Fast Dexelization of Polyhedral Models Using Ray-Tracing Cores of GPU

Masatomo Inui[1], Kohei Kaba[2] and Nobuyuki Umezu[3]

[1]Ibaraki University, masatomo.inui.az@vc.ibaraki.ac.jp
[2]Ibaraki University, 19nm425n@vc.ibaraki.ac.jp
[3]Ibaraki University, nobuyuki.umezu.cs@vc.ibaraki.ac.jp

Corresponding author: Masatomo Inui, masatomo.inui.az@vc.ibaraki.ac.jp

Abstract. State-of-the-art GPUs are equipped with special hardware called RT cores dedicated to image processing for a type of 3D computer graphics called ray-tracing. In this paper, we propose a novel method for fast dexelization of a complex polyhedral model using RT cores. NVIDIA Corporation provides an API library for ray-tracing computations named Optix. The function of an RT core is automatically available via the API function provided by Optix. In order to evaluate the effectiveness of the RT core in dexel processing, a dexelization software for polyhedral models was implemented using the API of Optix, and certain computational experiments were conducted. Via the experiments, the effectiveness of the RT core of GPU in dexel modeling was verified.

Keywords: voxelization, RT core, Optix, parallel processing, GPU
DOI: https://doi.org/10.14733/cadaps.2021.786-798

1 INTRODUCTION

Boundary representation (B-reps) is a standard method of solid modeling for CAD systems of mechanical products. In B-reps modeling, three-dimensional (3D) objects are represented as collections of interconnected closed surface elements. Operations related to B-reps models, e.g., Boolean set operation, require techniques like intersection calculation of the surface elements, trimming operation of the elements based on calculations, and reconstruction of adjacency relationships of the trimmed elements. These operations are computationally expensive, and the associated topological reconstruction process tends to be unstable owing to the existence of unavoidable floating-point errors in the intersection calculations.

1.1 NC Milling Simulation Using Dexel Model

Techniques using voxels, dexels [21], rays [16], or layered depth images (LDI) [18] based on the uniform decomposition of the 3D space have been widely used as solid modeling methods that do not suffer from the aforementioned problems. With the popularization of these methods, dexel-based solid modeling is rapidly becoming a standard method to represent object shapes in NC milling simulation [7], [10], [20]. A milling operation is geometrically equivalent to a Boolean subtraction of the swept volume of a cutter moving along a cutter path from a geometric model representing the workpiece shape. In dexel modeling, the workpiece shape is represented by a bundle of Z-axis-aligned segments defined corresponding to each grid point of a square mesh in the XY plane (consult Figure 1(a)). As no topological information is used during model representation, Boolean operations in dexel modeling are much more robust than the operations used in B-reps modeling.

During dexel modeling, near-vertical surfaces inevitably exhibit significant shape errors caused by finite grid resolution. The triple-dexel model was proposed to overcome this non-uniformity in shapes and to realize an accurate shape representation (consult Figure 1(b)) [1]. In this representation, besides being defined by a Z-axis-aligned dexel model, the 3D shape is also defined by an X-axis-aligned dexel model based on a square mesh in the YZ plane and a Y-axis-aligned dexel model based on a mesh in the ZX plane. Thus, via triple-dexel representation, the object space can be divided into a set of cubic cells (voxels) defined by properly positioned X-, Y-, and Z-axis-aligned dexels (consult Figure 2). This makes triple-dexel modeling capable of representing the same 3D shapes with the same resolutions using less memory than voxel models.

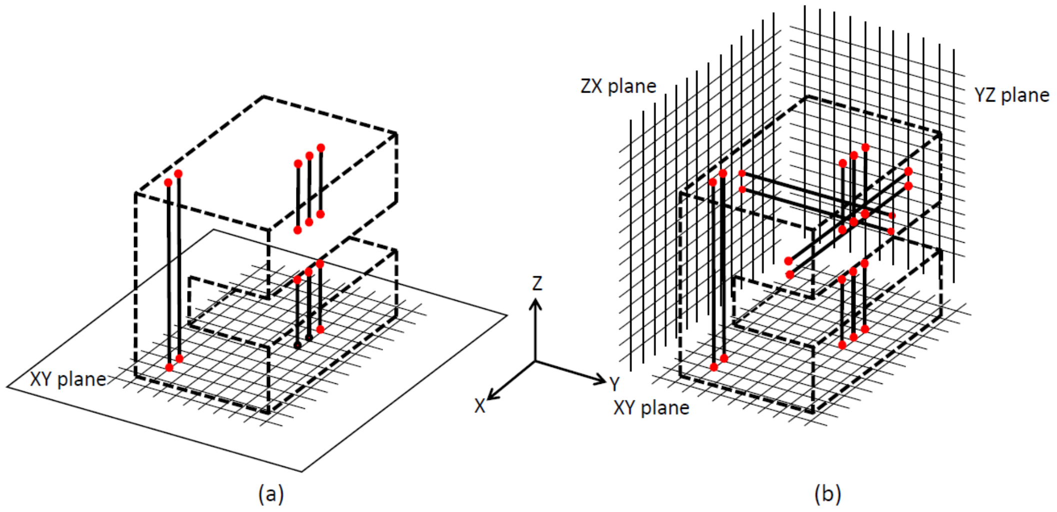

Figure 1: (a) The dexel model and (b) the triple-dexel model with X, Y, Z-axis-aligned dexels.

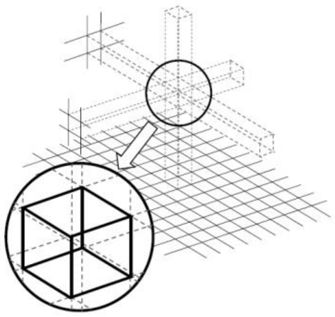

Figure 2: Cubic cell (voxel) defined by properly positioned X-, Y-, and Z-axis-aligned dexels.

In dexel (and triple-dexel) modeling, Boolean operations of 3D objects can be decomposed into simple Boolean computations using collinear dexel segments corresponding to each grid point. These computations are independent of segments lying on other lines. Thus, a dexel-wise Boolean computation can be conducted in a parallel manner. Techniques using graphics processing units (GPUs) have recently become popular in the implementation of parallel computation software. Current GPUs are designed using thousands of streaming processors (SPs) on a single chip. In programming environments such as compute unified device architecture (CUDA) [3], this enables programmers to utilize GPUs as general-purpose parallel processors in which each SP executes a single unit computation (or thread), e.g., simultaneous Boolean operations on dexels.

1.2 Conversion of a B-reps Model into a Dexel Model

During dexel-based milling simulation, initial workpiece models are usually available as B-reps models. The simulation converts them to their equivalent dexel models during the initial step. The workpiece model is affixed to a table in the milling environment during the simulation. During conventional 3-axis machining, the orientation of the workpiece is changed after the machining for a certain side is finished, and it is remounted on the table to machine another of its sides. As the definition of the dexel model is based on a square mesh fixed in the coordinate plane, rigid body transformations, especially rotation of the object, are difficult. Therefore, it is necessary to initially convert the dexel model to a B-reps model, and revert it to a dexel model again following any necessary rigid body transformations.

A dexel model can be converted into an equivalent polyhedral model using an algorithm called Quad Pillars, developed previously by the authors themselves [9]. As a triple-dexel model is equivalent to a voxel model, polyhedrization of the model is possible by employing surface extraction algorithms of cell structures, such as the marching cubes algorithm [15]. During the conversion of a B-reps model into a dexel model, the algorithm proposed in [16] is usually used. We refer to this algorithm as the "ray algorithm" in the following discussion. It defines a set of rays perpendicular to the coordinate plane corresponding to all grid points of the square mesh in the plane. The intersections between the rays and the B-reps model are computed, and subsequently converted into dexel segments.

A B-reps model representing a resultant shape from a milling operation often comprises a high number of small polygons — sometimes exceeding even 50,000,000 polygons. This causes the ray algorithm to take upwards of 10 minutes to convert the model into an accurate dexel model defined based on a 5,000×5,000 resolution grid. Attempting to reduce of the conversion time has become a serious problem in milling simulation. State-of-the-art GPUs are equipped with special hardware named RT cores dedicated to image processing called ray-tracing in 3D computer graphics. It is expected that various types of dexel processing can be accelerated using RT cores. As an initial step in the application of RT cores in dexel modeling, we propose a novel method for fast dexelization of complex polyhedral models using RT cores in this paper.

In the next section, algorithms for conversion of B-reps models to cell decomposition models, such as dexels and voxels, are briefly reviewed. Polyhedron-to-dexel conversion using the ray algorithm is discussed in the third section. Programming using RT cores and the implementation method of the ray algorithm using the RT core are illustrated in same section. Section 4 presents some results of the computational experiments that we conducted. Finally, concluding remarks are presented in section 5.

2 RELATED WORKS

In this section, existing studies related to model conversion of B-reps models into cell decomposition models are reviewed.

A dexel model [21] and a ray representation [16] can be regarded as identical representation methods of a solid object, enabling the conversion of a B-reps model into a dexel model using the ray algorithm. Another method that can be used for B-reps-to-dexel conversion is depth peeling. Depth peeling was originally developed to achieve order-independent transparency [5], but it proved to be applicable for obtaining an LDI model from a B-reps solid model. LDI operates by recording depth values of surfaces of 3D objects from a certain viewpoint in a multiple-layer system [18]. In the first layer, the depth of the nearest surface of the solid from the viewpoint is recorded corresponding to each pixel, and in the second layer, the depth of the next surface of the solid from the viewpoint is recorded. Depth peeling refers to the iteration of this process until all the surfaces of the solid have been recorded, and the result is an LDI of the object. If orthogonal projection is used to display a solid object, the measurements recorded via depth peeling are identical to those obtained from the ordered intersections between the solid and parallel rays corresponding to each pixel, enabling facile conversion of the LDI into an equivalent dexel model. Zhao et al. proposed a robust depth peeling method [23] and Wang et al. developed a GPU-accelerated depth peeling algorithm [22].

Conversion of B-reps models to voxel models has been extensively studied in the field of computer graphics. If a spatial cell structure containing a solid object within it is considered, the voxel model represents the object as a set of small cubes by replacing the cells that overlap the object with cubes. Three replacement criteria are used during this process — (1) replacement of a cell completely included in the object, (2) replacement of a cell partially overlapping the object, (3) replacement of a cell whose center point lies within the object. The last replacement criterion can be realized if a ray representation or a dexel model of the object is obtained. More sophisticated methods have also been proposed, e.g., subdivision of the solid object [13] or a method based on the computation of the distance between a cell and the object's surface [11].

With the popularization of hardware equipped for graphics processing, various conversion techniques using hardware functions have been developed. Karabassi et al. developed a conversion algorithm using the depth buffer mechanism [12]. A slicing-based voxelization algorithm was proposed by Chen and Fang. Their method generates slices of the underlying model in the frame buffer by employing appropriate clipping planes and obtains the voxel model based on the image of the section [2]. Heidelberger et al. [6] presented an effective algorithm for rapid layered depth image generation that can be extended to voxelization.

A voxelization algorithm developed by Dong et al. initially converts any solid object into a discrete voxel space. The resultant voxels are encoded as 2D textures and stored in three intermediate sheet buffers. Finally, these buffers are synthesized into one worksheet representing a voxel model [4]. Li and McMain proposed a GPU-accelerated voxelization algorithm in their Minkowski sum computation software [14]. Their method proceeds by generating possible surface elements corresponding to the Minkowski sum shape of two objects. Voxels corresponding to the Minkowski sum shape are then selected based on the identified surface elements. NVIDIA, a hardware developer of GPUs, posted a note explaining GPU-accelerated method to convert a polyhedral solid model into a voxel model [19].

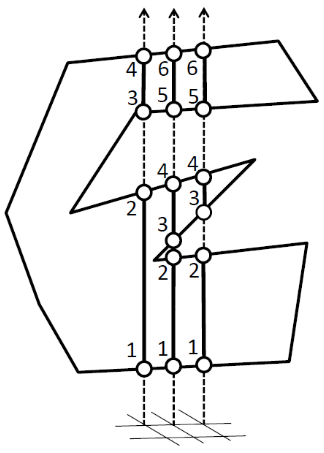

Figure 3: Polyhedron-to-dexel conversion.

3 HARDWARE-ACCELERATED RAY ALGORITHM

In this paper, we describe a method to accelerate the ray algorithm using RT cores of GPUs. Based on the obtained dexel model, the voxel model of the solid is easily obtained. By using RT cores, polyhedral models composed of a significant number of polygons can be converted into high-resolution dexel models over considerably short durations. To the best of our knowledge, this is the first study investigating the computation of a dexel model using RT cores.

3.1 The Ray Algorithm

Figure 3 illustrates the fundamental processing steps involved in the ray algorithm to convert a polyhedral solid model into its equivalent dexel model. We assume that no gaps or overlaps exist between the surface polygons in the polyhedral model. Several polyhedral models — STL ones in particular — do not satisfy this condition. However, polyhedral models in boundary representations always satisfy this condition.

Consider an axis-aligned regular square mesh on the XY-plane that contains the projection of the object onto the XY-plane. The appropriate resolution of the mesh is determined by considering the representation accuracy of the dexel model and the memory capacity of the computer. During NC milling simulation, the resolution is usually set to be approximately equal to or greater than 5,000 × 5,000. Corresponding to each grid point, a ray perpendicular to the coordinate plane (XY plane) is extended along the Z-axis, and its points of intersection with the surface polygons of the object are computed. These intersection points are then sorted along the direction of the ray. By connecting odd-numbered intersection points with even-numbered intersection points using lines, a dexel model equivalent to the polyhedral model is obtained. If the dexel model is required along the X-axis or the Y-axis, the same computation is repeated for equivalent grids on the YZ-plane or the ZX-plane.

The mutual independence of the computations corresponding to each grid point renders parallel processing effective in accelerating the ray algorithm. To further improve the conversion performance, the acceleration of the following two processes is necessary.

1. Computation of the points of intersection between surface polygons of the polyhedron and a ray.
2. Sorting of intersection points along the ray.

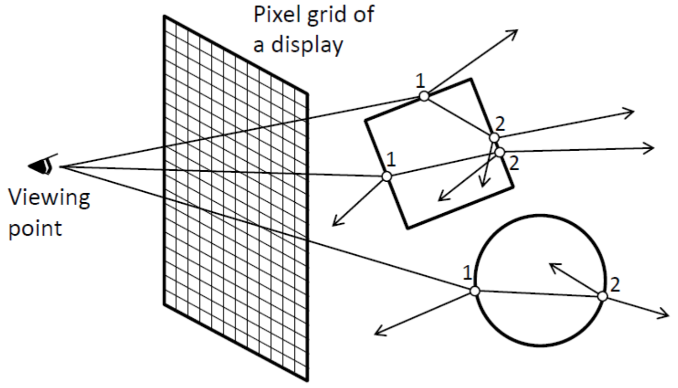

Figure 4: Computation process during ray-tracing.

3.2 Real-Time Ray-Tracing

Ray-tracing is a computer graphics technique used to generate photo-realistic images. As depicted in Figure 4, this method transmits a ray of light passing through each pixel of the display from the viewing point. When the ray collides with a solid shape placed in a virtual space, it is divided into two rays, one of which is reflected from the solid surface, and the other, which is refracted and advances into the solid's interior. This process is recursively repeated, and when any light source is reached, the corresponding color component is transmitted back to the viewing point along the reverse direction of the ray to generate a three-dimensional computer image. Although this technique is known to produce high quality images, its computational cost is very high and its use in applications requiring real-time performance is difficult.

Most currently available GPUs adopt many core architectures and comprise several computational units, e.g., CUDA cores in NVIDIA's GPUs. Among the GPUs supplied by NVIDIA, products whose names include the prefix, RTX-, comprise a set of processing cores specialized for ray-tracing computations. They are called RT (ray-tracing) cores. For example, RTX-2080 GPU comprises 46 RT cores in addition to 2944 CUDA cores. By using RT cores, ray-tracing images of complex 3D virtual environments including multiple polyhedral objects can be generated in real-time (60 frames per second) for a full-HD display.

During ray tracing, the following two operations are repeatedly executed.

- A straight line (ray) is extended through each grid point of the pixel-grid of a display. The ray can be set to be perpendicular to the display in the orthogonal projection.
- Corresponding to each ray, its point of intersection with the solid surface is computed. A part of the ray subsequently proceeds into the solid's interior, and the aforementioned intersection point computation with respect to the solid's surface is recursively executed. As a result, a series of ordered intersection points are obtained along the ray.

These operations are almost identical to the ones necessary during the conversion of polyhedral models into dexel models using the ray algorithm. Therefore, fast dexelization of the polyhedral models is enabled by using RT cores.

3.3 Optix, API for Ray-Tracing Computations

NVIDIA Corporation provides an API library named Optix [17] for ray-tracing computations. The function of an RT core is automatically available through the API function of Optix. In order to evaluate the effectiveness of the RT core in dexel processing, a dexelization software of the polyhedral model was implemented using the API of Optix, and computational experiments were performed.

Optix is implemented using C++ and it defines following three basic classes:

- **Context class:** A fundamental class for defining ray-tracing functions. Almost all events are defined as instances of this class.
- **Geometry class:** A class for defining geometric properties and operations, e.g., coordinates of polygons, a bounding box, and a function for intersection computations. We use geometry triangles class which is specialized for dealing with triangular polygons.
- **Material class:** A class that defines the reflectivity, refractive index and other material properties of a solid object.

Though various functions can be implemented via Optix, the following three kinds of functions are used in our polyhedron-to-dexel conversion software.

- **Ray-generation program:** This is a function called by the launch function that starts Optix processing. Ray-tracing processing is initiated here. Functions in this program are defined in the context class.
- **Closest hit program:** This function is called when a ray collides with the nearest polygon during the ray-tracing process. Information about intersection points that is necessary in the polyhedron-to-dexel conversion is recorded by using this function. Functions in this program are defined in the material class.

- **Miss program:** This function is called when a ray does not intersect any solids during the ray-tracing process. This function is used to terminate the ray-tracing operation. Functions in this program are defined in the context class.

To initiate Optix processing, the launch function of the context class is invoked based on the number of rays. Functions of Optix can be used in the threads invoked in the CUDA environment. In this case, a function named *rtTrace* is called in each thread, together with the information of the polyhedral model and the corresponding ray.

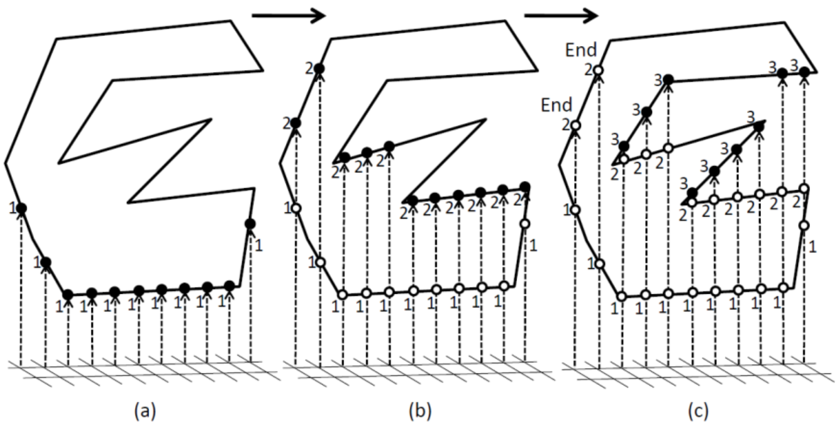

Figure 5: Polyhedron-to-dexel conversion process using Optix.

3.4 The Ray Algorithm Using Optix

The processing flow of our dexelization algorithm using Optix is as follows (consult Figure 5). In our implementation, the processing steps between Step 5 and Step 7 are executed by threads in a CUDA program, enabling the parallel computation of dexels for multiple rays.

Step 1: A context is generated. Optix constants and buffer for I/O are defined, followed by the definition of the ray-generation program and the miss program.

Step 2: Geometry triangles are generated. Following that, vertex data array of the triangles of the input model are transferred to the Optix buffer.

Step 3: A material class is created to define a closest hit program.

Step 4: The number of rays are specified to the launch function to initiate Optix processing.

Step 5: A set of rays is defined to be used during the Optix processing. Grid points on the XY-plane are indicated as starting points for the rays, and a direction perpendicular to the XY-plane is indicated as direction of the rays. These operations are simultaneously executed by multiple threads.

Step 6: Ray-tracing computations are performed using the *rtTrace* function for each thread.

Step 7: Information regarding the intersection points is stored using the closest hit program which is invoked whenever the intersection points are calculated (consult Figure 5(a)). The starting point of the ray is then updated to be the intersection point, and Step 6 is executed again (consult Figure 5(b)). If no intersection point is detected for a ray, the miss program is called and the exit flag is raised (consult the two lines at the left end in Figure 5(c)). In this case, the Optix process is terminated for the ray.

Step 8: The obtained intersection points for the ray have already been sorted along the ray. For each ray, pairs of odd-numbered intersection point and even-numbered intersection point are checked, and then connected to obtain the dexels corresponding to the ray.

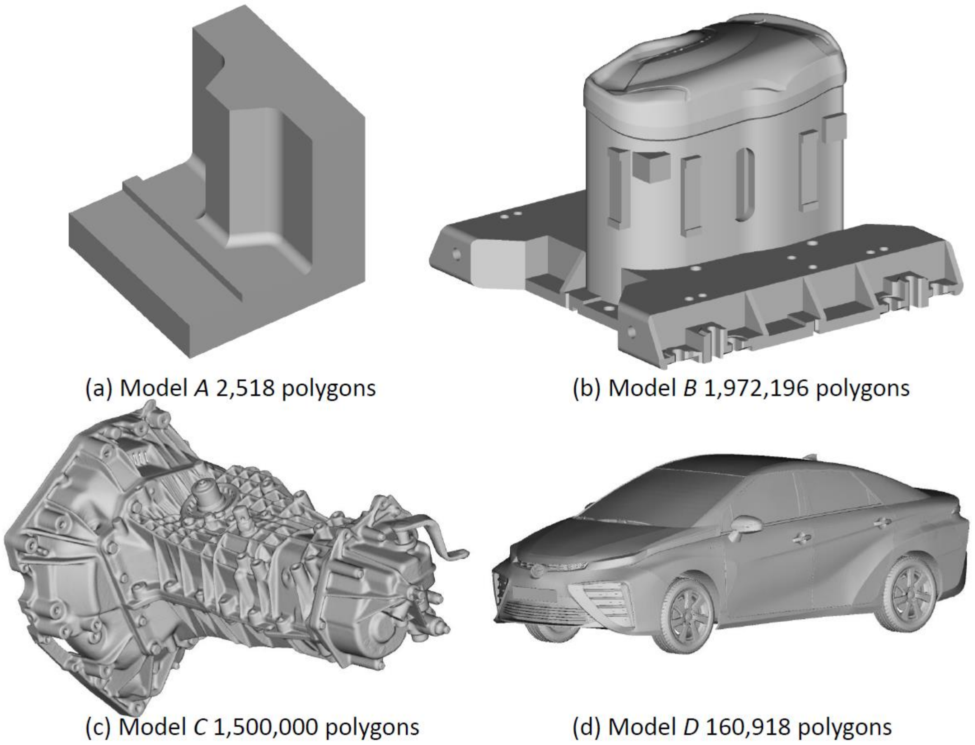
(a) Model *A* 2,518 polygons
(b) Model *B* 1,972,196 polygons
(c) Model *C* 1,500,000 polygons
(d) Model *D* 160,918 polygons

Figure 6: Sample models *A*, *B*, *C* and *D*.

4 COMPUTATIONAL EXPERIMENTS

To evaluate the performance of polyhedron-to-dexel conversion using RT cores, Step 1 to Step 7 of the aforementioned algorithm are implemented using Optix. A polyhedral model in the STL format is taken to be the input. Our software generates a set of vertical rays starting from grid points of the square mesh placed on the XY-plane, and then computes all intersection points between the rays and the polyhedral model. The obtained points are already order along the rays during the ray-tracing process. The software was implemented using VisualStudio 2017, CUDA 10.1, Optix 6.0, and Cmake 3.14. The specifications of the PC used in the experiment are CPU: Intel Core i9-9900 (3.60 GHz), RAM: 32GB, GPU: GeForce RTX-2080.

Instead of the workpiece models representing the milling result, we use six polyhedral models (model *A*, *B*, *C*, *D*, *E*, and *F*) depicted in Figure 6, 7, 8 and 10 in our conversion experiments for the purpose of maintaining confidentiality. Number of polygons of the models and resolutions of the square mesh for the dexel representation are illustrated in the figures. Figure 7 shows the conversion results for four models given in Figure 6. In Figure 7, ordered intersection points between the vertical rays and the models are illustrated. By connecting odd-numbered intersection point and even-numbered intersection point with a line segment, dexel models equivalent to the models given in Figure 6 are obtained. The time required for the processing was 0.436 s for *A*, 0.524 s for *B*, 0.554 s for *C* and 0.369 s for *D*. We have implemented another ray-algorithm-based conversion software. This software realizes the conversion using CUDA cores of the GPU. In the conversion, this software needs 7.26 s for model *B* and 1.95 s for model *D*. The conversion software using RT cores is 13.9 times faster for model *B* and 5.3 times faster for model *D*.

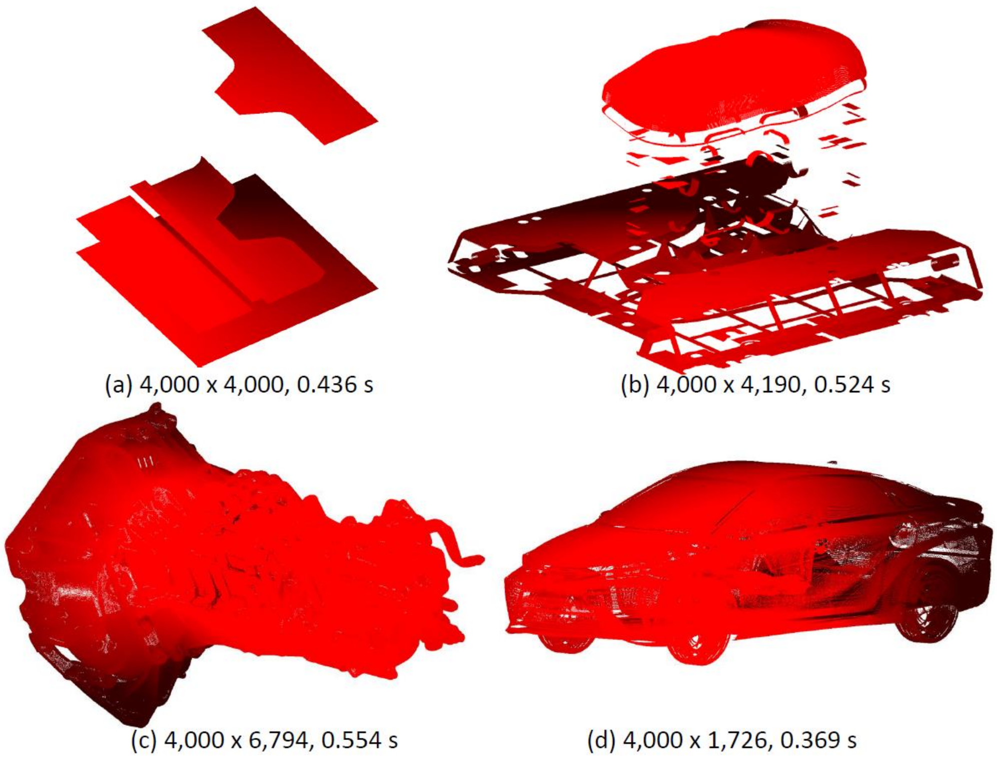
(a) 4,000 x 4,000, 0.436 s
(b) 4,000 x 4,190, 0.524 s
(c) 4,000 x 6,794, 0.554 s
(d) 4,000 x 1,726, 0.369 s

Figure 7: Sorted intersection points obtained by our software for models given in Figure 6.

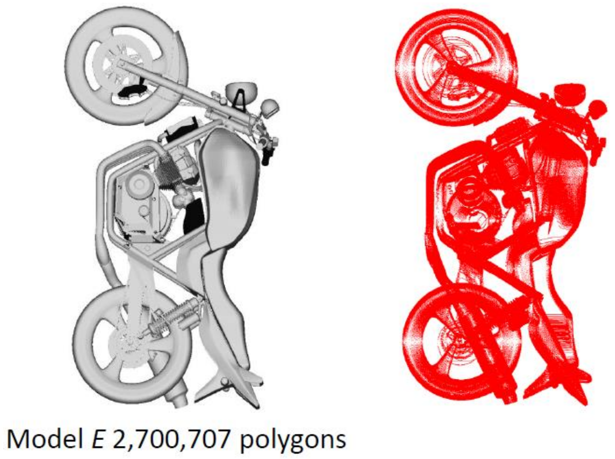

Figure 8: Sorted intersection points obtained by our software for model *E*.

Num. Cells	1,000,000	4,000,000	6,250,000	16,000,000	36,000,000
Time (s)	0.442	0.474	0.469	0.544	0.687

Table 1: Necessary computation time for obtaining the conversion result for model *E* with different numbers of the cell of the square mesh.

In the ray algorithm, the number of the cells of the square mesh is a critical factor for the performance of the software. Using a square mesh with a higher resolution, a more accurate dexel model can be obtained; however, the time and cost for the computation also increase. We executed experiments to investigate the relationship between the number of the cells and necessary conversion time. Table 1 shows the necessary computation time when the resolution of the square mesh varies and the number of the cells is increased. In the experiments, model *E* with 2,700,707 polygons was used as shown in Figure 8. Figure 9 shows a graph representing the relationship between the number of the cells and the necessary conversion time. As can be observed from the graph, the necessary computation time is proportional to the number of the cells. It is found that the slope of the graph is gentle and the calculation time does not increase much with increasing the mesh resolution. Since the number of RT cores and CUDA cores is limited, it is not possible to process more rays in parallel. GPU has a scheduling function, and if more rays need to be processed, this function repeatedly executes the parallel computation on the limited number of rays until all the rays have been processed. The increase in processing time with the increase in the number of cells is considered to be a result of this scheduling function.

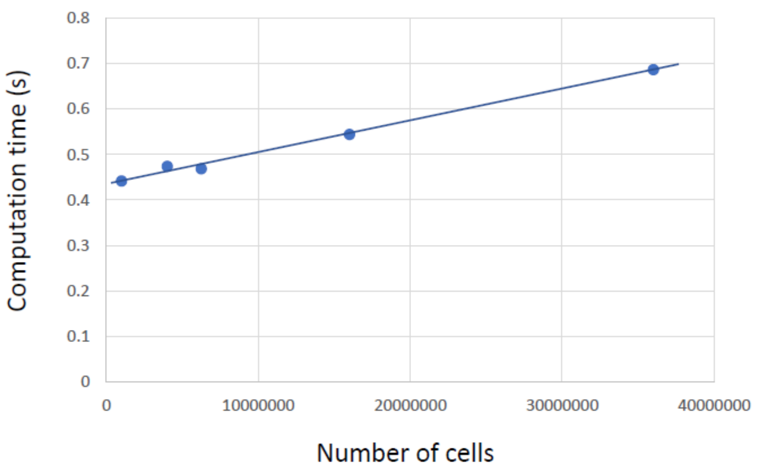

Figure 9: A graph of the necessary computation time for obtaining the conversion result for model *E* with different numbers of the cell of the square mesh.

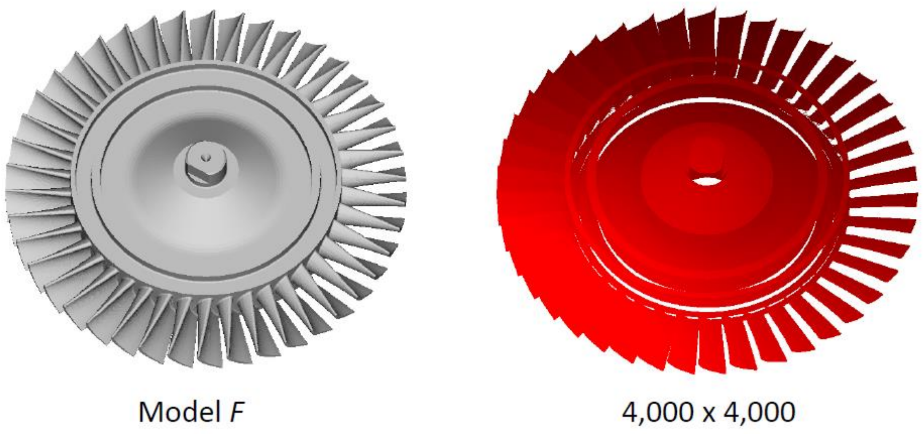

Figure 10: Sorted intersection points obtained by our software for model *F*.

Another critical factor of the conversion is the number of the polygons of the input model. We executed experiments to investigate the relationship between the number of the polygons of the model and necessary conversion time. In this experiment, we prepare five polyhedral model (model *F*) of the same shape but with different number of polygons. The shape of the model and its conversion result is given in Figure 10. In the conversion, a square mesh with 4,000 × 4,000 resolution was used for defining a dexel model. Table 2 shows the necessary computation time when the number of polygons varies. Figure 11 shows a graph representing the relationship between the number of the polygons and the necessary conversion time. As can be observed from the graph, the necessary computation time is also proportional to the number of the polygons, but the slope of the graph is gentle and the calculation time does not increase much. Based on these observations, it can be concluded that the use of RT core enables the rapid conversion of a complex polyhedral model comprising a high number of polygons into its equivalent dexel model of very high resolution.

Num. Polys	197,450	393,832	787,664	1,572,504	3,145,008
Time (s)	0.431	0.427	0.447	0.510	0.564

Table 2: Necessary computation time for converting model *F* with different numbers of the polygons.

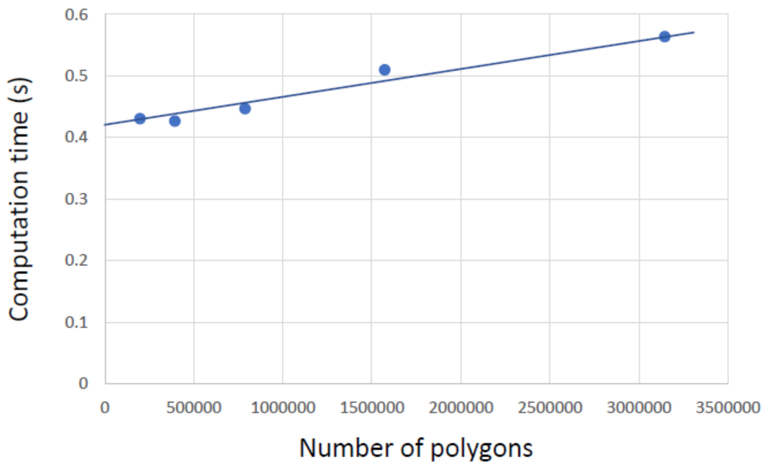

Figure 11: A graph of the necessary conversion time for model *F* with different numbers of the polygons.

5 CONCLUSIONS

State-of-the-art GPUs are equipped with special hardware called RT cores that are dedicated to ray-tracing. In this paper, we proposed a novel method for fast dexelization of a complex polyhedral model using RT cores. NVIDIA Corporation provides an API library named Optix for ray-tracing computations. The function of the RT core is automatically available through the API function of Optix. In order to evaluate the effectiveness of the RT core in dexel processing, a dexelization software of polyhedral models was implemented using the API of Optix, and computational experiments were performed. Based on the experimental results, it was confirmed that the time required for the conversion to the dexel model was only increased slightly even if the resolution of the dexel model was increased, or if the number of the polygons of the input model was increased.

We implemented a Minkowski sum computation software [8] and an NC machining simulation software using dexel representations of solid models. In our previous implementation, CUDA cores of GPUs had been used to realize parallel processing. We intend to re-implement the software for computation using RT cores. At present, it is necessary to mediate the API for computer graphics using systems such as Optix to use RT cores, and several restrictions remain on programming. For instance, the number of polygons that can be simultaneously handled in Optix is limited, and it is difficult to realize a computation using objects with more than hundred million polygons. We are currently investigating the problems and limitations of using RT cores and Optix, and examining their solutions.

ACKNOWLEDGEMENTS

This research was supported by the JSPS KAKENHI Grant Number 17K06075.

Masatomo Inui, https://orcid.org/0000-0002-1496-7680
Kohei Kaba, https://orcid.org/0000-0002-7580-4682
Nobuyuki Umezu, https://orcid.org/0000-0002-7873-7833

REFERENCES

[1] Benouamer, M.O.; Michelucci, D.: Bridging the gap between CSG and Brep via a triple ray representation, Proceedings of ACM Symposium on Solid Modeling and Applications, 1997, 68–79. https://doi.org/10.1145/267734.267755
[2] Chen, H; Fang, S: Fast voxelization of 3d synthetic objects, ACM Journal of Graphics Tools, 3(4), 1998, 33–45. https://doi.org/10.1080/10867651.1998.10487496
[3] CUDA compute unified device architecture programming guide, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, NVIDIA.
[4] Dong, Z.; Chen, W.; Bao, H.; Zhang, H.; Peng, Q.: Real-time voxelization for complex polygonal models. 12th Pacific Conference on Computer Graphics and Applications, 2004, 43–50. https://doi.org/10.1109/PCCGA.2004.1348333
[5] Everitt, C: Interactive order-independent transparency, In Technical Report. NVIDIA Corporation, 2001. https://my.eng.utah.edu/~cs5610/handouts/order_independent_transparency.pdf
[6] Heidelberger, B.; Teschner, M.; Gross, M.: Real-time volumetric intersections of deforming objects, Proceedings of Vision, Modeling, Visualization, 2003, 461–468.
[7] Huang. Y.; Oliver, J.H.: NC milling error assessment and tool path correction, SIGGRAPH 1994 Proc. 21st Annual Conference on Computer Graphics and Interactive Techniques, 1994, 287–294. https://doi.org/10.1145/192161.192231
[8] Inui, M.; Umezu, N.; Kitamura, Y.: Visualizing sphere-contacting areas on automobile parts for ECE inspection, Journal of Computational Design and Engineering, 2(1), 2015, 55-66. https://doi.org/10.1016/j.jcde.2014.11.006
[9] Inui, M.; Umezu, N.: Quad Pillars and Delta Pillars: Algorithms for Converting Dexel Models to Polyhedral Models, Journal of Computing and Information Science in Engineering, 17(3), September 2017, 9 pages, https://doi.org/10.1115/1.4034737
[10] Inui, M.; Kobayashi, M.; Umezu, N.: Cutter Engagement Feature Extraction Using Triple-Dexel Representation Workpiece Model and GPU Parallel Processing Function, Computer-Aided Design and Applications, 16(1), 2019, 89-102. https://doi.org/10.14733/cadaps.2019.89-102
[11] Jones, M.W.; Bærentzen, J.A.; Sr´amek, M.: 3D Distance Fields: A Survey of Techniques and Applications, IEEE Transaction on Visualization and Computer Graphics, 12(4), 206, 881-599. https://doi.org/10.1109/TVCG.2006.56

[12] Karabassi, E.A.; Papaioannou, G.; Theoharis, T.: A Fast Depth-Buffer-Based Voxelization Algorithm, Journal of Graphics Tools, 4(4), 1999, 5-10. https://doi.org/10.1080/10867651.1999.10487510

[13] Lai, S.; Cheng, F.: Voxelization of free-form solids represented by catmull-clark subdivision surfaces, Proceedings of the 4th international conference on Geometric Modeling and Processing (GMP'06), July 2006, 595–601. https://doi.org/10.1007/11802914_45

[14] Li, W.; McMains, S.: A GPU-based voxelization approach to 3D Minkowski sum computation, Proceedings of the 14th ACM Symposium on Solid and Physical Modeling (SPM '10), September 2010, 31–40. https://doi.org/10.1145/1839778.1839783

[15] Lorensen, W.E.; Cline, H.E.: Marching cubes: A high resolution 3D surface construction algorithm, Computer Graphics (Proceedings of ACM SIGGRAPH), 21(4), 1987, 163-169. https://doi.org/10.1145/37401.37422

[16] Menon, J.; Marisa, R.J.; Zagajac, J: More powerful solid modeling through ray representations, IEEE Computer Graphics and Applications, 14(3), May 1994, 22-35. https://doi.org/10.1109/38.279039

[17] NVIDIA OptiX™ Ray Tracing Engine, https://developer.nvidia.com/optix

[18] Shade, J.; Gortler, S.; He, L.W.; Szeliski, S.: Layered depth images, Proceedings of the 25th annual conference on Computer graphics and interactive techniques (SIGGRAPH '98), July 1998, 231–242. https://doi.org/10.1145/280814.280882

[19] Takeshige, M.: The Basics of GPU Voxelization, https://developer.nvidia.com/content/basics-gpu-voxelization

[20] Tukora, B.; Szalay, T.: Multi-dexel based material removal simulation and cutting force prediction with the use of general-purpose graphics processing units, Advances in Engineering Software, 43(1), January 2012, 65-70. https://doi.org/10.1016/j.advengsoft.2011.08.003

[21] VanHook, T.: Real-time shaded milling display, Computer Graphics (Proceedings of ACM SIGGRAPH), 20(4), 1986, 15-20. https://doi.org/10.1145/15886.15887

[22] Wang, C.C.L.; Manocha, D: GPU-based offset surface computation using point samples, Computer-Aided Design, 45(2), 2013, 321–330. https://doi.org/10.1016/j.cad.2012.10.015

[23] Zhao, H.; Wang, C.C.L.: Parallel and efficient Boolean on polygonal solids, The Visual Computer, 27, 507–517, 2011, https://doi.org/10.1007/s00371-011-0571-1