Simple Geometry Compression for Ray Tracing on GPU

Kirill Garanzha[1] Alexander Bely[2] Vladimir Galaktionov[1]

[1] Keldysh Institute of Applied Mathematics, Russian Academy of Sciences

[2] CentiLeo

Abstract

In this short paper we describe simple approach to loosely compress the vertex data for ray tracing large triangular models on the GPU. The disadvantage of the GPU is limited memory capacity. The advantage of the GPU is high performance computation. Sometimes it is hard to load large models to the GPU and we suggest compressing the vertices that form 3D models and acceleration data structure using 16bit representations instead of 32bit floats. At the same time the aggregate vertex precision varies between 22 and 24 bits and the amount of cracks is minimized. The advantages of NVIDIA CUDA platform are used to implement our approach efficiently.

Keywords: *ray tracing, GPU, CUDA, compression.*

1. INTRODUCTION

Scenes for feature film rendering and CAD/medicine visualization may have large geometric complexity and can easily contain ten or hundred million polygons. Demands for greater photorealism, more realistic materials, complex lighting and global illumination push computational bounds which often results in long render times and more research in data compression methods.

Complex lighting can be implemented with ray tracing rather simply. However efficient ray tracing implementation on the GPU is still a challenge. The GPUs such as NVIDIA GTX480 and GTX580 have around 1.5Tflops of compute power and only 1.5GB of main memory. Our challenge is to load a model with several ten million triangles to these GPUs.

One of the most popular 3D model representations is triangular representation which encodes the surface of the 3D model. One of the most popular acceleration structures for ray tracing is the Bounding Volume Hierarchy (**BVH**) of Axis Aligned Bounding Boxes (**AABBs**) [4].

In this paper we present a simple and efficient method for compact vertex coordinates and BVH storage which is dedicated for convenient ray tracing queries on modern GPU (our geometry compression is optimized for GPU ray tracing target unlike general compressed geometry/mesh representations by Deering and Chow [8] [9]). We spend an average of 15.5bytes per triangle for vertex coordinate and BVH data storage. This allows spending 750-950MB (for vertex coordinates and BVH) of GPU memory to ray trace 50-60 million triangle models (see Figure 1) which are common for wide range of CAD visualization tasks. Several features of NVIDIA CUDA platform [7] which are implemented in hardware are used to simplify and accelerate our data decompression method. Our basic approach is quantization of vertex coordinates in the limited space of bounding box.

2. ALGORITHM

Every node of the BVH contains the AABB and the link to the pair of children nodes (the BVH leaf contains the link to the primitives).

Figure 1: *The scene contains 58M triangles and is rendered on GTX 480 with 1.5GB memory: 3 area light sources, specular or diffuse reflections (the maximum ray path length is 3). The image is rendered progressively at 1024x768 (every image pass takes 130ms to render).*

As the rays are usually non-coherent this may result in a non-coherent memory access for the threads within the same CUDA warp. When the rays processed by the same CUDA warp access non-coherent data the best way to improve the memory access time is to access the data packed in float4 or int4 elements (basic 16byte CUDA types). The memory access time is further improved when we use CUDA textures [7]:

```
float4 elem = tex1Dfetch(texAABBdata, addr);
```

BVH data layout and fetching. We use the ray tracer implementation similar to the one by Aila and Laine [1]. The ray traverses through the binary BVH where two sibling nodes are stored together (the top root node is not stored; see Figure 2, left image). These 2 sibling nodes are sorted during ray traversal and the ray prefers to descend to the closest one until the leaf node is reached. Each AABB needs 6 words to encode the box, two sibling AABBs need 12 words to encode two boxes. During the ray traversal we fetch these 12 words using three float4 texture fetches (see Figure 2, array *Aabbs*) and then consider them as 2 boxes inside the "*ray-pair of boxes*" intersection.

Vertex data layout and fetching. An input triangle mesh is converted to a quadrilateral mesh using triangle connectivity information [3]. We process triangles one by one. Each triangle is connected to some neighboring triangle if they share a common edge and have the same material. From multiple candidates we select a pair of connected triangles that form a quad with the smallest perimeter. A quad mesh has 1.5 times less connectivity information (four vertex links per quad) than corresponding triangle mesh (six vertex links per two triangles).

Every new scene primitive has 4 vertices and one shared edge. Such a quad is stored as four consecutive 3D vertices (see Figure 2, array *Primitives*).

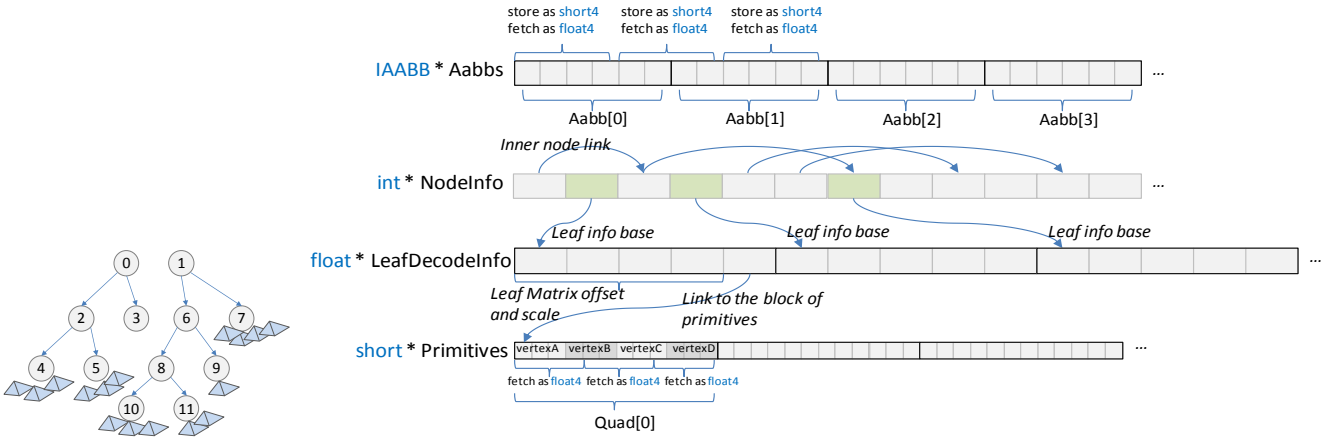

Figure 2: *Left: hierarchical view of the BVH without root; all the sibling nodes are stored together. Right: memory representation of the BVH and the list of primitives. Each i-th Aabb from Boxes array and i-th word from NodeInfo array forms the BVH node. The inner node (32nd bit of NodeInfo[i] equal to zero) refers to the pair of children nodes. The leaf node (green rectangle) has 32nd bit of NodeInfo[i] equal to one and refers to the block of five 4byte words of array LeafDecodeInfo. The first 4 words encode leaf decode matrix offset and scale, the fifth word is casted to integer value which refers to the block of several primitives united by this leaf.*

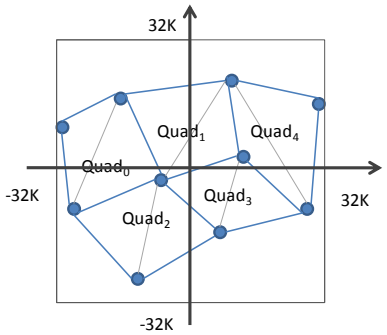

Figure 3: *Geometry Quantization. The BVH leaf is enclosed into a bounding cube and the vertex coordinates are quantized within the range -2^15..2^15.*

Vertex data compression. In the BVH generated using Surface Area Heuristic (SAH) [6] the leaves are well separated from each other. We assume that leaf extents are small enough compared to the whole mesh. All the quad vertices within the leaf are embedded into a 16-bit cube (coordinates ranging from -2^{15} to 2^{15}) and quantized (see Figure 3) using *Mencode* matrix (see the code for quad vertex compression in the *Listing 1*). We then linearize all quad vertices eliminating the connectivity information and vertex link indirection. Each quad now takes 24 bytes (i.e. 12 bytes/triangle). Decode information is stored per leaf: *float3 Center* and *float maxwidth* components of *Mdecode* matrix are stored in the first 4 words of the block allocated for this leaf in the array *LeafDecodeInfo* (see Figure 2).

Vertex decompression (decoding) is very fast. We use CUDA texture fetch whose mode is set to *cudaReadModeNormalizedFloat*:

```
// Bind the array of linearized quads to texQuad.
texture<short4,1,cudaReadModeNormalizedFloat> texQuads;
cudaBindTexture(0, texQuads, Primitives, numPrims * sizeof(QUAD_COMPRESSED));

// Read two connected triangles (shared edge is used in intersection test to
// reduce the number of cross/dot products
float3 A, B, C, D; {
  float4 q1 = tex1Dfetch(texQuads, 3 * quadIdx);
  float4 q2 = tex1Dfetch(texQuads, 3 * quadIdx + 1);
  float4 q3 = tex1Dfetch(texQuads, 3 * quadIdx + 2);
  A = make_float3(q1.x, q1.y, q1.z);
  B = make_float3(q1.w, q2.x, q2.y);
  C = make_float3(q2.z, q2.w, q3.x);
  D = make_float3(q3.y, q3.z, q3.w);
}
```

With this normalized fetch mode the float values are mapped from original short value range $-2^{15}..2^{15}$ to the values in the range *-1..1*.

The rays are specified in the world space. When we intersect them with the BVH-leaf they are transformed to the local leaf space (i.e. *-1..1*) using the *Center* and *maxwidth* components of the *Mdecode* which are stored per leaf in the *LeafDecodeInfo* array.

16-bit quantization (16bit vertex precision within the leaf) can encode 2^{48} voxels in a localized BVH-leaf 3D space (assumed to be relatively small compared to the world space). The amount of the leaves which can be placed in the row along any of the scene dimensions can determine the final precision of this vertex representation. If we can place 100-250 BVH-leaves along *x*, *y* or *z* scene dimension then the vertex precision would be 23-24bits in the world space. This precision depends on the size of the AABB of the average BVH leaf compared to the whole scene AABB.

```
Matrix4 CompressVertices(QUAD_COMPRESSED * outLeafQuads, QUAD * inLeafQuads,
                         int numQuads)
{
  AABB SBox = make_aabb();
  for(int i = 0; i < numQuads; i++) {
    Extend(SBox, inLeafQuads[i].A);
    Extend(SBox, inLeafQuads[i].B);
    Extend(SBox, inLeafQuads[i].C);
    Extend(SBox, inLeafQuads[i].D);
  }

  // Scale data
  float3 Center = (SBox.Min + SBox.Max) * 0.5f;
  float3 Size = SBox.Max - SBox.Min;
  float maxwidth = max(Size.x, max(Size.y, Size.z));

  // geometry is stored in a 16bit quantization
  // space in a cube [-32K,-32K,32K] .. [32K,32K,32K] with a center at [0,0,0]
  Matrix4 Mencode = scale(65534.0f / maxwidth) *
                    translate(-Center.x, -Center.y, -Center.z);

  // used to get the original position of object during ray traversal
  Matrix4 Mdecode = translate(Center.x, Center.y, Center.z) * scale(maxwidth);

  // optimized vertex coordinates
  // (shift and scale the vertices to the local compression space)
  for(int i = 0; i < numQuads; i++) {
    outLeafQuads[i].A = make_short3(transformPoint(Mencode, inLeafQuads[i].A));
    outLeafQuads[i].B = make_short3(transformPoint(Mencode, inLeafQuads[i].B));
    outLeafQuads[i].C = make_short3(transformPoint(Mencode, inLeafQuads[i].C));
    outLeafQuads[i].D = make_short3(transformPoint(Mencode, inLeafQuads[i].D));
  }

  return Mdecode;
}
```

Listing 1*: Code fragment for compressing quad vertices of the BVH leaf.*

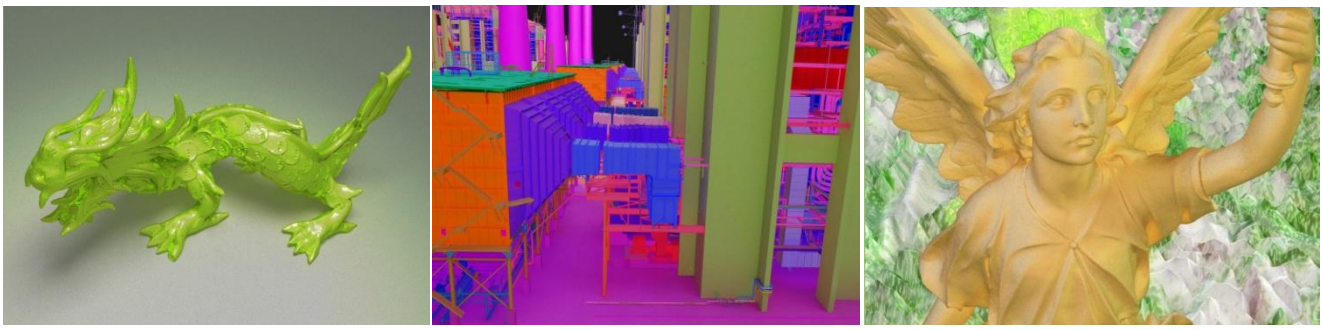

***Figure 4**: Dragon, PowerPlant and Lucy rendered with 3 light sources and enabled compression mode. No any visible artifacts/cracks.*

We have tested this simple vertex coordinate compression with a variety of large 3D models (CAD scenes, laser scanned objects, etc.) and have not observed any cracks between the leaves of the leaves of the BVH.

Although, the cracks are possible for small models which contain the primitives with large extents. Example: two primitives are connected in the original mesh. When the BVH is generated the primitives fall into different leaves. These leaves have independent bounding cubes (which are used for quantization, see Figure 3). For large extent primitives (compared to the scene extent) this can be the reason of visible cracks between the leaves of the BVH.

In our scenes we can mix the models which have 32-bit vertex precision or reduced vertex precision.

BVH data compression. For large models which contain several tens of millions of polygons we generate the SAH-based BVHs with up to 16 quads (equal to 32 triangles) per leaf. These "fat" leaves reduce the BVH size (the arrays *Aabbs*, *NodeInfo* and *LeafDecodeInfo* are smaller if the leaves have more primitives).

Speculative ray traversal [1] has more effect (compared to non-speculative) for the BVHs with "fat" primitives.

We also apply a 16bit quantization to the plane coordinates of the AABB. Each of the 6 plane coordinates is stored in a 2byte word. This encoding is done similarly to the vertex data encoding: all the world-space bounding boxes of the BVH-leaves are embedded into a global 16bit bounding cube (similar to the one on the Figure 3). All the BVH bounding boxes are stored using short data type. Ray traversal decodes the AABBs using the same normalized texture fetch mode which converts the data stored in short type to the float type.

With this quantization the short representation bounding boxes can be slightly extended compared to the boxes computed originally with float type. This extension may result in a few more ray-primitive intersection tests during the course of ray traversal.

3. RESULTS

We test the implementation of our compression method using CUDA 3.2, 64bit WindowsXP and GTX480 card (with 1.5GB of memory where only 1.3GB can be allocated with *cudaMalloc*).

The BVH is built offline using the Surface Area Heuristic (SAH) [6]. All the scenes are rendered with 1024x768 images resolution with 3 area light sources and 3 bounce rays.

Figure 1 represents the 58M triangle scene (a PowerPlant with 13M triangles, a Lucy model with 28M triangles, a Thai model with 10M triangles and a Dragon model with 7M triangles). In a compressed mode we spend 900MB for BVH storage (max 16 quads / leaf) and vertex data storage (which result in 15.5 bytes per triangle storage). Without this kind of memory optimization this kind of model could consume up to 2.7GB of storage which can't fit into GTX480 memory.

	non-compressed (4 triangles / BVH leaf)		compressed (quads + quantization + 16quads/BVH leaf)	
	Storage, MB	Render, ms	Storage, MB	Render, ms
Dragon,7M triangles	420	45	109	51
Thai, 10M triangles	600	47	155	55
PowerPlant, 13M triangles	780	97	201	110
Lucy, 28M triangles	1680	n/a	434	65
All together, 58M triangles	3480	n/a	900	130

***Table 1:** Non-compressed vs. compressed stats.*

In Table 1 we present comparison statistic for two modes: non-compressed (36 bytes for vertex data storage per triangle, 28 bytes for AABB storage, 4 triangles per BVH leaf) and compressed (grouping into quads, quantization and 16 quads per BVH leaf).

In the non-compressed mode each triangle requires up to 60 bytes to store the vertices and acceleration structure. Because of this we can't ray trace the models larger than 13M triangles on a 1.5GB graphics card.

With a maximum of 32 quads per primitive we can reduce the storage by 5% and decrease the ray tracing time by another 10%.

With a compression mode enabled we have 3.9x compression rate for the data storage and 10% slower ray tracing (if we compare render timings for PowerPlant and Dragon which both fit into GPU for both compressed/non-compressed modes). However, compressed-mode allows loading the model with 60-70M triangles to the GPU.

When compression mode is switched the amount of data to be accessed is reduced and there should be less influence of data access BW on the ray tracing time. But at the same time we have increased the number of primitives per BVH leaf from 2 quads (4 triangles) per leaf till 16 quads per leaf: the average number of ray-primitive intersection tests per ray was increased by 90%. Instead of 2x slowdown we got 10% slowdown for models that fit into memory (for both modes) because of the more utilization of GPU computation units per memory access units. This simple

experiment confirms that for GPU it can be better to store less and compute/recomputed more.

Potential improvement. The advantage of our approach is its simplicity compared to the hierarchical AABB compression [2], [5] which is harder to implement on GPU. Though, in the future we would like to experiment with such compression methods as well as with the larger primitive (not quad, but a triangle fan or "cluster" [3]).

Another interesting direction of future work would be to compress the dataset on the GPU as well as acceleration structure generation on the GPU.

4. CONCLUSION

In this paper we have presented a simple method to compress the vertex coordinates dataset and the bounding volume hierarchy by almost 4x and keep efficient ray tracing at the same time. This method allows rendering 4x larger models on the same GPU at low implementation cost.

5. AKNOWLEDGEMENTS

This work was supported by the Russian Leading Scientific Schools Foundation, project NSh- 8129.2010.9, RFBR, grants 10-01-00302.

6. REFERENCES

[1] Aila T., Laine S.: Understanding the Efficiency of Ray Traversal on GPUs. In Proc. of High Performance Graphics (2009).

[2] Fabianowski B. and Dingliana J.: Compact BVH Storage for Ray Tracing and Photon Mapping. Proceedings of the 9th Eurographics Ireland Workshop, Dublin, Ireland, 2009.

[3] Garanzha K.: The Use of Precomputed Triangle Clusters for Accelerated Ray Tracing in Dynamic Scenes. Proceedings of the Eurographics Symposium on Rendering 2009, Girona, Spain, 2009.

[4] Kay T., Kajiya J.: Ray tracing complex scenes. In Proc. of SIGGRAPH (1986), 269–278.

[5] Mahovsky J.: Ray Tracing with Reduced-Precision Bounding Volume Hierarchies. PhD thesis, University of Calgary, Calgary, Alberta, Canada, 2005.

[6] MacDonald J., Booth K.: Heuristics for ray tracing using space subdivision. The Visual Computer 6, 3 (1990), 153–166.

[7] NVIDIA. 2011. NVIDIA CUDA Programming Guide Version 4.0.

[8] Chow M.: "Optimized Geometry compression for real-time rendering", Proceedings on the IEEE Visualization'97.

[9] Deering M.: "Geometry Compression", Computer Graphics, 1995, 13-20.