Fast Parallel Surface and Solid Voxelization on GPUs

Michael Schwarz* Hans-Peter Seidel

Max-Planck-Institut Informatik

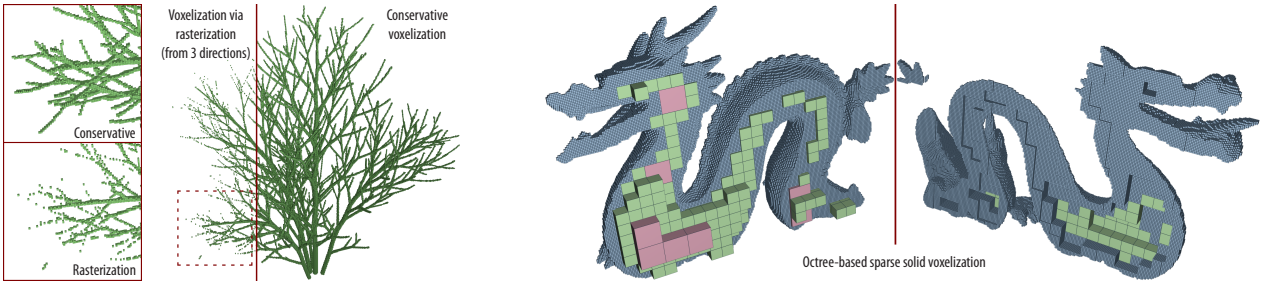


Figure 1: *Our methods can rapidly create surface and solid voxelizations, surpassing the limitations of rasterization-based approaches. Left: The conservative voxelization robustly captures all overlapped voxels. Right: Exploiting the uniformity of large voxel regions, we can directly perform solid voxelization into a sparse octree, significantly reducing the memory requirements.*

Abstract

This paper presents data-parallel algorithms for surface and solid voxelization on graphics hardware. First, a novel conservative surface voxelization technique, setting all voxels overlapped by a mesh's triangles, is introduced, which is up to one order of magnitude faster than previous solutions leveraging the standard rasterization pipeline. We then show how the involved new triangle/box overlap test can be adapted to yield a 6-separating surface voxelization, which is thinner but still connected and gap-free. Complementing these algorithms, both a triangle-parallel and a tile-based technique for solid voxelization are subsequently presented. Finally, addressing the high memory consumption of high-resolution voxel grids, we introduce a novel octree-based sparse solid voxelization approach, where only close to the solid's boundary finest-level voxels are stored, whereas uniform interior and exterior regions are represented by coarser-level voxels. This representation is created directly from a mesh without requiring a full intermediate solid voxelization, enabling GPU-based voxelizations of unprecedented size.

Keywords: voxelization, GPU computing, overlap testing, parallel data structure creation, octree

1 Introduction

Binary voxel representations are heavily employed in computer graphics. They are typically derived as discrete approximations of objects via a process called voxelization. In a *surface voxelization*, all voxels are set that fulfill some overlap or distance criterion with respect to a surface, whereas a *solid voxelization* sets all voxels considered interior to an object.

*e-mail: mschwarz@mpi-inf.mpg.de

ACM Reference Format
Schwarz, M., Seidel, H. 2010. Fast Parallel Surface and Solid Voxelization on GPUs.
ACM Trans. Graph. 29, 6, Article 179 (December 2010), 9 pages. DOI = 10.1145/1866158.1866201
http://doi.acm.org/10.1145/1866158.1866201.

Offering a regular representation that is independent from object and surface complexity, voxelizations are used in domains as diverse as 3D shape matching and visibility processing. In the latter, for instance, rays may be cast against a dynamically created scene voxelization to determine occlusion when computing lighting [Nichols et al. 2010] or ambient occlusion [Reinbothe et al. 2009]. Another application is collision detection, where a voxel-wise comparison of the voxelizations of two objects easily identifies collisions. Similarly, bitwise combinations of solid voxelizations are naturally applicable for performing constructive solid geometry tasks. Voxelizations are also helpful for establishing computational domains, e.g. for fluid simulations with obstacles or for light propagation.

Since voxelization is often a performance-critical, integral part, several algorithms have been proposed to perform voxelization of triangle meshes on the GPU, leveraging the fast rasterization pipeline for this 3D scan conversion. However, this inviting use of the rasterizer causes most surface voxelization approaches to suffer from gaps and missed thin features in the voxelization, mainly because a voxel is only set if the triangle overlaps the corresponding pixel's center. Although this can be alleviated by manually enlarging the triangle's pixel footprint in the geometry shader stage, resulting algorithms are typically rather slow, hampering their utility for real-time applications. Concerning solid voxelization, a fast, straightforward approach exists [Eisemann and Décoret 2008] but unfortunately only works with OpenGL, as it relies on bitwise blending operations, which are no longer exposed in Direct3D. By contrast, alternative solutions processing voxel slices sequentially are generally supported by all graphics APIs but are much slower.

In this paper, we approach voxelization with new data-parallel algorithms that utilize GPUs as general, massively parallel compute devices instead of building on the standard pipeline. Basically, they replace the fixed-function 2D rasterizer by a collection of custom "3D rasterizers". This has several advantages. First, we don't have to indirectly modify the rasterizer's pixel coverage test by appropriately enlarging input triangles and discarding superfluous fragments, which constitutes extra work and occupies processing units, but can directly apply our desired coverage criterion. Second, we may dynamically choose the coordinate plane (xy, xz or yz) optimal for processing on a per-triangle basis, whereas in the standard pipeline this involves a separate rendering for each coordinate plane as well as a subsequent data merging step. This is due to the texture-based encoding of the voxelization, e.g. across multiple channels of

a texture array. By contrast, we are free to adopt any storage organization scheme and can access this arbitrarily.

In particular, this freedom enables us to account for the fact that surface voxelizations are typically populated rather sparsely and that, analogously, solid voxelizations usually encompass many uniform regions. For instance, we could store such a voxelization in a sparse octree, where a coarser-level node is only refined if the represented voxel space is non-uniform, i.e. finest-level nodes merely exist for voxels close to the object's boundary. As we show in this paper, such a space-efficient voxelization representation can be created directly from an input mesh without having to first derive an intermediate full voxelization. Hence, it becomes easily possible to cope with voxelizations of grid size 4096^3, which would require 8 GB of memory if represented fully as bit array. As an additional benefit, the hierarchical representation enables faster processing of the voxel data, e.g. during ray traversal.

This paper covers a wide spectrum of voxelization procedures with dedicated data-parallel solutions that take the above considerations into account. Initially, we focus on surface voxelization (Sec. 3) and present a new parallel algorithm for fast conservative voxelization, where all voxels overlapped by a triangle are identified and set, utilizing a new triangle/box overlap test. Adapting this test, we then address creating a thinner, 6-separating voxelization. Subsequently, we turn to solid voxelization (Sec. 4) and introduce both a direct, triangle-parallel approach and an advanced tile-based algorithm. Finally, we present a novel octree-based sparse solid voxelization technique (Sec. 5), demonstrating the direct scan-conversion of triangles into a sparse spatial data structure on the GPU.

Numerous applications that utilize voxelizations can directly profit from our methods. Generally, they enable speeding up these applications, improving their attainable quality (e.g. because conservative voxelization becomes affordable in real-time scenarios), or using larger voxelizations without negatively affecting memory consumption or construction time.

2 Related work

Real-time voxelization Many voxelization algorithms with various properties have been devised. Among the most relevant real-time approaches, Fang and Chen [2000] construct a surface voxelization slice-wise, rendering the geometry once for each voxel slice while restricting the view volume to this slice. To reduce the number of rendering passes, Li et al. [2005] employ depth peeling to capture all surface points and then scatter them into a flattened voxel grid via point rendering. The peeling process is done along all three orthogonal voxel grid axes to decrease the number of missed voxels. While these approaches spend one texel for storing a voxel, Dong et al. [2004] encode binary voxels in separate bits of multiple multi-channel render targets, allowing to treat many slices in a single rendering pass. A fragment's depth is used to derive the voxel and its bit is set via additive alpha blending. They also consider all three grid axes as viewing direction, each time rendering only the triangles whose normal's dominant direction is parallel to the current axis, and then compose the resulting voxelizations. By contrast, Eisemann and Décoret [2006], employing the same efficient encoding, render only from one viewing direction but use the more robust bitwise OR-blending.

All these approaches easily miss thin structures and may suffer from gaps, especially if only one viewing direction is utilized, because they are based on the point sampling provided by conventional rasterization. In their conservative voxelization algorithm, Zhang et al. [2007] hence use conservative rasterization [Hasselgren et al. 2005] to capture all pixels overlapped by a triangle and then derive a per-pixel depth range along the viewing direction, identifying all overlapped voxels. Sometimes, however, additional voxels may be set due to robustness problems in the depth range computation.

Fewer methods target solid voxelization; most are restricted to closed, watertight input objects and determine all voxels whose center is inside. This classification is based on the parity of the number of intersections a ray originating at the voxel center has with the object, where an odd value indicates being interior. Hence, rendering the object from one direction and for each fragment flipping the inside/outside state of all affected voxels below directly yields the desired voxelization. Fang and Chen [2000] present an according slice-wise approach, whereas Eisemann and Décoret [2008] process all slices at a time, achieving high performance.

Rasterization Testing whether a sample point is covered by a triangle is typically done via three linear edge functions [Pineda 1988], whose sign indicates in which half-space of an edge a point is located. Once set up, these functions can be efficiently evaluated for multiple points. Akenine-Möller and Aila [2005] extend this approach to test a 2D axis-aligned box for being overlapped by a triangle. They show that the only modification required is selecting a different evaluation point based on the signs of an edge's normal components, where the involved point offset can be directly baked into the function coefficients; this insight is used extensively in our coverage tests.

Recently, a few approaches have been presented that perform rasterization on massively parallel hardware, omitting the fixed-function rasterizer. Some of them operate triangle-parallelly, using one thread per triangle, which loops over the pixels in the triangle's bounding box and tests them for coverage [Liu et al. 2010; Fatahalian et al. 2009]. Others pursue a tile-based strategy [Seiler et al. 2008; Abrash 2009; Eisenacher and Loop 2010], where first triangles are assigned to all pixel tiles they overlap. Then, tiles are processed in parallel (typically allocating one thread per sample point), each processing its triangle list sequentially. This approach enables keeping a tile's pixel data in fast memory and avoids conflicting writes by multiple triangles overlapping the same pixel.

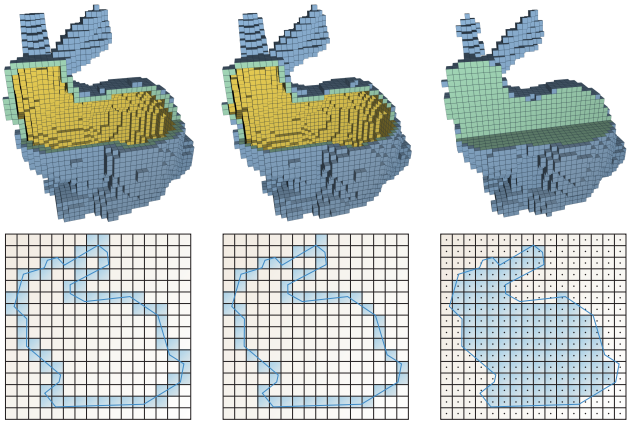
Parallel data structure construction Efficient GPU-based construction algorithms have been developed for several spatial data structures, like kd-trees [Zhou et al. 2008] and BVHs [Lauterbach et al. 2009]. Sun et al. [2008] build a texture-based full octree representation of a given solid voxelization, while Zhou et al. [2010] construct a sparse octree of a point cloud with full per-node adjacency information. Moreover, two roughly identical algorithms for constructing uniform grids for triangle soups [Kalojanov and Slusallek 2009; Ivson et al. 2009] have been devised; they are basically also applicable to assigning triangles to pixel tiles.

3 Surface voxelization

For surface voxelization, many applications demand a conservative criterion that sets a voxel if it is (partially) overlapped by a triangle (cf. Fig. 2 a), where we also consider voxels that are merely touched by a triangle to be overlapped by it. After initially concentrating on this conservative voxelization in the following subsections, we additionally introduce a different criterion in Sec. 3.4 that results in thinner voxelizations.

3.1 Triangle/box overlap test

Conservative voxelization requires identifying all voxels a triangle overlaps and hence it is crucial to utilize a fast test for triangle/voxel overlap. Such a test should further efficiently support an organization into an initial setup stage that merely depends on the triangle and a subsequent actual test part for a specific voxel that utilizes this setup data, since this enables rapid sequential and parallel test-



(a) Conservative (b) 6-separating (c) Solid

Figure 2: *Different kinds of voxelizations covered.*

ing of multiple (identically-sized) voxels. A new triangle/box test that fulfills these requirements is detailed in the following.

Given a triangle $\mathcal{T}$ with vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ and an axis-aligned box $\mathcal{B}$ (e.g. a voxel) with minimum corner $\mathbf{p}$ and maximum corner $\mathbf{p}+\Delta\mathbf{p}$, we observe that $\mathcal{T}$ overlaps $\mathcal{B}$ iff

a) $\mathcal{T}$'s plane overlaps $\mathcal{B}$ and

b) for each of the three coordinate planes (xy, xz, yz), the 2D projections of $\mathcal{T}$ and $\mathcal{B}$ into this plane overlap.

To test for the plane overlap, similar to Haines and Wallace [1991], we determine $\mathcal{T}$'s normal $\mathbf{n}$ and the *critical point*

$$\mathbf{c} = \left(\begin{cases}\Delta\mathbf{p}_x, & \mathbf{n}_x > 0\\ 0, & \mathbf{n}_x \le 0\end{cases}, \begin{cases}\Delta\mathbf{p}_y, & \mathbf{n}_y > 0\\ 0, & \mathbf{n}_y \le 0\end{cases}, \begin{cases}\Delta\mathbf{p}_z, & \mathbf{n}_z > 0\\ 0, & \mathbf{n}_z \le 0\end{cases}\right)^{\mathrm{T}}$$

and check whether $\mathbf{p}+\mathbf{c}$ and the opposite box corner $\mathbf{p}+(\Delta\mathbf{p}-\mathbf{c})$ are on different sides of the plane or one of them is on the plane, that is whether

$$(\langle\mathbf{n},\mathbf{p}\rangle + d_1)\,(\langle\mathbf{n},\mathbf{p}\rangle + d_2) \le 0, \tag{1}$$

where $d_1 = \langle\mathbf{n}, \mathbf{c}-\mathbf{v_0}\rangle$ and $d_2 = \langle\mathbf{n}, (\Delta\mathbf{p}-\mathbf{c})-\mathbf{v_0}\rangle$.

For the 2D projection overlap tests, we utilize edge functions [Pineda 1988], each evaluated at that corner of $\mathcal{B}$'s projection, a 2D axis-aligned box, that yields the largest value and hence is "most interior" with respect to the edge (cf. Fig. 3 a). More precisely, using the xy coordinate plane as example, we compute

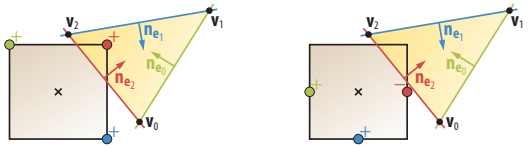
$$\begin{aligned}\mathbf{n}^{\mathrm{xy}}_{\mathbf{e}_i} &= (-\mathbf{e}_{i,\mathrm{y}}, \mathbf{e}_{i,\mathrm{x}})^{\mathrm{T}} \cdot \begin{cases}1, & \mathbf{n}_z \ge 0\\ -1, & \mathbf{n}_z < 0\end{cases}\\ d^{\mathrm{xy}}_{\mathbf{e}_i} &= -\langle\mathbf{n}^{\mathrm{xy}}_{\mathbf{e}_i}, \mathbf{v}_{i,\mathrm{xy}}\rangle + \max\left\{0, \Delta\mathbf{p}_x\mathbf{n}^{\mathrm{xy}}_{\mathbf{e}_i,\mathrm{x}}\right\} + \max\left\{0, \Delta\mathbf{p}_y\mathbf{n}^{\mathrm{xy}}_{\mathbf{e}_i,\mathrm{y}}\right\}\end{aligned} \tag{2}$$

for all three edges $\mathbf{e}_i = \mathbf{v}_{i+1 \bmod 3} - \mathbf{v}_i$ and test whether

$$\bigwedge_{i=0}^{2}\left(\langle\mathbf{n}^{\mathrm{xy}}_{\mathbf{e}_i}, \mathbf{p}_{\mathrm{xy}}\rangle + d^{\mathrm{xy}}_{\mathbf{e}_i} \ge 0\right) \tag{3}$$

holds true, indicating overlap. Because the evaluation points for the edge functions differ, it is additionally necessary to verify that $\mathcal{T}$'s axis-aligned bounding box actually overlaps $\mathcal{B}$.

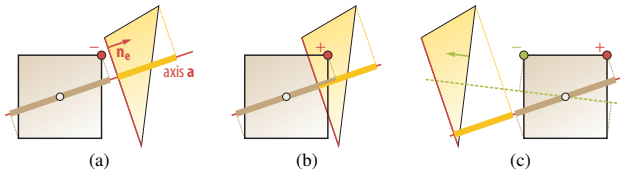
Consequently, for a given triangle $\mathcal{T}$ and box extent $\Delta\mathbf{p}$, $\mathcal{T}$'s bounding box, $\mathbf{n}$, d_1, d_2, and $\mathbf{n}^{\mathrm{xy}}_{\mathbf{e}_i}$, $d^{\mathrm{xy}}_{\mathbf{e}_i}$, $\mathbf{n}^{\mathrm{xz}}_{\mathbf{e}_i}$, $d^{\mathrm{xz}}_{\mathbf{e}_i}$, $\mathbf{n}^{\mathrm{yz}}_{\mathbf{e}_i}$, $d^{\mathrm{yz}}_{\mathbf{e}_i}$ $(i = 0, 1, 2)$ can be determined in a setup stage. The actual overlap test for a box with minimum corner $\mathbf{p}$ then requires merely testing for bounding box overlap and checking the criteria in Eqs. 1 and 3.



(a) Conservative (b) 6-separating

Figure 3: *Critical points for evaluating the edge functions in the 2D projection overlap tests, annotated with the function result's sign.*

Comparison The current standard triangle/box overlap test by Akenine-Möller [2001] is based on the separating axis theorem (SAT). The tests for the coordinate axes (x, y, z) and the normal $\mathbf{n}$ are equivalent to our bounding box and plane overlap tests, respectively. Interestingly, the remaining 9 axes tested essentially correspond to our 2D edge normals $\mathbf{n}^{\mathrm{xy}}_{\mathbf{e}_i}$, $\mathbf{n}^{\mathrm{xz}}_{\mathbf{e}_i}$, $\mathbf{n}^{\mathrm{yz}}_{\mathbf{e}_i}$ $(i = 0, 1, 2)$. However, while the SAT approach requires testing the projections of $\mathcal{T}$ and $\mathcal{B}$ onto an axis for overlap, our method merely necessitates evaluating an edge function and checking the result's sign. As illustrated in Fig. 4, the SAT test for one of these axes actually performs unnecessary work. Of the two configurations where an axis is separating (a, c), i.e. the projections onto it don't overlap, only the one where the box is in the exterior half-space of the corresponding edge (a) needs to be captured; the other one is already handled by the axis for the more adjacent edge (c). Overall, the SAT-based triangle/box overlap test requires more instructions than our approach, and a setup-based formulation additionally involves more set-up quantities in the per-box test part, hence consuming more registers when implemented on the GPU.



(a) (b) (c)

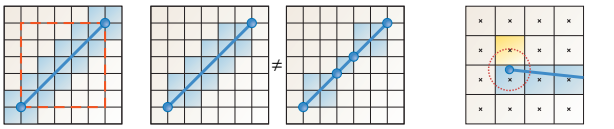
Figure 4: *Different configurations for SAT-based overlap test.*

3.2 Triangle-parallel conservative voxelization

To obtain a conservative voxelization, all voxels overlapped by an input mesh's triangles must be determined. One natural data-parallel approach to this computation is processing all triangles in parallel, launching one thread per triangle. For each triangle, first the bounding box is determined and then all voxels inside the bounding box are tested for overlap utilizing our triangle/box overlap test. If an overlap test passes, the corresponding voxel is set.

Note that we also consider voxels that are merely touched by the bounding box, which is important to make the voxelization independent of the tessellation of planar surfaces (cf. Fig. 5 a). Moreover, since only voxels overlapped by the bounding box are processed in the first place, the bounding box overlap test can be omitted when running the triangle/box overlap test.

Voxel updates Each voxel's state is encoded by a single bit in a linear array. With multiple triangles being processed concurrently, some of them may try to update the same 32-bit value at the same time. We hence employ the atomic OR function to avoid conflicting writes and ultimately missing any update. Moreover, when looping over the voxels within the bounding box, we make the inner-most loop proceed in x direction, where adjacent voxels are stored in consecutive bits. Instead of writing each set voxel instantaneously, we buffer the 32-bit value a voxel's bit belongs to in a register and only write it to memory once all relevant voxels within this 32-bit value have been processed, potentially saving many atomic updates.



(a) Consistent vs. inconsistent voxelization (b) Vertex voxelization

Figure 5: *The voxelization should be independent of planar surface tessellations. (a) Hence, touched voxels outside the bounding box must also be considered. (b) But the simple distance criteria by Huang et al. [1998] may wrongly include additional voxels.*

Test specialization The algorithm can be improved by reducing the number of tested voxels and case-specificly skipping unnecessary parts of the triangle/box overlap test. Key to this is the observation that because a triangle is planar, its voxelization is at most three voxels thick in the dominant axis direction of the triangle's normal $\mathbf{n}$. We exploit this by specializing the test for all three coordinate axes. For instance, if the z axis is $\mathbf{n}$'s dominant axis, it suffices to loop over the triangle's extent in the xy coordinate plane. For each visited voxel column (extending along z), first the 2D overlap test for the triangle's xy projection is run. If it passes, we determine the range of voxels within the column overlapped by the triangle's plane. To this end, based on the signs of $\mathbf{n}_x$ and $\mathbf{n}_y$, those two opposing corners of the voxel column (in the xy plane) are determined where the plane reaches its minimum and maximum z value within the column, respectively. These are then projected along the z axis onto the triangle's plane to determine the covered z range. Subsequently, all voxels in this range are subjected to the remaining two 2D overlap tests for the xz and yz planes. Note that no further plane overlap test is required, since by construction only voxels overlapped by the triangle's plane are processed.

If the (unclipped) depth range for a voxel column encompasses only one voxel, this voxel is guaranteed to be overlapped by the triangle. Consequently, the outstanding 2D overlap tests for the xz and yz planes can be omitted. One may hence even defer the setup for these two tests. Similarly, if the (unclipped) bounding box of a triangle is only one voxel thick, the whole triangle/box overlap test simplifies to a single 2D projection overlap test. Even more, in case the (unclipped) bounding box covers merely one voxel in at least two directions, all voxels overlapped by the bounding box can be directly set without any further tests.

Implementation Overall, nine different cases can be distinguished where the discussed test optimizations apply: 1D bounding boxes (one-voxel extent in ≥ 2 directions) and three possible axes along which they extend; 2D bounding boxes and three possible coordinate planes in which they extend; 3D bounding boxes and three possible dominant axes of the triangle's normal.

One straightforward implementation option is determining the occurring case when processing a triangle and then running the according specialization of the overlap test. Although it is possible to merge the specializations for the 1D cases with y and z extent as well as for the 2D cases with xz and xy extent, this approach easily suffers from high thread divergence, since a warp of them is executed in SIMD lockstep but the respective triangles can belong to many different cases. Moreover, the number of voxels to test per triangle may vary widely, resulting in underutilization.

This can be alleviated by pursuing a multi-kernel approach instead, where initially the case for each triangle is determined and written to a buffer, along with the triangle id. The stored case value encodes the bounding box type, the axis or plane of extent, and the number of voxel columns to process. The buffer is then sorted and the number of cases for each bounding box type is derived via compaction. Thanks to this reordering, all triangles in a warp typically belong to the same case and have a similar number of voxels to process. In practice, we utilize a separate kernel for each bounding box type, enabling a higher degree of concurrency for the 1D and 2D cases because of their lower register requirements.

3.3 Separability

Our conservative voxelization constitutes a *supercover* of the input mesh and hence is 26-separating for closed surfaces [Cohen-Or and Kaufman 1995]. 26-separability is a topological property and means that no path of 26-adjacent voxels exists that connects a voxel on one side of the surface and a voxel on the other side. Two voxels are 26-adjacent if they share a common vertex, edge or face; they are 6-adjacent if they share a face. In the following, to additionally capture non-closed surfaces, we use the term separability in a looser sense for them, considering only paths that intersect the mesh surface.

A comparison of our approach to the criteria for separating voxelizations by Huang et al. [1998] reveals that our plane overlap test is mathematically equivalent to their definition of a 26-separating plane voxelization. Their extension to triangle meshes includes all voxels that both satisfy this plane criterion and have their center inside the positive half-spaces of all the 3D edge planes, as well as all voxels whose center are within a distance of half a voxel's diagonal to an edge or vertex. However, this easily includes voxels outside the voxelization of the triangle's plane (cf. Fig. 5 b), rendering the mesh's voxelization dependent on the tessellation of planar surface parts. By contrast, our method operates by essentially removing all voxels from the triangle plane's voxelization that are entirely outside the triangle. Interestingly, each involved 2D projection overlap test turns out to be equivalent to performing an 8-separating (the 2D analog to 26-separability) pixelization of the edges according to Huang et al.'s respective criterion and further including the triangle's interior. Note that we are hence basically voxelizing the edges in addition to the interior and that such an edge voxelization solely depends on the edge and not the triangle.

3.4 6-separating surface voxelization

In some applications, instead of a conservative voxelization, which is 26-separating (and hence 6-connected), a "thinner" surface voxelization that is 6-separating (and hence 26-connected) is preferred (cf. Fig. 2 b). Thanks to the observations in Sec. 3.3, we can easily adapt our overlap test to create such a 6-separating voxelization. Harnessing the criterion for 6-separating plane voxelization from Huang et al. [1998], the offsets in the plane test become

$$\begin{aligned} d_1 &= \langle \mathbf{n}, \tfrac{1}{2}\Delta\mathbf{p} - \mathbf{v_0}\rangle + \tfrac{1}{2}\Delta\mathbf{p}_\diamond|\mathbf{n}_\diamond| \\ d_2 &= \langle \mathbf{n}, \tfrac{1}{2}\Delta\mathbf{p} - \mathbf{v_0}\rangle - \tfrac{1}{2}\Delta\mathbf{p}_\diamond|\mathbf{n}_\diamond| \end{aligned} \qquad \text{with } \diamond = \arg\max_{\square=x,y,z} |\mathbf{n}_\square|.$$

Similarly, Huang et al.'s criterion for 4-separating line pixelization is adopted for the 2D projection overlap tests (cf. Fig. 3 b); e.g. the offsets from Eq. 2 become

$$d^{xy}_{\mathbf{e}_i} = \langle \mathbf{n}^{xy}_{\mathbf{e}_i}, \tfrac{1}{2}\Delta\mathbf{p}_{xy} - \mathbf{v}_{i,xy}\rangle + \tfrac{1}{2}\Delta\mathbf{p}_\diamond|\mathbf{n}^{xy}_{\mathbf{e}_i,\diamond}|, \quad \text{with } \diamond = \arg\max_{\square=x,y} |\mathbf{n}^{xy}_{\mathbf{e}_i,\square}|.$$

We again optimize the computations by taking the dominant axis of the triangle's normal into account. Apart from the test setup, the only change required is adapting the determination of the considered voxel range in a voxel column to the modified plane overlap test. Concretely, we project a voxel column's center (in the xy plane) along the z axis onto the triangle's plane and take the according voxel(s). Only in case this projection touches two voxels, more than one voxel needs to be considered. In contrast to the conservative voxelization, 1D and 2D bounding boxes additionally require plane overlap testing; we hence omit specializations for them.

4 Solid voxelization

Solid voxelization mandates a closed, watertight object and sets all voxels whose center is inside this object (cf. Fig. 2 c). To capture fine details and handle non-solid parts of a model like thin sheets, a surface voxelization may be performed subsequently, setting the additional voxels via bitwise OR.

Recall that solid voxelization essentially boils down to rasterizing the object into a multi-sliced frame buffer, where a fragment effects flipping the inside/outside state of all voxels below. In this section, we consider two according data-parallel algorithms: a triangle-parallel approach and a tile-based method.

4.1 Direct, triangle-parallel solid voxelization

Similar to the surface voxelization approaches, we parallelize over all triangles, dedicating one thread per triangle. For each triangle, we first determine the bounding box in the yz plane and derive the yz range of covered voxel centers. If this range is non-empty, we loop over the contained voxel columns. For each, the center is tested against the triangle's yz projection, using edge functions. The edge normals $\mathbf{n}^{\mathrm{yz}}_{\mathbf{e}_i}$ are determined as in Eq. 2 and $d^{\mathrm{yz}}_{\mathbf{e}_i} = -\langle \mathbf{n}^{\mathrm{yz}}_{\mathbf{e}_i}, \mathbf{v}_{i,\mathrm{yz}} \rangle$. If the test passes, the center is projected along the x axis onto the triangle's plane. The resulting x coordinate q in voxel space, where each voxel (i, j, k), $0 \le i < n_\mathrm{x}$, is of size 1^3, then yields the x range $\lfloor q + \frac{1}{2} \rfloor =: \bar{q}, \ldots, n_\mathrm{x} - 1$ of voxels whose corresponding bits have to be flipped. This is performed utilizing the atomic XOR function.

Note that if an edge or vertex shared by adjacent triangles overlaps the voxel center, the triangle overlap test has to report an overlap for only exactly one of the triangles to avoid erroneously counting one surface intersection multiple times. We hence adopt the top-left fill rule from ordinary rasterization, which ignores overlapping edges except left edges ($\mathbf{n}^{\mathrm{yz}}_{\mathbf{e}_i,\mathrm{y}} > 0$) and top edges ($\mathbf{n}^{\mathrm{yz}}_{\mathbf{e}_i,\mathrm{y}} = 0 \wedge \mathbf{n}^{\mathrm{yz}}_{\mathbf{e}_i,\mathrm{z}} < 0$). In practice, we modify the test from Eq. 3 to

$$\bigwedge_{i=0}^{2} \left((\langle \mathbf{n}^{\mathrm{yz}}_{\mathbf{e}_i}, \mathbf{p}_{\mathrm{yz}} \rangle + d^{\mathrm{yz}}_{\mathbf{e}_i}) + f^{\mathrm{yz}}_{\mathbf{e}_i} > 0 \right),$$

where the introduced quantity $f^{\mathrm{yz}}_{\mathbf{e}_i}$ is set to the smallest floating-point number if the left or top edge criteria hold and to zero otherwise. Moreover, we ensure a consistent ordering of the vertices.

This triangle-parallel voxelization approach has two main weaknesses. First, the number of relevant voxel columns can vary significantly among the triangles processed by one warp, leading to underutilization. Second, especially for larger voxel grids, flipping all voxels in the derived x range poses high memory bandwidth requirements. Moreover, many atomic operations may have to be sequentialized, negatively affecting performance.

4.2 Tile-based solid voxelization

To elude the shortcomings of the triangle-parallel method, we also pursued a tile-based approach to solid voxelization. At first, we assign triangles to tiles (in the yz plane) of 4×4 voxel columns, yielding a work queue of tile/triangle pairs sorted by tile. The tiles are then processed in parallel, using one warp of threads per tile, each looping over its assigned triangles.

Tile assignment Initially, we determine for all triangles in parallel the number of tiles overlapped by them. The resulting buffer is subjected to an exclusive scan, yielding the required size of the work queue and an offset into this queue for each triangle. Subsequently, all triangles are visited in parallel again, each determining the tiles overlapped by it and writing the corresponding tile/triangle pairs into the work queue, starting at its offset. This queue is then sorted by tile. Finally, we derive a list of all tiles in the queue along with an offset to where a tile's triangle list in the work queue starts via compaction.

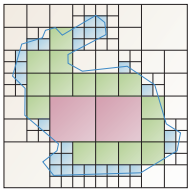
When determining a triangle's tile count, we first derive the triangle's bounding box and check whether any voxel column center is overlapped at all. If so, we test all tiles that contain voxel centers overlapped by the bounding box against the triangle, again using edge functions, each evaluated at the critical voxel column center within the tile. While this testing is more expensive than just enumerating the tiles covered by the bounding box, it results in a shorter queue, saving memory, speeding up sorting and avoiding processing irrelevant triangles later on. Since the testing has to be repeated when writing the tile/triangle pairs, we considered caching the binary results for the first 32 visited tiles per triangle in a buffer, which didn't pay off, though.

Tile processing The triangles assigned to a tile are processed sequentially. We allocate one thread per voxel column, testing its center for overlap and determining the range of affected voxels in the column, analogous to Sec. 4.1. To keep writes to global memory to a minimum, each thread maintains an active 32-bit segment of its column and a flip bitmask, where each bit corresponds to a 32-bit segment. When an overlapping triangle triggers the flipping of all voxels with x index $\ge \bar{q}$, we first check whether $\bar{q}$ is in the active segment. If this is not the case, we flush the active segment's value to global memory via bitwise XOR and choose $\lfloor \bar{q}/32 \rfloor$ as new active segment, initializing its value with zero. We then flip all bits for voxels $\ge \bar{q}$ in the active segment, as well as all bits in the flip bitmask for the segments succeeding the active one. Once a tile has been completely processed, the active segment is flushed and the bits of all segments whose bit is set in the flip bitmask are flipped. Note that no atomic memory operations are required.

Since a warp comprises 32 threads, to utilize all threads, we have to either choose a larger tile size, which reduces sample test efficiency [Fatahalian et al. 2009], or process two tiles per warp, which is susceptible to varying per-tile triangle counts, or process two triangles per warp, which leaves the second half-warp idle at the end of the triangle list if its length is odd. We opted for the last option, which causes two threads to process the same voxel column. To avoid conflicting and superfluous writes to global memory, these twin threads communicate. For instance, when one thread flushes an active segment but its twin thread has the same segment active, the value is incorporated in the twin thread's value instead of writing it to global memory.

Finally, to avoid fetching a triangle's data and setting up the overlap test quantities redundantly by all threads, we interleave the per-voxel-column processing with a parallel triangle setup stage where each thread fetches and sets up one triangle, storing the setup data in shared memory. In practice, we are currently limited to preparing 14 triangles at once due to the shared memory's limited size.

5 Sparse solid voxelization



The voxelization methods presented so far all operate on a full grid, explicitly storing each voxel's state. However, voxelizations typically contain larger blocks of uniform voxel state, suggesting a more space-efficient representation using a sparse hierarchical structure where nodes representing uniform regions are not further refined. Instead of creating a voxelization and then transforming it into such a representation, the voxelization should be performed directly into the hierarchical structure. We devised an according method for solid voxelization that uses an octree as representation, noting that it could be applied to surface voxelization analogously.

Because adding individual nodes on the fly during voxelization is ill-suited in a massively parallel context, we create the adapted octree structure up-front and subsequently populate it:

- First, all finest-level (i.e. level-0) nodes required during the voxelization have to be determined. Since an octree node has either no or eight children, it sufficies to merely determine all required level-1 nodes instead (Sec. 5.1).
- Based on this node set, the whole sparse octree is then constructed bottom-up (Sec. 5.2).
- Subsequently, the input object is voxelized into the finest-level nodes (Sec. 5.3).
- Finally, the inside/outside parity flips are propagated hierarchically to obtain a solid voxelization (Sec. 5.4).

In our implementation (cf. Fig. 6), each level-0 node stores a voxel sub-grid of size 4^3 for encoding and processing efficiency, thus essentially representing two finer levels. All nodes of one level are stored contiguously, sorted according to the Morton space-filling curve. Hence, a node's eight children are contiguous in memory and the first child has an index that is a multiple of eight. Each node maintains a pointer to its first child as well as pointers to its parent and its neighbors in x direction, which help accelerating the voxelization and propagation stages.

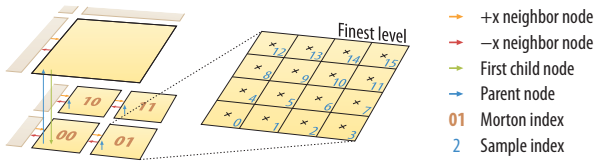


Figure 6: *Overview of the octree structure (2D analog).*

5.1 Determination of active level-1 nodes

To determine the level-1 nodes overlapped by the input mesh, referred to as *active* nodes, we could adopt a strategy similar to deriving the work queue in Sec. 4.2. However, it is more space-efficient to generate a surface voxelization into a grid with level-1 resolution and to extract the active voxels from it. Because all level-1 nodes whose voxel region is not uniform have to be identified as active, we perform a conservative voxelization, using the optimized version from Sec. 3.2 with the following modifications. First, the triangle is shifted in +x direction by half a level-0 sub-grid (SG) voxel to account for sampling at the voxel center. Second, we enlarge the triangle's bounding box in −x direction by one SG voxel and adapt the 2D edge functions accordingly, shifting all critical points that are on the voxel's max x face in +x direction by one SG voxel. This ensures that the final voxelization boundary consists solely of level-0 nodes (see the node marked with $*$ in Fig. 7 for an example).

After the level-1 voxelization is created, we derive a list of active nodes sorted in Morton order, with a node's Morton code being determined by interleaving the bits of its xyz grid position (i, j, k) (mapping it to the binary number $k_{n-1}j_{n-1}i_{n-1}\ldots k_0j_0i_0$). First, we launch one thread per 32-bit block of the voxelization. Each counts the number of set bits and writes it to a buffer. Then a scan is run on the buffer to derive offsets into the list of active level-1 nodes ($\textsc{ActNodes}_1$) as well as the list's length. Finally, again one thread is launched per block, writing the Morton codes of the block's set voxels into $\textsc{ActNodes}_1$, starting at its designated offset. To directly obtain a sorted list, we use blocks of extent 4×4×2, employ the block's index in Morton order when accessing the buffer and the scan result, and traverse the bits within the block in Morton order.

5.2 Octree construction

Given the list $\textsc{ActNodes}_1$, we then construct the sparse octree. First, the 8 length($\textsc{ActNodes}_1$) child nodes of the active level-1 nodes are allocated and initialized ($\textsc{Nodes}_0$), setting the x-neighbor pointers between adjacent nodes with the same parent node. Subsequently, by checking the Morton codes, we determine for each active level-1 node whether the next node in $\textsc{ActNodes}_1$ has a different level-2 parent, writing 1 if so and 0 otherwise into a buffer $\textsc{Flag}_1$, which is then subject to an exclusive scan, yielding $\textsc{ParentIndex}_1$. Note that its last entry plus one equals the number of active level-2 nodes N_2^{a}. We then allocate and initialize the $8N_2^{\mathrm{a}}$ level-1 nodes ($\textsc{Nodes}_1$) similar to the level-0 nodes. Moreover, we establish child and parent pointers for the active level-1 nodes and their children, respectively. Here, the i-th entry in $\textsc{ActNodes}_1$ corresponds to $\textsc{Nodes}_1[j]$ with $j = 8\,\textsc{ParentIndex}_1[i] + (\textsc{ActNodes}_1[i] \bmod 8)$ and has $\textsc{Nodes}_0[8i]$ as its first child.

Subsequently, a list of active level-2 nodes ($\textsc{ActNodes}_2$) is derived from $\textsc{ActNodes}_1$ by stripping the Morton codes of the three least significant bits and performing compaction, utilizing $\textsc{Flag}_1$ and $\textsc{ParentIndex}_1$. $\textsc{ActNodes}_2$ is then processed like $\textsc{ActNodes}_1$ before to establish the level-2 nodes. The remaining levels are treated analogously. In practice, we stop this process before reaching a single root node at a level where all nodes are typically present (we chose two levels below the root node), as this later speeds up traversal. Consequently, all nodes at this top level n must be created, and during initialization x-neighbor pointers are established for all adjacent nodes. Finally, starting at the top level, x-neighbor pointers have to be propagated from nodes in level i to their child nodes in level $i-1$, such that adjacent level-$(i-1)$ nodes with different parent nodes are also linked.

Note that our octree construction algorithm shares several similarities to the approach by Zhou et al. [2010], like proceeding bottom-up in a breadth-first fashion and using Morton ordering. The main differences are in the stored payload, with Zhou et al. operating on a given point cloud, as well as the adjacency information maintained, where they derive pointers to all 26 neighbors of a node.

5.3 Voxelization into octree

Once the octree is created, we voxelize the input mesh into this sparse structure, allocating one thread per triangle. As in Sec. 4, essentially a 2D rasterization in the yz plane is performed, but instead of flipping all relevant bits in a covered voxel column, we only update the level-0 node overlapped by the triangle and propagate the flip to the other affected voxels in the column in a later stage. For each triangle, we first determine the bounding box and check whether it overlaps any SG voxel center in the yz plane. If so, we derive the yz range of level-0 voxels (tiles) containing overlapped SG voxel centers. Edge functions for overlap testing are then set up as in Sec. 4. Pursuing a two-level approach, we loop over the tiles and check for triangle overlap. If an active tile is identified, we then loop over its 4×4 SG voxel columns, testing their centers for overlap. In case an overlap is reported, we project the center along the x axis onto the triangle's plane and derive the x index $\bar{q}$ of the first SG voxel that needs to be flipped. The level-0 node containing that voxel is identified, and all voxels in the corresponding column of its voxel sub-grid within the local x-index range $(\bar{q} \bmod 4), \ldots, 3$ are flipped via an atomic XOR operation.

To access a node, we initially perform a full descent in the octree from the top level. Within a tile, x-neighbor pointers are then utilized for fast navigation. When entering a new tile, we derive the common ancestor node of the previous and the desired node from their Morton codes. Depending on this ancestor's level, we either ascent to this node and then descent or perform a full descent.

5.4 Hierarchical inside/outside propagation

After the voxelization into the level-0 nodes has been performed, we hierarchically propagate bit flips in +x direction to achieve a solid voxelization (cf. Fig. 7). If the last bit in a level-0 node's SG voxel column is set, all SG voxels in the same column belonging to nodes in +x direction must be flipped. We hence loop over all (even-indexed) level-0 nodes in parallel, identifying those with no level-0 −x neighbor. Starting at such head nodes, we navigate along the +x-neighbor pointers, propagating bit flips to the next node. Note that after that, at the end of such node strings, where no more level-0 nodes abut, all four level-0 nodes with the same level-1 parent have all their SG voxels with local x index 3 in agreement.

We utilize this to propagate the flip information to such level-1 parent nodes. More precisely, we keep a flip propagation flag and an inside flag for each level-1 node and initialize both to zero. If a level-1 node's child is at the end of an x-linked node string and its SG voxels with local x index 3 are set, we set the flip flag. Subsequently, the flip information is propagated along x-linked level-1 node strings, starting at level-1 head nodes. If a node's flip flag is set, we flip both the flip and the inside flag of its +x neighbor.

This propagation to the next coarser level and then within this level is performed analogously for the remaining levels. After that, we propagate the inside flag information from the top level down to the level-0 nodes. For each node of level i that has its inside flag set and has children, we flip these child nodes' inside flags or their SG voxels, respectively. Note that a level-i node without children is inside the solid object if its inside flag is set and outside otherwise.

5.5 Coverage factor computation

Some applications prefer a scalar-valued voxelization where each voxel stores a coverage factor (or a function of it), i.e. the percentage of the voxel covered by the object. These are typically derived from a finer-resolution binary solid voxelization. Our octree-based representation directly supports such scenarios via a simple transformation, offering 64× supersampling. First, we derive coverage factors for level-0 nodes as the fraction of set bits in their voxel sub-grid. Subsequently, coverage factors for level-1 nodes are determined by averaging the coverage factors of their children. If a node has no children, its inside flag defines the coverage factor. This process is applied iteratively for all remaining levels.

6 Implementation

We implemented the described voxelization approaches in CUDA and hence use the according terminology throughout this paper. The methods take an indexed triangle set as input, provided in a vertex and an index buffer. At first, the vertices are always transformed to voxel space and the voxel grid is initialized with zeroes.

Whenever processing entities in parallel where each has a varying amount of work to perform, e.g. tiles looping over triangles or triangles looping over voxels or tiles, we typically employ persistent threads [Aila and Laine 2009], which often yielded considerable performance improvements. For scan and compaction, we utilized `chag::pp` [Billeter et al. 2009] and sorting was done with the radix sort from CUDPP [Satish et al. 2009], processing only as many bits as are significant. Note that recent results show that considerably faster implementations are possible for these parallel primitives; for instance, radix sort can be sped up by a factor of about three [Merrill and Grimshaw 2010].

Atomics Except for the tile-based solid voxelization, all introduced approaches make heavy use of atomic memory operations (rasterization-based voxelization techniques employ the GPU's fixed-function ROP units to set or flip bits). Unfortunately, in current (GT200-based) hardware they are expensive and can easily dominate the runtime if the same memory location is touched by multiple atomics. This, however, is hardly avoidable in conservative voxelization where a voxel is set by all triangles overlapping it, instead of just the one covering its center in a 2D projection (as with conventional rasterization). Some relief is provided by making a warp access triangles with a certain stride, which increases the chance that concurrently executed atomics refer to different memory locations. This is because contiguously stored triangles are typically close in 3D space.

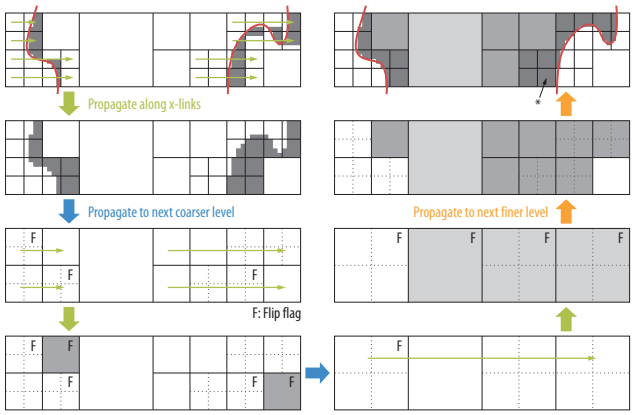


Figure 7: *Illustration of the hierarchical propagation stage.*

However, the new GF100-class GPUs offer significant improvements in this respect. Furthermore, the promotion of atomics to a core feature in OpenCL 1.1 [Khronos OpenCL Working Group 2010] suggests that these important operations will be well supported by future architectures.

7 Results

We tested our methods for multiple models and various voxel grid sizes. All reported results were obtained on an Intel Core 2 Quad 2.83 GHz with an NVIDIA GeForce GTX 285, using isotropic voxels and making the grid closely encompass the whole model. Corresponding preliminary results for an NVIDIA GeForce GTX 480 are provided in the supplemental material, showing speed-ups of up to 16.5 times.

Surface voxelization Table 1 lists performance data for some example models, covering a wide range of triangle counts. Note that the shrub model with its many branches, berries and leaves features a high surface complexity, resulting in a large number of overlapped voxels. We considered three different realizations of our surface voxelization methods: a simple one, which tests all voxels covered by a triangle's bounding box, a specialized one, which determines the case and executes the respective specialization of the overlap test, and one which first sorts the triangles by case before running case-optimized overlap tests. In general, the specialized version turned out to perform fastest and sorting didn't pay off. Interestingly, voxelization is often slower for lower-resolution grids on the GTX 285, probably due to conflicting atomic writes, which is especially noticeable for the Asian dragon. To somewhat quantify the penalty, we replaced atomic operations by according read/modify/write sequences, observing an increase in overall performance by typically 1.5 to 3 times (with a maximum of 13 times).

Compared to pipeline-based conservative voxelization [Zhang et al. 2007], our data-parallel methods are consistently faster, achieving speed-ups of 2.5 to 18.2 times. Moreover, the 6-separating voxelization takes only 56% to 93% of the time of our conservative

Model	Grid size	Pipeline-based conserv. voxel.	Our conservative voxelization					6-separating voxelization			
			Simple	Specialized	Sorted	#voxels	1D/2D/3D	Simple	Specialized	Sorted	#voxels
Stanford bunny (69,666 tris) 	128^3	10.39	1.62	1.65	2.94	54k	10%/49%/41%	1.60	1.50	2.37	33k
	256^3	10.24	3.99	2.51	4.19	222k	0%/20%/79%	3.38	1.98	3.16	133k
	512^3	14.80	11.97	5.96	8.40	900k	0%/ 9%/91%	12.40	4.59	6.26	535k
	1024^3	72.64	60.96	20.69	26.75	3627k	0%/ 3%/97%	58.08	17.09	18.42	2147k
Shrub (751,399 tris) 	128^3	188.16	27.71	25.44	30.39	102k	75%/18%/ 7%	27.13	15.58	18.31	74k
	256^3	236.50	22.57	17.81	23.07	426k	57%/18%/25%	21.37	10.88	13.49	274k
	512^3	336.30	36.10	25.42	28.94	1705k	46%/17%/38%	34.18	15.83	16.61	1022k
	1024^3	804.45	118.23	63.49	65.55	6776k	22%/27%/52%	110.71	40.95	35.82	3924k
Stanford dragon (871,414 tris) 	128^3	116.27	12.67	11.72	16.99	39k	89%/10%/ 1%	12.54	8.91	13.51	25k
	256^3	142.88	9.14	8.26	16.84	162k	64%/28%/ 8%	9.09	6.60	12.84	98k
	512^3	194.15	10.93	10.69	20.67	660k	24%/39%/36%	10.99	8.15	15.55	387k
	1024^3	331.81	24.17	23.00	32.18	2664k	8%/24%/68%	23.74	16.24	23.45	1539k
XYZ RGB Asian dragon (7,218,906 tris) 	128^3	1261.99	234.04	227.37	192.17	22k	99%/ 1%/ 0%	230.13	153.94	187.02	16k
	256^3	1496.62	111.11	106.63	134.08	91k	97%/ 3%/ 0%	108.87	76.89	113.59	61k
	512^3	—	73.78	68.99	130.76	374k	88%/11%/ 1%	74.55	54.70	97.29	233k
	1024^3	—	92.48	85.26	141.11	1516k	60%/32%/ 8%	95.28	66.59	105.32	897k

Table 1: *Running time (in ms) for different surface voxelization methods, along with the number of resulting voxels and the encountered percentage of cases with a 1D, 2D and 3D bounding box. For comparison, the pipeline-based approach by Zhang et al. [2007] is included.*

voxelization, mainly thanks to setting 26% to 42% fewer voxels.

Solid voxelization As the results in Table 2 show, the tile-based method becomes faster than the triangle-parallel approach if the grid size is high or the model features many triangles. In these cases, the overhead of assigning triangles to tiles and always testing all voxel columns in a tile for overlap is offset by the saving in memory bandwidth and the avoidance of atomic operations. Also note the effectiveness of the tile assignment stage in skipping triangles that don't overlap a tile's voxel column centers. As an extreme example, less than 0.3% of the (typically tiny) triangles are put in the work queue for the Asian dragon and a 128^3 grid.

When compared against the fastest pipeline-based version [Eisemann and Décoret 2008], at least one of our methods is faster in half of the listed cases, which is encouraging. Considering that solid voxelization is essentially a rasterization problem, this means we manage to outperform the hardware rasterizer and the blend stage for a task they were designed for. Note that as the rasterizer processes triangles sequentially, the voxelization time can be dominated by the fixed setup cost per triangle. Consequently, our data-parallel approaches excel for smaller grid sizes and models with a high triangle count.

Sparse solid voxelization Table 3 reveals that our octree-based solid voxelization is indeed very sparse. In particular, we are able to represent a voxelization covering a grid of size 4096^3 with just 216 MB (bunny model), comprising both the data and the octree structure. This is just 2.6% of the 8 GB that would be required for representing the grid explicitly. Note that our memory consumption can be even further decreased by storing the x-neighbor pointers separately and getting rid of them after the voxelization process.

With respect to our full solid voxelization methods, the running time of the sparse approach compares favorable for the bunny model, performing even twice as fast for a 1024^2 grid. By contrast, it turns out to be more expensive for models with a higher triangle count, but it additionally provides an octree representation and can support larger grid sizes. The timing break-down indicates that the determination of active level-1 nodes is rather costly and can even clearly dominate the overall time (Asian dragon). Again, this mainly can be traced back to the heavy use of atomic operations.

Model	Grid size	Pipe-line	Tri-parall.	Tile-based					
				T_{total}	T_{pairs}	T_{sort}	T_{tiles}	#pairs	#tris
Stanford bunny	128^3	0.52	0.48	1.53	0.38	0.32	0.48	42k	64.2
	256^3	0.58	1.48	2.02	0.39	0.44	0.80	81k	32.2
	512^3	1.43	9.27	4.72	0.49	0.78	2.88	145k	14.6
	1024^3	8.75	74.86	18.24	0.76	1.39	14.08	301k	7.6
Stanford dragon	128^3	2.69	1.45	2.51	1.09	0.32	0.47	40k	81.8
	256^3	2.73	2.03	4.72	1.21	0.64	1.11	156k	84.7
	512^3	3.16	7.57	7.19	1.41	1.96	2.91	476k	67.3
	1024^3	13.56	51.95	16.41	1.79	3.54	8.76	944k	34.0
Asian dragon	128^3	18.30	8.84	7.70	5.06	0.19	0.44	21k	81.5
	256^3	18.35	8.82	8.73	5.48	0.44	0.75	84k	89.1
	512^3	18.52	13.77	12.63	6.66	1.47	2.18	342k	96.3
	1024^3	20.46	50.94	24.26	7.66	4.91	7.92	1366k	99.6

Table 2: *Running time (in ms) for different solid voxelization methods. The break-down for the tile-based approach considers determining the work queue (T_{pairs}), sorting it (T_{sort}), and processing the tiles (T_{tiles}). Moreover, the number of tile/triangle pairs in the queue and the average triangle list length per active tile are provided.*

8 Conclusion

We have presented data-parallel algorithms for both surface and solid voxelization. Our conservative voxelization method outperforms previous GPU-based approaches by up to one order of magnitude, making it accessible to real-time applications. It employs a new triangle/box overlap test that should also be useful in other domains. Moreover, our algorithm easily can be adapted to a different overlap criterion, as we have shown by introducing a fast 6-separating surface voxelization method.

Two strategies for solid voxelization have been demonstrated. Their underlying approaches offer data-parallel alternatives to the hardware rasterizer and compare favorable in many cases. In particular, our tile-based method provides an efficient solution for rasterization tasks where larger amounts of data may be updated per fragment (e.g. color and depth, which cannot be done with a single atomic operation).

Finally, an octree-based sparse voxelization approach has been introduced that offers a compact hierarchical representation and allows voxelizations of resolutions not possible before on GPUs. It basically constitutes an alternative, domain-adapted, custom rendering pipeline that surpasses previous limitations, enabling the direct rendering into a sparse spatial data structure. An interesting avenue for future work is applying this strategy to other tasks where the standard rasterization pipeline proves too restrictive.

Acknowledgements

The (original) bunny and dragon models are courtesy of the Stanford 3D scanning repository; the shrub model is from the Xfrog public plants library.

Model	Stanford bunny				Stanford dragon				XYZ RGB Asian dragon			
Binary grid size	512^3	1024^3	2048^3	4096^3	512^3	1024^3	2048^3	4096^3	512^3	1024^3	2048^3	4096^3
Scalar-valued grid size	128^3	256^3	512^3	1024^3	128^3	256^3	512^3	1024^3	128^3	256^3	512^3	1024^3
Total	4.87	9.15	25.17	93.27	14.50	23.59	37.71	100.32	47.71	102.96	143.90	178.40
Determine active level-1 nodes	1.49	1.92	3.60	12.74	8.07	10.63	9.02	17.79	27.07	80.91	101.09	77.79
Construct octree	1.01	1.74	3.99	12.36	0.97	1.52	3.37	9.43	0.90	1.35	2.37	6.04
Voxelize into octree	1.47	3.43	10.65	42.61	4.45	9.59	20.57	55.36	17.35	17.96	35.81	82.55
Propagate inside/outside	0.44	1.36	5.12	19.71	0.32	0.92	3.28	13.16	0.23	0.56	1.98	7.54
Compute coverage factor	0.12	0.37	1.37	5.36	0.10	0.28	1.02	3.98	0.07	0.18	0.60	2.29
Stored level-0 nodes	5.30%	2.71%	1.37%	0.69%	3.82%	2.00%	1.01%	0.51%	2.02%	1.10%	0.57%	0.29%
Octree size	3.2 MB	13.2 MB	53.7 MB	216 MB	2.3 MB	9.7 MB	39.6 MB	159 MB	1.2 MB	5.3 MB	22.3 MB	90.8 MB

Table 3: *Running time (in ms) for our sparse solid voxelization method, including a break-down. The percentage of explicitly stored level-0 nodes provides a measure of sparseness. The given octree size accounts for both the structure and the data.*

References

Abrash, M. 2009. Rasterization on Larrabee. *Dr. Dobb's*. http://www.drdobbs.com/high-performance-computing/217200602.

Aila, T., and Laine, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics 2009*, 145–149.

Akenine-Möller, T., and Aila, T. 2005. Conservative and tiled rasterization using a modified triangle set-up. *Journal of Graphics Tools 10*, 3, 1–8.

Akenine-Möller, T. 2001. Fast 3D triangle-box overlap testing. *Journal of Graphics Tools 6*, 1, 29–33.

Billeter, M., Olsson, O., and Assarsson, U. 2009. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of High Performance Graphics 2009*, 159–166.

Cohen-Or, D., and Kaufman, A. 1995. Fundamentals of surface voxelization. *Graphical Models and Image Processing 57*, 6, 453–461.

Dong, Z., Chen, W., Bao, H., Zhang, H., and Peng, Q. 2004. Real-time voxelization for complex models. In *Proceedings of Pacific Graphics 2004*, 43–50.

Eisemann, E., and Décoret, X. 2006. Fast scene voxelization and applications. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2006*, 71–78.

Eisemann, E., and Décoret, X. 2008. Single-pass GPU solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, 73–80.

Eisenacher, C., and Loop, C. 2010. Data-parallel micropolygon rasterization. In *Eurographics 2010 Short Papers*, 53–56.

Fang, S., and Chen, H. 2000. Hardware accelerated voxelization. *Computers & Graphics 24*, 3, 433–442.

Fatahalian, K., Luong, E., Boulos, S., Akeley, K., Mark, W. R., and Hanrahan, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of High Performance Graphics 2009*, 59–68.

Haines, E. A., and Wallace, J. R. 1991. Shaft culling for efficient ray-cast radiosity. In *Proceedings of Eurographics Workshop on Rendering 1991*, 122–138.

Hasselgren, J., Akenine-Möller, T., and Ohlsson, L. 2005. Conservative rasterization. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley Professional, ch. 42, 677–690.

Huang, J., Yagel, R., Filippov, V., and Kurzion, Y. 1998. An accurate method for voxelizing polygon meshes. In *Proceedings of IEEE Symposium on Volume Visualization 1998*, 119–126.

Ivson, P., Duarte, L., and Celes, W. 2009. GPU-accelerated uniform grid construction for ray tracing dynamic scenes. Tech. Rep. 14/09, Pontifícia Universidade Católica do Rio de Janeiro.

Kalojanov, J., and Slusallek, P. 2009. A parallel algorithm for construction of uniform grids. In *Proceedings of High Performance Graphics 2009*, 23–28.

Khronos OpenCL Working Group. 2010. *The OpenCL Specification*. Version: 1.1.

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. 2009. Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2, 375–384.

Li, W., Fan, Z., Wei, X., and Kaufman, A. 2005. Flow simulation with complex boundaries. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley Professional, ch. 47, 747–764.

Liu, F., Huang, M.-C., Liu, X.-H., and Wu, E.-H. 2010. FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2010*, 75–82.

Merrill, D., and Grimshaw, A. 2010. Revisiting sorting for GPGPU stream architectures. Tech. Rep. CS2010-03, Department of Computer Science, University of Virginia.

Nichols, G., Penmatsa, R., and Wyman, C. 2010. Interactive, multiresolution image-space rendering for dynamic area lighting. *Computer Graphics Forum 29*, 4, 1279–1288.

Pineda, J. 1988. A parallel algorithm for polygon rasterization. *Computer Graphics (Proceedings of SIGGRAPH 88) 22*, 4, 17–20.

Reinbothe, C. K., Boubekeur, T., and Alexa, M. 2009. Hybrid ambient occlusion. In *Eurographics 2009 Annex (Areas Papers)*, 51–57.

Satish, N., Harris, M., and Garland, M. 2009. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium 2009*, 1–10.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3, 18:1–18:15.

Sun, X., Zhou, K., Stollnitz, E., Shi, J., and Guo, B. 2008. Interactive relighting of dynamic refractive objects. *ACM Transactions on Graphics 27*, 3, 35:1–35:9.

Zhang, L., Chen, W., Ebert, D. S., and Peng, Q. 2007. Conservative voxelization. *The Visual Computer 23*, 9–11, 783–792.

Zhou, K., Hou, Q., Wang, R., and Guo, B. 2008. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics 27*, 5, 126:1–126:9.

Zhou, K., Gong, M., Huang, X., and Guo, B. 2010. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*. To appear.