GPU Octrees and Optimized Search

Daniel Madeira
Anselmo Montenegro
Esteban Clua
Computation Institute, UFF

Thomas Lewiner
Matmídia Laboratory, PUC-RIO

Abstract

Octree structures are widely used in graphic applications to accelerate the computation of geometric proximity relations. This data strucutre is fundamental for game engine architectures for a correct scene management and culling process. With the increasing power of graphics hardware, processing tasks are progressively ported of to those architectures. However, octrees are essentially hierarchical structures, and octree searches mainly sequential processes, which is not suited for GPU implementation. On one side, several strategies have been proposed for GPU octree data structure, most of them use hierarchical searches. On the other side, recent works introduced optimized searches which avoid hierarchical traversals.

In this work, we propose a GPU octree that allows for those optimized searches, which uses the GPU streaming to search for large of points at once. Moreover, we propose a parallelization of those optimized search to speed up the single point search.

Finally, the proposed structure takes advantage of the recent graphics hardware architectures to improve the GPU octree data structure.

Keywords:: Octree, GPU, Geometric Search, Hardware Acceleration

Author's Contact:

{dmadeira,anselmo,esteban}@ic.uff.br
lewiner@gmail.com

1 Introduction

A great amount of graphics algorithms rely on spatial proximity: collision detection, surface geometry approximation, particle-based fluid simulation, light ray-based rendering among many others. Brute-force search procedures to detect proximity relations typically induce a quadratic complexity, which is prohibitive for large data sets. Therefore, several optimizations have been proposed to accelerate those searches, most of them in a divide-and-conquer fashion: dividing the space into smaller blocks that can be processed independently. This division needs to be stored in memory, leading to a trade-off between memory consumption and search procedures acceleration.

Classical structures have been devised inside this trade-off: binary space partitions, k-d trees, octrees and multi-grids [Samet 1990]. Among them, the octree structure is certainly one the most widespread in image processing, geometric modeling, medical imaging, collision detection, point based rendering, isosurface visualization and volumetric rendering, among many other fields.

With the increasing power of graphics hardware (GPU), many graphics and non-graphics applications are ported to those parallelized stream-processing architectures, which is a very delicate but productive task [Buck et al. 2004; Zamith et al. 2008]. The ideal situation would be to handle all the application stage, and all of the geometry and rasterization stages on the GPU. However, several parts of the a typically application, such as user interaction or pure hierarchical traversals, are essentially sequential, requiring for a CPU implementation. This implies continuous CPU-GPU communications, which is often a bottleneck, since the data bus between CPU and GPU is limited.

This work proposes a GPU octree data structure that allows for optimized searches [Castro et al. 2008]. This structure is fully addressed in the GPU, reducing the CPU-GPU traffic.

2 Related Work

Recently, several works have proposed octree implementations on the GPU. Most of them represent the octree in the usual way of a general tree: each node has a reference for its eight children. The difference between them is how to reference the children of a node on the octree [Benson and Davis 2002; Lefebvre et al. 2005; Ziegler et al. 2007; Vasconcelos et al. 2008; Ajmera et al. 2008].

Some other representations have been proprosed, some of them replacing the explicit use of pointers to children by simple calculus of their location in an index table [Gargantini 1982; Glassner 1984; Warren and Salmon 1993; Lefebvre and Hoppe 2006; Bastos and Celes 2008].

Moreover, current GPUs offer much more efficient manipulation of data structures [Fatica and Luebke 2007; Nickolls et al. 2008] avoiding the restriction of using texture as mass memory, and this work introduces a simple way to efficiently take advantage of this evolution.

Usual searches in octrees proceeds in the divide-and-conquer manner: starting from the root node, the search decides at each node which child may contain a given location, and recurse on that child until reaching the leaves of the tree. This leads to an average logarithmic complexity of the search. Although sequential, this procedure is the base of most GPU octree proposals [Benson and Davis 2002; Lefebvre et al. 2005; Ziegler et al. 2007], or at least for the search of an isolated point [Vasconcelos et al. 2008; Bastos and Celes 2008]. Castro et al. [Castro et al. 2008] improved this search strategy for hash table representations of octrees. Instead of starting from the root, they begin searching at a given depth of the octree, and then traverse the octree up- or down-wards. A statistical optimization stated the initial depth to minimize the expected traversal steps, leading to an amortized constant time for the search. We use this strategy in this work.

Contributions

In this work, we propose a GPU octree search structure based on hash table that enjoys the new GPU architecture for the data structure and optimized search [Castro et al. 2008].

This optimized search strategy actually treats each search location and octree depth independently, which is optimal for the GPU parallel structure, and is the strength of our method. In particular, we can perform the search for a sequence of points in parallel, streaming the optimized search. We also introduce a parallel version of the optimized search, which allow to efficiently search for a single point at a time.

Our GPU octree search algorithm actually shows to be very fast. In the experiments reported in this work, our algorithm runs at least 3 times faster than the CPU algorithm, achieving, in some cases, an speed-up factor of 50.

3 Review of Octree Representations

An octree is a hierarchical data structure based on a recursive decomposition of a 3D region. Each node represents a cube in the region, and the root node represents the whole region. The cube of a non-leaf node is divided into 8 octants, thereby generating 8 children. In most applications, the data is stored in the leaves. In this section we present some common octree representations.

3.1 Pointer Octree

The most classic representation of octrees uses pointers, as a traditional tree. Usually, each node stores 8 pointers, one for each of its child, and some data. In leaf nodes, the pointers to the children are void, while in intermediate nodes, the data is void [Samet 1990]. A pointer from a child to his father can also be added, in order to facilitate upward traversal.

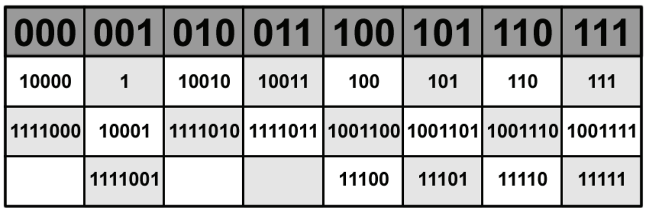
For octrees where each node is either a leaf or has exactly 8 children, it is possible to reduce the number of pointers by storing only a pointer to the first child and to the next sibling. However, this increases the traversal time.

3.2 Hashed Octree

It is possible to replace pointers by indexes. In that case, the references to a child node must be replaced by a calculus on the father's index, and the nodes must be stored in an index table.

Those structures are more compact that pointer octree, although depending on the pointer dereferencing time compared to the reference child index computation, it may be slower to access than pointer octrees.

However, they allow a direct access in constant time to any node of the octree, provided its index, while pointer octrees only allow direct access to the root.



000	001	010	011	100	101	110	111
10000	1	10010	10011	100	101	110	111
1111000	10001	1111010	1111011	1001100	1001101	1001110	1001111
	1111001			11100	11101	11110	11111

Figure 1: *Hash representation of the quadtree. The hash function uses a* 3 *bits key (top row) to group the tree nodes (bottom rows).*

The index can be generated *ad hoc* for a static octree in order to reduce the index table, as proposed on GPU through perfect hashing [Lefebvre and Hoppe 2006; Bastos and Celes 2008]. However, any significant change in the octree structure implies a complete rebuilding of the indexes, and the child references calculus are replaced by extra memory storage.

The index can also be systematically generated from the node geometrical position and/or by the node position in the octree hierarchy. This is the representation used in this work. A common choice for such index is Morton codes, reviewed at the next section. For usual octrees, those indexes are not consecutive, and thus the hash table must group them to avoid wasting too much memory. A common strategy is to group nodes that have the same last k bits of the Morton code, with $k = \lceil \log(n) \rceil + 1$, where n is the number of nodes (see Figure 1). In other words, the hash key h assigned to a morton code m is defined by $h(m) = m \mod k$.

3.3 Morton Index Generation

For spatial ordering of the nodes and to generate the indexes for the hashed octree, we use the Morton code. This method is efficient to generate unique index for each node, while offers god spatial locality and easy computation. Another advantage of Morton code is their hierarchical order, since it is possible to create a single index for each node, while preserving the tree hierarchy.

The index can be calculated from the tree hierarchy, recursively when traversing the tree. The root has index 1, and the index of each child node is the concatenation of its parent index with the direction of their octant, coded over 3 bits. The bottom-up traversal is also possible, as if to find the parent index we only have to truncate the last 3 bits of a child index.

The index can equivalently be computed from the geometric position of the node's cube and its size. Considering that the root's cube is a unit cube, and denoting by (x, y, z) the coordinates of the cube center and by 2^{-l} the cube side, i.e. the node is at depth l, we can generate the Morton code by:

$$1 x_l y_l z_l x_{l-1} y_{l-1} z_{l-1} x_{l-2} y_{l-2} z_{l-2} \ldots x_1 y_1 z_1 \ , \qquad (1)$$

where $x_l x_{l-1} \ldots x_1$ is the binary decomposition of $\lfloor 2^l x \rfloor$.

The generation of this index can be accelerated using integer dilation and contraction [Stocco and Schrack 1995].

4 Our GPU Octree Structure

We use hash table for the GPU representation of the Octree. This involves two specific design choices: the GPU memory used and the hash collision handling.

4.1 GPU Storage

Recent graphics hardware use CUDA architecture, which greatly improves the storage on GPU. CUDA stains for Computer Unified Architecture, and brings a new parading for memory distribution among the GPU. The texture memory can be addressed as a global memory for a GPU code, with linear and random addressing from the threads and blocks of each internal grid. This allow more efficient implementations of applications not related to the graphical pipeline and much more efficient octree implementations than presented in [Benson and Davis 2002], [Lefebvre et al. 2005] and [Ziegler et al. 2007].

4.2 Collision handling

A collision in the hash table occurs when two nodes have the same hash key. A simple option is to leave colliding nodes in the same entry of the hash table (open hashing). However, this requires either to allocate enough space in *all* the entries of the hash table to contain the maximal number of colliding nodes, which would require a static or controlled data and would waste a lot of memory, or to let a variable size structure in each entry of the hash table. This last option is commonly used in CPU implementation, since variable size container are easily implemented with pointers. However, this is much less efficient on GPU.

Another way to avoid collisions is to keep one of the colliding nodes in the position assigned by its hash key h, and to place the other ones at empty positions in the hash table (closed hashing). Those empty positions must be systematically chosen, by a collision function $c(h)$, to ensure that we can retrieve the other nodes!

Looking for a node n then resumes to looking at the position of its hash key. If the position is empty, then node n does not belong to the hash table. If not empty and if the node at the hash key position is different from n, then we look at position $h' = c(h)$. We repeat until n is found or a empty node is achieved. In this work, we use a linear colliding function, i.e. $c(h) = h + c_0$ for a fixed c_0.

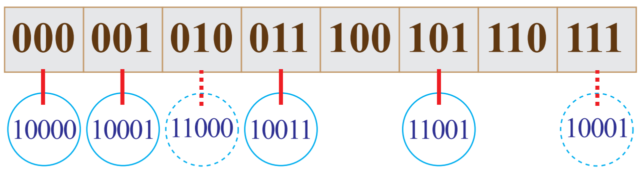


Figure 2: *Closed hashing: a colliding entry is sent to a new location, here its original location shifted by 2.*

5 Octree Search Algorithms

A direct search procedure in an octree returns the leaf whose cube contains a given position in space. In this section, we will recall usual algorithms for such search procedure, and introduce our procedure for our GPU octree.

5.1 CPU Algorithms

In pointer octrees, a search can only be done starting from the root node and traversing hierarchically the tree until the desired leaf is reached. This method has complexity of $\log_8(n)$, and $O(n)$ at worse case. The algorithm below shows a search in a pointer octree. In this method, the position and size of each traversed cube can be directly deduced from the recursion.

Algorithm 1: Classical search for point p.

```
1  start with the root $r$ and a $c$ a unit cube : $n = r$ ;
2  while $n$ is not a leaf do
3      retrieve the child $n'$ of $n$ containg $p$ ;
4      set $n = n'$ and $c$ the octant of $n'$ ;
5  end
6  return $n$;
```

Optimized searches offer a different access method. Since the leaves are the most distant nodes from the root node, it is better to start from a node closer to the desired leaves than from the root node.

However, to access a random node in the hashed octree, we need its Morton code, computable from position and depth. We know the position p from the search input, but the depth must be estimated. The optimized search [Castro et al. 2008] proposes to estimate this depth by the weighted median $\hat{l}$ of the expected depth, since it minimizes the number of traversal operations.

Algorithm 2: Optimized search for point p.

```
1  compute Morton code $m_{\max}$ of $p$ at maximal depth;
2  compute code $m$ of $p$ at depth $l = \hat{l}$ from $m_{\max}$ ;
3  access the node $n$ corresponding to $m$ in the hash table;
   // upward traversal, in case $n$ is below the leaf
4  while $m$ does not belong in the hash table do
5      decrease the depth $l$, removing 3 bits from $m$;
6      access the parent of $n$ in the hash table with $m$;
7  end
   // downward traversal, in case $n$ is not a leaf
8  while $n$ exists in the hash table do
9      increase the depth of $m$, adding 3 bits of $m_{max}$;
10     access the child of $n$ in the hash table with $m$;
11 end
12 return $n$ as the last valid access to the table ;
```

5.2 GPU Algorithms

The main contribution of this work is to parallelize the optimal search algorithm, presented in the previous section, to port it to GPU architecture. To do so, we take advantage of the fact that the search for each point is independent and, that, in practical applications, e.g. collision, many queries are needed simultaneously.

We first describe how to parallelize the search for a single point p given g available threads. The main idea is to search at many levels in parallel. Since we know from the statistical optimization that the leaf is probably near level $\hat{l}$, we concentrate the threads around that level. In the CPU implementation, the upward and downward traversals test 1 level up or down. Here, to avoid duplicate efforts, the upward traversals are handled by the first half of the thread, skipping $\frac{g-1}{2}$ levels instead of 1, and the downward traversals. Although the CPU search can be as fast as the GPU search, the advantage of this method is fully maintain the octree in the GPU

Algorithm 3: GPU search for point p given g threads.

```
1  compute Morton code $m_{\max}$ of $p$ at maximal depth;
2  for $iter \in 1..iter_{\max}$ do
3      foreach thread $t_i \in (0 \cdots g-1)$ do
4          if $t_i \leq \frac{g-1}{2}$ then
5              assign $l_i = \hat{l} + t_i + iter \cdot \frac{g-1}{2}$ ;
6          else
7              assign $l_i = \hat{l} + t_i - (iter + 2) \cdot \frac{g-1}{2}$ - 1 ;
8          end
9          compute code $m$ of $p$ at depth $l_i$ from $m_{\max}$ ;
10         access the node $n$ corresp. to $m$ in the hash;
11         if $n$ is a leaf then return $n$ ;
12     end
13 end
```

Observe that, in the CPU implementation, a leaf was detected testing for invalidate downward traversal. Here, we need to explicitly store for each node if it is a leaf or not. In practical applications, where the leaves are the only nodes containing data, this does not induce any memory overhead.

Now, computing the search for many points at the same resumes to divide the number of available threads by the number of points to search for, and use the above algorithm in parallel for each point.

6 Experiments and Results

We implemented the above algorithms inside the CUDA architecture, using the hashed octree with closed hashing. To handle the colisions, we used the linear colliding function. Although there could be more optimized methods, we chose it for its simplicity.

In our experiments, we used five models, at different resolutions. The results were obtained on a Core 2 Duo T9400 CPU, running at 2.53GHz, with a GeForce 9600M GT, with 32 processors.

We first tested our approach by searching for random points, and compare the timings between the CPU and GPU implementations. We searched simultaneously for 10 to 300 points.

The absolute timing comparisons are reported on Table 3, our algorithm achieved an execution time at least 3 times faster than CPU algorithm for a small number of searches. When searching for 300 points, our algorithm runs in average 45 times faster.

On the detailed results reported on Tables 1 (CPU) and 2 (GPU), we clearly see that the increase of the number of search points generates a linear increase of the execution time with a high proportion, while the results on the GPU shows a clear benefit of our parallelization.

In Figure 3 we can observed that the execution time of our algorithm grows slowly, in a linear pattern.

Table 1: *Total execution time (in seconds) for the CPU algorithm.*

# points	**10**	**50**	**100**	**200**	**300**
Ant	1	2	5	10	14
Armadillo	1	4	8	17	25
Bunny	1	3	7	14	21
Drill vripped	1	2	4	6	10
Drill zip	1	3	7	15	21

We further test for the influence of the hash table size. While in the first test, the octrees of all models and all implementations were stored in a hash table with the same size, in this second test, we varied the size of the hash table. In the results are presented in Table 4, we check that the influence of the size on our GPU algorithm is little, while keeping it big enough to contain the whole octree.

Table 2: *Total execution time (in seconds) for our GPU algorithm.*

# points	10	50	100	200	300
Ant	0.27	0.28	0.36	0.39	0.43
Armadillo	0.29	0.29	0.38	0.41	0.44
Bunny	0.28	0.35	0.37	0.4	0.37
Drill vripped	0.28	0.35	0.37	0.34	0.37
Drill zip	0.28	0.28	0.21	0.37	0.38

Table 3: *Speed-up of our algorithm for the test cases. Our algorithm runs at least 3 times faster than the CPU algorithm.*

	10	50	100	200	300
Ant	3,70	7,14	13,89	25,64	32,56
Armadillo	3,45	13,79	21,05	41,46	56,82
Bunny	3,57	8,57	18,92	35,00	56,76
Drill vripped	3,57	5,71	10,81	17,65	27,03
Drill zip	3,57	10,71	33,33	40,54	55,26

7 Conclusion

This work presented a new GPU approach for searching elements in octrees, based on hash table representation and optimized search. The solution is based on the fact that, in such contexts, each octree level is independent from the others. We proposed a parallelization of the search algorithm, which lead to a totally streamed implementation. This method combines the memory efficiency of the hashed octree with a significantly smaller execution time thanks to the structure of recent graphics hardware.

Many diferent applications can benefit from this speed-up, since it can be implemented to any scene structure that supports octree representation.

As future works, more efficient hash methods can be analyzed and used with the presented strategy. It is also important to validade the proposed approach for complete applications, such as real time 3D collision detection.

Also with the 2 searchs presented, we can now implement the octree generation on the GPU.

Table 4: *Execution times (in seconds) for the Ant model, with different sizes for the hash table.*

	Size of the hash table				
	2,000,000	1,000,000	500,000	250,000	100,000
CPU	5 s	2 s	1 s	1 s	1 s
GPU	0.33 s	0.29 s	0.28 s	0.28 s	0.27 s

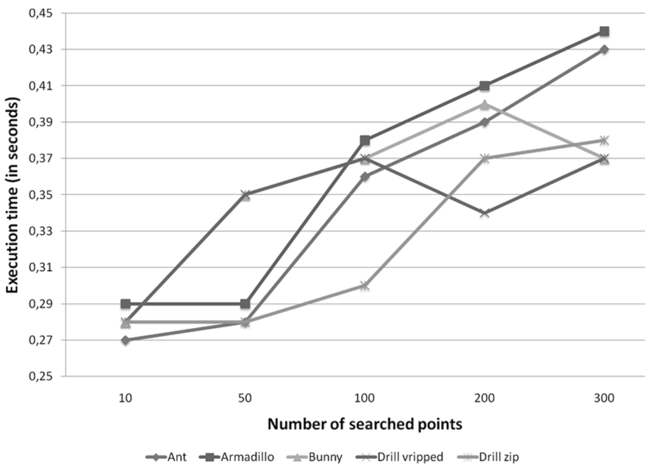


Figure 3: *Execution time of the GPU results of Table 2. The graph shows the linear complexity of our algorithm.*

References

AJMERA, P., GORADIA, R., CHANDRAN, S., AND ALURU, S. 2008. Fast, parallel, GPU-based space filling curves and octrees.

BASTOS, T., AND CELES, W. 2008. GPU-accelerated Adaptively Sampled Distance Fields. In *IEEE International Conference on Shape Modeling and Applications, 2008. SMI 2008*, 171–178.

BENSON, D., AND DAVIS, J. 2002. Octree textures. In *Siggraph*, vol. 21, 785–790.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: stream computing on graphics hardware. In *Siggraph*, 777–786.

CASTRO, R., LEWINER, T., LOPES, H., TAVARES, G., AND BORDIGNON, A. L. 2008. Statistical optimization of octree searches. *Computer Graphics Forum 27*, 1557–1566.

FATICA, M., AND LUEBKE, D., 2007. High performance computing with CUDA. Supercomputing 2007 tutorial. In Supercomputing 2007 tutorial notes, November.

GARGANTINI, I. 1982. Linear octrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing 4*, 20, 365–374.

GLASSNER, A. 1984. Space subdivision for fast ray tracing. *Computer Graphics & Applications 4*, 10, 15–22.

LEFEBVRE, S., AND HOPPE, H. 2006. Perfect spatial hashing. In *Siggraph*, 579–588.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. Octree textures on the GPU. In *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, ch. 37, 595–613.

NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. 2008. Scalable parallel programming with cuda. *Queue 6*, 2, 40–53.

SAMET, H. 1990. *The design and analysis of spatial data structures*. Addison-Wesley.

STOCCO, L., AND SCHRACK, G. 1995. Integer dilation and contraction for quadtrees and octrees. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing, 1995. Proceedings*, 426–428.

VASCONCELOS, C., SÁ, A., CARVALHO, P., AND GATTASS, M. 2008. Quadn4tree: A gpu-friendly quadtree leaves neighborhood structure. In *CGI: Computer Graphics International*. 1101.

WARREN, M. S., AND SALMON, J. K. 1993. A parallel hashed octree n-body algorithm. *Supercomputing, IEEE*, 12–21.

ZAMITH, M., CLUA, E. W. G., PAGLIOSA, P., CONCI, A., VALENTE, L., FEIJO, B., LEAL, R., AND MONTENEGRO, A. 2008. The GPU used as a math co-processor in real time applications. *Journal of Computer in Entertainment: CIE 6*, 1–19.

ZIEGLER, G., DIMITROV, R., THEOBALT, C., AND SEIDEL, H. 2007. Real-time quadtree analysis using HistoPyramids. In *IS&T and SPIE Conference on Electronic Imaging*, vol. 6496. 0L.