DOI: 10.1111/cgf.14670
Pacific Graphics 2022
N. Umetani, E. Vouga, and C. Wojtan
(Guest Editors)

Volume 41 (*2022*), *Number 7*

Efficient Direct Isosurface Rasterization of Scalar Volumes

A. Kreskowski, G. Rendle and B. Froehlich

Virtual Reality and Visualization Research Group, Bauhaus-Universität Weimar, Germany

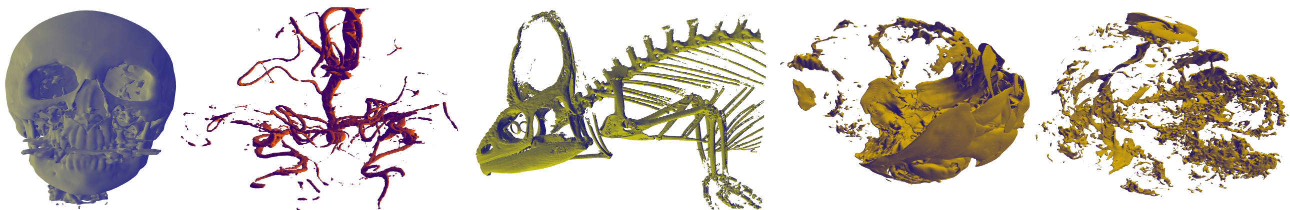


Figure 1: *Comparison of average draw times in milliseconds between our direct isosurface rasterization approach (ours) and min-max octree ray marching (oct), both at a rendering resolution of* 3840×2160 *pixels.*
CT Head (0.19_{ours} | 1.54_{oct}) *Vertebra* (0.26_{ours} | 1.95_{oct}) *Chameleon* (1.26_{ours} | 2.59_{oct}) *Timeseries Supernova* (0.29_{ours} | 1.41_{oct})

Abstract

In this paper we propose a novel and efficient rasterization-based approach for direct rendering of isosurfaces. Our method exploits the capabilities of task and mesh shader pipelines to identify subvolumes containing potentially visible isosurface geometry, and to efficiently extract primitives which are consumed on the fly by the rasterizer. As a result, our approach requires little preprocessing and negligible additional memory. Direct isosurface rasterization is competitive in terms of rendering performance when compared with ray-marching-based approaches, and significantly outperforms them for increasing resolution in most situations. Since our approach is entirely rasterization based, it affords straightforward integration into existing rendering pipelines, while allowing the use of modern graphics hardware features, such as multi-view stereo for efficient rendering of stereoscopic image pairs for geometry-bound applications. Direct isosurface rasterization is suitable for applications where isosurface geometry is highly variable, such as interactive analysis scenarios for static and dynamic data sets that require frequent isovalue adjustment.

CCS Concepts

• ***Computing methodologies*** → *Computer Graphics;*

1. Introduction

Isosurfaces are traditionally visualized using either *direct* or *indirect* rendering approaches. Direct isosurface visualization approaches are normally based on ray marching, and are referred to as 'direct' because no intermediate representation of the isosurface is created [EHK*06,PSL*98]. Conversely, indirect isosurface visualization techniques most commonly use variants of the marching-cubes technique [LC87] to extract an explicit isosurface representation from the volume dataset, which is subsequently rendered, often with a standard triangle mesh rendering pass. This two-step process requires significant memory bandwidth as well as additional memory for storing and accessing the isosurface representation.

In this paper, we present *direct isosurface rasterization*, an approach that avoids this memory and bandwidth overhead by leveraging the capabilities of task and mesh shading pipelines, a modern graphics feature, to produce indexed triangles as graphics primitives on the fly and have them directly consumed by the rasterization hardware. We refer to this process of extracting and directly consuming the geometry as *transient isosurface extraction*. We use task shaders to identify isosurface-containing cells inside small volume blocks, and pass sets of these cells on to mesh shaders, which extract the isosurface primitives and pass them on to the rasterization units. To complement the extraction and rendering stage, we introduce a raster-occlusion-culling-based technique that uses a similar approach, employing task shaders to identify volume blocks that contain isosurface geometry, and using mesh shaders to efficiently generate proxy geometry that is then rasterized to identify the potentially visible set of blocks.

© 2023 The Authors. Computer Graphics Forum published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

Even though ray-marching-based approaches have become a de facto standard for applications where isovalues need to be continuously adjusted, their performance generally has a linear dependence on the display resolution, which is ever increasing as display technologies advance. In contrast, rasterization units in modern GPUs are now so highly optimized that the cost of rasterizing primitives is much less dependent on resolution. Our observation is that both direct and indirect approaches need to access the same volume cells in order to render the visible part of an isosurface. If those cells can be identified and processed quickly, and primitives directly streamed into the rasterizer, the cost of isosurface extraction might be compensated by fast rasterization. It is our hypothesis that isosurface extraction and rasterization should be faster when cells generate multiple fragments (cover multiple pixels), because in these cases multiple rays would have to compute the isosurface intersections in the same cell. Therefore, we believe that direct isosurface rasterization should outperform ray casting with increasing display resolution for a given volume.

To validate our hypothesis, we effectively employ task and mesh shading pipelines to create an efficient direct isosurface visualization purely based on rasterization. Our contributions can be summarized as follows:

- a novel and efficient pipeline design for direct isosurface rasterization that requires negligible preprocessing and only a small amount of additional memory for auxiliary data structures;
- a lightweight two-level raster occlusion culling approach complementing our isosurface rasterization approach.

Our evaluation shows that direct isosurface rasterization outperforms ray-marching approaches using min-max octrees in most situations.

2. Related Work

Our work is mainly influenced by research in the fields of interactive indirect isosurface rendering, direct isosurface rendering, and GPU-based occlusion culling. In the following section, we provide an overview of work that is closely related to ours, and highlight the features that distinguish our approach from previous work.

2.1. Interactive Indirect Isosurface Rendering

One of the first approaches that addresses indirect isosurface rendering is the *continuation method* for isosurface polygonization from Wyvill et al. [WCB86]. The approach is based on discretization of 3D space into volumetric cells, followed by efficient identification of cells that are intersected by the isosurface. The subsequently introduced *Marching Cubes* (MC) algorithm [LC87] laid the foundation for many further developments in the area of real-time isosurface extraction from volumetric data. The algorithm's appeal lies in a simple approach that generates geometry from each cubic cell intersected by the isosurface, by first determining which cell corners lie inside the isosurface, and then generating vertices and triangles representing the isosurface based on pre-computed lookup tables. The algorithm lends itself well to implementation on massively parallel hardware, and is therefore at the core of our GPU-based isosurface extraction stage. For a comprehensive review of MC algorithms up to the year 2006, we refer the reader to the survey by Newman and Yi [NY06], and address only GPU-based MC variations in this literature review.

The first widely available GPU-based implementation of the MC algorithm was presented by NVIDIA [NVI22] as part of the CUDA sample code base provided with the *nvsdk*. The authors identified the necessity for two preprocessing passes: a stream compaction pass that determines per-cell occupancy, and a subsequent stream expansion pass to determine the number of triangles created per occupied cell. Although the preprocessing passes achieve efficient extraction by avoiding excessive synchronization between extraction threads, they are the most expensive processing stage, representing an opportunity for optimization in further works. The approach does not scale well to larger volumes, since several additional GPU buffers are involved that have linear memory requirements with respect to the volume resolution.

Dyken et al. [DZTS08] developed a competitive approach to isosurface extraction by using more efficient stream compaction and expansion operations based on the *HistoPyramid* data structure introduced by Ziegler et al. [ZTTS06]. The HistoPyramid's hierarchical approach led to isosurface extraction times that were reportedly faster than those achieved with the *nvsdk* implementation [LCDW16], but the amount of memory required to create the HistoPyramids can exhaust the GPU memory similarly quickly to the *nvsdk* technique [LCDW16].

To address this issue, Liu et al. [LCDW16] introduced the *Parallel Marching Blocks* (PMB) algorithm. The approach performs in-kernel stream compaction by detecting occupancy on a voxel-block level, meaning that the GPU memory requirement for compaction buffers is significantly reduced. The in-kernel stream compaction operation is accelerated through implementation of a local prefix sum algorithm similar to that used by Hughes et al. [HLJ*13]. Furthermore, isosurface geometry is also extracted in a block-based manner, allowing for the removal of duplicate vertices within a block which effectively reduces the memory footprint of the extracted isosurface. The authors reported isosurface extraction on volumes that were 64× larger than those that could be handled by the *nvsdk* and *HistoPyramids* approaches. Usher and Pascucci [UP20] also follow a block-wise stream compaction and expansion approach to reduce the memory footprint of the isosurface extracted from a very large volume.

In addition to variants of the MC algorithm, dual polygonization approaches such as *SurfaceNets* [Gib98] or *Dual Contouring* [JLSW02] allow for higher quality isosurface representations since vertices may be positioned arbitrarily within a cell, instead of being constrained to its edges. However, those algorithms come at the expense of additional computation steps, such as the creation of distance maps [Gib98] or the necessity to solve quadratic error functions in order to find the optimal placement of the dual surface's vertices. In this paper, we opt to explore the more lightweight isosurface extraction and rendering approach based on a variant of MC, although a consideration of dual contouring approaches may be worthwhile in further research.

In contrast to prior work, our approach requires no additional memory for an explicit isosurface representation, because the extracted mesh is directly consumed by the rasterizer. Our work is influenced by the PMB algorithm, in that we determine occupancy

© 2023 The Authors.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

on a block level. We also implement a local prefix sum algorithm for stream compaction, but avoid using atomic variables, to improve extraction performance. Furthermore, in contrast to PMB, we exploit view-dependent visibility information by computing the potentially visible set of blocks in a GPU-driven occlusion culling stage [KT14, LL20, LJSL21].

2.2. Direct Isosurface Rendering

The predominant technique used for direct visualization of single or multiple isosurfaces is ray marching, also referred to as ray casting. For a comprehensive overview of isosurface ray marching and basic acceleration techniques such as empty space skipping, we refer the reader to the comprehensive introduction to real-time volume graphics by Engel et al. [EHK*06]. We also intentionally omit a discussion of work addressing out-of-core rendering of multiresolution volumes, since this is out of the scope of our current work. For an in-depth review of GPU-driven visualization of large-scale volumes, we refer the interested reader to the work of Beyer et al. [BHP15].

Empty space skipping aims to reduce the number of times that a ray samples the volume. Liu et al. [LCD09] achieve extremely fine-grained empty space skipping by only marching rays through active volume cells, which are efficiently located with a HistoPyramid texture [ZTTS06] and rendered as point primitives to create rays close to the isosurface. The authors accelerate their approach by exploiting early-depth testing: a GPU feature allowing a fragment's interpolated z-coordinate to be compared with the current depth buffer value at the corresponding location, after which the fragment can be discarded if occluded, reducing shading calculations. This feature can be better exploited by rasterizing primitives in a front-to-back order, increasing the number of fragments that are discarded, which Liu et al. achieve through a CPU-based sorting of volume slices. The authors later introduced the *IsoBAS* acceleration structure [LCD15], yielding faster identification of active cells and requiring less memory overhead. Jiang et al. [JRZ*16] also propose an acceleration structure that can efficiently locate active cells for both direct and indirect visualization approaches, again rendering active cells by ray marching. The aforementioned methods [LCD09, LCD15, JRZ*16] can be considered hybrid object-order and image-order approaches, meaning that they are dependent on rendering resolution.

Sphere Tracing [Har96] and Segment Tracing [GGPP20] are techniques that vary the distance between samples along a ray to improve performance with respect to other ray casting-based approaches. However, Sphere Tracing requires signed distance functions, which need to be derived from the original volume in a preprocessing step. In addition, Galin et al. [GGPP20] explicitly state that Segment Tracing works only for a particular class of hierarchical skeletal implicit surfaces, which means that both techniques are not directly applicable to a wide range of volumes. Galin et al. further state that their Segment Tracing approach is outperformed by octrees with a depth of 7 or more, and that the additional memory required for auxiliary octree data is as low as 1 to 4 megabytes. Regarding the memory requirements for the input volume itself, we consider the memory overhead negligible and therefore deem min-max octree ray casting to be a fitting approach for general accelerated direct isosurface rendering. We compare our approach against min-max octree ray casting in the evaluation section.

In the proposed approach, active cells are efficiently identified and processed without the help of a large hierarchical auxiliary structure, by leveraging task and mesh shaders to check cells within potentially visible subvolumes, and dynamically allocate GPU resources to active cells. By extracting the isosurface as a triangle mesh instead of performing ray marching in active cells, we reduce dependence on the rendering resolution.

2.3. GPU-based Occlusion Culling of Dynamic Scenes

In order to execute isosurface extraction on only the potential visible set (PVS) of voxel blocks, our pipeline includes a GPU-driven occlusion culling stage. Historically, occlusion culling on the GPU was facilitated by occlusion query objects [CCG*07] that allowed retrieval of visibility information after rasterization of proxy objects. However, effective use of this feature was rather involved, and required careful use of spatial and temporal coherence, as first shown by Bittner et al. [BWPP04], and later by Mattausch et al. [MBW08].

Kubisch and Tavenrath [KT14] subsequently introduced *raster occlusion culling* as the state-of-the-art approach to efficient visibility determination for general scenes. Raster occlusion culling replaces expensive individual hardware occlusion queries with one efficient proxy geometry rasterization pass, followed by a GPU-driven stream compaction operation that creates a dense list of occupied subvolumes.

The approach usually outperforms those that process the depth texture of a previously established occluder representation, such as hierarchical occlusion maps [ZMHH97] and iterative depth warping [LKE18]. Recently, follow-up works have aimed to avoid rasterizing all leaf-level objects in the scene by applying the concept of spatial coherence exploitation through hierarchical grouping of objects [MBW08] to raster occlusion culling. These works include Lee and Lee's [LL20] proposal for an iterative version of raster occlusion culling, which works on a select number of hierarchy levels, and Lee et al.'s [LJSL21] introduction of a fully GPU-driven hierarchical raster occlusion culling algorithm, which aims to reduce the overall rasterization overhead by performing rasterization on a coarser level for group nodes, and refining the visibility results through ray traversal down to the leaf level.

In this work, we propose a lightweight raster occlusion culling pass that is a hybrid between basic raster occlusion culling as presented by Kubisch and Tavenrath [KT14] and hierarchical raster occlusion culling as presented by Lee and Lee [LL20]. We resolve potential visibility only for fixed size blocks, but use a custom task and mesh shader pipeline to efficiently detect occupied blocks, before creating proxy geometry for each one. We note that Kubisch [Kub14] presents an excellent introduction to raster occlusion culling using the of task and mesh shader pipeline.

Our culling approach is most similar to the probabilistic culling method proposed by Ibrahim et al. [IRR*22] for efficient particle rendering using pipelines based on task and mesh shaders. However, in contrast to Ibrahim et al., we perform occlusion culling on

© 2023 The Authors.
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

only one of the two levels, and ensure that our visibility determination is conservative rather than probabilistic.

3. Efficient Direct Isosurface Rasterization

In our direct isosurface visualization technique we employ the task and mesh shader pipeline to extract graphics primitives representing the isosurface 'on the fly' and have them directly consumed by the rasterization hardware. In the extraction and rendering stage, the task shader produces a compact list of isosurface-containing cells inside visible blocks using the Marching Cubes algorithm, and the mesh shader instances subsequently extract primitives from those cells using the Marching Cubes cell classification, before passing the primitives directly to the rasterization units (see Subsection 3.4). To avoid extracting isosurfaces in occluded regions, our pipeline includes a novel raster occlusion culling [KT14, LJSL21] approach to identify potentially visible blocks, detailed in Subsection 3.5. The occlusion culling stage also employs task and mesh shaders, using task shaders to identify blocks containing the isosurface, and mesh shaders to produce proxy geometry that is rasterized to determine the visibility of each block.

We hypothesize that utilizing the highly-optimized rasterization hardware to render geometric primitives will compensate for the cost of extracting the primitives, leading to better performance than approaches based on ray marching as rendering resolution increases.

Notes on Terminology. Although this description uses terminology specific to *NVIDIA*'s hardware and software stack and the *OpenGL* graphics API, our method should be applicable to similar graphics cards from other manufacturers. For instance, the method could be adapted for *AMD* hardware by replacing the term *warp* with *wavefront*, and observing that wavefronts are executed as larger workgroups with more than 32 threads. Similarly, our approach is easily mapped to other graphics APIs (e.g. *Vulkan* or *DirectX12*) by replacing the term *task shader* by the term *amplification shader*.

3.1. Task and Mesh Shaders

Task and mesh shaders [Kub18], introduced with the Turing generation of NVIDIA graphics hardware, are intended to provide graphics programmers with more flexibility than the classic geometry pipeline consisting of vertex, geometry and tessellation shaders. The pipeline functions as follows: the programmer launches a number of task shaders, which can each emit a variable number of mesh shaders. Mesh shaders can generate a variable number of geometric primitives, and pass them directly to the rasterization hardware. Data can be passed from the task shader group to its emitted mesh shader groups in a memory interface block.

Task and mesh shaders are similar to compute shaders, in that they operate in workgroups comprised of multiple threads, with each thread executing one invocation of the shader program. Like compute shaders they have no predefined input and output, however they have limited output sizes to conform with optimal memory allocation schemes. In contrast to compute shaders, the size of a workgroup is limited to 32 threads. Execution of the shader programs in workgroups allows for communication between threads in the same group, meaning that parallel reductions can be efficiently implemented for stream compaction and expansion operations. In our pipeline, we use this property similarly to Kubisch [Kub14] to create compact lists of occupied blocks, improving GPU utilization for subsequent processing. Since mesh shaders are also executed in groups, they support efficient implementation of typical compute shader algorithms, such as the Marching Cubes algorithm, with the crucial difference that the primitives extracted by mesh shader workgroups can be directly consumed by the rasterizer.

The possibility to create a variable number of mesh shaders from one task shader means that task and mesh shader pipelines are well suited for implementation of algorithms that use tree expansion to dynamically allocate computational resources to relevant tasks. We use this feature to allocate workgroups that create geometry for occupied subvolumes.

3.2. Hierarchical Computation Structure

We exploit spatial coherence in the data volume by applying parts of the pipeline on subvolumes with carefully chosen granularity. We denote the dimensions of the *volume* as $X \times Y \times Z$ voxels. Isosurface geometry is extracted by defining *cells*, each of which is a cube with a voxel value at each vertex, and applying the Marching Cubes cell classification [LC87] on each cell. Cells that produce geometry are referred to as *occupied cells*. We define a *block* as a subvolume comprised of $U \times V \times W$ cells, where our experience shows that $U = 8, V = 8, W = 4$ is a sensible setting which allows 8-bit indexing of the 256 cells contained in a block. However, blocks are only a virtual subdivision of the volume and do not require any rearrangement of the data.

3.3. Preprocessing

The proposed approach requires the availability of the minimum and maximum data values for each block, which are precomputed and stored with the volume [EHK*06, LCDW16]. The min-max grid is stored as a 2-channel volume texture, with resolution $\frac{X}{U} \times \frac{Y}{V} \times \frac{Z}{W}$. The texture allows the occupancy of each block to be determined with a single texture lookup, and is referenced during the occlusion culling pass where the proxy geometry for occupied regions of the volume is determined. The min-max grid could be recomputed for smaller volumes on the fly, e.g. when streaming time-varying data sets.

3.4. Transient Isosurface Extraction and Rendering of PVS_x

In this subsection, we detail our approach to transient isosurface extraction and rendering. We propose a task and mesh shader pipeline, shown in Figure 4, consisting of a task shader that locates occupied cells and assigns them to mesh shader workgroups, and a mesh shader that performs transient triangle mesh extraction for a given list of occupied cells. The geometry extracted by each mesh shader instance is then directly passed on to the rasterizer.

Note that the following description of the task and mesh shader pipeline refers to the first and third components in Figure 2, labelled

© 2023 The Authors.
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

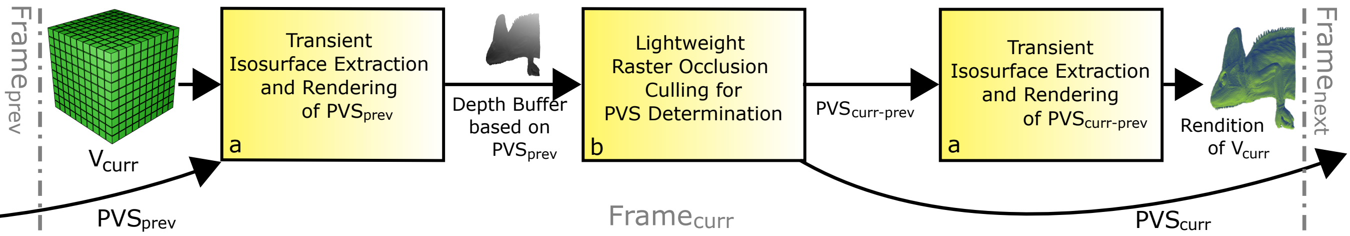
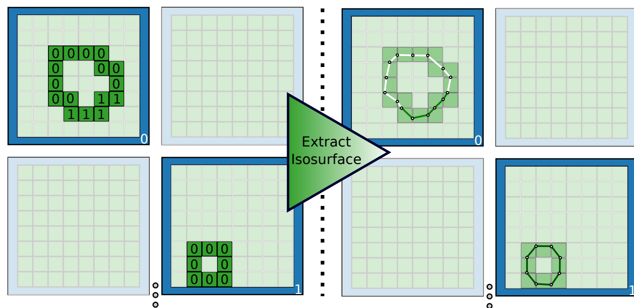
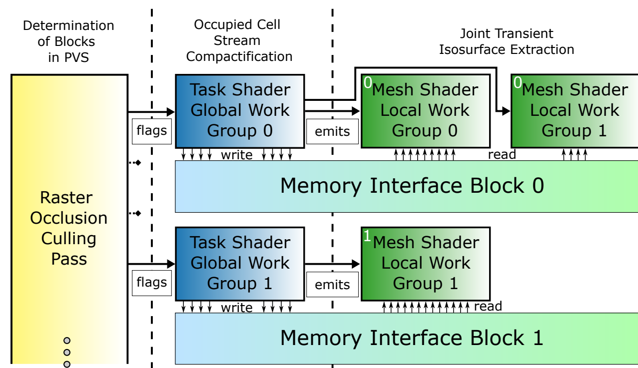


Figure 2: *Overview of our direct isosurface rasterization pipeline. The high-level design follows that of a typical raster occlusion culling pipeline [Kub14, KT14] to extract and render triangle mesh data. In our case, we render the potentially visible parts of an isosurface from volume V_{curr}. For brevity, we use the term 'PVS' in this illustration to refer to the indices of the potentially visible set of blocks, rather than the blocks themselves. A detailed visual overview of the implementation of the transient isosurface extraction and rendering stages (**a**, cmp. Subsection 3.4) as well as the lightweight raster occlusion culling for PVS determination pass (**b**, cmp. Subsection 3.5), is shown in Figures 4 and 5, respectively.*



(a) *Illustration of active task shader workgroups (blue rim, saturated), and occupied cells (left-hand side, green, labeled) located by each workgroup. Same labels within a task shader workgroup indicate occupied cells belonging to the same occupied cell list and are handled by the same mesh shader workgroup. Each mesh shader work group extracts a part of the potentially visible isosurface (right-hand side, different color per isosurface of different work groups) and passes the geometry on to the rasterizer.*



(b) *Tree expansion in our rendering pipeline for the example given in (a).*

Figure 3: *Tree expansion example for sparsely occupied volume presented in (a), showing task shader workgroups emitting mesh shader workgroups (b). The memory interface block (see Listing 1) which is shared between a task shader workgroup and its emitted mesh shader workgroups provides mesh shader workgroups with a list of occupied cells to process, increasing the chance of high warp occupancy.*

with **(a)**, and that both passes work on mutually exclusive sets of blocks depending on their visibility state in the current and previous frame [KT14, LJSL21].

3.4.1. Task Shader: Cell Classification and Task Generation

Each task shader workgroup processes one block, for which it identifies occupied cells, creates a compact list of those occupied cells, and emits mesh shader workgroups that will each process the cells corresponding to their respective part of the list. This approach makes effective use of the tree expansion feature of task and mesh shader pipelines. Identifying occupied cells and then emitting mesh shader workgroups which act only on occupied cells minimizes the number of mesh shader invocations, which has a significant positive effect on performance.

Each task shader workgroup writes into a *memory interface block* following the structure shown in Listing 1. A 32-bit index denotes the block worked on by the task and mesh shader workgroups. After the task shader workgroup has processed a block, the *dense_occupancy_index* buffer contains a compact list of the occupied cells in that block. Each mesh shader workgroup works on part of the list of occupied cells, starting from an offset contained in the *occupied_cell_list_offset* buffer, with length contained in the *occupied_list_length* buffer.

Listing 1: *Extract and draw shader interface for 256 cells per block*

```
taskNV out Task {
  uint32_t baseID;
  uint8_t  dense_occupancy_index[256];
  uint8_t  occupied_cell_list_offset[32];
  uint8_t  occupied_list_length[32];
} INTERFACE_TASK_MESH_EXTRACT_AND_DRAW;
```

We process cells in blocks of size 256 to enable indexing of all cells within a block with a single 8-bit integer. This approach reduces the size of the interface block, which we found to improve performance. Each thread in a mesh shader workgroup will process a single cell, meaning that the maximum number of cells that each mesh shader workgroup can process is 32. However, we found the number of vertices produced by each workgroup to be a decisive factor in achieving high performance. Empirically, we found that

© 2023 The Authors.
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

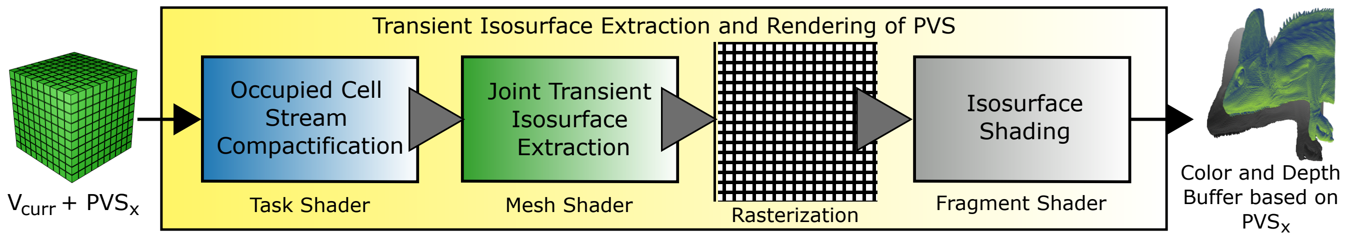


Figure 4: *Detail view of our 'Transient Isosurface Extraction and Rendering of PVS' pass (compare Overview in Figure 2). Based on the dense input volume and one of the two sets of PVS indices, PVS_{prev} or $PVS_{curr-prev}$, we efficiently identify sparse occupied cells within our task shader workgroup (blue) and pass compactified cell lists on to the mesh shader stage (green), in which a workgroup extracts the geometry of up to 32 sparsely distributed cells. The extracted geometry is then directly passed on to the rasterizer and subsequently shaded in the fragment shader instances. Note that, depending on whether this stage is the first or third in the overview pipeline, the result may be a rendering of the best occluders (using PVS_{prev}) based on the last frame, or the complete image for the current frame (using $PVS_{curr-prev}$). For more details on the transient isosurface extraction we refer the reader to Subsection 3.4.*

the locally optimal limit on the number of output vertices for our test data sets was 96. Since the number of vertices extracted from a cell varies between 3 and 12, assigning 32 cells to each mesh shader workgroup would often result in the limit of 96 vertices being exceeded. Therefore, we assign a dynamic number of cells to each mesh shader workgroup. The smallest number of cells processed by each mesh shader workgroup is 8, considering that in the case where every cell produces 4 disconnected triangles, giving 12 vertices, one mesh shader workgroup could operate on 8 cells before reaching the vertex limit. Accordingly, there is a maximum number of 32 lists, since 256 cells in total / 8 cells per list = 32. This number is reflected in the array sizes of the $occupied_cell_list_offset$ and $occupied_list_length$ buffers.

The process followed by the task shader can be divided into the following steps:

1. Individual cell classification and vertex count generation
2. Stream compaction and task generation

Individual Cell Classification and Vertex Count Generation. The task shader workgroup first uses the MC algorithm to determine whether each cell in the block is occupied, and if so, how many vertices will be required to represent the isosurface inside the cell. To process a block of 256 cells with 32 threads, each thread works on several cells. Each thread in the workgroup samples the voxels which are required to identify the state of all cells that it is responsible for. Subsequently, each thread looks up the state of its cells, stores the cells' occupancy flags, and writes the number of vertices that the cells would create into a shared memory buffer.

Note that the task shader stops executing the MC algorithm after identifying the per-cell vertex count, because it requires the vertex counts to determine which mesh shader workgroups will work on which occupied cells , but does not require any further information about the primitives extracted by MC. It may seem counter-intuitive to stop the MC algorithm after determining the per-cell vertex count, since it would be possible for the task shader to extract the required primitives, having already sampled the volume, and send indexed vertex buffers to mesh shader instances for subsequent rasterization. This, however, would drastically increase the amount of interface block memory required, compared to our approach of sending 8-bit indices marking occupied cells, because the interface block would require capacity for 12 vertices per cell. Passing a large amount of memory between shader instances would reduce performance notably.

Stream Compaction and Task Generation. The per-cell occupancy calculated in the cell classification stage is used to perform a reduction [BCF*17] to obtain the offset of each occupied cell's index in the dense set of indices written into the interface block's *dense_occupancy_index* buffer. Since the reduction is performed by 32 threads inside a workgroup, each group of 32 cells is processed in successive iterations. From each iteration, a cell obtains a local offset within its group of 32 cells. In order to calculate offsets to the start of each group, the last thread in the workgroup shares the total number of occupied cells within each group, allowing each thread to calculate a global offset for all cells within a block (similar to Liu et al. [LCDW16]). In contrast to Liu et al., we are able to omit expensive atomic operations, because offsets can be efficiently shared among all threads through *subgroupBroadcast*-operations.

Next, a single thread traverses the occupied cell buffer and divides the cells into lists that will each be processed by a different mesh shader workgroup. The thread begins to build a list by passing over the *dense_occupancy_index* buffer and accumulating the number of vertices that would be created for each occupied cell. As soon as the list comprises of 32 cells, or the maximum number of 96 vertices would be exceeded, one list is ended and a new one started. The start offset and the length for each completed cell list are stored in the $occupied_list_length$ and $occupied_cell_list_offset$ buffers respectively (see Listing 1). Finally, one thread writes the number of mesh shader workgroups created into the *gl_TaskCountNV* output variable.

3.4.2. Mesh Shader: Transient Isosurface Extraction

Each mesh shader workgroup, consisting of 32 threads, uses its *gl_WorkGroupID* to directly look up the the start offset and length of the list of cells that it will operate on.

Each active thread determines the ID and location of its cell, and

© 2023 The Authors.
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

performs the MC algorithm to extract the vertices and indices that represent the isosurface. We then use another efficient intra-warp reduction to determine offsets that each thread uses to write indices and projected vertices into the corresponding output arrays. The generated vertices are evenly distributed across the entire warp and projected cooperatively by all threads. One designated thread writes the total primitive count produced by each mesh shader workgroup. After the mesh shader workgroup has finished execution, the primitives are passed on to the rasterizer, where shading can be computed by a standard fragment shader.

Considerations regarding unique vertex extraction across cells. As our pipeline extracts vertices from occupied cells independently, some vertices may be duplicated in the vertex buffer produced by a mesh shader workgroup. Although it would be possible to detect these cases and remove duplicated vertices, as shown by Liu et al. [LCDW16], the potentially sparse nature of the occupied cells within a block means that unique vertices are hard to identify without additional analysis that may increase branch divergence.

3.5. Raster Occlusion Culling for Regular Volume Data

To achieve low draw times for volumes that exhibit high depth complexity, we determine a potentially visible set (PVS) of blocks to avoid extracting geometry that is not visible. This is especially relevant for our object-order approach to isosurface extraction, in that we benefit from reducing the number of task shader workgroups that are launched to process blocks that will not contribute to the displayed isosurface. We determine a PVS of blocks with a custom occlusion culling pass that conceptually follows a raster occlusion culling scheme [Kub14, KT14], shown in Figure 5.

3.5.1. Task Shader: Block-wise Stream Compaction

The raster occlusion culling stage follows a similar idea to the isosurface extraction stage, in that a sparsely-occupied set of subregions is reduced to a dense list of occupied subregions in the task shader, before primitives are extracted from occupied subregions in the mesh shader. In this stage, the task shader analyzes a localized group of 256 blocks, and creates mesh shader workgroups that process sets of 32 occupied blocks. The occupancy of each block can be determined by sampling the min-max texture at the corresponding location, and determining if the isovalue lies between the sampled minimum and maximum values. A dense list of occupied blocks is then generated in the same manner as a dense list of occupied cells is created in the isosurface extraction stage. Every mesh shader workgroup emitted by one task shader workgroup will operate on 32 blocks, apart from the final workgroup which may operate on between 1 and 32 blocks. Since the mesh shader workgroups operate on lists of fixed lengths, no list offset or length buffers are required in the interface block (shown in Listing 2).

Listing 2: *Raster occlusion culling shader interface for 256 blocks per workgroup.*

```
taskNV out Task {
  uint32_t baseID;
  uint32_t num_occupied_blocks;
  uint8_t  block_occupancy_index[256];
} INTERFACE_TASK_MESH_RASTER_CULL;
```

3.5.2. Mesh Shader: Proxy Geometry Creation

The mesh shader in the raster occlusion culling stage functions similarly to that of the isosurface extraction stage (see Subsection 3.4.2). Each thread in the mesh shader workgroup computes an offset into the *block_occupancy_index* array based on the thread and workgroup indices, ensuring that the index is less than the *num_occupied_blocks* variable. Each active thread creates proxy geometry for one occupied block. In contrast to the implementation by Kubisch [Kub14], the proxy geometry for each block does not comprise the three potentially visible sides of a cube, but a single quad containing the bounding box of the block when projected into screen space, with minimal z-coordinate of the block's corners. This reduces the number of triangles created per occupied block from 6 to 2, restricting the number of primitives created by a mesh shader workgroup to 64 and the number of vertices created to 128, instead of $32 \times 7 = 224$ vertices corresponding to 3 visible cube faces. Rasterizing fewer primitives is also beneficial because efficient raster occlusion culling implementations leverage the representative fragment test [KTB*18] extension, which allows the rasterizer to terminate early-depth testing even earlier for every primitive, if at least one fragment shader instance was invoked, that is, when one fragment of the primitive is visible. Since this extension cannot terminate early-depth testing for multiple associated primitives, rasterizing fewer primitives to begin with greatly increases the likelihood of early termination of the rasterization and therefore leads to fewer buffer writes in the subsequent fragment shader stage. Moreover, when using multi-view rendering features to rasterize proxy geometry per-eye in a joint pass, generating one quad per eye facilitates a more straightforward implementation. In the case where all potentially visible sides of a cube would need to be rendered, one would need to account for rendering more than 3 sides if the block is located between the viewer's eyes on any of the three cardinal axes.

After the vertex and index buffers for the blocks' quad proxies are written, the geometry is passed on to the rasterizer. As shown by other techniques based on raster occlusion culling [KT14, LL20], each fragment that passes the early depth test and representative fragment test writes into a preallocated slot of a visibility buffer, based on the implicit ID of each block. After the fragment shader stage, we run a bit compactification kernel for the currently created indices PVS_{curr} and subsequently build two sets of PVS data in a final compute shader pass. The only difference between the compute passes and the reference implementation of Kubisch [Kub14] is that we create both PVS_{curr}, which is used in the next frame to render the best occluders based on temporal coherence, and $PVS_{curr-prev}$, which determines the blocks which became visible since the last frame, in one common compute shader pass. A compute shader is used here because larger block sizes can be launched for the PVS compactification and output passes than if mesh shaders were used, which allows better utilization of the streaming multiprocessors on which the compute kernels are executed.

To ensure that our results can be reproduced, we provide a minimal rendering framework and example implementing our proposed technique, as well as the reference techniques, in both mono and stereo modes as supplementary material accompanying this paper.

© 2023 The Authors.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

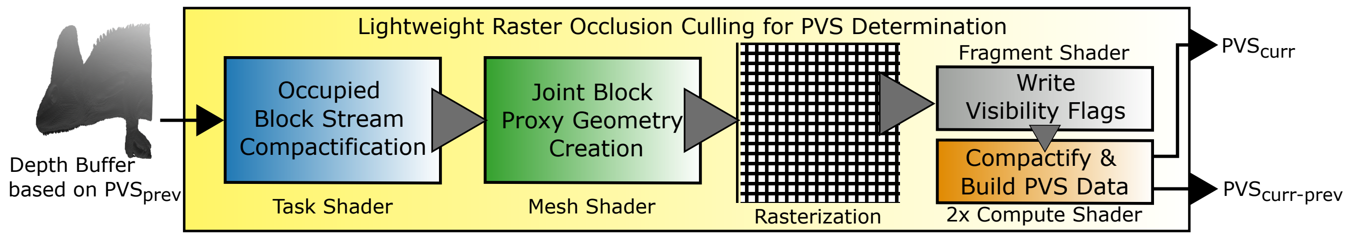


Figure 5: *Detail view of our 'Lightweight Raster Occlusion Culling' pass (compare overview in Figure 2). The task shader provides the mesh shader with a compact list of occupied blocks. Proxy geometry for each block is created in the mesh shader to test block visibility. Visibility flags are then to create compactify potentially visible block indices, as in [Kub14]. For details on our domain-specific extension of occlusion culling for dense volume data, we refer the reader to Subsection 3.5.*

4. Evaluation and Discussion

We compare the performance of our rendering pipeline against two reference isosurface rendering approaches: a direct, ray-marching-based isosurface visualization approach, and a 'best-case' indirect isosurface visualization technique.

Ray-marching Reference. The ray-marching reference technique is accelerated using a min-max octree to facilitate empty space skipping. The min-max octree is constructed by creating an MIP texture from the min-max texture used to determine block occupancy in our pipeline (see Subsection 3.3). For straightforward octree creation, we choose to construct the octree on a base level with equal size along all three cardinal axes. The comparison with a ray-marching technique is intended to assess whether our pipeline can outperform direct isosurface visualization approaches when faced with different rendering resolutions, volume sizes, and volume sparseness.

Optimized Mesh Reference. The comparison against an ideal indirect isosurface visualization technique serves to assess how close our approach comes to the best possible performance. For this reference, we record only the time taken to render a highly-optimized mesh representation of the isosurface. We extract vertex positions and normals for the mesh representation with the same extraction approach used in our pipeline, and write an explicit representation of the geometry to GPU memory. Duplicate vertices are removed, and the mesh is divided into meshlets. Rendering this reference mesh, which does not consider time taken for geometry extraction and occlusion culling, represents both the most efficient version of a forward mesh renderer for an optimized mesh, and an infinitely fast indirect isosurface extraction approach.

Occlusion Culling Granularity. To provide a fair comparison between our approach and the ray-marching reference, we perform raster occlusion culling for our approach on the same granularity as the leaf-level of the octree in the ray-marching reference; specifically for regions comprised of $8 \times 8 \times 8$ cells. Since our extraction stage operate on blocks of $8 \times 8 \times 4$ cells, we launch two task shader workgroups for each potentially visible region.

Shading. The fragment shader implementation is identical for both the rasterization-based approaches. Vertex normals, sampled on the fly using the central difference method, are passed to the fragment shader and used to perform Gooch Shading [GGSC98]. The ray-marching reference is entirely implemented in a fragment shader, and computes the normals in the same way when an isosurface intersection is found, also to apply Gooch Shading.

Name	# Voxels	Bit Depth	Size in GB
CT Head	256^3	8	0.02
Vertebra	512^3	8	0.13
Chameleon	1024^3	16	2.00
Supernova	30×512^3	32	15.30

Table 1: *Test Volume Data Sets. In addition to the static volumes* CT Head, Vertebra *and* Chameleon, *we evaluate our rendering approach with 30 time steps of the* Supernova *data set.*

Test System Specifications. Our test machine is equipped with a NVIDIA RTX 3090 graphics card with 24 GB of video RAM, as well as an *Intel Core i9-9900X* CPU running at 3.50 GHz, with 128 GB of RAM.

Software. All measurements were taken in an isosurface rendering library containing implementations of our direct isosurface rendering approach, the min-max-octree isosurface ray-marching approach, and an isosurface extraction, optimization and rendering approach. All extraction and rendering passes are implemented using a combination of C++, OpenGL, and its shading language GLSL, with the corresponding extensions for task and mesh shaders. The ray-marching approach uses three iterations of the binary search algorithm for isosurface refinement and a step size of 0.5 times the cell edge length. The isosurface extraction approach uses Kapoulkine's *meshoptimizer* C++ library [Kap22] to create meshlets with at most 64 vertices after the potentially visible surface is extracted, thereby following Kubisch's [Kub18] recommendation for default meshlet sizes.

4.1. Test Scenarios

We evaluate our pipeline in three different scenarios. The first scenario (Subsection 4.1.1) compares the rendering performance of our pipeline with that of the two aforementioned reference techniques, under varying rendering resolutions for static viewpoints

© 2023 The Authors.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

and static isovalues, using three volumetric data sets of increasing size (see Table 1). Note that for straightforward implementation of the octree-accelerated ray-marching approach, all volumes were padded such that size in each dimension was a power of two.

In our second scenario (Subsection 4.1.2), we compare the rendering performance of our approach against that ray-marching reference approach when a new time step of a temporally varying data set is rendered every frame. Rendering resolution is fixed in this scenario. We use the first 30 out of 60 time steps of the *Supernova* sequence, since those fit into the available graphics memory. In addition to evaluating the behavior when rendering temporally-varying data, we evaluate the performance of both approaches when the isovalue is constantly varied, simulating a scenario where the user adjusts the isovalue to explore the data set. The *Chameleon* data set is used for this scenario, which is particularly interesting as the occupancy of cells in the volume varies drastically for different isovalues.

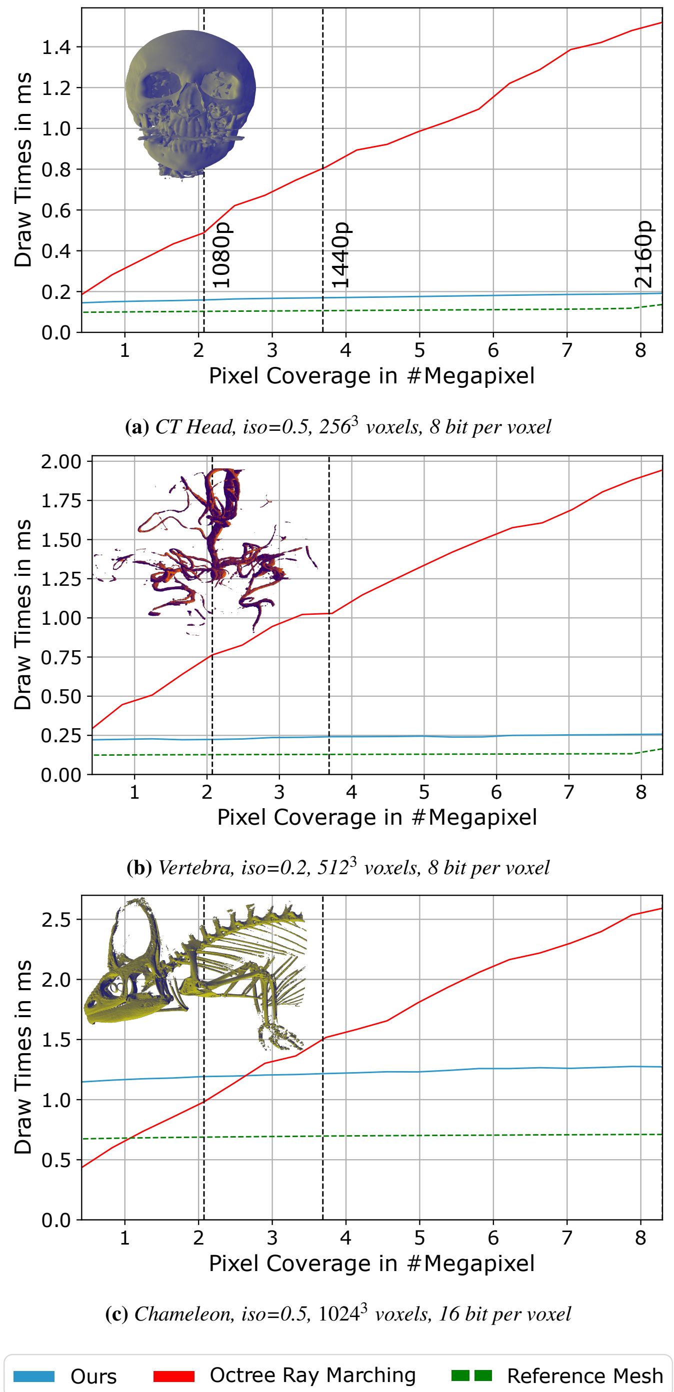
In our third scenario (Subsection 4.1.3), we compare and evaluate the stereo rendering performance of all three implemented rendering techniques. This scenario is intended to demonstrate that rasterization-based techniques, such as our approach and the optimized mesh reference, enable us to make use of mesh shaders' multi-viewport rendering capabilities [KBU*17] to share a large portion of the work for both eyes. In this section we prove that our approach is better suited to multi-viewport rendering than the ray-marching reference.

4.1.1. Rendering Resolution-dependent Performance

The comparison of our transient isosurface rendering approach against the octree-accelerated ray-marching and indirect isosurface rendering approaches is presented in Figure 6. Rendering performance (draw time) is evaluated for a range of rendering resolutions. In all cases, the volumes were positioned to fill the screen, such that the number of pixels rendered directly corresponds to the number of rays generated in the ray-marching-based implementation. The isosurfaces remained completely within the camera's viewing frustra. The isovalues were chosen arbitrarily, but performance at other isovalues for smaller data sets reflects the tendencies shown. We perform an in-depth evaluation of the performance across different isovalues for the largest test data set in Subsection 4.1.2.

Our approach outperforms the ray-marching reference approach when rendering the reference data sets at most of the rendering resolutions. The only exception is observed when rendering the *Chameleon* data set, where the ray-marching approach is more efficient for low-to-medium resolutions. For the maximum test resolution of 3840×2160 pixels, we achieve an average speedup over the ray-marching approach of $7.92_{CT\ Head}$, $7.64_{Vertebra}$ and $2.05_{Chameleon}$, respectively.

The reference mesh approach always outperforms our approach, by a factor of $1.61_{CT\ Head}$, $2.03_{Vertebra}$ and $1.85_{Chameleon}$, respectively. For the time-varying *Supernova* data set (compare Figure 7), the reference approach is 2.1 times faster than ours. However, considering that the isosurface representation is heavily optimized in this reference, we argue that our approach is competitive with both offline optimization and indirect rendering scenarios, in which the



(a) *CT Head, iso=0.5, 256^3 voxels, 8 bit per voxel*

(b) *Vertebra, iso=0.2, 512^3 voxels, 8 bit per voxel*

(c) *Chameleon, iso=0.5, 1024^3 voxels, 16 bit per voxel*

Figure 6: *Average draw time comparison for 3D volume data sets under varying rendering resolution with an aspect ratio from 16:9. Our approach (blue) clearly outperforms the octree ray-marching method (red) even at the lowest resolutions for small volumes, independently if the isosurface is rather dense (a) or sparse (b). For the Chameleon data set, our approach is faster starting at 2.7 megapixels resolution. The reference mesh (dashed-green) renders approximately 1.4 to 2.1 times faster across all of our data sets.*

© 2023 The Authors.
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

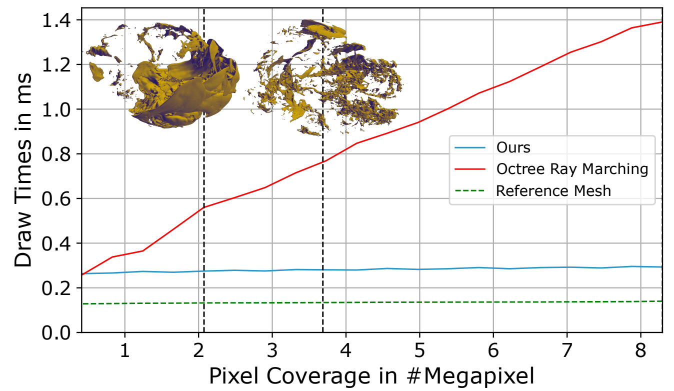
time taken to extract the explicit isosurface representation would need to be added to the presented rendering times.

In contrast to ray marching-based techniques, but similarly to the reference mesh rasterization technique, our proposed approach is hardly influenced by the rendering resolution. This is because nearly all stages of our approach work in volume space and are therefore not affected by the rendering resolution. The isosurface rasterization stage, which is dependent on rendering resolution, executes only a lightweight fragment shader program implementing Gooch Shading. More complex fragment shaders would lead to a higher dependence on display resolution, affecting our technique and the reference mesh approach.

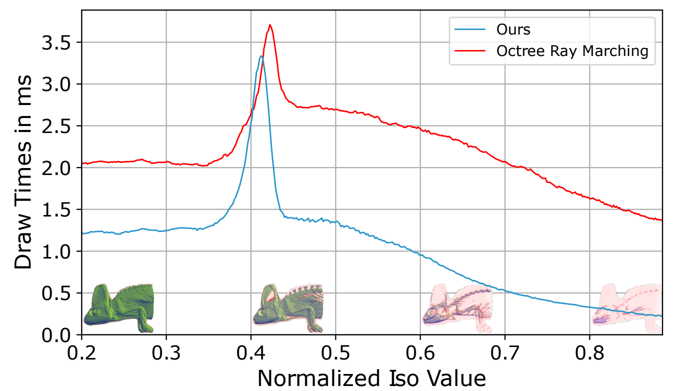
4.1.2. Performance for Time-Varying Data

To investigate how temporal coherence affects rendering performance, we evaluate the performance of all three approaches when rendering isosurfaces from time-varying data sets (see Figure 7). In contrast to the previous tests, the volume data, and therefore the set of occupied cells, changes from one frame to the next, which we expect to result in slightly decreased occlusion culling performance for our approach. Despite this, a similar pattern to that shown by the static test cases is revealed (see Figure 7a), in that our approach still outperforms the ray-marching reference for almost all tested resolutions, with a maximum speedup factor of 4.74 at a rendering resolution of 3840×2160 pixels.

We also evaluate our approach in a scenario where the isovalue is constantly varied, with the results shown in Figure 7b. Rendering resolution is kept constant at 3840×2160 pixels, but the isovalue is varied from 0.2 to 0.9 over 5 seconds. At an isovalue of 0.4, the skin starts to disappear and many high-frequent and noisy parts of the underlying bone structure become visible. Thus, many more cells containing isosurfaces need to be processed and rendered. Our block-based occlusion culling is less effective when the isosurface is noisy, reducing the performance of our approach to a level similar to the octree ray-marching approach.



(a) *Supernova, iso=0.08, 512^3 voxels, 32 bit per voxel $\times$ 30 time steps. Draw times averaged across all steps for different rendering resolutions.*



(b) *Chameleon rendered for a resolution of* 3840×2160 *pixels for a continuously changing isovalue from 0.2 to 0.9 over about 5 seconds. Starting from iso=0.2, the superimposed renditions show the extracted isosurface in iso increments of 0.2. The iso range around 0.4 is particularly challenging for our approach, since the skin starts to disappear and many high-frequent and noisy parts of the isosurface are rendered.*

Figure 7: *Draw time comparison for time-varying data. We repeat the evaluation from Subsection 4.1.1 using 30 time steps of a time-series (a) and exchange the active volume after each frame. We furthermore use a static volume for which we smoothly vary the isovalue in a given range after each frame (b).*

4.1.3. Stereoscopic Rendering Performance

We compare the monoscopic and stereoscopic rendering performance of all three approaches using a standard stereoscopic viewing setup with an interpupillary distance of 6 cm. In stereoscopic mode, we render each eye into a different layer of a two-layered custom framebuffer with resolution 3840×2160. The geometry passes in our approach and the optimized mesh reference are split after the mesh shader stage, where we use the multi-view rendering feature by writing primitives to one vertex array per view. Figure 8 shows the overhead incurred for stereoscopic rendering, compared to monoscopic rendering. The octree ray-marching technique cannot benefit from multi-view rendering, and so takes approximately twice as long to render two views, as expected. For our approach, the overhead for rendering two views creates is only 20 to 40%.

5. Conclusion and Future Work

This work presents a novel and efficient approach for direct rasterization of isosurfaces from scalar volumetric data sets. Our direct rasterization approach is largely independent of rendering resolution, and outperforms approaches based on ray marching for increasing resolution. This is made possible by integrating an effective occlusion culling technique into the task and mesh shader pipeline such that the extraction and rendering focuses on visible parts of the isosurface. Our approach requires negligible preprocessing and only a small amount of additional memory for auxiliary data structures because no explicit isosurface representation is stored, which also minimizes bandwidth requirements with graphics memory. Since the proposed technique is based on rasterization of standard graphics primitives, it lends itself well to being combined with multi-viewport rendering, where the transient isosurface extraction stage can be executed only once for both views in cases where viewing perspectives are very similar, such as for stereoscopic rendering scenarios.

© 2023 The Authors.
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

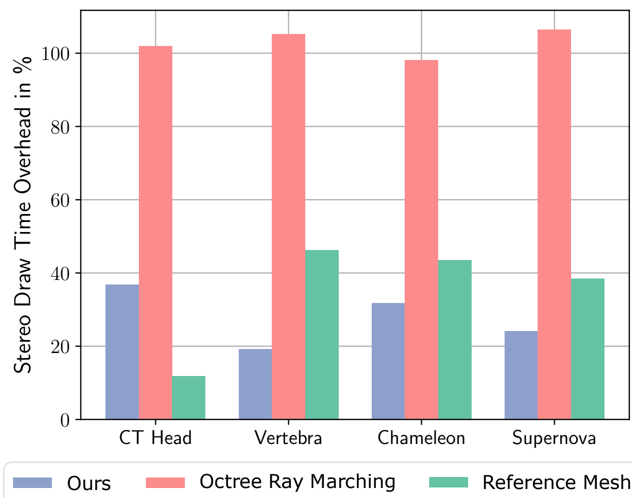


Figure 8: *Comparison of draw time overhead for rendering at resolutions of stereoscopic vs monoscopic* 3840×2160 *pixels. Using multi-view rendering, stereoscopic rendering merely creates an overhead of 19.2 to 36.8% for our proposed technique (blue) and 11.8 to 46.2% for the reference mesh (green), whereas the ray-marching approach (red) does not have any benefit since the rays are entirely separate for both eyes and therefore takes twice the time compared to monoscopic rendering.*

Our implementation is currently limited to the extraction of opaque isosurfaces. Efficient order-independent transparency techniques, such as per-pixel linked lists [KSS17, SBF15] or moment-based order-independent transparency [MKKP18] could be easily used in combination with our technique, but their additional resource requirements scale linearly with the rendering resolution, potentially losing the advantage over ray marching. In any case, ray-marching approaches remain the method of choice for visualization techniques based on volume compositing.

Our approach does not currently tackle output-sensitive rendering. However, it could be extended to select an appropriate octree level for extraction of the isosurface geometry. To support continuous level-of-detail isosurface rendering, it is worthwhile to explore the combination of the fundamental idea of transient isosurface extraction with dual contouring approaches [JLSW02], since related work suggests its suitability for crack-free level-of-detail rendering of implicit geometry. An alternative for applications with highly varying level-of-detail requirements across the volume could be the application of a hybrid technique, identifying regions of the volume that would benefit from ray marching instead of transient isosurface extraction. Although switching to a ray-marching approach in minification cases could be a solution in such situations, other advantages of our approach, such as support for multi-viewport rendering, could not be exploited.

Although there is clearly room for further research in the field of direct isosurface rasterization for volumetric data, this contribution shows that direct rasterization of a geometry representation generated on-the-fly can outperform non-trivial ray-marching-based approaches even at moderately high resolutions. Therefore direct rasterization is an option to consider when no explicit geometry representation is available, or when explicit geometry is expensive to generate such as for procedural geometry and procedural 3D or 4D volumes.

Acknowledgments The datasets used throughout this work are courtesy of: North Carolina Memorial Hospital (*CT Head*); Volvis Project Group of Tübingen University, Germany and Michael Meißner, Viatronix Inc., USA. (*Vertebra*), Digital Morphology Project at the University of Texas at Austin, USA (*Chameleon*) and Dr. John Blondin at North Carolina State University through SciDAC Institute for Ultrascale Visualization (*Supernova*).

Our research is funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) under the project ID 444831328, SPP2236 - AUDICTIVE - Auditory Cognition in Interactive Virtual Environments. We thank the members of the Virtual Reality and Visualization Research Group at Bauhaus-Universitat Weimar (http://www.uni- weimar.de/vr) for their support and the reviewers of this paper for their constructive feedback that helped to improve the paper significantly.

Open Access funding enabled and organized by Projekt DEAL.

References

[BCF*17] Bolz J., Chajdas M., Fredriksen J.-H., Galazin A., Greig A., Hagan A., Hector T., Henning N., Kessenich J., Koch D., Leese G., Lottes T., Neto D., Petit K., Potter R., Riley C., Simpson R.: GLSL Extension: KHR_shader_subgroup, 2017. URL: https://github.com/KhronosGroup/GLSL/blob/master/extensions/khr/GL_KHR_shader_subgroup.txt. 6

[BHP15] Beyer J., Hadwiger M., Pfister H.: State-of-the-art in gpu-based large-scale volume visualization. *Comput. Graph. Forum 34*, 8 (dec 2015), 13–37. 3

[BWPP04] Bittner J., Wimmer M., Piringer H., Purgathofer W.: Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3 (2004), 615–624. 3

[CCG*07] Cunniff R., Craighead M., Ginsburg D., Lefebvre K., Licea-Kane B., Triantos N.: OpenGL Extension: ARB_occlusion_query, 2007. URL: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_occlusion_query.txt. 3

[DZTS08] Dyken C., Ziegler G., Theobalt C., Seidel H.-P.: High-speed marching cubes using histopyramids. *Computer Graphics Forum 27*, 8 (2008), 2028–2039. 2

[EHK*06] Engel K., Hadwiger M., Kniss J. M., Rezk-Salama C., Weiskopf D.: *Real-time volume graphics*, 1st edition ed. A K Peters Ltd., 2006. 1, 3, 4

[GGPP20] Galin E., Guérin E., Paris A., Peytavie A.: Segment Tracing Using Local Lipschitz Bounds. *Computer Graphics Forum 39*, 2 (2020), 545–554. 3

[GGSC98] Gooch A., Gooch B., Shirley P., Cohen E.: A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1998), SIGGRAPH '98, Association for Computing Machinery, p. 447–452. 8

[Gib98] Gibson S. F. F.: Using distance maps for accurate surface representation in sampled volumes. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization* (New York, NY, USA, 1998), VVS '98, Association for Computing Machinery, p. 23–30. 2

© 2023 The Authors.

[Har96] HART J. C.: Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10 (Dec. 1996), 527–545. 3

[HLJ*13] HUGHES D. M., LIM I. S., JONES M. W., KNOLL A., SPENCER B.: Ink-compact: In-kernel stream compaction and its application to multi-kernel data visualization on general-purpose gpus. *Computer Graphics Forum 32*, 6 (2013), 178–188. 2

[IRR*22] IBRAHIM M., RAUTEK P., REINA G., AGUS M., HADWIGER M.: Probabilistic occlusion culling using confidence maps for high-quality rendering of large particle data. *IEEE Transactions on Visualization and Computer Graphics 28*, 1 (2022), 573–582. 3

[JLSW02] JU T., LOSASSO F., SCHAEFER S., WARREN J.: Dual contouring of hermite data. *ACM Trans. Graph. 21*, 3 (jul 2002), 339–346. 2, 11

[JRZ*16] JIANG Y., RHO S., ZHANG Y., JIANG F., YIN J.: Multi-scale stream reduction for volume rendering on gpus. *Microprocessors and Microsystems 47* (2016), 133–141. 3

[Kap22] KAPOULKINE A.: meshoptimizer, 2022. URL: https://github.com/zeux/meshoptimizer. 8

[KBU*17] KUBISCH C., BROWN P., URALSKY Y., SMITH T., AND P. K.: OpenGL Extension: NV_mesh_shader, 2017. URL: https://www.khronos.org/registry/OpenGL/extensions/NV/NV_mesh_shader.txt. 9

[KSS17] KESSENICH J., SELLERS G., SHREINER D.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*, 9th ed. Addison-Wesley Professional, 2017. 11

[KT14] KUBISCH C., TAVENRATH M.: Opengl 4.4 Scene Rendering Techniques. In *NVIDIA GPU Technology Conference (2014)* (2014). 3, 4, 5, 7

[KTB*18] KUBISCH C., THANGUDU K., BOLZ J., BROWN P., WERNESS E., KNOWLES P.: OpenGL Extension: GL_NV_representative_fragment_test, 2018. URL: https://www.khronos.org/registry/OpenGL/extensions/NV/NV_representative_fragment_test.txt. 7

[Kub14] KUBISCH C.: nvpro-samples: gl_occlusion_culling, 2014. URL: https://github.com/nvpro-samples/gl_occlusion_culling. 3, 4, 5, 7, 8

[Kub18] KUBISCH C.: Introduction to turing mesh shaders, 2018. URL: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/. 4, 8

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1987), SIGGRAPH '87, Association for Computing Machinery, p. 163–169. 1, 2, 4

[LCD09] LIU B., CLAPWORTHY G. J., DONG F.: Fast isosurface rendering on a gpu by cell rasterization. *Computer Graphics Forum 28*, 8 (2009), 2151–2164. 3

[LCD15] LIU B., CLAPWORTHY G. J., DONG F.: Isobas: A binary accelerating structure for fast isosurface rendering on gpus. *Computers & Graphics 48* (2015), 60–70. 3

[LCDW16] LIU B., CLAPWORTHY G. J., DONG F., WU E.: Parallel marching blocks: A practical isosurfacing algorithm for large data on many-core architectures. *Computer Graphics Forum 35*, 3 (2016), 211–220. 2, 4, 6, 7

[LJSL21] LEE G. B., JEONG M., SEOK Y., LEE S.: Hierarchical raster occlusion culling. *Computer Graphics Forum 40*, 2 (2021), 489–495. 3, 4, 5

[LKE18] LEE S., KIM Y., EISEMANN E.: Iterative depth warping. *ACM Trans. Graph. 37*, 5 (oct 2018). 3

[LL20] LEE G. B., LEE S.: Iterative GPU occlusion culling with BVH. In *High Performance Graphics Posters (2020)* (2020). 3, 7

[MBW08] MATTAUSCH O., BITTNER J., WIMMER M.: CHC++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Proceedings Eurographics 2008) 27*, 2 (Apr. 2008), 221–230. 3

[MKKP18] MÜNSTERMANN C., KRUMPEN S., KLEIN R., PETERS C.: Moment-based order-independent transparency. *Proc. ACM Comput. Graph. Interact. Tech. 1*, 1 (jul 2018). 11

[NVI22] NVIDIA: nvsdk: Cuda samples, 2022. URL: https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/marchingCubes. 2

[NY06] NEWMAN T. S., YI H.: A survey of the marching cubes algorithm. *Computers & Graphics 30*, 5 (2006), 854–879. 2

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. In *Proceedings Visualization '98 (Cat. No.98CB36276)* (1998), pp. 233–238. 1

[SBF15] SCHOLLMEYER A., BABANIN A., FROEHLICH B.: Order-independent transparency for programmable deferred shading pipelines. *Comput. Graph. Forum 34*, 7 (oct 2015), 67–76. 11

[UP20] USHER W., PASCUCCI V.: Interactive visualization of terascale data in the browser: Fact or fiction? In *2020 IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)* (2020), pp. 27–36. 2

[WCB86] WYVILL G., CRAIG M., BRIAN W.: Data structure for soft objects. *The Visual Computer 2*, 4 (Aug. 1986), 227–234. 2

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF K. E.: Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., p. 77–88. 3

[ZTTS06] ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.-P.: On-the-fly point clouds through histogram pyramids. In *11th International Fall Workshop on Vision, Modeling and Visualization 2006 (VMV2006)* (Aachen, Germany, 2006), Kobbelt L., Kuhlen T., Aach T., Westermann R., (Eds.), European Association for Computer Graphics (Eurographics), Aka, pp. 137–144. 2, 3

© 2023 The Authors.

14678659, 2022, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1111/cgf.14670 by Bauhaus-Universitaet Weimar, Wiley Online Library on [23/05/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License