EXAMENSARBETE INOM TEKNIK,
GRUNDNIVÅ, 15 HP

STOCKHOLM, *SVERIGE 2019*

GPU Volume Voxelization

Exploration of the performance characteristics of different GPU-based implementations

GRIGORY GLUKHOV

ALEKSANDRA SOLTAN

KTH
SKOLAN FÖR ELEKTROTEKNIK OCH DATAVETENSKAP

GPU Volume Voxelization

Exploration of the performance characteristics of different GPU-based implementations

Aleksandra Soltan
Grigory Glukhov

Examiner Johan Montelius
Supervisor Thomas Sjöland

A thesis presented for the degree of
Bachelor of Information and Communication Technology

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
SE-100 44 Stockholm, Sweden
June 2019

page intentionally left blank

Abstract

In recent years, voxel-based modelling has seen a reintroduction to computer game development through massive graphics hardware improvements. Nevertheless, polygons continue to be the default building block of 3D objects, introducing a need for the transformation of polygon meshes into voxel-based models; this process is known as voxelization. Efficient voxelization algorithms take advantage of the flexibility and control offered by modern, programmable GPU pipelines. However, the variability in possible approaches poses the question of how different GPU-based implementations affect voxelization performance.

This thesis explores the impact of GPU-based improvements by comparing four different implementations of a solid voxelization algorithm. The implementations include a naive transition from the CPU to the GPU, a non-branching execution path approach, data pre-processing, and a combination of the two previous approaches. Benchmarking experiments run on four, standard polygonal models and three graphics cards (NVIDIA and AMD) provide runtime and memory usage data for each implementation. A comparative analysis is performed on the basis of this data to determine the performance impact of the GPU-based adjustments to the voxelization algorithm implementation.

Results indicate that the non-branching execution path approach yields clear improvements over the naive implementation, while data pre-processing has inconsistent performance and a large initial performance cost; the combination of the two improvements unsurprisingly leads to combined results. Therefore, the conclusive recommendation is using the non-branching execution path technique for GPU-based improvements.

Keywords

voxelization, GPU, GPGPU, SIMT, thread divergence, Vulkan API

Sammanfattning

Voxel-baserad modellering har på senare år blivit återintroducerat till datorspelsutveckling tack vare massiva förbättringar i grafikhårdvara. Trots detta fortsätter polygoner att vara standarden för uppbyggnaden av 3D-objekt. Detta gör det nödvändigt att kunna transformera polygonytor till voxel-baserade modeller; denna process kallas för voxelisering. Effektiva voxeliseringsalgoritmer tar vara på den flexibilitet och kontroll som ges av moderna, programmerbara GPU-pipelines. Variationen i möjliga tillvägagångssätt gör det dock intressant att veta hur olika GPU-baserade implementationer påverkar prestandan av voxeliseringen.

Denna avhandling undersöker påverkan av GPU-baserade förbättringar genom att jämföra fyra olika implementationer av en solid-voxeliseringsalgoritm. Implementationerna inkluderar en naiv övergång från CPU:n till GPU:n, en metod med en non-branching exekveringsväg, förbehandling av data, och en kombination av det två tidigare metoderna. Benchmarking-experiment görs på fyra standardpolygonmodeller och tre grafikkort (NVIDIA och AMD) förser data för exekveringstid och minnesåtgång för varje implementation. En jämförande analys görs med detta data som grund för att bestämma den påverkan som de GPU-baserade ändringarna har på prestandan av voxeliseringsalgoritmens implementation.

Resultaten indikerar att implementationen med en non-branching exekveringsväg ger klara förbättringar över den naiva implementationen, medans förbehandlingen av data presterar inkonsekvent och har en stor initial prestandakostnad; kombinationen av dem båda ledde, inte överraskande, till blandade resultat. Den slutgiltiga rekommendationen är således att använda tekniken med en non-branching exekveringsväg för GPU-baserade förbättringar.

Nyckelord

voxelization, GPU, GPGPU, SIMT, tråd divergering, Vulkan API

Acknowledgements

Special thanks go to Igor Glukhov for helping us make figures, Michael Schwarz for responding to our email regarding details of the implementation of his algorithm, Erik Bauer for translating our abstract to Swedish, as well as Johan Montelius and Thomas Sjöland for answering thesis-related questions. Additionally, we would like to thank the Stanford Computer Graphics Laboratory for publishing the 3D meshes used in our experiments. Finally, we thank Mutate for hosting this thesis for two months.

Stockholm, June 2019
Aleksandra Soltan and Grigory Glukhov

Contents

List of Figures

List of Tables

1 Introduction

For decades polygons have been the default building block of 3D models in computer graphics. However, recent claims regarding "unlimited detail" [1], improved scalability [2], and intuitive content manipulation [2] have reignited interest in voxel representation of 3D data. Voxels are discrete cubes used to construct volumetric objects; due to voxels' comparability to real world atoms, they offer a higher level of detail and greater freedom for manipulation of 3D models. Voxel model representation has long been widely used in medical imaging like CAT scans and MRIs [3, 4]; recently, its use in computer game development has accelerated, with applications such as Global Illumination [5], terrain representation [6], and pathfinding [7].

The process of transforming a polygon representation of a 3D model into a voxel-based one is called voxelization. Several notable algorithms detailing this process come from a 2010 report by Schwarz and Seidel [8]; their binary voxelization methods are the current defining work in the field, having inspired several papers proposing novel approaches to voxelization [9–13]. Schwarz and Seidel utilize the programmability offered by modern GPUs through NVIDIA's CUDA parallel computing platform, which allows far more flexibility and control over how data is computed and processed on the GPU. Breaking out of the limitations of fixed-function rasterization leads to new approaches like direct voxelization into Sparse Voxel Octrees [8, 14, 15].

Within Schwarz and Seidel's tile-based solid voxelization algorithm there are opportunities for varying GPU-based implementations, ranging from a simple, naive approach to advanced, multiple-pass techniques with data pre-processing. Therefore, this thesis proposes four different implementations in order to compare the effects of individual, GPU-based refinements on the performance of a solid voxelization algorithm with output to Sparse Voxel Octrees. Based on start-to-end running time and memory usage, final conclusions will offer an efficient GPU-implementation of a parallelized, direct voxelization algorithm.

1.1 Background

Spurred by consumer demand for increasingly realistic and immersive gaming experiences, the development of GPUs has seen some of the greatest advancements in computational technology of the past thirty years [16]. The uses for modern GPUs extend far past 3D modelling and computer graphics, to tasks that were traditionally performed by the CPU. The most powerful consumer-grade CPU today has 28 cores [17], whereas any modern GPU is composed of hundreds to thousands of cores; this allows handling of thousands of software threads simultaneously [18]. Due to the highly-parallelized structure of GPUs, their data-parallel computing power dwarfs that of traditional CPUs: for algorithms well-suited to a GPU-based implementation, studies often find speedups between 5 and 20 times the performance on even a state-of-the-art

CPU[1] [16]. GPUs are specialized to have far better performance for parallel and floating-point computations.

There are two approaches to hardware-accelerated 3D data processing: fixed-function and programmable pipelines. Both are exposed through APIs, however fixed-function access is far more limited. Traditionally, GPU programmers were provided functionality for configuring the rendering pipeline, but the way vertices were transformed or lighting was calculated was fixed. More recently, graphic cards have become better at handling general, parallel computations. New APIs like Vulkan [19] provide the ability to submit custom instructions (shaders) for handling vertex and fragment processing, allowing a more flexible approach to rendering 3D geometry. Additional functionality for using GPUs for general computation tasks (known as GPGPU) is exposed through compute shaders [20] and such APIs as OpenCL [21] and CUDA [22].

The graphics pipeline consists of a sequence of steps that transform a mathematical model into an image on a screen (see Figure 1). These steps can be divided into four sections: "vertex geometry processing and transformation, triangle processing (through rasterization) and fragment generation, texturing and lighting, and fragment-combination operations for assembling the final image" [23]. Though the introduction of programmable pipelines means shaders are replacing many stages of traditional fixed-functionality, the standard pipeline model continues to be a good overview of the steps necessary for processing and rendering 3D graphics [23].

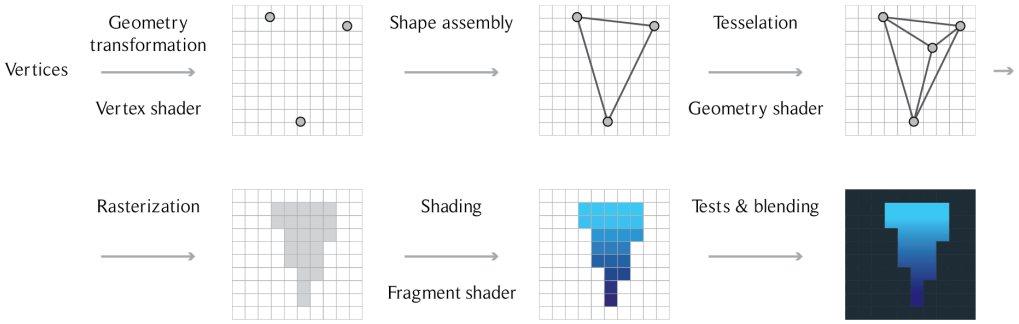


Figure 1: Graphics pipeline (courtesy of Igor Glukhov).

Voxelization is a process whereby a polygonal mesh is transformed into a voxel-based model (see Figure 2). A voxel (or "volumetric pixel") is a cubic representation of volume in a uniform grid [24], like a 3D pixel. Up until recently, voxel use was by and large limited to medical imaging and other static models; however, their 'stackability,' simple storage of volumetric data, and clear definition of the neighborhood surrounding each voxel [24] (as well as modern hardware support for large computational requirements) allow for wider applicability, especially in computer gaming. Voxels are advantageous for realistic modeling of cellular automata (such as free-form materials like smoke or clay,

[1] See section 1.6.3 for a discussion of CPU to GPU comparisons.

which benefit from representation through atomically-analogous structures) [25] and real-time interactivity and model manipulation [26] (meaning that, for example, users can burrow into a game environment by removing voxels to reveal others stacked below, whereas removal of surface polygons is not well-defined by default).

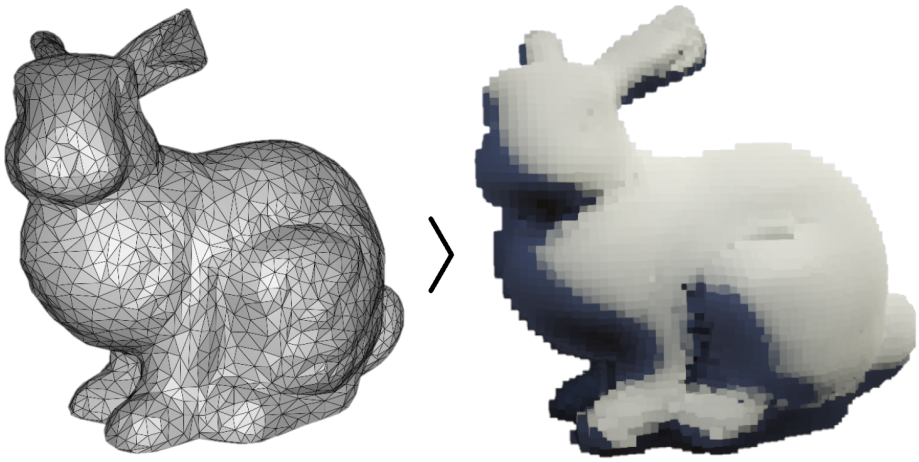


Figure 2: Voxelization input polygon mesh (left) and voxelization result (right). Input is the Stanford Bunny (courtesy of the Stanford Computer Graphics Laboratory [27]).

Voxelization can be divided into two types: surface and solid. Surface voxelization concerns transforming a continuous polygonal surface. In recent years, a range of voxelization approaches have been proposed, including tile-based rasterization [8, 9], triangle stream processing [14, 28], and voxelization based on inputted image data [29]. Solid voxelization is more involved than surface-only, as it requires voxelizing the space contained within a model as well. A straightforward approach simply sets the state of all voxels between two surface boundaries as being 'inside', by flipping bits or flags [8], but another popular method is employing slicing algorithms [30, 31], or some combination of the two [32]. Common differences between modern voxelization algorithms involve GPU- versus CPU-based implementation, fixed-function versus programmable pipeline implementation, data structure output (generally, grid versus Sparse Voxel Octree), and use of in-core versus out-of-core (extended memory) algorithms; combinations of these attributes can have significant effects on performance, and motivated our pursuit of a Schwarz and Seidel-based study.

Solid voxelization is more valuable as it provides results that contain volume information- a big benefit of voxel-based 3D data representation. Additionally, GPU-based approaches typically yield faster results and programmable pipelines offer more flexibility in the results of voxelization. The combination of these factors makes the Schwarz and Seidel tile-based, solid voxelization algorithm a natural fit as the basis of our work. However GPU-based implementations are

not straightforward and present many opportunities for improvement, which are the focus of this study.

1.2 Problem definition

This study is performed for a host company with an existing CPU-based implementation of the Schwarz and Seidel algorithm. Considering GPU optimization for data-parallel computing and the inherent parallelizability of the voxelization process, we assume that translating the implementation to be GPU-based would result in performance improvements. However, this is not a trivial task and leaves space for evaluating various approaches. Therefore, this report aims to compare different GPU-based implementations of one voxelization algorithm in order to answer the question of how certain GPU hardware-aware approaches affect memory requirements and execution time.

1.3 Purpose

The purpose of this degree project is to improve the performance of a tile-based, solid voxelization algorithm by taking advantage of GPU-specific attributes for increasing efficiency. This report explores four different GPU-based implementations of the same algorithm in order to compare their performance effects. It aims to answer how naive, non-branching execution path, and pre-computational approaches affect the runtime performance and memory usage of the voxelization algorithm.

1.4 Goals

The goal of this project is to measure the effects of GPU-specific implementation differences on a solid, tile-based voxelization algorithm. This has been divided into the following sub-goals:

1. Measure the execution time in ms of the solid voxelization process from beginning to end:
   a. For a naive implementation
   b. For a non-branching execution implementation
   c. For a data pre-processing implementation
   d. For an implementation combining the two previous approaches

2. Measure the maximal memory usage in bytes of the solid voxelization process from beginning to end:
   a. For a naive implementation
   b. For a non-branching execution implementation
   c. For a data pre-processing implementation
   d. For an implementation combining the two previous approaches

1.5 Research methodology

Evidence of the benefits of varying GPU-based implementations will be quantitative (through benchmarking experiments), but rely on a qualitative analysis of the relevance of chosen improvements. Our literature study and implementations are motivated by an analysis of academic research in the field of voxelization and GPU programmability. Our quantitative data collection method is experimental [33], in order to determine algorithm performance and conclude the impact of the different implementations. Performance measurement on multiple, voxelized models will result in data regarding time and memory usage. Runtime directly impacts the usage of the algorithm, therefore reducing it will improve workflow for the host company and possibly for future product users. Increased memory usage imposes additional limitations for our voxelizations, such as the polygon count in input geometry or voxel count in the results, so cutting it down is in the interest of performance as well.

1.6 Delimitations

1.6.1 Existing algorithm

The primary delimitation of this thesis involves its study of an existing algorithm, rather than development of a novel voxelization approach. Though various GPU implementations are applied, the abstract logic is based on the work of Schwarz and Seidel in their tile-based, solid voxelization algorithm. We do not intend to create our own algorithm and test it, as that would require far more analysis than just GPU-based performance effects. We do, however, choose the Schwarz and Seidel approach based off predetermined criteria for the requirements demanded of the voxelization algorithm covered in this study, as well as extend it with design decisions related to requirements from the host company.

1.6.2 Single algorithm

Our secondary delimitation is keeping the scope within one algorithm, as opposed to comparing the performance of several voxelization algorithms. Our initial objective was to qualitatively research and assess an algorithm with a potentially improved performance compared to the host company's implementation of the Schwarz and Seidel algorithm. However, after defining criteria for the algorithm based off the attributes of the Schwarz and Seidel approach as well as requirements put forth by the host company, we actually found the current algorithmic approach to be preferable out of the ten others considered. All other methods are also relatively modern (all proposed after 2000) and contain a variety of interesting techniques, but do not fulfill the set criteria (see Table 1).

Our criteria, in order of importance, are listed and defined below:

- **Volume voxelization** - As voxelized models are to be used in the procedural engine (meaning users interact with our results), we want volumetric

data. Additionally, introducing underlying volumetric data can help compress resulting voxels.

- **GPU-compatible** - Must be implementable on the GPU using shaders rather than the fixed-function pipeline.
- **Attribute conserving** - As the voxelization should be able to be volumetric, inferring voxel attributes (color, reflectivity, etc.) could prove a large improvement to the result of the voxelization. This might cause an overall slower voxelization, however it is necessary for the host company results.
- **Output to Sparse Voxel Octrees** - The host company uses Sparse Voxel Octrees for their voxel data, so algorithms that do not provide Sparse Voxel Octree output will need another conversion step for the final result.
- **In-core** - Out-of-core algorithms are good for very large voxelizations, either extremely large input meshes or large output grids. Our intended use is limited in scope, so we relax this criterion and determine that an in-core algorithm is sufficient for our use.

Algorithm	Volume voxelization	GPU-compatible	Output to Sparse Voxel Octrees	Attribute conserving	Out-of-core
Our algorithm	X	X	X	X	
Schwarz and Seidel [8] tile based	X	X			
Schwarz and Seidel [8] sparse	X	X	X		
Pantaleoni [9]		X		X	
Fang and Chen [30]	X	X			
Liao [32]	X	X			
Weng et al. [13]	X		X		
Dong et al. [34]	X	X			
Baert et al. [14]			X		X
Loop et al. [29]		X	X		X
Pätzold and Kolb [28]		X	X	X	X

Table 1: Algorithm comparison on basis of criteria fulfillment

1.6.3 GPU-specific evaluation

Our tertiary delimitation regards restricting the comparative analysis to GPU-based implementations, and disregarding CPU-based alternatives. Apart from focusing on the GPU from an obvious performance improvement perspective, studying CPU versus GPU performance can be tricky and require careful implementation, experimentation, and evaluation [35], which are outside the scope of this study. Comparisons between the two can skew favorably to GPUs when studies do not consider that non-parallelizable tasks that are not offloaded to the GPU must still be handled by the CPU, or that CPU implementation should be optimized to a fair level through parallelization on multiple cores, cache-friendly memory access, etc. [35]. Additionally, some algorithms are inherently less parallelizable and are thus better suited for general-purpose CPUs. However, when

provided an initial implementation of the Schwarz and Seidel algorithm from our host company, it was CPU-based; transferring it to the naive, GPU-based implementation yielded significant performance improvements.

1.7 Structure of the thesis

Chapter 2 presents relevant background information about voxelization concepts and GPU programmability, as well as related work regarding surface and solid voxelization. Chapter 3 presents the experimental methodology, including design, implementation, data collection, and setup. Chapter 4 presents the tested system, from design specifics to the actual implementation of the various approaches. Chapter 5 presents relevant results, followed by a discussion. Finally, Chapter 6 presents conclusions and future work.

2 Background

This section introduces background concepts relevant to the topics covered in this thesis.

2.1 GPU programmability

2.1.1 GPU computing

GPU computing takes advantage of the massively parallelized hardware available in modern graphics cards to perform many equivalent computations or instructions concurrently and far more efficiently than a CPU. This is achieved by bundling multiple arithmetic logic units (ALUs) into a SIMD unit and then placing several SIMDs in a compute unit; a standard example is AMD's Graphics Core Next microarchitecture, which includes compute units (i.e. the hardware that enables compute shader programmability) with 4 SIMDs, each with 16 ALUs [36]. SIMD stands for Single Instruction Multiple Data, meaning executing the same instruction over multiple work units- this is how the GPU is able to perform a huge number of equivalent computations in parallel.

One argument passed into the compute shader is a thread index, but this does not correspond to an individual thread as one might expect with a CPU. Instead, the compute shader accepts a bundle of threads; this is commonly referred to as either a "wavefront," a "workgroup," or a "warp" [37] (the last term is NVIDIA-specific and describes a group of 32 threads). Threads do not take individual branches, but rather all execute the same instructions concurrently; in some cases, this may mean that results from some threads are thrown out after the instruction is executed [37]. Also, having threads executing the same instructions poses a thread divergence issue: if an execution path branches into two, how do we handle the different cases? The solution is SIMT (Single Instruction Multiple Threads) architecture. SIMT falls under SIMD architecture, and applies to processors that map thread execution paths during runtime. When the processor reaches a split in the path, like an if statement, it disables some of

the threads while the others execute the if case, and then vice versa for the else; eventually, the threads converge again. As a result, all enabled threads within a single warp still abide by the rule of executing the same instruction sequence [38].

Instead of managing individual threads, compute units switch out work-groups during memory operations in order to minimize latency [39]. GPUs are designed for high throughput, hiding memory access latency by having a large amount of work in-flight [38]. The SIMD nature of modern GPUs complicates memory access of individual threads, as coalesced reads of contiguous blocks of memory should result in faster execution time, but certain access patterns are slower due to bank conflicts [38]. GPU shared memory is designed to be accessed quickly and by all threads currently executing. However, accesses are not coalesced and served in the same transaction as in "global memory"; instead, the data is stored in banks which are then accessed by the enabled threads. The problem here is that if two or more threads attempt to access different information in the same bank (i.e. a bank conflict occurs), the bank must process multiple transactions, which results in slower performance [38].

2.1.2 Graphics APIs

Interaction with the GPU is done via an API, like OpenGL [40], DirectX [41], Metal [42], or Vulkan [19]. APIs provide methods for the application to draw 2D and 3D objects using hardware acceleration. More recent iterations provide increased flexibility in how the objects are drawn by including the ability to supply shaders (instructions on how vertices or colored fragments are processed) to the GPU. Some APIs also provide methods for using the GPU for general purpose computation, not just 2D and 3D geometry primitives. For this thesis we use Vulkan's general purpose computing capabilities.

2.1.3 Compute shaders

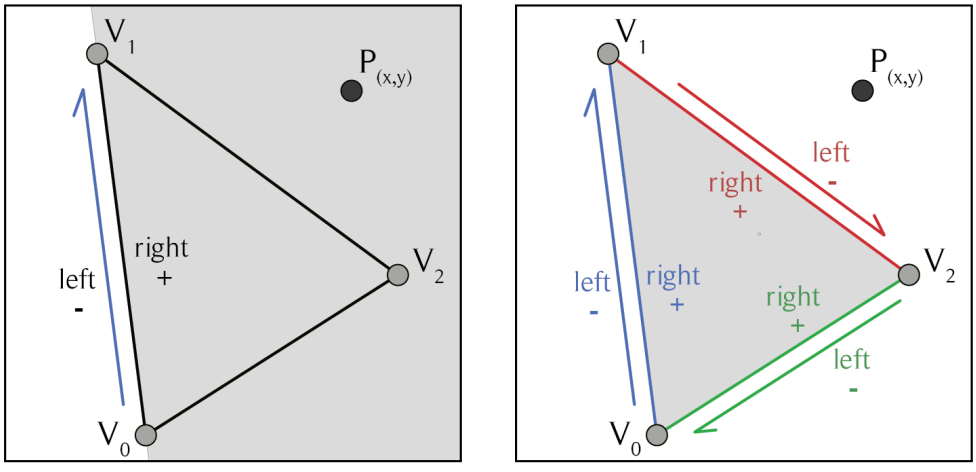
Shaders are the backbone of the GPU programmable pipeline because they allow developers far more control over the specifics of what happens to vertices and pixels, rather than relying on fixed functions which may be simpler to use, but are limited in their range of capabilities. When shaders were introduced, each type would generally correspond to a certain stage in the graphics pipeline; for example, the vertex shader was designed to take the vertices of the triangles in a mesh as input and process them to output values associated with those vertices, such as position, color, texture coordinates, or triangle normal [24]. However, having separate processors for different shaders can lead to poor load-balancing and utilization, so "unified shaders" were developed; unified shaders consolidated individual processors into one large grid which could handle vertex shader, pixel shader, and geometry shader tasks [16, 20], as well as offer additional general purpose computing functionality. Compute shaders, which were introduced as part of DirectX 11, are an example of a shader that does not correspond to any particular stage and acts as a form of GPU computing [24].

2.2 Voxelization

2.2.1 Rasterization

Rasterization, a 2D variant of voxelization, is the process of identifying all pixels contained within each triangle making up a polygonal mesh and outputting the collections of pixels as fragments [23]. Since the 1990s, polygons have been the default for representing geometric objects in computer graphics. While they can be quadrilateral, polygons are almost always triangles with an order assigned to their vertices; this order is important in identifying the inside versus the outside of the polygon [23]. Complex 3D models are made up of polygons (generally between hundreds [43] to hundreds of thousands [44] in the context of modern video games) connected to form a mesh. However, because a screen display is 2D, rasterization is required to transform these models into a form that can be displayed with pixels. This is a two-step process, beginning with computation of triangle data (such as edge functions) based off the input from the previous pipeline stage; more specifically, the triangle processing stage receives the coordinate locations of all polygon vertices identified by vertex geometry processing, as well as additional information such as the vertex normal, texture coordinates or color [45]. The second step involves traversing the triangle pixel-by-pixel and determining if the two overlap for each case.

The classic algorithm for testing if a point lies outside any edge of a triangle involves edge functions (see Figure 3), and was proposed for rasterization by Juan Pineda in 1988 [46]. Over forty years, many improved parallel rasterization algorithms have been proposed which take advantage of massive hardware improvements [47, 48] and pipeline programmability [49]; however, Pineda's method continues to be the basis of modern rasterization techniques.



(a) Single edge function.

(b) Full triangle test.

Figure 3: Point-triangle edge function test (courtesy of Igor Glukhov).

Pineda [40] defines an edge as a vector with two points (X, Y) and $(X + dX, Y + dY)$ and its edge function as

$$E(x, y) = (x - X)dY - (y - Y)dX$$

where (x, y) is the point (or pixel) being evaluated. This is mathematically equivalent to calculating the cross-product of the vector from (X, Y) to $(X + dX, Y + dY)$ and the vector from (X, Y) to (x, y).

The output of the edge function is positive when the point is to the "right" of the edge, negative if it is to the "left" of the edge, and 0 if it is on the edge. This has a clear application to rasterization, as one can simply evaluate all pixels for "right" or "left" positioning in relation to each edge in the triangle ("right" signifying the inside of the polygon for clockwise vertex order and "left" for counterclockwise) in order to determine overlap.

2.2.2 Triangle-box test

The triangle-box overlap test developed and employed by Schwarz and Seidel [8] in their algorithms is based off Pineda's edge functions [46] and a Möller and Aila paper which proposes a 2D variant of the test [50]. In order to perform a conservative surface voxelization, the test needs to determine all voxels that a triangle overlaps. Additionally, in order to allow efficient sequential and parallel processing of many voxels at once, setup should be quick and only rely on data regarding the triangle.

Schwarz and Seidel's [8] test takes as input a triangle T with vertices v_0, v_1, v_2 and an axis-aligned box B (representing a voxel) with a minimum corner p and maximum corner $p+\Delta p$; axis-aligned means that the edges of the box are aligned with the coordinate axes (i.e. for a 2D case, the vertical edges are parallel to the y-axis and horizontal edges are parallel to the x-axis). They begin by calculating the normal n of T (resulting from the cross products of all edge pairs) and the critical point, given by

$$c = \left(\begin{cases} \Delta p_x, & n_x > 0 \\ 0, & n_x \leq 0 \end{cases}, \begin{cases} \Delta p_y, & n_y > 0 \\ 0, & n_y \leq 0 \end{cases}, \begin{cases} \Delta p_z, & n_z > 0 \\ 0, & n_z \leq 0 \end{cases} \right)^\intercal$$

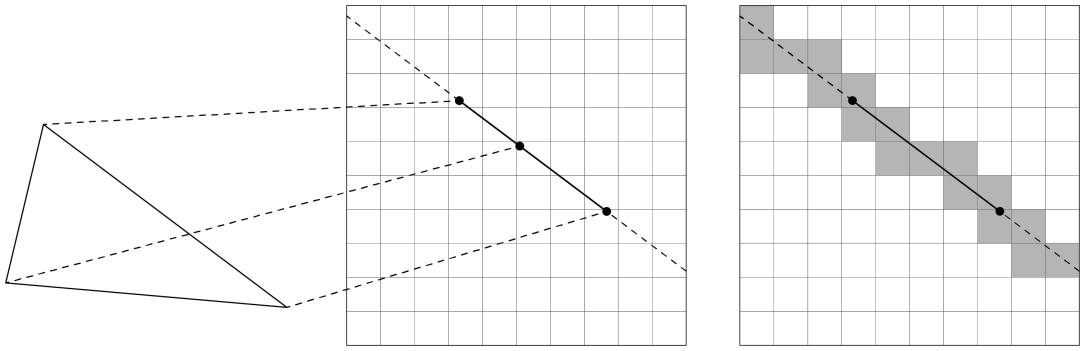
With these values, they can perform a plane overlap test (see Figure 4) to determine whether the voxel in question is split by the triangle's plane, if

$$((n, p) + d_1)((n, p) + d_2) \leq 0$$

where $d_1 = (n, c - v_0)$ and $d_2 = (n, (\Delta p - c) - v_0)$.

This culls all voxels that do not intersect the plane, but does not guarantee that the voxels that do are also overlapped by the triangle, since the triangle is just a small section of an infinite plane. More 2D projection tests need to be run, using Pineda's edge function logic.

For each of the three coordinate planes (xy, xz, yz), Schwarz and Seidel determine if the 2D projections of T and B in those planes overlap. For each



(a) Line segment is an analog for a triangle.

(b) Triangle-plane intersection test

Figure 4: Plane overlap test (courtesy of Igor Glukhov). A line segment is a 1D analog of a 2D angle.

2D projection of B, the edge functions are evaluated at the corner that is 'closest' to the projection of T. For example, for the xy plane the 2D edge normal is

$$n_{e_i}^{xy} = (-e_{i,y}, e_{i,x})^\intercal \cdot \begin{Bmatrix} 1, & \Delta n_z \geq 0 \\ -1, & \Delta n_z < 0 \end{Bmatrix}$$

And by extension

$$d_{e_i}^{xy} = -\langle n_{e_i}^{xy}, v_{i,xy} \rangle + \max\left\{0, \Delta p_x n_{e_i,x}^{xy}\right\} + \max\left\{0, \Delta p_y n_{e_i,y}^{xy}\right\}$$

where $e_i = v_{i+1 \mod 3} - v_i$. With those values, they can test whether

$$\bigwedge_{i=0}^{2} \left(\langle n_{e_i}^{xy}, p_{xy} \rangle + d_{e_i}^{xy} \geq 0\right)$$

is true and, if so, know that there is overlap between the triangle and the box. Additionally, they need to check that T's axis-aligned bounding box, which is the box with the minimum dimensions to contain T, overlaps B. The same tests can be run as for B and T, given T's bounding box, n, d_1, d_2, and $n_{e_i}^{xy}$, $d_{e_i}^{xy}$, $n_{e_i}^{xz}$, $d_{e_i}^{xz}$, $n_{e_i}^{yz}$, $d_{e_i}^{yz}$ $(i = 0, 1, 2)$.

2.2.3 Sparse Voxel Octrees

Similar to a binary tree, an octree is a data structure which can be used to represent volumetric regions. To construct a traditional octree, a 3D cubical volume is recursively divided into eight congruent, disjoint cubes until a certain minimum cube size or level is reached. Each node corresponds to a cube (or, more specifically, a parent cube made up of eight child cubes); leaf nodes contain child cubes that require no more subdivision and are entirely contained or entirely outside of the volume [51]. Continuous 3D models cannot necessarily be broken

down into cubes that are entirely contained or not, but once they are voxelized into a discrete form, octrees become a natural representation. Each node corresponds to a voxel (or a collection of voxels in multiples of 8). A Sparse Voxel Octree does not allocate nodes for empty regions of the total volume; instead of including empty subtrees, it has leaf nodes that do not necessarily correspond to individual voxels (i.e. at the highest resolution) [52].

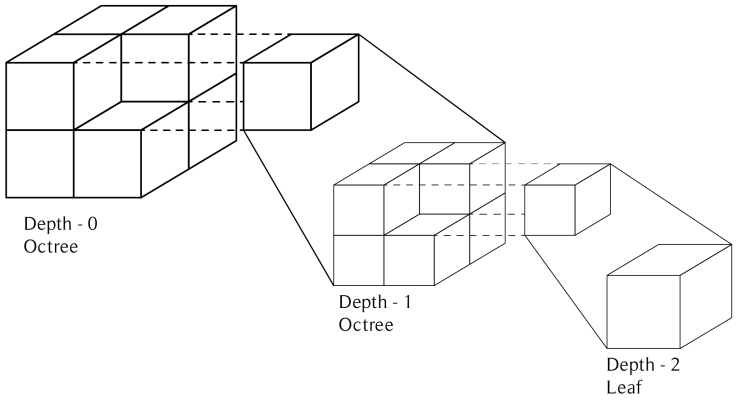


Figure 5: An example octree of maximum depth 2 (courtesy of Igor Glukhov). Note that a Sparse Voxel Octree can have leaf nodes at any level, not just at the maximum depth.

One downside of voxel-based models is higher memory requirements compared to polygon meshes. As resolution grows, the voxel storage increases by $O(n^3)$ [24], whereas since every polygon is stored as a single unit, increasing the polygon count results in a growth of O(n). The purpose of Sparse Voxel Octrees is compacting memory layout, which saves space and reduces memory bandwidth [52]; the hierarchical structure also improves the performance of traversal algorithms, which are usually a linear function of the number of nodes in the tree [51]. Laine and Karras [52] offer extensive information on Sparse Voxel Octrees, including their method of storing voxel data, like shading attributes, in conjunction with its parent, to avoid allocating storage for individual leaf nodes.

Sparse Voxel Octrees offer the advantages of voxel-based 3D data representation, while reducing rendering and memory computational requirements [52]. Additionally, there are still existing assets and tools for working with or creating meshes in traditional polygon form [53–56]. Therefore, voxelization offers the convenience of working with meshes in polygonal form and the advantages of representing data in voxel form.

2.2.4 Schwarz and Seidel algorithm

Schwarz and Seidel [8] introduce several versions of surface and solid voxelization algorithms adapted for parallel GPU-based implementation, rather than

using fixed-function rasterization. This frees the implementation from accounting for the gaps that appear in rasterization-pipeline-based surface voxelization techniques. They extend to solid voxelization with a tile-based approach, which serves as the basis for the algorithm studied in this paper (with the addition of Sparse Voxel Octree format conversion). In tile-based solid voxelization, all voxels contained in the 3D grid constituting the voxelization space are organized into columns along one axis (they choose this as the x-axis and we choose z, but ultimately this is irrelevant), which extend from the upper to the lower bound of the discrete space; 16 of these columns together make up a 4x4 tile.

Tile-based solid voxelization consists of two stages: tile assignment and tile processing. Tile assignment involves determining overlapped tile-triangle pairs (see Figure 7). Within the 3D grid space lies the polygon mesh, and those voxels that overlap the model will be part of the final discrete representation. To assign triangles to tiles, Schwarz and Seidel [8] execute these steps:

1. Perform a first pass for all triangles in parallel to determine the number of tiles that overlap their 2D projections in the yz plane (in our case, it would be the xy plane). This requires storing just the integer number of overlapped tiles in a data buffer.

2. Scan the buffer for the size of the required work queue and the queue offset of each triangle.

3. Perform a second parallel pass through all triangles, this time to identify overlapped tile-triangle pairs and writing them into the work queue, starting at the triangle offset. This requires storing the offset, as well as the values of tiles per triangle.

4. Sort the queue by tile.

5. Transform by compaction the list of triangle-tile pairs into a tile-triangle list, removing duplicate entries.

In order to check which tiles overlap a triangle, Schwarz and Seidel [8] use a triangle-point test, which is simpler than the triangle-box test, but uses similar principles:

1. Derive the triangle's bounding box and check whether the box overlaps any voxel column center (i.e. whether it overlaps the tile).

2. If so, test those tiles whose voxel columns are overlapped by the bounding box against the triangle, to determine voxel column center point overlap. Both steps are performed using edge functions- the checks are analogous to the second step in the triangle-box test.

Tile processing is the second stage of the algorithm and involves the 'filling' aspect of solid voxelization (see Figure 7):

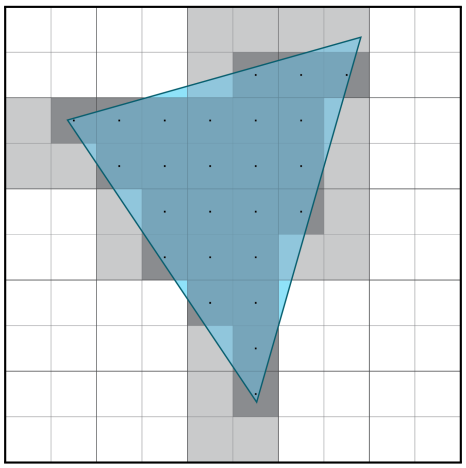


Figure 6: Tile assignment (courtesy of Igor Glukhov). Light-gray are marked tiles, dark-gray are overlapping columns.

1. Allocate one thread per voxel column to execute in parallel, though triangles assigned to a tile are processed sequentially. The thread passes through each voxel and tests for triangle-point intersection while maintaining an active 32-bit memory segment and a flip bitmask corresponding to all its segments (active and flushed).

2. If an overlap occurs, all voxels at the x index (for Schwarz and Seidel- z index for us) and above until the upper bound of the column (or down until the lower bound, as in our implementation (Figure 7 reflects our approach)) are flipped. However, before voxels are set, perform a check that the x index is in the thread's active segment.

3. If not in the active segment, the active segment's value is flushed to global memory by a XOR operation, flipping all bits, and a new active segment is chosen by $\frac{q}{32}$ (q being the x index) and initialized to zero.

4. Flip all bits for the corresponding voxels in the active segment, as well as all bits in the flip bitmask corresponding to succeeding segments. When the tile is processed, the bits within all segments whose bit in the bitmask is flipped will be flipped themselves (i.e. the voxels corresponding to those bits will be set). Atomic operations are avoided because each thread is responsible for its own region of output memory.

Additionally, each thread sets up one triangle for the overlap test and stores the precomputations in shared memory, in order to avoid all threads executing this redundantly.

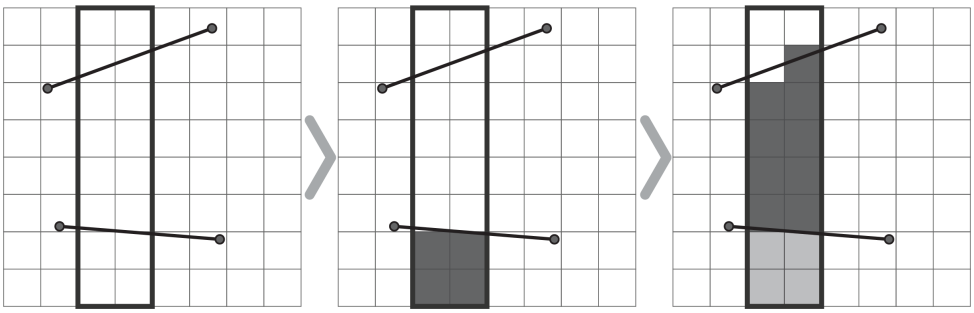


Figure 7: Tile processsing (courtesy of Igor Glukhov). Dark-gray are filled voxels, light-gray are voxels that were recently flipped back to empty state.

2.3 Related work

2.3.1 Surface voxelization

Though our work focuses on solid voxelization, we still felt it was relevant to study recent surface voxelization algorithms in order to determine if there were especially effective examples that could be paired with a volume filling algorithm to fit our requirements. Baert et al. [14] propose an algorithm for voxelization to Sparse Voxel Octrees which utilizes out-of-core (or external memory) algorithms to allow all involved data structures to extend past available memory. However, the mesh to Sparse Voxel Octree conversion requires an intermediate 3D voxel grid. Additionally, implementation is not GPU-based, and performance results are slower than other, GPU-based methods. Pätzold and Kolb [28] extend the previous algorithm to provide direct voxelization to Sparse Voxel Octrees (also referred to as being "grid-free"), while remaining out-of-core and GPU-based. Additionally, they also implement attribute-conservation, meaning that triangle properties are conserved per-voxel to allow easier attribute determination during the rendering stage. Pantaleoni [9] algorithm is a programmable, CUDA pipeline-based approach (taking advantage of vertex and fragment shaders to create a flexible and extended pipeline) to surface voxelization. Apart from proposing only a surface algorithm, VoxelPipe also uses anti-aliased buffers (A-buffers) in its bucketing mode for storing the list of fragments which touch each voxel; the Schwarz and Seidel [8] approach is abstracted for general-purpose GPUs and therefore do not rely on any particular storage buffer or pipeline-specific techniques. Finally, an interesting sparse voxelization algorithm proposed by Loop et al. [29] voxelizes a scene based on a set of images and information about their origin (camera location, direction, etc.). The parallelized, programmable GPU-based technique reconstructs 3D models in real-time, however the voxel culling process relies on image data, which is not relevant for our purposes.

2.3.2 Solid voxelization

We researched many solid voxelization algorithms in order to compare them against the Schwarz and Seidel approach. Though several were discarded, there were a few that proposed interesting methods, even if we decided to pursue Schwarz and Seidel [8] in the end. Garcia and Ottersten [12] offer a CPU-based, solid voxelization algorithm that takes as its input an incomplete point cloud dataset, rather than a polygonal mesh. They approximate the point cloud to a 3D axis-aligned voxel grid by simply setting any voxels that contain at least one point. This then allows surface voxelization through slice-wise construction of a shell over the grid, followed by setting all voxels within each slice; the result is a watertight surface and solid voxelization of the model. Liao [32] proposes a solid voxelization algorithm with a focus on real-time representation of interior materials through dynamic use of different slice functions. Similarly to Schwarz's bit-flipping propagation method, Liao [32] employs surface parity flag toggling to indicate whether depth buffer elements for each slice are volumetrically contained within a model. He notes that the algorithm can be improved through parallelization and by taking advantage of programmable pipelines.

Fang and Chen [30] were one of the first to implement voxelization on the GPU. They created a slicing technique, using traditional (fixed-function) graphics pipeline to voxelize one sheet (slice) of voxels at a time. Their approach works for both solid and surface voxelization and is able to voxelize 3D geometry in a variety of representations, supported by the traditional rendering pipeline for drawing. Their approach is voxel-parallel and uses operations typically used for color blending to achieve greater performance on GPUs. Due to the relaxed precision of color blending techniques in traditional pipelines the results can be somewhat inaccurate, requiring use of anti-aliasing and thus increasing computational requirements for the algorithm. An additional limitation is the inherently dense approach, effectively checking every voxel in the axis-aligned bounding box (AABB) of the drawn geometry. Dong et al. [34] put forth a programmable GPU-based, albeit older, approach to real-time voxelization. The algorithm is made up of three stages: rasterization, where the triangle mesh is converted into voxel space; texelization, where the voxels are encoded into three directional (x-, y-, and z-axes) sheet buffers; and finally synthesis of the three buffers into a final 2D texture (referred to as a "worksheet") representation. Eisemann and Decoret [31] improve upon the Fang and Chen [30] approach, making use of more recent graphics APIs and pipeline features, such as 3D textures and bunching multiple layers of voxels per slice. They also present uses for the resulting voxelization for calculating normals and density estimation. However, as with the previous algorithm, they rely on traditional fixed-function pipeline methods (specifically, using OpenGL on DirectX9 cards).

3 Experimental methodology

3.1 Tested implementations

We test the following GPU-based implementations of a tile-based, solid voxelization algorithm in order to compare and determine how GPU-specific approaches impact performance (see sections 4.1 and 4.2 for further design and implementation descriptions):

1. **The naive implementation** is the simplest CPU to GPU migration of the algorithm, meaning we assume that GPU architecture is somewhat similar to CPU architecture in terms of performance characteristics.

2. **The non-branching execution path implementation** involves reducing conditional branching in the GPU instructions. This is done to minimize the thread divergence of the SIMT blocks on graphics cards, since branching requires additional overhead for masking active threads and is generally slower than arithmetic operations [57]; because of this we expect the non-branching execution path approach to give a consistent runtime performance improvement.

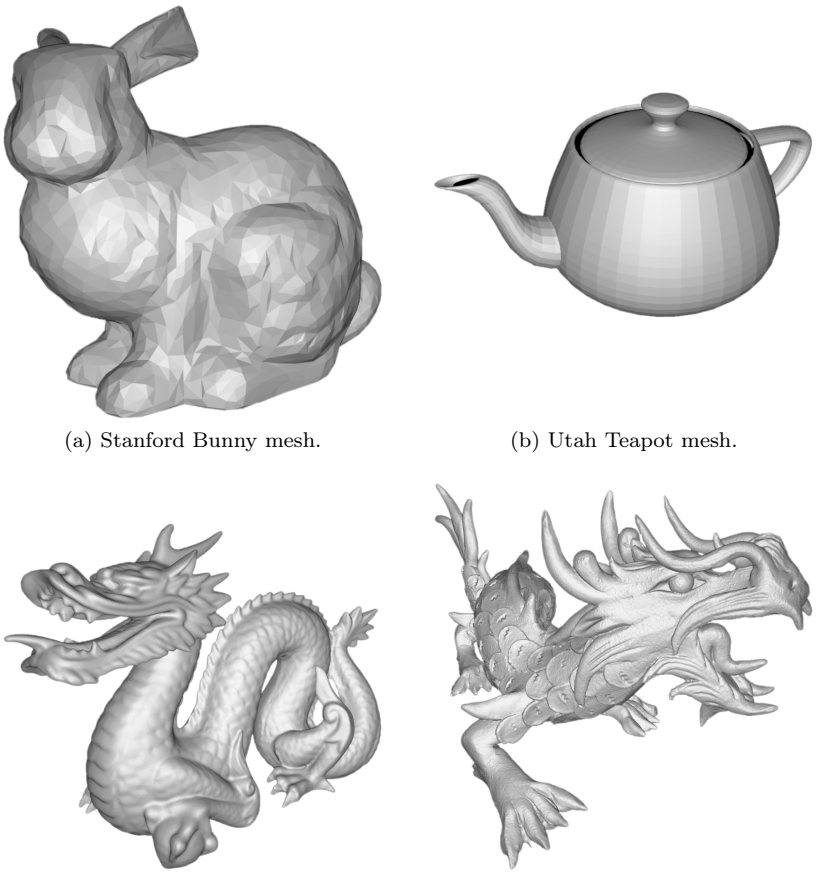
3. **The data pre-processing implementation** reuses common vertices in the input geometry by redefining a triangle from three vertex coordinates to three vertex indices. This reduces the memory usage of the input geometry, since vertices in 3D meshes are usually shared between multiple triangles. We expect the reduced memory usage to increase performance in memory-bandwidth-limited cases, since less data needs to be transferred and some data will be reused, increasing the cache hit rate. However, this makes the memory access non-uniform, which may decrease performance in cases where the computation is *not* limited by the memory bandwidth.

4. **The combination implementation** is a combination of the second and third approaches, to see if combining these potential improvements will have a greater positive impact. We expect the performance effect to be a combination of the effects of the two composing approaches, although it is possible that one will dominate over the other.

3.2 Experimental setup and data collection

3.2.1 Models

We run performance experiments on the four implementations by voxelizing a range of standard polygon meshes (four, variantly-sized models):



(a) Stanford Bunny mesh.

(b) Utah Teapot mesh.

(c) Stanford Dragon mesh.

(d) XYZ RGB Dragon mesh.

Figure 8: The tested meshes, consisting of (a) 4 968, (b) 6 320, (c) 871 414, and (d) 7 218 906 triangles (courtesy of Stanford Computer Graphics Laboratory [27]).

3.2.2 Experimental design and implementation

All the implementations are first tested manually to make sure they work as expected (i.e. the voxelized output is correct). Then, the four approaches are executed sequentially to form a 'run'. In the experiments, we perform 16 runs for each of the four result grid sizes (128^3, 256^3, 512^3, and 1024^3 voxel grids). This process is performed once per tested input mesh (see Figure 8). The resulting process is then run once for each of the three testbeds. See Figure 9 for an overview.

The times for each step are recorded by saving a time-stamp with a C++ standard high-resolution clock on the CPU, *before* issuing the first instruction

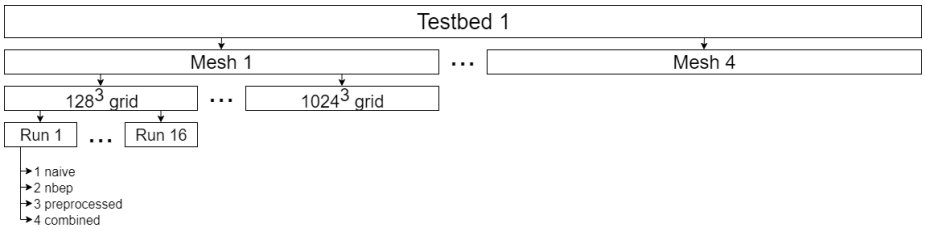


Figure 9: Experiment design structure.

to the GPU for a given step and *after* the GPU is done processing. Since the implementation is done on top of the host company's infrastructure, some additional variance and overhead might be present; however, overhead should stay consistent between experiments on the same test bed. The time measurements include setup, allocation, data transfers, and synchronization and are thus representative of real-world performance.

3.2.3 Data collection

We record the following data points:

- **The number of triangles in the input mesh** - Used to identify cases where setup time takes longer based on the size of the inputted model.
- **The size of the resulting voxel grid** - Used to identify cases where voxelization may take longer because of a larger resulting grid size.
- **The number of tile-triangle overlaps** - Used to identfy how big a given workload is per tile, and whether the voxelization performance suffers as a result.
- **The sizes of GPU-allocated data buffers** - Used to identify the GPU memory requirements for a given implementation.
  - Input geometry buffer
  - Result buffer
- **The duration of each step in high resolution clock cycles** - Used to identify which stage of voxelization has the biggest effect on runtime.
  - Setup
  - Tile counting
  - Prefix scanning
  - Sorting
  - Marking unique tiles
  - Stream reducing

– Voxelization

– Conversion

Based off this data, we analyze the performance impact of each GPU-based implementation approach. We identify the mean runtimes and memory usages (inferred from sizes of GPU-allocated data buffers) for the sets of data regarding time duration and memory usage of the voxelization process for each model on all three testbeds, and then plot the mean. Since the graphics card used affects runtime, we have one runtime graph for each testbed. We compare the durations of voxelization to the result grid sizes for each implementation.

The GPU memory requirement is calculated for each of the 7 voxelization steps, based on input mesh and output grid size parameters. Since each step is done independently, the biggest requirement sets the limitation for the algorithm. Note that the instructions (shaders) and pipeline infrastructure metadata introduce additional overhead, however it is dependent on the platform (graphics drivers, GPU manufacturer, API version, etc.) and the difference between approaches should be negligible for larger output grids or input meshes, where the overhead would impose a practical limitation.

3.3 Testing environment

3.3.1 Hardware

Testbed	GPU	CPU	RAM
1	NVIDIA GTX 1070	Core i7-6700	16GB @ 2133 MHz
2	NVIDIA RTX 2070	Core i7-6700K	32GB @ 2400 MHz
3	AMD RX Vega 64	Ryzen 5 1600X	16GB @ 2400 MHz

Table 2: Testbed hardware setups

3.3.2 Software

The graphics card driver version used is 430.64 for the NVIDIA cards and 19.5.2 for the AMD RX 670 card. The system is run on the Windows 10 OS. The code is written using C++, compiled with Microsoft Visual C compiler and the Vulkan graphics API; we utilize the infrastructure provided through the host company's game engine, but all code related to the voxelization process is of our own doing (any other references to the host company's work are left intentionally vague).

4 Design and implementation

Though the three individual, GPU-based approaches vary in implementation, their differences are primarily shader-based; overall, the algorithmic logic between them remains by and large the same. Therefore, we provide a general

design and implementation descriptions in the following sections, which precede and serve as a basis for the GPU-based approach specifics.

4.1 Design

The base design for our algorithm implementation is illustrated by the flowchart in Figure 10.

4.1.1 Naive approach

As mentioned in section 3.1, the naive approach is the simplest migration from a CPU- to GPU-based implementation; this means that there are GPU-specific characteristics that we disregard. Firstly, we ignore that branching execution paths are known to be slow due to thread divergence handling (explained in section 2.1.1). Instead, we treat each thread as if it were independent, as we would on a CPU. Secondly, we assume that vector operations are still less efficient than simple type operations, when in fact they are performed equivalently on a GPU; for example, vector multiplication is just as fast as single. This is due to the GPU's capability of performing many parallel computations at once, rendering matrix operations trivial. Additionally, the input geometry data is not processed and simply passed to the GPU by vertex positions.

4.1.2 Non-branching execution path approach

The non-branching implementation aims to avoid thread divergence by reducing the number of branches in execution paths- the less divergence handling required, the quicker the voxelization performance. This is done by avoiding if statements; if statements force some threads to take the if case while others take the else, so rewriting the code to eliminate them decreases cases of thread divergence.

4.1.3 Data pre-processing approach

For our data pre-processing approach we split the input geometry buffer into two, reducing their total size by reusing vertex position data. Rather than storing the input geometry as one contiguous array of vertices, the input geometry is processed, yielding a set of unique vertices and an array of indices of vertices per triangle. This reduces average triangle memory usage, as triangles sharing a vertex do not have duplicate position data (see Figure 12). The reduced memory footprint will reduce memory bandwidth used by the initial stages of the algorithm.

4.1.4 Combined approach

The combined approach incorporates data pre-processing into the non-branching execution path design. This combines their strengths and weaknesses and may

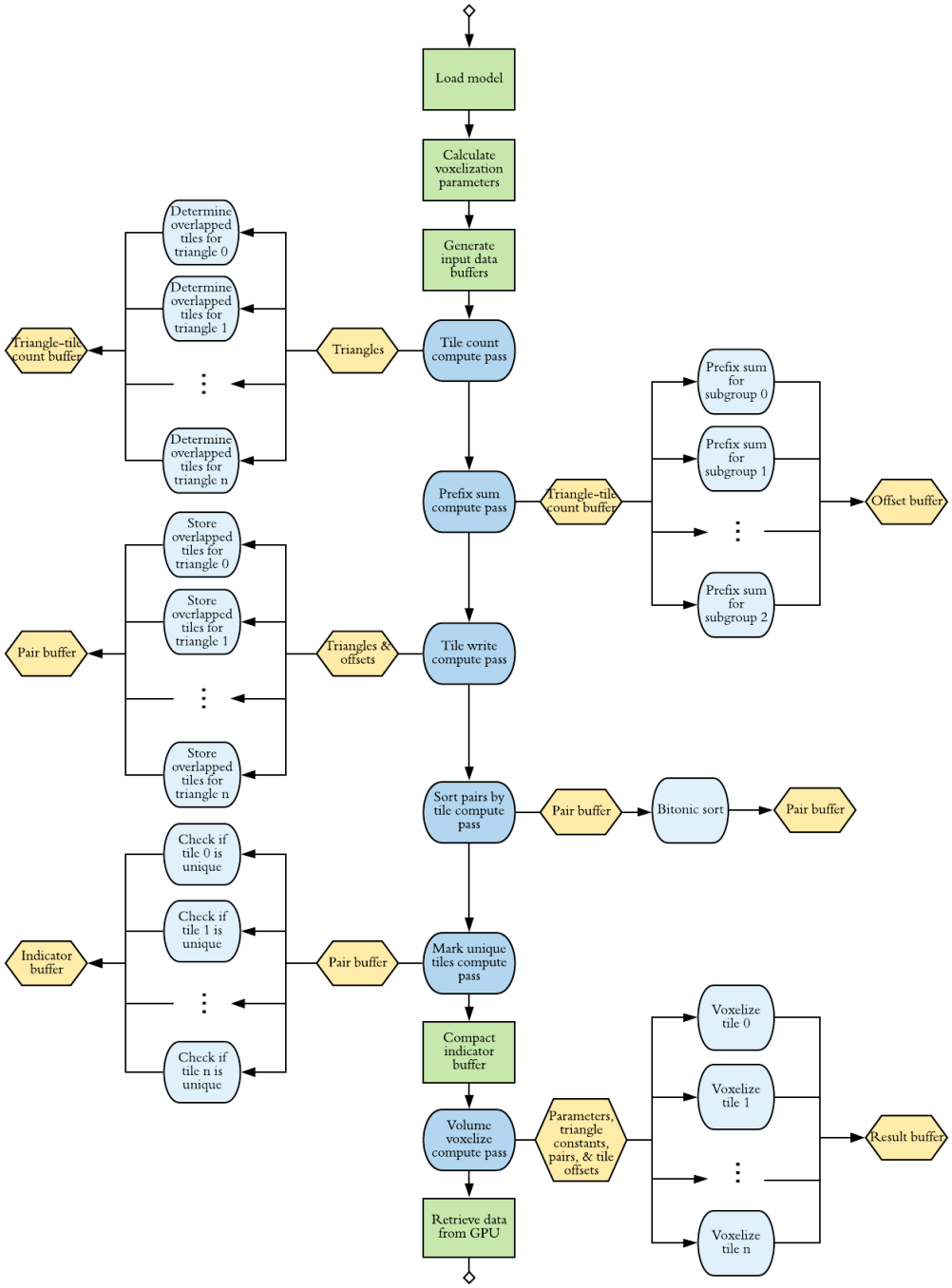


Figure 10: Base implementation of solid voxelization algorithm

not be as beneficial as the sum of their individual performance effects, in case they counteract each other.

4.2 Implementation

The base implementation of our algorithm follows these steps:

1. The GPU Voxelizer class performs all voxelization processes, namely interacting with the GPU, allocating any necessary resources, issuing commands, and retrieving the result. It contains a Model struct, which represents the polygon mesh with one array, containing all vertex positions corresponding to each triangle, without regard for duplicate vertices.

2. The overarching volume voxelization method takes 2 arguments: the path to the model file and the size of the output grid.

3. The volume voxelization method:

    - Performs a GPU capabilities check to make sure it supports all necessary operations.
    - Loads the mesh model from the provided file, calculating the size of the loaded mesh.
    - Calculates the transformation necessary for the model to be within the result grid.
    - Allocates GPU resources, namely data buffers for input and output and a pipeline for a given compute shader, which includes all necessary information, such as which buffers are going to be used for input and output, the descriptor sets for those buffers, and the command buffer for the compute pass.
    - Performs the voxelization passes:
        (a) *The first pass* takes the geometry data and voxelization parameters as input, transforms the vertex position with the transformation calculated during the model loading process, and calculates the per-triangle constants and the number of tiles overlapped by a given triangle. It is triangle-parallel, meaning that one thread is allocated per triangle.
        (b) *The second pass* takes the number of overlapped tiles calculated by the previous one and performs an exclusive scan on them, outputting offsets for a given triangle to write its overlapped tile-triangle pair to. This is also done in triangle-parallel fashion, allocating the smallest number of workgroups that fully contain the number of triangles. The size of a workgroup is dependent on hardware, but is typically around 1024 threads.

(c) *The third pass* takes triangle constants from the first pass and the offsets from the second pass and writes the overlapping tile-triangle pairs into the output buffer. This is also done per-triangle. The pairs contain tile coordinates and the triangle index (see Figure 11).

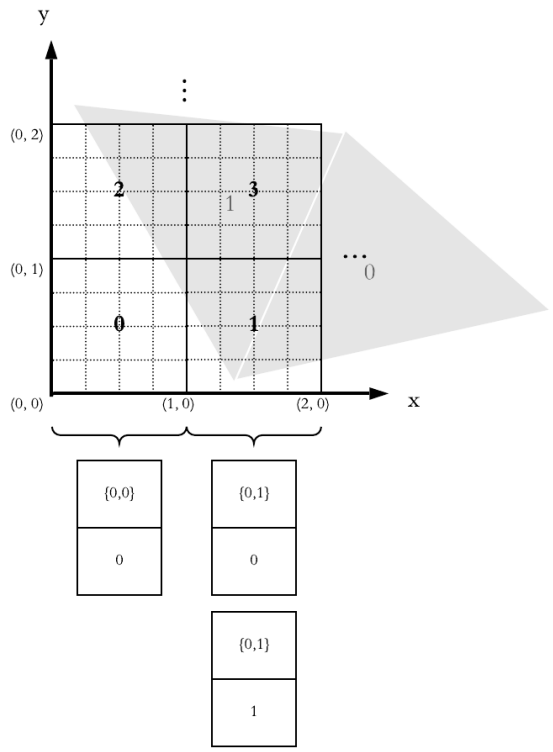


Figure 11: Tile data structure

(d) *The fourth pass* sorts the pairs. We employ bitonic sort along with an optional padding pass, populating the rest of the tile-triangle pair buffer with dummy pairs, until the buffer tile count is an exact power of two. The sorting is done in-place, meaning the input and the output are the same buffer and in pair-parallel fashion, allocating one thread per pair.

(e) *The fifth pass* marks unique tiles in the pair buffer. The marks are written to a different buffer, with a pair's index in the buffer if it is different from the previous one or zero if it is the same.

(f) *The sixth pass* is currently done on the CPU. It performs stream compaction on the unique marker buffer, populating a buffer

with only non-zero values. This is done linearly by the CPU by appending a non-zero index to the offset buffer for the last pass.

(g) *The seventh pass* is the tile-parallel voxelization pass. Its inputs are triangle constants, voxelization parameters, tile offsets from the previous pass, and the tile-triangle pairs. For each tile, it goes through its overlapped triangle list and flips all voxels below the intersection point of a given column in the tile. The final output is the result voxel grid. The pass is done in tile-parallel fashion, allocating one workgroup per tile, which consist of 4x4 threads, each dedicated to its own voxel column.

4. The last step of the voxelization is translating the results from the intermediate, dense, grid format to the host company's Sparse Voxel Octree format. This step is excluded from our results, as the specifics of the format are not important and are likely to be different depending on the voxelization's intended use.

4.2.1 Naive approach

The naive approach does not have any special implementation characteristics outside of the base implementation described in the previous subsection.

4.2.2 Non-branching execution path approach

We avoid the if statements present in the naive approach by taking advantage of a typecasting trick: when converting a boolean to a float, the behavior is well-defined in that a false value results in 0.0, while a true results in 1.0 [40]. Based on this, instead of conditionally manipulating data, the manipulation always occurs, but does not change the data if the condition is false. For example, when calculating triangle constants, instead of having an if condition to check whether the normal is negative to reflect the edge normal direction (as in the naive implementation),

```
if normal.z < 0:
    edgeNormalDirection *= -1
```

the edge normal direction is always multiplied by 1.0 when the condition is false and -1.0 when the condition is false:

```
edgeNormalDirection *= 1 - 2 * float(normal.z < 0)
```

4.2.3 Data pre-processing approach

The pre-processing implementation is mainly done on the CPU. The triangle data structure is changed from the naive implementation by introducing a second array: the index buffer (bottom array in Figure 12). Each element corresponds to the position of a triangle's vertices in the vertex buffer (top array in Figure 12). The input geometry is processed per vertex: we insert a vertex into a set

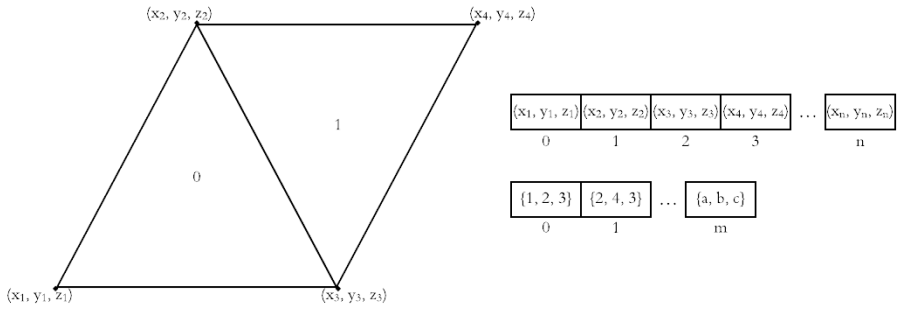


Figure 12: Triangle data structure

(if no equivalent vertex is already present (to avoid duplicates)) and append the index of the vertex in the set to the triangle buffer. To assemble a triangle's three vertex positions on the GPU, the shader retrieves the indices of the three vertices and then retrieves the vertex positions by their indices. This makes memory access less predictable, as many triangles may share the same vertex positions. However, it also lowers the bandwidth requirement for a triangle, as indices are smaller compared to vertex positions and the common positions will be cached, increasing their following retrieval time.

4.2.4 Combined approach

The combined approach simply includes changes from both the data pre-processing and non-branching execution path implementations.

5 Results and analysis

5.1 Major results

The following subsections present relevant data collected from 3072 voxelization runs over three testbeds (NVIDIA GTX 1070 card, NVIDIA RTX 2070 card, and AMD RX Vega 64). All raw, aggregated, and graphed data can be accessed through the links in the Appendix. However, for the purpose of drawing conclusions on the performance of the four algorithm implementations, we only directly reference the mean runtimes (with and without considering the time-demanding pre-processing setup), the relative performance of the GPU-based implementation changes compared to the naive approach, and the memory requirements. Since the experimental voxelizations were performed on four different models, we also chose the graphs for models which best showcase the behavior of the four GPU-based approaches.

5.1.1 Mean runtime performance

The following three pairs of graphs (Figures 13, 14, and 15) present the mean runtime in ms of the voxelization process for the Stanford Bunny (smallest mesh) and the XYZ RGB Dragon (largest mesh) models on each testbed. The grid size unit on the x-axis refers to the number of voxels on one face of the resulting discrete space, meaning that for a grid size of 128, the result grid is actually a 128x128x128 voxel cube.

The different approaches resulted in essentially equivalent runtimes for the smaller mesh (left graph in Figures 13, 14, and 15), while the larger mesh revealed clearer variation between performance (right graph in Figures 13, 14, and 15) (see section 5.1.2 for a summary of the relative performance effects on all models). However, in both cases the non-branching execution path approach improved the mean runtime compared to the naive approach, whereas the performance effect of the pre-processed and combination approaches varied between graphics cards.

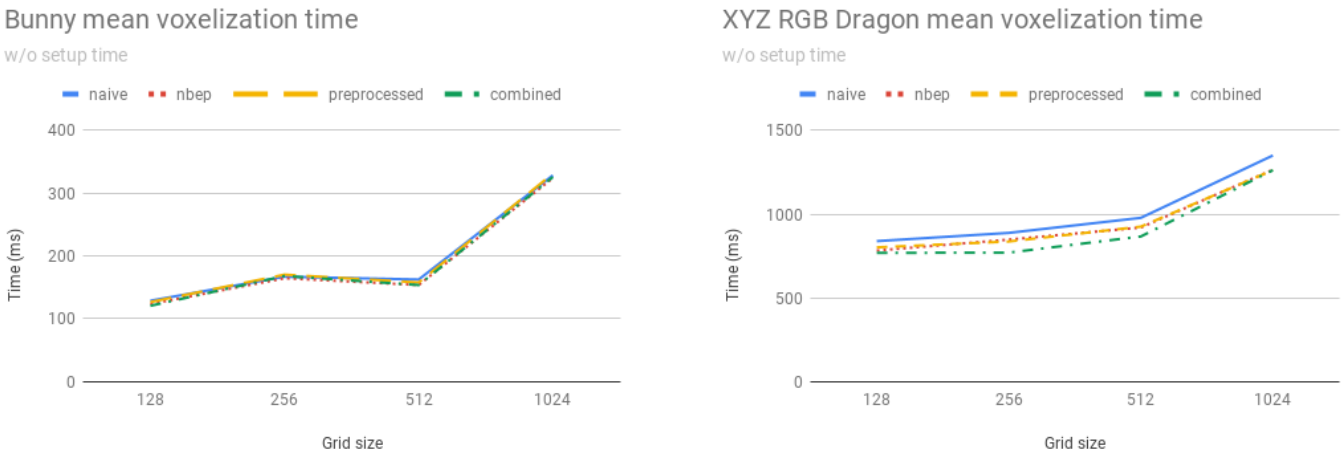


Figure 13: Mean runtime performance results for Stanford Bunny and XYZ RGB Dragon meshes on NVIDIA GTX 1070 card. Does not include the pre-processing step.

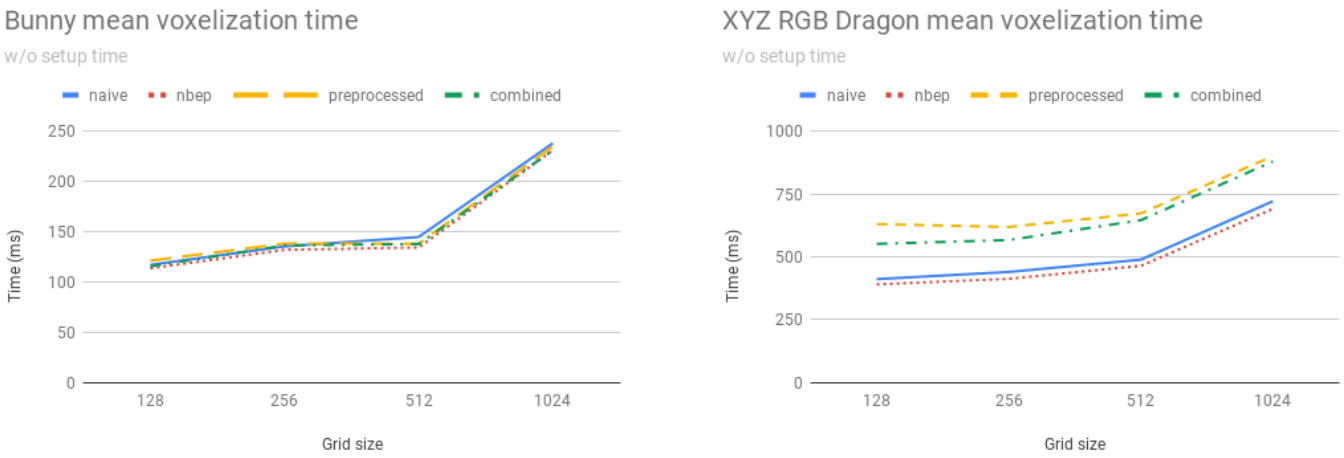


Figure 14: Mean runtime performance results for Stanford Bunny and XYZ RGB Dragon meshes on NVIDIA RTX 2070 card. Does not include the preprocessing step.

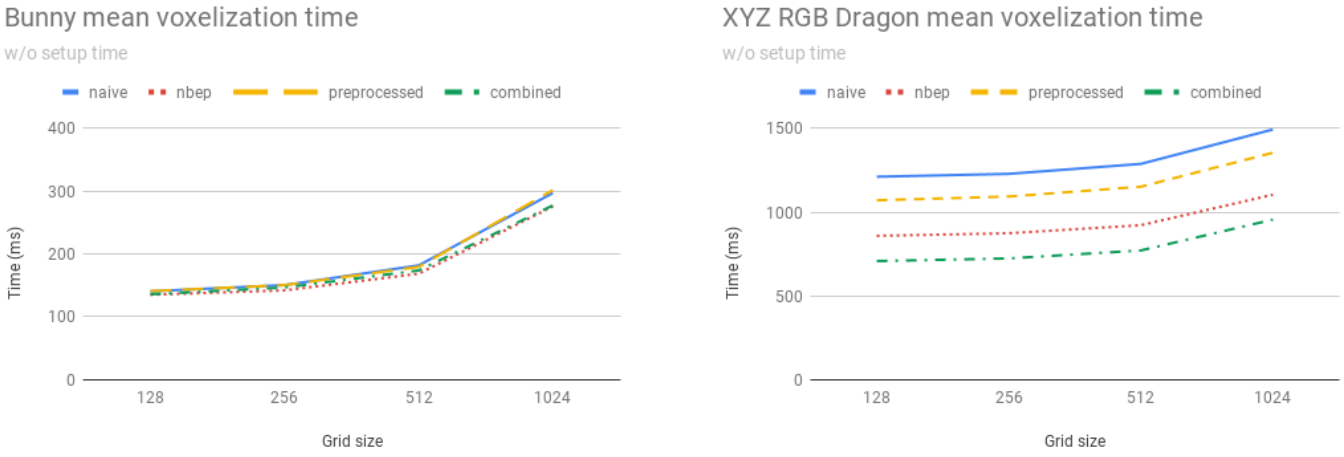


Figure 15: Mean runtime performance results for Stanford Bunny and XYZ RGB Dragon meshes on AMD RX Vega 64 card. Does not include the preprocessing step.

	Stanford Bunny	Utah Teapot	Stanford Dragon	XYZRGB Dragon
128	127 ms	117 ms	224 ms	752 ms
256	150 ms	132 ms	239 ms	775 ms
512	157 ms	151 ms	283 ms	842 ms
1024	283 ms	201 ms	539 ms	1104 ms

Table 3: Average voxelization times for different meshes, excluding pre-processing time.

5.1.2 Relative runtime performance

The following three pairs of graphs (Figures 16, 17, and 18) present the relative runtime performance impact of the changed voxelization implementations; the naive approach is used as the baseline.

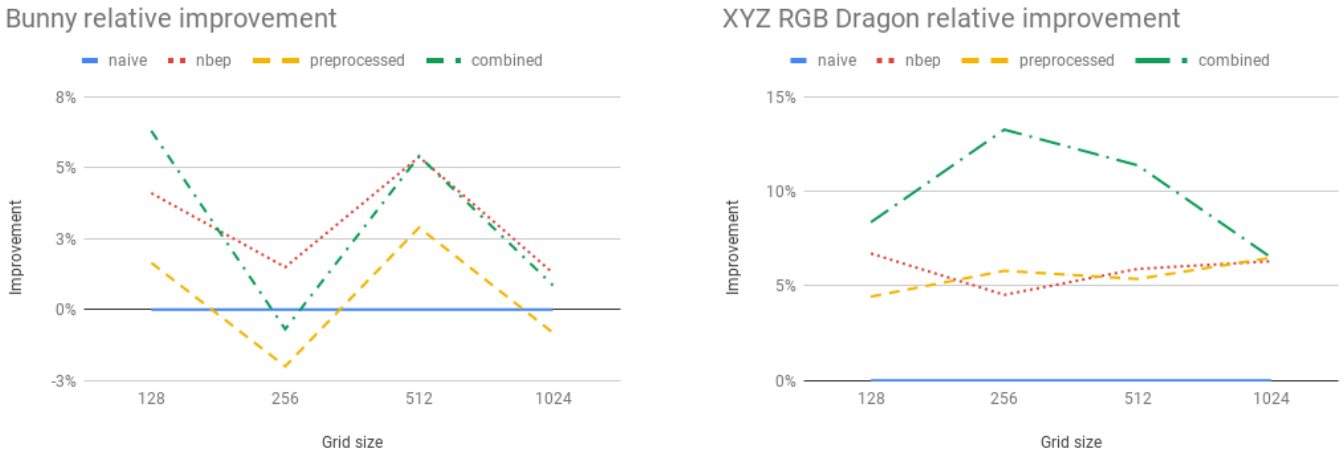


Figure 16: Relative runtime improvement results for Stanford Bunny and XYZ RGB Dragon meshes on NVIDIA GTX 1070 card. Does not include the pre-processing step.

For the Stanford Bunny mesh (Figure 8a) the non-branching execution path had the best performance for all grid sizes with an average improvement of 5% over the naive approach. The pre-processing approach performed closer to the naive one, ranging from being 1% slower to 2% faster, disregarding the pre-processing step (which is negligible for the 4968 triangles of the Bunny mesh). The combined approach always performed worse than the non-branching execution path, but faster than the naive approach. The results for the Utah Teapot mesh (Figure 8b) are very similar to the ones for the Bunny.

For the Stanford Dragon mesh (Figure 8c, 871414 triangles), the results di-

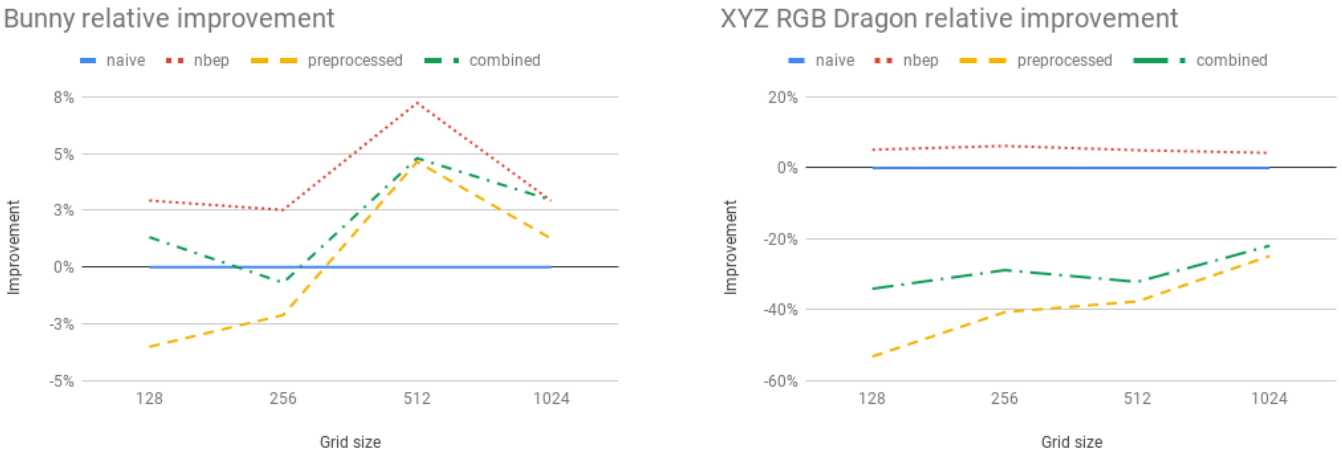


Figure 17: Relative runtime improvement results for Stanford Bunny and XYZ RGB Dragon meshes on NVIDIA RTX 2070 card. Does not include the pre-processing step.

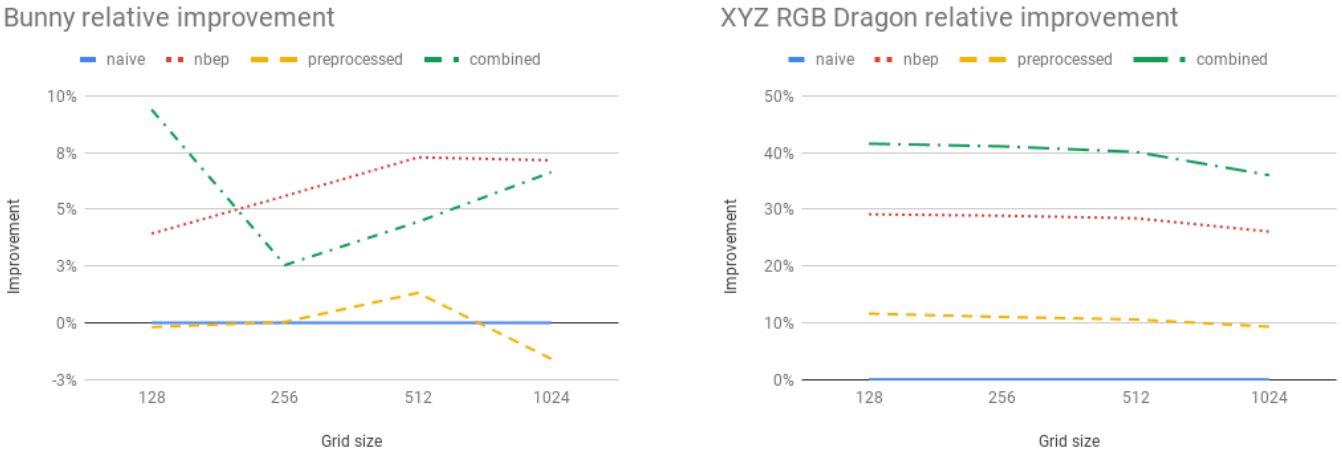


Figure 18: Relative runtime improvement results for Stanford Bunny and XYZ RGB Dragon meshes on AMD RX Vega 64 card. Does not include the pre-processing step.

verged for AMD and NVIDIA graphics cards. On NVIDIA, the non-branching execution path was always the fastest, being about 4% faster than the naive

approach. The pre-processed approach performed similarly to the naive one, ranging from 3% slower to 3% faster. The combined approach was marginally faster than the pre-processed approach, but always performed worse than the non-branching execution path alone. On the AMD RX Vega 64 however, all other implementations were faster than the naive approach. The smallest improvement was from the pre-processed one, which ran 6% faster than naive. The non-branching execution path performed significantly better, being on average 17% faster than the naive approach. The combined approach had the best improvement, at around 21% quicker runtime performance.

Finally, for the largest XYZ RGB Dragon mesh (Figure 8d, 7218906 triangles), the non-branching execution path was not always the quickest, however was still consistently 5% faster than the naive approach on NVIDIA GPUs. On the AMD RX Vega 64 graphics card, the non-branching execution path averaged 28% runtime performance improvement. On the NVIDIA RTX 2070, the pre-processed approach diminished performance significantly, running on average 40% slower than the naive approach. However, on the GTX 1070 the pre-processed approach performed on par with the non-branching execution path one, and on the AMD RX Vega GPU it also had a significant performance improvement of 10%. The combined approach was a simple combination of the two other ones, being over 40% faster on the AMD RX Vega, 10% faster on the GTX 1070, and 30% slower on the RTX 2070.

5.1.3 Mean runtime performance with pre-processing

The following three pairs of graphs (Figures 19, 20, and 21) present the mean runtime performance, but with the addition of the pre-processing step of the data pre-processing approach. The distinction between the pre-processed implementation with and without the actual pre-processing stage is that the former includes the first step for pre-processing the input geometry and compressing the vertex array, while the latter uses the data from the pre-processing stage (i.e. keeping the shaders and pipelines for processing the new geometry structure), but does not consider the runtime added by the actual pre-processing. The reason for this distinction becomes clear when we see that for large meshes, like the XYZ RGB Dragon, pre-processing alone runs for 5 times longer than the rest of the voxelization process.

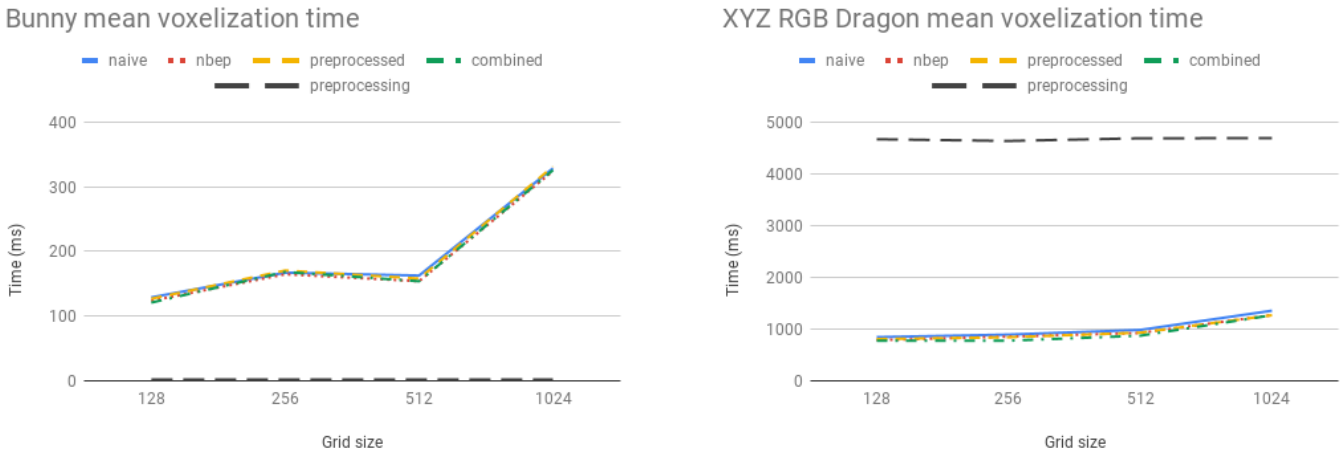


Figure 19: Mean runtime performance results with pre-processing step for the Stanford Bunny and XYZ RGB Dragon meshes on NVIDIA GTX 1070 card.

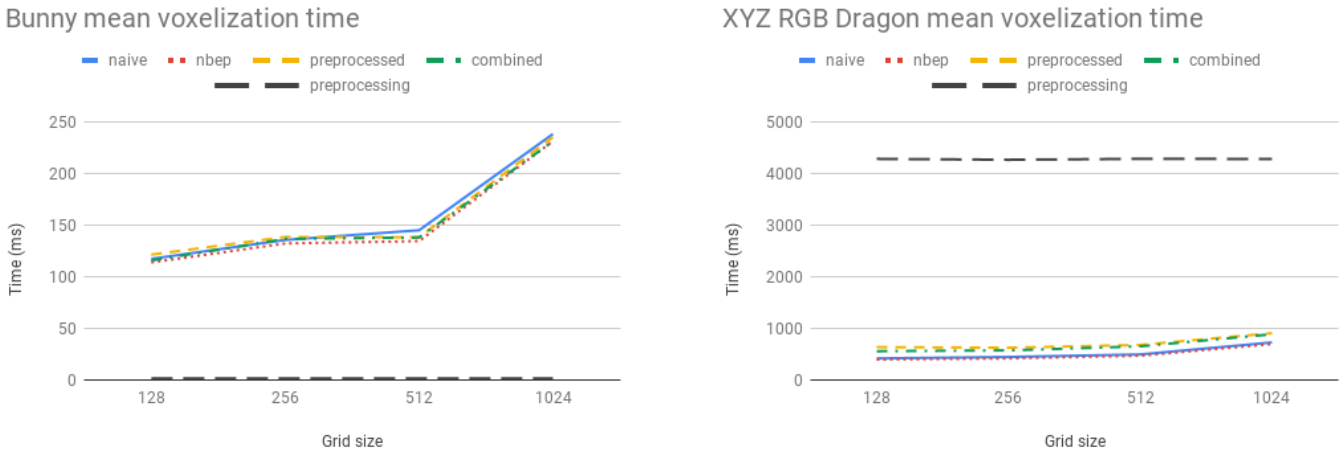


Figure 20: Mean runtime performance results with pre-processing step for the Stanford Bunny and XYZ RGB Dragon meshes on NVIDIA RTX 2070 card.

The data pre-processing step took on average 1.5 ms for the Stanford Bunny, 1.6 ms for Utah Teapot, 501 ms for Stanford Dragon, and 4570 ms for the XYZ RGB Dragon meshes. In other words, pre-processing the geometry took roughly the same amount of time as the rest of voxelization to the 1024^3 voxel grid for

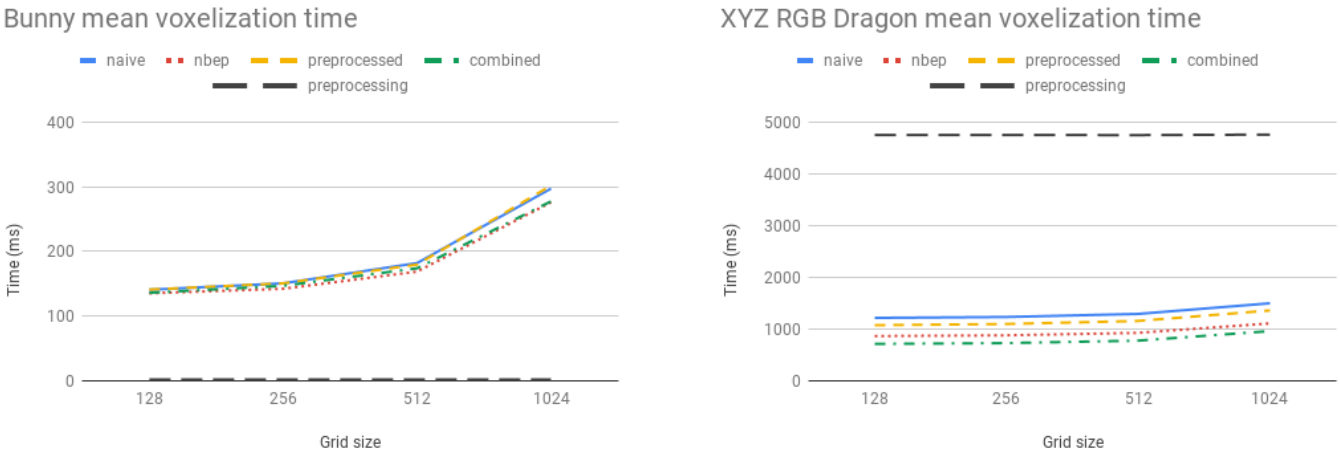


Figure 21: Mean runtime performance results with pre-processing step for the Stanford Bunny and XYZ RGB Dragon meshes on AMD RX Vega 64 card.

the Stanford Dragon mesh, and took significantly longer than the rest of the voxelization process for the XYZ RGB Dragon mesh.

5.1.4 GPU memory requirement

The following pair of graphs (Figure 22) presents the maximum memory requirements (in mebibytes) of the voxelization processes on the Stanford Bunny and XYZ RGB Dragon models. In this case, the four implementations are collapsed into two pairs: the naive and non-branching approaches, which have the exact same memory footprint, and the pre-processed and combined approaches, which are designed to improve memory performance. VRAM stands for Video Random Access Memory and represents memory for data local to the graphics cards.

The memory requirements of the algorithm depend on the number of triangles in the input geometry and the result grid size. The result of the data preprocessing technique was a roughly 20% smaller memory footprint of the input geometry. For particularly large meshes, such as the Stanford Dragon or XYZ RGB Dragon, a relatively large amount of memory is required to process the geometry in the first voxelization stage. For such cases, the compression achieved through reusing vertex positions with our pre-processing approach lowers the memory requirement for the whole algorithm. However, for larger output grids the final voxelization stage is still the limiting factor, which does not depend on triangle position data representation.

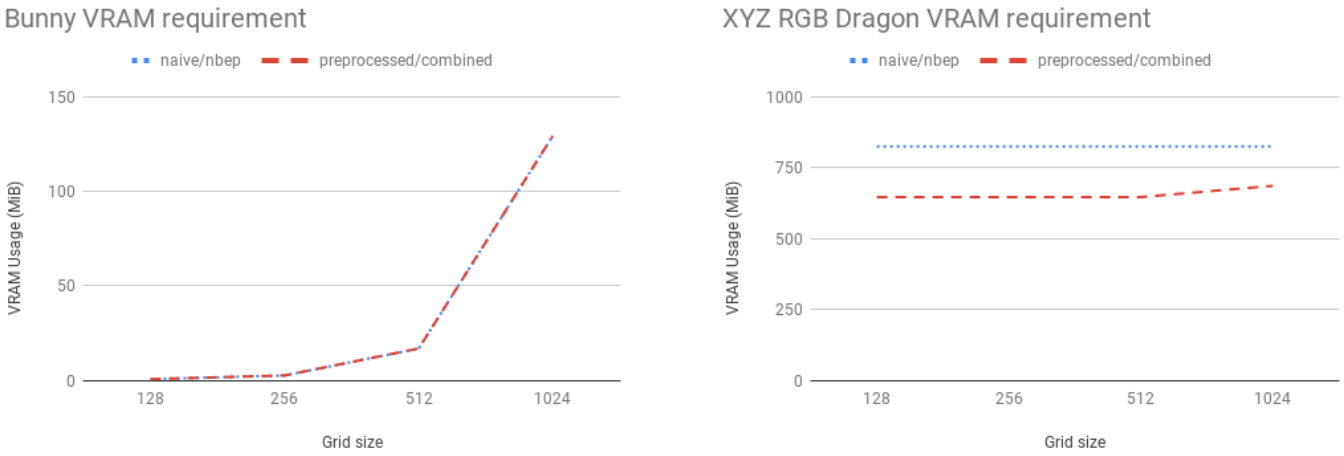


Figure 22: Memory requirements for Stanford Bunny and XYZ RGB Dragon mesh voxelization.

5.2 Discussion

5.2.1 Runtime performance

Unsurprisingly, the voxelization runtime varies significantly between different hardware configurations and input geometry. If we consider the naive approach as the baseline, the non-branching execution path increased the performance of the voxelization algorithm without introducing new limitations. The data pre-processing approach would only occasionally increase performance, but the gained performance would be much smaller than the cost of doing the pre-processing step. The combined approach had the negative cost of the pre-processing step and did not yield significantly better performance than just the non-branching execution path one.

The results for the non-branching execution path approach align with our expectations of a small, consistent runtime performance improvement at no additional cost. The data pre-processing provided unsatisfactory runtime performance results, likely due to resulting non-uniform memory access. Even in memory-bandwidth-limited cases (GTX 1070, RX Vega 64) the performance gained was marginal compared to the cost of doing the pre-processing, and in non-memory-bandwidth-limited cases (RTX 2070) it degraded the performance significantly. The combined approach performance was dominated by the effects of the pre-processing, which was somewhat expected, since it has the bigger impact.

We note that the differences in runtime performance between different approaches were significantly bigger on the AMD GPU than on the NVIDIA GPUs,

however the non-branching execution path is still the most consistent and, usually, bigger improvement for both vendors.

5.2.2 Memory performance

The data pre-processing step reduced the memory requirement for the input geometry, however it only made a difference for the biggest XYZ RGB Dragon mesh. For other meshes, both pre-processed and non-pre-processed approaches had the same memory requirements for the 1024^3 voxel grid. The memory requirement of the whole algorithm is typically limited by the last step in the voxelization process, which depends on the number of triangles of the input mesh and the size of the output grid. Since the result grid memory footprint grows exponentially, reducing the linearly-growing footprint of the input geometry is insufficient to significantly reduce the memory limitation of the voxelization algorithm.

6 Conclusions and future work

6.1 Conclusions

Of the four approaches studied, the non-branching execution path is clearly superior. It does not introduce additional costs or bottlenecks to the algorithm and gives a consistent performance improvement over the naive approach. The data pre-processing approach we used did not give a consistent runtime performance improvement and is therefore not recommended, even if it is pre-computed on the input mesh. Doing the pre-processing incurs a significant cost that outweighs any potential performance improvement. The combined approach performance was dominated by the data pre-processing and would usually result in worse performance than just the non-branching execution path.

The memory requirement grows exponentially with the resulting grid size, therefore the reduction in the linearly-growing input geometry memory footprint is insufficient to reduce the memory limitation of the voxelization algorithm. Thus, we conclude that the vertex-reusing pre-processing technique we explored in this thesis is not fit for significantly reducing the memory requirements of the voxelization algorithm.

We conclude that the non-branching execution path is a worthwhile improvement to the voxelization algorithm, resulting in a consistently reduced runtime of the algorithm and incurring no additional costs. It is also relatively simple to implement and will likely work for other GPU-based algorithms.

6.2 Limitations

There are a few limitations regarding our study and results, including other possible improvements for GPU-based implementations of different algorithms that we did not explore, such as reducing memory bank conflicts or improving cache hit rate for the input geometry [58].

We also only tested our implementations on GTX 1070, RTX 2070 and RX Vega 64 graphics cards, which all belong to the two latest generations of graphics hardware. The improvements might differ significantly compared to older technology [38].

6.3 Future work

After implementing the different approaches and running the experiments, we found a paper detailing a method by Sander et al. [58] for improving the cache hit rate, which could have affected the data pre-processing approach performance. Unfortunately, we did not have time to incorporate it, but this could be tested in future runs.

As GPGPU hardware evolves, other improvements may arise or the ones presented might become obsolete. Also, memory and processing speeds increase at different rates, meaning reducing the proportion of one to the other might be more beneficial in the future and require a different approach altogether.

References

[1] Unlimited detail real-time rendering technology preview. Youtube video, August 2011. https://www.youtube.com/watch?v=00gAbgBu8R4. [Accessed: June 8, 2019].

[2] Dan Tabar and Branislav Siles. The inevitable volumetric future. Youtube video, September 2018. https://www.youtube.com/watch?v=JHJ4TKJom88. [Accessed: June 8, 2019].

[3] Stefan Zachow, Michael Zilske, and Hans-Christian Hege. 3D reconstruction of individual anatomy from medical image data: Segmentation and geometry processing. *Proceedings of the CADFEM Users Meeting*, January 2007.

[4] James Duncan and Gene Gindi, editors. *Information Processing in Medical Imaging*, number 15 in International Conference, June 1997. Springer-Verlag Berlin Heidelberg. ISBN 978-3540690702.

[5] Andrei Simion, Victor Asavei, Sorin Andrei Pistirica, and Ovidiu Poncea. Practical GPU and voxel-based indirect illumination for real time computer games. In *2015 20th International Conference on Control Systems and Computer Science*, pages 379–384, May 2015. doi: 10.1109/CSCS.2015.47.

[6] Miguel Cepero. Procerual voxel terrain. Youtube video, October 2010. https://www.youtube.com/watch?v=PwtKI_Cx6Dw. [Accessed: June 8, 2019].

[7] Fangyu Li, Sisi Zlatanova, Martijn Koopman, Xueying Bai, and Abdoulaye Diakité. Universal path planning for an indoor drone. *Automation in Construction*, 95:275–283, 2018. ISSN 0926-5805.

[8] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics (TOG)*, 29(6):1–10, December 2010. ISSN 1557-7368.

[9] Jacopo Pantaleoni. VoxelPipe: a programmable pipeline for 3D voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on high performance graphics*, HPG '11, pages 99–106. ACM, 2011. ISBN 978-1450308960.

[10] Randall Rauwendaal and Mike Bailey. Hybrid computational voxelization using the graphics pipeline. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):15–37, March 2013. ISSN 2331-7418. http://jcgt.org/published/0002/01/02/.

[11] Samuli Laine. A topological approach to voxelization. *Computer Graphics Forum*, 32(4):77–86, 2013. ISSN 01677055.

[12] Frederic Garcia and Björn Ottersten. CPU-based real-time surface and solid voxelization for incomplete point cloud. In *2014 22nd International Conference on Pattern Recognition*, pages 2757–2762. IEEE, August 2014. ISBN 978-1479952090.

[13] Jianguang Weng, Yueting Zhuang, and Hui Zhang. Error-bounded solid voxelization for polygonal model based on heuristic seed filling. In *International Symposium on Visual Computing*, pages 607–612. Springer, 2005. ISBN 978-3-540-32284-9.

[14] Jeroen Baert, Ares Lagae, and Philip Dutré. Out-of-core construction of Sparse Voxel Octrees. In *Proceedings of the 5th high-performance graphics conference*, pages 27–32. ACM, 2013.

[15] Cyril Crassin and Simon Green. *Octree-Based Sparse Voxelization Using The GPU Hardware Rasterizer*. CRC Press, Patrick Cozzi and Christophe Riccio, July 2012.

[16] David Luebke and Greg Humphreys. How GPUs work. *Computer*, 40(2): 96–100, February 2007. ISSN 1558-0814.

[17] Intel CPUs. Web page. `https://www.intel.com/content/www/us/en/products/docs/processors/xeon/w-3175x-brief.html?wapkw=28-core+xeon+w-3175x`. [Accessed: June 8, 2019].

[18] Kevin Krewell. What's the difference between a CPU and a GPU? December 2009.

[19] Vulkan API homepage. Web page, . `https://www.khronos.org/vulkan/`. [Accessed: June 8, 2019].

[20] Compute shaders. Web page, . `https://www.khronos.org/opengl/wiki/Compute_Shader`. [Accessed: June 8, 2019].

[21] OpenCL overview. Web page, . `https://www.khronos.org/opencl/`. [Accessed: June 8, 2019].

[22] CUDA overview. Web page, . `https://developer.nvidia.com/cuda-zone`. [Accessed: June 8, 2019].

[23] James D Foley, Andries van Dam, Steven K Feiner, and John F Hughes. *Computer graphics: principles and practice. Third Edition*, volume 12110. Addison-Wesley Professional, July 2013. ISBN 978-0321399526.

[24] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/CRC Press, 2018. ISBN 978-1138627000.

[25] Hideki Arata, Yoshiaki Takai, Nami K Takai, and Tsuyoshi Yamamoto. Free-form shape modeling by 3D cellular automata. In *Proceedings Shape Modeling International'99. International Conference on Shape Modeling and Applications*, pages 242–247. IEEE, 1999.

[26] Atomontage volumetric technology reel - april 2018. Youtube video, April 2018. https://www.youtube.com/watch?v=nr5JqYYye3w&t=84s. [Accessed: June 8, 2019].

[27] Repository of meshes used for voxelization testing. Stanford Computer Graphics Laboratory. http://graphics.stanford.edu/data/3Dscanrep/. [Accessed: June 8, 2019].

[28] Martin Pätzold and Andreas Kolb. Grid-free out-of-core voxelization to Sparse Voxel Octrees on GPU. In *Proceedings of the 7th Conference on high-performance graphics*, HPG '15, pages 95–103. ACM, August 2015. ISBN 978-1450337076.

[29] Charles Loop, Cha Zhang, and Zhengyou Zhang. Real-time high-resolution sparse voxelization with application to image-based modeling. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 73–79. ACM, July 2013.

[30] Shiaofen Fang and Hongsheng Chen. Hardware accelerated voxelization. *Computers & Graphics*, 24(3):433–442, 2000.

[31] Elmar Eisemann and Xavier Decoret. Single-pass GPU solid voxelization for real-time applications. pages 73–80, 2008. ISBN 978-1568814230.

[32] Duoduo Liao. GPU-accelerated multi-valued solid voxelization by slice functions in real time. In *Proceedings of the 24th Spring Conference on Computer Graphics*, pages 113–120. ACM, April 2008.

[33] Anne Håkansson. Portal of research methods and methodologies for research projects and degree projects. pages 67–73, Las Vegas USA, December 2013. KTH, Skolan för informations- och kommunikationsteknik (ICT), Programvaruteknik och Datorsystem, SCS, CSREA Press U.S.A. ISBN 160-1322437.

[34] Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, and Qunsheng Peng. Real-time voxelization for complex polygonal models. In *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings*, pages 43–50. IEEE, October 2004. ISBN 076-9522343.

[35] Sangjin Han. On fair comparison between CPU and GPU. February 2013.

[36] Ilija Petrovic. GCN architecture whitepaper. June 2012.

[37] Fabian Giesen. A trip through the graphics pipeline 2011, part 13. Blog post, October 2011. https://fgiesen.wordpress.com/2011/10/09/a-trip-through-the-graphics-pipeline-2011-part-13/. [Accessed: June 8, 2019].

[38] Vasily Volkov. *Understanding latency hiding on GPUs*. PhD thesis, UC Berkeley, August 2016.

[39] Sebastian Aaltonen. Optimizing GPU occupancy and resource usage with large thread groups, May 2017. https://gpuopen.com/optimizing-gpu-occupancy-resource-usage-large-thread-groups/. [Accessed: June 8, 2019].

[40] Mark Segal and Kurt Akeley. *The OpenGL® Graphics System: A Specification (Version 4.5 (Core Profile)-May 28, 2015)*, May 2015.

[41] Direct3D 11 programmable pipeline. Web page, . https://docs.microsoft.com/en-us/windows/desktop/direct3d11/overviews-direct3d-11-graphics-pipeline. [Accessed: June 8, 2019].

[42] Metal 2. Web page, . https://developer.apple.com/metal/. [Accessed: June 8, 2019].

[43] Unity: documentation of recommended polygon count. Web page, . https://docs.unity3d.com/Manual/ModelingOptimizedCharacters.html. [Accessed: June 8, 2019].

[44] Mathijs de Jonge. Example of high polygon mesh. https://twitter.com/dejongemathijs/status/647309631329538048. [Accessed: June 8, 2019].

[45] Brian Caulfield. What's the difference between ray tracing and rasterization? March 2018. https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/. [Accessed: June 8, 2019].

[46] Juan Pineda. A parallel algorithm for polygon rasterization. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 17–20. ACM, August 1988.

[47] Jon Hasselgren. Efficient compression and rasterization algorithms for graphics hardware. 2006. ISSN 1652-4691.

[48] Yafei Wang, Zhenjie Chen, Liang Cheng, Manchun Li, and Jiechen Wang. Parallel scanline algorithm for rapid rasterization of vector geographic data. *Computers & geosciences*, 59:31–40, 2013.

[49] Jordi Roca, Victor Moya, Carlos Gonzalez, VicenteEscandell, Albert Murciego, Agustin Fernandez, and Roger Espasa. A SIMD-efficient 14 instruction shader program for high-throughput microtriangle rasterization. *The Visual Computer*, 26(6-8):707–719, 2010.

[50] Daniel Cohen-Or and Arie Kaufman. Fundamentals of surface voxelization. *Graphical Models and Image Processing*, 57(6):453–461, 1995. ISSN 1077-3169.

[51] Hanan Samet. An overview of quadtrees, octrees, and related hierarchical data structures. In *Theoretical Foundations of Computer Graphics and CAD*, pages 51–68. Springer, 1988.

[52] Samuli Laine and Tero Karras. Efficient Sparse Voxel Octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, August 2011. ISSN 10772626. doi: 10.1109/TVCG.2010.240.

[53] Blender. Web page, . `https://www.blender.org/`. [Accessed: June 8, 2019].

[54] 3ds max. Web page, . `https://www.autodesk.com/products/3ds-max/overview`. [Accessed: June 8, 2019].

[55] ZBrush. Web page, . `http://pixologic.com/`. [Accessed: June 8, 2019].

[56] Turbosquid 3D model repository. Web page, . `https://www.turbosquid.com/`. [Accessed: June 8, 2019].

[57] Piotr Bialas and Adam Strzelecki. Benchmarking the cost of thread divergence in cuda. In *International Conference on Parallel Processing and Applied Mathematics*, pages 570–579. Springer, 2015.

[58] Pedro Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (TOG)*, 26(3):89–es, 2007. ISSN 1557-7368.

Appendix

Raw data

The raw, aggregated, and graphed data gathered from the experiments can be found at:
`https://drive.google.com/open?id=1SDkuNLgjfBDIEu_Qk4eUlsVdyPChzYjy`

TRITA TRITA-EECS-EX-2019:166