Conservative Sampling of Solids in Image Space

Yuen-Shan Leung, and Charlie C.L. Wang, *Member, IEEE*

***Abstract*— This paper presents a new method for sampling B-rep solid models into *Layered Depth Images* (LDI). The boundary of the sampled models represented by LDI is closed and the sampled models are guaranteed to bound the input B-rep models on the rays of LDI (called *conservative*). Our sampling method can be fully implemented by shader programs supported by various graphics hardware. Experimental results demonstrate the efficiency of the proposed method, and the applications in intersecting volume evaluation and Minkowski sum computation are given at the end of this paper to show the versatility of our approach.**

***Index Terms*— Sampling, Solid Model, Image Space, Layered Depth Images, GPU.**

I. Introduction

The representation of geometric objects based on volumetric data structure has advantages in many applications, including collision detection, haptic rendering and Boolean operations, as it provides a compact and robust description of solid models. Recently, more and more approaches start to sample B-rep models into volumetric data and store them in image space (e.g., [1]–[5]). Many of them employ a representation called *Layered Depth Image* (LDI), which is originally proposed in [6] for rendering purpose. An LDI represents a model H by a 2D array of pixels viewed from a single camera with parallel rays that pass through the centers of pixels. Each LDI pixel stores depth values of the intersection points between the ray and the boundary surface of H; usually there are multiple samples on a ray (therefore multiple layers of images). An LDI representation of solid H should always have an even number of samples sorted by depth on its rays, and the portion between the $(2i+1)$-th and $(2i+2)$-th samples ($i = 0, 1, \cdots$) on a ray must locate inside the solid H. This *boundary closure* property has been employed in many applications to compute discrete forms of volumetric metrics (e.g., intersecting volume [7]). One advantage of using LDI is that the sampling process and other related computations can be accelerated by modern graphics hardware equipped with a *Graphics Processing Unit* (GPU). However, the existing methods may fail (e.g., [1]) or give poor performance (e.g., [5]) when the layer-complexity of an input model is high (e.g., the models shown in Fig.1). In such a circumstance, the LDI sampling step would become the performance bottleneck of the whole system.

In this paper, we present a robust and efficient algorithm for sampling the *boundary representation* (B-rep) of a solid model H into an LDI representation. While most existing GPU approaches (ref. [1]–[5]) require a pre-/post-sorting step in their algorithms, our method generates sorted samples during the process. The main idea is to exploit the bitwise information obtained from solid voxelization and fetch the depths of LDI samples with a novel fast bit checking technique together with the MAX/MIN blending operations. The LDI solids generated by our approach are guaranteed to bound the input B-rep models on the rays of LDI, therefore they can preserve *conservativeness* in image space. Usually, there are two kinds of conservativeness on a solid model, namely *overestimated* and *underestimated*. An overestimated conservative sampling means that the result along each ray encloses every interior segment of an input object on this ray, whereas an underestimated sampling generates segments that lie fully inside the input object (see Fig.2 for an illustration). In the rest of this paper, the term conservativeness means an overestimated one without any further specification.

Major Results:

- We present an efficient solid sampling approach in this paper. Our approach can generate samples of Layered Depth Images (LDI) in $(2 + n/32)$ rendering passes for an input solid H with n layers in the sampling direction. Moreover, this method is scalable and does not limit the maximal number of layers on the input solid.
- Our sampling method can generate samples ordered by depth value without any additional sorting (or linear search) step. It relies on a novel fast bit checking technique developed in this paper.
- LDI solids generated by this method is conservative to an input solid H in the image space, which is very important for many applications (e.g., collision detection and Boolean operations).

Our paper is organized as follows. Section II provides a literature review of related research work. Our sampling method is detailed in Section III. Experimental results and applications are shown and discussed in Section IV and V. Finally, we conclude our paper in Section VI.

II. Related Work

In this section, we give a literature review of relevant work on using accelerated graphics hardware to generate LDI solids from B-rep models. Prior work can be classified into three groups based on when to conduct the sorting step to obtain LDI samples ordered by depth.

One category is to work at a primitive level – sorting the geometries before rasterization. Wexler et al. [8] decompose the scene into smaller sub-scenes and sort them from front-to-back. The sorted sub-scenes are then extracted to form a set of batches and classic depth peeling will be performed

Manuscript submitted on October 20, 2011.
Revision prepared on February 6, 2012.
Authors are with the Department of Mechanical and Automation Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong (Tel: (852) 3943 8052; Fax: (852) 2603 6002; E-mail: cwang@mae.cuhk.edu.hk).

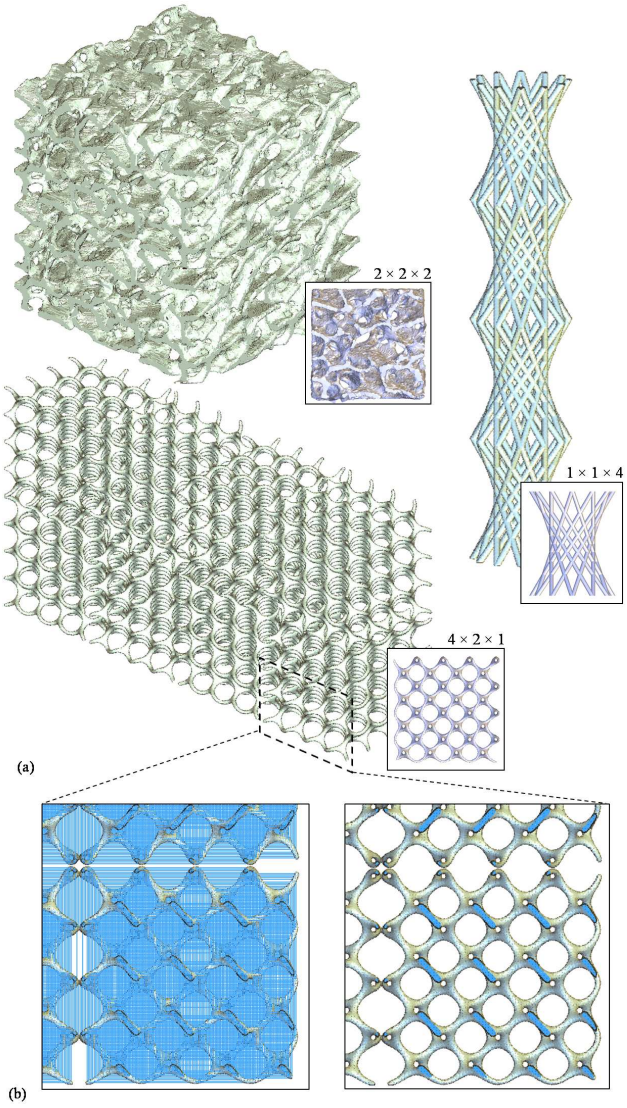


Fig. 1. Our goal is to efficiently sample models with complex layers into samples sorted by depth in image space without pre- or post-sorting. (a) Different types of models with complicated topology, which are difficult to be modeled in conventional geometric modeling systems. (b) Scaffold of trabecular bone consists of $2 \times 2 \times 2$ similar structure units; the hyperboloid model and the nylon lattice model are both created by merging several modeling units with complex topology. As shown in (b), when sampling these models into image space, unsorted samples (left) give an incorrect solid – illustrated by the blue line segments. A correct solid (right) has samples stored on rays in the order of depth.

on each batch. A recent approach, coherent layer peeling [9], also pre-sorts objects and then peels their sorted surfaces in each iteration in order to achieve approximately linear running time. However, this pre-sorting strategy is time-consuming for models with a complex geometry (e.g., the ones shown in Fig.1).

Some approaches sort samples at a fragment level after rasterization. Depth peeling [10] is a classic method which repeatedly rasterizes the same geometry to sort all possible fragments by depth. However, when a ray intersects a silhouette edge shared by two faces having different orientations, one of the two intersections will be missed in depth peeling as they have the same depth value. The boundary closure property,

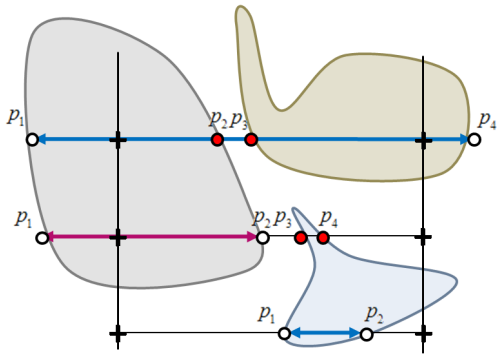


Fig. 2. An illustration of different types of conservativeness on rays, where the range between two crosses represents one voxel. An overestimated conservative sampling on the top ray gives a 1D solid between p_1 and p_4 by discarding the gap between p_2 and p_3 on the given solid, and the underestimated sampling on the middle ray misses the small solid between p_3 and p_4. Our sampling algorithm generates 1) two samples p_1 and p_4 for the top ray, 2) two samples p_1 and p_4 for the middle ray, and 3) two samples p_1 and p_2 for the bottom ray.

which requires two samples to be output here, is not satisfied. To solve this problem, Heidelberger et al. [1] proposed a similar framework that uses stencil buffer instead of depth buffer to obtain LDI samples; however, this approach requires an extra sorting step afterwards. Moreover, as all the graphics hardware has a fixed 8-bit buffer for stencil buffer, triangles on an input model to be sampled must be decomposed when the layer complexity of the model is higher than $2^8 = 256$ layers. Although the strategy in [11] can be used to detect the saturation of stencil buffer and then govern the decomposition of input models, such an extension of [1] will further slow down the speed as more data communication between GPU and CPU is needed for the geometry decomposition. These pioneering works that use graphics hardware to accelerate the sampling process involves highly redundant passes of rendering, which lead to the performance bottleneck. Several recent approaches focus on reducing the number of rendering passes, including [2]–[4]. However, they all request an extra sorting step to reorder the resultant fragments. Performing a post-sorting step on GPU will become a bottleneck when sampling models with a large number of layers. Consequently, approaches that work without additional sorting are more attractive.

Eisemann and Decoret in [12], [13] proposed solid voxelization schemes for fast approximating the shape of objects using bitmask. These methods work well for volumetric rendering effects, but voxels cannot provide detailed depth values on samples, thus limiting the accuracy of solids in image space. Recently, Liu et al. [4] succeeded in capturing up to 32 depth values simultaneously with a correct front-to-back order in one single geometry pass. Their method is good for rendering purpose but has a similar problem as depth peeling when rays intersect silhouette edges. Another algorithm was later proposed by them to build a fully programmable pipeline using CUDA instead of the conventional rasterization pipeline – this new pipeline is called FreePipe [5]. In FreePipe, they can handle multiple fragments and therefore sort them in one single pass. However for a scene with a high depth complexity,

TABLE I
COMPARISON OF DIFFERENT LDI SAMPLING APPROACHES

	Stencil buffer [1]	FreePipe [5]	Ours
Sorting Required	Yes	Yes	No
Layer Limitation	Yes	No	No

this sorting step integrated with rasterization still degrades the overall performance of sampling because of its non-linear nature. Ideally, we wish to obtain LDI samples sorted by their depth value in a fast way without any extra sorting step during the sampling process. A method called *conservative sampling* which uses an optimal trade-off between speed and accuracy is proposed in this paper for this purpose. A comparison states the difference between existing methods and ours is shown in Table I.

III. GPU ACCELERATED CONSERVATIVE SAMPLING

The basic idea of our approach is to approximate the volume of an input model via binary encoding and count the occurrences of entering/leaving voxels to index each arrival fragment that is sampled from the input watertight model H. Samples falling in the same voxel are processed by MAX blending algorithm so that the samples are output in a conservative manner – only one or two samples are generated within each voxel.

A. Algorithm Overview

Our LDI sampling algorithm consists of two phases: bitwise mask generation and depth value retrieval.

In the first phase, we compute the solid voxelization of a model H with a resolution of $r_x \times r_y \times r_z$ by graphics hardware accelerated rasterization procedure, where $r_x \times r_y$ is the same as the resolution of the LDI solid to be obtained. r_z is a parameter to be selected by users to control the *Level of Details in Conservation* (LOD-C). The larger r_z is employed, the closer the sampled result is to the exact result. When the value of r_z is small, some short line segments are neglected on the ray (e.g., the gaps between p_2 and p_3 shown in Fig.2). However, the 1D solid determined by the sampling will bound the exact 1D solid of H on the ray (i.e., conservativeness is preserved as illustrated in Fig.2). Two bitwise masks, T_1 and T_2, are obtained in this phase to govern the retrieval of depth values for samples on each ray in the second phase, and both masks are stored in the texture memory as 3D textures (see Fig.3 for an illustration). The first bitwise mask is obtained by the single pass technique of solid voxelization [13]. The method is based on an algorithm that can be implemented as a fragment shader to process the binary information of an array of voxels at (x, y, z) with $z = 0, \cdots, r_z - 1$. The key idea is to let the shader generate a binary number having the value of "1" in all bit-positions lower than z for every fragment, where z is obtained based on the fragment's depth. This binary number is blended into the existing information of the array of voxels in T_1 by the exclusive disjunction operator (i.e., XOR), and the accumulated blending results in a solid voxelization of the rendered model H. However, as mentioned in [13], some very

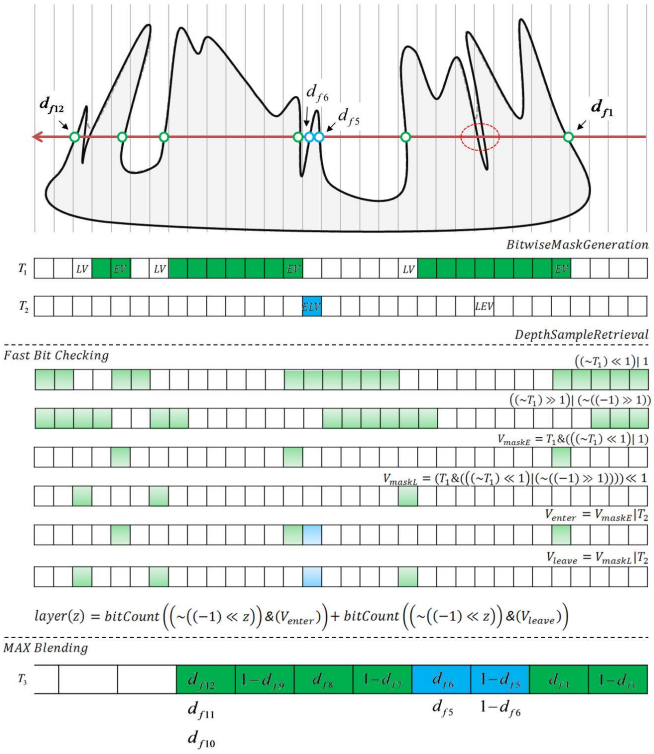


Fig. 3. An algorithm illustration on a 32-bit integer variable. A fragment falling in the voxel (x,y,z) undergoes Fast Bit Checking to get its depth order, regardless of the number of layers in the model.

thin geometries will be missed in T_1 by this method (e.g., the segment between d_{f5} and d_{f6} in Fig.3). Stimulated by the conservative voxelization in [13], we conduct the second pass of rendering to find out the voxels which have fragments falling in but are missed in T_1. These voxels are stored in another texture T_2 (see Fig.3).

The second phase of our algorithm outputs the sorted depth values of the conservative LDI samples by using the *Multi-Render-Target* (MRT). Eight targets where each is attached to a 2D texture having four channels (i.e., RGBA) can output a maximum of 32 layers of samples in each rendering pass. Specifically, starting from the voxels with the smallest index in the z-direction, a fragment shader is employed to output the depth values of the fragments falling in the 32 voxels from $(x, y, 32t)$ to $(x, y, 32t + 31)$ ($t = 0, 1, \cdots$ indicating the t-th pass of rendering). The resultant samples are stored in a 3D texture T_3 with a resolution of $r_x \times r_y \times n_{\max}$ where $n_{\max}$ is the maximal number of layers of the sampled LDI solid. For a fragment that falls in a voxel (x, y, z), we first conduct a novel bit checking technique (see Section III-C) to detect the number of layers of samples in front of it (i.e., in the voxels (x, y, b) with $b < z$). If there are p layers ahead, the depth value of this fragment is then merged into the depth value at (x, y, p) (see the details in Section III-D). Starting from $t = 0$, by rendering the input model H for m times, a total of $32m$ layers of LDI samples, which are sorted by depth values, can be obtained and stored in T_3.

The two phases of our conservative sampling algorithm in

Algorithm 1 BitwiseMaskGeneration

Input: B-rep of the water-tight solid model H
Output: T_1 and T_2

```
glEnable(XOR);
// Rendering H by calling shader programs;
for all fragments f_k do
    z ⇐ GetVoxelPosition(depth(f_k));
    Update all voxels ahead of z in T_1; // Ref. [13]
end for
glDisable(XOR);
glEnable(OR);
// Rendering H by calling shader programs;
for all fragments f_k do
    z ⇐ GetVoxelPosition(depth(f_k));
    if T_1(x,y,z) = T_1(x,y,z-1) = 0 then
        T_2(x,y,z) ⇐ 1;
    end if
end for
glDisable(OR);
```

Algorithm 2 DepthSampleRetrieval

Input: B-rep of the water-tight solid model H, T_1 and T_2
Output: T_3

```
glEnable(BLEND);
glBlendEquation(MAX);
i ⇐ 0;
repeat
    // Rendering H by calling shader programs;
    for all fragments f_k do
        z ⇐ GetVoxelPosition(depth(f_k));
        if z ≥ 32i and z < 32(i+1) then
            p ⇐ CountLayersAhead(T_1, T_2, z);
            if (x,y,z) is EV then
                // Entering voxel
                T_3(x,y,p) ⇐ 1 − depth(f_k);
            end if
            if (x,y,z) is LV then
                // Leaving voxel
                T_3(x,y,p) ⇐ depth(f_k);
            end if
            if (x,y,z) is ELV then
                // Entering-Leaving voxel
                T_3(x,y,p) ⇐ 1 − depth(f_k);
                T_3(x,y,p+1) ⇐ depth(f_k);
            end if
        end if
    end for
    i ⇐ i + 1;
until no fragment is rendered;
glDisable(BLEND);
```

pseudo-code are listed in **Algorithms** *BitwiseMaskGeneration* and *DepthSampleRetrieval*.

B. Voxel Classification

For fragments generated during rasterization, it is essential to identify the type of voxel they fall in and discard fragments that lack contribution to conservative results. Unlike the common solid and empty voxels which are either fully inside or outside the given solid H, we classify the other voxels into four categories. Assume n fragments (f_1,f_2,...,f_n) with $depth(f_i) < depth(f_{i+1})$ are routed into one voxel (x, y, z), the voxel is identified as

- *Entering Voxel* (EV): n is an odd number, and f_1 and f_n are entering the solid H;
- *Leaving Voxel* (LV): n is an odd number, and f_1 and f_n are leaving the solid H;
- *Entering-Leaving Voxel* (ELV): n is an even number, and f_1 is entering the solid H while f_n is leaving H;
- *Leaving-Entering Voxel* (LEV): n is an even number, and f_1 is leaving the solid H while f_n is entering H.

Note that no fragment falls into the solid/empty voxels. The type of voxels is very important for deciding which fragments are generated for the conservative sampling result. According to our definition of conservativeness, the output samples of different types of voxels are:

- EV: f_1 – the fragment with the minimal depth value;
- LV: f_n – the fragment with the maximal depth value;
- ELV: both f_1 and f_n – two fragments with the minimal and the maximal depth values respectively;
- LEV: ideally, the output is expected to be two endpoints at the 1D gap with the maximal length.

This selection is implemented by MAX blending method introduced in Section III-D. However, the blending method can hardly generate the expected result for LEVs. To simplify the computation, LEVs are considered as solid voxels in our approach, which does still satisfy the conservativeness. After the bitwise mask generation step of our algorithm, the type of a voxel (x, y, z) can be easily detected by the information stored in T_1 and T_2 as follows.

- EV: $T_1(x, y, z) = 1$, $T_1(x, y, z - 1) = 0$;
- LV: $T_1(x, y, z) = 0$, $T_1(x, y, z - 1) = 1$;
- ELV: $T_1(x, y, z) = T_1(x, y, z - 1) = 0$, $T_2(x, y, z) = 1$.

Specifically, when $T_1(x, y, z) = T_1(x, y, z - 1) = 0$, we need to further detect whether there is any fragment falls into the voxel (x, y, z). This information is stored in the second bitwise mask in T_2 (see Fig.3 for an illustration). In order to neglect the computation on LEVs, we do not check the voxels with $T_1(x, y, z) = T_1(x, y, z - 1) = 1$.

C. Fast Bit Checking

In order to generate LDI samples sorted by depth value, for a fragment f_k falling in the voxel (x, y, z), we need to find an efficient way to decide where the depth value of f_k must be located (i.e., the value of p in $T_3(x, y, p)$). According to our definition of conservativeness, every entering voxel (or leaving voxel) generates one LDI sample and every entering-leaving voxel (or leaving-entering voxel) has two LDI samples generated. Therefore, for a fragment f_k falling in the voxel (x, y, z), we can determine where to locate its depth value in T_3 by counting the number of EV, LV, ELV and LEV in front of it (i.e., among the voxels (x, y, b) with $b = 0, 1, \cdots, z - 1$).

Intuitively, we can obtain the result by querying the texture column using bitmask – '$1 \ll i$' and the logic 'AND' operator for the i-th bit. However, this method is heavy in computation since it needs to loop through the whole column to find the value of p.

We propose a fast bit checking method to solve the problem for determining the number of layers in front of a voxel (x, y, z) on a ray (x, y). The method consists of two parts: 1) arithmetic shift and logic operations on bitmask, and 2) fast bit counting. In T_1, two consecutive bits '10' and '01' indicate the entering and the leaving voxels respectively. Let T_1 be the integer variable for representing the voxels (x, y, b) in T_1 with $b = 0, 1, \cdots, r_z - 1$. Then, the bitmask

$$V_{maskE} = T_1 \& (((\sim T_1) \ll 1)|1) \tag{1}$$

gives some bitwise information to specify where EVs are located, where the part '$|1$' is for filling the first bit of shifted T_1 (see Fig.3 for an illustration). Similarly, the bitwise information on LVs can be found by

$$V_{maskL} = (T_1 \& (((\sim T_1) \ll 1)|(\sim ((-1) \gg 1)))) \ll 1, \tag{2}$$

with '$|(\sim ((-1) \gg 1))$' filling the last bit of shifted T_1. Together with T_2, the location of all entering samples and leaving samples can be found in

$$V_{enter} = V_{maskE}|T_2, \tag{3}$$

$$V_{leave} = V_{maskL}|T_2. \tag{4}$$

Fig.3 gives an illustration of an example. Therefore, the number of layers, p, in front of the voxel (x, y, z) can be obtained by

$$\begin{aligned} p = {} & bitCount((\sim ((-1) \ll z)) \& V_{enter}) \\ & + bitCount((\sim ((-1) \ll z)) \& V_{leave}). \end{aligned} \tag{5}$$

Note that, the value of the above formula can be evaluated efficiently either by the $bitCount$ function from OpenGL version above 4.0 or by the population count based on a variable-precision SWAR algorithm [14]. The fast bit checking method proposed here can achieve $50 - 100$ times of speedup compared with the exhaustive bitwise checks.

D. MAX Blending for Depth Value Retrieval

As mentioned above, we conduct the blending mechanism provided by graphics hardware to select the LDI samples among the fragments falling in the same voxel. In different types of voxels, different requirements (i.e., fragments with the minimal or maximal depth values) are given for conservative sampling. However, current graphics hardware does not support MAX and MIN blending together. Therefore, we transform the value of depth in T_3 to fit the hardware constraint. In the resultant texture T_3 which stores the LDI samples, for a sample stored in $T_3(x, y, p)$, $T_3(x, y, p)$ gives an LDI sample that starts to enter the solid H when p is odd; $T_3(x, y, p)$ is a leaving sample when p is even. When there are multiple fragments falling in an EV which has p layers in front (p should be even), the fragment with the minimal depth will be selected. There is an even number of layers in front of an LV. Moreover, all the depth values are normalized into the range $[0, 1]$ during rasterization. Based on these observations, we can map the requirement for selecting the minimal depth into the selection of the maximal depth (by MAX blending). Specifically, for the fragment f with a depth value d_f that falls in a voxel with p layers in front, the blending function at the texture T_3 should be

$$T_3(x, y, p) = \begin{cases} \max(1 - d_f) & \text{if } p \text{ is odd} \\ \max(d_f) & \text{if } p \text{ is even} \end{cases} \tag{6}$$

In this way, the depth value of an LDI sample stored in T_3 is $T_3(x, y, z)$ when z is even and the stored depth value becomes $1 - T_3(x, y, z)$ when z is odd.

E. Implementation Details

In order to reduce the number of passes, we need *Multiple Render Target* (MRT) provided by graphics hardware, which allows a shader program to render data to textures directly and these textures can also be used as input to other shader programs. Our implementation is based on OpenGL Shading Language (GLSL). Using MRT of GLSL can output data to multiple textures simultaneously. In the bitwise mask generation phase of our algorithm, we perform voxelization with the volume of $r_x \times r_y \times r_z$ so that the 3D textures T_1 and T_2 have the sizes of $r_x \times r_y \times \text{ceil}(r_z/128)$ with GL_RGBA32UI_EXT as the pixel format. Since every pixel in the 3D texture can have, four 32-bit channels, RGBA, a total of 128-bit can be output per pixel. Modern graphics hardware can afford at most 8 MRTs, thus a maximal of $8 \times 128 = 1024$ bits information can be generated by a fragment shader in one pass of rendering. Therefore, a single pass voxelization can produce voxels with $r_z = 1024$ (i.e., LOD-C). In most cases of conservative sampling, using $r_z = 1024$ generates accurate enough LDI samples.

To be general, formulas of the fast bit checking technique given in Section III-C assume that the whole row of voxels (x, y, b) with $b = 0, 1, \cdots, r_z - 1$ is represented by one single binary number in r_z-bits. However, the implementation in GLSL is based on 8 pixels with RGBA channels (i.e., 8×4 integers in 32-bits). Shifting across these 32 integers need to be carefully processed in the shader program (see the source code provided in [17]).

After the bitwise mask generation phase of our algorithm, we acquire the maximal number of layers, $n_{\max}$, before the depth value retrieval phase so that the resolution of the third 3D texture, T_3, can be determined as $r_x \times r_y \times \text{ceil}(n_{\max}/4)$. Again, the RGBA channels of each pixel in T_3 can store the depth values of 4 LDI samples. To obtain $n_{\max}$, we can add one pass of rendering between the two phases of our algorithm. A fragment shader that counts the layers by the function in Eq.(5) is employed to render the number of layers along each ray into the framebuffer. The maximal number among all pixels in the framebuffer can be obtained by either reading back to CPU or the efficient prefix-sum technique [15]. Our experimental tests by reading back to CPU can also obtain the value of $n_{\max}$ very efficiently on LDI with a resolution of 512×512.

TABLE II
COMPUTATIONAL TIME STATISTICS (IN MILLISECOND).

Models	Fig.	No. of Triangles	Stencil Buffer		Our Approach		FreePipe	
			LDI	Time$^+$	LDI	Time	LDI	Time$^+$
Scaffold Trabecular-bone ($2 \times 2 \times 2$)	1	876K	34-42-42	781 (575)	32-38-42	206	36-42-44	247 (212)
Scaffold Hyperboloid ($1 \times 1 \times 4$)	1	130K	12-12-32	38.9 (11)	12-12-32	20.5	12-12-32	78.6 (71.1)
Lattice of Nylon ($4 \times 2 \times 1$)	1	654K	40-32-16	174 (73)	40-32-14	30.2	40-32-16	138 (125)
Chain	5	402K	6-*Incorrect*-6	236 (6)	6-192-6	35.5	6-266-6	495 (487)
Brush	6	278K	100-42-10	563 (373)	100-40-10	63.7	100-44-10	328 (294)
Geodesic	8	51.2K	24-14-20	72.2 (48)	24-14-18	62.3	24-14-20	81.3 (74.2)

*Models are sampled into LDI with the resolution of 512×512, and LOD-C=1024 is employed in conservative sampling.
+The numbers shown in the brackets indicate the time spent in sorting LDI samples.

TABLE III
TIMING FOR CONSERVATIVE SAMPLING WITH DIFFERENT GRAPHIC CARDS (IN MILLISECOND).

Models	GeForce 9800 GTX			GeForce GTX 295			ATI Radeon HD 5870		
	Sten. Buffer	Ours	FreePipe	Sten. Buffer	Ours	FreePipe	Sten. Buffer	Ours	FreePipe
Scaffold Hyperboloid	70.5 (10)	107	245 (208)	58.8 (11)	46.7	145 (112)	53.6 (16)	26.2	N/A
Lattice of Nylon	526 (58)	502	525 (381)	381 (68)	204	265 (189)	573 (65)	152	N/A
Chain	499 (8)	60.0	1,584 (1527)	582 (7)	58.7	596 (553)	723 (4)	42.7	N/A
Brush	539 (241)	1020	1,010 (861.7)	624 (285)	281	683 (585)	559 (16)	160	N/A
Geodesic	67.8 (35)	135	294 (255)	60.4 (35)	63.7	164 (130)	91 (74)	59.2	N/A

*FreePipe is implemented by CUDA in [5], which cannot run on the ATI graphics cards.
+The numbers shown in the brackets indicate the time spent in sorting LDI samples.

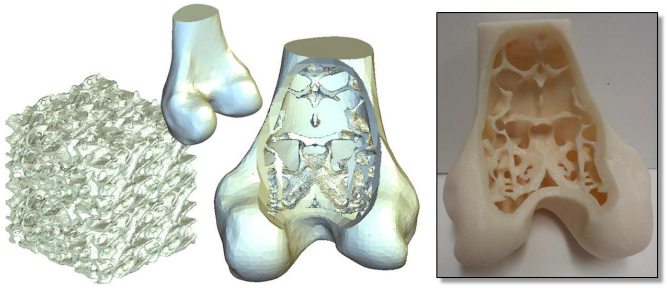


Fig. 4. The scaffold of a trabecular-bone (left) can be inserted into the Bone model (middle) and then fabricated by a rapid prototyping machine (right).

IV. EXPERIMENTAL RESULTS AND DISCUSSION

Our experimental tests were carried out on a computer with Intel Core i5 CPU 750 2.67GHz + 4GB RAM and GeForce GTX580 graphics card. All shader programs are written in GLSL. Source code has been provided in [17].

Firstly, three models are shown in Fig.1 to demonstrate how our work can benefit the modeling of scaffolds with extremely complex geometries, where several unit cells are merged. The scaffold of trabecular-bone can be inserted into the Bone model and then fabricated by a rapid prototyping machine (see Fig.4). Our second example, the Chain model in Fig.5, has more than 260 layers. An incorrect result is generated by stencil buffer based method [1] (see Fig.5(a)) as stencil buffer only has 8 bits and thus can only process models having less than 256 layers when the extension similar to [11] is not used. Another example with many layers is the Brush model shown in Fig.6. Table II lists the computational statistics for sampling the models at the resolution of 512×512 where LOD-C=1024 is employed for conservative sampling. From Table II, we observed that our performance is comparable to Heidelberger et al.'s method [1] and Liu et al.'s method [5] on models with a small number of layers but outperforms models with a large number of layers (e.g. Figs.5 and 6). The main reason is that prior methods need to perform the comparison sort after obtaining depth lists whereas ours don't. Usually, the sorting step takes around 70% of the total time, which becomes the main bottleneck in the process.

In order to study the performance of our method comparing to other existing methods on different graphics hardwares, we further test our benchmarks on three other graphics cards: GeForce GTX 295, GeForce 9800 GTX, and ATI Radeon HD 5870. According to the public available benchmark for graphics hardware (`http://www.videocardbenchmark.net/gpu_list.php`), their performances are scored as GTX 580 – 3934, Radeon HD 5870 – 2743, GTX 295 – 1720, and 9800 GTX – 1134, where the score is the higher the better. The statistics of timing on these graphics cards are listed in Table III. Together with Table II, it is easy to find that the speedup of our method comparing to stencil buffer and FreePipe based methods is more significant on the advanced modern graphics cards (e.g., GTX 580 and Radeon HD 5870). This is because that the speed of visiting texture memory on these cards has been greatly improved, which is one trend of graphics hardware development. Moreover, tests on ATI Radeon card also verify the generality of our method, which can be applied to all graphics hardware supporting OpenGL 2.0 or above. An interesting observation is that the the speed of our method is slower than the stencil buffer based method on GeForce 9800 GTX when testing the 'brush' model. This is also because

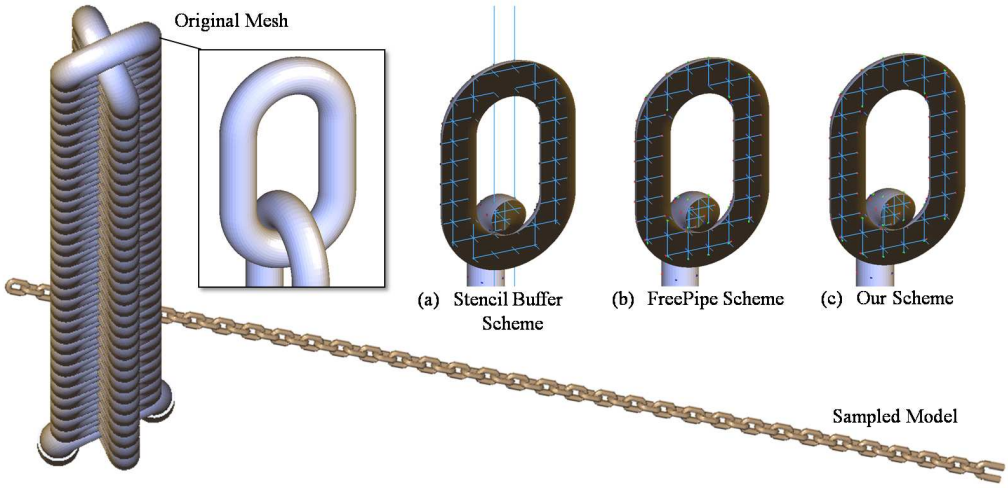


Fig. 5. Sampling on the Chain model by different methods: (a) an incorrect result is generated by stencil buffer scheme [1] – caused by inability to support more than 255 layers in stencil buffer which has 8 bits only, (b) a result generated by the FreePipe scheme [5], and (c) the samples generated by our scheme which has merged some layers that are very close to each other in a conservative manner (i.e., have less layers compared with the exact result).

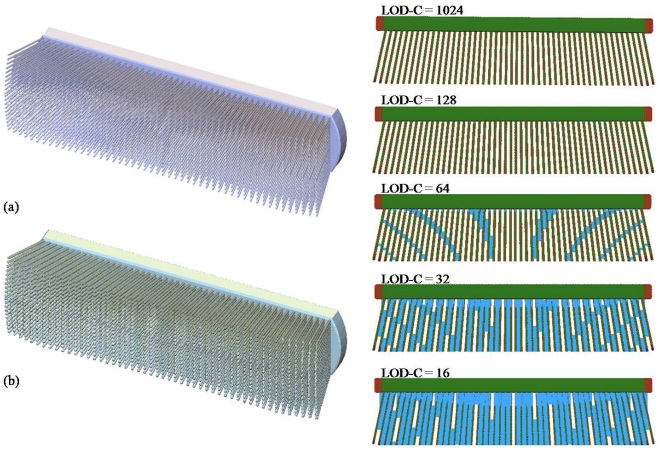


Fig. 6. Sampling the Brush model - with different LOD-Cs. The samplings with LOD-C in 1024, 128, 64, 32 and 16 take 63.7, 34.5, 26.3 and 24.2 (ms) respectively. The blue regions illustrate the bounding volumes given by the conservative LDI samples.

of slow 3D texture memory visit provided on GeForce 9800 GTX.

To further speed up the sampling procedure, we can sample the scene with a smaller LOD-C, especially when we know the samples on rays are distributed sparsely. Fig.7 shows how the value of LOD-C affects the sampling result. It is obvious that a smaller LOD-C gives faster performance and bigger bounding volume. But it doesn't mean that the result must look bad. We compared the appearance of sampling results in two different levels of conservativeness. In Fig.6, the sampling results from LOD-C=1024 and LOD-C=128 have similar appearance and interior volume.

A. Discussion

To capture samples simultaneously, we need to attach different color targets in framebuffer to several textures. Unfortunately, current specification only allows a fixed number of attachments to be bounded, and the number of passes in the second phase depends on running devices – graphics hardware. Specifically, the smaller number of MRTs is allowed on graphics hardware, the more rendering passes are needed. For applications that need other attributes (e.g. normals, colors) to be output together with depth values, putting them together might increase the number of passes needed and then degrade the performance. In addition, undesirable mismatches would occur in MAX blending step. This problem may be alleviated in future by the new feature, SGIX_blend_alpha_minmax, which output all four color components but determines blending results by a comparison of the alpha component only.

V. Applications

We test the LDI solid sampling technique presented in this paper on two applications: intersecting volume evaluation and Minkowski sum.

A. Intersecting Volume Evaluation

Our LDI sampling algorithm enables a fast query of intersecting LDI volume of two overlapped models based on an extension of intersecting volume evaluation by voxels. For two models M_A and M_B, we can use the solid voxelization technique in [13] to compute the solid voxels in one rendering pass for each model, and store the results in two textures T_A and T_B. Another rendering pass with a quadrangle can be used to activate the fragment shader to compute the Boolean intersection result of T_A and T_B. The resultant solid voxels are stored in the third 3D texture, T_{res}. Using T_{res} as the texture T_1 of our *DepthSampleRetrieval* algorithm with triangles of both M_A and M_B passed to the fragment shader will result in LDI samples of the overlapped volume. See Fig.8 for an example of how the intersecting volume in LDI representation is computed for two models in different locations.

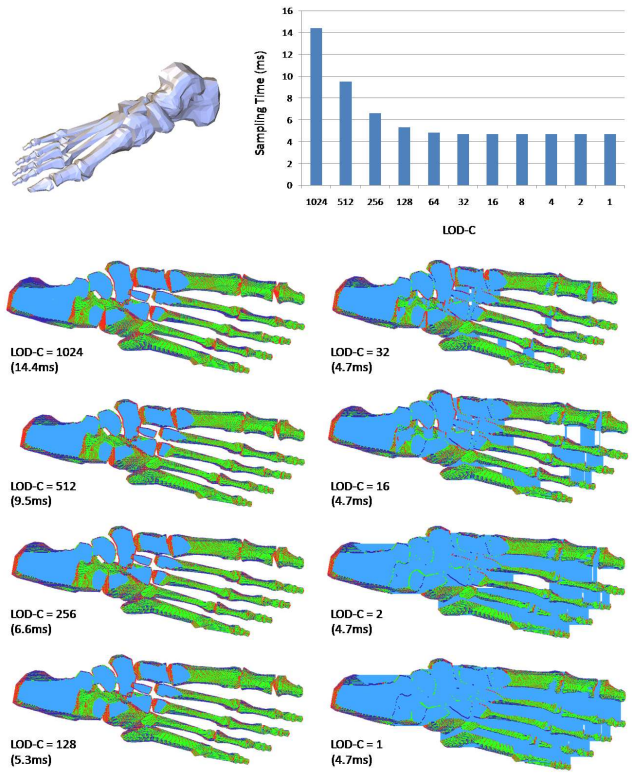


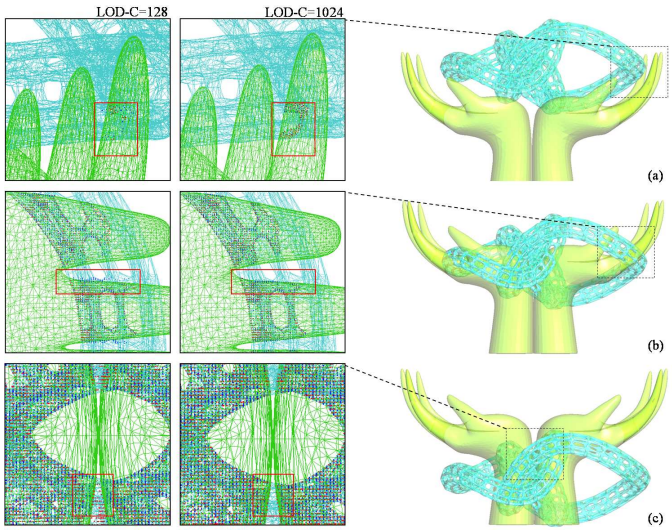
Fig. 7. Comparisons of bounding volume (displayed in blue) and sampling time when different LOD-Cs are used.

B. Minkowski Sum Computation

The same strategy in our scheme can be used to compute the Minkowski sum, $M_A \oplus M_B$, of two polygonal mesh models M_A and M_B. The method presented in [16] computes the solid result of $M_A \oplus M_B$ and stores it in a 3D texture T_M. Their method consists of three steps: 1) polygon culling for generating the superset of polygon soup, 2) surface voxelization to convert the polygon soup into voxels and 3) voxel-based flooding to generate solid results in voxel representation, all of which can run on graphics hardware. Again, the resultant voxel set T_M is employed as T_1 in the second phase of our algorithm *DepthSampleRetrieval*. By adding the passes of rendering both M_A and M_B, we can obtain LDI samples as the result of $M_A \oplus M_B$, which is more accurate than the results in voxels. Fig.9 shows the results of applying our method to further enhance results obtained from [16], and Table IV shows the computational statistics in the resolution of $512 \times 512 \times 512$. Note that only the time of our *DepthSampleRetrieval* algorithm is listed in Table IV as this is the cost of improving the accuracy of Minkowski sum.

VI. Conclusion

We present a novel technique in this paper to efficiently convert voxels obtained from solid voxelization into the *Layered Depth Image* (LDI) representation, which is more accurate. The sampling result is conservative of the input model. Our algorithm can be implemented fully on accelerated graphics hardware that supports the shading language. The parallel nature of our algorithm contributes to the high-speed performance of this approach. Experimental results show that our approach is faster than other sampling methods in generating of LDI solids. Lastly, we demonstrate the versatility of our method by two applications - intersecting volume evaluation and Minkowski sum computation.



Locations	(a)	(b)	(c)
LOD-C: 128	14.4 ms	17.2 ms	28.6 ms
LOD-C: 1024	34.6 ms	40.3 ms	38.9 ms

Fig. 8. Computing the intersecting volume between the Hands model (with 30k triangles) and the Geodesic model (with 51k triangles) in different locations. From the zoomin views of the resultant intersecting volume in LDI samples, it is easy to find that using a higher LOD-C generates a volume that can capture more details. The computational statistics are shown in the table.

TABLE IV
Timing for Computing LDI Samples from Voxels of Minkowski Sum

Model M_A		Model M_B		VBO	Time	No. of
	Trgl.		Trgl.	Trgl.	in Sec.	Samples
Bunny	25.3k	Ball	0.5k	68.9k	0.072s	602.0k
Bull	12.4k	Knot	0.99k	372.5k	0.137s	517.2k
Ear	32.2k	Frame	0.096k	55.8k	0.089s	544.8
Grate2	0.94k	Grate1	0.54k	425.5k	1.4s	1402.3k
Helix	74.0k	Torus	1.6k	474.7k	0.052s	139.6k

Acknowledgment

This research is supported by the HKSAR Research Grants Council (RGC) General Research Fund (GRF) Grants (Ref.: CUHK/417109 and CUHK/417508). Most models tested in this paper are downloaded from Internet, and the helix model was made by the software – JewelCAD. The authors also would like to thank the group of Prof. Pheng-Ann Heng for sharing the ATI Radeon HD 5870 graphics card.

References

[1] B. Heidelberger, M. Teschner, and M.H. Gross, "Real-time volumetric intersections of deforming objects," *Proc. of VMV 2003*, pp.461-468, 2003.

[2] L. Bavoil, S.P. Callahan, A. Lefohn, J.L.D. Comba, and C.T. Silva, "Multi-fragment effects on the GPU using the k-buffer," *Proc. of ACM I3D'07*, pp.97-104, 2007

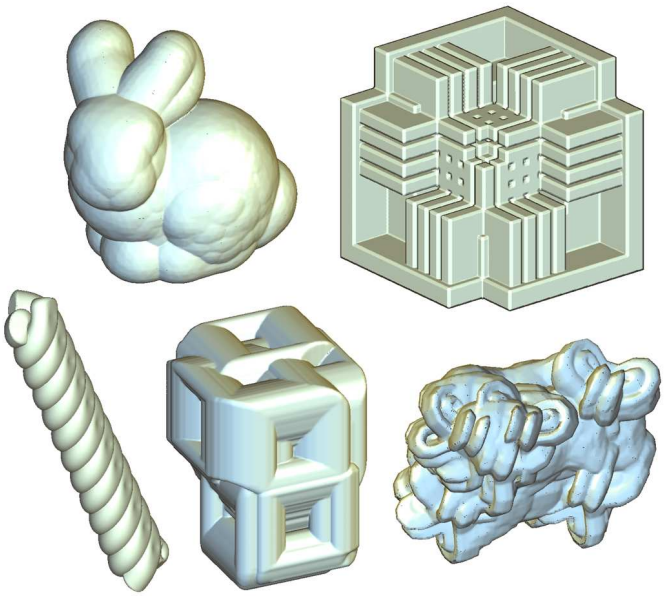


Fig. 9. Examples of using our LDI sampling approach ($512 \times 512 \times 512$) to improve the accuracy of results obtained from voxel-based Minkowski sum computation on GPU.

[3] K. Myers, and L. Bavoil, "Stencil routed A-buffer," *Proc. of ACM SIGGRAPH 2007 Sketches (SIGGRAPH '07)*, Article 21, 2007.

[4] B. Liu, L.-Y. Wei, Y.-Q. Xu, and E.-H. Wu, "Multi-layer depth peeling via fragment sort," *Proc of 11th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pp.452-456, 2009.

[5] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects," *Proc. of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp.75-82, 2010.

[6] J. Shade, S. Gortler, L.-W. He, and R. Szeliski, "Layered depth images," *Proc. of SIGGRAPH '98*, pp.231-242, 1998.

[7] F. Faure, S. Barbier, J. Allard, and F. Falipou, "Image-based collision detection and response between arbitrary volume objects," *Proc. of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp.155-162, 2008.

[8] D. Wexler, L. Gritz, E. Enderton, and J. Rice, "GPU-accelerated high-quality hidden surface removal," *Proc. of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pp.7-14, 2005.

[9] N. Carr, R. Mech, and G. Miller, "Coherent layer peeling for transparent high-depth-complexity scenes," *Proc. of the 23rd ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, pp.33-40, 2008.

[10] C. Everitt, "Interactive order-independent transparency," *Technical report*, NVIDIA Corporation, 2001.

[11] L. Bavoil, and K. Myers, "General purpose z-buffer CSG rendering with consumer level hardware," *Technical Report 2000-003*, VRVis, 2000.

[12] E. Eisemann, and X. Decoret, "Fast scene voxelization and applications," *Proc. of the 2006 ACM Symposium on Interactive 3D Graphics and Games*, pp.71-78, 2006.

[13] E. Eisemann, and X. Decoret, "Single-pass GPU solid voxelization for real-time applications," *Proc. of Graphics Interface 2008*, pp.73-80, 2008.

[14] H.G. Dietz, "The aggregate magic algorithms," *Aggregate.Org Online Technical Report*, University of Kentucky.

[15] S. Sengupta, A. Lefohn, and J.D. Owens, "A work-efficient step-efficient prefix sum Algorithm," *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pp.26-27, 2006.

[16] W. Li, and S. McMains, "A GPU-based voxelization approach to 3D Minkowski sum computation," *ACM Symposium on Solid and Physical Modelling*, pp.31-40, 2010.

[17] Y.-S. Leung, and C.C.L. Wang, Conservative LDI Sampling, `http://www2.mae.cuhk.edu.hk/~cwang/ConLDISampling.html`, The Chinese University of Hong Kong, 2012.

APPENDIX I
POPULATION COUNT (ONES COUNT)

In order to be self-contained, we would like to list the pseudo-code from [14] that efficiently counts the number of one in a binary number x. This is based on 32-bit recursive reduction using SWAR, but the first step is mapping 2-bit values into sum of two 1-bit values in a sneaky way.

1) $x- = ((x \gg 1)\&0x55555555)$;
2) $x = (((x \gg 2)\&0x33333333) + (x\&0x33333333))$;
3) $x = (((x \gg 4) + x)\&0x0f0f0f0f)$;
4) $x+ = (x \gg 8)$;
5) $x+ = (x \gg 16)$;
6) *return* $(x\&0x0000003f)$.