of primitives. The stackless algorithm performs a little better than the dynamic algorithm, even though the stackless algorithm utilizes the slower leaf-subalgorithm. It should be noted that the leaf algorithm actually performs better than the dynamic stack algorithm in the second case.

10 Results

The tests from the previous section will be discussed in this section.

In the case of separation, the stackless algorithm and the post-indexed stackless algorithm perform significantly better than the two stack algorithms. This is assumed to be becuse the stack algorithms have to setup the stack, which causes some overhead. The leaf algorithm on the other hand has to compare all of the leaves from one of the objects to the root of the other object, thus performing a lot of unnecessary, redundant comparisons.

In the intersection test the stack algorithm is the fastest. The stackless, the post-indexed and the alternately stepping stack algorithms perform almost equally well in this case.

In tests 1 and 2, the leaf algorithm actually performs a little better than the stackless, the post-indexed and the alternately stepping algorithms.

In the case of small proximity, the leaf algorithm does too many iterations, just as in the case of separation. When the number of overlaps between the BVHs is significantly smaller than the number of leaves in the objects, the leaf algorithm will do many unneccesary comparisons.

It seems that the stackless algorithm and the post-indexed algortihm perform very well in these tests. Following the separation test, it seems that a low number of iterations gives an advantage to these two algorithms. It should be noted, that in test three, the alternately stepping algorithm, the stackless algorithm and the post-indexed stackless algorithm are lucky to have fewer iterations than the dynamic stack algorithm.

In the close proximity tests, the dynamic stack algorithm seems to perform better than the others, except in the third test, where the stackless algorithm, and the post-indexed stackless algorithm are equally fast. In the close proximity tests, the stackless algorithm seems to be as fast or possibly a little faster than the post-indexed algorithm and the alternately stepping stack algorithm.

It is assumed that the algorithms are robust, since all algorithms in all tests and setups finds the same number of leaf overlaps and geometry overlaps.

In the scalability tests, the dynamic algorithm scales best in the number of faces over time. The three alternately stepping algorithms scale very equally, and the leaf algorithm scales worst, see figure 33. It is interesting to see that

the three alternately stepping algorithms scale equally, compared to the dynamic stack algorithm. This indicates that the traversal rule has a lot of influence on how well the algorithm scales.

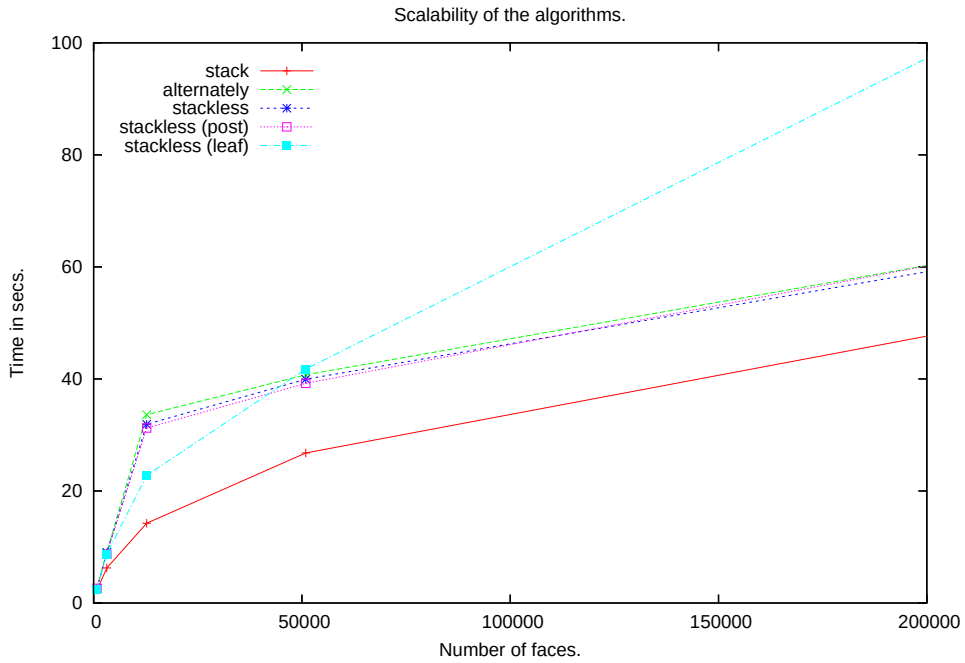


Figure 33: The three alternately stepping algorithms scale equally well, but the dynamic stack algorithm scales best.

When comparing equal objects with different sized volumes, it is apparant that this is where the force of the dynamic stack algorithm is, as was stated in section 4. The leaf algorithm does not handle this case very well - finding a lot of overlaps between BVs.

The stackless algorithm seems to perform slightly better than both the post-indexed stackless algorithm, and the alternately stepping stack algorithm.

When comparing objects with a large difference in number of primitives, the stackless algorithm performs best, but not a lot better than the other algorithms, except for the leaf algorithm. The leaf algorithm performs best if the object with the fewest leaves is taken as the first argument, as seen in test 2, as do the other algorithms.

The stackless algorithm and the post-indexed stackless algorithm do not seem to suffer a lot of penalty from using the leaf algorithm for collision between objects with different number of primitives.

All in all, the stackless algrorithm and the post-indexed stackless algorithm seem to perform a little bit (but not significant) better than alternately stepping stack algorithm. This is actually a surprise since it was assumed that the handling of the stack would slow down the alternately stepping stack algorithm. This leads to the hypotesis that the handling of the stack is not as expensive as expected.

The leaf algorithm is often not very good, but on some occations it performs better than some of the other algorithms. Mostly when there is much overlap.
The stackless algorithm and the post-indexed stackless algorithm seem to perform almost equally well. It is assumed that it is cache misses that cause the post-indexed algorithm to perform a little worse then the pre-indexed stackless algorithm.

The dynamic algorithm seem to perform best in most cases where there is much overlap, whereas the stackless algorithm seems performs better with little or no overlap. It should be noted, that in almost every test, the stackless algorithm had more iterations of its loop compared to the dynamic stack algorithm. This leads to thinking that in case of an equal number of iterations the stackless algorithm would be fastest. This is actually backed up by the fact that the stackless algorithm is almost always faster than the alterantely stepping stack algorithm, and these two algorithm always have the same number of iterations.

11 Conclusion

The two introduced algorithms, the stackless, and the post-indexed stackless, find the same collisions as a commonly used dynamic stack algorithm. The performance is unfortunatly not always as good as the dynamic stack algorithm on the tested hardware architecture.

The stackless algorithms are implemented without use of the stack, and are as such capable of being run on hardware where dynamic memory allocation is either impossible or very expensive, for instance on a GPU.

In future perspective, it would be very interesting to see if it is possible to create another predictable traversal rule which gives fewer iterations compared to a dynamic algorithm.

It is also of interest to try and use the predictable parts of the stackless algorithm combined with a dynamic stack algorithm, and possible create an algorithm with a compacked stack, making it possible to maintain the stack in e.g. a few bytes.

References

[1] Bullet, http://www.continuousphysics.com/bullet/.

[2] Robert Bridson, Ronald Fedkiw, and John Anderson. *Robust Treatment of Collisions, Contact and Friction for Cloth Animation.* Proceedings of SIGGRAPH '02, pp. 594-603, 2002.

[3] Christer Ericson. *Real-Time Collision Detection.* Morgan Kaufmann, 2005.

[4] Kenny Erleben, Jon Sporring, Knud Henriksen, and Henrik Dohlmann. *Physics-Based Animation.* Charles River Media, 2005.

[5] Jeffrey Goldsmith and John Salmon. *Automatic Creation of Object Hierarchies for Ray Tracing.* IEEE CGA, 1987.

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

[7] Niels Thrane and Lars Ole Simonsen. *A Comparison of Accelerating Structures for GPU Assisted Ray Tracing.* Master's thesis, University of Aarhus. Department of Computer Science, 2005.