High Definition Interactive Animated Ray Tracing on CELL Processor using Coherent Grid Traversal

David R. Chapman[*]
University of Maryland Baltimore County

Abstract

The IBM/Toshiba/Sony CELL processor exhibited on the SONY Playstation 3 offers a large amount of parallel processing power for it's price; this power can be exploited during ray-tracing. The additional computing power stems from multiple cores with entirely SIMD instruction sets, slow branch performance, and no cache-miss detection. Unfortunately, this processor design is not well suited for typical Ray Acceleration Scheme (RAS) traversals that rely heavily on stack recursion and branching. Recently, packet-based traversals of RASs have demonstrated speed increases of at least an order of magnitude over traditional approaches when computing at high image resolutions on traditional desktop processors. Frustum-based uniform grid traversals in particular not only demonstrate this incredible performance, but can apparently be optimized for execution on parallel SIMD processors such as the CELL. Additionally, uniform grid traversals allow for animated scenes via dynamic scene voxelization.

Keywords: Ray tracing, CELL Processor, multi-core

1 Introduction

Ray tracing is described as "embarrassingly parallel", because the algorithm can easily be partitioned onto parallel threads of execution. For example, the correctness of ray-object intersection tests does not depend on the order in which they are executed. Thus, many rays and objects could be intersected in parallel while computing an image.

Unfortunately, CPUs are not designed to efficiently execute parallel algorithms such as Ray tracing. Historically, CPUs have in large part been designed to execute a single sequential thread of program logic, for which any instruction may depend on the results of it's predecessors. Valuable CPU real-estate is expended to employ heuristic hardware to execute in parallel sequentially organized threads with ever diminishing performance gains. For Ray tracing, this real-estate could be better utilized by performing more arithmetic computation.

The CELL processor exhibited on the Sony Playstation 3 is a radical processor design that provides 200 GFlops, which for comparison is 35 times more than an AMD Opteron CPU [Pham et al. 2005]. This incredible performance is possible under the assumption that many instructions will be executed in parallel. This assumption typically holds true for Ray tracing, and thus the CELL processor is an intuitive candidate for Ray tracing. The CELL processor contains 9 cores; 8 of which are Synergistic Processing Units (SPUs), which have entirely SIMD instruction sets. The 9th core is a traditional PowerPC processor.

[*]e-mail: dchapm2@umbc.edu

Although performing Ray tracing on the CELL processor is clearly intelligible. It is not entirely obvious how to do so efficiently. There are additional challenges when attempting to perform High Definition Animated Ray tracing. For example, although it is easy to parallelize Ray tracing via Multithreading, CELL also provides massive SIMD power, but it is more difficult to SIMDize RAS traversals. Many traversals are optimized for traditional processors, and make heavy use of branching and stack recursion, which cannot easily be mapped to the SIMD execution model.

Another reason why High Definition Ray tracing is challenging, is because the running time of Ray tracing is linear with the number of rays in the scene. In other words, when the screen size is multiplied by two in each dimension, Ray tracing typically slows down by a factor of 4. Fortunately, in practice packet and frustum RAS traversal algorithms do not suffer as much from poor performance with large screen size. However, these schemes typically have even more complicated traversals, which can make SIMD execution even more difficult.

The goal is to adapt a fast a Ray Acceleration Scheme for a Ray Tracer on the CELL processor of a Playstation 3. The Ray Acceleration Scheme must allow for Animated Scenes, at real time or interactive framerates, while running at High Definition screen resolutions.

2 Related Work

Ray tracing has already been implemented on the IBM CELL Processor. Carsten Benthin implemented Coherent Ray tracing using a Packet based Bounding Volume Hierarchy traversals. This paper demonstrated a Ray tracing system on the CELL processor can provide performance roughly 9 times that of a traditional processor. This paper demonstrates the importance of the traversal implementation, as it obviates that 80%+ of the processing time is spent performing traversals. Unfortunately, their Bounding Volume traversal implementation was complicated enough that it required branching during the traversals, which slows down the CELL performance. In addition, their method required an expensive branch mis-prediction nearly 40

Packet and or frustum traversals for Ray Acceleration datastructures have been developed for all of the major datastructures such as BVH, BSP-tree, Octree, KD-tree, and uniform grid [Wald et al. 2007; Haitao Du 2003; Reshetov et al. 2005; Benthin et al. 2006]. A packet traversal is one in which a group of similarly oriented rays is traced through the datastructure at a time, rather than tracing individual rays at a time. A frustum traversal is a variation on this in which packets are bounded by bounding-frustums, and these bounding-frustums are traced through the datastructure at a time. These traversals all provide significantly higher performance at high screen resolutions than traditional traversal algorithms without packets. This is because all of the traversal cost is amortized over the packets; as screen resolution increases, the number of rays per packet increases, but the number of packets remains constant, thus the average cost per ray is lessened.

Unfortunately, not all of the traversals are suitable for animated scenes. For example, Octrees, KD trees, and BSP trees are generally thought of as too slow for dynamic reconstruction. For example, KD-tree construction takes seconds to minutes to perform using typical algorithms. Although work has been completed that allows for certain restricted forms of animation using KD-tree deformations [Günther et al. 2006], It is difficult to apply these techniques to random motion. Uniform Grids and BVH trees, on the other hand, are generally considered to provide fast enough rebuilding time for animated scenes.

Uniform Grids also appear to be most SIMDizable RAS. Although Foley and Sugarman had developed SIMD traversal algorithms for KD-trees, their algorithms have not been applied to packet based KD traversals [2005]. Szirmay-Kalos et al. have developed an extremely fast fully SIMD approximation to Ray tracing, but their approximation is unfortunately lossy, so it would be difficult to incorporate into this project [2005]. In general, the difficulty with SIMDizing RASs because of the hierarchical nature of the datastructure, and the branching and stack requirements of tree traversal break the SIMD paradigm.

Uniform grids are the only popular non-hierarchical RAS, and thus, simple traversals can be developed that do not require use of a stack [Wyvill 1988]. These traversals are easier to SIMDize. In addition, Uniform Grids can be reconstructed very quickly, and every frame if necessary. Thus they make a good candidate for animated scenes [Wald et al. 2006].

I attempt to modify a SIMD frustum-based uniform grid traversal algorithm by Wald et al. [2006], for efficient implementation on a Playstation 3. Their algorithm takes a packet of rays, and produces a bounding frustum. It then outwardly traverses the frustum slice by slice through the grid while computing all voxels for which the frustum intersects. All of the rays in the frustum are intersected with all objects in the slice, and frustum culling and mailboxing are used to weed out unnecessary intersection tests [Wald et al. 2006]. We propose to implement this algorithm on the cell processor, because it not only provides fast ray acceleration at high resolutions much like the other packet and frustum based traversal algorithms, but the traversal algorithm is stack-less, and overwhelmingly SIMD. For example, computing bounds on the next slice, as well as object frustum culling are all totally SIMD operations. This makes effective use of the SPU instruction set. The paper does not provide a good method of traversing rays generated by reflection or refraction. In addition, the technique for packet based shadow tracing is slightly convoluted, so as a result, secondary Ray tracing will be an extension to the project and nothing more. In addition, there are still parts of the traversal that were designed for traditional CPUs, such as the iterative loop to traversing a packet through the grid. Modifications will still be necessary for this algorithm to be implemented efficiently on the CELL processor [Wald et al. 2006].

3 Implementation and Lessons Learned

This project was originally intended to be a joint project with Charles Lohr to implement an optimized Ray Tracer on a Playstation 3, where Mr. Lohr would develop the ray-object intersection routines, and this project would implement the RAS. The implementation of this project is unfinished and parts of it are untested. The RAS is not functional enough to be combined with Mr. Lohr's raytracer. But this project offers many lessons learned, which provide a theoretical contribution to SIMD Ray acceleration. We have modified the Coherent Grid Traversal algorithm in a way which we expect to perform efficiently on the CELL architecture, and describe the design decisions that provide a theoretical contribution to researchers that plan to develop this or similar acceleration schemes on similar SIMD architecture. empirical performance analysis, unfortunately, is not yet available due to the unfinished state of the implementation as of this writing.

We have learned many lessons about how to implement Coherent Grid Traversal on the Cell. Foremost, we have learned to make special considerations to work around the processor's weaknesses which we describe in detail. By doing so we expect to receive great performance increases as compared to implementations for traditional processors.

3.1 Bandwidth Limitations

At High Definition resolutions, the simple act of rendering pixels provides challenging limits on bandwidth usage. Resources must be expended to transfer pixels from the processor to the framebuffer. We have noticed on the Playstation 3, at 720p, simple programs that we have written to test the rate of writing pixels to the framebuffer run between 30 and 60 frames per second, which is significantly slow because our implementation targets real-time rendering. This imposes limitations on our implementation, because this leaves little bandwidth between processor and RAM to perform anything other than writing pixels.

A lesson learned is that the bandwidth required to write pixels is another reason to avoid large scenes with more than several thousand objects, because these scenes, along with their supporting datastructures, would not fit entirely on the CELL SPU's local store. Rendering of large scenes would not only require loading of objects from RAM require an efficient method of software caching, it would require even more expensive bandwidth from the processor to RAM, which would compete with the already slow bandwidth from processor to framebuffer. Efficiently loading objects from RAM, is clearly a very hard problem that we have not attempted to solve. We assume that all objects reside on the 256KB local storage.

We have noticed that the task of partitioning screen pixels into parallel chunks is a difficult consideration. Horizontal scanlines are the fastest way to write pixels on the Playstation 3. Unfortunately, they cause difficulty for the Coherent Grid Traversal. The frustum traversal requires that frustums of rays are generated from coherent packets. These packets can be effectively produced by assigning all of the rays from a block of pixels on the screen. Unfortunately, the framebuffer effectively requires that the screen is partitioned into scanlines rather than blocks. We are currently implementing a solution for which the screen is partitioned into blocks of scanlines, and that these partitions are processed in parallel on the SPUs. for 720p, the blocks of scanlines are 4 pixels vertical and 1280 pixels horizontal, and thus can be generated as blocks but written to the framebuffer in scanline order. This solution will work for 720p, but is not scalable to higher resolutions. The RGBA framebuffer requires 4 bytes per pixel, and we doubt that we will implement larger blocks because the total temporary storage required by this method is 20 KB, which is already a significant portion of the 256KB available on the SPU. The small vertical resolution means that we cannot traverse coherent blocks larger than 4x4 pixels. According to Wald et al. [2006], is an efficient size for 1024x1024 which is a comparable resolution to 720p. Unfortunately, larger screen resolutions would require even more storage space on the SPU, which we would not be able to acquire.

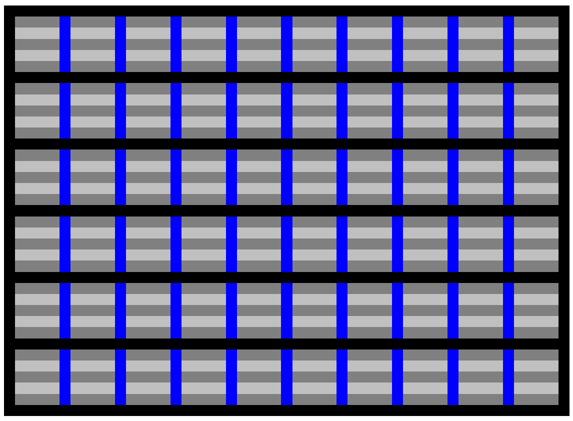

Figure 1: *Example of partitioning the screen into blocks of scanlines. Blocks are rectangular regions of pixels, where scanlines are horizontal lines of pixels. Notice how the each four scanlines are combined into a long block, but each block is processed by the ray tracer in 4x4 chunks delimited by the blue separators in the figure.*

3.2 Storage Limitations

Each SPU contains 256KB local store, which is does not directly interface with Random Access Memory, unless explicitly told to do so. Due in part to time constraints, and in part to obtain the maximum possible performance, we will restrict the scenes that we wish to Ray trace to those that will fit entirely onto the SPU's local store. Due to the mirrored programming model, this means that all of the contents of the entire scene, including the Uniform Grid, data objects, program logic, and temporary storage must reside on the local store of a single SPU, and this local store will be mirrored across all of the SPUs.

Due to this tight use of memory, we intend to use grid sizes smaller than the "ideal" 5 voxels per primitive shown by Wald et al. [2006]. There are many reasons for this. One reason is because we feel that the storage required for large grids will be a limiting factor. We will supply 32 bytes per grid cell in order to specify references to contained primitives. For a model with 2,000 spheres, a 10,000 voxel grid would require 312.5 KB, which is more than the 256KB available for all data combined. We do not expect to have storage for grids larger than 16x16x16, which would require over 130KB. Another reason to use smaller grids is to downplay the importance of mailboxing, because we do not intend to implement mailboxing. Mailboxing is an optimization for Coherent Grid Traversals, for which objects that reside in multiple grid cells are only intersected once, by keeping track of which objects have already been intersected, and allowing intersections only with new objects. We do not believe that the constant-time cost of ray-object intersection is as severe on the Playstation 3 as it is on a traditional computer, because we do not load objects from RAM. However, we believe that the cost of mailboxing, is more expensive on a Playstation 3, because a fair amount of branching is required in order to keep track of the history of intersected objects. By using a smaller grid, we reduce the necessity of mailboxing in two ways. Firstly, it is less likely that objects will span the multiple grid cells, because the size of a grid cell increases is the number of cells decreases. Secondly, there are fewer grid cells to traverse, which means that even less time will be spent on redundant intersections. We feel that smaller grid sizes are applicable to our implementation in several ways. However, the drawback of this strategy is that more objects reside in each grid cell, which makes frustum culling even more important.

Frustum culling is an optimization to Coherent Grid Traversal that we intend to implement. Although we have not written the code for this yet, the algorithm is very clear, because given four frustum planes, it is trivial to determine whether a sphere intersects the frustum, just by comparing the sphere's position with the plane equations. Only objects that intersect the bounding frustum are tested for intersection with the rays within the frustum. The overhead induced by frustum culling is that each object must be tested for intersection with the frustum before it is tested with the individual rays. In addition, an unpredictable branch is required, where if the object is not in the frustum, then it will not be tested for the more complicated ray-sphere intersections tests provided by Mr. Lohr. One might expect that this piece of the algorithm is not apparently suitable for the Playstation 3 because of it's use of branching, but we contribute that much as the case with the CPU implementation, we expect that frustum culling should still provide a large speed increase for the CELL Coherent Grid Traversal, because the complexity of code required to frustum cull is still significantly less than Mr. Lohr's code to provide 16 rays with sphere intersection tests. As a result, many could be quickly discarded by frustum culling before attempting to perform the intersection tests. This is similar to what Wald et al. [2006] found for frustum culling performance benefit on traditional CPUs, as the number of ray-object intersections was reduced by a factor of 4 - 8. Since we are using smaller grid sizes than Wald et al. [2006], and not implementing the mailboxing optimization, frustum culling will likely provide even more benefit.

3.3 Branching Limitations

Although Coherent Grid Traversal is non-hierarchical and requires no recursion, it still has a large demand for looping, which if mispredicted, separates the basic blocks of execution, and causes many pipeline stalls. On the CELL SPU, branch hinting is also very limited compared to than of traditional processors. For if statements, it is often less expensive to compute both clauses of the statement, and then undo the incorrect clause using bitwise operations. Branch prediction does not exist on the CELL SPUs to the extent that it exists for traditional processors. However, branch hint instructions allow for fast execution provided that the hint is able to correctly predict the branch. The algorithm for Coherent Grid Traversal is described in Figure 3 Top. The algorithm requires that for any particular bounding frustum of rays, all of the grid cells that intersect the frustum are traversed outwardly from the camera. The outer loop traverses slice by slice along the axis that is most parallel to the direction of the Frustum. The two inner loops traverse the square regions within each slice. Unfortunately, this exact looping structure

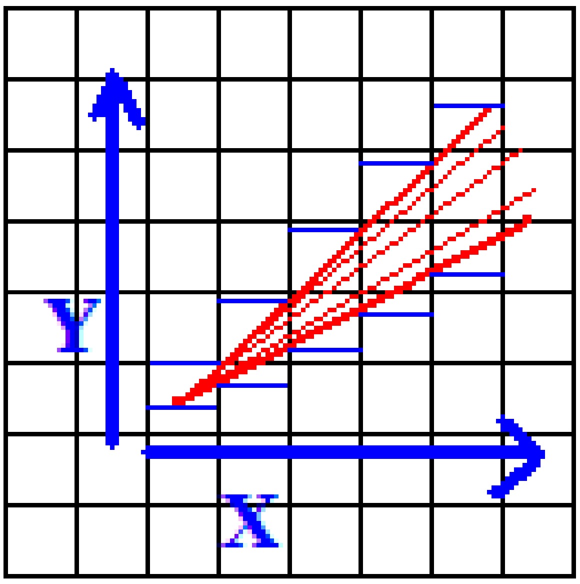

Figure 2: *2D Example of Coherent Grid Traversal. Assuming that the frustum is aligned primarily with the X axis such as with this example, Coherent Grid Traversal requires that for every unit of X axis going outwardly from the camera, every vertical slice must be accessed, and for every vertical slice, the blue lines, denoting the frustum boundaries show which grid cells must be traversed. Coherent Grid Traversal extends this traversal algorithm into 3D, by using a 2D traversal for every slice.*

does not map well to the CELL processor. The outer loop cannot be predicted, because it only decrements the number of slices and then calls the inner loops, and the Cell Processor cannot issue a branch hint to such a complicated structure. Also, the inner loops cannot be predicted. The inner loop could execute between 1 and 3 iterations, which is not enough to merit branch prediction, because either an "always predict branch" or "always predict no branch" would frequently mis-predict. As a result, the obvious traversal pseudocode will mis-predict over 50

Rather than using these 3 nested loops, We decided to perform one single loop that traverses grid cells in the same order as these 3 nested loops (See Figure 3 Bottom). This code looks more complicated but it maps better to the Cell architecture. The if/else can be implemented without branching. Thus, this single loop is simple enough for the Cell processor to issue branch hints to improve looping performance.

By using the "always predicting branch" strategy, this loop will only mis-predict the branch of the last iteration. The drawback is that the solution to each of the 3 if/else clauses must be computed for every iteration, and then the incorrect clauses must be forgotten. However, on an architecture like CELL, where branch prediction is not very powerful, it is beneficial to avoid mis-prediction as much as possible. This code for traversal has been written, but not tested or benchmarked, so we can only theorize that the modified loop is faster than the 3 nested loops on CELL architecture.

3.4 Efficient Voxelization

Animated scenes require modification to the uniform grid datastructure. We will allow for any animation by voxelizing the uniform grid every frame. Because we use the mirrored parallel programming model for the CELL SPUs, we need every SPU to voxelize the scene in order to insert data into the uniform grid, thus this voxelization will occur once for every SPU. The voxels will contain 32 bytes each in order to specify their contained primitives to intersect, these 32 bytes form a static list of 16 2-byte references to objects. Assuming that objects are smaller than grid cells, we can apply a simplified voxelization scheme that requires less code and running time, and that produces no false negatives, and thus will not reduce the resulting image quality. See figure 4 for a 2D illustration of the following algorithm. For each object, a bounding box is computed, and each of the eight corner points is voxelized. This simplified approach is acceptable only under the assumption that the grid-cells are larger than the objects. This assumption is reasonable, because it is consistent with other assumptions made throughout the project. Namely, that all of the grid voxels fit within 150KB local storage, which means that grid cells must be relatively large. It is also important to notice that this method is very efficient for the CELL SPU to voxelize, primarily, because the CELL will always attempt to voxelize exactly 8 points. Because the number of points is known to be a small constant, namely eight, no looping is required to determine each point to voxelize. Moreover, voxelization is inherently parallelizable, because each of the 8 corner points can be voxelized independently.

It is necessary that each object be voxelized quickly, and voxelized only once. For example, if all eight corner points lie withing a single voxel, then the object should be listed only once, not eight times. We ensure both properties elegantly by using a fixed sized queue for each voxel, and by writing to each queue in parallel. Each voxel contains a queue of up to 16 two byte indexes. Each index is a reference to a particular object that intersects or resides within a grid cell. When an object is voxelized, it's reference is pushed onto the 32 byte voxel queue. For every object, the queues that correspond to the voxels of the eight corner points are each computed separately and in parallel. This means that if two or more corner points happen to reside within the same grid cell, then the object would only be gridded once within that grid cell. For example, let's assume that the sphere, and thus it's bounding box, are entirely contained within a single grid cell. Because each corner point would perform it's own independent push operation on the same 32 byte queue, they would all produce the same resulting queue, and this value would be written to the same location in memory eight times (once for each duplicate corner point), which is the same as a single voxelization. On the other extreme, if all eight corner points lie in different voxels, then each of the eight queues will be written to only once, so the object will be properly voxelized into all eight cells. Due to the queue's small size, pushing an element onto a queue requires only 6 SIMD logical instructions, because appending a new element to a queue only requires that all 32 bytes be shifted left by two bytes, and then the new element written to the first two bytes.

4 Conclusion and Discussion

We describe a solution for ray acceleration on the Playstation 3 that should perform efficiently at High Definition resolutions, and we believe will allow for Interactive or Real Time ray-tracing at these resolutions of scenes with several hundred to a few thousand spheres, provided that the scene does not make use of secondary rays. The implementation modifies Coherent Grid Traversal to perform efficiently on the Playstation 3 architecture, and we describe the parts of this modification that provide an intellectual contribution. The described system is designed and implemented in fragments, and we are looking forward to debugged it so that it can be benchmarked.

The most immediate step of future work is to finish debugging the implementation. The voxelization algorithm is working perfectly, but the looping algorithm still has yet to be debugged properly. We believe that our alternate looping structure described in the paper has benefits over the traditional loop, but we have yet to construct hard empirical statistics from both methods.

An eventual goal would be for this system to be used as part of a computer game or simulation, where the simulation is executing on the PowerPC core, and the real time ray-tracer is executing on the SPUs. We feel that we are taking a step in this direction by writing all of the code onto the SPUs, in order to leave the traditional PowerPC cores available for execution of traditional programs such as games. However, we recognize that more research must be done beyond the scope of this project in order for games or simulations to be practical using our implementation. For example, games tend to make use of meshes and particle systems that require very large storage of many triangles, whereas our implementation requires that all objects, datastructures, and binaries reside entirely in the local storage of each SPU. It is worth exploring ways of batching traditional meshes from RAM, but we believe that many batched Stochastic features in games such as terrain and particle systems could be replaced with procedural features such as fractals or noise which require significantly less storage. We leave these directions open to future research.

References

BENTHIN, C., WALD, I., SCHERBAUM, M., AND FRIEDRICH, H. 2006. Ray tracing on the cell processor. *IEEE Symposium on Interactive Ray Tracing 2006* (September), 15–23.

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM Press, New York, NY, USA, 15–22.

GÜNTHER, J., FRIEDRICH, H., WALD, I., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum 25*, 3 (Sept.), 517–525. (Proceedings of Eurographics).

HAITAO DU, NOZAR TABRIZI, Y. L. N. B. M. F. 2003. Interactive ray tracing on reconfigurable SIMD morphosys. *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 144–149. ISSN 1530-1591.

PHAM, D., ASANO, S., BOLLIGER, M., DAY, M. N., HOFSTEE, H. P., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., RILEY, M., SHIPPY, D., STASLAK, D., SUZUOKI, M., WANG, M., WARNOCK, J., WEITZEI, S., WENDEL, D., YAMAZAKI, T., AND YAZAWA, K. 2005. The design and implementation of a first generation cell processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International 1* (February), 184–592.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3, 1176–1185.

SZIRMAY-KALOS, L., ASZODI, B., LAZANYI, I., AND PREMECZ, M. 2005. Approximate ray-tracing on the GPU with distance impostors. *COMPUTER GRAPHICS FORUM 24*, 3, 695–704. ISSN 0167-7055.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM Press, New York, NY, USA, 485–493.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1.

WYVILL, G. 1988. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer 4*, 2 (March), 65–83. ISSN 0178-2789.

```
for (i = startSlice to endSlice)
{
   for (j = lowerBoundY to upperBoundY)
   {
      for (k = lowerBoundZ to upperBoundZ)
      {
         ComputeIntersection( cell[i, j, k] );
      }
   }
}
```

```
i = startSlice
j = startY
k = startZ
while (i <= endSlice)
{
   ComputeIntersection( cell[i, j, k] );

   if (j < endY)
   {
       j+=1
   }
   else if (k < endZ)
   {
      j = startY
      k+=1
   }
   else
   {
      i+=1
      j = startY
      k = startZ
   }
}
```

Figure 3: *Pseudocode for different Coherent Grid Traversal implementations, assuming that the frustum is primarily aligned to the X axis. Top shows the original implementation. Bottom shows the modified version for Playstation 3. Notice that the if/else blocks do not require branching, because all 3 clauses can be computed, and bitwise operations can be used to remember only the correct result. Therefore, 3 loops (Left) are compressed into 1 (Right).*

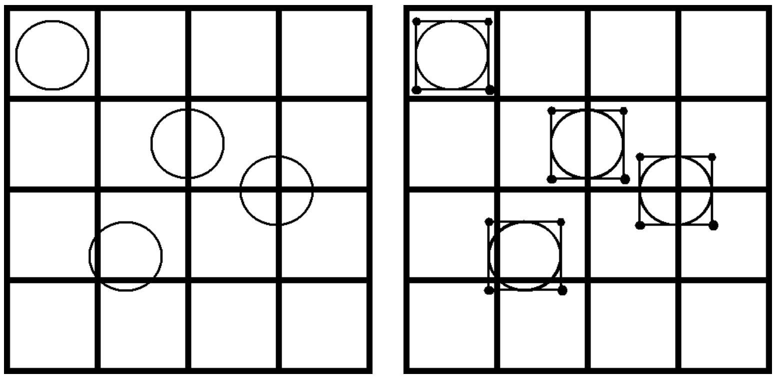

Figure 4: *2D Example of Voxelization algorithm. Bounding boxes are computed for every sphere, and each of the corner points of the box is voxilized. Assuming that the spheres are smaller than the grid cells, this algorithm produces no false negatives, which means that any subsequent ray-tracing will be correct. The lower left sphere shows an example of a false positive, where the lower left grid cell does not contain the sphere, but a bounding box contains a point in this cell. We expect this case to be rare, but even so, it only causes a minor performance drop during ray-tracing. It would not lead to incorrect results, just unnecessary intersection tests.*