GPU Ray Tracing – Comparative Study on Ray-Triangle Intersection Algorithms

Vladimir Shumskiy

Moscow Institute of Physics and Technology
Air Graphics
v.a.shumskiy@gmail.com

Abstract. I present a comparative study on GPU ray tracing implemented for two different types of ray-triangle intersection algorithms used with BVH (Bounding Volume Hierarchy) spatial data structure evaluated for performance on three static scenes. I study how number of triangles placed in a BVH leaf node affects rendering performance. I propose GPU-optimized SIMD ray-triangle intersection method evaluated on GPU for path-tracing and compare it's performance with plain Moller-Trumbore and Unit Triangle intersection methods.

Keywords: ray-triangle intersection, GPU programming, Direct3D, Direct-Compute, performance study, ray tracing, bounding volume hierarches.

1 Introduction

While modern graphics cards (GPUs) allow for general computation in a parallel manner, one of the most prominent applications for a GPU is image synthesis. This is thanks to the inherent parallel nature of ray tracing and other global illumination algorithms – the decomposition of images into pixels provides a natural way of creating individual tasks for many parallel processors. Unlike the GPUs a few years ago, modern ones allow us full programmability similar to general CPUs, while the streaming computation model has its own specific issues. This has to be taken into account when adopting the data structures, traversal algorithms and intersection test routines for ray tracing on GPU architecture.

Testing framework for this paper is based on formerly published papers that implement ray-tracing with spatial data structures in GPU. We use bounding volume hierarchies as described in [5] and few different ray-triangle intersection methods, especially Moller-Trumbore [8] and Unit Triangle [14] routine. While performance of each of those algorithms was successfully studied separately, little attention was paid to how triangle-intersection method can affect spatial data structure traversal performance and vice versa. Furthermore, the performance has not been carefully compared on a current programmable GPU architecture, especially using a cross-vendor APIs like OpenCL, DirectCompute or C++ AMP. In this paper we first present such a comparison study dealing with efficiency of two different types of ray-triangle intersection algorithms for ray tracing on GPU.

This paper is further structured as follows. Section 2 summarizes the previous work of ray-triangle intersection on both CPU and GPU and performance comparison on those algorithms. Section 3 describes our choices for implementation. Section 4 shows the result from measurements on two GPUs for a set of scenes. Further it discusses the bottlenecks of a contemporary GPU architecture for ray tracing algorithms. Section 5 concludes the paper with possible perspectives for future work.

2 Previous Work

Due to important role in computer graphics plenty of research has been done in the field of intersection testing algorithms. Algorithms proposed by Snyder and Barr [11], Badouel [3], Moller-Trumbore [8], Woop [14], Wald and Shevtsov et al. [10], have been successfully compared and studied [9], [2], [3], [8]. In our research we divide algorithms on those which use precomputed data and on those which do not. Based on previous work we decided to use Moller-Trubmore algorithm as a minimal storage, fast non-precomputed type and Swen Woop's Unit Triangle Test as the precomputed one as it requires only 48 bytes per triangle and doesn't need to store vertex list. In this section we describe chosen algorithms along with BVH spatial data structure.

We omit ray-packet algorithms in our work, because the coherence of the rays within the packet is very important since the vector instructions are fully used only if all rays go through the same branch of computation. In situations like physical simulation, collision detection or ray-tracing in scenes, where rays bounces into multiple directions (spherical or bumpmapped surfaces), coherent ray packets break down very quickly to single rays or do not exist at all. Ray-packets have proven [1], [6] to be ineffective in the above mentioned tasks.

2.1 Moller-Trumbore's Algorithm

The algorithm proposed by Moller and Trumbore does not test intersection with the triangle's embedding plane and therefore does not require the plane equation parameters. This is a big advantage mainly in terms of memory consumption especially on the GPU execution performance. The algorithm goes as follows [8]:

1. In a series of transformations the triangle is first translated into the origin and then transformed to a right-aligned unit triangle in the y-z plane, with the ray direction aligned with x. This can be expressed by a linear equation

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} \quad \frac{1}{P \cdot E_1} \begin{pmatrix} Q & E_2 \\ P & T \\ Q & D \end{pmatrix} \tag{1}$$

Where $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$, $T = O - V_0$, $P = D \times E_2$ and $Q = T \times E_1$.

2. This linear equation can now be solved to find the barycentric coordinates of the intersection point (u,v) and its distance t from the ray origin.

2.2 Unit Triangle Algorithm

The so called Unit Triangle intersection algorithm performs ray transformations and consists of two stages [14]. First the ray is transformed, using a triangle specific affine triangle transformation, to a coordinate system, in which the triangle looks like the unit triangle Δunit with the edge points (1, 0, 0), (0, 1, 0) and (0, 0, 0). In the second stage, a simple intersection test of the transformed ray with the unit triangle is done. The affine triangle transformation to a triangle $\Delta = (a, b, c)$ is an affine transformation $T\Delta(x) = m \cdot x + n$ with $m \in MatR(3 \times 3)$ and $n \in R3$ that maps the triangle Δ to the unit triangle Δunit.

2.3 Bounding Volume Hierarchies

Bounding volume hierarchies were successfully implemented on GPU. Thrane and Simonsen [12] in fact compare kd-trees, uniform grids, and bounding volume hierarchies implemented on a GPU (2005-year hardware). They conclude the performance of BVHs is low, however higher than the performance of other two data structures when no ray packets are used. Carr et al. [4] implemented a variant of BVHs in combination with geometry images. Günther et al. [5] use ray packets and yield interactive performance comparable or exceeding CPU-based implementation, but only for primary and shadow rays. Recently, Lauterbach et al. [7] present an algorithm for fast BVH construction on a GPU, where they report performance comparable to kdtrees [16] only for one scene. Torres et al. [13] published an algorithm for stack-less BVH traversal, where the use of stack is replaced by ropes connecting the nodes of a BVH in a sibling order.

3 Implementation

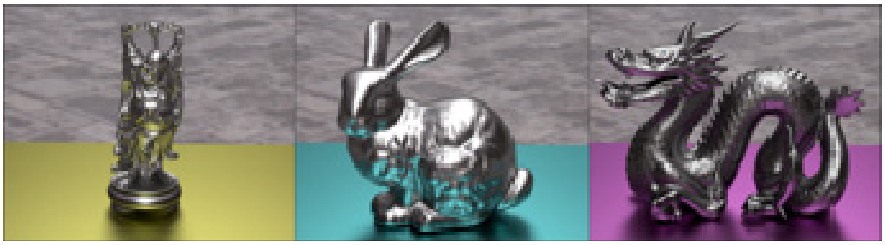


Fig. 1. Testing scenes: Stanford Buddha, Bunny and Dragon

We have implemented a standalone compact program called RenderBro, that does not need the support of other 3rd party libraries along with Autodesk 3DS Max Plugin (both can be obtained at http://renderbro.com). Standalone program is capable of loading 3D scene form OBJ file format along with MTL materials files. 3DS Max plugin is capable to work with any kind of geometry loaded into editor in question. While the data structures are built offline on a CPU, the created data structures and scene geometry are transferred to a GPU and used for ray tracing algorithm entirely on the GPU. This methodology is sufficient to study the efficiency of shooting rays with different intersection algorithms. The traversal and intersection algorithms were implemented using Microsoft DirectX DirectCompute (Compute Shaders). Although this implementation limits target platforms to Microsoft Windows, it gives freedom to choose any GPU vendor to run GPU ray tracing, such as ATI/AMD, NVIDIA, Intel. DirectCompute code can be translated to C++ AMP version, which can be executed on any OS. We designed our solution to support as many hardware as possible, though only DirectX 10 compatible or newer hardware is supported. All shaders were compiled using latest Windows SDK 8.0 D3DCompiler_45.

In Moller-Trumbore setting the geometry of a scene consisting solely of triangles is represented by a list Lv of vertices and list of materials Lm, where each triangle has a list of three indices to Lv plus an index to the Lm. In the Unit Triangle Test each triangle is represented directly by the affine transformation matrix. For our tests we implemented path-trace setting with physically-based importance sampled shading including Phong, Blinn-Phong, Lambertian Diffuse, Oren-Nayar Diffuse, Ashikhmin-Shirley, Glass and Perfect Mirror BSDFs.

The BVHs were built in top-down fashion with surface area heuristics using the centroids of bounding boxes for scene triangles, following the paper by Günther et al. [5]. Each BVH node consists of AABB extents and indices to child nodes. If it's a leaf node, child indices are replaced with triangle offset along with triangle count. Those parameters are packed in 32 byte BVH node structure. As a BVH does not need to store the min and max intersection distances along the ray, only the node address is saved to the stack. Stack does not need to be shortened to only several entries, which minimizes the number of traversal steps. Serialization of write operation may occur as threads record their information. Each BVH node can contain any number of triangles. This hypothetically reduces the number of nodes of a hierarchy along with GPU memory needed and gives space for GPU traversal optimization. Traversal is done with "while-if" method and listed in appendix 8.3.

3.1 GPU Optimized Intersections

Moller-Trumbore and Unit triangle intersection tests are pretty straight-forward to implement and require a little knowledge of GPU architecture (see appendix, section 8.1). Such methods can experience poor register usage in architectures like VLIW which is used in many AMD GPUs.

In this work we introduce a method that strongly benefit from denser GPU register usage. The main idea is to exploit the wide vector width SIMD (Single Instruction Multiple Data) by testing intersection with one ray and four triangles at a time.

Firstly, at precompute time we need to try to fill BVH with four triangles per node, so each node will contain four pointers to triangle list. If this is not possible we would have one triangle per node at worst. Secondly, we need to align GPU scene data according to BVH node structure. So if particular node has only one triangle, we need to place three degenerate triangles in a triangle list to fulfill the alignment. Of course, this will result in a GPU memory footprint by the means of performance. Thirdly, when performing BVH traverse we will be able to linearly fetch four triangles. Here we will need to construct additional vectors, like:

```
// fetch triangle vertices
float3 v01, v11, v21;
float3 v02, v12, v22;
float3 v03, v13, v23;
float3 v04, v14, v24;
...
// construct per-component dir & orig vectors
float4 dir4x = ray.dir.xxxx;
float4 dir4y = ray.dir.yyyy;
float4 dir4z = ray.dir.zzzz;
float4 orig4x = ray.orig.xxxx;
float4 orig4y = ray.orig.yyyy;
float4 orig4z = ray.orig.zzzz;
```

This allows us to compute temporary values on per-component SIMD basis, so all non SIMD operations like scalar addition and multiplication can be performed on each triangle simultaneously. For example, when performing scalar multiplication on GPU we use only one computing block while using new approach we will perform four multiplication operations by the same cost (see appendix 8.2 for full listing):

```
// one triangle per pass
float divisor = dot(pvec, e1);

// four triangles per pass
float4 divisor4 = pvecx*e1x + pvecy*e1y + pvecz*e1z;
```

Table 1. Test scene properties, number of triangles per BVH leaf node, rendering times for different ray-triangle intersection method for Path Trace setting with 500 samples and max ray depth of 16. GPU NVIDIA GeForce GT 240M. Image resolution: 800x600.

Scene	Triangles per BVH node	GPU Scene Size, MB	Moller-Trumbore, seconds (average)	Unit Triangle, seconds (average)	Quad Moller-Trumbore, seconds (average)	Quad Unit Triangle, seconds (average)
Buddha, 100.006 triangles	1	9,35	103,33	103,94	-	-
	2	7,26	95,13	95,44	-	-
	3	6,38	92,72	93,07	-	-
	4	5,89	90,91	91,17	89,67	85,36
	8	5,15	92,57	94,01	-	-
	16	4,73	104,53	108,83	-	-
Bunny, 69.678 triangles	1	6,51	72,18	73,52	-	-
	2	4,87	65,48	66,67	-	-
	3	4,45	62,07	63,94	-	-
	4	3,99	60,98	62,17	57,35	52,84
	8	3,45	62,1	63,29	-	-
	16	3,14	69,67	70,61	-	-

Table 1. (*continued*)

Dragon, 100.012 triangles	1	9,35	210,89	211,67	-	-
	2	7,26	210,71	211,61	-	-
	3	6,35	210,68	211,44	-	-
	4	5,86	210,52	211,21	173,37	168,85
	8	5,11	210,69	211,37	-	-
	16	4,65	210,91	211,66	-	-

Table 2. Test scene properties, number of triangles per BVH leaf node, rendering times for different ray-triangle intersection methods for Path Trace setting with 500 samples and max ray depth of 16. GPU AMD Radeon HD 6870. Image resolution: 800x600.

Scene	Triangles per BVH node	GPU Scene Size, MB	Moller-Trumbore, seconds (average)	Unit Triangle, seconds (average)	Quad Moller-Trumbore, seconds (average)	Quad Unit Triangle, seconds (average)
Buddha, 100.006 triangles	1	9,35	49,27	49,43	-	-
	2	7,26	49,78	50,31	-	-
	3	6,38	51,94	52,18	-	-
	4	5,89	54,15	55,16	34,3	31,37
	8	5,15	66,3	68,64	-	-
	16	4,73	95,22	98,38	-	-
Bunny, 69.678 triangles	1	6,51	33,27	33,79	-	-
	2	4,87	33,57	34,41	-	-
	3	4,45	34,68	35,55	-	-
	4	3,99	38,05	39,06	23,9	20,64
	8	3,45	46,57	47,86	-	-
	16	3,14	60,26	62,56	-	-
Dragon, 100.012 triangles	1	9,35	99,27	99,75	-	-
	2	7,26	99,41	100,96	-	-
	3	6,35	104,6	105,61	-	-
	4	5,86	108,35	111,5	57,99	54,91
	8	5,11	132,57	138,06	-	-
	16	4,65	188,55	198,07	-	-

Table 3. Test scene properties, number of triangles per BVH leaf node, rendering times for different ray-triangle intersection methods for Path Trace setting with 1000 samples and max ray depth of 16. GPU NVIDIA GeForce GTX 560. Image resolution: 800x600.

Scene	Triangles per BVH node	GPU Scene Size, MB	Moller-Trumbore, seconds (average)	Unit Triangle, seconds (average)	Quad Moller-Trumbore, seconds (average)	Quad Unit Triangle, seconds (average)
Buddha	4	5,89	66,59	67,07	63,87	62,13
Bunny	4	3,99	56,82	57,52	53,18	51,17
Dragon	4	5,86	84,76	85,94	80,37	78,64

Table 4. Test scene properties, number of triangles per BVH leaf node, rendering times for different ray-triangle intersection methods for Path Trace setting with 1000 samples and max ray depth of 16. GPU AMD HD7850. Image resolution: 800x600.

Scene	# Triangles per BVH node	GPU Scene Size, MB	Moller-Trumbore, seconds (average)	Unit Triangle, seconds (average)	Quad Moller-Trumbore, seconds (average)	Quad Unit Triangle, seconds (average)
Buddha	4	5,89	38,03	39,41	33,43	31,35
Bunny	4	3,99	36,48	37,97	31,64	29,85
Dragon	4	5,86	65,56	67,09	56,72	54,92

4 Results

In this section we describe the results for measurement on three different scenes. We used scenes from individual objects courtesy of Stanford scene repository. These scenes are frequently used to test the performance of ray tracing and global illumination algorithms. Images were rendered in 800x600 pixels resolution. All performance results in this paper were measured on 4 different GPUs:

1. NVIDIA GeForce GT 240M (2009), 48 CUDA cores on 1210MHz, 1 GByte of memory with bandwidth of 54.4 GB/sec.
2. AMD HD 6870 (2010), 2 TFLOPs, 1120 Stream Processors on 900MHz, 1GByte of memory with bandwidth of 134.4 GB/sec.
3. NVIDIA GeForce GTX 560 (2011), 2.1 TFLOPs, 336 CUDA cores on 1620-1900MHz, 1 GByte of memory with bandwidth of 128 GB/s.
4. AMD HD 7850 (2012), 1.76 TFLOPs, 1024 Stream Processors on 860MHz, 2GByte of memory with bandwidth of 153.6 GB/s.

The static properties of data structures for all three scenes along with average computation time for path-tracing are shown in Tables 1 and 2. Performance results for BVHs build with different count of triangles per leaf node are shown in columns 4 and 5. Those results demonstrate that both the number of triangles per leaf node and the selected intersection method remarkably affect the performance. Results lead us to assumption that 4 triangles per leaf node is the optimal number for BVH traversal.

Moller-Trumbore kernel had proven to be up to 5% faster than Unit Triangle in all tests while Quad Unit Triangle kernel shown to be up to 14% faster than Quad Moller-Trumbore.

Our proposed Quad Unit-Triangle method brings moderate improvements of 5% to 11% on different generations of NVIDIA hardware except for Dragon scene setup on GT 240M GPU where it gain about 18% (tables 1 and 3). The situation is better on VLIW AMD/ATi hardware where it had shown to be up to 2x times faster than Moller-Trumbore (table 2). Furthermore we managed to analyze performance of the latest GPU generation AMD HD 7850 (table 4). As we expected, it showed consistent results in spite of new architecture and showed that Quad Unit-Triangle is approximately 20% faster than Moller-Trumbore method.

Good results are gained for VLIW architecture used in AMD GPUs where more functional units are available and may be scheduled by the compiler or hardware simultaneously. According to description of VLIW architecture, it's possible to perform compute operations along with memory access. We assume that this result is mainly achieved by performing more linear memory access to GPU global memory, avoiding branching by unrolling triangle-intersection loop and taking advantage of denser GPU register usage.

Things are bit different for NVIDIA GPUs like Fermi which internally operate in a SIMD manner by ganging multiple (32) scalar threads together into SIMD warps. If a warp's threads diverge, the warp serially executes both branches, temporarily disabling threads that are not on that path. Thus, ray tracing performance certainly can benefit from loop unrolling and more linear memory access. But the true cause of performance improvement lies much deeper in the GPU architecture and goes beyond the scope of this article.

5 Conclusion and Future Work

We have shown that triangle intersection routines that tend to have good performance when used separately can behave badly when used together with acceleration structures like BVH's due to incoherent memory access, lack of registers, and so on. So we focus our work on finding the robust combination of triangle intersection method and spatial data structure. For now it's the Quad Triangle intersection method used along with BVH.

As future work, the implementation could be extended by several other data structures, such as Kd-Trees, Uniform Grids along with few different ray-triangle intersection methods that can be efficiently mapped to GPUs and are likely to show unexpected results when used together.

Furthermore, shown results can hardly be called ambiguous as they are pretty much view dependent. Dragon scene showed 18% performance improvement on NVIDIA GT 240M for one angle of view. For different angles performance may vary, showing both improvement and deterioration. So, a part of our future work will be devoted to analysis of more complex and dynamic scenes where view dependency is not that great.

Unfortunately, we didn't manage to make in-depth performance study on latest discreet NVIDIA and integrated Intel GPUs. We would like to complete out research by taking this GPUs into account as a part of future work.

References

1. Aila, T., Laine, S.: Understanding the Efficiency of Ray Traversal on GPUs. In: Proceedings of High-Performance Graphics 2009, pp. 145–150. ACM, New York (2009)
2. Arenberg, J.: Ray-Triangle Intersection with Barycentric Coordinates. In: Haines, E. (ed.) Ray Tracing News, November 4, vol. 1(11) (1988)
3. Badouel, F.: An efficient Ray-Polygon intersection, Graphic Gems, pp. 390–393. Academic Press (1990)
4. Carr, N.A., Hoberock, J., Crane, K., Hart, J.C.: Fast GPU ray tracing of dynamic meshes using geometry images. In: GI 2006: Proceedings of Graphics Interface 2006, pp. 203–209. Canadian Information Processing Society, Toronto (2006)
5. Günther, J., Popov, S., Seidel, H.-P., Slusallek, P.: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In: Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007, pp. 113–118 (September 2007)
6. Havel, J., Herout, A.: Yet Faster Ray-Triangle Intersection (Using SSE4). IEEE Transactions on Visualization and Computer Graphics 16(3), 434–438 (2010), doi:10.1109/TVCG.2009.73
7. Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., Manocha, D.: Fast BVH Construction on GPUs. Computer Graphics Forum 28(2), 375–384 (2009) (Proceedings of Eurographics 2007)
8. Möller, T., Trumbore, B.: Fast, minimum storage ray-triangle intersection. Journal on Graphic Tools 2(1), 21–28 (1997)
9. Segura, R.J., Feito, F.R.: Algorithms to Test RayTriangle Intersection. Comparative Study. In: Skala, V. (ed.) WSCG 2001 Conference Proceedings (February 2001)
10. Shevtsov, M., Soupikov, A., Kapustin, A.: Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In: Proceedings of GraphiCon 2007, pp. 33–39 (2007)
11. Snyder, M., Barr, A.H.: Raytracing complex models containing surface tesselations. In: Proceedings of the 14th Annual Conference on Computer Graphics, vol. 21(4), pp. 119–128 (1987)
12. Thrane, N., Simonsen, L.O.: A comparison of acceleration structures for GPU assisted ray tracing. M.Sc. Thesis, University of Aarhus, Denmark (2005)
13. Torres, R., Martin, P.J., Gavilanes, A.: Ray Casting using a Roped BVH with CUDA. In: 25th Spring Conference on Computer Graphics (SCCG 2009), Budmerice, Slovakia, pp. 107–114 (April 2009)
14. Woop, S., Schmittler, J., Slusallek, P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. ACM Transactions Graphics 24(3), 434–444 (2005)

15. Wald, I.: Realtime ray tracing and interactive global illumination. PhD thesis, Saarland University (2004)
16. Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time KD-tree construction on graphics hardware. In: SIGGRAPH Asia 2008: ACM SIGGRAPH Asia 2008 Papers, New York, pp. 1–11 (2008)

A Appendix

A.1 Casual Moller-Trumbore GPU Ray-Triangle Intersection Routine (HLSL code)

```
float intersect(float3 orig, float3 dir, float3 v0,
float3 v1, float3 v2)
{
  float3 e1 = v1 - v0;
  float3 e2 = v2 - v0;

  float3 normal = normalize(cross(e1, e2));
  float b = dot(normal, ray.dir);

  float3 w0 = ray.orig - v0;
  float a = -dot(normal, w0);
  float t = a / b;

  float3 p = ray.orig + t * ray.dir;
  float uu, uv, vv, wu, wv, inverseD;
  uu = dot(e1, e1);
  uv = dot(e1, e2);
  vv = dot(e2, e2);

  float3 w = p - v0;
  wu = dot(w, e1);
  wv = dot(w, e2);
  inverseD = uv * uv - uu * vv;
  inverseD = 1.0f / inverseD;

  float u = (uv * wv - vv * wu) * inverseD;
  if (u < 0.0f || u > 1.0f)
    return -1.0f;

  float v = (uv * wu - uu * wv) * inverseD;
  if (v < 0.0f || (u + v) > 1.0f)
    return -1.0f;

  UV = float2(u,v);
  return t;
}
```

A.2 Quad Moller-Trumbore GPU Triangle-Ray Intersection Routine (HLSL code)

```
float intersect(float3 orig, float3 dir, float3 v01,
float3 v11, float3 v21, float3 v02, float3 v12, float3
v22, float3 v03, float3 v13, float3 v23, float3 v04,
float3 v14, float3 v24)
{
  float3 e11 = v11 - v01;
  float3 e21 = v21 - v01;
  float3 e12 = v12 - v02;
  float3 e22 = v22 - v02;
  float3 e13 = v13 - v03;
  float3 e23 = v23 - v03;
  float3 e14 = v14 – v04;
  float3 e24 = v24 - v04;
  float4 v0x = float4(v01.x, v02.x, v03.x, v04.x);
  float4 v0y = float4(v01.y, v02.y, v03.y, v04.y);
  float4 v0z = float4(v01.z, v02.z, v03.z, v04.z);
  float4 e1x = float4(e11.x, e12.x, e13.x, e14.x);
  float4 e1y = float4(e11.y, e12.y, e13.y, e14.y);
  float4 e1z = float4(e11.z, e12.z, e13.z, e14.z);
  float4 e2x = float4(e21.x, e22.x, e23.x, e24.x);
  float4 e2y = float4(e21.y, e22.y, e23.y, e24.y);
  float4 e2z = float4(e21.z, e22.z, e23.z, e24.z);
  float4 dir4x = ray.dir.xxxx;
  float4 dir4y = ray.dir.yyyy;
  float4 dir4z = ray.dir.zzzz;
  float4 pvecx = dir4y*e2z - dir4z*e2y;
  float4 pvecy = dir4z*e2x - dir4x*e2z;
  float4 pvecz = dir4x*e2y - dir4y*e2x;
  float4 divisor = pvecx*e1x + pvecy*e1y + pvecz*e1z;
  float4 invDivisor = float4(1, 1, 1, 1) / divisor;
  float4 orig4x = ray.orig.xxxx;
  float4 orig4y = ray.orig.yyyy;
  float4 orig4z = ray.orig.zzzz;
  float4 tvecx = orig4x - v0x;
  float4 tvecy = orig4y - v0y;
  float4 tvecz = orig4z - v0z;
  float4 u4;
  u4 = tvecx*pvecx + tvecy*pvecy + tvecz*pvecz;
  u4 = u4 * invDivisor;
  float4 qvecx = tvecy*e1z - tvecz*e1y;
  float4 qvecy = tvecz*e1x - tvecx*e1z;
  float4 qvecz = tvecx*e1y - tvecy*e1x;
  float4 v4;
  v4 = dir4x*qvecx + dir4y*qvecy + dir4z*qvecz;
  v4 = v4 * invDivisor;
  float4 t4;
  t4 = e2x*qvecx + e2y*qvecy + e2z*qvecz;
  t4 = t4 * invDivisor;
  float t = -1.0f;

  if(t4.x < t && t4.x > 0)
    if(u4.x >= 0 && v4.x >= 0 && u4.x + v4.x <= 1)
      t = t4.x;

  if(t4.y < t && t4.y > 0)
    if(u4.y >= 0 && v4.y >= 0 && u4.y + v4.y <= 1)
      t = t4.y;

  if(t4.z < t && t4.z > 0)
    if(u4.z >= 0 && v4.z >= 0 && u4.z + v4.z <= 1)
      t = t4.z;

  if(t4.w < t && t4.w > 0)
    if(u4.w >= 0 && v4.w >= 0 && u4.w + v4.w <= 1)
      t = t4.w;

  return t;
}
```

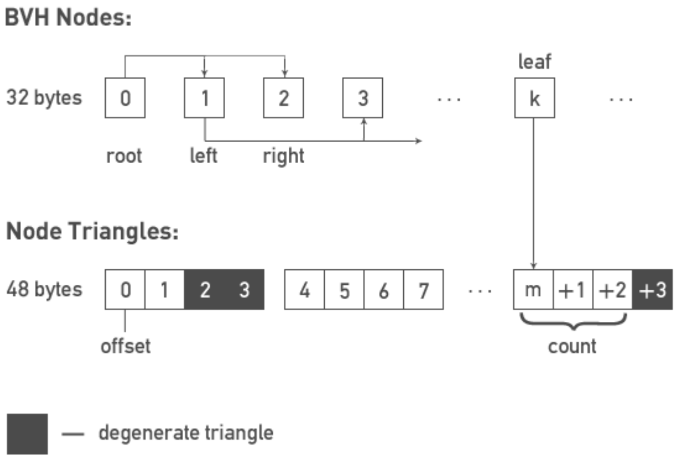
A.3 BVH Traversal Routine (HLSL code)

```
struct BvhCell
{
 float4 vmin; //float3 min + uint children
 float4 vmax; //float3 max + uint count
};
bool RayIntersectScene(Ray ray)
{
 uint stack[64], stackPos = 0, node = 0;
 float t  = FLT_MAX;
 bool intersect = false;
 BvhCell cellLeft, cellRight;
 BvhCell current = GetNode(node);

 while(1)
 {
  uint count  = GetNodeTriangleCount(current);
  if(count > 0)
  { // Leaf Node
   uint offset = GetNodeTriangleOffset(current);
   intersect = RayTrisTest(ray, t, offset, count);
   if(stackPos > 0)
   {
    node = stack[--stackPos];
```

```
      current  = LoadNode(node);
    }
    else return intersected;
  }
  else
  {
    uint  leftNode  = GetLeftChildID(node);
    uint  rightNode = GetRightChildID(node);
    float lMin, rMin;
    cellLeft  = GetNode(leftNode);
    cellRight = GetNode(rightNode);
    bool wantLeft = RayAABBTest(ray, cellLeft, lMin);
    bool wantRight = RayAABBTest(ray, cellRight,
                              rMin);
    if(wantLeft && wantRight)
    {
      bool firstLeft = leftMin < rightMin;
      if(firstLeft)
      {
        current = cellLeft;
        node    = leftNode;
        stack[stackPos++] = rightNode;
      }
      else
      {
        current = cellRight;
        node    = rightNode;
        stack[stackPos++] = leftNode;
      }
    }
    else if(wantRight)
    {
      current = cellRight;
      node    = rightNode;
    }
    else if(wantLeft)
    {
      current = cellLeft;
      node    = leftNode;
    }
    else
    {
      if(stackPos > 0)
      {
        node = stack[--stackPos];
        current = GetNode(node);
      } else return intersected;
      }
    }
  }
 return intersected;
}
```

A.4 BVH Data Layout


BVH Nodes:
32 bytes
0
1
2
3
...
leaf
k
...
root
left
right
Node Triangles:
48 bytes
0
1
2
3
4
5
6
7
...
m
+1
+2
+3
offset
count
— degenerate triangle