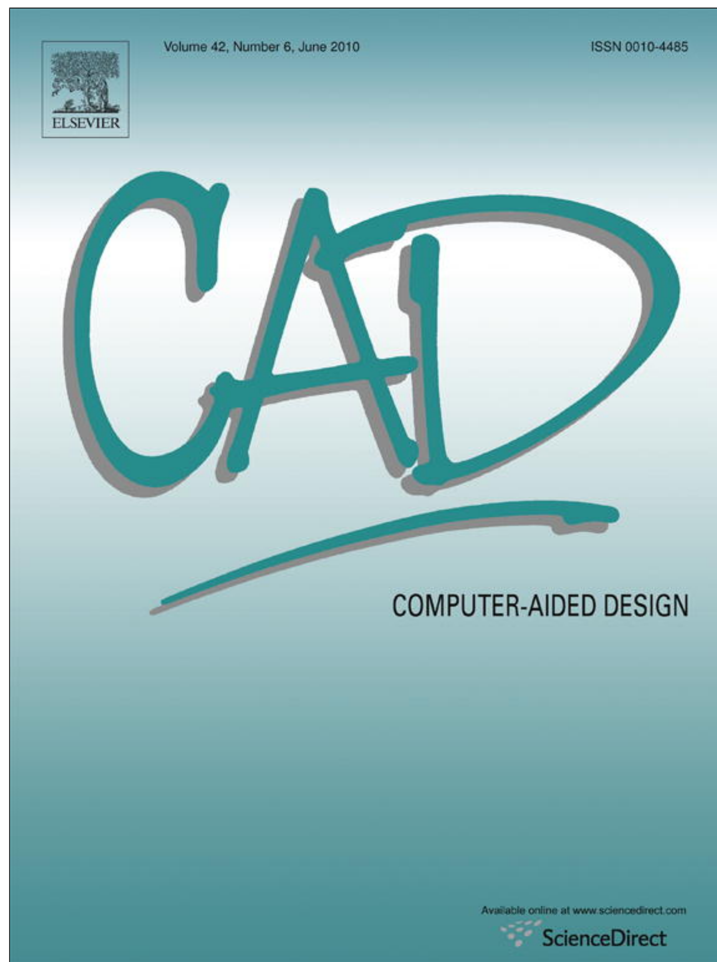
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

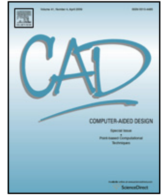
Author's personal copy

Computer-Aided Design 42 (2010) 535–544

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad



Solid modeling of polyhedral objects by Layered Depth-Normal Images on the GPU

Charlie C.L. Wang [a,*], Yuen-Shan Leung [a], Yong Chen [b]

[a] Department of Mechanical and Automation Engineering, The Chinese University of Hong Kong, China
[b] Department of Industrial and Systems Engineering, University of Southern California, USA

ARTICLE INFO

Article history:
Received 2 September 2009
Accepted 7 February 2010

Keywords:
Solid modeler
Complex objects
Layered Depth-Normal Images
GPU

ABSTRACT

We introduce a novel solid modeling framework taking advantage of the architecture of parallel computing on modern graphics hardware. Solid models in this framework are represented by an extension of the ray representation — *Layered Depth-Normal Images* (LDNI), which inherits the good properties of Boolean simplicity, localization and domain decoupling. The defect of ray representation in computational intensity has been overcome by the newly developed parallel algorithms running on the graphics hardware equipped with *Graphics Processing Unit* (GPU). The LDNI for a solid model whose boundary is represented by a closed polygonal mesh can be generated efficiently with the help of hardware accelerated sampling. The parallel algorithm for computing Boolean operations on two LDNI solids runs well on modern graphics hardware. A parallel algorithm is also introduced in this paper to convert LDNI solids to sharp-feature preserved polygonal mesh surfaces, which can be used in downstream applications (e.g., finite element analysis). Different from those GPU-based techniques for rendering CSG-tree of solid models Hable and Rossignac (2007, 2005) [1,2], we compute and store the shape of objects in solid modeling completely on graphics hardware. This greatly eliminates the communication bottleneck between the graphics memory and the main memory.

© 2010 Elsevier Ltd. All rights reserved.

1. Introduction

The purpose of this research is to exploit a solid modeler for freeform objects completely running on GPUs, which will greatly improve the efficiency of shape modeling in various applications (e.g., virtual sculpting, microstructure design, and rapid prototyping). Models in the above applications usually have a very complex shape and topology. Furthermore, the solid modeling operations (such as Boolean operations, offsetting, or hollowing) will be applied to the complex objects for many times. Therefore, an efficient and robust solid modeler is needed.

The market-available solid modelers (e.g., ACIS and Parasolid) use the *boundary representation* (B-rep) to present the shape of an object in computers. Although these approaches based on intersection calculation and direct manipulation of B-rep are accurate, they lack of efficiency and prone to robust problems. The efficiency and robustness problems become more serious when these market-available tools are used to model objects with complex geometry (e.g., the models shown in Fig. 1). Volumetric representation is a good alternative as it can compactly approximate the shape and topology of complex objects by a set of volume data. Modeling based on uniformly sampled volume data is simple but very expensive in time and memory. Adaptive sampling based approaches can reduce the memory cost but in general are not fast enough. At present, the programmable components of the *Graphics Processing Unit* (GPU) allow the use of its high-performance parallel architecture to accelerate many graphics and scientific applications, which originally run on CPU and the main memory. The data communication between the graphics memory and the main memory is still a bottleneck. Thus, a solid modeler completely running on the graphics hardware is a practical way to eliminate such bottleneck at present. Although the rendering of CSG-tree on GPU has been studied in [1,2], the problem becomes much more complicated if the resultant shape of solid modeling operations needs to be retained. For rendering, only the visible elements are kept and displayed by pixels of a single image. To the best of our knowledge, the computation power of modern GPUs on consumer graphics cards however has not been exploited for the solid modeling purpose.

By extending the *ray representation* (ray-rep) in solid modeling (Ref. [4]), we conduct *Layered Depth-Normal Images* (LDNI) in our GPU-based solid modeler, which can achieve a balance between required memory and computing time. In our approach, every solid model is represented by three LDNIs equipped with normals, where each LDNI is perpendicular to one orthogonal axis (i.e., x-, y-, or z-axis). For a given sampling rate w, the required memory

* Corresponding author. Tel.: +852 26098052; fax: +852 26036002.
E-mail address: cwang@mae.cuhk.edu.hk (C.C.L. Wang).

0010-4485/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cad.2010.02.001

Author's personal copy

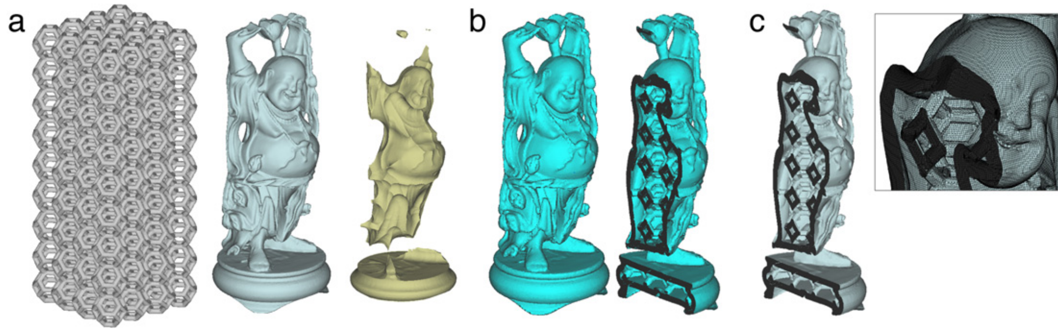


Fig. 1. An example of using Boolean operations to build the interior structure of a hollowed solid model to balance the stiffness and the fabrication time in rapid prototyping [3]. Our GPU-based solid modeler can generate high quality solids with fine details. (a) Input models: left, Truss with 941.9k triangles; middle, Buddha with 497.7k triangles; right, Offset of Buddha with 213.3k triangles. (b) The resultant LDNI solid is obtained by computing: "(Buddha ∩ Truss) ∪ (Buddha \ Offset)" — the model is directly rendered by using sample points. (c) The final mesh surface with 804.6k quadrangles generated from the LDNI solid.

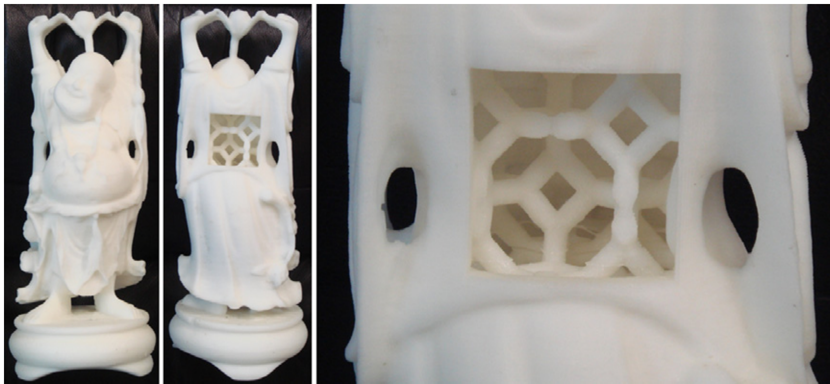


Fig. 2. The Buddha with internal structure is fabricated from the mesh model shown in Fig. 1 by using rapid prototyping.

of LDNI solids is only $O(w^2)$ for most practical models which is similar to the adaptively sampled implicit representations but can be more easily visited in parallel. Besides, the sampling of LDNI from closed 2-manifold polygonal meshes can be efficiently completed by a rasterization technique implemented with the help of graphics hardware. Boolean operations on LDNI solids can be easily implemented running in parallel on GPUs. Solid models in many downstream applications of solid modeling (e.g., rapid prototyping and finite element analysis) still need to have the *boundary representation* (B-rep). A parallel contouring algorithm akin to the dual contouring [5] has been developed for this purpose. Moreover, as the sample points in LDNI solids are coupled with normal vectors, they can be directly rendered as surfels [6] whose size and shape will be changed according to the variation of viewing parameters. Fig. 1 shows an example of modeling complex (both in geometry and topology) models using our GPU-based solid modeling framework, and the real model fabricated from the mesh model by rapid prototyping is shown in Fig. 2.

1.1. Related work

Solid modeling based on B-rep has been investigated for many years. Most of the existing approaches are based on the intersection calculation followed by a direct manipulation of boundary representation. Surveys can be found in [7,8]. The topology correctness of resultant models relies on the robust intersection computation (Ref. [9]). Although a recently published approach [10] proposed a topologically robust algorithm for Boolean operations using approximate arithmetic, the computation still needs to face the topology regularization problem in some extreme cases (such as the examples shown in [11]). Therefore, many approaches start to seek help from volumetric representation. However, most volumetric representations, which are easier to be implemented on GPU than adaptive sampled ones, require the memory in $O(w^3)$ complexity. Here we only need $O(w^2)$ for a LDNI solid — details will be explained in Section 2.

The purpose of the techniques presented in this paper is different from the point-sampled geometry approaches [12–14] which focus on interactive rendering. In these approaches, the shape of a model is described by a set of surface points coupled with normals (i.e., surfel). As mentioned before, the CAD/CAM applications such as CNC and rapid prototyping planning need to have B-rep of solid models. Although we can generate B-rep from the surfels, the structural information of samples that can be used to speed up the solid modeling operations and the contouring process is missed. Moreover, the point-sampled geometry does not give an efficient way to evaluate the *inside/outside* of a point. Similar to point-sampled geometry, the surface information of solids encoded by LDNI is also stored by a set of points coupled with normal vectors. However, the samples in LDNI representation are well organized in a data structure so that the following solid modeling operations and contouring can be implemented easily and completed in an interactive speed. Recently, Nielson et al. in [15] developed an efficient data structure for representing high resolution level sets. However, whether it can be applied to solid modeling is still under research. Moreover, their method cannot preserve sharp feature as no Hermite data is recorded.

Our LDNI solid representation is an extension of ray-rep in the solid modeling literature [4,16,17]. However, different from our LDNI representation, the Ray-rep only stores depth values without surface normals. Furthermore, the algorithm presented in [17] which converts models from ray-rep to B-rep does not take the advantage of structurally stored information, thus it involves a lot of global search and could be very time-consuming. Another line of research related to our work is the so-called *Marching Intersections* (MI) approach [18,19]. The representation of MI is similar to our

Author's personal copy

LDNI representation but MI does not use normal vectors at samples in the surface reconstruction. This leads to the major deficiency of MI. As discussed in [20], aliasing error cannot be eliminated along sharp edges without normal vectors. None of these approaches takes the advantage of high parallel computing capability which is available on consumer-level PCs nowadays. Although recently, the LDNI is adopted to generate fast volumetric tests by Trapp and Döllner in [21], they did not consider the problem of using LDNI to represent the shape in solid modeling.

Stimulated by the original work of Layered Depth Images (LDI) in [22], we recently proposed to use Layered Depth-Normal Images in adaptive sampling, modeling and mesh generation (Refs. [23,24]). Although the primal idea of using LDNI in solid modeling has been addressed in [23,24], the highly parallel algorithms for sampling, Boolean operations and contouring have not been exploited there, which are the unavoidable issues of developing a solid modeler completely running on the GPU for complex objects.

1.2. Main result

The work presented in this paper includes the following main results.

- We introduce a special kind of LDI — *Layered Depth-Normal Images* (LDNI) as an extension of the *ray representation* (ray-rep), which can be efficiently sampled from the B-rep[1] of a solid model and be well mapped to the texture memory on graphics hardware.
- The parallel algorithm for the Boolean operations on two LDNI solids runs well on modern graphics hardware. Although it is not a necessary step after each solid modeling operation, a highly parallel algorithm running on GPU is also developed to generate mesh surfaces from a LDNI solid.
- The above results lead to a novel solid modeling framework, which can handle solid modeling problem more efficiently on modern graphics hardware.

In the rest of this paper, we first brief the LDNI representation in Section 2, and then present the efficient sampling method using graphics hardware rasterization in Section 3. How to map the computation of Boolean operations on the GPU in parallel is addressed in Section 4. The direct rendering method of LDNI solids and the sharp-feature preserved parallel algorithm to generate mesh surfaces from LDNI solids is then developed in Sections 5 and 6. After demonstrating the functionality of our modeling framework in Section 7, our paper ends with the discussion and conclusion sections.

2. Layered depth-normal images for solid models on GPU

In this section, we first explain the principle of *Layered Depth-Normal Images* (LDNI) as an extension of ray-rep briefly, and point out that LDNI is actually a semi-implicit representation. Then, we detail the data structure for storing LDNI in the texture memory of graphics hardware.

2.1. LDNI as a semi-implicit representation

The ray-rep of a solid model H along a specific viewing direction can be considered as a two-dimensional image with $w \times w$ pixels, where on the ray passing the center of each pixel the segments inside the solid H are recorded. As an extension of ray-rep, a LDNI with a specified viewing direction is a two-dimensional image with $w \times w$ pixels, where each pixel contains a sequence of numbers

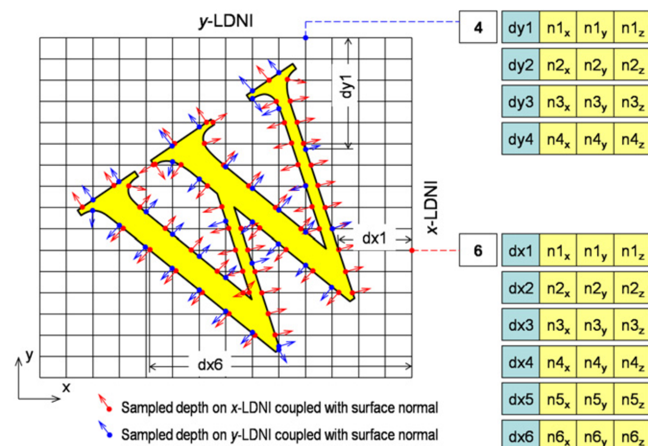


Fig. 3. A two-dimensional illustration of LDNI, where the dot represents the location of sampled depth and the arrow denotes the unit surface normal vector at this point. Red color is employed for the x-LDNI that is perpendicular to x-axis, and blue is for y-LDNI. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

that specify (1) the depths from the intersections (between a ray passing through the center of pixel along the viewing direction and the boundary surface ∂H of H) to the viewing plane, and (2) the normal vector of ∂H at the intersections. Therefore, each sample on rays in a LDNI is a Hermite data. The samples on a ray are sorted in ascending order by their depth values.

We employ a structured set of three LDNIs sampled along x-, y-, and z-axes to present a solid model H in LDNI representation (denoted by $\tilde{H}$). All three images are with the same resolution $w \times w$, and the images are located to let their rays intersect at the $w \times w \times w$ nodes of uniform grids in $\Re^3$. Fig. 3 gives a 2D illustration of the LDNI representation, where the red dots and arrows indicate the Hermite data points recorded on the x-LDNI and the blue ones illustrate the Hermite data points on the y-LDNI. The example information stored in one pixel on the x-LDNI (linked by the red dash line) and one pixel on the y-LDNI (linked by the blue dash line) is also illustrated in Fig. 3, in which the slots with blue background represent the depth values and the yellow slots denote unit normal vectors. We can find that the information stored in LDNI representation is different from other uniformly sampled implicit representation — here only the set of Hermite data points on the surface of a model is sampled and stored. Similar to *Point Set Surfaces* (PPS) in [25], the inside/outside status of a point $\mathbf{p}$ to $\tilde{H}$ can be detected by its closest Hermite sample on $\tilde{H}$ — thus LDNI representation is in fact a semi-implicit representation.

2.2. Data structure on GPU

The data structure of LDNI representation on GPU is discussed below. A solid model represented by LDNI is stored as a list of 2D textures in graphics memory. More specifically, if the maximum number of samples among all pixels on the LDNI along the direction t is n^t_{max}, there are n^t_{max} textures with resolution $w \times w$ for the t-LDNI. Fig. 4 gives an example of the textures of z-LDNI. On the ray passing a pixel p, if there are only m samples (with $m < n^t_{max}$), a special value M (e.g., ∞ or 0) will be filled at the location of this pixel on the ith texture when $i = m + 1, \ldots, n^t_{max}$ — for example, the white pixels in Fig. 4. Therefore, when exploring the samples on the ray of p, we can start searching the pixel value at p from the first texture, then the second, until the special value M is met or we have already searched n^t_{max} textures. Simply, four channels of textures, RGBA, could be used to store the three components of normal vector and the depth value at Hermite samples respectively. Unlike [21], we adopt 2D textures but not 3D textures to store LDNI.

[1] The B-rep of an input solid model must be polygonal surfaces that are closed, manifold (i.e., having a disk-like surface neighborhood around each vertex) and free of self-intersections.

Author's personal copy

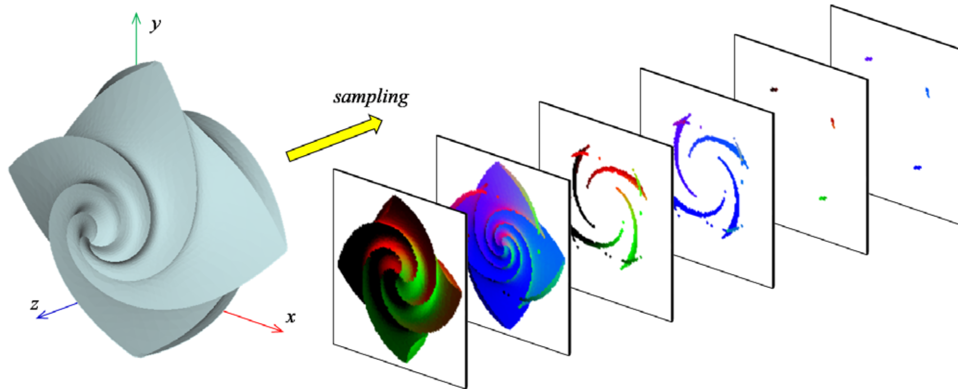


Fig. 4. A solid model represented by LDNI can be stored as a list of 2D textures in graphics memory — an illustration without encoding.

The information stored on a ray of LDNI is with the size $n^t_{\max}$. On most practical models, $n^t_{\max}$ is a constant number that satisfies $n^t_{\max} \ll w$; in the worst case, $n^t_{\max} \rightarrow w$ on all rays. Therefore, the memory complexity of LDNI is $O(w^2)$ on most practical models but become $O(w^3)$ in the worst case.

An encoding method is presented below to reduce the required memory. Simply using four components, RGBA, on a texel to store the depth value and the normal vector at a sample wastes the texture memory. From the study in [26], we found that the shape presented by LDNI is more sensitive to the depth values on samples than the normal vectors. Therefore, using less memory to store the normal vectors is a good way to further reduce the required memory to store LDNI in graphics memory. Moreover, the x-, y- and z-components of a unit normal vector $\hat{\mathbf{n}}$ are normalized by $\hat{\mathbf{n}}_x^2 + \hat{\mathbf{n}}_y^2 + \hat{\mathbf{n}}_z^2 \equiv 1$. We adopt the following method to make the samples of LDNI stored more compactly. A LDNI solid with $n_{\max}$ layers are stored by $\frac{1}{2}n_{\max}$ textures. On modern graphics hardware, the pixels at a texture usually have four color channels: RGBA, and each color channel has 32-bits. For the samples on the $2i$th layer and the $(2i+1)$th layer of LDNI, we store their depth values in the G and A components respectively with 32-bits. By selecting an appropriate origin, we can easily make the depth values of all samples be positive. Then, the $\hat{\mathbf{n}}_x$ and $\hat{\mathbf{n}}_y$ components of the unit normal vector at a sample are truncated into 16-bits and stored in the R channel for the sample on the $2i$th layer, and the B channel for the sample on the $(2i+1)$th layer. Lastly, the signs of $\hat{\mathbf{n}}_z$ are encoded onto the depth values in G or A components. By this way, we can reduce half of the required memory. Such encoding can be easily implemented during the sampling by shader programs.

3. Sampling B-rep into LDNI

Using graphics accelerated hardware to construct LDNI representation from the closed 2-manifold boundary of a solid model H is similar to the scan-conversion algorithm that the scan line along view direction alternatively passes H between the interior and exterior. Two strategies can be employed to convert a mesh surface ∂H into samples intersected by the rays and ∂H: (1) the widely used depth-peeling [27] and (2) ours using stencil buffer akin to [28].

Why not depth-peeling? For a correctly sampled solid model $\tilde{H}$ represented by LDNI, the number of samples on the ray passing a pixel must be even so that the samples clearly state the *inside/outside* regions for the sampled solid along the ray. However, the depth-peeling cannot ensure this as it is based on the depth values of intersections. The edges on an input mesh model are called silhouette-edges if only one of its two adjacent polygons faces towards the viewpoint. During sampling, it is possible to have a ray passing through silhouette-edges. The depth-peeling algorithm (e.g., [27]) will report only one intersection point with the silhouette. However, for a valid solid representation along the ray, two (or zero) samples must be reported at this intersection. This misreporting will lead to the incorrect specification of *inside/outside* along the ray.

In order to avoid the misreporting cases in depth-peeling, we consider using stencil buffer to count the number of intersections along the rays, which can guarantee a valid number of samples being reported although the reported samples are not sorted by their depth values. When a ray that passes through the center of a pixel intersects a silhouette edge, the stencil buffer will report two intersections as long as both the faces adjacent to the silhouette edge cover the center of pixel. If they do not cover the center (even if this is generated by numerical error), no intersection will be reported which is also acceptable to LDNI sampling. More details of stencil buffer at silhouette-edges can be found in [29]. Similar to [28], the boundary surface mesh of H has to be rendered several times for the sampling of LDNIs. The viewing parameters are determined by the working envelope, which is slightly larger than the bounding box of the model. Orthogonal projection is adopted for rendering so that the intersection points from parallel rays can be generated. The number of times that the rendering will repeat is determined by the depth complexity $n^t_{\max}$ of the model H along the given direction t. The value of depth complexity n_p at every pixel p can be read from the stencil buffer after the first rendering, in which the stencil test configuration allows only the first fragment to pass per pixel but still increase the stencil buffer in the later fragment pass. After that, $n_{\max} = \max(n_p)$ can be determined by scanning n_p on all pixels in parallel using [30]. If $n_{\max} > 1$, additional rendering passes with $k = 2$ to $n_{\max}$ will generate the remaining layers and the stencil test configuration allows only the kth fragment to pass. For the pixels with $n_p < n_{\max}$, layers from $n_p + 1$ to $n_{\max}$ do not contain valid depth values and are assigned with special mark M. Not only depth values but also the normal vectors at the intersection points must be sampled and stored (details will be given below). The above algorithm generates an unsorted LDNI as fragments are in general rendered in arbitrary. Therefore, a post-step is needed to sort the samples at each pixel by their depth values.

Speed up by shader program. The bottleneck of the above sampling method is the communication between the main memory and the texture memory of graphics hardware. In order to avoid repeatedly sending the geometry and connectivity data from the main memory to the graphics hardware during the sampling, we first compile a graphics object list onto the graphics hardware so that we can call the object list directly from the graphics hardware to repeatedly render the models. By this standard rendering procedure, we can only send the model to be sampled through the data communication bottleneck once. In order to further reduce

Author's personal copy

Table 1
Logic operator *OPRT* (A, B).

Status of A and B		Type of Operations		
inside A	*inside B*	'∪'	'∩'	'\'
True	False	True	False	True
False	True	True	False	False
True	True	True	True	False
False	False	False	False	False

the amount of data sent through this communication bottleneck, instead of sending $9m$ vertex coordinates and $3m$ face normals for a mesh surface with m triangles, we pass a vertex table with $3n$ float coordinates (for n vertices) and a triangular face table with $3m$ integer indices for vertices of triangles to the graphics hardware. In general, $n \approx \frac{m}{2}$ — i.e., we reduce the amount of communication from $12m \times 4 = 48m$ bytes to $4.5m \times 4 = 18m$ bytes. A geometry shader program is also developed to assemble the topology information into triangles to be rendered by the streamline. A fragment shader program is adopted to encode the sampled depth values and normal vectors into the two channels of frame-buffer. Note that the encoding mentioned at the end of last section is implemented here.

As we use CUDA kernel programs [31] in the next section to compute solid modeling operations in parallel, after sampling two layer of LDNI onto the frame-buffer, we can instantly map the pixel values at frame-buffer to the texture memory, which can be accessed by CUDA kernel program.[2] Before computing a Boolean operation, the unsorted samples will be sorted through a CUDA kernel program.

4. Boolean operations on LDNI

As the fundamental operation of solid modeling, Boolean operations are widely used in various CAD/CAM applications. A solid model in LDNI representation is actually a set of well-organized 1D volumes that inherits the good property of Boolean simplicity from ray-rep. When computing the Boolean operation of two LDNI solid models $\tilde{H}_A$ and $\tilde{H}_B$, the Boolean operations can simply be conducted by the depth-normal samples on each ray as long as the rays of $\tilde{H}_A$ and $\tilde{H}_B$ are overlapped. This request can be easily satisfied during sampling. More specifically, when sampling H_B into a LDNI solid $\tilde{H}_B$, we carefully choose the origin of sampling envelope to ensure that the rays of $\tilde{H}_B$ overlap the rays of $\tilde{H}_A$. More specifically, the two input models are identically oriented and with the same sampling frequency.

The algorithm of Boolean operations on rays is briefed below. On two overlapped rays $R_A \in \tilde{H}_A$ and $R_B \in \tilde{H}_B$, if either R_A or R_B is empty (i.e., with $n_A = 0$ or $n_B = 0$), the processing will be very simple (see line 21-26 in *Algorithm BooleanOperationOnRay*). For the case neither R_A nor R_B is empty, we can perform a 1D Boolean operation easily by moving on the samples of R_A and R_B according to their depth values. During the movement, the resultant samples are generated and stored on a new ray R_{res}, where the resultant samples are those leading to a change of *inside/outside* status based on the analysis of current position of the sample and the type of Boolean operation performed with the help of a logic operator, *OPRT* (A, B). The operator *OPRT* (A, B) is as defined in Table 1. In short, we can check whether a depth value d is *inside* or *outside* a 1D volume by detecting whether there are odd (or even) number of samples whose depth values are less than d. Then, the *inside* or *outside* status on a resultant 1D volume is determined by *OPRT* (A, B). After scanning all samples on R_A and R_B, we can output the resultant samples into a list of sample, R_{res}. Pseudo-code is listed in **Algorithm** *BooleanOperationOnRay* (in Appendix), where n_A and n_B are the numbers of samples on R_A and R_B, $R_A[i]$ and $R_B[i]$ denote the ith sample on the ray, and $R_A[i].d$ and $R_B[i].d$ represent the depths at samples. Lines 1-20 show the method of scanning all samples of R_A and R_B, which is similar to the Boolean operations on Ray-rep except the manipulation on normal vectors.

Robustness enhancement. A step of *small interval removal* is given to enhance the robustness of Boolean operation computation. The 1D volumes whose thickness are smaller than ϵ will be removed from the 1D volume of the resultant LDNI samples. $\epsilon = 10^{-5}$ is chosen in our implementation. This is because the depth values are encoded in single precision float on graphics hardware, and $\epsilon = 10^{-5}$ is slightly larger than the smallest number that can be presented by single precision float (i.e., with 32-bits). We incorporate this robustness enhancement into **Algorithm** *BooleanOperationOnRay* (R_A, R_B) by changing its Step 17. Instead of simply insert s at the tail of R_{res}, we compare its depth value with the depth of the last sample in R_{res}. If the difference on their depths is smaller than ϵ, we remove the last sample from R_{res}; otherwise, we add s at the tail of R_{res}. By this, the result of Boolean operations on tangential contact models can be easily corrected; however, it is very tough to directly compute Boolean operations on B-rep models.

Implementation on GPU. The above algorithm for Boolean operations on two LDNI solids $\tilde{H}_A$ and $\tilde{H}_B$ maps to the parallel computation on the GPU very well. We just need to write a kernel program by CUDA [31] for **Algorithm** *BooleanOperationOnRay* (R_A, R_B) and run it for all rays on $\tilde{H}_A$ and $\tilde{H}_B$ in parallel. The resultant samples will be kept in the graphics memory for further use, so that the time for data communication between graphics hardware and main memory can be saved.

Furthermore, we can classify the rays from $\tilde{H}_A$ and $\tilde{H}_B$ into four groups:

- Group I: Neither R_A nor R_B are empty;
- Group II: R_A is not empty but R_B is empty;
- Group III: R_A is empty but R_B is not empty;
- Group IV: Both R_A and R_B are empty.

Our tests on many examples show that there are in general more than 60% rays belonging to Group IV. Therefore, a lot of time will be wasted if we simply run **Algorithm** *BooleanOperationOnRay* in parallel on all rays. To avoid this, we employ the all-prefix-sum scan technique in [30] to remove rays in Group IV according to above criteria, and then call **Algorithm** *BooleanOperationOnRay* (R_A, R_B) in parallel on the rest rays in Group I–III.

5. GPU-based direct rendering of LDNIs

Many solid modeling applications like virtual sculpting and microstructure design request the function for displaying the solid model during the procedure of modeling. Different from B-rep, there is no straightforward method for displaying the surface of a solid model in LDNI representation. Although we can convert it into a B-rep mesh to display (by the algorithm presented in the following section), the communication between the graphics hardware and the main memory will be a bottleneck to slow down the update rate — such refreshment can hardly be in a real-time (>25 fps) or interactive rate (∼10 fps). Based on this reason, we develop a method for directly rendering the samples of LDNI on GPU. As the sample points in LDNIs are coupled with normal vectors, they are rendered as surfels [6] whose size and shape are changed according to the variation of viewing parameters.

The most important issue is to reduce the amount of communication between the graphics hardware and the host. First of all,

[2] Such an instant mapping is supported by DirectX using *cudaD3D10RegisterResource* and *cudaD3D10ResourceGetMappedPointer*.

Author's personal copy

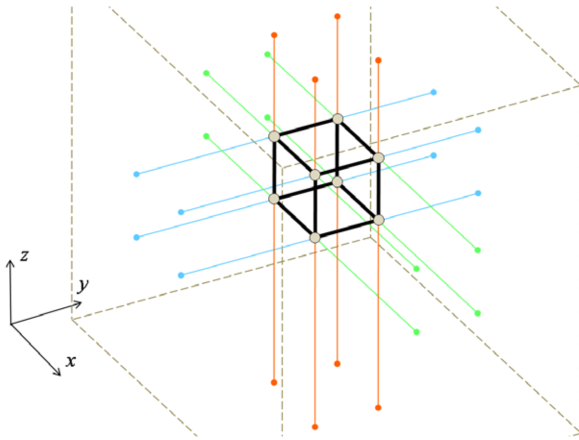


Fig. 5. The *inside/outside* status of eight nodes on a cell can be detected on-site locally by exploring 12 related rays (rays in different directions are displayed in different colors). The black segments are the cell-edges overlapped with rays. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

we employ the all-prefix-sum scan technique in [30] to obtain the total number, n_{pnts}, of samples in a LDNI solid. To activate the engine of displaying while minimizing the amount of data communication, we adopt geometry shader to create the surfels for rendering. If each geometry shader can generate $m_{\max}$ primitives, we send $\ulcorner n_{pnts} \setminus m_{\max} \urcorner$ point primitives from the host. Then, each shader will splat one point into $m_{\max}$ points with the positions and normals are acquired from the LDNI samples already stored in the graphics memory. By this, we can achieve real-time rendering of more than ten million surfels, which is fast enough for directly rendering the samples of LDNIs (for example, the direct rendering results shown in Fig. 1).

6. Contouring LDNI solid to B-rep

This section will present the contouring algorithm which converts a LDNI solid model $\tilde{H}$ into a closed polygonal mesh S (i.e., B-rep). We develop a parallel dual contouring algorithm running on GPU. The most difficult part is how to generate the mesh surface in a streaming mode without requesting additional memory as it is very limited on the graphics hardware. Our algorithm outputs the resultant mesh model by two passes, where the vertex table (geometry) of S is generated in the first pass and the second pass constructs the face table (topology) which specifies the polygons linking the vertices.

The first issue if how to detect *inside/outside* consistently on a node intersected by three rays. As the images of a LDNI representation are located so that the intersections of their rays intersect at the $w \times w \times w$ nodes of uniform grids in $\Re^3$, our algorithm will employ this grid structure to construct vertex and face tables. However, instead of constructing such grids explicitly, we check the *inside/outside* configuration of each grid node on-site by using the samples on the three rays intersecting at this node. Therefore, we avoid spending additional $O(w^3)$ memory on contouring. Although the numerical error could lead to inconsistent classification of *inside/outside* from three rays at very few nodes, it can be easily solved by a majority vote — a node is classified to be *inside* the model if it shows *inside* on at least two rays passing this node.

The method to locate vertices on the resultant mesh surface is introduced here. For a solid represented by LDNIs with resolution $w \times w$, the rays of LDNIs actually establish $(w-1) \times (w-1) \times (w-1)$ cubic cells with cell-edges overlapping with part of the rays (see Fig. 5), and each cell is labeled by its location index $[i, j, k]_{(i,j,k=0,\ldots,w-2)}$. A cell is at the boundary of the LDNI solid model $\tilde{H}$ if any of its grid nodes is *inside* while there is at least one node *outside*. Such a cell is called *boundary cell*. Vertices of the final polygonal mesh S are only constructed in the boundary cells. Every boundary cell contains one vertex which has an unique ID, $(i(w-2)^2 + j(w-2) + k)$, if the cell's index is $[i, j, k]$. The position of this vertex is determined by the position that minimizes the *Quadratic Error Function* (QEF) defined by the Hermite samples falling on the cell-edges of the cell $[i, j, k]$. Using the position minimizing QEF helps reconstructing sharp features on the resultant mesh surface S. This is also the reason why we choose dual contouring but not the Marching Cubes [32] as the strategy of our parallel contouring algorithm. The first pass of our algorithm will check the boundary cells in parallel and output the vertices, both the positions and IDs, in the boundary cells. This is easily implemented in a streaming way. To further speed up, the all-prefix-sum scan technique in [30] is employed to prevent spending unnecessary time on the non-boundary-cells.

The second pass of our algorithm checks the edges of cells in parallel. If there is an inside/outside sign change on the two grid nodes of a cell-edge, one quadrilateral face is constructed by linking the vertices in its four neighboring cells. Note that only the IDs of vertices for quadrilateral faces are output as records in the face table. All faces should be constructed in such an orientation that its normal faces outwards. The second pass is also easy to be implemented to run in parallel and streaming mode. Again, the all-prefix-sum scan technique in [30] is utilized to remove those rays not adjacent to any boundary cells from computation.

7. Experimental results

We have implemented the above algorithm and tested various examples with massive number of triangles on a consumer-level PC with Intel Core 2 Quad CPU Q6600 2.4 GHz + 4 GB RAM and GeForce GTX295 graphics card. We first study the performance of our parallel sampling algorithm and the graphics memory occupied by the resultant LDNI solid. Statistics are shown in Table 2. All examples given in this paper are tested at the resolution of 512 × 512. From Table 2, we can conclude that our parallel sampling algorithm is able to generate the LDNI representation from closed polygonal mesh surfaces efficiently.

Our first example of solid modeling shown in Fig. 1 is to construct the interior truss structure of a hollowed solid model, which is a very important modeling step for the rapid prototyping in CAD/CAM [3]. Five more examples are shown in Figs. 6 and 7, where the sharp features are well preserved in our modeling framework as our LDNI representation keeps the Hermite data on each sample. Lastly, the example of cube and sphere is used to test the robustness of our solid modeler when handling degenerated cases (i.e., tangential contact cases) in Fig. 8. After testing variety of freeform models, it is not difficult to conclude that our algorithm is very robust and efficient.

In order to compare the proposed algorithm with the state-of-the-art, we also implemented two other programs for Boolean operations. One calls the API functions provided by the commercial software package ACIS R15 [34], and the other employs the Boolean operation functions on 3D selective Nef Complexes in the newest version of CGAL library [33]. The comparisons of computing time are listed in Table 3. It is not difficult to find that our GPU-based solid modeler works well on the models with massive number of triangles, which cannot be computed by ACIS and CGAL. It is surprising that although ACIS can work out the Boolean operation on tangential contacted models by using exact arithmetic, it takes very long time to generate the results. CGAL performs better on this aspect. More than that, ACIS takes a long time to report failure on the complex models (e.g., "Dragon \ Bunny" takes 1.14 h).

To further demonstrate the speed improvement on the GPU-based parallel algorithms comparing to the CPU-based algorithm in [23,24], we list the statistics for the same sets of examples in

Author's personal copy

Table 2
Statistics of LDNI sampling and memory usage at resolution 512 × 512.

Model	Faces num	Vertices num	Sampling time (s)	LDNI layers	Memory (MB)
Buddha	498k	249k	0.400	14-20-10	88
Truss	942k	467k	0.773	34-88-36	316
Bunny	70k	35k	0.127	20-10-12	84
Dragon	277k	138k	0.253	14-12-12	76
Filigree	260k	130k	0.244	28-30-10	136
Lion	400k	200k	0.322	24-18-12	108
Helix	74k	37k	0.130	6-6-32	88

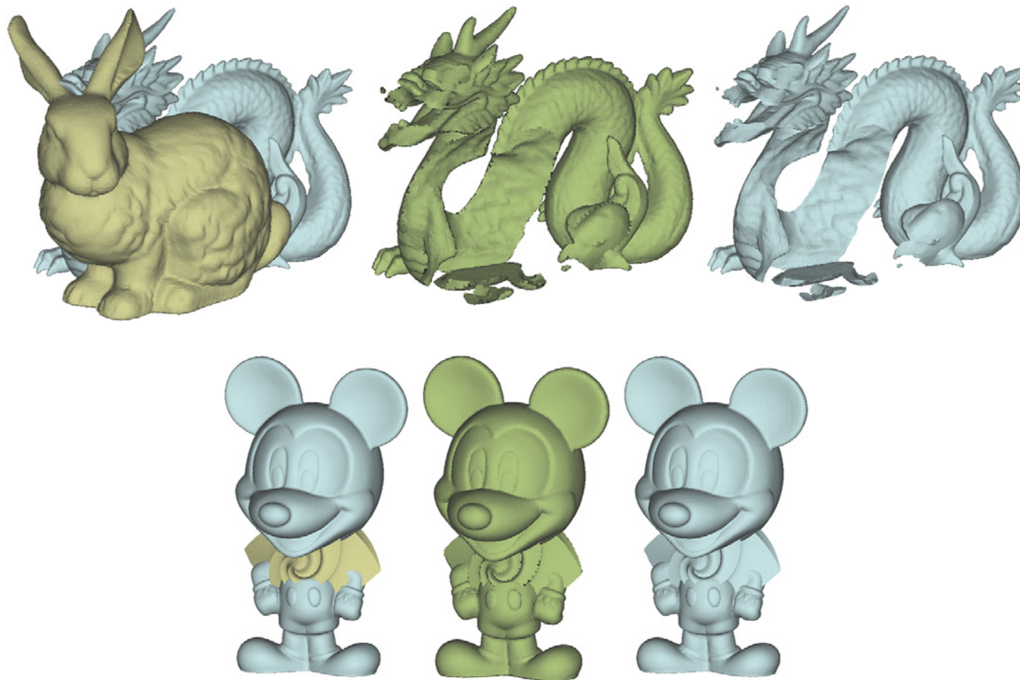


Fig. 6. Two examples: "Dragon \ Bunny" (top) and "Mickey ∪ Octa-Flower" (bottom). The results preserve details very well — see the geometry details from Bunny on the resultant dragon, and the sharp features on the resultant Mickey.

Table 3
Computational statistics in time (second).

Example	Figure	Face num		ACIS	CGAL	Our GPU-based solid modeler[a]		
		First	Second			Sampling	Boolean	Contouring
RP Example[b]	1	942k 213k[c]	498k	Failed	Failed	1.466	0.354	1.551
Dragon \ Bunny	6	277k	69.7k	Failed	Failed	0.361	0.113	0.757
Mickey ∪ Octa-Flower	6	80.1k	15.8k	135.22	Failed	0.197	0.078	0.656
Buddha ∪ Filigree	7	498k	260k	Failed	Failed	0.622	0.111	0.703
Bunny \ (Helix ∪ Box)	7	69.7k	74.0k	106.08	Failed	0.351	0.164	0.720
Lion \ (Lion-off ∪ Box)	7	400k	99.2k	7120.98	Failed	0.527	0.209	1.537
Box ∪ Sphere \ Box	8	4.80k	760	43.388	0.864	0.195	0.147	0.314
Chair \ Octa-Flower	9	464	15.8k	1.72	7.82	0.173	0.084	0.687
Ring-A ∪ Ring-B	9	12.3k	14.3k	Failed	34.9	0.185	0.078	0.968

[a] The solids are computed at the resolution of 512 × 512.
[b] We actually compute "Buddha \ (Offset \ Truss)" in the RP example to speed up the computation.
[c] For the offset model of Buddha model.

Table 4
Computing time (Second) of CPU-based LDNI approach [23,24].

Example	Sampling	On Single-Core CPU		On Quad-Core CPU	
		Boolean	Contouring	Boolean	Contouring
RP Example	8.044 (6.578)	1.092	13.244	1.092	7.972
Dragon \ Bunny	1.710 (1.349)	0.327	6.585	0.328	3.902
Mickey ∪ Octa-Flower	0.934 (0.737)	0.250	4.040	0.266	2.356
Buddha ∪ Filigree	2.534 (1.912)	0.094	5.366	0.234	3.245
Bunny \ (Helix ∪ Box)	1.767 (1.416)	0.516	9.142	0.905	5.288
Lion \ (Lion-off ∪ Box)	2.777 (2.250)	0.874	13.135	0.905	7.644

* The values in brackets are the time spending on reading back from the graphics memory to the host.

Author's personal copy

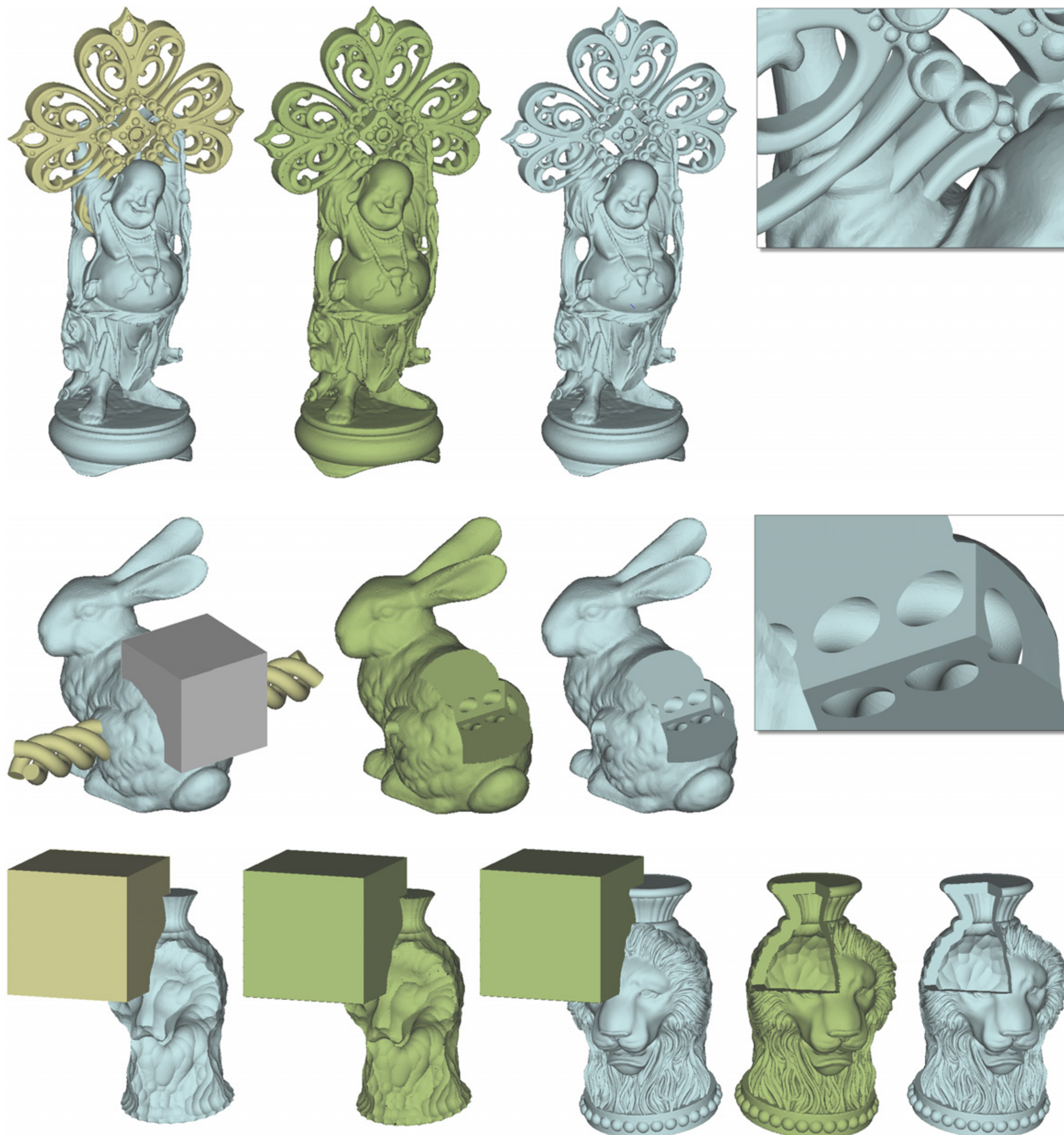


Fig. 7. Three more examples: "Buddha ∪ Filigree", "Bunny \ (Helix ∪ Box)", and "Lion \ (Lion-off ∪ Box)".

Table 4. It is not hard to find that GPU-based parallel algorithms proposed in this paper can achieve more than 5 times speed up — even after using multiple-core parallelization and adaptive algorithms in [23,24]. When comparing to the results evaluated on PC with single-core CPU, the speed up is even higher. The speed up is gained from (1) the reduction of communication between the graphics hardware and the host, and (2) the highly parallel computational power provided by the graphics hardware. If we consider about the Boolean operation step itself, the speed up ratio seems is not that high. However, we find that in the GPU-based Boolean operation step, the memory allocation to generate the texture space to store the resultant samples takes from 18.6% to 58.6% of the total Boolean time. The time needed for Boolean step can be further reduced if we pre-allocate the texture memory — of course, the program implemented in this way will not be efficient in memory consumption. Nevertheless, this will benefit the applications relying on repeated Boolean operations (e.g., virtual sculpting, CNC simulation, microstructure design, etc.). A video to demonstrate the function of our implementation can also be downloaded from the link: http://www.mae.cuhk.edu.hk/~yleung/cadgpumodeler.wmv.

8. Discussions

The major limitation on our current implementation is the memory occupied by the sampled LDNI solids. When increasing the resolution from 512 × 512 to 1024 × 1024, the graphics memory on our graphics card has been used up in the RP example as the displaying of OS GUI and the 3D models need some graphics memory as well. We are developing a new data structure like [15], which is stimulated by the data structure for storing sparse matrix, to store the sampled LDNI more compactly. This is considered as our work in the near future. Currently, the high resolution computation can also be achieved in our framework by volume tiling. However, the extra cost we should pay is the time of data communication between the graphics memory and the host (i.e., the differences of times in Tables 3 and 4).

Another limitation of solid in LDNI is an old problem of ray-rep — it is rotation sensitive. Specifically, when we have a solid in ray-rep, the current ray-rep is not correct any more after a simple rotation. Therefore, a fast re-sampling method needs to be developed to solve this problem.

Author's personal copy

Table 5
Shape error reported by the metro tool [35].

Example	Res.: 128 × 128		Res.: 256 × 256		Res.: 512 × 512	
	E_{mean} (%)	$E_{\max}$ (%)	E_{mean} (%)	$E_{\max}$ (%)	E_{mean} (%)	$E_{\max}$ (%)
Mickey ∪ Octa-Flower	7.97×10^{-3}	0.333	2.03×10^{-3}	0.291	6.56×10^{-4}	0.277
Bunny \ (Helix ∪ Box)	6.89×10^{-3}	0.396	4.04×10^{-3}	0.301	1.32×10^{-3}	0.293
Lion \ (Lion-off ∪ Box)	1.69×10^{-2}	1.27	4.48×10^{-3}	0.988	2.10×10^{-3}	0.262
Box ∪ Sphere \ Box	1.53×10^{-3}	0.0426	5.76×10^{-4}	0.0202	1.53×10^{-4}	0.0106
Chair \ Octa-Flower	1.54×10^{-3}	0.808	4.37×10^{-4}	0.117	9.02×10^{-4}	0.0503
Ring-A ∪ Ring-B	3.45×10^{-3}	0.108	1.09×10^{-3}	0.0681	3.03×10^{-4}	0.0292

* The errors are reported in percentage with reference to the diagonal lengths of the models' bounding boxes.

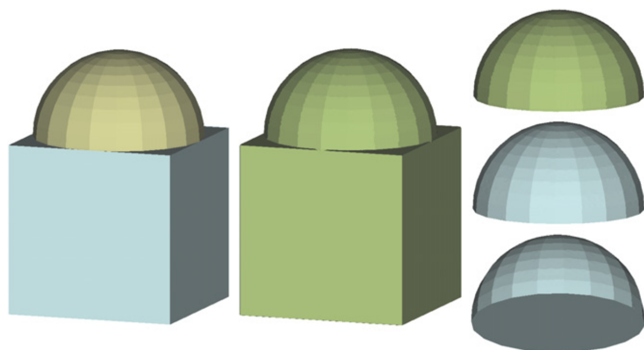


Fig. 8. A tangential contact example to test the robustness of our approach: "Box ∪ Sphere \ Box".

Our current implementation lacks of other more complicated solid modeling operations, e.g., general offsetting and Minkowski sum. Special parallel algorithms need to be developed to further enhance the current solid modeling framework on LDNI.

We present the highly parallel algorithm in this paper which can fully run on the GPU. The most critical issues solved here are: (1) how to reduce the communication between graphics memory and host, and (2) how to enhance the degree of parallelism. Differently, the approach proposed in [23,24] provides methods on the adaptive sampling, modeling and mesh generation so that a better accuracy control can be achieved with limited memory. These two approaches focus on different aspects of the solid modeling on complex objects. We leave the problem about whether to choose high speed or adaptive accuracy to the users. Moreover, the conversion from LDNI to B-rep could be an issue to generate water-tight model when the modeled object has very small features (e.g., the feature size less than $\sqrt{3}$ times of the diagonal length of cubic cells in contouring).

In this approach, the solid modeling operations are computed on the sampled LDNI representation, which will generate shape approximation error on the results. The surface errors can be measured using the publicly available Metro tool [35] by comparing with the exact Boolean operation's results obtained from ACIS (or CGAL). From Table 5, we can find that the error generated by this method is small, and the accuracy converges while increasing the sampling rates.

9. Conclusion

In this paper, we propose a novel solid modeling framework using Layered Depth-Normal Images (LDNI) to represent solid models on GPU. All steps of the framework including sampling, computing of Boolean operations and contouring map to the architecture of modern graphics hardware quite well so that a fully GPU-based solid modeler can be implemented by our approach. Results with massive number of triangles have been successfully tested on our prototype implementation, where most of them fail on the state-of-the-art commercial (or open-source) solid modeling kernels (i.e., ACIS and CGAL). In short, our solid modeling framework running on GPU can compute operations on complex models more efficiently and effectively.

Acknowledgements

This work is partially supported by CUHK Direct Research Grant (CUHK/2050400) and Hong Kong RGC CERG Grant (CUHK/417508). The third author would like to acknowledge the support of a National Science Foundation grant CMMI-0927297 and CMMI-0927397. The authors would like to thank Mr. Yunbo Zhang to help implement the Boolean operation program by ACIS and Mr. Pu Huang for implementing the Boolean operation program by CGAL.

Appendix

The pseudo-code of the **Algorithm** *BooleanOperationOnRay* is shown below.

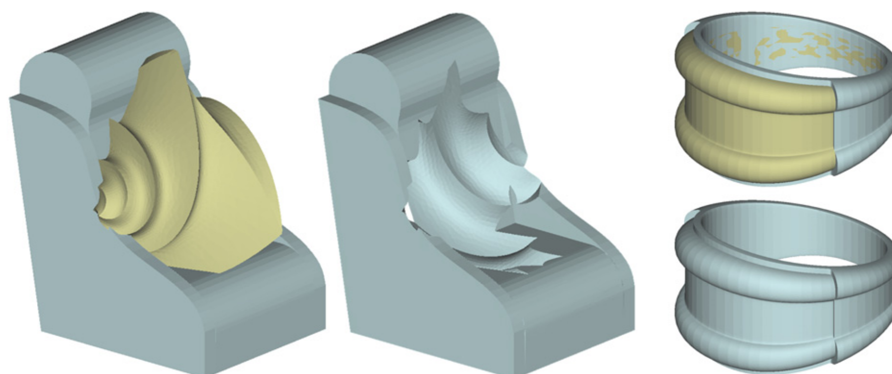


Fig. 9. The results of our approach on two more examples — "Chair \ Octa-Flower" (left) and "Ring-A ∪ Ring-B" (right) that can be successfully computed by CGAL [33].

Author's personal copy

Algorithm 1 BooleanOperationOnRay(R_A,R_B)

1: Initialize a *NULL* result sample list R_{res};
2: **if** $(n_A > 0)$ AND $(n_B > 0)$ **then**
3: $inside A = inside B = false$;
4: $lastConfig = OPRT(inside A, inside B)$;
5: $i_A = i_B = 0$;
6: **while** $(i_A < n_A)$ AND $(i_B < n_B)$ **do**
7: **if** $(i_A < n_A)$ AND $((i_B == n_B)$ OR $(R_A[i_A].d < R_B[i_B].d))$ **then**
8: $s = R_A[i_A]$ and $i_A = i_A + 1$;
9: $inside A = ((i_A\%2) == 0)$;
10: **else**
11: $s = R_B[i_B]$ and $i_B = i_B + 1$;
12: $inside B = ((i_B\%2) == 0)$;
13: For '\' operation, reverse the normal of s;
14: **end if**
15: $config = OPRT(inside A, inside B)$;
16: **if** $(config! = lastConfig)$ **then**
17: Insert s at the tail of R_{res};
18: $lastConfig = config$;
19: **end if**
20: **end while**
21: **else if** $(n_A == 0)$ AND $(n_B > 0)$ **then**
22: For '∪' operation, copy all samples from R_B to R_{res};
23: **else if** $(n_A > 0)$ AND $(n_B == 0)$ **then**
24: For '∩' operation, remove all samples from R_A;
25: Copy all sample from R_B to R_{res};
26: **end if**
27: **return** R_{res};

References

[1] Hable J, Rossignac J. Cst: Constructive solid trimming for rendering breps and csg. IEEE Trans Vis Comput Graph 2007;13(5):1004–14.
[2] Hable J, Rossignac J. Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. In: SIGGRAPH'05: ACM SIGGRAPH 2005 papers. New York (NY, USA): ACM; 2005. p. 1024–31. doi:http://doi.acm.org/10.1145/1186822.1073306.
[3] Chua C, Leong K, Lim C. Rapid prototyping: Principles and applications. World Scientific; 2003.
[4] Hartquist E, Menon J, Suresh K, Voelcker H, Zagajac J. A computing strategy for applications involving offsets, sweeps, and minkowski operations. Comput Aided Des 1999;31(3):175–83.
[5] Ju T, Losasso F, Schaefer S, Warren J. Dual contouring of hermite data. In: SIGGRAPH'02: Proceedings of the 29th annual conference on computer graphics and interactive techniques. New York (NY, USA): ACM; 2002. p. 339–46. doi:http://doi.acm.org/10.1145/566570.566586.
[6] Pfister H, Zwicker M, van Baar J, Gross M. Surfels: Surface elements as rendering primitives. In: SIGGRAPH'00: Proceedings of the 27th annual conference on computer graphics and interactive techniques. New York (NY, USA): ACM Press/Addison-Wesley Publishing Co.; 2000. p. 335–42. doi:http://doi.acm.org/10.1145/344779.344936.
[7] Hoffmann C. Geometric and solid modeling. San Mateo, Calif: Morgan Kaufmann; 1989.
[8] Rossignac J, Requicha A. Solid modeling. In: Encyclopedia of electrical and electronics engineering. John Wiley and Sons; 1999.
[9] Hoffmann C. Robustness in geometric computations. ASME J Comput Inform Sci Eng 2001;1:143–56.
[10] Smith JM, Dodgson NA. A topologically robust algorithm for boolean operations on polyhedral shapes using approximate arithmetic. Comput Aided Des 2007;39(2):149–63. doi:http://dx.doi.org/10.1016/j.cad.2006.11.003.
[11] Mortenson M. Geometric modeling. 2nd ed. New York: Wiley; 1997.
[12] Pauly M, Keiser R, Kobbelt LP, Gross M. Shape modeling with point-sampled geometry. In: SIGGRAPH'03: ACM SIGGRAPH 2003 papers. New York (NY, USA): ACM; 2003. p. 641–50. doi:http://doi.acm.org/10.1145/1201775.882319.
[13] Adams B, Dutré P. Interactive boolean operations on surfel-bounded solids. ACM Trans Graph 2003;22(3):651–6. doi:http://doi.acm.org/10.1145/882262.882320.
[14] Zhang N, Qu H, Kaufman A. Csg operations on point models with implicit connectivity. In: CGI'05: Proceedings of the computer graphics international 2005. Washington (DC, USA): IEEE Computer Society; 2005. p. 87–93.
[15] Nielsen MB, Nilsson O, Söderström A, Museth K. Out-of-core and compressed level set methods. ACM Trans Graph 2007;26(4):16. doi:http://doi.acm.org/10.1145/1289603.1289607.
[16] Ellis JL, Kedem G, Lyerly TC, Thielman DG, Marisa RJ, Menon JP, et al. The ray casting engine and ray representatives. In: SMA'91: Proceedings of the first ACM symposium on solid modeling foundations and CAD/CAM applications. New York (NY, USA): ACM; 1991. p. 255–67. doi:http://doi.acm.org/10.1145/112515.112548.
[17] Menon JP, Voelcker HB. On the completeness and conversion of ray representations of arbitrary solids. In: SMA'95: Proceedings of the third ACM symposium on solid modeling and applications. New York (NY, USA): ACM; 1995. p. 175–286. doi:http://doi.acm.org/10.1145/218013.218057.
[18] Rocchini C, Cignoni P, Ganovelli F, Montani C, Pingi P, Scopigno R. Marching intersections: An efficient resampling algorithm for surface management. In: SMI'01: Proceedings of the international conference on shape modeling & applications. Washington, DC, USA: IEEE Computer Society; 2001. p. 296.
[19] Rocchini C, Cignoni P, Ganovelli F, Montani C, Pingi P, Scopigno R. The marching intersections algorithm for merging range images. Vis Comput 2004;20(2):149–64. doi:http://dx.doi.org/10.1007/s00371-003-0237-8.
[20] Kobbelt LP, Botsch M, Schwanecke U, Seidel H-P. Feature sensitive surface extraction from volume data. In: SIGGRAPH'01: Proceedings of the 28th annual conference on computer graphics and interactive techniques. New York (NY, USA): ACM; 2001. p. 57–66. doi:http://doi.acm.org/10.1145/383259.383265.
[21] Trapp M, Döllner J. Real-time volumetric tests using layered depth images. In: Mania ERK, editor. Proceedings of eurographics 2008, eurographics. The Eurographics Association; 2008. p. 235–8.
[22] Shade J, Gortler S, He L-w, Szeliski R. Layered depth images. In: SIGGRAPH'98: Proceedings of the 25th annual conference on computer graphics and interactive techniques. New York (NY, USA): ACM; 1998. p. 231–42. doi:http://doi.acm.org/10.1145/280814.280882.
[23] Chen Y, Wang CCL. Layered depth-normal images for complex geometries - part one: Accurate sampling and adaptive modeling. In: Proceedings of ASME IDETC/CIE 2008 conference, 28th computers and information in engineering conference. ASME; 2008.
[24] Wang CCL, Chen Y. Layered depth-normal images for complex geometries - part two: manifold-preserved adaptive contouring. In: Proceedings of ASME IDETC/CIE 2008 conference, 28th computers and information in engineering conference. ASME; 2008.
[25] Gross M, Pfister H. Point-based graphics. Burlington, MA: Morgan Kaufmann; 2007.
[26] Wang CCL, Chen Y. Layered depth-normal images: A sparse implicit representation of solid models, Technical Report, The Chinese University of Hong Kong, 2007.
[27] Everitt C. Interactive order-independent transparency, Technical report, NVIDIA Corporation, 2001.
[28] Heidelberger B, Teschner M, Gross MH. Real-time volumetric intersections of deforming objects. In: VMV. 2003. p. 461–8.
[29] Blythe D. Advanced graphics programming techniques using opengl, SIGGRAPH99 Course Notes, 1999. http://www.opengl.org/resources/code/samples/sig99/advanced99/notes/notes.html.
[30] Harris M, Sengupta S, Owens JD. Parallel prefix sum (scan) with cuda. In: Nguyen H, editor. GPU Gems 3. Addison Wesley; 2007.
[31] NVIDIA, CUDA programming guide. http://developer.nvidia.com/object/cuda.html version 2.0.
[32] Lorensen WE, Cline HE. Marching cubes: A high resolution 3d surface construction algorithm. SIGGRAPH Comput Graph 1987;21(4):163–9. doi:http://doi.acm.org/10.1145/37402.37422.
[33] CGAL. Computational geometry algorithm library. http://www.cgal.org version 3.4.
[34] Spatial, 3D ACIS modeling. http://www.spatial.com. R15.
[35] Cignoni P, Rocchini C, Scopigno R. Metro: Measuring error on simplified surfaces. Computer Graphics Forum 1998;17(2):167–74.