Vision, Modeling, and Visualization (2015)
D. Bommes, T. Ritschel and T. Schultz (Eds.)

Vector-to-Closest-Point Octree for Surface Ray-Casting

I. Demir[1] and R. Westermann[1]

[1]Computer Graphics & Visualization Group, Technische Universität München, Germany

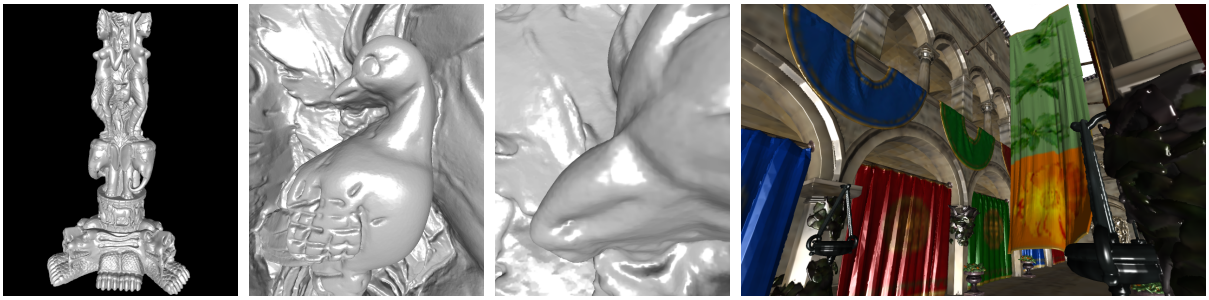

Figure 1: *Rendering from a hierarchical Vector-to-Closest-Point (VCP) grid at different zoom steps. Left: The Thai statue (10 million polygons) renders in less than 10ms per frame at a maximum grid resolution of* $2048 \times 1024 \times 1024$. *Right: For the Sponza model the hierarchy is 3 levels shallower than for a sparse voxel hierarchy in order to achieve roughly the same rendering quality (for the Thai statue the gain is between 1 (left) and 5 levels (right)).*

Abstract

GPU voxel-based surface ray-casting has positioned as an interesting alternative to rasterization-based rendering approaches, because it allows using many processing units simultaneously, can effectively exploit thread level parallelism, and enables fine-granularity occlusion culling on the pixel level. Yet voxel-based techniques face the problem that an extremely high resolution is necessary to avoid block artifacts at high zoom levels. In this work, we propose a novel improvement of voxel-based ray-casting to overcome this limitation. By using a hierarchical Vector-to-Closest-Point (VCP) representation, we can inherit the advantages of a voxel-based approach at a much smoother approximation of the surface. We demonstrate that, although the VCP grid consumes more memory per cell, it requires less memory overall, because it builds upon a significantly shallower tree hierarchy. In a number of examples we demonstrate the use of our approach for high-quality rendering of high resolution surface models.

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

1. Introduction and Related Work

Voxel-based surface ray-casting on the GPU [GMIG08, CNLE09, LK10a] has been introduced as an interesting alternative to rasterization-based polygon rendering and triangle ray-tracing. It inherits the benefits of ray-based GPU rendering techniques [PBMH02, CHH02, EVG04, FS05, CHCH06, GPSS07, PGSS07, HSHH07, AL09] to effectively exploit the GPU's massively parallel design and use many processing units simultaneously. In addition it can effectively perform fine-granularity occlusion culling on the ray level. Moreover, the regular voxel grid gives rise to efficient ray traversal schemes, and it allows generating levels of detail with prescribed (screen-space) error in a very simple and efficient way.

A limitation of voxel-based techniques is that they produce visual artifacts when the size of the voxels exceeds

the pixel size in screen space. In this case, the underlying voxel grid becomes visible as block structures. As a consequence, classical voxel-based techniques need to build very deep trees so that even for extreme zoom-ins the voxel size can still match the pixel size. It is worth noting that the high-resolution representation is also required in areas where the initial surface is smooth.

Different solutions have been proposed to remedy this problem. "Far Voxels" have been introduced as an efficient LOD structure for polygonal models by Gobbetti and Marton [GM05]. They render the polygon structure where the details are high, and render a voxel-based approximation of polygon clusters at coarser resolution levels. Laine et al. [LK10b] introduce a sparse voxel octree (SVO), where the voxels are supplemented by piecewise linear contour representations, which are then used during rendering to avoid block artifacts and generate smooth object silhouettes. Their method was later enhanced by Kämpe et al. [KSA13] using directed acyclic graphs (DAG) instead of octrees to reduce the memory consumption when encoding identical regions. Similar to "Far Voxels", Reichl et al. [RCBW12] employ a hybrid rendering method which combines rasterization and ray-casting. "Sphere Tracing" is a rendering technique proposed by Hart for implicit surfaces defined by signed distance functions [Har96], which was later improved upon by Keinert et al. [KSK*14]. Bastos et al. presented a method built on Sphere Tracing to efficiently render Adaptively Sampled Distance Fields (ADFs) on the GPU [BC08]. While their method is restricted to signed distance isosurfaces, our method is able to handle arbitrary surfaces.

A different approach has been proposed already 17 years ago by Gibson [Gib98], and used for voxel-based surface rendering on fixed-function graphics hardware by Westermann et al. [WSE99]. Gibson proposed to make use of a distance field to obtain a higher order interpolant of a level-set surface in an intensity volume. Westermann et al. employed this idea to voxelize a polygon surface into a 3D scalar distance field and render the implicit surface via slice-based volume rendering. For a thorough overview of algorithms for computing signed distance transforms from polygon models as well as a number of applications of such transforms, let us refer to [JBS06, FPRJ00]. In the context of fluid simulation, Auer et al proposed a rendering method for a uniform low-resolution closest-point simulation grid [AMT*12].

Contribution. In this work we extend on previous works and present a novel approach to overcome the limitations of voxel-based surface rendering. Inspired by the distance-to-closest-surface representation by Gibson [Gib98], we introduce the hierarchical *Vector-to-Closest-Point* representation (VCP) to improve the surface approximation quality of a grid-based representation. As demonstrated in Fig. 1, compared to a "standard" voxel hierarchy, the VCP representation results in a significantly shallower tree, and it reduces the memory consumption and tree traversal costs during rendering. The surface can be determined as the 0-level-set of the VCP function, and smooth normals can be computed by interpolating VCP vectors. Our rendering specific novel contributions are:

- The use of a hierarchical VCP grid for parallel ray-casting on the GPU.
- An efficient high-quality technique for intersecting a ray with the surface encoded in a VCP grid.
- A significantly shorter model hierarchy compared to a "classical" voxel representation.
- The option to render a smooth surface at arbitrary zoom-ins from a VCP grid.

2. VCP Representation

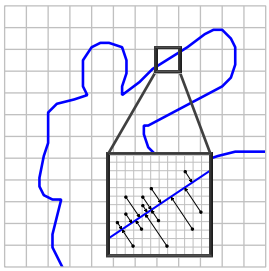

Figure 2: *Illustration of the VCP representation in 2D. At each grid point, a vector to the point closest to the original geometry (blue) is stored; this is visualized by a few such vectors (black). The grid is adaptively refined in the vicinity of the object.*

To begin with, let us consider a geometric object, represented by a polygonal surface, which is embedded in three-dimensional Cartesian space. The space is discretized via a uniform Cartesian grid. This is illustrated in Fig. 2 in two dimensions. Now we can compute for each grid point the VCP vector wrt. the given geometry, i.e., the vector pointing to the closest surface point. An algorithm to efficiently compute a VCP grid for very large polygon meshes is presented below. Note that storing the full grid at a high resolution would waste a significant amount of memory, since during rendering the VCP data is only required in a narrow ε-band around the object's surface. Therefore our approach uses an additional regular grid, which comprises *blocks* of 2^3 cubical cells. A VCP representation is computed only for those (non-empty) blocks which overlap the narrow band, while for all other (empty) blocks we store one single scalar value indicating the minimum distance of all vertices of this block to the surface. These values are used during ray-casting to adaptively vary the step-size, i.e., a step as large as the stored minimum distance can never cross the surface. We will later describe how to employ a hierarchical representation to adaptively prune empty space comprising multiple empty blocks.

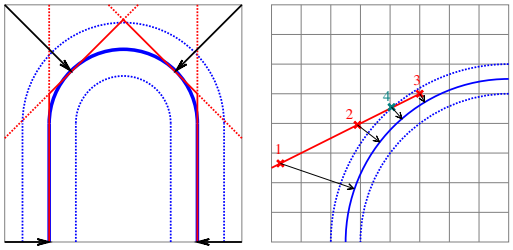

Figure 3: *Left: If only distance information is stored at the grid points, the feature (blue) is blurred or lost during linear interpolation since it is located within one grid cell. Our interpolation scheme based on the VCP representation (black arrows), however, conserves the feature to a better degree as illustrated by the red lines. Hence we can choose a smaller* ε*-band leading to more precise rendering results. Right: Ray-casting through the VCP grid. At each step (marks 1, 2, 3) along the ray (red) the VCP distance to the object (blue) is obtained. Once it is less than* ε*, i.e., the distance from the dotted blue curve to the feature, this is considered as a hit (mark 3). Interpolating between the current and the previous step results in a point on the ray with VCP distance of about* ε *(mark 4).*

In principle, to implicitly encode the surface one could also use a scalar distance field, which stores at every grid point the shortest distance to the surface. Such a representation, however, requires the use of signed distance values in order to accurately determine the distance-0 isosurface. Computing signed distances, on the other hand, is non-trivial and generally speaking not even possible, for instance if the surface is non-orientable or not closed. By contrast, our technique can be applied to arbitrary geometry. In addition, the distance values in a discrete grid approximate the surface with an error that is linear in the grid spacing, while each closest point defines the exact position of a surface point. Hence, a better approximation quality is achieved with a VCP representation, as illustrated in Fig. 3 (left).

2.1. Ray-Casting

The VCP grid is rendered via parallel GPU ray-casting. For each ray, we simultaneously compute the first and last intersection point with the object's bounding box, and we let the rays march through the VCP grid using varying step-sizes. This process is illustrated in Fig. 3 (right). At each sampling point, the allowed size of the next step is computed, either by reading the scalar distance value from the hit (empty) block, or by using the length of the interpolated VCP vector in case a non-empty block is hit. If this distance is less than ε, a ray-surface intersection point is assumed, otherwise the value is used as the step size for the next sampling point.

To determine the exact intersection point, the location between the current and the previous sampling point at which the distance is exactly ε is interpolated. By doing this for every ray, a smooth surface being a constant distance away from the exact surface is rendered. It is clear that by decreasing ε we can come arbitrarily close to this surface. Once a ray-surface intersection is found, a local lighting model using the surface normal vector at the intersection point is evaluated. The normal vector is given by the normalized VCP vector, since it is always perpendicular to the surface.

At this point, the procedure could be extended to compute shadows by sending another ray from the hit position to the light source. Likewise, additional non-camera rays could be sent from the hit point to render reflection and refraction effects. This can be implemented in a straightforward way, since our grid structure is independent of the camera position.

2.2. VCP Interpolation

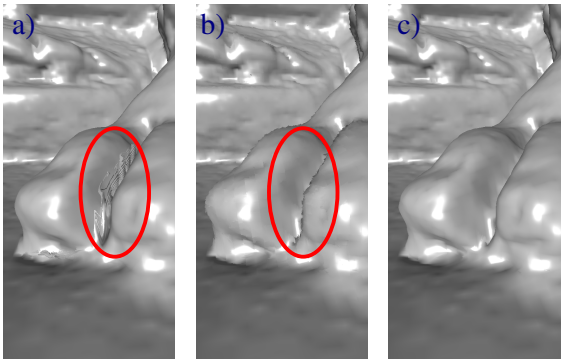

Figure 4: *Comparison of three interpolation schemes. a) Trilinear interpolation produces artifacts since adjacent VCP vectors pointing in opposite directions are canceled out. b) Our intermediary scheme also leads to artifacts since triangles are substituted by planes which have no boundary. c) Combining both methods avoids artifacts of either case and is computationally cheap.*

To obtain the VCP at an arbitrary location, we interpolate between the 8 given vectors at the vertices of the respective cell. For this, it is crucial to have an interpolation scheme which is consistent with the VCP geometry. A straight forward way would be to simply use trilinear interpolation. However, as shown in Fig. 4a, this method produces artifacts: Suppose we have VCP vectors $(-\varepsilon,0,0)$ and $(\varepsilon,0,0)$ at two adjacent grid points, meaning that they are pointing to regions that are close in space but distant wrt. the surface. Now linearly interpolating in the middle of these two points would yield the vector $(0,0,0)$ and hence result in an intersection point during ray-casting. To remedy this problem, we propose a specially designed interpolation scheme which fits the purpose of interpolating VCP data consistently. This method is illustrated in Fig. 3

(left). When interpolating at p' we iterate over all 8 grid points p_i and think of each VCP as describing a plane. More precisely, each vector vcp_i describes a (unique) plane P_i such that vcp_i starting from p_i is a perpendicular on P_i, i.e., $vcp_i \perp P_i$ and $p_i + vcp_i \in P_i$. We now obtain the interpolated VCP wrt. p_i as the perpendicular from p' to P_i, i.e., $vcp_i^\star = \left(\left(vcp_i \cdot \left(p_i + vcp_i - p'\right)\right) / \left(vcp_i \cdot vcp_i\right)\right) \cdot vcp_i$. Finally, the shortest vector is returned as the result, i.e., $vcp^\star = \arg\min_{vcp_i^\star} \{\|vcp_i^\star\|\}$. However this approach also leads to rendering artifacts as can be seen in Fig. 4b. This comes from the fact that by substituting planes for a mesh which originally consisted of triangles the boundaries of the triangles are lost. As a costly solution to this problem, one could not only store VCP vectors but additionally the triangles to which they refer. A much more feasible solution is to combine our method with the trilinear interpolation scheme such that the VCP with the greater length is returned as the final result. In this case artifacts of either case are avoided leading to a smooth rendering result. This is depicted in Fig. 4c.

3. VCP Octree

The concept underlying our construction of a hierarchical octree structure is illustrated in Fig. 5. The octree is built bottom-up from the VCP grid outlined before. Each block is initially considered as a leaf. For each block, we compute the minimum distance d_{min}, as either the minimum of all scalar per-vertex distances of empty blocks or the minimum of all VCP distances of the vertices of non-empty blocks. Whenever the minimum distance is greater than a certain threshold, i.e. $d_{min} > t$, we mark the corresponding block as empty. In this way we ensure that VCP information is only stored in the close vicinity of the surface, thus lowering the memory requirement. To maximize the number of empty cells we minimize t such that ray-casting produces no visual artifacts. For this we set t equal to 2 times the distance of two diagonally neighboring grid points wrt. the finest resolution. This ensures that an intersection with the object is never lost during the hierarchy construction. The octree is then constructed according to the following merging principles, where we consider groups of 2^3 blocks:

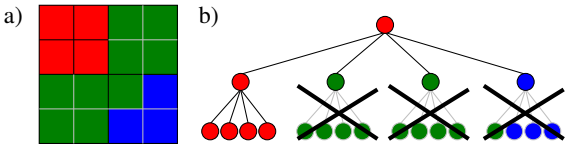

Figure 5: *a) In this 2D-illustration, each cell is surrounded by 4 neighboring grid points, where a VCP is given at each grid point. Green cells correspond to empty leaves. Blue / red cells correspond to non-empty leaves that can / cannot be merged at the next level. b) The resulting octree after eliminating all nodes that can be combined.*

- If all blocks are marked as empty, they are merged into an empty node in the next octree level (Fig. 5b, nodes with only green child nodes). The merged leaf nodes are removed, the new node becomes a leaf node representing a grid cell at the next coarser level, and it is assigned d_{min} as the minimum distance to the surface.
- If all nodes are leaves but some of them are non-empty, we check whether it is possible to simplify the block to one single cell. For this, we test how well the VCP at the vertices of the non-empty blocks can be interpolated from the 2^3 vertices of the single cell. We do so by computing the Euclidean distances d_i between the interpolated and the exact VCP vectors. If the maximum of these distance is below a given threshold, the nodes are merged to a non-empty leaf node at the next level, storing the minimum distance of all merged nodes. If the nodes cannot be merged, an internal node is inserted at the next level pointing to corresponding leaves (Fig. 5b, nodes with only green and blue child nodes).
- Otherwise, an internal node is inserted at the next level pointing to corresponding nodes (Fig. 5b, nodes with red child nodes).

3.1. Mesh-based Generation

In this section we describe how a full VCP grid is constructed from a given triangle mesh. To begin with, let us consider the simplified case of computing the VCP from an arbitrary point p in space to a single triangle given by points p_1, p_2, p_3. This can be done efficiently by calculating the barycentric coordinates of the (perpendicular) projection p' from p onto the triangle as proposed by Heidrich [Hei05]. Let $u = p_2 - p_1, v = p_3 - p1, n = u \times v, c = n \cdot n, w = p - p_1$. Then we obtain the barycentric coordinates as $\gamma = ((u \times w) \cdot n)/c, \beta = ((w \times v) \cdot n)/c, \alpha = 1 - \beta - \gamma$. If we have $0 \leq \alpha, \beta, \gamma \leq 1$ then p' lies inside the triangle and equals also the VCP, i.e., $vcp = p'$. This holds because the VCP vector to a plane is necessarily perpendicular. If p' is outside of the triangle the VCP must coincide with an edge (or a vertex). For this we compute the closest point vector to each edge and return the vector such that its length minimal. To compute the closest point vector from p to a line segment given by p_1, p_2, let $u = p_2 - p_1$ and $\lambda = ((p - p_1) \cdot d)/(d \cdot d)$. Let $\lambda^\star = clamp(\lambda, 0, 1)$. We finally obtain the VCP as $vcp = p_1 + \lambda^\star \cdot d$. Note that it is also possible to store other attributes such as color or texture coordinates at this point.

We now extend our approach to arbitrary meshes consisting of multiple triangles. Starting with an empty grid we insert each triangle into the grid in the following manner. First, we compute the bounding box for the current triangle and then update each grid point within this box. More precisely, each grid point is updated if it is currently empty or it is pointing to another triangle with a greater VCP distance. Next, to expand the VCP data computed so far, we utilize

a region growing algorithm. We iterate k times over all grid points and compare the current grid point p with each surrounding grid point q in the following way. If p is empty, i.e., points to no triangle, let p point to the same triangle as q. Otherwise, if p,q point to triangles T_p, T_q, update p such that it points to T_q if the VCP distance from p to T_q is less than the current distance, i.e., to T_p. The number k depends linearly on the aforementioned value ε.

4. GPU Implementation Issues

To reduce the number of memory indirections on the GPU, the octree is not constructed up to one single root node. Instead, the construction process is stopped at a certain level where the memory requirement falls below a given threshold. The data generated so far constitutes the indirection pool of the octree. To traverse it efficiently on the GPU the following information is encoded at each node as a 32-bit signed integer. For internal nodes a pointer to the node index of its first child is stored; all child nodes are then stored at subsequent locations in memory (up to index `0x40000000 - 1`). An empty leaf is encoded by an integer greater than or equal to `0x40000000` $= 2^{30}$ which represents the distance d_{min} to the nearest VCP wrt. all grid points on the boundary of the spatial region referred by that node. We compute this value by the formula $\left\lfloor \min\{d_{min}/D, 1\} \cdot \left(2^{30}-1\right) + 2^{30} \right\rfloor$, where D represents a certain cutoff-value. Finally, non-empty leaves are stored as negative integers `-(leafIdx + 1)`, where `leafIdx` points to the index in the VCP leaf data structure which is constructed as given in the next paragraph.

4.1. VCP Leaf Data

For each non-empty leaf, 2^3 VCP vectors have to be stored. Note that leaves can overlap if they belong to adjacent blocks in space as shown in Fig. 6a. In this case they share 4 VCP vectors which allows us to reduce the memory requirement. For this we sort the leaves by using the following algorithm. First, leaves are ordered by their corresponding octree level. Second, when leaves are on the same level (only in this case the possibility of adjacency arises), they are sorted in xyz-order where without loss of generality the x-axis has the greatest length. Now we construct the leaf data as a 3D texture in such a way that cells share their VCP vectors wherever it is possible as illustrated in Fig. 6b. Finally, the leaf indices are assigned to the indirection pool. Another important aspect in achieving memory efficiency is the data type used to store VCP vectors. To reduce the memory requirement we propose a custom data type which consumes only 32 bit per VCP vector. We begin with transforming the VCP vector into spherical coordinates (r,θ,φ) given as the radius, azimuthal angle, $\theta \in [-\pi/2, \pi/2]$, and polar angle, $\varphi \in [0, 2\pi]$. We obtain the radius by normalizing the vector length wrt. a fixed maximum radius, i.e., $r = min\{\|pcs\|/r_{max}, 1\}$. To improve

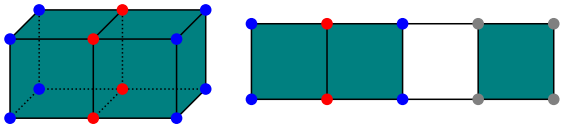

Figure 6: *Left: Two adjacent blocks overlap at 4 grid points (red) meaning that they share the corresponding 4 VCP vectors, which allows us to reduce the required amount of memory. Right: Cells with overlapping points are placed next to each other. Between non-overlapping cells there are empty regions (white background) which remain unused.*

the precision for small radii, we apply the square root to the radius, i.e., $r \leftarrow \sqrt{r}$. Now the three components are stored as normalized unsigned integers into 32 bit where r occupies the first 13 bits, θ the next 9 bits and φ the remaining 10 bits.

4.2. Reducing Run-time Memory Traffic

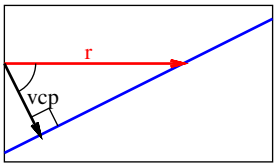

Figure 7: *The step size can be increased when considering the surface (blue) as locally planar. Given the angle between the vector vcp (black) and the ray direction r (red), the distance to surface along the ray can then be computed by applying the law of cosines.*

It is not necessary to traverse the octree at each step along the ray beginning from the root. Instead, we can simply remember the node index directly before the leaf. In many cases we can reuse this index as starting point, in particular when the step size is small which is the real bottleneck of our technique. To further optimize the number of global memory accesses we store at each non-empty leaf the 8 VCP values in local memory. This enables us to avoid reaccessing the same values in global memory as long as we stay in the same cell while marching along the ray, i.e., when the step size is very small and would hence require a lot of memory look-ups. Finally, as illustrated in Fig. 7, we can increase the step size. Considering the surface locally as planar would allow us to pick a step size of $s = d/\left(\frac{vcp}{d} \cdot r\right)$, where r denotes the normalized ray direction. In practice, however, this leads to rendering artifacts because the surface is not planar on a larger scale. Therefore, we restrict the step size optimization by a threshold, in our case 2ε. Fig. 8 shows a comparison of the optimized vs. the unoptimized approach that clearly demonstrates its advantage. A similar algorithm was employed by Hart in the context of sphere tracing [Har96].

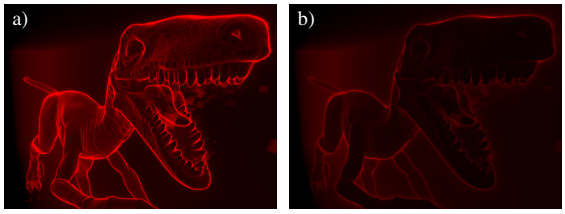

Figure 8: *Comparison of the unoptimized (a) vs. the optimized approach (b). Here the number of accesses to global memory is encoded by color intensity. Clearly, the optimized version is much more efficient. Note that this also results in significantly greater frame rates.*

4.3. Dynamic Memory Management

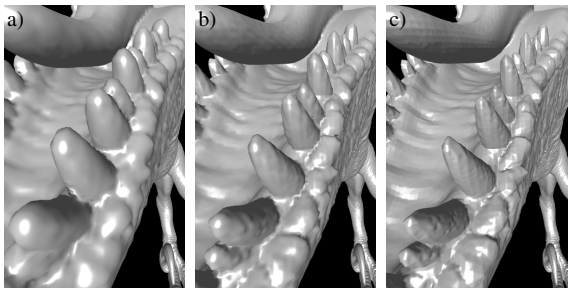

Figure 9: *Rendering was performed a) at a coarser resolution (to demonstrate the effect more clearly), b) at a finer resolution, c) including further refined subvolumes.*

To improve the rendering quality, we have developed a system that dynamically loads more refined subsets of the given geometry into GPU memory in places, which are located close to the camera and visible in the current viewport. For this, we divide the bounding box of the entire scene into subvolumes. Then, we consider each subvolume as a unique object and construct the VCP hierarchy by restricting the VCP grid on the respective region. During the rendering process, we continually compute the required set of subvolumes and load them into memory. Loading is done in the background, which exploits the multithreading capability of DirectX 11 and does not interrupt the rendering process. While ray-casting, we check at each step whether there is a refined VCP hierarchy available in GPU memory as long as the distance to the camera is less than a threshold, in our case $1/4$ of the diameter of the object's bounding box. In this case, we perform a look-up in the refined structure in the same way as explained in the previous section. The effect is demonstrated in Fig. 9. As a minor drawback of this approach, one can sometimes spot popping artifacts upon switching to different subvolume levels. However, we regard this as acceptable, since our method dramatically reduces the runtime memory requirement on the GPU, thus enabling a significantly finer resolution.

4.4. GPU-CPU Upstreaming

To determine which subvolumes are currently needed, we identify these subvolumes which are hit by rays at early steps. While ray-casting, we compute at each step which subvolume is currently hit. Then we increase a counter associated with that particular subvolume by the number $1/(a+\varepsilon)$, where a is the accumulated step size. Afterwards, the counters are ordered decreasingly, which results in a priority list. Finally, the associated subvolumes are loaded into GPU memory according to their priority. To minimize the number of interlocked accesses to global memory (required for the counter to be consistent), we run a separate rendering process for this issue. Here, it is sufficient to use a very small texture as rendering target, in our implementation of size 24×16 pixels.

5. Results

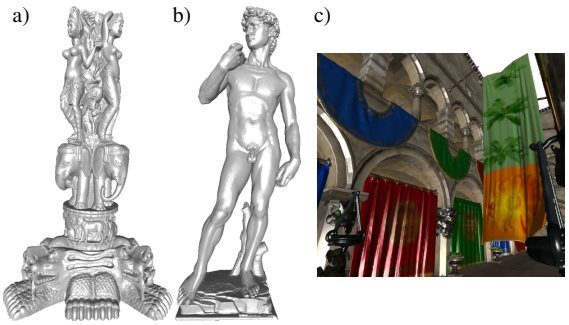

Figure 10: *Objects used in our test cases. From left to right: Thai Statue, David, Sponza Atrium. All images were rendered with the implementation of our method.*

In this section, we show several results of our method. Then, we compare our approach by performance and quality against ray-casting (1) on signed distance octrees, (2) on voxel octrees, i.e., binary voxel grids organized as octrees, and (3) the sparse voxel octree (SVO) implementation by Laine et al. [LK10b]. All measurements were performed on a desktop PC equipped with an Intel Xeon X5675 CPU at 3 GHz and an NVIDIA GeForce GTX 580 graphics adapter with 3 GiB of memory. A viewport of resolution 1920×1200 was used for all renderings. In Fig. 10 renderings of the utilized objects are depicted. Table 1 shows relevant statistical properties about our test scenarios. Note that the resolution for all techniques is chosen such that roughly the same amount of memory is consumed.

A comparison of the performance is presented in table 2. Note that similar frame rates are achieved in all scenarios,

Object	Tri.	VCP Res/Mem		Sgn Dist. Res/Mem		Voxel Res/Mem		SVO Res/Mem	
Thai	10M	$2K \times 1K^2$	1.05 GiB	$2K \times 1K^2$	1.05 GiB	$4K \times 2K^2$	1.07 GiB	$2K \times 1K^2$	860 MiB
David	960M	$2K \times 1K^2$	980 MiB	$2K \times 1K^2$	980 MiB	$4K \times 2K^2$	960 MiB	$2K \times 1K^2$	770 MiB
Sponza	150K	$1K \times 512^2$	290 MiB	$1K \times 512^2$	290 MiB	$1K \times 512^2$	350 MiB	N/A	

Table 1: *For different objects, the following data is shown: the number of triangles of the original mesh; resolution and memory consumption for different techniques.*

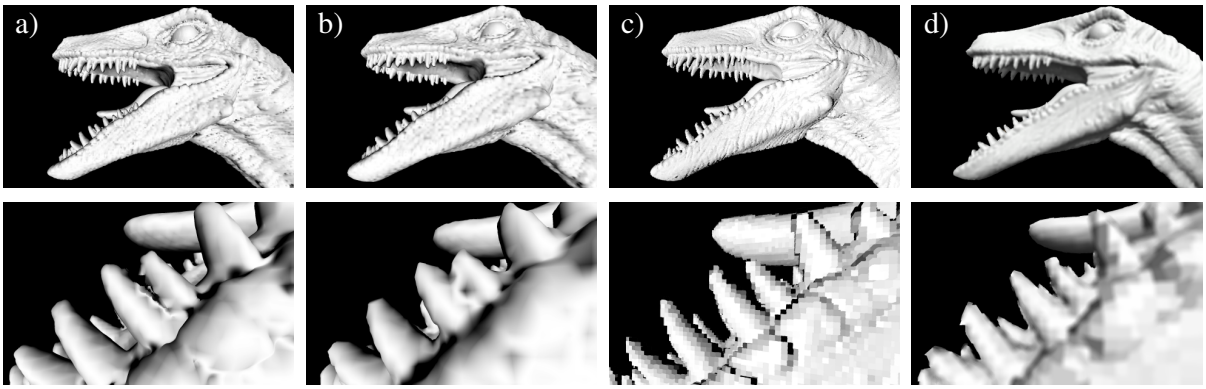

Figure 11: *Comparison of rendering quality for different ray-casting techniques: a) VCP b) signed distance field c) voxel grid d) SVO. To demonstrate the differences, the resolution was deliberately chosen at a lower level, with the constraint that the memory consumption is roughly equal for each technique.*

Scene	VCP	Sgn. Dist.	Voxel	SVO
Thai 1	8.6	8.2	8.0	8.1
Thai 2	11.7	11.1	10.6	11.0
Thai 3	20.4	19.8	19.4	18.2
David	9.1	8.7	8.2	8.9
Sponza	18.3	17.7	18.7	N/A
Raptor (Top)	16.7	16.2	15.9	14.2
Raptor (Bottom)	13.2	12.9	12.7	16.8

Table 2: *Comparison of rendering times per frame. Each value is given in milliseconds. The Thai scenes belong to Fig. 1 (from left to right). David and Sponza are (monochrome) renderings of the objects shown in Fig. 10. The raptor scenes correspond to Fig. 11.*

which we attribute to the fact the same geometry is rendered by a ray-casting technique with roughly the same memory requirement.

Next, we study the different techniques wrt. the rendering quality they achieve. The results are shown in Fig. 11 with close-ups in the bottom row. Our method preserves fine details better than the signed distances field for the previously mentioned reasons. Moreover, the use of voxel ray-casting produces clearly recognizable artifacts due to the underlying block structure of the voxel grid. The SVO method avoids such artifacts; however, in generally smoother regions finer details appear somewhat blurred. It should be noted that the quality of all techniques can be significantly increased by using a greater resolution.

6. Conclusion and Future Work

In this work, we have presented a novel rendering technique built upon a data representation based on Vectors-to-Closest-Points wrt. the geometry of the scene. By using a regular grid as the underlying structure, we achieve an efficient GPU-implementation of ray-casting. Since the grid is organized in an octree-like fashion, the memory consumption is minimized. Furthermore, by relying on VCP vectors rather than employing voxel grids, our technique has the advantage of avoiding artifacts, which are inherent to all voxel-based methods. We have also presented an algorithm to generate VCP hierarchies, i.e., octree-representations of VCP grids, from polygon meshes in an computationally efficient manner. Our algorithm can be used for arbitrary large meshes by processing their faces sequentially as we have demonstrated for the David statue. We have also proposed and implemented several ways to significantly enhance the performance of our ray-casting algorithm by exploiting the underlying VCP representation. To circumvent the memory restrictions on the GPU, we have implemented an algorithm that dynamically loads only a certain subset of the whole ge-

ometry at a finer resolution into GPU memory. This subset is determined by the analyzing which parts of the scene in the current viewport lie in close vicinity to the camera. Finally, we have presented various results that clearly demonstrate the potential of our approach in the context of interactively rendering large meshes.

In the future, we plan to implement a DAG structure inspired by Sparse Voxel DAGs [KSA13]. By recognizing identical VCP regions, we expect to significantly lower the memory requirement, in particular for scenes exhibiting a high degree of self-similarity. We further aim for rendering shadows and specular reflections by integrating non-camera rays into our implementation.

Acknowledgment

This work was supported by the European Union under the ERC Advanced Grant 291372—SaferVis—Uncertainty Visualization for Reliable Data Discovery.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009* (2009). 1

[AMT*12] AUER S., MACDONALD C. B., TREIB M., SCHNEIDER J., WESTERMANN R.: Real-time fluid effects on surfaces using the closest point method. *Computer Graphics Forum 31*, 6 (2012). 2

[BC08] BASTOS T., CELES W.: Gpu-accelerated adaptively sampled distance fields. In *Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on* (June 2008), pp. 171–178. 2

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Proc. Graphics Interface* (2006), pp. 203–209. 1

[CHH02] CARR N. A., HALL J. D., HART J. C.: The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), pp. 37–46. 1

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering, feb 2009. to appear. 1

[EVG04] ERNST M., VOGELGSANG C., GREINER G.: Stack implementation on programmable graphics hardware. In *Vision Modeling and Visualization* (2004), pp. 255–262. 1

[FPRJ00] FRISKEN S. F., PERRY R. N., ROCKWOOD A. P., JONES T. R.: Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (2000), SIGGRAPH '00, pp. 249–254. 2

[FS05] FOLEY T., SUGERMAN J.: Kd-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), pp. 15–22. 1

[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* (1998), VVS '98, pp. 23–30. 2

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

[GMIG08] GOBBETTI E., MARTON F., IGLESIAS GUITIÁN J. A.: A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *Vis. Comput. 24*, 7 (July 2008), 797–806. 1

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

[Har96] HART J. C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12* (1996), 527–545. 2, 5

[Hei05] HEIDRICH W.: Computing the barycentric coordinates of a projected point. *Journal of Graphics, GPU, and Game Tools 10*, 3 (2005), 9–12. 4

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), pp. 167–174. 1

[JBS06] JONES M. W., BAERENTZEN J. A., SRAMEK M.: 3d distance fields: A survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics 12*, 4 (July 2006), 581–599. 2

[KSA13] KÄMPE V., SINTORN E., ASSARSSON U.: High resolution sparse voxel dags. *ACM Trans. Graph. 32*, 4 (July 2013), 101:1–101:13. 2, 8

[KSK*14] KEINERT B., SCHÄFER H., KORNDÖRFER J., GANSE U., STAMMINGER M.: Enhanced Sphere Tracing. pp. 1–8. 2

[LK10a] LAINE S., KARRAS T.: Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2010), I3D '10, pp. 55–63. 1

[LK10b] LAINE S., KARRAS T.: *Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation*. NVIDIA Technical Report NVR-2010-001, NVIDIA Corporation, Feb. 2010. 2, 6

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712. 1

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (2007), 415–424. 1

[RCBW12] REICHL F., CHAJDAS M. G., BÜRGER K., WESTERMANN R.: Hybrid Sample-based Surface Rendering. pp. 47–54. 2

[WSE99] WESTERMANN R., SOMMER O., ERTL T.: Decoupling polygon rendering from geometry using rasterization hardware. In *Proceedings of the 10th Eurographics Conference on Rendering* (1999), EGWR'99, pp. 45–56. 2