

# Dual-Split Trees – Supplemental Materials

DAQI LIN, University of Utah

KONSTANTIN SHKURKO, University of Utah

IAN MALLETT, University of Utah

CEM YUKSEL, University of Utah

## ACM Reference Format:

Daqi Lin, Konstantin Shkurko, Ian Mallett, and Cem Yuksel. 2019. Dual-Split Trees – Supplemental Materials. 1, 1 (March 2019), 18 pages. <https://doi.org/10.1145/3306131.3317028>

## 1 NODE DEFINITIONS

Following is the C++ code for node definition of BVH (Listing 1), Dual-Split Tree (Listing 2), BIH [Wächter and Keller 2006] (Listing 3), H-Tree [Havran et al. 2006] (Listing 4), Compact BVH [Fabianowski and Dingliana 2009] (Listing 5).

**Listing 1.** *BVH Node Definition*

```
1 | // BVH uses variable-sized node, so a "BVHNode" only represents a 4-byte segment
2 | // We use an union to allow the segment to be used as both integer and floating point number
3 | // An internal node is 1 "idx" + 12 "plane" -- 52 bytes:
4 | //   -- The first 2 bits of idx indicate if left and right child are leaves: 0 -- not leaf 1 -- leaf
5 | //   -- The next 30 bits of idx store the address of the left child (right child is stored
6 | //     next to the left child)
7 | //   -- The 12 planes are left child bounding box's lower and upper positions
8 | //     and right child bounding box's lower and upper positions
9 | // A leaf is 1 idx -- 4 bytes:
10 | //   -- The idx stores the leaf's starting address in the global triangle index list
11 | struct BVHNode
12 | {
13 |     union
14 |     {
15 |         unsigned idx;
16 |         float plane;
17 |     };
18 | }
```

**Listing 2.** *Dual-Split Tree Node Definition*

```
1 | // Dual-Split Trees use variable-sized nodes, so a "DSTNode" only represents a 4-byte segment
2 | // We use an union to allow the segment to be used as both integer and floating point number
3 | // An internal node is 1 "header_offset" + 2 "plane" -- 12 bytes
```

---

Authors' addresses: Daqi Lin, University of Utah; Konstantin Shkurko, University of Utah; Ian Mallett, University of Utah; Cem Yuksel, University of Utah.

---

© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

```

4 | // -- The first 6 bits of "header_offset" is the header, as described in the paper
5 | // -- The next 26 bits of "header_offset" is the integer offset to the left child. In the case
6 | //   where a carving node acts as a leaf, it stores the leaf's offset in the global triangle index list
7 | // -- The 2 "planes" are either 2 splitting planes or 2 carving planes, depending on the node type
8 | // A leaf is 1 "header_offset" -- 4 bytes
9 | // -- The first 6 bits are "000001"
10 | // -- The next 26 bits store the leaf's offset in the global triangle index list
11 | struct DSTNode
12 | {
13 |     union
14 |     {
15 |         unsigned header_offset;
16 |         float plane;
17 |     }
18 | }

```

**Listing 3.** *BIH Node Definition*

```

1 | // We store BIH in a compact way, using variable-sized nodes, so a "BIHNode" only represents a 4-byte segment
2 | // We use an union to allow the segment to be used as both integer and floating point number
3 | // An internal node is 1 "header_offset" + 2 "plane" -- 12 bytes
4 | // -- The first 2 bits of "header_offset" store the splitting axis (00, 01, 10)
5 | // -- The 3rd bit of "header_offset" stores 1 if the left child is a leaf and 0 if not. Note that
6 | //   this is the only extra information we store compared to the node representation in [Wachter and Keller 2006],
7 | //   but it allows a compact storage of BIH in memory as our Dual-Split Trees, for fair comparison.
8 | // -- The next 29 bits store the integer offset to the left child
9 | // -- The 2 "planes" are the 2 splitting planes
10 | // A leaf is 1 "header_offset"
11 | // -- The first 2 bits of "header_offset" is "11" to mark leaf.
12 | // -- The next 30 bits store the leaf's offset in the global triangle index list
13 |
14 | struct BIHNode
15 | {
16 |     union
17 |     {
18 |         unsigned header_offset;
19 |         float plane;
20 |     }
21 | }

```

**Listing 4.** *HTree Node Definition*

```

1 | // We store HTree in a compact way, using variable-sized nodes, so an "HTreeNode" only represents a 4-byte segment
2 | // In this compact representation, a two-plane node (BV2) or a SKD-tree node requires 12 bytes instead of 16 bytes
3 | // as in [Havran et al. 2006], a six-plane node (BV6) only requires 28 bytes instead of 32 bytes.
4 | // A leaf only uses 4 bytes instead of 16 bytes
5 | // We use an union to allow the segment to be used as both integer and floating point number.
6 | // Any node has a "header_offset" 4-byte segment, the first 3 bits of "header_offset" specify the node type
7 | // 0 -- SKD Node with splitting axis X, 1 -- SKD Node with splitting axis Y, 2 -- SKD Node with splitting axis Z

```

```

8 | // 3 -- Leaf
9 | // 4 -- BV2 with bounding axis X, 5 -- BV2 with bounding axis Y, 6 -- BV2 with bounding axis Z
10 | // 7 -- BV6
11 | // An SKD-node is 1 "header_offset" + 2 "plane" -- 12 bytes
12 | //     -- the first 3 bits of "header_offset" is 0-2
13 | //     -- the next 2 bits of "header_offset" encodes left child size
14 | //         0 -- left child has 7 words (28 bytes)
15 | //         1 -- left child has 3 words (12 bytes)
16 | //         2 -- left child was 1 word (4 bytes)
17 | //     -- The 2 "planes" are the 2 splitting planes
18 | //     -- the next 27 bits encodes the integer offset to the left child
19 | // A BV2 node is 1 "header_offset" + 2 "plane" -- 12 bytes
20 | //     -- the first 3 bits of "header_offset" is 4-6
21 | //     -- the last 27 bits encodes the integer offset to the child
22 | //     -- The 2 "planes" are the 2 bounding planes
23 | // A BV6 node is 1 "header_offset" + 6 "plane" -- 28 bytes
24 | //     -- the first 3 bits of "header_offset" is 7 (111)
25 | //     -- the last 27 bits encode the integer offset to the child
26 | //     -- The 6 "planes" store the bounding box
27 | // A leaf node is 1 "header_offset" -- 4 bytes
28 | //     -- the first 3 bits of "header_offset" is 3 (011)
29 | //     -- the last 29 bits encode the leaf's offset in the global triangle index list
30 |
31 | struct HTreeNode
32 | {
33 |     union
34 |     {
35 |         unsigned header_offset;
36 |         float plane;
37 |     }
38 | }

```

Listing 5. Compact BVH Node Definition

```

1 | struct CompactBVHNode
2 | {
3 |     //24 bytes
4 |     float m[3];
5 |     float M[3];
6 |
7 |     //8 bytes
8 |     struct
9 |     {
10 |         unsigned left : 28;
11 |         unsigned l : 3;
12 |         unsigned isLeftLeaf : 1;
13 |         unsigned right : 28;
14 |         unsigned L : 3;
15 |         unsigned isRightLeaf : 1;
16 |     };

```

```
17 | };
```

## 2 TRAVERSAL KERNELS

We provide commented C++ code for the implementations of the traversal kernels of BVH (Listing 6), Dual-Split Tree (Listing 7), BIH (Listing 8), H-Tree (Listing 9), compact BVH (Listing 10).

The compact BVH is implemented faithfully to the node structure and traversal algorithm (Algorithm 3) provided by Fabianowski and Dingliana [2009].

**Listing 6.** *BVH Traversal Kernel*

```
1 | bool RayTracing::BVHTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
2 | {
3 |     //precompute the reciprocal of ray direction to avoid dividing later
4 |     vec3 invdir = 1.f / ray.dir;
5 |     //precompute the signs of ray directions
6 |     const int sign[3] = { invdir.x < 0, invdir.y < 0, invdir.z < 0 };
7 |     bool isLeaf = false;
8 |     //initialize the offset to point to the root node
9 |     int offset = 0;
10 |    //initialize closest hit to infinity
11 |    hitRecord.t = INFINITY;
12 |
13 |    //discard rays that miss the scene bounding box
14 |    if (BoxIntersection(ray.origin, invdir, sign, &bbox, hitRecord.t) == INFINITY) return false;
15 |    //create traversal stack
16 |    BVHStackItem stack[MAX_LEVEL];
17 |    int stack_ptr = -1;
18 |    while (true)
19 |    {
20 |        if (isLeaf)
21 |        {
22 |            // global triangle index list -- the last triangle associated with the leaf will have 1 at MSB
23 |            int trioffset = bv->nodes[offset].idx;
24 |            for (int i = trioffset;; i++)
25 |            {
26 |                int triId = globalTriangleIndexList[i];
27 |                //mask the lower 31 bits as triangle index
28 |                //hitRecord records the closest hit
29 |                TriangleIntersection(ray, model->tris[triId & 0x7fffffff], hitRecord);
30 |                //check if the triangle is the last triangle
31 |                if (triId < 0) break;
32 |            }
33 |        }
34 |        else
35 |        {
36 |            //first 2 bits of the node indicates if left and right child are leaves: 0 -- not leaf 1 -- leaf
37 |            char isLeftLeaf = bv->nodes[offset].idx >> 30 & 3;
38 |            char isRightLeaf = isLeftLeaf & 1;
```

```

39 |         isLeftLeaf = isLeftLeaf >> 1;
40 |
41 |         //next 30 bits of the internal node is the offset of the left child
42 |         //compute right child offset -- if left child is leaf: offset by 1 / otherwise: offset by 13
43 |         int first = offset + (bvh->nodes[offset].idx & 0x3fffffff), second = first + 12*(!isLeftLeaf) + 1;
44 |
45 |         // "An Efficient and Robust Ray-Box Intersection Algorithm" -- Shirley et al.
46 |         // returns the near hit with the bounding box
47 |         // return infinity if there is not hit / hit at back / hit is farther than the closest hit (hitRecord.t)
48 |         float first_t = BoxIntersection(ray.origin, invdir, sign, (aabb*)&bvh->nodes[offset + 1], hitRecord.t);
49 |         float second_t = BoxIntersection(ray.origin, invdir, sign, (aabb*)&bvh->nodes[offset + 7], hitRecord.t);
50 |
51 |         // if first child has farther hit than second child
52 |         if (first_t > second_t)
53 |         {
54 |             // update offset and flag, visit second child first
55 |             offset = second;
56 |             isLeaf = isRightLeaf;
57 |             // if hit first child, push it to stack (isLeaf flag, offset, near hit with bounding box)
58 |             if (first_t != INFINITY) stack[++stack_ptr] = BVHStackItem(isLeftLeaf, first, first_t);
59 |         }
60 |         else
61 |         {
62 |             // check if first child is hit
63 |             if (first_t != INFINITY)
64 |             {
65 |                 offset = first;
66 |                 isLeaf = isLeftLeaf;
67 |             }
68 |             // the case when first_t == second_t == INFINITY, hit neither child, pop stack
69 |             else goto pop;
70 |             // if hit second child, push it to stack
71 |             if (second_t != INFINITY) stack[++stack_ptr] = BVHStackItem(isRightLeaf, second, second_t);
72 |         }
73 |         continue;
74 |     }
75 | pop:
76 |     while (true)
77 |     {
78 |         // if stack is empty, terminate program and report hit or no hit
79 |         if (stack_ptr == -1) return hitRecord.t < INFINITY;
80 |         BVHStackItem item = stack[stack_ptr--];
81 |         // discard node if near hit with its bounding box is not closer than current closest hit
82 |         // otherwise update states, visit the node
83 |         if (item.tmin < hitRecord.t)
84 |         {
85 |             offset = item.offset;
86 |             isLeaf = item.isLeaf;
87 |             break;
88 |         }
89 |     }

```

```

90 || }
91 || return false;
92 || }

```

**Listing 7.** *Dual-Split Tree Traversal Kernel*

```

1 || bool RayTracing::DSTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
2 || {
3 || //make sure that if not shadowray, normalize first
4 || float tmin = 0;
5 || float tmax = INFINITY;
6 ||
7 || // stores first 4 byte of a dual split tree node (header+offset)
8 || unsigned header_offset;
9 || //stores the highest 5 bits of the header
10 || unsigned header5;
11 || //the 6th header bit -- 1 : leaf or leaf contained 0 : is not/does not contain a leaf
12 || bool leafbit;
13 || //idx is the current node index
14 || unsigned idx = 0;
15 || hitRecord.t = INFINITY;
16 ||
17 || const vec3 invdir = 1.f / ray.dir;
18 || const int dir_sgn[3] = { invdir.x < 0, invdir.y < 0, invdir.z < 0 };
19 || if (BoxIntersection(ray.origin, invdir, dir_sgn, &bbox, tmax) == INFINITY) return false;
20 || //make sure ray starts at origin
21 || if (tmin < 0) tmin = 0;
22 ||
23 || // a StackItem stores the node's address, tmin and tmax
24 || StackItem stack[MAX_LEVEL];
25 || int stack_ptr = -1;
26 ||
27 || while (true)
28 || {
29 || //get the header+offset
30 || header_offset = dst->nodes[idx].header_offset;
31 || header5 = header_offset >> 27;
32 || //get the leaf bit at the 26th bit
33 || leafbit = (header_offset >> 26) & 1;
34 ||
35 || //the highest bit is 0 -- either a split node or a leaf
36 || if (header5 >> 4 == 0)
37 || {
38 || // if it is a leaf
39 || if (leafbit)
40 || {
41 || //get global triangle index list offset in lower 26 bits
42 || idx = header_offset & 0x3FFFFFF;
43 || goto leaf;
44 || }

```

```

45 | //split planes for the split node
46 | float cursplit[2];
47 | cursplit[0] = dst->nodes[idx + 1].plane;
48 | cursplit[1] = dst->nodes[idx + 2].plane;
49 | //get the left child offset relative to parent
50 | idx += header_offset & 0x3FFFFFF;
51 |
52 | //decode the splitting axis
53 | unsigned axis = header5 >> 2;
54 | //get ray direction sign at current axis
55 | unsigned sign = dir_sgn[axis];
56 | //diff: right child offset relative to left child
57 | //if left child is a leaf: 1 Otherwise: 3
58 | unsigned diff = header5 & 3;
59 | //ray-plane intersection with two splitting planes
60 | float ts1 = (cursplit[sign] - ray.origin[axis]) * invdir[axis];
61 | float ts2 = (cursplit[sign ^ 1] - ray.origin[axis]) * invdir[axis];
62 | // sign bit is multiplied by diff
63 | // if diff is 1
64 | //if sign is 1 -> sign ^ diff = 1 ^ 1 = 0 -> left child offset
65 | //otherwise -> sign ^ diff = 0 ^ 1 = 0 -> right child offset
66 | // if diff is 3
67 | //if sign is 1 -> sign ^ diff = 3 ^ 3 = 0 -> left child offset
68 | //otherwise -> sign ^ diff = 0 ^ 3 = 3 -> right child offset
69 | sign *= diff;
70 | // determine traversal order
71 | if (tmax >= ts2) // far child is intersected
72 | {
73 |     float tnext = max(tmin, ts2);
74 |     if (tmin <= ts1) // near child is also intersected
75 |     {
76 |         stack.stack[++stack_ptr] = StackItem(idx + (sign ^ diff), tnext, tmax);
77 |         idx += sign;
78 |         tmax = min(tmax, ts1);
79 |     }
80 |     else //near child is not intersected
81 |     {
82 |         idx += sign ^ diff;
83 |         tmin = tnext;
84 |     }
85 |     continue;
86 | }
87 | else //far child is not intersected
88 | {
89 |     if (tmin <= ts1) //near child is intersected
90 |     {
91 |         idx += sign;
92 |         tmax = min(tmax, ts1);
93 |         continue;
94 |     }
95 |     //neither child is intersected

```

```

96 |         else goto pop;
97 |     }
98 | }
99 | //Carve Node
100 | else
101 | {
102 |     float cursplit[2];
103 |     cursplit[0] = dst->nodes[idx + 1].plane;
104 |     cursplit[1] = dst->nodes[idx + 2].plane;
105 |
106 |
107 |     //carve type 1 at 30th and 29th bits
108 |     //00-xy 01-xz 11-yz (dual axes carve node)
109 |     //10-xx/yy/zz (single axis carve node)
110 |     char carvetype1 = header5 >> 2 & 3;
111 |
112 |     //carve type 2 at 28th and 27th bits
113 |     //if node is single axis carve node -> gives axis
114 |     //otherwise -> gives plane signs
115 |     char carvetype2 = header5 & 3;
116 |
117 |     //single axis carve node
118 |     if (carvetype1 == 2)
119 |     {
120 |         float ts1, ts2;
121 |         unsigned sign = dir_sgn[carvetype2];
122 |         ts1 = (cursplit[sign] - ray.origin[carvetype2]) * invdir[carvetype2];
123 |         ts2 = (cursplit[sign ^ 1] - ray.origin[carvetype2]) * invdir[carvetype2];
124 |         //compute trimmed tmin and tmax
125 |         tmax = min(ts1, tmax);
126 |         tmin = max(ts2, tmin);
127 |         //does not hit bounding volume, pop stack
128 |         if (tmin > tmax) goto pop;
129 |         //get offset bits
130 |         int offset = header_offset & 0x3FFFFFFF;
131 |
132 |         //if containing a leaf
133 |         //offset treated as global trianlge index list offset and go to leaf
134 |         if (leafbit)
135 |         {
136 |             idx = offset;
137 |             goto leaf;
138 |         }
139 |         //otherwise
140 |         //offset treated as child offset relative to parent
141 |         idx += offset;
142 |         continue;
143 |     }
144 |     //dual axes carve node
145 |     else
146 |     {

```



```

147 | // get dual-axes
148 | unsigned axis1 = carvetype1 >> 1;
149 | unsigned axis2 = (carvetype1 & 1) + 1;
150 |
151 | // simplified corner carving test -- see Figure 4
152 | float t_min_0, t_min_1, t_max_0, t_max_1;
153 | t_min_0 = (cursplit[0] - ray.origin[axis1]) * invdir[axis1];
154 | t_max_0 = tmax;
155 | //plane sign in first axis is same as ray direction
156 | if (dir_sgn[axis1] == (carvetype2 >> 1))
157 | {
158 |     t_max_0 = t_min_0;
159 |     t_min_0 = tmin;
160 | }
161 | t_min_1 = (cursplit[1] - ray.origin[axis2]) * invdir[axis2];
162 | t_max_1 = tmax;
163 | if (dir_sgn[axis2] == (carvetype2 & 1))
164 | {
165 |     t_max_1 = t_min_1;
166 |     t_min_1 = tmin;
167 | }
168 |
169 | //compute trimmed tmin and tmax
170 | tmin = max(tmin, max(t_min_0, t_min_1));
171 | tmax = min(tmax, min(t_max_0, t_max_1));
172 | //does not hit bounding volume, pop stack
173 | if (tmin > tmax) goto pop;
174 | //if containing a leaf
175 | //offset treated as global triangle index list offset and go to leaf
176 | int offset = header_offset & 0x3FFFFFF;
177 | if (leafbit)
178 | {
179 |     idx = offset;
180 |     goto leaf;
181 | }
182 | //otherwise
183 | //offset treated as child offset relative to parent
184 | idx += offset;
185 | continue;
186 | }
187 | }
188 |
189 | // leaf -- go through all triangles
190 | leaf:
191 | for (unsigned i = idx; i++)
192 | {
193 |     int triId = globalTriangleIndexList[i];
194 |     TriangleIntersection(ray, model->tris[triId & 0x7FFFFFFF], hitRecord);
195 |     if (triId < 0) break;
196 | }
197 |

```

```

198 || pop:
199 ||     while (true)
200 ||     {
201 ||         if (stack_ptr == -1) return hitRecord.t < INFINITY;
202 ||         StackItem item = stack.stack[stack_ptr--];
203 ||         idx = item.idx;
204 ||         tmin = item.tmin;
205 ||         if (tmin < hitRecord.t)
206 ||         {
207 ||             tmax = min(hitRecord.t, item.tmax);
208 ||             break;
209 ||         }
210 ||     }
211 || }
212 || return false;
213 || }

```

Listing 8. BIH Traversal Kernel

```

1 || bool RayTracing::BIHTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
2 || {
3 ||     //make sure that if not shadowray, normalize first
4 ||     float tmin = 0;
5 ||     float tmax = INFINITY;
6 ||
7 ||     // stores first 4 byte of a BIH node (header+offset)
8 ||     unsigned header_offset;
9 ||     unsigned header2; // the first two bits of the header, specifying leaf or splitting axis
10 ||    bool sizebit; // the 3rd bit of the header, specifying left child size
11 ||    //idx is the current node index
12 ||    unsigned idx = 0;
13 ||    hitRecord.t = INFINITY;
14 ||
15 ||    const vec3 invdir = 1.f / ray.dir;
16 ||    const int dir_sgn[3] = { invdir.x < 0, invdir.y < 0, invdir.z < 0 };
17 ||    if (BoxIntersection(ray.origin, invdir, dir_sgn, &bbox, tmax) == INFINITY) return false;
18 ||    //make sure ray starts at origin
19 ||    if (tmin < 0) tmin = 0;
20 ||
21 ||    // a StackItem stores the node's address, tmin and tmax
22 ||    StackItem stack[MAX_LEVEL];
23 ||    int stack_ptr = -1;
24 ||
25 ||    while (true)
26 ||    {
27 ||        //get the header+offset
28 ||        header_offset = bih->nodes[idx].header_offset;
29 ||
30 ||        header2 = header_offset >> 29;
31 ||        sizebit = header2 & 1;

```

```

32     header2 >>= 1;
33
34     // the first two bits are 11 -- a leaf
35     if (header2 == 3)
36     {
37         //get global triangle index list offset in lower 30 bits
38         idx = header_offset & 0x3FFFFFFF;
39         for (unsigned i = idx;;i++)
40         {
41             int triId = globalTriangleIndexList[i];
42             TriangleIntersection(ray, model->tris[triId & 0x7FFFFFFF], hitRecord);
43             if (triId < 0) break;
44         }
45     }
46     else // an internal node
47     {
48         //splitting planes for the node
49         float cursplit[2];
50         cursplit[0] = bih->nodes[idx + 1].plane;
51         cursplit[1] = bih->nodes[idx + 2].plane;
52         //get the left child offset relative to parent
53         idx += header_offset & 0x1FFFFFFF;
54         //get ray direction sign at current axis (stored in header2)
55         unsigned sign = dir_sgn[header2];
56
57         //ray-plane intersection with two splitting planes
58         float ts1 = (cursplit[sign] - ray.origin[axis]) * invdir[axis];
59         float ts2 = (cursplit[sign ^ 1] - ray.origin[axis]) * invdir[axis];
60
61         //decode left child size from sizebit 0--left child has 3 words, 1--left child has 1 word
62         unsigned diff = 3 >> sizebit;
63         // sign bit is multiplied by diff
64         // if diff is 1
65         //if sign is 1 -> sign ^ diff = 1 ^ 1 = 0 -> left child offset
66         //otherwise -> sign ^ diff = 0 ^ 1 = 0 -> right child offset
67         // if diff is 3
68         //if sign is 1 -> sign ^ diff = 3 ^ 3 = 0 -> left child offset
69         //otherwise -> sign ^ diff = 0 ^ 3 = 3 -> right child offset
70         sign *= diff;
71
72         // determine traversal order
73         if (tmax >= ts2) // far child is intersected
74         {
75             float tnext = max(tmin, ts2);
76             if (tmin <= ts1) // near child is also intersected
77             {
78                 stack.stack[++stack_ptr] = StackItem(idx + (sign ^ diff), tnext, tmax);
79                 idx += sign;
80                 tmax = min(tmax, ts1);
81             }
82             else //near child is not intersected

```

```

83 |     {
84 |         idx += sign ^ diff;
85 |         tmin = tnext;
86 |     }
87 |     continue;
88 | }
89 | else //far child is not intersected
90 | {
91 |     if (tmin <= ts1) //near child is intersected
92 |     {
93 |         idx += sign;
94 |         tmax = min(tmax, ts1);
95 |         continue;
96 |     }
97 |     //neither child is intersected
98 |     else goto pop;
99 | }
100 | }
101 |
102 | pop:
103 |     while (true)
104 |     {
105 |         if (stack_ptr == -1) return hitRecord.t < INFINITY;
106 |         idx = item.idx;
107 |         StackItem item = stack.stack[stack_ptr--];
108 |         tmin = item.tmin;
109 |         if (tmin < hitRecord.t)
110 |         {
111 |             tmax = min(hitRecord.t, item.tmax);
112 |             break;
113 |         }
114 |     }
115 | }
116 | return false;
117 | }

```

**Listing 9.** *H-Tree Traversal Kernel*

```

1 | bool RayTracing::HTreeTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
2 | {
3 |     //make sure that if not shadowray, normalize first
4 |     float tmin = 0;
5 |     float tmax = INFINITY;
6 |
7 |     // stores first 4 byte of a dual split tree node (header+offset)
8 |     unsigned header_offset;
9 |     //stores the first 3 bits of the header (node type)
10 |    unsigned nodeType;
11 |    //idx is the current node index
12 |    unsigned idx = 0;

```

```

13 hitRecord.t = INFINITY;
14
15 const vec3 invdir = 1.f / ray.dir;
16 const int dir_sgn[3] = { invdir.x < 0, invdir.y < 0, invdir.z < 0 };
17 if (BoxIntersection(ray.origin, invdir, dir_sgn, &bbox, tmax) == INFINITY) return false;
18 //make sure ray starts at origin
19 if (tmin < 0) tmin = 0;
20
21 // a StackItem stores the node's address, tmin and tmax
22 StackItem stack[MAX_LEVEL];
23 int stack_ptr = -1;
24
25 while (true)
26 {
27 //get the header+offset
28 header_offset = htree->nodes[idx].header_offset;
29 nodeType = header_offset >> 29;
30 if (nodeType == 3) //leaf
31 {
32 //get global triangle index list offset in lower 29 bits
33 idx = header_offset & 0x1FFFFFFF;
34 for (unsigned i = idx;;i++)
35 {
36 int triId = globalTriangleIndexList[i];
37 TriangleIntersection(ray, model->tris[triId & 0x7FFFFFFF], hitRecord);
38 if (triId < 0) break;
39 }
40 }
41 else if (nodeType == 7) // BV6
42 {
43 // a bounding box intersection test which also considers tmin
44 if (!BV6Intersection(ray, (aabb*)&htree->nodes[idx + 1], tmin, tmax)) goto pop;
45 //add the child offset relative to parent
46 idx += header_offset & 0x7FFFFFFF;
47 continue;
48 }
49 else if (nodeType <= 2) // SKD
50 {
51 //splitting planes for the node
52 float cursplit[2];
53 cursplit[0] = htree->nodes[idx + 1].plane;
54 cursplit[1] = htree->nodes[idx + 2].plane;
55 //get the left child offset relative to parent
56 idx += header_offset & 0x7FFFFFFF;
57 //get ray direction sign at current axis (stored in nodeType)
58 unsigned sign = dir_sgn[nodeType];
59
60 //ray-plane intersection with two splitting planes
61 float ts1 = (cursplit[sign] - ray.origin[axis]) * invdir[axis];
62 float ts2 = (cursplit[sign ^ 1] - ray.origin[axis]) * invdir[axis];
63

```

```

64 |         //decode left child size
65 |         unsigned diff = 7 >> (header_offset >> 27 & 3);
66 |         // sign bit is multiplied by diff
67 |     // if diff is 1
68 |         //if sign is 1 -> sign ^ diff = 1 ^ 1 = 0 -> left child offset
69 |         //otherwise -> sign ^ diff = 0 ^ 1 = 0 -> right child offset
70 |     // if diff is 3
71 |         //if sign is 1 -> sign ^ diff = 3 ^ 3 = 0 -> left child offset
72 |         //otherwise -> sign ^ diff = 0 ^ 3 = 3 -> right child offset
73 |     sign *= diff;
74 |         // determine traversal order
75 |     if (tmax >= ts2) // far child is intersected
76 |     {
77 |         float tnext = max(tmin, ts2);
78 |         if (tmin <= ts1) // near child is also intersected
79 |         {
80 |             stack.stack[++stack_ptr] = StackItem(idx + (sign ^ diff), tnext, tmax);
81 |             idx += sign;
82 |             tmax = min(tmax, ts1);
83 |         }
84 |         else //near child is not intersected
85 |         {
86 |             idx += sign ^ diff;
87 |             tmin = tnext;
88 |         }
89 |         continue;
90 |     }
91 |     else //far child is not intersected
92 |     {
93 |         if (tmin <= ts1) //near child is intersected
94 |         {
95 |             idx += sign;
96 |             tmax = min(tmax, ts1);
97 |             continue;
98 |         }
99 |         //neither child is intersected
100 |         else goto pop;
101 |     }
102 |     }
103 |     else //BV2
104 |     {
105 |         nodeType -= 4; //convert to bounding axis
106 |         float cursplit[2];
107 |         cursplit[0] = htree->nodes[idx + 1].plane;
108 |         cursplit[1] = htree->nodes[idx + 2].plane;
109 |
110 |         float ts1, ts2;
111 |         unsigned sign = dir_sgn[nodeType];
112 |         ts1 = (cursplit[sign] - ray.origin[nodeType]) * invdir[nodeType];
113 |         ts2 = (cursplit[sign ^ 1] - ray.origin[nodeType]) * invdir[nodeType];
114 |

```

```

115 |         //compute trimmed tmin and tmax
116 |         tmax = min(ts1, tmax);
117 |         tmin = max(ts2, tmin);
118 |         //does not hit bounding volume, pop stack
119 |         if (tmin > tmax) goto pop;
120 |
121 |         idx += header_offset & 0x7FFFFFFF;
122 |     }
123 |
124 | pop:
125 |     while (true)
126 |     {
127 |         if (stack_ptr == -1) return hitRecord.t < INFINITY;
128 |         StackItem item = stack.stack[stack_ptr--];
129 |         idx = item.idx;
130 |         tmin = item.tmin;
131 |         if (tmin < hitRecord.t)
132 |         {
133 |             tmax = min(hitRecord.t, item.tmax);
134 |             break;
135 |         }
136 |     }
137 | }
138 | return false;
139 | }

```

Listing 10. Compact BVH Traversal Kernel

```

1 | bool RayTracing::CompactBVHTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
2 | {
3 |     vec3 invdir = 1.f / ray.dir;
4 |     //idx is the current node index
5 |     unsigned idx = 0;
6 |     // using the notation in Compact BVH Storage for Ray Tracing and Photon Mapping (Fabianowski et al.)
7 |     // a: tmin, b: tmax
8 |     float a = 0;
9 |     float b = INFINITY;
10 |    hitRecord.t = INFINITY;
11 |
12 |    // update a and b to near and far hit with the scene bounding box
13 |    // CompactBVHBoxIntersection: the same function as BVH's BoxIntersection
14 |    // but passing one argument that gets the ray-box far hit to initialize b
15 |    a = CompactBVHBoxIntersection(ray.origin, invdir, &bbox, hitRecord.t, b);
16 |    //miss the scene bounding box
17 |    if (a == INFINITY) return false;
18 |    //make sure ray starts at origin
19 |    if (a < 0) a = 0;
20 |    // initialize stack
21 |    CompactBVHStackItem stack[MAX_LEVEL];
22 |    int stack_ptr = -1;

```

```

23 | while (true)
24 | {
25 |     if (idx >> 31) // if the node is a leaf (the isLeaf boolean is packed with the index at the MSB)
26 |     {
27 |         //get lowest 28 bits as triangle offset
28 |         int trioffset = idx & 0xFFFFFFFF;
29 |         for (int i = trioffset;; i++)
30 |         {
31 |             int triId = bvh->triIds[i];
32 |             TriangleIntersection(ray, model->tris[triId & 0x7FFFFFFF], hitRecord);
33 |             if (triId < 0) break;
34 |         }
35 |     }
36 |     else
37 |     {
38 |         //get child assignment indicator l and L
39 |         //please see Compact BVH Storage for Ray Tracing and Photon Mapping
40 |         //http://www.fabianowski.eu/research/egie2009.pdf
41 |         //each of ls and Ls is stored in 3 bits
42 |         //for ls: the bits represent the xmin, ymin, zmin planes
43 |         //for Ls: the bits represent the xmax, ymax, zmax planes
44 |         //if a bit is 1, the corresponding plane belongs to left child
45 |         //otherwise, it belongs to right child
46 |
47 |         //cbvh is the pointer to the compact BVH
48 |         unsigned ls = cbvh->nodes[idx].l;
49 |         unsigned Ls = cbvh->nodes[idx].L;
50 |
51 |         //initialize left and right child near and far hit values to parent values
52 |
53 |         ///////////////////////////////////////////////////////////////////
54 |         //the following part is identical to the the Algorithm 3 provided in the paper
55 |         float al = a, ar = a;
56 |         float bl = min(b, hitRecord.t), br = b;
57 |
58 |         //m and M stores (xmin,ymin,zmin) and (xmax,ymax,zmax)
59 |         vec3 t1 = vec3((cbvh->nodes[idx].m[0] - ray.origin[0]) * invdir[0],
60 |             (cbvh->nodes[idx].m[1] - ray.origin[1]) * invdir[1],
61 |             (cbvh->nodes[idx].m[2] - ray.origin[2]) * invdir[2]);
62 |
63 |         vec3 t2 = vec3((cbvh->nodes[idx].M[0] - ray.origin[0]) * invdir[0],
64 |             (cbvh->nodes[idx].M[1] - ray.origin[1]) * invdir[1],
65 |             (cbvh->nodes[idx].M[2] - ray.origin[2]) * invdir[2]);
66 |         vec3 t3 = INFINITY * invdir;
67 |
68 |         // the "slab" test for compact BVH
69 |         for (int k = 0; k < 3; k++)
70 |         {
71 |             float t1l, t1r, t2l, t2r;
72 |             //if the min plane at axis k belongs to left child
73 |             if ((ls >> k) & 1) { t1l = t1[k]; t1r = -t3[k]; }

```



```

74 |         else { t1l = -t3[k]; t1r = t1[k]; }
75 |         //if the min plane at axis k belongs to right child
76 |         if ((Ls >> k) & 1) { t2l = t2[k]; t2r = t3[k]; }
77 |         else { t2l = t3[k]; t2r = t2[k]; }
78 |         a1 = max(a1, min(t1l, t2l));
79 |         b1 = min(b1, max(t1l, t2l));
80 |         ar = max(ar, min(t1r, t2r));
81 |         br = min(br, max(t1r, t2r));
82 |     }
83 |
84 |     //if hit both children
85 |     if (a1 <= b1 && ar <= br)
86 |     {
87 |         //always visit the closer child first
88 |         if (a1 <= ar)
89 |         {
90 |             a = a1;
91 |             b = b1;
92 |             //push far child to stack
93 |             stack[++stack_ptr] = DSStep(cbvh->nodes[idx].right | (cbvh->nodes[idx].isRightLeaf << 31), ar, br);
94 |             //update current index
95 |             idx = cbvh->nodes[idx].left | (cbvh->nodes[idx].isLeftLeaf << 31);
96 |         }
97 |         else
98 |         {
99 |             a = ar;
100 |            b = br;
101 |            stack[++stack_ptr] = DSStep(cbvh->nodes[idx].left | (cbvh->nodes[idx].isLeftLeaf << 31), a1, b1);
102 |            idx = cbvh->nodes[idx].right | (cbvh->nodes[idx].isRightLeaf << 31);
103 |        }
104 |        continue;
105 |    }
106 |    //only hit left child
107 |    else if (a1 <= b1)
108 |    {
109 |        a = a1;
110 |        b = b1;
111 |        idx = cbvh->nodes[idx].left | (cbvh->nodes[idx].isLeftLeaf << 31);
112 |        continue;
113 |    }
114 |    //only hit right child
115 |    else if (ar <= br)
116 |    {
117 |        a = ar;
118 |        b = br;
119 |        idx = cbvh->nodes[idx].right | (cbvh->nodes[idx].isRightLeaf << 31);
120 |        continue;
121 |    }
122 |    //the above part is identical to the the Algorithm 3 provided in the paper
123 |    //////////////////////////////////////
124 | }

```

```
125 || pop:
126 ||     while (true)
127 ||     {
128 ||         if (stack_ptr == -1) return hitRecord.t < INFINITY;
129 ||         CompactBVHStackItem item = stack[stack_ptr--];
130 ||         if (item.tmin < hitRecord.t)
131 ||         {
132 ||             idx = item.idx;
133 ||             a = item.tmin;
134 ||             b = item.tmax;
135 ||             break;
136 ||         }
137 ||     }
138 || }
139 || return false;
140 || }
141 || `
```

## REFERENCES

- Bartosz Fabianowski and John Dingliana. 2009. Compact BVH storage for ray tracing and photon mapping. In *Proc. of Eurographics Ireland Workshop*. 1–8.
- Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. 2006. On the fast construction of spatial hierarchies for ray tracing. In *IEEE Symposium on Interactive Ray Tracing*. IEEE, 71–80.
- Carsten Wächter and Alexander Keller. 2006. Instant ray tracing: The bounding interval hierarchy. *Rendering Techniques 2006 (2006)*, 139–149.