RayCaching: Amortized Isosurface Rendering for Virtual Reality

F. Nysjö, F. Malmberg and I. Nyström

Centre for Image Analysis, Uppsala University, Uppsala, Sweden
{fredrik.nysjo, filip.malmberg, ingela.nystrhom}@it.uu.se

Abstract
Real-time virtual reality requires efficient rendering methods to deal with high-resolution stereoscopic displays and low latency head-tracking. Our proposed RayCaching method renders isosurfaces of large volume datasets by amortizing raycasting over several frames and caching primary rays as small bricks that can be efficiently rasterized. An occupancy map in form of a clipmap provides level of detail and ensures that only bricks corresponding to visible points on the isosurface are being cached and rendered. Hard shadows and ambient occlusion from secondary rays are also accumulated and stored in the cache. Our method supports real-time isosurface rendering with dynamic isovalue and allows stereoscopic visualization and exploration of large volume datasets at framerates suitable for virtual reality applications.

Keywords: ray tracing, rendering, point-based graphics, rendering, immersive VR, virtual environments

ACM CCS: Computing methodologies → Ray tracing; Visibility; Point-based models; Virtual reality

1. Introduction

Isosurfaces in sampled and procedural data are encountered in applications in many areas, including scientific visualization, constructive solid geometry (CSG), computer animation and games. Efficient real-time rendering of isosurfaces in large volume datasets can be a challenge, especially for virtual reality (VR) applications in which high framerates and low latency need to be maintained. In medical visualization, a user may want to explore a dataset such as a computed tomography (CT) image with dynamic isovalue, which also requires the isosurface rendering to be able to quickly update the isosurface when the isovalue changes. Global illumination effects, such as shadows and ambient occlusion (AO), provide important depth cues that should also be incorporated in the rendering.

Indirect volume rendering techniques can provide efficient rendering of isosurfaces in static volume data and in data in which the complete isosurface does not have to be recomputed in every frame. Polygonization algorithms, such as Marching Cubes [LC87][NY06], extract the isosurface in form of a triangle mesh that can be efficiently rasterized. Point-based methods, such as surface splatting [ZPVBG01], also use rastertization, but extract the isosurface as a point cloud instead of a polygonal mesh; each point typically represents a disc-shaped surface element (surfel) [PZVBG00] that can be rendered using, for example, elliptical-weighted-average (EWA)-splatting [BHJK05] that provides smooth interpolation of normals and other surface attributes. However, for dynamic isovalue, indirect volume rendering often requires the complete isosurface to be recomputed when the isovalue changes, which can be expensive.

Direct volume rendering techniques avoid storing an explicit representation of the isosurface and can therefore be more suitable for dynamic volume data and dynamic isovalue. Image-order methods, such as GPU-based isosurface raycasting [KW03], perform ray-marching in the volume to render the isosurface; when distance values are available, sphere tracing [Har96] can be used for a more efficient ray-marching. Object-order methods, such as texture mapping [EHK*06], are also available. However, direct volume rendering typically depends on fast acceleration structures for empty space skipping, which need to be updated when the volume data or the isovalue changes. For VR, the cost of raycasting every pixel can be prohibitive, and for stereoscopic rendering, there is little exploitation of spatial or temporal coherence between the eyes, which can lead to additional expensive sampling of the volume data.

In this paper, we propose an isosurface rendering method, RayCaching, that combines some of the best features of indirect volume rendering with direct volume rendering. Our technical contributions described in this paper can be summarized as follows:
The proposed isosurface rendering method, RayCaching, where we use an occupancy clipmap to cache primary rays from volume raycasting as bricks in a point cloud that can be efficiently rasterized by splatting (Figure 1). The brick cache is reused between frames, such that the computational cost of the raycasting is amortized (distributed) over several frames. Our method provides level of detail (LOD) and allows stereoscopic visualization and exploration of large volume datasets with dynamic isovalue.

- The extension of our method to compute raycasted hard shadows and AO for cached bricks.

2. Related Work

The GigaVoxels method by Crassin et al. [CNLE09] uses ray-guided streaming and an octree of pre-filtered bricks, a so-called brick-map, for isosurface raycasting. The larger bricks used in GigaVoxels are updated in object-order and raycasted, whereas our smaller bricks are updated in image-order and rasterized. In contrast to GigaVoxels and similar methods [LK10] allowing out-of-core rendering of very large datasets (up to at least 8192^3 voxels), our method is currently restricted to volume datasets that fit in the GPU memory. Liu et al. [LCD09] rasterize pre-classified grid cells and compute high-quality normals by lookup in the original volume data. Their method allows dynamic isovalue and transparency, but is restricted to static grid-sampled volume data and does not provide LOD. The IsoBAS method by the same authors [LCD15] shares similar restrictions, but is one of the fastest isosurface rendering methods available today. Other direct isosurface rendering methods, using empty-space skipping and brick caching, have been proposed by, for example, Parker et al. [PSL*98] and Hadwiger et al. [HSS*05]. Wang et al. [WYC15] show how exploiting texture caching and mapping multiple GPU threads (warps or wavefronts) to single rays can improve the raycasting performance. Guthie et al. [GG16] combine warp-based direct volume rendering with data compression to render large datasets. Savchuk et al. [SB18] propose an isosurface raycasting method for VR in which they in a first pass perform raycasting at a coarser pixel resolution and in a second pass either interpolate colour values from the previous pass or perform additional raycasting for classified pixels in order to generate the final target resolution image.

Fast indirect volume rendering of large datasets with dynamic isovalue has also been proposed. Parallel Marching Blocks by Liu et al. [LCDW16] can extract indexed MC triangle meshes for large datasets (2048^3 x 4096 voxels) at interactive rates with dynamic isovalue but without LOD. Scholz et al. [SBD15] amortize the computational cost of generating a triangle mesh over several frames, such that the complete isosurface can be updated in a few seconds; similar to our method, their method provides view-dependent LOD and does not require any pre-processing of the volume data.

Whereas methods, such as GigaVoxels and sparse-voxel octrees [LK10], typically use an octree data structure to accelerate raycasting and to provide LOD, clipmaps have been used for similar purposes, for example, in large-scale terrain rendering [TMJ98] and for computing global illumination [Wri15]. Yusov [Yus16] uses clipmaps to provide LOD for particle-based cloud rendering using raycasting and pre-computed scattering.

The render cache method by Walter et al. [WDP99] addresses the problem of decoupling the rendering sampling rate from the rendering framerate. Shaded samples from raycasting are added as 3D points to a fixed-size point cache and rendered with sample interpolation for hole filling. The method was originally implemented in software and later extended to the GPU [ZWL05][VLBW06]. Our method shares similarities with the render cache, but stores bricks with attributes instead of individual shaded samples, and uses a clipmap to handle space occupancy and LOD instead of maintaining a one-to-one mapping between pixels and cached points. Also, our method is aimed at isosurface rendering and not raytracing or path tracing of polygonal scenes. Smit et al. [SVLBF10] propose an approach similar to the render cache, but for isosurface rendering in VR.

While the rasterization part of our method typically exhibits low depth complexity (has little overdraw), the rasterization of many small polygons, such as points and micro-polygons, can lead to bad GPU utilization and therefore become a potential
bottleneck \cite{FBH10}. Evans \cite{Eva15} presents a novel splatting method in which a compute shader and 64-bit atomics replace the traditional rasterization pipeline when splatting highly dense point clouds extracted from signed distance fields. The method allows splatting of billions of points per second on a standard PlayStation 4 game console, but requires hardware support for 64-bit atomics and uses deferred shading and temporal anti-aliasing that could make the method less suitable for VR.

3. Methods

In Sections 3.1–3.7, the proposed method is described in detail. Our method, outlined in Figure 2, exploits spatial and temporal coherency by caching intersections from primary rays as bricks in a point cloud that is rasterized by splatting. Each brick corresponds to a binary voxel in an occupancy clipmap used to avoid redundant caching. The clipmap is centred around the viewer and provides LOD so that each brick voxel obtains (approximately) the same projected size, as illustrated in Figure 3. Cached bricks are stored in a circular buffer and updated by sampling in the original volume data. Empty or no longer visible bricks are automatically evicted from the cache. In Section 3.8, we extend our method to compute and accumulate raycasted hard shadows and AO for cached bricks.

3.1. Amortized isosurface raycasting

To amortize or distribute the computational cost of isosurface raycasting over several frames, we divide the viewport into $16 \times 16$ tiles and cast a fixed number of primary rays for each tile per frame and eye. For each tile, we store and update a ray counter used to select 2D ray starting points within the tile from a low-discrepancy Halton sequence \cite{Hal64}. For each ray, we compute 3D entry and exit points on the volume bounding geometry, and ray-march the volume until the first isosurface intersection along the ray is found. Once an intersection is found, we perform a few steps of interval refinement by bisection to compute the final intersection position, and update the timestamp for that location to the current frame index. The intersection point is then tested against the occupancy clipmap, and cached as a brick in our brick cache (Figure 4). The timestamp is always updated when an intersection is found, even when the previous clipmap test fails. Timestamps are stored in a separate clipmap (Section 3.3).

3.2. Brick cache

The brick cache stores bricks of size $2 \times 2 \times 2$ brick voxels (Figure 5) in a circular buffer managed on the GPU. Each time an intersection from the raycasting passes the clipmap test (Figure 4), a new brick is inserted at the back of the result buffer that stores bricks to be processed. For each of the eight brick voxels in a brick, we sample the original volume data, and compute and store...
Table 1: Memory layout for 128-bit brick header.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position (XYZ)</td>
<td>96 bit</td>
</tr>
<tr>
<td>LOD</td>
<td>4 bit</td>
</tr>
<tr>
<td>Accumulation frame</td>
<td>4 bit</td>
</tr>
<tr>
<td>Brick voxel bitmask</td>
<td>8 bit</td>
</tr>
<tr>
<td>Hidden flag</td>
<td>1 bit</td>
</tr>
<tr>
<td>(Unused bits)</td>
<td>15 bit</td>
</tr>
</tbody>
</table>

Table 2: Memory layout for 32-bit brick voxel.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed distance</td>
<td>8 bit</td>
</tr>
<tr>
<td>Normal (θφ)</td>
<td>16 bit</td>
</tr>
<tr>
<td>Opacity</td>
<td>8 bit</td>
</tr>
</tbody>
</table>

The brick is then copied from the result buffer to the back of the brick cache. Bricks at the front of the cache are also selected and removed in each frame, to be updated and re-inserted at the back. Bricks determined to be empty or no longer visible based on their opacity values and a timestamp comparison with the current frame index are evicted from the cache. The memory layouts of the brick header and brick voxels are shown in Tables 1 and 2.

To compute the brick voxels, we divide each brick voxel into a $3 \times 3 \times 3$ sub-grid and sample the original volume data at those grid points using trilinear interpolation. For normal and signed distance to the isosurface, we compute weighted averages of the image gradients and positions of non-empty grid cells. For opacity, we use extinction-based downsampling [KB08] with non-empty grid cells considered opaque, and weight the extinction values so that grid cells with normal facing away from the vector between the grid cell and the clipmap centre obtain a lower extinction value.

In our implementation, bricks are stored in a single storage buffer object (GL_ARB_shader_storage_buffer_object). A buffer counter and offset to the first brick in the buffer is stored at the beginning of the buffer. The capacity of the brick cache should be set to provide a sufficient number of bricks to fully cover the visible isosurface; in our current implementation, we empirically set the capacity to one million bricks, which was found sufficient for the datasets we tested.

3.3. Occupancy clipmap

To avoid caching redundant bricks, we use an occupancy clipmap to test and store occupancy for bricks. Clipmaps [TMJ98] are similar to mipmaps, but use nested levels of identical resolution (Figure 6). Each voxel in our occupancy clipmap stores an initially empty sub-voxel bitmask; the bitmask indicates whether the space of a subvoxel is occupied by a brick. During raycasting, intersections are queried
against the clipmap; if the space is occupied, then the intersection is discarded, otherwise a brick is inserted into the cache and the subvoxel of that brick is set to one.

Clipmap voxels are stored as either 32-bit or 64-bit integers, and accessed and updated by atomic GPU integer instructions. In our implementation, we use clipmaps with eight levels and base resolution 128$^3$ voxels, and subvoxel bitmasks up to 4$^3$ bits. Each clipmap level thus requires 8–16 MB of storage, and the effective clipmap resolution becomes 128$^3$–512$^3$ binary voxels depending on the mask size. Timestamps are stored in a separate clipmap of coarser resolution (64$^3$ voxels), such that neighbouring bricks share timestamp.

When the camera moves too far from the clipmap centre, the clipmap needs to be re-centred (Figure 7). Before re-centring, cached bricks clear the bits for their clipmap subvoxels; after re-centring, bricks are tested against the (now empty) clipmap, to determine whether they should be re-inserted into the cache. Scattering of brick voxels is used to create new bricks when bricks move from a higher LOD to a lower LOD. To avoid the need to use gathering to create new bricks when bricks move from a lower LOD to a higher LOD, during the raycasting, we assign new intersections a probability to generate hidden bricks inside the inner 1/8th of a clipmap LOD, so that those hidden bricks are available and can be used during re-centring.

The point cloud we obtain by using an occupancy clipmap and a timestamp clipmap corresponds to the minimal visible set of points needed for rendering the current view (Figure 3).

3.4. Importance sampling

To improve the convergence rate of the isosurface raycasting, we use an importance sampling scheme that keeps track of which viewport tiles that should be importance sampled in a frame. After regular non-importance rays are generated for each viewport tile in a frame, tiles with new cached intersections are appended to a circular buffer. In a second raycasting pass, each viewport tile is mapped on a tile in the circular buffer, so that the tiles in the buffer are sampled by more rays. Similarly to Walter et al. [WDP99], we split the number of rays cast per frame, so that half of our rays become importance rays. Examples of the importance sampling are shown in Section 4.

3.5. Predictive sampling

In addition to importance sampling, we use predictive sampling to fill in occluded areas before they become visible in the current view. Our predictive sampling estimates a new camera or head pose to be used during the raycasting step; the pose is predicted a few frames ahead, from the current and previous pose, and does not need to be accurate; in our experiments, we found prediction of 10–15 frames to be sufficient. During other steps of the rendering such as splatting, the original camera or head pose is used.

Certain VR application programming interfaces, such as the Valve OpenVR SDK [Val], can provide a more accurate predicted head pose estimated from sensor data and the head-mounted display’s (HMD’s) inertial measurement unit, which can be used for better prediction sampling when rendering for VR.

3.6. Brick rasterization

Forward shading is generally recommended for the current generation of HMDs, such as the Oculus Rift and the HTC Vive, because it can be combined with multi-sampled anti-aliasing (MSAA) that removes otherwise distracting aliasing from the enlarged pixels of the HMD. Methods, such as EWA splatting [BHZK05] and direct isosurface raycasting, often use (or require) deferred shading, which is difficult to combine with MSAA and therefore requires a screen-space anti-aliasing solution, such as fast-approximate anti-aliasing (FXAA) [Lot09], which can be less suitable for VR because of shimmering pixels and other image artefacts.

Our method uses forward shading, and renders the isosurface by rasterizing brick voxels as billboarded point sprites (GL_POINTS). Points for brick voxels are drawn with an indirect draw command (glDrawElementsIndirect) to avoid potential synchronization overhead from having to read back the current draw count from the GPU. Each point is splatted as an anti-aliased disc that can be rendered using either alpha blending or alpha-to-coverage. For disc splatting, we use the affine approximation from [BK03].

The brick cache only contains the bricks needed for rendering the opaque isosurface (the closest hits along the rays), but we use opacity values for compositing and anti-aliasing of brick voxels. For rendering with alpha blending, bricks need to be sorted in back-to-front order to avoid flickering and incorrect blend-order. While techniques, such as an A-buffer [Car84] [LHL14] and weighted order-independent blending [MB13], could be used to avoid sorting, we use a fast approximate sorting scheme that outputs an index buffer with bricks sorted according to which binary voxel layer in the clipmap they map to. The clipmap is divided into ‘onion layers’, with each layer corresponding to a layer of binary voxels in the clipmap. Sorting is performed by binning bricks into the layers, and computing prefix sums of the bins. To provide consistent blend-order within layers, layers are further divided into four sub-bins so that adjacent bricks map to different sub-bins. In our implementation, we use a parallel prefix sum, similar to the one presented in [ABOS09], to compute the prefix sums of the bins. Hidden bricks are skipped during sorting to avoid them being rendered. The resulting index buffer allows the bricks to be traversed in approximately sorted order. For rendering with alpha-to-coverage, the sorting step can be omitted at the cost of slightly lower image quality compared to alpha blending.

An image quality comparison between direct isosurface raycasting and our method with disc splatting is shown in Figure 8. As an alternative rastertization method, we also tried splatting the brick voxels with local ray-marching in the original volume data, using a fixed number of small steps to find the isosurface inside the brick voxel. Similarly to Liu et al. [LCD09], the original volume data can be used to compute a high-quality normal for each fragment. In our experiments, we found three ray-marching steps to give a sufficiently good result. However, raycasting code and resource usage need to be duplicated to the splatting shader, and local ray-marching will be less efficient than anti-aliased disc splatting.
3.7. Stereoscopic rendering

To enable stereoscopic viewing, we divide the raycasting in each frame between the left and right eyes, so that equal number of rays are cast for each eye. Bricks from intersections are stored in a single cache for both eyes. The occupancy clipmap prevents us from caching redundant bricks when an isosurface point is visible from more than one eye. To render the left and right eyes, we rasterize the bricks twice, without reusing any shading information between the eyes. Occluded isosurface areas that become visible from only one eye will appear as holes until a sufficient number of rays have been cast for that eye to populate the brick cache. Because brick voxels are rasterized as projected discs, there should be no other discrepancies between the eyes except possible from face culling at grazing view angles.

3.8. Hard shadows and AO

Global information, such as hard shadows and AO, provides important depth cues. Shadow mapping would be the default choice for computing hard shadows for isosurfaces, but has the drawback that it requires the isosurface to be drawn from the light’s point of view, which is generally not possible with the information in our brick cache. Similarly, screen-space AO methods are widely used in real-time rendering, but can provide inconsistent results between the eyes in VR. For these reasons, we favour object space methods that sample the original volume data.

To compute raycasted hard shadows and AO, we cast and accumulate secondary rays each time a brick voxel in the cache is being updated. For each $2 \times 2 \times 2$ brick, we store and update an accumulation frame counter that is used with a weighted moving average filter to accumulate new samples. For hard shadows, we cast a single long ray in the direction of the light source and jitter the ray origin in each frame. For AO, we cast four shorter rays in random directions on a sphere generated from a Halton sequence, using the same ray directions for every brick voxel in a frame to improve coherence. We take up to 100 steps for shadow rays and 10 steps for AO rays, and compute visibility in the range $[0,1]$. Visibility is accumulated similar to in Mattausch et al. [MSW11], but we clamp our weighted moving average filters sooner, after only 4–8 frames.
When having a point cloud representation of a scene, AO could alternatively be computed directly from the surfels of the point cloud [Chr08]. However, since our brick cache currently only stores information about the isosurface visible from the viewer’s point of view, we have not further explored this approach.

4. Results

We implemented our method in OpenGL 4.5 and C++11. Most of the experiments were conducted on an Intel Core i5-3570K with 16GB RAM and an NVIDIA GeForce GTX 1070; a separate laptop with an Intel Core i7-6700HQ with 16GB RAM, an NVIDIA GeForce GTX 965M and integrated Intel HD 530 graphics was also used to evaluate the performance on mid-end and low-end GPUs. We measured frame times, using OpenGL timer queries (GL_ARB_timer_query), and framerates for the test datasets in Table 3. Figures 1 and 9 show the CT datasets for which rendering with dynamic isovalue is of interest. Figure 10 shows the MANIX CT dataset rendered with hard shadows and AO, with frame timings presented in Tables 4 and 5. All outputs were rendered to a 16-bit...
Figure 11: Illustration of brick cache convergence and importance sampling behaviour of our method. First row: brick cache and rendered isosurface after N frames, starting from empty cache and using 32 rays per tile for raycasting. Second row: viewport tiles with colours indicating how many times (0–8) each tile is importance sampled in the frame. Third row: brick cache and rendered isosurface after N frames, when changing between two distinct isovalues and using 32 rays per tile for raycasting. Fourth row: viewport tiles with colours indicating importance sampling. The scene is rendered in 384³ clipmap resolution.

RGBA surface with 4×MSAA. For the RayCaching and direct isosurface raycasting methods, we used stochastic jittering and a step length of 0.004; no additional acceleration structures were used for any of the raycasting. OpenVR [Val] was used for the VR implementation. When rendering for an Oculus Rift HMD on the GTX 1070, we were able to render isosurfaces for our test datasets at the desired 90 fps target rate when using 32 rays per tile. VR rendering of the stag beetle dataset is shown in Figure 12. However, some problems with undersampling were observed in the most distant bricks, resulting in lower isosurface quality in those areas compared to the reference methods.

Table 4 shows GPU timings in milliseconds when rendering the scene in Figure 10 with our method in stereo at 1920 × 1080 resolution with 4 × MSAA. Thirty-two rays per viewport tile were used for raycasting and 8000 cached bricks were updated per frame. A lower 384³ clipmap resolution had to be used for the Intel GPU because of lack of support for 64-bit integer atomics.

Table 3: Volume datasets used for evaluation.

<table>
<thead>
<tr>
<th>Volume</th>
<th>Resolution (W × H × D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANIX</td>
<td>512 × 512 × 460</td>
</tr>
<tr>
<td>PELVIX</td>
<td>512 × 512 × 354</td>
</tr>
<tr>
<td>OBELIX</td>
<td>512 × 512 × 1558</td>
</tr>
<tr>
<td>Stag beetle</td>
<td>832 × 832 × 494</td>
</tr>
<tr>
<td>Armadillo</td>
<td>512 × 512 × 512</td>
</tr>
</tbody>
</table>

Table 5 shows GPU timings for the same scene with our method compared to reference implementations of Marching Cubes and direct isosurface raycasting, including raycasted shadows and AO computed from the volume. For Marching Cubes, an additional z-prepass was used to avoid computing shadow and AO rays for hidden surfaces. For direct isosurface raycasting, no anti-aliasing was used.

© 2019 The Authors Computer Graphics Forum © 2019 Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd
Figure 12: Stereoscopic VR rendering of stag beetle dataset (832 x 832 x 494 voxels) for an Oculus Rift HMD. The scene is rendered in 512³ clipmap resolution, and the number of non-empty brick voxels (splats) in the view is 409,594. Colours indicate clipmap LOD.

Table 4: Breakdown of the frame time of our method (RayCaching) when rendering the scene in Figure 10 (isosurface only) on three different GPUs.

<table>
<thead>
<tr>
<th>Operation</th>
<th>GTX 1070</th>
<th>GTX 965M</th>
<th>HD 530</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raycasting + bricks</td>
<td>0.8</td>
<td>2.7</td>
<td>8.8</td>
</tr>
<tr>
<td>Sorting</td>
<td>0.4</td>
<td>0.5</td>
<td>0.8</td>
</tr>
<tr>
<td>Splatting</td>
<td>0.8</td>
<td>1.6</td>
<td>4.3</td>
</tr>
<tr>
<td>Total</td>
<td>2.0</td>
<td>4.8</td>
<td>13.9</td>
</tr>
</tbody>
</table>

The timings show GPU time in milliseconds.

Table 5: Performance comparison when rendering the scene in Figure 10 on three different GPUs, with and without raycasted shadows and AO, with the methods Marching Cubes (MC) (4.76M triangles), direct isosurface raycasting (DIR) and RayCaching (ours).

<table>
<thead>
<tr>
<th>Operation</th>
<th>GTX 1070</th>
<th>GTX 965M</th>
<th>HD 530</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC (isosurface only)</td>
<td>5.0</td>
<td>10.4</td>
<td>39.5</td>
</tr>
<tr>
<td>MC (shadows + AO)</td>
<td>40.6</td>
<td>258.0</td>
<td>600.0</td>
</tr>
<tr>
<td>DIR (isosurface only)</td>
<td>5.7</td>
<td>16.0</td>
<td>62.0</td>
</tr>
<tr>
<td>DIR (shadows + AO)</td>
<td>14.7</td>
<td>89.9</td>
<td>143.8</td>
</tr>
<tr>
<td>Ours (isosurface only)</td>
<td>2.0</td>
<td>4.8</td>
<td>13.9</td>
</tr>
<tr>
<td>Ours (shadows + AO)</td>
<td>3.5</td>
<td>10.1</td>
<td>23.9</td>
</tr>
</tbody>
</table>

The timings show GPU time in milliseconds.

5. Conclusions and Future Work

In this paper, we proposed an isosurface rendering method, RayCaching, that allows stereoscopic rendering of large volume datasets with dynamic iso-value. Performance suggests that our method is suitable for the intended VR application. While our method is mainly compared against direct isosurface raycasting without acceleration structures for empty-space skipping, our method would also benefit if such acceleration structures were used. Compared to isosurface extraction and mesh rendering, our method requires the original volume data during rendering, but can be faster for primary rays and enables amortizing of secondary rays.

A limitation of our method is that dynamic transformations (translations or rotations) of volume objects inside the clipmap will invalidate the brick cache. If a unique occupancy clipmap is used per volume object, then such transforms should instead be expressed as a translation of the clipmap in order to avoid invalidating the cache. To address the observed undersampling issue, a possible future direction would be to explore temporal supersampling of brick voxels. We would also like to further explore the use of grid cell rasterization to improve the rendering quality: instead of sampling the original volume with trilinear interpolation in a fragment shader during rasterization, as in Liu et al. [LCD09], we could compute and store eight pre-filtered corner values inside each brick voxel for the sampling. Transparency is another important aspect of isosurface rendering that we have not addressed in this paper. By allowing primary rays failing the clipmap test to continue ray-marching inside the volume, we could obtain additional layers of points that could be used for transparency rendering. The number of transparent layers would be defined by the maximum number of clipmap tests along a ray. The approximate sorting scheme described in Section 3.6 would...
enable transparent layers to be blended in correct layer order, but
tickering might still occur if the splat radius is increased such that
splits from non-adjacent bricks within layers overlap.

We foresee medical applications for our method, for example,
to enable fast rendering of isosurfaces in virtual craniomaxillo-
facial surgery planning [NOM*K7, where large medical CT vol-
unes and CSG operations on signed distance fields need to be vi-
ualized and interacted with in stereoscopic 3D with head-tracking.
Source code for our method is available at https://bitbucket.org/
FredrikNysjo/raycaching.

Acknowledgements

We thank Ingrid Carlbom and Johan Nysjö for their comments
on the manuscript. We further would like to thank the anony-
mous reviewers for their comments and valuable input. The med-
cal CT datasets were obtained from the OsirIX DICOM Image
Library (www.osirix-viewer.com/resources/dicom-image-library/).
The stag beetle dataset was obtained from [GGK05], whereas the
Armadillo model was obtained from The Stanford 3D Scanning
Repository.

References

[ABOS09] Assarsson U., Billeter M., Olsson O., Sintorn E.: GPU
primitives – Case study: Hair rendering. In Presented at the
‘Beyond Programmable Shading Course’ at ACM SIGGRAPH
‘09 (2009).

[BUZH05] Botts M., Hornung A., Zwicker M., Kobelt L.: High-
quality surface splatting on today’s GPUs. In Proceedings of
the Second Eurographics/IEEE VGTC Conference on Point-
17–24.

[BC03] Botts M., Kobelt L.: High-quality point-based rendering
on modern GPUs. In Proceedings of the 11th Pacific Conference

[Carp84] Carpenter L.: The A-buffer, an anti-aliased hidden sur-
face method. In Proceedings of the 11th Annual Conference on
Computer Graphics and Interactive Techniques (1984), ACM
SIGGRAPH ’84, pp. 103–108.

[Car87] Christensen P. H.: Pixar Technical Memo 08-01: Point-

[CNLE09] Crassin C., Neyret F., LeFebvre S., Eisemann E.: Gi-
gaVoxels: Ray-guided streaming for efficient and detailed voxel

[EHK*06] Engel K., Hadwiger M., Kniss J. M., Rezk-Salama C.,

[Eva15] Evans A.: Learning from failure: A survey of promis-
ing, unconventional and mostly abandoned renderers for ‘Dreams
PS4’, a geometrically dense, painterly UGC game. In Presented
at the ‘Advances in Real-Time Rendering Course’ at ACM SIG-

[FBH*10] Fatahalian K., Boulos S., Hegarty J., Akeley K.,
Mark W. R., Moreton H., Hanrahan P.: Reducing shad-
ing on GPUs using quad-fragment merging. In Proceedings of the
37th International Conference and Exhibition on Computer Graphics and Interactive Techniques (2010), ACM
SIGGRAPH ’10.

[GG16] Guthe S., Goesele M.: Variable length coding for GPU-
based direct volume rendering. In Proceedings of Vision, Model-
ling, and Visualization 2016 (2016).

[GGK05] Gröller M. E., Ga¨leser G., Kastner J.: Stag bee-
ac.at/research/publications/2005/dataset-stagbeetle/.

[Hal64] Halton J. H.: Algorithm 247: Radical-inverse quasi-
random point sequence. Communications of the ACM 7, 12
(1964), 701–702.

[Har96] Hart J. C.: Sphere tracing: A geometric method for the
anti-aliased ray tracing of implicit surfaces. Visual Computer 12,
10 (1996), 527–545.

[HGS*05] Hadwiger M., Sigc C., Scharsach H., Bühler K.,
Gross M.: Real-time ray-casting and advanced shading of dis-
crete isosurfaces. In Computer Graphics Forum (2005), vol. 24,
pp. 303–312.

[KB08] Kraus M., Bürger K.: Interpolating and downsampling
RGBA volume data. In Vision, Modeling, and Visualization 2008:

GPU-based volume rendering. In Proceedings of the 14th IEEE
Visualization 2003 (VIS’03) (2003), IEEE Computer Society,
p. 38.

[LCS*07] Lorensen W. E., Cline H. E.: Marching cubes: A high res-
olution 3D surface construction algorithm. Computer Graphics

[LCD09] Liu B., Clapworthy G. J., Dong F.: Fast isosurface ren-
dering on a GPU by cell rasterization. Computer Graphics Forum
28, 8 (2009), 2151–2164.

[LCD15] Liu B., Clapworthy G. J., Dong F.: IsoBAS: A bi-
inary accelerating structure for fast isosurface rendering on
GPUs. Computers & Graphics 48, Supplement C (2015),
60–70.

marching blocks: A practical isosurfac ing algorithm for large
data on many-core architectures. In Computer Graphics Forum

[LLH*14] LeFebvre S., Hornus S., Lasram A.: Per-pixel lists for
single pass A-buffer. In GPU Pro 5: Advanced Rendering Tech-
Supporting Information

Additional supporting information may be found online in the Supporting Information section at the end of the article.

Video S1