# 4.3 Bounding Volume Hierarchies

Bounding volume hierarchies (BVHs) are an approach for ray intersection acceleration based on primitive subdivision, where the primitives are partitioned into a hierarchy of disjoint sets. (In contrast, spatial subdivision generally partitions space into a hierarchy of disjoint sets.) Figure 4.2 shows a bounding volume hierarchy for a simple scene. Primitives are stored in the leaves, and each node stores a bounding box of the primitives in the nodes beneath it. Thus, as a ray traverses through the tree, any time it doesn't intersect a node's bounds, the subtree beneath that node can be skipped.

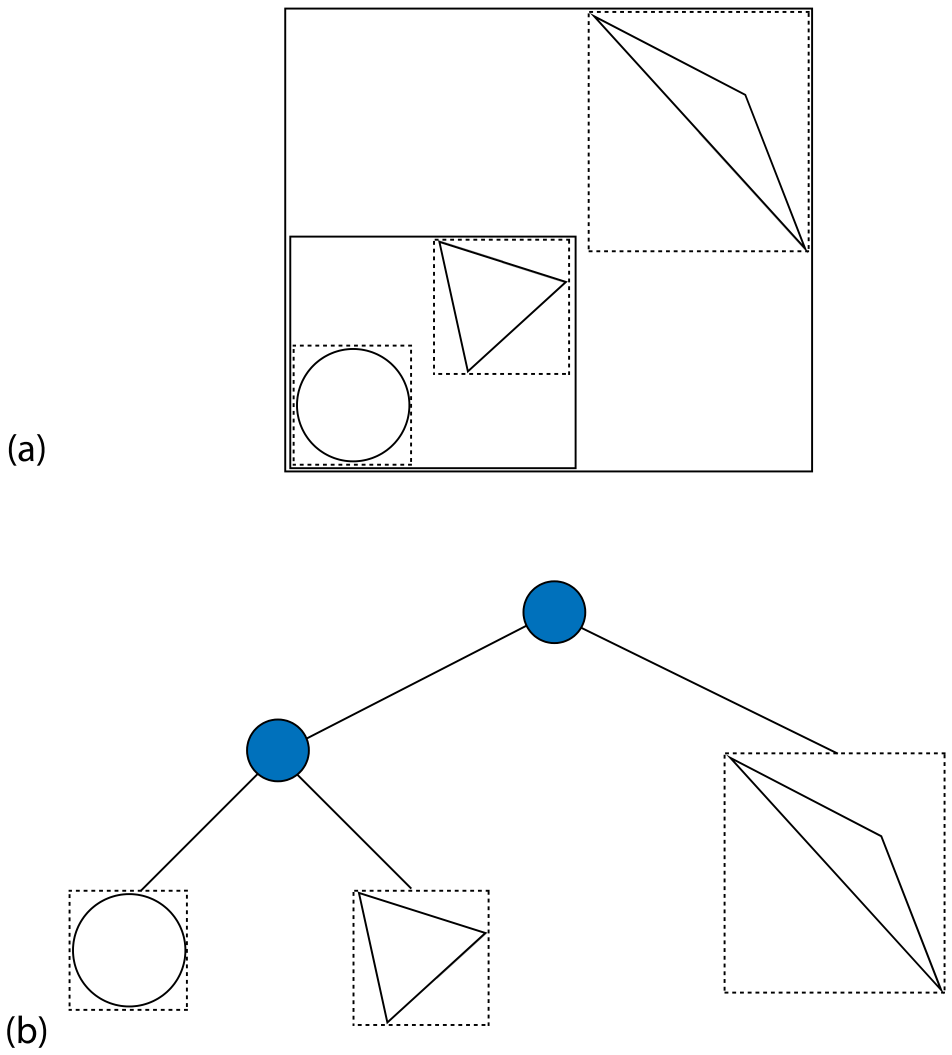



Figure 4.2: Bounding Volume Hierarchy for a Simple Scene. (a) A small collection of primitives, with bounding boxes shown by dashed lines. The primitives are aggregated based on proximity; here, the sphere and the equilateral triangle are bounded by another bounding box before being bounded by a bounding box that encompasses the entire scene (both shown in solid lines). (b) The corresponding bounding volume hierarchy. The root node holds the bounds of the entire scene. Here, it has two children, one storing a bounding box that encompasses the sphere and equilateral triangle (that in turn has those primitives as its children) and the other storing the bounding box that holds the skinny triangle.

One property of primitive subdivision is that each primitive appears in the hierarchy only once. In contrast, a primitive may overlap multiple spatial regions with spatial subdivision and thus may be tested for intersection multiple times as the ray passes through them.[†] Another implication of this property is that the amount of memory needed to represent the primitive subdivision hierarchy is bounded. For a binary BVH that stores a single primitive in each leaf, the total number of nodes is $2n - 1$, where $n$ is the number of primitives. There will be $n$ leaf nodes and $n - 1$ interior nodes. If leaves store multiple primitives, fewer nodes are needed.

BVHs are more efficient to build than kd-trees, which generally deliver slightly faster ray intersection tests than BVHs but take substantially longer to build. On the other hand, BVHs are generally more numerically robust and less prone to missed intersections due to round-off errors than kd-trees are.

The BVH accelerator, `BVHAccel`, is defined in accelerators/bvh.h and accelerators/bvh.cpp. In addition to the primitives to be stored and the maximum number of primitives that can be in any leaf node, its constructor takes an enumerant value that describes which of four algorithms to use when partitioning primitives to build the tree. The default, `SAH`, indicates that an algorithm based on the "surface area heuristic," discussed in Section 4.3.2, should be used. An alternative, `HLBVH`, which is discussed in Section 4.3.3, can be constructed more efficiently (and more easily parallelized), but it doesn't build trees that are as effective as `SAH`. The remaining two approaches use even less computation to build the tree but create fairly low-quality trees.

*<<BVHAccel Public Types>>=*
```
enum class SplitMethod { SAH, HLBVH, Middle, EqualCounts };
```

*<<BVHAccel Method Definitions>>=*
```
BVHAccel::BVHAccel(const std::vector<std::shared_ptr<Primitive>> &p,
         int maxPrimsInNode, SplitMethod splitMethod)
     : maxPrimsInNode(std::min(255, maxPrimsInNode)), primitives(p),
       splitMethod(splitMethod) {
    if (primitives.size() == 0)
        return;
    <<Build BVH from primitives>> ⊞
}
```

*<<BVHAccel Private Data>>=*
```
const int maxPrimsInNode;
const SplitMethod splitMethod;
std::vector<std::shared_ptr<Primitive>> primitives;
```

## 4.3.1 BVH Construction

There are three stages to BVH construction in the implementation here. First, bounding information about each primitive is computed and stored in an array that will be used during tree construction. Next, the tree is built using the algorithm choice encoded in `splitMethod`. The result is a binary tree where each interior node holds pointers to its children and each leaf node holds references to one or more primitives. Finally, this tree is converted to a more compact (and thus more efficient) pointerless representation for use during rendering. (The implementation is more straightforward with this approach, versus computing the pointerless representation directly during tree construction, which is also possible.)

*<<Build BVH from `primitives`>>=*
```
<<Initialize primitiveInfo array for primitives>> ⊞
<<Build BVH tree for primitives using primitiveInfo>> ⊞
<<Compute representation of depth-first traversal of BVH tree>> ⊞
```

For each primitive to be stored in the BVH, we store the centroid of its bounding box, its complete bounding box, and its index in the `primitives` array in an instance of the `BVHPrimitiveInfo` structure.

*<<Initialize `primitiveInfo` array for primitives>>=*
```
std::vector<BVHPrimitiveInfo> primitiveInfo(primitives.size());
for (size_t i = 0; i < primitives.size(); ++i)
    primitiveInfo[i] = { i, primitives[i]->WorldBound() };
```

*<<BVHAccel Local Declarations>>=*
```
struct BVHPrimitiveInfo {
    BVHPrimitiveInfo(size_t primitiveNumber, const Bounds3f &bounds)
        : primitiveNumber(primitiveNumber), bounds(bounds),
          centroid(.5f * bounds.pMin + .5f * bounds.pMax) { }
    size_t primitiveNumber;
    Bounds3f bounds;
    Point3f centroid;
};
```

Hierarchy construction can now begin. If the HLBVH construction algorithm has been selected, `HLBVHBuild()` is called to build the tree. The other three construction algorithms are all handled by `recursiveBuild()`. The initial calls to these functions are passed all of the primitives to be stored in the tree. They return a pointer to the root of the tree, which is represented with the BVHBuildNode structure. Tree nodes should be allocated with the provided `MemoryArena`, and the total number created should be stored in `*totalNodes`.

One important side effect of the tree construction process is that a new array of pointers to primitives is returned via the `orderedPrims` parameter; this array stores the primitives ordered so that the primitives in leaf nodes occupy contiguous ranges in the array. It is swapped with the original `primitives` array after tree construction.

*<<Build BVH tree for primitives using `primitiveInfo`>>=*

```
MemoryArena arena(1024 * 1024);
int totalNodes = 0;
std::vector<std::shared_ptr<Primitive>> orderedPrims;
BVHBuildNode *root;
if (splitMethod == SplitMethod::HLBVH)
    root = HLBVHBuild(arena, primitiveInfo, &totalNodes, orderedPrims);
else
    root = recursiveBuild(arena, primitiveInfo, 0, primitives.size(),
                          &totalNodes, orderedPrims);
primitives.swap(orderedPrims);
```

Each `BVHBuildNode` represents a node of the BVH. All nodes store a `Bounds3f`, which represents the bounds of all of the children beneath the node. Each interior node stores pointers to its two children in `children`. Interior nodes also record the coordinate axis along which primitives were partitioned for distribution to their two children; this information is used to improve the performance of the traversal algorithm. Leaf nodes need to record which primitive or primitives are stored in them; the elements of the `BVHAccel::primitives` array from the offset `firstPrimOffset` up to but not including **`firstPrimOffset` + `nPrimitives`** are the primitives in the leaf. (Hence the need for reordering the primitives array, so that this representation can be used, rather than, for example, storing a variable-sized array of primitive indices at each leaf node.)

*<<BVHAccel Local Declarations>>+=*

```
struct BVHBuildNode {
    <<BVHBuildNode Public Methods>>
    Bounds3f bounds;
    BVHBuildNode *children[2];
    int splitAxis, firstPrimOffset, nPrimitives;
};
```

We'll distinguish between leaf and interior nodes by whether their children pointers have the value `nullptr` or not, respectively.

*<<BVHBuildNode Public Methods>>=*

```
void InitLeaf(int first, int n, const Bounds3f &b) {
    firstPrimOffset = first;
    nPrimitives = n;
    bounds = b;
    children[0] = children[1] = nullptr;
}
```

The `InitInterior()` method requires that the two children nodes already have been created, so that their pointers can be passed in. This requirement makes it easy to compute the bounds of the interior node, since the children bounds are immediately available.

*<<BVHBuildNode Public Methods>>+=*

```
void InitInterior(int axis, BVHBuildNode *c0, BVHBuildNode *c1) {
    children[0] = c0;
    children[1] = c1;
    bounds = Union(c0->bounds, c1->bounds);
    splitAxis = axis;
    nPrimitives = 0;
}
```

In addition to a `MemoryArena` used for allocating nodes and the array of `BVHPrimitiveInfo` structures, `recursiveBuild()` takes as parameters the range $[\mathbf{start}, \mathbf{end})$. It is responsible for returning a BVH for the subset of primitives represented by the range from `primitiveInfo[start]` up to and including `primitiveInfo[end-1]`. If this range covers only a single primitive, then the recursion has bottomed out and a leaf node is created. Otherwise, this method partitions the elements of the array in that range using one of the partitioning algorithms and reorders the array elements in the range accordingly, so that the ranges from $[\mathbf{start}, \mathbf{mid})$ and $[\mathbf{mid}, \mathbf{end})$ represent the partitioned subsets. If the partitioning is successful, these two primitive sets are in turn passed to recursive calls that will themselves return pointers to nodes for the two children of the current node.

`totalNodes` tracks the total number of BVH nodes that have been created; this number is used so that exactly the right number of the more compact `LinearBVHNode`s can be allocated later. Finally, the `orderedPrims` array is used to store primitive references as primitives are stored in leaf nodes of the tree. This array is initially empty; when a leaf node is created, `recursiveBuild()` adds the primitives that overlap it to the end of the array, making it possible for leaf nodes to just store an offset into this array and a primitive count to represent the set of primitives that overlap it. Recall that when tree construction is finished, `BVHAccel::primitives` is replaced with the ordered primitives array created here.

*<<BVHAccel Method Definitions>>+=*

```
BVHBuildNode *BVHAccel::recursiveBuild(MemoryArena &arena,
        std::vector<BVHPrimitiveInfo> &primitiveInfo, int start,
        int end, int *totalNodes,
        std::vector<std::shared_ptr<Primitive>> &orderedPrims) {
    BVHBuildNode *node = arena.Alloc<BVHBuildNode>();
    (*totalNodes)++;
    <<Compute bounds of all primitives in BVH node>> ⊞
    int nPrimitives = end - start;
    if (nPrimitives == 1) {
        <<Create leaf BVHBuildNode>> ⊞
    } else {
        <<Compute bound of primitive centroids, choose split dimension dim>> ⊞
        <<Partition primitives into two sets and build children>> ⊞
    }
    return node;
}
```

*<<Compute bounds of all primitives in BVH node>>=*

```
Bounds3f bounds;
for (int i = start; i < end; ++i)
    bounds = Union(bounds, primitiveInfo[i].bounds);
```

At leaf nodes, the primitives overlapping the leaf are appended to the `orderedPrims` array and a leaf node object is initialized.

*<<Create leaf BVHBuildNode>>=*

```
int firstPrimOffset = orderedPrims.size();
for (int i = start; i < end; ++i) {
    int primNum = primitiveInfo[i].primitiveNumber;
    orderedPrims.push_back(primitives[primNum]);
}
node->InitLeaf(firstPrimOffset, nPrimitives, bounds);
return node;
```

For interior nodes, the collection of primitives must be partitioned between the two children subtrees. Given $n$ primitives, there are in general $2^(n-1)-2$ possible ways to partition them into two nonempty groups. In practice when building BVHs, one generally considers partitions along a coordinate axis, meaning that there are about $3n$ candidate partitions. (Along each axis, each primitive may be put into the first partition or the second partition.)

Here, we choose just one of the three coordinate axes to use in partitioning the primitives. We select the axis associated with the largest extent when projecting the bounding box centroid for the current set of primitives. (An alternative would be to try all three axes and select the one that gave the best result, but in practice this approach works well.) This approach gives good partitions in many scenes; Figure 4.3 illustrates the strategy.

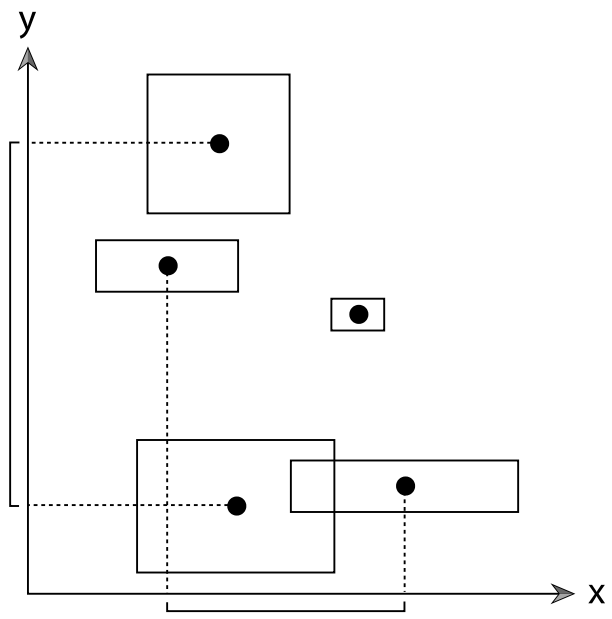



Figure 4.3: Choosing the Axis along Which to Partition Primitives. The `BVHAccel` chooses an axis along which to partition the primitives based on which axis has the largest range of the centroids of the primitives' bounding boxes. Here, in two dimensions, their extent is largest along the $y$ axis (filled points on the axes), so the primitives will be partitioned in $y$.

The general goal in partitioning here is to select a partition of primitives that doesn't have too much overlap of the bounding boxes of the two resulting primitive sets—if there is substantial overlap then it will more frequently be necessary to traverse both children subtrees when traversing the tree, requiring more computation than if it had been possible to more effectively prune away collections of primitives. This idea of finding effective primitive partitions will be made more rigorous shortly, in the discussion of the surface area heuristic.

*<<Compute bound of primitive centroids, choose split dimension `dim`>>=*
```
Bounds3f centroidBounds;
for (int i = start; i < end; ++i)
    centroidBounds = Union(centroidBounds, primitiveInfo[i].centroid);
int dim = centroidBounds.MaximumExtent();
```

If all of the centroid points are at the same position (i.e., the centroid bounds have zero volume), then recursion stops and a leaf node is created with the primitives; none of the splitting methods here is effective in that (unusual) case. The primitives are otherwise partitioned using the chosen method and passed to two recursive calls to `recursiveBuild()`.

*<<Partition primitives into two sets and build children>>=*
```
int mid = (start + end) / 2;
if (centroidBounds.pMax[dim] == centroidBounds.pMin[dim]) {
    <<Create leaf BVHBuildNode>> ⊞
} else {
    <<Partition primitives based on splitMethod>> ⊞
    node->InitInterior(dim,
                       recursiveBuild(arena, primitiveInfo, start, mid,
                                      totalNodes, orderedPrims),
                       recursiveBuild(arena, primitiveInfo, mid, end,
                                      totalNodes, orderedPrims));
}
```

We won't include the code fragment <<*Partition primitives based on `splitMethod`*>> here; it just uses the value of `BVHAccel::splitMethod` to determine which primitive partitioning scheme to use. These three schemes will be described in the following few pages.

A simple `splitMethod` is `Middle`, which first computes the midpoint of the primitives' centroids along the splitting axis. This method is implemented in the fragment <<*Partition primitives through node's midpoint*>>. The primitives are classified into the two sets, depending on whether their centroids are above or below the midpoint. This partitioning is easily done with the `std::partition()` C++ standard library function, which takes a range of elements in an array and a comparison function and orders the elements in the array so that all of the elements that return `true` for the given predicate function appear in the range before those that

return `false` for it.[†] `std::partition()` returns a pointer to the first element that had a `false` value for the predicate, which is converted into an offset into the `primitiveInfo` array so that we can pass it to the recursive call. Figure 4.4 illustrates this approach, including cases where it does and does not work well.

If the primitives all have large overlapping bounding boxes, this splitting method may fail to separate the primitives into two groups. In that case, execution falls through to the `SplitMethod::EqualCounts` approach to try again.

*<<Partition primitives through node's midpoint>>=*

```
Float pmid = (centroidBounds.pMin[dim] + centroidBounds.pMax[dim]) / 2;
BVHPrimitiveInfo *midPtr =
    std::partition(&primitiveInfo[start], &primitiveInfo[end-1]+1,
        [dim, pmid](const BVHPrimitiveInfo &pi) {
            return pi.centroid[dim] < pmid;
        });
mid = midPtr - &primitiveInfo[0];
if (mid != start && mid != end)
    break;
```

When `splitMethod` is `SplitMethod::EqualCounts`, the <<*Partition primitives into equally sized subsets*>> fragment runs. It partitions the primitives into two equal-sized subsets such that the first half of the $n$ of them are the $n/2$ with smallest centroid coordinate values along the chosen axis, and the second half are the ones with the largest centroid coordinate values. While this approach can sometimes work well, the case in Figure 4.4(b) is one where this method also fares poorly.

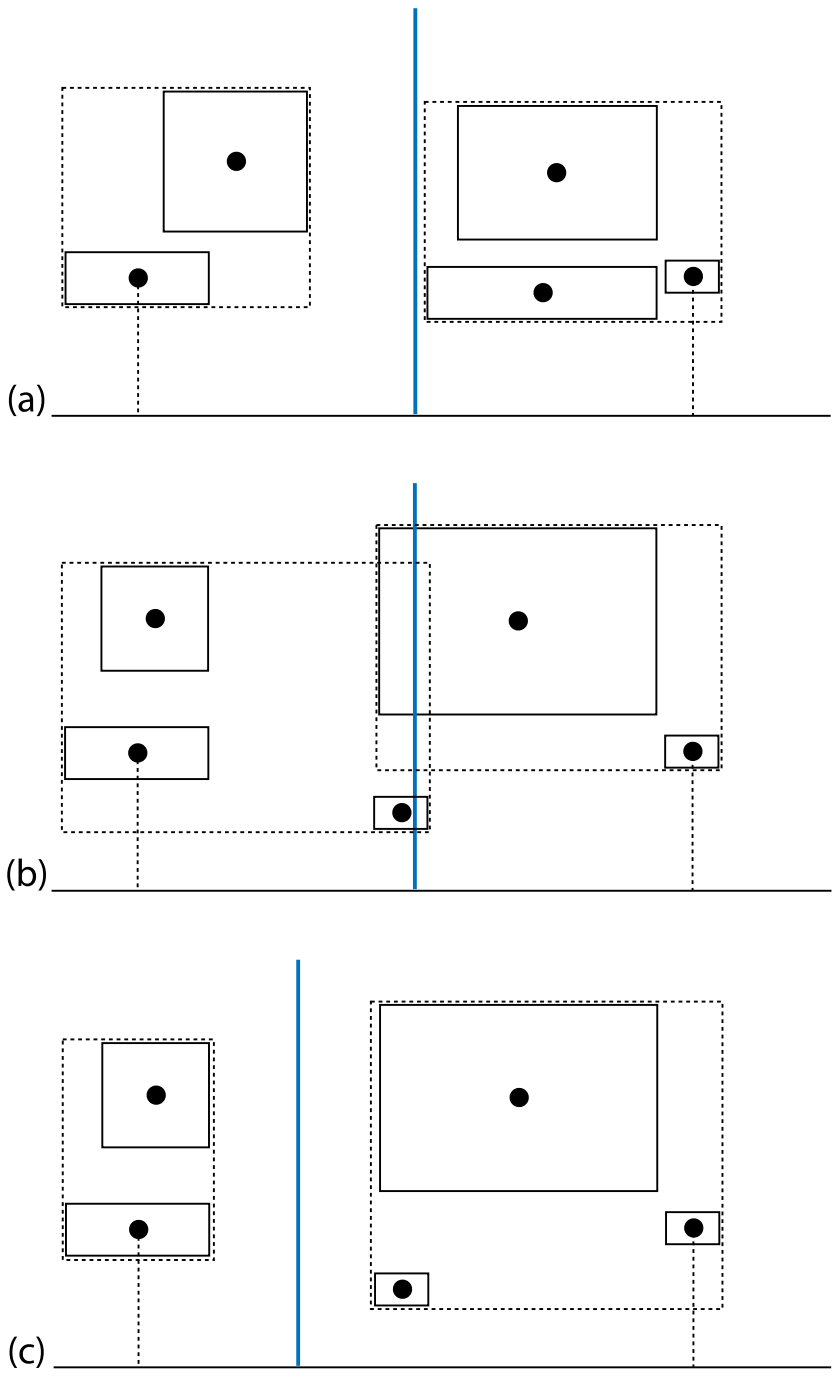



Figure 4.4: Splitting Primitives Based on the Midpoint of Centroids on an Axis. (a) For some distributions of primitives, such as the one shown here, splitting based on the midpoint of the centroids along the chosen axis (thick blue line) works well. (The bounding boxes of the two resulting primitive groups are shown with dashed lines.) (b) For distributions like this one, the midpoint is a suboptimal choice; the two resulting bounding boxes overlap substantially. (c) If the same group of primitives from (b) is instead split along the line shown here, the resulting bounding boxes are smaller and don't overlap at all, leading to better performance when rendering.

This scheme is also easily implemented with a standard library call, `std::nth_element()`. It takes a start, middle, and ending pointer as well as a comparison function. It orders the array so that the element at the middle pointer is the one that would be there if the array was fully sorted, and such that all of the elements before the middle one compare to less than the middle element and all of the elements after it compare to greater than it. This ordering can be done in $O(n)$ time, with $n$ the number of elements, which is more efficient than the $O(n \log n)$ of completely sorting the array.

*<<Partition primitives into equally sized subsets>>=*

```
mid = (start + end) / 2;
std::nth_element(&primitiveInfo[start], &primitiveInfo[mid],
                 &primitiveInfo[end-1]+1,
    [dim](const BVHPrimitiveInfo &a, const BVHPrimitiveInfo &b) {
        return a.centroid[dim] < b.centroid[dim];
    });
```

## 4.3.2 The Surface Area Heuristic

The two primitive partitioning approaches above can work well for some distributions of primitives, but they often choose partitions that perform poorly in practice, leading to more nodes of the tree being visited by rays and hence unnecessarily inefficient ray–primitive intersection computations at rendering time. Most of the best current algorithms for building acceleration structures for ray-tracing are based on the "surface area heuristic" (SAH), which provides a well-grounded cost model for answering questions like "which of a number of partitions of primitives will lead to a better BVH for ray–primitive intersection tests?," or "which of a number of possible positions to split space in a spatial subdivision scheme will lead to a better acceleration structure?"

The SAH model estimates the computational expense of performing ray intersection tests, including the time spent traversing nodes of the tree and the time spent on ray–primitive intersection tests for a particular partitioning of primitives. Algorithms for building acceleration structures can then follow the goal of minimizing total cost. Typically, a greedy algorithm is used that minimizes the cost for each single node of the hierarchy being built individually.

The ideas behind the SAH cost model are straightforward: at any point in building an adaptive acceleration structure (primitive subdivision or spatial subdivision), we could just create a leaf node for the current region and geometry. In that case, any ray that passes through this region will be tested against all of the overlapping primitives and will incur a cost of

$$\sum_{i=1}^{N} t_{\text{isect}}(i),$$

where $N$ is the number of primitives and $t_{\text{isect}}(i)$ is the time to compute a ray–object intersection with the $i$ th primitive.

The other option is to split the region. In that case, rays will incur the cost

$$c(A, B) = t_{\text{trav}} + p_A \sum_{i=1}^{N_A} t_{\text{isect}}(a_i) + p_B \sum_{i=1}^{N_B} t_{\text{isect}}(b_i), \tag{4.1}$$

where $t_{\text{trav}}$ is the time it takes to traverse the interior node and determine which of the children the ray passes through, $p_A$ and $p_B$ are the probabilities that the ray passes through each of the child nodes (assuming binary subdivision), $a_i$ and $b_i$ are the indices of primitives in the two children nodes, and $N_A$ and $N_B$ are the number of primitives that overlap the regions of the two child nodes, respectively. The choice of how primitives are partitioned affects both the values of the two probabilities as well as the set of primitives on each side of the split.

In `pbrt`, we will make the simplifying assumption that $t_{\text{isect}}(i)$ is the same for all of the primitives; this assumption is probably not too far from reality, and any error that it introduces doesn't seem to affect the performance of accelerators very much. Another possibility would be to add a method to `Primitive` that returned an estimate of the number of CPU cycles its intersection test requires.

The probabilities $p_A$ and $p_B$ can be computed using ideas from geometric probability. It can be shown that for a convex volume $A$ contained in another convex volume $B$, the conditional probability that a uniformly distributed random ray passing through $B$ will also pass through $A$ is the ratio of their surface areas, $s_A$ and $s_B$:

$$p(A|B) = \frac{s_A}{s_B}.$$

Because we are interested in the cost for rays passing through the node, we can use this result directly. Thus, if we are considering refining a region of space $A$ such that there are two new subregions with bounds $B$ and $C$ (Figure 4.5), the probability that a ray passing through $A$ will also pass through either of the subregions is easily computed.

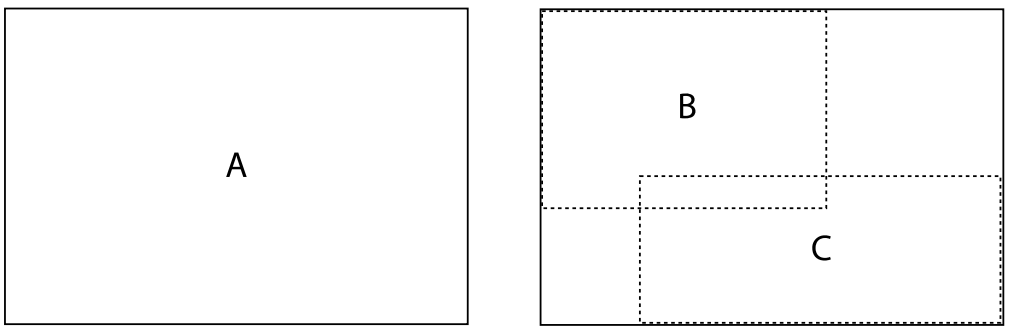



Figure 4.5: If a node of the bounding hierarchy with surface area $s_A$ is split into two children with surface areas $s_B$ and $s_C$, the probabilities that a ray passing through $A$ also passes through $B$ and $C$ are given by $s_B/s_A$ and $s_C/s_A$, respectively.

When `splitMethod` has the value `SplitMethod::SAH`, the SAH is used for building the BVH; a partition of the primitives along the chosen axis that gives a minimal SAH cost estimate is found by considering a number of candidate partitions. (This is the default `SplitMethod`, and it creates the most efficient trees for rendering.) However, once it has refined down to a small handful of primitives, the implementation switches over to partitioning into equally sized subsets. The incremental computational cost for applying the SAH at this point isn't worthwhile.

*<<Partition primitives using approximate SAH>>=*
```
if (nPrimitives <= 4) {
    <<Partition primitives into equally sized subsets>> ⊞
} else {
    <<Allocate BucketInfo for SAH partition buckets>> ⊞
    <<Initialize BucketInfo for SAH partition buckets>> ⊞
    <<Compute costs for splitting after each bucket>> ⊞
    <<Find bucket to split at that minimizes SAH metric>> ⊞
    <<Either create leaf or split primitives at selected SAH bucket>> ⊞
}
```

Rather than exhaustively considering all $2n$ possible partitions along the axis, computing the SAH for each to select the best, the implementation here instead divides the range along the axis into a small number of buckets of equal extent. It then only considers partitions at bucket boundaries. This approach is more efficient than considering all partitions while usually still producing partitions that are nearly as effective. This idea is illustrated in Figure 4.6.

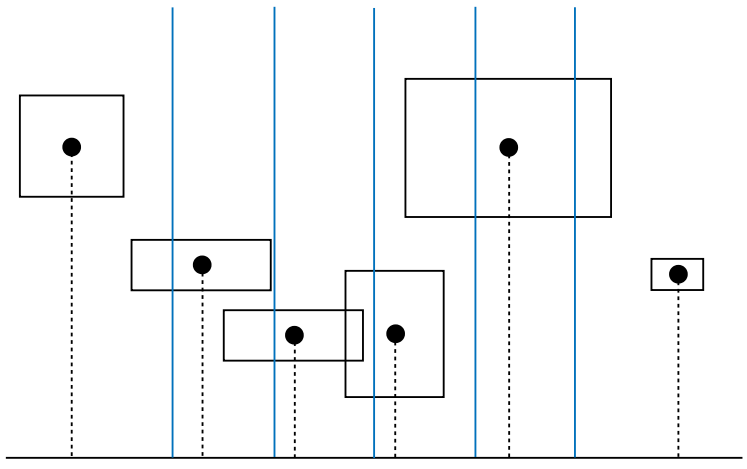


Figure 4.6: Choosing a Splitting Plane with the Surface Area Heuristic for BVHs. The projected extent of primitive bounds centroids is projected onto the chosen split axis. Each primitive is placed in a bucket along the axis based on the centroid of its bounds. The implementation then estimates the cost for splitting the primitives along the planes along each of the bucket boundaries (solid blue lines); whichever one gives the minimum cost per the surface area heuristic is selected.

*<<Allocate BucketInfo for SAH partition buckets>>=*

```
constexpr int nBuckets = 12;
struct BucketInfo {
    int count = 0;
    Bounds3f bounds;
};
BucketInfo buckets[nBuckets];
```

For each primitive in the range, we determine the bucket that its centroid lies in and update the bucket's bounds to include the primitive's bounds.

*<<Initialize BucketInfo for SAH partition buckets>>=*

```
for (int i = start; i < end; ++i) {
    int b = nBuckets *
        centroidBounds.Offset(primitiveInfo[i].centroid)[dim];
    if (b == nBuckets) b = nBuckets - 1;
    buckets[b].count++;
    buckets[b].bounds = Union(buckets[b].bounds, primitiveInfo[i].bounds);
}
```

For each bucket, we now have a count of the number of primitives and the bounds of all of their respective bounding boxes. We want to use the SAH to estimate the cost of splitting at each of the bucket boundaries. The fragment below loops over all of the buckets and initializes the `cost[i]` array to store the estimated SAH cost for splitting after the `i`th bucket. (It doesn't consider a split after the last bucket, which by definition wouldn't split the primitives.)

We arbitrarily set the estimated intersection cost to 1, and then set the estimated traversal cost to $1/8$. (One of the two of them can always be set to 1 since it is the relative, rather than absolute, magnitudes of the estimated traversal and intersection costs that determine their effect.) While the absolute amount of computation for node traversal—a ray–bounding box intersection—is only slightly less than the amount of computation needed to intersect a ray with a shape, ray–primitive intersection tests in `pbrt` go through two virtual function calls, which add significant overhead, so we estimate their cost here as eight times more than the ray–box intersection.

This computation has $O(n^2)$ complexity in the number of buckets, though a linear-time implementation based on a forward scan over the buckets and a backward scan over the buckets that incrementally compute and store bounds and counts is possible. For the small $n$ here, the performance impact is generally acceptable, though for a more highly optimized renderer addressing this inefficiency may be worthwhile.

*<<Compute costs for splitting after each bucket>>=*

```
Float cost[nBuckets - 1];
for (int i = 0; i < nBuckets - 1; ++i) {
    Bounds3f b0, b1;
    int count0 = 0, count1 = 0;
    for (int j = 0; j <= i; ++j) {
        b0 = Union(b0, buckets[j].bounds);
        count0 += buckets[j].count;
    }
    for (int j = i+1; j < nBuckets; ++j) {
        b1 = Union(b1, buckets[j].bounds);
        count1 += buckets[j].count;
    }
    cost[i] = .125f + (count0 * b0.SurfaceArea() +
                       count1 * b1.SurfaceArea()) / bounds.SurfaceArea();
}
```

Given all of the costs, a linear scan through the `cost` array finds the partition with minimum cost.

*<<Find bucket to split at that minimizes SAH metric>>=*

```
Float minCost = cost[0];
int minCostSplitBucket = 0;
for (int i = 1; i < nBuckets - 1; ++i) {
    if (cost[i] < minCost) {
        minCost = cost[i];
        minCostSplitBucket = i;
    }
}
```

If the chosen bucket boundary for partitioning has a lower estimated cost than building a node with the existing primitives or if more than the maximum number of primitives allowed in a node is present, the `std::partition()` function is used to do the work of reordering nodes in the `primitiveInfo` array. Recall from its usage earlier that this function ensures that all elements of the array that return `true` from the given predicate appear before those that return `false` and that it returns a pointer to the first element where the predicate returns `false`. Because we arbitrarily set the estimated intersection cost to 1 previously, the estimated cost for just creating a leaf node is equal to the number of primitives, `nPrimitives`.

*<<Either create leaf or split primitives at selected SAH bucket>>=*

```
Float leafCost = nPrimitives;
if (nPrimitives > maxPrimsInNode || minCost < leafCost) {
    BVHPrimitiveInfo *pmid = std::partition(&primitiveInfo[start],
        &primitiveInfo[end-1]+1,
        [=](const BVHPrimitiveInfo &pi) {
            int b = nBuckets * centroidBounds.Offset(pi.centroid)[dim];
            if (b == nBuckets) b = nBuckets - 1;
            return b <= minCostSplitBucket;
        });
    mid = pmid - &primitiveInfo[0];
} else {
    <<Create leaf BVHBuildNode>> ⊞
}
```

## 4.3.3 Linear Bounding Volume Hierarchies

While building bounding volume hierarchies using the surface area heuristic gives very good results, that approach does have two disadvantages: first, many passes are taken over the scene primitives to compute the SAH costs at all of the levels of the tree. Second, top-down BVH construction is difficult to parallelize well: the most obvious parallelization approach—performing parallel construction of independent subtrees—suffers from limited independent work until the top few levels of the tree have been built, which in turn inhibits parallel scalability. (This second issue is particularly an issue on GPUs, which perform poorly if massive parallelism isn't available.)

*Linear bounding volume hierarchies* (LBVHs) were developed to address these issues. With LBVHs, the tree is built with a small number of lightweight passes over the primitives; tree construction time is linear in the number of primitives. Further, the algorithm quickly partitions the primitives into clusters that can be processed independently. This processing can be fairly easily parallelized and is well suited to GPU implementation.

The key idea behind LBVHs is to turn BVH construction into a sorting problem. Because there's no single ordering function for sorting multi-dimensional data, LBVHs are based on *Morton codes*, which map nearby points in $n$ dimensions to nearby points along the 1D line, where there is an obvious ordering function. After the primitives have been sorted, spatially nearby clusters of primitives are in contiguous segments of the sorted array.

Morton codes are based on a simple transformation: given $n$-dimensional integer coordinate values, their Morton-coded representation is found by interleaving the bits of the coordinates in base 2. For example, consider a 2D coordinate $(x, y)$ where the bits of $x$ and $y$ are denoted by $x_i$ and $y_i$. The corresponding Morton-coded value is

$$\cdots y_3\, x_3\, y_2\, x_2\, y_1\, x_1\, y_0\, x_0 .$$

Figure 4.7 shows a plot of the 2D points in Morton order—note that they are visited along a path that follows a reversed "z" shape. (The Morton path is sometimes called "z-order" for this reason.) We can see that points with coordinates that are close together in 2D are generally close together along the Morton curve.[†]

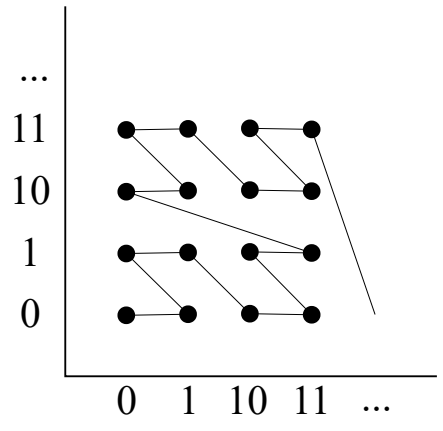
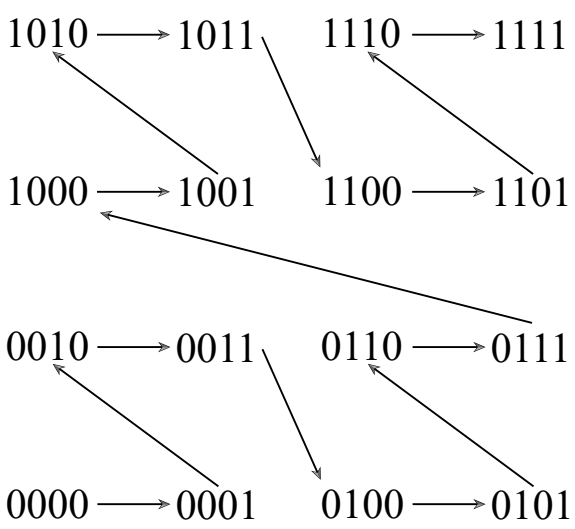



Figure 4.7: The Order That Points Are Visited along the Morton Curve. Coordinate values along the $x$ and $y$ axes are shown in binary. If we connect the integer coordinate points in the order of their Morton indices, we see that the Morton curve visits the points along a hierarchical "z"-shaped path.

A Morton-encoded value also encodes useful information about the position of the point that it represents. Consider the case of 4-bit coordinate values in 2D: the $x$ and $y$ coordinates are integers in $[0, 15]$ and the Morton code has 8 bits: $y_3\ x_3\ y_2\ x_2\ y_1\ x_1\ y_0\ x_0$. Many interesting properties follow from the encoding; a few examples include:

- For a Morton-encoded 8-bit value where the high bit $y_3$ is set, then we know that the high bit of its underlying $y$ coordinate is set and thus $y \geq 8$ (Figure 4.8(a)).
- The next bit value, $x_3$, splits the $x$ axis in the middle (Figure 4.8(b)). If $y_3$ is set and $x_3$ is off, for example, then the corresponding point must lie in the shaded area of Figure 4.8(c). In general, points with a number of matching high bits lie in a power-of-two sized and aligned region of space determined by the matching bit values.
- The value of $y_2$ splits the $y$ axis into four regions (Figure 4.8(d)).



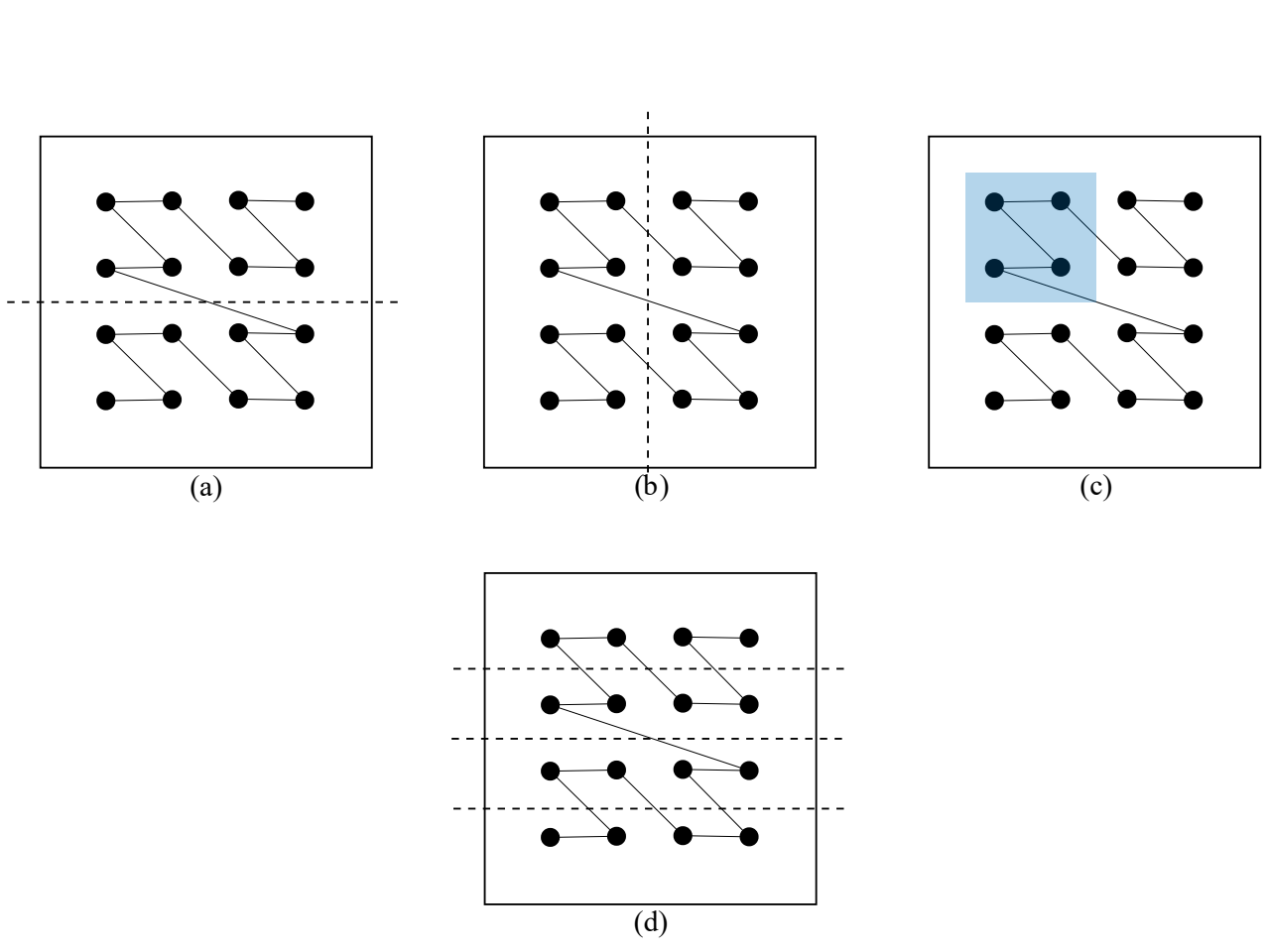



Figure 4.8: Implications of the Morton Encoding. The values of various bits in the Morton value indicate the region of space that the corresponding coordinate lies in. (a) In 2D, the high bit of the Morton-coded value of a point's coordinates define a splitting plane along the middle of the $y$ axis. If the high bit is set, the point is above the plane. (b) Similarly, the second-highest bit of the Morton value splits the $x$ axis in the middle. (c) If the high $y$ bit is 1 and the high $x$ bit is 0, then the point must lie in the shaded region. (d) The second-from-highest $y$ bit splits the $y$ axis into four regions.

Another way to interpret these bit-based properties is in terms of Morton-coded values. For example, Figure 4.8(a) corresponds to the index being in the range $[8, 15]$, and Figure 4.8(c) corresponds to $[8, 11]$. Thus, given a set of sorted Morton indices, we could find the range of points corresponding to an area like Figure 4.8(c) by performing a binary search to find each endpoint in the array.

LBVHs are BVHs built by partitioning primitives using splitting planes that are at the midpoint of each region of space (i.e., equivalent to the `SplitMethod::Middle` path defined earlier). Partitioning is extremely efficient, as it's based on properties of the Morton encoding described above.

Just reimplementing `Middle` in a different manner isn't particularly interesting, so in the implementation here, we'll build a *hierarchical linear bounding volume hierarchy* (HLBVH). With this approach, Morton-curve-based clustering is used to first build trees for the lower levels of the hierarchy (referred to as "treelets" in the following) and the top levels of the tree are then created using the surface area heuristic. The `HLBVHBuild()` method implements this approach and returns the root node of the resulting tree.

*<<BVHAccel Method Definitions>>+=*
```
BVHBuildNode *BVHAccel::HLBVHBuild(MemoryArena &arena,
        const std::vector<BVHPrimitiveInfo> &primitiveInfo,
        int *totalNodes,
        std::vector<std::shared_ptr<Primitive>> &orderedPrims) const {
    <<Compute bounding box of all primitive centroids>> ⊞
    <<Compute Morton indices of primitives>> ⊞
    <<Radix sort primitive Morton indices>> ⊞
    <<Create LBVH treelets at bottom of BVH>> ⊞
    <<Create and return SAH BVH from LBVH treelets>> ⊞
}
```

The BVH is built using only the centroids of primitive bounding boxes to sort them—it doesn't account for the actual spatial extent of each primitive. This simplification is critical to the performance that HLBVHs offer, but it also means that for scenes with primitives that span a wide range of sizes, the tree that is built won't account for this variation as an SAH-based tree would.

Because the Morton encoding operates on integer coordinates, we first need to bound the centroids of all of the primitives so that we can quantize centroid positions with respect to the overall bounds.

*<<Compute bounding box of all primitive centroids>>=*
```
Bounds3f bounds;
for (const BVHPrimitiveInfo &pi : primitiveInfo)
    bounds = Union(bounds, pi.centroid);
```

Given the overall bounds, we can now compute the Morton code for each primitive. This is a fairly lightweight calculation, but given that there may be millions of primitives, it's worth parallelizing. Note that a loop chunk size of 512 is passed to `ParallelFor()` below; this causes worker threads to be given groups of 512 primitives to process rather than one at a time as would otherwise be the default. Because the amount of work performed per primitive to compute the Morton code is relatively small, this granularity better amortizes the overhead of distributing tasks to the worker threads.

*<<Compute Morton indices of primitives>>=*
```
std::vector<MortonPrimitive> mortonPrims(primitiveInfo.size());
ParallelFor(
    [&](int i) {
        <<Initialize mortonPrims[i] for ith primitive>> ⊞
    }, primitiveInfo.size(), 512);
```

A `MortonPrimitive` instance is created for each primitive; it stores the index of the primitive in the `primitiveInfo` array as well as its Morton code.

*<<BVHAccel Local Declarations>>+=*

```
struct MortonPrimitive {
    int primitiveIndex;
    uint32_t mortonCode;
};
```

We use 10 bits for each of the $x$, $y$, and $z$ dimensions, giving a total of 30 bits for the Morton code. This granularity allows the values to fit into a single 32-bit variable. Floating-point centroid offsets inside the bounding box are in $[0, 1]$, so we scale them by $2^{10}$ to get integer coordinates that fit in 10 bits. (For the edge case of offsets exactly equal to 1, an out-of-range quantized value of 1024 may result; this case is handled in the forthcoming LeftShift3() function.)

*<<Initialize mortonPrims[i] for ith primitive>>=*

```
constexpr int mortonBits = 10;
constexpr int mortonScale = 1 << mortonBits;
mortonPrims[i].primitiveIndex = primitiveInfo[i].primitiveNumber;
Vector3f centroidOffset = bounds.Offset(primitiveInfo[i].centroid);
mortonPrims[i].mortonCode = EncodeMorton3(centroidOffset * mortonScale);
```

To compute 3D Morton codes, first we'll define a helper function: LeftShift3() takes a 32-bit value and returns the result of shifting the $i$th bit to be at the $3i$th bit, leaving zeros in other bits. Figure 4.9 illustrates this operation.

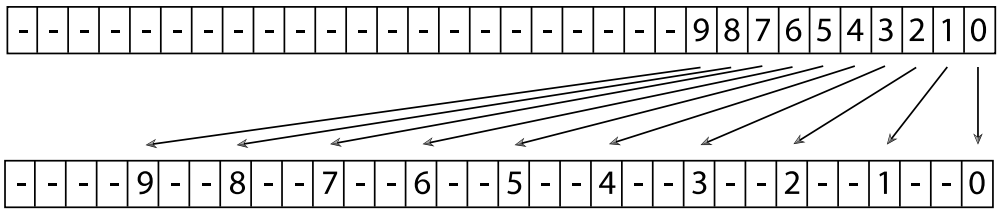



Figure 4.9: Bit Shifts to Compute 3D Morton Codes. The LeftShift3() function takes a 32-bit integer value and for the bottom 10 bits, shifts the $i$th bit to be in position $3i$—in other words, shifts it $2i$ places to the left. All other bits are set to zero.

The most obvious approach to implement this operation, shifting each bit value individually, isn't the most efficient. (It would require a total of 9 shifts, along with logical ORs to compute the final value.) Instead, we can decompose each bit's shift into multiple shifts of power-of-two size that together shift the bit's value to its final position. Then, all of the bits that need to be shifted a given power-of-two number of places can be shifted together. The LeftShift3() function implements this computation, and Figure 4.10 shows how it works.

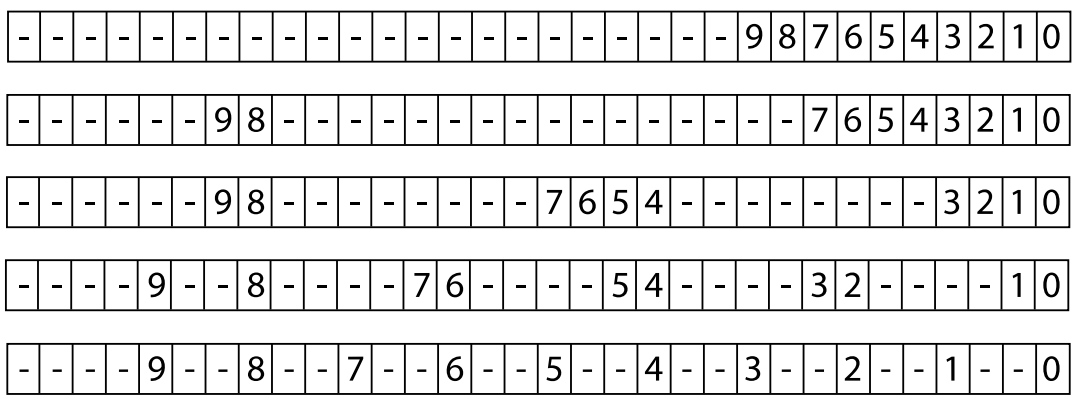



Figure 4.10: Power-of-Two Decomposition of Morton Bit Shifts. The bit shifts to compute the Morton code for each 3D coordinate are performed in a series of shifts of power-of-two size. First, bits 8 and 9 are shifted 16 places to the left. This places bit 8 in its final position. Next bits 4 through 7 are shifted 8 places. After shifts of 4 and 2 places (with appropriate masking so that each bit is shifted the right number of places in the end), all bits are in the proper position. This computation is implemented by the LeftShift3() function.

*<<BVHAccel Utility Functions>>=*

```
inline uint32_t LeftShift3(uint32_t x) {
    if (x == (1 << 10)) --x;
    x = (x | (x << 16)) & 0b00000011000000000000000011111111;
    x = (x | (x <<  8)) & 0b00000011000000000111100000001111;
```

```
    x = (x | (x <<  4)) & 0b0000001100001100001100001100001100000011;
    x = (x | (x <<  2)) & 0b0000100100100100100100100100100100001001;
    return x;
}
```

`EncodeMorton3()` takes a 3D coordinate value where each component is a floating-point value between 0 and $2^{10}$. It converts these values to integers and then computes the Morton code by expanding the three 10-bit quantized values so that their $i$th bits are at position $3i$, then shifting the $y$ bits over one more, the $z$ bits over two more, and ORing together the result (Figure 4.11).

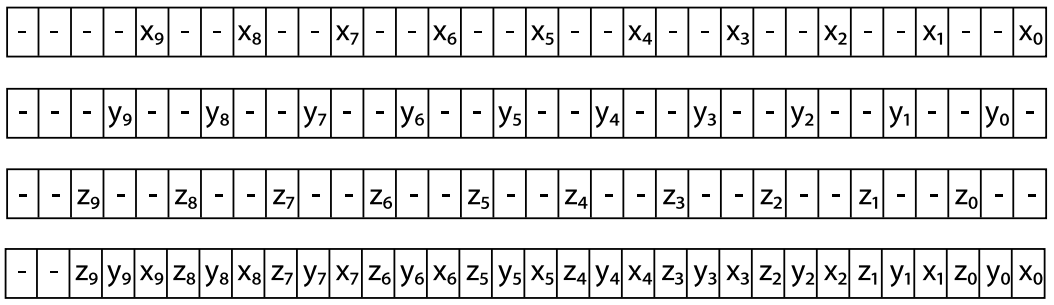



Figure 4.11: Final Interleaving of Coordinate Values. Given interleaved values for $x$, $y$, and $z$ computed by `LeftShift3()`, the final Morton-encoded value is computed by shifting $y$ and $z$ one and two places, respectively, and then ORing together the results.

*<<BVHAccel Utility Functions>>+=* ▲ ▼

```
inline uint32_t EncodeMorton3(const Vector3f &v) {
    return (LeftShift3(v.z) << 2) | (LeftShift3(v.y) << 1) |
            LeftShift3(v.x);
}
```

Once the Morton indices have been computed, we'll sort the `mortonPrims` by Morton index value using a radix sort. We have found that for BVH construction, our radix sort implementation is noticeably faster than using `std::sort()` from our system's standard library (which is a mixture of a quicksort and an insertion sort).

*<<Radix sort primitive Morton indices>>=*

```
RadixSort(&mortonPrims);
```

Recall that a radix sort differs from most sorting algorithms in that it isn't based on comparing pairs of values but rather is based on bucketing items based on some key. Radix sort can be used to sort integer values by sorting them one digit at a time, going from the rightmost digit to the leftmost. Especially with binary values, it's worth sorting multiple digits at a time; doing so reduces the total number of passes taken over the data. In the implementation here, `bitsPerPass` sets the number of bits processed per pass; with the value 6, we have 5 passes to sort the 30 bits.

*<<BVHAccel Utility Functions>>+=* ▲

```
static void RadixSort(std::vector<MortonPrimitive> *v) {
    std::vector<MortonPrimitive> tempVector(v->size());
    constexpr int bitsPerPass = 6;
    constexpr int nBits = 30;
    constexpr int nPasses = nBits / bitsPerPass;
    for (int pass = 0; pass < nPasses; ++pass) {
        <<Perform one pass of radix sort, sorting bitsPerPass bits>> ⊞
    }
    <<Copy final result from tempVector, if needed>> ⊞
}
```

The current pass will sort `bitsPerPass` bits, starting at `lowBit`.

*<<Perform one pass of radix sort, sorting `bitsPerPass` bits>>=*

```
int lowBit = pass * bitsPerPass;
<<Set in and out vector pointers for radix sort pass>> ⊞
<<Count number of zero bits in array for current radix sort bit>> ⊞
<<Compute starting index in output array for each bucket>> ⊞
<<Store sorted values in output array>> ⊞
```

The `in` and `out` references correspond to the vector to be sorted and the vector to store the sorted values in, respectively. Each pass through the loop alternates between the input vector `*v` and the temporary vector for each of them.

*<<Set in and out vector pointers for radix sort pass>>=*
```
std::vector<MortonPrimitive> &in  = (pass & 1) ? tempVector : *v;
std::vector<MortonPrimitive> &out = (pass & 1) ? *v : tempVector;
```

If we're sorting $n$ bits per pass, then there are $2^n$ buckets that each value may land in. We first count how many values will land in each bucket; this will let us determine where to store sorted values in the output array. To compute the bucket index for the current value, the implementation shifts the index so that the bit at index `lowBit` is at bit 0 and then masks off the low `bitsPerPass` bits.

*<<Count number of zero bits in array for current radix sort bit>>=*
```
constexpr int nBuckets = 1 << bitsPerPass;
int bucketCount[nBuckets] = { 0 };
constexpr int bitMask = (1 << bitsPerPass) - 1;
for (const MortonPrimitive &mp : in) {
    int bucket = (mp.mortonCode >> lowBit) & bitMask;
    ++bucketCount[bucket];
}
```

Given the count of how many values land in each bucket, we can compute the offset in the output array where each bucket's values start; this is just the sum of how many values land in the preceding buckets.

*<<Compute starting index in output array for each bucket>>=*
```
int outIndex[nBuckets];
outIndex[0] = 0;
for (int i = 1; i < nBuckets; ++i)
    outIndex[i] = outIndex[i - 1] + bucketCount[i - 1];
```

Now that we know where to start storing values for each bucket, we can take another pass over the primitives to recompute the bucket that each one lands in and to store their `MortonPrimitives` in the output array. This completes the sorting pass for the current group of bits.

*<<Store sorted values in output array>>=*
```
for (const MortonPrimitive &mp : in) {
    int bucket = (mp.mortonCode >> lowBit) & bitMask;
    out[outIndex[bucket]++] = mp;
}
```

When sorting is done, if an odd number of radix sort passes were performed, then the final sorted values need to be copied from the temporary vector to the output vector that was originally passed to `RadixSort()`.

*<<Copy final result from `tempVector`, if needed>>=*
```
if (nPasses & 1)
    std::swap(*v, tempVector);
```

Given the sorted array of primitives, we'll find clusters of primitives with nearby centroids and then create an LBVH over the primitives in each cluster. This step is a good one to parallelize as there are generally many clusters and each cluster can be processed independently.

*<<Create LBVH treelets at bottom of BVH>>=*
*<<Find intervals of primitives for each treelet>>* ⊞
*<<Create LBVHs for treelets in parallel>>* ⊞

Each primitive cluster is represented by an `LBVHTreelet`. It encodes the index in the `mortonPrims` array of the first primitive in the cluster as well as the number of following primitives. (See Figure 4.12.)

*<<BVHAccel Local Declarations>>+=* ▲ ▼
```
struct LBVHTreelet {
   int startIndex, nPrimitives;
   BVHBuildNode *buildNodes;
};
```

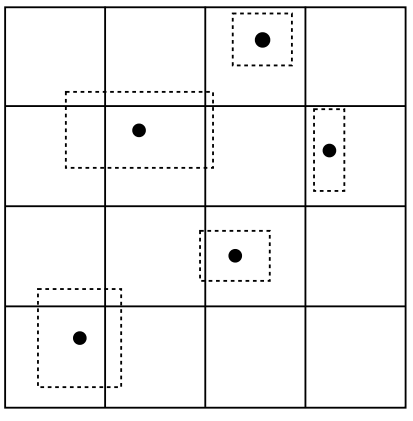


Figure 4.12: Primitive Clusters for LBVH Treelets. Primitive centroids are clustered in a uniform grid over their bounds. An LBVH is created for each cluster of primitives within a cell that are in contiguous sections of the sorted Morton index values.

Recall from Figure 4.8 that a set of points with Morton codes that match in their high bit values lie in a power-of-two aligned and sized subset of the original volume. Because we have already sorted the `mortonPrims` array by Morton-coded value, primitives with matching high bit values are already together in contiguous sections of the array.

Here we'll find sets of primitives that have the same values for the high 12 bits of their 30-bit Morton codes. Clusters are found by taking a linear pass through the `mortonPrims` array and finding the offsets where any of the high 12 bits changes. This corresponds to clustering primitives in a regular grid of $2^{12} = 4096$ total grid cells with $2^4 = 16$ cells in each dimension. In practice, many of the grid cells will be empty, though we'll still expect to find many independent clusters here.

*<<Find intervals of primitives for each treelet>>=*

```
std::vector<LBVHTreelet> treeletsToBuild;
for (int start = 0, end = 1; end <= (int)mortonPrims.size(); ++end) {
    uint32_t mask = 0b00111111111111000000000000000000;
    if (end == (int)mortonPrims.size() ||
        ((mortonPrims[start].mortonCode & mask) !=
         (mortonPrims[end].mortonCode & mask))) {
        <<Add entry to treeletsToBuild for this treelet>> ⊞
        start = end;
    }
}
```

When a cluster of primitives has been found for a treelet, `BVHBuildNodes` are immediately allocated for it. (Recall that the number of nodes in a BVH is bounded by twice the number of leaf nodes, which in turn is bounded by the number of primitives). It's simpler to pre-allocate this memory now in a serial phase of execution than during parallel construction of LBVHs.

One important detail here is the `false` value passed to `MemoryArena::Alloc()`; it indicates that the constructors of the underlying objects being allocated should not be executed. To our surprise, running the `BVHBuildNode` constructors introduced significant overhead and meaningfully reduced overall HLBVH construction performance. Because all of the members of `BVHBuildNode` will be initialized in code to follow, the initialization performed by the constructor is unnecessary here in any case.

*<<Add entry to `treeletsToBuild` for this treelet>>=*

```
int nPrimitives = end - start;
int maxBVHNodes = 2 * nPrimitives - 1;
BVHBuildNode *nodes = arena.Alloc<BVHBuildNode>(maxBVHNodes, false);
treeletsToBuild.push_back({start, nPrimitives, nodes});
```

Once the primitives for each treelet have been identified, we can create LBVHs for them in parallel. When construction is finished, the `buildNodes` pointer for each `LBVHTreelet` will point to the root of the corresponding LBVH.

There are two places where the worker threads building LBVHs must coordinate with each other. First, the total number of nodes in all of the LBVHs needs to be computed and returned via the `totalNodes` pointer passed to `HLBVHBuild()`. Second, when leaf nodes are created for the LBVHs, a contiguous segment of the `orderedPrims` array is needed to record the indices of the primitives in the leaf node. Our implementation uses atomic variables for both—`atomicTotal` to track the number of nodes and `orderedPrimsOffset` for the index of the next available entry in `orderedPrims`.

*<<Create LBVHs for treelets in parallel>>=*

```
std::atomic<int> atomicTotal(0), orderedPrimsOffset(0);
orderedPrims.resize(primitives.size());
ParallelFor(
    [&](int i) {
        <<Generate ith LBVH treelet>> ⊞
    }, treeletsToBuild.size());
*totalNodes = atomicTotal;
```

The work of building the treelet is performed by `emitLBVH()`, which takes primitives with centroids in some region of space and successively partitions them with splitting planes that divide the current region of space into two halves along the center of the region along one of the three axes.

Note that instead of taking a pointer to the atomic variable `atomicTotal` to count the number of nodes created, `emitLBVH()` updates a non-atomic local variable. The fragment here then only updates `atomicTotal` once per treelet when each treelet is done. This approach gives measurably better performance than the alternative—having the worker threads frequently modify `atomicTotal` over the course of their execution. (See the discussion of the overhead of multi-core memory coherence models in Appendix A.6.1.)

*<<Generate ith LBVH treelet>>=*

```
int nodesCreated = 0;
const int firstBitIndex = 29 - 12;
LBVHTreelet &tr = treeletsToBuild[i];
tr.buildNodes =
    emitLBVH(tr.buildNodes, primitiveInfo, &mortonPrims[tr.startIndex],
             tr.nPrimitives, &nodesCreated, orderedPrims,
             &orderedPrimsOffset, firstBitIndex);
atomicTotal += nodesCreated;
```

Thanks to the Morton encoding, the current region of space doesn't need to be explicitly represented in `emitLBVH()`: the sorted `MortonPrims` passed in have some number of matching high bits, which in turn corresponds to a spatial bound. For each of the remaining bits in the Morton codes, this function tries to split the primitives along the plane corresponding to the bit `bitIndex` (recall Figure 4.8(d)) and then calls itself recursively. The index of the next bit to try splitting with is passed as the last argument to the function: initially it's $29 - 12$, since $29$ is the index of the $30$th bit with zero-based indexing, and we previously used the high $12$ bits of the Morton-coded value to cluster the primitives; thus, we know that those bits must all match for the cluster.

*<<BVHAccel Method Definitions>>+=* ▲ ▼

```
BVHBuildNode *BVHAccel::emitLBVH(BVHBuildNode *&buildNodes,
        const std::vector<BVHPrimitiveInfo> &primitiveInfo,
        MortonPrimitive *mortonPrims, int nPrimitives, int *totalNodes,
        std::vector<std::shared_ptr<Primitive>> &orderedPrims,
        std::atomic<int> *orderedPrimsOffset, int bitIndex) const {
    if (bitIndex == -1 || nPrimitives < maxPrimsInNode) {
        <<Create and return leaf node of LBVH treelet>> ⊞
    } else {
        int mask = 1 << bitIndex;
        <<Advance to next subtree level if there's no LBVH split for this bit>> ⊞
        <<Find LBVH split point for this dimension>> ⊞
        <<Create and return interior LBVH node>> ⊞
    }
}
```

After `emitLBVH()` has partitioned the primitives with the final low bit, no more splitting is possible and a leaf node is created. Alternatively, it also stops and makes a leaf node if it's down to a small number of primitives.

Recall that `orderedPrimsOffset` is the offset to the next available element in the `orderedPrims` array. Here, the call to `fetch_add()` atomically adds the value of `nPrimitives` to `orderedPrimsOffset` and returns its old value before the addition. Because these operations are atomic, multiple LBVH construction threads can concurrently carve out space in the `orderedPrims` array without data races. Given space in the array, leaf construction is similar to the approach implemented earlier in <<*Create leaf BVHBuildNode*>>.

*<<Create and return leaf node of LBVH treelet>>=*

```
(*totalNodes)++;
BVHBuildNode *node = buildNodes++;
Bounds3f bounds;
int firstPrimOffset = orderedPrimsOffset->fetch_add(nPrimitives);
for (int i = 0; i < nPrimitives; ++i) {
    int primitiveIndex = mortonPrims[i].primitiveIndex;
```

```
    orderedPrims[firstPrimOffset + i] = primitives[primitiveIndex];
    bounds = Union(bounds, primitiveInfo[primitiveIndex].bounds);
}
node->InitLeaf(firstPrimOffset, nPrimitives, bounds);
return node;
```

It may be the case that all of the primitives lie on the same side of the splitting plane; since the primitives are sorted by their Morton index, this case can be efficiently checked by seeing if the first and last primitive in the range both have the same bit value for this plane. In this case, `emitLBVH()` proceeds to the next bit without unnecessarily creating a node.

*<<Advance to next subtree level if there's no LBVH split for this bit>>=*

```
if ((mortonPrims[0].mortonCode & mask) ==
    (mortonPrims[nPrimitives - 1].mortonCode & mask))
    return emitLBVH(buildNodes, primitiveInfo, mortonPrims, nPrimitives,
                    totalNodes, orderedPrims, orderedPrimsOffset,
                    bitIndex - 1);
```

If there are primitives on both sides of the splitting plane, then a binary search efficiently finds the dividing point where the `bitIndex`th bit goes from 0 to 1 in the current set of primitives.

*<<Find LBVH split point for this dimension>>=*

```
int searchStart = 0, searchEnd = nPrimitives - 1;
while (searchStart + 1 != searchEnd) {
    int mid = (searchStart + searchEnd) / 2;
    if ((mortonPrims[searchStart].mortonCode & mask) ==
        (mortonPrims[mid].mortonCode & mask))
        searchStart = mid;
    else
        searchEnd = mid;
}
int splitOffset = searchEnd;
```

Given the split offset, the method can now claim a node to use as an interior node and recursively build LBVHs for both partitioned sets of primitives. Note a further efficiency benefit from Morton encoding: entries in the `mortonPrims` array don't need to be copied or reordered for the partition: because they are all sorted by their Morton code value and because it is processing bits from high to low, the two spans of primitives are already on the correct sides of the partition plane.

*<<Create and return interior LBVH node>>=*

```
(*totalNodes)++;
BVHBuildNode *node = buildNodes++;
BVHBuildNode *lbvh[2] = {
    emitLBVH(buildNodes, primitiveInfo, mortonPrims, splitOffset,
             totalNodes, orderedPrims, orderedPrimsOffset, bitIndex - 1),
    emitLBVH(buildNodes, primitiveInfo, &mortonPrims[splitOffset],
             nPrimitives - splitOffset, totalNodes, orderedPrims,
             orderedPrimsOffset, bitIndex - 1)
};
int axis = bitIndex % 3;
node->InitInterior(axis, lbvh[0], lbvh[1]);
return node;
```

Once all of the LBVH treelets have been created, `buildUpperSAH()` creates a BVH of all the treelets. Since there are generally tens or hundreds of them (and in any case, no more than 4096), this step takes very little time.

*<<Create and return SAH BVH from LBVH treelets>>=*

```
std::vector<BVHBuildNode *> finishedTreelets;
for (LBVHTreelet &treelet : treeletsToBuild)
    finishedTreelets.push_back(treelet.buildNodes);
return buildUpperSAH(arena, finishedTreelets, 0,
                     finishedTreelets.size(), totalNodes);
```

The implementation of this method isn't included here, as it follows the same approach as fully SAH-based BVH construction, just over treelet root nodes rather than scene primitives.

*<<BVHAccel Private Methods>>=*

```
BVHBuildNode *buildUpperSAH(MemoryArena &arena,
    std::vector<BVHBuildNode *> &treeletRoots, int start, int end,
    int *totalNodes) const;
```

# 4.3.4 Compact BVH For Traversal

Once the BVH tree is built, the last step is to convert it into a compact representation—doing so improves cache, memory, and thus overall system performance. The final BVH is stored in a linear array in memory. The nodes of the original tree are laid out in depth-first order, which means that the first child of each interior node is immediately after the node in memory. In this case, only the offset to the second child of each interior node must be stored explicitly. See Figure 4.13 for an illustration of the relationship between tree topology and node order in memory.

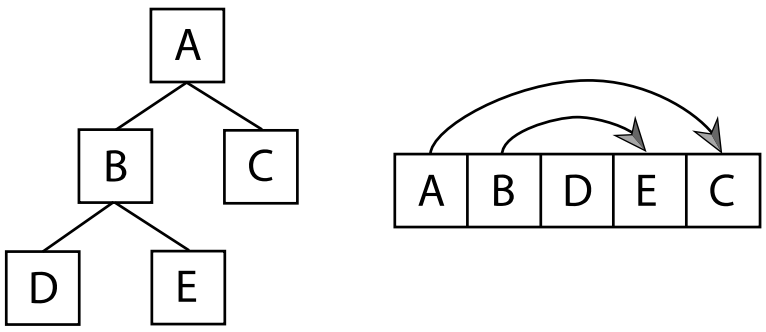



Figure 4.13: Linear Layout of a BVH in Memory. The nodes of the BVH (left) are stored in memory in depth-first order (right). Therefore, for any interior node of the tree (A and B in this example), the first child is found immediately after the parent node in memory. The second child is found via an offset pointer, represented here with lines with arrows. Leaf nodes of the tree (D, E, and C) have no children.

The `LinearBVHNode` structure stores the information needed to traverse the BVH. In addition to the bounding box for each node, for leaf nodes it stores the offset and primitive count for the primitives in the node. For interior nodes, it stores the offset to the second child as well as which of the coordinate axes the primitives were partitioned along when the hierarchy was built; this information is used in the traversal routine below to try to visit nodes in front-to-back order along the ray.

*<<BVHAccel Local Declarations>>+=*
```
struct LinearBVHNode {
    Bounds3f bounds;
    union {
        int primitivesOffset;    // leaf
        int secondChildOffset;   // interior
    };
    uint16_t nPrimitives;  // 0 -> interior node
    uint8_t axis;          // interior node: xyz
    uint8_t pad[1];        // ensure 32 byte total size
};
```

This structure is padded to ensure that it's 32 bytes large. Doing so ensures that, if the nodes are allocated such that the first node is cache-line aligned, then none of the subsequent nodes will straddle cache lines (as long as the cache line size is at least 32 bytes, which is the case on modern CPU architectures).

The built tree is transformed to the `LinearBVHNode` representation by the `flattenBVHTree()` method, which performs a depth-first traversal and stores the nodes in memory in linear order.

*<<Compute representation of depth-first traversal of BVH tree>>=*
```
nodes = AllocAligned<LinearBVHNode>(totalNodes);
int offset = 0;
flattenBVHTree(root, &offset);
```

The pointer to the array of `LinearBVHNodes` is stored as a `BVHAccel` member variable so that it can be freed in the `BVHAccel` destructor.

*<<BVHAccel Private Data>>+=*
```
LinearBVHNode *nodes = nullptr;
```

Flattening the tree to the linear representation is straightforward; the `*offset` parameter tracks the current offset into the `BVHAccel::nodes` array. Note that the current node is added to the array before the recursive calls to process its children (if the node is an interior node).

*<<BVHAccel Method Definitions>>+=*

```
int BVHAccel::flattenBVHTree(BVHBuildNode *node, int *offset) {
    LinearBVHNode *linearNode = &nodes[*offset];
    linearNode->bounds = node->bounds;
    int myOffset = (*offset)++;
    if (node->nPrimitives > 0) {
        linearNode->primitivesOffset = node->firstPrimOffset;
        linearNode->nPrimitives = node->nPrimitives;
    } else {
        <<Create interior flattened BVH node>> ⊞
    }
    return myOffset;
}
```

At interior nodes, recursive calls are made to flatten the two subtrees. The first one ends up immediately after the current node in the array, as desired, and the offset of the second one, returned by its recursive `flattenBVHTree()` call, is stored in this node's `secondChildOffset` member.

*<<Create interior flattened BVH node>>=*

```
linearNode->axis = node->splitAxis;
linearNode->nPrimitives = 0;
flattenBVHTree(node->children[0], offset);
linearNode->secondChildOffset =
    flattenBVHTree(node->children[1], offset);
```

## 4.3.5 Traversal

The BVH traversal code is quite simple—there are no recursive function calls and only a tiny amount of data to maintain about the current state of the traversal. The `Intersect()` method starts by precomputing a few values related to the ray that will be used repeatedly.

*<<BVHAccel Method Definitions>>+=* ▲

```
bool BVHAccel::Intersect(const Ray &ray,
        SurfaceInteraction *isect) const {
    bool hit = false;
    Vector3f invDir(1 / ray.d.x, 1 / ray.d.y, 1 / ray.d.z);
    int dirIsNeg[3] = { invDir.x < 0, invDir.y < 0, invDir.z < 0 };
    <<Follow ray through BVH nodes to find primitive intersections>> ⊞
    return hit;
}
```

Each time the `while` loop in `Intersect()` starts an iteration, `currentNodeIndex` holds the offset into the `nodes` array of the node to be visited. It starts with a value of 0, representing the root of the tree. The nodes that still need to be visited are stored in the `nodesToVisit[]` array, which acts as a stack; `toVisitOffset` holds the offset to the next free element in the stack.

*<<Follow ray through BVH nodes to find primitive intersections>>=*

```
int toVisitOffset = 0, currentNodeIndex = 0;
int nodesToVisit[64];
while (true) {
    const LinearBVHNode *node = &nodes[currentNodeIndex];
    <<Check ray against BVH node>> ⊞
}
```

At each node, we check to see if the ray intersects the node's bounding box (or starts inside of it). We visit the node if so, testing for intersection with its primitives if it's a leaf node or processing its children if it's an interior node. If no intersection is found, then the offset of the next node to be visited is retrieved from `nodesToVisit[]` (or traversal is complete if the stack is empty).

*<<Check ray against BVH node>>=*

```
if (node->bounds.IntersectP(ray, invDir, dirIsNeg)) {
    if (node->nPrimitives > 0) {
        <<Intersect ray with primitives in leaf BVH node>> ⊞
    } else {
        <<Put far BVH node on nodesToVisit stack, advance to near node>> ⊞
    }
} else {
    if (toVisitOffset == 0) break;
    currentNodeIndex = nodesToVisit[--toVisitOffset];
}
```

If the current node is a leaf, then the ray must be tested for intersection with the primitives inside it. The next node to visit is then found from the `nodesToVisit` stack; even if an intersection is found in the current node, the remaining nodes must be visited, in case one of them yields a closer intersection. However, if an intersection is found, the ray's `tMax` value will be updated to the intersection distance; this makes it possible to efficiently discard any remaining nodes that are farther away than the intersection.

*<<Intersect ray with primitives in leaf BVH node>>=*

```
for (int i = 0; i < node->nPrimitives; ++i)
    if (primitives[node->primitivesOffset + i]->Intersect(ray, isect))
        hit = true;
if (toVisitOffset == 0) break;
currentNodeIndex = nodesToVisit[--toVisitOffset];
```

For an interior node that the ray hits, it is necessary to visit both of its children. As described above, it's desirable to visit the first child that the ray passes through before visiting the second one, in case there is a primitive that the ray intersects in the first one, so that the ray's `tMax` value can be updated, thus reducing the ray's extent and thus the number of node bounding boxes it intersects.

An efficient way to perform a front-to-back traversal without incurring the expense of intersecting the ray with both child nodes and comparing the distances is to use the sign of the ray's direction vector for the coordinate axis along which primitives were partitioned for the current node: if the sign is negative, we should visit the second child before the first child, since the primitives that went into the second child's subtree were on the upper side of the partition point. (And conversely for a positive-signed direction.) Doing this is straightforward: the offset for the node to be visited first is copied to `currentNodeIndex`, and the offset for the other node is added to the `nodesToVisit` stack. (Recall that the first child is immediately after the current node due to the depth-first layout of nodes in memory.)

*<<Put far BVH node on `nodesToVisit` stack, advance to near node>>=*

```
if (dirIsNeg[node->axis]) {
   nodesToVisit[toVisitOffset++] = currentNodeIndex + 1;
   currentNodeIndex = node->secondChildOffset;
} else {
   nodesToVisit[toVisitOffset++] = node->secondChildOffset;
   currentNodeIndex = currentNodeIndex + 1;
}
```

The `BVHAccel::IntersectP()` method is essentially the same as the regular intersection method, with the two differences that `Primitive`'s `IntersectP()` methods are called rather than `Intersect()`, and traversal stops immediately when any intersection is found.