GDC

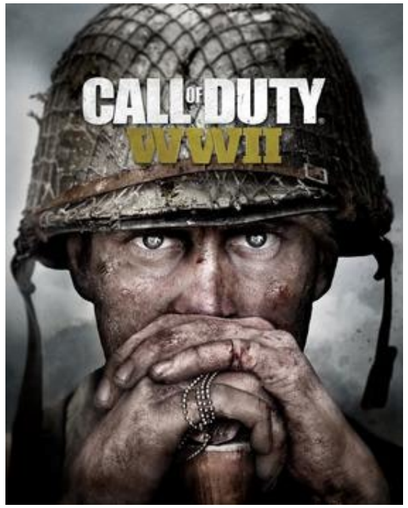
Model Collision: Balancing Speed and Size in Black Ops Cold War

Andrew Shurney
Associate Principal Software Engineer
Activision: Central Tech

© 2021 Activision Publishing, Inc.

About Me

- Andrew Shurney
- Associate Principal Software Engineer
- Activision: Central Tech since 2016
- Call of Duty



- Pipeline/systems work + occasional collision

Outline

- Impetus
- Triangle Formats
- AABB Tree Construction and Structure
- Fine Tuning Optimization

Problem Overview

- Really big worlds eat a lot of data
  - Collision is only allocated the leftovers
- Need to minimize in-memory footprint
- No headroom for a size/speed trade-off
  - We want both.
- Collision database mostly optimized
  - Down to 280MB, but still too big

Problem Scope

- Low hanging fruit: model bullet meshes
- Usually less than 200 tris
  - Some outliers w/ up to 6000
- Mostly small objects
  - Some up to 8192 uints^3
- Only triangle meshes
- Only ray casts

Initial State

- Each triangle stored as three planes
  - Very efficient ray intersection
  - 48 bytes per triangle
- Models with ≥256 triangles use AABB tree
  - 8 children per node
  - 24 bytes per AABB
- Scalar comparison function

Triangle Storage

- Vertex based
  - Indexed/Non-indexed
  - Build planes to test against
- Matrix
  - Just store the planes
  - Very cheap tests

Matrix-Based Triangles

- Store transformation to barycentric coords
  - 9 floats + 2 bits per triangle[1]
  - Chosen by longest normal axis
  - Very cheap tests

$$\begin{bmatrix} 0 & \frac{\overrightarrow{E_{2z}}}{\vec{n}_x} & -\frac{\overrightarrow{E_{2y}}}{\vec{n}_x} & \frac{(\overrightarrow{v_3}\times\overrightarrow{v_1})_x}{\vec{n}_x} \\ 0 & -\frac{\overrightarrow{E_{1z}}}{\vec{n}_x} & \frac{\overrightarrow{E_{1y}}}{\vec{n}_x} & -\frac{(\overrightarrow{v_2}\times\overrightarrow{v_1})_x}{\vec{n}_x} \\ 1 & \frac{\vec{n}_y}{\vec{n}_x} & \frac{\vec{n}_z}{\vec{n}_x} & -\frac{\vec{n}\bullet\overrightarrow{v_1}}{\vec{n}_x} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -\frac{\overrightarrow{E_{2z}}}{\vec{n}_y} & 0 & \frac{\overrightarrow{E_{2x}}}{\vec{n}_y} & \frac{(\overrightarrow{v_3}\times\overrightarrow{v_1})_y}{\vec{n}_y} \\ \frac{\overrightarrow{E_{1z}}}{\vec{n}_y} & 0 & -\frac{\overrightarrow{E_{1x}}}{\vec{n}_y} & -\frac{(\overrightarrow{v_2}\times\overrightarrow{v_1})_y}{\vec{n}_y} \\ \frac{\vec{n}_x}{\vec{n}_y} & 1 & \frac{\vec{n}_z}{\vec{n}_y} & -\frac{\vec{n}\bullet\overrightarrow{v_1}}{\vec{n}_y} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\overrightarrow{E_{2y}}}{\vec{n}_z} & -\frac{\overrightarrow{E_{2x}}}{\vec{n}_z} & 0 & \frac{(\overrightarrow{v_3}\times\overrightarrow{v_1})_z}{\vec{n}_z} \\ -\frac{\overrightarrow{E_{1y}}}{\vec{n}_z} & \frac{\overrightarrow{E_{1x}}}{\vec{n}_z} & 0 & -\frac{(\overrightarrow{v_2}\times\overrightarrow{v_1})_z}{\vec{n}_z} \\ \frac{\vec{n}_x}{\vec{n}_z} & \frac{\vec{n}_y}{\vec{n}_z} & 1 & -\frac{\vec{n}\bullet\overrightarrow{v_1}}{\vec{n}_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

1: Baldwin, Doug, and Michael Webber. "Fast Ray-Triangle Intersection by Coordinate Transformation." *Journal of Computer Graphics Techniques (JCGT)*, vol. 5, no. 3, 2016, pp. 39–49., doi:April 16, 2021. http://jcgt.org/published/0005/03/03/paper.pdf

Vertex-Based Triangles

- Store actual 3D points
- Can index them to save space
- Need to generate planes for tests
  - Relatively expensive for robust detection (no cracks)[2]

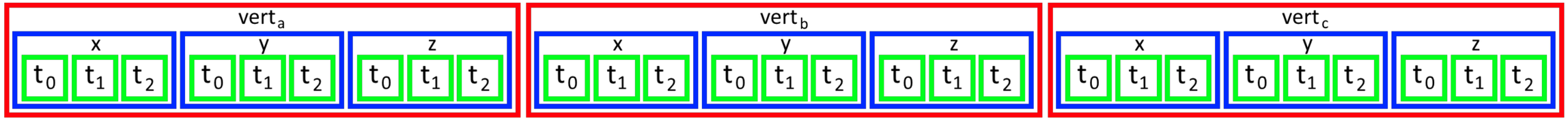
2: Hammon, Earl. "Extreme SIMD: Optimized Collision Detection in 'Titanfall'." *Game Developers Conference*, 2018, www.gdcvault.com/play/1025126/Extreme-SIMD-Optimized-Collision-Detection.

Napkin Math

- Current:
  - Tri-size: $4_{float/side} \times 3_{sides} \times 4_{bytes\ per\ float} \times 8_{bits\ per\ byte} = 384_{bits/tri}$
  - Test: 4_{dots}
- Baldwin Matricies:
  - Tri-size: $9_{floats} + 2_{bits} = 290_{bits/tri}$, $\frac{290}{384} \approx 75\%$
  - Test upper bound: $4_{dots}/SIMD_{4,8} = (400, 800)\%$ speedup
- Vertex Triangles:
  - Tri-size: $3_{verts} \times 3_{vals/vert} \times (\leq 16)_{bits/val} = \leq 144_{bits/tri} \approx 37\%$
  - Test: $\frac{expensive}{4,8} \leq 4_{dots}???$

Initial Experiments

- XYZ F32 Tris
  - Coordinates swizzled for generic SIMD



- Indexed F32 Tris
  - U16 or U8 indices, as model requires
  - 7 shuffles required per 4 triangles

Swizzled 4 Triangle Load

```
MultiTri BuildTri(const float* vertData, uint triCount) {
    const float* const x = vertData;
    const float* const y = x + triCount;
    const float* const z = y + triCount;
    MultiTri tri;
    for (uint vI = 0; vI < ARRAY_COUNT(tri.verts); ++vI) {
        tri.verts[vI].x = _mm_loadu_ps(x + (triCount * 3 * vI));
        tri.verts[vI].y = _mm_loadu_ps(y + (triCount * 3 * vI));
        tri.verts[vI].z = _mm_loadu_ps(z + (triCount * 3 * vI));
    }
    return tri;
}
```

Indexed 4 Triangle Load

```
MultiTri BuildTri(const vec3* verts, const T* inds, uint triCount, uint triWidth) {
    MultiTri tri;
    for (uint vIndex = 0; vIndex < 3; ++vIndex) {
        __m128 x0y0z0ww = _mm_setzero_ps();
        __m128 x1y1z1ww = _mm_setzero_ps();
        __m128 x2y2z2ww = _mm_setzero_ps();
        __m128 x3y3z3ww = _mm_setzero_ps();
        switch (triWidth) {
            case 4: x3y3z3ww = _mm_loadu_ps(verts[inds[3 * 3 + vIndex]]);
            case 3: x2y2z2ww = _mm_loadu_ps(verts[inds[2 * 3 + vIndex]]);
            case 2: x1y1z1ww = _mm_loadu_ps(verts[inds[1 * 3 + vIndex]]);
            case 1: x0y0z0ww = _mm_loadu_ps(verts[inds[0 * 3 + vIndex]]);
        }
        __m128 x0y0x1y1 = _mm_shuffle_ps(x0y0z0ww, x1y1z1ww, _MM_SHUFFLE(1, 0, 1, 0));
        __m128 x2y2x3y3 = _mm_shuffle_ps(x2y2z2ww, x3y3z3ww, _MM_SHUFFLE(1, 0, 1, 0));
        __m128 z0wwz1ww = _mm_shuffle_ps(x0y0z0ww, x1y1z1ww, _MM_SHUFFLE(3, 2, 3, 2));
        __m128 z2wwz3ww = _mm_shuffle_ps(x2y2z2ww, x3y3z3ww, _MM_SHUFFLE(3, 2, 3, 2));
        tri.verts[vIndex].x = _mm_shuffle_ps(x0y0x1y1, x2y2x3y3, _MM_SHUFFLE(2, 0, 2, 0));
        tri.verts[vIndex].y = _mm_shuffle_ps(x0y0x1y1, x2y2x3y3, _MM_SHUFFLE(3, 1, 3, 1));
        tri.verts[vIndex].z = _mm_shuffle_ps(z0wwz1ww, z2wwz3ww, _MM_SHUFFLE(2, 0, 2, 0));
    }
    return out;
}
```

Initial Results

- XYZ F32 Tris
  - ~23% size reduction
  - ~65% speedup
- Indexed F32 Tris
  - ~67% size reduction
  - ~18% speedup

Simple Vert Compressing

- XYZ 16-bit fixed point
  - Unpack and scale to reach F32
  - Known to support all models
- Adaptive 16 or 8 bit fixed point
  - Use model’s longest axis
  - Can store verts in 5.3 or 13.3 bit fixed
- Axially adaptive 16 or 8 bit fixed point
  - Determines vert width per axis

Unpacking Fixed Point in SIMD

```
const __m128 SCALE = _mm_set1_ps(0.125f); //13.3 or 5.3. 1/(1<<3) = 0.125
__m128i packedVX;
if constexpr (sizeof(T) == 2)
    packedVX = _mm_loadu_si64(x + (triCount * 3 * vIndex));
else {
    packedVX = _mm_loadu_si32(x + (triCount * 3 * vIndex));
    packedVX = _mm_unpacklo_epi8(packedVX, _mm_setzero_si128());
}
packedVX = _mm_unpacklo_epi16(packedVX, _mm_setzero_si128());
outV[vIndex].x = _mm_mul_ps(_mm_cvtepi32_ps(packedVX), SCALE);
```

Simple Compression Results

- XYZ 13.3-bit fixed point
  - ~56% size reduction
  - ~41% speedup
- Adaptive 13.3 or 5.3 bit fixed point
  - ~60% size reduction
  - ~35% speedup
- Axially adaptive 13.3 or 5.3 bit fixed point
  - ~62% size reduction
  - ~5% speedup

Need Something New

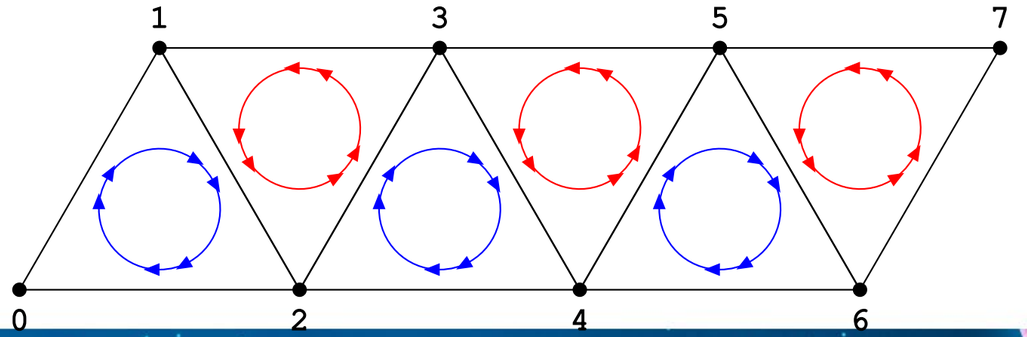
- Indexed results approach size targets
  - Below speed targets
- Flat vert lists hit speed targets
  - Until I push them toward size targets

Or Maybe Something Old

Triangle strips

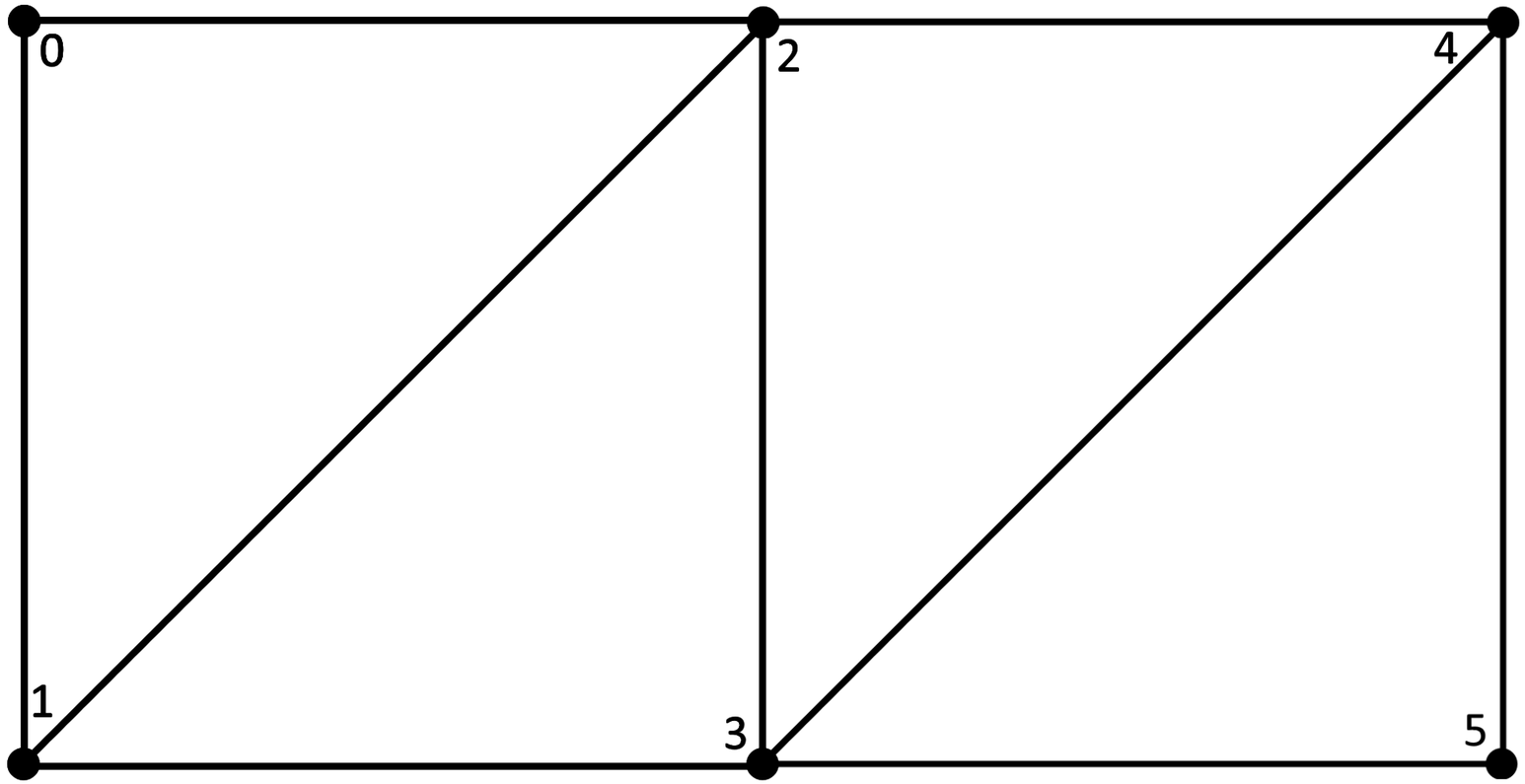
Triangle Strip Properties

- Vertex count N+2 instead of 3N
- Specify 2 verts, then each new is new tri
- Each triangle has alternating winding
- Each triangle must share leading edge
- Plenty of papers/libraries to generate strips from meshes



Expanding a Triangle Strip

- Need to expand to flat verts for intersection



	Vert A	Vert B	Vert C
Tri 0	0	1	2
Tri 1	2	1	3
Tri 2	2	3	4
Tri 3	4	3	5

Expanding a Triangle Strip

- Expand and swizzle 0, 1, 2, 3, 4, 5

	Vert A	Vert B	Vert C
Tri 0	0	1	2
Tri 1	2	1	3
Tri 2	2	3	4
Tri 3	4	3	5

	Tri 0	Tri 1	Tri 2	Tri 3
Vert A	0	2	2	4
Vert B	1	1	3	3
Vert C	2	3	4	5

Expanding a Triangle Strip: SSE

```
__m128 xL = _mm_loadu_ps(firstV + vCount*0);                    // xL: 0,1,2,3
__m128 xH = _mm_loadu_ps(firstV + 2 + vCount*0);                // xH: 2,3,4,5
outV[2].x = xH;                                                 // v2:                    2,3,4,5
outV[1].x = _mm_shuffle_ps(xL, xL, _MM_SHUFFLE(3,3,1,1)); // v1: 0,1,2,3:0,1,2,3->1,1,3,3
outV[0].x = _mm_shuffle_ps(xL, xH, _MM_SHUFFLE(2,0,2,0)); // v0: 0,1,2,3:2,3,4,5->0,2,2,4
```

	Tri 0	Tri 1	Tri 2	Tri 3
Vert C	2	3	4	5
Vert B	1	1	3	3
Vert A	0	2	2	4

Expanding a Triangle Strip: AVX

```
__m256 xL = _mm256_loadu_ps(firstV + vCount*0);          //xL: 0,1,2,3,4,5,6,7
__m256 xH = _mm256_loadu_ps(firstV + 2 + vCount*0);      //xH: 2,3,4,5,6,7,8,9
outV[2].x = xH;                                          //v2: 2,3,4,5,6,7,8,9
xH = _mm256_permute_ps(xH, _MM_SHUFFLE(2,0,0,0));        //xH: 2,2,2,4,6,6,6,8
outV[1].x = _mm256_permute_ps(xL, _MM_SHUFFLE(3,3,1,1)); //v1: 1,1,3,3,5,5,7,7
outV[0].x = _mm256_blend_ps(xH, xL, 0b00010001);         //v0: 0,2,2,4,4,6,6,8
```

	Tri 0	Tri 1	Tri 2	Tri 3	Tri 4	Tri 5	Tri 6	Tri 7
Vert C	2	3	4	5	6	7	8	9
Vert B	1	1	3	3	5	5	7	7
Vert A	0	2	2	4	4	6	6	8

Is It Viable?

- Tri strips F32
  - ~58% size reduction
  - ~69% speedup

- Faster than flat tris!
- Close to indexed size w/o compression

Is It Still Compressible?

- Same strategy, 13.3 or 5.3 fixed
- Apply per tri strip/list patch
  - Should avoid axial's slowdowns
  - Better compression than per model

Fixed to Float Tri Stirp: SSE

```
__m128i xVals;
const __m128i ZERO = _mm_setzero_si128();
const __m128i SCALE = _mm_set1_ps(0.125f);
if constexpr (sizeof(T) == 2){
    xVals = _mm_loadu_si128((__m128i*)vals);      // 0,1,2,3,4,5,a,b
} else {
    xVals = _mm_loadu_si64(vals);                 // 0,1,2,3,4,5,a,b,0,0,0,0,0,0,0,0
    xVals = _mm_unpacklo_epi8(xVals, ZERO);       // 0,1,2,3,4,5,a,b
}
__m128i xHi16 = _mm_bsrli_si128(xVals, 4);        // 2,3,4,5,a,b,0,0
__m128i xLo16 = _mm_unpacklo_epi16(xVals, ZERO); // 0,1,2,3
xHi16 = _mm_unpacklo_epi16(xHi16, ZERO);          // 2,3,4,5
*outXLo = _mm_mul_ps(_mm_cvtepi32_ps(xLo16), SCALE);
*outXHi = _mm_mul_ps(_mm_cvtepi32_ps(xHi16), SCALE);
```

Fixed to Float Tri Strip: AVX2

```
const __m256 SCALE = _mm256_set1_ps(0.125f);
__m256i xLo16, xHi16;
if constexpr (sizeof(T) == 2) {
    xLo16 = _mm256_loadu_si256((__m256i*)vals);                 // 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
    xLo16 = _mm256_permute4x64_epi64(xLo16, 0b10010100);        // 0,1,2,3,4,5,6,7,4,5,6,7,8,9,a,b
} else {
    __m128i xLo8 = _mm_loadu_si128((__m128i*)vals);             // 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
    xLo16 = _mm256_castsi128_si256(pl128);                      // 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,0...
    // -> 0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7,4,5,6,7,8,9,a,b,4,5,6,7,8,9,a,b
    xLo16 = _mm256_permutevar8x32_epi32(xLo16, _mm256_set_epi32(2, 1, 2, 1, 1, 0, 1, 0));
    xLo16 = _mm256_unpacklo_epi8(xLo16, _mm256_setzero_si256()); // 0,1,2,3,4,5,6,7,4,5,6,7,8,9,a,b
}
xHi16 = _mm256_shuffle_epi32(xLo16, _MM_SHUFFLE(0, 0, 2, 1));     // 2,3,4,5,0,1,0,1,6,7,8,9,4,5,4,5
xLo16 = _mm256_unpacklo_epi16(xLo16, _mm256_setzero_si256());     // 0,1,2,3,4,5,6,7
xHi16 = _mm256_unpacklo_epi16(xHi16, _mm256_setzero_si256());     // 2,3,4,5,6,7,8,9
*outXLo = _mm256_mul_ps(_mm256_cvtepi32_ps(xLo16), SCALE);
*outXHi = _mm256_mul_ps(_mm256_cvtepi32_ps(xHi16), SCALE);
```

Compressed Tri Strip Results

- Fixed Point Tri Strips
  - ~75% size reduction
  - ~68% speedup

- We have a winner

Next Chunk: The Trees

- Trees now ~39% of our data
  - Up from ~9%
- Should be plenty of room to optimize them

Model BVH Tree Structure

- Current tree
  - 32-bit index per triangle/child node offset
  - Only models >= 256 triangles
  - F32 AABB bounds
  - No bounds for leaf nodes
  - 32-bit node structure
  - Each node is up to 8 wide

Run-Length Leaf Compression

- Dealing with index trees
- Sort data to tree order
  - More cache friendly
- Leaf data now sequential
- Only need store index of first child
- `node->leafs[c]` → `node->firstLeaf + c`
- Free ~80% reduction in leaf data

AABB Compression

- Scale unpacking 8 bit fixed cheap
- AABBs only need to contain
  - `nextafterf()` is your friend
- Store all bounds as 8 bit [0,1)
- Slightly looser bounds-75% data reduction
  - More cache efficient trees

Packing the Node Structure

- Node was 32bits overhead
  - 1bit is leaf, 4bit child count, 27bit mask
- Mask not used
- Child count can be 3bits with a 1 bias
- Node now 8bits overhead
  - 2bits u8,u16,u32, or u64 offsets
  - 2bits u8,u16,u32, or u64 offset scale

Node Structure Comparison

```
struct OldNode {
  union {
    uint32_t this;        // &this == &OldNode
    struct {
      uint32_t leaf : 1;  // If 1, always 1 offset
      uint32_t cc : 4;    // childCount
      uint32_t mask : 27;
      uint32_t cOffs[cc]; // child = &this + cOffs[ci]
      float bounds[6][cc];// minX,maxX,...,maxZ
    };
  };
};
```

Old Node

- Node size: 228bytes
- Leaf size: 36bytes ← No Bounds

New Node

- Node size: [57,81]bytes
- Leaf size: [50,51]bytes

```
struct NewNode {
  static const uint8_t ccBias = 1;
  union {
    uint8_t thisNode8;   // Choose via offsScale
    uint16_t thisNode16; // &thisNode == &NewNode
    uint32_t thisNode32;
    uint64_t thisNode64;
    struct {
      uint8_t leaf : 1;       // If 1, always 1 offset
      uint8_t cc : 3;         // childCount
      uint8_t offsScale : 2; // 0=8,3=64 for thisNode
      uint8_t offsWidth : 2; // 0=8,3=64 for cOffs
      union {
        uint8_t cOffs8[cc+ccBias];  // Choose via offsWidth
        uint16_t cOffs16[cc+ccBias];// c = &this+cOffs[ci]
        uint32_t cOffs32[cc+ccBias];// Always 1 for leaf
        uint64_t cOffs64[cc+ccBias];
      };
      uint8_t bounds[6][cc+ccBias]; // [0,1)
    };
  };
};
```

Final BVH Structure

- Adaptive width child node/leaf offsets
- Run-length compressed leaf indices
- 8bit fixed node bounds (including leaves)
- 8bit node structure
- Each node is up to 8 wide
- Used for all models

New Trees, New Numbers

- Fixed Point Tri Strips w/ Tweaked Trees
  - ~78% size reduction
  - ~99% speedup

Make One More Pass

- One final look at overhead
  - Searching for remaining fruit

Final Tweaks

- Needed a 20bit structure per tri patch
- 20bit = 32bits
- 12 bits of wasted padding ☹

```
enum TriPatchType : uint32_t {
    STRIP,
    LIST,
    COUNT
};
```

```
struct TriPatch {
    static const uint32_t tcBias = 1;
    uint32_t firstVert :14; // Max 16k verts/model
    uint32_t triCount  : 4; // w/ bias, [1,16]
    TriPatchType type  : 1;
    uint32_t thinVerts : 1; // 8 or 16 bit verts
    uint32_t padding   :12;
};
```

Unpacking Fixed Points, Revisited

```
__m128i xVals;
const __m128i ZERO = _mm_setzero_si128();
const __m128i SCALE = _mm_set1_ps(0.125f);
if constexpr (sizeof(T) == 2){
    xVals = _mm_loadu_si128((__m128i*)vals);      // 0,1,2,3,4,5,a,b
} else {
    xVals = _mm_loadu_si64(vals);                 // 0,1,2,3,4,5,a,b,0,0,0,0,0,0,0,0
    xVals = _mm_unpacklo_epi8(xVals, ZERO);       // 0,1,2,3,4,5,a,b
}
__m128i xHi16 = _mm_bsrli_si128(xVals, 4);        // 2,3,4,5,a,b,0,0
__m128i xLo16 = _mm_unpacklo_epi16(xVals, ZERO); // 0,1,2,3
xHi16 = _mm_unpacklo_epi16(xHi16, ZERO);          // 2,3,4,5
*outXLo = _mm_mul_ps(_mm_cvtepi32_ps(xLo16), SCALE);
*outXHi = _mm_mul_ps(_mm_cvtepi32_ps(xHi16), SCALE);
```

Unpacking Fixed Point, Improved

```
__m128i xV;
const __m128i ZERO = _mm_setzero_si128();
const __m128i SCALE = _mm_set1_ps(0.125f);
if constexpr (sizeof(T) == 2){
    xV = _mm_loadu_si128((__m128i*)vals);                    // 0,1,2,3,4,5,a,b
} else {
    xV = _mm_loadu_si64(vals);                               // 0,1,2,3,4,5,a,b,0,0,…
    xV = _mm_unpacklo_epi8(xV, _mm_set1_epi8(xHi8));         // 0,1,2,3,4,5,a,b
}
__m128i xHi16 = _mm_bsrli_si128(xV, 4);                      // 2,3,4,5,a,b,0,0
__m128i xLo16 = _mm_unpacklo_epi16(xV, ZERO);                // 0,1,2,3
xHi16 = _mm_unpacklo_epi16(xHi16, ZERO);                     // 2,3,4,5
*outXLo = _mm_mul_ps(_mm_cvtepi32_ps(xLo16), SCALE);
*outXHi = _mm_mul_ps(_mm_cvtepi32_ps(xHi16), SCALE);
```

Final TriPatch Structure

```
struct TriPatch {
    static const uint32_t tcBias = 1;
    uint32_t firstVert   :14; // Max 16k verts/mdl
    uint32_t triCount    : 4; // w/ bias, [1,16]
    TriPatchType type    : 1;
    uint32_t thinVerts   : 1; // 8 or 16 bit verts
    uint32_t xThinHiBits : 4;
    uint32_t yThinHiBits : 4;
    uint32_t zThinHiBits : 4;
};
```

Final Numbers

- Fixed Point Tri Strips w/ Trees
  - ~80% size reduction
  - ~99% speedup

- 1/5th the size
- Twice as fast
- Collision for all models: 16MB

Ship It.

Takeaways

- Triangle strips are great for collision data
- Simple transforms often inlined in SIMD
- If you know your epsilon, fixed point is nice
- Doing more/loading less can be perf win
- Keep static/dynamic trees separate
  - Large compression possible in static

Acknowledgements

- Special Thanks
  - Peter-Pike Sloan: BVH construction tips
  - James Snider: Carte blanche in Treyarch's collision database

- References

1. Baldwin, Doug, and Michael Webber. "Fast Ray-Triangle Intersection by Coordinate Transformation." Journal of Computer Graphics Techniques (JCGT), vol. 5, no. 3, 2016, pp. 39–49., doi:April 16, 2021. http://jcgt.org/published/0005/03/03/paper.pdf
2. Hammon, Earl. "Extreme SIMD: Optimized Collision Detection in 'Titanfall'." Game Developers Conference, 2018, www.gdcvault.com/play/1025126/Extreme-SIMD-Optimized-Collision-Detection

Questions?

- andrew.shurney@activision.com

- www.linkedin.com/in/andrew-shurney