Memory-Conserving Bounding Volume Hierarchies with Coherent Ray Tracing

Jeffrey A. Mahovsky, *Member, IEEE* and Brian Wyvill, ???

Abstract—Bounding volume hierarchies (BVH) are a commonly-used method for speeding up ray tracing, but they can consume a large amount of memory for complex scenes. We present a new scheme for storing BVHs that reduces the storage requirements by up to 79%. This scheme has significant computational overhead, but this can be reduced to negligible levels by shooting bundles of rays through the BVH (coherent ray tracing). This gives the speed of a coherency-based ray tracer combined with large memory savings, permitting rendering of larger scenes and reducing the cost of a hardware ray tracer.

Index Terms— Graphics data structures and data types, Object hierarchies, Raytracing, Trees

1 Introduction

RAY tracing is a fundamentally slow operation since millions of rays must be intersected with potentially millions of objects in a three-dimensional scene. Bounding volume hierarchies (BVH) are an effective method for drastically reducing the number of ray-object intersection tests [1]. Other hierarchical schemes are octrees [2], k-d trees [3], and voxel hierarchies [4].

A BVH consists of a tree of nodes enclosing the scene's geometry. Each node is a three-dimensional bounding volume, often a box with faces parallel to the coordinate axes (known as an axis-aligned bounding box.) A node's bounding box fully encloses its children, or some geometry within the scene. When rendering, rays are tested for intersection against the nodes (boxes) in the BVH. If a ray hits a node, then the node's children are tested for intersection as well, and so on, recursively.

One problem with the BVH is the large number of nodes that may be needed to minimize rendering time. Our experimentation shows that roughly half as many BVH nodes are needed as there are objects in the scene. Hence, for a scene with 20 million objects, 10 million BVH nodes are needed. Clearly, this is a lot of overhead, so it is important to minimize the storage requirements of the BVH.

A typical BVH node will have three parts:

1. The dimensions of the axis-aligned bounding box: xmin, ymin, zmin, xmax, ymax, and zmax.
2. The number of objects enclosed by the node, if a leaf node. This value can be -1 to indicate that the node is a branch node.
3. A pointer that points to the two children (if a branch node), or points to a list of objects enclosed by the node (if a leaf node.)

The box dimensions consume 8 bytes per component if using double precision floating point, for a total of 48 bytes. The number of objects field and the pointer can be assumed to be 4 bytes each, totaling 8 bytes. Hence, each BVH node consumes 56 bytes. Note that there is only one pointer used for both children, as both children can be allocated at once as an array of two objects. This eliminates the need for a second pointer and reduces the amount of memory management.

Based on these estimates, a BVH containing 10 million nodes will consume 560 million bytes of memory. This is a lot of overhead, and could potentially be better used to store geometry and/or textures.

Single-precision floating point values may be used for the box dimensions, reducing the node's storage requirements to 32 bytes. The use of single-precision FP may introduce subtle artifacts into the image, so double-precision coordinates will be used for testing.

2 Memory-efficient BVH encoding

2.1 Summary

It is possible to replace the 64-bit double precision floating point box dimensions with 8-bit unsigned integers without affecting image quality, reducing the node size to 12 bytes. (Additionally, the 32-bit number of objects value is shortened to 16 bits. The 32-bit value wasn't strictly necessary, but it preserved proper structure member alignment and made the structure's size an even multiple of 8 bytes.) Using 12 bytes per BVH node equals a savings of 79% versus using 64-bit double precision, and 63% versus 32-bit single precision floating point.

The key to the memory savings is that the 8-bit unsigned integers only encode as much information as is absolutely necessary to reconstruct an approximation of the original double-precision coordinates. The reconstructed boxes are slightly larger than the originals, but are guaranteed to fully enclose the original boxes. This ensures that no boxes will be accidentally missed due to the loss of precision in the encoding, and correct images will be produced. There will be some false hits, however, resulting in some excess hierarchy traversal.

Encoding of the 8-bit box coordinates is based on a hier-

• *J. A. Mahovsky is with the University of Calgary, Calgary, Alberta, Canada. E-mail: mahovskj@cpsc.ucalgary.ca.*
• *B. Wyvill is with the University of Calgary, Calgary, Alberta, Canada. E-mail: blob@cpsc.ucalgary.ca.*

Manuscript received (insert date of submission if desired).

archy of coordinate systems. Each node has its own integer coordinate system in the range [0, 255] in x, y, and z. The node's children are then represented by coordinate values in this [0, 255] coordinate system. These children then have their own [0, 255] coordinate system, and so forth. When rendering, these [0, 255] coordinates are converted back into the normal double precision world space coordinates for the ray-box intersection tests. This conversion adds some computational overhead.

2.2 Hierarchy Construction

BVH construction is based on the method described by Shirley et al. [5]. To summarize, construction is a recursive operation where nodes are subdivided into two children until some termination criteria is met. Each node is split in half by a splitting plane parallel to one of the principal axes, and the center points of the objects are compared to the splitting plane. (We use geometry consisting of only triangles, thus the centroid of the triangles are used.) The objects with centers on one side of the plane belong to one child, and vice-versa.

Construction terminates when the number of objects in a node is less than a threshold value, or the depth of the hierarchy reaches a maximum value. Our experiments show that a threshold value of between 4 and 8 maximum objects per node will minimize rendering time. (A value of 7 was chosen for the tests in this paper. Values closer to 4 reduce rendering time by a few percent but greatly increase the number of BVH nodes.) The maximum hierarchy depth was chosen to be 60, although no scenes had hierarchies that deep. BVHs are guaranteed to terminate and not produce hierarchies of infinite depth (at least when using the algorithm in the reference.) 60 was chosen because it is a depth greater than that normally encountered in a BVH (using our test scenes), and will prevent the BVH construction from producing an unusually deep tree if given some sort of degenerate scene data.

First, the regular BVH construction method is presented as pseudocode:

```
procedure Build(node, list, depth, axis)
    (Determine the bounding box of the objects in 'list' and assign
    to node.xmin, node.xmax, node.ymin, node.ymax, node.zmin
    and node.zmax.)
    if list.length <= 7 or depth = 60
        (Leaf node)
        node.numobjects = list.length
        node.pointer = list
    else
        (Branch node)
        node.numobjects = -1
        children[] = new BVH_Node[2]
        node.pointer = children
        (Choose splitting plane based on 'axis' parameter and node-
        bounding box.)
        child0list = (objects from 'list' on - side of splitting plane)
        child1list = (objects from 'list' on + side of splitting plane)
        Build(children[0], child0list, depth + 1, (axis + 1) mod 3)
        Build(children[1], child1list, depth + 1, (axis + 1) mod 3)
```

To generate 8-bit BVH nodes, Build() uses the dimensions of the parent node to compute the 8-bit coordinate values for the current node. Each node has its own [0, 255] integer coordinate system for its children, forming a hierarchy of coordinate systems.

```
procedure Build(node, list, depth, axis,
        pXOrigin, pYOrigin, pZOrigin,
        pXWidth, pYWidth, pZWidth)

    (Determine the bounding box of the objects in 'list' and assign
    to temporary values nxmin, nxmax, nymin, nymax, nzmin and
    nzmax.)

    (Compute the 8-bit coordinate values with range [0, 255] based
    on the actual coordinates nxmin, etc. and the dimensions of the
    parent node. The floor() and ceil() operations enlarge the box
    by rounding the minimum coordinates down, and the maxi-
    mum coordinates up. This ensures the 8-bit precision box co-
    ordinates enclose the actual box.)

    node.xmin = floor(((nxmin – pXOrigin) / pXWidth) * 255)
    node.ymin = floor(((nymin – pYOrigin) / pYWidth) * 255)
    node.zmin = floor(((nzmin – pZOrigin) / pZWidth) * 255)
    node.xmax = ceil(((nxmax – pXOrigin) / pXWidth) * 255)
    node.ymax = ceil(((nymax – pYOrigin) / pYWidth) * 255)
    node.zmax = ceil(((nzmax – pZOrigin) / pZWidth) * 255)

    (Clamp these values to [0, 255] as floating point imprecision
    may result in values very slightly outside this range.)

    if list.length <= 7 or depth = 60
        (Leaf node)
        node.numobjects = list.length
        node.pointer = list
    else
        (Branch node)
        node.numobjects = -1
        children[] = new BVH_Node[2]
        node.pointer = children
        (Choose splitting plane based on 'axis' parameter and node
        bounding box coordinates nxmin, nymin, nzmin, nxmax,
        nymax, nzmax.)
        child0list = (objects from 'list' on - side of splitting plane)
        child1list = (objects from 'list' on + side of splitting plane)

        (Convert the node's 8-bit coordinate values back to the regu-
        lar coordinate system with double precision values. Convert
        these into the x, y, and z origin and width values for the chil-
        dren's coordinate system. Note that this computes a slightly
        larger box than nxmin, nxmax, etc.)

        xOrigin = pXOrigin + (node.xmin * pXWidth) / 255
        yOrigin = pYOrigin + (node.ymin * pYWidth) / 255
        zOrigin = pZOrigin + (node.zmin * pZWidth) / 255
        xWidth = (node.xmax – node.xmin) * pXWidth / 255
        yWidth = (node.ymax – node.ymin) * pYWidth / 255
        zWidth = (node.zmax – node.zmin) * pZWidth / 255

        Build(children[0], child0list, depth + 1, (axis + 1) mod 3,
        xOrigin, yOrigin, zOrigin, xWidth, yWidth, zWidth)
        Build(children[1], child1list, depth + 1, (axis + 1) mod 3,
        xOrigin, yOrigin, zOrigin, xWidth, yWidth, zWidth)
```

One pitfall encountered with computing the 8-bit integer coordinates is due to the inherent imprecision of floating-point arithmetic. Before converting to integer, values slightly greater than 255 or less than zero may result, producing integer coordinates of -1 or 256 after applying floor() and ceil(). The 8-bit coordinates must be clamped to [0, 255] to avoid this problem.

Hierarchy construction starts by calling Build() with the origin and width parameters being equal to the scene boundaries. To simplify things, the scene is scaled to fit within a cube of dimensions (-1, -1, -1) to (1, 1, 1). Thus, the Build() call is:

```
Build(rootnode, allobjects, 0, 0, -1, -1, -1, 2, 2, 2)
```

2.3 Hierarchy Traversal

Hierarchy traversal is straightforward. If a ray hits a node, both of that node's children are tested for intersection. The process proceeds recursively until a leaf node containing geometry is reached, and the geometry is tested for intersection. In pseudocode:

```
procedure Traverse(node, ray)

  if ray hits node
    if node.numobjects != -1
      (intersect ray with geometry contained within the node)
  else
      Traverse(node.child0, ray)
      Traverse(node.child1, ray)
```

For the purpose of this paper it does not matter how the "ray hits node" test is performed. We use the Pluecker coordinate-based method described in [6] as it is somewhat faster than the older slabs-based method [7]. Also, better performance is achieved if the order of child testing is optimized to match the direction of the ray and the axis of the splitting plane that separated the children [8], but that is beyond the scope of this paper.

Several changes were made to Traverse() to accommodate the BVH nodes with 8-bit box dimensions. It must generate the actual coordinates of the bounding box based on the node's 8-bit coordinate values and the parent node's dimensions. These coordinates are then used for the children's traversal. In pseudocode:

```
procedure Traverse(node, ray,
    pXOrigin, pYOrigin, pZOrigin,
    pXWidth, pYWidth, pZWidth)

  (Generate the node's actual box dimensions from the node's 8-
  bit dimensions and the parent's actual dimensions.)
  xmin = pXOrigin + pXWidth * node.xmin
  ymin = pYOrigin + pYWidth * node.ymin
  zmin = pZOrigin + pZWidth * node.zmin
  xmax = pXOrigin + pXWidth * node.xmax
  ymax = pYOrigin + pYWidth * node.ymax
  zmax = pZOrigin + pZWidth * node.zmax

  if ray hits node
    if node.numobjects != -1
    (Intersect ray with geometry contained within the node.)
  else
    (Compute the width parameters, and scale by 1/255 so the
    children's coordinates are properly scaled when the width is
    multiplied by the integer coordinate.)
    xWidth = (xmax - xmin) * 1/255
    yWidth = (ymax - ymin) * 1/255
    zWidth = (zmax - zmin) * 1/255
    Traverse(node.child0, ray, xmin, ymin, zmin, xWidth,
    yWidth, zWidth)
    Traverse(node.child1, ray, xmin, ymin, zmin, xWidth,
    yWidth, zWidth)
```

Hierarchy traversal starts by calling Traverse() with the origin and width parameters set to the scene boundaries. To simplify things, the scene is scaled to fit within a cube of dimensions (-1, -1, -1) to (1, 1, 1). Thus, traversal starts with:

```
Traverse(rootnode, ray, -1, -1, -1, 2/255, 2/255, 2/255)
```

Note that the width is 2/255 and not 2. This is to ensure that the reconstructed box coordinates are scaled properly.

2.4 Test Results

Several test scenes were used, varying from thousands to millions of triangles (Fig. 1). Only triangles were used for geometry, although results should be similar using other types of objects.

Teapot: Polygonal Utah teapot with mirrored background. 2,262 triangles, 2 lights.
Bunny: Psychedelic reflective Stanford bunny in a colored box. 69,463 triangles, 2 lights.
Branch: Branch model from the U of Calgary VIC lab. 312,706 triangles, 2 lights.
Poppy: Poppy model from the U of Calgary VIC lab [9]. 395,460 triangles, 2 lights.
Powerplant-front: Front view of UNC powerplant model [10]. 12,748,512 triangles, 1 light.
Powerplant-north: North/bottom view of UNC powerplant model. 12,748,512 triangles, 3 lights.
Powerplant-boiler: View inside the boiler of the UNC powerplant. 12,748,512 triangles, 1 light.
Lucy: Angelic statue [11]. 28,057,792 triangles, 2 lights.

Tests were run on a 3.0 GHz Pentium-4 system with 2GB RAM running Microsoft Windows XP. The compiler was Microsoft Visual C++ .NET 2003.

Tests were run three times, taking the lowest time. The times varied by a small fraction of a percent. Images were rendered at 2048x2048 pixel resolution with no anti-aliasing. This would be roughly equivalent to 1024x1024 pixel resolution with 4-sample anti-aliasing. The test images were also compared to the reference images to ensure that no pixels differed.

During testing, it was discovered that 8-bit integer to double conversions were a major computational bottleneck. The solution was to use a 256-entry look-up table (LUT) for the conversions, containing the double-precision values [0, 255]. This improved performance considerably.

Several different types of statistics were recorded for each of the scenes:

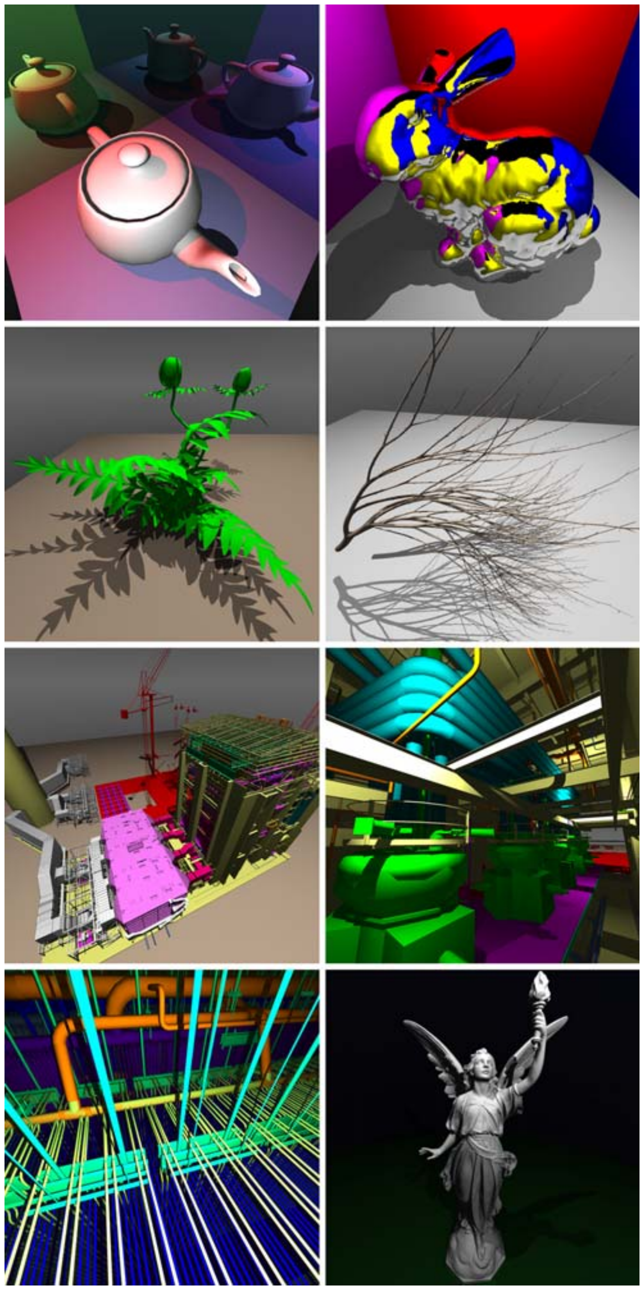
Fig. 1. Test scenes. Top to bottom: Teapot, Bunny, Poppy, Branch, Powerplant-front, Powerplant-north, Powerplant-boiler, Lucy.

Nodes: The number of BVH nodes in the scene.
Node storage: Total bytes consumed by the BVH nodes. Reference nodes consume 56 bytes each and 8-bit nodes consume 12 bytes each.
Nodes tested (shadow/closest): Total number of ray-BVH node intersection tests.
Tris tested (shadow/closest): Total number of ray-triangle intersection tests.
Time (s): Rendering time.
Time w/LUT (s): Rendering time using a 256-entry look-up table for 8-bit integer to double conversions.

Traversal statistics were split into shadow/closest categories to determine if there was any significant difference between regular rays and shadow rays. Regular rays (primary, reflected, and refracted rays) return the closest intersection point, and shadow rays return just a true/false value, permitting special optimizations.

Table 1 presents the results for BVH nodes using 64-bit double precision versus 8-bit integer values. The percentage relative to 64-bit double precision is shown, with lower values being better.

The results show substantial memory savings. Hierarchy memory usage is reduced by nearly 79% for all scenes when using BVH nodes with 8-bit integer values. Half a gigabyte of memory is saved for the complex Lucy scene.

The statistics show that despite the nodes having their dimensions truncated to 8-bits, the total number of nodes traversed and the number of ray-triangle intersection tests is typically increased by 1% or less (although Powerplant-front and Powerplant-north show somewhat larger increases.) The increase is consistent for even the most complex scenes, hence the algorithm scales well. Additional tests were run using 12- and 16-bit precision, where the increase in node traversal and triangle intersection tests was virtually zero. Rendering time was nearly unchanged, suggesting that the bottleneck was not the excess traversal but rather the decompression of the nodes.

Unfortunately, rendering time was increased by between 46.08% and 65.95% when using 8-bit precision. Using a look-up table for the integer to double conversions reduced the overhead to between 27.76% and 41.35%. In the following section, we propose a solution to this problem.

4 Coherent Ray Tracing

4.1 Summary

Coherent ray tracing exploits the similarity between rays to improve performance. For example, the 64 rays for any 8 by 8 pixel region of the screen will be similar, and will intersect similar areas of the scene. Other researchers have used coherency to reduce the memory bandwidth requirements of ray tracing, by shooting 4 rays at once with Intel's SSE instructions [12], or by shooting 64 rays at once for a prototype hardware design [13]. These batches of rays are traversed through the hierarchy (either a BVH or BSP/k-d tree) as groups, and are intersected with the geometry of the scene in groups. This can reduce the number of hierarchy nodes and geometry fetches from memory by up to a factor of n, when shooting bundles of n rays. Major rendering speed increases have been observed, as fetches from memory are no longer a bottleneck and the CPU or hardware can run at full speed without waiting for memory accesses to complete.

Coherent ray tracing solves the problem of the integer BVH node decompression overhead by amortizing the decompression cost over many rays. The decompression need only be performed once each time the node is loaded from memory, and not for each ray. This should dramatically reduce the overhead.

TABLE 1
64-BIT DOUBLE VS. 8-BIT INTEGER

Teapot	**64-bit double**	**8-bit integer**	**%**
Nodes	1,045	1,045	100.00%
Node storage (bytes)	58,520	12,540	21.43%
Nodes Tested (Shadow)	217,657,769	219,271,249	100.74%
Tris Tested (Shadow)	58,595,114	59,116,061	100.89%
Nodes Tested (Closest)	159,362,863	160,401,051	100.65%
Tris Tested (Closest)	39,671,827	39,935,527	100.66%
Time (s)	17.48	25.66	146.80%
Time w/LUT (s)		23.73	135.76%

Bunny			
Nodes	29,231	29,231	100.00%
Node storage (bytes)	1,636,936	350,772	21.43%
Nodes Tested (Shadow)	257,473,712	258,894,504	100.55%
Tris Tested (Shadow)	85,176,399	85,424,896	100.29%
Nodes Tested (Closest)	362,223,897	365,230,447	100.83%
Tris Tested (Closest)	101,742,900	102,329,316	100.58%
Time (s)	30.73	44.89	146.08%
Time w/LUT (s)		39.26	127.76%

Poppy			
Nodes	164,905	164,905	100.00%
Node storage (bytes)	9,234,680	1,978,860	21.43%
Nodes Tested (Shadow)	159,363,367	160,756,199	100.87%
Tris Tested (Shadow)	28,432,754	28,644,869	100.75%
Nodes Tested (Closest)	96,539,404	97,453,770	100.95%
Tris Tested (Closest)	12,120,010	12,208,745	100.73%
Time (s)	12.27	18.29	149.06%
Time w/LUT (s)		16.07	130.97%

Branch			
Nodes	134,995	134,995	100.00%
Node storage (bytes)	7,559,720	1,619,940	21.43%
Nodes Tested (Shadow)	210,384,634	212,163,626	100.85%
Tris Tested (Shadow)	31,688,171	31,895,734	100.66%
Nodes Tested (Closest)	225,745,754	227,718,420	100.87%
Tris Tested (Closest)	20,044,002	20,206,784	100.81%
Time (s)	19.24	29.88	155.30%
Time w/LUT (s)		26.2	136.17%

Powerplant - Front	**64-bit double**	**8-bit integer**	**%**
Nodes	5,612,947	5,612,947	100.00%
Node storage (bytes)	314,325,032	67,355,364	21.43%
Nodes Tested (Shadow)	291,090,401	297,062,657	102.05%
Tris Tested (Shadow)	45,951,546	46,806,043	101.86%
Nodes Tested (Closest)	456,537,468	464,241,744	101.69%
Tris Tested (Closest)	77,264,665	78,357,240	101.41%
Time (s)	33.72	50.74	150.47%
Time w/LUT (s)		43.97	130.40%

Powerplant - North			
Nodes	5,612,947	5,612,947	100.00%
Node storage (bytes)	314,325,032	67,355,364	21.43%
Nodes Tested (Shadow)	2,752,008,349	2,780,655,289	101.04%
Tris Tested (Shadow)	240,590,073	245,020,915	101.84%
Nodes Tested (Closest)	1,507,036,580	1,564,907,034	103.84%
Tris Tested (Closest)	142,308,159	144,182,126	101.32%
Time (s)	170.91	283.62	165.95%
Time w/LUT (s)		241.58	141.35%

Powerplant - Boiler			
Nodes	5,612,947	5,612,947	100.00%
Node storage (bytes)	314,325,032	67,355,364	21.43%
Nodes Tested (Shadow)	1,434,810,614	1,443,320,976	100.59%
Tris Tested (Shadow)	158,896,106	160,393,303	100.94%
Nodes Tested (Closest)	1,672,287,732	1,682,661,656	100.62%
Tris Tested (Closest)	233,628,723	235,702,651	100.89%
Time (s)	128.44	207.98	161.93%
Time w/LUT (s)		170.65	132.86%

Lucy			
Nodes	11,702,495	11,702,495	100.00%
Node storage (bytes)	655,339,720	140,429,940	21.43%
Nodes Tested (Shadow)	734,968,446	740,996,576	100.82%
Tris Tested (Shadow)	182,251,169	183,525,921	100.70%
Nodes Tested (Closest)	333,597,274	336,596,062	100.90%
Tris Tested (Closest)	77,618,571	78,133,207	100.66%
Time (s)	53.65	79.95	149.02%
Time w/LUT (s)		74.3	138.49%

Our implementation of coherent ray tracing does not use any special hardware or vector CPU instructions. We shoot 64 or 256 rays at a time from the eye and send them through the hierarchy. We want to shoot as many rays as possible, but if too many rays are shot at once, the rendering speed stops increasing because the set of rays becomes larger than the CPU caches. 64 rays fit within level 1 CPU cache (typically 8-16 kilobytes to today's CPUs.) Assuming approximately 100 bytes per ray, 6400 bytes of storage is needed. We also tested with 256 rays to test the technique's scalability, but did not observe any further speed increase. This is likely because 256 rays are too large to fit within the level 1 cache of today's CPUs, and the cache misses cancel the advantage from the greater coherency.

After ray traversal and intersection completes, the intersection results for the rays are examined. When shooting 64 rays, there will be a maximum of 64 intersection points. For each intersection point, a shadow ray is directed to the first light source. These (up to 64) shadow rays are collected into a group that is traced through the scene. The process repeats for each light source. Depending on the material properties of the intersections, groups of transmitted and/or reflected rays are also created and traced through the scene.

Slight modifications are needed to the BVH traversal to accommodate groups of rays. Instead of sending a single ray to the traversal function, the function retrieves rays via a separate stack of pointers to rays. A parameter is passed to the function indicating the number of ray pointers to read from the top of the stack. The rays are retrieved and are tested for intersection with the BVH node. Intersecting rays have their pointers pushed onto the stack and these are passed to the node's children. (Recall that for a BVH, both children are tested against the same set of rays that intersect the parent.) When the children's tests finish, the "top of stack" pointer is decremented by the number of rays that were passed to the children, and the recursion un-rolls. If shooting a maximum of 64 rays at once into a hierarchy a maximum of 60 levels deep, the stack must accommodate 64 * 60 or 3840 ray pointers.

Ray-triangle intersection proceeds in a similar fashion, with the function reading ray pointers from the top of the stack and intersecting them with each triangle belonging to the node.

4.2 Test Results

TABLE 2
64-BIT DOUBLE VS. 8-BIT INTEGER FOR 64-RAY (8X8 PIXELS) COHERENT RAY TRACING

Teapot	64-bit double 8x8 pixels	8-bit integer 8x8 pixels	%
Nodes	1,045	1,045	100.00%
Node storage (bytes)	58,520	12,540	21.43%
Nodes Loaded (Shadow)	3,946,135	3,974,323	100.71%
Nodes Tested (Shadow)	225,842,488	227,455,968	100.71%
Tris Loaded (Shadow)	1,291,052	1,306,696	101.21%
Tris Tested (Shadow)	61,336,086	61,857,033	100.85%
Nodes Loaded (Closest)	2,640,985	2,658,237	100.65%
Nodes Tested (Closest)	159,362,863	160,401,051	100.65%
Tris Loaded (Closest)	734,506	739,671	100.70%
Tris Tested (Closest)	39,671,827	39,935,527	100.66%
Time (s)	11.82	12.14	102.71%
Time w/LUT (s)		12	101.52%

Powerplant - Front	64-bit double 8x8 pixels	8-bit integer 8x8 pixels	%
Nodes	5,612,947	5,612,947	100.00%
Node storage (bytes)	314,325,032	67,355,364	21.43%
Nodes Loaded (Shadow)	8,177,312	8,352,056	102.14%
Nodes Tested (Shadow)	293,915,132	299,777,638	101.99%
Tris Loaded (Shadow)	2,548,282	2,607,956	102.34%
Tris Tested (Shadow)	46,154,482	46,992,405	101.82%
Nodes Loaded (Closest)	10,161,150	10,342,016	101.78%
Nodes Tested (Closest)	456,537,468	464,241,744	101.69%
Tris Loaded (Closest)	3,066,046	3,121,845	101.82%
Tris Tested (Closest)	77,264,665	78,357,240	101.41%
Time (s)	20.8	21.68	104.23%
Time w/LUT (s)		20.82	100.10%

Bunny			
Nodes	29,231	29,231	100.00%
Node storage (bytes)	1,636,936	350,772	21.43%
Nodes Loaded (Shadow)	6,249,540	6,286,164	100.59%
Nodes Tested (Shadow)	294,155,424	295,696,250	100.52%
Tris Loaded (Shadow)	2,409,237	2,418,628	100.39%
Tris Tested (Shadow)	96,453,960	96,714,495	100.27%
Nodes Loaded (Closest)	7,939,636	8,016,146	100.96%
Nodes Tested (Closest)	362,223,897	365,498,249	100.90%
Tris Loaded (Closest)	2,923,517	2,957,223	101.15%
Tris Tested (Closest)	101,742,900	102,585,741	100.83%
Time (s)	20.2	20.94	103.66%
Time w/LUT (s)		20.47	101.34%

Powerplant – North			
Nodes	5,612,947	5,612,947	100.00%
Node storage (bytes)	314,325,032	67,355,364	21.43%
Nodes Loaded (Shadow)	51,310,807	51,835,651	101.02%
Nodes Tested (Shadow)	2,864,358,472	2,892,203,588	100.97%
Tris Loaded (Shadow)	5,649,641	5,773,878	102.20%
Tris Tested (Shadow)	253,591,881	257,898,843	101.70%
Nodes Loaded (Closest)	24,830,582	25,754,488	103.72%
Nodes Tested (Closest)	1,507,036,580	1,564,907,034	103.84%
Tris Loaded (Closest)	2,675,686	2,715,441	101.49%
Tris Tested (Closest)	142,308,159	144,182,126	101.32%
Time (s)	91.98	97.39	105.88%
Time w/LUT (s)		94.94	103.22%

Poppy			
Nodes	164,905	164,905	100.00%
Node storage (bytes)	9,234,680	1,978,860	21.43%
Nodes Loaded (Shadow)	4,079,456	4,108,482	100.71%
Nodes Tested (Shadow)	160,676,978	161,999,680	100.82%
Tris Loaded (Shadow)	1,892,513	1,901,481	100.47%
Tris Tested (Shadow)	28,690,704	28,875,393	100.64%
Nodes Loaded (Closest)	1,987,097	2,006,623	100.98%
Nodes Tested (Closest)	96,539,404	97,551,516	101.05%
Tris Loaded (Closest)	649,529	654,854	100.82%
Tris Tested (Closest)	12,120,010	12,246,848	101.05%
Time (s)	8.24	8.53	103.52%
Time w/LUT (s)		8.41	102.06%

Powerplant – Boiler			
Nodes	5,612,947	5,612,947	100.00%
Node storage (bytes)	314,325,032	67,355,364	21.43%
Nodes Loaded (Shadow)	31,336,418	31,575,760	100.76%
Nodes Tested (Shadow)	1,449,581,682	1,458,092,044	100.59%
Tris Loaded (Shadow)	6,484,661	6,569,452	101.31%
Tris Tested (Shadow)	161,214,896	162,712,093	100.93%
Nodes Loaded (Closest)	34,192,615	34,403,589	100.62%
Nodes Tested (Closest)	1,672,287,732	1,679,719,406	100.44%
Tris Loaded (Closest)	7,753,776	7,845,348	101.18%
Tris Tested (Closest)	233,628,723	235,423,672	100.77%
Time (s)	70.58	74.15	105.06%
Time w/LUT (s)		71.62	101.47%

Branch			
Nodes	134,995	134,995	100.00%
Node storage (bytes)	7,559,720	1,619,940	21.43%
Nodes Loaded (Shadow)	6,302,553	6,349,897	100.75%
Nodes Tested (Shadow)	215,401,086	217,165,042	100.82%
Tris Loaded (Shadow)	2,520,422	2,533,958	100.54%
Tris Tested (Shadow)	32,651,548	32,855,559	100.62%
Nodes Loaded (Closest)	4,304,145	4,340,357	100.84%
Nodes Tested (Closest)	225,745,754	227,718,420	100.87%
Tris Loaded (Closest)	965,348	971,072	100.59%
Tris Tested (Closest)	20,044,002	20,206,784	100.81%
Time (s)	13	13.46	103.54%
Time w/LUT (s)		13.33	102.54%

Lucy			
Nodes	11,702,495	11,702,495	100.00%
Node storage (bytes)	655,339,720	140,429,940	21.43%
Nodes Loaded (Shadow)	28,377,605	28,543,457	100.58%
Nodes Tested (Shadow)	743,310,568	749,281,424	100.80%
Tris Loaded (Shadow)	21,636,645	21,770,691	100.62%
Tris Tested (Shadow)	182,388,463	183,615,654	100.67%
Nodes Loaded (Closest)	10,576,058	10,647,876	100.68%
Nodes Tested (Closest)	333,597,274	336,596,062	100.90%
Tris Loaded (Closest)	7,569,463	7,623,863	100.72%
Tris Tested (Closest)	77,618,571	78,133,207	100.66%
Time (s)	34.67	35.64	102.80%
Time w/LUT (s)		34.6	99.80%

The testing method is the same as before, except *nodes loaded* and *tris loaded* fields have been added. Nodes loaded indicates the number of BVH nodes loaded from memory and decompressed back into double-precision form. Tris loaded indicates the number of triangles loaded from memory. These fields monitor the effectiveness of shooting bundles of rays versus single rays. If rays are shot in bundles of 64, the values of nodes loaded and tris loaded would ideally be 1/64th of the nodes tested and tris tested. This ideal value is never reached because the ray bundles can be broken up as they traverse the hierarchy, since not all rays traverse the same nodes. (For the non-coherent case in the previous section, the number of nodes loaded equals the number of nodes tested, and the number of triangles loaded equals the number of triangles tested.) The results using coherent ray tracing are presented in Table 2.

These results are much better than the previous results. Exploiting ray coherency reduces the overhead by more than an order of magnitude. Using the LUT, the times range from 99.80% of the reference (slightly faster) to 103.22%. Note that the reference is now a ray coherency-

TABLE 3
64-BIT DOUBLE VS. 8-BIT INTEGER FOR 256-RAY (16X16 PIXELS) COHERENT RAY TRACING

Teapot	64-bit double 16x16 pixels	8-bit integer 16x16 pixels	%
Time (s)	47.7	48.87	102.45%
Time w/LUT (s)		48.09	100.82%
Bunny			
Time (s)	80.24	82.13	102.36%
Time w/LUT (s)		80.45	100.26%
Poppy			
Time (s)	31.91	32.96	103.29%
Time w/LUT (s)		32.3	101.22%
Branch			
Time (s)	50.29	51.87	103.14%
Time w/LUT (s)		50.88	101.17%
Powerplant - Front			
Time (s)	79.33	81.99	103.35%
Time w/LUT (s)		79.59	100.33%
Powerplant - North			
Time (s)	369.56	384.75	104.11%
Time w/LUT (s)		375.96	101.73%
Powerplant - Boiler			
Time (s)	276.2	284.95	103.17%
Time w/LUT (s)		277.3	100.40%
Lucy			
Time (s)	133.95	134.03	100.06%
Time w/LUT (s)		131.58	98.23%

based reference, which is much faster than the non-coherency reference. Hence, the speed of a coherency-based ray tracer has been combined with the memory savings of BVH nodes using 8-bit integer coordinates.

Comparing loaded versus tested statistics for the scenes confirms that ray coherency drastically reduces the number of node and triangle fetches from memory. As a result, rendering times are decreased substantially as well. With no special hardware or CPU instructions, speed-ups of 48% to 86% are obtained.

Increasing the coherency by increasing the image resolution and the bundle size should reduce the overhead further. To test this, the scenes were also rendered at 4096x4096 pixel resolution, shooting a 16x16 pixel group of rays at once (256 rays). Although 4096x4096 pixel resolution may seem excessively high, it is roughly equivalent to 2048x2048 anti-aliased pixels. Today (2004), Apple Computer's 30-inch Cinema Display sports 2560x1600 pixel resolution [14], and large, high resolution displays like these will become more commonplace as technology improves.

For brevity, only the rendering times are presented for the 4096x4096 pixel results in Table 3. The memory savings is the same regardless of the size of the ray bundles.

With the increased coherency, rendering times now fall within the 98.23% to 101.73% range. The overhead from using low-precision boxes has become negligible.

5 CONCLUSIONS AND FUTURE WORK

Our results show that a BVH node's box dimensions can be effectively represented as 8-bit values in a hierarchy of coordinate systems, instead of using 32- or 64-bit floating point values. This reduces the size of each BVH node by 79% versus using 64-bit floating point and by 63% versus 32-bit floating point, saving hundreds of megabytes of memory for scenes with millions of objects.

Correct images are produced despite the low precision, because the low precision boxes are guaranteed to enclose the 64-bit double precision boxes they approximate. Unfortunately, this incurs a run-time overhead that reduces rendering speed by up to 41%, because the 8-bit dimensions must be converted back into a usable 64-bit floating point form before performing a ray-box overlap test. A small amount of excess hierarchy traversal and ray-triangle intersection tests also happens, reducing performance slightly.

Coherent ray tracing solves the performance problem by amortizing the conversion cost over many rays. A node's dimensions are converted once for a group of many rays, instead of once for each ray. This reduces the run-time overhead to negligible levels, while retaining the 63% or 79% memory savings.

The memory savings permits rendering of larger scenes, since more memory can be devoted to geometry. Scenes that do not fit within physical memory with a conventional BVH may now fit, reducing the need for out-of-core rendering. The cost of a hardware ray tracer is also reduced because the memory array can be smaller. This savings could be invested in a faster grade of memory, accelerating the ray tracing.

The results also suggest that scene geometry may also be stored in a more memory-efficient form that requires computationally-intensive decompression. The challenge with scene geometry is compressing it without affecting the image quality.

ACKNOWLEDGMENTS

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC.)

REFERENCES

[1] J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, vol. 7, no. 5, 1987.

[2] A. Glassner, "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, vol. 4, no. 10, 1984.

[3] D. Fussell and K. R. Subramanian, "Fast Ray Tracing using K-D Trees," Technical Report TR-88-07, Department of Computer Sciences, University of Texas at Austin, 1988.

[4] D. Jevans and B. Wyvill, "Adaptive Voxel Subdivision for Ray Tracing," *Proc. Graphics Interface '89*, pp. 164-172, 1989.

[5] P. Shirley and R.K. Morley, *Realistic Ray Tracing, Second Edition*, Natick, Mass.: A.K. Peters Ltd., pp. 131-143, 2003.

[6] J. Mahovsky and Brian Wyvill, "Fast Ray-Axis Aligned Bounding Box Overlap Tests with Plücker Coordinates," *Journal of Graphics Tools*, vol. 9 no. 1, 2004.

[7] B. Smits, "Efficiency Issues for Ray Tracing," *Journal of Graphics*

Tools, vol. 3 no. 2, 1998.

[8] J. Mahovsky, “Follow up to ‘Fast Ray-Axis Aligned Bounding Box Overlap Tests with Plücker Coordinates’,” Technical Report 2004-759-24, University of Calgary, 2004.

[9] P. MacMurchy, “The Use of Subdivision Surfaces in the Modeling of Plants,” Master’s dissertation, Dept. of Computer Science, University of Calgary, 2004.

[10] Various, “The Walkthru Project – Power Plant Model Release,” http://www.cs.unc.edu/~geom/Powerplant/. 2004.

[11] Various, “The Stanford 3D Scanning Repository,” http://graphics.stanford.edu/data/3Dscanrep/. 2004.

[12] I. Wald, P. Slusallek and C. Benthin, “Interactive Rendering with Coherent Ray Tracing,” *Computer Graphics Forum*, vol. 20 no. 3, 2001.

[13] J. Schmittler, I. Wald and P. Slusallek, “SaarCOR – A Hardware Architecture for Ray Tracing,” *proc. Graphics Hardware 2002*, pp. 27-36, 2002.

[14] Apple Computer, Inc., “Apple Cinema Displays,” http://www.apple.com/displays/specs.html. 2004.

Jeffrey A. Mahovsky is a PhD Candidate at the University of Calgary. His degrees include a Bachelor of Applied Science in Electronic Systems Engineering (1999) from the University of Regina and a Master of Applied Science in Electronic Systems Engineering (2001) from the University of Regina. His current research interests focus on ray tracing techniques. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Brian Wyvill …