# Fast Parallel Construction of Stack-less Complete LBVH Trees with Efficient Bit-trail Traversal for Ray Tracing

Arturo García [*‡] Sergio Murguia [†] Ulises Olivares [‡] Félix F. Ramos [‡]

[*]Intel Corporation [‡]CINVESTAV México

## Abstract

This paper presents an efficient space partitioning approach for building high quality Linear Bounding Volume Hierarchy (LBVH) acceleration structures for ray tracing. This method produces more regular axis-aligned bounding boxes (AABB) into a complete binary tree. This structure is fully parallelized on GPU and the process to efficiently parallelize the construction is reviewed in detail in order to guarantee the fastest build times. We also describe the traversal algorithm that is based on a stack-less bit trail method to accelerate the frame rates of rendering of 3D models in graphics processing units (GPU). We analyze diverse performance metrics such as build times, frame rates, memory footprint and average intersections by AABB and by primitive, and we compare the results with a middle split and SAH (surface area heuristic) splitting method where we show that our structure provides a good balance between fast building times and efficient ray traversal performance. This partitioning approach improves the ray traversal efficiency of rigid objects resulting on an increase of frame rates performance of 30% SAH and 50% faster than middle split.

**CR Categories:** I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

**Keywords:** Ray tracing, stack-less bounding volume hierarchy, bit-trail traversal algorithm, radix sort, acceleration structures, data parallel computing, GPGPU

## 1 Introduction

Fast construction algorithms are needed to rebuild acceleration structures from scratch in order to produce dynamic scenes. However, it is important to find the trade-off between fast construction times and ray traversal performance. The use of sophisticated heuristics to estimate the cost of ray-traversal has a negative impact in the construction times. Nonetheless, fast split methods tend to have the fastest construction times but a poor traversal performance. BVH based implementations use SAH [Goldsmith and Salmon 1987] for building the data structure in which the original scene $S$ is recursively partitioned using a greedy strategy for minimizing the expected traversal cost [MacDonald and Booth 1990] and building times of $O(NlogN)$ [Wald et al. 2009].

This paper proposes a new method for building Linear Bounding Volume Hierarchy (LBVH) acceleration structures for ray tracing based on a splitting approach that produces a regular data structure into a complete binary tree. This partitioning method offers fast construction times of high quality BVH structures and offers efficient traversal performance.

Throughout this paper, the splitting method is described and then the fast parallel construction of the acceleration structure on GPU. We review in detail the optimization strategies based on the use of high data parallel computing algorithms. It also describes the stack-less traversal and we go through the insights of how this algorithm helps to accelerate the traversal performance.

## 2 Related Work

Several acceleration structures have been proposed in order to decrease construction times and to obtain the highest frame rates. Nevertheless, these structures have a trade-off between fast construction times and ray traversal performance.

Pantaleoni et al. [Pantaleoni and Luebke 2010] proposed a hierarchical grid decomposition of the LBVH algorithm [Lauterbach et al. 2009] to exploit spacial coherence of the 3D model minimizing the memory bandwidth usage and producing a compact structure. This algorithm presented faster construction times than the original proposed by Lauterbach.

Garanzha et al. [Garanzha et al. 2011] extended and made some improvements of HLBVH [Pantaleoni and Luebke 2010], proposing an algorithm that employed work queues. This approach accelerated the overall construction speed by a factor of 5 to the 10×. The main contribution was focused on replacing breadth-first tree traversal primitives used to perform the object partitioning with a single pipeline based on efficient work-queues.

Wald in [Wald 2010] designed a framework for the fast construction of SAH BVH that uses an architecture designed by Intel, Many Integrated Core (MIC) Architecture to achieve performance improvements that competed with the best implementations in GPU and CPU architectures [Pantaleoni and Luebke 2010] [Garanzha et al. 2011].

Bauszat et al. [Bauszat et al. 2010] designed a representation of a BVH using the lowest memory possible per node. This method reduced the memory consumption to less than 1 percent of a standard BVH, but it impacted and decreased the traversal performance.

Hapala et al. [Hapala et al. 2013] implemented a traversal algorithm for BVH that did not need a stack but stores a parent pointer for each node to enable the traversal back upwards in the tree.

Zhefu et al. [Zhefu et al. 2013] presented a CPU implementation based on BVH partition, which could remove unnecessary rays in subspace which increased the traversal performance.

Sulaiman et al. [Sulaiman1 and Bade 2011] implemented a median splitting method to build balanced BVH structures for collision detection that finds the midpoint of the corresponding AABB by using

[*]arturo.garcia@intel.com
[†]fory_murguia@hotmail.com
[‡]garcia, uolivares, framos@gdl.cinvestav.mx


VRCAI 2014, November 30 – December 02, 2014, Shenzhen, China.

http://dx.doi.org/10.1145/2670473.2670488

the longest axis. The splitting method used here is similar to the approach presented in this work, but the splitting mask used to cut the axis is modeled for ray tracing efficiency.

Murguia et al. [Murguia et al. 2013] designed a stack-less LBVH that used a middle split approach based on the Morton code of primitives [Lauterbach et al. 2009], this method gives fast construction times but poor traversal performance.

As we reviewed in this section, the effort has been focused on improving one of the three performance areas: construction times, memory footprint or traversal efficiency. The problem to find the balance between these three factors in order to provide the best performance for ray tracing of dynamic scenes in real time is still latent.

## 3 Parallel Construction

This section addresses a new approach for the construction of LBVH structure using a complete tree. This section is divided in three main parts. The first part describes a set of rules to determine the number of primitives in each node, this approach guarantees the resulting structure will be a complete tree. The second part describes the node structure of the tree. Finally, the third part depicts in detail a new partitioning scheme.

### 3.1 Complete Binary Tree

In the Stack-less LBVH approach like presented by Murguia et al. [Murguia et al. 2013], one of the main problems is that the resulting tree is not well balanced and in order to navigate through it efficiently, it must preserve memory regions that are not used. This causes a lot of memory waste and in some cases the number of actual nodes used can be as low as $15\%$. In order to prevent this, a tree was created in which every level is filled completely with the exception of the last level and in that level all the nodes will be as far left as possible. These kind of trees are known as complete binary tree [NIST 2013], Figure 1 is an example.

By using complete binary trees, the number of nodes required is simply $2 * N$ where $N$ represents the number of primitives, but in order to have a complete tree, it is necessary to determine how many primitives will be stored on every branch of the tree ensuring that only the deepest level could be incomplete. It can be observed in Figure 1 that the subtree in every internal node is also a complete tree. Furthermore, the number of leaf nodes on the left branch is never lower than the number of leafs on the right side.

Assuming that the number of leaf nodes $N$, is greater than 2, where $k$ represents the radix of the most significant bit of the binary representation of $N$, and $r$ represents the complement $r = |N| - 2^k$ (see Figure 2). Then, we can have two different cases: the binary representation of $N$ is $10...{}_2$ or is $11...{}_2$. In the first case, there are not enough leafs to fully complete the left side and thus the right side should be full, if $N = 2^k + r$ then we should have $2^{k-1} + r$ leafs on the left side and $2^{k-1}$ on the right side, see node 3 of the tree in Figure 1. In the second case, there are enough leafs to complete the left side and thus will have $2^k$ leafs while the right side will hold $r$ leafs, for this case, $r \geq 2^{k-1}$; the node 1 of Figure 1 is an example of this case.

The process just described provides the rules to determine the spatial partition (cut) in a given axis, it is represented by these two cases:

$$|N| = 2^k + r = \begin{cases} 10...{}_2, & (1) \\ 11...{}_2, & (2) \end{cases}$$

$$10...{}_2 = \begin{cases} 2^{k-1} + r, & \text{for } left \quad (3a) \\ 2^{k-1}, & \text{for } right \quad (3b) \end{cases}$$

$$11...{}_2 = \begin{cases} 2^k, & \text{for } left \quad (4a) \\ r, & \text{for } right \text{ with } r \geq 2^{k-1} \quad (4b) \end{cases}$$

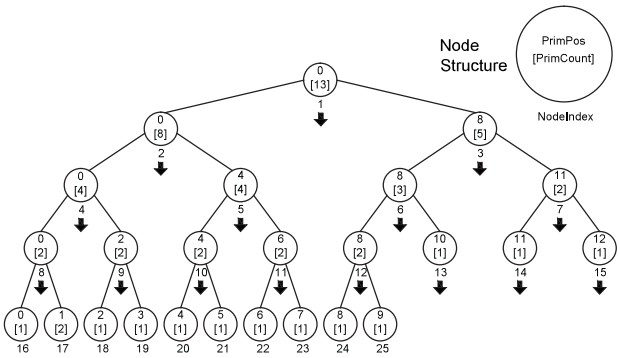



**Figure 1:** *Complete Binary Tree.*

The complete LBVH is a linear representation of a complete binary tree that is built with the split approach described above. The tree is stored in breadth-fist order in an array where the resulting tree is a heap. This linear representation benefits from more compact storage and better locality of reference. In a heap, the left child of node $n$ is located at $2n$ and the right child is at $2n + 1$.

While traversing, the next node to be visited is computed using arithmetic operations. A 32-bit integer is used to store the trail of each ray, which allows restarting the traversal from the last node hit, instead of the root [Laine 2010]. Also, using the direction of the ray, the trail can decide whether the left or right child must be visited first.

Besides the common properties of a heap, the linear binary tree representation possesses the following characteristics:

- The root node is at position $i = 1$. The node at position $i = 0$ is invalid. In this way, the arithmetic operations to traverse the tree are simpler.
- The first right ancestor of node $n$ is found by removing all the least significant bits that are 0 in the binary notation of $n$.
- The binary notation of an auxiliary 32-bit integer (the "trail") can be used to substitute the stack (commonly used to store the visited nodes during ray traversal). Each bit represents one level in the tree. A zero means that the level still has unvisited nodes while a number one means that all nodes in that level have already been visited.

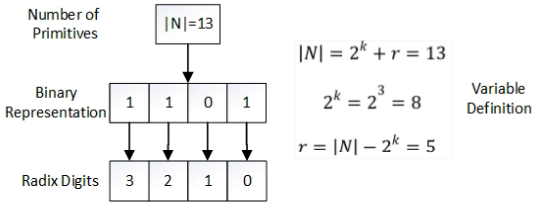



**Figure 2:** *Partitioning scheme using the rules described above.*

### 3.2 Node Structure

The size of each node is 32 bytes. Six 32-bit floats are stored per node representing an AABB and two 32-bit integers storing the total of primitives in the AABB (PrimCount), and a primitive offset (PrimPos). Additionally, PrimCount stores the split axis, and whether the node is leaf or internal as shown in Figure 3. The axis is stored on the two least significant bits of PrimCount and the node flag ($leaf = 1, internal = 0$ ) on its third least significant bit.

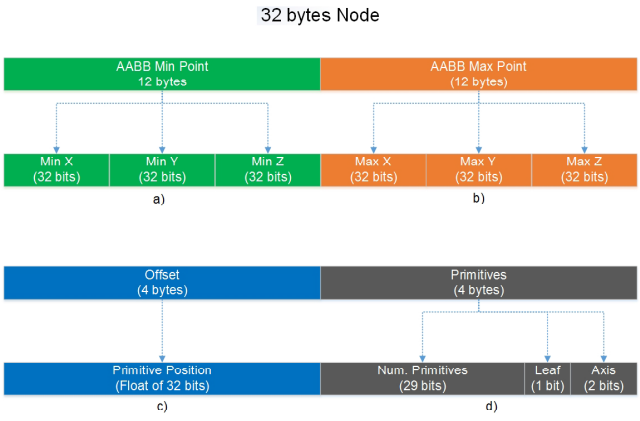



**Figure 3:** *Node Structure composed by 32 bytes. a) represents the min point of each axis of the AABB. b) represents the max point of the AABB. c) represents the position of the primitive in the tree. d) represents the number of primitives, if the node is a leaf and the split axis.*

In this phase the complete binary tree is built, this process is straightforward since we know the total of primitives of the 3D model to be rendered, then we can create a complete tree following the rules described in the equations 1 and 2 which will correspond to the size of the cut in $x$, $y$ or $z$ axis for each AABB. The process initiates computing in parallel the centroids of the primitives and the AABB's, initializing the root node of the tree with the total number of primitives in the 3D model and the primitive offset to 0, then each level of the tree is processed top-down in parallel, this process is done in $log_2 N$ cycles for $N$ primitives. The algorithm stores two types of nodes: internal node and leaf.

### 3.3 Space Partitioning

Once we have selected the number of primitives that every side will have, the next step is to decide which primitives will go left and which will go to the right. To do so, we find the lowest and highest values of the coordinates of the primitive centers using the axis that has the longest range, we select the primitives with the lowest value on that axis to be in the left side and the rest on the right side. Geometrically, we are selecting the longest side of the bounding box that contains all the primitive centers, and splitting the primitives based on the order given by the projection of the centers on that side. Using this method will generate boxes that tend to be more regular, by reducing the length of the longest side.

Once the complete tree is built we know how many primitives will be contained in each node, then we need to perform the space partition in order to assign the primitives to each node. The space partitions are executed on the longest axis in order to improve the traversal performance by localizing the major density of adjacent primitives in the left branch of the tree. Figure 4 shows the space partitions of primitives in a space of 2 dimensions.

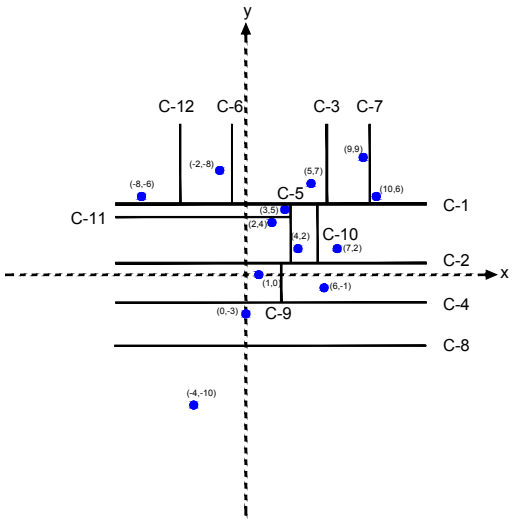



**Figure 4:** *Space Partitioning*

The space partitioning is done following these steps:

1. Find the longest axis
2. Store split axis
3. Sort primitives
4. Split primitives
5. Assign primitives to each sibling

#### 3.3.1 Find the Longest Axis

This is determined by doing a parallel reduction based on an algorithm presented by Hillis et al. [Hillis and Steele 1986] to sum an array of $n$ numbers that can be computed in time $O(logn)$ by organizing the addends at the leaves of a binary tree and performing the sums bottom-up at each level of the tree in parallel. Our function returns the longest axis by calculating the $max$ and $min$ values in $x$, $y$ and $z$ of an array of primitives, then the function calculates the distance between the $max$ and $min$ points for each axis and returns the axis that has the longest distance. Figure 5 shows the parallel process to get the max and min values of an array of numbers.

#### 3.3.2 Store Split Axis

Once the longest axis is determined, it is stored in the two least significant bits of the PrimeCount register of the node structure: $x = 00$, $y = 01$ and $z = 10$.

#### 3.3.3 Sort Primitives

A fast data parallel radix sort implementation is used to sort the primitives against the split axis in ascending order based on the centroids computed on the pre-build stage.

Due to the linear representation of the tree is a heap, the nodes with the same number of primitives are localized and can be sorted in parallel independently of its level in the complete binary tree, for example in Figure 6 the nodes 8,9,10,11 and 12 of level 3 and node 7 of level 2 are sorted in parallel. This can be done modifying the radix sort algorithm to perform segmented scans [Blelloch 1990] where each segment is able to sort against a different axis. In the example previously described, the nodes 8 to 11 are sorted against

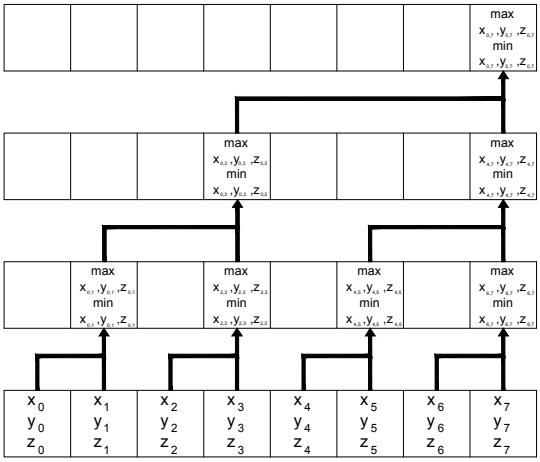



**Figure 5:** *Parallel reduction algorithm to find the min and max points in an axis*

$x$ and node 7 and 12 in $y$ axis. The soring groups can be easily identify by analyzing the binary tree array that was created during the pre-build phase, Figure 6 shows the linear representation of the array and depicts how the nodes with the same number of primitives are grouped in a dispatch, reducing the number of kernel dispatches.

Since the number of levels on the tree is $\lceil \log_2 N \rceil$, building the tree with $N$ primitives has an expected complexity of $O(N \log N)$.

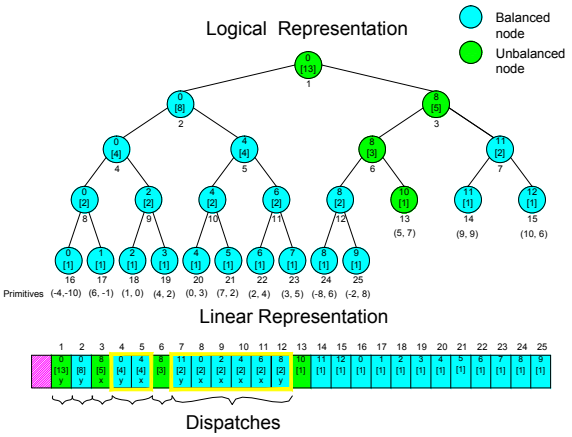



**Figure 6:** *CLBVH Data Representation*

#### 3.3.4 Split Primitives

This step is straight forward because the split was calculated previously when the tree was pre-built to determine the number of primitives for each sibling node, for example, in Figure 7 we have an array of 13 primitives in the root node to be divided to create the sibling nodes, then we apply the partition process described in the equations 1 and 2, we get that the first 8 primitives of the sorted array will be assigned to the right node and 5 to the left node (see nodes 2 and 3).

#### 3.3.5 Assign Primitives to Each Sibling:

Finally the sorted primitives are assigned, to the right and left sibling nodes according to the number of primitives calculated in the previous step

Figure 7 shows the parallel building process described in this section.

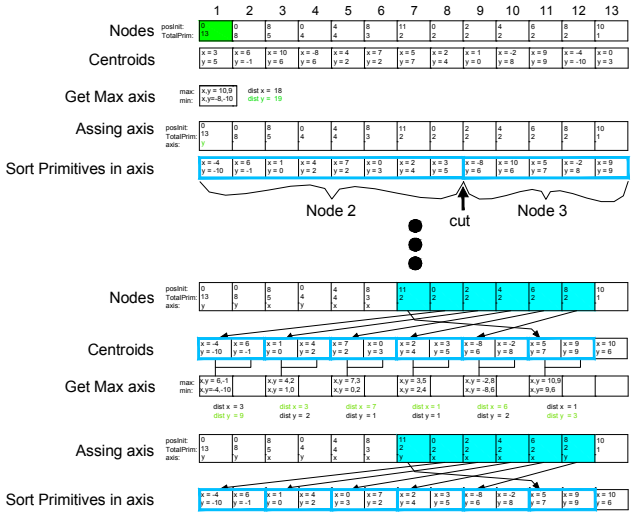



**Figure 7:** *Parallel Construction Process*

## 4 Efficient Bit-traversal

This section presents an efficient bit-traversal algorithm that is a variation of the bit-trail algorithm presented by Murguia et al. [Murguia et al. 2013] where a 32-bit trail is used instead of a stack and ray traversal restarts on the last node hit instead of the root. This algorithm was selected due to its effectiveness during traversal and its linear representation. Three main modification were done to the original bit-trail traversal algorithm:

1. It is not needed to track the sign of the ray along the current split axis
2. Ray checks for intersections in the left and right sibling nodes (AABB's) at a time
3. The algorithm doesn't allow empty nodes

The ray checks for intersection in the left and right boxes, this increases the box intersection but it decreases the primitive intersections by 50% which increase the frame rate performance. The trail is a 32-bit integer. Each bit in the trail represents a level on the tree with the following meaning:

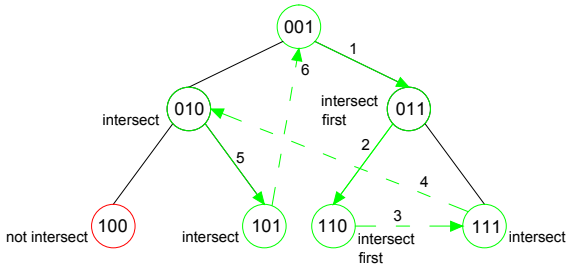
1. $bit = 0$: The level on the current subtree contains unvisited nodes
2. $bit = 1$: All nodes in the current level of the subtree have been visited

The algorithm starts with both $nodeNum$ and $trail$ set to 1. After the first intersection is found, the traversal stores the intersected nodeNum and it is passed to the next traversal iteration. The trail pushes nodes when an internal node is intersected and pops nodes when either an intersection was not found or when a leaf node is intersected. A push is computed using a bitwise shift-left operation and a pop using a bitwise shift-right operation. Figure 8 illustrates a walk trough example of a "Complete Stack-Lest LBVH" where each node in the tree is labeled with its index (nodeNum) in binary notation and the implementation in GPU is shown in listing 1.

```
...
unsigned int nodeNum = 1;
int trail = 1 << (firstbithigh(nodeNum)));

[allow_uav_condition]do
{
  int hitBoxes = 0;
  const int p = firstbitlow(trail + 1);

  if(isLeaf)
  {
    const int iPos=uNodes[nodeNum].iPos;

    ...

    //pop
    trail = (trail >> p) + 1;
    nodeNum = (nodeNum >> p) ^ 1;
  }
  else
  {
    result+=2;
    nodeNum = (nodeNum << 1);

    //Check if nodes intersects with the ray
    hitL=IntBox(ray.vfOrigin,invDir,nodeNum);
    hitR=IntBox(ray.vfOrigin,invDir,nodeNum+1);

    //Check if left node intersects with the ray
        intersection better than current one &
       it is not behind the camera
    if((hitL[0] <= bInt.fT)&&(hitL[1] >= 0.0f))
      hitBoxes++;

    //Check if right node intersects with the
       ray
    if((hitR[0] <= bInt.fT)&&(hitR[1] >= 0.0f))
    {
      hitBoxes++;

      //Check if only right node intersects with
          the ray
      if((hitBoxes == 1) || (hitR[0] < hitL[0]))
        nodeNum++;
    }
    if(hitBoxes > 0)
      trail = (trail << 1) | (hitBoxes & 1);
    else {
      //does not intersect, change to next node
      trail = (trail >> p) + 1;
      nodeNum = (nodeNum >> (p+1)) ^ 1;
    }
  }
}
while (trail > 1);
...
```

**Listing 1:** *Bit-Trail Traveral.*

During the downward traversal, the algorithm checks which box is hit first by the ray. If the left node is hit first, a 0 is pushed into $nodeNum$ and a 1 otherwise, if the ray intersects both boxes then a 0 is pushed onto the trail and if it only hits one box, a 1 is pushed. The traversal goes up the tree when an intersection was not found or when a leaf node is intersected, then it's set $trail = trail + 1$; and the number $n$ of consecutive 0's starting from the Least Significant Bit (LSB) in trail is counted. Next, the algorithm sets $trail = trail >> n$ and $nodeNum = nodeNum >> n$ (which corresponds to a pop). Finally, the LSB of nodeNum is inverted to visit the next branch. The traversal ends when $nodeNum = 1$, which means that the ray returns to the root and no more intersections were found.




| Step | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Node | 001 | 011 | 110 | 111 | 010 | 101 | 001 |
| Trail | 001 | 010 | 100 | 101 | 011 | 111 | 001 |

**Figure 8:** *Bit-Trail Traversal*

## 5 Results

This section presents the performance numbers of the Stack-Less Complete LBVH acceleration structure rendering 3D models in a ray tracing application. The metrics taken compare the construction time of the acceleration structure, rendering frame rate, memory footprint and ray traversal efficiency of our acceleration structure implementation with a stack-less SLBVH [Murguia et al. 2013] and a SAH-BVH implementation based on the implementation presented in the PBRT framework by Pharr et al.[Pharr and Humphreys 2004]. The tests were executed in a 3.19GHz Intel Core i7 CPU with 6Gb of DDR3 RAM compiled as a 32-bit application and the GPU is a NVIDIA Geforce GTX780 video card.

The models used are Stanford Bunny (69,451 primitives), Crytek Sponza (279,163 primitives), Armadillo (345,944 primitives), Happy Buddha (1,087,716 primitives) and Welsh Dragon (2,097,152 primitives). Figure 9 shows a ray-tracing application [Garcia et al. 2012] executing 3D models rendering in real time using the Stack-Less Complete LBVH acceleration structure. The application is using gloss mapping, Phong shading, one ray for shadows with one light source. Eight cameras with different positions and directions were setup to measure frame rates of the scenes rendered at $1024x1024$ pixels.

**Figure 9:** *Bunny (69K primitives), Armadillo (345K primitives), Happy Buddha (1M primitives) and Welsh Dragon (2M primitives) ray-traced in real time using a Stack-less Complete LBVH acceleration structure.*

### 5.1 Construction Performance

In the numbers shown in Figure 10, we present the construction performance of three acceleration structures: SAH-BVH, our stack-less Complete LBVH and stack-less LBVH.

If SAH-BVH structure is built in CPU and the other two accelerations structures are built in GPU, then we can't have a fair comparison with the SAH-BVH construction method. The advantages of this construction will be clearly seen in the rendering frame rates where the three structures are running in GPU. The construction times of our structure shows that the parallel algorithm can build the BVH tree of a 3D model with 1M primitives in $.37s$. This construction method can be used in combination with dynamic rebuilding strategies [Wald et al. 2009] to enable the ray tracing of animated scenes.

### 5.2 Rendering Frame Rate

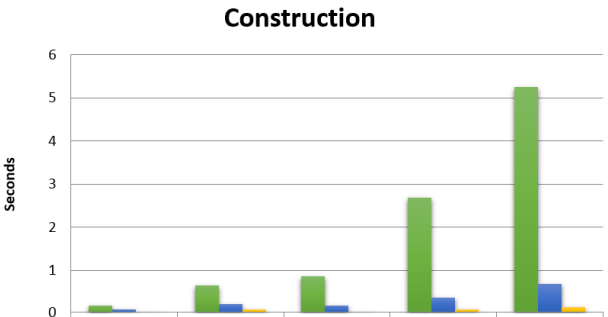
The rendering times are shown in Figure 11. Five models were traversed using a stack-based SAH-BVH, the stack-less Complete LBVH and a stack-less LBVH in a GPU NVIDIA Geforce GTX780 video card. The ray traversal on our acceleration structure is consistently $1.3\times$ faster than in the SAH-BVH structure and on average $1.5\times$ faster than the SLBVH for all the model except in the Sponza. In heterogeneous models like Sponza our partition method doesn't produce the regular distribution of the AABB's as in the case of rigid objects, this impacts rendering performance because a ray has to check for more box intersections. In the stack-less LBVH the geometry is not well-partitioned given that the middle-split scheme based on Morton codes is not optimal for balanced distributions, impacting the traversal performance.

### 5.3 Memory Footprint

Figure 12 shows the memory footprint of the three acceleration structures. SAH-BVH and stack-less Complete LBVH need the same amount of memory to store the structure. Even the memory footprint in the in the stack-less LBVH is lower than other stack-less approaches it stores empty nodes in order to enable it's stack-less traversal algorithm, this situation has a big impact in the memory needed to allocate the structure.

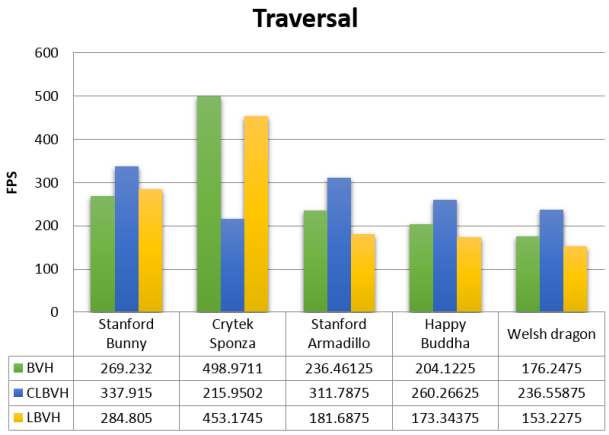
### 5.4 Traversal Efficiency

Figure 13 shows the graphics of ray intersections per box in red (plotting the $1\%$) and per primitive in green (plotting the $7\%$) for the three structures and figure 14, which shows the average intersections per box and primitive in the Happy Buddha model. The Stack-Less Complete LBVH intersects on average $30\%$ more boxes



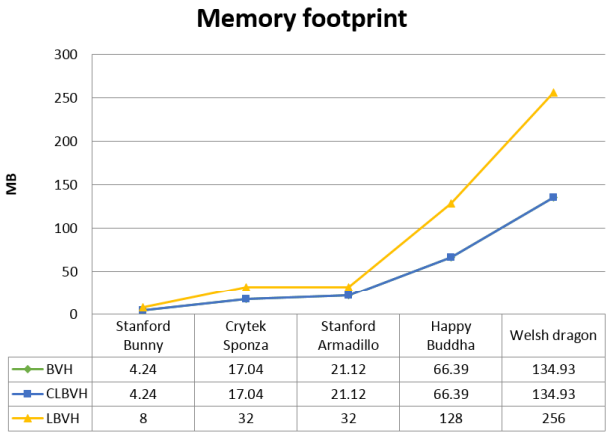

| | Stanford Bunny | Crytek Sponza | Stanford Armadillo | Happy Buddha | Welsh dragon |
|---|---|---|---|---|---|
| BVH | 0.151885 | 0.649513 | 0.858426 | 2.684844 | 5.246824 |
| CLBVH | 0.073718 | 0.189189 | 0.152146 | 0.374995 | 0.682959 |
| LBVH | 0.007143 | 0.074054 | 0.012953 | 0.064554 | 0.126499 |

**Figure 10:** *Construction Performance in Seconds*




| | Stanford Bunny | Crytek Sponza | Stanford Armadillo | Happy Buddha | Welsh dragon |
|---|---|---|---|---|---|
| BVH | 269.232 | 498.9711 | 236.46125 | 204.1225 | 176.2475 |
| CLBVH | 337.915 | 215.9502 | 311.7875 | 260.26625 | 236.55875 |
| LBVH | 284.805 | 453.1745 | 181.6875 | 173.34375 | 153.2275 |

**Figure 11:** *Rendering Frame Rates per Second*




| | Stanford Bunny | Crytek Sponza | Stanford Armadillo | Happy Buddha | Welsh dragon |
|---|---|---|---|---|---|
| BVH | 4.24 | 17.04 | 21.12 | 66.39 | 134.93 |
| CLBVH | 4.24 | 17.04 | 21.12 | 66.39 | 134.93 |
| LBVH | 8 | 32 | 32 | 128 | 256 |

**Figure 12:** *Memory footprint (MB). BVH and CLBVH use the same amount of memory.*

and $50\%$ less primitives than the SAH-BVH. Our structure intersects more boxes because the traversing algorithm is checking two boxes (left and right siblings) at a time, but this gives the advantage to check for less intersection in the leafs providing a better performance. In the case of the Stack-Less SLBVH with middle split partitioning the distribution is producing a structure where the primitives are not optimally distributed and many share the same leaf node producing an unbalance tree that leads to few box intersection and higher primitive intersections which impacts the traversal performance.

## 6 Conclusions

This work presented a novel partitioning approach to build high quality BVH structure. It also presented a highly efficient process to fully parallelize the construction where it used a modification of a fast paralleled implementation of a segmented radix sort algorithm to distribute the primitives based on the order given by the projection of the centers on it's longest box side. The partition method took full advantage of optimized data parallel scan algorithms to quickly determine the longest side of an AABB. We presented very competitive build times of this parallel construction.

We used a bit-trail traversal method to accelerate the ray-traversal

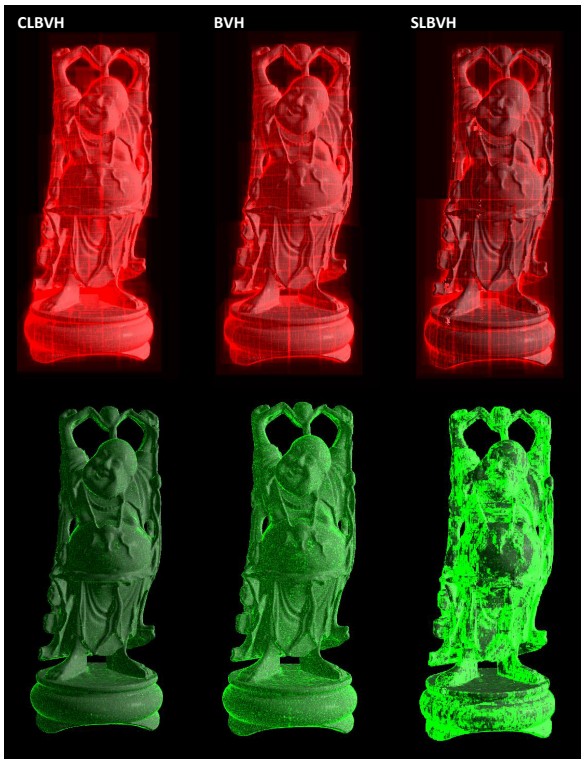



**Figure 13:** *Intersections per box (red) and per primitive (green)*

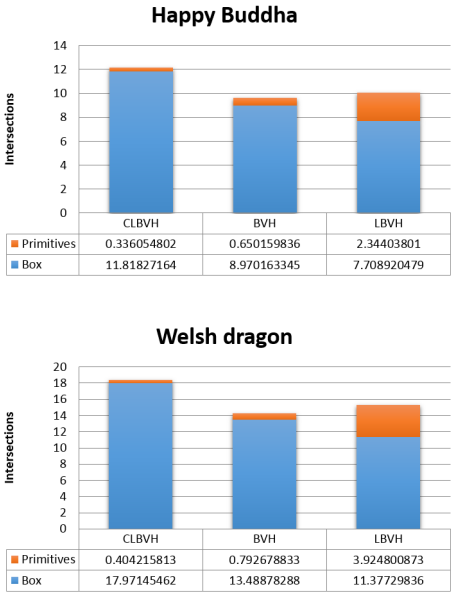



**Figure 14:** *Average intersections per box and per primitive*

and the frame rates presented a better performance compared with a SAH-BVH structure for rigid objects. In addition we presented the memory footprint and the intersections averages where the numbers reinforced the advantages in the traversal performance compared with a SAH-BVH structure.

The Stack-Less Complete LBVH partitioning and traversal method offers fast build times for high quality structures, efficient memory storage and provides high performance traversal for rigid objects. For future work we will apply this acceleration method in combination with predictive and adaptive strategies for accelerating the rendering of dynamic scenes in real time.

## Acknowledgments

The authors would like to thank the Stanford Computer Graphics Laboratory for the Stanford Bunny, Armadillo and Happy Buddha models [University 2013]; Bangor University for the Welsh Dragon model [University 2011].

## References

BAUSZAT, P., EISEMANN, M., AND MAGNOR, M. A. 2010. The minimal bounding volume hierarchy. In *VMV*, Eurographics Association, R. Koch, A. Kolb, and C. Rezk-Salama, Eds., 227–234.

BLELLOCH, G. E. 1990. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA.

GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, HPG '11, 59–64.

GARCIA, A., AVILA, F., MURGUIA, S., AND REYES, L. 2012. *Interactive Ray Tracing Using DirectX11 on the Compute Shader*, 1 ed. A K Peters/CRC Press, 353–376.

GOLDSMITH, J., AND SALMON, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7*, 5 (May), 14–20.

HAPALA, M., DAVIDOVIČ, T., WALD, I., HAVRAN, V., AND SLUSALLEK, P. 2013. Efficient stack-less bvh traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics*, ACM, New York, NY, USA, SCCG '11, 7–12.

HILLIS, W. D., AND STEELE, JR., G. L. 1986. Data parallel algorithms. *Commun. ACM 29*, 12, 1170–1183.

LAINE, S. 2010. Restart trail for stackless bvh traversal. In *Proceedings of the Conference on High Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, HPG '10, 107–111.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast bvh construction on gpus. In *IN PROC. EUROGRAPHICS 09*.

MACDONALD, D. J., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *Vis. Comput. 6*, 3 (may), 153–166.

MURGUIA, S., AVILA, F., REYES, L., AND GARCIA, A. 2013. *Bit-trail Traversal for Stackless LBVH on DirectCompute*, 1 ed. CRC Press, 319–335.

NIST, 2013. Complete binary tree.

PANTALEONI, J., AND LUEBKE, D. 2010. Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, HPG '10, 87–95.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

SULAIMAN1, H. A., AND BADE, A. 2011. The construction of balanced bounding-volume hierarchies using spatial object median splitting method for collision detection. In *International Journal of New Computer Architectures and their Applications*, 396–403.

UNIVERSITY, B., 2011. Eg 2011 welsh dragon. http://eg2011.bangor.ac.uk/dragon/Welsh-Dragon.html.

UNIVERSITY, S., 2013. The stanford 3d scanning repository.

WALD, I., MARK, W. R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2009. State of the art in ray tracing animated scenes. In *Computer Graphics Forum*.

WALD, I. 2010. Fast construction of sah bvhs on the intel many integrated core (mic) architecture. *IEEE Transactions on Visualization and Computer Graphics*.

ZHEFU, W., HONG, Y., AND BIN, C. 2013. Divide and conquer ray tracing algorithm based on bvh partition. In *Virtual Reality and Visualization (ICVRV), 2013 International Conference on*, 49–55.