Zero-byte AABB-trees

Pierre Terdiman – v 1.2

Introduction

So you have a standard indexed triangle mesh, like this:

```
int              mNbVerts;
const float*     mVerts;          // XYZ for each vertex
int              mNbTris;
int*             mTriangles32;    // 3 vertex indices/tri
```

And then you want to raycast against that mesh. You want this operation to be as fast as possible, but you also want to use as little memory as possible. What are your options?

Option a): you just test each triangle of that mesh, one at a time.

Option b): you precompute an acceleration structure (say an AABB-tree) for the mesh, and then you raycast against that structure.

Option a) uses no memory but it is the slowest possible solution. Option b) gives you the fastest solution, but it uses an arbitrary amount of extra memory, which can be quite significant. This is a classical tradeoff between memory and speed.

Well. In this paper we will see a way to combine both options into option c): a zero-byte AABB-tree (ZBT). It uses no extra memory like a), but still gives you (almost) the speed of b).

I had the idea while working on PhysX's "bucket pruner" and Opcode 2.0, but never got the time to try it before now – it was just an entry in a huge TODO list. It appears that it has been independently discovered and published meanwhile (6), but while the idea and basic approach are the same, the implementation details in that paper are quite different.

The basics

A standard AABB-tree node looks like this:

```
struct AABBTreeNode
{
	AABB			mBounds;
	AABBTreeNode*		mPosChild;
	AABBTreeNode*		mNegChild;
	int			mTriangleIndex;
};
```

You have the AABB for each node, links to the “positive” and “negative” children (this is a binary tree with 2 children per node), and a triangle index. For internal nodes *mTriangleIndex* is not used, but *mPosChild* and *mNegChild* are. For leaf nodes, *mTriangleIndex* is used, but *mPosChild* and *mNegChild* are not (pointers are NULL).

This can easily be optimized into this common form:

```
struct AABBTreeNode
{
	AABB	mBounds;
	int	mData;
};
```

There is now a bit in *mData* indicating if the node is a leaf or an internal node. If the node is a leaf, the remaining 31 bits of *mData* encode the triangle index. Otherwise it encodes the index of the positive node child, and the negative node child immediately follows the positive one in memory.

Thus our initial vanilla node size is *sizeof(AABBTreeNode) = 32 bytes* (or 20 bytes if we compress the bounds). Please refer to (1) for details.

Getting rid of bounds

Bounds take the lion’s share of the node structure, so let’s attack them first.

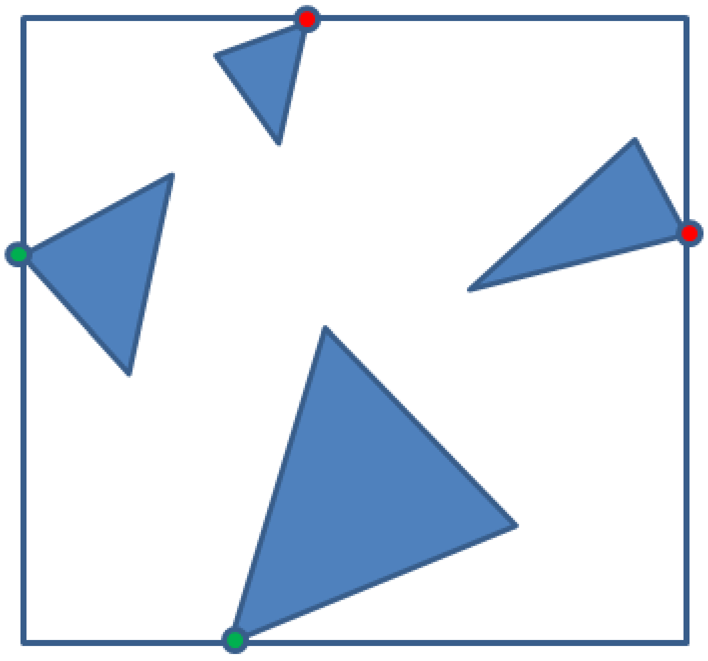
We define the AABB as two Min/Max vectors (as opposed to the Center/Extents format):

```
struct AABB
{
	float	mMin[3];	// Min point
	float	mMax[3];	// Min point
};
```

Any AABB in the tree has been derived from the initial set of triangles during tree building. And any of the 6 coordinates making up the AABB (or 4 coordinates in 2D) is in fact coming from one of the mesh’s vertices. So we can always encode the AABB using the indices of these contributing vertices:

```
struct VertexBasedAABB
{
	int	mMin[3];	// Vertex indices
	int	mMax[3];	// Vertex indices
};
```

For example in 2D, the min (green) indices would reference the green vertices in the diagram below, while the max (red) vertices would reference the red vertices.



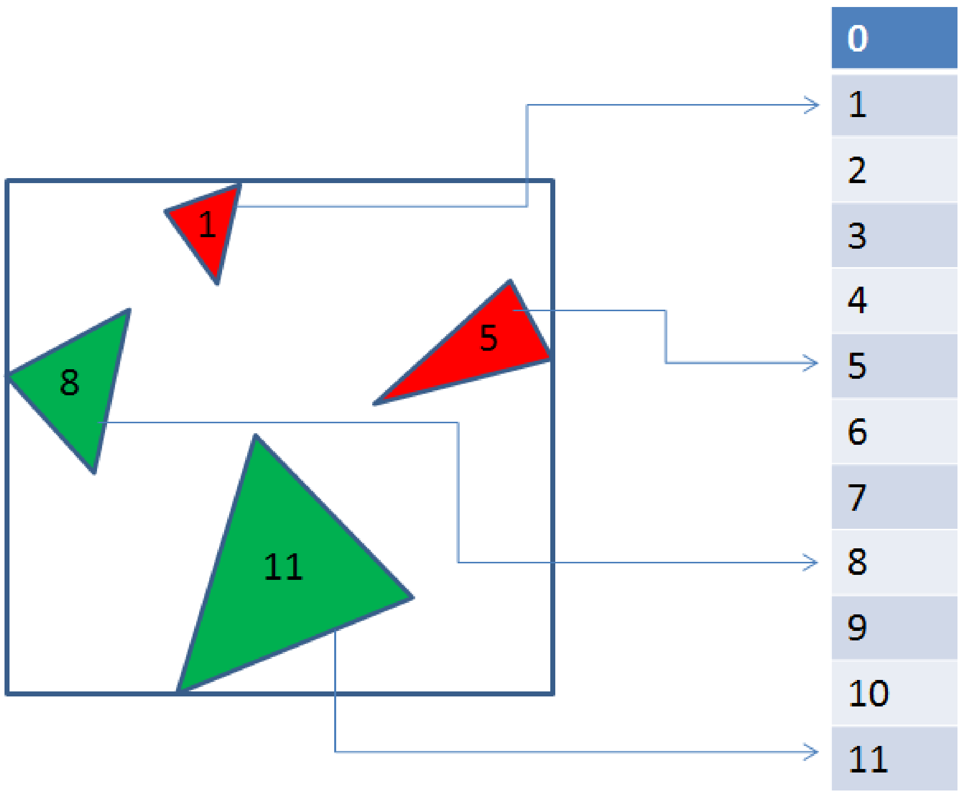
At runtime, we just look up the given vertices and retrieve the proper coordinate value, which gives back our initial AABB. This just works.

Now this alone does not buy us much since the size of the box is the same as before (sizeof(AABB)==sizeof(VertexBasedAABB)). One way to compress this would be to reorganize the mesh vertices, in such a way that the 6 (4 in 2D) encoded vertices would follow each other in memory. We would then only need to store the index of the first vertex, and we would know that the remaining ones immediately follow.

Unfortunately we cannot easily reorganize vertices since they are shared by several triangles. On the other hand we can freely reorganize triangles. And the box can also be encoded using triangles:

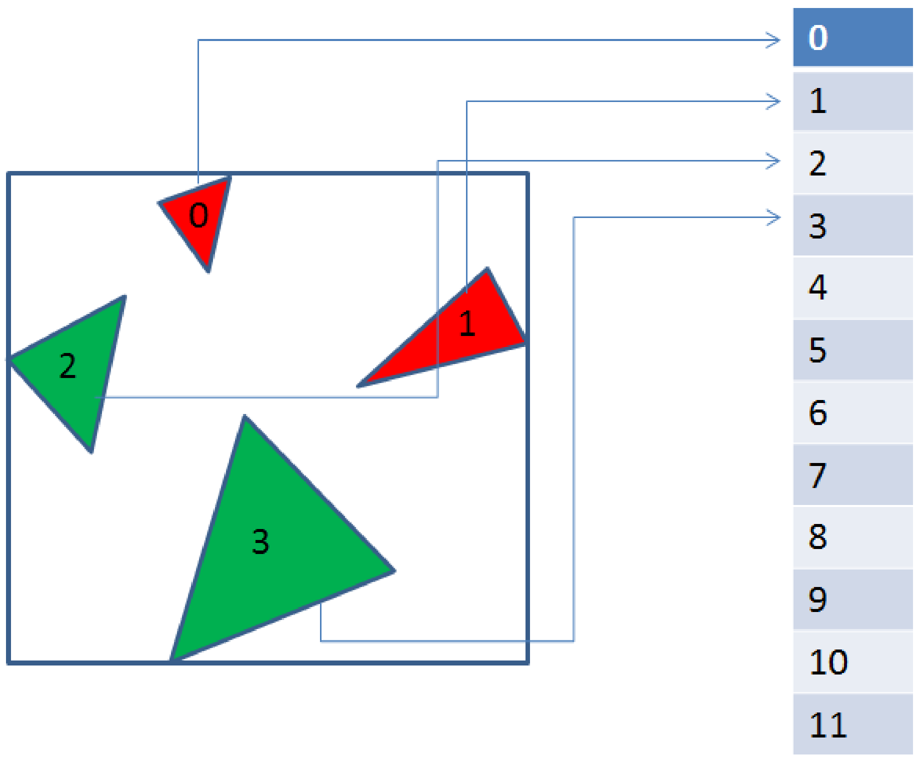
```
struct TriangleBasedAABB
{
	int	mMin[3];	// Triangle indices
	int	mMax[3];	// Triangle indices
};
```

That is, we do not directly reference the desired vertices, but we reference triangles referencing them. At runtime we now look up these triangles and compute the bounds around all their vertices.



In the above diagram we would look up 12 vertices in total instead of just 4 (in 2D. That would be 18 and 6 in 3D). This is worse for performance but better for memory usage, since we can now reorganize / reshuffle the triangles.

For example say that the above triangles have indices 1, 5, 8, 11 (their positions in the initial array of triangles). We can group the 6 (4 in 2D) triangles referenced by the box together, i.e. make them follow each other in the array of triangles. In our example their indices would become, say, 0, 1, 2, 3:

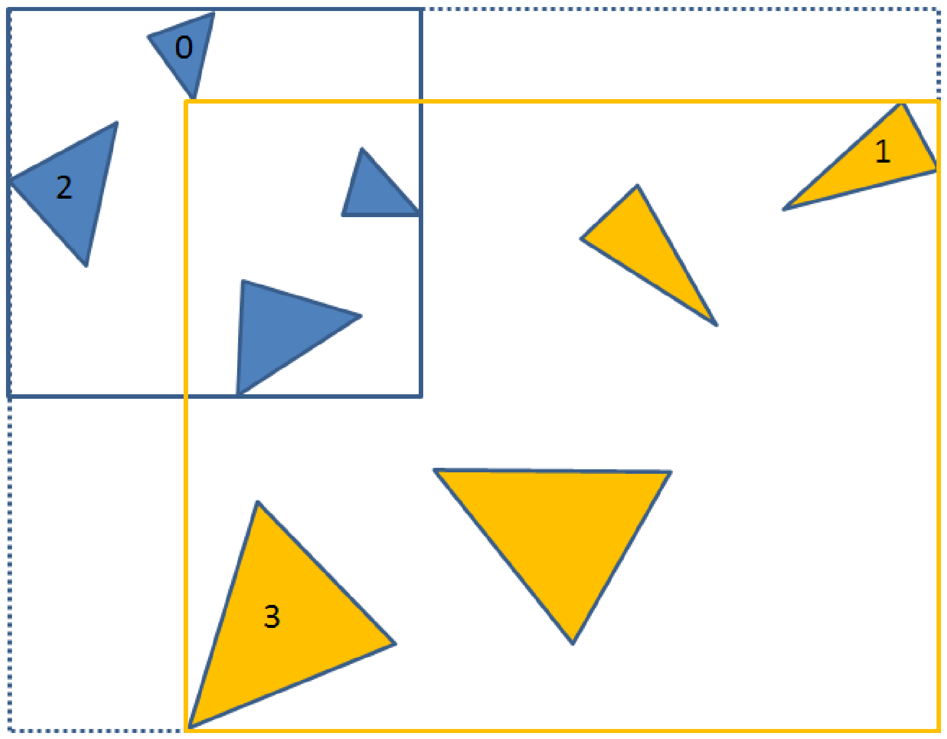


The set of triangles can now be encoded with just one number, the starting index in the array of triangles:

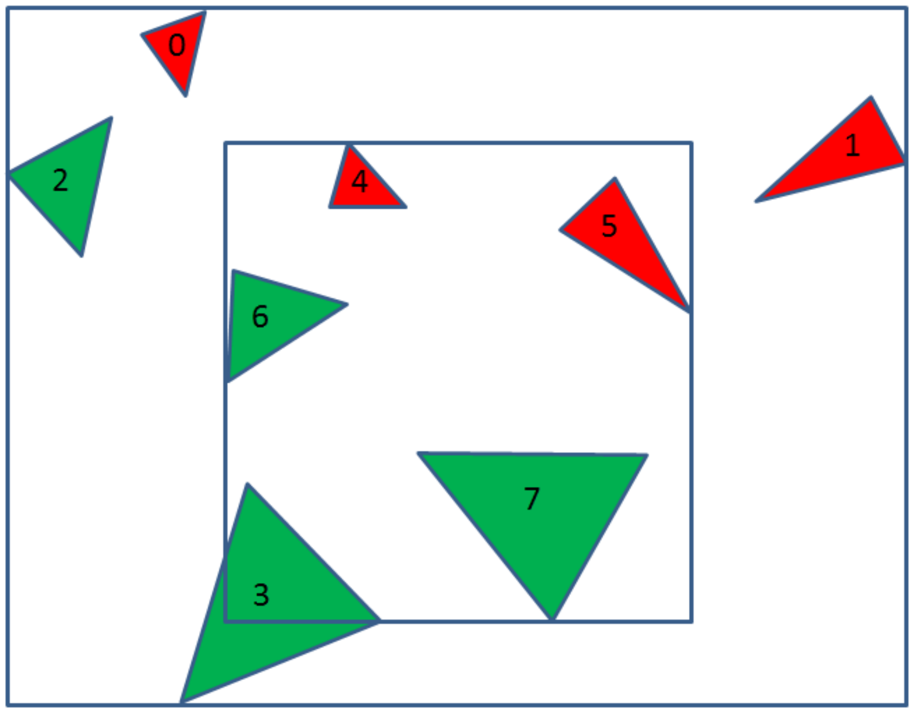
```
struct TriangleBasedAABB
{
        int     mStartIndex;        // Triangle index
};
```

Once this is done for a node, we repeat the process for children nodes. There is a catch here. As we know since Gomez (2), the children AABBs for a node share half of their coordinates with their parent. See for example the following diagram: triangles 0, 1, 2, 3 are the ones needed to recompute the top-level (dashed) bounding box. According to the tree building code, the children nodes should contain the blue triangles in one child, and the orange triangles in the other child. But triangles 0 and 2 will be needed to recompute the blue bounding box, and triangles 1 and 3 will be needed to recompute the orange bounding box. Since these triangles have been relocated to the start of the triangle array in the previous step, we cannot relocate them again to another position, for computing the children bounds.

And it seems the previous encoding scheme with just one index does not actually work.



There is a trick to make it work though: simply remove the 6 (4 in 2D) already relocated triangles from the mesh, for the rest of the tree building process. Ignore these parent triangles when building the children nodes. With this scheme, we get something like the following diagram:

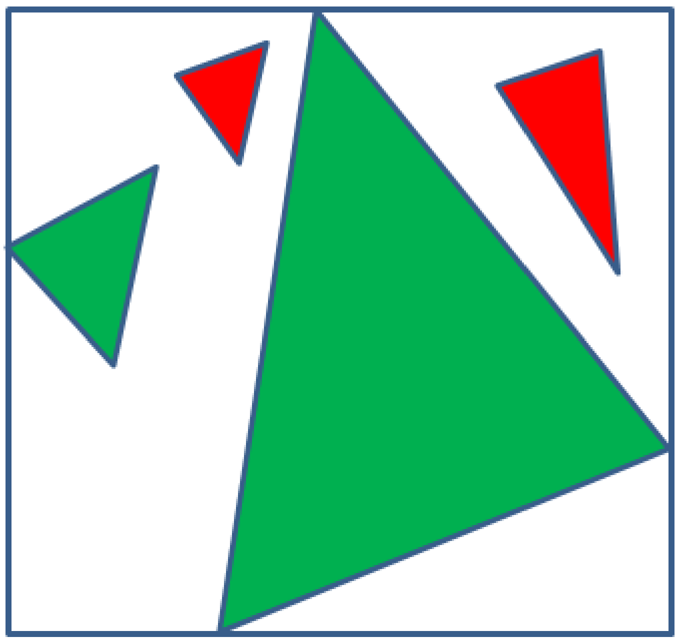


Triangles 0 to 3 are relocated to the start of the triangle array for the parent node. Then they are ignored. The first child node now identifies the remaining triangles (relocated to positions 4 to 7) as the ones needed to recompute its bounding box. And we can encode the AABB for this first child node with a single index again ('4' in this case).

This has two consequences: on one hand, there will be a need to test all these triangles at runtime, for each internal node. For example, when performing a raycast against the top level node, we will need to perform raycasts against triangles 0 to 3 at the same time we recompute the node's bounds. This is simply because these triangles, ignored by the building code after their relocation, will never appear in leaf nodes. If we do not test them while recomputing the bounds, our raycast query will potentially miss triangles and give wrong results.

On the other hand, the total number of nodes in the tree will be greatly reduced compared to a regular tree. From the point of view of the building code, it is roughly as if there was suddenly 6 (4 in 2D) times less triangles to process. This is not important right now, but it will be later. It guarantees that the number of nodes will be smaller than the number of triangles, which is not usually the case when building 'complete' BV-trees (they have 2*N-1 nodes with N = number of triangles).

Now, so far we have always talked about "6" triangles, but there is not always a need for 6 of them. A given AABB can actually be defined by 1 to 6 triangles. For example the following 2D bounds are properly defined by just the 2 green triangles:



It is always possible to enforce "6" triangles by simply choosing random extra triangles in each node, until we have 6 of them. If we always have a known number of triangles per node, we do not need to record that number anywhere and we save memory.

However as we saw these triangles are tested at runtime, so is a good idea for performance to minimize the number of triangles needed to encode the bounds. But then of course we need to store a number of triangles in each node, giving us this format:

```
struct TriangleBasedAABB
{
      int   mStartIndex;      // Triangle index
      int   mNbTris;
};
```

Let’s recap. So far, our node structure looks like this:

```
struct AABBTreeNode
{
      // Bounds
      int   mStartIndex;      // Triangle index
      int   mNbTris;
      // Leaf/Internal node data
      int   mData;
};
```

That is 12 bytes/node, which is much better than the 32 bytes we started from. We are not done yet though.

Clearly we do not need 32 bits for *mNbTris*, since this is a number between 0 and 6. We only need 3 bits, and we can immediately merge this with *mData*.

Remember that *mData* originally had 1 bit to mark leaf nodes, and 31 bits for a child node index. It turns out that we do not need that leaf bit anymore, thanks to the changes made to encode bounds.

For a leaf, *mData* encoded a triangle index. But we can now encode that triangle index in *mStartIndex*, merging the triangles-from-internal-nodes (needed to compute the bounds), and the triangle-from-leaf-nodes (needed to compute the ray-triangle intersection). Since we must test the triangles making up the bounds for ray intersection anyway, there is really no difference between the two. And thus, it is a simple book-keeping matter to merge all these triangles together (relocating the ‘leaf triangles’ after the ‘internal node triangles’). Since we had only one triangle per leaf, *mNbTris* can now be up to 7, which is perfect (it still fits on 3 bits and we do not waste one of the 8 bit combinations anymore). The 31 bits of *mData* for leaf nodes are now free, and we can drop the *mData* leaf bit by stating that the child node index encoded in *mData* is 0 for leaves.

For internal nodes *mData* encoded a (child) node index, and this does not change. But we can now use 3 bits of *mData* to encode the number of triangles, leaving 29 bits to encode the node index - which is way more than enough.

Thus the node structure is now:

```
struct AABBTreeNode
{
      int   mStartIndex;      // Triangle index
      int   mData;            // Child node index/Number of tris
};
```

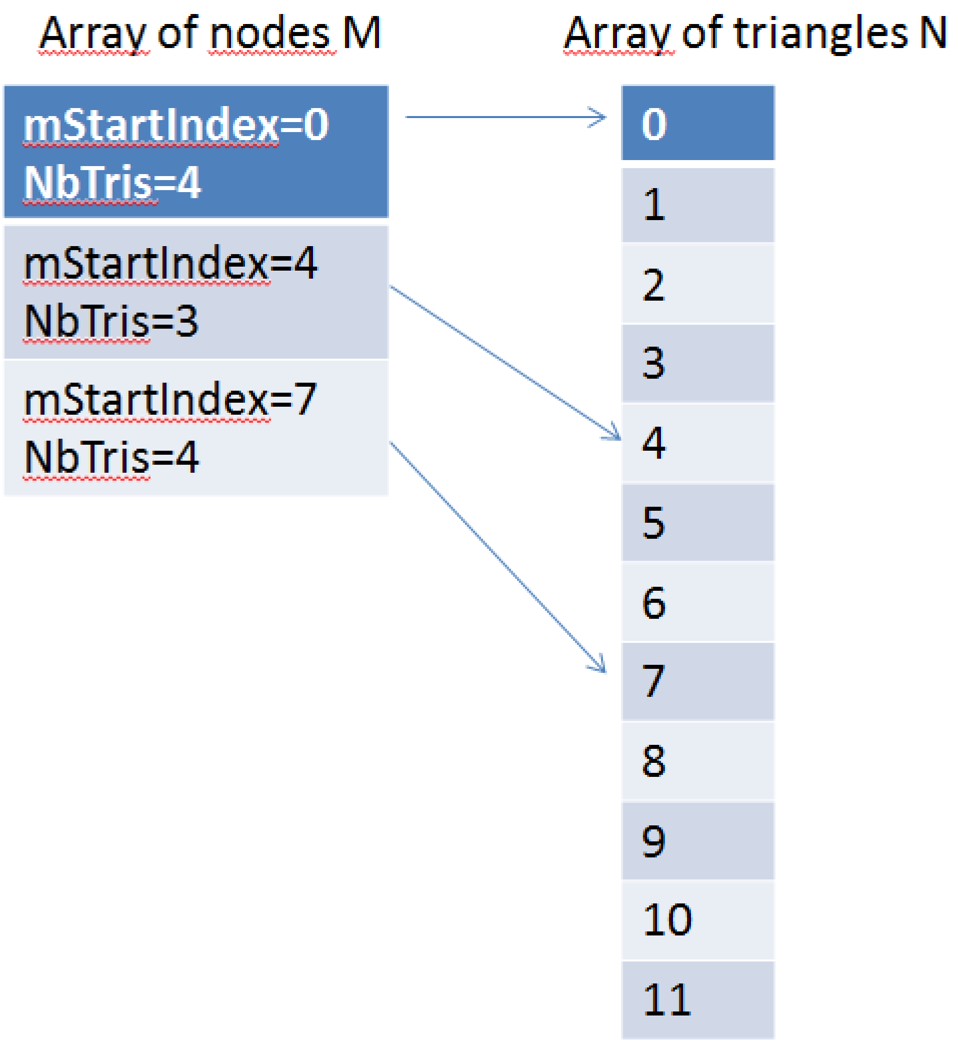
Getting rid of triangle index

We are down to 8 bytes per node: this is nice but not enough. Gomez had 11 bytes/node years ago already so the improvements so far are minimal.

Fortunately we can do better.

We now have a set of N triangles and a set of M nodes. By construction (and as we previously mentioned) we will always have M<=N. And since the tree nodes reference chunks of the set of triangles, we can easily make *mStartIndex* fully implicit.

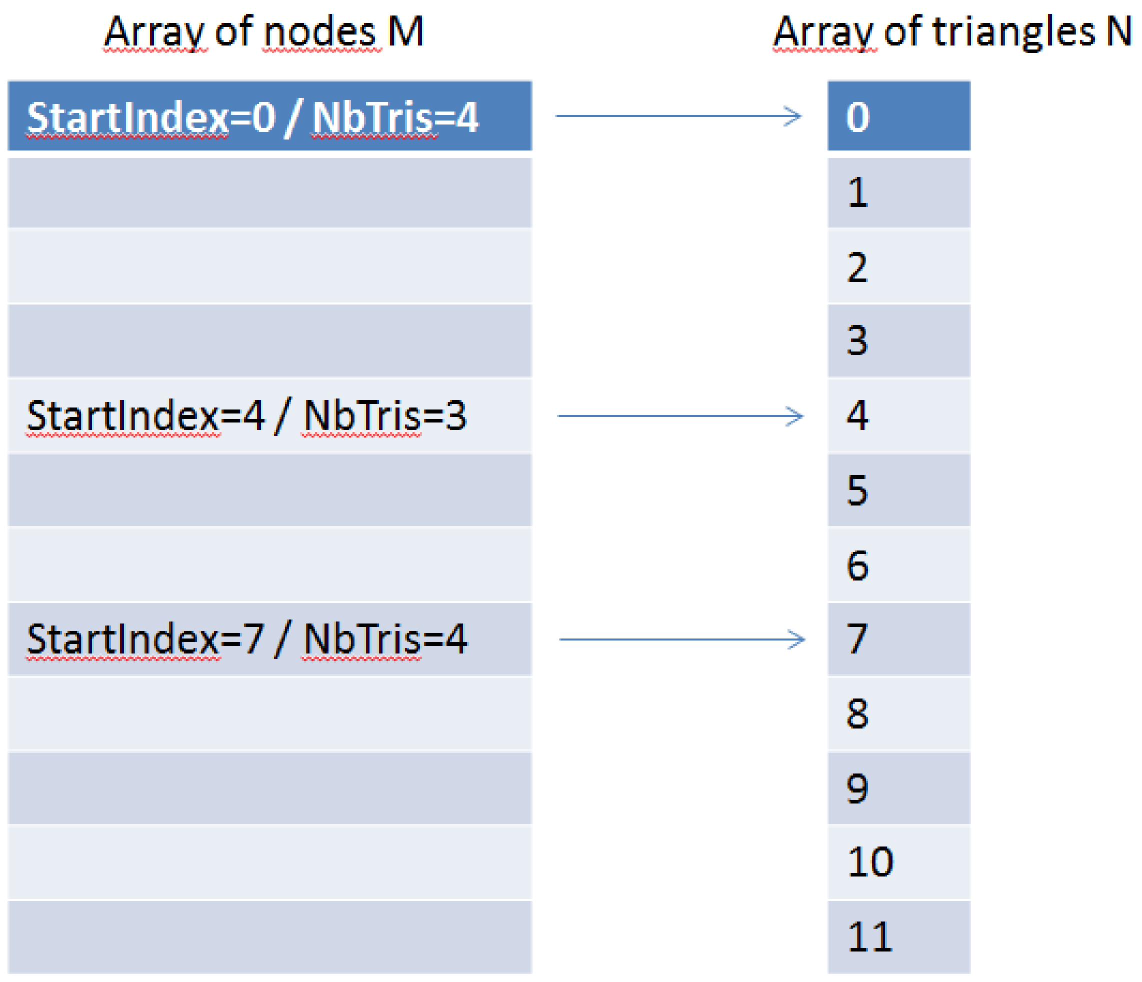
We start from this:



And then we:

- Allocate more nodes than necessary, enforcing M=N.
- Spread the nodes in the array so that node[i].mStartIndex = i

That is:



We now have a 1:1 mapping between node indices and (start) triangle indices. There is no need to store *mStartIndex* anymore; it is now equal to the node indices encoded in *mData*.

Granted, this allocates more nodes than necessary, but we are going to use that in just a moment. For now the important point is that our node structure is down to just 4 bytes per node:

```
struct AABBTreeNode
{
	int	mData;		// Child node index/Number of tris
};
```

Merging with mesh data

The final step should be obvious after enforcing M=N and looking at that last diagram. The natural next move is to merge the two arrays. We only need 4 bytes for encoding the node, but in reality we can easily get away with 3. We mentioned that we had 29 bits in *mData* to encode a node index, but 21 bits would be enough to support meshes up to ~2 million triangles.

That is 3 bytes left in *AABBTreeNode*, 3 vertex indices per triangle in the mesh data from the very first paragraph. For the same reason we do not need the full 32 bits to encode a triangle or node index, we do not need the full 32 bits to encode vertex indices in the mesh data. So we simply store the 3 bytes from the node in the MSBs of vertex indices in the mesh data.

And just like that, we are down to 0 byte/node.

Precomputed Node Sorting

Before looking at runtime performance, let's deal with node sorting. Traditional node sorting would be expensive with this scheme since the AABBs are not immediately available and must be recomputed at runtime. This is the kind of situation where *Precomputed Node Sorting* (PNS) shines. Unfortunately as seen in (3), we need 1 byte per node to encode the PNS data, and unfortunately we do not have any space left for this now.

Or…. Do we?

Recall that we got rid of the triangle indices by spreading the array of nodes over the array of triangles. So for example if a node has a position P and is referencing 6 triangles, the next node will be located at position P+6 in memory (and this is a computation we will need to do at runtime to access the "negative" nodes that were always located just after the positive ones, if you remember).

P+6…

This means that we actually have plenty of space left for each node. We encoded our remaining 3 bytes in the first referenced triangle (P[0]) but we did not touch P[1] to P[5], which still contain available space. For a node referencing 6 triangles, we actually still have (6-1)*3 = 15 bytes available!

So the trick is simply to compute the PNS byte as usual, and store it in the next referenced triangle. For a node at position P, the PNS byte would be located in the MSB of the first vertex of triangle P+1.

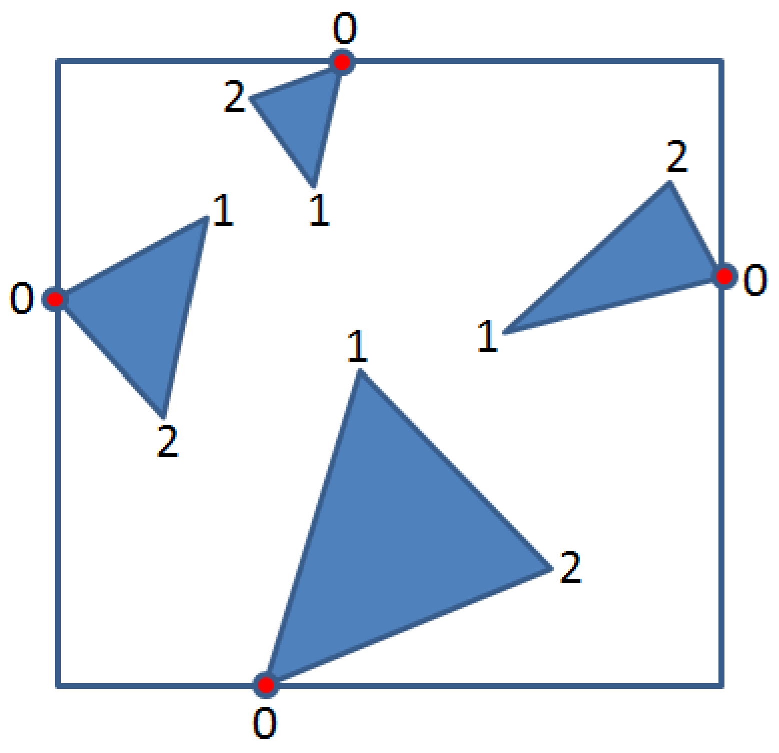
This works just fine as long as the number of triangles referenced in each node is higher than 1. If the number is equal to 1, the node maps to a single triangle and there is indeed no space left anywhere to store a PNS byte.

But the beauty of PNS is that the sorting is approximate, and getting it wrong in a few nodes does not matter much (it cannot really be any worse than not doing any sorting). So for the rare (internal) nodes referencing a single triangle, we do not compute the PNS byte at all and we let the code read and use an incorrect PNS byte. It makes things infinitesimally less efficient than they could be, but it still works just fine.

Box cache

The bounds are now recomputed at runtime. Fetching triangles is not a big issue since they are now fully sorted and share their addresses with the tree nodes. So there is no extra cache misses to access the triangles, it is the same as accessing the nodes. Fetching vertices however is very costly. For an AABB referencing 6 triangles, the code needs to fetch 3*6 = 18 vertices. This is a lot of potential cache misses, and it is clearly the slowest part and the biggest issue in the scheme.

One way to improve this would be to shuffle the vertices within a triangle, making sure that the coordinate contributing to the AABB is always given by the triangle's first vertex. That way the code only needs to fetch 1 vertex per triangle, not 3:



However this only works when keeping 6 triangles per node. In cases where, say, an AABB is fully defined by a single triangle, one must consider all 3 vertices of that triangle, there is no way to avoid the work.

But another option is to use a box cache. Once the bounds are recomputed, they are stored in a small cache and retrieved from there whenever possible. We found that a 4K entry cache gave a good performance boost for a reasonably small amount of memory. Note that the cache is shared between all meshes: there is one cache per scene, not one cache per mesh. On one hand it keeps the amount of memory dedicated to the cache acceptable (about 260Kb in our tests). On the other hand it complicates things for multithreaded queries, where one needs to allocate one cache per thread. This is hardly a problem in practice but it is worth mentioning.

Results

We benchmarked the Zero-Byte Tree (ZBT) using PEEL (4). All scenes use “raycast closest” calls, i.e. we are looking for the closest hit (rather than “any hit” or “all hits”). Some scenes use long rays, some scenes use short rays. Some scenes use frustum rays as if the scene was raytraced, some scenes use random radial rays, some scenes use vertical rays. Some scenes have few meshes, with each mesh containing a large amount of triangles. Some scenes have thousands of meshes, with each mesh containing a small amount of triangles.

A number of libraries have been tested at the same time, to give a fair idea of ZBT’s performance compared to regular BV trees.

Opcode 1.3 (1) is an old library which was one of the fastest around when it was released (around 2003). The original code does not use SIMD instructions, but it has been recompiled with /SSE2 as a PEEL plug-in (giving it a performance boost, but not as much as if it had been written using SIMD intrinsics). The library supports various trees, and in these tests we use “quantized no-leaf trees”.

Opcode 2.0 is a new rewritten version of Opcode. It uses SIMD natively and all the tricks in the book at time of writing.

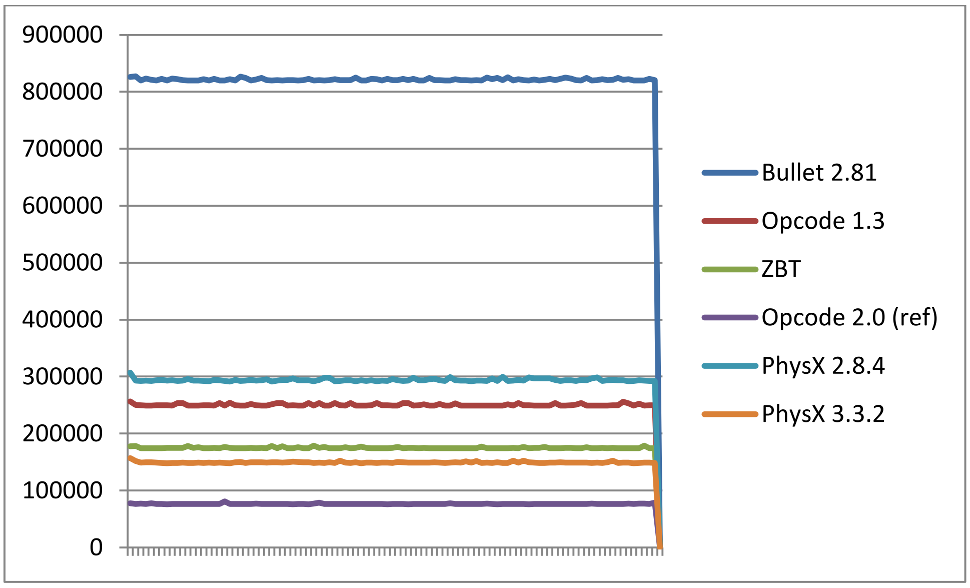
PhysX 2.8.4 is an old version of the popular PhysX library. PhysX 3.3.2 is the latest released version of that same library.

Bullet 2.81 is a recent version of the popular Bullet library. It has been recompiled with the same optimization settings as the other libraries (maximizing speed).

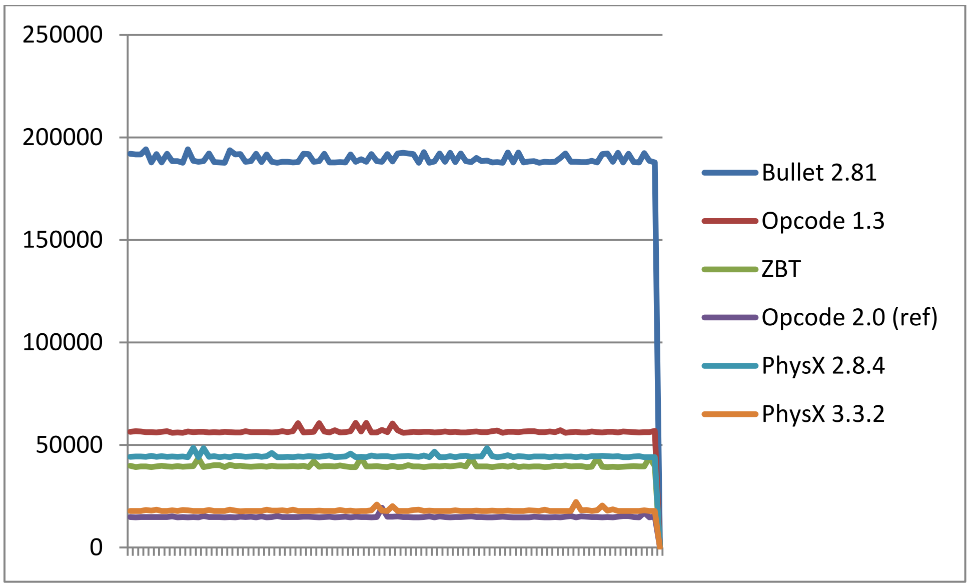
Both Bullet and PhysX have a high per-raycast overhead compared to Opcode, so even when they use exactly the same algorithms and data-structures for the midphase, Opcode usually ends up faster. So in a way it is a bit unfair to compare Opcode-based implementations with Bullet and PhysX. Nonetheless, it is important to include these libraries as fair points of reference, since they are used in actual games to perform actual raycasts, using the same interface and entry points as these benchmarks.

The results are for single-threaded queries but the relative performance of involved libraries does not change much with multi-threaded queries.

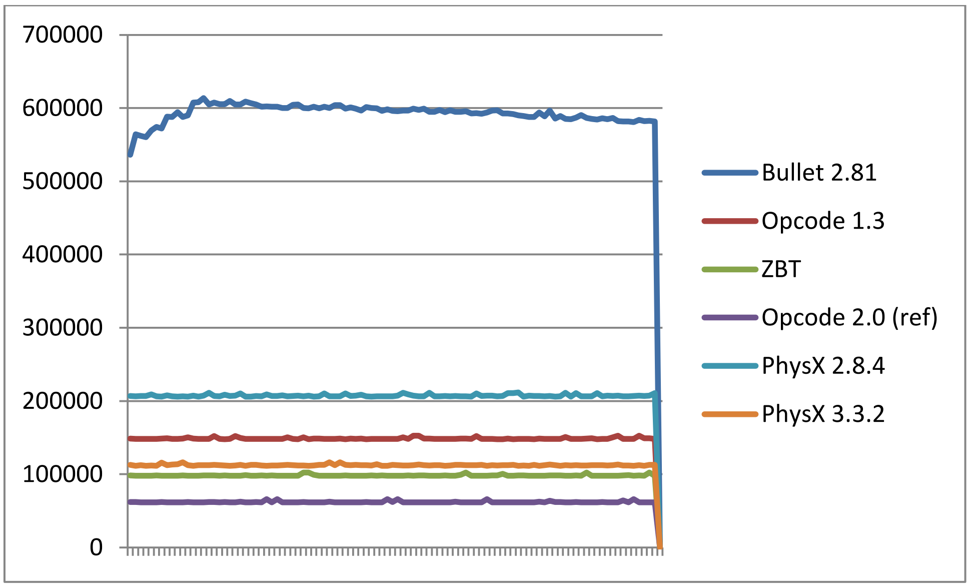
Bunny_RT:



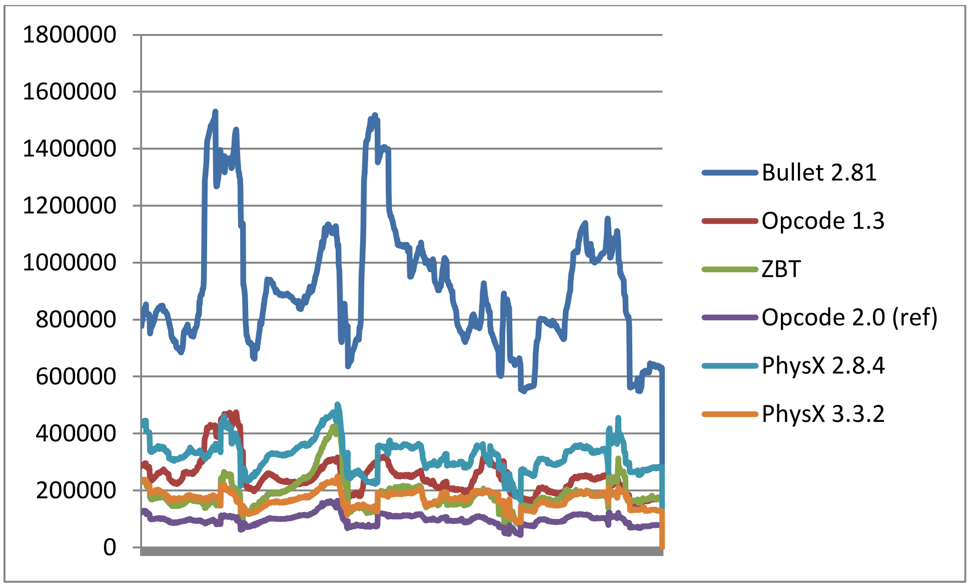
PlanetSideBulletRays:



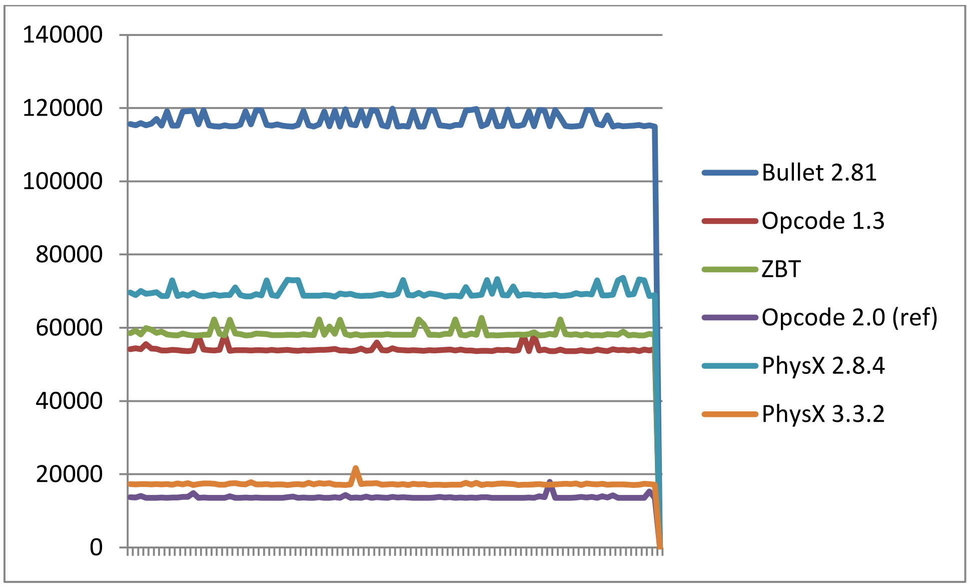
KP_RT:


700000
600000
500000
400000
300000
200000
100000
0
Bullet 2.81
Opcode 1.3
ZBT
Opcode 2.0 (ref)
PhysX 2.8.4
PhysX 3.3.2

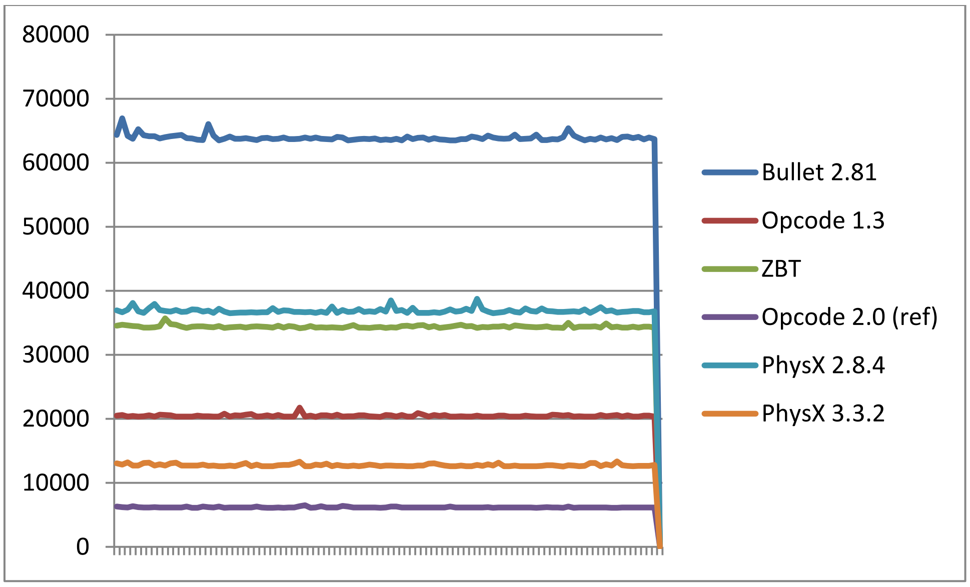
KP_RT2:


1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
Bullet 2.81
Opcode 1.3
ZBT
Opcode 2.0 (ref)
PhysX 2.8.4
PhysX 3.3.2

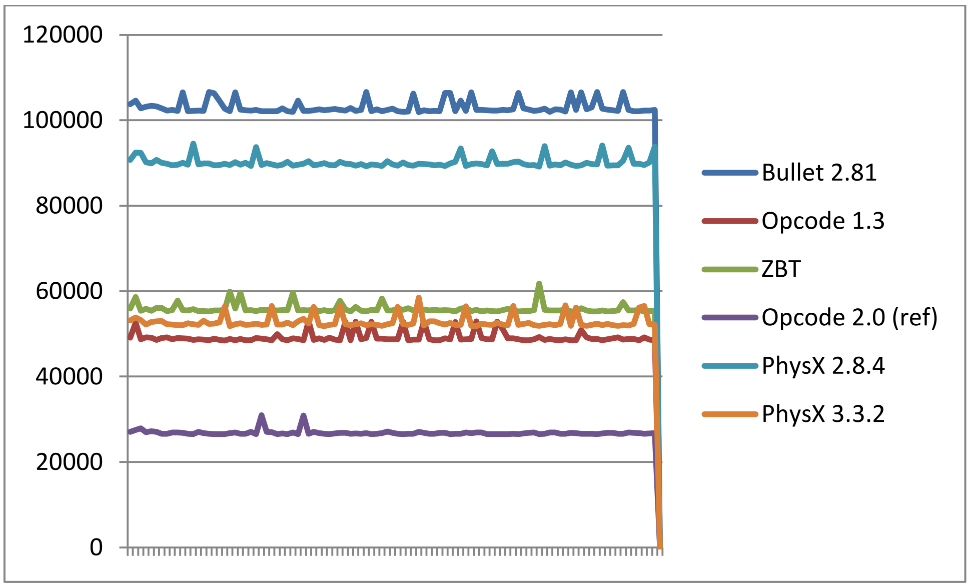
InsideRays_TestZone:



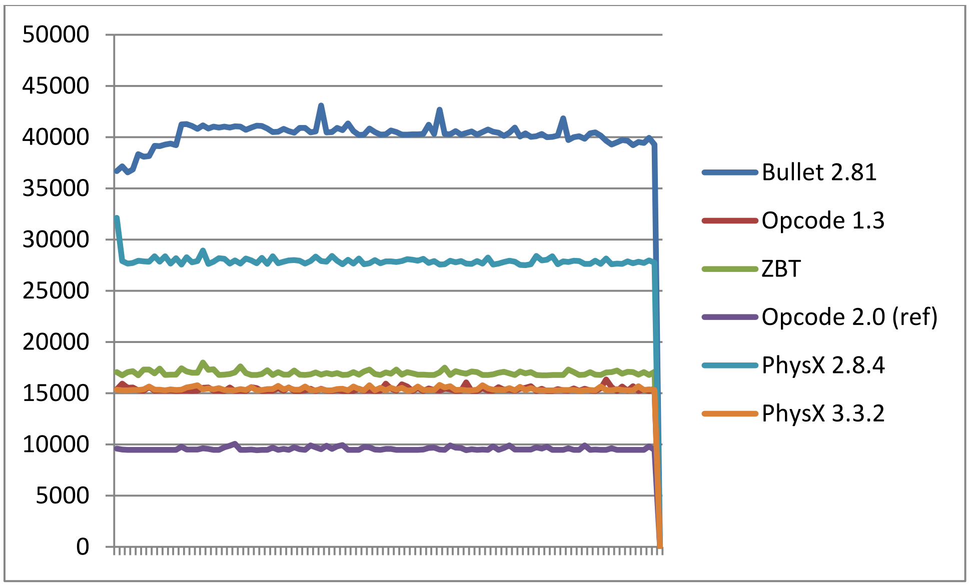
OutsideRays_TestZone:



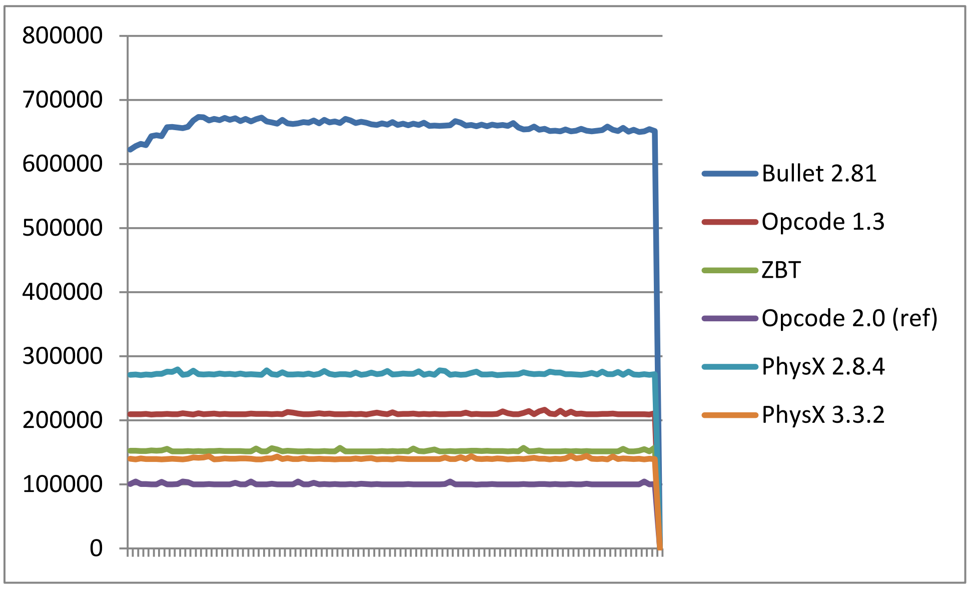
SceneRaycastVsStaticMeshes_Archipelago16384: (vertical rays)



SceneRaycastVsStaticMeshes_KP2_Short:

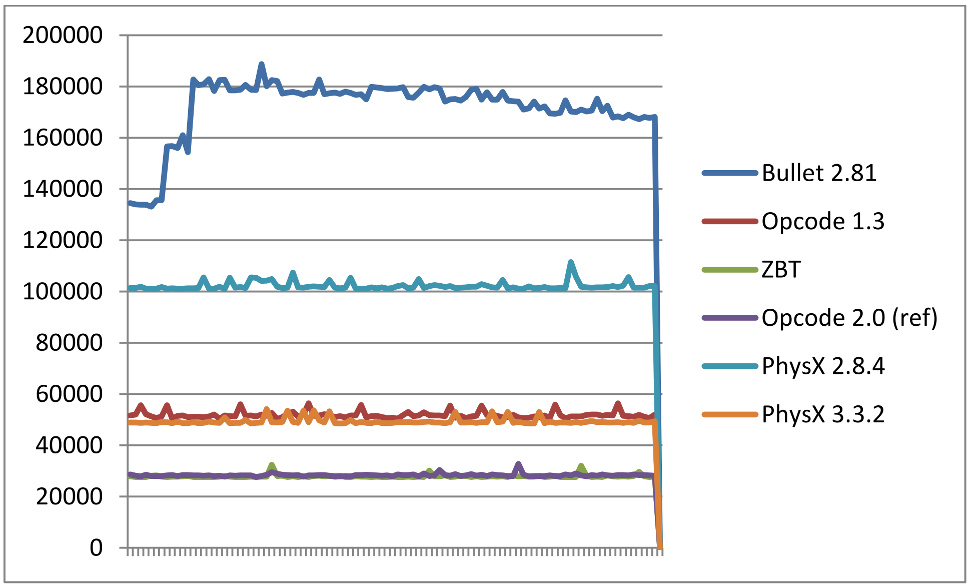


SceneRaycastVsStaticMeshes_KP1

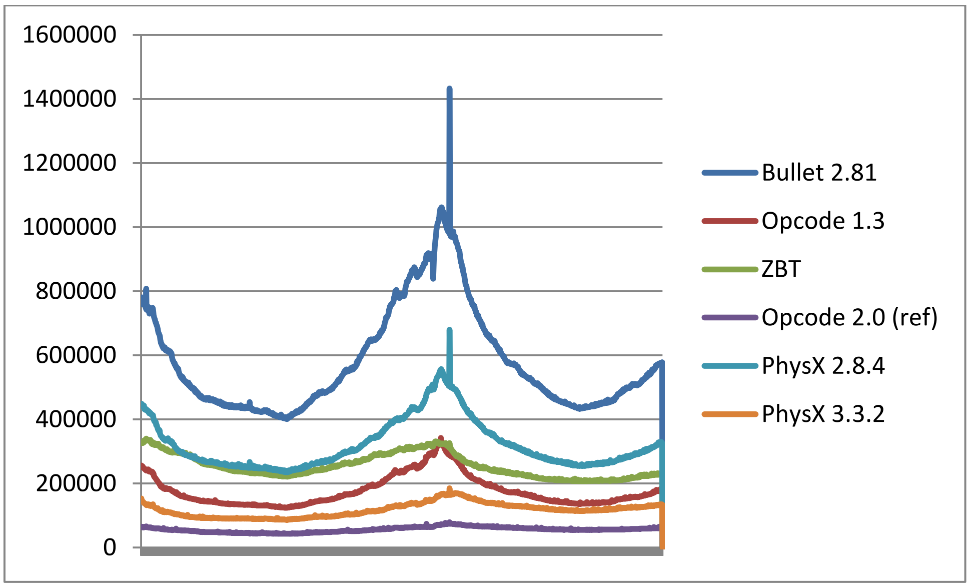


6384:

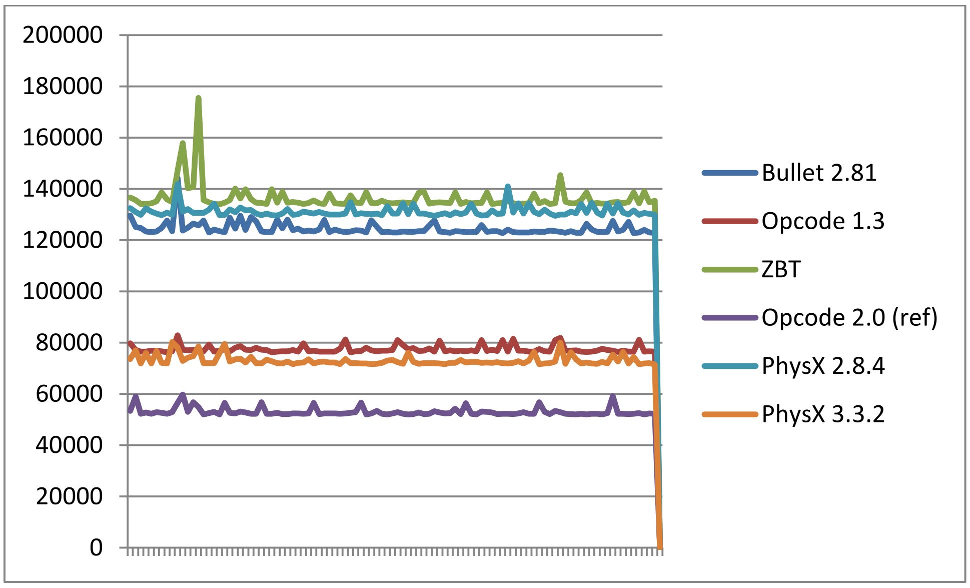
SceneRaycastVsStaticMeshes_KP16384_NoHit:



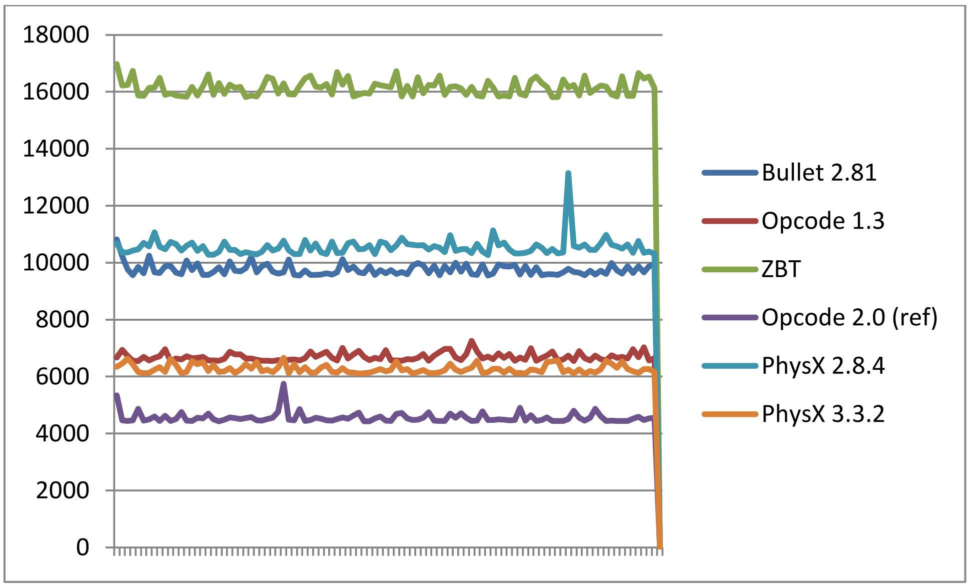
SceneRaycastVsStaticMeshes_MeshSurface:



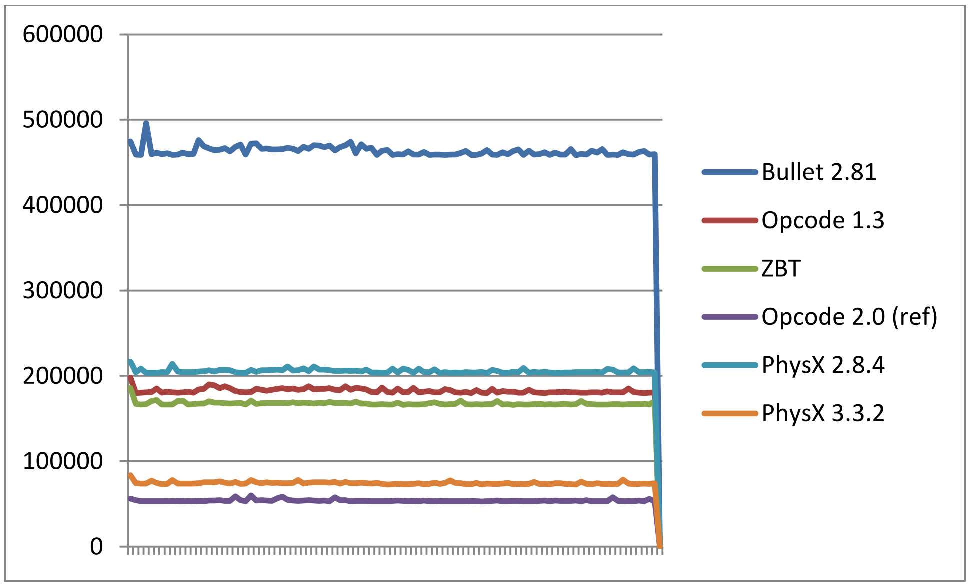
SceneRaycastVsStaticMeshes_Terrain_Long:



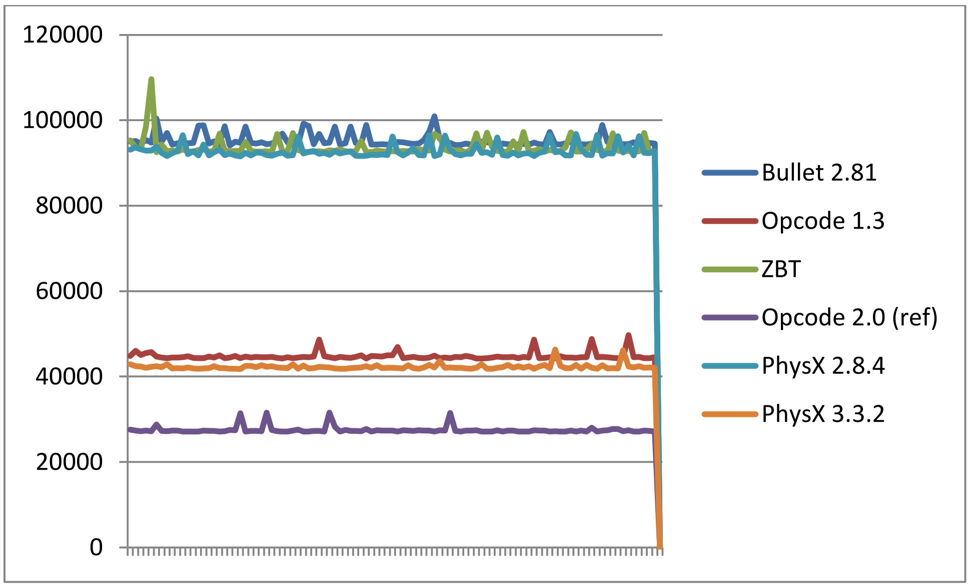
SceneRaycastVsStaticMeshes_Terrain_Short:



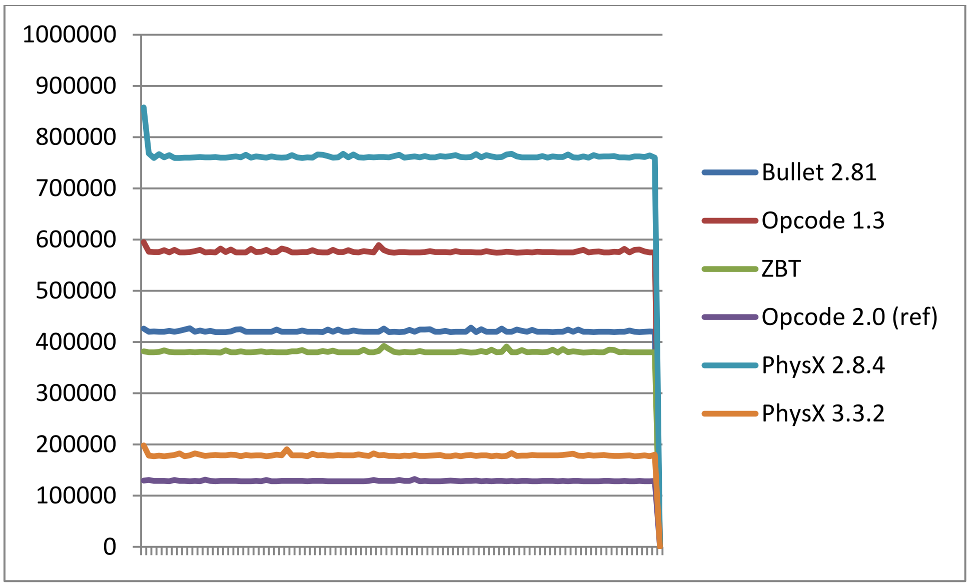
SceneRaycastVsStaticMeshes_TessBunny16384_2:



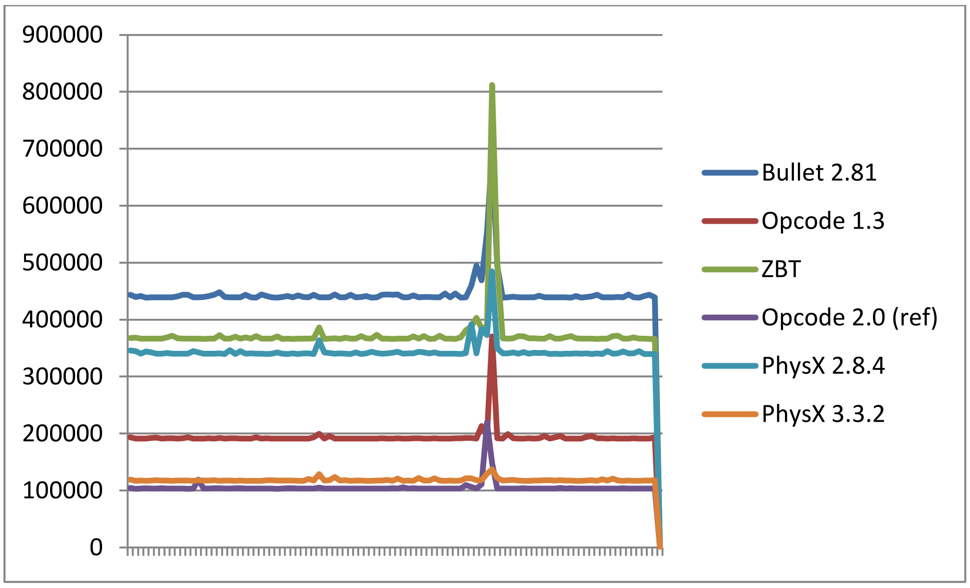
SceneRaycastVsStaticMeshes_TessBunnyShort:



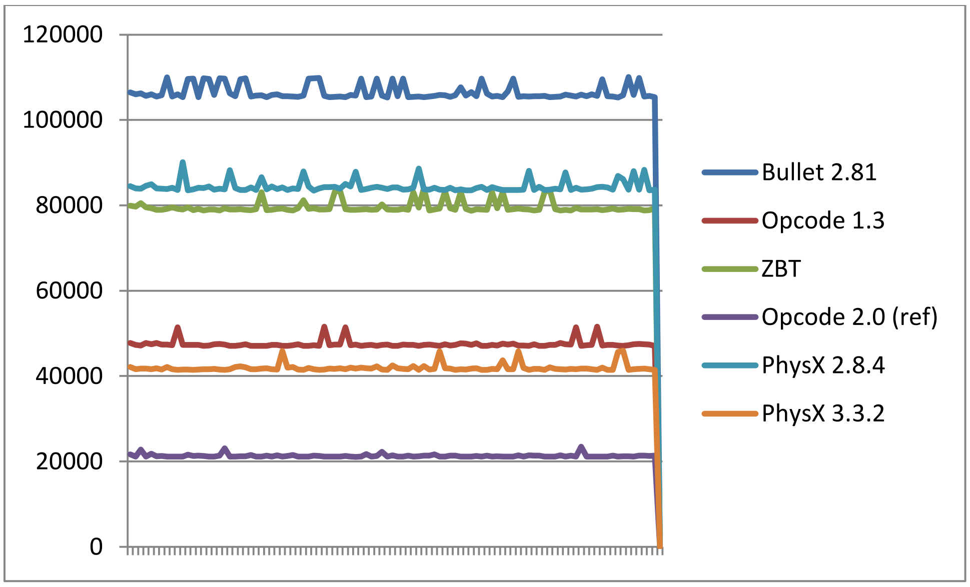
SceneRaycastVsStaticMeshes_TestZone_LongRays:



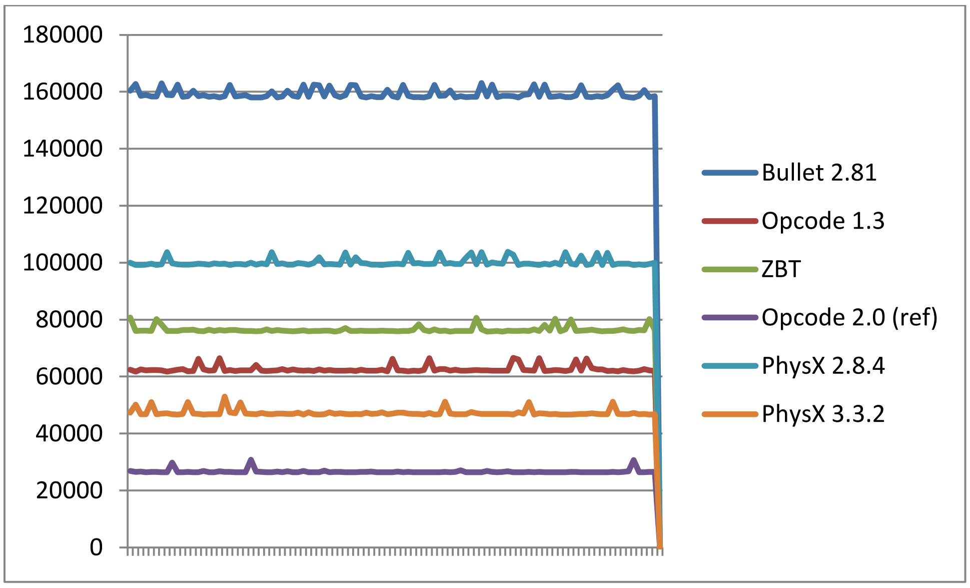
SceneRaycastVsStaticMeshes_TestZone_ShortRays:



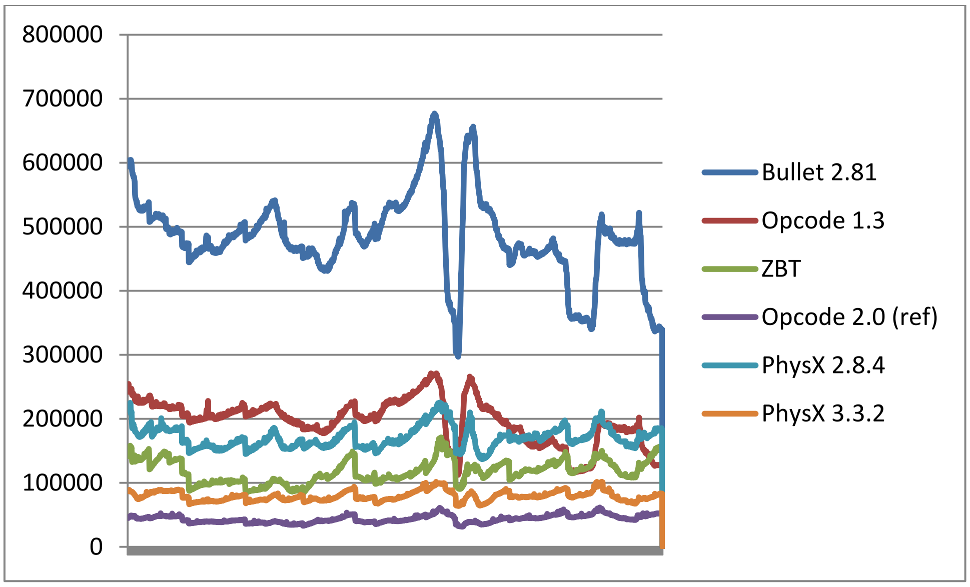
SceneRaycastVsStaticMeshes_TestZone_VerticalRays:



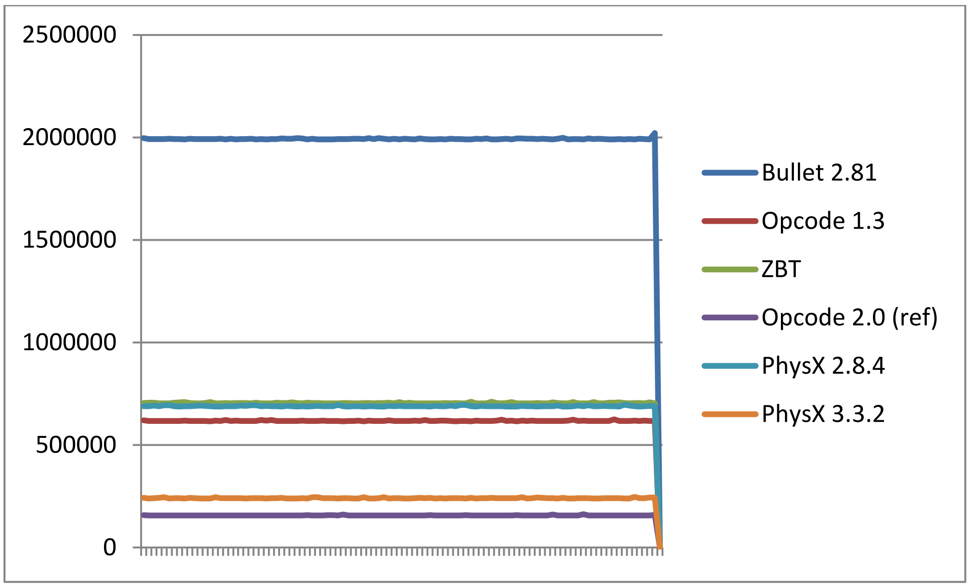
Terrain_RT:



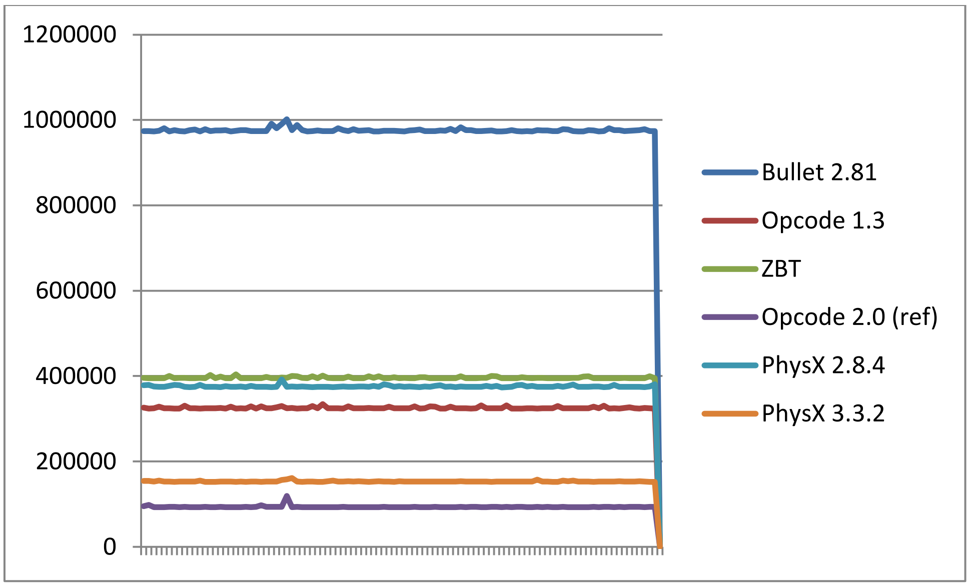
Terrain_RT2:



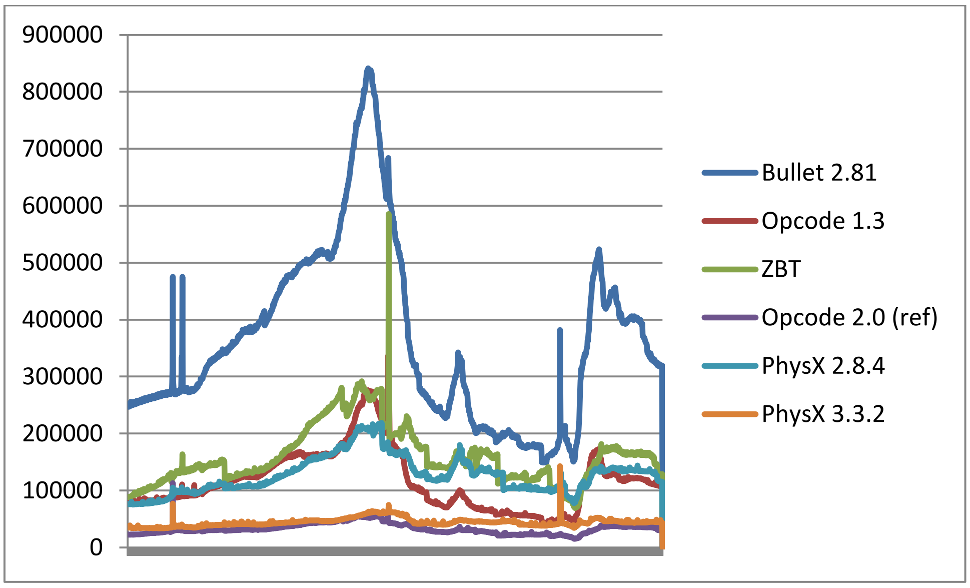
TestZone_RT:



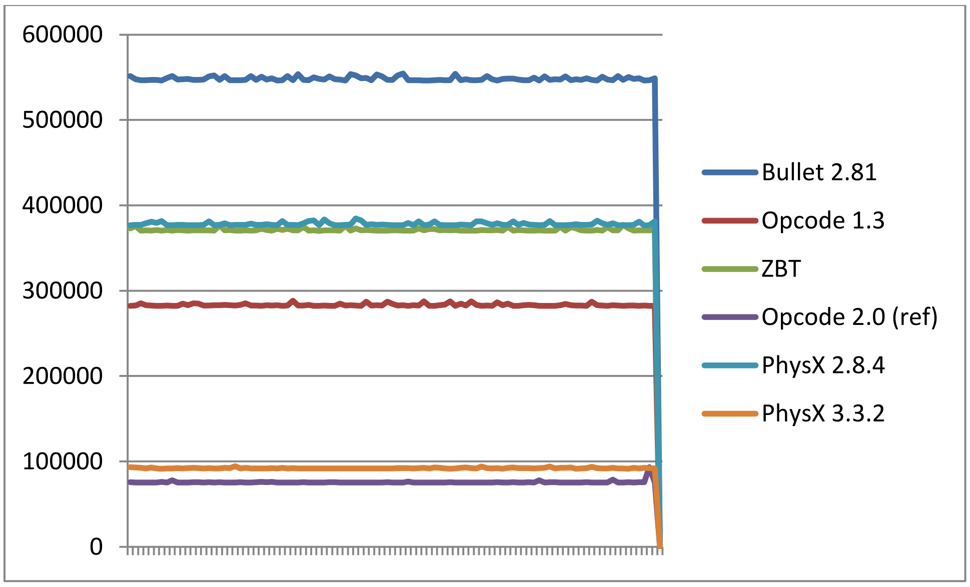
TestZone_RT2:



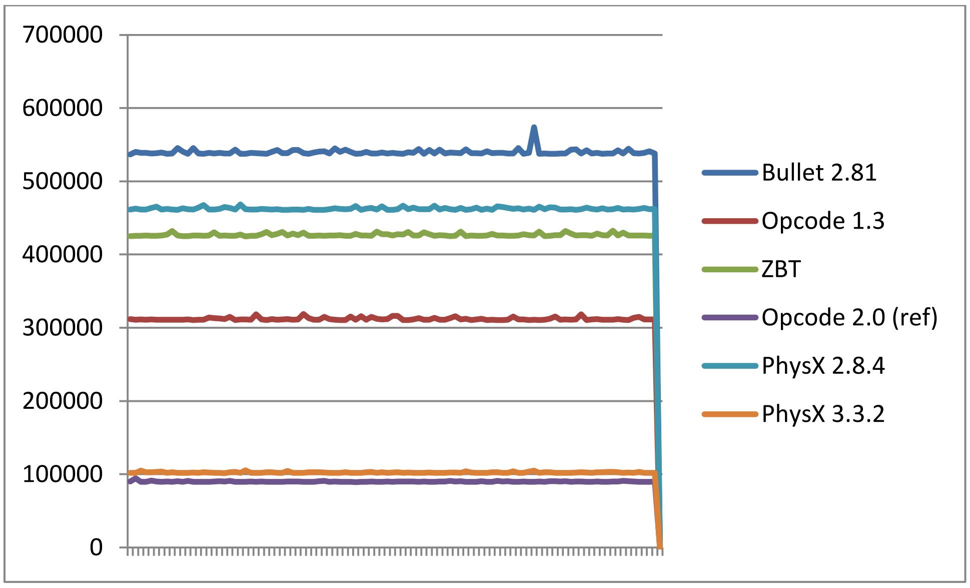
TestZone_RT3:



ValveRaycastsMidGame1:



ValveRaycastsOnlyRays:



Comments:

- Opcode 2.0 is always faster in these tests. This is expected, since this library contains all the knowledge and tricks accumulated by the author over the course of 10 years (working on PhysX, looking at Bullet, etc). This gives us a "speed of light" reference to compare ZBT to.

- Recall that ZBT uses zero bytes, i.e. we are comparing apples to oranges a bit here. Tweaking the amount of memory used by the mesh data structure usually has a direct impact on performance. To be "fair" we should compare ZBT to the brute-force O(n) approach where we raycast against each triangle of a mesh, since this is the only other approach using no extra memory. But of course this would be pointless since ZBT is orders of magnitude faster than this, for obvious reasons.

- The performance is surprisingly good. ZBT is often close to, or faster than Opcode 1.3. I suspect this is due to the native SIMD code and node sorting support, which were both missing from Opcode 1.3.

- ZBT is also faster than PhysX 2.8.4 and Bullet 2.81 in a majority of cases.

- In most cases it cannot compete with PhysX 3.3.2 or Opcode 2.0. Nonetheless in rare cases it still manages to be very close (or even faster than PhysX 3.3.2).

- ZBT is rarely the slowest. Despite using no memory, it remains competitive with regular BV-trees. The approach is not just an interesting engineering idea: it looks like a viable, usable solution. This is really the best we could hope for, since obviously ZBT was not designed for speed.

Limitations

The approach has a few limitations.

First, we implicitly assumed that a mesh was using 32bit vertex indices. This is needed for us to be able to store the remaining last 3 bytes of tree data within the triangle mesh structure. This does not work for meshes that use 16bit vertex indices. In practice however, using a 16bit indexed mesh + a regular BV tree still takes more memory than using a 32bit indexed mesh alone. So one can always switch all meshes to 32bit indices, which also simplifies the code everywhere these indices are accessed (there is no need to support 2 different index sizes anymore).

In a similar way, the approach does not work as-is for non-indexed vertices.

Finally, the bounds are not explicitly stored anymore so they cannot easily be "refit" (5). In other words the approach does not work well with deformable meshes. When the mesh deforms, the array of triangles must be re-shuffled again, and the tree rebuilt. This is slower than a refit operation.

Differences with already published material

I only noticed after the work was done that a similar idea had been published already (6). There are large differences between the two versions though:

- They do not recompute the full AABB in each node. Instead they limit the computation to a single axis (chosen implicitly depending on the tree's current depth). This is faster (only 2 triangles are needed instead of 6) but of course it does not prune sub-trees as efficiently as when using the full AABB.
- They use a fixed number of triangles per node, which forces them to duplicate triangles sometimes.
- They do not mention vertex shuffling, even though their approach could benefit from it (even with 2 triangles and 1 axis per box only).
- They do not use a box cache.
- There is no mention of node sorting and it is unclear if it is used. In any case they would need to find some memory somewhere to use something like PNS.
- They cannot use an SAH-based tree construction, because their scheme uses a fully implicit tree structure where each child node index can be derived from their parent index. This makes the problem easier to solve (there is no need to find room for storing node indices) but it imposes severe restrictions on the building code. Our implementation has no such limitations.

In fact I believe the work in this paper covers this part of their "limitations and future work" section:

"Hopefully, we can find ways in the future to implicitly represent other creation schemes, like the SAH, as well."

Future work

A lot of things have not been covered due to lack of time.

Performance for “raycast any” and “raycast multiple” (raycast queries returning boolean answers or all touched triangles) should be analyzed. There is no technical reason why this would give fundamentally different results from “raycast closest” calls though.

Performance of overlap and sweep queries should be analyzed. Performing extra overlap tests or extra sweep tests on the triangles making up the AABBs is likely to be more costly than performing extra raycast tests, so ZBT may suffer here.

We mentioned that the approach does not work for non-indexed vertices. However it could be adapted to work on these, which would probably increase performance. If we would store 6 contiguous non-indexed vertices for each node, there would be no cache misses anymore from fetching these vertices. The address for a node would be the same as the address for the vertices themselves, and recomputing the bounds would become faster. However there would still be a need for storing 3 bytes of tree data somewhere, so this would only work with compressed vertices of some kind. Using quantized non-indexed vertices could be a good middle-ground, the memory gains from quantization counterbalancing the losses from not using indexed vertices.

Hybrid trees could be investigated. These would be regular BV-trees keeping a relatively high number of triangles per leaf (maybe hundreds), and each leaf would be a Zero-Byte Tree. This could give back most of the performance of Opcode2 for just a fraction of its memory usage.

References

(1) Opcode collision library, http://www.codercorner.com/Opcode.pdf
(2) Miguel Gomez, Game Programming Gems 2, “Compressed Axis-Aligned Bounding Box Trees”
(3) Precomputed Node Sorting, http://www.codercorner.com/blog/?p=734
(4) PEEL, http://www.codercorner.com/blog/?p=748
(5) Gino van den Bergen, “Efficient Collision Detection of Complex Deformable Models using AABB Trees”, http://www.cs.cmu.edu/~djames/pbmis/etc/jgt98deform_AABB.pdf
(6) No-memory BVH, http://graphics.tu-bs.de/publications/Eisemann11NMH/