Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Streamed Ray Tracing of Single Rays on the Cell Processor

Florian Bingel† and Andre Hinkenjann‡

Bonn-Rhein-Sieg University of Applied Sciences, Sankt Augustin, Germany

Abstract

In this paper we present an approach to efficiently trace single rays on the Cell Processor, instead of using ray packets. To benefit from the performance of this processor, a data structure is chosen which allows traversal without excessive accesses to main memory. Together with careful optimization for SIMD processing, a performance comparable to a packet based ray tracer, running on the same hardware, is achieved. In special cases, when the coherency of the traced rays get very low, it even outperforms the packet based approach.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray Tracing;

1. Introduction

Most of today's real time ray tracing approaches support ray packets to benefit from the coherence of the rays. This works well for simple ray casting algorithms and ray tracing of scenes producing highly coherent secondary rays. For transparent objects like windows or flat mirrors this usually is the case, while curved surfaces like lenses, liquids or bumpy structures can lead to a wide spreading of secondary rays. Also, in more sophisticated global illumination algorithms the rays sampling a scene in several ways are not necessarily coherent. In that case the performance of packet based ray tracers usually degenerates to a fraction of the speed compared to tracing coherent rays. To overcome this issue, it is useful to trace only single rays and to take advantage of the coherence of the rays, when its available.

In this paper we present a ray tracing approach that utilises the Cell Broadband Engine (Cell B.E.), known as the processor of Sony's Playstation 3, to trace single rays using well known structures like Uniform Grids. A streaming architecture, which does not depend on efficient caching, is used to keep a high workload on the processor. Special techniques from the Cell B.E. architecture are used to overcome the high latency when accessing main memory and to utilize the SIMD-capabilities of the processor. The Cell processor contains a so called Power Processing Element (PPE), which is a common general purpose processor. It is not very fast, contains 2 MByte second level cache and is intended for controlling. The compute-power of the Cell is provided by eight Synergistic Processing Elements (SPE). This simple but very fast SIMD-Processors each contain a 256 KByte Local Store (LS), which is located in the processing element and therefore can be accessed very fast (within 6 cycles). In contrast, the SPEs cannot directly access the main memory. Instead, special direct memory access (DMA) - instructions load from and store to the main memory. All data necessary for a program and the program itself has to fit into the local store. This memory hierarchy makes the development more complicated, but the programmer can completely control which data is in the LS. A software managed cache, provided by IBM's Cell-SDK [cel09], can be used if the memory access pattern makes it reasonable. The documentation recommends the use of double-/multi buffering schemes, where parts of the needed data is loaded into the LS and processed, while the loading of another part is in progress. This is meant to overcome the limitations of the size of the LS and to hide the latency of several hundred or thousands of processor cycles while loading/storing data from/to the main memory.

2. Related Work

Since its release in 2006, few ray tracers for the Cell processor were published. A demo version of IBM's Interac-

† e-mail: florian.bingel@h-brs.de
‡ e-mail: andre.hinkenjann@h-brs.de

tive Ray Tracer (iRT) is available for free and uses precomputed datasets that are rendered either directly on a Playstation 3 via a joy-pad controlled interface or remotely on the QS21/QS22 hardware. It allows to render static scenes with simple shading, texturing and shadowing and supports ambient occlusion for global-illumination-like effects. The recursion depth limited to four is enough for simple scenes but not for advanced optical effects with many transmissive or reflective objects. In 2006 in [BWSF06] a packet based ray tracer was published. The algorithms for data structure traversal and intersection test, adopted from x86-implementations, where successfully transferred to the Cell architecture primarily using software caches and software-hyperthreading, which implements an advanced double-/multi-buffering approach. A cell ray tracer, based on the bounding interval hierarchy [WK06] data structure and 4*4 packet tracing was published in [BMH09] in 2009. In combination with the VR-Framework *basho* [HM04] a complete immersive visualization system is available which provides a modern software architecture using plugins. This ray tracer was used for comparison to the approach introduced in chapter 4. In [Hap09] a comparison of a BIH and a kd-Tree implementation of a ray tracer on the Cell was published. Different techniques are evaluated to speed up the rendering process, but the implementation suffers from non optimal load balancing between the PPE and the SPEs.

Several recent papers address the problem of coherent and especially incoherent rays. In [MMAM07] a large buffer is used to store generated rays and sort them to discover coherence for a common packet tracer. This generates a large memory footprint and does not pay off, according to their results. Another approach, as published in [WBB08] or [DHH*08], implements single ray ray tracing on a flattened bounding volume hierarchy. This addresses the SIMD capabilities of current and future processors and showed a performance comparable to current packet tracers when rendering scenes producing many incoherent rays. In [BWB08] a packet tracer uses big packets which are merged with other packets as the number of masked rays increases. The inactive rays can be removed from the rendering loop this way, but a clever way of combining packets containing coherent rays is needed.

3. Streamed Ray Tracing

Our single ray tracing approach uses a typical uniform grid for acceleration. There are two important reasons for this. At first, a uniform grid is a regular data structure. This allows for traversal without doing excessive accesses to main memory. Only the location of the grid (usually the bounding box of an object), its resolution (in cells / dimension) and a small bit-lookup-array are needed. This array contains one bit for every grid cell which indicates if a cell contains objects. By pre-loading this data into the local store of the SPEs, it is possible to traverse the data structure until a non-empty cell is intersected. The second reason for using a uniform grid is that during traversal only a small amount of data needs to be stored (e.g. current cell), and not a stack which is usually needed while traversing tree-like structures, like BVHs or kd-trees. This allows traversing many rays consecutively and to use this as a multi-buffering scheme for overlapping memory accesses. To be independent of any coherence of the rays, the triangle data, contained in the grid cells, is not cached. Instead it is loaded from main memory to a buffer in SPEs local store during traversal. The rays are streamed through the several steps of the ray tracer. This scheme is similar to that in [MMAM07], but does no sorting of the rays at all. The rays are processed in the order they occur. One condition for that scheme is an iterative approach, instead of doing a common recursive ray tracing (like in [MMAM07]).

3.1. Construction of the Uniform Grid

The construction of the grid is completely done on the PPU using common approaches. This means that the build is not very fast, most effort was made to enhance the rendering part. But the construction of a grid is generally very fast, compared to other data structures (for construction times, see chapter 4). An approach to efficiently implement this on a streaming processor was published in [KS09] and we are considering to use this for our system.

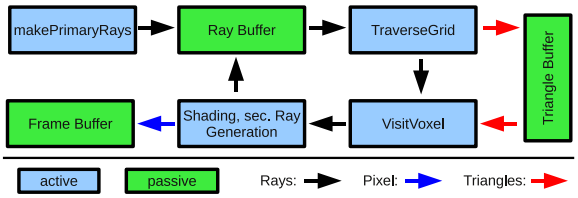

Figure 1: *The ray pipeline. The processing elements are blue, the green ones are buffering elements for rays, pixel and triangles. The arrows show the flow of the data.*

3.2. The Ray Pipeline

Figure 1 shows the steps each ray takes until the final pixel color is calculated. At first, a fixed number of eight rays is generated and written into a ray buffer. These rays are processed at the same time, but completely independent from each other. In one iteration, eight rays are read from the buffer and traversed through the grid until the first hit of a non-empty cell. The data for these cells is asynchronously loaded into the triangle-buffer. When the traversal for all eight rays is done, the triangle data of the first processed ray usually is already loaded and it can be intersected against the triangles. This is done in "VisitVoxel". If an intersection was found, additional data (normals, texture coordinates,...) of the hit triangle is asynchronously loaded. If no intersection was found, the ray is written back into the ray buffer. This step is repeated for all of the eight rays. After finishing this part, shading of the hit points is done for all rays that hit

a triangle. In the shading-routine the material data of the hit triangles is loaded from a software managed cache [cel09]. This works well because usually the memory footprint of the complete material data is small. Also the secondary rays are generated here and written into the ray buffer. Now the ray buffer is checked, if it holds less than eight rays. If this is the case, new primary rays are generated. If not, the processing continues with the traversal. The ray buffer is implemented as a ring buffer and is accessed in FIFO-style. The procedure continues until less than eight rays are left and no more primary rays can be generated. Since this covers only a small fraction of all rays and only occurs at the end of a rendering step, these last rays are processed individually, without any interleaving of the memory accesses. A more dynamic processing of different numbers of rays would generally lead to a bigger overhead and is not suitable for a streaming processor like a SPE.

3.3. SIMD-Traversal of the Grid

The grid traversal of the rays is a hot spot of the ray tracer. Depending on the scene, this part can take more than 50% of the total run-time. Therefore, this part is heavily optimised. The traversal is implemented according to the algorithm of [AW87], but without any branches. The use of SIMD programming features and intrinsics make it possible to traverse a ray in only 21 processor cycles per grid cell.

Listing 1: *The while loop of the traversal of the grid cells.*

```
while(!out)
{
  tempCellIndex = cellIndex;
  vector uint bitIndex = spu_rlmaskqw(tempCellIndex,3);

  //shift one bit to the correct position for the current cell
  vector uint bitVectorVal = spu_slqw(oneOne,tempCellIndex);

  //compare all three comp. of tMax to find smallest Element
  vector unsigned int compare = spu_cmpgt(tMaxRotate1, tMax);
  vector unsigned int compare2 = spu_cmpeq(tMaxRotate1, tMax);
  compare = spu_or(compare,compare2);
  tMaxRotate2 = (vector float)spu_cmpgt(tMaxRotate2, tMax);
  compare = spu_and(compare,(vector unsigned int)tMaxRotate2);

  //add smallest element to currCell and tMax
  vector signed int currStep = spu_and(step, compare);
  vector float currDelta = spu_and(tDelta, compare);
  currCell = spu_add(currCell,currStep);
  tMax = spu_add(tMax,currDelta);
  tMaxRotate1 = spu_shuffle(tMax,tMax, rot3_1);
  tMaxRotate2 = spu_shuffle(tMax,tMax, rot3_2);
  cellIndex = spu_splats(spu_extract(cellIndex, 0));
  cellIndex = spu_add(cellIndex,indexStep);
  vector unsigned int outOfBox = spu_cmpeq(currCell,justOut);
  outOfBox = spu_orx(outOfBox);

  //Load the Bitvector into a register
  vector uint lookupVector = si_lqx(bitLookupVector,bitIndex);

  //rotate the bit for the current cell to the first byte
  lookupVector = spu_rlqwbytebc(lookupVector,tempCellIndex);
  voxelNotEmpty = spu_and(lookupVector,bitVectorVal);
  outOfBox = spu_or(outOfBox,voxelNotEmpty);
  cellIndex = spu_and(cellIndex,(vector signed int)compare);
  cellIndex = spu_orx(cellIndex);
  out = spu_extract(outOfBox,0);
}
```

Listing 1 shows the main part of the traversal, the while-loop where one ray is traversed through one grid cell. Please note that this loop computes values from consecutive iterations. While the current cell's (stored in "tempCellIndex") bit value is loaded, the next cell is computed in "cellIndex". The six cycles latency only of the "si_lqx()" instruction, which loads from the local store, allows this on-demand loading. The compiler reorders the C-instructions above to minimize dependency stalls of the instructions. The loop exits if the ray leaves the grid or if a non-empty cell was hit.

3.4. Atomic Read of the Grid Information

During the traversal of the grid, when a non-empty cell is found, the number of triangles in that cell as well as their location in main memory is not known. An array, containing this information for every grid cell is located in main memory. Since it should not be loaded using a cache, due to the possible incoherency of the rays, a different way of loading has to be used. The Cell B.E. architecture provides a low latency atomic access to main memory, intended for synchronization purposes of the SPEs. This operation implements a high priority, 128 Byte load in about 200 - 300 processor cycles, opposed to at least 800 - 900 cycles latency needed for a standard DMA access. By using atomic loads, the number of the triangles in the current grid cell and a pointer to the triangles in main memory can be loaded very efficiently without the use of any caching. Table 1 gives a comparison of different possibilities of loading the two values. For the benchmarking a very cache friendly dataset was used, so the times for the software cache are best case values and can be higher in practice. It should be mentioned that the "atomic" nature of the load is not needed at all, only the high priority and the short latency makes it useful for that case. Due to the property of the implementation of this instruction (getllar()), which does not lock the memory but just reserves it, accesses from other SPEs to the same memory location are possible.

Table 1: *The table shows the average cycles needed to load 8 bytes from main memory. By using an asynchronous atomic load, the latency can be minimized without the need of a software managed cache. This allows completely incoherent, but efficient loads.*

SW-Cache	DMA sync.	DMA async.	Atomic sync.	Atomic async.
315	986	609	470	269

3.5. The Micro-Cache

After the traversal for every ray (processed in one iteration) a buffer is needed to hold the triangle data of the grid cell that was traversed. It makes sense to reuse that data when a ray intersects the same grid cell as one of its direct predecessors.

By using a simple comparison of the grid cell indices, a ray, coherent to another ray, can use the grid information and the triangle buffer already loaded in the same iteration of the ray pipeline. A very small size of 8 entries, using 2 vectors each containing 4 grid cell indices, and the SIMD commands of the SPEs make it possible to test for already loaded grid cells with virtually no overhead cost. In scenes producing high coherent rays this can lead to a speed-up of about 20%, whereas in scenes producing low-coherent rays no drawback is noticed due to the low overhead of the "caching".

3.6. Triangle Intersection

To intersect the triangles, two different algorithms were tested. To take advantage of the SIMD capabilities, the triangles are always organized in packets of four. The packet size matches the SIMD width of the SPEs.

We compared two algorithms: a version from the Cell SDK, included in the example library [cel09], based on the well known Moeller-Trumbore-Test [MT97], and a SIMD-optimized test [MSK07] are used to intersect one ray with four triangles at a time. The loop, iterating over the triangles always processes two triangle packets at once. This loop-unrolling showed an increase of performance of about 10% because of a higher processor load. Shevtsov's algorithm performed only slightly better than the SDK-Version. Because of a more expensive pre-calculation during build-up of the dataset, needed for Shetsov's version, the SDK-Version was used.

3.7. Shading, Texturing and Secondary Ray Generation

After a hit was found, shading is started. The implementation currently supports phong shading and texturing. This is the only part of the whole system where caches are used. The material data usually consists of some ten materials, allowing an efficient caching even without good coherency. The textures are also loaded using a cache, which showed a bigger but acceptable impact. Future versions should get rid of the caches and load that data directly from main memory, similar to the loading of triangle data. The shading routine also includes the generation of the secondary rays. They are written into the ray buffer to be processed in a subsequent iteration of the whole process. Since these rays emerge in a grid cell whose data is still in the triangle buffer, they are directly intersected with the triangles. This prevents an additional load of that data when first traversing these rays. This optimization resulted in a speed-up of 1 to 7%.

3.8. Extension to a Simple Hierarchy

The bit array, indicating if a grid cell is occupied or not, must reside completely in the local store of the SPEs during traversal. Since only about 100 KByte can be used for this, the size of the grid is limited to about 800000 cells. The performance of larger scenes, like the "fairy" scene (cf. figure 4) suffers from this limitation. To overcome this problem, a simple hierarchy was included in an extended version of the ray tracer. Every grid cell, containing more than a fixed number of triangles, contains another uniform grid consisting of 8*8*8 cells. When loading the grid information (see chapter 3.4), the complete information needed to traverse this small grid can be loaded from main memory in one step. The atomic load always performs a 128 Bytes load, so the bit array for 8^3 cells, as well as additional information like pointers to triangle data, fits perfectly into this size. The additional traversal step is done after the Traversal() and before the VisitVoxel() steps of the ray pipeline. This way the extended grid can efficiently handle much bigger scenes than the simple uniform grid.

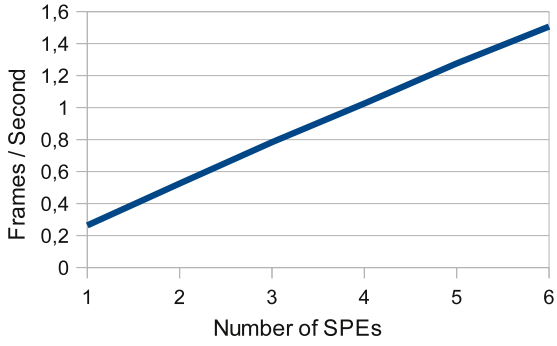

Figure 2: *Render times for the "eye"-scene, rendered with one to six SPEs. The renderer shows a nearly linear speed-up because all renderers run completely independent. The small divergence is a result of the heavy load to the memory controller which connects all SPEs to the main memory.*

3.9. Parallelization and Load Balancing

Since each SPE runs a complete instance of the ray tracer, and since there are six SPEs available on a Playstation 3, six individual renderers can be run. The process of the ray tracing itself is easily parallelized. A "sort first" [MCEF08] approach is utilized by dividing the view port into tiles and assigning each SPE one tile at a time. This is done by a global counter which all SPEs read and increment using an atomic access. The next tile to be rendered can be calculated from the counter value and the predefined tile size. The speed-up for using one to six SPEs is nearly linear (cf. figure 2). The memory controller which connects all SPEs to main memory limits the parallel processing by only a few percent. Different tile sizes and tile shapes where tested. Figure 3 shows that the performance of the renderer is heavily dependent of the tile size. By rendering small tiles of only $8 * 8$ pixels the overhead for synchronization limits the rendering speed. The blue bars show a performance increase for square tiles up to $32 * 32$ pixels. The red bars show the render times for equally

sized blocks of 1280 pixels, beginning with a nearly square tile ($40 * 32$) to one single line of the image. Rendering a single line does provide the best memory access pattern, but suffers from a low coherency of the rendered tiles. Since the pipeline of the renderer uses only the coherency of the 8 pixels rendered at once (see chapter 3.5), it is more efficient to generate $4 * 2$ Rays, instead of a row of 8. This can be done by assigning tiles of a size of $640 * 2$ pixels to render, the best trade-of of a good memory access and a high coherency for the rendering step. Due to the limitations of the SPEs local store, it is not possible to use tiles with more than 1280 pixels.

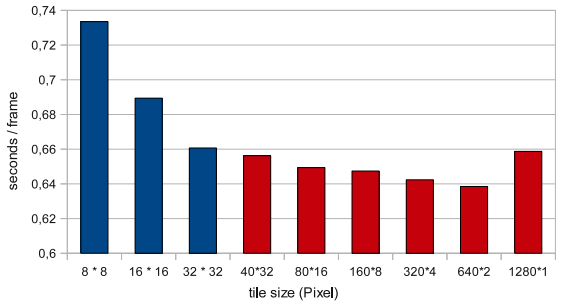

Figure 3: *Different tile sizes and tile shapes result in different rendering times. Larger tile sizes lower the synchronization overhead. The maximum possible size of 1280 pixels in different shapes is shown by the red bars. A more line-based setting provides a faster memory access. Rendering single lines (last bar) does not provide good coherency of the rays, thus lowering the speed.*

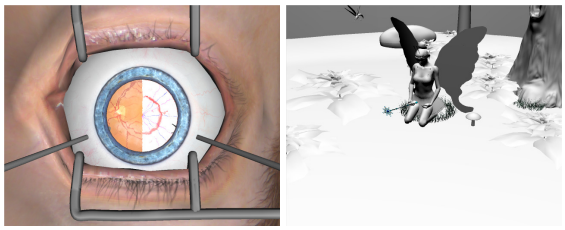

Figure 4: *Different scenes ray traced with a* $1280 * 1024$ *pixel resolution. The first scene is an eye model of an eye surgery simulator (courtesy VRmagic [VRm09]). The second scene (FairyForest scene from [fai09]) is provided to compare the results to other ray tracers.*

4. Results

To measure the efficiency of the presented ray tracing approach, three different scenes where tested. Figure 4 shows two of the scenes. The eye scene shows a model of a human eye, part of an eye-surgery simulator. It consists of 50.000 triangles, but many transparent layers generate many secondary rays. The next scene is well known from many other publications and is taken from [fai09]. It is made of 170.000 triangles which takes advantage of the hierarchical grid, where the simple uniform grid does not perform well. In figure 7 a scene, called "spherebox", is shown which was constructed to force the generation of incoherent rays. This scene is also made of about 50.000 triangles. The resolution of all renderings was $1280 * 1024$. To compare the results to a common packet based ray tracer, an implementation of the BIH data structure [WK06] in combination with a $4 * 4$ packet tracing was used, which is further described in [BMH09]. It runs on the Cell, using 6 available SPEs of a Playstation 3. It should be mentioned that the BIH does not perform as good as a well constructed kd-Tree or BVH, but allows for very fast build times. This means that it is possible to render dynamic scenes. An implementation of the BIH compiler on one SPE, like the one mentioned in [WRH09], shows a speed-up of up to seven compared to the PPU implementation. The data structure builds in this paper are done on the PPU, but a future implementation of the grid-build on the SPE is expected to show a similar behaviour.

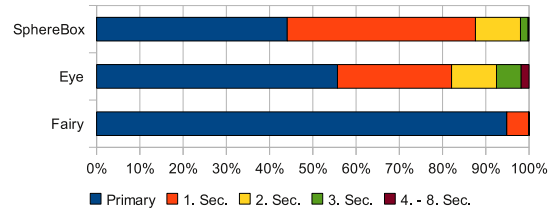

Figure 5: *The relation of primary and secondary rays for recursion depths up to 8 for the scenes "spherebox" (fig. 7), "eye" and "Fairy Forrest" (fig. 4).*

4.1. Ray Distributions

The different scenes chosen generate different characteristics of ray-distributions. In the FairyForest scene nearly all rays are primary rays, thus having a high degree of coherence while rendering. The eye scene does generate many secondary rays, but they are very coherent due to the fact, that the lens system of the eye bundles the rays, and some transparent layers in the eye do not spread the rays at all. The spheres scene represents a scene generating many secondary rays having a wide spreading, resulting in low ray coherence. Figure 5 shows the fractions of the rays of different recursion depths to all rays traced.

4.2. Benchmarks

In figure 6 three charts show the results for three different ray tracers. The first one uses the uniform grid as explained in this paper. The second one uses the extension to the simple hierarchical Grid as mentioned in chapter 3.8. The third

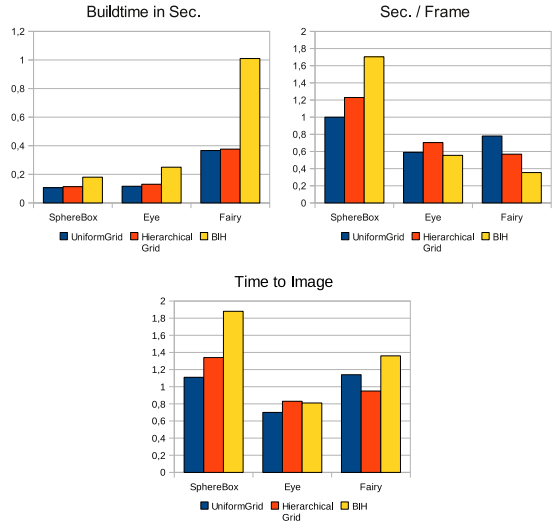

Figure 6: *The benchmarks for the three scenes, using the different ray tracers, show that the single ray approach can compete with the packet based BIH ray tracer.*

one is the packet-based ray tracer using a BIH data structure as mentioned above.

The first chart shows the buildtimes of the different data structures for the different scenes. As expected, both the uniform grid and the hierarchical grid are built faster than the BIH structure but the huge difference in building the "fairy"-scene is remarkable. For the two smaller scenes, "sphereBox" and "eye", even the slow PPU builds the data structures fast enough to reach interactive frame rates.

The second chart shows the pure render times for the same data structures and scenes. Because of many reflections, the "sphereBox" results in the largest render time. Both grid-based ray tracers are advantageous over the BIH ray tracer due to many incoherent rays in this scene. The ray packets lead to a low efficiency of the BIH traversal. In the "eye"-scene all ray tracers are almost equally fast, only the overhead of the hierarchical grid does not pay off for this small scene. It is interesting that the uniform grid can compete with the BIH even for highly coherent secondary rays produced in this scene. The "fairy" scene is the biggest scene here and it shows a clear advantage of the BIH ray tracer. Since most of the rays are primary rays, the ray packets benefit from the very high coherence of the rays. Using the uniform grid, this scene can not be rendered efficiently because of its limitations of the grid size. In this case the hierarchical grid is the better one of the grids.

Since our ray tracers are intended for rendering dynamic scenes, the time to image, the sum of both times mentioned above, is important. For both the "sphereBox" and "eye" scenes the relation of the times of the three ray tracers is similar to the render times, showing an advantage of the grids for the "sphereBox" and an almost equal performance for the "eye". For the "fairy" scene the BIH ray tracer suffers from the very slow buildtime, leading to even slower performance than the uniform grid.

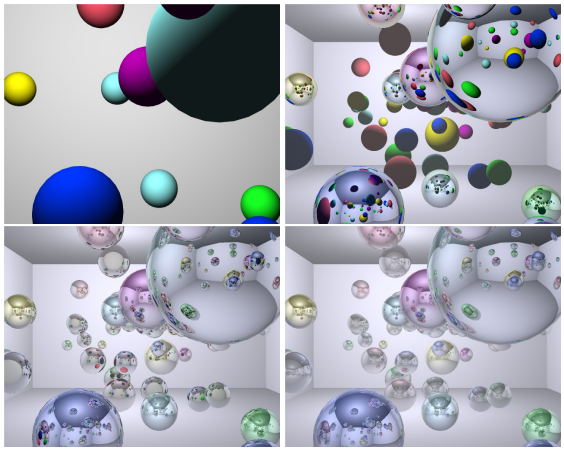

Figure 7: *The "spherebox" scene, rendered with recursion depth 0 (ray casting), 1, 2, and 9. The differences of the recursion depths from 3 to 9 are only hardly noticeable.*

4.3. The Impact of the Recursion Depth

To analyse the impact of the recursion depth to the render times, the "spherebox"-scene is used. In figure 7 this scene is rendered with different recursion depths. The difference in the images of depths 2 and 9 is very small, only a small fraction of rays is active in the third and following recursions (see figure 5). Figure 8 shows the render times for the three ray tracers for only primary rays and the recursion depths from 1 to 9. Starting at a depth of four, the render times for the grids are stable at 1 second. The few additional rays don't influence the rendering very much. For the BIH-ray tracer the times increase for every additional recursion step, which indicates a bad efficiency for the ray packets from a recursion depth of 3 upwards. This behaviour is an important feature of the single-ray ray tracers proposed here.

5. Conclusion and Future Work

We successfully implemented a ray tracer on the Cell processor which is capable of tracing single rays in a streaming fashion. It can compete with another ray tracer for this processor, which implements the traditional packet based tracing. Future work is to remove the dependency on the texture cache, allowing incoherency in the shading part, too. A drawback right now is the lack of shadows, an extension is needed. We would like to implement this approach on alternative platforms, like GPUs, to find out if the approach presented here is portable to other architectures.

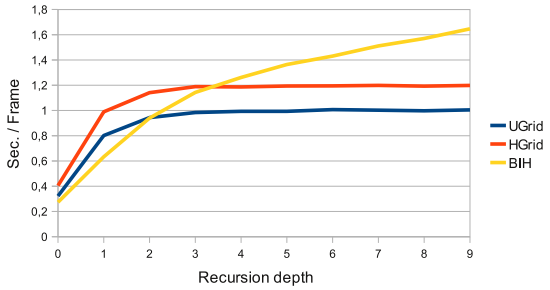

Figure 8: *The render times for the packet based BIH ray tracer increases even for recursion depths where not many rays are active any more. The single-ray ray tracers do not show this behaviour.*

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) under grant no 1762X07. We would like to thank VRmagic for providing the eye model and IBM Deutschland GmbH for the close co-operation.

References

[AW87] AMANTIDES J., WOO A.: A fast voxel traversal algorithm for ray tracing. In *Eurographics 1987* (1987).

[BMH09] BINGEL F., MANNUSSF., HINKENJANN A.: Ray tracing on a cell cluster for virtual environments. In *CGI '09: Computer Graphics International* (New York, NY, USA, 2009), ACM.

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive ray packet reordering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug. 2008), pp. 131–138.

[BWSF06] BENTHIN C., WALD I., SCHERBAUM M., FRIEDRICH H.: Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 25–23.

[cel09] Cell broadband engine resource center, December 2009. `http://www.ibm.com/developerworks/power/cell/`.

[DHH*08] DAMMERTZ, H., HANIKA, J., KELLER, A.: Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4 (June 2008), 1225–1233.

[fai09] The utah 3d animation repository, December 2009. `http://www.sci.utah.edu/ wald/animrep/`.

[Hap09] HAPALA M.: Data structures for ray tracing on cell architecture. In *CESCG 2009* (Slovakia, April 2009).

[HM04] HINKENJANN A., MANNUSSF.: basho - a virtual environment framework. In *Proc. of 7th Symposium on Virtual Reality, SVR 2004* (2004).

[KS09] KALOJANOV J., SLUSALLEK P.: A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 23–28.

[LJS*08] LI B., JIN H., SHAO Z., LI Y., LIU X.: Optimized implementation of ray tracing on cell broadband engine. In *Multimedia and Ubiquitous Engineering, 2008. MUE 2008. International Conference on* (April 2008), pp. 438–443.

[MCEF08] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses* (New York, NY, USA, 2008), ACM, pp. 1–11.

[MMAM07] MANSSON E., MUNKBERG J., AKENINE-MOLLER T.: Deep coherent ray tracing. In *IEEE Symposium on Interactive Ray Tracing, 2007. RT '07.* (Sept. 2007), pp. 79–85.

[MSK07] MAXIM SHEVTSOV A. S., KAPUSTIN A.: Ray-triangle intersection algorithm for modern cpu architectures. In *GraphiCon'2007* (Moscou, 23–27 juillet 2007).

[MT97] MOELLER T., TRUMBORE B.: Fast, minimum storage ray-triangle intersection. *J. Graph. Tools 2*, 1 (1997), 21–28.

[VRm09] Vrmagic gmbh, December 2009. `http://www.vrmagic.com`.

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs -. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug. 2008), pp. 49–57.

[WK06] WÄCHTER C., KELLER E.: Instant ray tracing: The bounding interval hierarchy. In *In Rendering Techniques 2006 - Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 139–149.

[WRH09] WEIER M., ROTH T., HINKENJANN A.: Efficient strategies for acceleration structure updates in interactive ray tracing applications on the cell processor. In *ISVC '09: Proceedings of the 5th International Symposium on Advances in Visual Computing* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 987–998.