A Memory Efficient Uniform Grid Build Process for GPUs

Eugene M. Taranta II Sumanta N. Pattanaik
University of Central Florida

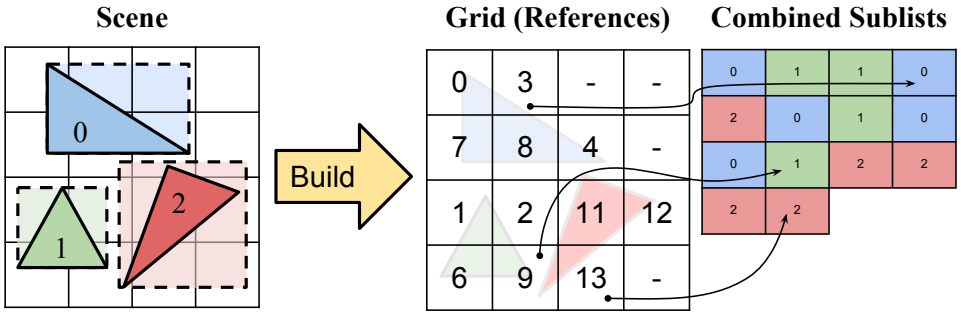


Figure 1. Given a scene comprising a set of geometric primitives and a grid density, the uniform grid build process yields a linear array of references, each reference pointing to a sublist of geometric primitives—those that intersect the associated voxel.

Abstract

Uniform grids are a common acceleration structure used to speed up intersection queries across various rendering and physics applications. These structures are attractive because they are easy to understand and exhibit fast build and query times. Recent advances in parallel algorithms have further increased their construction speed, although they still require substantial memory allocations throughout the build process and in their final representation. To address these memory consumption issues, we introduce a novel but easy to understand approach. Although our process is general, we demonstrate its effectiveness in an example real-time ray-tracing application where we see a 28% to 38% reduction in memory allocated for optimal grid densities and a 31% to 38% reduction in their final memory footprint. As grid densities increase, savings approach 45% and higher. We also show that our process does not sacrifice performance in terms of overall frame rates.

 ISSN 2331-7418

1. Introduction

A wide variety of applications currently leverage uniform grids. To illustrate a few examples, rendering and physics simulations often use grids to accelerate collision detection [Ericson 2004; Millington 2010]. Zheng et al. [2012] use grids to carry out self-compacting concrete flow simulations and predict flow and fill behaviors in complex structures. Razavi et el. [2015] describe an educational haptics-based drilling simulation tool for dentistry. Their system, in part, employs a hashed uniform grid structure. In order to visualize millions of dynamic particles, Krone et al. [2012] utilize marching cubes over a uniform grid of density estimates; and Reda et al. [2013] embed glyphs in a similar density grid to render solid ball-and-stick surfaces and volumes for interactive visualizations of molecular boundaries. One can also use grid structures to assist with photon gathering in progressive photon mapping [Hachisuka and Jensen 2010; Pedersen 2013], map overlaps [Magalhães et al. 2015], smoothed particle hydrodynamics [Akinci et al. 2013], collision detection of polydisperse sphere packings [Weller et al. 2013], and ray tracing [Kalojanov and Slusallek 2009], as well as many others approaches. Consequently, uniform grid improvements have broad appeal given their adaptability to a wide range of problems. In this paper, we specifically deal with build process and accelerator representation inefficiencies with respect to memory utilization. The approach we take is based on our previous work [Taranta II and Pattanaik 2014], which leverages recent advances in GPU hardware to accelerate atomic operations.

To better understand the landscape, Fujimoto et al. [**?**] first described uniform grids as a 3D extension of raster grids for ray tracing, although at the time they called their accelerator SEADS: spatially enumerated auxiliary data structure. More than twenty years later, Kalojanov and Slusallek [2009] developed the first GPU-based parallel algorithm for uniform grid construction, which Kalojanov et al. [2011] later extended to two-level grids. One issue with both approaches is that the workload between threads is nonuniform because each thread works on a different primitive, e.g., a triangle; and since primitives vary in size, the effort required to process a primitive also varies. Liu and Rokne [2013] addressed this issue by considering primitive-cell pairs—each thread analyzes exactly one triangle-cell pair in order to evenly distribute the workload, which also leads to more precise grid definitions. We further improved the process by eliminating an expensive sorting step that remained in the build pipeline by utilizing atomic operations [Taranta II and Pattanaik 2014]. Although the state of the art is now highly optimized, the build process and accelerator still consume large amounts of memory, thereby limiting grid density and scene complexity.

As we previously saw a significant improvement in build speed using atomic functions, we now take this a step further to eliminate a subset of memory allocations that occur throughout the build process as well as to reduce the accelerator's final memory

footprint. We achieve this reduction by employing additional atomic operations on housekeeping data and by replicating select calculations between steps, rather than storing intermediate results in cache. To evaluate our new build process, we use ray tracing as an example application; however, our approach is general and can be applied to any situation in which scene data is stored in a regular grid structure. As we demonstrate, these modifications have a slight negative impact on build time, although rendering performance improves, and so the overall frame rate remains relatively unchanged. More importantly, we also show that memory utilization in all tests is significantly reduced.

2. Build Process

Uniform grids are a spatial subdivision structure that partitions a scene into equally sized, axis-aligned rectangular cuboids called voxels. Common approaches enumerate these voxels and put them into one-to-one correspondence with a linear array. Namely, each occupied voxel in the accelerator contains a reference to a list of primitives that intersect it. In a ray-tracing application, to resolve ray-primitive collision queries, one can use a 3D-DDA [Amanatides and Woo 1987] to incrementally march a ray through the scene one voxel at a time. Upon passing through an occupied cell, the ray is cast against its associated geometry to test if the ray intersects any of the voxel's primitives. Given a scene comprising a set of geometric primitives and a specified grid density, our goal is to efficiently construct a uniform grid accelerator using a GPU (see Figure 1).

As an overview, the construction process is made up of three phases: Compute-Coarse-Pairs, Evaluate-Coarse-Pairs, and Extract-Grid. The first phase determines the maximum number of possible triangle-voxel overlaps that may exist in the scene by inspecting the axis-aligned bounding box (AABB) of each triangle. Since the AABB of a triangle often overlaps more voxels than the triangle itself, phase two evaluates each individual triangle-voxel coarse pair to determine if a true overlap exists. Finally, the last phase constructs the acceleration structure using key information collected in the prior phases. In what follows, we first detail the new construction process and then highlight how our approach differs from the state of the art.

As discussed, the Compute-Coarse-Pairs phase determines the maximum number of triangle-voxel overlaps that may exist in the current geometric configuration; see Figure 2 and Algorithm 1. We first allocate and zero-initialize the *coarse pairs count* array in order to store the overlap count upper bound for each triangle. The process then calculates the AABB of each triangle in parallel and stores each AABB's voxel occupancy count in the coarse pairs count array. Next, we cache the last coarse pairs count element and perform an in-place exclusive scan [Sengupta et al. 2007] to convert the array into a set of *indices*. Finally, we add the last element of the indices array

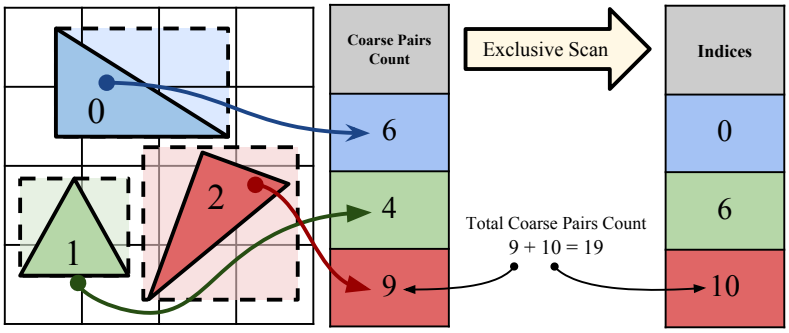


Figure 2. Compute-coarse-pairs. The maximum number of triangle-voxel overlaps that may exist in the current configuration is determined by casting the AABB of each triangle against the grid. Coarse pair counts are thereafter converted into indices for use in the next phase, and the total coarse pairs count is determined from the last indices and coarse pairs count elements.

Algorithm 1. Compute-Coarse-Pairs ( )

IntArray $coarsePairsCnt[triangleCount] = \{0\}$
foreach $triangleId \in [0, triangleCount - 1]$ **in parallel**
  AABB $voxBB =$ VOXEL-BOUNDING-BOX($triangleId$)
  $coarsePairsCnt[triangleId]$ = GET-VOXEL-COUNT(voxBB);
int $totalCoarsePairs \leftarrow coarsePairs[triangleCount - 1]$
$indices \leftarrow$ EXCLUSIVE-SCAN($coarsePairs$)
$totalCoarsePairs \leftarrow totalCoarsePairs + indices[triangleCount - 1]$

with the cached count to yield the *total coarse pairs* count. Note that "pair" refers to a single AABB-voxel overlap, which may or may not represent a real triangle-voxel overlap.

With all coarse pairs now identified, the Evaluate-Coarse-Pairs phase extracts all real triangle-voxel overlaps from the candidate overlaps; see Figures 3 and 4 as well as Algorithm 2. First, the process allocates a zero-initialized array equal in size to the total coarse pairs count. We refer to this as the *tags* array, and for each value stored in the indices array, we set the corresponding tag element to one, except element zero which remains zero. Next, we perform an in-place inclusive scan [Sengupta et al. 2007] on the tags array so that each element becomes a valid triangle ID. Once this transformation is complete, we refer to the tags array as the *coarse pairs* array, and there is still one entry for every AABB-triangle overlap.

We can now determine the precise overlaps. We start by allocating the *overlap counts* array, which is equal in size to the number of voxels in the scene and is used to track the number of real overlaps that exist per voxel. Next, we evaluate each coarse pair in parallel. Since each parallel task receives a unique task ID, we use this ID to determine which coarse pair to evaluate. That is, we use the task ID to select a

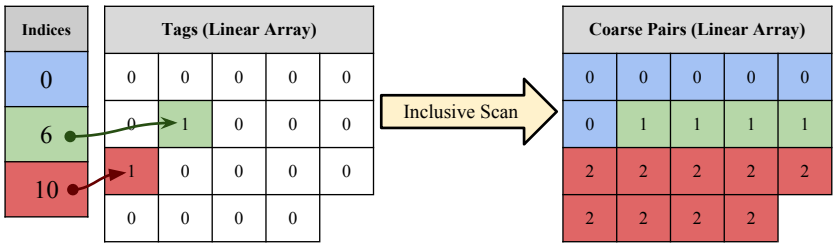


Figure 3. Coarse-pairs-evaluation I: The linear tags array is allocated, which is total coarse pairs count in length. For each index in indices except the first, its corresponding tag is set to 1. Finally, the tags array is converted into the coarse pairs array using an inclusive scan.

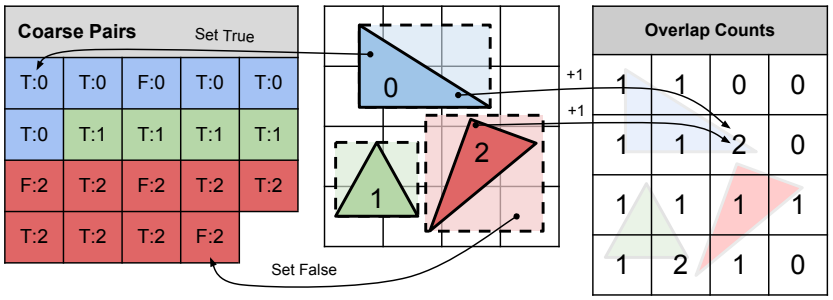


Figure 4. Coarse-pairs-evaluation II: Each coarse pair is evaluated in parallel to determine if the triangle truly overlaps the voxel. When an overlap exists, the most significant bit of the coarse pair element is set to true and its associated overlap counter is atomically incremented.

triangle ID from the coarse pairs array. Then with a triangle selected, we look up its starting position from the indices array. The difference between the task ID and the starting position gives us a voxel ID to process relative to the triangle's AABB—we treat the triangle's AABB as a region that is subdivided and enumerated into its own voxel set. For example, in Figure 3, task ID 8 processes the third voxel of triangle 1's AABB. Based on the indices array, triangle 1's starting position in the coarse pairs array is 6. Therefore, the task processes voxel ID 2 of triangle 1's AABB, i.e., 8-6=2. Thereafter, we convert the relative voxel ID into an exact grid voxel ID, from which we then calculate the grid voxel's AABB. Using the grid voxel's bounding box, we are able to perform a fast triangle-box overlap test [Akenine-Möller 2002] to determine if the triangle and grid voxel actually overlap. If an overlap is found, we set the most significant bit of the coarse pairs element to indicate that an overlap was identified. At the same time, we also atomically increment the corresponding count in the overlap counts array (see Figure 4). There is also a global overlap counter, not shown in the figure, that we atomically increment every time a real overlap is found. This can be done efficiently by synchronizing parallel tasks. In CUDA [Nickolls et al. 2008], for

Algorithm 2. Evaluate-Coarse-Pairs ( )

```
// Part I (Figure 3)
IntArray tags[totalCoarsePairs] = {0}
foreach triangleId ∈ [1, triangleCount − 1] in parallel
    tags[indices[triangleId]] = 1
coarsePairs ← INCLUSIVE-SCAN(tags)

// Part II (Figure 4)
int flags ← AllocRequiredBitFlag + AllocInProgBitFlag
IntArray overlapCounts[gridSize] = {flags}
int totalOverlapCnt ← 0
foreach coarsePairsIdx ∈ [0, totalCoarsePairs − 1] in parallel
    int triangleId ← coarsePairs[coarsePairsIdx]
    int triAabbVoxId ← coarsePairsIdx − indices[triangleId]
    int gridVoxId ← CONVERT-TO-GRID-VOX(triangleId, triAabbVoxId)
    if OVERLAP(triangleId, gridVoxId) then
        coarsePairs[coarsePairsIdx] ← triangleId + OverlapDetectedBitFlag
        ATOMIC-INCREMENT( overlapCounts[gridVoxId] )
        ATOMIC-INCREMENT( totalOverlapCount[gridVoxId] )
```

example, one can use a warp-wide reduction to coalesce up to thirty-two updates into a single atomic write.

With all of the true overlaps identified, we now enter the Extract-Grid phase as shown in Figure 5 and Algorithm 3. Based on the global overlaps count found in the previous step, we allocate the *triangle list* array, which will ultimately store the individual overlapping triangle list for each voxel. By the end of this phase, we will have converted the overlap counts array into the *grid* array, and each occupied voxel will have a reference into the triangle list where the overlapping triangle IDs are stored. Now to begin extraction, we again evaluate each coarse pair element in parallel. This

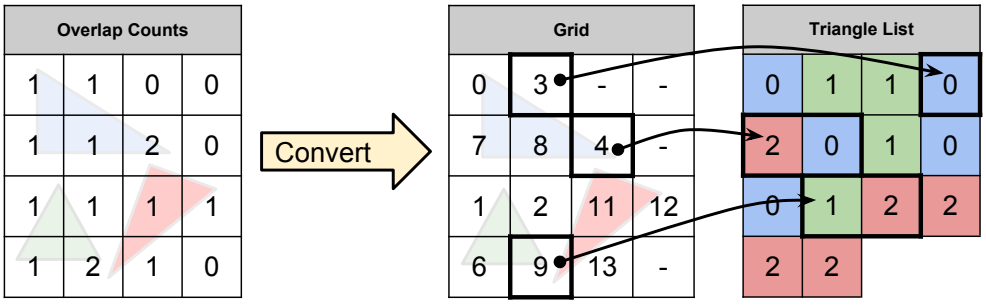


Figure 5. Extract-grid. Overlaps counts are converted into triangle list indices as the triangle list is filled in. Sublists for some grid voxels are highlighted though none of the additional flags (i.e., allocation required, etc.) are shown.

Algorithm 3. Extract-Grid ( )

```
int triangleListIdx ← 0
grid ← overlapCounts // alias
foreach coarsePairIdx ∈ [0, totalCoarsePairs − 1] in parallel
    int triangleId ← coarsePairs[coarsePairIdx]
    if NOT-BIT-SET (triangleId, OverlapDetectedBitFlag) then
        return
    triangleId ← triangleId − OverlapDetectedBitFlag
    int triAabbVoxId ← coarsePairsIdx − indices[triangleId]
    int gridVoxId ← CONVERT-TO-GRID-VOX(triangleId, triAabbVoxId)

    int terminateTriSubList ← 0
    int oldValue ←ATOMIC-CLEAR(grid[gridVoxId], AllocRequiredBitFlag)
    if BIT-SET(oldValue, AllocRequiredBitFlag) then
        int cnt ← oldValue − (AllocRequiredBitFlag + AllocInProgBitFlag)
        int start ← ATOMIC-ADD(triangleListIdx, cnt)
        int idx ← start + cnt − 1
        ATOMIC-WRITE(grid[gridVoxId], idx)
        terminateTriSubList ← TerminateBitFlag
    else
        while BIT-SET(oldValue, AllocInProgBitFlag) do
            oldValue ←ATOMIC-READ(grid[gridVoxId])
        int oldIdx ←ATOMIC-DECREMENT(grid[gridVoxId])
        int idx ← oldIdx − 1

    triangleList[idx] ← triangle + terminateTriSubList
```

time, if the most significant bit of the coarse pairs element is set, then the overlap is valid and we save this information into the grid and triangle list arrays. Note that we convert the task ID into a grid voxel ID using the same technique as before. However, not discussed previously, is that we also set the two most significant bits of each element in the overlap counts array. We use these two bits as flags to help synchronize construction of the uniform grid, where one bit is the *allocation required* flag and the other is the *allocation in progress* flag. When a coarse pairs element contains a true overlap, the process reads the corresponding overlap counts element and simultaneously clears the allocation required flag; for example, in CUDA we use an atomic-and operation to simultaneously read the current value and clear the allocation required flag. From here, depending on the state of the flags, we take one of three actions:

1. *Allocation required.* If the allocation required flag is set, the process needs to reserve enough space in the triangle list to store the sublist of triangle IDs overlapping the associated grid voxel. For this condition, we use a zero-initialized global counter called *triangle list index*. Let n denote the associated overlap count for the voxel, minus the upper bit flags. Further, let i denote the current

triangle list index value. When an allocation is required, we first atomically increment the triangle list index by n, thereby reserving space in the triangle list array at locations $[i, i + n - 1]$. Next, we write the triangle ID into the triangle list array at position $i + n - 1$. We also write $i + n - 1$ into the overlap counts element, thus converting the counter into a grid array element—a reference into the triangle list. Note that this update also clears the allocation in progress flag.

2. *Allocation in progress*. In this case, a different task cleared the the allocation required flag, but the actual allocation work is not complete. Therefore, the thread "spins" by continuing to read the overlap counts element until the allocation in progress flag clears. Once both flags are clear, the process continues into the allocation complete state.

3. *Allocation complete*. In this state, both flags are clear and the overlap counter contains a triangle list reference. The process atomically then decrements the reference stored in the grid element and stores the associated triangle ID in the triangle list at the specified location. After all of the overlaps are processed, the grid array element will hold the value i, where the triangle ID sublist starts.

For traversals through a uniform grid to work, the ray-tracing process must know which grid voxels are occupied and where each triangle sublist terminates. To address the former, if the allocation in progress flag is still set, then the voxel is empty. To address the latter, when an allocation occurs, we also set the most significant bit of the triangle ID written into position $i + n - 1$. Therefore, when the ray-tracing process hits an occupied voxel, it can walk through the triangle sublist until it encounters this bit flag.

2.1. Summary of Modifications

The previous approach differs from our new build process in two ways. First, the previous approach stores true voxel overlaps in a separate array, and so in the Extract-Grid phase, no additional work is required to calculate the associated voxel IDs. Our method, however, eliminates this extra storage by using bit flags in the coarse pairs array, but, as a consequence, we also have to later recalculate voxel IDs. Second, the previous approach stores a triangle list start and end reference per voxel, whereas our method only requires a single reference. This optimization, however, comes at the cost of some additional atomic operations to synchronize access to the triangle list during the extract-grid phase. Although these changes result in additional work, we show in our evaluation that differences in performance are minor, especially when considering the reduction in memory usage.

3. Evaluation and Results

To evaluate our new build algorithms, we developed a ray-tracing application and tested twelve different scenes of varying complexity over a wide range of grid resolutions. The scene complexity ranged between $261,978$ and $10,500,549$ triangles. Following recommendations from prior work [Cleary et al. 1983; Wald et al. 2006], we defined the grid density λ parameter as

$$N_x = d_x \sqrt[3]{\frac{\lambda N}{V}},\ N_y = d_y \sqrt[3]{\frac{\lambda N}{V}},\ N_z = d_z \sqrt[3]{\frac{\lambda N}{V}},$$

where vector $\vec{d}$ is the extent of the scene's bounding box, V is its volume, and N is the number of primitives. Further, we rendered all scenes at 1920 x 1280 using primary ray tracing and dot-normal shading. We made this decision to verify that our uniform grid modifications did not impact traversal performance, since coherent rays are most likely to experience performance degradation due to changes to the underlying accelerator. Further, we also rendered each scene 1000 times with the camera position and orientation randomized in each iteration, where the camera position is selected from anywhere within the scene's AABB. This randomization ensures that open views, obstructed views, and combinations thereof are evaluated across both conditions (with and without our accelerator modifications). The test system was a 3.2 GHz Quad-Core Intel Xeon Mac Pro with 6 GiB of 1066 MHz DDR3 memory, and an SSD; and the GPU was an NVIDIA GeForce GTX 680 GPU with 2048 MiB of memory.

We present memory usage in Figure 6 for the Conference Room and Matinee scenes, and build time and rendering performance for the same test scenes in Figure 7. Note, however, that these trends are identical for all test scenes. For all practical grid densities, our method improved memory usage. Even for the Conference Room scene at its lowest density ($\lambda = .5$), our approach reduced the maximum amount of memory consumed during the build process by 25%, and these savings increase rapidly, appearing to hit an asymptote near 48%. Our method also improved the final accelerator's memory footprint in a similar way for all scenes. While ray-tracing applications using a standard uniform grid are unlikely to select such high grid densities, applications that build hierarchal data structures upon uniform grids or utilize grids in a different fashion may benefit from these additional savings. In the Matinee scene, improvements start near 20%, rise rapidly, and level off near 47%, which again is similar for all scenes. With respect to speed, our approach slightly improves rendering time in milliseconds for all grid resolutions, whereas the build time suffers slightly. These differences balance out in such a way that the aggregate time spent building and rendering each frame remains relatively constant; see the speedup curve in Figure 7.

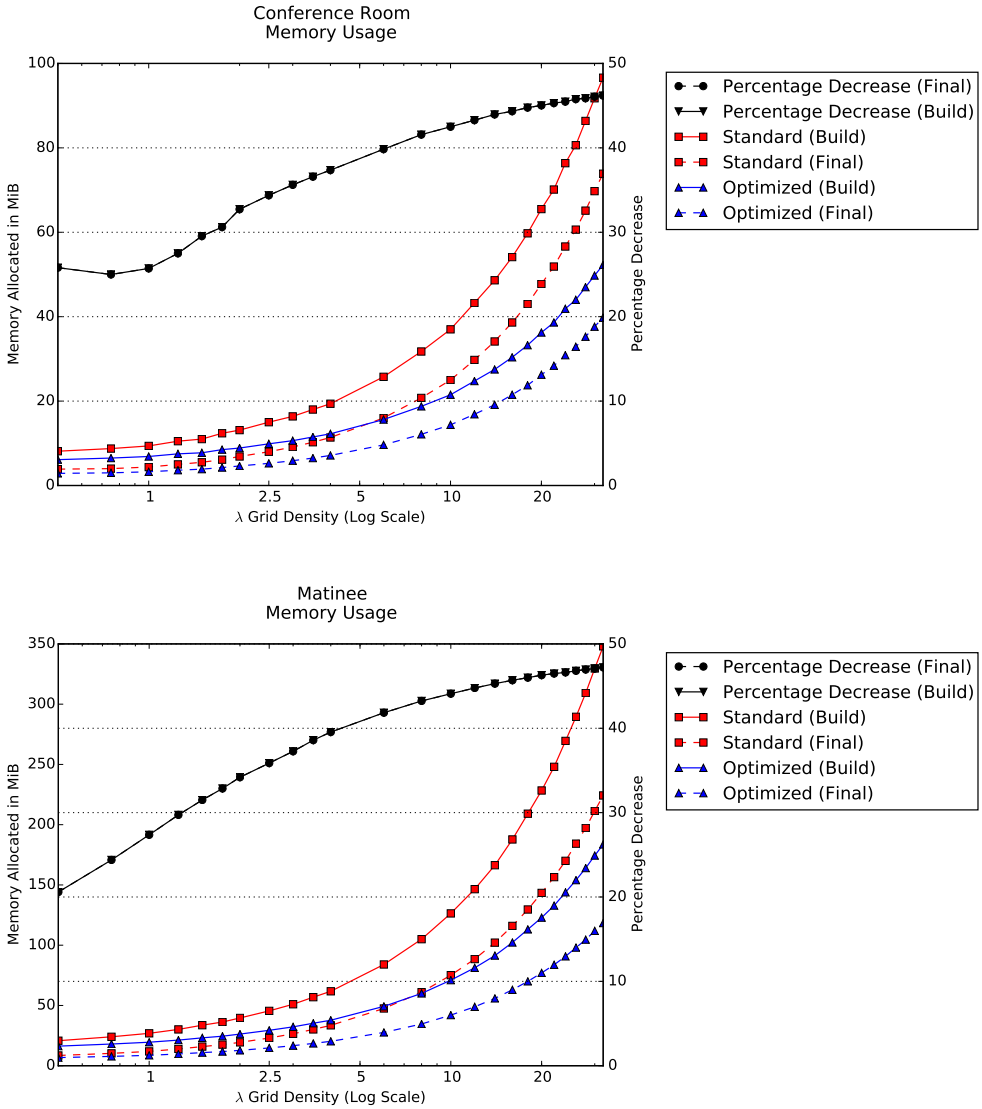


Figure 6. Memory usage for the Conference Room and Matinee scenes with varying grid densities, using the state-of-the-art build process and our memory optimized build process, shown as *standard* and *optimized*, respectively. Shown are both the maximum memory consumed during construction and the final memory required: *build* and *final*. The curves shown here are similar to those in all of our test scenes, where memory savings start above 20% with $\lambda = 0.75$ and quickly approach an asymptote of a better than 45% reduction in memory utilization. Further, note that the standard (final) memory requirement is greater than the optimized (build) requirement for medium and larger densities.

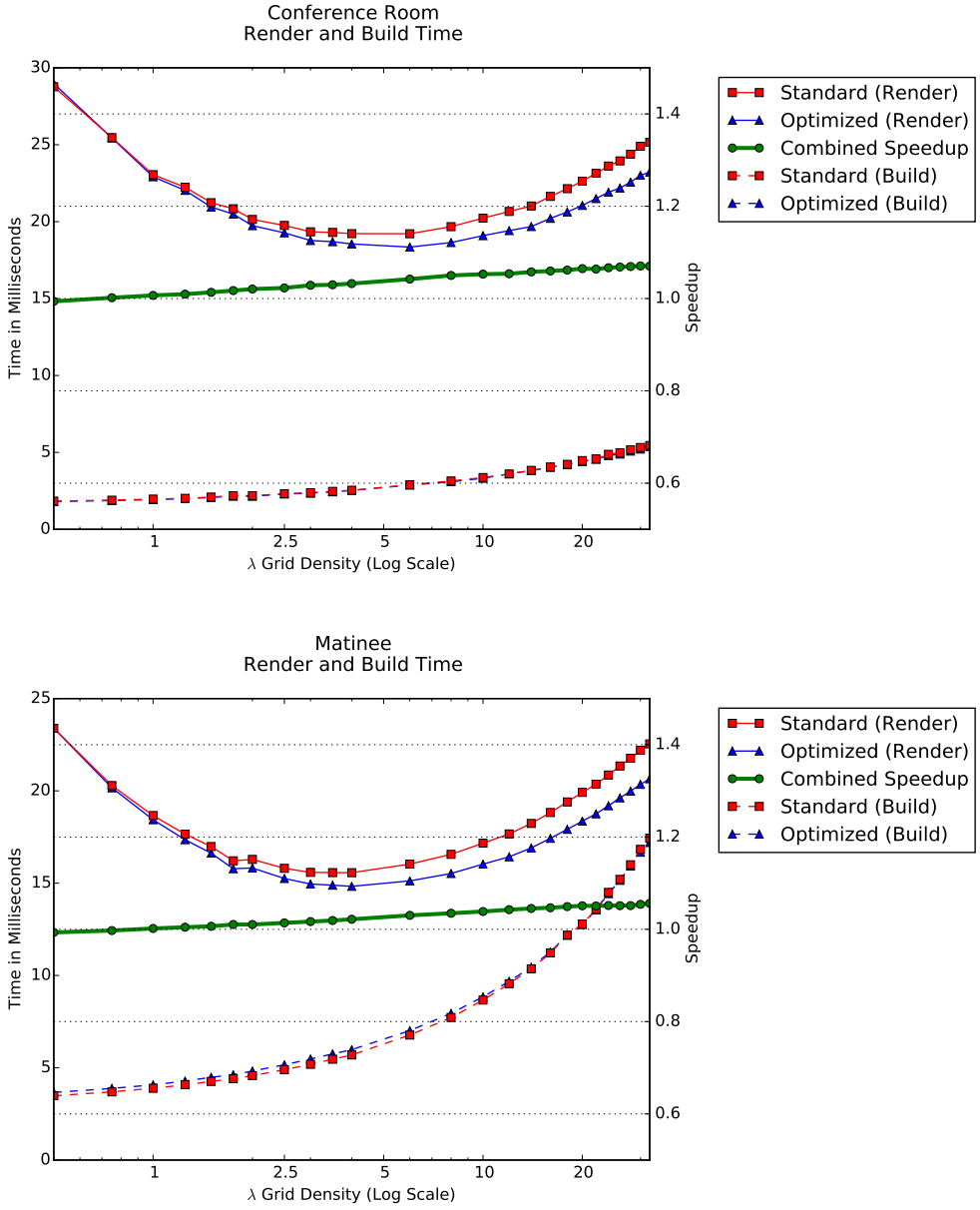


Figure 7. Rendering and build times for the Conference Room and Matinee scenes with varying grid densities, using the state-of-the-art build process and our memory optimized build process, shown as *standard* and *optimized*, respectively. The curves shown here are similar to those of most of our test scenes (exceptions are noted in the discussion, Section 4). In test scenes with lots of open space such as Cyrtek, The Bedroom, and those above, there is a noticeable difference in rendering time as the density increases, where our approach is faster.

	Standard			Memory Optimized			% Reduction	
Scene	**λ**	**Max**	**Final**	**λ**	**Max**	**Final**	**Max**	**Final**
Crytek Sponza	3.5	20	10	3.5	12	6	37	34
Conference Room	3.0	16	9	4.0	12	7	25	22
The Bedroom	3.0	23	12	3.0	15	8	35	33
The Shop Girls	3.5	24	15	3.5	16	9	35	38
Science Fiction	1.5	24	13	1.5	17	9	28	33
Matinee	1.75	36	17	1.75	24	11	32	33
Christmas	2.0	50	25	2.5	37	20	25	21
Artistry	2.5	70	29	2.5	43	18	38	37
Natural History	2.0	70	34	2.0	47	23	33	32
Hairball	0.75	234	65	0.75	152	56	35	13
San Miguel	0.5	188	88	0.75	171	79	9	10
Asian Dragon	2.5	263	178	3.0	195	124	26	30

Table 1. Maximum and final MiB memory allocations using the *standard* build process and our *memory optimized* process. The grid densities (λ) selected are those that maximize combined build and render performance. Note that in some cases the memory optimized density is larger and, therefore, the percentage reduction result compares two different densities.

	Standard			Memory Optimized			Combined
Scene	**λ**	**Build**	**Render**	**λ**	**Build**	**Render**	**Speedup**
Crytek Sponza	3.5	2.85	7.63	3.5	2.83	7.41	1.024
Conference Room	3.0	2.39	19.34	4.0	2.53	18.55	1.031
The Bedroom	3.0	3.25	17.56	3.0	3.26	17.02	1.026
The Shop Girls	3.5	3.12	20.67	3.5	3.20	20.29	1.013
Science Fiction	1.5	3.17	13.82	1.5	3.35	13.27	1.022
Matinee	1.75	4.40	16.21	1.75	4.62	15.78	1.010
Christmas	2.0	5.57	18.20	2.5	6.53	17.34	0.996
Artistry	2.5	7.01	29.65	2.5	7.44	29.26	0.999
Natural History	2.0	7.60	16.60	2.0	8.30	16.22	0.987
Hairball	0.75	27.99	45.78	0.75	31.53	45.96	0.952
San Miguel	0.5	23.36	27.02	0.75	28.31	25.67	0.933
Asian Dragon	2.5	19.40	60.00	3.0	22.68	57.00	0.997

Table 2. Speed comparison of the state-of-the-art build process and our memory optimized build process for each test scene, shown as *standard* and *memory optimized*, respectively. The grid densities (λ) selected are those that maximize combined build and render performance. All times are in milliseconds. The combined speedup compares the best standard to the best memory optimized performance.

Table 1 lists memory consumption results for the state-of-the-art build process alongside our memory optimized build process for the various test scenes. We selected those results that gave the best combined build and rendertime performance. In all cases, our new build process improves memory usage, even in situations where the optimal grid density is larger using the new process, such as with the San Miguel scene. For scenes having an equal optimal density, maximum memory usage drops by 28–37% and the final output size also drops by 13–34%. In Table 2, we show build and render millisecond time performance for the same conditions. As is consistent with the previously discussed results, our process marginally impacts build performance, though we are able to render the scenes slightly faster. In the end, the overall difference in combined speed is negligible—less than 2%. Two notable exceptions are the Hairball and San Miguel scenes, which we discuss further in the next section.

4. Discussion

Based on the results shown in Figure 6, we were able to significantly improve memory utilization for both the build process and final output. As an example, the previous method requires 232 MiB to construct the accelerator for the Natural History scene at its optimal grid density ($\lambda = 2.0$), whereas with our method, memory consumption is now down to 163 MiB, a 37% reduction in memory usage. The size of the uniform grid itself also dropped from 34 to 23 MiB, a 32% improvement, which is similar for most scenes. An additional benefit of our approach is that for medium to large grid densities, our process's maximum memory consumption is less than the size of the accelerator generated by the previous process. Therefore, our approach allows one to reach higher grid densities where insufficient memory resources previously posed a problem; this will benefit a variety of practical applications. We observed this benefit in some of our own test scenes. Specifically, the maximum density of the Hairball scene was 12 and 30 for the previous and new processes, respectively. Similarly, Asian Dragon peaked at 14 and 18, and San Miguel at 4 and 10.

In Table 2 we see that the combined rendering and build performance is mostly unaffected. In general, our approach slightly impacts build performance, but marginally improves rendering performance. This effect on build performance is likely due to the repeated work required to calculate coarse pair voxels in the extract-grid step (see Algorithm 3) and because of the additional atomic operations, which falls in line with the slight decline observed in performance as the scenes become more complex. With respect to the speedup in rendering performance, this is likely due to the fact that for each voxel through which a ray passes, we only access one triangle list reference; whereas, the previous approach accesses two references, so scenes with vast amounts of empty space benefit from our optimization. We expect other applications utilizing uniform grids for collision queries to experience this same boost in perfor-

mance. However, given scenes such as Hairball and San Miguel that are both dense and complex, improvements in rendering performance are not enough to compensate for the degradation in build performance. Another limitation of our method is that large non-axis aligned triangles can still generate significant overhead and consume large amounts of memory due to its mostly empty AABB. One can circumvent this issue by decomposing triangles into smaller pieces in a preprocessing step.

It is worth noting that we can further optimize the build process by introducing some risk. In Algorithm 3 (EXTRACT-GRID), the process spins while waiting for an allocation to complete by repeatedly reading the associated grid array element. Once the allocation in progress flag clears, we then acquire the triangle list index via an atomic decrement operation. Rather than using an atomic read, however, we can instead use an atomic decrement operation so that as soon as the allocation in progress flag clears, the next access acquires the triangle index position, thereby eliminating one atomic operation. This modification requires that we initialize the grid element with a third high bit set, so that the decrement does not clear the allocation in progress flag. Although we did not experience any issue with this approach, we cannot guarantee that the allocation will complete before the count rolls under and unintentionally clears the allocation in progress flag. With this modification, build performance is indeed faster and a bit closer to the original build performance, but because of this potential race condition, we do not formally report our results.

5. Conclusion

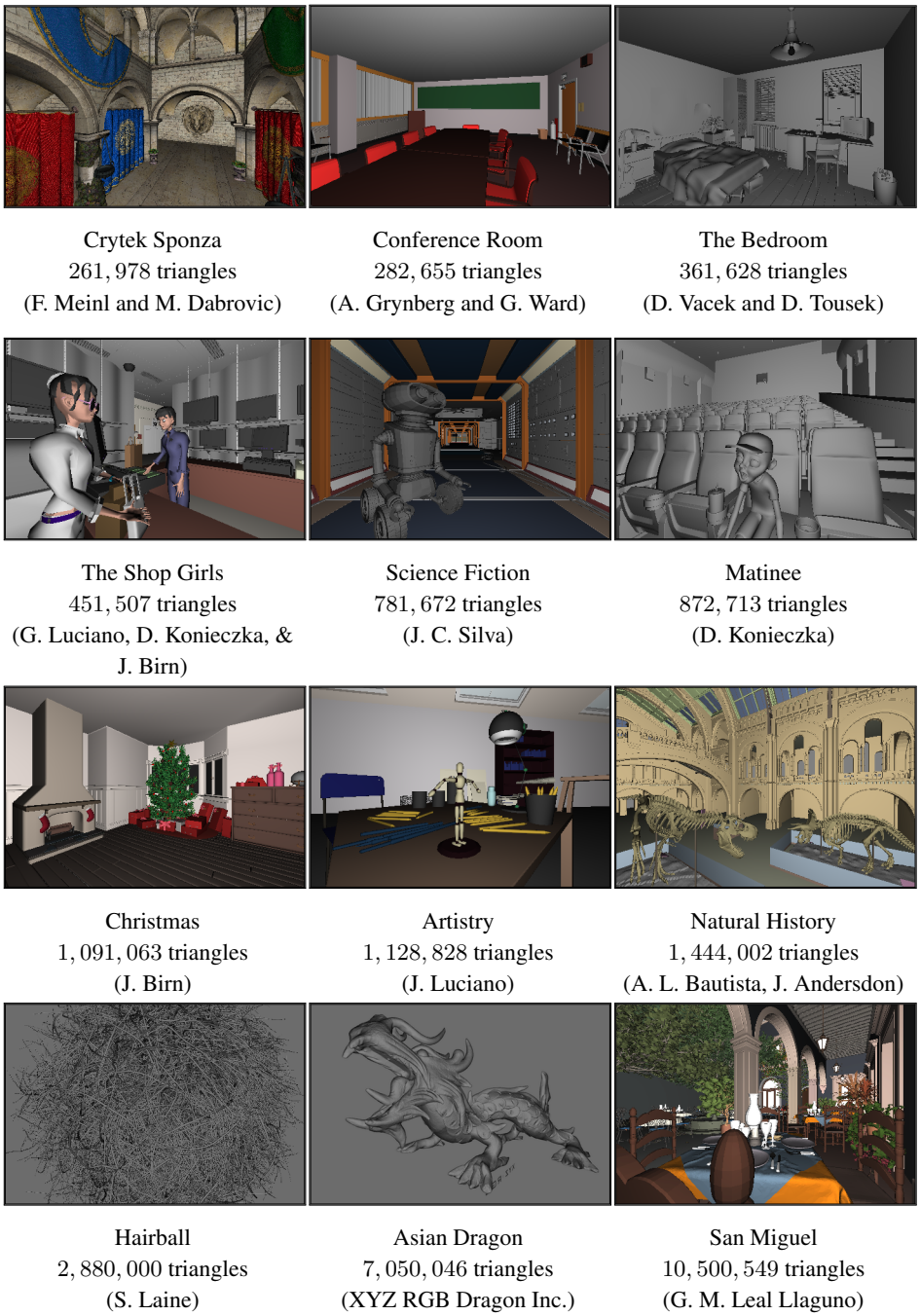
We have presented a new uniform grid build procedure designed to reduce memory consumption during the build process and in the final grid representation. Further, we used a ray-tracing application to evaluate our approach, using scenes of varying complexity, and in all cases we were able to demonstrate a significant improvement in memory consumption. Under practical conditions, our method uses 28% to 38% less memory during the build process as compared to the state of the art and similarly, our method requires 31% to 38% less memory to store the accelerator. For large grid densities, these savings jump up to approximately 45%. Our evaluation also shows little to no loss in the combined build and render time performance and in some cases, the possibility of a small gain in rendering performance. We expect that the optimizations discussed and demonstrated will also extend to other applications using uniform grids.

Acknowledgements

In addition to the test scene authors listed in Figure 9, we acknowledge NSF (Award Number IIS-1064427) for providing partial funding for this research.

A. Test Scenes

Figure 9 contains information about our twelve test scenes, including the scene name, a screen capture, the triangle count, and acknowledgments.



Crytek Sponza
261, 978 triangles
(F. Meinl and M. Dabrovic)

Conference Room
282, 655 triangles
(A. Grynberg and G. Ward)

The Bedroom
361, 628 triangles
(D. Vacek and D. Tousek)

The Shop Girls
451, 507 triangles
(G. Luciano, D. Konieczka, & J. Birn)

Science Fiction
781, 672 triangles
(J. C. Silva)

Matinee
872, 713 triangles
(D. Konieczka)

Christmas
1, 091, 063 triangles
(J. Birn)

Artistry
1, 128, 828 triangles
(J. Luciano)

Natural History
1, 444, 002 triangles
(A. L. Bautista, J. Andersdon)

Hairball
2, 880, 000 triangles
(S. Laine)

Asian Dragon
7, 050, 046 triangles
(XYZ RGB Dragon Inc.)

San Miguel
10, 500, 549 triangles
(G. M. Leal Llaguno)

Figure 9. Test scenes used to validate our memory optimized build process.

References

AKENINE-MÖLLER, T. 2002. Fast 3D triangle-box overlap testing. *Journal of Graphics Tools 6*, 1 (Jan.), 29–33. URL: http://dx.doi.org/10.1080/10867651.2001.10487535. 54

AKINCI, G., AKINCI, N., OSWALD, E., AND TESCHNER, M. 2013. Adaptive surface reconstruction for SPH using 3-level uniform grids. In *21st International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision in co-operation with EUROGRAPHICS Association, WSCG 2013, Plzen, Czech Republic, June 24-27, 2013*, Eurographics Association, Aire-la-Ville, Switzerland, 195–204. URL: http://hdl.handle.net/11025/10610. 51

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, Eurographics Association, Aire-la-Ville, Switzerland, 3–10. URL: http://dx.doi.org/10.2312/egtp.19871000. 52

CLEARY, J. G., WYVILL, B. M., VATTI, R., AND BIRTWISTLE, G. M. 1983. Design and analysis of a parallel ray tracing computer. In *Graphics Interface '83*, National Research Council of Canada, Toronto, ON, Canada, 33–38. URL: http://graphicsinterface.org/proceedings/gi1983/gi1983-5. 58

ERICSON, C. 2004. *Real-time collision detection*. CRC Press. URL: https://www.crcpress.com/Real-Time-Collision-Detection/Ericson/p/book/9781558607323. 51

HACHISUKA, T., AND JENSEN, H. W. 2010. Parallel progressive photon mapping on GPUs. In *ACM SIGGRAPH ASIA 2010 Sketches*, ACM, New York, SA '10, 54:1–54:1. URL: http://doi.acm.org/10.1145/1899950.1900004. 51

KALOJANOV, J., AND SLUSALLEK, P. 2009. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, HPG '09, 23–28. URL: http://doi.acm.org/10.1145/1572769.1572773. 51

KALOJANOV, J., BILLETER, M., AND SLUSALLEK, P. 2011. Two-level grids for ray tracing on GPUs. *Computer Graphics Forum 30*, 2, 307–314. URL: http://dx.doi.org/10.1111/j.1467-8659.2011.01862.x. 51

KRONE, M., STONE, J., ERTL, T., AND SCHULTEN, K. 2012. Fast visualization of gaussian density surfaces for molecular dynamics and particle system trajectories. The Eurographics Association, Aire-la-Ville, Switzerland, M. Meyer and T. Weinkaufs, Eds. URL: http://dx.doi.org/10.2312/PE/EuroVisShort/EuroVisShort2012/067-071. 51

LIU, X., AND ROKNE, J. G. 2013. A micro 64-tree structure for accelerating ray tracing on a GPU. In *Proceedings of Graphics Interface 2013*, Canadian Information Processing Society, Toronto, ON, Canada, GI '13, 165–172. URL: http://dl.acm.org/citation.cfm?id=2532129.2532158. 51

MAGALHÃES, S. V. G., ANDRADE, M. V. A., FRANKLIN, W. R., AND LI, W. 2015. Fast exact parallel map overlay using a two-level uniform grid. In *Proceedings of the 4th International ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*, ACM, New York, BigSpatial'15, 45–54. URL: http://doi.acm.org/10.1145/2835185.2835188. 51

MILLINGTON, I. 2010. *Game physics engine development: how to build a robust commercial-grade physics engine for your game*. CRC Press, Boca Raton, FL. 51

NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. 2008. Scalable parallel programming with CUDA. *Queue 6*, 2 (Mar.), 40–53. URL: http://doi.acm.org/10.1145/1365490.1365500. 54

PEDERSEN, S. A. 2013. *Progressive photon mapping on GPUs*. Master's thesis, Norwegian University of Science and Technology, Norway. URL: http://hdl.handle.net/11250/253364. 51

RAZAVI, M., TALEBI, H., ZAREINEJAD, M., AND DEHGHAN, M. 2015. A GPU-implemented physics-based haptic simulator of tooth drilling. *The International Journal of Medical Robotics and Computer Assisted Surgery 11*, 4, 476–485. URL: https://dx.doi.org/10.1002/rcs.1635. 51

REDA, K., KNOLL, A., I. NOMURA, K., PAPKA, M. E., JOHNSON, A. E., AND LEIGH, J. 2013. Visualizing large-scale atomistic simulations in ultra-resolution immersive environments. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, IEEE, Los Alamitos, CA, 59–65. URL: http://dx.doi.org/10.1109/LDAV.2013.6675159. 51

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for GPU computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, Eurographics Association, Aire-la-Ville, Switzerland, GH '07, 97–106. URL: http://dl.acm.org/citation.cfm?id=1280094.1280110. 52, 53

TARANTA II, E. M., AND PATTANAIK, S. N. 2014. Macro 64-regions for uniform grids on GPU. *The Visual Computer 30*, 6-8, 615–624. URL: http://dx.doi.org/10.1007/s00371-014-0974-x. 51, 67

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions On Graphics 25*, 3 (July), 485–493. URL: http://doi.acm.org/10.1145/1141911.1141913. 58

WELLER, R., FRESE, U., AND ZACHMANN, G. 2013. Parallel collision detection in constant time. In *Workshop on Virtual Reality Interaction and Physical Simulation*, The Eurographics Association, Aire-la-Ville, Switzerland, J. Bender, J. Dequidt, C. Duriez, and G. Zachmann, Eds. URL: http://dx.doi.org/10.2312/PE.vriphys.vriphys13.061-070. 51

ZHENG, J., AN, X., AND HUANG, M. 2012. GPU-based parallel algorithm for particle contact detection and its application in self-compacting concrete flow simulations. *Computers & Structures 112/113*, 193–204. URL: http://dx.doi.org/10.1016/j.compstruc.2012.08.003. 51

Index of Supplemental Materials

Included in our supplemental material (`http://www.jcgt.org/published/0005/03/04/source.zip`) is a fully commented CUDA implementation of the build process described in this paper, build_mem_optimized.cu. Also included is the state-of-the-art variant, build_standard.cu from [Taranta II and Pattanaik 2014] for reference and comparison. The last item, build_helpers.cu, contains some utility functions used by both.

Author Contact Information

Eugene M. Taranta II
Department of Computer Science
University of Central Florida
4000 Central Florida Blvd
Orlando, FL 32816
etaranta@gmail.com

Sumanta N. Pattanaik
Department of Computer Science
University of Central Florida
4000 Central Florida Blvd
Orlando, FL 32816
sumant@cs.ucf.edu
http://graphics.cs.ucf.edu/

Taranta II, Eugene M. and Pattaniak, Sumanta N., A Memory Efficient Uniform Grid Build Process For GPUs, *Journal of Computer Graphics Techniques (JCGT)*, vol. 5, no. 3, 50–67, 2016
http://jcgt.org/published/0005/03/04/

Received:	2015-06-14		
Recommended:	2016-03-28	Corresponding Editor:	Wojceich Jarosz
Published:	2016-09-29	Editor-in-Chief:	Marc Olano