Lightweight Bounding Volumes for Ray Tracing

David Cline, Kevin Steele and Parris Egbert *
Brigham Young University

February 13, 2006

Abstract

This paper presents a memory-efficient auxiliary data structure for ray tracing called a *lightweight bounding volume hierarchy*, or LBVH. The new data structure reduces memory requirements in three ways: using implicit indexing, limited precision numbers, and a high branching factor. We show that LBVHs can be nearly as effective as standard bounding volumes in terms of speed while using significantly less memory.

1 Introduction

In order for ray tracing to be efficient, an auxiliary data structure such as a bounding volume hierarchy (BVH) or voxel grid must be used to speed intersection tests. This auxiliary data structure can use a significant amount of memory. To illustrate, consider the following reasonably efficient implementation of a BVH node similar to the one described in [Shirley and Morley, 2003]:

```
struct BVHnode {
    float minBounds[3];
    float maxBounds[3];
    BVHnode *left, *right;
};
```

Each BVH node contains six floating point numbers and two pointers, yeilding 32 bytes. [Smits, 1998] shows how to eliminate one of the pointers in the BVH node structure by storing the nodes in an array in traversal order and using "skip" pointers instead of child pointers. Our technique takes this idea a step further, eliminating all pointers from the acceleration data structure.

Memory requirements can also be decreased by using reduced precision numbers. [Terdiman, 2001] describes the use of "quantized trees" to lower memory costs in a collision detection context. [Mahovsky, 2005] presents reduced precision hierarchies in a ray tracing context, using unsigned bytes instead of floating point numbers in

*e-mail: cline@rivit.cs.byu.edu, steele@cs.byu.edu, egbert@cs.byu.edu

the BVH nodes. To maintain precision, Mahovsky employs a hierarchical encoding scheme, which increases the BVH traversal time, however. Our algorithm also uses reduced precision numbers to lower memory requirements, but we use a simpler, fixed encoding scheme, and maintain precision by nesting multiple bounding volume hierarchies.

A third way to decrease the amount of memory taken by a bounding volume hierarchy is to increase the branching factor (number of children per node). If we consider a hierarchy's "bounding ability" to be proportional to the number of leaf nodes, increasing the branching factor decreases the memory cost for the same bounding ability. We refer to the amount of memory that a hierarchy uses per leaf node as the *amortized cost* of leaf nodes. For a constant branching factor of k, the amortized cost of leaf nodes can be calculated as $k/(k-1)$ times the cost of a single BVH node. Hence, if the BVH nodes defined at the beginning of this section are arranged into a binary tree, the amortized cost of the leaf nodes is twice that of a single node, or 64 bytes. Bounding volume methods with a higher branching factor, such as the Salmon-Goldsmith algorithm [Goldsmith and Salmon, 1987] have a lower amortized cost per leaf node.

Other acceleration data structures, such as the voxel grids [Fujimoto et al., 1986, Klimaszewski and Sederberg, 1997], and axis-aligned BSP trees [Wald et al., 2001] may fare a little better than standard bounding volumes in terms of memory footprint, but they too can require substantial amounts of memory. For voxel grids, each voxel typically contains a pointer (4 bytes), and non-empty voxels have a list of objects that intersect the voxel bounds. To achieve maximum speed, the number of voxels is often set to several times the number of objects enclosed in the grid. This causes a large percentage of the objects to span multiple voxels, which in turn increases the number of object pointers that must be stored. BSP nodes can be described compactly, using only one float and one pointer (8 bytes), so the amortized cost of leaf nodes, not including any object pointers in the leaves, is 16 bytes. However, as with voxel grids, objects may span multiple spatial partitions, so it is likely that several pointers will need to be stored for each object enclosed by the hierarchy.

2 Lightweight Bounding Volumes

In this section we show that in the case of bounding volume hierarchies, implicit indexing can be used to create auxiliary data structures that have no pointers at all. We then show that by using limited precision arithmetic, lightweight bounding volume hierarchies can be created that are easy to initialize, efficient to traverse, and require only a few bytes per enclosed object to store.

2.1 The LBVH Data Structure

A lightweight bounding volume hierarchy (LBVH) is an essentially complete k-ary tree that is stored in an array, and indexed like a heap. Node zero in the array is designated as the root node, and and for any node q, its parent is node $\lfloor (q-1)/k \rfloor$, and its children are nodes $(qk+1)$ to $(qk+k)$. The data members of the LBVH node use limited precision

numbers, typically two-byte shorts or unsigned bytes, to conserve space. One possible LBVH node is defined as follows:

```
struct LBVHnode {
    short   minBounds[3];
    short   maxBounds[3];
};
```

This structure only requires 12 bytes. Our implementation is based on a branching factor of four, so the amortized cost of leaf nodes is 4/3 the cost of a single node, or 16 bytes. We found four to be a good tradeoff between the size of the hierarchy and traversal speed. In our experiments, using a branching factor of four decreased speed by about 1.5% while saving a third of the memory cost of the hierarchy. By contrast, using a branching factor of 8 only saved an additional 9.5% of the memory cost, while being 14% slower.

As mentioned, child pointers are not needed since the hierarchy is indexed like a heap. Explicit object lists at the leaf nodes are also not needed. Instead, a simple function, which will be defined in section 2.3, determines which objects are enclosed by a given leaf.

In addition to the array of limited precision nodes, the bounding box of the hierarchy, B, is stored at full precision in world space. When the hierarchy is traversed, the bounding box will be used to transform rays from world space to the *range* of the limited precision numbers used by the LBVH (i.e. *range* = 255 for unsigned bytes and $2^{15}-1$ for shorts). The transformation is a translation followed by a scale:

$$translate = (-B_{xmin}, -B_{ymin}, -B_{zmin}) \tag{1}$$

$$scale = \left(\frac{range-1}{B_{xmax}-B_{xmin}}, \frac{range-1}{B_{ymax}-B_{ymin}}, \frac{range-1}{B_{zmax}-B_{zmin}} \right) \tag{2}$$

Note that we subtract 1 from *range* in equation 2 to avoid overflow, which can happen because of numerical imprecision.

To maintain tight bounds, we do not use a single LBVH for the entire scene. Instead, we enclose each polygonal mesh in its own LBVH, and then enclose these in a second level hierarchy that encompasses the whole scene. Without this step, large structures such as walls or ground planes could steal precision from finely tessellated parts of the scene.

2.2 LBVH Initialization

This section discusses how to initialize an LBVH for an unorganized list of objects (e.g. triangles). Our implementation uses a variant of median split adapted to handle a branching factor of four rather than two. Instead of splitting a node once to create two children as in the median split algorithm, we split twice to create four children. In addition, we do not split object lists at the exact median. Rather, we determine how many objects should go into each node in the tree to make sure that all leaf nodes

1. Calculate the bounding box of the n objects, B.
2. Calculate *translate* and *scale* using equations 1 and 2.
3. Calculate the number of nodes in the hierarchy, m.
   3a. $m = \lfloor 4n/3 \rfloor$.
   3b. Increment m by 1 until $m\%4$ equals 1.
4. Calculate how many objects will go into each node.
   4a. Temporarily assign one object to each leaf node.
   4b. Find number of objects in non-leaf nodes (sum number in children).
5. Recursively partition the objects, starting at the root node.
   5a. Store the bounding box of objects enclosed by the current node in the node.
   5b. If the current node is a leaf, return.
   5c. Sort the objects on the longest axis (in world space).
   5d. Divide the objects into two intermediate partitions.
      Let a, b, c and d be the number of objects that will be assigned to the children of the current node.
      Place $a+b$ objects in the first intermediate partition, and $c+d$ objects in the second intermediate partition.
   5e. Compute the bounding box of each intermediate partition.
   5f. Sort the objects in the intermediate partitions along their longest axes.
   5g. Place a, b, c and d objects in the four children of the current node.
   5h. Recursively partition each of the four child nodes.

Figure 1: Algorithm to initialize an LBVH for an unorganized list of n objects.

contain the same number of objects. Constructing the tree in this way guarantees that the BVH will be fully balanced, which is required for implicit indexing. Figure 1 gives pseudocode for initializing an LBVH. The remainder of this section describes the initialization steps in detail.

Determining the size of the LBVH. As an example of LBVH initialization, consider the problem of bounding 7 objects in an LBVH with one object in each leaf node (refer to 2). In step 1 of the algorithm, the bounding box of the seven objects is calculated. Step 2 calculates the translate and scale vectors associated with the bounding box (equations 1 and 2). In step 3, the number of nodes that will be in the hierarchy is determined using the amortized cost of leaf nodes. We use a branching factor of four, so the hierarchy will contain 4/3 as many nodes as leaves, or $\lfloor 7 \times (4/3) \rfloor = 9$ total nodes. (In step 3b, we increase this value slightly if necessary to make sure that all interior nodes in the hierarchy have four children.)

Calculating the number of objects enclosed by LBVH nodes. Step 4 of the algorithm determines the number of objects that will be enclosed by each node in the hierarchy. We start by temporarily assigning one object (or some small fixed number)

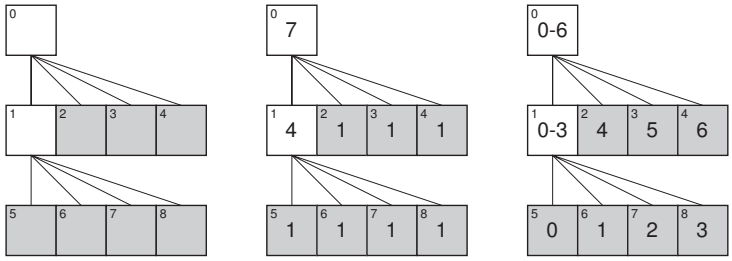

Figure 2: A lightweight bounding volume hierarchy. (A) The LBVH is stored as an array of nodes. In this case, 9 total nodes are in the array, with 7 leaf nodes, shown in gray (step 3 of figure 1). (B) Before an LBVH can be initialized, the number of objects enclosed by each node in the hierarchy must be calculated. First, a predetermined number of objects is assigned to each leaf node, in this case one object per leaf. Then, the number of objects enclosed by internal nodes is calculated by summing the number of objects contained by their children (step 4 of figure 1). Finally, the LBVH is initialized based on repeated sorting of the array of objects (step 5 of figure 1). (C) shows the indices of the objects that are ultimately enclosed by the LBVH nodes. Note that the first leaf node on the bottom row of the hierarchy always encloses object zero.

to each leaf node, and then calculate the number of objects in internal nodes by summing the number of objects in their children. Figure 2(a) shows the hierarchy for our example, and figure 2(b) shows the number of objects enclosed by LBVH nodes.

Initializing the bounding volume hierarchy. Once the number of objects assigned to the nodes in the LBVH has been established, the objects can be partitioned among the nodes (step 5). Consider again the example of enclosing seven objects with an LBVH. To intialize the hierarchy, we transform the bounding box of the objects using equations 1 and 2, and store the result in the root node of the LBVH. Next, the objects are sorted along the longest axis of the bounding box (in world space, not limited precision space). The sorted list of objects is then partitioned so that the left partition has five objects (the sum of the number of objects enclosed by the first two children of the root, 4+1), and the right partition has two objects (the sum of the number of objects enclosed by the second two children of the root, 1+1). These partitions are then divided again by computing their bounding boxes and sorting the objects based on the corresponding longest axes. This process is repeated recursively for all internal nodes in the hierarchy, resulting in the configuration shown in figure 2(c).

2.3 LBVH Traversal

A ray can be intersected with an LBVH in the same way as with any other bounding volume hierarchy, except that the ray must be transformed into the coordinate system of the LBVH, and the indices of enclosed objects must be calculated by the traversal

routine. During traversal, the object index corresponding to leaf j is found using the equation

$$i = \begin{cases} (j - l_b)\, n & \text{if } j \leq l_0 \\ (j - l_b + m - 1)\, n & \text{otherwise} \end{cases} \quad (3)$$

where i is the beginning index objects enclosed by leaf node j, l_b is the node index of the first node on the bottom row of the hierarchy (5 for the hierarchy shown in figure 2), m is the number of leaf nodes in the hieararchy, and n is the number of objects enclosed by each leaf node. Figure 3 gives pseudocode for LBVH traversal.

```
Transform the ray to LBVH space (translate and scale).
Push the root node onto the stack.
While the stack is not empty
    Pop a node off the stack.
    If the transformed ray intersects the node
        If the node is a leaf
            Intersect un-transformed ray with objects in the leaf.
            If the ray is a shadow ray and the ray intersected an object
                Return the intersection point.
        Else
            Push the children of the node onto the stack.
Return the closest intersection found or no intersection.
```

Figure 3: Algorithm to intersect a ray with an LBVH.

3 Comparison to Other Acceleration Methods

This section compares the performance of lightweight bounding volumes to median split BVHs and voxel grids. To make the comparison fair, all of the methods are built as two level hierarchies, with a separate bounding structure used for each polygon mesh. We rendered several scenes of varying complexity using each of the different acceleration schemes. The test scenes are shown in figure 4.

	Skull	Bunny	Buddha	Bunnies	Dragon	Angel
Geometry	0.61	1.85	29.0	100.1	192.8	749.1
BVH	1.40	4.24	66.4	228.9	440.6	-
Voxel grids	0.56	0.94	13.3	44.4	64.8	252.2
LBVH short	0.35	1.06	16.6	57.2	110.2	428.1
LBVH byte	0.17	0.53	8.3	28.6	55.1	214.0

Table 1: Memory usage of different bounding volume schemes compared with typical memory usage for the geometry itself. All table entries are given in megabytes.

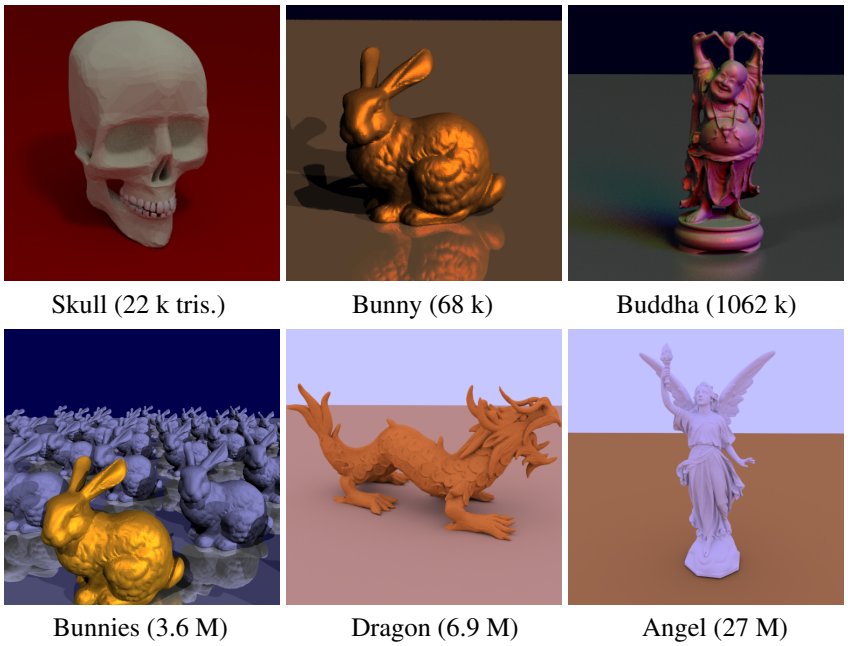

Figure 4: Scenes used to test LBVH performance, showing the number of triangles in each scene.

Memory usage. Table 1 compares the memory usage of different bounding volume schemes to the "typical" memory usage for the geometry itself. Voxel grids were initialized to have the same number of voxels as the number of enclosed objects. Bounding volumes were initialized to enclose one primitive in each leaf node. Standard BVH memory statistics are based on the BVHnode structure given in section 1. Memory usage for LBVHs using short (2 byte) integers and unsigned bytes are both listed. Triangle memory usage statistics are based on 28 bytes per triangle, assuming shared vertices. We arrive at this figure as follows: A triangle stores three 4 byte indices (12 bytes), and in a typical mesh vertices are shared by six triangles, so that each triangle carries an additional overhead of one half the cost of a vertex. If vertices store a position, normal and uv texture coordinates as 4 byte floats, a vertex takes 32 bytes, and each triangle incurs 16 additional bytes of overhead. Of course, some scenes may not require normals or texture coordinates, but decreasing the memory requirements of the triangles actually makes our technique more attractive because it increases the proportion of memory taken by the acceleration data structure.

As can be seen from table 1, standard bounding volumes with 1 primitive per leaf take more than twice as much memory as the geometry they enclose, whereas lightweight bounding volumes take only about 30 or 60 percent as much memory as the geometry (1/4 or 1/8 the memory cost of standard BVHs). Most of the voxel grid examples in the table are on par with LBVHs in terms of size, but the only example that is comparable in terms of speed, the Skull, requires 60 percent more memory than the

	Skull	Bunny	Buddha	Bunnies	Dragon	Angel
BVH	18.5	6.6	9.2	15.3	8.0	-
Voxel grids	19.4	13.0	62.7	43.2	45.3	63.6
LBVH short	20.3	7.8	9.6	18.7	7.9	9.5
LBVH byte	21.1	9.5	16.9	23.3	37.7	57.4

Table 2: Comparison of render times for different bounding volume schemes. In all cases the images were rendered at 1024×1024 pixels with a single ray sample per pixel. Table entries are given in seconds.

LBVH implementation based on short integers. We believe this to be a general trend. Voxel grid methods gain speed by increasing the number of voxels in the grid, which in turn increases the number of object pointers that must be stored.

Render speed. Table 2 compares render times for standard and lightweight bounding volumes as well as voxel grids. As expected, LBVH times are slightly, but not significantly slower than BVHs. In most cases LBVHs using short integers perform within about 10 percent of standard BVHs, with the worst cases being about 20 percent slower. However, we were unable to load the largest test model (the angel) using standard bounding volumes on a machine with 2 GB of memory. Voxel grids slightly outperformed LBVHs in the "Skull" test, but as mentioned, they required significantly more memory than LBVHs in this case. In the other tests, voxel grids used less memory, but ran much slower.

Except for the smallest scenes, the one byte version of LBVHs ran much more than the two byte version. There appears to be a direct relationship between the number of primitives in an LBVH and the performance gained by using shorts instead of bytes. Thus, a promising strategy might be to divide large polygon meshes into smaller pieces that have only a few thousand polygons each, and then encode each of these in a separate byte-based LBVH.

Number of primitives per leaf node. For the majority of the cases that we tried, placing a single primitive in each leaf node resulted in the fastest render times, but only by a small margin. This was the case for both standard and lightweight bounding volumes. Render time for our implementation tended to degrade gracefully with the addition of more primitives to leaf nodes, with an increase in render time of about 10% at 4 objects per leaf, and 30% at 8 objects per leaf. Based on these results, we suggest 4 objects per leaf as a default setting. That way, the memory needs of the hierarchy reduce to roughly 4 bytes per primitive, which is the same amount of memory used up just pointing to the objects with an explicit indexing scheme.

Acknowledgements We would like to thank the reviewers for their comments and suggestions, as well as Stanford 3D Scanning Repository and XYZ RGB Inc. for the models used in this project.

References

[Fujimoto et al., 1986] Fujimoto, A., Tanaka, T., and Iwata, K. (1986). ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics & Applications*, pages 16–26.

[Goldsmith and Salmon, 1987] Goldsmith, J. and Salmon, J. (1987). Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics & Applications*, 7(5):14–20.

[Klimaszewski and Sederberg, 1997] Klimaszewski, K. S. and Sederberg, T. W. (1997). Faster ray tracing using adaptive grids. *IEEE Computer Graphics & Applications*, 17(1):42–51.

[Mahovsky, 2005] Mahovsky, J. (2005). Ray tracing with reduced-precision bounding volume hierarchies. *PhD thesis, University of Calgary*.

[Shirley and Morley, 2003] Shirley, P. and Morley, R. K. (2003). *Realistic Ray Tracing (Second Edition)*, chapter 9, pages 141–142, ISBN: 1–56881–198–5. A K Peters.

[Smits, 1998] Smits, B. (1998). Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3(2):1–14.

[Terdiman, 2001] Terdiman, P. (2001). Memory-optimized bounding-volume hierarchies. *http://www.codecorner.com/Opcode.pdf*.

[Wald et al., 2001] Wald, I., Slusallek, P., Benthin, C., and Wagner, M. (2001). Interactive rendering with coherent ray tracing. In Chalmers, A. and Rhyne, T.-M., editors, *EG 2001 Proceedings*, volume 20(3), pages 153–164. Blackwell Publishing.