Visual Comput (2007) 23: 783–792
DOI 10.1007/s00371-007-0149-0

ORIGINAL ARTICLE

Conservative voxelization**

Long Zhang
Wei Chen*
David S. Ebert
Qunsheng Peng

Published online: 28 June 2007

L. Zhang · W. Chen (✉) · Q. Peng
State Key Lab of CAD&CG, Zhejiang University, 310027, Hangzhou, China
chenwei@cad.zju.edu.cn

David S. Ebert
The School of Electrical and Computer Engineering, Purdue University, USA

Abstract We propose a novel hardware-accelerated voxelization algorithm for polygonal models. Compared with previous approaches, our algorithm has a major advantage that it guarantees the conservative correctness in voxelization: every voxel intersecting the input model is correctly recognized. This property is crucial for applications like collision detection, occlusion culling and visibility processing. We also present an efficient and robust implementation of the algorithm in the GPU. Experiments show that our algorithm has a lower memory consumption than previous approaches and is more efficient when the volume resolution is high. In addition, our algorithm requires no preprocessing and is suitable for voxelizing deformable models.

Keywords Voxelization · GPU · Conservative correctness

1 Introduction

Voxelization [15, 21] denotes the reformulation process that converts various boundary-based representations like parametric surfaces and polygonal models into a volumetric representation, i.e., a 3D array of voxels. During the last decade, much effort has been devoted to this topic and voxelization has been widely applied to many applications, e.g., volumetric modeling [22], virtual medicine [16], haptic rendering [18], collision detection [3, 9, 12], 3D spatial analysis [2], physically-based simulation [17] and real-time rendering [6].

With the rapidly growing power of modern graphics hardware, hardware-accelerated voxelization has attracted much research interest. The main idea is to use the rasterization functionality of graphics hardware to accelerate voxelization. This is based on the fact that rasterization and voxelization are similar scan-conversion operations.

The common steps of hardware-accelerated voxelization are illustrated in Fig. 1. First, each polygon of the input model is projected onto the 2D viewing plane. Typically orthogonal projection is used to map the bounding box of the model into the viewing volume. Subsequently, each projected polygon is rasterized. For each resultant fragment, its screen coordinate (x, y) together with its depth value z defines a voxel (x, y, z^*), where $z^* = \lfloor z \cdot \mathrm{res}_z \rfloor$, and res_z denotes the volumetric resolution along the projection direction. The last step records the information of the voxel in the fragment (x, y) of the frame buffer, and is called *volume encoding*. Its core is a mapping between the fragment color and the corresponding voxels. The bottom left image of Fig. 1 shows an example that uses four color channels to represent the information of four voxels.

Previous hardware-accelerated voxelization algorithms have a common problem: some voxels that intersect the input model are missed. This is undesirable for some applications such as collision detection, occlusion culling and visibility processing. For instance, conservative correctness is crucial in voxelization-based collision detection because it guarantees that two models do not intersect if their voxelization results do not.

*This work was done while Wei Chen was a visiting scholar at Purdue university.

**This work was partially supported by NSF of China project (No. 60503056), the 973 program of China (No.2002CB312100) and the 863 program of China (No. 2006AA01Z314).

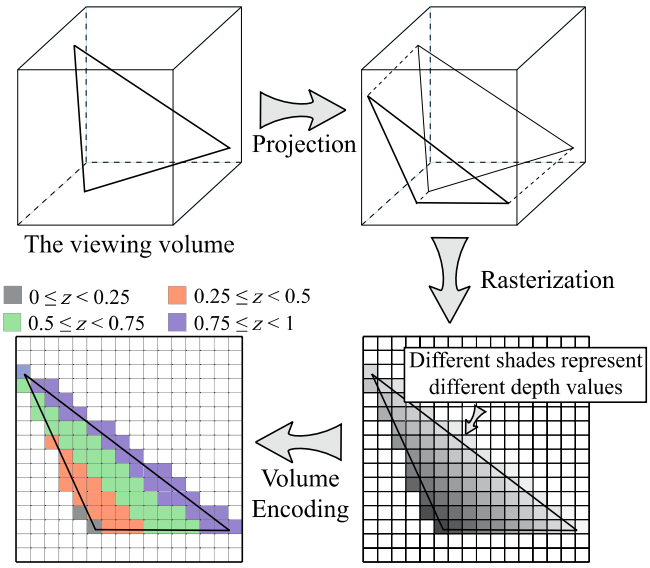


Fig. 1. The common pipeline of hardware-accelerated voxelization algorithms

One reason for the loss of voxels is that the standard rasterization performed within GPUs does not generate all fragments that intersect the projected polygon. This problem has been elegantly solved by the *conservative rasterization* algorithm [1, 11]. Another reason is that multiple voxels intersecting a polygon may be projected into the same fragment, in which case only one voxel is generated by traditional techniques.

The main contribution of this paper is a novel technique that recovers all voxels corresponding to each rasterized fragment. By seamlessly incorporating some conservative rasterization algorithm, we achieve conservatively correct results. We present our approach as follows. Section 2 gives a brief review of hardware-accelerated voxelization algorithms. In Sect. 3 we outline our algorithm and present an efficient and robust implementation in the GPU. Section 4 presents an alternative solution to conservative voxelization. The implementation details are described in Sect. 5. Experimental results and technical discussions are given in Sect. 6. Finally, we draw conclusions and highlight future work in Sect. 7.

2 Related work

The slice-based algorithm [4, 8] is a pioneering work that makes use of graphics hardware to accelerate voxelization. It employs a multi-pass rendering technique to generate the resultant volume slice by slice. In each pass, a pair of near and far clipping planes is set and the geometry between the two planes is rasterized to generate a volume slice. It is apparent that the pass number is identical to the slice number, or the volume resolution along the projection direction. Thus, the efficiency decreases rapidly with the increase of the volume resolution.

Karabassi et al. [14] proposed to project the input model to six faces of its bounding volume. For each projection, the frontmost fragments are stored and later read back from the depth buffer. This way, only six rendering passes are required. The algorithm works well only for convex objects, heavily restricting its usage in practical applications.

Based on the concept of depth peeling [7], Heidelberger et al. [13] introduced a fast layered depth image (LDI) generation approach, which can be extended to voxelization. Though the algorithm achieves relatively accurate results, its performance is greatly dependent on the scene depth complexity. In addition, the accessed depth-lists have to be sorted at each frame.

Dong et al. [5] first proposed to encode the information of multiple voxels in one fragment, which dramatically reduces the rendering passes and increases the efficiency. They also proposed to solve the aforementioned projection artifact by voxelizing the model along three orthogonal directions. Each polygon is projected along the direction in which the projected polygon has the largest area. Then the volumetric representations in three directions are composited to get the final results. A similar idea was adopted in [17] to generate the complex boundary in fluid simulation. The composition operation is, however, quite costly in the GPU when the volume resolution is high. In addition, it only partially alleviates the artifacts caused by projection and cannot ensure conservatively correct results.

Eisemann et al. [6] presented a scene voxelization approach for the purpose of rendering. Basically, they used the same pipeline as [5]. The difference is that their approach performs perspective projection along only one direction (the viewing direction), and hence achieves high efficiency. This scheme misses a considerable number of voxels, especially when the input polygon is nearly parallel to the viewing direction. The resultant artifacts are negligible for most rendering applications, while they are undesirable for other applications like collision detection.

A work that is related to our conservative voxelization algorithm is the conservative rasterization technique introduced by Hasselgren et al. [11]. Another solution to conservatively correct rasterization was proposed in [1]. Conservative correctness in rasterization is of essential importance for our algorithm. In practice, we adopt the method of [11] and combine it with our approach.

3 The conservative voxelization algorithm

We introduce our algorithm in terms of triangular models. For an arbitrary polygonal model, we first tessellate it into a triangular model.

Under an orthogonal projection, a column of voxels $V(x, y) = \{x, y, z | z = 0, \ldots, z_{\text{res}} - 1\}$ are projected into

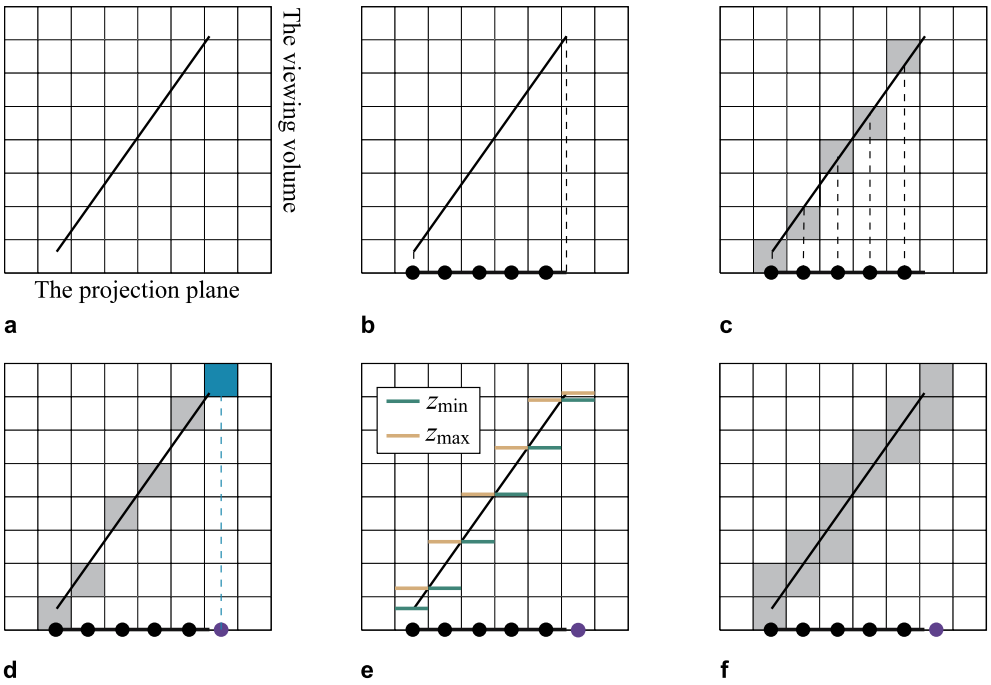


Fig. 2a–f. Illustrations of previous GPU-based algorithms and our solution in 2D. **a** The input data; **b** multiple voxels are projected to a single fragment; **c** the result by previous GPU-based techniques; **d** incorporating conservative rasterization. By computing the actual depth range of each fragment (**e**), our approach yields conservatively correct results (**f**)

the same fragment (x, y) in the viewing plane. When one triangle T intersects with multiple voxels in $V(x, y)$, only one fragment (x, y) will be generated by rasterizing the projected triangle T^p (Fig. 2b). In other words, each fragment (x, y) generated by rasterizing T^p corresponds to one or more voxels in $V(x, y)$ that intersect the triangle T.

As illustrated in Fig. 2c,d, previous approaches generate one voxel per fragment by using the depth value in the fragment center. In contrast, we propose to compute the exact intersection between the triangle T and the prism formed by the column of voxels $V(x, y)$. This is identical to computing the depth range in the intersection region of the projected triangle T^p and the fragment rectangle R (see Fig. 3). If the depth range for some fragment (x, y) is $[z_{\min}, z_{\max}]$, this scheme results in a set of voxels $(x, y, z^*_{\min}), (x, y, z^*_{\min}+1), \cdots, (x, y, z^*_{\max})$, where $z^*_{\min} = \lfloor z_{\min} \cdot \mathrm{res}_z \rfloor$ and $z^*_{\max} = \lfloor z_{\max} \cdot \mathrm{res}_z \rfloor$. Figure 2e and f simply depict our key idea.

If the fragment R is fully covered by T^p (called an *interior* fragment), the depth range can be easily calculated. Suppose that the depth value in the fragment center is z_c and the partial derivatives of the depth value are $(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y})$, the minimal and maximal depth values will be $z_{\min} = z_c - \Delta z$, $z_{\max} = z_c + \Delta z$, and $\Delta z = \frac{1}{2}(|\frac{\partial z}{\partial x}| + |\frac{\partial z}{\partial y}|)$.

In the case that R is partially covered by T^p (this called a *boundary* fragment), the computation is more complicated. One simple scheme is to directly employ the depth range derived above. This leads to over-conservative results, especially when the voxelizing triangle is nearly parallel to the projection direction.

To obtain an accurate depth range, we need to consider the intersection between R and T^p. This is a polygon (denoted by I) with up to seven vertices as shown in Fig. 3a. The polygon I is convex, and the maximal and minimal depth values must appear on its vertices. We can compute the depth values of all vertices and find the depth range. As illustrated in Fig. 3b, the vertices of I consist of the intersection points between the triangle and the fragment rectangle, the vertices of the triangle that are inside the fragment, and the vertices of the fragment that are inside the triangle.

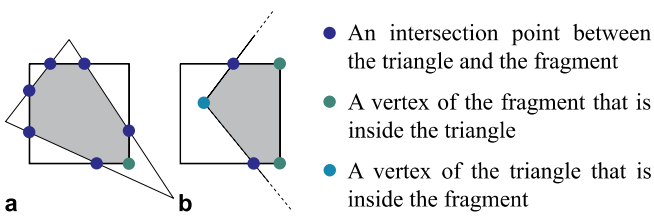


Fig. 3a,b. Intersection of a triangle with a fragment. **a** The intersection polygon has up to seven vertices; **b** three kinds of vertices of the intersection polygon

Let $v_{\min}$ be the vertex of R that satisfies $z(v_{\min}) = \min_{p \in R} z(p)$. If $v_{\min}$ lies in T^p, then $z_{\min} = z(v_{\min})$ $min = z_c - \Delta z$. Otherwise, the minimal depth value will not appear in any vertex of R except if the vertex is on the boundary of T^p, i.e., it coincides with some intersection point. The strict proof is presented in Theorem 1 in the Appendix. Algorithm 1 summarizes the process to compute the minimal depth value. The maximal depth value can be evaluated in the same way.

Algorithm 1. Computing the minimal depth $z_{\min}$

```
if v_min ∈ T^p then
    z_min ← z(v_min)
else
    z_min ← HUGE_VALUE
    for each intersection point Q_i do
        z_min ← min(z_min, z(Q_i))
    end for
    for each vertex P_i of T do
        if P_i ∈ R then
            z_min ← min(z_min, z(P_i))
        end if
    end for
end if
```

3.1 Computing the minimal depth value

For the sake of simplicity, we use the following denotations. The vertices of T^p are denoted by $P_i(x_i, y_i)$ and the corresponding depth values are z_i, $i = 1, 2, 3$. $R = [x_{\min}, x_{\max}; y_{\min}, y_{\max}]$. The left, right, bottom and top edges of R are denoted by e_L, e_R, e_B and e_T, respectively.

To determine whether $v_{\min}$ is inside or outside T^p, we compute the barycentric coordinate $\mathbf{C}_{\min}$ of $v_{\min}$ with respect to T^p. If and only if all three components of $\mathbf{C}_{\min}$ are non-negative, $v_{\min} \in T^p$. $\mathbf{C}_{\min}$ is calculated by $\mathbf{C}_{\min} = \mathbf{C} - \Delta\mathbf{C}$, where $\mathbf{C}$ denotes the barycentric coordinate of the fragment center, $\Delta\mathbf{C}$ is the difference between $\mathbf{C}$ and $\mathbf{C}_{\min}$, and is computed as:

$$\Delta\mathbf{C} = \frac{1}{2} \cdot sign\left(\frac{\partial z}{\partial x}\right) \cdot \frac{\partial \mathbf{C}}{\partial x} + \frac{1}{2} \cdot sign\left(\frac{\partial z}{\partial y}\right) \cdot \frac{\partial \mathbf{C}}{\partial y}. \tag{1}$$

The signed area of T^p is:

$$S = x_1 y_2 - x_2 y_1 + x_2 y_3 - x_3 y_2 + x_3 y_1 - x_1 y_3. \tag{2}$$

And we have

$$\frac{\partial \mathbf{C}}{\partial x} = \frac{(y_2 - y_3,\ y_3 - y_1,\ y_1 - y_2)}{S}$$
$$\frac{\partial \mathbf{C}}{\partial y} = \frac{(x_3 - x_2,\ x_1 - x_3,\ x_2 - x_1)}{S}, \tag{3}$$

and

$$\frac{\partial z}{\partial x} = (z_1, z_2, z_3) \cdot \frac{\partial \mathbf{C}}{\partial x}, \quad \frac{\partial z}{\partial y} = (z_1, z_2, z_3) \cdot \frac{\partial \mathbf{C}}{\partial y}. \tag{4}$$

The computation of $\Delta\mathbf{C}$ is performed in the vertex processing stage. For each vertex P_i, we compute $\Delta\mathbf{C}$ and output $\mathbf{C}^* = \mathbf{C}(P_i) - \Delta\mathbf{C}$ as a 3D vector[1]. $\mathbf{C}_{\min}$ is then automatically obtained by a bilinear interpolation in the rasterization stage.

The intersection between T^p and R is obtained by computing the intersection between each edge of T^p and each edge of R. For each pair of edges, we first calculate the intersection point between their corresponding lines. Let $a_{ij}^x = \frac{1}{x_j - x_i}$, $b_{ij}^x = \frac{-x_i}{x_j - x_i}$, $a_{ij}^y = \frac{1}{y_j - y_i}$, $b_{ij}^y = \frac{-y_i}{y_j - y_i}$; the intersection point Q of $\overline{P_i P_j}$ with e_T is computed by

$$Q = \mathcal{L}(P_i, P_j, t), \quad t = a_{ij}^y \cdot y_{\max} + b_{ij}^y, \tag{5}$$

where $\mathcal{L}$ denotes the linear interpolation function. The intersections between $\overline{P_i P_j}$ and other edges of R are computed similarly. Subsequently, we examine whether Q is on both edges. If this is true, we set $z(Q) = \mathcal{L}(z_i, z_j, t)$, otherwise, we assign $z(Q)$ a huge value. Finally, we compute the minimal value among the depth values associated with 12 intersection points.

Note that the GPU has native support to 4D vector processing. It allows us to simultaneously compute the intersection points between one edge of T^p and four edges of R. Likewise, the comparison of 12 depth values can also be greatly accelerated. Specifically, it can be done by four comparisons (two comparisons of two 4D vectors, one comparison of two 2D vectors and one comparison of two scalars).

3.2 Handling special cases

In computing the depth ranges, there are several ill-posed cases that need to be handled carefully.

3.2.1 Intersection of parallel lines

The computation of the intersection of two lines becomes numerically unstable when they are approximately parallel. An example is illustrated in Fig. 4a. The edge $\overline{P_i P_j}$ intersects with the fragment R at Q_L and Q_T. If $\overline{P_i P_j}$ is nearly horizontal, i.e., $y_i \approx y_j$, computing Q_T by Eq. 5 will result in a large error [20].

If we simply omit Q_T, the result may be no longer conservatively correct when Q_T has the minimal or maximal depth value in the intersection region. Our solution is to replace Q_T with Q_R, where Q_R is the intersection point of $\overline{P_i P_j}$ with the prolonged line of e_R. This scheme does not decrease the depth range because the depth value of Q_T must be in the range of $z(Q_L)$ and $z(Q_R)$. In addition, this scheme is robust. As $\overline{P_i P_j}$ is nearly horizontal, Q_R can be correctly computed.

[1] The barycentric coordinates for three vertices of T^p are (1,0,0), (0,1,0) and (0,0,1).

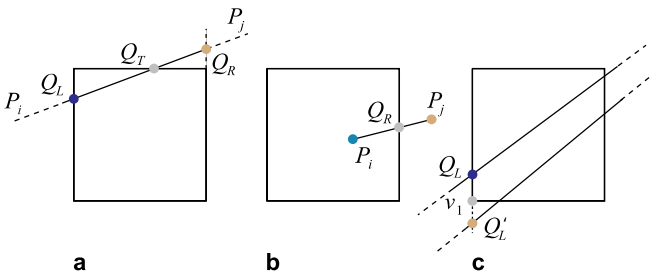


Fig. 4a–c. Robust computation of the depth range. A gray dot represents a vertex of the intersection polygon which cannot be robustly computed. We replace it with the orange point which is actually outside (but very close to) the intersection region. Three ill-posed cases: **a** The edge $\overline{P_iP_j}$ is approximately parallel to one edge of the fragment rectangle; **b** the edge $\overline{P_iP_j}$ is nearly degenerate; **c** the projected triangle is nearly degenerate

The practical implementation is as follows. When $|\frac{y_j-y_i}{x_j-x_i}|$ is smaller than some threshold ε_I, we set $a_{ij}^y = 0$, $b_{ij}^y = 2$, to discard all intersections of $\overline{P_iP_j}$ with e_T and e_B. Accordingly, we slightly enlarge R in its height to include both Q_L and Q_R. To determine whether a certain intersection point $Q(x, y)$ is on e_L or e_R, we test whether it satisfies $y_{\min} - \varepsilon_I \le y \le y_{\max} + \varepsilon_I$. The condition is always satisfied when $\overline{P_iP_j}$ intersects e_T or e_B.

3.2.2 Degenerate edges

Two vertices in a triangle may be projected to very close positions. Figure 4b illustrates an example that the near degenerate edge $\overline{P_iP_j}$ intersects e_R at Q_R. As both $|x_j - x_i|$ and $|y_j - y_i|$ are considerably small, the intersection of $\overline{P_iP_j}$ with all edges of R cannot be correctly calculated. Moreover, all intersection points (Q_R in this example) must be omitted. To maintain the conservative correctness, we assume that both vertices of the edge are inside the fragment. The implementation is similar to the former case. Given that $|x_j - x_i| < \varepsilon_I$ and $|y_j - y_i| < \varepsilon_I$, we consider the vertex $P_k(x_k, y_k)$ of T^p to be inside R if and only if $x_{\min} - \varepsilon_I \le x_k \le x_{\max} + \varepsilon_I$, $y_{\min} - \varepsilon_I \le y_k \le y_{\max} + \varepsilon_I$.

3.2.3 Degenerate triangles

When a triangle is nearly parallel to the projection direction, the projected area S is extremely small. When $|S|$ is smaller than a certain threshold ε_S, both the barycentric coordinate and depth derivatives cannot be reliably computed. In other words, we can neither determine whether a point is inside the projected triangle nor compute its depth value. Figure 4c shows an example that the bottom left vertex v_1 of R is inside the triangle and has to be removed. Based on Theorem 2 presented in the Appendix, we understand that there must be two intersection points, say Q_L and Q'_L, satisfying $\|Q_L - v_1\| \le \sqrt{2}h$, $\|Q'_L - v_1\| \le \sqrt{2}h$, where h is the minimal height of the triangle. As $|S|$ is small, h is also small. Similarly to the above two cases, we can safely ignore v_1 and adopt both Q_L and Q'_L by slightly enlarging the fragment R.

In summary, we list the detailed implementation in Algorithm 2. The computation is performed with vertex processing and is duplicated three times for each triangle. The next-generation GPUs support carrying out the computation with geometry processing, which leads to higher efficiency.

Algorithm 2. Robust computation of the depth range

Let x_E and y_E be the enlarged width and height of R
Let ε_m denote the machine error in GPU
$x_E, y_E \leftarrow \varepsilon_m$
for each edge $\overline{P_iP_j}$ in Triangle T^p **do**
  if $|y_j - y_i| < \max(\varepsilon_I, \varepsilon_I|x_j - x_i|)$ **then**
    $a_{ij}^y \leftarrow 0, b_{ij}^y \leftarrow 2$
    $y_E \leftarrow \max(y_E, \varepsilon_I)$
  else
    $a_{ij}^y \leftarrow \frac{1}{y_j - y_i}, b_{ij}^y \leftarrow \frac{-y_i}{y_j - y_i}$
  end if
  Compute a_{ij}^x, b_{ij}^x similarly.
end for
Compute the projected area S according to Eq. 2
if $|S| < \varepsilon_S$ **then**
  $\mathbf{C}^* \leftarrow (-1, -1, -1)$
  $x_E \leftarrow \max(x_E, \sqrt{2}h)$, $y_E \leftarrow \max(y_E, \sqrt{2}h)$
else
  Compute $\mathbf{C}^*$ according to Sect. 3.1
end if

4 An alternative approach

We tried another pre-computation based approach to calculate the depth range. Each intersection point of the projected triangle with the fragment rectangle is assigned a mark according to its location, where 'L', 'R', 'B', 'T' denote the cases that it lies on the left, right, bottom and top edges of the fragment, respectively. The special mark 'N' means that the intersection point does not exist. The marks of six potential intersection points make up an *intersection code*. The location of the vertex with extreme depth value is then completely determined by the

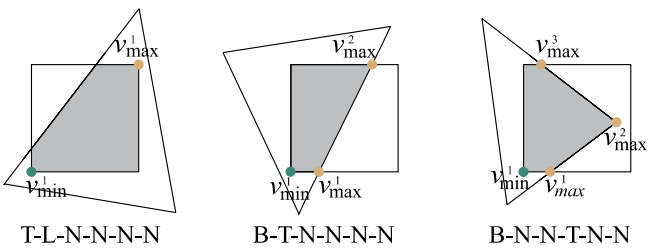


Fig. 5. Example cases for the pre-computation scheme and the corresponding intersection codes

intersection code and the signs of $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. Figure 5 shows some typical cases and the corresponding intersection codes. The vertices that potentially have minimal or maximal depth value are marked as $v^i_{\min}$ and $v^i_{\max}$, $i = 1, 2, 3$, assuming $\frac{\partial z}{\partial x} > 0$ and $\frac{\partial z}{\partial y} > 0$. In all cases, the minimal and maximal depth values will appear at no more than three vertices. Accordingly, we can precompute the locations of the extreme points for all cases and store them as a lookup table for later access.

Intuitively, this scheme will run faster than the method presented in the last section. However, it cannot be conveniently implemented in current graphics hardware. On the one hand, the algorithm implies a switch-case like program structure, whose implementation in hardware is expensive. On the other hand, it is costly to compute the locations of the intersection points. New hardware features will probably overcome these problems and make this approach feasible.

5 Hardware implementation details

After obtaining the depth range of a fragment, we need to recover the corresponding voxels and record the information in the frame buffer.

5.1 Volume encoding

We adopt the encoding technique in [5], which records the occupancy information of a voxel in one bit, where '1' means that the voxel intersects with the input model and '0' means no intersection. In this way, a texel of a texture with the 'RGBA8' format can be used to record information for 32 voxels, and a $\text{res}_x \times \text{res}_y$ texture can be used to record the voxelization result at the volume resolution of $\text{res}_x \times \text{res}_y \times 32$. When the desired resolution res_z is larger than 32, we have to divide the volume into several sub-volumes and perform voxelization in multi-passes. In each pass, we use the OpenGL logical 'OR' operation to encode multiple voxels intersecting with different triangles into one fragment in the frame buffer, as in [6].

Given a depth range $(z_{\min}, z_{\max})$, a color value, or *bit-mask* that represents the corresponding voxels is obtained by a simple texture lookup. The texture that serves as the lookup table is of size 32×32. It records the bit-masks corresponding to all $(z^*_{\min}, z^*_{\max})$ pair, where $z^*_{\min} = \lfloor z_{\min} \times 32 \rfloor$, $z^*_{\max} = \lfloor z_{\max} \times 32 \rfloor$. The texture is precomputed and stored in the video memory.

The number of required rendering passes can be efficiently reduced by exploiting the 'MRT (multiple render targets)' functionality of modern graphics hardware. This enables multiple textures to be bound into the frame buffer simultaneously. In our test platform, we can set up to four render targets. This facilitates processing 128 slices in a single pass. Accordingly, the lookup table texture must be modified because the number of permutations of $z^*_{\min}$ and $z^*_{\max}$ increases to 128×128. In addition, each bit-mask consists of 128 bits and has to be stored in four texels. In practice, we precompute all bit-masks and store them in a 512×128 texture.

5.2 Combination with conservative rasterization

Ideally, conservative rasterization should be naturally supported by graphics hardware, which only requires a minor modification to the hardware rasterization. Unfortunately, such a feature is currently not supported. In practice, we adopt the approach introduced in [11]. The basic idea is to enlarge each triangle by half a pixel and draw the resulting bounding polygon. One can either draw the bounding polygon directly, or draw an optimal bounding triangle and cull the redundant fragments. We choose the second scheme because it is more efficient in most of our experiments.

6 Experimental results

We have implemented the proposed algorithm on a PC with an AMD AthlonXP 3000+ CPU, 512 M RAM and NVidia 6600 GT video card. The shaders are written in the Cg shading language [19].

6.1 Accuracy analysis

Theoretically, our approach could lead to an accurate voxelization result: a voxel is generated if and only if it intersects the input model. In practice, it generates some redundant voxels that actually have no intersection with the model.

In Sect. 3.2, we discuss several special cases of the computation of the depth ranges. When a certain vertex of the intersection polygon cannot be robustly calculated, we replace it with another point that is outside the intersection region. This scheme may lead to over-conservative results. However, these special cases rarely take place in our experiments. Meanwhile, the introduced voxelization errors are small and negligible because the virtually added vertices are very close to the intersection region.

The floating computation in GPUs has a limited precision, which leads to some errors in the computed depth ranges[2]. To address this problem, we simply enlarge the depth range by a small constant value (10^{-5} in our implementation). This scheme may also lead to over-conservative results, even if very rarely.

To verify our algorithm, we build a software-based voxelization system that achieves conservatively correct results by taking the exact intersection of triangles with voxels. We compare the results with those generated by our conservative voxelization algorithm running in the GPU. For all tested datasets, our approach does not miss

[2] Most GPUs support single precision floating computation.

any voxel and the number of redundant voxels is no more than 0.1% of the total number.

6.2 Performance analysis

Table 1 shows the configuration and timings for various models under different volume resolutions. The major time consumption lies in the computation of the depth range for each fragment using Algorithm 1. The cost for the interior fragments is low and much higher for the boundary fragments. Although a large amount of fragments are produced when the volume resolution is high, the average cost for the fragment processing is not high because most fragments are interior fragments. As a result, our approach is very efficient for high volume resolutions.

On the contrary, however, our algorithm is not very efficient for complex models, even when the volume resolution is low. The reason is that voxelizing complex models generates many boundary fragments, yielding a high average cost of the fragment processing. It also slows down the SIMD branching of the fragment processing [10]. Decreasing the volume resolutions has little effect because the number of the boundary fragments is no less than the total triangle numbers with the conservative rasterization scheme.

6.3 Comparison with Dong's Method

The algorithm proposed by Dong et al. [5] is the fastest voxelization algorithm so far that provides an approximately conservatively correct voxelization results. We compare the efficiency of our algorithm with Dong's algorithm in Fig. 6.

For complex models and low volume resolution, Dong's approach runs faster than ours. This is because Dong's method does not compute the depth range for each fragment, which is costly for complex models. Our approach outperforms Dong's approach when the volume resolution becomes higher. For high volume resolutions, the cost is quite high. For example, at the volume resolution of 512^3, the time for composition is about 300 ms (Fig. 6). Likewise, at the volume resolution of 1024^3, composition will cost at least 2400 ms because there are 8 times more voxels at 512^3. Apparently, composition tends to be a heavy bottleneck with an increase of the volume resolution.

Table 1. The voxelization timings in ms

Model	#Triangle	128^3	256^3	512^3	1024^3
Torus	1600	11.7	11.8	35.2	117.6
Mannequin	23 402	11.9	23.5	35.3	82.3
Bunny	69 451	35.3	35.3	47.0	94.1
Buddha	209 962	82.3	94.1	129.4	258.8
Hand	654 666	214.0	261.3	284.0	313.5
Dragon	871 326	294.0	353.0	360.0	418.3

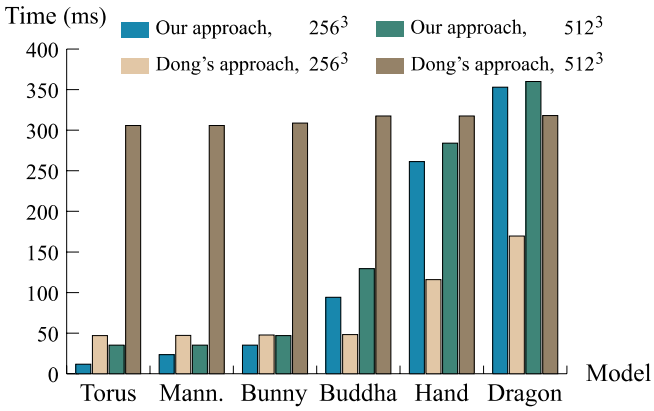


Fig. 6. Performance comparison between Dong's method and our method. The testing models are listed in Table 1

Figure 7 compares the voxelization results generated by our method and Dong's method. The rendering of the volumes is accomplished by blending the volume slices, which are orthogonal to the viewing direction. The blueness is proportional to the thickness of the volume. As shown in the figure, our results contain more voxels than those in Dong's approach. The right column shows the difference between each pair of volumes.

Our approach has a lower memory consumption than Dong's method. Dong's approach needs to store the volume representation in three directions and requires several large-sized textures to serve as lookup tables for the composition operation. Instead, our approach stores the volume representation only in one direction and consumes several small textures for lookup operations. In fact, running Dong's method at a volume resolution of 1024^3 would be out of video memory in our test platform, while ours supports even larger volume resolutions.

6.4 Application to collision detection

We develop a collision detection system based on the proposed approach. By voxelizing two test models with a common bounding box, their intersection is efficiently obtained through a direct comparison of the two resulting volumes. Generally speaking, the error of collision detection is inversely proportional to the volume resolution. Because our approach is efficient at high volume resolutions, high accuracy of collision detection can be achieved. In addition, our approach is conservatively correct, facilitating correct classification of colliding regions. The colliding voxels represent potential intersection regions and the non-colliding voxels represent regions where two models do not collide. Therefore, a large portion of regions where two models do not collide can be efficiently pruned. Additionally, our approach is suitable for the collision detection of deformable models because no pre-processing for the deformation is required.

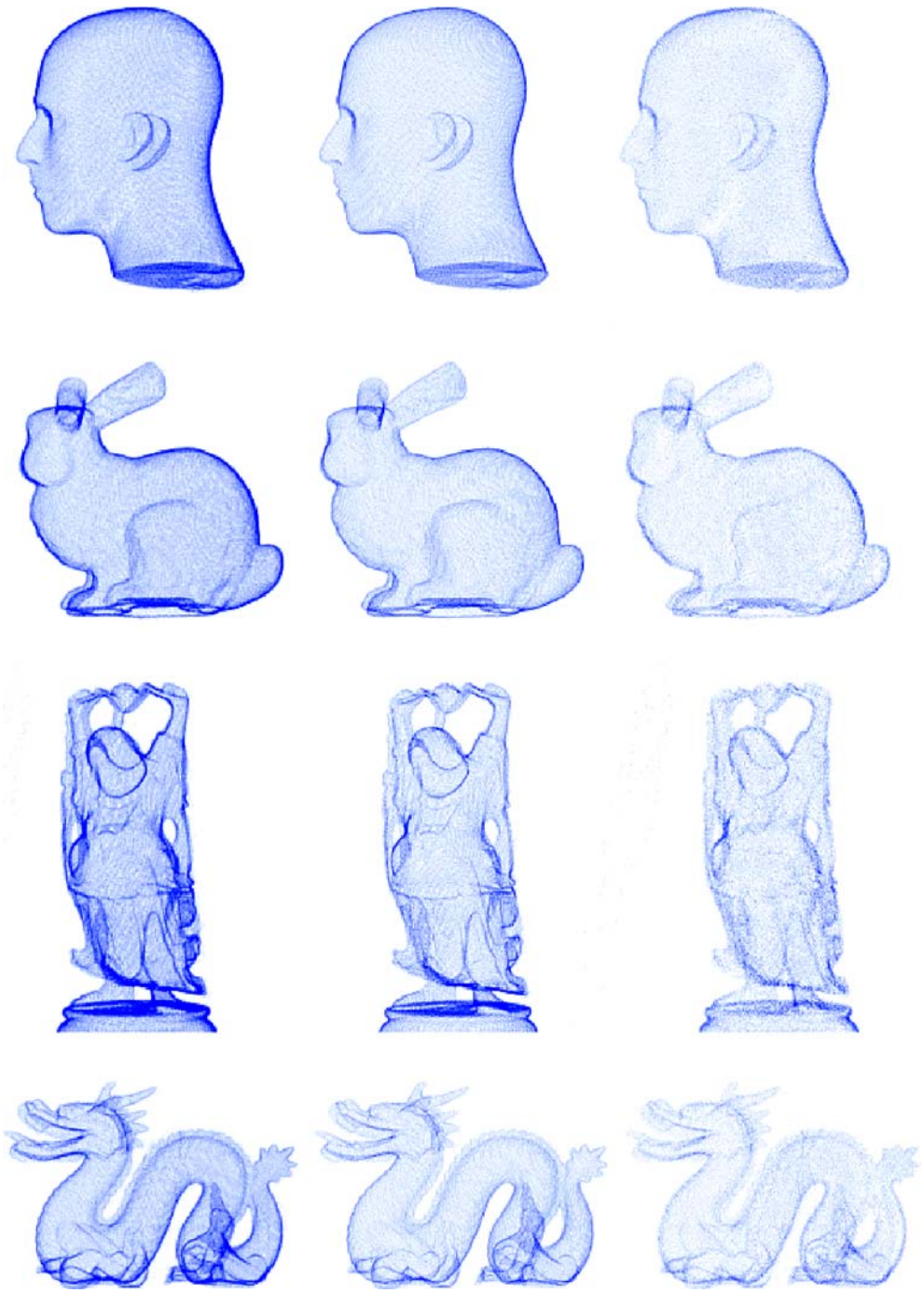


Fig. 7. Comparison of our approach (*left column*) and Dong's approach (*middle column*). The right column shows the differences. From *top* to *bottom*: Mannequin, bunny, Buddha and dragon models

Figure 8 shows a collision detection example between the Buddha model (209 962 triangles) and the MoHand model (5399 triangles). The MoHand model keeps morphing among three key frames. The voxelizations of the two models take 94 and 12 ms at a volume resolution of 256^3. The subsequent collision query costs 8 ms. In total, the collision detection is accomplished in approximately 114 ms.

Fig. 8. Interactive collision detection based on conservative voxelization

7 Conclusions and future work

In this paper we have investigated the problems induced in the projection and rasterization stages of GPU-based voxelization. We have proposed an easy-to-use technique to achieve conservatively correct results. Our approach traverses the handled model only once and yields high efficiency, as demonstrated in our experiments.

As far as future work is concerned, we would like to try more efficient algorithms such as the one proposed in Sect. 4 by using new graphics hardware features. We shall also seek to exploit more useful applications of conservative voxelization. With the newest video cards, the OpenGL logical operations are only available for 8-bit integer textures. If logical operations for 16-bit or 32-bit integer textures are supported, fewer rendering passes are required and the performance will be improved dramatically.

A Appendix

Theorem 1. *Let $f(x, y)$ be a scalar function defined in $\mathbb{R}^2$, which satisfies that $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ are constants. Let R be a rectangle, T be a triangle, and I be the nonempty intersection region between R and T. Let $\mathbf{v}$ be a vertex of R, $\mathbf{v} \in (I - \delta T)$, where δT is the boundary of T. If $f(\mathbf{v}) = \min_{\mathbf{x}\in I} f(\mathbf{x})$, then $f(\mathbf{v}) = \min_{\mathbf{x}\in R} f(\mathbf{x})$.*

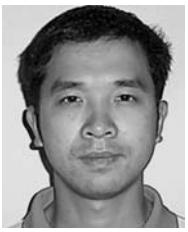
Proof. Let $\mathbf{x}$ be an arbitrary point in R. We have $f(\mathbf{x}) - f(\mathbf{v}) = (\mathbf{x} - \mathbf{v}) \cdot (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$. Because $\mathbf{v} \in (I - \delta T)$, we can find some $\varepsilon > 0$, such that $\mathbf{x}' = \mathbf{v} + \varepsilon(\mathbf{x} - \mathbf{v}) \in I$. Then $f(\mathbf{x}') - f(\mathbf{v}) = \varepsilon(\mathbf{x} - \mathbf{v}) \cdot (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$. Because $f(\mathbf{v}) = \min_{\mathbf{x}\in I} f(\mathbf{x})$, $\varepsilon(\mathbf{x} - \mathbf{v}) \cdot (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}) \leq 0$, and $(\mathbf{x} - \mathbf{v}) \cdot (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}) \leq 0$. Therefore, $f(\mathbf{x}) \leq f(\mathbf{v})$. □

Theorem 2. *Let l_1 and l_2 be two orthogonal straight lines intersecting at point P, $P \in \triangle ABC$. The intersection points of the two lines with the triangle are denoted by D, E, F and G respectively. We must have $PD \leq \varepsilon$, $PE \leq \varepsilon$ or $PF \leq \varepsilon$, $PG \leq \varepsilon$, where $\varepsilon = \sqrt{2}h$, and h is the minimal height of $\triangle ABC$.*

Proof. Without loss of generality, we assume that edge BC is largest and AH is the corresponding height, $AH = h$. Let $\angle(l_1, AH) = \theta$, $\angle(l_2, AH) = \varphi$. From $l_1 \perp l_2$, we know that $\min(\theta, \varphi) \leq \pi/4$. If $\theta \leq \varphi$, we create a line l' passing D, $l' /\!/ BC$, and $PI \perp l'$, $PI \cap l' = I$. Then $PD = PI/\cos\theta$. Since $PI \leq h$ and $\theta \leq \pi/4$, $PD \leq \sqrt{2}h$. Similarly, $PE \leq \sqrt{2}h$. If $\varphi \leq \theta$, We can prove $PF \leq \sqrt{2}h$ and $PG \leq \sqrt{2}h$ in the same way. □

References

1. Akenine-Möller, T., Aila, T.: Conservative and tiled rasterization using a modified triangle set-up. ACM J. Graph. Tools **10**(3), 1–8 (2005)
2. Beckhaus, S., Wind, J., Strothotte, T.: Hardware-based voxelization for 3D spatial analysis. In: Proceedings of the 5th International Conference on Computer Graphics and Imaging, pp. 15–20. ACTA Press, Canmore, Alberta, Canada (2002)
3. Boyles, M., Fang, S.: Slicing-based volumetric collision detection. ACM J. Graph. Tools **4**(4), 23–32 (2000)
4. Chen, H., Fang, S.: Fast voxelization of 3D synthetic objects. ACM J. Graph. Tools **3**(4), 33–45 (1999)
5. Dong, Z., Chen, W., Bao, H., Zhang, H., Peng, Q.: Real-time voxelization for complex polygonal models. In: Proceedings of Pacific Graphics 2004, pp. 43–50, IEEE Comput. Graph. Soc. Press (2004)
6. Eisemann, E., Décoret, X.: Fast scene voxelization and applications. In: Proceedings of 2006 Symposium on Interactive 3D Graphics and Games, pp. 71–78. ACM Press, New York (2006)
7. Everitt, C.: Interactive order-independent transparency. Technical report, NVIDIA Corporation (2001)
8. Fang, S., Chen, H.: Hardware accelerated voxelization. Comput. Graph. **24**(3), 433–442 (2000)
9. Gagvani, N., Silver, D.: Shape-based volumetric collision detection. In: Proceedings of the IEEE Symposium on Volume Visualization 2000, pp. 57–61. ACM Press, New York (2000)
10. Harris, M., Buck, I.: GPU flow-control idioms. GPU Gems II **34**, 547–555 (2005)
11. Hasselgren, J., Akenine-Möller, T., Ohlsson, L.: Conservative rasterization. GPU Gems II **42**, 677–690 (2005)
12. He, T., Kaufman, A.: Collision detection for volumetric objects. In: Proceedings of IEEE Visualization 1997, pp. 27–35. IEEE Computer Society Press, New York (1997)
13. Heidelberger, B., Teschner, M., Gross, M.: Volumetric collision detection for deformable objects. Technical Report No.395, Institute of Scientific Computing, ETH Zürich (2003)
14. Karabassi, E.A., Papaioannou, G., Theoharis, T.: A fast depth-buffer-based voxelization algorithm. ACM J. Graph. Tools **4**(4), 5–10 (1999)
15. Kaufman, A., Shimony, E.: 3D scan-conversion algorithms for voxel-based graphics. In: Proceedings of ACM Workshop on Interactive 3D Graphics, pp. 45–76. ACM Press, Chapel Hill, NC (1986)
16. Kreeger, K., Kaufman, A.: Mixing translucent polygons with volumes. In: Proceedings of IEEE Visualization 1999, pp. 191–198. (1999)
17. Li, W., Fan, Z., Wei, X., Kaufman, A.: Flow simulation with complex boundaries. GPU Gems II **47**, 677–690 (2005)
18. McNeely, W., Puterbaugh, K., Troy, J.: Six degree-of-freedom haptic rendering using voxel sampling. In: Proceedings of ACM SIGGRAPH 1999, pp. 401–408 (1999)
19. NVIDIA Corporation: Cg Specification (2006)
20. Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, New York (1992)
21. Wang, S., Kaufman, A.: Volume sampled voxelization of geometric primitives. In: Proceedings of IEEE Visualization 1993, pp. 78–84. IEEE Computer Society Press (1993)
22. Wang, S., Kaufman, A.: Volume-sampled 3D modeling. IEEE Comput. Graph. Appl. **14**(5), 26–32 (1994)




LONG ZHANG received his bachelor's degree in 2003 from the Department of Applied Mathematics, Zhejiang University. He is currently a PhD candidate of the State Key Lab of CAD&CG, Zhejiang University. His research interests include real-time rendering and computer animation.

DR. WEI CHEN is an associate professor at State Key Lab of CAD&CG at Zhejiang University, P.R. China. From June 2000 to June 2002 he was a joint PhD student in Fraunhofer Institute for Graphics, Darmstadt, Germany and he received his PhD degree in July 2002. From June 2006 to June 2008 he is a visiting scholar at Purdue University, USA. His current research interests include visualization and efficient modeling and photo-realistic rendering.

DAVID S. EBERT is a full professor in the School of Electrical and Computer Engineering at Purdue University. His research interests include scientific, medical, and information visualization; computer graphics; animation; procedural techniques; volume rendering; illustrative visualization; realistic rendering; procedural texturing; modeling; and modeling natural phenomena. Ebert has a PhD from the Computer and Information Science Department at The Ohio State University. He has served on the ACM Siggraph Executive Committee and was the editor in chief of IEEE Transactions on Visualization and Computer Graphics.

PROF. QUNSHENG PENG received his PhD degree from the School of Computing studies of East Anglia University, UK in 1983 and is currently a professor and doctoral supervisor at the State Key lab of CAD&CG, Zhejiang University. His research areas are virtual reality, computer animation, visualization and infrared simulation.