Volume xx (200y), Number z, pp. 1–14

Robust Ray-Bounding Volume Hierarchy Traversal with Reduced Precision Integer Arithmetic

J. Mahovsky[1] and B. Wyvill[1] and A. Davis[2] and A. Ibrahim[2]

[1]University of Calgary, Calgary, Alberta, Canada
[2]University of Utah, Salt Lake City, Utah, USA

Abstract

We present a new ray-bounding volume hierarchy (BVH) traversal scheme that uses reduced precision integer arithmetic instead of floating point. Error bounds in the traversal's ray-box test guarantee that a correct image is produced, regardless of the precision of the arithmetic used. Thus, simpler integer arithmetic hardware can be used in a hardware-based ray tracer instead of floating point. We present a theoretical hardware ray-box test circuit and demonstrate a 6-fold reduction in circuit area. Memory is also saved because the BVH node dimensions are represented with lower-precision values.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Raytracing

1. Introduction

Bounding volume hierarchies (BVH) are an effective method for speeding up ray tracing and avoiding the need to test every ray against every object in the scene [KK86]. BVHs are one method in a range of acceleration schemes, such as grids [CW88], octrees [Gla84], grid hierarchies [JW89], and BSP or k-d trees [FS88] [Hav01].

The bottleneck in ray tracers using a BVH (or any other acceleration scheme) is the traversal of the hierarchy or grid. An efficient ray-BVH traversal method is crucial to maximize performance. Most BVHs use axis-aligned bounding boxes (AABBs) as bounding volumes, although other bounding shapes can also be used. The core operation in BVH traversal is the ray-AABB test and is the most important operation to optimize.

We present a new method for performing ray-AABB tests using reduced-precision integer arithmetic. By using 16-24 bit integer arithmetic instead of 32-bit floating-point, the cost of a hardware ray-AABB test is greatly reduced. Circuit area is reduced by a factor of six, allowing a possible six-fold increase in ray-AABB test speed because six times as many ray-AABB circuits can be placed on a chip. Ray tracing is a 'trivially parallel' problem, because rays are independent, thus six times as many rays can be traversed through a BVH at once. This ignores the details of object intersections and shading, which much be handled seperately, so the overall rendering speedup will be less than six.

Our reduced-precision ray-AABB test is based on the Plücker-coordinate derived technique [MW04]. This method is a prime candidate for integer implementation because it consists of only multiplies, subtractions, and comparisions of finite values. By analyzing the error characteristics of the arithmetic operations, a fully-robust, 'conservative' ray-AABB test is derived. The conservative test assumes a ray-box hit if there is uncertainty due to numerical imprecision. This ensures that no hierarchy nodes are accidentally missed due to imprecision, but results in some excess hierarchy traversal. The excess traversal is greater when using lower numeric precisions.

Robust integer implementation of the traditional 'slabs' ray-box test [KK86] is more difficult, because it must handle very large or infinite values when the ray is parallel or nearly parallel to a box face. For comparision, the Plücker-AABB test requires only a simple changes to work with reduced precision integer arithmetic.

Several hardware ray tracing systems have been developed to date, but they use floating-point arithmetic. Two of the most promising systems are SaarCOR and the AR350 chip. SaarCOR [SWS02] is targeted at interactive applications and has been implemented as an FPGA prototype but

not (to date) in a custom VLSI chip. Advanced Rendering Technology's AR350 ray tracing chip [Hal01] is designed to accelerate off-line rendering for commercial 3-D software packages. An earlier version of the AR350 used 20-bit logarithmic arithmetic [Wri99], but a more recent design uses floating-point. Another system, TigerSHARK, is a flexible architecture based on an array of floating-point digital signal processors [HA96]. Ray tracing has also been implemented on a GPU [Pur04], however the implementation is somewhat awkward due to limitations in GPU programmability.

Other researchers have also investigated the use of reduced precision for 3-D rendering. The QSplat point rendering system [RL00] uses a hierarchy of low-precision bounding spheres to encode the coordinates of point samples. In another project, Hao et al. successfully reduced the precision required for a traditional scan-converting 3-D graphics pipeline [HV01]. Their approach resulted in subtle changes to the rendered images, however.

2. BVH Construction

There are two kinds of BVH, differing by the method of construction. BVHs constructed in a top-down fashion are much more effective than those constructed bottom-up [GS87]. In this paper, we use top-down constructed BVHs. We also only allow two child nodes for each BVH node, thus our BVH has some k-d tree-like properties.

The BVH is constructed using an algorithm similar to that described in [SM03]. The algorithm starts by computing the minimum bounding box that encloses all the triangles in the scene. (Our scenes contain only triangles, but the BVH could enclose other geometry types as well.) A splitting plane is chosen along one of the principal x, y, or z axes to divide the bounding box in half. These halves become the node's two children. The center points of the triangles are computed and compared with the splitting plane, partitioning the triangles into two sets (one for each child node.) This process repeats recursively with the two children until some termination criteria is met.

The hierarchy terminates when a node contains seven objects or less. The value seven was determined empirically to be the best tradeoff between hierarchy size and rendering time. A lower value improves rendering time slightly, at the expense of excessively large hierarchies. Additionally, a node will not be split along an axis if it is already too thin along that axis, instead the next axis will be chosen. If the node is too thin along all the *x*, *y*, and *z* axes, it is not split and becomes a leaf node. In this case, the node may contain more than seven objects.

Tree construction heuristics can be applied to BVHs to reduce rendering time. We have had success with the Surface Area Heuristic (SAH) [MB89], observing nearly a doubling of rendering performance. Unfortunately, preprocessing times became prohibitive so we did not use the SAH for this paper.

A typical BVH node has three parts:

1. The coordinates 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 negative to indicate that the node is a branch node. A value of -1 denotes a branch node that was split along the *x* axis, -2 for the *y* axis, and -3 for the *z* axis. This information is used to improve the BVH traversal order.
3. A pointer to the two children (if a branch node), or a pointer to a list of objects enclosed by the node (if a leaf node.)

For our 'reference' BVH implementation, we chose to represent the box coordinates with 32-bit floating point values. In this case, a BVH node consumes 32 bytes of memory. The six box coordinates consume 24 bytes, the number of objects field consumes 4 bytes, and the pointer consumes 4 bytes. For the integer BVH implementations, box coordinate precisions of 12, 16, 20, and 24 bits were tested. In these cases, each BVH node consumes 17, 20, 23, or 26 bytes. This results in BVH memory savings of 19% to 47%, versus using 32-bit floating-point values.

Floating-point arithmetic is still used during integer hierarchy construction. First, the scene is uniformly scaled to fit the $[(-1,-1,-1),(1,1,1)]$ cube. BVH construction proceeds like the floating-point case, except the BVH coordinates are scaled to the appropriate integer range and are rounded to the nearest integer. Thus, the coordinates have an error of $\pm\frac{1}{2}$. For example, when using a 12-bit integer coordinate system, the BVH coordinates are multiplied by 2047 to fit the integer range $[-2047, 2047]$ in *x*, *y*, and *z*. (The value of -2048 is not used because we want symmetry.) Additionally, nodes are not split along an axis if they are already smaller than the integer coordinate system grid in that direction.

3. Plücker-Coordinate Based BVH Traversal

The integer BVH traversal algorithm is based on the Plücker-coordinate based ray-axis aligned box overlap test [MW04]. Although this method was derived using Plücker coordinates, it does not require the ray or boxes to be converted into a Plücker coordinate form. Other methods for performing a ray-box test are the 'slabs' method [KK86] and by using the separating axis theorem (SAT) [GLGT99]. (The SAT test is actually a line segment-box test, but can be adapted to ray tracing by choosing appropriate line endpoints. However, this test is not any simpler than the Plücker method and is somewhat awkward to use for ray tracing because it is a line segment-box test, and cannot handle infinitely long rays. Additionally, the ray or line segment must be represented as

a line midpoint coordinate and a forward and backward distance vector, instead of the origin, direction, and length that is standard for ray tracing. The SAT-based test is only based on multiplication, addition, subtraction, and comparisions of finite quantities, so a similar analysis and integer implementation as presented in this paper for the Plücker method is possible.)

The Plücker technique tests the ray against the edges of the box to determine if the ray passes through the box. The algorithm is broken into eight cases, based on the signs of the ray's *i*, *j*, and *k* direction components. For example, the case where all three components are negative is called 'MMM' (for 'minus minus minus.') Splitting the algorithm into eight separate cases permits the testing of only six of the box edges instead of all twelve. The origin and endpoint (if not infinite) of the ray are also tested against the box, as the edge tests only determine if the ray stabs the box regardless of the origin and endpoint.

Before the Plücker algorithm can be described, a ray data structure must be defined. We use a ray with:

1. An origin with *x*, *y*, and *z* components.
2. A direction vector with *i*, *j*, and *k* components.
3. A flag indicating whether the ray is finite or infinitely long.
4. A length *t*, if the ray is not of infinite length. (Alternatively, a value of -1 can indicate an infinite length instead of the flag.)
5. An endpoint (ex, ey, ez), if the ray is not of infinite length.

In a software implementation of the Plücker algorithm, there are eight separate ray-BVH traversal functions, one for each case. In a hardware implementation, 2:1 multiplexer circuits can select the correct ray origin and direction values to feed to a single ray-box testing circuit based on the direction vector component signs. The directions of some of the comparisons (e.g. '>' or '<') also vary depending on the case, but these can be handled with a small amount of logic circuitry. The details of this are described in section 7.

Pseudocode for case 'MMM' is as follows:

```
procedure Traverse_MMM(node, ray)

// Part A: The box must be on the forward
// extension of the ray, else it misses
if ray.x < node.xmin OR
   ray.y < node.ymin OR
   ray.z < node.zmin
   return MISS

// Part B: If the ray is not infinite, check
// the ray's endpoint
if ray.infinite == false AND
   (ray.ex > node.xmax OR
    ray.ey > node.ymax OR
    ray.ez > node.zmax)
   return MISS

// Part C: Subtract the ray origin from the
// box coordinates
xa = node.xmin - ray.x
ya = node.ymin - ray.y
za = node.zmin - ray.z
xb = node.xmax - ray.x
yb = node.ymax - ray.y
zb = node.zmax - ray.z

// Part D: Test the ray against the six
// silhouette edges of the box
if ray.i * ya - ray.j * xb < 0 OR
   ray.j * xa - ray.i * yb < 0 OR
   ray.k * xa - ray.i * zb < 0 OR
   ray.i * za - ray.k * xb < 0 OR
   ray.j * za - ray.k * yb < 0 OR
   ray.k * ya - ray.j * zb < 0
   return MISS

// Part E: Test the children if a branch
// node or the triangles if a leaf node
if node.numtris < 0
// Branch node
   Traverse_MMM(node.child[1], ray)
   Traverse_MMM(node.child[0], ray)
else
// Leaf node
   TriIntersect(node, ray)
```

The other seven cases are similar, but variables are switched around. For part A, there are three comparisons, one for each coordinate axis. The type of each comparision is determined by the corresponding ray direction vector component sign. For example, if *ray.i* is negative, the comparison is $(ray.x < node.xmin)$. If *ray.i* is positive, the comparison is $(ray.x > node.xmax)$. For part B, the rule is reversed. If *ray.i* is negative, the comparison is $(ray.ex > node.xmax)$, and $(ray.ex < node.xmin)$ if *ray.i* is positive.

Part C is the same for all eight cases. For part D, examine the pattern of 'a' and 'b' variables (e.g. 'xa' and 'xb'). If *ray.i* is positive, change all the 'a' variables that are multiplied with *ray.i* to 'b', and all the 'b' variables to 'a'. For example, case PMM is:

```
if ray.i * yb - ray.j * xb < 0 OR
   ray.j * xa - ray.i * ya < 0 OR
   ray.k * xa - ray.i * za < 0 OR
   ray.i * zb - ray.k * xb < 0 OR
   ray.j * za - ray.k * yb < 0 OR
   ray.k * ya - ray.j * zb < 0
   return MISS
```

The rule is similar for *ray.j* and *ray.k*.

The pseudocode for part E incorporates the improved BVH traversal order described in [Mah04]. The best order to test a node's children is not simply child[0] and then child[1], but depends on the case (e.g. 'MMM') and the axis of the splitting plane that separated the children during BVH construction. (Cases 'MMM' and 'PPP' do not depend on the splitting plane axis.) This improves performance by up

to 243% for some scenes. Testing child[1] and then child[0] is the optimal order for case 'MMM', as shown in the pseudocode.

4. Reduced Precision BVH Traversal

Reduced precision integer BVH traversal is similar to the floating-point traversal, except integer arithmetic is used for the calculations and comparisons, and the ray parameters and box coordinates are represented with integer values.

Two error conditions may occur when performing ray-box intersection tests using reduced precision integer arithmetic. The first is a *false hit*, where the low-precision computation indicates the box is hit when it would actually be missed if computed with full precision. A false hit does not produce an incorrect image, but it does result in unnecessary traversal of the BVH. The second error condition is a *false miss*, where the integer computation indicates a miss when the result should be a hit. This can produce incorrect images because geometry contained within the hierarchy that should be tested for ray intersection is ignored. Hence, the goal is to make false misses impossible and to minimize the number of false hits. As the precision is lowered, the number of false hits will increase, reducing the effectiveness of the BVH.

The 16-bit precision case will be explained here, but the derivation can be extended to other amounts of precision. The 16-bit integer estimates are denoted with a '16' subscript. For this case, the ray origin (x,y,z), ray direction (i,j,k), ray endpoint (ex,ey,ez), and node box dimensions $(xmin,ymin,zmin,xmax,ymax,zmax)$ have been scaled and rounded to 16-bit integer values with range $[-32767,32767]$. The ray direction components have been scaled and rounded such that the largest component is ± 32767. Note that this is not the same as normalizing the direction vector.

Examine the first parts of the box intersection code (for the 'MMM' ray case):

```
// Part A: The box must be on the forward
// extension of the ray, else it misses
if ray.x < node.xmin OR
   ray.y < node.ymin OR
   ray.z < node.zmin
   return MISS

// Part B: If the ray is not infinite, check
// the ray's endpoint
if ray.infinite == false AND
   (ray.ex > node.xmax OR
    ray.ey > node.ymax OR
    ray.ez > node.zmax)
   return MISS
```

Converting this portion to use 16-bit integer arithmetic is straightforward. Simply substitute the integer values for the floating-point ones. Examining the comparisons reveals that this cannot produce false misses, but may produce false hits. This happens because values that differ when represented with full precision may become equal when rounded to integers. For example, if $ray.x = 34.6$ and $node.xmin = 34.7$, then according to the comparison above $(ray.x < node.xmin)$, a miss occurs. When rounded to integers, $ray.x_{16} = 35$ and $node.xmin_{16} = 35$. The integer comparison $(ray.x_{16} < node.xmin_{16})$ now fails to reject the ray and a false hit occurs.

For a hit to occur, $(ray.x >= node.xmin)$. It is impossible for an actual hit to become an integer false miss because at worst, *ray.x* and *node.xmin* will become equal when rounded to integer. Also, if their actual values are equal, their integer values will also be equal. The same reasoning applies for the other five comparisions.

The rest of the ray-box test involves computation, so interval arithmetic will be used to derive error bounds on the results obtained from the integer approximations. These error bounds will be used in the comparisons for the six ray-edge tests. Values that include error information as intervals will be denoted with an uppercase letter, while scalars will be lowercase. It is important to note that an actual implementation using this idea would not use interval arithmetic. Rather, just the scalar values are computed and compared to the error bound derived from this interval arithmetic proof.

The next stage (Part C) of the ray-box test subtracts the ray origin from the box coordinates. Converting to integer and eliminating 'node.' and 'ray.' for brevity:

$$\begin{aligned} xa_{17} &= xmin_{16} - x_{16} \\ ya_{17} &= ymin_{16} - y_{16} \\ za_{17} &= zmin_{16} - z_{16} \\ xb_{17} &= xmax_{16} - x_{16} \\ yb_{17} &= ymax_{16} - y_{16} \\ zb_{17} &= zmax_{16} - z_{16} \end{aligned}$$

Note that the result of the subtraction is 17 bits, not 16. This is to prevent overflow of the result, because subtraction of two values with range $[-32767,32767]$ produces a result with range $[-65534,65534]$.

Using interval arithmetic, error bounds are obtained for the subtraction result. Taking the first subtraction as an example, the terms can be re-written as intervals. The error in the integer terms is $\pm\frac{1}{2}$, because the integer values are a rounded approximation of the full-precision value.

$$\begin{aligned} xa_{17} &= xmin_{16} - x_{16} \\ XA_{17} &= [xmin_{16} - \frac{1}{2}, xmin_{16} + \frac{1}{2}] - [x_{16} - \frac{1}{2}, x_{16} + \frac{1}{2}] \end{aligned}$$

The rule for interval subtraction is:

$$[a,b] - [c,d] = [a-d, b-c]$$

Thus:

$$XA_{17} = [xmin_{16} - x_{16} - 1, xmin_{16} - x_{16} + 1]$$

$$XA_{17} = [xa_{17}-1, xa_{17}+1]$$

Hence, all the scalar subtraction results have a range of $[-65534, 65534]$ and an error of ± 1. Remember that in an actual implementation, only the scalar xa_{17} is computed, and not the interval XA_{17}.

Next, the ray is tested against the six edges of the box (Part D):

```
if i * ya - j * xb < 0 OR
   j * xa - i * yb < 0 OR
   k * xa - i * zb < 0 OR
   i * za - k * xb < 0 OR
   j * za - k * yb < 0 OR
   k * ya - j * zb < 0
   return MISS
```

The first expression $i*ya - j*xb < 0$ will be analyzed. This expression gives the relative orientation of the ray with respect to the edge. If the result is negative, the ray is on the 'outside' of the edge and misses the box. Re-written in integer form, it becomes:

$$i_{16}*ya_{17} - j_{16}*xb_{17} < ERRBOUND$$

$ERRBOUND$ is chosen to ensure that no false misses can occur. $ERRBOUND$ is the same for all six edge tests. Note that $ERRBOUND$ is a constant. It is possible to use true interval arithmetic for the calculations, but that would add too much overhead. A static error bound value is not as 'tight' as one computed based on the actual values of the parameters, but is trivial to implement.

To simplify the analysis, the expression $i_{16}*ya_{17} - j_{16}*xb_{17} < ERRBOUND$ is first split into terms p and q, where:

$$\begin{aligned} p_{32} &= i_{16}*ya_{17} \\ q_{32} &= j_{16}*xb_{17} \\ p_{32} - q_{32} &< ERRBOUND \end{aligned}$$

p and q are 32 bits in length, as multiplying two signed values of length m and n bits gives a result with length $m+n-1$ bits.

Again, interval arithmetic is used to determine the error characteristics. Compute the interval P32 based on p_{32}, recalling that the error in the ray's i, j, and k components is $\pm\frac{1}{2}$, and the error in ya_{17} is ± 1:

$$P_{32} = [i_{16} - \frac{1}{2}, i_{16} + \frac{1}{2}] * [ya_{17}-1, ya_{17}+1]$$

The rule for interval multiplication is:

$$\begin{aligned} [a,b]*[c,d] = [&\min(a*c, a*d, b*c, b*d), \\ &\max(a*c, a*d, b*c, b*d)] \end{aligned}$$

Expanding:

$$\begin{aligned} P_{32} = [&\min(i_{16}*ya_{17} - i_{16} - \frac{ya_{17}}{2} + \frac{1}{2}, \\ &i_{16}*ya_{17} + i_{16} - \frac{ya_{17}}{2} - \frac{1}{2}, \\ &i_{16}*ya_{17} - i_{16} + \frac{ya_{17}}{2} - \frac{1}{2}, \\ &i_{16}*ya_{17} + i_{16} + \frac{ya_{17}}{2} + \frac{1}{2}), \\ &\max(i_{16}*ya_{17} - i_{16} - \frac{ya_{17}}{2} + \frac{1}{2}, \\ &i_{16}*ya_{17} + i_{16} - \frac{ya_{17}}{2} - \frac{1}{2}, \\ &i_{16}*ya_{17} - i_{16} + \frac{ya_{17}}{2} - \frac{1}{2}, \\ &i_{16}*ya_{17} + i_{16} + \frac{ya_{17}}{2} + \frac{1}{2})] \end{aligned}$$

It is impractical to compute the eight min and max subexpressions because the overhead of the additional computations would be too great. However, a maximum theoretical error bound can be obtained on the multiplication result. Again, this may not be as 'tight' as the actual interval result obtained from fully evaluating the eight min and max subexpressions.

Examining the interval P_{32} reveals that $i_{16}*ya_{17}$ is common to all eight min and max subexpressions. The other portion of the subexpressions can be considered error terms. We want to compute a maximum possible value for the error, which has the form:

$$err = \pm i_{16} \pm \frac{ya_{17}}{2} \pm \frac{1}{2}$$

The maximum possible value for err is obtained by substituting the maximum possible values for i_{16} (32767) and ya_{17} (65534). Hence,

$$\begin{aligned} err &= 32767 + \frac{65534}{2} + \frac{1}{2} \\ &= 65534.5 \end{aligned}$$

Using this simplification, P_{32} can be re-written as:

$$\begin{aligned} P_{32} &= [i_{16}*ya_{17} - err, i_{16}*ya_{17} + err] \\ &= [i_{16}*ya_{17} - 65534.5, i_{16}*ya_{17} + 65534.5] \end{aligned}$$

Recall that the scalar p_{32} equals $i_{16}*ya_{17}$:

$$P_{32} = [p_{32} - 65534.5, p_{32} + 65534.5]$$

Q_{32} is of the same form and has the same error bounds as P_{32}, so its derivation is not shown. Hence, the scalars p_{32} and q_{32} both have error bounds of ± 65534.5.

Finally, p_{32} and q_{32} are subtracted and compared to the constant $ERRBOUND$, which is yet to be determined. The result of the subtraction is 33 bits to prevent overflow from subtracting two 32-bit values. The error in the subtraction is determined by subtracting the intervals P_{32} and Q_{32}:

$$P_{32} - Q_{32} < ERRBOUND$$

$$[p_{32}-65534.5, p_{32}+65534.5] \; - \\ [q_{32}-65534.5, q_{32}+65534.5] \; < \; ERRBOUND$$

$$[p_{32}-q_{32}-131069, p_{32}-q_{32}+131069] < ERRBOUND$$

Thus, the result of the subtraction $p_{32}-q_{32}$ has an error of ± 131069.

Recall that originally, the result's comparison was with zero and not $ERRBOUND$. A negative result meant that the ray was oriented 'outside' that particular edge and the ray missed the box. A zero or positive result meant that the ray intersected the edge or was oriented 'inside' the edge. If all six edge comparisons were zero or positive, the box was hit by the ray. Since there is an error of ± 131069 on the integer result, any values between -131069 and $+131069$ represent an uncertain situation where it is unclear whether the ray misses or not. We wish to convert these uncertain results to false hits to conservatively avoid missing boxes that should have been hit. Hence, any result greater than or equal to -131069 is now considered a hit, and any result less than -131069 is guaranteed to be a true miss. Thus:

$$ERRBOUND = -131069$$

Substituting for p_{32} and q_{32}, the final expression for the first integer ray-box edge test becomes:

$$i_{16} * ya_{17} - j_{16} * xb_{17} < -131069$$

The other five edge tests have a similar form, and use the same $ERRBOUND$ value.

In general, the $ERRBOUND$ for a given initial wordlength of n bits for the ray and box parameters is:

$$ERRBOUND = -(2^{n+1}-3)$$

The derivation in this section used $n = 16$ bits.

5. Simplification by Truncation of Multiplication Results

The technique can be improved by realizing that the 32-bit multiplication results p_{32} and q_{32} can have their 16 lower bits truncated without adverse effects. This simplifies the subtraction of p and q, and the comparison with $ERRBOUND$ since the wordlengths are halved. Removing the lower 16 bits is equivalent to dividing by 65536 and taking the floor() of the result. Thus, p_{32} and q_{32} become p_{16} and q_{16}.

The error of ± 65534.5 on p_{32} and q_{32} must also be adjusted. One may be tempted to simply divide the error by 65536, giving an error of ± 0.999977. This is only partially correct. Some information is lost when the bottom 16 bits of p_{32} and q_{32} are dropped. This may shift the value of p_{16} and q_{16} downwards relative to their 32-bit values by up to $\frac{65535}{65536}ths$. The upper error bound of $+0.999977$ is still correct, but the lower error bound must have $\frac{65535}{65536}$ subtracted from it to compensate for the possible downward shift. The lower error bound becomes $-0.999977 - \frac{65535}{65536} = -1.999962$.

$ERRBOUND$ must be recomputed based on the new error bounds for p_{16} and q_{16}. Using $hiword()$ to denote the trimming of the lower 16 result bits:

$$p_{16} = hiword(p_{32}) \\ q_{16} = hiword(q_{32})$$

$$p_{16} - q_{16} < ERRBOUND$$

Applying interval arithmetic to determine the error:

$$P_{16} = [p_{16}-1.999962, p_{16}+0.999977] \\ Q_{16} = [q_{16}-1.999962, q_{16}+0.999977]$$

$$P_{16} - Q_{16} < ERRBOUND$$

$$[p_{16}-1.999962, p_{16}+0.999977] \; - \\ [q_{16}-1.999962, q_{16}+0.999977] \; < \; ERRBOUND$$

Using the interval subtraction rule:

$$[p_{16}-q_{16}-2.999939, p_{16}-q_{16}+2.999939] < ERRBOUND$$

Hence, $ERRBOUND = -2.999939$ if expressed as a decimal value. Substituting for p_{16}, q_{16}, and $hiword()$ gives:

$$hiword(i_{16} * ya_{17}) - hiword(j_{16} * xb_{17}) < -2.999939$$

However, since the comparison with $ERRBOUND$ is an integer one, this can be rewritten as:

$$hiword(i_{16} * ya_{17}) - hiword(j_{16} * xb_{17}) < -2$$

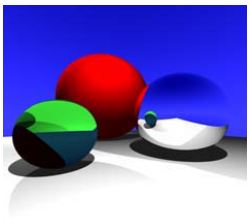
This works because the next lowest integer value is -3, which is both less than -2.999939 and less than -2. An integer value of -2 is both not less than -2.999939 and not less than -2, so the correct behavior is preserved.

Thus, $ERRBOUND$ is -2. In fact, $ERRBOUND$ is always -2 if the lower n bits of p and q are truncated, given an initial wordlength of n bits for the ray and box parameters. It is also possible to truncate fewer than n bits provided $ERRBOUND$ is adjusted accordingly.

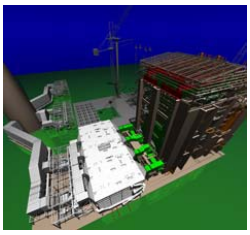
6. Test Results

A variety of test scenes were used to evaluate the integer technique (Figure 1). The Bunny [Vara], Branch, Poppy [Mac04], Power plant [Varb], and Lucy [Vara] models were obtained from other researchers or from the Internet.

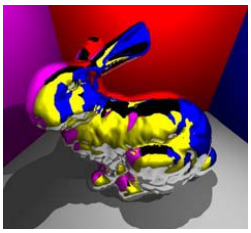
Images were rendered at 4096x4096 pixel resolution and compared pixel-for-pixel with reference images to verify



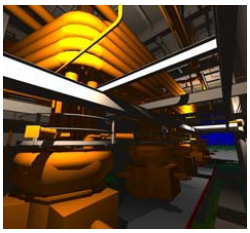
Spheres
Triangle mesh spheres - solid, reflective, and refractive
Triangles: 24578 Lights: 2
BVH Nodes: 9997
Primary rays: 4194304
Secondary rays: 1220088
Shadow rays: 4658758



Powerplant - Front
Front view of the UNC powerplant model
Triangles: 12748512 Lights: 1
BVH Nodes: 5604263
Primary rays: 4194304
Secondary Rays: 1112742
Shadow rays: 3257733



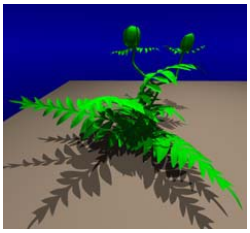
Bunny
Psychedelic reflective Stanford bunny in a colored box
Triangles: 69463 Lights: 2
BVH Nodes: 29231
Primary rays: 4194304
Secondary rays: 1565756
Shadow rays: 8380660



Powerplant - North
North/bottom view of the UNC powerplant model
Triangles: 12748512 Lights: 3
BVH Nodes: 5604263
Primary rays: 4194304
Secondary Rays: 57961
Shadow rays: 12593457

Branch
Branch model from the U of Calgary Graphics Jungle
Triangles: 312706 Lights: 2
BVH Nodes: 135005
Primary rays: 4194304
Shadow rays: 6057826

Powerplant - Boiler
Inside the boiler of the UNC powerplant model
Triangles: 12748512 Lights: 1
BVH Nodes: 5604263
Primary rays: 4194304
Shadow rays: 4194304



Poppy
Poppy model from the U of Calgary Graphics Jungle
Triangles: 395460 Lights: 2
BVH Nodes: 164903
Primary rays: 4194304
Shadow rays: 5527312

Lucy
Angelic statue
Triangles: 28057792 Lights: 2
BVH Nodes: 11702357
Primary rays: 4194304
Secondary rays: 1556156
Shadow rays: 11500918

Figure 1: *Test scenes.*

correctness. Reference images were rendered with 32-bit floating-point arithmetic.

Table 1 presents the test statistics. The first data column presents the statistics for the reference implementation using floating-point. The next eight columns give statistics for the integer method. These are split into four precision categories: 24 bits, 20 bits, 16 bits, and 12 bits. Each precision category is split into two cases. The first case (labeled n/2n bits) does not truncate the multiplication results p and q. For example, the 24-bit integer case produces p and q values that are 48 bits in length. The second case (labeled n/n bits) truncates the p and q values back to n bits.

Several statistics were measured for each test scene and precision:

- *Ray-Tri Tests (Pri/Sec/Sha)*: The number of ray-triangle tests performed for primary, secondary, and shadow rays.
- *Ray-Node Tests (Pri/Sec/Sha)*: The number of ray-node tests performed for primary, secondary, and shadow rays.

Statistics for the integer cases are expressed as percentages relative to the reference case. Percentages greater than 100% indicate that the integer algorithm is testing more nodes or triangles than the reference case. This increase (or overhead) is due to false hits in the ray-box test, causing additional BVH traversal and additional ray-triangle tests. As the precision is reduced, the percentages will increase as the number of false hits increases. The authors consider an acceptable increase to be in the neighborhood of a few percent.

Timings are not provided because the integer results are a simulation, and are not tuned for speed. The integer times were slower than the reference due to the amount of additional bit manipulation needed to accomodate the variable precision settings. In particular, truncating the p and q terms required an expensive shift operation. This shift would not be required in hardware, as the lower bits of p and q would simply not be connected to anything (or computed at all.)

The results show that for simple to moderately complex scenes, 16-bit integer arithmetic is sufficient. The Spheres,

Spheres	Reference	20/40bits	20/20bits	16/32bits	16/16bits	12/24bits	12/12bits
Ray-Tri Tests (Pri)	26,452k	100%	100%	101%	102%	124%	129%
Ray-Tri Tests (Sec)	18,341k	100%	100%	102%	102%	126%	133%
Ray-Tri Tests (Sha)	49,349k	100%	100%	101%	101%	119%	124%
Ray-Node Tests (Pri)	160,697k	100%	100%	100%	100%	104%	105%
Ray-Node Tests (Sec)	78,983k	100%	100%	100%	100%	106%	107%
Ray-Node Tests (Sha)	206,994k	100%	100%	100%	100%	104%	105%
Bunny							
Ray-Tri Tests (Pri)	67,780k	100%	100%	101%	101%	113%	117%
Ray-Tri Tests (Sec)	34,002k	100%	100%	101%	102%	120%	124%
Ray-Tri Tests (Sha)	85,170k	100%	100%	101%	101%	109%	112%
Ray-Node Tests (Pri)	245,978k	100%	100%	100%	100%	104%	105%
Ray-Node Tests (Sec)	116,331k	100%	100%	100%	100%	105%	107%
Ray-Node Tests (Sha)	257,453k	100%	100%	100%	100%	104%	105%
Branch							
Ray-Tri Tests (Pri)	20,068k	101%	101%	119%	124%	802%	1103%
Ray-Tri Tests (Sha)	31,701k	101%	101%	117%	121%	605%	814%
Ray-Node Tests (Pri)	225,773k	100%	100%	102%	102%	145%	161%
Ray-Node Tests (Sha)	210,371k	100%	100%	102%	103%	153%	172%
Poppy							
Ray-Tri Tests (Pri)	12,127k	101%	101%	110%	112%	380%	491%
Ray-Tri Tests (Sha)	28,424k	101%	101%	110%	112%	304%	374%
Ray-Node Tests (Pri)	96,549k	100%	100%	101%	102%	130%	139%
Ray-Node Tests (Sha)	159,347k	100%	100%	102%	102%	128%	137%
Power - Front							
Ray-Tri Tests (Pri)	76,331k	101%	101%	115%	118%	509%	655%
Ray-Tri Tests (Sec)	8,492k	101%	101%	109%	111%	335%	433%
Ray-Tri Tests (Sha)	47,259k	101%	101%	115%	118%	415%	529%
Ray-Node Tests (Pri)	456,926k	100%	100%	103%	103%	149%	164%
Ray-Node Tests (Sec)	47,322k	100%	100%	102%	103%	139%	154%
Ray-Node Tests (Sha)	304,137k	100%	100%	103%	103%	141%	153%
Power - North							
Ray-Tri Tests (Pri)	133,066k	101%	102%	121%	127%	706%	942%
Ray-Tri Tests (Sec)	1,179k	101%	102%	124%	131%	532%	698%
Ray-Tri Tests (Sha)	237,767k	101%	102%	123%	128%	665%	855%
Ray-Node Tests (Pri)	1,499,541k	100%	100%	102%	103%	145%	158%
Ray-Node Tests (Sec)	10,267k	100%	100%	102%	103%	132%	142%
Ray-Node Tests (Sha)	2,766,692k	100%	100%	102%	103%	136%	145%
Power - Boiler							
Ray-Tri Tests (Pri)	233,896k	103%	104%	163%	180%	3222%	4897%
Ray-Tri Tests (Sha)	159,376k	103%	104%	157%	172%	1963%	2607%
Ray-Node Tests (Pri)	1,655,241k	100%	100%	106%	108%	240%	301%
Ray-Node Tests (Sha)	1,418,359k	100%	100%	104%	106%	179%	201%
Lucy							
Ray-Tri Tests (Pri)	49,486k	102%	102%	136%	146%	3501%	4992%
Ray-Tri Tests (Sec)	28,229k	100%	100%	101%	101%	158%	190%
Ray-Tri Tests (Sha)	182,481k	102%	103%	139%	151%	2286%	3280%
Ray-Node Tests (Pri)	259,503k	100%	100%	106%	107%	423%	551%
Ray-Node Tests (Sec)	74,198k	100%	100%	100%	100%	112%	117%
Ray-Node Tests (Sha)	735,050k	101%	101%	108%	110%	367%	477%

Table 1: *Test statistics. 24-bit results are omitted because they are all 100%.*

Precision	*Bytes / node*	*Nodes*	*BVH size (bytes)*
32-bit FP	32	11,702,427	374,477,664
24-bit int	26	11,702,427	304,263,102
20-bit int	23	11,702,427	269,155,821
16-bit int	20	11,702,005	234,040,100
12-bit int	17	11,549,639	196,343,863

Table 2: *BVH size for Lucy model for varying precisions*

Bunny, Poppy, and Branch scenes show an increase of only a few percent in the number of BVH nodes tested. The Branch scene shows an increase of 24% in the number of triangles tested, but note that the number of triangles tested is much smaller than the number of nodes tested. Hence, the 24% increase may not be as damaging to performance as it appears.

20-bit arithmetic is needed for the more complex scenes. In the power plant and Lucy scenes, an acceptable overhead of less than 4% is observed using 20 bits of precision. Using 24 bits results in less than 0.5% overhead, and is more than sufficient for the scenes tested.

12 bits of precision did not perform well, resulting in significant increases in ray-node and ray-triangle tests for even the simplest scenes. One interesting result is that the number of hierarchy nodes was reduced by more than half for the power plant scenes, saving a large amount of memory. (The other scenes did not show a significant reduction in the number of BVH nodes, thus no statistics are presented.) Recall that hierarchy nodes are not generated if they are smaller than the integer coordinate system can properly represent. The boxes were too small to represent with the 12-bit coordinate grid, causing early termination of the hierarchy. Unfortunately, the BVH was so inefficient due to false hits that this was of little value.

Truncation of the p and q values does not greatly affect the results, except at very low precisions. For higher precisions, keeping all p and q bits seems to be unnecessary. One compromise is to keep only a few additional p and q bits. For example, the 16-bit case produces p and q values of length 32. These could be truncated to 20 bits instead of 16. Additional testing showed that keeping four additional bits was nearly as effective as keeping all of the additional bits.

There was also no significant difference between the relative statistics for primary, secondary, and shadow rays.

Significant memory savings is also achieved while still producing images identical to those produced when using 32-bit floating-point. Table 2 details the memory footprint of the Lucy scene's BVH for varying precisions.

7. Theoretical Hardware Implementation

The purpose of this section is provide: a synopsis of VLSI circuit design issues, describe a basic design for a Plücker coordinate based ray-AABB intersection pipeline, and use this pipeline to quantify the expected benefit from using reduced precision integer arithmetic.

When looking at the history of complex high-performance chips such as microprocessors, it is clear that 35% of the Moore's Law scaling trajectory has been the result of better architectures while the remaining 65% has been the result of improved integrated circuit technology. As transistor sizes have shrunk, design options have been primarily constrained by the available die area and more recently by power considerations.

As a dedicated VLSI accelerator, ray tracing has one significant advantage: the abundant opportunity to exploit inter-ray parallelism. By reducing circuit size, more rays can be processed at once, increasing performance greatly. Given the "trivially parallel" nature of ray tracing, an appropriate metric to describe the aggregate benefit of a particular design is the *area-delay product*. Reducing area by a factor of n will allow n parallel units to be constructed that can improve performance by n. Reducing a particular unit's delay by m will also increase the performance by m.

By trading large 32-bit floating point circuits for smaller lower-precision integer units, more rays can be processed within a particular area. In addition, the need for pre- and/or post- normalization duties in floating point devices causes increased operational latency and typically longer pipelines. 32-bit IEEE-754 floating point is only slightly worse than 24-bit integers when it comes to power. The mantissa circuits are the dominant power consumers in floating point units. The represented mantissa is only 23 bits but the standard employs a hidden bit and most implementations employ extended internal precision to compensate for rounding issues.

A definitive view of the size, speed, and power consumption of arithmetic circuits is difficult to obtain since there are so many variables in algorithm, design method, circuit approach, and target operating conditions. For example, reducing the delay between pipeline stages yields a higher frequency of operation but at a significant area penalty. Equivalent arithmetic circuits in an Intel XScale running at a peak frequency of 400 MHz are an order of magnitude smaller than those in a Pentium 4 running at a peak frequency of 3.8 GHz [Fle05]. The addition of scan chain latches to enhance testability adversely affects both area and frequency. Other choices such as a particular Booth encoding for multipliers, or the choice between a wide variety of adders (carry-save, kogge-stone, brent-kung, etc.) has a significant effect on both area and delay [WH04]. Similarly the choice to pursue static vs. dynamic circuit strategies has a similar effect.

Another major influence comes from the basic design

flow. Modern VLSI CAD tools are capable of synthesizing a wide variety of integrated circuits from a program-like description using a language such as Verilog or VHDL. Synthesis CAD tools map the program onto a set of standard cells which perform relatively primitive functions such as and-or-invert with a variety of fan-in and drive strength options. While standard cells are themselves designed at the transistor level by companies such as Artisan, the cells are designed to a common form factor that simplifies the synthesis process. Full custom designs allow the designer to individually lay out individual transistors and wires for arbitrarily complex circuits. This fine grain control of the circuit results in significantly smaller and faster circuits, but requires significantly more expertise and design time. Most modern chips are hybrid designs which employ both synthesis and full-custom blocks. As synthesis capabilities improve, there is an increased reliance on the synthesis design flow. Both the cost and the benefit of full custom designs increases with design complexity. A full custom design for the Plücker based ray-AABB intersection unit described here will likely improve on a synthesized design by more than an order of magnitude in the value of the area-delay product. However the relative benefit of the reduced precision integer vs. the floating point approach is likely to be similar when the same design flow is used.

In order to achieve an understanding of the potential benefits of the reduced precision Plücker-coordinate integer approach, we have designed a simple ray-AABB intersection pipeline and have synthesized a variety of arithmetic circuits. A high level view of the pipeline is shown in Figure 2.

There are eight possible cases for a particular ray based on the sign bits of the *ri*, *rj*, and *rk* direction components. Recall that a software implementation requires eight seperate blocks of code to handle the cases. A hardware design can use a single circuit with multiplexors (muxes) to route the appropriate values to the circuit components based on the case. The muxes add some complexity to the design, but their contribution to the total circuit area is small.

The Plücker pipeline consists of 2 basic phases. The first phase determines whether or not the AABB exists on the forward extension of the ray and is within the ray's length. If the first phase indicates that a hit is possible, then the second phase determines whether the ray intersects (stabs) the AABB. Phase 1 and phase 2 could also be pipelined but in order to balance this pipeline, it would be necessary to analyze both the memory and register fill aspects of the design. The analysis presented here assumes that the two phases operate sequentially.

Details of the first pipeline phase are shown in Figure 3. In this figure: *2:1* is a word-wide 2-to-1 multiplexor; *b:2:1* is a single bit 2-to-1 multiplexor; and *"< >"* is a word-wide magnitude comparator capable of determining less-than, equal-to or greater-than. The input values to this stage come from latches containing the AABB coordinates, the ray origin, ray direction, ray endpoint, and a flag indicating whether the ray is infinitely long or finite.

If the first phase is successful, indicated by a HIT, then the second phase executes. A high level schematic of the second phase is shown in Figure 4. In this phase, ray and node *x*, *y*, and *z* values are subtracted to create six values (labeled *xa*, *ya*, *za*, *xb*, *yb*, and *zb*) which feed 3 similar circuits (labeled *ri:rj*, *rk:ri*, and *rj:rk*). Each of these three circuits tests two of the edges. The sign bits of the ray direction components (ri_{sign}, rj_{sign}, rk_{sign}) provide the remaining circuit inputs. A HIT signal from all three circuits indicates that the ray intersects the AABB.

The AND/OR gates are shown to make the schematics correct but their contribution is negligible in terms of both area and delay. Integer circuits were synthesized for 16, 20, and 24-bit precision. The detailed data for these circuits is shown in Table 3. The circuits were synthesized using commercial EDA tools (Synopsis Module- and Design-Compiler) and the standard cell library supplied by Artisan for a 250nm bulk CMOS TSMC process. The process has 5 layers of metal interconnect. In particular, the Artisan library contains 441 cells and multiple drive strengths are available for each logic function. The back-end place and route chores were performed using the Cadence Silicon Ensemble tool.

The add/subtract circuit is a fast carry look-ahead design. The multiplier is a radix 8 Booth encoded design. Since the CAD tools do a particularly sub-optimal job of generating the non-arithmetic circuits, the data presented here is based on custom commercial designs [Fle05]. The full custom data was obtained from a different integrated circuit process, and the actual area and delay numbers are proprietary. Therefore the data has been normalized to the corresponding 32-bit integer adder and the area and delay are calculated accordingly.

Both phases of the pipeline require 20 latches, 12 multipliers, 12 subtractors, 12 magnitude comparitors, 18 word-width multiplexors, and 6 single-bit multiplexors. 24-bit integer units were chosen for the comparison because 24 bits are more than sufficient the all the scenes tested, and provides a 'worst case scenario' for circuit area. We assume that the multiplication results are truncated back to 24 bits in length, as in Section 5. One minor detail is that the subtraction results are actually 25 bits in length, and not 24 bits as assumed for our estimates. This is not really important because our estimates are approximate.

The data in the table shows some anomalies that are an artifact of the synthesis tools. In particular the 24 multiply is better than the 20 bit multiply and the same as the 16 bit multiply. The difference is small but this anomaly would not be present in a custom design. This data does not account for wire complexity and wire-delay issues. Since the wiring complexity of these circuits in a 5-metal process is low, these issues should not alter the conclusion.

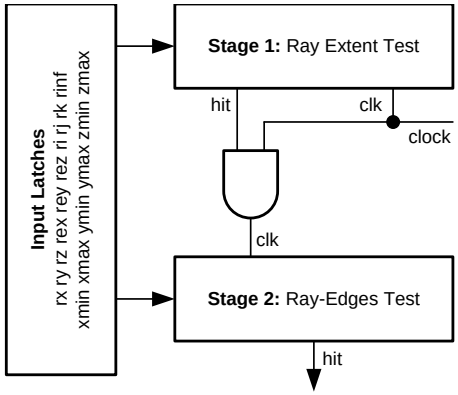


Figure 2: *Overview of ray-AABB circuit.*

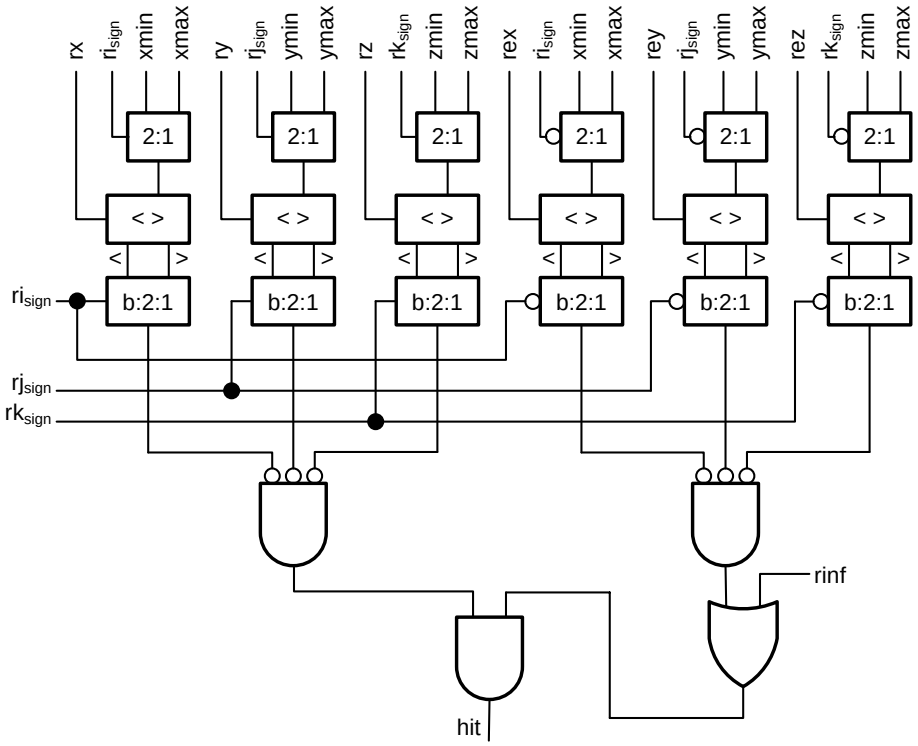


Figure 3: *Ray-AABB extent test circuit.*

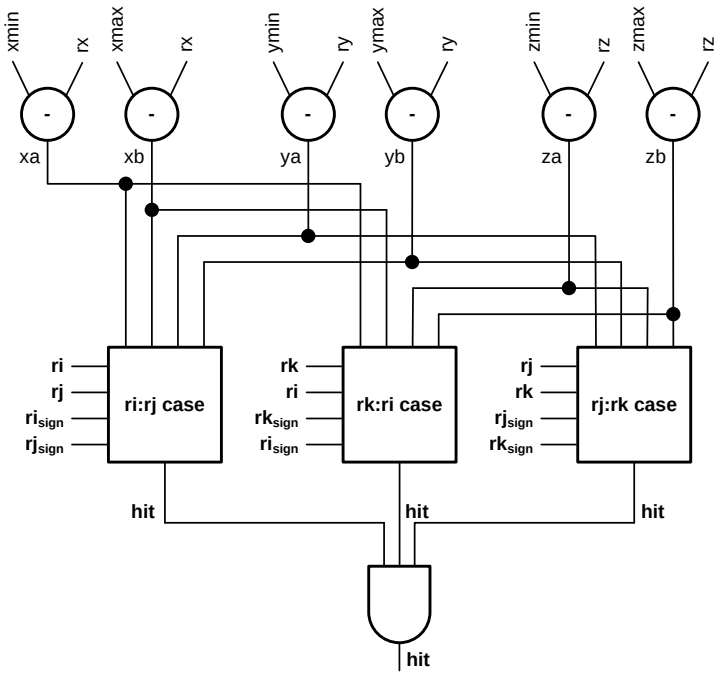


Figure 4: *Ray-AABB edges test circuit.*

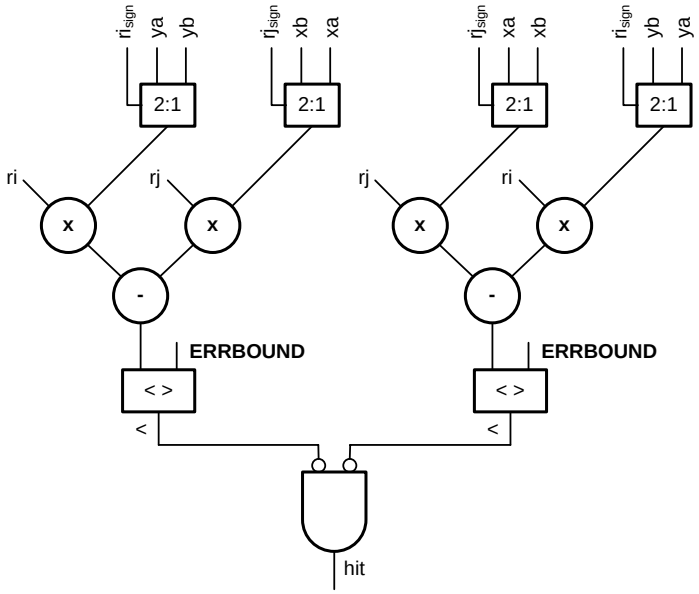


Figure 5: *ri:rj ray-AABB edges test circuit.*

Function	*Internal pipe stages*	*Area* (mm^2)	*Delay* (ns)	*Area* $*$ *Delay* $(mm^2 * ns)$
Integer Subtract (16-bit)	0	0.007964	1.1	0.008760
Integer Subtract (20-bit)	0	0.011005	1.1	0.012106
Integer Subtract (24-bit)	0	0.013677	1.2	0.016412
Integer Multiply (16-bit)	1	0.026342	2.5	0.065855
Integer Multiply (20-bit)	1	0.037390	2.8	0.104692
Integer Multiply (24-bit)	1	0.026342	2.5	0.065855
FP Subtract (32-bit)	6	0.105673	2.7	0.285317
FP Multiply (32-bit)	0	0.146793	6.2	0.910117
FP Multiply (32-bit)	4	0.174797	2.3	0.402033
Integer Magnitude Compare (24-bit)	0	0.005439	0.47736	0.002596
FP Magnitude Compare (32-bit)	0	0.013083	0.47736	0.006245
2:1 Multiplexor (1-bit)	0	0.000030	0.17646	0.000005
2:1 Multiplexor (24-bit)	0	0.000519	0.17646	0.000092
2:1 Multiplexor (32-bit)	0	0.001333	0.17646	0.000235
Scan Latch (24-bit)	0	0.001902	0.17647	0.000336
Scan Latch (32-bit)	0	0.002673	0.17647	0.000472

Table 3: *Circuit area and delay*

Component	*Number*	24 *bit Int Area* (mm^2)	32 *bit FP Area* (mm^2)
Latch	20	0.05346	0.03803
Multiply	12	2.09756	0.31610
Subtract	12	1.26808	0.16412
Compare	12	0.15699	0.06527
Mux	18	0.02399	0.00935
1-bit mux	6	0.00018	0.00018
Total area:		3.60026 mm^2	0.59305 mm^2
Pipe stage delay:		2.7 ns	2.5 ns
Area*delay:		9.72071 $mm^2 * ns$	1.48263 $mm^2 * ns$

Table 4: *24-bit Int and 32-bit FP ray-BVH area and delay*

Performance estimates for the integer and FP ray-AABB designs are presented in Table 4. The data shows that the performance of a 24-bit integer approach is more than 6 times better than conventional 32-bit floating point in both area and area-delay product. Furthermore the larger phase 2 pipeline contains 16 stages for the floating point circuit and only 8 stages for the integer circuit. Since the phases in this design are sequential this provides an additional factor of 2 benefit. However the throughput benefit will be dependent on the percentage of rays that actually hit within the first stage and require computation from the second.

8. Conclusions and Future Work

Our integer-based ray-BVH traversal technique permits the use of lower precision integer arithmetic for ray-BVH traversal instead of floating-point. This is most useful for hardware designers who are implementing ray tracing, as the complexity of floating point arithmetic can be avoided. Error bounds on the ray-box overlap test ensure that correct images are produced regardless of the integer precision used. Memory savings is also achieved because a BVH node's bounding box coordinates can be represented with lower precision.

The test results show that the precision may be reduced to as low as 16 bits for simple to moderate scenes. For more complex scenes, 20 bits of integer precision are needed to ensure the BVH works efficiently. The penalty for using too low a precision for a complex scene is unnecessary BVH traversal and onject intersection tests, reducing the effectiveness of the BVH. When using 20 bits of precision, the additional traversal and intersection tests amounted to only a few percent for the most complex scenes. Using 24 bits reduced

any overhead to less than 0.5%, and was more than adequate for the scenes we tested.

Our theoretical hardware analysis shows that using 24-bit integer arithmetic circuits instead of 32-bit floating-point gives a six-fold savings in circuit area. This can translate into a large speed-up, because six times as many ray-BVH intersection circuits can exist on a chip, thus six times as many rays can be processed at once.

9. Acknowledgements

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (iCORE). Thanks to Mr. Peter MacMurchy and Mrs. Julia Taylor-Hell for the Poppy and Branch models.

References

[CW88] CLEARY J. G., WYVILL G.: Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer 4*, 2 (July 1988), 65–83. 1

[Fle05] FLETCHER T.: Intel Corporation, personal communication, 2005. 9, 10

[FS88] FUSSELL D., SUBRAMANIAN K. R.: *Fast Ray Tracing using K-D Trees*. Tech. Rep. TR-88-07, University of Texas at Austin, 1988. 1

[Gla84] GLASSNER A.: Space subdivision for fast ray tracing. *IEEE CG&A 4*, 10 (1984), 15–22. 1

[GLGT99] GREGORY A., LIN M. C., GOTTSCHALK S., TAYLOR R.: H-collide: A framework for fast and accurate collision detection for haptic interaction. In *Proc. Virtual Reality Conference 1999* (1999), pp. 38–45. 2

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE CG&A 7*, 5 (1987), 14–20. 2

[HA96] HUMPHREYS G., ANANIAN C.: *TigerSHARK: A Hardware Accelerated Ray-tracing Engine*. Tech. rep., Princeton University, 1996. 2

[Hal01] HALL D.: The AR350: Today's ray trace rendering processor. In *Proc. Eurographics/SIGGRAPH workshop on Graphics hardware - Hot 3D Session 1* (2001). 2

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, Praha, Czech Republic, Apr. 2001. 1

[HV01] HAO X., VARSHNEY A.: Variable-precision rendering. In *Proc. 2001 symposium on Interactive 3D graphics* (2001), pp. 149–158. 2

[JW89] JEVANS D., WYVILL B.: Adaptive voxel subdivision for ray tracing. In *Proc. Graphics Interface '89* (1989), pp. 164–172. 1

[KK86] KAY T., KAJIYA J.: Ray tracing complex scenes. In *Proc. SIGGRAPH '86* (1986), pp. 269–278. 1, 2

[Mac04] MACMURCHY P.: *The Use of Subdivision Surfaces in the Modeling of Plants*. Master's thesis, University of Calgary, 2004. 6

[Mah04] MAHOVSKY J.: *Follow up to 'Fast Ray-Axis Aligned Bounding Box Overlap Tests with Plücker Coordinates'*. Tech. Rep. 2004-759-24, University of Calgary, 2004. Updated January 2005. 3

[MB89] MACDONALD J., BOOTH K.: Heuristics for ray tracing using space subdivision. In *Proceedings of Graphics Interface '89* (Toronto, Ontario, June 1989), Canadian Information Processing Society, pp. 152–63. 2

[MW04] MAHOVSKY J., WYVILL B.: Fast ray-axis aligned bounding box overlap tests with Plücker coordinates. *Journal of Graphics Tools 9*, 1 (2004), 35–46. 1, 2

[Pur04] PURCELL T.: *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004. 2

[RL00] RUSINKIEWICZ S., LEVOY M.: QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of ACM SIGGRAPH* (2000), pp. 343–352. 2

[SM03] SHIRLEY P., MORLEY R. K.: *Realistic Ray Tracing*, second ed. A. K. Peters, Ltd., 2003. 2

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: SaarCOR - A hardware architecture for ray tracing. In *Proc. Graphics Hardware 2002* (2002), pp. 27–36. 1

[Vara] VARIOUS: The Stanford 3D Scanning Repository. On-line: http://graphics.stanford.edu/data/3Dscanrep/. 6

[Varb] VARIOUS: The Walkthru Project - power plant model release. On-line: http://www.cs.unc.edu/~geom/Powerplant/. 6

[WH04] WESTE N., HARRIS D.: *CMOS VLSI Design : A Circuits and Systems Perspective*, third ed. Addison-Wesley, 2004. 9

[Wri99] WRIGLEY A.: U.S. patent 5,933,146 – Method and apparatus for constructing an image of a notional scene by a process of ray tracing, 1999. 2