Vision, Modeling, and Visualization (2012)
M. Goesele, T. Grosch, B. Preim, H. Theisel, and K. Toennies (Eds.)

Hybrid Sample-based Surface Rendering

F. Reichl, M.G. Chajdas, K. Bürger, and R. Westermann

Technische Universität München

Abstract

The performance of rasterization-based rendering on current GPUs strongly depends on the abilities to avoid overdraw and to prevent rendering triangles smaller than the pixel size. Otherwise, the rates at which high-resolution polygon models can be displayed are affected significantly. Instead of trying to build these abilities into the rasterization-based rendering pipeline, we propose an alternative rendering pipeline implementation that uses rasterization and ray-casting in every frame simultaneously to determine eye-ray intersections. To make ray-casting competitive with rasterization, we introduce a memory-efficient sample-based data structure which gives rise to an efficient ray traversal procedure. In combination with a regular model subdivision, the most optimal rendering technique can be selected at run-time for each part. For very large triangle meshes our method can outperform pure rasterization and requires a considerably smaller memory budget on the GPU. Since the proposed data structure can be constructed from any renderable surface representation, it can also be used to efficiently render isosurfaces in scalar volume fields. The compactness of the data structure allows rendering from GPU memory when alternative techniques already require exhaustive paging.

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

1. Introduction

For high-resolution polygon models, much of the available polygon throughput on recent GPUs is often wasted: When a geometric screen space error below pixel size has to be guaranteed, it cannot be avoided that single pixels are covered by multiple triangles. Furthermore, since occlusion queries achieve their full potential at a rather coarse granularity, a high pixel overdraw is often introduced. Consequently, when high resolution triangle meshes are rendered on large viewports, the amount of triangles to be rendered can quickly exceed the GPU's memory and throughput capacities.

A possible option to overcome these limitations is to seek a graphics pipeline abstraction that does not built upon the projection of polygons into the pixel raster, but determines in front-to-back order the fragments that are seen through a pixel. One such abstraction is ray-tracing, which has become an alternative to rasterization due to advancements in algorithms and graphics hardware technology [GPSS07, AL09, PBD*10]. However, rasterization is still faster than ray-tracing for the computation of eye-rays, due to the initial cost per ray for traversal and ray-triangle intersection.

A different option is to employ sample-based surface representations in combination with regular sampling structures that can be ray-cast efficiently on the GPU [CNLE09,LK10]. This option is particular charming because it provides both the abilities to perform early-ray termination and adaptive LoD selection. Thus, it can effectively minimize the numbers of samples to be accessed for a particular view. Recent findings in the context of terrain rendering have even shown advantages of ray-casting over rasterization [DKW], even when a geometric LoD structure can be employed to effectively reduce the number of rendered triangles.

Our contribution: In this work we propose a hybrid GPU pipeline for computing eye-ray intersections with arbitrary models. It performs GPU rasterization and ray-casting simultaneously, deciding at run-time which technique to use for each part of the model.

To efficiently perform ray-casting, we introduce a novel sample-based surface representation. It can be created efficiently at multiple resolutions and provides an effective mechanism to reduce the number of evaluated surface samples. Compared to a triangle mesh, the proposed representation has a significantly lower memory consumption. Thus, it

DOI: 10.2312/PE/VMV/VMV12/047-054

Figure 1: *A 1 billion triangle model is rendered on a 1920×1080 viewport using rasterization and ray-casting simultaneously. Ray-casting works on a sample-based surface LoD representation at an effective maximum sampling resolution of* $8K^2 \times 32K$. *Red and green surface areas indicate parts of the model that are rendered via rasterization and ray-casting, respectively. On a GTX 680 graphics card the hybrid approach always renders the model in less than 30 ms (> 33 fps) at a screen space error below pixel size.*

is less sensitive to bus bandwidth limitations. The representation is built on a regular 2D sampling structure, on which parallel ray traversal can be performed efficiently in front-to-back order.

We demonstrate that the proposed graphics pipeline can be implemented efficiently on recent GPUs, and that significant performance gains can be achieved for high-resolution polygon models. To the best of our knowledge, for the first time we can show that a rendering pipeline based on ray-casting can be faster than rasterization for eye-ray intersections of arbitrary polygon models. Since the sample-based representation can be constructed from any available surface representation, it can also be used for rendering isosurfaces in large volume data sets. To enable interactive selection of different isosurfaces, we have implemented the construction of the sample-based representation in CUDA on the GPU. We show that for data sets as large as 4096^3, even construction of all data displayed in a typical view requires less than 200 ms from scratch. Compared to direct volume rendering, the memory requirement at run-time is reduced of a factor of up to 10.

2. Related work

Recently, advances in hardware and software technology have shown the potential of ray-tracing as an alternative to rasterization, especially for high-resolution models with many inherent occlusions. Developments in this field include advanced space partitioning and traversal schemes [WIK*06, WMS06], and optimized GPU implementations [AL09, PBD*10], to name just the most recent. Rasterization and ray-tracing have been employed consecutively to generate primary-ray intersections and secondary effects, respectively [LBIM05, OLG*07]. All these approaches can be classified as "conventional ray-tracing approaches", since they operate on the polygon object representation and perform classical ray-polygon intersection tests.

Especially for the rendering of very large models, hybrid approaches combining techniques such as geometry LoDs, pre-computed imposters, and point rendering have been proposed [ACW*99, CN01, CAZ01, GBBK04]. "Far Voxels" have been introduced as an efficient LOD structure for polygon models by Gobbetti and Marton [GM05]. They approximate polygon clusters by pre-computed voxel primitives and switch to order-independent volume splatting at coarser resolution levels. Thus, early-ray termination cannot be exploited and a significant rasterization overhead is introduced by the rendering of pixel-sized splat primitives.

Recently, "Gigavoxels" were introduced [CNLE09] for the rendering of very large polygon models. "Gigavoxels" build upon octree textures, which were first introduced by Benson and Davis [BD02] and DeBry et al. [DGPR02], and later realized on the GPU by Lefebvre et al. [LHN05] and Lefohn et al. [LSK*06]. [LK10] Laine et al., extending on this work, showed that octrees containing single voxels at the leaf nodes can also be efficiently built. Both methods also resample the polygonal surfaces onto a discrete grid to employ ray-casting, but they require a considerable amount of CPU and GPU memory. Thus, for very high-resolution polygon meshes and large viewports, their application is limited.

Lischinski and Rappoport [LR98] have introduced the Layered Depth Cube (LDC), which samples a models from three mutually orthogonal directions onto regular grids. Building on this concept, Bürger et al. [BHKW07, BKW09] proposed GPU methods for ray-tracing secondary effects and for surface painting. Dick et al. [DKW] have shown

that terrain rendering on a regular sampling grid structure can even be faster than pure rasterization for high resolution models. Novak and Dachsbacher [ND12] extended on this work and proposed splitting the model into parts which can be represented as height fields, so that GPU height field ray-casting can be employed. For high-resolution polygon models, however, finding the local piecewise height field parameterizations requires an extensive preprocess. Furthermore, it is not guaranteed that any such height field at a reasonable size exists for a desired maximum geometrical error. The effectiveness of the approach strongly depends on the surface geometry—for those reasons, their work is focused on secondary effects where local geometrical errors have a less noticeable impact.

3. Data Representation

Our method requires a two-step preprocessing of a triangle mesh which we will describe in this section. Similar to Novak and Dachsbacher, we start with a spatial subdivision followed by a resampling of each surface part into a 2D structure. However, we use a regular space partitioning which can be constructed efficiently. In this way, we also support non-static data such as isosurfaces in scalar fields, where rebuilding an adaptive acceleration structure would be too costly during runtime. In general, the regular subdivision leads to surface parts which cannot be represented by a single height field any more. We will show how our structure resolves this problem and still allows for very efficient ray-casting. Furthermore, our representation allows the creation of several levels of detail with guaranteed geometric error to further speed up the rendering for larger viewing distances.

3.1. Model Partition

Given a triangle mesh, its bounding box is partitioned into a set of bricks using a regular grid. For each brick containing a part of the surface, a 2D sampling grid as described in the next section is constructed. The resolution of these grids depends on the size of the finest geometric details, i.e., the size of the smallest triangle in the entire mesh. In our current implementation, the size of a sampling cell in these grids is equal to half of this smallest triangle size. For instance, for the David model in Fig. 1 this corresponds to an effective total resolution of $8K^2 \times 32K$ samples. If a brick contains only triangles which are at least two times as large as this cell size, though the sampling structure is created to facilitate the construction of a LoD hierarchy, it is deleted upon finishing this process. We will call such bricks *unresolved*. The rational behind this is that a sampling grid is rendered if its cell size is approximately the pixel size. Thus, when triangles are larger than this size, they would cover many pixels and rasterization would be preferred.

An additional criterion takes into account the fill rate of a sampling structure. For a constructed sample-based representation, the fill rate measures the fraction of effectively used entries in this representation. If this fraction is too low, which indicates that a huge number of entries are wasted, the respective grid is also classified as unresolved. Figure 2 shows a partitioning that has been generated using these rules.

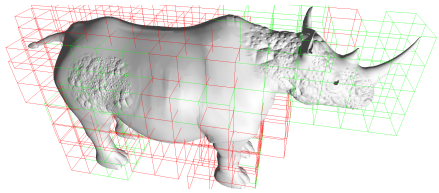

Figure 2: *Subdivision (finest level) of a triangle model into bricks using the refinement rules given in this paper. Green boxes indicate bricks for which a 2D sampling structure has been built. Red boxes indicate unresolved brick which are rendered via rasterization.*

3.2. Sample-based Data Structure

Each brick stores the triangles contained in it, clipped at the brick boundaries. From those triangle lists, our sample-based data structure is built in the second step of the preprocess. It builds upon the orthogonal fragment buffer (OFB) proposed in [BKW09]. An OFB resamples a surface along three orthogonal directions and generates for each direction a set of depth layers, each stored in a 2D sampling grid (see fig. 3).

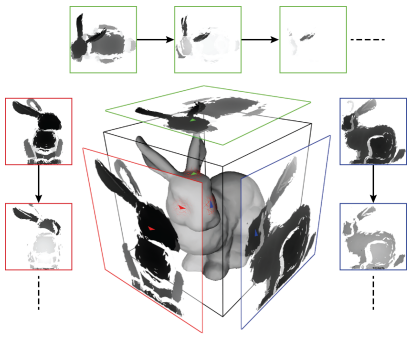

Figure 3: *An OFB is created by depth-peeling a mesh from three orthogonal directions; the distance of the surface from the respective sampling grid are stored in several layers of 2D sampling grids.*

The OFB for each brick is constructed by depth-peeling the surface from three orthogonal directions at a fixed resolution. Each depth layer stores the distance of the surface from the respective sampling grid, which we will refer to as the *sample value*. The depth values are quantized to the grid resolution to avoid holes in the sample-based surface representation, so the final OFB basically contains a conservative voxelization of the polygon model. In addition, surface attributes such as normals and colors are rendered directly into each layer.

This original OFB format, however, has severe limitations for large polygon meshes: First, for parts of the model with high depth complexity but low fill rate in each depth stack, it introduces a substantial memory overhead. Second, ray-casting this structure requires to search all depth layers to find the first sample that is hit by a view ray. For high depth complexities and high resolution sampling grids, this introduces a severe performance bottleneck.

To address these shortcomings, we only store a single 2D sampling grid of the same resolution, the *primary grid*, for each brick. The sampling direction closest to the average normal of all triangles in a brick is selected as primary direction. All sample values and attributes of all OFB layers are stored at the respective position in the primary grid; conceptually, the OFB layers are treated as a 3D grid for this step (see fig. 4, left). When the same sample is contained in multiple directions, its attributes are averaged before storage. The original OFB is discarded afterwards as it is no longer required.

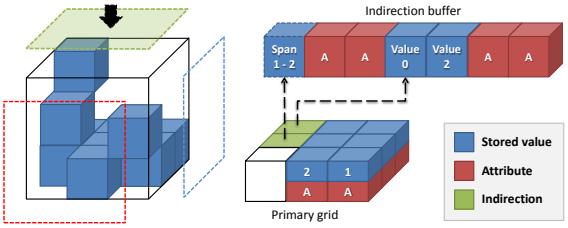

Figure 4: *Construction of our sample-based representation from an existing OFB: The sample values stored in the OFB layers (left) are stored into a single 2D structure, the primary grid (right). In this example, the top-down view is selected as primary direction. If only a single sample exists for a grid position, its distance from the primary grid is stored directly along with its attributes. Otherwise, a pointer into the indirection buffer containing the sample values is stored. Successive voxels are stored as a single compact RLE span.*

For each OFB direction, multiple layers can be present. This requires the storage of a variable number of values and attributes for each sampling position in the primary grid. For this reason we employ a two-level data structure consisting of the 2D primary grid and a 1D *indirection buffer*. If a grid position contains only a single sample value, it is stored directly in the primary grid along with all its attributes, much alike a traditional height field. In any other case, all sample values for a grid position are written to the indirection buffer in ascending order, followed by the attributes of all samples. A pointer to the first written value as well as the number of values is stored at the respective position of the primary grid instead (see fig. 4, right).

To further reduce the storage overhead, a runlength encoding of sample values is performed during construction: For each span of successive values to be written into the indirection buffer, only the first value is stored along with the total length of the span.

For a sufficiently high resolution of the initial 3D space partitioning, the 2D sampling structures as described will only contain very few samples in typical cases as will be shown in section 6. Even if this is not the case, we will demonstrate that the compact span representations can be tested very efficiently to find possible intersections between a ray and the surface.

3.3. Level of Detail

Once the initial resolution level has been processed, a 2D sampling grid and indirection buffer has been created for every brick containing a surface part. To create a sample-based model representation at a coarser resolution, we first generate for every 2^3 adjacent bricks one new brick. For these bricks, a 2D sampling grid at half the initial resolution and an indirection buffer are created. However, to determine the samples along the primary directions, instead of rendering the surface model we examine the already existing samples at the finer level. A sample is generated if at least one sample in the merged cells at the finer level exists. In this case, the attributes are averaged. This process is repeated recursively until a user-defined level is reached.

Differing from the described process is the handling of unresolved bricks. If a brick is classified as unresolved because a) its 2^3 child bricks are all unresolved and b) the minimum size of the triangles in the child bricks exceeds twice the brick's cell size, the surface samples for this brick are generated via rendering the triangles. An unresolved brick stores the references to the triangle lists of its children. After the construction of the sample-based LoD surface hierarchy, which corresponds to a sparse octree refined along the surface, the sampling structures of all unresolved bricks are deleted. Thus, for surface parts which are modeled by large triangles no sample-based representation is built, and these regions will always be rendered via rasterization up to a certain coarse level.

4. Hybrid Rendering

In every frame, the octree that was computed in the preprocess is first traversed on the CPU, and the bricks which have to be *rasterized* are determined. Here, a brick is only considered if it satisfies a user-defined screen space error, usually one pixel or below, or if the finest octree level is reached. Unresolved bricks are always rasterized. Bricks for which both the triangles and the sample-based data structure are stored are rendered via rasterization, if a) the view sampling frequency is higher than the brick resolution or b) a *render oracle* determines that rasterization will be faster than ray-casting. The oracle is evaluated on the CPU for every brick

that is selected for rendering and determines the most efficient rendering technique for that brick. We use the oracle proposed in [DKW] for hybrid terrain rendering. To evaluate the cost of rasterization it considers both the number of triangles in a brick as well as the area they subtend in the view plane. The ray-cast oracle is based on a measure of how steep the ray from the eye position descends on the sampling grid that is to be cast. It thus estimates the length of the ray's projection into this grid.

The bricks selected for rasterization are rendered in front to back order, with z-buffering and occlusion queries enabled. All bricks which are not rendered via rasterization are ray-cast. The ray-caster uses a single draw call to traverse the octree completely on the GPU. It uses a data structure similar [CNLE09]—in a dedicated buffer, for each brick residing in GPU memory there exists an entry containing pointers to the brick's sampling plane as well as a base offset in the indirection buffer. Also stored is a single pointer to all 8 of the brick's children as well as a flag indicating the render oracle decision for this brick. This buffer is updated by the CPU each frame and traversed by a stackless octree raycaster. Early ray termination is implemented by testing the depth values generated by the rasterizer before entering a brick. For every brick that is flagged as "ray-cast", the sample-based data structure is rendered as described next.

For larger datasets, coarser LoDs are rendered for bricks not yet residing in CPU memory. These bricks are then loaded asynchronously to hide disc latencies, and uploaded to the GPU as soon as they become available. As long as memory is available, all data is cached in GPU and CPU memory in a least-recently-used manner.

4.1. Sample-based Ray-Casting

Ray-casting a brick is performed using an extension of standard height field ray-casting: For each brick a ray is passing through, the ray is projected into the 2D sampling plane and DDA-like ray marching is performed in front-to-back order to find the grid cells hit by the ray. For every cell a flag-bit indicates whether a single sample or a pointer into the indirection buffer is stored. In the first case, a single height value needs to be tested for intersection; in the later case, the ray is tested for intersections with the sample spans stored at each cell, i.e., whether it intersects one of the 3D cells associated with the stored samples. Intersections with spans consisting of multiple samples are handled in a single test. Once an intersection is found, the surface attributes can be read from the data buffer using the index of the sample that was intersected. If no intersection is found, ray marching continues with the next cell until the end of the sampling grid is reached.

To further accelerate the ray traversal process, we employ 2D maximum/minimum mipmaps [OKL06, TIS08]. A mipmap representation can effectively reduce the number of ray-casting steps until the first intersection, and in our particular scenario it can be used to discard those cells where an intersection cannot occur. For each brick, such a min/max hierarchy is computed in the preprocess. These hierarchies are then used in the traversal process to skip regions that are completely below or above the ray.

5. Isosurface Ray-Casting

The proposed sample-based representation can also be constructed very efficiently from a scalar volume field. To enable interactive rendering of isosurfaces in Cartesian grid data sets, we have implemented the construction process on the GPU using the CUDA API to allow for interactive isovalue changes. We perform two sweeps over the volume from three orthogonal sampling directions with sampling planes of resolution equal to the volume resolution; each sampling cell is associated with a single CUDA thread. In he first sweep, the number of isosurface intersections per cell is counted and stored in the sampling plane. In the second sweep, for each intersection the gradient normals are calculated and written into a linear buffer along with the intersection position. To assign each thread a starting write offset in this buffer, a parallel prefix sum operation of the intersection counts is performed.

To merge the sampling directions into the primary direction, two passes following the same principle are applied—we first count the number of intersections using the virtual 3D grid given by the three existing sampling planes. A prefix sum is applied again to calculate the final indirection buffer offsets, and in a second pass, the merged surface attributes are written to the indirection buffer. If only a single intersection was found, these values are directly written to the sampling plane.

The CUDA construction process is embedded into a GPU-based out-of-core volume rendering framework, including a pre-computed LoD hierarchy and a corresponding octree containing per-brick min/max values to avoid processing bricks which do not contain a surface part. The framework is built conceptually similar to the ones described in [CNSE10, IGGM, GMIG08].

At runtime, the min/max octree is traversed in a single pass on the GPU to determine the bricks in the LoD hierarchy which have to be rendered for the current view. For each selected brick for which a sample-based representation is not yet resident on the GPU, the volume data is asynchronously loaded from the CPU—or disc if necessary—and the sample-based representation is constructed using the CUDA kernel. Bus transfer is minimized by using separate LRU caches for volume- and sample-based data, whereas only a small part of the available memory is assigned to the volume data caches. If a brick should be rendered at a higher sampling rate than the brick resolution, we fall back to classical isosurface ray-casting using tri-linear interpolation.

6. Results

In this section, we analyse the performance and memory consumption of the hybrid rendering pipeline in comparison to pure rasterization and ray-casting of our sample-based representation on the GPU. All timings were measured on a standard desktop PC, equipped with an Intel Core 2 Quad Q9450 2.66 GHz processor, 8 GB of RAM, and an NVIDIA GeForce GTX 680 graphics card with 2048 MB of local video memory. Rendering was always to a 1920×1080 viewport unless mentioned otherwise. Models with an inverse aspect ratio were rendered in landscape mode. The screen space error-tolerance was set to one pixel. In all polygonal test scenes, per-vertex attributes like colors and normals were resampled to the sample-based data structures.

As can be seen from all presented results, transitions from rasterized to ray-cast bricks are seamless in all cases and no noticeable quality differences or cracks exist between the two rendering methods. Since ray-casting does not provide a cost-efficient method for anti-aliasing, techniques such as MSAA need to also be disabled for the rasterizer. To compensate this, we apply SMAA [JESG12] as a post-process in our current implementation, which lead to overall good result.

6.1. Polygon Meshes

Table 1 provides information concerning the used polygonal models. For the views in Fig. 5 and Fig. 1, Table 2 compares the performance of pure rasterization, pure sample-based ray-casting, and the hybrid approach. Numbers denoted with a * are estimates, because not all visible data could be stored in GPU memory at the required resolution. In this case, the rendering times would be vastly dominated by CPU/GPU data transfer.

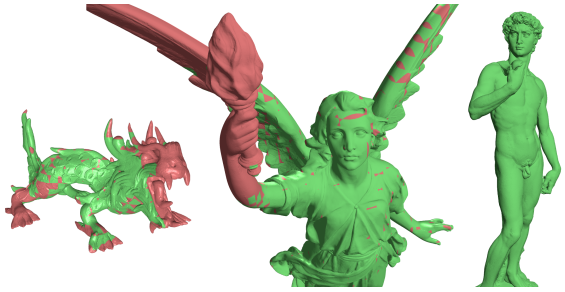

Figure 5: *Dragon, Lucy, and David (view 1) are rendered via rasterization (red) and ray-casting (green).*

Especially for the David model, the hybrid pipeline shows a significant speed-up over pure rasterization. This is in particular due to the use of the sample-based LoD hierarchy to reduce the required memory per frame. Regardless of the view, the David model can always be rendered in less than 30 ms. On the other hand, even for the smaller models which fit entirely into GPU memory, and where large parts

Model	Tris	Res	Mem
Dragon	9 M	$2K^3$	0.15 / 0.12
Lucy	28 M	$8K^3$	0.74 / 0.57
David	950 M	$8K^2 \times 32K$	24.44 / 18.80

Table 1: *Model statistics: number of triangles, effective sampling resolution, total file size (triangle data, sample-based representation, LoD), and size of triangle data only (in GB).*

Views	t_{rast}	t_{ray}	t_{hybrid}	Mem_{rast}	Mem_{ray}
Dragon	9.8	11.7	7.6	145	11
Lucy	25	29.8	21.8	477	58
David 1	900*	9.7	9.7	13083	15
David 2	190*	23.2	21.4	3810	65
David 3	79.4	34.1	27.5	1840	91

Table 2: *Rendering times in ms and GPU memory requirements in MB for a single frame as depicted in Figs. 5 and 1 (views from Fig. 1 are labeled David 1, 2 and 3 from left to right). Numbers denoted with a * are estimates without CPU/GPU memory transfer.*

are rasterized, a noticeable performance gain is achieved. This demonstrates the efficiency of sample-based GPU ray-casting and emphasizes the importance of avoiding pixel overdraw via early ray-termination. This is confirmed by Fig. 6, where a sequence of views around the Lucy model was performed. The hybrid approach is always at least on par with pure rasterization and ray-casting, and usually outperforms both. The memory requirement is reduced significantly. For comparison, the sparse voxel octree representation [LK10] of the Dragon model requires 156 MB of GPU memory, whereas our representation uses about 30 MB.

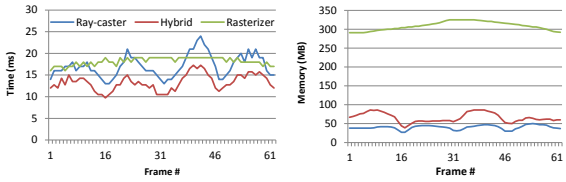

Figure 6: *Recorded flight around Lucy. Render times and memory consumption are analysed.*

Fig. 7 illustrates why ray-casting can be performed on the sample-based data structure very efficiently. It shows the number of surface samples stored at the cell of the 2D sampling grid where the surface intersection is actually found. Interestingly, in most cases only one single sample is stored, meaning that locally the surface is a height field over the 2D sampling domain. This confirms our expectation, that at a reasonable subdivision depth the surface parts in each brick tend to become height fields, even though the domain was chosen to be aligned with one of the brick faces.

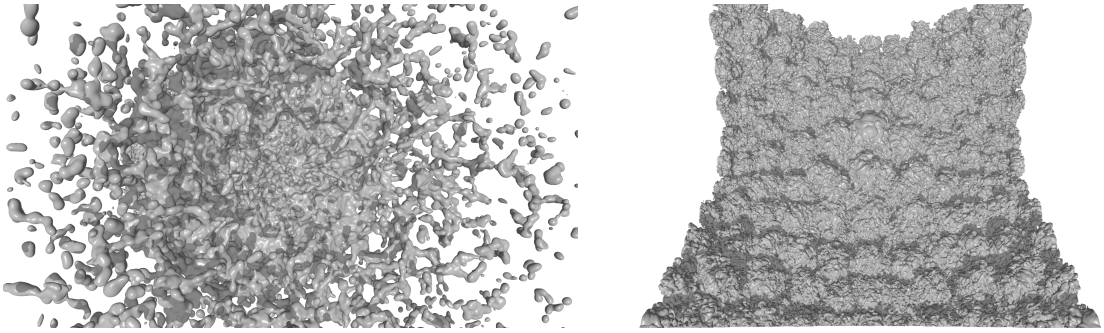

Figure 8: *For the Ejecta (4096^3) and Richtmyer ($2048^2 \times 1920$) data sets, isosurface rendering onto a 1920×1080 viewport takes always less than 25 ms.*

6.2. Isosurface Ray-Casting

The following volume data sets were used: Ejecta is a simulation of the impact of a supernova ejecta on a companion star. It has a resolution of 4096^3, stores 2 bytes per voxel, and consumes 128 GB for the finest level of detail. Richtmyer is a simulation of a Richtmyer-Meshkov instability. It has a resolution of $2048^2 \times 1920$, stores 1 byte per voxel, and consumes 7.5 GB. Timing and memory statistics for Direct Volume Rendering (DVR) and Sample-Based Rendering (SBR) related to the views in Fig. 8 are given in Table 3. The viewport size was set to 1920×1080. For a fair DVR comparison, we simply disabled the SBR extension of our volume rendering system, visualizing each brick using classical isosurface ray-casting. Otherwise, the exact same system was used, including a LoD hierarchy on the volume data.

In all cases, we chose a brick size of 32^3. While smaller bricks can decrease the memory requirements for a particular isosurface, a spatially coherent organization of bricks on the GPU becomes more difficult and GPU cache mechanisms work less effective. In our experiments, and confirmed by state-of-the-art volume rendering systems [CNSE10, IGGM], the selected brick size has shown the best tradeoff.

It can be seen that the sample-based isosurface representation has a significantly lower memory footprint on the GPU than DVR. In DVR, especially for large viewports even the data required for rendering a single frame might exceed the available GPU memory. Due to the compactness of our representation, the entire data needed to render the isosurface in the Richtmyer data set fits onto the GPU. One can also see that the sample-based representation can be built at rates vastly exceeding CPU/GPU bus bandwidth. Thus, even if the isosurface is changed, noticeable losses in performance are not introduced.

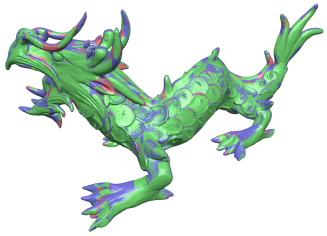

Figure 7: *Color coding of number of samples that are stored at the grid cell where the surface intersection is found. Green indicates one sample, blue indicates more than one sample in one span, red indicates more than one span.*

7. Conclusion and Future Work

We have presented a hybrid GPU pipeline for the rendering of polygon models with high geometric complexity. The pipeline does not replace but evolves the current graphics pipeline abstraction in that it uses rasterization and ray-casting simultaneously to generate eye-ray intersections. We have demonstrated that for very large triangle meshes at extreme resolution the hybrid pipeline can overcome performance limitations of pure rasterization. This has been achieved by introducing a sample-based surface representation which can be compactly encoded and traversed efficiently by many rays in parallel on the GPU. Due to the

	DVR		SBR		
Data set	Mem	t_r	Mem	t_r	Build (ms)
Ejecta	2.98	412	0.30	23	196
Richtmyer	1.02	29	0.22	21	134
Finest LoD	3.24	819	0.73	67	324

Table 3: *Memory and timing statistics for direct (DVR) and sample-based (SBR) volume rendering. Mem: memory consumption (in GB) for a single frame. t_r: rendering times (in ms, including data upload to GPU if required). Build: time required to convert all visible surface parts (with LoD error below one pixel) for the depicted images. Last row: statistics for the selected isosurface at the finest LoD of the entire Richtmyer data set.*

ability to efficiently compute a LoD hierarchy on this representation, GPU resources can be employed effectively at run-time. We see our work as a step towards next generation rendering architectures that scale with respect to the overall GPU throughput and can thus satisfy the future demands of real-time graphics.

In the future, we will in particular investigate the parallelization of sample-based ray-casting on multi-core architectures. Multi-core architectures provide parallelism only to a modest degree, but they are highly optimized for minimizing latency in a single sequential task. Since sample-based ray-casting often suffers from latency issues caused by adjacent rays following different patterns through the sampling grids, it can be a good candidate for multi-core parallelism.

Acknowledgements: We would like to thank the Digital Michelangelo Project at Stanford for providing the David statue.

References

[ACW*99] ALIAGA D., COHEN J., WILSON A., BAKER E., ZHANG H., ERIKSON C., HOFF K., HUDSON T., STUERZLINGER W., BASTOS R., WHITTON M., BROOKS F., MANOCHA D.: MMR: an interactive massive model rendering system using geometric and image-based acceleration. In *Proceedings of I3D '99* (1999), pp. 199–206. 2

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on gpus. In *Proc. of HPG '09* (2009), pp. 145–149. 1, 2

[BD02] BENSON D., DAVIS J.: Octree textures. In *Proceedings of SIGGRAPH '02* (2002), pp. 785–790. 2

[BHKW07] BÜRGER K., HERTEL S., KRÜGER J., WESTERMANN R.: GPU Rendering of Secondary Effects. In *Proceedings of VMV '07* (2007), pp. 51–60. 2

[BKW09] BÜRGER K., KRÜGER J., WESTERMANN R.: Sample-based surface coloring. *IEEE TVCG 16*, 5 (2009). 2, 3

[CAZ01] COHEN J. D., ALIAGA D. G., ZHANG W.: Hybrid simplification: combining multi-resolution polygon and point rendering. In *Proceedings of VIS '01* (2001), pp. 37–44. 2

[CN01] CHEN B., NGUYEN M. X.: POP: A hybrid point and polygon rendering system for large data. *Proceedings of VIS '01* (2001), 45–52. 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of I3D '09* (2009), pp. 15–22. 1, 2, 5

[CNSE10] CRASSIN C., NEYRET F., SAINZ M., EISEMANN E.: Efficient rendering of highly detailed volumetric scenes with gigavoxels. in book: Gpu pro, 2010. 5, 7

[DGPR02] DEBRY D. G., GIBBS J., PETTY D. D., ROBINS N.: Painting and rendering textures on unparameterized models. In *Proceedings of SIGGRAPH '02* (2002), pp. 763–768. 2

[DKW] DICK C., KRÜGER J., WESTERMANN R.: GPU ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers*. 1, 2, 5

[GBBK04] GUTHE M., BORODIN P., BALÁZS Á., KLEIN R.: Real-time appearance preserving out-of-core rendering with shadows. In *Proceedings of EGSR '04* (jun 2004), pp. 69–79 + 409. 2

[GM05] GOBBETTI E., MARTON F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Transactions on Graphics 24*, 3 (2005), 878–885. 2

[GMIG08] GOBBETTI E., MARTON F., IGLESIAS GUITIÁN J. A.: A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24* (July 2008), 797–806. 5

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of Symposium on Interactive Ray Tracing '07* (Sept. 2007), pp. 113–118. 1

[IGGM] IGLESIAS GUITIÁN J., GOBBETTI E., MARTON F.: View-dependent exploration of massive volumetric models on large-scale light field displays. *The Visual Computer 26*, 1037–1047. 5, 7

[JESG12] JIMENEZ J., ECHEVARRIA J. I., SOUSA T., GUTIERREZ D.: Smaa: Enhanced morphological antialiasing. *Proceedings of Eurographics '12* (2012). 6

[LBIM05] LÁSZLÓ S.-K., BARNABÁS A., ISTVÁN L., MÁTYÁS P.: Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum 24*, 3 (2005). 2

[LHN05] LEFEBVRE S., HORNUS S., NEYRET F.: *GPUGems 2*. Addison-Wesley, 2005, ch. Octree Textures on the GPU, pp. 595–613. 2

[LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees. In *Proceedings of I3D '10* (2010), pp. 55–63. 1, 2, 6

[LR98] LISCHINSKI D., RAPPOPORT A.: Image-Based Rendering for Non-Diffuse Synthetic Scenes. In *Proceedings of EGSR '98* (1998), pp. 301–314. 2

[LSK*06] LEFOHN A. E., SENGUPTA S., KNISS J., STRZODKA R., OWENS J. D.: Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics 25*, 1 (2006), 60–99. 2

[ND12] NOVÁK J., DACHSBACHER C.: Rasterized bounding volume hierarchies. *Computer Graphics Forum (Proc. of Eurographics) 31*, 2 (2012), 403–412. 3

[OKL06] OH K., KI H., LEE C.-H.: Pyramidal displacement mapping: a gpu based artifacts-free ray tracing through an image pyramid. In *Proceedings of VRST '06* (2006), pp. 75–82. 5

[OLG*07] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113. 2

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics 29*, 4 (2010), 66:1–66:13. 1, 2

[TIS08] TEVS A., IHRKE I., SEIDEL H.-P.: Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Proceedings of I3D '08* (2008), pp. 183–190. 5

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics* (2006), 485–493. 2

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for hardware accelerated ray tracing of dynamic scenes. In *In proceedings of GH '06* (2006), pp. 67–77. 2