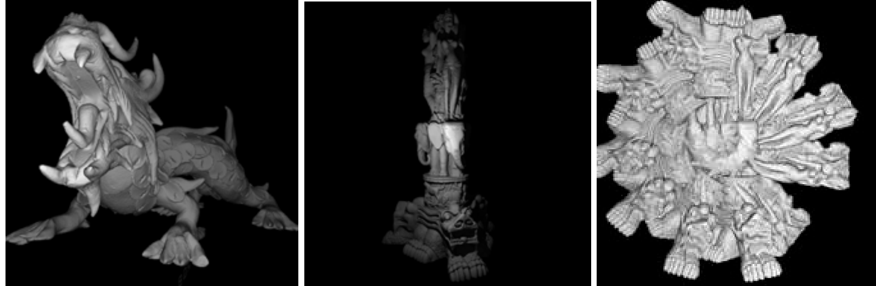
Efficient acceleration structures layout for rendering on 32- and 64-bit many-core architectures

Maxim Shevtsov, Alexei Soupikov and Alexander Reshetov
Intel Corporation
{maxim.y.shevtsov, alexei.soupikov,alexander.reshetov}@intel.com



a) Asian Dragon model, 7.2M triangles, 64-bit extension consumes only 2Mb of 1.3Gb acceleration structure, extension processing time is <0.5% of rendering time

b) Thai Statue model, 10M triangles, 64-bit extension consumes only 2.1Mb of 1.4Gb acceleration structure, extension processing time is <0.5% of rendering time

d) Thai Statue model replicated 7 times (70M triangles total) 64-bit extension consumes only 4Mb of 7Gb acceleration structure, extension processing time is <0.5% of rendering time

Figure 1. 64-bit extension overhead for large models. Models are ray-traced with shadows at 1024x1024 on a 2-way 3GHz Intel ®Core™2 Duo machine (4 threads for construction/rendering) with 8 GB RAM

ABSTRACT

Any rendering solution needs fast acceleration structures to reduce the complexity of solving search problems. We address the following problems of constructing acceleration structures for the large number of primitives: compact memory layout, efficient traversal capability, memory address space independence, parallel construction capability and 32/64 bit efficiency.
This proposed kd-tree layout solution solves the above problems completely with highest possible efficiency. It is easily applied to other hierarchical acceleration structure types as well.

KEYWORDS: Rendering, acceleration structure, kd-tree, ray-tracing, proximity search.

1. Introduction

Rendering algorithms use acceleration structures to reduce the complexity of solving search problems [1]. Making their usage practical for high-speed parallel requires addressing of the following problems:

- Efficient traversal capability – compact representation should not slow down the traversal step
- Memory address space independence –an acceleration structure could be saved/loaded/transferred easily
- Parallel construction capability - the data format of acceleration structure should support creation in multiple parallel threads
- 32 and 64 bit efficiency – the acceleration structure size should not explode on 64 bit architectures. The 32 bit mode acceleration structure mode should have exactly the same binary representation on 64 bit architectures.

In this paper we propose specific memory layout solving the above problems. We use kd-tree as example, but the solution we proposed is also applicable to a wide range of partitioning hierarchies.

2. Previous work

The kd-tree is basically a binary tree in which each node corresponds to a spatial cell. Inner node of kd-tree represents splitting plane and refers to the two child nodes. Each leaf node, in contrast, stores primitives counter and refers to a corresponding list of primitives.

A compact kd-tree layout has been proposed in [3]. It uses only eight bytes per node. Storing offsets instead of pointers makes the data structure independent of the base address changes:

```
/* basic 8-byte layout for a kd-tree node */
struct KDTreeNode {
       union{
             //position of axis-aligned split plane
             float split_position;
             // or number of leaf primitives
             unsigned int items;
       }
       unsigned int dim_offset_flag;
//'dim_offset_flag' bits encode multiple data:
// bits[0..1]: encode the split plane dimension
// bits[2..30]: encode an unsigned address offset
// bit[31]: encodes whether node is an inner node or
leaf
};
// macros for extracting node information
#define DIMENSION(n) (n->dim_offset_flag & 0x3)
#define ISLEAF(n) (n->dim_offset_flag & (UINT)(1<<31))
#define OFFSET(n) (n->dim_offset_flag & 0x7FFFFFFC)
```

3. Solution

Efficient leaf/node test

During traversal the leaf/internal node test is executed at each traversal step and the branch depends on its results, so its performance is critical. Having 0 as a leaf indicator allows reducing the test to exactly 1 instruction before branch:

```
and Node, 0x03
jz  ProcessLeaf
```

```
/* 8-byte layout for a kd-tree node */
struct KDTreeNode {
      union{
            float split_position;
            unsigned int items;
      }
      int dim_offset_flag;
// 'dim_offset_flag' bits encode data in a new way
// bits[0..1] : indicate either
// • a leaf(if set to 0)
    // if 'items' field is >=0 it is true leaf
    // otherwise it is 64-bit extension
// • an inner node with split plane dimension
//   (if set to 1,2,3 for x,y,z axis corresp.)
// bits[2..31] : encode a signed address offset
};
```

32 and 64 bit efficiency

To handle an unpredictability of resulting tree size a construction algorithm usually allocates memory by continuous regions. The number of links between those sub-trees is relatively small (<<1% of total number of total number of kd-tree nodes).

The typical region size is way smaller than 4GBs. So the nodes can use 32-bit offsets to reference children within the same region. The only nodes that need 64-bit offsets are the nodes having children located in another memory region. *Since the number of such nodes is small they are encoded as extensions of 32-bit nodes. We use negative values of the primitive counter to indicate that the leaf is special, and it is a 64-bit extended node.*

Multiple threads construction

When the tree is constructed in multiple threads each thread builds some sub-tree [2]. Thus different threads may create a parent node and its children nodes. So when a parent is created the offset to children nodes may be unknown. That fact prevents from allocating 64-bit offset data next to a node. The 64-bit extended node data (actual 64-bit offset, actual leaf/axis and actual counter/split fields) is stored in a special per-thread table.

64-bit extension node is a special type of leaf with the following values of its fields:

a) *-(entry+1),* where *entry* is a table entry number, is stored in **items** field;

a) *(tbl)<<2* where *tbl* is a table number, is stored in **dim_offset_flag** field; note that 2 least-significant bits are zeroed, indicating a leaf.

Each construction thread creates its own 64-bit node table to avoid contention. If a table becomes full the correspondent thread just increases its size by re-allocation and data copy. Since the table is small it does not affect construction performance. The tests on models with up to 70M polygons demonstrated that 256-entry per-thread tables were never full. Storage or transmission of the tree located in multiple memory regions requires only adjustment of cross-region offsets in the table.

```
// relocation table's entries
struct TableEntry{
      //actual leaf/node but with offset==0
        KDTreeNode node;
      //true offset
        __int64 offset;
};
#define ISLEAF(n) (!(n.dim_offset_flag&0x3))
#define DIMENSION(n) ((n.dim_offset_flag&0x3)-1)
#define IS_64BIT_EXT(n) (n.items<0)
#define HASITEMS(n) (n.items)

#define MAKELEAF(n,its,ofs) n.items = its; \
  n.dim_offset_flag = ofs;
#define ENCODE64BIT_EXT(n,table_id,entry_id) \
      MAKELEAF(n,-(entry_id+1),table_id<<2)

#define DECODE64BIT_EXT(node, newadr) \
   int tab_id = (node.dim_offset_flag)>>2; \
   int entry_id = -node.items-1; \
   TableEntry e = m_tables[tab_id][entry_id]; \
   newadr = &node + e.offset; \
   node =  e.node;
```

Modifications of traversal algorithm

The tree constructed by 32-bit code can be rendered by 64-bit code without any modifications. To avoid testing 64-bit extension indicator at each traversal step the extended nodes are stored as leaves. Since the probability of traversing leaf is very small comparing to probability of traversing internal node the additional 64-bit extension test is performed at a very small fraction of traversal steps.

```
register KDTreeNode node = m_root;
// ADRINT is  int or __int64 (32/64-bit architectures)
ADRINT newadr = &node;
traverse_loop:
while (!ISLEAF(node)){
       //get dimension, traversal order, etc
         const ADRINT adr0 = newadr+…;//front child
         const ADRINT adr1 = newadr+…;//back child
         //traverse of either back/front child or both
         //…
}
//processing leaves
const int nitems = HASITEMS(node);
if(nitems>0){
       //...
}
#ifndef _M_X64
else if(IS_64BIT_EXT(node)){
//64-bit extensions processing:
//newadr is patched using relocation table
         DECODE64BIT_EXT(node, newadr);
      goto traverse_loop;
 }
#endif// _M_X64
//popping from stack etc
```

4. Memory/Performance considerations

Our experiments with variety of models proved that memory overhead from using proposed 64-bit extensions is negligible (refer to Figure 1 for examples models).

Also managing per-thread 64-bit node tables doesn't affect construction performance. Our measurements show that construction slowdown is <1% and thus is negligible too. We also performed tests for 1-128 construction threads with wide range of models (1-100M polygons). They demonstrated that 128-entry per-thread tables are far most than sufficient to connect portions of a tree constructed with different threads.

The performance of rendering using new layout supporting 64-bit extensions is the same as of rendering the efficient layout supporting 32-bits only (see Figure 1 for details). Even on complex models and high memory regions granularity the slowdown using the proposed layout was less than 0.5% comparing with 32-bit only offsets and one continuous memory region for the whole tree.

References

[1] V. Havran: "*Heuristic Ray Shooting Algorithms*". PhD thesis, Czech Technical University in Prague, 2001.

[2] M. Shevtsov, A. Soupikov, and A. Kapustin.: "*Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes*". In Proceedings of Eurographics 2007.

[3] I. Wald, "*Realtime ray tracing and interactive global illumination*", PhD thesis, Saarland University, 2004.