ISSN 1213-6972 Journal of WSCG http://www.wscg.eu Vol.28, No.1-2, 2020

Clustered Grid Cell Data Structure for Isosurface Rendering

Fredrik Nysjö
Centre for Image Analysis, Dept. of Information Technology,
Uppsala University, Sweden
{fredrik.nysjo}@it.uu.se

ABSTRACT

Active grid cells in scalar volume data are typically identified by many isosurface rendering methods when extracting another representation of the data for rendering. However, the use of grid cells themselves as rendering primitives is not extensively explored in the literature. In this paper, we propose a cluster-based data structure for storing the data of active grid cells for fast cell rasterisation via billboard splatting. Compared to previous cell rasterisation approaches, eight corner scalar values are stored with each active grid cell, so that the full volume data is not required during rendering. The grid cells can be quickly extracted and use about 37 percent memory compared to a typical efficient mesh-based representation, while supporting large grid sizes. We present further improvements such as a visibility buffer for cluster culling and EWA-based interpolation of attributes such as normals. We also show that our data structure can be used for hybrid ray tracing or path tracing to compute global illumination.

Keywords

Point-based rendering, Visibility, Ray tracing

1 INTRODUCTION

Isosurface rendering requires finding the intersections of the isosurface for a particular isovalue in the data. The scalar volume data, for example, a computed tomography (CT) image, is typically sampled with tri-linear interpolation when finding intersections and computing smooth normals for shading. If dynamic isovalue is not required, indirect volume rendering techniques generally provide fast rendering and good data reduction. Mesh-based methods such as Marching Cubes (MC) [15] and dual contouring [11] find the active grid cells (the grid cells containing isosurface intersections) of the data and extract a polygonal representation of the isosurface. Point-based methods such as elliptical weighted-average (EWA)-based surface splatting [22, 1] typically extract a point for each active grid cell and render these points as small disks. Sparse storage of larger voxel bricks and hierarchical data structures such as octrees can also be used to provide better data reduction and faster traversal to direct volume rendering techniques such as isosurface raycasting.

Compared to the previously mentioned techniques, the use of grid cells themselves as rendering primitives is not extensively explored in the literature. Parker et al. [20] describe a method for analytically computing ray-surface intersections for active grid cells during isosurface raycasting. Zhang et al. [21] propose storing active edges instead of active grid cells, which are rendered by disk splatting or expanded and rendered as small point clouds. Cell rasterisation (CR) was proposed by Liu et al. [13] as a fast method for rasterising the active grid cells of a volume. Active grid cells are rasterised as point primitives via so-called billboard splatting, and the original volume data is sampled in the cells in the fragment shader to find intersections and compute smooth normals. Compared to indirect volume rendering techniques such as mesh- and point-based methods, cell rasterisation provide the same isosurface as isosurface raycasting. Liu et al. also use a data structure for fast sorting and traversal of the cells in front-to-back order for proper transparency rendering. However, their method provide no data reduction, since only grid positions are stored with the cells, and therefore the original volume data needs to be available in GPU memory for sampling during rendering.

In this paper, we propose a memory efficient data structure for storing the active grid cells of a volume. The data structure is easy to implement, does not require the original volume data to be available during rendering, and supports ray tracing in addition to rasterisation. Our main contributions in the paper are:

- A clustered grid cell data structure (CGC) for storing grid positions and corner scalar values of active grid cells. The data structure supports large grid sizes

Figure 1: Our proposed data structure for isosurface rendering: Left: 16^3 block volume with each block (blue wireframe) storing a pointer to a list of clusters (pink wireframes) of active grid cells extracted from a voxelised model; Middle-left: isosurface rasterised with cell rasterisation using scalar values stored with active grid cells and visibility culling of individual clusters (indicated by colors); Middle-right: final shaded isosurface after EWA-based normal interpolation and deferred normalized Blinn-Phong shading; Right: hybrid ray tracing using our data structure with cell rasterisation for primary rays and path tracing with three bounces and an environment map for secondary rays (after one frame, using 1 sample per pixel). Each cluster shown in the first two images consists of up to 120 active grid cells and an axis-aligned bounding box. A lower resolution (256^3 voxels) voxelisation of the Buddha model was used to clearer show the clusters and the effect of the attribute interpolation.

($128K \times 128K \times 128K$) and can be used for fast cell rasterisation via billboard splatting.

- Improvements to the cell rasterisation including visibility culling of clusters and EWA-based interpolation for smooth attribute interpolation of normals and other attributes.
- Showing that the proposed data structure can be used for hybrid ray tracing or path tracing to compute global illumination.

2 RELATED WORK

For direct isosurface raycasting, larger bricks or tiles are generally used instead of individual grid cells for data reduction and empty space skipping, as in Hadwiger et al. [7]. The voxel database (VDB) data structure by Museth [19] is commonly used in offline rendering and uses a shallow tree of internal nodes and leaf nodes to efficiently store volume data in tiles. Recently, Hoetzlein [9] also introduced GVDB for real-time rendering on the GPU. Our data structure shares some similarities with the VDB data structure, but does not, for example, provide random access to voxels, and is restricted to isosurface rendering by using smaller tiles grouped into clusters.

Sparse voxel octrees (SVOs) have been proposed for storage and ray tracing of large voxel models [12]. The SVO typically stores surface voxels in the leaf nodes of the octree, and is traversed during ray tracing. Jablonski et al. [10] propose an alternative method for rendering SVOs via billboard splatting and a screen-space voxel buffer. Large number of individual voxels can also be efficiently rasterised as cubes via billboard splatting and fast ray-box intersection tests [16]. However, for high quality isosurface rendering, a drawback of traditional SVOs representing voxels as small cubes is the limited contour information they provide, leading to a blocky appearance unless each voxel is projected to less than a few pixels. Efficient sparse voxel octrees (ESVOs) [12] improves this by storing additional contour information in the octree and voxels. Heitz et al. [8] use filtered signed distance fields and differential cone tracing to render SVOs with accurate anti-aliased contours. Marcus [17] proposes a SVO data structure storing scalar values (signed distances in their case) at the corners of each voxel, which is similar to the corner values we store for cell rasterisation, but used during ray tracing of the SVO.

Visibility culling can be important in rasterisation to improve performance and reduce overdraw. Livnat et al. [14] perform point-based view-dependent isosurface extraction from an octree, using a visibility framebuffer to prune non-visible regions. They further compute surface normals in a post-processing step for far

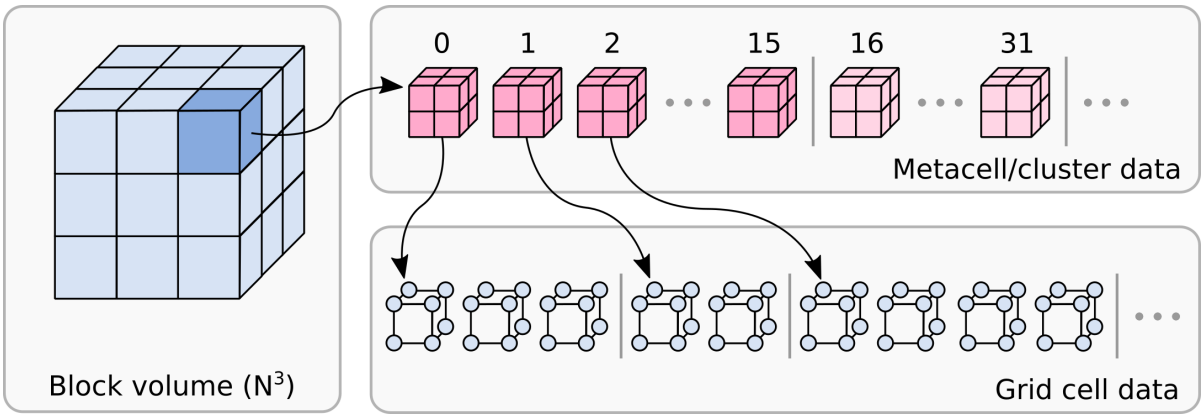


Figure 2: Overview of the clustered grid cell (CGC) data structure for storing active grid cells. Each block in the block volume (blue) stores a pointer to metacells (red) extracted in the block. The metacells are grouped into clusters of 16 metacells, with each metacell storing a pointer to the corner values of up to eight grid cells. The last metacell in each cluster stores a cluster bound instead of a pointer. Blocks, metacells, and grid cell corners are stored in separate buffers on the GPU. The memory layout for blocks and metacells is further described in Table 1.

points (minified surfaces), while using MC triangle normals for near points (magnified surfaces). For general occlusion culling of clustered geometry, a hierarchical Z-buffer [5] is often used in combination with pre-computed cluster bounds for conservative culling of clusters. For back-face culling, a normal cone can similarly be pre-computed and stored with each cluster. In deferred splatting [6] and the method for cluster culling we use in this paper based on a visibility buffer, the visibility culling is not conservative but does not require generating a hierarchical Z-buffer or storing cluster bounds or normal cones.

Rasterisation of small primitives (small triangles and points) can further lead to bad GPU utilisation when each primitive only occupies a few pixels. Evans [4] presents a point-based method replacing the standard rasterisation pipeline with stochastic splatting in a compute shader. Splatting is performed using 64-bit atomic instructions, thereby avoiding executing GPU threads for non-covered fragments in 2×2 pixel quads or fragments with zero alpha value. A drawback of this approach is that 64-bit atomic instructions are only available and exposed to the programmer on game consoles (PlayStation 4) and on certain NVIDIA GPUs (Maxwell and later generations).

3 METHODS

In Section 3.1, we describe the proposed CGC data structure for storing clusters of active grid cells, and the isosurface extraction. In Sections 3.2–3.4, we describe cell rasterisation using billboard splatting, and present further improvements such as a visibility buffer for back-face and occlusion culling and EWA-based attribute interpolation of attributes such as normals. In the final Sections 3.5 and 3.6, we discuss memory usage and hybrid ray tracing.

Block data	**Size**	**Metacell data**	**Size**
Metacell count	32 bit	Grid position (XYZ)	48 bit
Metacell pointer	32 bit	Cell mask	16 bit
Min value	32 bit	Cell pointer	32 bit
Max value	32 bit		
Total:	128 bit	**Total:**	96 bit

Table 1: Memory layout for blocks and metacells. Blocks and metacells are stored in separate GPU buffer textures, using 16 bytes per block and 12 bytes per metacell. Corner scalar values for active grid cells are stored in a separate buffer texture.

3.1 Data Structure

An overview of our proposed data structure is shown in Figures 1–2. We store extracted grid cells in a linear buffer on the GPU. Eight corner values per cell are stored as two RGBA values, with the component type matching the scalar precision of the volume data. To avoid storing a grid position with each cell, we group up to eight neighboring cells into a $2 \times 2 \times 2$ metacell storing a pointer, a cell mask, and a cell pointer. The metacells are stored in a second linear buffer, with each segment of 16 consecutive metacells representing a cluster.

For the isosurface extraction, we divide the volume data into $16 \times 16 \times 16$ number of blocks, and extract the active grid cells of each block in Morton order. To prevent metacells of the same cluster from spilling over into different volume blocks, we pad the last cluster of each block with empty metacells such that its size become 16. We also use the last metacell of each cluster to store a cluster bound instead of cell metadata. We perform the isosurface extraction on the CPU, and upload extracted metacells and grid cells to separate GPU buffer textures.

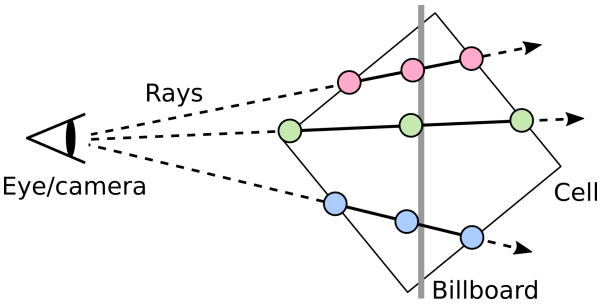


Figure 3: Cell rasterisation using billboard splatting: for each fragment rasterised for the billboard (gray line intersecting the cell), a ray-box intersection test is performed with the cell, and interpolated samples (indicated in colors) computed along the ray to determine the isosurface intersection. In our implementation, three samples per ray is used, which was found sufficient.

The memory layout for blocks and metacells is shown in Table 1. In our current implementation, grid positions are stored with 16 bit per component, allowing a maximum grid size of $128K \times 128K \times 128K$ voxels to be represented.

3.2 Cell Rasterisation

The basic idea of cell rasterisation using billboard splatting is illustrated in Figure 3. In the original cell rasterisation paper [13], a vertex shader is used to render active cells as a point sprites (`GL_POINTS` primitives). For each rasterised fragment of a point sprite billboard, a ray-box intersection test is performed with the cell, and the original volume data is sampled along the ray with tri-linear interpolation to find isosurface intersections. When an intersection is found, a normal gradient is also computed from the volume data for shading.

For efficient rendering of the clusters of grid cells in our data structure, we replace the vertex shader of previous approaches with a tesselation shader taking a single patch (`GL_PATCHES` primitive) per cluster as input and expands it into points for the cells in the cluster. Clusters that were culled in the last frame (Section 3.3) are rendered at $1/8$ rate (with at most on point per metacell per frame) until they become visible. Stored corner values are interpolated with manual tri-linear interpolation in the fragment shader. For the ray-box intersection test, we use the efficient slab test mentioned in Majercik et al. [16], which also can be used to rasterise cells as cubic voxels. To allow smooth interpolation of normals and other attributes for shading, we rasterise cells to a geometry buffer (G-buffer) and compute deferred normalized Blinn-Phong shading in a separate pass after attribute interpolation (further described in Section 3.4). A cell normal is computed in the tesselation shader from the corner values, and further used for back-face culling.

3.3 Visibility Buffer

To further improve the rasterisation performance, we introduce a visibility buffer to avoid performing full cell rasterisation of clusters that are fully occluded or back-facing. The visibility buffer stores a bit per cluster indicating the visibility after each frame. An RG32UI texture is used during the cell rasterisation pass to capture cluster IDs visible cells. After the cell rasterisation, we clear the current visibility buffer bits and update the visibility buffer from the ID texture in a separate compute shader pass. Similar to the disk-based deferred splatting approach by Guennebaud et al. [6], we render cells of non-visible clusters as single pixel splats for updating the visibility buffer. During fast camera movements, non-visible clusters becoming visible might result in flickering artifacts. We therefore introduce a second visibility buffer to record of new clusters becoming visible, and perform full cell rasterisation for those clusters in a second pass. During fast camera movements, there might still be some flickering artifacts, which can be seen in the accompanying video.

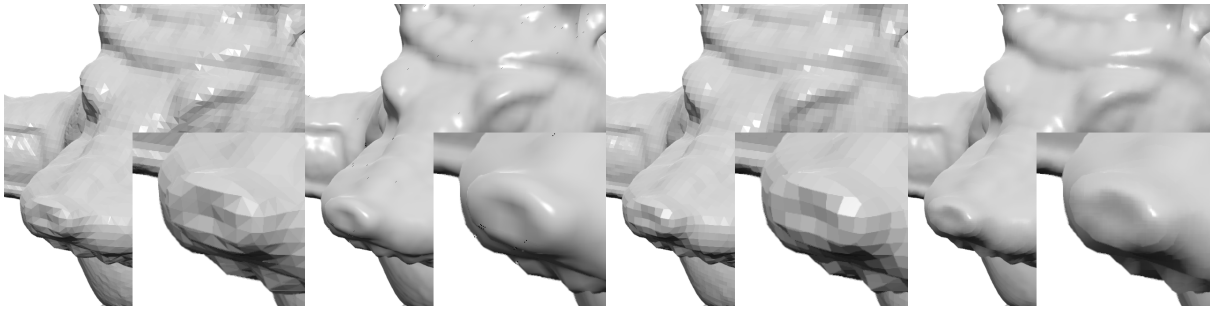
3.4 EWA-based Attribute Interpolation

Normals and other attributes should be smoothly interpolated before computing shading. However, our data structure does not allow random access to neighboring grid cells. For smooth interpolation of normals and other attributes, we introduce a second EWA-based attribute splatting pass in which cells are rendered as round splats with a weighted footprint, and attributes accumulated to the G-buffer. The results of the EWA-based interpolation is shown in Figure 4.

Laine et al. [12] propose performing interpolation as a post-processing step (after ESVO ray tracing) by computing a weighted average of neighboring pixels in a Poisson disk centered around each target pixel. The interpolation can use a large number of samples (up to 96) per pixel, which may be expensive on GPUs with low memory bandwidth. Specular surfaces are also not handled correctly because the interpolation is performed after shading. Jablonski et al. [10] propose a smaller 3×3 Gaussian filter for interpolating G-buffer normals before shading. We investigated replacing the post-processing in [12] with a stochastic interpolation scheme using temporal anti-aliasing (TAA) and fewer samples per frame. However, our EWA-based interpolation provided smoother normals and was possible to use without TAA. Another option would be storing additional corner values (for example, 32 instead of 8) with each grid cell, at the cost of significantly increased memory usage.

3.5 Memory Usage

Average memory usage per grid cell in our data structures, when including storage for metacells and the



(a) MC mesh (b) CR [13] (c) CGC (ours) (d) CGC+EWA (ours)

Figure 4: Isosurface comparison for voxelised Armadillo model: (a) MC mesh (without pre-computed normals); (b) isosurface from cell rasterisation with sampling in the original volume data; (c)–(d) isosurface from cell rasterisation of our CGC data structure, without and with EWA-based normal interpolation. For our method, the normal gradient of each cell was computed in the vertex shader from the stored corner values. A lower resolution (256^3 voxels) voxelisation of the Armadillo model was used to better show the effect of the normal interpolation.

block volume, is 11 bytes for 8-bit scalar data and 19 bytes for 16-bit scalar data. If 4-bit scalars provide sufficient precision for the application, for example, when scalars represent coverage values or truncated signed distances such as in some sculpting applications, the average memory usage may be further reduced to 7 bytes per cell. In addition to storage for the main data structure, we allocate 0.13 MB visibility buffers with capacity for 1M unique clusters, which was found sufficient even for the hairball dataset.

Volume	Resolution ($W \times H \times D$)	Type	Size (GB)
Dragon	$512 \times 512 \times 512$	8 bit	0.13
Plastic Skull	$512 \times 512 \times 512$	8 bit	0.13
Stag beetle CT	$832 \times 832 \times 494$	16 bit	0.68
Chameleon CT	$1024 \times 1024 \times 1080$	8 bit	1.13
Raptor	$2048 \times 1024 \times 512$	8 bit	1.07
Buddha	$1024 \times 2048 \times 1024$	8 bit	2.15
Hairball	$1024 \times 1024 \times 1024$	8 bit	1.07

Table 2: Volume datasets used for evaluation.

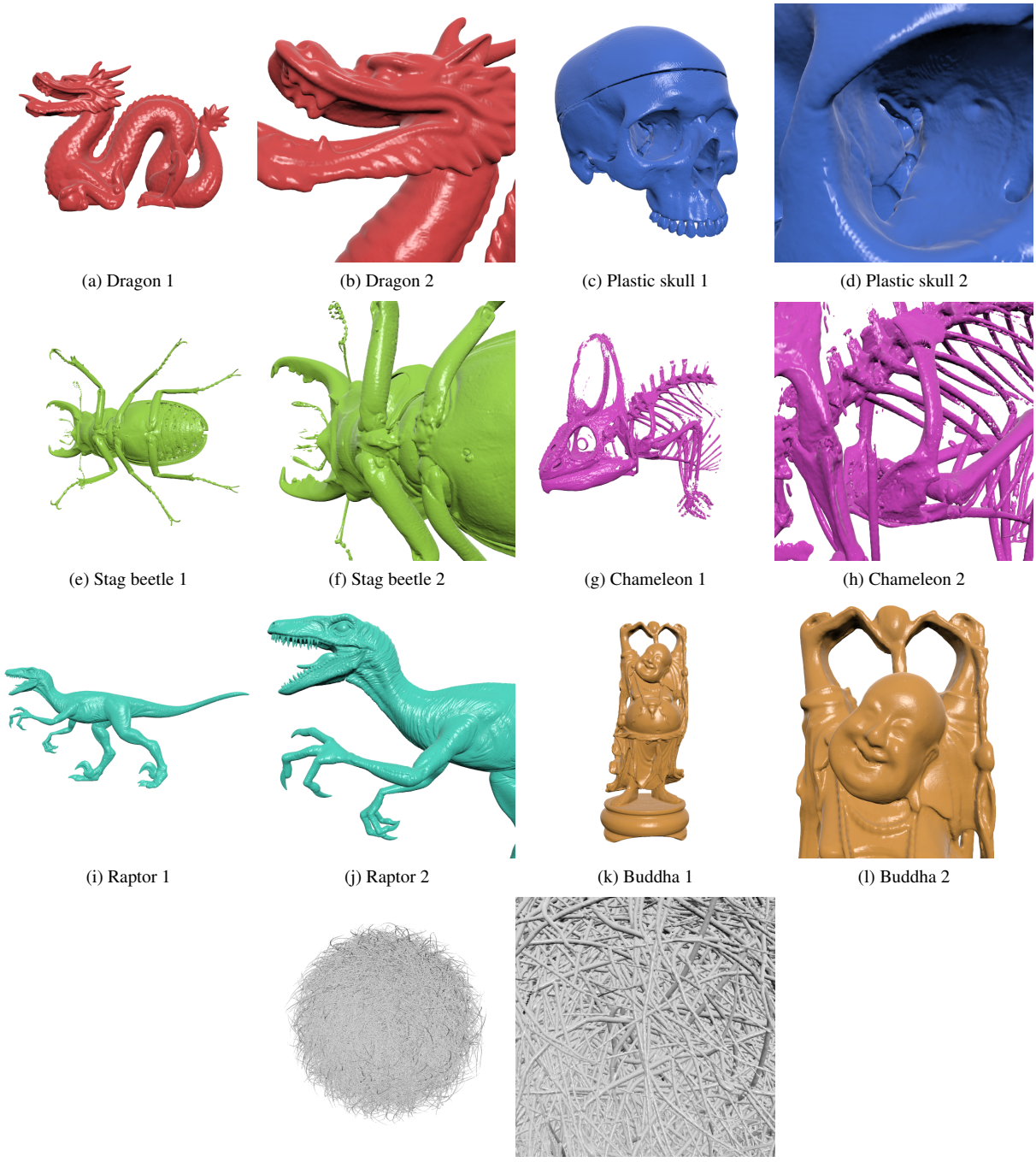
3.6 Hybrid Ray Tracing and Path Tracing

The data structure presented in this paper can also be used directly for ray tracing or path tracing of the extracted isosurface. Using the pre-computed cluster bound stored in the last metacell of each cluster, intersections with rays can be efficiently computed hierarchicaly by traversing and testing blocks, clusters, metacells, and individual voxels with the same efficient slab test used in the cell rasterisation. Shadow and ambient occlusion rays only need to fetch the metacell data, whereas additional bounces for path tracing also need to fetch grid cell data in order to compute a normal at each intersection point. Primary rays can be either traced or rasterised (hybrid ray tracing). Results from a path tracing implementation using hybrid ray tracing are shown in Figures 1 and 6.

4 EXPERIMENTS AND RESULTS

We implemented our method in OpenGL 4.5 and C++. All experiments were performed on an AMD Ryzen 7 2700 with 32GB RAM and an NVIDIA GeForce GTX 1070, and on a separate laptop with an Intel Core i7-6700HQ with 16GB RAM and an NVIDIA GeForce GTX 965M to evaluate performance on a lower-end GPU. Frame times were measured via OpenGL timer queries (`GL_ARB_timer_query`) for the scenes in Figure 5 rendered at 1920×1080 resolution. To generate the non-CT volume datasets in Table 2 for the test scenes, a CPU-based implementation of the slicemap-based binary voxelisation method of Eisemann et al. [3] was used. Each input mesh was voxelised to three times a target resolution, followed by downsampling to generate a coverage representation.

A comparison of memory usage and extraction times for our data structure (CGC) and a typical efficient mesh-based representation extracted with MC is presented in Table 3. The size of the original volume data is also presented in Table 2. For the MC implementation, we used the code from [2] after modifying it to generate indexed meshes without duplicate vertices, and store vertex positions in 16-bit signed normalized format with a scaling factor. While the MC memory usage could be further reduced using compression or converting indexed meshes into triangle strips, such optimization would require further mesh processing and increased extraction time. A performance comparison of MC, original cell rasterisation (CR) [13] using sampling in the original volume data, and our cell rasterisation using the CGC data structure is presented in Table 4. A corresponding isosurface comparison is also shown in Figure 4.



(a) Dragon 1 (b) Dragon 2 (c) Plastic skull 1 (d) Plastic skull 2

(e) Stag beetle 1 (f) Stag beetle 2 (g) Chameleon 1 (h) Chameleon 2

(i) Raptor 1 (j) Raptor 2 (k) Buddha 1 (l) Buddha 2

(m) Hairball 1 (n) Hairball 2

Figure 5: Test scenes for the datasets in Table 2.

5 DISCUSSION

In this paper, the clustered grid cell (CGC) data structure for isosurface rendering was proposed. We demonstrated that this data structure can be used for fast cell rasterisation via billboard splatting, as well as for hybrid ray tracing of isosurfaces extracted from large volumes. The data structure does not require the original volume data to be available during rendering, and uses about 37% memory compared to a typical mesh-based representation.

The cell rasterisation performance of our method was comparable to mesh rendering, except for the larger test scenes with high depth complexity, in which our method was faster and benefited from the visibility culling. Good use cases for our method would include digital sculpting and visualization of data that need stor-

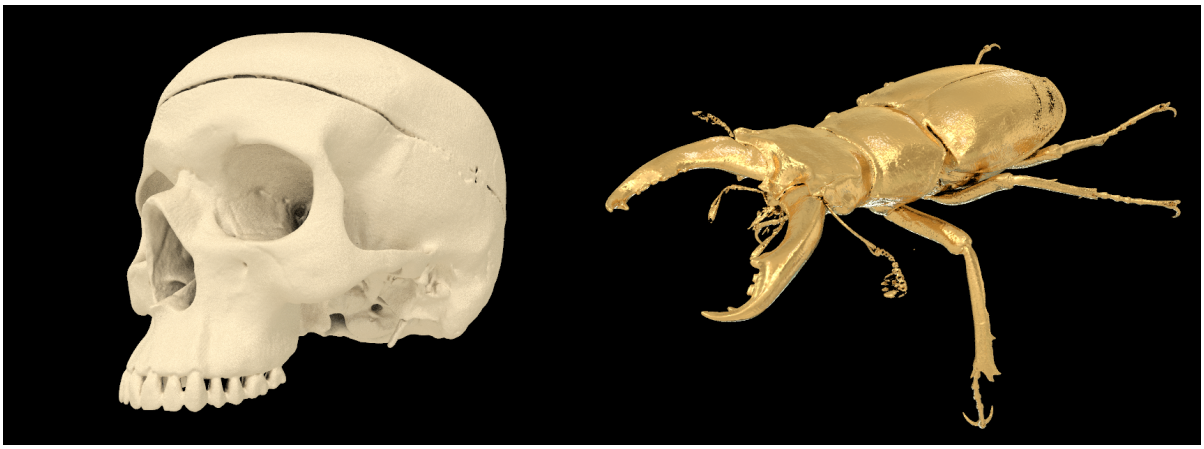


Figure 6: Hybrid ray tracing: Plastic skull dataset (512^3 voxels) and Stag beetle CT dataset ($834 \times 834 \times 494$ voxels) rendered with our data structure using cell rasterisation for primary rays and path tracing with three bounces and an HDR environment map for secondary rays (using 1 sample per pixel). Both scenes are rendered at interactive rate (less than 16 milliseconds per frame) at 1920×1080 resolution, with images showing result after 100 frames.

	CGC (ours)				MC [15]			
Volume	**Active cells** (count)	**Metacells** (count)	**Memory** (MB)	**Extraction** (time, s)	**Vertices** (count)	**Triangles** (count)	**Memory** (MB)	**Extraction** (time, s)
Dragon	668,804	182,400	7.5	0.8	668,860	1,337,714	20.1	1.5
Plastic skull	2,107,816	569,888	23.7	0.9	2,108,153	4,216,444	63.2	1.7
Stag beetle CT	3,140,758	769,536	59.5	2.6*	3,045,308	6,099,032	91.5	4.1*
Chameleon CT	3,492,569	911,088	38.9	11.4	3,430,406	6,861,504	102.9	12.4
Raptor	2,864,152	765,136	32.1	10.7	2,864,515	5,729,028	85.9	11.1
Buddha	8,464,696	2,264,208	94.9	17.6	8,464,912	16,929,714	253.9	24.3
Hairball	36,034,268	8,952,736	395.7	8.3	36,585,310	73,442,740	1,100.8	16.8

Table 3: Memory usage and isosurface extraction times for the datasets in Table 2, for our CGC data structure using the cell format in Table 1, and with corresponding MC indexed triangle meshes extracted and shown for comparison. For MC meshes, memory usage is without vertex normals and with indices stored using 4 bytes per index and vertex positions stored in 16-bit signed normalized format (with scaling factor computed from mesh bounds) using 6 bytes per vertex. The same normalized isovalue 0.5 was used for all datasets.

ing additional attributes per voxel or cell. A limitation of our method is the need to use deferred rendering for the EWA-splatting of normals and other attributes. However, as we demonstrate, the data structure can also be used for cell rasterisation with sampling in the original volume data.

It could be interesting to explore approaches using compute-based rasterisation for far grid cells and traditional rasterisation for near grid cells. We also aim to investigate if clusters could be more efficiently rendered with mesh shaders that were recently introduced with the NVIDIA Turing GPU architecture. Other future work could be improving ray tracing performance by re-arranging the clusters of min-max blocks into bounding volume hierarchies.

The source code for our implementation is available at `https://bitbucket.org/FredrikNysjo/grid_cells`.

ACKNOWLEDGEMENTS

The author would like to thank Filip Malmberg and Ingela Nyström for valuable input on the manuscript and the method. Further thanks to Johan Nysjö and Ingrid Carlbom for comments and discussion, and to the anonymous reviewers for their feedback. The STL mesh for the plastic skull dataset was obtained from a CT scan provided by Uppsala University hospital. The Dragon, Buddha, and Hairball models were downloaded from Morgan McGuire's Computer Graphics Archive [18], whereas the Raptor model was downloaded from the AIM@Shape Shape Repository (`http://visionair.ge.imati.cnr.it/ontologies/shapes`). Remaining datasets were obtained from `https://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle` and the Digital Morphology library (`http://digimorph.org`).

Scene	MC [15]		CR [13]		CGC (ours)	
	GTX 1070 (time, ms)	**GTX 965M** (time, ms)	**GTX 1070** (time, ms)	**GTX 965M** (time, ms)	**GTX 1070** (time, ms)	**GTX 965M** (time, ms)
Dragon 1	0.3	0.8	0.4	1.2	0.6 (0.4)	1.9 (1.1)
Dragon 2	0.3	0.9	0.6	1.7	0.7 (0.5)	2.2 (1.3)
Plastic skull 1	0.9	2.5	0.7	1.9	1.0 (0.6)	2.8 (1.7)
Plastic skull 2	1.0	3.0	1.2	3.3	1.5 (0.9)	4.0 (2.5)
Stag beetle 1	1.4	3.9	0.9	2.5	1.2 (0.9)	3.5 (2.5)
Stag beetle 2	1.8	6.5	1.5	4.4	1.6 (1.1)	4.9 (3.3)
Chameleon 1	1.5	4.3	1.4	3.5	2.0 (1.4)	5.2 (3.6)
Chameleon 2	1.5	3.6	1.3	3.6	1.6 (1.0)	4.9 (3.0)
Raptor 1	1.2	3.6	1.4	4.1	2.1 (1.5)	6.3 (4.4)
Raptor 2	1.2	3.0	1.0	2.8	1.5 (0.9)	4.5 (2.7)
Buddha 1	3.9	11.3	4.5	12.1	7.0 (4.9)	17.9 (12.4)
Buddha 2	4.0	9.2	1.7	4.5	2.5 (1.7)	7.0 (4.6)
Hairball 1	25.9	61.5	6.3	18.4	8.5 (6.2)	24.3 (18.4)
Hairball 2	26.5	66.4	6.2	18.0	6.7 (5.1)	19.3 (15.4)

Table 4: Performance comparison when rendering the scenes in Figure 5 at 1920×1080 pixels render target resolution on two different GPUs (GTX 1070 and GTX 965M), using Marching Cubes (MC), cell rasterisation with sampling in the original volume data (CR [13]), and cell rasterisation of our data structure (CGC). For the CR implementation, the visibility buffer and cluster sorting of our method is used instead of the original data structure. The times show GPU timings in milliseconds of the rasterisation time (depth and G-buffer pass). Rasterisation time for our method without EWA-splatting is also presented in the parentheses.

6 REFERENCES

[1] BOTSCH, M., HORNUNG, A., ZWICKER, M., AND KOBBELT, L. High-Quality Surface Splatting on Today's GPUs. In *Proceedings of the Second Eurographics / IEEE VGTC Conference on Point-Based Graphics* (2005), SPBG'05, Eurographics Association, pp. 17–24.

[2] BOURKE, P. Polygonising a scalar field. `http://paulbourke.net/geometry/polygonise`. Accessed on January 1, 2020.

[3] EISEMANN, E., AND DÉCORET, X. Single-Pass GPU Solid Voxelization for Real-Time Applications. In *Proceedings of Graphics Interface 2008* (2008), pp. 73–80.

[4] EVANS, A. Learning from Failure: a Survey of Promising, Unconventional and Mostly Abandoned Renderers for 'Dreams PS4', a Geometrically Dense, Painterly UGC Game. Presented at the 'Advances in Real-Time Rendering Course' at ACM SIGGRAPH '15 (2015).

[5] GREENE, N., KASS, M., AND MILLER, G. Hierarchical z-buffer visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (1993), ACM, pp. 231–238.

[6] GUENNEBAUD, G., BARTHE, L., AND PAULIN, M. Deferred Splatting. *Computer Graphics Forum 23*, 3 (2004), 653–660.

[7] HADWIGER, M., SIGG, C., SCHARSACH, H., BÜHLER, K., AND GROSS, M. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. In *Computer Graphics Forum* (2005), vol. 24, pp. 303–312.

[8] HEITZ, E., AND NEYRET, F. Representing Appearance and Pre-Filtering Subpixel Data in Sparse Voxel Octrees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics* (2012), pp. 125–134.

[9] HOETZLEIN, R. K. GVDB: Raytracing Sparse Voxel Database Structures on the GPU. In *Proceedings of High Performance Graphics* (2016), pp. 109–117.

[10] JABLONSKI, S., AND MARTYN, T. Real-Time Voxel Rendering Algorithm based on Screen Space Billboard Voxel Buffer with Sparse Lookup Textures. In *Proceedings of the 24th Int. Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (2016), WSCG'2016, pp. 27–36.

[11] JU, T., LOSASSO, F., SCHAEFER, S., AND WARREN, J. Dual Contouring of Hermite Data. In *ACM transactions on graphics (TOG)* (2002), vol. 21, pp. 339–346.

[12] LAINE, S., AND KARRAS, T. Efficient Sparse Voxel Octrees - Analysis, Extensions, and Implementation. Tech. rep., NVIDIA Research, 2010.

[13] LIU, B., CLAPWORTHY, G. J., AND DONG, F. Fast Isosurface Rendering on a GPU by Cell

Rasterization. *Computer Graphics Forum 28*, 8 (2009), 2151–2164.

[14] LIVNAT, Y., AND TRICOCHE, X. Interactive Point-Based Isosurface Extraction. In *Proceedings of Visualization '04* (2004), pp. 457–464.

[15] LORENSEN, W. E., AND CLINE, H. E. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics 21*, 4 (1987), 163–169.

[16] MAJERCIK, A., CRASSIN, C., SHIRLEY, P., AND MCGUIRE, M. A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering. *Journal of Computer Graphics Techniques (JCGT) 7*, 3 (2018), 66–81.

[17] MARCUS, R. Level-of-Detail Independent Voxel-Based Surface Approximations. Tech. rep., Utrecht University, 2017.

[18] MCGUIRE, M. Computer graphics archive, July 2017. Accessed on January 1, 2020.

[19] MUSETH, K. VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Transactions on Graphics (TOG) 32*, 3 (2013), 27.

[20] PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. Interactive Ray Tracing for Isosurface Rendering. In *Proceedings of the Conference on Visualization'98* (1998), pp. 233–238.

[21] ZHANG, H., AND KAUFMAN, A. Interactive Point-based Isosurface Exploration and High-quality Rendering. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1267–1274.

[22] ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. Surface Splatting. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (2001), ACM SIGGRAPH '01, ACM, pp. 371–378.