# Efficient Voxelization Using Projected Optimal Scanline

Yumin Zhang[1], Steven Garcia[1], Weiwei Xu[2], Tianjia Shao[2] and Yin Yang[1]
[1]Department of Electrical and Computer Engeneering, the University of New Mexico, New Mexico, U.S.A
[2]State Key Lab of CAD&CG, Zhejiang University, China

**Abstract**

In the paper, we propose an efficient algorithm for the surface voxelization of 3D geometrically complex models. Unlike recent techniques relying on triangle-voxel intersection tests, our algorithm exploits the conventional parallel-scanline strategy. Observing that there does not exist an optimal scanline interval in general 3D cases if one wants to use parallel voxelized scanlines to cover the interior of a triangle, we subdivide a triangle into multiple axis-aligned slices and carry out the scanning within each polygonal slice. The theoretical optimal scanline interval can be obtained to maximize the efficiency of the algorithm without missing any voxels on the triangle. Once the collection of scanlines are determined and voxelized, we obtain the surface voxelization. We fine tune the algorithm so that it only involves a few operations of integer additions and comparisons for each voxel generated. Finally, we comprehensively compare our method with the state-of-the-art method in terms of theoretical complexity, runtime performance and the quality of the voxelization on both CPU and GPU of a regular desktop PC, as well as on a mobile device. The results show that our method outperforms the existing method, especially when the resolution of the voxelization is high.

*Keywords:* 3D voxelization, Scanline, Integer arithmetic, Bresenham's algorithm.

## 1. Introduction

Real world geometries have a diverse range of forms and shapes, usually consisting of various kinds of primitives like lines, triangles, polygons, curved surfaces, etc. In order to visualize, animate, render and analyze such geometries with digital computers, a discrete representation is essential. Voxelization, as one of the most widely used discretizing approaches, converts a continuous geometry into a set of volumetric pixels or *voxels* which best approximates the original shape. Voxelization plays a fundamental role in computer graphics, and it often stands as an important geometric preprocessing step in many applications, such as virtual reality [1], medical imaging/visualization [2], global rendering [3], collision detection [4, 5, 6], computer animation or simulation [7, 8, 9, 10, 11], and other interesting areas [12]. A *surface voxelization* of a 3D model produces a set of boxes/voxels that encapsulates its geometric boundary, which is often represented as a triangle mesh in computer graphics. We evaluate voxelization algorithms by their *efficiency*, *accuracy*, *separability* and *minimality*, following the framework of Cohen-Or and Kaufman [13]. Many existing voxelization algorithms [14] [15] are based on *overlap tests*, wherein every potential voxel candidate undergoes a sequence of tests to determine whether it intersects a triangle. These methods produce *super covers* of input models (see Sec. 3 for a quick terminology review). A drawback of the methods is that the triangle-voxel intersec-

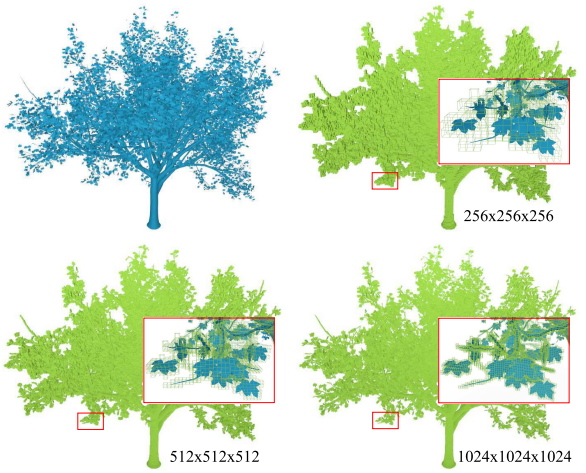



Figure 1: The results of the voxelization of a tree model (842K triangles) using the proposed method under the resolutions of 256, 512 and 1024 respectively.

tion test could be relatively expensive, compared to the *scan-conversion* algorithms [16, 17, 18]. The 3D scan-conversion algorithms are extensions of their 2D counterparts (i.e. the famous Bresenham's algorithm [19] and very efficient. However, due to the complication of 3D scenarios, the resulting voxelization produced by these algorithms is only a subset of the cover of the original model.

In this paper, we propose a new algorithm that can produce a cover using parallel scanlines. A cover is *N-tunnel-free*, which is an important feature of a high-quality surface voxelization. The biggest challenge in our method is to determine an appropriate set of parallel scanlines. The scanlines should not miss any voxel on the cover, and the distance between two consecutive scanlines or the *scanline interval* (SI) should be optimal in order to achieve high efficiency. We will show that there does not exist a "gold standard" allowing us to set a constant SI in a general 3D case. Instead, our method subdivides an input triangle into multiple axis-aligned slices. The voxelization of each slice degenerates to a 2D case. We derive a theoretically optimal SI for setting up the scanlines for each slice (see detailed explanation in Sec. 4.2). We further derive an integer-only version of the algorithm with minor accuracy compromise, and an enhanced version for generating the super cover. We test our algorithm on various 3D models with both CPU and GPU with a desktop computer, as well as the mobile platform of IOS device (an Apple iPhone 6). Experiments, for example, voxelization of a tree model (Fig. 1), show that the proposed method presents a good performance, especially for a high-resolution voxelization. To the best of our knowledge, our method is the first one to generate a N-tunnel-free cover or the super cover using the scanline strategy.

## 2. Related Work

The *separating axis theorem* (SAT) [20, 21] provides a general guideline for an overlap test between two convex polygons: the triangle-box overlap test is simply a base case in the SAT [22]. Akenine-Möller [23] adopted a standard triangle-box overlap test following the SAT, which consists of 13 sub-tests: three for the box against the minimal box of the triangle, one for overlap test between the box and the plane determined by the triangle, and nine for the projections of the triangle against the box. The triangle intersects the box if it passes all the tests. In a recent contribution, Schwarz and Seidel [14] provided the *sufficient and necessary condition* of the box-triangle intersection tests: 1) the box intersects the triangle's plane; and 2) the projections of the box and the triangle overlap on all of the three coordinate planes. They use nine edge functions to evaluate the second condition, which essentially correspond to the nine sub-tests in [23]. They also improved the algorithm by reducing the number of candidate voxels and skipping unnecessary tests based on the observation that a triangle is of at most three-voxel thickness in its *dominant axis* direction of the triangle's normal. Based on their method, Pantaleoni [15] further reduced the number of candidate voxels in the inner loop of the 2D projection overlap test by computing a tighter bound. Crassin and Green [24] presented a simple voxelization pipeline basically following the work in [14, 15]. The resulting voxelization is not a correct 6-tunnel-free one, because only the coverage of the center of each voxel is tested against the triangles to generate fragments. They employed the idea in [25] to fix the problem. However, the method can not generate super covers because the voxels captured at the triangle edges could be redundant.

Overlap tests form the basis of many existing rasterization algorithm. McCool and colleagues [26] examined four corners of a pixel tile against each edge of a triangle by evaluating the signs of its three edge functions: a point is inside a triangle if and only if all of the three edge functions at the point are positive. A conservative 2D rasterization algorithm was presented in [27, 28], which modified the triangle setup and selected a different evaluation point to reduce the computation. Haines and Wallace [29] observed that the overlap test between a box and a plane can be done by projecting the box to the diagonal that best aligns with the plane normal, and only two corresponding corners would be involved in the test. It is suggested that if the entire box is in either the positive or the negative half-plane, only one of the box corners needs to be tested.

Instead of using overlap tests, Huang and colleagues [30] voxelized a surface by evaluating a distance threshold. If the distance from voxel center to the triangle is smaller than a certain threshold value, an overlap is determined. Varadhan and colleagues [31] presented an algorithm computing the *max-norm* distance between voxels and other geometric primitives, which was formulated as an optimization problem. Brimkov and Barneva [32] presented a nice surface voxelization namely the *graceful plane*, which is 6-tunnel-free and jump free. Graceful planes are the thinnest possible discrete voxelizations in which any geometry primitives are connected sets of voxels. Fei and colleagues [33] proposed a point-tessellated voxelization method. Taking advantage of the powerful tessellator in GPU hardware, the method efficiently generates watertight voxelizations. The resulting voxelizations approximate the

ground truth (i.e. super covers) well and are suitable for applications where accuracy is not the major concern, such as video games and virtual realities. Chao and colleagues [34] set up a set of sampling points of the input triangle. The voxelization of the triangle is simply the union of all the point voxelization. The method also produces an approximation and does not guarantee to generate a cover.

The 3D/2D line voxelization algorithms lie in the most inner loop of our algorithm and must be efficent. Liu and Chen [35] extended 2D Bresenham's algorithm for 3D line segments. Au and Woo [36] investigated the 3D Bresenham's algorithm using the Voronoi diagram. Our method borrows the ideas in these 3D line rasterization techniques, which generate the line voxelization by incrementally evaluating stepping parameters along the straight line segment to determine the corresponding voxel sequence [37, 38, 39]. We improve this algorithm so that only integer operations are involved, which could further speed up the computation with the support of dedicated hardware [40, 41].

As a fundamental algorithm, voxelization has been extensively implemented/tested on modern GPUs architectures. Dong and colleagues [42] proposed a fast voxelization algorithm on the GPU for complex polygon models, which achieved a real-time frame rate. GPU-accelerated algorithm [43, 44] as well as GPU-oriented data structures [45, 46] have been researched in order to optimally utilize the hardware resources. Our method is parallelizable and has been implemented using `nVidia CUDA` in our experiments.

## 3. Terminology Review

To make the paper self-contained, we briefly review some useful terminologies [13, 14, 38, 47] regarding the properties and quality of a voxelization. The generated voxel is assumed to have a unit size in all of its $x$, $y$ and $z$ directions, and we use $\mathbb{Z}^3$ to denote the set consisting of all the integer coordinates or *grid points* corresponding to the centers of all the voxels. Hereinafter, we use bold lowercase letters, e.g. $\mathbf{p}(p_x, p_y, p_z)$ to denote a point defined in $\mathbb{R}^3$ with real coordinates $p_x$, $p_y$ and $p_z$, and bold uppercase letters like $\mathbf{P}(P_x, P_y, P_z)$ to denote a grid point or a voxel with integer indices of $P_x$, $P_y$ and $P_z$ in $\mathbb{Z}^3$.

Two neighboring voxels are *26-adjacent* if they share a face, an edge or a corner. Similarly, two voxels are *18-adjacent* if they are connected by a face or an edge, or *6-adjacent* if connected by just a face. An *N-path* is a voxel sequence in which any two consecutive voxels are $N$-adjacent, for $N \in \{6, 18, 26\}$. By connecting the centers of every two adjacent voxels along an $N$-path, we obtain its *polygon arc*. Let $\mathcal{S}$ be a continuous surface patch and $\mathcal{V}$ be its voxelization. We say that an $N$-path penetrates $\mathcal{V}$ if its polygon arc passes through $\mathcal{S}$. If there does not exist an $N$-path in $\mathbb{Z}^3 \setminus \mathcal{V}$ penetrating $\mathcal{V}$, $\mathcal{V}$ is called *N-tunnel-free*. If $\mathcal{S}$ is completely encapsulated by $\mathcal{V}$, and every voxel in $\mathcal{V}$ is intersecting $\mathcal{S}$, $\mathcal{V}$ is called a *cover* of $\mathcal{S}$. Intuitively, a cover with $N$-tunnel-free property is a *good* voxelization of the surface. [1]

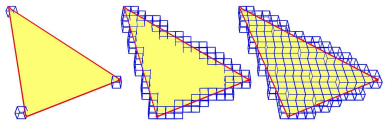


Figure 2: The voxelization of vertices, edges and the interior area of a triangle.

## 4. Scanline-based Voxelization

As shown in Fig. 2, our algorithm consists of three steps, namely the voxelization of triangles' vertices, edges, and the interior. Our voxelization is 26-tunnel-free, and it can be downgraded to 18- or 6-tunnel-free to generate thinner voxelizations if necessary. The first step of vertex or point voxelization is trivial. Given a point $\mathbf{p}_0(x_0, y_0, z_0)$, its corresponding voxel indices can be easily obtained as $\mathbf{P}_0(\lfloor x_0+1/2 \rfloor, \lfloor y_0+1/2 \rfloor, \lfloor z_0+1/2 \rfloor)$. Next, we will detail the second and the third step of how to voxelize the edges and the interior of an input triangle.

### 4.1. Line Voxelization

Our line voxelization is based upon the work of Amanatides and Woo [37], which efficiently generates a 6-path line voxelization. We further generalize this method to an integer-only version so that only integer operations are needed.

*RLV: regular line voxelization.* Assume that two ends of the line segment, given by $\mathbf{p}_0(x_0, y_0, z_0)$ and $\mathbf{p}_1(x_1, y_1, z_1)$, are contained by the voxels $\mathbf{P}_0(X_0, Y_0, Z_0)$ and $\mathbf{P}_1(X_1, Y_1, Z_1)$ respectively. Let $\mathbf{v} = [v_x, v_y, v_z]^\top$ be a unit vector directing from $\mathbf{p}_0$ to $\mathbf{p}_1$. If we cast a ray from $\mathbf{p}_0$ to $\mathbf{p}_1$, this ray will first intersect a facet of $\mathbf{P}_0$, and the voxel adjacent to this intersected facet will be marked as part of the final voxelization. As the ray moves forward, it will intersect $|X_1 - X_0|$ $yz$ facets, $|Y_1 - Y_0|$ $xz$

[1] Strictly speaking, there are small differences between concepts of *cover*, *super cover* and *minimum cover* as discussed in existing literature [13]. Think of a simple 3D point coinciding with a voxel's corner. A super cover will be all the eight voxels incident to the point. Any one of these eight voxels is a minimum cover, and any subset of these eight voxels is a cover. For a closed triangularized manifold, however it is not practically necessary to differentiate these minor differences.

facets, and $|Z_1 - Z_0|$ $xy$ facets before it reaches $\mathbf{p}_1$. We define the $x-$direction *distance function* $l^x(i)$ as the distance along the ray from $\mathbf{p}_0$ to the $i^{\text{th}}$ $yz$ facet, where $0 \le i \le |X_1 - X_0|$. The projection of $l^x(i)$ on the $x$ axis is $l^x(i) \cdot |v_x|$. Recalling that the dimension of a voxel $h$ equals to 1, we have $l^x(i) \cdot |v_x| = d_0^x + i$ or:

$$l^x(i) = \frac{d_0^x}{|v_x|} + \frac{i}{|v_x|}, \tag{1}$$

where $d_0^x$ is the distance (along $x$ axis) between $\mathbf{p}_0$ and its first intersecting $yz$ facet, which can be evaluated as:

$$d_0^x = \begin{cases} X_0 - x_0 + \frac{1}{2} & \text{if} \quad v_x > 0 \\ \frac{1}{2} - (X_0 - x_0) & \text{if} \quad v_x < 0 \end{cases}. \tag{2}$$

If $v_x = 0$, we have $l^x(i) \to \infty$ which means the ray will never hit a $yz$ facet and the line voxelization degenerates to 2D.

In each iteration, we track the *step distances*, denoted by $l^x$, $l^y$, and $l^z$, from the current voxel to the next $yz$, $xz$, and $xy$ facet. Let $m^x = 1/|v_x|$, $m^y = 1/|v_y|$, and $m^z = 1/|v_z|$ denote the slopes of three distance functions. Assume that at certain step $j$, $l^x = l_{min} \triangleq \min\{l^x, l^y, l^z\}$, indicating that the ray will intersect a $yz$ facet and $\mathbf{P}'(X_j + \Delta X, Y_j, Z_j)$ will be marked next. Three step distance variables for the next step can be updated incrementally as: $l^x \leftarrow l^x - l_{min} + m^x = m^x$, $l^y \leftarrow l^y - l_{min}$, and $l^z \leftarrow l^z - l_{min}$. This process stops when the ray reaches $\mathbf{p}_1$, where the last voxel generated will be $\mathbf{P}_1(X_1, Y_1, Z_1)$.

The minimum step distances are not unique if the line segment intersects a voxel at one of its edges or corners, which is referred to as a *singular point*. We can either pick an arbitrary voxel candidate with minimum step distance or pick all the voxels incident to the singular point. In the latter case, the corresponding line voxelization forms a super cover and we call this modified method *super-cover line voxelization* (SLV). Fig. 3 (a) and (b) show the results of RLV and SLV.

*ILV: integer-only line voxelization.* The integer-only line voxelization eliminates runtime floating-point arithmetics in RLV. To do so, we use the grid points of $\mathbf{P}_0$ and $\mathbf{P}_1$ to approximate $\mathbf{p}_0$ and $\mathbf{p}_1$, and $d_0^x$, $d_0^y$, $d_0^z$ in Eq. (2) equal to $1/2$. Let $\mathbf{v}$ be $[X_1 - X_0, Y_1 - Y_0, Z_1 - Z_0]^\top$, and the distance functions are re-written as:

$$\begin{cases} l^x(i) = \dfrac{1}{|X_1 - X_0|} i + \dfrac{1}{2|X_1 - X_0|} \\ l^y(i) = \dfrac{1}{|Y_1 - Y_0|} i + \dfrac{1}{2|Y_1 - Y_0|} \\ l^z(i) = \dfrac{1}{|Z_1 - Z_0|} i + \dfrac{1}{2|Z_1 - Z_0|} \end{cases}. \tag{3}$$

It is noteworthy that true values of distance functions are of less interest, as we only need the relative order among $l^x$, $l^y$, and $l^z$ to determine the next voxel. Therefore, we multiply both sides of Eq. (3) by a scaling factor $s = 2|X_1 - X_0||Y_1 - Y_0||Z_1 - Z_0|$ resulting in three integer-valued distance functions denoted by $L^x(i) = s \cdot l^x(i)$, $L^y(i) = s \cdot l^y(i)$, and $L^z(i) = s \cdot l^z(i)$ such that:

$$\begin{cases} L^x(i) = 2M^x i + M^x \\ L^y(i) = 2M^y i + M^y \\ L^z(i) = 2M^z i + M^z \end{cases}, \tag{4}$$

where $M^x = |Y_1 - Y_0||Z_1 - Z_0|$, $M^y = |X_1 - X_0||Z_1 - Z_0|$ and $M^z = |X_1 - X_0||Y_1 - Y_0|$ are all integers. Then, we can utilize three integers $L^x = s \cdot l^x$, $L^y = s \cdot l^y$ and $L^z = s \cdot l^z$ as the integer-counterparts of $l^x$, $l^y$ and $l^z$ to determine the voxels to be generated along the ray. ILV is an approximation of RLV as it forces the ends of a line segments to be at the grid points.

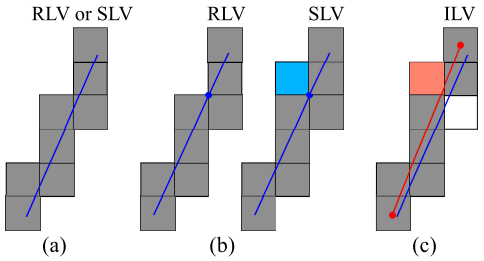


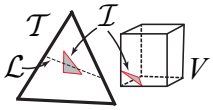

Figure 3: (a) RLV and SLV generate identical voxelization for a general line segment without singular points. (b) SLV captures more voxles than RLV to form a super cover if the line segment contains a singular point. (c) Small variations of the voxels generated by ILV and RLV due to the voxel center approximation.

### *4.2. Triangle Voxelization*

Triangle voxelization targets the interior of an input triangle $\mathcal{T}$, and outputs a 26-tunnel-free set of voxels $\mathcal{V}$ that completely encapsulates $\mathcal{T}$. Unlike SAT-based methods, we do not perform excessive overlap tests for voxels residing in the bounding box of $\mathcal{T}$. Instead, we carefully form a set of scanline segments: the voxelization of each scanline can be obtained with RLV/ILV, and the superset of all the scanline voxelizations will be the final output of this procedure. While the intuition could lead one to select a set of parallel scanline segments to cover the triangle, it turns out that there does not exist an optimal scanline interval (SI) for the general 3D case that guarantees not missing any voxels on the cover. Thus, the performance of a 3D scanline method is often unsatisfying and slower than SAT-based methods. In this section, we show that such technical challenge can be resolved by projecting $\mathcal{T}$ onto axis-aligned slices.

We show that an optimal scanline interval is available within a 2D slice, which guides us to set the most aggressive scan strategy slice by slice. Finally, we give an integer version of this method, with minor compromises of the scanning optimality using integer-based scanline intervals.

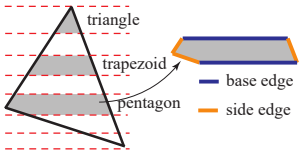
*Scanline interval in 3D.* Let $V$ be a voxel intersecting $\mathcal{T}$ as shown on right. We use $\mathcal{I}$ to denote the intersecting region on $\mathcal{T}$ such that $\mathcal{I} = \mathcal{T} \bigcap V$. Clearly, a 26-tunnel-free voxelization of $\mathcal{T}$ must include $V$. We can also learn from the line voxelization procedure that the voxelization of a scanline $\mathcal{L}$ includes $V$ only if $\mathcal{L} \bigcap V \neq \varnothing$. As the scanline is also on the triangle i.e. $\mathcal{L} \subset \mathcal{T}$, $\mathcal{L} \bigcap \mathcal{I}$ is also non-empty:




$$\left.\begin{array}{l} \mathcal{T} \bigcap V = \mathcal{I} \\ \mathcal{L} \bigcap V \neq \varnothing \\ \mathcal{L} \subset \mathcal{T} \end{array}\right\} \Rightarrow \mathcal{L} \bigcap \mathcal{I} \neq \varnothing.$$

In other words, the scanline $\mathcal{L}$ must intersect $\mathcal{I}$ as well, in order to produce voxel $V$. As the intersecting region $\mathcal{I}$ could approach to an infinitesimally small area, any finite scanline interval will miss $\mathcal{I}$ in the resulting voxelization. Therefore, we say there does NOT exist an optimal SI in 3D, and one has to resort to the 2D projection to solve this problem.

*Triangle splicing.* We first determine $\mathcal{T}$'s *dominant direction* – the coordinate axis that best aligns with the triangle's normal. Without loss of generality, assume that $z$ axis is its dominant direction, and we reorder $\mathcal{T}$'s three vertices $\mathbf{p}_0(x_0, y_0, z_0)$, $\mathbf{p}_1(x_1, y_1, z_1)$ and $\mathbf{p}_2(x_2, y_2, z_2)$ such that $z_0 \leq z_1 \leq z_2$. Their voxelizations are denoted with $\mathbf{P}_0(X_0, Y_0, Z_0)$, $\mathbf{P}_1(X_1, Y_1, Z_1)$ and $\mathbf{P}_2(X_2, Y_2, Z_2)$ respectively. Afterwards, $\mathcal{T}$ is sliced into a set of polygons $\{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_{Z_2-Z_0+1}\}$ along the $z$ axis by a series of $xy$ planes $\{\mathcal{S}_0, \mathcal{S}_1, ..., \mathcal{S}_{Z_2-Z_0+1}\}$. $\mathcal{T}_i$ must be convex, and it could be either a trapezoid, a triangle or a pentagon as shown here. Regardless of its geometric variations, we can always group $\mathcal{T}_i$'s edges into *side edges* and *base edges*. The side edges are the ones coincide with the original edges of $\mathcal{T}$, while the base edges, are the intersections between $\mathcal{T}_i$ and $\mathcal{S}_i$. The $Z$ index of the $i^{\text{th}}$ plane $\mathcal{S}_i$ is $Z_0 + i + 1/2$. It is clear that each sliced polygon $\mathcal{T}_i$ is restricted to within one-voxel-thickness along the $z$ axis. Therefore, the voxelization of $\mathcal{T}_i$ degenerates to a 2D case with a fixed $Z$ index.



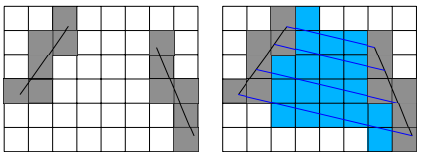



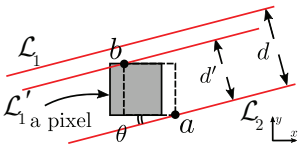
Figure 4: A set of parallel scanline is formed to cover the interior of $\mathcal{T}_i$. The scanline interval is set to be $\widehat{d}$.

*Optimized parallel scanline.* For a given $\mathcal{T}_i$, our scan starts with one of its base edges as shown in Fig. 4. As discussed, we seek an as-sparse-as-possible scan if the resulting voxelization remains 26-tunnel-free. Hereby, we provide an upper bound of the distance between two consecutive scanlines in 2D.

**Theorem 4.1.** *In 2D, there does not exist a voxel between a pair of parallel lines that does not intersect either of them if the distance d between the lines satisfies the following scanning condition:*

$$d \leq \widehat{d}, \qquad \widehat{d} = h \cdot (\sin\theta + \cos\theta), \tag{5}$$

*where h is the dimension of the voxel ($h = 1$ in our case) and $\theta$ is the smallest angle between the lines and the voxel's edges.*




*Proof* Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be the parallel lines and assume their distance satisfies Eq. (5): $d \leq \widehat{d}$. Assume that there exists a voxel between $\mathcal{L}_1$ and $\mathcal{L}_2$ intersecting neither of them as shown on the left. We can move the voxel along the $x$ axis toward $\mathcal{L}_2$ for a finite amount of distance until it hits $\mathcal{L}_2$ at one of its corners $a$. Then, there must exist another line $\mathcal{L}'_1$ parallel to both $\mathcal{L}_1$ and $\mathcal{L}_2$ passing through corner $b$ of the voxel, which is diagonal to $a$. The distance $d'$ between $\mathcal{L}'_1$ and $\mathcal{L}_2$ is: $d' = \sqrt{2}h \cdot \sin(\theta + \pi/4) = h \cdot (\sin\theta + \cos\theta) = \widehat{d}$. Since $\mathcal{L}'_1$ lies between $\mathcal{L}_1$ and $\mathcal{L}_2$, we have $d > d'$, which contradicts our assumption. ■

Theorem 4.1 suggests that as long as the scanning condition is satisfied, no missing voxels will be produced, and the resulting voxelization is guaranteed to be a cover of $\mathcal{T}_i$. On the other hand, if the scanline interval exceeds $\widehat{d}$, the resulting voxelization $\mathcal{V}_i$ is no longer 26-tunnel-free, and we can clearly see voxels missed as shown in Fig. 5. The voxelization of the entire triangle $\mathcal{V}$, is the union of the voxelization of each $\mathcal{T}_i$: $\mathcal{V} = \bigcup \mathcal{V}_i$. Because $\mathcal{T}_i$ and $\mathcal{T}_{i+1}$ always share a base edge, $\mathcal{V}_i$ overlaps $\mathcal{V}_{i+1}$ at voxels corresponding to this shared edge. Thus, $\mathcal{V}_i \bigcup \mathcal{V}_{i+1}$ is also 26-tunnel-free and so is $\mathcal{V}$.

*Integer scanline.* While the parallel scanline method is theoretically optimal, it needs to compute its intersec-

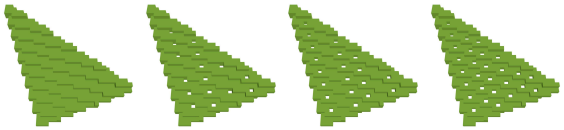


Figure 5: From left to right: the resulting voxelization of a triangle with scanline intervals of $\widehat{d}$, $1.05\widehat{d}$, $1.1\widehat{d}$ and $1.2\widehat{d}$.

tions between the side edges of $\mathcal{T}_i$ for each scanline in order to determine the starting and ending locations of the scanline voxelization. The previous voxelization of $\mathcal{T}$'s edges is completely disregarded. In addition, floating number arithmetic is unavoidable as the optimal interval $\widehat{d}$ itself is a floating number. Similar to ILV, we further tweak the parallel scanline algorithm and provide its integer-only counterpart, which requires only an incremental integer arithmetic at each step.

Let $\mathcal{V}_A$ and $\mathcal{V}_B$ be the voxelizations of two side edges $e_A$ and $e_B$ of $\mathcal{T}_i$, which contain two sets of ordered voxels along $e_A$ and $e_B$ respectively[2]. We restrict the starting and ending points, referred as end $A$ and end $B$, of a scanline to be the grid points in $\mathcal{V}_A$ and $\mathcal{V}_B$. Such constraint frees us from expensive calculations for finding the exact intersections between a scanline and side edges – we only need to identify the best scanline ends from sets $\mathcal{V}_A$ and $\mathcal{V}_B$.

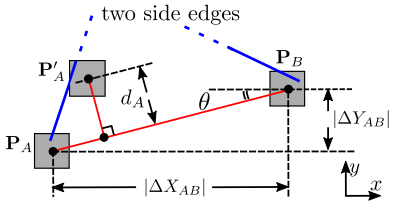



Figure 6: We find the next voxel $\mathbf{P}'_A \in \mathcal{V}_A$ as the one that is most distant from the current scanline while the scanning condition is still satisfied.

Undoubtedly, the first scanline connects the first entries in $\mathcal{V}_A$ and $\mathcal{V}_B$, say $\mathbf{P}_A(X_A, Y_A) \in \mathcal{V}_A$ and $\mathbf{P}_B(X_B, Y_B) \in \mathcal{V}_B$ as shown in Fig. 6. According to Eq. (5), we re-write the best SI as:

$$\widehat{d} = h \cdot (\sin\theta + \cos\theta) = \frac{|\Delta X_{AB}| + |\Delta Y_{AB}|}{\sqrt{\Delta X_{AB}^2 + \Delta Y_{AB}^2}}, \qquad (6)$$

where $\Delta X_{AB} = X_A - X_B$ and $\Delta Y_{AB} = Y_A - Y_B$. The line equation of the current scanline can be written in the form of $ax + by + c = 0$, where $a = \Delta Y_{AB}$, $b = \Delta X_{AB}$, $c = \Delta X_{AB} Y_A - \Delta Y_{AB} X_A$. It is known that the distance between a point $(x_0, y_0)$ and $ax+by+c = 0$ is $|ax_0+by_0+c|/\sqrt{a^2+b^2}$. Thus, $d_A$, between a voxel $\mathbf{P}'_A(X'_A, Y'_A) \in \mathcal{V}_A$ and the current scanline can be computed as:

$$d_A = \frac{1}{\sqrt{\Delta X_{AB}^2 + \Delta Y_{AB}^2}} \left| \Delta Y_{AB}(X'_A - X_A) - \Delta X_{AB}(Y'_A - Y_A) \right|. \qquad (7)$$

Enforcing $d_A \le \widehat{d}$ leads to $\left|\Delta Y_{AB}(X'_A - X_A) - \Delta X_{AB}(Y'_A - Y_A)\right| \le |\Delta X_{AB}| + |\Delta Y_{AB}|$ or equivalently:

$$C_1 \le \Delta Y_{AB} X'_A - \Delta X_{AB} Y'_A \le C_2. \qquad (8)$$

Here, $C_1 = \Delta Y_{AB} X_A - \Delta X_{AB} Y_A - |\Delta X_{AB}| - |\Delta Y_{AB}|$ and $C_2 = \Delta Y_{AB} X_A - \Delta X_{AB} Y_A + |\Delta X_{AB}| + |\Delta Y_{AB}|$. Every time we update $\mathbf{P}'_A$ with its next entry in $\mathcal{V}_A$, either its $X$ or $Y$ indices will be changed by $\pm 1$, meaning Eq. (8) can actually be incrementally evaluated:

$$C_1 \le C_0 + \Delta C \le C_2, \qquad C_0 = \Delta Y_{AB} X_A - \Delta X_{AB} Y_A, \quad (9)$$

where $\Delta C$ could be either $\pm\Delta Y_{AB}$ or $\pm\Delta X_{AB}$, depending on the orientation of $e_A$.

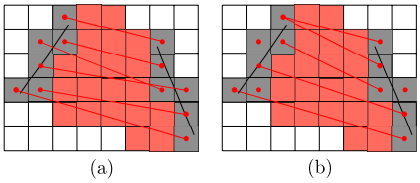
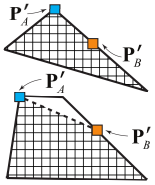



Figure 7: Compared to the naïve scanline strategy (a), the proposed method skips most redundant voxel generation (b).

Note that $C_0$, $C_1$ and $C_2$ are all integer constants depending on the current scanline configuration, and we can tell whether $\mathbf{P}'_A$ violates the scan condition with only four integer operations: three comparisons (one for determining the value of $\Delta C$) and one addition. As soon as the scan condition does not hold, we rollback to the previous entry in $\mathcal{V}_A$ and choose it as the end $A$ for the next scanline. Otherwise, current $\mathbf{P}'_A$ may still be conservative and we forward to the next entry in $\mathcal{V}_A$. Similarly, we can locate the best end $B$ in $\mathcal{V}_B$, and voxelize the resulting scanline using ILV. Fig. 7 shows an illustrative example of the proposed scanline method, as well as the naïve scanline strategy, in which scanlines are generated for every voxel pair in $\mathcal{V}_A$ and $\mathcal{V}_B$. Experiments show that our method is significantly faster than the naïve scanline since most of the redundant voxel generation is omitted. The complete triangle voxelization procedure is summarized in Alg. 1.

*Boundary treatment.* It is noteworthy that the numbers of voxels in $\mathcal{V}_A$ and $\mathcal{V}_B$ may differ significantly and it is possible that, for example as shown on the right, $\mathbf{P}'_A$ reaches the last entry in $\mathcal{V}_A$ before




[2]There will be three side edges if $\mathcal{T}_i$ is a pentagon. In this case, we simply combine the two connected side edges into one single side edge.

**Algorithm 1** Triangle Voxelization

```
1:  function VoxelizeTriangle(v0, v1, v2, n)
2:      {P0, P1, P2} ← GetVoxel(v0, v1, v2)
3:      i ← DominantAxisIndex(n)
4:      SortOnAxis(P0, P1, P2, i)
5:      MarkLineILV(P0, P1, Q0)
6:      MarkLineILV(P1, P2, Q1)
7:      MarkLineILV(P0, P2, Q2)
8:      Q1 ← Q0 ∪ Q1
9:      FillInterior(Q1, Q2, P0, P2, i)
10: ---------------------------------------------
11: function MarkLineILV(P0, P1, Q)
12:     ΔP[0] ← Sign(P1[0] − P0[0])
13:     ΔP[1] ← Sign(P1[1] − P0[1])
14:     ΔP[2] ← Sign(P1[2] − P0[2])
15:     L[0] ← M[0] ← |P1[1] − P0[1]||P1[2] − P0[2]|
16:     L[1] ← M[1] ← |P1[0] − P0[0]||P1[2] − P0[2]|
17:     L[2] ← M[2] ← |P1[0] − P0[0]||P1[1] − P0[1]|
18:     P_current ← P0
19:     while P_current ≠ P1 do
20:         ⟨L_min, L_index⟩ ← Min(L[0], L[1], L[2])
21:         P_current[L_index] ← P_current[L_index]+ΔP[L_index]
22:         L ← L − L_min
23:         L[L_index] ← 2M[L_index]
24:         MarkVoxel(P_current)
25:         Q.PushBack(P_current)
26: ---------------------------------------------
27: function FillInterior(Q1, Q2, P0, P2, axis)
28:     for i = 0 to P2[axis] − P0[axis] do
29:         slice ← P0[axis] + i + 1/2
30:         Q1sub ← GetSubSequence(Q1, slice)
31:         Q2sub ← GetSubSequence(Q2, slice)
32:         while Q1sub ≠ ∅ OR Q2sub ≠ ∅ do
33:             P_start ← GetNextInSlice(Q1sub)
34:             P_stop ← GetNextInSlice(Q2sub)
35:             MarkLineILV(P_start, P_stop)
```

$\mathbf{P}'_B$ does. Such boundary inconsistency is handled seperately based on the geometry of $\mathcal{T}_i$. If $\mathcal{T}_i$ is a triangle, the scanline terminates as soon as $\mathbf{P}'_A$ (or $\mathbf{P}'_B$) reaches the end because the entire triangle interior has been voxelized (the shadowed region). Otherwise, the corresponding base edge will become the new side edge, and the remaining un-voxelized region becomes a triangle.

## 5. Performance Analysis

In this section, we discuss the performance of the proposed voxelization algorithm. We also elaborate the detailed algorithmic difference between our method and existing SAT-based techniques.

*Cover property.* The cover property of our resulting surface voxelization is determined by which line voxelization algorithm, SLV, RLV or ILV, is used. It is not difficult to see that using SLV can form the super cover of the input model as the state-of-the-art does. When using RLV, the proposed method forms a 26-tunnel-free cover. As discussed previously, if the input model does not contain singular points, both RLV and SLV generate the super cover. ILV produces 26-tunnel-free surface voxelization which is a close approximation of a cover. The error induced by ILV is related to many factors, such as the voxel dimensions, the orientation of the line segment and the relative spatial position between the grid points and the line end points. Fig. 8 shows three typical cases where we use RLV and ILV to voxelize some line segments. We can see that the voxelizations vary a lot in the extreme cases (a) and (b), but in (c), which is like the average case, no error occurred.

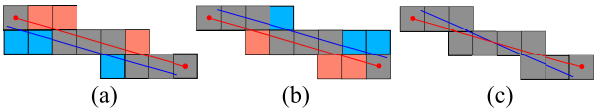



Figure 8: Voxelizing the line segments (blue) by RLV and the approximation (red) by ILV could result either different voxelizations in (a) and (b), or identical voxelizations in (c). The grey voxels are captured by both RLV and ILV, while the blue voxels and the red voxels are captured only by RLV or ILV respectively.

Optionally, our method can be downgraded to generate thinner 18- or 6-tunnel-free voxelizations by removing some base edge voxelizations (Fig. 9). Recall that the base edges are shared by two consecutive polygons and voxelized in both of the corresponding two slices. This feature is necessary to make the final voxliezation 26-tunnel-free. If we eliminate the voxelization of the base edge in either of the two slices, 26- or 18-paths form, and the voxelization can only be 6-tunnel-free and is not a cover anymore. The downgrading method makes thinner voxelizations as preferred in some applications.

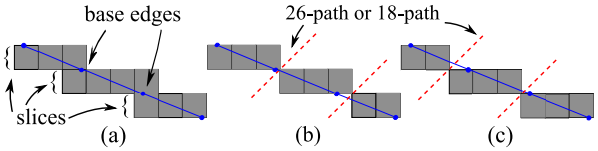



Figure 9: Our method (a) can be downgraded to 18- or 6-tunnel-free methods (b) or (c) by eliminating some base edge voxelizations. The resulting voxelizations are thinner and 6-tunnel-free.

*Algorithmic complexity.* Schwarz and Seidel [14] proposed the state-of-the-art voxelization algorithm. They

firstly determine the dominate axis of the triangle, say, $z$ axis, then project the triangle to the $xy$ plane. A set of voxels are determined by a 2D overlap test. Each voxel further determines a voxel array along $z$ axis which contains a number of voxel candidates (up to three) intersecting the triangle. All the voxel candidates are subject to two remaining 2D overlap tests. The complexity of the method is therefore $\mathbf{O}(N \times M)$, where $N$ and $M$ are the numbers of voxels along $x$ and $y$ axis of the bounding box respectively. We define a triangle-voxel overlap test as an *atomic operation*, which contains 9 sub-tests. Each sub-test has 2 multiplications, 2 additions and 1 comparisons. All of the $N \times M$ voxels should undergo the atomic operation, but most of them do not have to take all the sub-tests. There are two reasons. One is that a failed sub-test will imply no overlap and the rest sub-tests are not necessary. The other is that voxels in the same voxel array can share the $xy$ plane test result. Therefore, based on the observation that the thickness of the voxelization is at most three voxels along $z$ axis, the lower bound of the number of sub-tests for an overlapping voxel is 7, which yields 14 multiplications, 14 additions and 7 comparisons. Pantaleoni [15] further improved the previous method when determining the voxel candidates in the $xy$ plane. This method takes one coordinate as a constant and computes the range of the other coordinate through the edge functions. The voxels in the range are immediately taken as candidates without performing overlap tests. The rest steps are the same as the ones in the previous method: the $z$ coordinate range is computed per voxel array, and voxels in the range are subjected to the remaining two 2D projection overlap tests. We can see that the complexity of this method is $\mathbf{O}(W)$, where $W$ is the number of the voxles overlapping the triangle. For an overlapping voxel, 6 sub-tests are needed, which contains 12 multiplications, 12 additions and 6 comparisons.

In our method, we determine the dominant axis, followed by voxlelizing the three edges of the triangle. The rest work is to find the voxels overlapping the triangle interior. To this end, we further divide the triangle slice by slice. In each slice, we use 2D scanlines to find the overlapping voxels. We can see that the method actually compute indices of the overlapping voxels only. Therefore, the complexity of our method is also $\mathbf{O}(W)$. However, the atomic operation of our method is simply to extend one voxel along a 2D scanline, which only involves 2 additions plus 1 comparison. This is the major reason why our method is faster than the existing SAT-based method.

*Parallelization.* Both aforementioned SAT-based methods and our method can be parallelized and accelerated using multi-threading or GPU. However, it can be clearly seen that our method is parallelized for each triangle, while the SAT-based methods can be trivially parallelized for each voxel candidate. As a result, one may speculate that with the increased resolution of the vocalization, SAT-based method will eventually outperform our method. Interestingly, the fact is opposite – our method often demonstrates a better performance in practice. The reasons are two-fold. First of all, while scanning the interior of an individual triangle is *sequential*, parallelization at triangles utilizes modern GPU platform already. For instance, the latest `nVidia GTX 1080` GPU equips with $2,560$ cores, while most meshes we are dealing with nowadays have much more than $2,560$ triangles in general. More importantly, our method has a much simpler atomic operation than SAT-based methods (e.g. in [14, 15]). Therefore, a higher voxelization resolution will further exaggerate such dif-

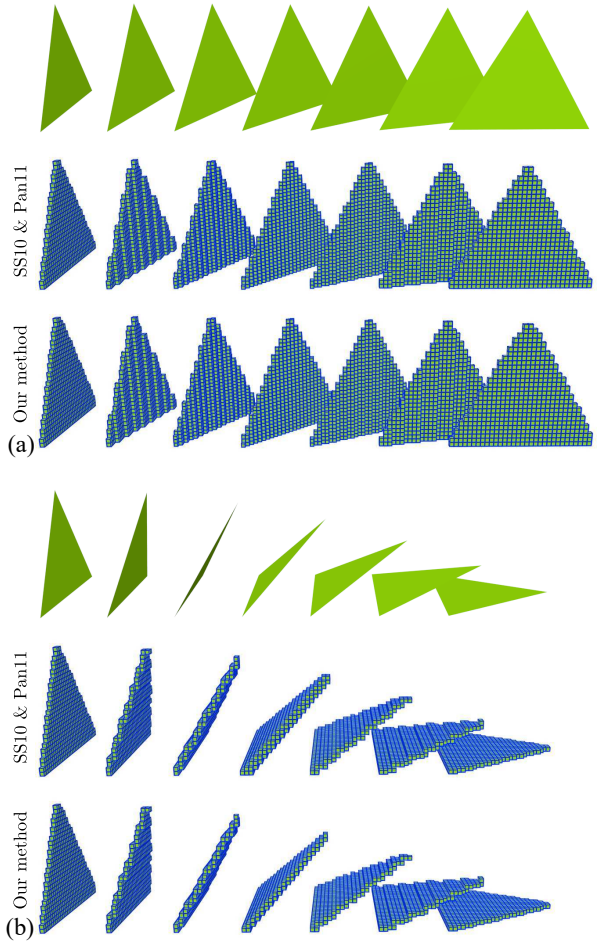



Figure 10: Voxelizing an equilateral triangle under the resolution of $32^3$. (a) The triangle rotate around $y$ axis. (b) The triangle rotates around $z$ axis.

ference (see Tab. 2). On the other hand, if the dimension of a voxel is comparable to a triangle patch, our method becomes less efficient. This is because interior voxels may be already generated during the line voxelization or even point voxelization stage and computing the SI based on Eq. (8) is less profitable to reduce the redundant voxelization. Furthermore, if the triangle numbers are very small (e.g. only one triangle), the SAT-based methods will beat our method easily by parallelization. Based on such analysis, one can clearly see that our method is alternative to and complements existing SAT-based methods: it is suitable for high-density voxelization (e.g. for accurate numerical integrations). Our experiment results confirm this conclusion.

## 6. Experiments and Results

Our method is tested on a desktop PC (with both CPU and GPU), and an Apple iPhone 6. For comparison, we also implemented the state-of-the-art SAT-based methods as in [14] and [15]. The desktop PC equips an Intel® i7-2600 CPU@3.4 GHz (4 physical cores), 12G RAM, and a NVIDIA GTX 970 video card. The mobile platform is an Apple iPhone 6 with Dual-core Typhoon CPU@1.4 GHz (ARM v8-based). Our CPU implementation is both single- and multi-thread using `C++` and our GPU implementation utilizes NVIDIA `CUDA 7.5`. Our IOS implementation is written in `Object-C` with `Xcode`.

*Implementation.* In [14] and [15], triangles can be preprocessed and classified to 1D, 2D or 3D cases. In 1D cases, the triangle is slim and enclosed in one voxel array aligned to a coordinate axis. All the voxels in the array are marked as overlapping. In 2D cases, each voxel in the bounding box undergoes a 2D projection overlap test. Voxels passing the test are marked immediately [14]. More efficiently, Pantaleoni [15] finds those overlapping voxels by fixing one coordinate and computing the range of the other coordinate. In more general 3D cases, as described previously, one projects the triangle to determine the set of voxel arrays, computes the range of the voxel candidates and processes the two remaining 2D projection overlap tests.

In our `CUDA` implementation we process all triangles in parallel, with each thread voxelizing a single triangle. Each voxel in our global grid is represented as a bit in a 32-bit integer array and each thread that is processing a triangle must update this array when a voxel is labeled. To accomplish this we take advantage of `atomic functions` from `CUDA`. Atomic memory operations alleviate the complexity involved in updating shared memory during a parallel computation.

| $y$ | # Comparisons | | | # Additions | | |
|---|---|---|---|---|---|---|
| | **SS10** | **Pan11** | **Ours** | **SS10** | **Pan11** | **Ours** |
| 0° | 2,483 | 0 | 838 | 4,966 | 0 | 1,111 |
| 15° | 6,356 | 3,806 | 863 | 12,712 | 7,612 | 1,236 |
| 30° | 8,177 | 5,343 | 1,061 | 16,354 | 10,686 | 1,687 |
| 45° | 7,835 | 5,523 | 1,027 | 15,666 | 11,046 | 1,744 |
| 60° | 8,151 | 5,246 | 1,068 | 16,302 | 10,492 | 1,701 |
| 75° | 6,357 | 3,786 | 863 | 12,750 | 7,572 | 1,236 |
| 90° | 2,483 | 0 | 839 | 4,966 | 0 | 1,113 |
| $z$ | **SS10** | **Pan11** | **Ours** | **SS10** | **Pan11** | **Ours** |
| 0° | 2,483 | 0 | 838 | 4,966 | 0 | 1,111 |
| 15° | 5,899 | 3,490 | 863 | 11,798 | 6,980 | 1,236 |
| 30° | 6,204 | 3,940 | 1,061 | 12,408 | 7,880 | 1,687 |
| 45° | 8,162 | 5,358 | 1,027 | 16,324 | 10,716 | 1,744 |
| 60° | 6,204 | 3,988 | 1,068 | 12,408 | 7,976 | 1,701 |
| 75° | 5,899 | 3,502 | 863 | 11,798 | 7,004 | 1,236 |
| 90° | 2,483 | 0 | 839 | 4,966 | 0 | 1,113 |

Table 1: Comparative statistics of complexity of SAT-based methods ([14] and [15]) and our method showing the total numbers of comparisons and additions during the voxelization. When the triangle is parallel to the coordinate plane, no atomic operation is needed in [15].

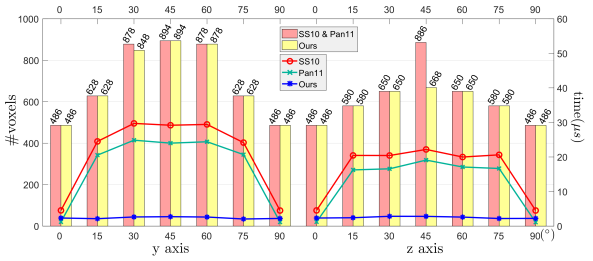



Figure 11: The numbers of voxels (bar plots) as well as the time performance (line plots) of [14] [15] and our method.

Specifically, we utilize the atomic `OR` function which ensures that all voxel array updates issued concurrently are performed without interruption with respect to other threads.

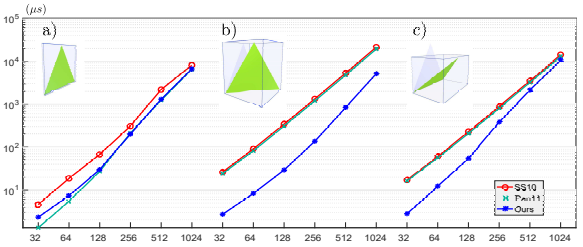



Figure 12: The time performance of triangle voxelization under various resolutions: a) the triangle is aligned in the $yz$ plane initially; b) the triangle is rotated around $y$ axis by 45°; and c) the triangle is rotated around $z$ axis by 45°.

*Voxelization of a single triangle.* We voxelize an equilateral triangle under the resolution of $32^3$ using the

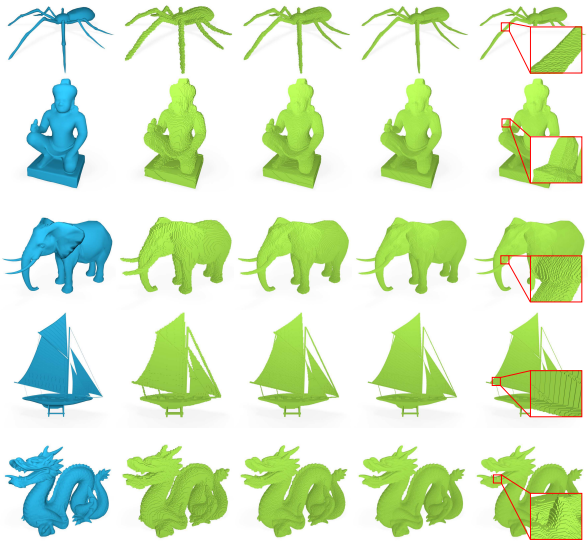


Figure 13: Voxelizing several 3D models under the voxelizations of $128^3$, $256^3$, $512^3$ and $1024^3$.

SAT-based methods and our method. The triangle is initially aligned with the $yz$ plane, and we rotate it around $y$ and $z$ axes gradually by 15° each time up to 90°. We use the single-thread implementation in this experiment. The resulting voxelizations are shown in Fig. 10. The corresponding computation costs are reported in Tab. 1, where we can see that our method invests much less computation in the general 3D scenarios. Note that we don't compare the multiplications because the atomic operation in our method does not involve multiplications.

We also notice that the computation costs of the SAT-based method largely depend on the orientation of the triangle, while our method is much more consistent with respect to different orientations. This observation is also verified in the time benchmark shown in Fig. 11, where we pick three typical poses of the rotating triangle, which are the initial one, the one rotated around $y$ axis by 45°, and the one rotated around $z$ axis by 45°, to test the time performance under various resolutions from $32^3$ to $1024^3$ as shown in Fig. 12. In Fig. 12 a), the triangle is essentially in 2D and [15] performs best. However, in more general 3D cases (Fig. 12 b) and c)) our method is faster. Note that the vertical axis is in logarithmic.

*More results on 3D models.* The voxelization results of more 3D models are shown in Fig. 13. The resolutions of the voxelization increases from $256^3$ to $4096^3$. Tab. 2 reports the comparative time performance using a single- and multi-thread CPU, `CUDA` and iOS implementations (i.e. Fig. 14) of both the SAT-based methods and our method. Due to the limited memory, voxelization either at $2048^3$ on the iPhone or at $4096^3$ on GPU is not available. As Tab. 2 exhibits, the performance gap between the SAT and our method is getting larger with the increased voxel resolutions. Such result is consistent with our performance analysis as discussed in Sec. 5. The performance reported in Tab. 2 is based on the integer version of our algorithm. We find that the acceleration of using integer version algorithm over the floating number version is very minor (~ 3%). However, further acceleration of using integer version algorithm may be possible if dedicated hardware is adopted [41].

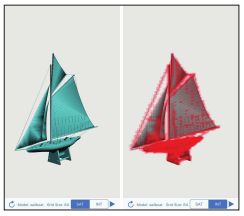


Figure 14: Snapshots of our iOS implementation.

To justify the voxelization error, we voxelize the 3D models by different methods and compare the voxel numbers of the resulting voxelizations. The results are reported in Tab. 3. We can see that, as expected, our floating number version (SLV/RLV) generates the same number of voxels as the SAT-based method does, and the relative error induced by our integer version is small (less than 2.5%). We highlight the difference of two voxelizations generated by the SAT-based method and our integer version method respectively in Fig. 15 (under the $64^3$ resolution).

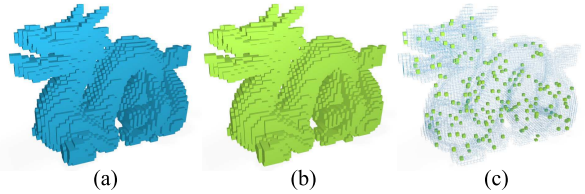



Figure 15: Voxelizations of the 3D dragon model by the SAT-based method (a) and our integer version method (b) under the $64^3$ resolution are presented. The difference is highlighted in (c).

*Applications.* As mentioned, the voxelization plays an essential role in many graphics applications. The proposed high-performance voxelization algorithm can be directly used in physics-based animations – both for mesh generation and collision culling. It is also important for global illumination (Fig. 16).

## 7. Conclusion

We present a high-performance algorithm for the voxelization of complex 3D models. Our method avoids (relatively) expensive triangle-voxel intersection tests

| Model | Resolution | CPU$^s$ ($ms$) | | | CPU$^m$ ($ms$) | | | GPU ($ms$) | | | iOS ($ms$) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Pan11** | **Ours** | **Acc** | **Pan11** | **Ours** | **Acc** | **Pan11** | **Ours** | **Acc** | **Pan11** | **Ours** | **Acc** |
| Spider (3,341 tris) | $256^3$ | 3.7 | 2.7 | 37% | 1.8 | 1.6 | 13% | 0.30 | 0.20 | 50% | 8.5 | 7.4 | 15% |
| | $512^3$ | 8.8 | 4.6 | 91% | 3.1 | 2.5 | 24% | 0.85 | 0.45 | 88% | 20.1 | 10.9 | 84% |
| | $1024^3$ | 24.8 | 9.7 | 156% | 8.8 | 4.7 | 87% | 2.6 | 1.1 | 132% | 83.3 | 29.0 | 187% |
| | $2048^3$ | 87.9 | 29.9 | 194% | 37.0 | 17.8 | 107% | 20.7 | 8.6 | 139% | – | – | – |
| | $4096^3$ | 492.3 | 187.7 | 162% | 148.8 | 62.8 | 137% | – | – | – | – | – | – |
| Cambridge demon (8,822 tris) | $256^3$ | 11.1 | 8.2 | 35% | 5.3 | 3.9 | 36% | 0.82 | 0.63 | 30% | 21.5 | 16.7 | 29% |
| | $512^3$ | 30.5 | 14.5 | 110% | 13.2 | 5.6 | 136% | 1.8 | 1.5 | 20% | 83.6 | 36.0 | 132% |
| | $1024^3$ | 99.2 | 33.3 | 198% | 29.7 | 12.1 | 145% | 6.8 | 4.2 | 62% | 284.2 | 99.2 | 186% |
| | $2048^3$ | 476.1 | 117.0 | 307% | 144.6 | 48.3 | 199% | 33.1 | 19.5 | 70% | – | – | – |
| | $4096^3$ | 2255 | 759.8 | 197% | 699.4 | 241.8 | 189% | – | – | – | – | – | – |
| Elephant (10,150 tris) | $256^3$ | 11.0 | 7.9 | 39% | 4.8 | 4.2 | 14% | 1.9 | 1.8 | 6% | 21.9 | 16.2 | 35% |
| | $512^3$ | 28.4 | 14.1 | 101% | 9.0 | 6.9 | 30% | 4.1 | 2.3 | 78% | 79.9 | 36.0 | 122% |
| | $1024^3$ | 86.0 | 31.3 | 175% | 26.7 | 14.8 | 80% | 15.4 | 4.6 | 234% | 380.5 | 95.2 | 300% |
| | $2048^3$ | 325.2 | 103.9 | 213% | 132.3 | 53.2 | 148% | 85.4 | 22.7 | 276% | – | – | – |
| | $4096^3$ | 2066 | 711.3 | 190% | 594.6 | 216.6 | 175% | – | – | – | – | – | – |
| Sailboat (70,476 tris) | $256^3$ | 23.4 | 19.2 | 22% | 14.8 | 13.7 | 8% | 10.6 | 4.3 | 66% | 62.0 | 32.8 | 89% |
| | $512^3$ | 49.0 | 29.1 | 69% | 25.7 | 18.0 | 43% | 17.9 | 9.7 | 85% | 197.4 | 56.9 | 247% |
| | $1024^3$ | 133.9 | 50.9 | 163% | 58.8 | 27.9 | 111% | 48.8 | 22.9 | 113% | 491.5 | 110.2 | 346% |
| | $2048^3$ | 483.4 | 104.9 | 361% | 173.4 | 49.3 | 252% | 160.8 | 42.1 | 281% | – | – | – |
| | $4096^3$ | 1168 | 333.9 | 250% | 394.5 | 121.3 | 225% | – | – | – | – | – | – |
| Dragon (100,000 tris) | $256^3$ | 59.3 | 61.1 | −3% | 35.1 | 38.2 | −8% | 1.3 | 1.4 | −6% | 72.3 | 80.1 | 11% |
| | $512^3$ | 99.4 | 85.9 | 16% | 48.7 | 47.2 | 3% | 1.7 | 1.6 | 6% | 191.2 | 143.4 | 33% |
| | $1024^3$ | 197.6 | 128.3 | 54% | 83.3 | 62.5 | 33% | 6.5 | 5.2 | 13% | 618.0 | 302.4 | 104% |
| | $2048^3$ | 591.8 | 253.0 | 134% | 191.5 | 103.9 | 84% | 22.8 | 17.5 | 30% | – | – | – |
| | $4096^3$ | 2507 | 1007 | 149% | 629.5 | 309.7 | 103% | – | – | – | – | – | – |

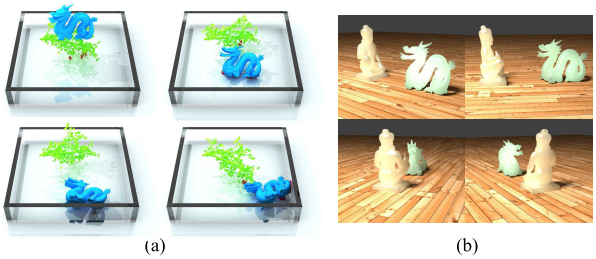
Table 2: The running time for GPU/CPU and iOS implementations of [15] (**Pan11**) and our method. The accelerations (**Acc**) are also highlighted. **CPU**$^s$ and **CPU**$^m$ are for the single- and multi-thread implementations. The number of threads that we use in the latter is four.

| 3D Model | Spider | | | Demon | | | Elephant | | | Sailboat | | | Dragon | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Resolution | $256^3$ | $512^3$ | $1024^3$ | $256^3$ | $512^3$ | $1024^3$ | $256^3$ | $512^3$ | $1024^3$ | $256^3$ | $512^3$ | $1024^3$ | $256^3$ | $512^3$ | $1024^3$ |
| SS10/Pan11 | 38.9K | 161K | 658K | 173K | 691K | 2.77M | 125K | 506K | 2.05M | 65.6K | 276K | 1.18M | 168K | 673K | 2.70M |
| SLV/RLV | 38.9K | 161K | 658K | 173K | 691K | 2.77M | 125K | 506K | 2.05M | 65.6K | 276K | 1.18M | 168K | 673K | 2.70M |
| ILV | 39.8K | 164K | 674K | 175K | 701K | 2.80M | 128K | 516K | 2.08M | 67.2K | 281K | 1.20M | 171K | 687K | 2.75M |
| ILV error | 2.3% | 1.9% | 2.4% | 1.2% | 1.4% | 0.7% | 2.4% | 2.0% | 1.5% | 2.4% | 1.8% | 1.7% | 1.8% | 2.1% | 1.9% |

Table 3: Comparative statistics of the voxel numbers of the resulting 3D model voxelizations under different resolutions and using different methods. Our floating number version (SLV/RLV) generates the same number of voxels as the SAT-based method does. The relative error induced by our integer version method (ILV) is less than 2.5%.

and voxelizes the surface geometry based on an efficient line voxelization algorithm. Since there is no optimal 3D scanline interval, we project a 3D triangle into axis-aligned slices and give the theoretically optimal scan-line strategy. On top of it, we further avoid floating number arithmetic during the voxelization. We provide a comprehensive analysis and comparative experiments between the proposed method and the state-of-the-art SAT-based methods. The time performance on CPU, GPU as well as mobile devices shows that our method is efficient, especially for high-resolution voxelizations when the encapsulating a triangle requires more interior voxels.

Since the voxelization is a discrete representation of the original object, many applications require that voxels store more information rather than the binary overlap flag, such as density, normal vector, material properties, etc. For instance, surface normal vector at the voxel's location should be well estimated for alias-free rendering [48, 49]; evaluating the occupancy functions



(a) (b)

Figure 16: Our methods are used in mesh generation and collision culling (a) and global radiosity (b).

(filtered techniques) [50] or distance functions (distance field techniques) [51, 52, 53, 54] at the voxels is essential to reconstruct the original object surface. Our method is basically designed for binary voxelization and can not directly support the non-binary application. As a potential solution, one can first voxelize the object using our method, then compute and store the associated quantities for each overlapping voxel using other related algorithms.

This method leaves us many interesting further directions to explore. For example, the current version of our method is not able to produce the voxelization with adaptive resolutions (e.g. like using an octree). This limitation may be resolved by applying our method at various density levels incrementally. It is also worthy to further investigate how to utilize the superior performance of the proposed method – we see runtime collision culling is a promising application.

## 8. Acknowledgement

We would like to thank anonymous reviewers for their constructive comments. Weiwei Xu is partially supported by NSFC (61322204) and the Fundamental Research Funds for the Central Universities (2017XZZX009-03). Tianjia Shao is partially supported by NSFC (61402402) and Microsoft Research Asia. Yin Yang is partially supported by NSF (CNS-1637092 and CHS-1464306).

[1] A. Kolb, L. John, et al., Volumetric model repair for virtual reality applications, EG, Short-Paper (2001) 249–256.

[2] K. Kreeger, A. Kaufman, Mixing translucent polygons with volumes, in: Proceedings of the conference on Visualization'99, 1999, pp. 191–198.

[3] F. Liu, M.-C. Huang, X.-H. Liu, E.-H. Wu, Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects, in: I3D, 2010, pp. 75–82.

[4] G. v. d. Bergen, Efficient collision detection of complex deformable models using AABB trees, Journal of Graphics Tools 2 (4) (1997) 1–13.

[5] T. He, A. Kaufman, Collision detection for volumetric objects, in: Proceedings of the 8th conference on Visualization'97, IEEE Computer Society Press, 1997, pp. 27–ff.

[6] M. Nießner, C. Siegl, H. Schäfer, C. Loop, Real-time collision detection for dynamic hardware tessellated objects.

[7] Q.-h. Zhu, Y. Chen, A. Kaufman, Real-time biomechanically-based muscle volume deformation using fem, in: Computer Graphics Forum, Vol. 17, Wiley Online Library, 1998, pp. 275–284.

[8] D. L. James, J. Barbič, C. D. Twigg, Squashing cubes: Automating deformable model construction for graphics, in: ACM SIGGRAPH 2004 Sketches, SIGGRAPH '04, 2004, p. 38.

[9] W. Li, Z. Fan, X. Wei, A. Kaufman, Flow simulation with complex boundaries, GPU Gems 2 (2005) 747–764.

[10] T. Zirr, C. Dachsbacher, Memory-efficient on-the-fly voxelization and rendering of particle data, IEEE Transactions on Visualization and Computer Graphics.

[11] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, E. Eisemann, Geometry and attribute compression for voxel scenes, in: Computer Graphics Forum, Vol. 35, Wiley Online Library, 2016, pp. 397–407.

[12] Y. Zhou, S. Sueda, W. Matusik, A. Shamir, Boxelization: folding 3d objects into boxes, ACM Transactions on Graphics (TOG) 33 (4) (2014) 71.

[13] D. Cohen-Or, A. Kaufman, Fundamentals of surface voxelization, Graphical models and image processing 57 (6) (1995) 453–461.

[14] M. Schwarz, H.-P. Seidel, Fast parallel surface and solid voxelization on gpus, in: ACM Transactions on Graphics (TOG), Vol. 29, ACM, 2010, p. 179.

[15] J. Pantaleoni, Voxelpipe: a programmable pipeline for 3d voxelization, in: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, ACM, 2011, pp. 99–106.

[16] A. Kaufman, Efficient algorithms for 3d scan-conversion of parametric curves, surfaces, and volumes, ACM SIGGRAPH Computer Graphics 21 (4) (1987) 171–179.

[17] A. Kaufman, Efficient algorithms for scan-converting 3d polygons, Computers & Graphics 12 (2) (1988) 213–219.

[18] A. Kaufman, E. Shimony, 3d scan-conversion algorithms for voxel-based graphics, in: Proceedings of the 1986 workshop on Interactive 3D graphics, ACM, 1987, pp. 45–75.

[19] J. E. Bresenham, Algorithm for computer control of a digital plotter, IBM Systems journal 4 (1) (1965) 25–30.

[20] S. Gottschalk, M. C. Lin, D. Manocha, Obbtree: A hierarchical structure for rapid interference detection, in: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, ACM, 1996, pp. 171–180.

[21] T. Akenine-Möller, E. Haines, N. Hoffman, Real-time rendering, CRC Press, 2008.

[22] D. Eberly, Intersection of convex objects: The method of separating axes, www. magic-software. com.

[23] T. Akenine-Möller, Fast 3D triangle-box overlap testing, in: ACM SIGGRAPH 2005 Courses, ACM, 2005, p. 8.

[24] C. Crassin, S. Green, Octree-based sparse voxelization using the gpu hardware rasterizer, OpenGL Insights (2012) 303–318.

[25] J. Hasselgren, T. Akenine-Möller, L. Ohlsson, Conservative rasterization, GPU Gems 2 (2005) 677–690.

[26] M. D. McCool, C. Wales, K. Moule, Incremental and hierarchical hilbert order edge equation polygon rasterizatione, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, ACM, 2001, pp. 65–72.

[27] J. Pineda, A parallel algorithm for polygon rasterization, in: SIGGRAPH, Vol. 22, 1988, pp. 17–20.

[28] T. Akenine-Möller, T. Aila, Conservative and tiled rasterization using a modified triangle set-up, Journal of Graphics, GPU, and Game Tools 10 (3) (2005) 1–8.

[29] E. A. Haines, J. R. Wallace, Shaft culling for efficient ray-cast radiosity, in: Photorealistic Rendering in Computer Graphics,

Springer, 1994, pp. 122–138.
[30] J. Huang, R. Yagel, V. Filippov, Y. Kurzion, An accurate method for voxelizing polygon meshes, in: Volume Visualization, IEEE Symposium on, IEEE, 1998, pp. 119–126.
[31] G. Varadhan, S. Krishnan, Y. J. Kim, S. Diggavi, D. Manocha, Efficient max-norm distance computation and reliable voxelization, in: SGP, 2003, pp. 116–126.
[32] V. E. Brimkov, R. P. Barneva, Graceful planes and lines, Theoretical Computer Science 283 (1) (2002) 151–170.
[33] Y. Fei, B. Wang, J. Chen, Point-tessellated voxelization, in: Proceedings of Graphics Interface 2012, Canadian Information Processing Society, 2012, pp. 9–18.
[34] Y. Chao, Y. Wei, X. Du, F. Yuan, Alarm thresholds of threaten regions based on triangle mesh voxelization, in: Fuzzy Systems and Knowledge Discovery (FSKD), 2015 12th International Conference on, IEEE, 2015, pp. 2475–2479.
[35] X. Liu, K. Cheng, Three-dimensional extension of bresenham's algorithm and its application in straight-line interpolation, Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture 216 (3) (2002) 459–463.
[36] C. Au, T. Woo, Three dimensional extension of bresenham's algorithm with voronoi diagram, Computer-Aided Design 43 (4) (2011) 417–426.
[37] J. Amanatides, A. Woo, et al., A fast voxel traversal algorithm for ray tracing, in: Eurographics, Vol. 87, 1987, p. 10.
[38] D. Cohen-Or, A. Kaufman, 3D line voxelization and connectivity control, Computer Graphics and Applications, IEEE 17 (6) (1997) 80–87.
[39] H.-H. Hsieh, C.-C. Chang, W.-K. Tai, H.-W. Shen, Novel geometrical voxelization approach with application to streamlines, Journal of Computer Science and Technology 25 (5) (2010) 895–904.
[40] M. Shand, P. Bertin, J. Vuillemin, Hardware speedups in long integer multiplication, ACM SIGARCH Computer Architecture News 19 (1) (1991) 106–113.
[41] Z. Luo, M. Martonosi, Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques, Computers, IEEE Transactions on 49 (3) (2000) 208–218.
[42] Z. Dong, W. Chen, H. Bao, H. Zhang, Q. Peng, Real-time voxelization for complex polygonal models, in: Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on, IEEE, 2004, pp. 43–50.
[43] E. Eisemann, X. Décoret, Single-pass gpu solid voxelization for real-time applications, in: Proceedings of graphics interface 2008, Canadian Information Processing Society, 2008, pp. 73–80.
[44] P. Ivson, L. Duarte, W. Celes, Gpu-accelerated uniform grid construction for ray tracing dynamic scenes, Master's thesis, Departamento de Informatica, Pontificia Universidade Catolica, Rio de Janeiro.
[45] K. Zhou, Q. Hou, R. Wang, B. Guo, Real-time kd-tree construction on graphics hardware, in: ACM Transactions on Graphics (TOG), Vol. 27, ACM, 2008, p. 126.
[46] K. Zhou, M. Gong, X. Huang, B. Guo, Data-parallel octrees for surface reconstruction, Visualization and Computer Graphics, IEEE Transactions on 17 (5) (2011) 669–681.
[47] L. Zhang, W. Chen, D. S. Ebert, Q. Peng, Conservative voxelization, The Visual Computer 23 (9-11) (2007) 783–792.
[48] S. W. Wang, A. E. Kaufman, Volume sampled voxelization of geometric primitives, in: Visualization, 1993. Visualization'93, Proceedings., IEEE Conference on, IEEE, 1993, pp. 78–84.
[49] M. Sramek, A. E. Kaufman, Alias-free voxelization of geometric objects, IEEE transactions on visualization and computer graphics 5 (3) (1999) 251–267.
[50] M. Sramek, A. Kaufman, Object voxelization by filtering, in: Volume Visualization, 1998. IEEE Symposium on, IEEE, 1998, pp. 111–118.
[51] D. E. Breen, R. T. Whitaker, A level-set approach for the metamorphosis of solid models, IEEE Transactions on Visualization and Computer Graphics 7 (2) (2001) 173–192.
[52] M. W. Jones, J. A. Baerentzen, M. Sramek, 3d distance fields: A survey of techniques and applications, IEEE Transactions on visualization and Computer Graphics 12 (4) (2006) 581–599.
[53] R. N. Perry, S. F. Frisken, Kizamu: A system for sculpting digital characters, in: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, ACM, 2001, pp. 47–56.
[54] P. Novotny, L. I. Dimitrov, M. Sramek, Enhanced voxelization and representation of objects with sharp details in truncated distance fields, IEEE transactions on visualization and computer graphics 16 (3) (2010) 484–498.