Comparison of Triangle Encodings and Bounding Volume Hierarchies for Ray Casts

Jorrit Rouwé – Guerrilla Games – Sept 21st 2019

Contents

1 Abstract

In games we use a lot of ray casts to probe the environment around the player. This environment consists mainly of triangle meshes. Much of the research in ray casting is done in the context of ray tracing (1) where there are 1000s of coherent (nearly parallel) rays to be calculated. Our ray casts are typically not coherent and if they originate from the same location, they're usually in opposite directions. For game play, we typically do in the order of 100 ray casts per frame. We need the ray cast result right away, because game logic often does new ray casts based on the result of the previous ray cast. Getting ray cast results 1 frame later (as is often the case with GPU based solutions) makes the game code complex and reduces the responsiveness of the game. Memory is always tight on a game console, so we want to store our meshes in a memory efficient format.

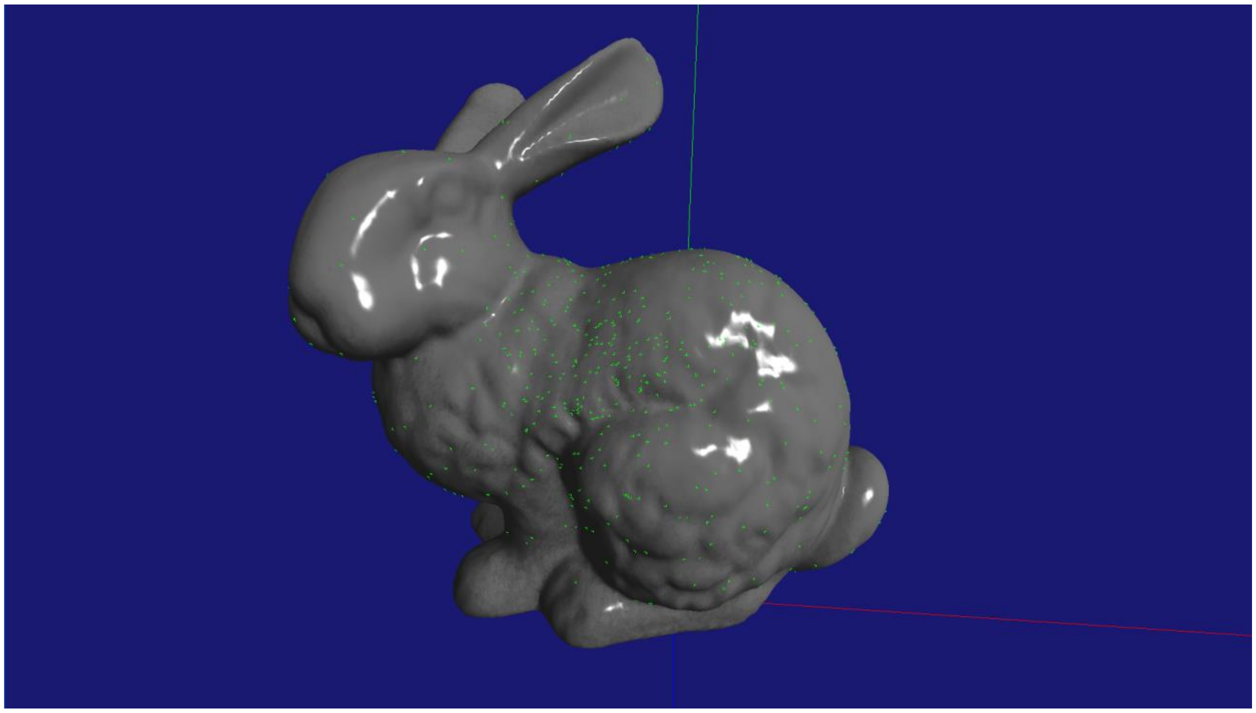
This article compares various triangle encodings and bounding volume hierarchies to select an efficient algorithm for our use case. We will limit ourselves to finding the closest triangle hit. We focus on CPU tests, but at the end of the document we also do some testing on GPU.

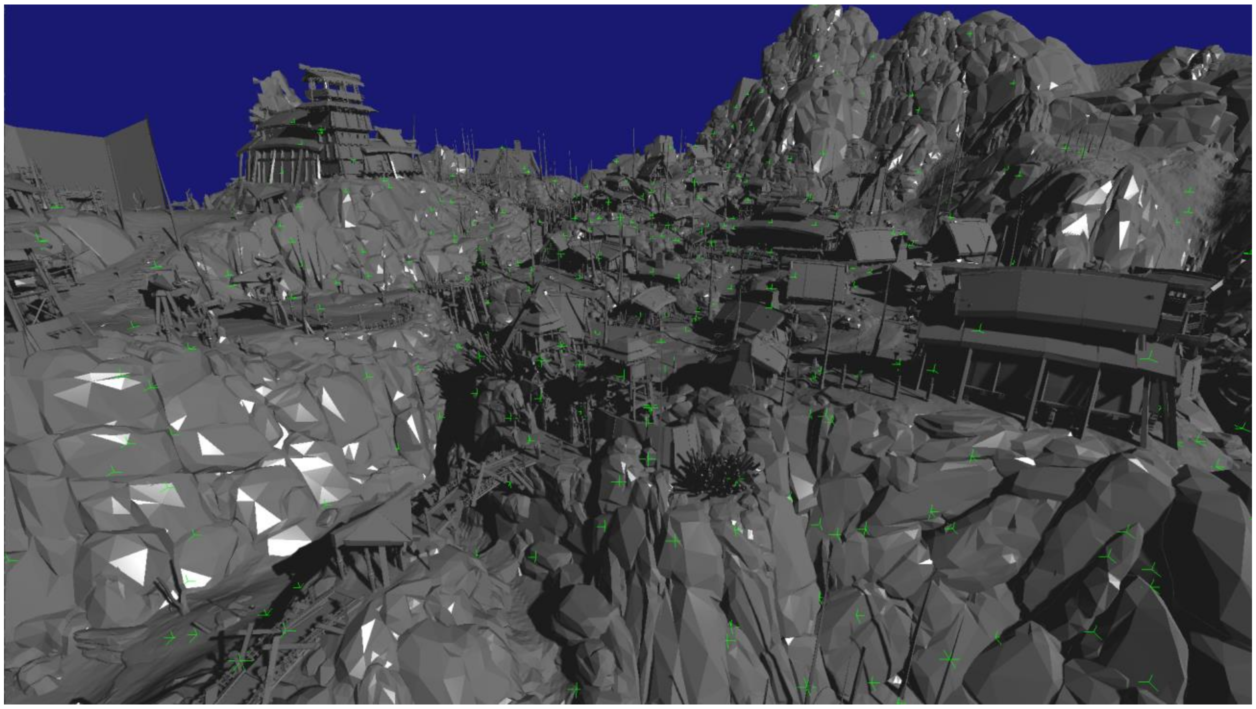
Since, on modern processors memory is much slower than calculations, we expect that it will pay off to do some more calculations to decompress data. We found that, for our use case, this is not completely true. As soon as the bounding volume hierarchy / triangle encoding scheme becomes more complex, the extra calculations do not weigh up against the reduced amount of memory accesses. We found that a simple encoding in combination with SIMD (2) instructions for hit testing gives a good balance between speed and memory requirements. We found that ray casting on GPU is faster, but only when there are many ray casts and the game code can wait (typically a frame) for the answer.

The source code and data needed to run these tests can be found at:
https://github.com/jrouwe/RayCastTest.

2 Introduction

This article will first focus on various triangle encodings, going from very simple to slightly more complex. Next we look at different ways to build a bounding volume hierarchy and which ones result in an efficient tree for our use case. Then we will investigate various bounding volume hierarchies. Finally we will look at the performance / memory trade off between the various solutions. To test our algorithms we use 2 scenes:


The ‘Stanford Bunny’ consisting of 69,451 triangles. Green markers indicate where the random rays hit.


The scene from ‘Mothers Heart’ – Horizon Zero Dawn consisting of 2,264,303 triangles.

Some notations: A triangle mesh has N vertices. We denote the i-th triangle as T_i with $i \epsilon [1, N]$. T_i has 3 vertices: v_{i1}, v_{i2}, v_{i3}. We can split a vertex up into its components: $v_{ij} = (x_{ij}, y_{ij}, z_{ij})$.

We define an axis aligned bounding box around a set of triangles by the minimum and maximum coordinates: B_{min} and B_{max}. Sometimes we use the bounding box of all triangles in the mesh and sometimes we use the triangles of a leaf node of the bounding volume tree. This depends on if it is possible to easily calculate the bounding box of a leaf.

3 Triangle Encodings

Triangles are usually interleaved with the bounding volume tree. A ‘block’ of triangles can be encoded in various ways. We’ve tested the following:

3.1 Float3

This is the most basic storage form. Triangles are stored as an array of floats:

x_{11}, y_{11}, z_{11},
x_{12}, y_{12}, z_{12},
x_{13}, y_{13}, z_{13},

x_{21}, y_{21}, z_{21},
…

3.2 Float3SOA4

Vertices are stored in Structure of Array format of floats. 4 triangles are packed and tested together using SIMD instructions:

x_{11}, …, x_{41}, y_{11}, …, y_{41}, z_{11}, …, z_{41},
x_{12}, …, x_{42}, y_{12}, …, y_{42}, z_{12}, …, z_{42},
x_{13}, …, x_{43}, y_{13}, …, y_{43}, z_{13}, …, z_{43},

x_{51}, …, x_{81}, y_{51}, …, y_{81}, z_{51}, …, z_{81},
…

Padding triangles need to be added to each leaf to ensure there are a multiple of 4 triangles.

3.3 Float3SOA8

This format is like Float3SOA4 except that there are 8 triangles per block. 8 triangles are tested at the same time using AVX2 instructions.

3.4 BitPack

This format packs x, y and z components of a vertex in a uint64 (21 bits per component).

To decompress a vertex we use: $v = (x, y, z) * (B_{max} – B_{min}) / (2^{21} – 1) + B_{min}$.

The bounding box comes from the leaf of the tree where possible.

3.5 BitPackSOA4

This format packs a block of 4 triangles in a Structure of Arrays format with 16 bit per vertex component.

x_{11}, …, x_{41}, y_{11}, …, y_{41}, z_{11}, …, z_{41},
x_{12}, …, x_{42}, y_{12}, …, y_{42}, z_{12}, …, z_{42},
x_{13}, …, x_{43}, y_{13}, …, y_{43}, z_{13}, …, z_{43},

followed by 32-bit float scale and offset values:

s_x, s_y, s_z,
o_x, o_y, o_z

To decompress a vertex we use:

$v_{ij} = (x_{ij}, y_{ij}, z_{ij}) * (s_x, s_y, s_z) + (o_x, o_y, o_z)$.

Where the scale and offset are derived from the bounding box of the 4 triangles:

$s = (B_{max} – B_{min}) / (2^{16} – 1)$
$o = B_{min}$

We do not use the bounding box of the tree because we only have limited precision.

3.6 Indexed16, Indexed32

This is the classic indexed triangle layout. There is a buffer of shared vertices (3 floats per vertex) and 3 16 or 32 bit indices per triangle that point into the vertex buffer.

For a fully connected mesh, each vertex is shared by about 6 triangles resulting in roughly 0.5 vertices per triangle giving a decent compression.

3.7 Indexed16SOA4, Indexed32SOA4

This format has a buffer of shared vertices (3 floats per vertex) and we store the indices of a block of 4 triangles in Structure of Arrays format:

I_{11} … I_{41},
I_{12} … I_{42},
I_{13} … I_{43}

Where I_{ij} is the 16 or 32 bit index of vertex j of triangle i.

We can use SIMD instructions to read the vertex indices for 4 triangles at the same time and we can use an AVX2 gather operation (_mm_i32gather_epi32) to fetch the vertex data. The triangles are then tested 4 at the same time using SIMD math.

3.8 Indexed16BitPackSOA4, Indexed32BitPackSOA4

This is the same as the previous format, however the vertices are stored in an uint64 per vertex like the BitPack format.

Since the vertices are potentially shared between multiple leaf nodes, we need to use the bounding box of all triangles to compress the vertices and we lose a little bit of precision.

3.9 Indexed8BitPackSOA4

Again there is a separate buffer of compressed vertices (uint64 per vertex like the BitPack format) and we use an 8-bit index to select them. Each triangle block contains:

OffsetToVertices (uint32),
TriangleBlock (4 triangles in 12 bytes),
TriangleBlock...

TriangleBlock contains 8-bit indices for 4 triangles:

I_{11} … $I_{41,}$
I_{12} … $I_{42,}$
I_{13} … I_{43}

The compressed vertex for an index can be found by:

<address_of_this_block> + OffsetToVertices + 8 * I_{ij}.

Again this can be done efficiently using the gather instructions and SIMD math to process 4 triangles at a time.

Since we only have 8-bit indices we can have max. 256 / 3 = 85 triangles per block when no vertices are shared.

While creating triangle blocks, we use a greedy algorithm that assumes that none of the vertices will be shared. This means we need space for 3 * <num_triangles_in_block> vertices. This means we can set OffsetToVertices to 256 - 3 * <num_triangles_in_block> before the last compressed vertex. When adding a vertex, we check if it is a duplicate that we can reach using our 8-bit index and if it is, we re-use it, otherwise we add a new one. If the tree is written in a depth first fashion, the re-use is turns out to be pretty good and around 12% of the vertices are duplicated.

3.10 UncompressedStrip, CompressedStrip

This format uses an alternative form of triangle stripping. Each vertex has 3 extra bits of context that indicates if a new strip starts and which of the 2 vertices from the last triangle to re-use, this is encoded as follows:

Each new vertex C in the list forms a new triangle ($v_{(i+1)1}$, $v_{(i+1)2}$, $v_{(i+1)3}$) = (A, B, C), A and B come from the previous triangle (v_{i1}, v_{i2}, v_{i3}).

To start a strip, the first vertex is marked:
STRIPIFY_FLAG_START_STRIP_V1 = 3

The second is marked:
STRIPIFY_FLAG_START_STRIP_V2 = 7

The following flags indicate where A and B come from when a strip is continued (the flags are OR-red together).

Where to take the first vertex for the new triangle from:
STRIPIFY_FLAG_A_IS_V1 = 0
STRIPIFY_FLAG_A_IS_V2 = 1
STRIPIFY_FLAG_A_IS_V3 = 2

Where to take the second vertex for the new triangle from (note that it wraps: V3 + 1 = V1):
STRIPIFY_FLAG_B_IS_A_PLUS_1 = 0
STRIPIFY_FLAG_B_IS_A_PLUS_2 = 4

UncompressedStrip uses 3 floats per vertex and stores the 3 bits in the least significant bit of each of the floats.

CompressedStrip uses an uint64 per vertex like the BitPack encoding but with 20 bits per component so there's room for the extra 3 bits. The bounding box comes from the leaf of the tree where possible.

3.11 Speed and size comparisons

The table below compares speed and size of each of the triangle encodings on an i7 7700HQ testing 1024 long rays against the Stanford Bunny:

Triangle Encoding	Time 1024 Rays (µs)	Bytes Per Triangle
Float3SOA8	91.4	36.0
Float3SOA4	169.4	36.0
BitPackSOA4	240.1	24.0
Indexed8BitPackSOA4	362.3	8.8[1]
Indexed16BitPackSOA4	376.3	10.0
Indexed32BitPackSOA4	379.4	16.0
Indexed16SOA4	400.2	12.0

[1] This number is taken from when the triangle encoder was used in a tree (since the 8 bit indices are not large enough to index all vertices in the model). Triangles are grouped together this way and give a much better vertex re-use. Measured value when inserting vertices in a random order was 17.7.

Indexed32SOA4	401.9	18.0
Float3	671.7	36.0
Indexed16	760.7	12.0
Indexed32	765.6	18.0
UncompressedStrip	826.6	13.1
CompressedStrip	891.9	8.8
BitPack	932.3	24.0

Note that all measurements in this document were repeated several times and the smallest time was taken. We want to compare algorithms, not measure hiccups that Windows might cause due to task switching etc.

The Indexed8BitPackSOA4 encoding offers a good balance between speed and size. The disadvantage of the format is that it uses the bounding box of all triangles for compression, so loses a bit of precision.

4 Generating an AABB Tree

Various Bounding Volume Trees were tested. Each of them was generated from an Axis Aligned Bounding Box (AABB) tree. We will first discuss the generation of that tree.

4.1 Splitting

When generating an AABB Tree, at each level in the tree we must decide how to split the remaining triangles into two batches. We've tried the following methods:

- Longest Axis: Calculate bounding box for triangle centers and split at the center of the box along the longest axis.
- Mean: Calculate mean and standard deviation of triangle centers. Split at the mean along the axis with highest standard deviation.
- Morton: Use Morton codes to sort triangle centers and split when the highest bit that differs flips (3).
- Binning: Use Surface Area Heuristic approach as outlined in (4).
- Fixed Leaf Size: This is an extension over the Binning algorithm. It first groups triangles in multiples of X (usually 4) and then runs the Binning algorithm on those groups. This has the advantage that all leaf nodes get multiples of X triangles so we don't have to add padding triangles for the SOA formats.

For an AABB tree with Float3 encoding and 8 triangles per leaf:

Splitter	Time 1024 Rays (μs)
Binning	0.62
Mean	0.68
Longest Axis	0.73

Morton	0.74
Fixed Leaf Size	0.81

The binning method is clearly the best method, but it is also the most expensive method for generating a tree.

The Fixed Leaf Size method was devised to work better with the SOA encodings, so let's see how that works. For the Float3SOA4 encoding:

Splitter	Time 1024 Rays (µs)
Binning	0.60
Mean	0.65
Longest Axis	0.67
Fixed Leaf Size	0.71
Morton	0.71

It looks like the reduced amount of triangle blocks to test cannot compensate the reduced efficiency of the tree.

4.2 Leaf size

We experimented with the amount of triangles per leaf node: 4, 8, 16. Increasing the amount of triangles will reduce the tree depth and therefore the amount of space required for it.

For an AABB tree with Float3 vertices and the Binning splitter:

Triangles Per Leaf	Time 1024 Rays (µs)
4	0.62
8	0.62
16	0.70

We selected 8 triangles per leaf as a nice tradeoff between speed and memory consumption.

4.3 Node order

The tree and triangles are always stored in a single packed memory block. Some AABB tree encodings support varying the order in which nodes are stored in the tree:

- Depth first, nodes with triangles are added directly behind parent node
- Depth first, nodes with triangles go behind all other nodes
- Breadth first, nodes with triangles are added directly behind parent node
- Breadth first, nodes with triangles go behind all other nodes

For an AABB tree with Float3 encoding, the Binning splitter and 8 vertices per leaf:

Sorting Order	Time 1024 Rays (µs)
Depth First	0.527
Depth First Triangles Last	0.539
Breadth First Triangles Last	0.546
Breadth First	0.547

The difference between these modes is not very big but Depth First wins by a small margin.

5 Bounding Volume Tree Types

Various tree types were tested. They are listed below.

5.1 AABBTree

This is the simplest storage form. Each node in this tree stores:

BoundingBoxMin (3 floats)
BoundingBoxMax (3 floats)
Properties (uint32)

The highest bit of the Properties field indicates if the node contains triangles or if it points to another node. If the node contains triangles, the remaining bits specify how many triangles and the triangle block immediately follows the node. If the node does not contain triangles, the first child node will follow this node immediately and the remaining bits in Properties specify the offset to the second child.

Several different strategies for walking the tree during a Ray Cast were tried, the fastest looked like this:

```
float closest = FLT_MAX;
const Node *stack[stack_size];
const Node *node = <root of tree>;
int top = -1;
for (;;)
{
    // Test if node contains triangles
    if (!node->HasTriangles())
    {
        const Node *left_child = node->GetLeftChild();
        const Node *right_child = node->GetRightChild();

        // Test bounds of left child
        float left_fraction = RayAABox(ray, left_child->GetBounds());
        bool left_intersects = left_fraction < closest;

        // Test bounds of right child
        float right_fraction = RayAABox(ray, right_child->GetBounds());
        bool right_intersects = right_fraction < closest;

        if (left_intersects && right_intersects)
        {
            // Both collide
            if (left_fraction < right_fraction)
            {
                // Left child before right child
                stack[++top] = right_child;
                node = left_child;
            }
            else
            {
                // Right child before left child
                stack[++top] = left_child;
                node = right_child;
            }
            continue;
        }
        else if (left_intersects)
        {
            // Only left collides
            node = left_child;
            continue;
        }
        else if (right_intersects)
        {
            // Only right collides
            node = right_child;
            continue;
        }
    }
    else
    {
        // Node contains triangles, do triangle tests
        ...
    }

    // Fetch next node
    if (top < 0)
        break;
    node = stack[top--];
}
```

5.2 AABBTreeSplitAxis

This tree uses the same layout as the AABBTree except that the Properties field contains 2 extra bits to specify which axis was used as split axis while creating the tree. The nodes are sorted so that the first child always has the lowest value along the split axis. This can be used to decide at each level to which child to recurse to, to have the highest chance of finding the closest hit first:

```
int sign_direction[] = { ray.direction.x > 0? 1 : 0, ray.direction.y > 0? 1 : 0, ray.direction.z
> 0? 1 : 0 };

float closest = FLT_MAX;
const Node *stack[stack_size];
const Node *node = <root of tree>;
int top = -1;
for (;;)
{
    // Test if node is closer than closest hit result
    if (RayAABoxHits(ray, node->GetBounds(), closest))
    {
        // Test if node contains triangles
        if (!node->HasTriangles())
        {
            // Use split axis to determine which child to visit first
            const Node *children[] = { node->GetLeftChild(), node->GetRightChild() };
            uint32 sign = sign_direction[node->GetSplitAxis()];
            const Node *first = children[sign ^ 1];
            const Node *second = children[sign];
            stack[++top] = second;
            node = first;
            continue;
        }
        else
        {
            // Node contains triangles, do triangle tests
            ...
        }
    }

    // Fetch next node
    if (top < 0)
        break;
    node = stack[top--];
}
```

5.3 AABBTreePNS

This tree uses Precomputed Node Sorting (5). It has the same layout as AABBTree but uses 8-bits of the Properties field to store the precomputed bits.

The ray cast function looks like:

```
// Calculate bit index for PNS
uint32 x = signbit(ray.direction.x);
uint32 y = signbit(ray.direction.y);
uint32 z = signbit(ray.direction.z);
uint32 bit_index = z | (y << 1) | (x << 2);

float closest = FLT_MAX;
const Node *stack[stack_size];
const Node *node = <root of tree>;
int top = -1;
for (;;)
{
    // Test if node is closer than closest hit result
    if (RayAABoxHits(ray, node->GetBounds(), closest))
    {
        // Test if node contains triangles
        if (!node->HasTriangles())
        {
            // Use PNS to determine which child to visit first
            const Node *children[] = { node->GetLeftChild(), node->GetRightChild() };
            uint bit = (node->GetPNSBits() >> bit_index) & 1;
            const Node *first = children[bit];
            const Node *second = children[bit ^ 1];
            stack[++top] = second;
            node = first;
            continue;
        }
        else
        {
            // Node contains triangles, do triangle tests
            ...
        }
    }

    // Fetch next node
    if (top < 0)
        break;
    node = stack[top--];
}
```

5.4 AABBTreeCompressed

This format minimizes the storage needed for the tree. We compress the two child bounding boxes relative to its parent bounding box by storing only the values that change. Although the 2 child bounding boxes have 12 values, only 6 are different from parent to child and these 6 values are compressed in 7 bits per component. We store one additional bit per component to indicate if the left or the right child has the changed value:

If BoundsMinX bit is 0:

> child0.min.x = parent.min.x + <compressed min.x> * (parent.max.x - parent.min.x) / (1^7 – 1)
> child1.min.x = parent.min.x

If BoundsMinX bit is 1:

> child0.min.x = parent.min.x
> child1.min.x = parent.min.x + <compressed min.x> * (parent.max.x - parent.min.x) / (1^7 – 1)

Same for X and Z and the maximum value X, Y and Z.

We store a header for the tree:

RootBoundsMin (3 floats)
RootBoundsMax (3 floats)
RootTriangleCount (uint32)

If the mesh has few enough polygons, we store the triangle count in RootTriangleCount and the triangle block immediately follows this header. If RootTriangleCount = 0 the header is followed by the root node of the tree.

A tree node looks like:

NodeBoundsMin (uint32: 3 * 7-bit components + 3 bits + 8 bits triangle count for left child)
NodeBoundsMax (uint32: 3 * 7-bit components + 3 bits + 8 bits triangle count for right child)
RightChildOffset (uint32)

Again the left child follows the node immediately and the right child can be found at RightChildOffset. If left/right child triangle count = 0 it means that the left/right child is a node. If it is not zero it is a triangle block.

The ray cast function looks like:

```
float closest = FLT_MAX;

struct StackEntry
{
    Vec3            mBoundsMin;
    Vec3            mBoundsMax;
    const void *    mPtr;
    uint32          mTriangleCount;
};
StackEntry stack[stack_size];
```

```
Vec3 node_min(header->mRootBoundsMin);
Vec3 node_max(header->mRootBoundsMax);
const void *ptr = <root of tree>;
uint32 triangle_count = header->mRootTriangleCount;

int top = -1;
for (;;)
{
    // Test if node contains triangles
    if (triangle_count == 0)
    {
        const Node *node = reinterpret_cast<const Node *>(ptr);

        // Get child bounding boxes
        Vec3 child_bounds_min[2], child_bounds_max[2];
        node->UnpackBounds(node_min, node_max, child_bounds_min, child_bounds_max);

        // Test bounds of left child
        float left_fraction = RayAABox(ray, child_bounds_min[0], child_bounds_max[0]);
        bool left_intersects = left_fraction < closest;

        // Test bounds of right child
        float right_fraction = RayAABox(ray, child_bounds_min[1], child_bounds_max[1]);
        bool right_intersects = right_fraction < closest;

        if (left_intersects && right_intersects)
        {
            // Both collide
            if (left_fraction < right_fraction)
            {
                // Test left child before right child
                ++top;
                stack[top].mBoundsMin = child_bounds_min[1];
                stack[top].mBoundsMax = child_bounds_max[1];
                stack[top].mPtr = node->GetRightPtr();
                stack[top].mTriangleCount = node->GetRightTriangleCount();
                node_min = child_bounds_min[0];
                node_max = child_bounds_max[0];
                ptr = node->GetLeftPtr();
                triangle_count = node->GetLeftTriangleCount();
            }
            else
            {
                // Test right child before left child
                ++top;
                stack[top].mBoundsMin = child_bounds_min[0];
                stack[top].mBoundsMax = child_bounds_max[0];
                stack[top].mPtr = node->GetLeftPtr();
                stack[top].mTriangleCount = node->GetLeftTriangleCount();
                node_min = child_bounds_min[1];
                node_max = child_bounds_max[1];
                ptr = node->GetRightPtr();
                triangle_count = node->GetRightTriangleCount();
            }
            continue;
        }
        else if (left_intersects)
        {
            // Only left collides
            node_min = child_bounds_min[0];
            node_max = child_bounds_max[0];
            ptr = node->GetLeftPtr();
            triangle_count = node->GetLeftTriangleCount();
            continue;
        }
        else if (right_intersects)
        {
            // Only right collides
            node_min = child_bounds_min[1];
            node_max = child_bounds_max[1];
            ptr = node->GetRightPtr();
            triangle_count = node->GetRightTriangleCount();
            continue;
        }
    }
    else
    {
        // Node contains triangles, do triangle tests
        ...
    }

    // Fetch next node
    if (top < 0)
        break;
    node_min = stack[top].mBoundsMin;
    node_max = stack[top].mBoundsMax;
    ptr = stack[top].mPtr;
    triangle_count = stack[top].mTriangleCount;
    --top;
}
```

5.5 SKDTree

An SKD-Tree (6) stores the split axis and the maximum bounding value for the left half space and the minimum bounding value for the right half space. As such it stores much less data, but, since it cuts the volume only on 1 axis, nodes have a lower chance of being discarded.

We store a uint32 Properties field per node. This contains 2-bits for the split axis (0 = x, 1 = y, 2 = z, 3 = node is a leaf and contains triangles). The remaining bits are either used for the triangle count or the offset to the right node. When a node is not a leaf node, the Properties field will be followed by 2 floats (LeftPlane, RightPlane). The left node immediately follows the planes.

The ray cast function looks like:

```
float closest = FLT_MAX;

struct StackEntry
{
    const Node *   mNode;
    float          mFractionNear;
    float          mFractionFar;
};
StackEntry stack[stack_size];
float fraction_near, fraction_far;
const Node *node = <root of tree>;
RayAABox(ray, root_bounds_min, root_bounds_max, fraction_near, fraction_far);
int top = -1;
for (;;)
{
    if (fraction_near <= fraction_far)
    {
        // Test if node contains triangles
        if (!node->HasTriangles())
        {
            const Node *left_child = node->GetLeftChild();
            const Node *right_child = node->GetRightChild();
            float left_fraction_near = fraction_near, left_fraction_far = fraction_far;
            bool left_intersects = GetHitFraction(ray_origin, ray_direction, node->GetPlaneAxis(),
node->GetLeftPlane(), -1.0f, left_fraction_near, left_fraction_far);

            float right_fraction_near = fraction_near, right_fraction_far = fraction_far;
            bool right_intersects = GetHitFraction(ray_origin, ray_direction, node-
>GetPlaneAxis(), node->GetRightPlane(), 1.0f, right_fraction_near, right_fraction_far);

            if (left_intersects && right_intersects)
            {
                // Both collide
                if (left_fraction_near < right_fraction_near)
                {
                    // Left child before right child
                    ++top;
                    stack[top].mNode = right_child;
                    stack[top].mFractionNear = right_fraction_near;
                    stack[top].mFractionFar = right_fraction_far;
                    node = left_child;
                    fraction_near = left_fraction_near;
                    fraction_far = left_fraction_far;
                }
                else
                {
                    // Right child before left child
                    ++top;
                    stack[top].mNode = left_child;
                    stack[top].mFractionNear = left_fraction_near;
                    stack[top].mFractionFar = left_fraction_far;
                    node = right_child;
                    fraction_near = right_fraction_near;
                    fraction_far = right_fraction_far;
                }
                continue;
            }
            else if (left_intersects)
            {
                // Only left collides
                node = left_child;
                fraction_near = left_fraction_near;
                fraction_far = left_fraction_far;
                continue;
            }
            else if (right_intersects)
            {
                // Only right collides
                node = right_child;
                fraction_near = right_fraction_near;
                fraction_far = right_fraction_far;
                continue;
            }
        }
        else
        {
            // Node contains triangles, do triangle tests
            ...
        }
    }

    // Fetch next node
    if (top < 0)
        break;
    node = stack[top].mNode;
    fraction_near = stack[top].mFractionNear;
    fraction_far = min(closest, stack[top].mFractionFar);
    --top;
}
```

Where:

```
bool GetHitFraction(Vec3 inOrigin, Vec3 inDirection, uint inAxis, float inCoordinate, float
inSide, float &outFractionNear, float &outFractionFar)
{
    float dist_to_plane = inOrigin[inAxis] - inCoordinate;
    float direction = inDirection[inAxis];

    // Check if ray is parallel to plane
    if (abs(direction) < 1.0e-12f)
    {
        // Check if ray is on the right side of the plane
        return inSide * dist_to_plane >= 0.0f;
    }
    else
    {
        // Update fraction
        float intersection = -dist_to_plane / direction;
        if (inSide * direction > 0.0f)
            outFractionNear = max(outFractionNear, intersection);
        else
            outFractionFar = min(outFractionFar, intersection);

        // Return if there is still a possibility for a hit
        return outFractionNear <= outFractionFar;
    }
}
```

5.6 QuadTree

So far all tree formats did not exploit SIMD. In order to improve this, we convert the AABBTree to a QuadTree. We do this by removing every other level of the tree so each node has 4 instead of 2 children. We can now test 4 nodes at the same time.

The root of the tree has:

RootProperties (uint32)

Each node has:

BoundsMinX (4 floats)
BoundsMinY (4 floats)
BoundsMinZ (4 floats)
BoundsMaxX (4 floats)
BoundsMaxY (4 floats)
BoundsMaxZ (4 floats)
NodeProperties (4 uint32s)

The Root/NodeProperties field reserves 5 bits to indicate how many triangles are in the child (0 means the child is a node, anything else means it is a triangle block), the remaining 27 bits are used to encode the offset of the child node or triangle block. If a child node is unused (because we did not have an exact multiple of 4 nodes) we ensure that the bounding box is invalid so the intersection routine will never find a collision.

The ray cast function looks like:

```
float closest = FLT_MAX;
const uint8 *buffer_start = <start of the tree buffer>;
const int stack_size = 128;
uint32 node_stack[stack_size];
float fraction_stack[stack_size];
node_stack[0] = header->mRootProperties;
fraction_stack[0] = 0;
int top = 0;
do
{
    // Test if node contains triangles
    uint32 node_properties = node_stack[top];
    uint32 tri_count = node_properties >> TRIANGLE_COUNT_SHIFT;
    if (tri_count == 0)
    {
        // It's a node
        const Node *node = reinterpret_cast<const Node *>(buffer_start + node_properties);

        // Test ray vs bounds of 4 children
        Vec4 bounds_minx = LoadFloat4(&node->mBoundsMinX);
        Vec4 bounds_miny = LoadFloat4(&node->mBoundsMinY);
        Vec4 bounds_minz = LoadFloat4(&node->mBoundsMinZ);
        Vec4 bounds_maxx = LoadFloat4(&node->mBoundsMaxX);
        Vec4 bounds_maxy = LoadFloat4(&node->mBoundsMaxY);
        Vec4 bounds_maxz = LoadFloat4(&node->mBoundsMaxZ);
        Vec4 fraction = RayAABox4(ray, bounds_minx, bounds_miny, bounds_minz, bounds_maxx,
bounds_maxy, bounds_maxz);

        // Load properties for 4 children
        Int4 properties = LoadInt4(&node->mNodeProperties);

        // Sort fraction so that highest values are first
        // (we want to first process closer hits and we process stack top to bottom)
        // Sorts properties in the same order
        // Uses SIMD sorting network: (7)
        Sort4Reverse(fraction, properties);

        // Count how many results are closer than our current closest value
        Int4 closer = Less(fraction, Replicate(closest));
        int num_results = CountTrues(closer);

        // Shift the results so that only the closer ones remain
        fraction = ShiftLeftComponents(fraction, 4 - num_results);
        properties = ShiftLeftComponents(properties, 4 - num_results);

        // Push them onto the stack
        fraction.StoreFloat4(&fraction_stack[top]);
        properties.StoreInt4(&node_stack[top]);
        top += num_results;
    }
    else
    {
        // We have triangles, now test them
        const void *triangles = buffer_start + (node_properties & OFFSET_MASK);
        ...
    }

    // Fetch next node that could give a closer hit
    do
```

```
        --top;
    while (top >= 0 && fraction_stack[top] >= closest);
}
while (top >= 0);
```

5.7 QuadTreeHalfFloat

The final tested format is the same as QuadTree but stores bounding box values in half float. These can be decoded to float using _mm_cvtph_ps and then the algorithm is the same as for QuadTree.

The node structure becomes:

BoundsMinX (4 half-floats)
BoundsMinY (4 half-floats)
BoundsMinZ (4 half-floats)
BoundsMaxX (4 half-floats)
BoundsMaxY (4 half-floats)
BoundsMaxZ (4 half-floats)
NodeProperties (4 uint32s)

This means a node is exactly 64 bytes, which is cache friendly. If we sort nodes so that the triangle blocks go last (which are not multiples of 64 bytes), we can be sure that all nodes are nicely cache line aligned.

6 Speed and Size Measurements

Testing ray against the tree + triangles for the Stanford Bunny, Float3 triangle encoding, the Binning splitter and 8 triangles per leaf:

Tree Type	Time 1024 Rays (µs)	Tree Size Without Triangles (bytes)
QuadTree	0.53	646340
QuadTreeHalfFloat	0.54	378212
AABBTree	0.62	651476
AABBTreeSplitAxis	0.67	651476
AABBTreePNS	0.68	651476
AABBTreeCompressed	0.94	139596
SKDTree	2.58	186132

We can see that the QuadTree performs the best and that QuadTreeHalfFloat is a good speed/size tradeoff.

Expanding the test with different triangle encodings:

Tree Type	Triangle Encoding	Time 1024 Rays (µs)	Bytes/Triangle
QuadTree	Indexed8BitPackSOA4	0.47	18.3
QuadTree	BitPackSOA4	0.47	38.2
QuadTree	Indexed16BitPackSOA4	0.48	20.5

QuadTreeHalfFloat	Indexed8BitPackSOA4	0.48	14.4
QuadTree	Float3SOA4	0.48	52.8
QuadTreeHalfFloat	BitPackSOA4	0.49	34.3
QuadTreeHalfFloat	Indexed16BitPackSOA4	0.49	16.6
QuadTree	Float3SOA8	0.50	57.3
QuadTreeHalfFloat	Indexed16SOA4	0.50	18.6
QuadTree	UncompressedStrip	0.50	26.4
QuadTree	Indexed16SOA4	0.51	22.5
QuadTreeHalfFloat	Float3SOA8	0.52	53.4
QuadTreeHalfFloat	Float3SOA4	0.53	48.9
QuadTree	Float3	0.53	45.3
QuadTreeHalfFloat	UncompressedStrip	0.53	22.6
QuadTree	Indexed16	0.53	21.4
QuadTree	BitPack	0.53	33.2
QuadTreeHalfFloat	Float3	0.54	41.4
QuadTree	CompressedStrip	0.55	20.6
QuadTreeHalfFloat	Indexed16	0.55	17.5
QuadTreeHalfFloat	BitPack	0.56	29.3
QuadTreeHalfFloat	CompressedStrip	0.56	16.7
AABBTreeCompressed	Indexed16SOA4	0.81	15.3
AABBTreeCompressed	Indexed8BitPackSOA4	0.85	10.8
AABBTreeCompressed	Indexed16BitPackSOA4	0.86	13.3
AABBTreeCompressed	Indexed16	0.89	14.0
AABBTreeCompressed	Float3SOA8	0.89	50.3
AABBTreeCompressed	BitPackSOA4	0.89	31.2
AABBTreeCompressed	Float3SOA4	0.89	45.8
AABBTreeCompressed	CompressedStrip	0.92	13.4
AABBTreeCompressed	UncompressedStrip	0.93	19.1
AABBTreeCompressed	Float3	0.94	38.0
AABBTreeCompressed	BitPack	0.95	26.0

Note that for brevity's sake we left out some tree and triangle encoding types who were neither fast nor small.

We can see that the smallest we can get is by using AABBTreeCompressed in combination with Indexed8BitPackSOA4 (10.8 bytes/triangle), but a very good speed/size tradeoff is to use QuadTreeHalfFloat in combination with Indexed8BitPackSOA4 (14.4 bytes/triangle).

Repeating these tests with the 'Mothers Heart' scene from Horizon Zero Dawn we get similar results:

Tree Type	Time 1024 Rays (µs)
QuadTree	1.51
QuadTreeHalfFloat	1.53

AABBTreeSplitAxis	2.14
AABBTree	2.30
AABBTreeCompressed	2.44
SKDTree	28.22

AABBTreePNS did not have enough bits to store offsets for this mesh.

Tree Type	Triangle Encoding	Time 1024 Rays (µs)	Bytes/Triangle
QuadTree	BitPackSOA4	1.41	39.5
QuadTree	Indexed8BitPackSOA4	1.45	20.2
QuadTree	Float3SOA4Packed	1.45	49.2
QuadTreeHalfFloat	BitPackSOA4	1.45	35.4
QuadTree	CompressedStrip	1.46	24.0
QuadTreeHalfFloat	Indexed8BitPackSOA4	1.46	16.0
QuadTree	Float3SOA4	1.47	54.4
QuadTreeHalfFloat	CompressedStrip	1.48	19.9
QuadTreeHalfFloat	Float3SOA4Packed	1.48	45.1
QuadTree	Float3SOA8	1.49	62.0
QuadTree	UncompressedStrip	1.50	31.2
QuadTreeHalfFloat	BitPack	1.51	29.8
QuadTree	Float3	1.51	46.0
QuadTreeHalfFloat	UncompressedStrip	1.51	27.0
QuadTreeHalfFloat	Float3SOA8	1.52	57.9
QuadTreeHalfFloat	Float3SOA4	1.53	50.3
QuadTreeHalfFloat	Float3	1.53	41.9
QuadTree	BitPack	1.55	33.9
QuadTree	Indexed32BitPackSOA4	1.55	28.8
QuadTreeHalfFloat	Indexed32BitPackSOA4	1.56	24.6
QuadTree	Indexed32SOA4	1.57	30.9
QuadTreeHalfFloat	Indexed32SOA4	1.62	26.7
QuadTree	Indexed32	1.62	28.3
QuadTreeHalfFloat	Indexed32	1.67	24.2
AABBTree	Indexed8BitPackSOA4	2.00	20.2
AABBTree	UncompressedStrip	2.05	31.3
AABBTree	CompressedStrip	2.07	24.3
AABBTree	BitPack	2.12	34.2
AABBTree	Float3SOA8	2.13	62.5
AABBTree	Indexed32SOA4	2.14	31.4
AABBTree	BitPackSOA4	2.15	40.0
AABBTree	Indexed32BitPackSOA4	2.18	29.3
AABBTreeCompressed	Indexed8BitPackSOA4	2.20	12.2
AABBTree	Float3SOA4	2.27	55.0
AABBTree	Indexed32	2.27	28.5

AABBTreeCompressed	BitPackSOA4	2.29	32.0
AABBTree	Float3	2.30	46.2
AABBTreeCompressed	UncompressedStrip	2.33	23.3
AABBTreeCompressed	CompressedStrip	2.36	16.3
AABBTreeCompressed	Float3SOA8	2.38	54.5
AABBTreeCompressed	Float3SOA4	2.38	47.0
AABBTreeCompressed	BitPack	2.40	26.2
AABBTreeCompressed	Indexed32BitPackSOA4	2.41	21.3
AABBTreeCompressed	Indexed32SOA4	2.44	23.4
AABBTreeCompressed	Float3	2.44	38.2
AABBTreeCompressed	Indexed32	2.50	20.5

7 Ray Casting on GPU

We also tested a number of ray casting algorithms on GPU. These were tested on a NVidia GTX 1050 using DirectX 11. Three different strategies were tested:

- BruteForce: Just test all triangles. We tested 2 triangle encodings: Float3SOA128 and BitPackSOA128. These look like the CPU versions but pack 128 triangles at a time so the GPU can do a coalesced read.
- AABBList: Group triangles into batches of 256 triangles and calculate an AABB per batch. In the first compute shader test all AABBs versus the rays and write (ray index, aabb index) per hit into an 'append and consume' buffer. A second compute shader then takes these results and performs collision checks against individual triangles. During construction, the triangles were grouped by a greedy algorithm that would find the triangle whose centroid x coordinate was lowest and group it with its closest triangles.
- AABBTree: This uses the same algorithm and data layout as the CPU version. We tested 2 triangle encodings: Float3 and UncompressedStrip.
- SKDTree: Again, same as the CPU version. We tested only Float3 triangle encoding.

For the GPU, less triangles per leaf seem better (although the results seem not very consistent). This is for the AABBTree with Float3 encoding and the Binning splitting algorithm:

Triangles Per Leaf	Time 1024 Rays (µs)
4	0.42
16	0.44
8	0.55

Comparing algorithms for the Stanford Bunny with 4 triangles per leaf:

Tree Type	Triangle Encoding	Time 1024 Rays (µs)	Bytes/Triangle
AABBTree	Float3	0.34	53.4

AABBTree	UncompressedStrip	0.41	37.1
QuadTree	Indexed8BitPackSOA4	0.47	18.3
AABBList	Float3SOA256	1.06	36.2
SKDTree	Float3	1.15	41.0
BruteForce	BitPackSOA128	35.01	24
BruteForce	Float3SOA128	41.67	36

The fastest CPU version is added to the table in yellow. You can see that the GPU ray cast is faster than the CPU version, but the overhead of DirectX 11 is very significant, so the GPU version is not really suitable to do small amounts of ray casts.

8 Conclusion

We've compared various algorithms and found that the QuadTreeHalfFloat with Indexed8BitPackSOA4 gives a good balance between speed and memory. Since most of the ray casts happen in small batches and on the CPU, the CPU implementation is currently the best way to go.

There are still many things that were left untested, especially in the area of GPU ray casting. Some ideas for future investigation:

- Investigate better estimation of node and leaf costs in the Binning algorithm to improve the quality of the tree (currently we assume both costs are the same).
- Test different types of tree / vertex compression with the GPU algorithm.
- Split up the GPU tree walk into a shader that walks the tree and a shader that tests triangles (similar to the AABBList algorithm).
- Implement a ZeroByteBVH (8)
- Look at other compression schemes, e.g. (9)

9 Bibliography

1. **Wiki.** Ray Tracing. *Wiki.* [Online] https://en.wikipedia.org/wiki/Ray_tracing_(graphics).
2. —. SIMD Instructions. *Wiki.* [Online] https://en.wikipedia.org/wiki/SIMD.
3. **Karras, Tero.** Thinking Parallel, Part III: Tree Construction on the GPU. [Online] http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/.
4. **Gunther, Johannes.** Realtime Ray Tracing on GPU with BVH-based Packet Traversal. [Online] http://www-sop.inria.fr/members/Stefan.Popov/media/BVHPacketsOnGPU_IRT07.pdf.
5. **Terdiman, Pierre.** Precomputed node sorting in BV-tree traversal. [Online] http://www.codercorner.com/blog/?p=734.
6. **Wiki.** Bounding Interval Hierarchy. [Online] https://en.wikipedia.org/wiki/Bounding_interval_hierarchy.
7. —. Sorting Network. [Online] http://en.wikipedia.org/wiki/Sorting_network.
8. **Terdiman, Pierre.** Zero-byte AABB-trees. [Online] http://www.codercorner.com/ZeroByteBVH.pdf.

9. **Segovia, Benjamin.** Memory Efficient Ray Tracing with Hierarchical Mesh Quantization. [Online] https://pdfs.semanticscholar.org/c329/a3456cb8ed426472d281bea42320a37a711a.pdf.