2017

Multi-level voxel representation for GPU-accelerated solid modeling

Gavin Young
Iowa State University

Multi-level voxel representation for GPU-accelerated solid modeling

by

Gavin Young

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Mechanical Engineering

Program of Study Committee:
Adarsh Krishnamurthy, Major Professor
Soumik Sarkar
Ming-Chen Hsu

The student author and the program of study committee are solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2017

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Dr. Adarsh Krishnamurthy for his support and guidance on my Master's study and research. His knowledge helped me through my research and writing of this thesis. I am extremely thankful to have had him as an advisor and mentor for my Master's. I would also like to thank the rest of my thesis committee: Dr. Soumik Sarkar and Dr. Ming-Chen Hsu, for their insight. I would like to thank my lab mates for always being willing to help and keeping the lab fun.

Finally, I would like thank my family for all the encouragement and support they have given me throughout my life.

This work was partly supported by the National Science Foundation under grant number CMMI-1644441. We would like to thank nVIDIA for proving us with the GPU to test our algorithms.

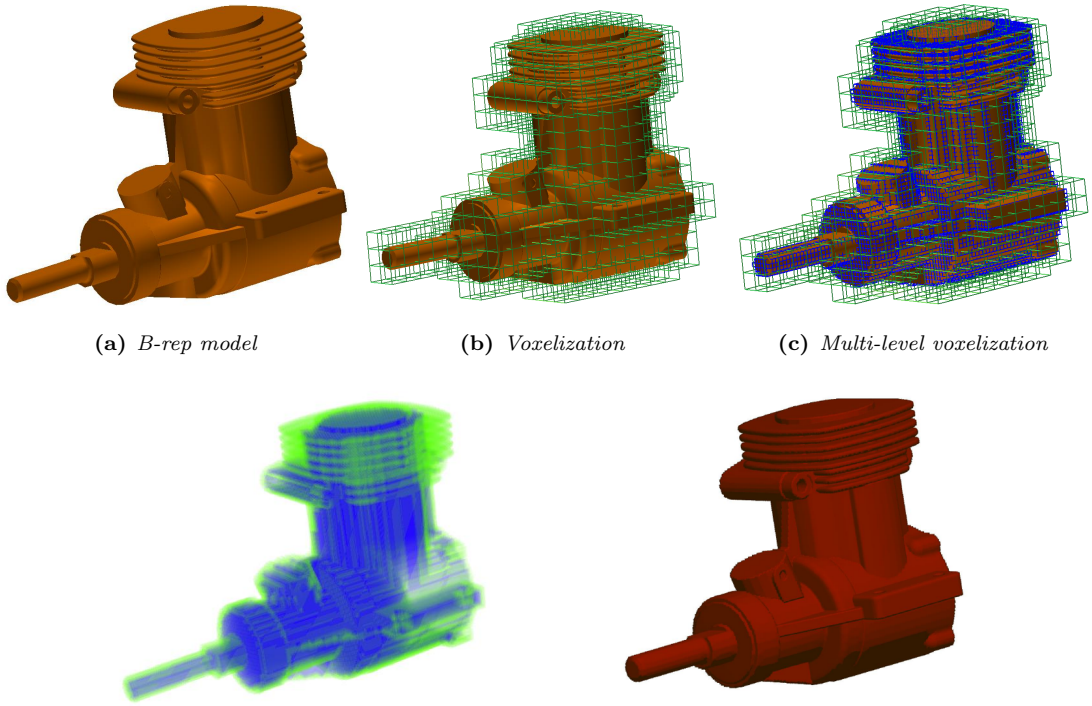
ABSTRACT

Solid models traditionally use boundary-representation (B-rep) to define and model their geometry. However, performing modeling operations such as Boolean operations or computing point membership classification with B-rep is computationally intensive, since B-reps do not have volumetric information. Voxelized representations, on the other hand, can be extended to include volumetric information of solid models. However, in order to use voxelized representations for solid modeling, efficient methods for voxelizing a B-rep solid model needs to be developed. In this thesis, GPU-accelerated methods are presented for creating and rendering a multi-level voxelization of a solid model that can be used along with the existing B-rep for modeling operations. Two GPU-accelerated algorithms are described; one for creating a multi-level voxelization given a B-rep of a solid model and another for ray casting to render the multi-level voxelization of the solid model. Compact and flat data structures are described that can be used to store the multi-level voxelization data and can be efficiently retrieved in parallel using GPU-algorithms for rendering and modeling operations. The GPU-accelerated multi-level voxelization method can generate models with an effective voxel count of up to 8 billion voxels. In addition, the GPU voxelization algorithm is more than 40x faster than the CPU implementation in generating the voxelization. Finally, we outline a few applications for the hybrid representation, which include fast point-membership classification, volume computation, and collision detection.

CHAPTER 1. INTRODUCTION

Traditional solid modeling makes use of boundary-representation (B-rep) to define and model solid geometry [Shareef and Yagel, 1995]. Even though B-rep is ideally suited for rendering using rasterization, performing modeling operations such as Boolean operations or computing point membership classification require complex geometric algorithms [Requicha and Voelcker, 1985]. In addition, several topological checks need to be performed with the B-rep data structure to ensure the validity of the B-rep solid model [Bajaj and Schikore, 1996]. Volume-based representations, such as voxel representation or distance fields, provide an alternative to B-rep to represent the solid model [Shareef and Yagel, 1995]. However, converting a B-rep to a voxel representation is computationally intensive. In addition, converting the voxel representation back to B-rep may lead to many small triangles that have to be edited to get a B-rep that is close to the original solid model [Tsuzuki et al., 2007]. In this thesis, we propose a hybrid B-rep and voxel representation, where both representations are stored simultaneously. This allows for the use of the representation that is better suited for a particular application. However, storing both representations require the fast regeneration of the voxelized model after any change to the B-rep model. We have developed a multi-level voxelization algorithm that is accelerated by the GPU to perform this compute-intensive operation.

Another requirement for an integrated representation is the voxel representation needs to be fine enough to capture small details in the B-rep solid model. However, an uniformly fine voxelized representation necessitates a large memory usage, making it impractical for modeling applications. Traditional CPU-based volume representations make use of hierarchical data structures such as octree [Grimm et al., 2004] to perform selective refinement around the boundary of the solid model and ultimately reduce memory usage. These octree data structures are usually generated using recursive algorithms. However, these recursive algorithms are not



(a) *B-rep model* **(b)** *Voxelization* **(c)** *Multi-level voxelization*

(d) *Volume rendering* **(e)** *Surface rendering*

Figure 1.1: *Multi-level voxelization and direct rendering of voxel models.*

suited for efficient parallelization using the SIMD GPU architecture. To make the hierarchical data-structure efficient for GPU-accelerated algorithms, we make use of a multi-level voxelization. Each level in our multi-level voxelization consists of an user-defined number of voxels in each direction ($\mathcal{O}(10)$ per direction) that is significantly higher than the 2 voxels per direction in an octree data structure. This allows for a fine voxelization of the solid model with only two levels for many practical applications.

Traditional voxel representations that are directly generated using CT scans store a density value for each voxel location indicating the density of the actual object. However, B-rep solid models do not have any volumetric information and hence can be assumed to be of uniform density. This can be exploited in generating a voxelized model by using a single binary digit to represent the occupancy of material in a voxel. This reduces the memory usage of the voxelized representation and can lead to large dense voxelized representations. This was exploited previously by researchers to encode the voxel grids using a compact binary

representation [Dong et al., 2004; Zhang et al., 2007; Eisemann and Décoret, 2008]. In this thesis, we extend the voxelization to store not only the occupancy information, but also encode the pointers to the original B-rep surfaces and triangles in the voxel. This allows for the use of the voxelized representation to store additional information such as surface normals, which can be used to create renderings of the model that are close to the raster rendering of the original B-rep.

With the advent of direct GPU programming capability (CUDA and OpenCL), newer voxelization methods make use of data parallel algorithms, which take advantage of direct GPU memory write access to perform the voxelization. This allows the generation of an octree voxelized representation on the GPU by first voxelizing only the surface of a model and then use occupancy propagation methods to fill the interior [Schwarz and Seidel, 2010]. However, these methods do not take advantage of the fast hardware rasterizer on the GPU that provides efficient ray-triangle intersection tests. There has been recent research on developing an hardware pipeline for voxelization [Pantaleoni, 2011], but it requires custom access to the GPU hardware. Our method integrates the data-parallel approaches while also utilizing the hardware rasterizer to perform the ray-triangle intersection test by directly rendering the triangles and making use of the stencil buffer to count the intersections. This allows the better utilization of the GPU hardware in performing the voxelization. In addition, we extend this method to efficiently perform multi-level voxelization that is better suited for the rendering pipeline, than using traditional octree representation.

The multi-level voxelization can be used to accelerate several solid modeling operations that can be computationally inefficient with B-reps. One such computation is point-membership classification (PMC), which identifies whether a given point in 3D space is inside an object or not. Traditionally this is an $\mathcal{O}(n)$ operation, where n is the number of triangles. This is further complicated in B-reps made up of spline surfaces. Using the multi-level voxelization with our flat data structures reduces it to a $\mathcal{O}(1)$ operation, with $2 * k - 1$ address lookups for level k voxelization. Another application of multi-level voxelization is fast collision detection. The voxels can be considered as oriented bounding boxes (OBBs) to perform intersection tests. These tests are data parallel and hence, can also be accelerated using the GPU.

In this thesis, we have developed GPU-accelerated methods to create and render a multi-level voxelization of a solid model, which can be simultaneously used with the existing boundary-representation for solid modeling operations. Our main contributions include:

- A GPU-accelerated method to create a multi-level voxelization given a boundary representation of a solid model. This method can generate a fine voxelization of both the boundary and the interior of the solid model more than 100x faster than existing CPU-based ray intersection methods.
- Flat storage data structures that can be used to efficiently store the multi-level voxelization data on the GPU for efficient retrieval for solid modeling applications.
- A GPU-accelerated ray casting method to render the multi-level voxelization of the solid model. This method can perform volume rendering of solid models at more than 30fps in full HD resolution (1920×1080).
- Augmented voxel data structure to store the surface normals of solid models. The surface normals can be used to interactively render the surfaces of the voxelized model with realistic lighting.
- Application of the multi-level voxelization for fast point-membership classification and GPU-accelerated collision detection.

This thesis is arranged as follows. We briefly describe some of the background and other work related to voxelization and ray casting in Chapter 2. We then explain our GPU-accelerated algorithm to construct the multi-level voxelization in Chapter 3. We describe our GPU-based ray-casting approach to render the volume and surface of a solid model in Chapter 4. We outline some of the solid modeling operations that can be accelerated by using the multi-level voxelization in Chapter 6. Finally, we provide the computational timing results for generating and rendering the multi-level voxelization in Chapter 5.

CHAPTER 2. BACKGROUND AND RELATED WORK

Voxelized models are more effective than B-reps in representing volume data. For example, using a voxel-based representation for 3D Magnetic Resonance Image (MRI) scans, will allow the information about internal structures to be maintained [Gibson, 1995]. Since the voxel representation contains information about the interior of the solid model, they can be modified for simulation applications to encode the material properties of the volume element. This is especially useful in using CT or MR images to construct voxel models that can be directly used in simulations [Rank et al., 2012].

In addition to the density information, the voxelized representation can be extended to hold multiple elements of data. An occupancy map is used to organize the voxels [Gibson, 1995]; it consists of a collection of pointers that have memory address location of a voxel location. These addresses will either be a null pointer or a voxel data structure. The voxel data structure will hold the information about the solid model, for example, color, material, opacity, etc. To initialize an occupancy map, the voxel addresses are stored based on the voxel positions relative to the object space. In our implementation we make use of a flat data structure to store the occupancy map. This data structure consists of unrolling the 3D voxels first along the x-direction, followed by y- and z-direction, sequentially.

Rasterization can be used for volume rendering of voxel models; however, the voxels must be rendered in back-to-front order with transparencies correctly applied. Li et al. [2003] make use of a volumetric partitioning method, where objects are stored in separate convex volumes, and the final image can be made by compositing these objects. However, care must be taken in rasterization-based methods to prevent z-fighting of adjacent voxels while rendering.

One of the more common ways to visualize voxelized data is to use ray-casting. Parallel rays along each rendering pixel location are sent from the camera towards the bounding-box

of object and intersections with the bounding-box are tracked using the parametric values of the ray [Marques et al., 2009]. For rendering voxelized representations, the voxels are sampled periodically along the ray and composited to generate the final pixel color. Ray-casting can be implemented with the GPU due to the data parallel nature of the ray-casting algorithm. In addition, the dataset can be stored in a 3D texture that allows for smooth interpolation of density values [Hadwiger et al., 2005]. However, using 3D textures for multi-level voxelization becomes tedious since a new 3D texture needs to be generated for each of the fine level voxels. In addition, there exists a limit on the maximum number of textures that can be simultaneously bound on the GPU ($\mathcal{O}(32)$, that limits the voxelization. Hence, we make use of a flat data structure for the voxelization.

Certain physical interactions between solid models such as collision detection can be performed efficiently using voxelized representations. There have been previous research on using GPU-accelerated algorithms for handling collision detection in voxelized models [Chen et al., 2004]. We show an application of using our multi-level voxelization to perform collision detection. In our approach the collision detection is reduced to performing overlap between oriented bounding-boxes (OBBs) that can be efficiently performed in parallel using the GPU.

Point membership classification (PMC) consists of classifying whether a point in 3D space lies inside or outside an object. PMC is considered as one of the core modeling operations in solid modeling [Requicha and Rossignac, 1992]. One of the standard PMC tests for B-rep consists of counting the number of intersections between a ray starting from the PMC test point and the object boundary [Tilove, 1980]. We make use of the PMC test to classify each voxel to be inside or outside the object. Once the voxelized model is generated, it can be used to accelerate the PMC test of the object by testing any new point against the voxelization instead of the B-rep model.

CHAPTER 3. CONSTRUCTING MULTI-LEVEL VOXELIZATION

In this chapter, we describe our new GPU-accelerated algorithm to create a multi-level voxelization of a B-rep solid model. Figure 3.1 shows an overview of the algorithm for creating the multi-level voxelization. In our implementation, the B-rep solid model is first finely tessellated to a set of triangles using a solid modeling kernel (such as ACIS). However, the method can be extended to B-reps consisting of spline surfaces (such as NURBS).

Algorithm Overview

To create a multi-level voxelization, we construct a grid of voxels in the region occupied by the object. We then make use of a rendering-based approach to classify the voxel centers as being inside or outside the B-rep solid model. To generate the fine level voxelization, we first make use of the tessellation to classify the coarse voxels that are on the boundary of the solid model. Once the boundary voxels are identified, we create an index array that identifies the memory location of each coarse level boundary voxel in the fine level voxel grid. We then make use of the same rendering approach to classify the grid of fine level voxels by rendering the clipped model inside each boundary voxel. Using this method on the GPU, a fine voxelization of the model (up to 1 billion voxels) with a relative voxel size of 0.001, can be generated (Figure 1.1(c)). This resolution is fine enough to perform modeling operations with the voxelization. Using the triangles of the B-rep model that intersect with the fine-level voxels, a surface normal can be encoded into each voxel. This normal information can then be used to calculate lighting while rendering the surface of the solid model using ray-casting.

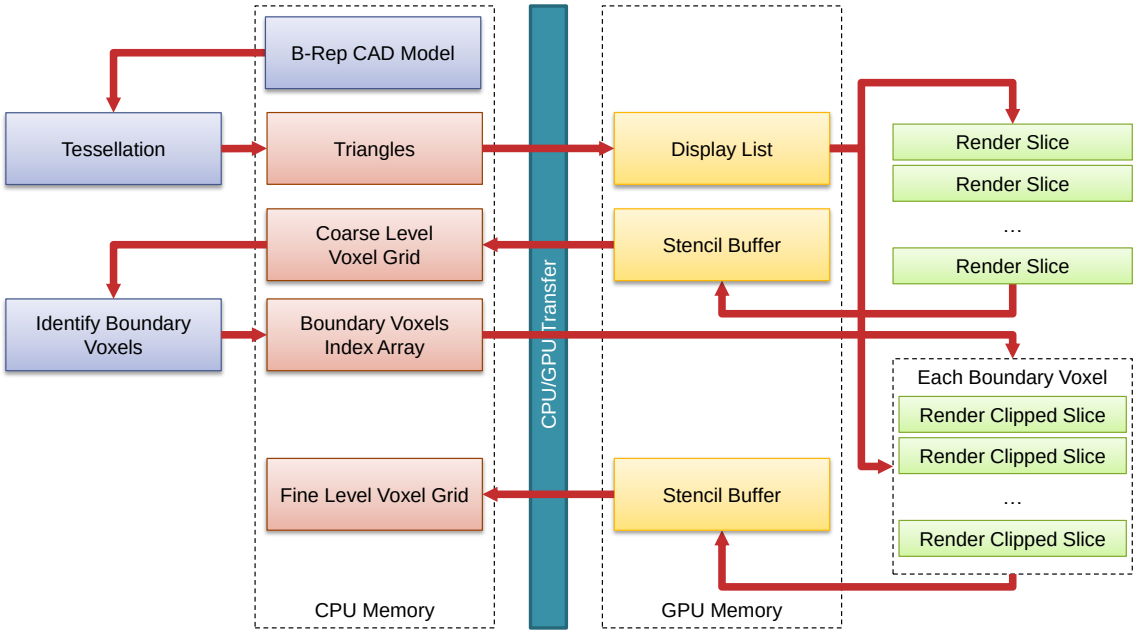


Figure 3.1: *Overview of the multi-level voxelization algorithm.*

Coarse-Level Voxelization

To generate the coarse level voxelization, we first divide the axis-aligned bounding-box (AABB) of the solid model into a regular grid of voxels. The voxels in our implementation are axis-aligned bounding-boxes themselves, without any restriction on them being cubes, rectangular boxes can be used as well. This allows for a better voxelization of models with skewed aspect ratios, without compromising on the accuracy of the voxelized representation.

A 2D example of performing voxelization using GPU rendering is shown in Figure 3.2; the method directly extends to 3D. The tessellated CAD model is transfered to the GPU using display lists. The CAD model is then rendered slice-by-slice by clipping it while rendering. Each pixel of this clipped model is then used to classify the voxel corresponding to the slice as being inside or outside the CAD model. This is performed by counting the number of fragments that are rendered in each pixel using the stencil buffer on the GPU. After the clipped model is rendered, an odd value in the stencil buffer indicates that the voxel on the particular slice is inside the CAD model, and vice versa. The process is then repeated by clipping the model with a plane that is offset by the voxel size along the view direction. Without loss of generality, the z-direction is chosen as the view direction in our implementation. We can get the coarse

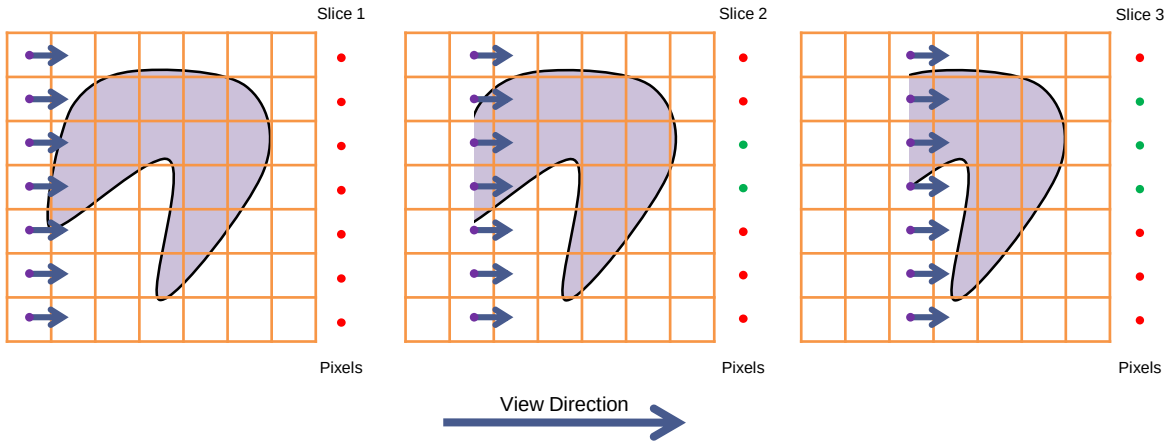


Figure 3.2: *Performing voxelization in 2D using GPU rendering. A clipped CAD model is rendered slice-by-slice and the number of rendered pixels is counted. The pixels that are rendered an odd number of times in each slice are inside the object.*

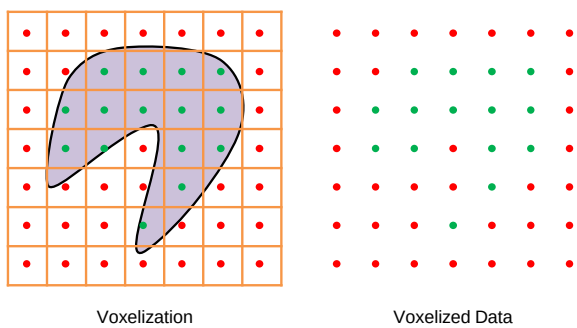


Figure 3.3: *Coarse voxelization after performing the inside-outside test for all slices. The voxel centers that are inside the solid model are marked in green.*

voxelization of the CAD model once all the slices are rendered and the voxel centers have been classified (Figure 3.3).

Identifying Boundary voxels

Identifying the boundary voxels of the coarse voxelization is performed by identifying the voxels that intersect with the triangles of the B-rep model. Since a valid B-rep model does not contain any triangles in the interior, if a voxel intersects with a triangle then the voxel contains a part of one of the boundary surfaces of the solid model. In our multi-level voxelization, we identify the boundary voxels for two specific reasons. First, the boundary voxels need to be identified in order to generate the fine level voxel grid only inside the boundary voxels.

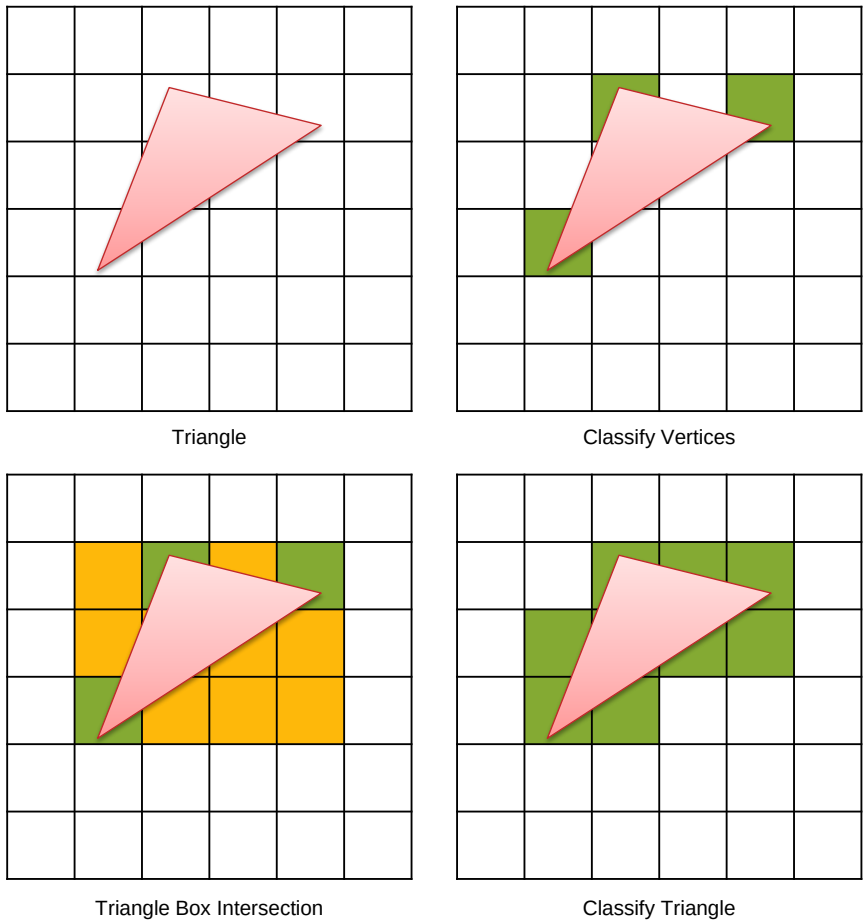


Figure 3.4: *Identifying the boundary voxels in 2D. The voxels in which the vertices of the triangle lie are first identified. The triangle is then intersected with the AABBs within the bounds along the 2 directions (marked in yellow) to identify all boundary voxels (green).*

This allows for a higher resolution voxelization without exponentially increasing the total voxel count. In addition, once the boundary voxels are identified, the average surface normals of the triangles that intersect with the voxel can be embedded into that voxel. This allows for realistic lighting calculation using only the voxelization to generate a surface rendering of the model using ray casting.

To identify the boundary voxels, we loop through every face and in turn every triangle of that face; we then find the voxels the triangle intersects with. To speed up this operation, we first identify the voxels that contain the triangle vertices. By first identifying the voxels that contain a vertex of the triangle, we can cull the voxels that need not be tested for intersections with the triangle. This is because once we know the boundaries of the triangle, we do not need

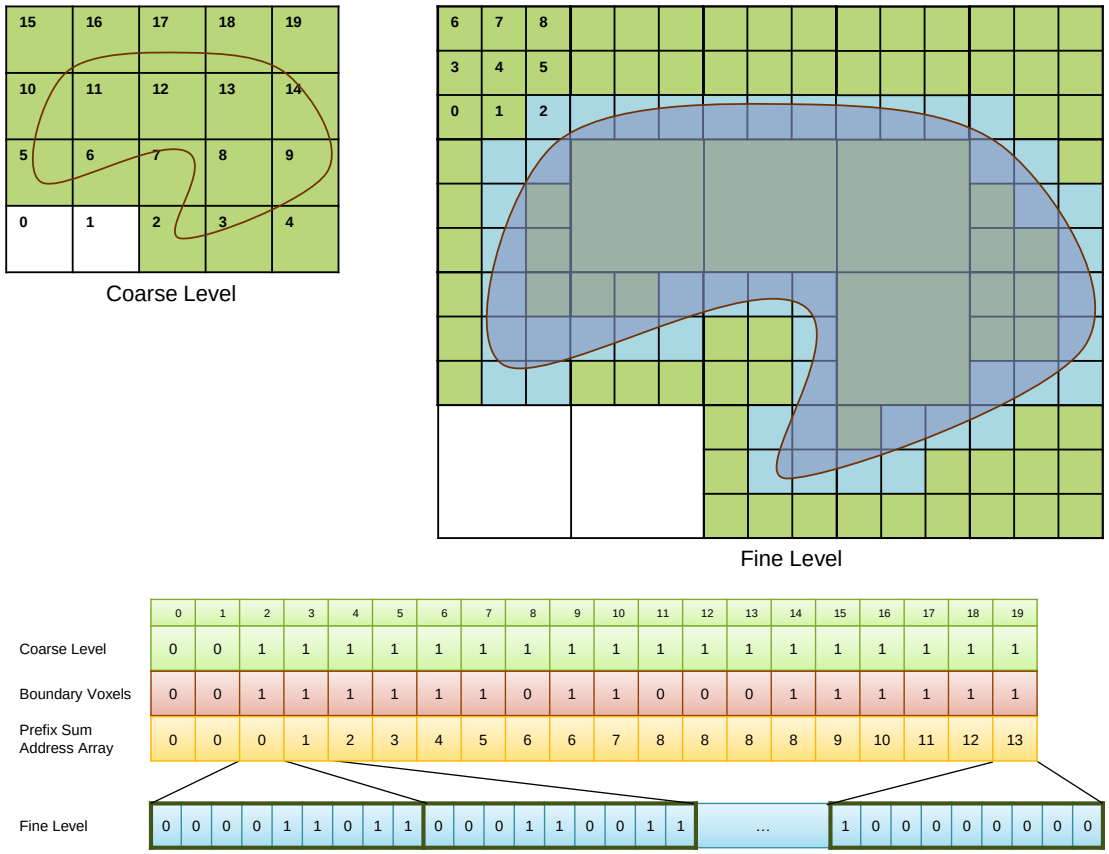


Figure 3.5: *Data structures for the multi-level voxelization.*

to test any of the voxels that lie outside the boundaries as shown in Figure 3.4. The index of the voxel that the vertex lies in can be calculated using the position of the vertex relative to the boundaries of the AABB of the object using

$$i = [N_x (x_p - x_{min})/(x_{max} - x_{min})], \tag{3.1}$$

$$j = [N_y (y_p - y_{min})/(y_{max} - y_{min})], \tag{3.2}$$

$$k = [N_z (z_p - z_{min})/(z_{max} - z_{min})], \tag{3.3}$$

$$v_{array} = k\, N_y\, N_x + j\, N_x + i, \tag{3.4}$$

where N contains the dimensions of the voxel grid generated to encapsulate the model, $*_{min}$ and $*_{max}$ corresponds to the minimum and maximum corner of the AABB, and the v_{array} is the index in the global array of the coarse voxelization. We then perform an intersection test with all

the voxels that lie within the bounds and the triangle using the separating-axis test [Gottschalk et al., 1996]. The rendering-based voxelization has already classified the voxel as being inside or outside the model, and hence, once the voxel has been identified as a boundary voxel we can change the value in the voxelization to indicate it as a boundary voxel. For illustration purposes, the boundary voxels are shown as a separate array in Figure 3.5; in practice, we use the same array but indicate boundary voxels using a different integer, say 2.

Once all the boundary voxels are identified, we make use of an exclusive prefix-sum array [Blelloch, 1990] to keep track of the address of the fine level voxelization. The size of the prefix sum array will be the same as the coarse level voxelization (see Figure 3.5). The prefix sum array will be referenced later when performing the fine level voxelization.

Fine-Level Voxelization

The fine level voxelization is implementing another level of voxelization inside the boundary voxels of the first coarse level. Fine level voxelization allows creating a higher-resolution voxel model without exponentially increasing the number of voxels. Voxelizing the fine level is performed using the same method as the coarse level. However the AABB of the boundary voxels are used to clip the model in the fine level voxelization except along the view direction (as shown in Figure 3.6). Along the view direction, the model is clipped in the back similar to the coarse voxelization using the plane that passes through the voxel centers that are being classified. However, the model is not clipped in the front in order to correctly count the boundary fragments that are rendered.

After classifying the voxel centers in the boundary voxel, the classification result along with any additional information about the voxel (such as coordinates, surface normals, etc.) need to be stored in a flat array data structure for better and faster retrieval on the GPU. However, since the number of boundary voxels will vary depending on the model and the coarse level voxelization, the size of the fine level voxelization is not constant. To keep track of the address locations of the boundary voxels in the fine level array, we make use of the exclusive prefix sum. The prefix sum array keeps track of the boundary voxels in the coarse level voxelization. In our implementation, all the boundary voxels are divided into the same number of user-defined

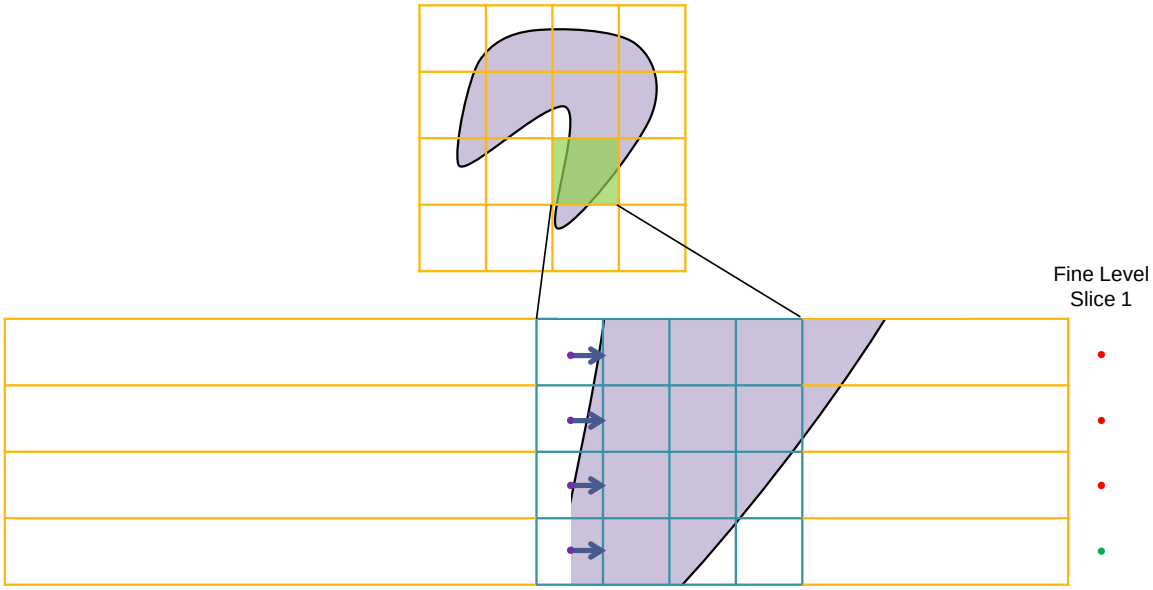


Figure 3.6: *Performing fine level voxelization in 2D using GPU rendering. The boundary voxel (shaded green) is further refined into a fine level voxel grid. The model is clipped in all directions except along the view direction to count the intersections using rendering.*

fine level voxels, and hence, using the prefix sum address array, we can directly access the memory location of the fine level voxelization. An example of this operation in 2D is shown in Figure 3.6.

After the fine level voxels have been classified as inside or outside the B-rep solid model, the boundary voxels in the fine level voxelization can also be identified. Classifying the boundary voxels for the fine level uses the same procedure as the coarse level and uses the tessellation of the B-rep. However, it is faster than the coarse level since instead of testing all the triangles in every face, we just need to test the triangles that have already been classified as intersecting with the coarse level voxel. We test the triangles that intersect with the coarse level voxels for intersection with the fine level voxels.

Once the boundary voxels in the fine level have been identified, we again store the average surface normal of all the triangles that intersect with the voxel. This surface normal is then used while rendering the voxelization. We can also directly render the voxels as wireframe to check for errors in voxelization. We assign different colors to the fine and coarse level voxels, and also to inside and boundary voxels as shown in Figure 1.1(c). The resolution in the image is set lower to better illustrate the differences between the coarse and fine level voxels.

CHAPTER 4. RENDERING MULTI-LEVEL VOXELIZATION

We render the multi-level voxelization using a ray-casting method implemented on the GPU. We can generate two kinds of rending, volume and surface rendering. Volume rendering is performed by sampling which voxels are inside or outside the model along a particular ray. The total of the number of voxels sampled is then used to calculate the color of the ray. Surface rendering, on the other hand, stops when a boundary voxel has been sampled along the ray; the relevant surface normal data of the voxel is then used to compute lighting calculations that assign a color to the ray to be rendered. The resulting image from the surface rendering is similar to the rasterized rendering of the B-rep solid model.

To perform the ray-casting, the voxel data for the classification of the inside voxels for the coarse and the fine voxels are used. In addition, the average surface normals for the coarse and fine level boundary voxels is also used for the surface rendering. We store these data in respective flat arrays on the GPU. We use flat arrays to store the data instead of 3D textures, since a 3D texture would be required for each boundary voxel while rendering. Due to restrictions on the maximum number of textures that can be bound while rendering, using 3D textures for multi-level voxelization is not feasible. By using a flat array, a single array can be used to store the data for all the fine level voxels. However, if only the coarse level voxels need to be rendered, we have another implementation that makes use of 3D textures. One of the main advantages of 3D textures over flat arrays, is that it is possible to perform fast linear interpolation with 3D textures, which provides a better rendering of smooth surfaces.

Volume Rendering

Volume rendering is displaying a 2D projection of a 3D data set [Kruger and Westermann, 2003]. The data set in our case is the multi-level voxelized model that was generated earlier.

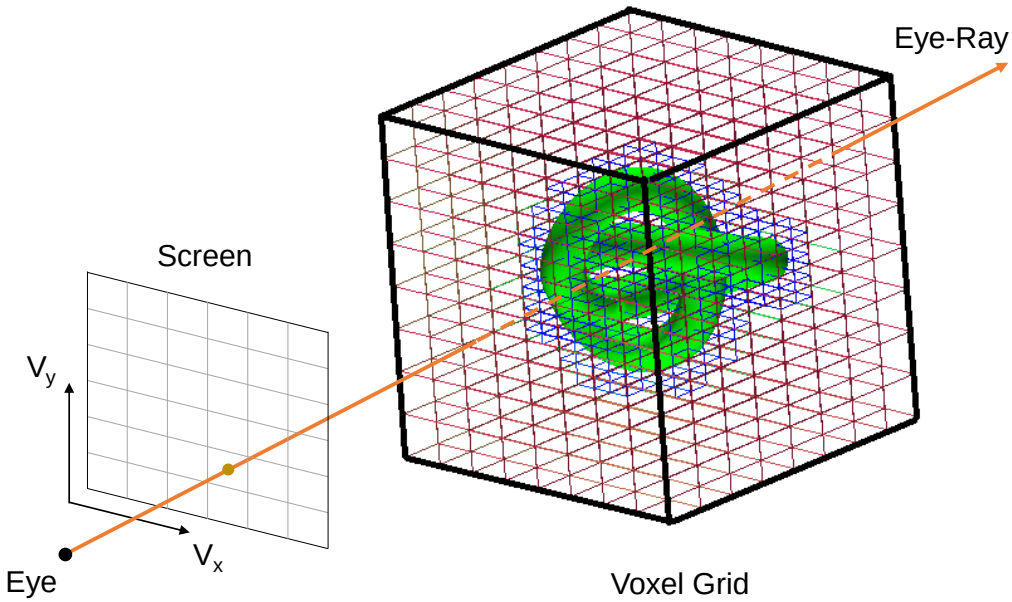


Figure 4.1: *Using ray casting to render the multi-level voxelization.*

Volume rendering is performed in our case by implementing a ray-casting algorithm on the GPU, where each pixel on the screen corresponds to a single ray [Kruger and Westermann, 2003].

Each ray is first tested for intersection with the AABB of the complete voxelization. The AABB of the voxelization is mapped to a canonical volume $[-1, 1] \times [-1, 1] \times [-1, 1]$. If the ray intersects with the AABB, the entry and exit points along the ray parameter are calculated. This ray is then marched from the entry to the exit point at uniform intervals while keeping track of the data from all the intersected voxels. Using the ray's origin and direction along the near and far intersections from a Ray-Box intersection, we can get the current position of the ray along the parameter t (Equation 4.1). Rays that don't intersect with the AABB can be culled since they will never intersect with the model.

$$\mathbf{p} = \mathbf{O} + t\,\mathbf{D} \tag{4.1}$$

The current ray position is then used to get the i, j, and k grid positions for the voxel in

intersects using

$$
\begin{aligned}
i &= (p_x + 1)/2 * N_x, \\
j &= (p_y + 1)/2 * N_y, \\
k &= (p_z + 1)/2 * N_z,
\end{aligned}
\tag{4.2}
$$

where $\mathbf{p}$ is the ray position and N is the x, y, and z dimensions of the voxel grid. These indexes are then used to get the location (v) in the data arrays that have the associated information for that voxel;

$$
v = k\, N_y\, N_x + j\, N_x + i. \tag{4.3}
$$

In the multi-level voxel model, the boundary voxels have a value different than an inside or outside voxel in the coarse level array. When a boundary voxel is sampled, the fine level voxel array is also sampled as the ray marches through the voxelization. The prefix sum address array is used to identify the location of the fine level voxelization corresponding to the current boundary voxel in the fine level voxelization array. The index in the prefix sum address array for the boundary voxel is then multiplied by the fine level voxelization resolution. The ray-position is also calculated relative to the AABB of the boundary voxel instead of the AABB of the model. This is necessary to get the correct index for the fine level voxelization inside the given boundary voxel.

$$
\begin{aligned}
i_2 &= [(p_x + 1)/2]\, N_x\, N_{2x} - i\, N_{2x}, \\
j_2 &= [(p_y + 1)/2]\, N_y\, N_{2y} - j\, N_{2y}, \\
k_2 &= [(p_z + 1)/2]\, N_z\, N_{2z} - k\, N_{2z},
\end{aligned}
\tag{4.4}
$$

where N_2 contains the dimensions of the fine level voxel grids, and i, j, k are the grid positions for the current boundary voxel. The index for the fine level voxel can be calculated in a similar manner to the coarse level, but with the fine level ray position and the fine level voxel dimensions. To get the position in the array containing all the fine level data, the value from the prefix sum array, PS, is combined with fine level index,

$$
V_2 = PS_v\, N_{2x}\, N_{2y}\, N_{2z} + k_2\, N_{2y}\, N_{2x} + j_2\, N_{2x} + i_2. \tag{4.5}
$$

Surface Rendering

Surface rendering can generate a realistic image of the model's surface compared to volume rendering. In our method, the GPU-accelerated surface rendering is implemented in a similar manner to the volume rendering. The same ray-casting method is used, but in addition to the voxel occupancy information, the surface normals of the boundary voxels are also used. Using the surface normals allows performing realistic lighting calculations on the GPU. When rendering the surfaces using the multi-level voxelization the coarse level surface normals are not used; only the fine level surface normals are used. The coarse level occupancy information is still used to identify the boundary voxels. The index in the array for the fine level surface normals is the same as the index (V_2) for the fine level occupancy array. The occupancy information is used to cull the voxels that need not be rendered. This is performed because only the boundary voxels contain the normal information and hence have to be rendered.

To calculate the color of each ray that intersects the boundary voxel using lighting calculations, the sum of the different lighting components ($C_{ambient}$, $C_{diffuse}$, $C_{specular}$, and $C_{emission}$) is calculated. $C_{ambient}$ is light that comes from all directions equally. $C_{diffuse}$ is light that comes from a point source and hits surfaces with an intensity that depends on whether or not the surface faces the light; the light radiates from the surface equally for this property. $C_{specular}$ is light from a point source that is reflected like a mirror and the light bounces in the direction defined by the surface shape. $C_{emission}$ is light emitted by the object. Note that all lighting components are 3-element vectors corresponding to the red, green, blue color values. The ray color can be calculated using

$$C = C_{ambient} + C_{diffuse} + C_{specular} + C_{emission}. \tag{4.6}$$

The ambient teflectivity coefficent (k_a) controls the amount of ambient light reflected from the surface,

$$C_{ambient} = k_a * G_{ambient}, \tag{4.7}$$

where $G_{ambient}$ refers to the value of the ambient light itself. $C_{diffuse}$ is calculated from the diffuse reflectivity coefficient (k_d), the light color (C_{light}), and the dot product of the light

direction($\mathbf{L}$) and surface normal ($\mathbf{n}$),

$$C_{diffuse} = k_d \, C_{light} \, (\mathbf{n} \cdot \mathbf{L}) \, . \tag{4.8}$$

$C_{specular}$ is calculated from the specular reflectivity coefficient (k_s), the light color, the shininess coefficient (f), and the dot product of the view vector ($\mathbf{v}$) and direction of perfect mirror-like reflection ($\mathbf{r}$).

$$C_{specular} = k_s \, C_{light} \, (\mathbf{v} \cdot \mathbf{r})^f \, . \tag{4.9}$$

$C_{emission}$ is a constant based on whether the model should be emitting light or not. The surface constants are chosen for each model to give a realistic surface appearance; an example of the surface rendering is shown in Figure 1.1(e). One limitation of our implementation of the surface rendering is that the object is made up of a single solid color. This can be extended in the future by storing the B-rep surface color along with the normals in the boundary voxels.

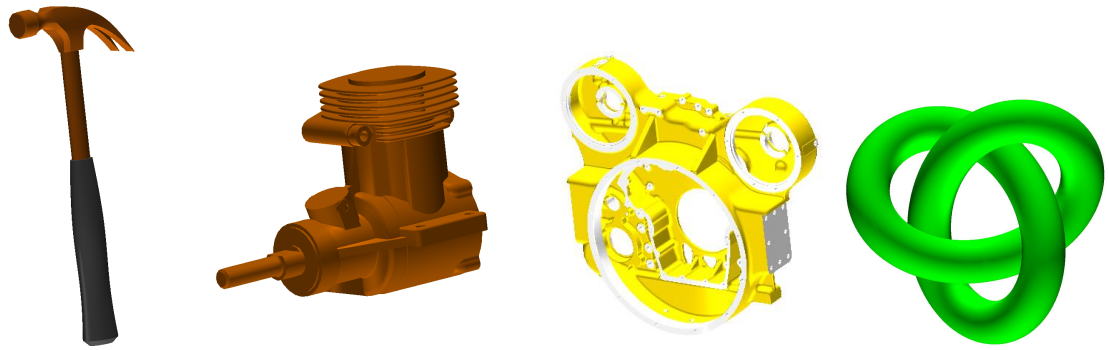
CHAPTER 5. TIMING RESULTS

To test our new GPU-accelerated voxelization and rendering methods we used four different models, Hammer (Figure 5.1(a)), Engine (Figure 5.1(b)), Engine Cover (Figure 5.1(c)), and Trefoil (Figure 5.1(d)). These models were chosen based on their complexity; the Hammer, Engine, and Engine Cover models correspond to models with low, medium, and high triangle counts, respectively. The Trefoil model was used to assess the performance of methods in a complex model with overlapping inside and outside regions along a given view direction. The models with different triangle counts can be used assess the impact of the size of the tessellation on the voxelization. For the Trefoil model, we assessed the accuracy of the voxelization and the appearance of the volume and surface renderings. We varied the resolution of the coarse level voxelization and the fine level voxelization separately to study the effect of each level on the computation time. In addition, we measured the frame rates achieved during volume and surface rendering of the different voxelized models in full HD (1920x1080) screen resolution. The timings were measured in a workstation with Intel Xeon CPU E5-2630 v3 @2.40GHz, 32 GB RAM, and nVIDIA GeForce GTX Titax X GPU with 12 GB GPU-RAM

Voxelization Timings

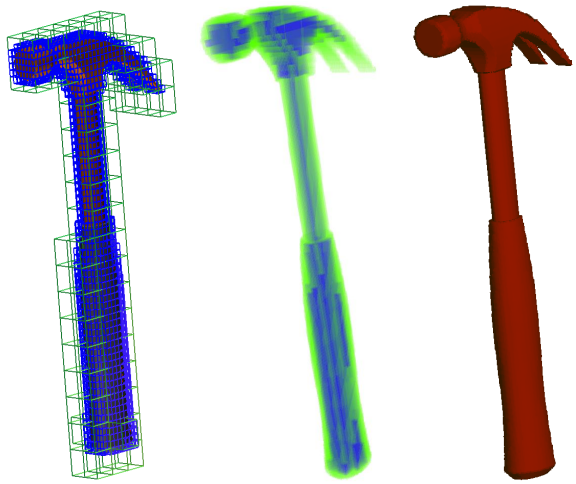
The voxelization timings were separated between the coarse and the fine levels, to compare the time for computing the different levels. Table 5.2 and Table 5.2 gives a comprehensive listing of all the timing results. It can be seen that computing the fine level voxelization takes the largest percentage of the total time. This is expected since the voxelization and identification of boundary voxels is repeated for every boundary voxel in the coarse level.

The time taken for 2-level voxelization can also be compared with its 1-level equivalent. For example, looking at the maximum number of a voxels along an axis, having 20 voxels



(a) *Hammer* **(b)** *Engine* **(c)** *Engine Cover* **(d)** *Trefoil*

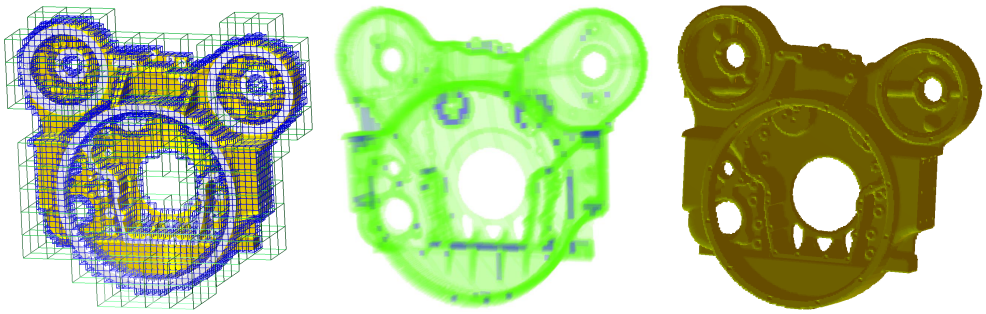
Figure 5.1: *Models used for testing the multi-level voxelization.*



(a) *Voxelization* **(b)** *Volume rendering* **(c)** *Surface rendering*

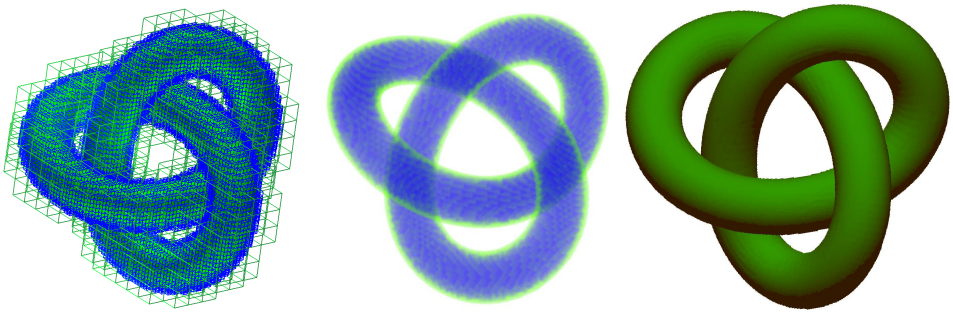
Figure 5.2: *Hammer Model.*

along an axis for the coarse level with 10 voxels along an axis for the fine level will have the same resolution as 200 voxels along the longest axis for a 1-level voxelization. The time for just the 1-level voxelization is significantly shorter than doing multiple levels. Because of this time difference, whether a multi-level voxelization is used over a single-level voxelization will depend on the application. The multi-level voxelization can achieve higher resolutions than a single-level, since the highest single-level voxel count along an axis is 300 in our testing, due to memory requirements of the voxel data structure. However, the multi-level voxelization can easily reach >1,000 equivalent single-level voxels along an axis within the CPU and GPU memory limits. As a result, using the multi-level voxelization is the only option if a higher



(a) *Voxelization* **(b)** *Volume rendering* **(c)** *Surface rendering*

Figure 5.3: *Engine Cover Model.*



(a) *Voxelization* **(b)** *Volume rendering* **(c)** *Surface rendering*

Figure 5.4: *Trefoil Model.*

resolution is desired for a particular model.

We also implemented a CPU version of the voxelization method using the Möller-Trumbore ray intersection algorithm [Möller and Trumbore, 2005]. We then tested the time taken for the CPU to voxelize the same models. It can be seen from Table 5.1 that except for the smallest resolutions, the GPU voxelization algorithm is $> 40\times$ faster than the CPU implementation. In addition, the GPU voxelization takes $< 150ms$ for intermediate resolutions, making it suitable for interactive updates in a CAD system.

Rendering Frame Rates

We measured the frame rates while interactively rendering the multi-level voxelization using GPU ray casting. The main objective is to maintain a frame rate of at least 30 frames-per-second (fps) in full HD screen resolution (1920x1080). The frame-rates achieved during surface rendering were all fairly consistent at around 61 fps. The Engine Cover model was the only

Table 5.1: *GPU and CPU time for performing a single coarse level voxelization. It can be seen that the GPU based voxelization is more than 40x faster than the CPU in most cases.*

Voxel Grid	Effective Voxels	GPU Time (s)	CPU Time (s)	Speedup
Hammer				
12x20x8	122,880	0.012	0.161	14×
48x100x28	134,400	0.024	11.181	466×
100x200x52	1,040,000	0.13	85.393	658×
Engine				
12x16x20	245,760	0.043	1.874	43×
64x76x100	486,400	0.202	234.485	1,161×
124x156x200	3,868,800	0.747	1,839.420	2,463×
Engine Cover				
20x20x8	204,800	0.063	32.632	518×
100x96x40	384,000	0.939	4,746	5,054×
200x188x76	2,857,600	3.467	35,299.8	10,181×

model that had the most values under 61 fps with the lowest being 35 fps. This may be because of the significantly larger number (10M) of fine level voxels in the model.

In addition to the complexity of the model, the distance of the camera to the model has a large effect on the rendering frame rates. The further away from the model the higher the frame rate and vice versa. This could be attributed to the fact that the further away from the camera the model is, fewer rays are intersecting with the model, and hence requires fewer memory lookups on the GPU. The rays that are not intersecting with the AABB of the model are culled leading to a higher fps.

The results of our GPU-accelerated ray-casting method show that we can achieve our goal of at least 30fps at full HD for all the models at the tested resolutions. The fps were significantly higher on most models.

Table 5.2: *Timings for voxelization and rendering at different resolutions for the different models tested.*

Level 1	Level 2	Level 1 Voxels	Level 2 Voxels	Effective Voxels	Level 1 Time (s)	Level 2 Time (s)	Total Time (s)	Frame Rate (fps)
Hammer: 5,668 Triangles								
8x12x4	4x4x4	192	3,072	12,288	0.032	0.179	0.240	61
8x12x4	10x10x10	192	48,000	192,000	0.036	1.721	1.782	61
8x12x4	20x20x20	192	384,000	1,536,000	0.032	12.869	12.951	61
12x20x8	4x4x4	1,920	7,872	122,880	0.036	0.327	0.400	61
12x20x8	10x10x10	1,920	123,000	1,920,000	0.035	2.047	2.119	45
12x20x8	20x20x20	1,920	984,000	15,360,000	0.036	15.355	15.473	61
48x100x28	1x1x1	134,400	0	134,400	0.133	0.000	0.156	61
48x100x28	4x4x4	134,400	170,304	8,601,600	0.410	4.039	4.508	61
48x100x28	10x10x10	134,400	2,661,000	134,400,000	0.426	13.321	13.996	52
100x200x52	1x1x1	1,040,000	0	1,040,000	0.887	0.000	0.956	61
100x200x52	4x4x4	1,040,000	661,248	66,560,000	2.775	15.372	18.328	53
100x200x52	6x6x6	1,040,000	10,332,000	1,040,000,000	2.884	41.374	45.125	45
Engine: 28,134 Triangles								
8x12x12	4x4x4	768	10,560	49,152	0.319	1.44	1.805	61
8x12x12	10x10x10	768	165,000	768,000	0.309	9.264	9.619	61
8x12x12	20x20x20	768	556,875	2,592,000	0.320	29.583	29.983	61
12x16x20	4x4x4	3,840	31,680	245,760	0.388	3.025	3.451	56
12x16x20	10x10x10	3,840	495,000	3,840,000	0.393	14.238	14.711	61
12x16x20	20x20x20	3,840	1,670,625	12,960,000	0.388	40.756	41.358	45
64x76x100	1x1x1	486,400	0	486,400	0.723	0.000	0.776	61
64x76x100	4x4x4	486,400	1,192,128	31,129,600	12.654	88.328	101.299	40
64x76x100	10x10x10	486,400	18,627,000	486,400,000	12.510	234.382	248.379	50
124x156x200	1x1x1	3,868,800	0	3,868,800	3.710	0.000	3.953	61
124x156x200	4x4x4	3,868,800	4,752,128	247,603,200	87.262	352.715	440.831	40
124x156x200	6x6x6	3,868,800	16,038,432	835,660,800	97.059	508.369	607.207	43

Table 5.3: *Timings for voxelization and rendering at different resolutions for the different models tested.*

Level 1	Level 2	Level 1 Voxels	Level 2 Voxels	Effective Voxels	Level 1 Time (s)	Level 2 Time (s)	Total Time (s)	Frame Rate (fps)
Engine Cover: 118,921 Triangles								
12x12x4	4x4x4	576	4,736	36,864	4.820	3.797	8.689	50
12x12x4	10x10x10	576	74,000	576,000	4.639	29.674	34.383	39
12x12x4	20x20x20	576	592,000	4,608,000	4.752	228.073	232.944	48
20x20x8	4x4x4	3,200	18,688	204,800	2.080	8.203	10.393	45
20x20x8	10x10x10	3,200	292,000	3,200,000	2.144	44.444	46.683	44
20x20x8	20x20x20	3,200	2,336,000	25,600,000	2.123	269.462	271.871	45
100x96x40	1x1x1	384,000	0	384,000	0.873	0.000	0.939	61
100x96x40	4x4x4	384,000	791,232	24,576,000	26.308	273.027	299.545	36
100x96x40	10x10x10	384,000	12,363,000	384,000,000	25.708	743.213	770.021	35
200x188x76	1x1x1	2,857,600	0	2,857,600	3.251	0.000	3.462	61
200x188x76	4x4x4	2,857,600	3,314,816	182,886,400	128.443	1,060.007	1,189.130	35
200x188x76	6x6x6	2,857,600	11,187,504	617,241,600	129.02	1,620.829	1751.220	34
Trefoil: 22,318 Triangles								
12x12x12	4x4x4	1,728	13,120	110,592	0.155	0.761	0.958	65
12x12x12	10x10x10	1,728	205,000	1,728,000	0.146	6.634	6.825	62
12x12x12	20x20x20	1,728	1,640,000	13,824,000	0.145	52.044	52.322	61
20x20x20	4x4x4	8,000	43,135	512,000	0.298	1.571	1.943	63
20x20x20	10x10x10	8,000	674,000	8,000,000	0.291	9.369	10.238	61
20x20x20	20x20x20	8,000	5,392,000	64,000,000	0.280	66.309	66.979	61
100x100x100	1x1x1	1,000,000	0	1,000,000	0.948	0.000	1.024	61
100x100x100	4x4x4	1,000,000	1,145,088	64,000,000	9.299	26.380	35.931	61
100x100x100	10x10x10	1,000,000	17,892,000	1,000,000,000	9.219	76.497	87.079	61
200x200x200	1x1x1	8,000,000	0	8,000,000	6.795	0.000	7.257	61
200x200x200	4x4x4	8,000,000	4,604,928	512,000,000	73.193	94.522	168.785	50
200x200x200	6x6x6	8,000,000	71,952,000	1,728,000,000	74.391	340.763	431.958	60

CHAPTER 6. APPLICATIONS OF MULTI-LEVEL VOXEL MODELS

In this chapter, we briefly outline some of the applications of the integrated B-rep and multi-level voxel models in solid modeling. Having a fast voxelized version of the B-rep model is ideal to accelerate some of the core solid modeling operations. The applications listed here can serve as a starting point for more complex usage of the integrated representation.

Collision Detection

The voxelization can be used to perform fast collision detection between CAD models. The voxels of each model can be considered as oriented bounding-boxes (OBBs), which can be checked for overlap using the separating-axis tests. Since the voxels are also independent, the pair-wise OBB overlap tests can be performed in parallel using the GPU. Figure 6.1 shows an example of interactively highlighting the OBBs that overlap between the Hammer and the Engine model. Performing collision detection using the hybrid representation allows for the detection of the case when a particular model is completely enclosed by another model; identifying this case using only B-reps is difficult.

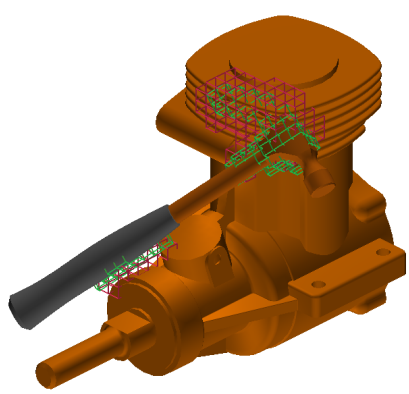


Figure 6.1: *Performing collision detection between voxelized models using OBB overlap tests.*

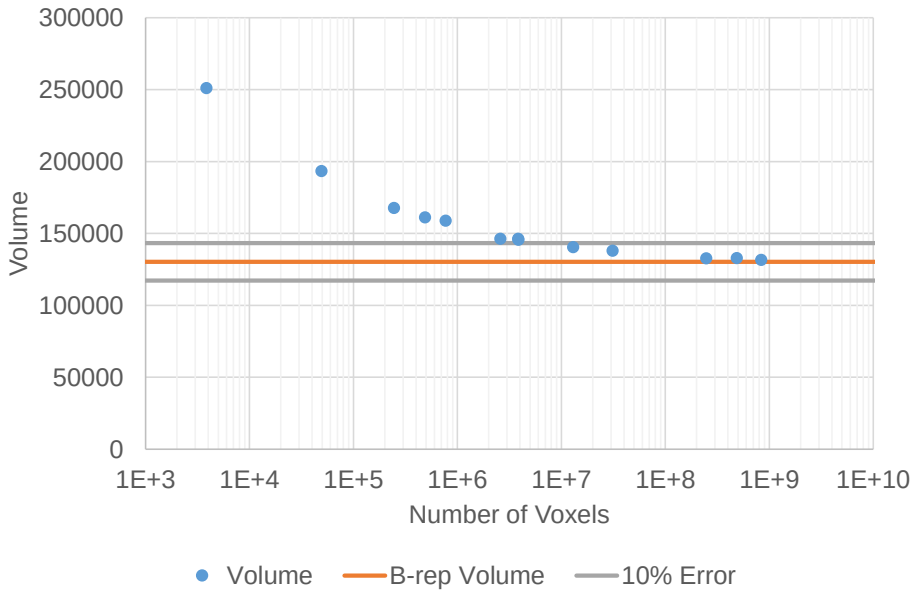


Figure 6.2: *Computing the volume of the Engine model using the voxelization gives an estimate of the error for PMC. The voxelized representation is within* 10% *of the B-rep volume with only a few million voxels.*

Point-Membership Classification

Another application of the hybrid representation is fast point membership classification (PMC) using the voxelization. Given any point in 3D space, it can be identified as belonging to inside or outside of the B-rep CAD model with $\mathcal{O}(1)$ lookups. In addition, since we also identify the boundary voxels in the hybrid representation, we can provide an error probability for the PMC based on whether the test point lies inside a boundary voxel. One way to quantify the error in our voxelized representation is to compute the volume enclosed by the voxelization. The volume of the voxelization can be easily calculated, since the size of each voxel and the number of voxels that are classified as inside the model is known. Figure 6.2 shows the volume computed with different resolutions of the voxelization for the Engine model. It can be seen that since we enclose the model completely with voxels, the voxelization volume will always be higher than the actual B-rep volume. It can be seen that the voxelization volume converges to the B-rep volume with higher resolution voxelization, and a few million voxels are enough to capture the volume to within 10% error.

CHAPTER 7. CONCLUSIONS

This thesis has proposed a new hybrid multi-level voxelization representation that can be used simultaneously with the boundary representation (B-rep) of a solid model. We have presented GPU-accelerated methods for creating the multi-level voxelization given a B-rep solid model and rendering the multi-level voxelization using ray casting. Our methods make use of the GPU rendering pipeline to accelerate the voxelization process. We have also developed flat data structures that can be used to efficiently store the multi-level voxelization along with B-rep surface information such as surface normals. The surface normals stored inside the boundary voxels can be used to generate a smooth surface rendering of the CAD model using only the multi-level voxelization.

We tested our methods on multiple models and the GPU-accelerated voxelization is $40\times$ faster than the CPU-based method for most high resolutions. This shows that this method can be used to interactively generate the voxelization while the user is editing the CAD model, leading to acceleration of some of the common solid modeling operations such as PMC, collision detection, clearance computation, etc. The GPU-accelerated ray-casting method developed to render the multi-level voxelization can achieve > 30fps for volume or surface rendering of the models, allowing for real-time interactions.

We have outlined some applications of the multi-level voxelization including fast PMC, and collision detection. These applications can be easily extended to B-reps that consist of spline surfaces. In addition, since the CAD system also contains the source feature, such as holes, extrusions, etc., that generated the B-rep surfaces, the voxel models can also encode a *Feature Link* between the CAD model features and the B-rep surfaces. The voxelization can store pointers to the surfaces that intersect with each voxel. This information along with the *Feature Link* will provide information required to trace back from the voxel the original feature in the

CAD model. This *Feature Link* is important to provide modifications to the CAD model based on the modifications to the voxelized model. This would allow for a true integrated B-rep and voxel representation for several applications such as shape editing and automatic design modifications.

BIBLIOGRAPHY

Bajaj, C. L., Schikore, D. R., 1996. Error-bounded reduction of triangle meshes with multivariate data. In: Electronic Imaging: Science & Technology. International Society for Optics and Photonics, pp. 34–45.

Blelloch, G. E., 1990. Prefix sums and their applications.

Chen, W., Wan, H., Zhang, H., Bao, H., Peng, Q., 2004. Interactive collision detection for complex and deformable models using programmable graphics hardware. In: Proceedings of the ACM Symposium on Virtual Reality Software and Technology. ACM, New York, NY, USA, pp. 10–15.

Dong, Z., Chen, W., Bao, H., Zhang, H., Peng, Q., 2004. Real-time voxelization for complex polygonal models. In: Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on. IEEE, pp. 43–50.

Eisemann, E., Décoret, X., 2008. Single-pass GPU solid voxelization for real-time applications. In: Proceedings of Graphics Interface. Canadian Information Processing Society, pp. 73–80.

Gibson, S. F. F., 1995. Beyond volume rendering: visualization, haptic exploration, and physical modeling of voxel-based objects. In: Visualization in Scientific Computing95. Springer, pp. 10–24.

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

Grimm, S., Bruckner, S., Kanitsar, A., Groller, E., 2004. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In: Volume Visualization and Graphics, 2004 IEEE Symposium on. IEEE, pp. 1–8.

Hadwiger, M., Sigg, C., Scharsach, H., Bühler, K., Gross, M., 2005. Real-time ray-casting and advanced shading of discrete isosurfaces. In: Computer Graphics Forum. Vol. 24. Wiley Online Library, pp. 303–312.

Kruger, J., Westermann, R., 2003. Acceleration techniques for gpu-based volume rendering. In: Proceedings of the 14th IEEE Visualization 2003 (VIS'03). IEEE Computer Society, p. 38.

Li, W., Mueller, K., Kaufman, A., 2003. Empty space skipping and occlusion clipping for texture-based volume rendering. In: Visualization, 2003. VIS 2003. IEEE. IEEE, pp. 317–324.

Marques, R., Santos, L. P., Leskovsky, P., Paloc, C., 2009. Gpu ray casting.

Möller, T., Trumbore, B., 2005. Fast, minimum storage ray/triangle intersection. In: ACM SIGGRAPH 2005 Courses. ACM, p. 7.

Pantaleoni, J., 2011. VoxelPipe: A programmable pipeline for 3D voxelization. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. HPG '11. ACM, New York, NY, USA, pp. 99–106.

Rank, E., Ruess, M., Kollmannsberger, S., Schillinger, D., Düster, A., 2012. Geometric modeling, isogeometric analysis and the finite cell method. Computer Methods in Applied Mechanics and Engineering 249, 104–115.

Requicha, A. A., Voelcker, H. B., 1985. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. Proceedings of the IEEE 73 (1), 30–44.

Requicha, A. A. G., Rossignac, J. R., 1992. Solid modeling and beyond. IEEE Computer Graphics Applications 12 (5), 31–44.

Schwarz, M., Seidel, H.-P., 2010. Fast parallel surface and solid voxelization on GPUs. ACM Transactions on Graphics 29 (6), 179:1–179:10.

Shareef, N., Yagel, R., 1995. Rapid previewing via volume-based solid modeling. In: Proceedings of the third ACM symposium on Solid modeling and applications. ACM, pp. 281–291.

Tilove, R. B., 1980. Set membership classification: A unified approach to geometric intersection problems. IEEE trans. Computers 29 (10), 874–883.

Tsuzuki, M. d. S. G., Takase, F. K., Garcia, M. A. S., Martins, T. d. C., 2007. Converting csg models into meshed b-rep models using euler operators and propagation based marching cubes. Journal of the Brazilian Society of Mechanical Sciences and Engineering 29 (4), 337–344.

Zhang, L., Chen, W., Ebert, D. S., Peng, Q., 2007. Conservative voxelization. The Visual Computer 23 (9), 783–792.