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*

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

Listing 2. *Dual-Split Tree Node Definition*

```
// Dual-Split Trees use variable-sized nodes, so a "DSTNode" only represents a 4-byte segment
// We use an union to allow the segment to be used as both integer and floating point number
// 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.

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

Listing 3. *BIH Node Definition*

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

struct BIHNode
{
    union
    {
        unsigned header_offset;
        float plane;
    }
}
```

Listing 4. *HTree Node Definition*

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

```
// 3 -- Leaf
// 4 -- BV2 with bounding axis X, 5 -- BV2 with bounding axis Y, 6 -- BV2 with bounding axis Z
// 7 -- BV6
// An SKD-node is 1 "header_offset" + 2 "plane" -- 12 bytes
//       -- the first 3 bits of "header_offset" is 0-2
//       -- the next 2 bits of "header_offset" encodes left child size
//           0 -- left child has 7 words (28 bytes)
//           1 -- left child has 3 words (12 bytes)
//           2 -- left child was 1 word (4 bytes)
//       -- The 2 "planes" are the 2 splitting planes
//       -- the next 27 bits encodes the integer offset to the left child
// A BV2 node is 1 "header_offset" + 2 "plane" -- 12 bytes
//       -- the first 3 bits of "header_offset" is 4-6
//       -- the last 27 bits encodes the integer offset to the child
//       -- The 2 "planes" are the 2 bounding planes
// A BV6 node is 1 "header_offset" + 6 "plane" -- 28 bytes
//       -- the first 3 bits of "header_offset" is 7 (111)
//       -- the last 27 bits encode the integer offset to the child
//       -- The 6 "planes" store the bounding box
// A leaf node is 1 "header_offset" -- 4 bytes
//       -- the first 3 bits of "header_offset" is 3 (011)
//       -- the last 29 bits encode the leaf's offset in the global triangle index list

struct HTreeNode
{
    union
    {
        unsigned header_offset;
        float plane;
    }
}
```

Listing 5. *Compact BVH Node Definition*

```
struct CompactBVHNode
{
  //24 bytes
  float m[3];
  float M[3];

  //8 bytes
  struct
  {
    unsigned left : 28;
    unsigned l : 3;
    unsigned isLeftLeaf : 1;
    unsigned right : 28;
    unsigned L : 3;
    unsigned isRightLeaf : 1;
  };
```

```
};
```

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*

```
bool RayTracing::BVHTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
{
  //precompute the reciprocal of ray direction to avoid dividing later
  vec3 invdir = 1.f / ray.dir;
  //precompute the signs of ray directions
  const int sign[3] = { invdir.x < 0, invdir.y < 0, invdir.z < 0 };
  bool isLeaf = false;
  //initialize the offset to point to the root node
  int offset = 0;
  //initialize closest hit to infinitiy
  hitRecord.t = INFINITY;

  //discard rays that miss the scene bounding box
  if (BoxIntersection(ray.origin, invdir, sign, &bbox, hitRecord.t) == INFINITY) return false;
  //create traversal stack
  BVHStackItem stack[MAX_LEVEL];
  int stack_ptr = -1;
  while (true)
  {
    if (isLeaf)
    {
      // global triangle index list -- the last triangle associated with the leaf will have 1 at MSB
      int trioffset = bvh->nodes[offset].idx;
      for (int i = trioffset;; i++)
      {
        int triId = globalTriangleIndexList[i];
        //mask the lower 31 bits as triangle index
        //hitRecord records the closest hit
        TriangleIntersection(ray, model->tris[triId & 0x7fffffff], hitRecord);
        //check if the triangle is the last triangle
        if (triId < 0) break;
      }
    }
    else
    {
      //first 2 bits of the node indicates if left and right child are leaves: 0 -- not leaf  1 -- leaf
      char isLeftLeaf = bvh->nodes[offset].idx >> 30 & 3;
      char isRightLeaf = isLeftLeaf & 1;
```

```
      isLeftLeaf = isLeftLeaf >> 1;

      //next 30 bits of the internal node is the offset of the left child
      //compute right child offset -- if left child is leaf: offset by 1 / otherwise: offset by 13
      int first = offset + (bvh->nodes[offset].idx & 0x3fffffff), second = first + 12*(!isLeftLeaf) + 1;

      //"An Efficient and Robust Ray-Box Intersection Algorithm" -- Shirley et al.
      //returns the near hit with the bounding box
      //return infinity if there is not hit / hit at back / hit is farther than the closest hit (hitRecord.t)
      float first_t = BoxIntersection(ray.origin, invdir, sign, (aabb*)&bvh->nodes[offset + 1], hitRecord.t);
      float second_t = BoxIntersection(ray.origin, invdir, sign, (aabb*)&bvh->nodes[offset + 7], hitRecord.t);

      //if first child has farther hit than second child
      if (first_t > second_t)
      {
        //update offset and flag, visit second child first
        offset = second;
        isLeaf = isRightLeaf;
        //if hit first child, push it to stack (isLeaf flag, offset, near hit with bounding box)
        if (first_t != INFINITY) stack[++stack_ptr] = BVHStackItem(isLeftLeaf, first, first_t);
      }
      else
      {
        //check if first child is hit
        if (first_t != INFINITY)
        {
          offset = first;
          isLeaf = isLeftLeaf;
        }
        //the case when first_t == second_t == INFINITY, hit neither child, pop stack
        else goto pop;
        //if hit second child, push it to stack
        if (second_t != INFINITY) stack[++stack_ptr] = BVHStackItem(isRightLeaf, second, second_t);
      }
      continue;
    }
  pop:
    while (true)
    {
      //if stack is empty, terminate program and report hit or no hit
      if (stack_ptr == -1) return hitRecord.t < INFINITY;
      BVHStackItem item = stack[stack_ptr--];
      //discard node if near hit with its bounding box is not closer than current closest hit
      //otherwise update states, visit the node
      if (item.tmin < hitRecord.t)
      {
        offset = item.offset;
        isLeaf = item.isLeaf;
        break;
      }
    }
```

```
  }
  return false;
}
```

Listing 7. *Dual-Split Tree Traversal Kernel*

```
bool RayTracing::DSTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
{
  //make sure that if not shadowray, normalize first
  float tmin = 0;
  float tmax = INFINITY;

  // stores first 4 byte of a dual split tree node (header+offset)
  unsigned header_offset;
  //stores the highest 5 bits of the header
  unsigned header5;
  //the 6th header bit -- 1 : leaf or leaf contained  0 : is not/does not contain a leaf
  bool leafbit;
  //idx is the current node index
  unsigned idx = 0;
  hitRecord.t = INFINITY;

  const vec3 invdir = 1.f / ray.dir;
  const int dir_sgn[3] = { invdir.x < 0, invdir.y < 0, invdir.z < 0 };
  if (BoxIntersection(ray.origin, invdir, dir_sgn, &bbox, tmax) == INFINITY) return false;
  //make sure ray starts at origin
  if (tmin < 0) tmin = 0;

    // a StackItem stores the node's address, tmin and tmax
  StackItem stack[MAX_LEVEL];
  int stack_ptr = -1;

  while (true)
  {
    //get the header+offset
    header_offset = dst->nodes[idx].header_offset;
    header5 = header_offset >> 27;
    //get the leaf bit at the 26th bit
    leafbit = (header_offset >> 26) & 1;

    //the highest bit is 0 -- either a split node or a leaf
    if (header5 >> 4 == 0)
    {
      // if it is a leaf
      if (leafbit)
      {
        //get global triangle index list offset in lower 26 bits
        idx = header_offset & 0x3FFFFFF;
        goto leaf;
      }
```

```
    //split planes for the split node
    float cursplit[2];
    cursplit[0] = dst->nodes[idx + 1].plane;
    cursplit[1] = dst->nodes[idx + 2].plane;
    //get the left child offset relative to parent
    idx += header_offset & 0x3FFFFFF;

    //decode the splitting axis
    unsigned axis = header5 >> 2;
    //get ray direction sign at current axis
    unsigned sign = dir_sgn[axis];
    //diff: right child offset relative to left child
    //if left child is a leaf: 1  Otherwise: 3
    unsigned diff = header5 & 3;
    //ray-plane intersection with two splitting planes
    float ts1 = (cursplit[sign] - ray.origin[axis]) * invdir[axis];
    float ts2 = (cursplit[sign ^ 1] - ray.origin[axis]) * invdir[axis];
    // sign bit is multiplied by diff
    // if diff is 1
      //if sign is 1 -> sign ^ diff = 1 ^ 1 = 0 -> left child offset
      //otherwise -> sign ^ diff = 0 ^ 1 = 0 -> right child offset
    // if diff is 3
      //if sign is 1 -> sign ^ diff = 3 ^ 3 = 0 -> left child offset
      //otherwise -> sign ^ diff = 0 ^ 3 = 3 -> right child offset
    sign *= diff;
    // determine traversal order
    if (tmax >= ts2) // far child is intersected
    {
      float tnext = max(tmin, ts2);
      if (tmin <= ts1) // near child is also intersected
      {
        stack.stack[++stack_ptr] = StackItem(idx + (sign ^ diff), tnext, tmax);
        idx += sign;
        tmax = min(tmax, ts1);
      }
      else //near child is not intersected
      {
        idx += sign ^ diff;
        tmin = tnext;
      }
      continue;
    }
    else //far child is not intersected
    {
      if (tmin <= ts1) //near child is intersected
      {
        idx += sign;
        tmax = min(tmax, ts1);
        continue;
      }
      //neither child is intersected
```

```
        else goto pop;
      }
    }
    //Carve Node
    else
    {
      float cursplit[2];
      cursplit[0] = dst->nodes[idx + 1].plane;
      cursplit[1] = dst->nodes[idx + 2].plane;


      //carve type 1 at 30th and 29th bits
      //00-xy 01-xz 11-yz (dual axes carve node)
      //10-xx/yy/zz (single axis carve node)
      char carvetype1 = header5 >> 2 & 3;

      //carve type 2 at 28th and 27th bits
      //if node is single axis carve node -> gives axis
      //otherwise -> gives plane signs
      char carvetype2 = header5 & 3;

      //single axis carve node
      if (carvetype1 == 2)
      {
        float ts1, ts2;
        unsigned sign = dir_sgn[carvetype2];
        ts1 = (cursplit[sign] - ray.origin[carvetype2]) * invdir[carvetype2];
        ts2 = (cursplit[sign ^ 1] - ray.origin[carvetype2]) * invdir[carvetype2];
        //compute trimmed tmin and tmax
        tmax = min(ts1, tmax);
        tmin = max(ts2, tmin);
        //does not hit bounding volume, pop stack
        if (tmin > tmax) goto pop;
        //get offset bits
        int offset = header_offset & 0x3FFFFFF;

        //if containing a leaf
          //offset treated as global trianlge index list offset and go to leaf
        if (leafbit)
        {
          idx = offset;
          goto leaf;
        }
        //otherwise
          //offset treated as child offset relative to parent
        idx += offset;
        continue;
      }
      //dual axes carve node
      else
      {
```

```
    // get dual-axes
    unsigned axis1 = carvetype1 >> 1;
    unsigned axis2 = (carvetype1 & 1) + 1;

    // simplified corner carving test -- see Figure 4
    float t_min_0, t_min_1, t_max_0, t_max_1;
    t_min_0 = (cursplit[0] - ray.origin[axis1]) * invdir[axis1];
    t_max_0 = tmax;
    //plane sign in first axis is same as ray direction
    if (dir_sgn[axis1] == (carvetype2 >> 1))
    {
     t_max_0 = t_min_0;
     t_min_0 = tmin;
    }
    t_min_1 = (cursplit[1] - ray.origin[axis2]) * invdir[axis2];
    t_max_1 = tmax;
    if (dir_sgn[axis2] == (carvetype2 & 1))
    {
     t_max_1 = t_min_1;
     t_min_1 = tmin;
    }

    //compute trimmed tmin and tmax
    tmin = max(tmin, max(t_min_0, t_min_1));
    tmax = min(tmax, min(t_max_0, t_max_1));
    //does not hit bounding volume, pop stack
    if (tmin > tmax) goto pop;
    //if containing a leaf
     //offset treated as global trianlge index list offset and go to leaf
    int offset = header_offset & 0x3FFFFFF;
    if (leafbit)
    {
     idx = offset;
     goto leaf;
    }
    //otherwise
     //offset treated as child offset relative to parent
    idx += offset;
    continue;
   }
  }

 // leaf -- go through all triangles
 leaf:
  for (unsigned i = idx;;i++)
  {
   int triId = globalTriangleIndexList[i];
   TriangleIntersection(ray, model->tris[triId & 0x7FFFFFFF], hitRecord);
   if (triId < 0) break;
  }

```

```
  pop:
    while (true)
    {
      if (stack_ptr == -1) return hitRecord.t < INFINITY;
      StackItem item = stack.stack[stack_ptr--];
      idx = item.idx;
      tmin = item.tmin;
      if (tmin < hitRecord.t)
      {
        tmax = min(hitRecord.t, item.tmax);
        break;
      }
    }
  }
  return false;
}
```

Listing 8. *BIH Traversal Kernel*

```
bool RayTracing::BIHTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
{
  //make sure that if not shadowray, normalize first
  float tmin = 0;
  float tmax = INFINITY;

  // stores first 4 byte of a BIH node (header+offset)
  unsigned header_offset;
  unsigned header2; // the first two bits of the header, specifying leaf or splitting axis
  bool sizebit; // the 3rd bit of the header, specifying left child size
  //idx is the current node index
  unsigned idx = 0;
  hitRecord.t = INFINITY;

  const vec3 invdir = 1.f / ray.dir;
  const int dir_sgn[3] = { invdir.x < 0, invdir.y < 0, invdir.z < 0 };
  if (BoxIntersection(ray.origin, invdir, dir_sgn, &bbox, tmax) == INFINITY) return false;
  //make sure ray starts at origin
  if (tmin < 0) tmin = 0;

    // a StackItem stores the node's address, tmin and tmax
  StackItem stack[MAX_LEVEL];
  int stack_ptr = -1;

  while (true)
  {
    //get the header+offset
    header_offset = bih->nodes[idx].header_offset;

    header2 = header_offset >> 29;
    sizebit = header2 & 1;
```

```
        header2 >>= 1;

        // the first two bits are 11 -- a leaf
        if (header2 == 3)
        {
            //get global triangle index list offset in lower 30 bits
            idx = header_offset & 0x3FFFFFFF;
            for (unsigned i = idx;;i++)
            {
                int triId = globalTriangleIndexList[i];
                TriangleIntersection(ray, model->tris[triId & 0x7FFFFFFF], hitRecord);
                if (triId < 0) break;
            }
        }
        else // an internal node
        {
      //splitting planes for the node
      float cursplit[2];
      cursplit[0] = bih->nodes[idx + 1].plane;
      cursplit[1] = bih->nodes[idx + 2].plane;
            //get the left child offset relative to parent
      idx += header_offset & 0x1FFFFFFF;
      //get ray direction sign at current axis (stored in header2)
      unsigned sign = dir_sgn[header2];

      //ray-plane intersection with two splitting planes
      float ts1 = (cursplit[sign] - ray.origin[axis]) * invdir[axis];
      float ts2 = (cursplit[sign ^ 1] - ray.origin[axis]) * invdir[axis];

            //decode left child size from sizebit 0--left child has 3 words, 1--left child has 1 word
            unsigned diff = 3 >> sizebit;
            // sign bit is multiplied by diff
      // if diff is 1
        //if sign is 1 -> sign ^ diff = 1 ^ 1 = 0 -> left child offset
        //otherwise -> sign ^ diff = 0 ^ 1 = 0 -> right child offset
      // if diff is 3
        //if sign is 1 -> sign ^ diff = 3 ^ 3 = 0 -> left child offset
        //otherwise -> sign ^ diff = 0 ^ 3 = 3 -> right child offset
      sign *= diff;

      // determine traversal order
      if (tmax >= ts2) // far child is intersected
      {
        float tnext = max(tmin, ts2);
        if (tmin <= ts1) // near child is also intersected
        {
          stack.stack[++stack_ptr] = StackItem(idx + (sign ^ diff), tnext, tmax);
          idx += sign;
          tmax = min(tmax, ts1);
        }
        else //near child is not intersected
```

```
        {
          idx += sign ^ diff;
          tmin = tnext;
        }
        continue;
      }
      else //far child is not intersected
      {
        if (tmin <= ts1) //near child is intersected
        {
          idx += sign;
          tmax = min(tmax, ts1);
          continue;
        }
        //neither child is intersected
        else goto pop;
      }
        }

  pop:
    while (true)
    {
      if (stack_ptr == -1) return hitRecord.t < INFINITY;
      idx = item.idx;
      StackItem item = stack.stack[stack_ptr--];
      tmin = item.tmin;
      if (tmin < hitRecord.t)
      {
        tmax = min(hitRecord.t, item.tmax);
        break;
      }
    }
  }
  return false;
}
```

Listing 9. *H-Tree Traversal Kernel*

```
bool RayTracing::HTreeTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
{
  //make sure that if not shadowray, normalize first
  float tmin = 0;
  float tmax = INFINITY;

  // stores first 4 byte of a dual split tree node (header+offset)
  unsigned header_offset;
  //stores the first 3 bits of the header (node type)
  unsigned nodeType;
  //idx is the current node index
  unsigned idx = 0;
```

```
  hitRecord.t = INFINITY;

  const vec3 invdir = 1.f / ray.dir;
  const int dir_sgn[3] = { invdir.x < 0, invdir.y < 0, invdir.z < 0 };
  if (BoxIntersection(ray.origin, invdir, dir_sgn, &bbox, tmax) == INFINITY) return false;
  //make sure ray starts at origin
  if (tmin < 0) tmin = 0;

    // a StackItem stores the node's address, tmin and tmax
  StackItem stack[MAX_LEVEL];
  int stack_ptr = -1;

  while (true)
  {
    //get the header+offset
    header_offset = htree->nodes[idx].header_offset;
    nodeType = header_offset >> 29;
        if (nodeType == 3) //leaf
    {
            //get global triangle index list offset in lower 29 bits
            idx = header_offset & 0x1FFFFFFF;
            for (unsigned i = idx;;i++)
            {
                int triId = globalTriangleIndexList[i];
                TriangleIntersection(ray, model->tris[triId & 0x7FFFFFFF], hitRecord);
                if (triId < 0) break;
            }
        }
        else if (nodeType == 7) // BV6
        {
            // a bounding box intersection test which also considers tmin
            if (!BV6Intersection(ray, (aabb*)&htree->nodes[idx + 1], tmin, tmax)) goto pop;
            //add the child offset relative to parent
      idx += header_offset & 0x7FFFFFF;
      continue;
        }
        else if (nodeType <= 2) // SKD
        {
      //splitting planes for the node
      float cursplit[2];
      cursplit[0] = htree->nodes[idx + 1].plane;
      cursplit[1] = htree->nodes[idx + 2].plane;
            //get the left child offset relative to parent
      idx += header_offset & 0x7FFFFFF;
      //get ray direction sign at current axis (stored in nodeType)
      unsigned sign = dir_sgn[nodeType];

      //ray-plane intersection with two splitting planes
      float ts1 = (cursplit[sign] - ray.origin[axis]) * invdir[axis];
      float ts2 = (cursplit[sign ^ 1] - ray.origin[axis]) * invdir[axis];

```

```
            //decode left child size
            unsigned diff = 7 >> (header_offset >> 27 & 3);
            // sign bit is multiplied by diff
      // if diff is 1
        //if sign is 1 -> sign ^ diff = 1 ^ 1 = 0 -> left child offset
        //otherwise -> sign ^ diff = 0 ^ 1 = 0 -> right child offset
      // if diff is 3
        //if sign is 1 -> sign ^ diff = 3 ^ 3 = 0 -> left child offset
        //otherwise -> sign ^ diff = 0 ^ 3 = 3 -> right child offset
      sign *= diff;
                  // determine traversal order
      if (tmax >= ts2) // far child is intersected
      {
        float tnext = max(tmin, ts2);
        if (tmin <= ts1) // near child is also intersected
        {
          stack.stack[++stack_ptr] = StackItem(idx + (sign ^ diff), tnext, tmax);
          idx += sign;
          tmax = min(tmax, ts1);
        }
        else //near child is not intersected
        {
          idx += sign ^ diff;
          tmin = tnext;
        }
        continue;
      }
      else //far child is not intersected
      {
        if (tmin <= ts1) //near child is intersected
        {
          idx += sign;
          tmax = min(tmax, ts1);
          continue;
        }
        //neither child is intersected
        else goto pop;
      }
        }
        else //BV2
        {
            nodeType -= 4; //convert to bounding axis
      float cursplit[2];
      cursplit[0] = htree->nodes[idx + 1].plane;
      cursplit[1] = htree->nodes[idx + 2].plane;

            float ts1, ts2;
            unsigned sign = dir_sgn[nodeType];
            ts1 = (cursplit[sign] - ray.origin[nodeType]) * invdir[nodeType];
            ts2 = (cursplit[sign ^ 1] - ray.origin[nodeType]) * invdir[nodeType];

```

```
            //compute trimmed tmin and tmax
            tmax = min(ts1, tmax);
            tmin = max(ts2, tmin);
            //does not hit bounding volume, pop stack
            if (tmin > tmax) goto pop;

            idx += header_offset & 0x7FFFFFF;
        }

  pop:
    while (true)
    {
      if (stack_ptr == -1) return hitRecord.t < INFINITY;
      StackItem item = stack.stack[stack_ptr--];
      idx = item.idx;
      tmin = item.tmin;
      if (tmin < hitRecord.t)
      {
        tmax = min(hitRecord.t, item.tmax);
        break;
      }
    }
  }
  return false;
}
```

Listing 10. *Compact BVH Traversal Kernel*

```
bool RayTracing::CompactBVHTraversal(const Ray & ray, SurfaceHitRecord & hitRecord)
{
  vec3 invdir = 1.f / ray.dir;
  //idx is the current node index
  unsigned idx = 0;
  // using the notation in Compact BVH Storage for Ray Tracing and Photon Mapping (Fabianowski et al.)
  // a: tmin,  b: tmax
  float a = 0;
  float b = INFINITY;
  hitRecord.t = INFINITY;

  // update a and b to near and far hit with the scene bounding box
  // CompactBVHBoxIntersection: the same function as BVH's BoxIntersection
  // but passing one argument that gets the ray-box far hit to initialize b
  a = CompactBVHBoxIntersection(ray.origin, invdir, &bbox, hitRecord.t, b);
  //miss the scene bounding box
  if (a == INFINITY) return false;
  //make sure ray starts at origin
  if (a < 0) a = 0;
  // initialize stack
  CompactBVHStackItem stack[MAX_LEVEL];
  int stack_ptr = -1;
```

```
  while (true)
  {
    if (idx >> 31) // if the node is a leaf (the isLeaf boolean is packed with the index at the MSB)
    {
      //get lowest 28 bits as triangle offset
      int trioffset = idx & 0xFFFFFFF;
      for (int i = trioffset;; i++)
      {
        int triId = bvh->triIds[i];
        TriangleIntersection(ray, model->tris[triId & 0x7FFFFFFF], hitRecord);
        if (triId < 0) break;
      }
    }
    else
    {
      //get child assignment indicator l and L
      //please see Compact BVH Storage for Ray Tracing and Photon Mapping
        //http://www.fabianowski.eu/research/egie2009.pdf
      //each of ls and Ls is stored in 3 bits
      //for ls: the bits represent the xmin, ymin, zmin planes
      //for Ls: the bits represent the xmax, ymax, zmax planes
      //if a bit is 1, the corresponding plane belongs to left child
      //otherwise, it belongs to right child

      //cbvh is the pointer to the compact BVH
      unsigned ls = cbvh->nodes[idx].l;
      unsigned Ls = cbvh->nodes[idx].L;

      //initialize left and right child near and far hit values to parent values

      ///////////////////////////////////////////////////////////////////////////
      //the following part is identical to the the Algorithm 3 provided in the paper
      float al = a, ar = a;
      float bl = min(b, hitRecord.t), br = b;

      //m and M stores (xmin,ymin,zmin) and (xmax,ymax,zmax)
      vec3 t1 = vec3((cbvh->nodes[idx].m[0] - ray.origin[0]) * invdir[0],
        (cbvh->nodes[idx].m[1] - ray.origin[1]) * invdir[1],
        (cbvh->nodes[idx].m[2] - ray.origin[2]) * invdir[2]);

      vec3 t2 = vec3((cbvh->nodes[idx].M[0] - ray.origin[0]) * invdir[0],
        (cbvh->nodes[idx].M[1] - ray.origin[1]) * invdir[1],
        (cbvh->nodes[idx].M[2] - ray.origin[2]) * invdir[2]);
      vec3 t3 = INFINITY * invdir;

      // the "slab" test for compact BVH
      for (int k = 0; k < 3; k++)
      {
        float t1l, t1r, t2l, t2r;
        //if the min plane at axis k belongs to left child
        if ((ls >> k) & 1) { t1l = t1[k]; t1r = -t3[k]; }
```

```
        else { t1l = -t3[k]; t1r = t1[k]; }
        //if the min plane at axis k belongs to right child
        if ((Ls >> k) & 1) { t2l = t2[k]; t2r = t3[k]; }
        else { t2l = t3[k]; t2r = t2[k]; }
        al = max(al, min(t1l, t2l));
        bl = min(bl, max(t1l, t2l));
        ar = max(ar, min(t1r, t2r));
        br = min(br, max(t1r, t2r));
      }

      //if hit both children
      if (al <= bl && ar <= br)
      {
        //always visit the closer child first
        if (al <= ar)
        {
          a = al;
          b = bl;
          //push far child to stack
          stack[++stack_ptr] = DSStep(cbvh->nodes[idx].right | (cbvh->nodes[idx].isRightLeaf << 31), ar, br);
          //update current index
          idx = cbvh->nodes[idx].left | (cbvh->nodes[idx].isLeftLeaf << 31);
        }
        else
        {
          a = ar;
          b = br;
          stack[++stack_ptr] = DSStep(cbvh->nodes[idx].left | (cbvh->nodes[idx].isLeftLeaf << 31), al, bl);
          idx = cbvh->nodes[idx].right | (cbvh->nodes[idx].isRightLeaf << 31);
        }
        continue;
      }
      //only hit left child
      else if (al <= bl)
      {
        a = al;
        b = bl;
        idx = cbvh->nodes[idx].left | (cbvh->nodes[idx].isLeftLeaf << 31);
        continue;
      }
      //only hit right child
      else if (ar <= br)
      {
        a = ar;
        b = br;
        idx = cbvh->nodes[idx].right | (cbvh->nodes[idx].isRightLeaf << 31);
        continue;
      }
      //the above part is identical to the the Algorithm 3 provided in the paper
      //////////////////////////////////////////////////////////////////////////
    }
```

```
  pop:
    while (true)
    {
      if (stack_ptr == -1) return hitRecord.t < INFINITY;
      CompactBVHStackItem item = stack[stack_ptr--];
      if (item.tmin < hitRecord.t)
      {
        idx = item.idx;
        a = item.tmin;
        b = item.tmax;
        break;
      }
    }
  }
  return false;
}
`
```

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.