High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

Watertight Ray Traversal with Reduced Precision

K. Vaidyanathan,[1] T. Akenine-Möller,[1] and M. Salvi

[1]Intel Corporation

Abstract

Reduced precision bounding volume hierarchies and ray traversal can significantly improve the efficiency of ray tracing through low-cost dedicated hardware. A key approach to enabling reduced precision computations during traversal is to translate the ray origin closer to the bounding volume hierarchy node after each traversal step. However, this approach precludes sharing of intersection computations between a parent node and its two children, which is an important optimization. In this paper, we introduce a novel traversal algorithm that addresses this limitation and achieves a significant reduction in the computational complexity of traversal compared to previous approaches. We also include an analysis that shows how our algorithm guarantees watertight intersections which is a key requirement for robust image quality, especially with reduced precision traversal where numerical errors can be large.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

Ray tracing [Whi80] is one of the most fundamental algorithms that can be used to implement physically-based rendering [Kaj86, Jen01, PH10] in a straightforward manner. Since it is based on the ability to query ray-scene intersections along arbitrary directions it is a powerful tool for simulating a wide range of light transport scenarios. Although great progress has been made over the last decade towards optimizing software implementations of ray tracing running on CPU cores [WWB*14] as well as graphics processing units (GPUs) [PBD*10], the time it takes to render an image can still be prohibitive for real-time applications.

Rasterization on GPUs is another widely used rendering technique and although it lacks the versatility of ray tracing, it can be substantially faster and therefore better suited for real-time rendering. A significant part of this speed advantage stems from the fact that the GPUs provide hardware acceleration for the core algorithms and can efficiently map parts of the rendering pipeline to many small SIMD processors that execute in parallel. Furthermore, the graphics hardware industry has spent significant efforts optimizing GPUs over time to derive better performance [KDK*11].

Hardware acceleration can also be applied to improve performance of ray tracing and although this field has received a fair amount of attention in academic research (see Section 2), it is yet to be widely adopted by the real-time graphics industry. With greater emphasis on optimizing GPUs for ray tracing performance, we can expect significant improvements in hardware ray tracing following the trends that have been observed with rasterization.

The key aspects of ray tracing that impact performance are shading and ray-scene intersection computations. Shading be efficiently evaluated on the GPU cores as long as coherency can be extracted for SIMD processing. However traversing spatial acceleration structures to determine ray-scene intersections can be a significant performance overhead. Our work focuses on the problem of traversing bounding volume hierarchies (BVHs) and is motivated by the observation that this can be done efficiently using reduced precision computations combined with BVH compression. These gains can be multiplied by reusing storage as well as ray-box intersection computations for the six bounding planes that are shared between a parent BVH node and its two children [FD09, EW11]. However, with previous reduced precision traversal methods [Kee14], such reuse is not possible.

Moreover, an essential requirement for ray traversal algorithms is watertight intersections [Ize13, WBW13]. This is especially important with reduced precision traversal, where incrementally computed values can accumulate errors and lead to missed intersections. In this paper, we introduce a novel BVH traversal algorithm that:

- pairs reduced precision intersection testing with a compressed BVH node structure that can significantly reduce memory traffic,
- reuses computations for bounding planes shared between a parent node and its children, achieving a significant reduction in intersection cost and
- guarantees watertight intersections, which is a requirement for producing correct results.

2. Previous Work

One of the earliest hardware architectures for ray tracing was introduced by Schmittler et al. [SWS02, SWW*04]. Later Woop et al. [WSS05, SWS06] presented a fully programmable ray-processing unit (RPU) that could also handle dynamic scenes. Nah et al. introduced dedicated hardware for traversal and intersection called the (T&I) engine [NPP*11] and derived a complete ray tracing architecture called RayCore [NKK*14]. Lee et al. [LSL*13] presented SGRT, which featured a parallel-pipelined T&I unit together with a Samsung reconfigurable processor (SRP) for shading and ray generation.

Spjut et al. [SKKB09] introduced TRaX, a programmable architecture for ray tracing with multi-threaded MIMD processing, which was better suited for divergent ray paths. Kopta et al. [KSBD10] further optimized MIMD architectures through sharing of resources such as caches and functional units. In later work, Kopta et al. [KSS*13] reworked the TRaX architecture to traverse ray streams, which were reordered by treelets [AK10] to improve cache hit rates. Lee et al. [LSH*15] proposed a MIMD architecture for ray traversal with a reorder buffer to schedule rays based on cache hits. Although these hardware implementations demonstrated significant efficiency improvements, they did not leverage reduced precision computations.

Efficient hardware implementations for ray-box and ray-triangle intersections have been proposed based on fixed point arithmetic [HK07, HRB*10] or better use of floating point precision [KNP15]. At the same time, the size of the spatial data structure or triangles can be reduced using compression [CSE06,MW06, SE10,KMKY10], implicit indexing and fixed surface reduction per node [BEM10], storing fewer planes per node [WK06, EWM08, FD09] or triangle strip representations [LYTM08]. In previous research, these two problems i.e., intersection testing and reducing the size of the spatial data structure, have been investigated mostly in isolation. The work of Keely [Kee14] is an important exception as it focuses on BVH compression as well as reduced precision traversal. Our approach also addresses these problems in a combined manner and achieves further savings in node size and intersection computations. In addition to this we provide an algorithm that is strictly watertight. While watertight bounding box [Ize13] and ray-triangle intersections [WBW13] have been derived for full-precision traversal, our focus is on quantized bounding boxes and reduced precision ray traversal.

3. Background

3.1. BVH Compression

We start with a description of a compression scheme for BVH bounding boxes along the lines of previous techniques such as Keely [Kee14] and Mahovsky and Wyvill [MW06]. Consider a BVH with bounding volumes represented by axis-aligned bounding boxes that are specified using its minimum coordinate $\mathbf{p}$ and maximum coordinate $\mathbf{q}$, where $(\mathbf{p},\mathbf{q}) \in \mathbb{R}^3$. These coordinates can be compressed by quantizing them to a local low resolution grid and decompressed as the BVH is traversed to evaluate ray-scene intersections. In the following description, the components of a point $\mathbf{p}$ are accessed as $\mathbf{p} = (p_x, p_y, p_z)$.

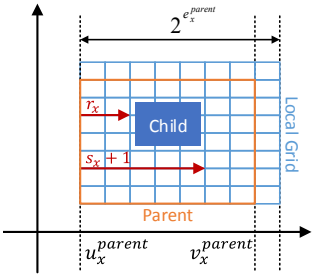


Figure 1: The coordinates of a bounding box can be compressed by computing quantized relative offsets $(\mathbf{r},\mathbf{s})$ in a local grid that is aligned with the parent bounding box.

Let $(\mathbf{u}^{\mathbf{parent}}, \mathbf{v}^{\mathbf{parent}})$ be the corresponding coordinates of the *quantized* bounding box. The next step is to compress the bounding boxes of the child nodes inside the parent. Since the BVH compression is lossy, the decompressed coordinates can be different from the original coordinates. The local low resolution grid is selected such that the origin of the grid is aligned with $\mathbf{u}^{parent}$ as shown in Figure 1 and the dimensions of the grid are selected to be the smallest value that is a power of two and greater than the parent box. The quantized bounding box is derived by computing relative offsets of the bounding box coordinates to the origin of the grid and quantizing the result to a small number of bits, N_b. For an axis i, these operations are described by

$$r_i = \left\lfloor \frac{\left(p_i - u_i^{parent}\right)}{2^{e_i^{parent}}} 2^{N_b} \right\rfloor, \quad s_i = \left\lfloor \frac{\left(q_i - u_i^{parent}\right)}{2^{e_i^{parent}}} 2^{N_b} \right\rfloor, \qquad (1)$$

where $2^{e_i^{parent}}$ is the grid dimension along x and $(\mathbf{r},\mathbf{s}) \in \mathbb{Z}^3_{>0}$ are the quantized offsets relative to the quantized parent bounding box. Using the floor rounding mode guarantees that $(r_i, s_i) \in [0, 2^{N_b} - 1]$. The exponent e_i^{parent} is given by [Kee14]

$$e_i^{parent} = \arg\min_k \left(2^k > \left(v_i^{parent} - u_i^{parent}\right)\right). \qquad (2)$$

The bounding boxes can be iteratively decompressed from the quantized offsets when the BVH is traversed from the root node as described in Equations 3 and 4, i.e.,

$$u_i = u_i^{parent} + r_i 2^{(e_i^{parent} - N_b)}, \qquad (3)$$

$$v_i = u_i^{parent} + (s_i + 1) 2^{(e_i^{parent} - N_b)}, \qquad (4)$$

where $(\mathbf{u},\mathbf{v})$ are the decompressed coordinates that are initialized with the coordinates of the root box at the start of the traversal. The value of s_i is incremented by one, which compensates for the negative error introduced by the $floor$ operation in Equation 1. This also guarantees $v_i - u_i > 0$, which is required for deterministic grid dimensions if a parent node is decompressed from a child node [Kee14]. Our representation is slightly different (see Section 4).

3.2. Reduced Precision Traversal

Intersections between rays and the compressed BVH are determined by traversing the BVH, decompressing the bounding boxes and computing ray-bounding box intersections at each node, which can be evaluated using Kay and Kajiya's slab test [KK86].

The slab test uses a parametrized ray representation $\mathbf{o}+t\mathbf{d}$, where $\mathbf{o}$ is the ray origin, $\mathbf{d}$ is the ray direction and t is the distance along the ray. Ray-bounding box intersections can be determined by comparing the parametric distances from the ray origin to each of the six planes of the bounding box. The parametric distances to the two planes along the axis i are given by

$$\lambda_i = (u_i - o_i)w_i, \quad (5)$$

$$\mu_i = (v_i - o_i)w_i, \quad (6)$$

where $w_i = \frac{1}{d_i}$ is the ray slope. Assuming the slopes (w_x, w_y, w_z) are positive, a ray intersects a bounding box if

$$\max(\lambda^{\max}, t^{min}) \leq \min(\mu^{\min} + 2\text{ulp}(\mu^{\min}), t^{max}), \quad (7)$$

where $\lambda^{\max}$ and $\mu^{\min}$ are the maximum and minimum values of λ_i and μ_i respectively and (t^{min}, t^{max}) represent the clip distances of the ray. The function ulp() returns the unit in the last place of a finite precision floating-point value. Adding this compensation term to $\mu^{\min}$ ensures watertight intersection results [Ize13]. When $w_i < 0$, the corresponding values of λ_i and μ_i have to be swapped.

The parametrc distances can be computed with reduced precision arithmetic if the origin of the ray (traversal point) is moved closer to the bounding box [Kee14]. This can be done by updating the origin of the ray at each traversal step. However, if the ray origin is modified, the values of $(\boldsymbol{\lambda}, \boldsymbol{\mu})$ computed for the parent node cannot be reused for the shared bounding planes of the child nodes. Therefore, for a pair of sibling nodes, Equations 5 and 6 have to be evaluated for twelve bounding planes, which could otherwise be reduced by half [FD09, EW11].

4. Our BVH Node Structure

The bounding boxes of a pair of sibling nodes share six bounding planes (coordinates) with the parent node. This can be leveraged to derive a compact node structure that stores six bounding planes for a pair of sibling nodes and a pair of 3-bit masks $(\mathbf{l}, \mathbf{m})$ that assign the planes to the left or right sibling [FD09, EW11]. We use a similar node structure but store quantized plane offsets instead of the full precision coordinates.

Assuming a depth-first layout for a binary tree we store an N_p-bit index to the right child pair, one bit to indicate an internal or leaf node, six compressed coordinates and a pair of 3-bit masks as shown in Table 1. If the index of the right child pair is relative to the parent node, then this index is always even and the least significant bit of the index can be dropped. Therefore, if N_p is the number of bits used for the child index, a single BVH can contain $2^{N_p+1} - 1$ nodes. Assuming $N_b = 6$ and $N_p = 21$, only 8 bytes are required to store a *pair* of nodes. This is just a third of the node size used in Keely's work [Kee14].

5. Algorithm

In this section, we derive a reduced precision traversal algorithm that can reuse computations for planes that are shared with the parent node by combining bounding box decompression with traversal.

Field	Number of Bits
Leaf node indicator	1
$(\mathbf{l}, \mathbf{m})$	6
$(\mathbf{r}, \mathbf{s})$	$N_b \times 6$
Index to right child pair	N_p

Table 1: Layout for a pair of sibling nodes. Only six bounding planes are stored and the remaining six bounding planes are shared with the parent node.

The parametric distances for six planes can be incrementally derived from the parent node as

$$\lambda_i = \lambda_i^{parent} + w_i r_i 2^{(e_i^{parent} - N_b)}, \quad (8)$$

$$\mu_i = \lambda_i^{parent} + w_i s_i 2^{(e_i^{parent} - N_b)}. \quad (9)$$

Since r_i is a reduced precision term, the multiplication operations in Equations 8 and 9 can be computed with lower precision. Note that with this traversal algorithm, the decompressed bounding boxes are no longer available. Therefore, the exponent $\mathbf{e}^{parent}$ cannot be directly computed. In Section 5.4, we introduce a low-cost method for computing exponents without decompressing the bounding boxes.

5.1. Watertight Compression and Traversal

With each traversal step, numerical errors can be introduced through finite precision floating-point arithmetic, which results in a lower bound $(\boldsymbol{\lambda}^{lower}, \boldsymbol{\mu}^{lower})$ and an upper bound $(\boldsymbol{\lambda}^{upper}, \boldsymbol{\mu}^{upper})$ for the parametric distances. Assuming the slope w_i is positive, a necessary and sufficient condition for watertight intersections is that for each axis i, $\lambda_i^{upper} \leq \lambda_i$ and $\mu_i^{lower} \geq \mu_i$.

Using an approach similar to Ize's work [Ize13], the upper bound for λ_i after the first traversal step, can be derived from Equation 8, i.e.,

$$\lambda_i^{upper} = \left(\lambda_i^{parent} + w_i r_i^{upper} 2^{(e_i^{parent} - N_b)}(1+\varepsilon)^2\right)(1+\varepsilon), \quad (10)$$

where ε is the machine epsilon of the underlying floating-point representation and errors are introduced by the computation of the slope w_i, the inner multiplication, and the outer addition. The value of r_i^{upper} can be derived from the leftmost expression in Equation 1, which results in

$$r_i^{upper} = \left(p_i - u_i^{lower}\right) 2^{(N_b - e_i^{parent})}. \quad (11)$$

Working backwards from Equation 11, our first step towards watertight intersections is to ensure that $u_i^{lower} \geq u_i$ by modifying Equation 3, resulting in

$$u_i = RU\left(u_i^{parent} + r_i 2^{(e_i^{parent} - N_b)}\right), \quad (12)$$

where $RU()$ is a rounding rule that rounds the result of the floating-point operations upward towards positive infinity [IEE08]. With fixed-function ray traversal, these rounding rules can be baked into the floating-point hardware. Next, we ensure that $r_i^{upper} \leq r_i$ by modifying Equation 1 into

$$r_i = \left\lfloor RD\left(p_i - u_i^{parent}\right) 2^{(N_b - e_i^{parent})} \right\rfloor, \quad (13)$$

where $RD()$ rounds the result of the floating-point operations downwards towards negative infinity [IEE08].

Finally, we ensure that $\lambda_i^{upper} \leq \lambda_i$ by modifying Equation 8 so that

$$\lambda_i = RD\left(\lambda_i^{parent} + RD\left(w_i r_i\right) 2^{\left(e_i^{parent} - N_b\right)}\right), \tag{14}$$

where the slope w_i is also rounded to the lower value. Therefore, the *first constraint* for watertightness i.e., $\lambda_i^{upper} \leq \lambda_i$, is met. Although we have eliminated positive errors in the value of λ_i, negative errors still remain. Moreover, these negative errors get accumulated with each traversal step. Unfortunately since the offset for μ_i is derived with respect to λ_i^{parent}, errors in λ_i are coupled with the errors in μ_i. Therefore, accumulated negative errors in λ_i would lower the value of μ_i^{lower} violating the second constraint for watertightness.

However, we can decouple λ_i and μ_i using an alternative compression scheme, where the offsets for q_i are derived with respect to v_i^{parent} as shown in Figure 2. Negative numerical errors in μ_i can then be independently eliminated similar to positive errors in λ_i. Algorithm 1 describes the compression part and Algorithm 2 describes the incremental computation of parametric distances. These distances are then compared to determine ray-box intersections for the sibling nodes as described in Equation 7.

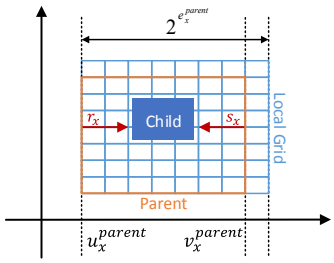


Figure 2: We decouple the minimum and maximum coordinates by computing the relative offset of the maximum co-ordinate $\mathbf{q}$ (not shown) with respect to the maximum decompressed coordinate of the parent box $\mathbf{v}^{parent}$ instead of the minimum coordinate $\mathbf{u}^{parent}$.

Algorithm 1 Bounding box compression for a pair of sibling nodes with decoupled minimum and maximum coordinates. Bounding box coordinates that are not shared with the parent box are indicated by a pair of three bit masks $(\mathbf{l}, \mathbf{m})$.

for all $i \in (x, y, z)$ **do**
  if $p_i^{left} \neq p_i^{parent}$ **then**
    $l_i = 1$
    $r_i = \left\lfloor RD\left(p_i^{left} - u_i^{parent}\right) 2^{(N_b - e_i)} \right\rfloor$
  else
    $l_i = 0$
    $r_i = \left\lfloor RD\left(p_i^{right} - u_i^{parent}\right) 2^{(N_b - e_i)} \right\rfloor$
  $u_i = RU\left(u_i^{parent} + r_i\right) 2^{(e_i - N_b)}$

  if $q_i^{left} \neq q_i^{parent}$ **then**
    $m_i = 1$
    $s_i = \left\lfloor RD\left(v_i^{parent} - q_i^{left}\right) 2^{(N_b - e_i)} \right\rfloor$
  else
    $m_i = 0$
    $s_i = \left\lfloor RD\left(v_i^{parent} - q_i^{right}\right) 2^{(N_b - e_i)} \right\rfloor$
  $v_i = RD\left(v_i^{parent} - s_i\right) 2^{(e_i - N_b)}$

Algorithm 2 Watertight traversal for a pair of sibling nodes. If the slope w_i is negative the minimum and maximum coordinates are swapped. Six parametric distances are computed and then assigned to the left or right child depending on the mask $(\mathbf{l}, \mathbf{m})$.

for all $i \in (x, y, z)$ **do**
  // Initialize (λ, μ) for left and right nodes
  $\lambda_i^{left} = \lambda_i^{parent}$
  $\mu_i^{left} = \mu_i^{parent}$
  $\lambda_i^{right} = \lambda_i^{parent}$
  $\mu_i^{right} = \mu_i^{parent}$
  // Compute new distances. Swap r_i and s_i if slope is negative
  if $w_i \geq 0$ **then**
    $\lambda_i = RD\left(\lambda_i^{parent} + RD\left(w_i r_i\right) 2^{\left(e_i^{parent} - N_b\right)}\right)$
    $\mu_i = RU\left(\mu_i^{parent} - RD\left(w_i s_i\right) 2^{\left(e_i^{parent} - N_b\right)}\right)$
  else
    $\lambda_i = RD\left(\lambda_i^{parent} + RD\left(|w_i| s_i\right) 2^{\left(e_i^{parent} - N_b\right)}\right)$
    $\mu_i = RU\left(\mu_i^{parent} - RD\left(|w_i| r_i\right) 2^{\left(e_i^{parent} - N_b\right)}\right)$
  // Assign computed distances to left and right nodes
  if ($l_i = 0$ and $w_i \geq 0$) or ($m_i = 0$ and $w_i < 0$) **then**
    $\lambda_i^{left} = \lambda_i$
  if ($m_i = 0$ and $w_i \geq 0$) or ($l_i = 0$ and $w_i < 0$) **then**
    $\mu_i^{left} = \mu_i$
  if ($l_i = 1$ and $w_i \geq 0$) or ($m_i = 1$ and $w_i < 0$) **then**
    $\lambda_i^{right} = \lambda_i$
  if ($m_i = 1$ and $w_i \geq 0$) or ($l_i = 1$ and $w_i < 0$) **then**
    $\mu_i^{right} = \mu_i$

5.2. Degenerate Axes

If a component of the ray direction d_i along an axis i is zero, then the corresponding values of (λ_i, μ_i) will evaluate to $\pm\infty$ and therefore cannot be derived incrementally. In such a scenario, we mark the axis i as a degenerate axis and instead of incrementally computing the values of (λ_i, μ_i), we incrementally compute the offset of the plane coordinates relative to the ray origin $(u_i - o_i, v_i - o_i)$. Note that is equivalent to incrementally computing the value of (λ_i, μ_i) with a slope $w_i = 1$. Once we have derived these coordinate offsets, we can directly compare them against the ray origin to derive the values of (λ_i, μ_i),

if $u_i - o_i \leq 0$ and $v_i - o_i \geq 0$ **then**
  $\lambda_i = -\infty$
  $\mu_i = \infty$
else
  $\lambda_i = \infty$
  $\mu_i = -\infty$.

5.3. Ray Setup

For each ray-BVH intersection, the values of the parametric distances $(\boldsymbol{\lambda}^{root}, \boldsymbol{\mu}^{root})$ are initialized by performing a full precision intersection test with the quantized root bounding box. The initial parametric distances and the ray slope are only computed once per ray and therefore can be evaluated in the shader before initiating hardware based traversal. The computed ray slopes can also be used for watertight ray-triangle intersections [WBW13].

	Showdown	Temple	Crown	San Miguel
Keely (8-Bit)	64.9 / 5.1	95.5 / 8.0	60.5 / 8.6	97.5 / 8.3
Keely (10-Bit)	64.0 / 4.7	91.8 / 6.6	58.1 / 7.8	94.2 / 7.4
Our	63.8 / 4.7	90.7 / 6.2	57.3 / 7.6	93.2 / 7.2
Full Precision	54.6 / 4.0	78.3 / 4.9	51.6 / 7.0	86.5 / 6.6

Table 2: The number of internal-nodes / primitive-leaves traversed for the four test scenes, using different traversal methods.

5.4. Computing Grid Dimensions

Since we combine bounding box decompression and traversal, the decompressed bounds are no longer available. However, the grid size d_i along an axis i can be computed separately. Note that an unsigned floating-point value with an N_b-bit mantissa is sufficient to represent the grid size. At each traversal step, the value of d_i can be iteratively derived from its previous value d_i^{parent} as

$$d_i = d_i^{parent} - (r_i + s_i) e_i^{parent}. \tag{15}$$

The exponent of d_i gives the grid exponent, i.e., $e_i = exponent(d_i)$. The initial value of the grid size $\mathbf{d}^{root}$ and the quantized coordinates of the root bounding box are precomputed during BVH compression.

5.5. Traversal Stack

In order to share bounding planes and traversal computations between a parent node and its children, the BVH should be traversed in a *topological* order i.e., a parent node should be traversed before its children. Typically such a traversal would require a stack, where each stack entry would store the traversal state for the parent node, which includes the full precision values of $\boldsymbol{\lambda}$, $\boldsymbol{\mu}$ and $\mathbf{d}$ as well as the index of the far child node pair. A single stack entry requires close to 32 bytes, which is large compared to an uncompressed BVH, where only the index of the far child is stored.

In order to reduce the memory requirements of the stack, we truncate it to the top four entries using a restart trail [Lai10]. With a four entry stack, the number of additional traversal steps resulting from restarts is less than 10%. We assume a dedicated storage of 128 bytes per ray for the truncated stack. The total size of this memory on-chip would depend on the maximum number of rays in flight, which is an architecture dependent parameter outside the scope of our analysis.

Keely [Kee14] avoids a stack entirely using bidirectional node references to retrace traversal steps [KSS*13]. However this approach does not guarantee topological ordering, which precludes sharing of bounding planes. Furthermore with a topological traversal order, the accumulated numerical errors in λ and μ are bounded by the maximum depth of the BVH, which is significantly smaller than the maximum number of traversal steps.

6. Results

In order to analyze the implementation and performance trade-offs of our approach, we compare it against the reduced precision traversal technique of Keely [Kee14], which we call *traversal point update* (TPU). We evaluate these traversal methods using the four test scenes shown in Figure 3, at a resolution of 512×512 pixels and 16 samples per pixel.

The Showdown and Temple scenes have less than one million triangles, which is representative of the geometric complexity observed in typical real-time applications. The Showdown scene is an outdoor environment with ray traced direct lighting, while Temple is an indoor scene one-bounce diffuse indirect lighting. The Crown scene has over five million triangles and several glossy and refractive surfaces, which we render with up to four indirect ray bounces. The San Miguel scene is a complex outdoor scene with over one million triangles and rendered with direct environment lighting.

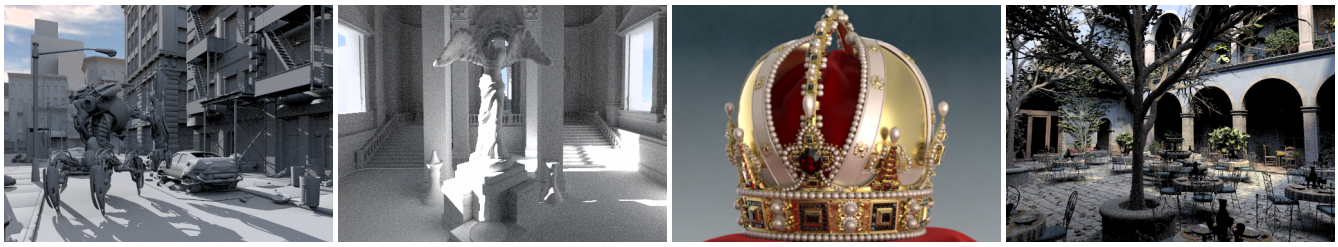
The Showdown, Temple and San Miguel scenes have been rendered using PBRT [PH10], while the Crown scene has been rendered using a renderer based on Embree [WWB*14]. For all the scenes, we extract the ray and geometry information from the renderer and simulate traversal in a standalone functional simulator. Watertightness is verified by comparing each ray intersection against the result produced by robust full precision traversal [Ize13]. We use a BVH builder based on a binned surface area heuristic (SAH) [Wal07]. We do not split primitives across spatial divisions [SFD09], which can further improve performance.

Modifications to TPU: Considering the significantly smaller node size that can be achieved by sharing bounding planes with the parent node, we use this node structure for both traversal methods. As discussed in Section 5.5, this requires a traversal stack. Therefore, we introduce a minor modification to TPU, where the de-compressed coordinates are stored in the stack along with the coordinates of the origin and the distance to the translated origin $t^{\min}$. The translated value of $t^{\max}$ is derived at each traversal step by adding the updated value of t^{min}. Note that this introduces an extra addition to the original method. With these modifications to TPU, we make it *practically watertight*. It is guaranteed that the decompressed bounding boxes are strictly watertight. However a few missed intersections can still result from accumulated numerical errors in the ray clip bounds $t^{\min}$ and $t^{\max}$.

6.1. Analysis

Figure 4 (left) shows the relative increase in the number of traversal steps with our method compared to full precision BVH nodes and traversal, when the plane offsets are quantized to different numbers of bits N_b. With 8-bit plane offsets the increase in traversal steps is less than 5% and with 6-bit offsets, it is between 8% and 17%. It is interesting to note that the San Miguel and Crown scenes, which have higher geometry complexity, also have a smaller relative increase in the number of traversal steps. Splitting larger triangles could potentially improve performance with the Showdown and Temple scenes, which we leave for future analysis.

We pick 6-bit plane offsets for the remaining part of our analysis, which allows a pair of nodes can be stored in 8 bytes and presents a good trade-off between cache-aligned node sizes and the number of traversal steps. Figure 4 (middle) shows the relative increase in the number of traversal steps with TPU using different mantissa widths for the intersection test. Similar to Keely [Kee14], we use a one-bit multiplier in our implementation of TPU to compute the offset to the new origin. Using 6-bit mantissas results in a large increase in the number of traversal steps but this drops significantly with 8-bit mantissas, where the increase is between 14% and 25%.



Showdown Temple Crown San Miguel

Figure 3: Higher quality renderings of the four test scenes used in our evaluation. The Showdown and San Miguel scenes are rendered with direct environment lighting, while the Temple and Crown scenes are path traced, i.e., with indirect lighting.

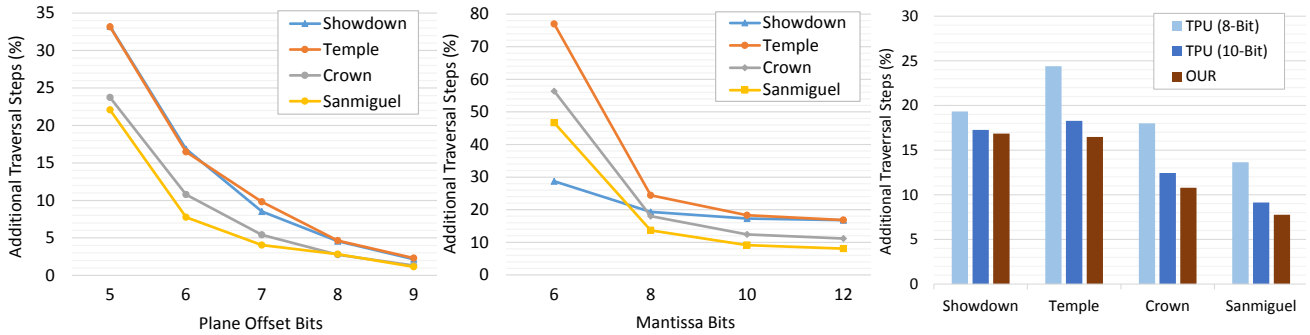


Figure 4: Relative increase in the number of traversal steps with different reduced precision options. Left: our method with different number of bits N_b for the plane offset. Middle: traversal point update (TPU) with 6-bit plane offsets and using different sizes of mantissa bits for the intersection arithmetic. Right: TPU with intersection tests using 8-bit and 10-bit mantissas and our method, both with 6-bit plane offsets.

Figure 4 (right) compares the relative increase in traversal steps using our technique against TPU with 8-bit as well as 10-bit mantissas. Since we use full precision values for the slopes and the parametric distances, our method presents the lower bound for the increase in traversal steps. With 10-bit mantissas, the performance of TPU is close to our method and with 8-bit mantissas, it introduces between 2% to 8% additional traversal steps. Table 2 shows a more detailed comparison of the different traversal methods.

6.2. Bandwidth

The memory traffic resulting from BVH node fetches depends on the system architecture and design choices such as the sizes and hierarchy of the caches, ordering of nodes, the number of parallel traversal tasks, etc. Although these aspects are outside the scope of our paper, the potential reduction in node bandwidth with the compact node structure of Section 4 remains one of the key motivations for our work. Therefore, we include a simplified bandwidth analysis using a functional implementation of ray traversal, where we measure the bytes fetched per ray based on the number of cache line misses that are observed. We use a simple ordered depth-first BVH layout [NPK*10, LSL*13] to improve cache line locality.

Figure 5 shows the bytes fetched per ray for the Temple scene, assuming a fully associative cache of 32 kB. Compared to a full precision BVH node pair (32 bytes), a node pair with 6-bit plane offsets compressed to 8 bytes brings a reduction of about 50% in node bandwidth when using cache lines of 64 bytes. With cache lines of 32 bytes, this is further reduced to about 40%. Considering the small size of a compressed node compared to the size of a cache line, a node layout scheme that re-orders nodes across multiple BVH levels [YM06], can potentially achieve further savings in bandwidth. Note that there is no additional bandwidth associated with the stack as it is stored on-chip in a truncated form.

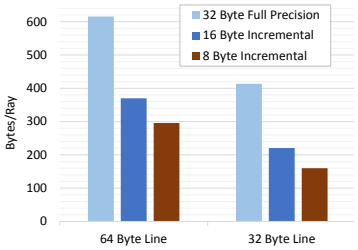


Figure 5: Number of node bytes fetched per ray for the Temple scene using a cache size of 32 kB, with 32 bytes and 64 bytes line sizes. We compare three configurations, namely, a full precision node pair (32 bytes), a node pair with 16-bit plane offsets (16 bytes), and a node pair with 6-bit plane offsets (8 bytes). The bounding planes are shared with the parent node for all configurations.

6.3. Intersection Errors

As discussed in Section 6, we compare our watertight algorithm against a modified version of TPU, which is more robust but not strictly watertight. Figure 6 reports the fraction of the ray-bounding box tests that result in false misses with TPU, compared to the watertight intersection test of Ize [Ize13]. Our algorithm on the other hand does not produce any false misses.

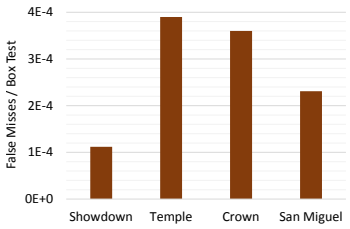


Figure 6: The ratio of false misses to the total number of bounding box intersection tests.

6.4. Arithmetic Complexity

Table 3 lists the different floating-point operations for TPU as well as our method. It also includes a technology-independent estimate of the arithmetic complexity in terms of the number of equivalent NAND-gates. These estimates were derived by synthesizing parameterized floating-point units from the Synopsys Designware Foundation library with two pipeline stages. Translating the ray origin closer to the node enables lower precision intersection tests with TPU compared to our approach. However, the number of operations and the overall arithmetic complexity is significantly lower with our approach, which requires 60% fewer gates compared to TPU.

The subtractor in step 6 (of TPU), requires full precision inputs in order to limit the impact of catastrophic cancellations [Gol91]. However since the output of this subtractor is truncated to 8 bits, its implementation can be optimized. We approximate the complexity of this operation with an 8-bit subtracter [Kee14]. One of the operands in the additions corresponding to step 4 and 8 has a one-bit mantissa. We synthesize this adder by forcing 23 mantissa bits of one operand to zero, which results in a 14% reduction in gate count. From Equation 15, we can see that both operands in the subtraction operation for computing the grid size (step 1) have the same exponent. Therefore these subtractors do not require alignment of the input radix. Synthesizing these adders with a shared exponent results in a significant gate count reduction, which is close to 50%.

Traversal Point Update			
No.	Computation Step	Floating-Point Operation	Gates
1	Grid exponent	6, 6-bit sub	1888
2	Decompressed bounds	6, 24-bit add	17400
3	Full precision origin	6, 24-bit add	17400
4	$t^{\min}$	2, 24-bit add	4980
5	$t^{\max}$	2, 24-bit add	4980
6	Plane distances (λ, μ)	12, 8-bit sub	11506
7		12, 8×8-bit multiply	9899
8	Distance to boxes	2, 24-bit add	4980
	Total		73,032

OUR			
No.	Computation Step	Floating-Point Operation	Gates
1	Grid exponent	6, 6-bit sub	1888
2	Plane distances (λ, μ)	6, 24-bit add	17400
3		6, 6×24 bit multiply	9790
	Total		29,078

Table 3: Arithmetic complexity for intersecting a pair of sibling nodes with traversal point update (TPU) [Kee14] and our method respectively.

7. Conclusion

Techniques like data compression and reduced precision arithmetic are commonly used to achieve greater efficiency in modern GPU pipelines. We aim to derive similar gains by leveraging these techniques for hardware ray tracing. Towards this goal, we have introduced a joint algorithm for reduced precision traversal and BVH compression that achieves a significant reduction in intersection cost and traversal bandwidth, across a variety of scenes. Our algorithm also produces robust watertight results, meeting the requirements for high quality rendering.

Having reduced the cost of processing BVH nodes, we note that the cost of processing triangles remains an important target for future optimizations. Unfortunately lossy compression that is well suited for a BVH, is not applicable to triangle data. It remains to be seen if fast lossless compression schemes, such as those used for color and depth data in a rasterization pipeline, can be applied to triangles in a raytracing architecture.

We also note that building a system architecture for ray tracing that leverages these energy efficient techniques, is an interesting challenge for the future. We are positive that with consistent efforts in that direction, ray tracing can become a strongly competitive choice for real time rendering

Acknowledgments We would like to thank David Blythe, Charles Lingle and Tom Piazza for supporting this research. We also thank David Baldwin for his contributions. The Showdown and the Temple scenes are courtesy of Epic Games, Inc. The Crown scene is courtesy of Martin Lubich and the San Miguel scene was modeled by Guillermo M. Leal Llaguno. We are grateful for these scenes.

References

[AK10] AILA T., KARRAS T.: Architecture Considerations for Tracing Incoherent Rays. In *High-Performance Graphics* (2010), pp. 113–122. 2

[BEM10] BAUSZAT P., EISEMANN M., MAGNOR M.: The Minimal Bounding Volume Hierarchy. In *Vision, Modeling, and Visualization* (Siegen, Germany, 2010), pp. 227–234. 2

[CSE06] CLINE D., STEELE K., EGBERT P.: Lightweight Bounding Volumes for Ray Tracing. *Journal of Graphics, GPU, and Game Tools, 11*, 4 (2006), 61–71. 2

[EW11] ERNST M., WOOP S.: Ray Tracing with Shared-Plane Bounding Volume Hierarchies. *Journal of Graphics, GPU, and Game Tools, 15*, 3 (2011), 141–151. 1, 3

[EWM08] EISEMANN M., WOIZISCHKE C., MAGNOR M.: Ray Tracing with the Single-Slab Hierarchy. In *Vision, Modeling, and Visualization* (2008), pp. 373–381. 2

[FD09] FABIANOWSKI B., DINGLIANA J.: Compact BVH Storage for Ray Tracing and Photon Mapping. In *Proceedings of Eurographics Ireland Workshop* (2009), pp. 1–8. 1, 2, 3

[Gol91] GOLDBERG D.: What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Computing Surveys, 23*, 1 (Mar. 1991), 5–48. 7

[HK07] HANIKA J., KELLER A.: Towards Hardware Ray Tracing using Fixed Point Arithmetic. In *IEEE Symposium on Interactive Ray Tracing* (2007), pp. 119–128. 2

[HRB*10] HEINLY J., RECKER S., BENSEMA K., PORCH J., GRIBBLE C.: Integer Ray Tracing. *Journal of Graphics, GPU, and Game Tools, 14*, 4 (2010), 31–56. 2

[IEE08] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70. doi:10.1109/IEEESTD.2008.4610935. 3

[Ize13] IZE T.: Robust BVH Ray Traversal. *Journal of Computer Graphics Techniques, 2*, 2 (2013), 12–27. 1, 2, 3, 5, 6

[Jen01] JENSEN H. W.: *Realistic Image Synthesis Using Photon Mapping*. AK Peters Ltd., 2001. 1

[Kaj86] KAJIYA J. T.: The Rendering Equation. In *Computer Graphics (Proceedings of SIGGRAPH 86)* (1986), ACM, pp. 143–150. 1

[KDK*11] KECKLER S. W., DALLY W. J., KHAILANY B., GARLAND M., GLASCO D.: GPUs and the Future of Parallel Computing. *IEEE Micro, 31*, 5 (2011), 7–17. 1

[Kee14] KEELY S.: Reduced Precision for Hardware Ray Tracing in GPUs. In *High-Performance Graphics* (2014), pp. 29–40. 1, 2, 3, 5, 7

[KK86] KAY T. L., KAJIYA J. T.: Ray Tracing Complex Scenes. In *Computer Graphics (Proceedings of SIGGRAPH 86)* (1986), ACM, pp. 269–278. 2

[KMKY10] KIM T.-J., MOON B., KIM D., YOON S.-E.: RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies. *IEEE Transactions on Visualization and Computer Graphics, 16*, 2 (2010), 273–286. 2

[KNP15] KIM D., NAH J.-H., PARK W.-C.: Geometry Transition Method to Improve Ray-Tracing Precision. *Multimedia Tools and Applications,* (2015), 1–12. 2

[KSBD10] KOPTA D., SPJUT J., BRUNVAND E., DAVIS A.: Efficient MIMD Architectures for High-Performance Ray Tracing. In *IEEE International Conference on Computer Design* (2010), pp. 9–16. 2

[KSS*13] KOPTA D., SHKURKO K., SPJUT J., BRUNVAND E., DAVIS A.: An Energy and Bandwidth Efficient Ray Tracing Architecture. In *High-Performance Graphics* (2013), ACM, pp. 121–128. 2, 5

[Lai10] LAINE S.: Restart Trail for Stackless BVH Traversal. In *High-Performance Graphics* (2010), pp. 107–111. 5

[LSH*15] LEE W.-J., SHIN Y., HWANG S. J., KANG S., YOO J.-J., RYU S.: Reorder Buffer: An Energy-efficient Multithreading Architecture for Hardware MIMD Ray Traversal. In *High-Performance Graphics* (2015), pp. 21–32. 2

[LSL*13] LEE W.-J., SHIN Y., LEE J., KIM J.-W., NAH J.-H., JUNG S., LEE S., PARK H.-S., HAN T.-D.: SGRT: A Mobile GPU Architecture for Real-time Ray Tracing. In *High-Performance Graphics* (2013), ACM, pp. 109–119. 2, 6

[LYTM08] LAUTERBACH C., YOON S.-E., TANG M., MANOCHA D.: ReduceM: Interactive and Memory Efficient Ray Tracing of Large Models. *Computer Graphics Forum, 27*, 4 (2008), 1313–1321. 2

[MW06] MAHOVSKY J., WYVILL B.: Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum, 25*, 2 (2006), 173–182. 2

[NKK*14] NAH J.-H., KWON H.-J., KIM D.-S., JEONG C.-H., PARK J., HAN T.-D., MANOCHA D., PARK W.-C.: RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices. *ACM Transactions on Graphics, 33*, 5 (2014), 162:1–162:15. 2

[NPK*10] NAH J.-H., PARK J.-S., KIM J.-W., PARK C., HAN T.-D.: Ordered Depth-first Layouts for Ray Tracing. In *ACM SIGGRAPH ASIA Sketches* (2010), pp. 55:1–55:2. 6

[NPP*11] NAH J.-H., PARK J.-S., PARK C., KIM J.-W., JUNG Y.-H., PARK W.-C., HAN T.-D.: T & I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing. *ACM Transactions on Graphics, 30*, 6 (2011), 160:1–160:10. 2

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics, 29*, 4 (2010), 66:1–66:13. 1

[PH10] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*, 2nd ed. Morgan Kaufmann, 2010. 1, 5

[SE10] SEGOVIA B., ERNST M.: Memory Efficient Ray Tracing with Hierarchical Mesh Quantization. In *Graphics Interface* (2010), pp. 153–160. 2

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial Splits in Bounding Volume Hierarchies. In *High-Performance Graphics* (2009), pp. 7–13. 5

[SKKB09] SPJUT J., KENSLER A., KOPTA D., BRUNVAND E.: TRaX: A Multicore Hardware Architecture for Real-time Ray Tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28*, 12 (2009), 1802–1815. 2

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: SaarCOR: A Hardware Architecture for Ray Tracing. In *Graphics Hardware* (2002), pp. 27–36. 2

[SWS06] SVEN WOOP E. B., SLUSALLEK P.: Estimating Performance of a Ray-Tracing ASIC Design. In *IEEE Symposium on Interactive Ray Tracing 2006* (September 2006), pp. 7–14. 2

[SWW*04] SCHMITTLER J., WOOP S., WAGNER D., PAUL W. J., SLUSALLEK P.: Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Graphics Hardware* (2004), pp. 95–106. 2

[Wal07] WALD I.: On Fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing* (2007), IEEE Computer Society, pp. 33–40. 5

[WBW13] WOOP S., BENTHIN C., WALD I.: Watertight Ray/Triangle Intersection. *Journal of Computer Graphics Techniques, 2*, 1 (2013), 65–82. 1, 2, 4

[Whi80] WHITTED T.: An Improved Illumination Model for Shaded Display. *Communications of the ACM, 23*, 6 (1980), 343–349. 1

[WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Eurographics Symposium on Rendering* (2006), Eurographics Association, pp. 139–149. 2

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics, 24*, 3 (2005), 434–444. 2

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics, 33*, 4 (2014), 143:1–143:8. 1, 5

[YM06] YOON S.-E., MANOCHA D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum, 25*, 3 (2006), 507–516. 6