Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

4-18-2019 2:30 PM

Haptics-enabled, GPU augmented surgical simulation platform for glenoid reaming

Vlad Popa, *The University of Western Ontario*

Supervisor: Tutunea-Fatan, Ovidiu-Remus, *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Master of Engineering Science degree in Biomedical Engineering
© Vlad Popa 2019

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Medical Education Commons, Software Engineering Commons, and the Systems Architecture Commons

Recommended Citation

Popa, Vlad, "Haptics-enabled, GPU augmented surgical simulation platform for glenoid reaming" (2019). *Electronic Thesis and Dissertation Repository*. 6143.
https://ir.lib.uwo.ca/etd/6143

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

Abstract

Surgical simulators are technological platforms that provide virtual substitutes to the current cadaver-based medical training models. The advantages of exposure to these devices have been thoroughly studied, with enhanced surgical proficiency being one of the assets gained after extensive use. While simulators have already penetrated numerous medical domains, the field of orthopedics remains stagnant despite a demand for the ability to practice uncommon surgeries, such as total shoulder arthroplasty (TSA).

Here we extrapolate the algorithms of an inhouse software engine revolving around glenoid reaming, a critical step of TSA. The project's purpose is to provide efficient techniques for future simulators, and the methods developed address the challenges of achieving real-time performance with high-volume computations and haptics input rates. The core of the engine revolves around the management and manipulation of voxels, which handle the representation of virtual objects, the collision between them, and the removal of material upon interaction. A partitioning ("Chunk") system was implemented for performant voxel organization and collision handling. Compared to object-wide single voxel buffers or 3D textures, chunks enable empty-space memory savings and optimized collision testing through region isolation. Overall, the engine can replicate the interaction between a ~30 million voxel scapula and a drill at 60 Hz visual, 1 kHz haptics, and 333 Hz collisions. We anticipate that the techniques developed will further the development of current and future simulators.

Keywords: Virtual Reality, Surgery, Simulator, Engine, Haptics, Voxels, Collision Detection, GPGPU

Acknowledgments

I express my humblest and most sincere thanks to my supervisor Dr. Ovidiu-Remus Tutunea-Fatan, whose guidance and direction have enabled the realization and success of this project. I am grateful for the opportunity provided by him, which has led to the development of the engine, and the improvement of my skillset as a programmer. I would also thank my colleague, Dr. Reza Faieghi, whose contributions to the project have shaped the project to where it is today, achieving its goals of real-time material removal.

Contents

Abstract ii

Acknowledgments iii

List of Figures vii

List of Tables ix

List of Algorithms x

List of Abbreviations, Symbols, and Nomenclature xi

Introduction 1

1.1 Importance of practicing surgical operations 1

1.2 Haptics-augmented VR surgical simulation platforms 2

1.3 Surgical simulators in clinical settings 2

1.4 The shoulder joint 3

1.5. Glenoid reaming in total shoulder arthroplasty 5

1.6 Motivation and project goals 6

1.7 Simulation engine techniques and development 6

1.7.1 Voxels for model representations 6

1.7.2 The chunk system 7

1.7.3 Entity-component system 8

1.7.4 Custom software development 9

1.7.5 Model Voxelization 10

1.7.6 Computed tomography scan data processing 11

1.7.7 Graphics rendering techniques 12

1.7.8 Engine loops and collision detection algorithms.. 15

1.8 Specific aims and thesis outline. 17

Engine Specifics.. 18

2.1 Initialization and object design 18

2.2 Chunks and the chunk system 20

2.2.1 Chunk voxel grid dimensions reasoning 22

2.3 Model voxelization and STL file to voxel grid conversion 22

2.4 Chunk grid construction using computed tomography scans.. 24

2.5 The haptics loop 27

2.6 The render loop 31

The chunk system 33

3.1 Memory optimizations 33

3.1.1 Against single buffer/texture data structures 33

3.2 Material removal 37

3.2.1 Collision testing with chunks 37

3.2.2 Baseline chunk collision performance with various tools and sizes 39

3.2.3 Chunk system collision testing performance compared to single-buffer-based algorithms... 45

3.2.4 Chunk system complexity 50

3.2.5 Chunk system vs. octrees 51

3.2.6 Independency from tool voxel resolution 52

3.3 Chunk voxel rendering 52

3.3.1 Neighbor-aware triangle mesh generation 52

3.3.2 Ray-box intersection voxel rendering 54

3.3.3 Comparison of the voxel rendering methods 56

Thesis closure 63

4.1 Summary 63

4.2 Future work 64

4.3 Conclusion 64

Bibliography 66

Curriculum Vitae 80

List of Figures

Figure 1.1: Replacement of the glenohumeral joint in TSA 4

Figure 1.2: Process of glenoid resurfacing 4

Figure 1.3. Scapula bone voxel representation. 12

Figure 1.4: Voxel representations of (a) cellular foam, (b) trabecular bone core 2, (c) trabecular bone core, (d) scapula, and (e) glenoid samples, which were used for tests. 15

Figure 2.1 Chunk partitioning system of voxelized scapula bone. 20

Figure 2.2: Before (A) and after (B) "*RectGridToCubicGridTransfer*" processing.............. 25

Figure 2.3: OOBB intersection test and intersecting chunks identification........................... 30

Figure 2.4: Flowchart of the engine's render and haptics loops. 31

Figure 3.1: Visual representation of active (red) vs. inactive (grey) chunks in generated Scapula chunk grid.. 34

Figure 3.2: Active voxels stored in memory across test models and data structures............. 36

Figure 3.3: Visualization of the chunk system collision testing mechanism. Chunk voxels (voxels in red WF) are iterated and compared against tool voxels.. 38

Figure 3.4: Visualization of tool-to-bone voxel collision testing. Tool voxels (green WF) are iterated through, transformed, and collision checked against bone voxels (red WF)............ 39

Figure 3.5. Demonstration of the real time collision and material system. 39

Figure 3.6: Static collision event using reamer tool for testing effect on performance of chunk sizes during collision testing... 40

Figure 3.7: Static collision event using peg-hole cylinder tool for testing effect on performance of chunk sizes during collision testing... 41

Figure 3.8: Chunk collision task computation time using the peg-hole cylinder tool with varying PC configurations 44

Figure 3.9: Voxel processing time of various collision algorithms on Windows 10, Intel i5 8600, Nvidia GTX 1070 PC. Chunk collision was performed with reamer. 46

Figure 3.10: Chunk system collision timings using peg-hole cylinder tool vs Reza et. al's timings at varying tool and model voxel resolutions. 48

Figure 3.11: Chunk system collision timings using peg-hole cylinder tool against those measured when using Zheng et. al's method 49

Figure 3.12: Chunk system collision timings using peg-hole cylinder tool against those measured when using Yau et. al's method.. 49

Figure 3.13: 720p render timings for triangle mesh voxel rendering of 32^3 chunks across tested models on Windows 10, Intel i5 8600, Nvidia GTX 1070 PC.................................... 57

Figure 3.14: 720p render timings for GL_POINT based voxel rendering of 32^3 chunks across tested models on Windows 10, Intel i5 8600, Nvidia GTX 1070 PC 58

Figure 3.15: Ratio of active to inactive voxels across tested models 59

Figure 3.16: 720p render timings for GL_POINT based voxel rendering for "Cellular Foam" model across varying Windows 10 PC configurations 61

List of Tables

Table 2.1: Timings (in ms) of model voxelization for models tested 23

Table 2.2 Timings (in ms) of CT scan chunk grid construction phases for models tested. 26

Table 3.1: General voxel grid characteristics of the CT scanned objects used for testing 35

Table 3.2: Chunk grid characteristics of the CT scanned objects used for testing 35

Table 3.3: Static collision event timings for various chunk dimensions using the reamer..... 41

Table 3.4: Static collision event timings for various chunk dimensions using peg-hole cylinder tool 42

Table 3.5: Static collision event timings at chunk dimensions of 32^3 using the Reamer tool while varying tool voxel-sizes 42

Table 3.6: Specifications for the Windows 10 PCs used over the course of engine testing ... 43

Table 3.7: Various statistics of the voxel rendering methods used....................................... 56

Table 3.8: Breakdown of the runtime for different phases of chunk triangle mesh generation 60

List of Algorithms

Algorithm 1.1: 1D index to 3D grid position (and reverse) calculations................................. 8

Algorithm 2.1: Calculation of STL model voxel grid properties... 23

Algorithm 2.2: CT scan voxel data structure 24

Algorithm 2.3: Calculation of parameters for rectangle and cubic CT scan voxel grid 24

Algorithm 2.4: Raw data OpenCL kernel pseudocode 24

Algorithm 2.5: Rectangle grid to cubic grid voxel transfer OpenCL kernel pseudocode 26

Algorithm 2.6: Cubic voxel grid to chunk voxel buffer OpenCL kernel pseudocode........... 26

Algorithm 2.7: Entity collision check pseudocode 28

Algorithm 2.8: Entity intersection volume calculation pseudocode 29

Algorithm 2.9: Min/max intersecting chunk coordinates calculation based on intersection volume... 29

Algorithm 2.10: Intersection chunk collision detection OpenCL kernel setup pseudocode... 29

List of Abbreviations, Symbols, and Nomenclature

AABB	Axis Aligned Bounding Box
CPU	Central Processing Unit
CT	Computed Tomography
GPGPU	General Purpose Graphic Processing Unit
GPU	Graphics Processing Unit
OOBB	Object Oriented Bounding Box
OpenCL	Open Computing Library
OpenGL	Open Graphics Library
RAM	Random Access Memory
SAT	Separating Axis Theorem
TSA	Total Shoulder Arthroplasty
VAO	Vertex Array Object
VOI	Volume of Interest
VPS	Voxmap Point-Shell
VR	Virtual Reality
VRAM	Video Random Access Memory
WF	Wireframe

Chapter 1

Introduction

1.1 Importance of practicing surgical operations

Continuous repetition of surgical training programs has been thoroughly demonstrated to increase the patent care of participating surgeons. Indeed, there exists a concise correlation between the successful outcome of a procedure and the familiarity that a surgeon has with an operation [1-3]. Furthermore, studies investigating the relationship between a surgeon's intimate knowledge of an operation and the rate of medical errors have verified that mortality rates dramatically decrease when a surgeon has previously encountered the surgical task to some capacity [4-7]. Clearly, surgical proficiency requires practice to obtain, as with any skill. Regrettably, a lack of skill is the leading cause of patient death [6-11]. There is thus a growing emphasis that surgeons should train whenever the opportunity presents itself [12, 13].

Surgical training extends beyond menial motor exercises, however. While performing mock procedures on synthetic models may enhance one's dexterity, for instance, human bodies are complex, and providing a realistic learning environment is complicated [14]. Traditional cadaver-based models (animal/human) and live patients currently fulfil said role, though these options are limited, costly, and pose safety risks [15]. Work hour restrictions, where surgeons aren't at times available, also impede surgeons from obtaining the experience they desperately need [12]. Considering the nature of these setbacks, a suitable alternative must provide a platform where surgeons may improve their competency at consistent intervals, at their discretion, and at a reasonable degree of realism. Surgical simulators are believed to be a solution that addresses all concerns [16].

1.2 Haptics-augmented VR surgical simulation platforms

Surgical simulators are advanced technological platforms that provide realistic virtual training models to medical professionals [17-19]. Through software and hardware peripherals, these simulators mimic the hospital environment within virtual reality and replicate the operating room experience for its users [20]. Being virtual, simulated operations are customizable, repeatable, and performed at the discretion of the user, all of which are advantages over traditional cadavers [21, 22]. There are two major components that encompass surgical simulators: the haptics-enabled hardware tool peripherals, and the software rendering engine [23]. Haptics devices provide the interface for which the user interacts the virtual space with, and they enable tactile information to be relayed back to the user, giving the sensation of a real operation. The visualization of the virtual space is controlled by a computer workstation running an advanced software physics and graphics rendering engine [24].

1.3 Surgical simulators in clinical settings

Modern VR surgical simulators are designed to conform to a narrow scope of use cases and specialize to deliver specific procedures rather than be a general-purpose tool that encompasses the entirety of the human body. Essentially, surgical simulators are finely turned to accelerate competency of problematic surgeries and provide precise qualitative feedback on a step-by-step basis [16, 24]. There are numerous examples of simulators that are extensively used in various fields, and several studies have reported an upsurge in residential program efficiency as a result [25, 26]. The Karlsruhe Endoscopic Surgery Trainer, a VR laparoscopy simulator, is a device capable of roughly imitating the human abdomen, and has been implemented with great success at the University Hospital of Tuebingen since 1996 [21]. A Rhinoplasty simulator created by Lee et al. utilizes image processing to facilitate the study of patient facial structures, ultimately easing pre-operative planning for plastic surgery [27]. Within dental care, amongst many examples [28-30], Wu et al. have written a VR simulator that models the process of dental grinding [31]. Finally, a few contemporary companies offer commercial solutions that incorporate pre-operative planning, VR patient data viewing, and psycho-motor skill education modules [32]. Together, these products mark a growing trend in the medical field where technology is being leveraged to assist the advancements made towards surgical procedures [33].

1.4 The shoulder joint

While there are a variety of simulators in circulation, the most common being for laparoscopy [34-38], there are few that address orthopaedics, which results in the field relying on the apprenticeship model for training [39]. With orthopedic surgeons experiencing setbacks, invested development in orthopaedic simulators would greatly benefit doctors [40-45]. Current work focuses on minimally invasive operations (arthroscopy), since such tasks are easier to program compared to more complex operations [46]. Even so, there is interest in developing simulators that replicate uncommon surgeries relating to arthritis [47-49]. As of 2013, over 15% of Canadians aged 15 and up were diagnosed with some form of arthritis [50]. However, despite an incidence rate increase, shoulder replacement surgeries remain infrequent enough that residents have few opportunities to observe or practice [51]. With bone machining simulators making strides within the field [24, 52], total shoulder arthroplasty (TSA) is a prime target to be addressed by surgical simulators [53], and is the focus of this project.

TSA is a medical procedure that restores shoulder joint function to patients who have experienced trauma or are suffering from glenohumeral arthritis. The shoulder is made of muscles, the Humerus, the shoulder blade (Scapula), and the collarbone (Clavicle) [54].

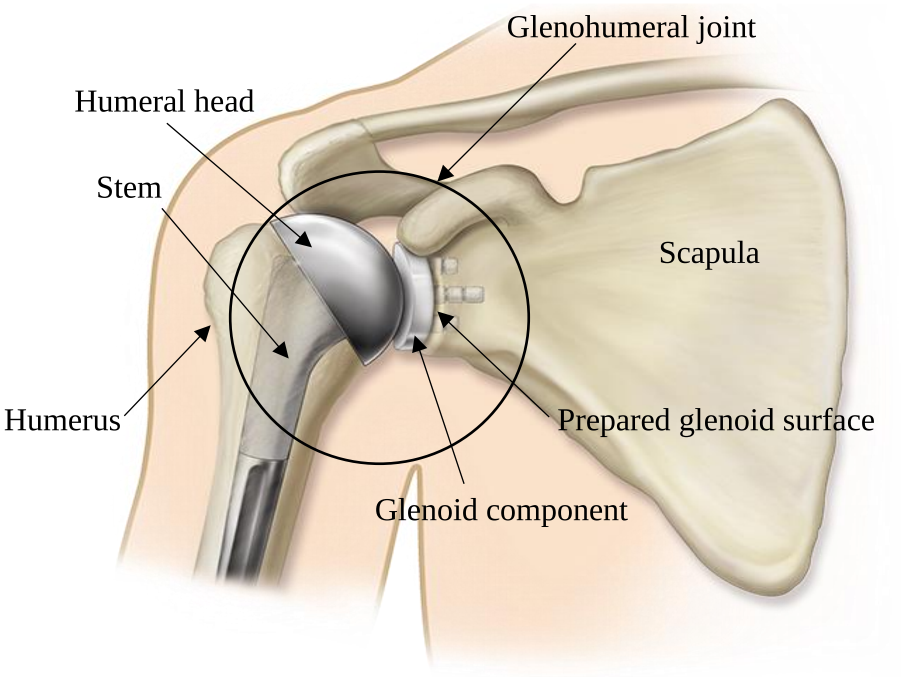


Figure 1.1: Replacement of the glenohumeral joint in TSA

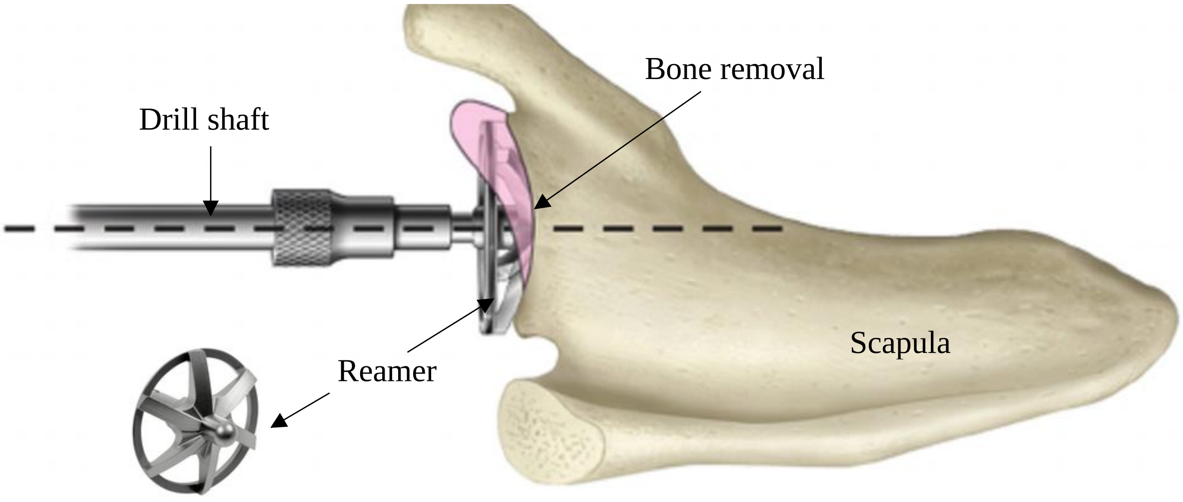


Figure 1.2: Process of glenoid resurfacing

Pathological disease usually occurs at the Glenohumeral joint (GHJ) of the shoulder (Fig. 1.1). When a patient is afflicted with osteoarthritis, the articular cartilage lining the surface of the humerus and the glenoid of the Scapula wears down and erodes [55]. Without a frictionless surface between them, the bones experience direct contact, and the resulting rubbing between the two leads to discomfort and pain [54, 56]. To correct this, the diseased portions of the bones are removed, and artificial implants are inserted into the humerus and glenoid [57, 58]. Correct placement and fixation of the implant into the cavity of the glenoid is critical to restoring the joint, avoiding loosening or breakage, and preventing further patient distress [59-65].

1.5. Glenoid reaming in total shoulder arthroplasty

During TSA, prosthetic replacement of all or part of the glenohumeral joint is performed in two major steps. First, the glenoid component of the implant is inserted into a resurfaced glenoid cavity, and second, the humeral component of the implant is lodged into a hollowed out humerus bone (Fig 1.1 – 1.2) [59, 66, 67]. The difficulty of the operation lies in preparing the glenoid surface through a process known as glenoid reaming [59]. The task involves the removal of 2 mm of glenoid cartilage layer, and then gently removing the subchondral plate without touching the cancellous bone. This resurfacing, performed with a drilling device known as the reamer, ends in a convex surface that holds the glenoid implant in place (Fig. 1.2). Prior to this procedure, however, a peg-hole is drilled into the glenoid cavity to assist in guiding the reamer [54, 59]. The problem here lies in the visualization. Due to the interposed position between the reamer and the glenoid, there is limited sight of the tool-bone interface during reaming [63, 68-71]. Combined with the necessity of correct implant application, glenoid reaming requires skill and practice.

1.6 Motivation and project goals

Two factors contributed to the motivation behind this project. Firstly, the practical goal of the engine was to simulate, in real-time and to a degree of accuracy, the process of glenoid resurfacing, such that surgeons can use the simulator to prepare for the task. Secondly, the methods and algorithms developed over the course of the project should positively contribute to existing and future simulator code bases, increasing their efficiency and performance. One of the necessities for accomplishing these requirements is the addition of force feedback through an input device [72-77]. Given this restriction, on top of delivering real-time rendering, haptics was also integrated into the engine.

1.7 Simulation engine techniques and development

1.7.1 Voxels for model representations

The engine developed for this project was designed to achieve a 1:1 correspondence between user input via a haptics device and the movement and interaction of the virtual objects being rendered on the screen. This simultaneous update between cause and effect is known as real-time [78-80]. More precisely, the term “Real-time” signifies that the visual refresh rate of the engine matches the monitor’s refresh rate (i.e.: 60 Hz), and that the update/input refresh rate matches that of the haptics input device, which is typically 1 kHz [74, 81, 82]. Achieving high frequencies is a common and challenging bottleneck for simulation engines as any code executed during the program’s runtime must perform at sub-millisecond speeds. To address the demands, developers have borrowed design patterns from game engines, which typically contain the latest advancements in graphics and in handling numerous detailed objects in virtual space [83]. The normal approach to representing objects in most surgical simulators, including this one, is voxels [84-87]. Voxels are discrete units of space containing metadata that describes the properties of the space the voxel occupies [88]. For visualization purposes, the object being represented by voxels can be perceived as one made entirely out of a set cubes, with some that are filled, and some that are empty. Voxels contain several advantages over alternative means of representing virtual objects. Voxels are easily queried given their grid-like structure, and any point of interest can be quickly identified. Compare this to triangle meshes, whose vertices must be arranged according to a node-map prior to any sort of search [89]. When it comes to collision detection, this ability translates into being able to easily identify areas where an intersection takes place. Furthermore, once a change occurs during an intersection, the difference can be visualized by simply changing the metadata of the voxels affected. With this new identifier, the rendering system can then choose whether to continue displaying the voxels on screen or remove them from view if it has detected that they are no longer supposed to be there (i.e.: the voxels now represent void space). Contrast this simplicity with that of triangle meshes, where not only do all the vertices that are in the intersection volume need to be identified, but must be re-triangulated to reflect the change, involving mathematics more complex than simply changing a density value [30, 89-91].

1.7.2 The chunk system

Of course, voxel representations of models need to be stored in memory and organized to some sort of custom data structure. Existing engines utilize either a single buffer or a 3D texture to house voxels [84, 92-94]. For 3D textures, they have the advantage of already being laid out as a 3D matrix, wherein a voxel can be queried by supplying a relative XYZ coordinate to the texture. For a 1D single buffer, the voxels may either be stored sparsely, where all model voxels are compacted into a linear array, but each contain metadata that indicates their position, or arranged according to a 3D lattice using a 3D XYZ to 1D index mathematical conversion algorithm (Alg. 1.1).

```
inline int3 IndexToZYXGridPos(int i, int3 dims) {
	return (int3)(i % dims.x, (i / dims.x) % dims.y, i / (dims.x * dims.y));
}

inline int ZYXGridPosToIndex(int3 pos, int3 dims) {
	return pos.x + (pos.y * dims.x) + (pos.z * dims.x * dims.y);
}
```

Algorithm 1.1: 1D index to 3D grid position (and reverse) calculations

The latter contains what are known as “empty” or “void” voxels, voxels that represent empty space in the 3D matrix where there is no presence of model material. Either way, these data structures are self contained into a single span of memory. The main feature of the engine developed for this project is the deviation from this norm, and the introduction of an alternative means of storing and processing voxels. The system is a partitioning scheme that subdivides uniform portions of the voxel representation into a grid of large units known as “Chunks”. Chunks are subsections of the overall voxel representation containing 32x32x32 voxels each. Like voxels, chunks are arranged as an axis-aligned 3D grid that encompasses the entirety of the model they represent. The chunk system was introduced to address two main concerns with voxel-based model dynamics. First, to reduce the increasing memory cost of higher resolution voxel models, and second, to lower the amount of computations required to process a collision event. The results of this system are described in chapter 3.

1.7.3 Entity-component system

Other techniques have also been adapted from game engines for the benefit of surgical simulators. The entity component system is one such example of a typical game engine optimization that has been ported over to this project. Entities are essentially scaffolds containing objects (“Components”) which house generic behaviours and properties that describe entities. The combination of components within an entity describe how an entity behaves and how it’s rendered in virtual space. Entities allow for modularity and easy customisability when creating simulation objects [83].

1.7.4 Custom software development

Performance and efficiency, as opposed to graphical fidelity, remained a priority for this project. The work performed remains minimal in the aspect of rendering virtual objects and instead focuses on the interaction and behaviour between objects. There exist game engines, such as the Unreal Engine, that have been utilized in the context of VR research [83]. However, such platforms have limited support for voxel-based physics, collisions, and visual rendering. Furthermore, performance enhancements using GPGPU means (ex: OpenCL) are not natively supported and require a non-trivial plugin to add functionality [83]. Creating an in-house engine from scratch allows the engine to be specialized for voxels and ends up being much easier to implement custom behaviours without having to learn the intricacies of a commercial engine beforehand.

C++ was chosen as the engine’s language as C++ enables finer grained tuning for performance optimizations, and it offers a plethora of 3rd party libraries to assist in development. Features such as manual memory management, classes, and access to APIs that are native to the platform, ease the development of graphically rich applications. These libraries include the Open Graphics Library (OpenGL), Open Compute Library (OpenCL), and OpenHaptics [95-97]. OpenCL, and its Nvidia counterpart CUDA, are APIs that enable general purpose programming on graphics cards. The main advantage is the notable speed up in task execution from using thousands of GPU cores in parallel [98, 99]. The choice to use the graphics card was made primarily due to the properties that are inherent to voxels [100].

Being discrete and grid-like, voxels are mostly independent entities from one another, and thus can be worked upon individually. As such, work involving voxels can be spread out across numerous cores/threads, of which the GPU has far more of than the CPU [101]. The result is a speed up in compute times when performing voxel-related tasks on the GPU compared to the CPU. OpenCL was chosen given its hardware support. Whereas CUDA runs only on Nvidia made graphics cards, OpenCL supports Nvidia, AMD, and Intel products, and is supported on all 3 major operating systems, Windows, Mac OSX and Linux. GPU APIs have also already been used in simulators with great effect, making writing software that takes advantage of the graphics card worth the effort [99, 102-107]. The visualization toolkit (VTK) is a C++ library that contains functions relating to the management and manipulation of voxels. It was considered briefly in early versions of the engine. However, it was ultimately dropped since creating the engine from scratch would enable precise development of the chunk system and allow a dependency free generic approach for voxel manipulation to be developed for current and future surgical simulators to use.

1.7.5 Model Voxelization

For voxel-voxel collision detection and material removal, the models that are to be used in the simulation must have a backing data structure that holds a voxelized representation of the model itself, which is queried and worked on during collision testing. Data for the models comes from various files. For bone models, their data come from computed tomography (CT) scans, which, while still requiring processing, are already in a voxel-like format where each line in the file represents 1 voxel. As such, the transition from file to voxel grid representation is straightforward. For the tools used in the simulation, however, their data is derived from STL files. STL is a file format that describes the surface geometry of a 3D object. Essentially, an STL file contains the triangulated surface of a model without any additional information such as color or texture UVs. When generating a voxel representation from an STL file, the algorithms described here are used [108, 109]. ASSIMP, a model extraction library, is used to read the file and obtain the model vertices, normals and indices. This data is also transferred to OpenGL buffers so that the triangle mesh of the model can be rendered during scene drawing. The triangle mesh, as opposed to the voxel representation, is drawn because the tool model's geometry remains static over the course of the simulation. This is opposite of the bone's geometry, which gets continually resurfaced. As such, there is no need to regenerate the tool's triangle mesh, so it gets used to save compute cycles as static triangle meshes are faster to compute than voxel geometries (see Chapter 3, 3.3).

1.7.6 Computed tomography scan data processing

Bone models derive their data from CT scans, and thus, have a more straightforward means of generating a voxel representation given that CT scans already come in the form of voxels. The main process is to convert the linear array of voxels in the file into a virtual 3D voxel grid. There are several steps to do so. The first is to read the data from the CT scan text file into a dynamic array of a custom data structure containing the voxel's 3D position vector and density. The next is to convert the positional data and density into a 3D matrix that is equal to the computed bounds of the CT scanned object. The 3D matrix is the voxel grid of the model, where each voxel is an int16_t containing the density. Transferring the CT scanned voxel to this grid requires subtracting the voxel's position with the minimum computed position in the object and then dividing by the CT scan voxel size. From there, the position is floored to the nearest integer, and then converted to a 1D index, which becomes the voxel's index in the 3D matrix (Alg. 1.1). With voxels being in a 3D matrix, it becomes easier to query their 3D position relative to the grid by using an index conversion algorithm. Overall, this saves 12 bytes of memory per voxel since storing a 3-component float vector is no longer required.

The next step is to convert this 3D matrix to the chunk grid, which is essentially performed through a series of range checks and subdivisions (see Chapter 2).

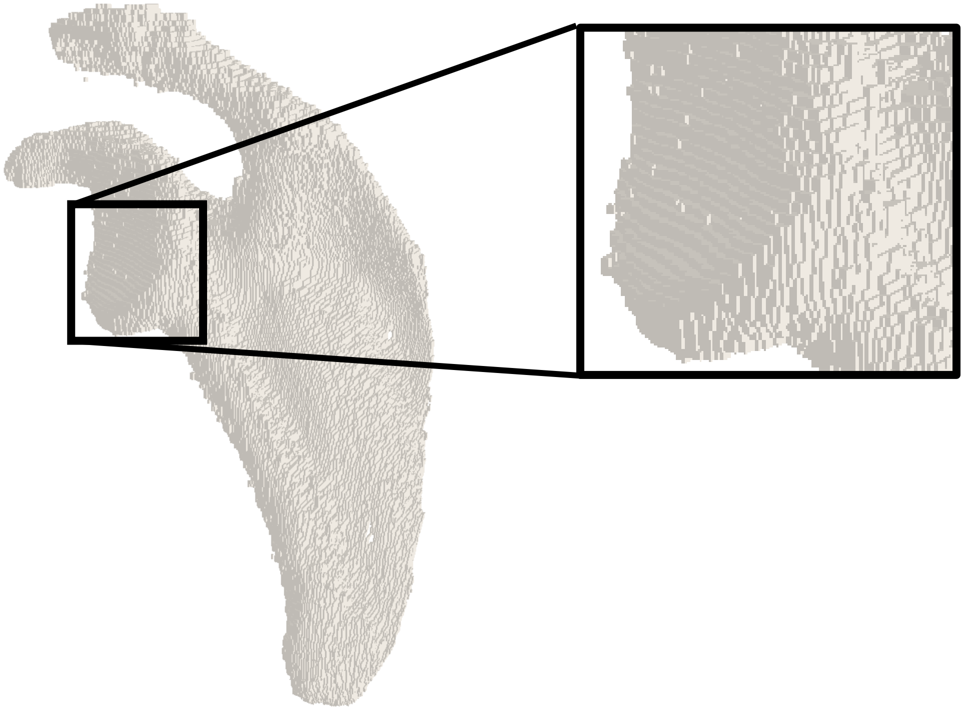


Figure 1.3. Scapula bone voxel representation.

1.7.7 Graphics rendering techniques

To achieve the target visual refresh rate of 60 Hz, a variety of rendering techniques were investigated. As the virtual representation of the tools remains static throughout the simulation, a simple triangle mesh shader was enough to draw them to the screen. To set up the tools, mesh data from STL files were loaded, using the ASSIMP importer library, into memory, and the vertex positions, normals and indices of the triangles were transferred into OpenGL buffers. At scene drawing, these buffers were bound, and a shader was executed. For each vertex, the gl_Position was computed by multiplying the camera's view-projection matrix, the model matrix, and the vertex position. The model's shadow was computed using the vertex normal and a directional lighting algorithm [110, 111]. Finally, the color was set stainless steel.

A different approach had to be taken for the representation of the bone model. Unlike the tool models, the bone's model would be subject to arbitrary and dynamic deformation over the course of the simulation, as material would be removed by the tool. As such, an algorithm would be required that could quickly update the visual representation of the bone model such that the engine could maintain a visual refresh rate of 60 Hz. Furthermore, the techniques to be used had to be compatible with the chunk system. Being a partitioning system, the algorithm would have to render each chunk individually, while still maintaining visual coherence across all chunks. Essentially, the final image would have to look as one continuous model that resembled, to some degree of accuracy, the bone it's representing.

A triangle mesh generation algorithm was the first to be introduced into the engine to render chunks. To summarize, the method would generate a series of vertices that combine to create cubes, where 1 voxel would result in one cube. Together these cubes would form the "voxelized" shape of the object (Figure 1.3). As an optimization, the algorithm would remove obscured cube faces (i.e.: faces that were not adjacent to a void/empty voxel), which would dramatically reduce the number of vertices to be rendered and speed up object drawing. The cubic mesh would be generated on a chunk-by-chunk basis and would take into consideration neighboring chunks. In observing neighbouring chunks, a continuous mesh could be generated, and the meshes would appear seamless when combined. While the method would produce the desired visual result, the issue would be the time required to regenerate chunk meshes after a collision event. Ultimately, any number of chunk mesh regeneration queries per visual refresh tick would come at a cost and would be provably slower than the alternative currently used.

Recently, a new technique has been developed to allow for direct rendering of voxel models using an OpenGL feature that enables the drawing of points with certain commands

[112]. The method, referred to as the GL_POINT based method, is as follows. For each voxel to be rendered, a box is created at the world coordinate of the voxel (transformed with the model matrix of the model it represents and the camera projection and view matrices), and then rasterized. Next, for every pixel of the rasterized box, a ray is traced through it. If the ray intersects the box at the pixel, then the pixel is drawn to the screen with the correct lighting based on the face of the box hit. The end visual result is the same as if using the triangle mesh generated from voxels. The key difference is that creating the final picture using this method does not require any triangle meshes to accomplish, but instead only the voxel buffer that you want to render. As such, the intermediary process of creating, and more importantly regenerating, a triangle mesh is completely skipped, drastically improving performance when the engine is under load via collision testing. The results of using this method over the traditional triangle mesh method is recounted in chapter 3, 3.3.

Other rendering techniques were investigated, such as generating triangle meshes using marching cubes or performing voxel-based volume rendering. However, marching cubes presents the same mesh regeneration problem, and the ray-tracing through volume rendering could not be adequately adapted for the chunk system. The key problem is that the ray-tracing of volume rendering would have to occur multiple times, once per chunk. This in contrast to the normal use of volume rendering, which is performed once per visual tick, as the voxels are stored in a singular buffer/3D texture and the rays would be able to check against every voxel in one pass [113]. The cost of performing the ray-tracing multiple times per frame would be very expensive, and so volume rendering was not further investigated.

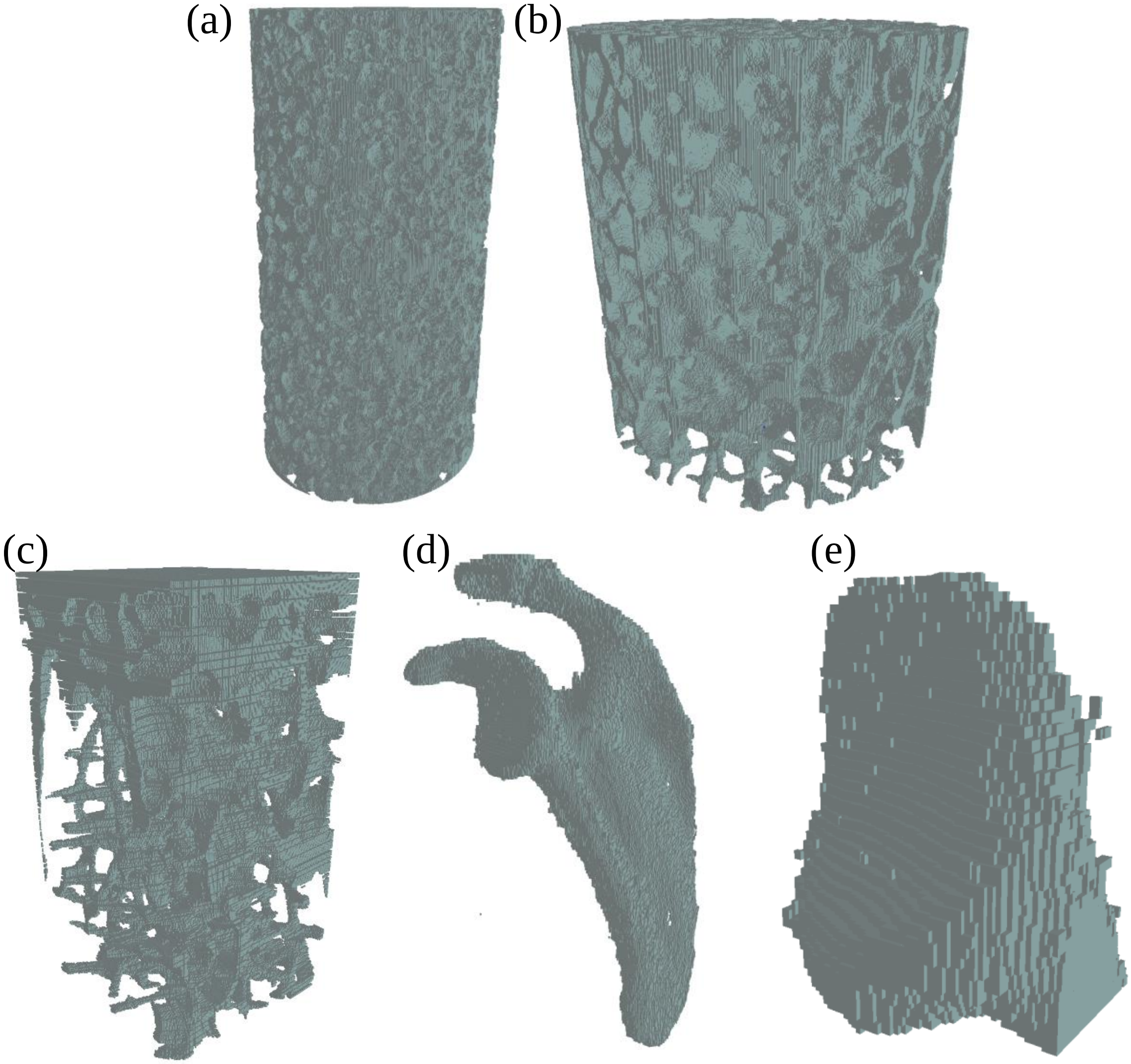


Figure 1.4: Voxel representations of (a) cellular foam, (b) trabecular bone core 2, (c) trabecular bone core, (d) scapula, and (e) glenoid samples, which were used for tests.

1.7.8 Engine loops and collision detection algorithms

The engine consists of two main loops, the render loop and the haptics loop, which are initiated once all simulation models are loaded and connected to entities, the shaders are created, the camera is created, and various uniform buffers are generated. The uniform buffers include various properties about the current scene, such as the camera's projection and view matrices, the model matrix of the current entity being rendered, and variables for directional lighting (i.e.: light direction, intensity, ambiance). The render loop updates the virtual camera based on user input and makes draw calls to render the virtual objects on screen. The haptics loop collects input data from the connected PhantomOmni haptics device and applies the device transformation matrix to the current tool being used. Doing so, there is a 1:1 correspondence between the user's motion with the device and the movement of the tool being displayed. After the tool is transformed, the engine makes an OOBB-OOBB intersection check between the tool and the bone. If the check passes, then an intersection volume is calculated, and the engine moves on to collision testing. The collision algorithm used to determine intersection voxels is based on an algorithm developed by Reza et al., and adapted to work with the partitioning system created for this engine [114]. The adapted collision algorithm runs reasonably well and can reach a stable 333 Hz update rate on a Windows 10, GTX 1070, Intel i5 8600, 8 GB RAM work station PC. Several tests were performed to assess the effectiveness of the chunk system against competing voxel-based collision testing algorithms. The algorithms analyzed were those developed by Zheng et. al, Yau et. al, and Reza et. al [75, 114, 115]. Each of these algorithms share 2 properties that contrast the voxel partitioning scheme of the engine. Firstly, these algorithms work on single 1D sparse voxel buffers that contain either the entire tool or bone voxel representation. This engine, on the other hand, subdivides the bone model into multiple discrete buffers, but still retains 1 voxel buffer for the tool. Secondly, the competing algorithms iterate from the tool voxels to the bone voxels. When performing collision testing, intersecting voxels must be identified, and the voxels from the bone must be updated to reflect any collisions that occur. When checking for intersections, one method is to take the coordinates and metadata of the tool voxels, transform them to the object space of the bone voxels, and then check for voxel-voxel overlap, which is what the single buffer algorithms do. The opposite procedure, transforming bone voxels to the tool's object space, is what this engine does, due to the chunk system implementation. The results of the two philosophies on voxel-based collision testing are displayed in Chapter 3.

1.8 Specific aims and thesis outline.

Overall, the engine was created:

- To simulate the complex interactions that occur during glenoid reaming, from the creation of the peg-hole to the resurfacing of the glenoid via a reamer.
- To develop techniques for current and future simulators that assist in the realization of real-time simulation of virtual surgeries.
- To develop techniques that optimize memory management of voxels and boost the performance of high-volume computations commonly associated with voxel-based tasks, specifically with collision testing and material removal.

In the end, the engine was able to succeed in achieving real-time speeds using a combination of a GL_POINT based voxel rendering technique and utilizing a voxel partitioning and management system known as the chunk system. Chunks usually had a smaller memory footprint compared to single buffers, and enabled performance optimizations that were only possible though partitioning a CT scanned object into discrete sections. We anticipate that the techniques and observations found over the course of the development of the engine will provide a valuable foundation and guide for future surgical simulators. Note that while this engine is focused on glenoid reaming, the loading of bones and tools is arbitrary, meaning that this engine is applicable to any joint.

Chapter 2

Engine Specifics

2.1 Initialization and object design

On initialization, the engine creates a Win32 window, initializes OpenGL 4.4 via GLEW 2.0, creates an OpenCL/OpenGL shared context, initializes objects relevant to the glenoid reaming simulation, and then launches a haptics loop and a rendering loop. The objects used for the simulation are the scapula bone, the head of a reaming drill (reamer), and a cylinder used to make a peg-hole in the glenoid. Before detailing the engine architecture any further, a note must be made on the design of the objects.

Interactable objects (i.e.: scapula, reamer, cylinder) are derived from the “Entity” class. Entities, within the context of the engine, are scaffolds that contain “components”, which are data structures that describe an object’s properties and behaviours within virtual space. By default, all entities contain 4 components, those being: an object-oriented bounding box (OOBB) component, an input component, a triangle-mesh component, and a Voxel-Grid (VG) component. The OOBB contains vectors and matrices that hold the box’s extents, center position, and rotation. The center position and rotation are used as the entity’s current position and rotation in virtual space, respectfully. The OOBB also contains several methods to facilitate collision detection between entities. The method to test OOBB intersection utilizes separating axis theorem (SAT) and is used to determine if two entities are “close enough” before proceeding with further collision tests [116].

The variables within the OOBB are modified by the input component, which processes user input from either the WinProc function, or from the haptics device, and adjusts the entity’s position/rotation accordingly. The triangle-mesh component is a data structure containing various OpenGL buffers for rendering the triangle mesh representation of an Entity. The reamer and cylinder (i.e. the surgical tools) are drawn with triangle meshes since their models remain static over the course of the simulation. For models whose representation is consistently updated, such as the Scapula, the VG component is used instead.

Interaction, collision, and, sometimes, rendering within the engine revolves around voxels. Voxels are discrete units of space containing metadata describing the contents of said space. The engine utilizes voxels to internally represent its objects because voxels enable a more precise and simplified means of simulating materials and, more importantly, material removal. Changes in a 3D voxel grid simply involve changing the metadata of affected voxels. For example, one could set the density values of collided voxels to 0 to signify removal. Compare this to triangle meshes, which involve complex mathematics to reposition triangle vertices in response to a change [30]. Voxels are also naturally suited for speedup via parallelism, given that they can be treated as discrete units. In the end, voxels end up being easier to work with, and much more performant.

The VG component houses OpenCL buffers that hold all the voxels used to represent an entity. All voxels are stored in GPU VRAM to avoid unnecessary CPU to GPU memory transfers, as all voxel manipulation tasks are run on OpenCL kernels. Additionally, voxels within the buffers are arranged in according to a 3D matrix. This is so that their 3D position in space, relative to the grid, can be queried using their array index and the voxel grid’s dimensions (Alg. 1.1). The VG component is created differently depending on the file data used to create the Entity. The voxel grid is the result of either a voxelization process of an STL file (surgical tools), or of a data transfer from a CT scan file (scapula). In the case of the surgical tools, the component contains a single OpenCL buffer that contains the object's voxels in a 3D matrix arrangement. However, the component for the scapula bone contains several OpenCL buffers that together make up the entity. These buffers have their own organization and query system and are known as "Chunks".

2.2 Chunks and the chunk system

A "Chunk" is a 32 × 32 × 32 grid of voxels stored in an OpenCL buffer, and the system designed to access and manipulate chunks is one of the main features of this engine. Chunks are essentially a means of partitioning the voxel representation of an object into uniform sectors (Fig 2.1), and were introduced to address certain limitations when representing models using a single buffer or 3D texture, a common practice found in similar projects [74, 114].

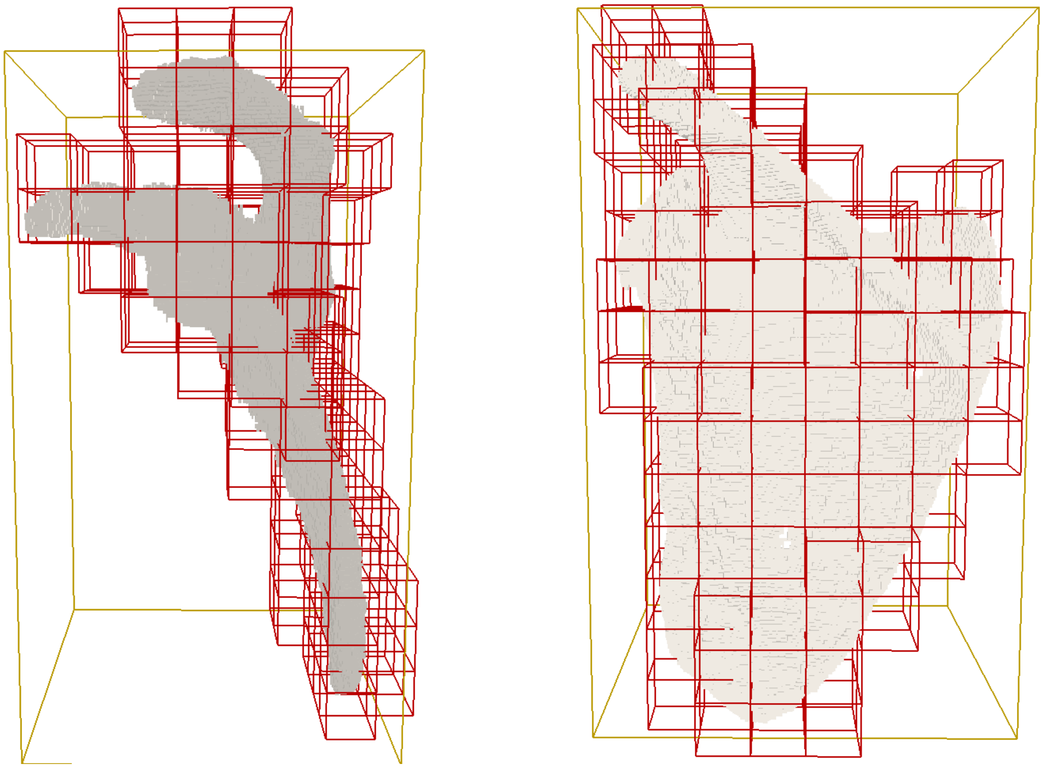


Figure 2.1 Chunk partitioning system of voxelized scapula bone.

When performing material removal, the underlining voxel and rendering data behind the affected model must be updated to show that a change has occurred. Here, a few scenarios can occur. If a single buffer is used to store the model's voxels, and the buffer is sparse, meaning that no memory is wasted on voxels representing empty space, then the voxels of the buffer must be shifted to new positions after each removal to reflect the change. The cost in time is dependent on the initial buffer size, and the number of voxels removed per update. Additionally, extra memory would have to be allocated for the voxel's 3D world space position as it couldn't be calculated from a buffer index (i.e.: voxels aren't stored in a 3D matrix arrangement). If a single, non-sparse, buffer is used to hold a model's voxels, then voxels may be arranged in a 3D matrix, 3D positions may be extrapolated from buffer indices, and no shifts would be required after updates since empty voxels are allowed in the buffer. Chunk voxel buffers are not sparse and hold this property. However, single buffers that encompass the bounds of an object may waste a lot of memory depending on the dimensions and contents of the object being represented. The scapula CT scan used for the simulation has chunk voxel grid dimensions of 256 * 256 * 224 (Table 3.2), with most of the voxels representing empty space. Chunks are a means of partitioning the model into multiple buffers, making it possible to allocate chunks only for active voxels, and omitting those that contain only empty space voxels without affecting the rest of the grid. Finally, if a single buffer is used to generate a triangle mesh representation of the object, then the entire triangle mesh must be regenerated after an update. Chunks do not have this problem as each chunk can be rendered to the screen separately from one another. As such, individual chunks come together like puzzle pieces to display the whole object they represent. On a final note, a single 3D Texture could be used instead of a single buffer to store the voxel representation of a model. Unfortunately, until OpenCL 2.0, 3D textures, unlike buffers, could not be read and written to within the same kernel [98]. There would have to be an extra step during collision detection where voxels are read in one kernel execution, and then collision changes are written in another kernel execution, which is slower. In conclusion, there are many advantages to partitioning voxels into uniform sectors, hence the introduction of chunks, and a system to correctly access them when needed.

2.2.1 Chunk voxel grid dimensions reasoning

Dimensions of 32 × 32 × 32 were determined to be a sweet-spot for chunk size when analysing performance during collision testing and material removal. Compared to the other dimensions tested, such as 16^3 and 64^3, 32^3 chunks minimize both the number of chunk collision tests required to complete a collision event, and the maximum number of voxels parsed per event. A more thorough analysis on the matter is explained in Chapter 3, which also details the chunk system more in depth.

2.3 Model voxelization and STL file to voxel grid conversion

After creating a Win32 window, the engine initializes the scapula, reamer and peg-hole cylinder entities. Surgical tool entities derive their data from .STL files, which are processed through a voxelization algorithm to generate a voxel representation. The voxel representation is used for the material removal stage of the simulation. The voxelization process is as follows. A .STL file containing mesh data for the tool is read and processed using ASSIMP 4, an open-source asset import library for C++ [117]. From the vertices extracted, the minimum and maximum coordinates are computed. These 3D vectors are then used to calculate various components of the model’s voxel-grid representation, given a user inputted voxel size, which defaults to 0.5 mm^3. The OOBB created for the entity is centered at the origin, and its extents are set to 1 `VoxelSize` greater than calculated model bounds to avoid the voxelization algorithm throwing an array out-of-bounds exception (Alg. 2.1).

```
CaclModelVoxelGridComponents
MinVoxelCoords <= round(MinVertexCoords – HalfVoxelSize, VoxelSize) –VoxelSize);
MaxVoxelCoords <= round (MaxVertexCoords + HalfVoxelSize, VoxelSize) + VoxelSize);
EntityOrientedBox <= OOBB(MinVoxelCoords, MaxVoxelCoords);
VoxelGridDims <= ceil((MaxVoxelCoords - MinVoxelCoords) / VoxelSize);
TotalVoxelCount <= VoxelGridDims.x * VoxelGridDims.y * VoxelGridDims.z;
```

Algorithm 2.1: Calculation of STL model voxel grid properties

Vertices, normals, and indices are then transferred to OpenGL vertex and element buffers that are bound to a vertex array object (VAO). The VAO is used later during scene rendering to draw the triangle mesh representation of the model. The vertex and index buffers are also used to create shared OpenCL buffers. These shared buffers are used for the voxelization process. Two additional OpenCL buffers are created to hold voxel-grid information (i.e.: `VoxelSize, MinVoxelCoords, EntityOrientedBoxExtents, VoxelGridDims,` and `VoxelCount`), and the entity's computed voxels, respectfully. Note that voxels are int16_t primitives. These integers hold density values, which vary from 0 (empty) to 1 (solid) for surgical tools.

After all buffers are created and initialized, the arguments for an OpenCL voxelization kernel are set, and the voxelization kernel is launched. The voxelization algorithm is taken from the following papers [108, 109, 114, 118, 119]. Once the process is complete, buffers containing the entity's voxels and voxel grid information are transferred to the entity.

Table 2.1: Timings (in ms) of model voxelization for models tested

Tasks	Reamer timings (ms)	Cylinder timings (ms)
Voxelize model	0.753728	0.559584
Adjust X-Axis voxel densities	0.442368	N/A

2.4 Chunk grid construction using computed tomography scans

Interactable entities whose voxels are not static over the course of the simulation have their voxels organized into a chunk grid (stored inside the VG component) for easier voxel manipulation. The scapula uses a chunk grid, and its data is derived from a CT scan file. First, raw data from a CT scan file is converted to a 1D array of structs containing the object-relative position of each voxel and its associated density (Alg. 2.2).

```
Data structure used to hold raw CT scan voxel data
struct CTScanVoxel
{
      vec3 Position;
      float Density;
};
```

Algorithm 2.2: CT scan voxel data structure

```
CalcVoxelGridParams: MinVoxelCoords, MaxVoxelCoords, RectVoxelSize
RectMinVoxelCoords <= floorMultiple(MinVertexCoords, RectVoxelSize)
RectMaxVoxelCoords <= ceilMultiple(MaxVertexCoords, RectVoxelSize)
RectVoxelGridDims <= round((RectMaxVoxelCoords – RectMinVoxelCoords) /
RectVoxelSize);
RectVoxelCount <= RectVoxelGrid.x * RectVoxelGrid.y * RectVoxelGrid.z;
CubicVoxelSize <= vec3(RectVoxelSize.x, RectVoxelSize.x, RectVoxelSize.x)
CubicVoxelGridDims.x <= RectVoxelGridDims.x
ubicVoxelGridDims.y <= RectVoxelGridDims.y
CubicVoxelGridDims.z <= ceil((RectMaxVoxelCoords.z – RectMinVoxelCoords.z) /
CubicVoxelSize.z);
CeilCubicVoxelGridDims <= ceilMultiple(CubicVoxelGridDims, ChunkDims)
CeilVoxelCount <= CeilCubicVoxelGridDims.x * CeilCubicVoxelGridDims.y *
CeilCubicVoxelGridDims.z
ChunkGridDims = CeilCubicVoxelGridDims / ChunkDims
```

Algorithm 2.3: Calculation of parameters for rectangle and cubic CT scan voxel grid

```
RawCTScanDataToRectGridTransfer: global float4* RawCTDataArray
// CT data is stored as float4, where .xyz is voxel position, and .w is
voxel density
I <= get_global_id(0)
P <= convert_int3_rtz(RawCTDataArray[I].xyz – RectMinVoxelCoords) /
VoxelSize)
J <= 3DGridPosToIndex(P, RectVoxelGridDims)
VoxelArrayOut[J].Density = convert_short(RawCTDataArray[I].w)
```

Algorithm 2.4: Raw data OpenCL kernel pseudocode

Second, the list is converted to a 1D int16_t array with a size equal to the calculated volume of the CT scanned object in voxel units. The dimensions of the voxel grid are calculated from the min and max position coordinates divided by scan voxel size (Alg. 2.3). Each element within this 1D array contains the density, while voxels are placed in the array according to their object-relative position minus the min position coordinates. A conversion equation transforms this 3D position into a 1D index (Alg. 2.4).

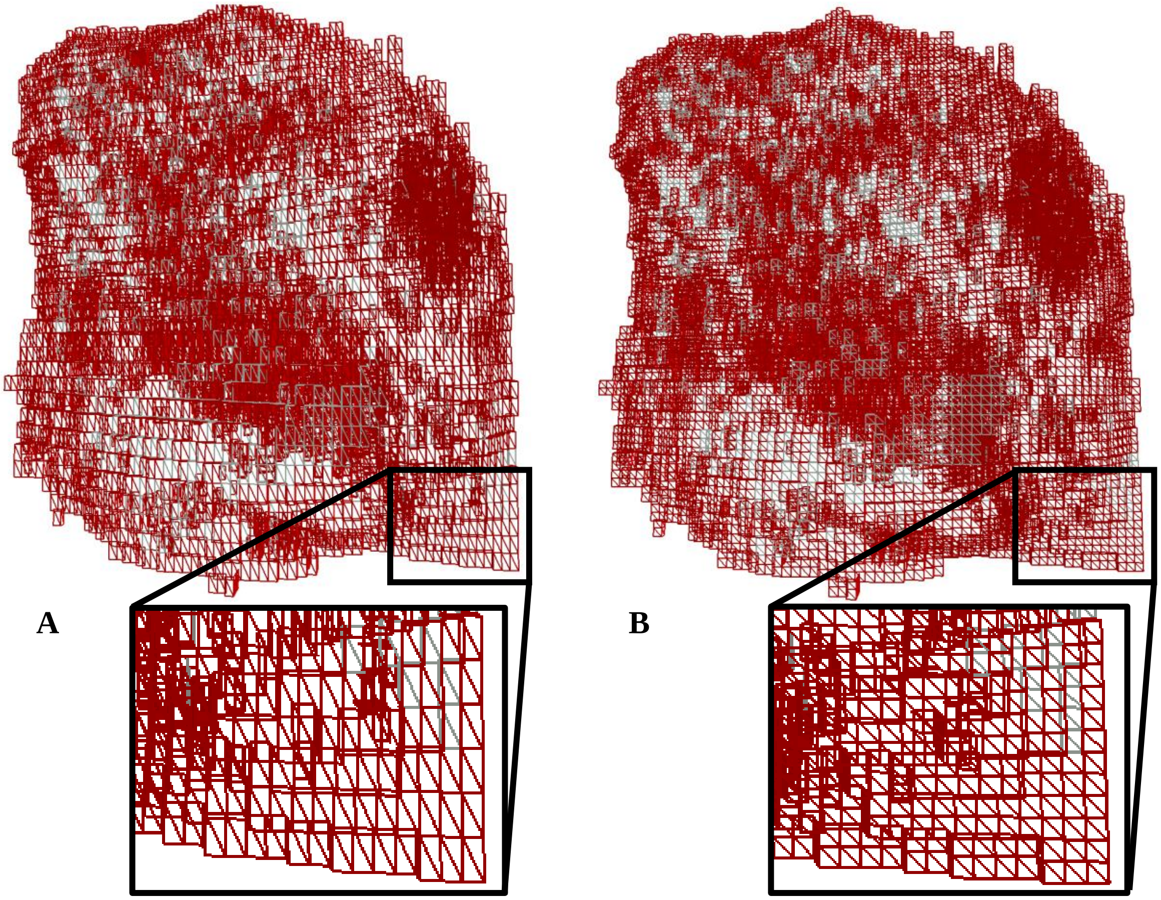


Figure 2.2: Before (A) and after (B) "*RectGridToCubicGridTransfer*" processing

```
RectGridToCubicGridTransfer
// adjusting only Z dimension!
// global_work_size <= CubicVoxelGridDims
// local_work_size <= 1
X <= get_global_id(0)
Y <= get_global_id(1)
Z <= get_global_id(2)
PrevRectGridIndex <= convert_int_rtz(((Z - 1)*
CubicVoxelSize.z)/RectVoxelSize.z)
```

```
CurrRectGridIndex <= convert_int_rtz((Z * CubicVoxelSize.z) /
RectVoxelSize.z)
RectGridDepthRatio = (Z*CubicVoxelSize.z) – (PrevRectGridIndex *
RectVoxelSize.z)
RectGridDepthRatio /= CubicVoxelSize.z
Int I = RectGridDepthRatio >= 0.5 ? CurrRectGridIndex : PrevRectGridIndex
Int J <= 3DGridCoordsToIndex(X, Y, Z, CubicVoxelGridDims)
Int K <= 3DGridCoordsToIndex(X, Y, I, RectVoxelGridDims)
CubicVoxelArray[J] = RectVoxelArray[K]
```

Algorithm 2.5: Rectangle grid to cubic grid voxel transfer OpenCL kernel pseudocode

Third, if the dimensions of the CT scan voxel are not equal (as is the case with the Scapula model used, which are 0.47 mm x 0.47 mm x 1.0 mm), then voxels are copied into a separate 1D array, and filler voxels are added at precise locations such that, in the end, all voxels in the array have equal dimensions and the same density per volume is retained (Fig 2.2, Alg. 2.5). This is done to accommodate a limitation with GL_POINT based voxel rendering (see Chapter 3, 3.3). The initial array is known as the RectVoxelGrid, while the final array is known as the CubicVoxelGrid (the names referring to the shape of the voxels they store) (Alg. 2.3).

```
CubicVoxelGridToChunkTransfer
// global_work_size <= ChunkDimsInVoxels
// local_work_size <= 1
X <= get_global_id(0)
Y <= get_global_id(1)
Z <= get_global_id(2)
VoxelPosRelChunkGrid <= (int3)(X, Y, Z) + ChunkMinPosInUnitVoxels
Int I = 3DGridCoordsToIndex(X, Y, Z, ChunkDims)
Int j = 3DGridCoordsToIndex(VoxelPosRelChunkGrid, CubicVoxelGridDims)
ChunkVoxelArray[I] = CubicVoxelArray[J]
If (CubicVoxelArray[J].Density >= 120) atomic_inc(ActiveVoxelsCounter)
```

Algorithm 2.6: Cubic voxel grid to chunk voxel buffer OpenCL kernel pseudocode

Table 2.2 Timings (in ms) of CT scan chunk grid construction phases for models tested.

Tasks (measured in ms)	*Cellular foam*	*Trabecular bone core 1*	*Trabecular bone core 2*	*Scapula*	*Glenoid*
Load raw data from CT text file	6930	936	13379	295	31
Raw data to rect. voxel grid	26	3.5	51	1.2	0.13
Rect. voxel grid to cubic voxel grid	175	23	106	78	0.67
Cubic voxel grid to chunk grid	141	17	76	62	1.2

Finally, the CubicVoxelGrid is subdivided into chunks, and placed into a chunk grid, which is a 1D array of chunk pointers arranged in a 3D matrix fashion. Note that chunks that have no active voxels (i.e.: voxels with densities less than or equal to 120) inside them have their buffers deleted to save memory, and a flag set to indicate that they are empty (Alg. 2.6).

2.5 The haptics loop

With the simulation objects initialized, the engine then allocates 2 CPU threads, where 1 thread is responsible for the rending loop, and the other maintains the haptics/update loop. The rendering thread is v-sync locked and runs at 60 Hz. The haptics thread is created with highest priority, runs at 1 kHz, and was written according to example code provided by the OpenHaptics SDK [95]. The haptics thread is a required component of the engine, given that this thread is responsible for deriving input from the connected PhantomOmni haptics device. As such, physics updates and collision testing are preformed within the haptics loop. Entities that are controlled with the haptics device are updated at 1 kHz, along with OOBB collision detection. The algorithm for material removal runs at 1/3 of the speed, or 333 Hz, due to the amount of time 1 execution of a collision test takes (~0.05 – 0.06 ms), and the number of tests taken per collision event between the reamer and the scapula (1 – 20 events) (Table 3.3). Its limited to allow for a stable experience, as the number of chunks processed per collision event is variable and could cause lag spikes if left without restriction. These values were taken from a Windows 10 workstation running an intel i5 8600, Nvidia GTX 1070, and 8 GB of RAM. At 1 kHz entity movement updates, 333 Hz material removal updates, and 60 Hz rendering, the surgeon will not see a discrepancy between their input with the haptics device and what’s shown on screen. The final initialization step, after the objects, is to initialize the camera, which has its own uniform buffer for camera variables (i.e.: view and projection matrices, aspect ratio, FOV, near and far plane), the geometry buffer, and the shaders, one for voxels, one for triangle meshes, and one for the final composite pass. The GBuffer is used for deferred rendering. Lighting in the engine uses a basic directional lighting algorithm [110]. With all objects set up, the haptics thread is launched, and the simulation begins.

At the start of the haptics loop, raw haptics input data (i.e.: position vectors and transformation matrix) is collected. This data is then mapped to the OOBB of current surgical tool being used. The initial tool is the peg-hole cylinder and is used to create a peg-hole in the glenoid. Upon pressing a hotkey, the tool switches to the reamer to finish the operation. The surgical tool being used has a 1:1 correspondence with the haptics device, meaning that the current center position and rotation of the entity's OOBB is equal to the position and transformation of the haptics device.

```
CheckForCollisions: BoneEntity, ToolEntity, deltaTime, hapticsReadData
ToolEntity.Update(deltaTime, hapticsReadData)
If (BoneEntity.CollidesWith(ToolEntity))
      PerformCollisionDetection()
```

Algorithm 2.7: Entity collision check pseudocode

Once the surgical tool entity is updated, the engine checks whether the OOBB of the tool (ToolEntity) collides with the scapula (BoneEntity) (Alg. 2.7).

```
ComputeIntersectionBox: BoneEntity, ToolEntity
vec3 Min(FLT_MAX)
vec3 Max(-FLT_MAX)
For Corner in ToolEntity.OBBox.Corners {
      if (BoneEntity.OBBox.PointInBox(Corner)) {
            Min <= min(Min, Corner)
            Max <= max(Max, Corner)
      } else {
            Vec3 Point        <= BoneEntity.Box.GetPointClosestTo(Corner)
            Vec3 BoxCenter    <= BoneEntity.Box.Center;
            Mat3 Rotation     <= BoneEntity.Box.Rotation;
            Vec3 Extents      <= BoneEntity.Box.Extents;
            Vec3 Delta        <= Corner - BoxCenter
            Vec3 ClosestPoint <= BoxCenter
            For (I = 0; I < 3; ++I) {
                  ClosestPoint += clamp(dot(delta, Rotation[I]), -Extents[I],
                  Extents[I]) * Rotation[I]
```

```
                }
                Min = min(Min, ClosestPoint)
                Max = max(Max, ClosestPoint)
        }
}
IntersectionBox <= ObjectBoundingBox(Min, Max)
```

Algorithm 2.8: Entity intersection volume calculation pseudocode

```
ObtainIntersectingChunkGridRanges
mat3 InvBoneRotation <= inverse(BoneEntity.OBBox.Rotation)
ivec3 ChunkMinPosInUnitChunks(INT_MAX)
ivec3 ChunkMaxPosInUnitChunks(-INT_MAX)
For (Corner in IntersectionBox.Corners) {
        TransformedCornerPoint <= (InvBoneRot * (Corner – BoneEntity.OBBox.Center)) + BoneEntity.OBBox.Center
        PointInUnitVoxels <= (TransformedCornerPoint + BoneEntity.Box.Extents) /
BoneEntity.VoxelSize;
        MinPointInUnitChunks<= floorMultiple(PointInUnitVoxels,CHUNK_SIZE)/CHUNK_SIZE;
        MaxPointInUnitChunks<= ceilMultiple(PointInUnitVoxels,CHUNK_SIZE)/CHUNK_SIZE;
        ChunkMinPosInUnitChunks <= min(ChunkMinPosInUnitChunks, MinPointInUnitChunks);
        ChunkMaxPosInUnitChunks <= max(ChunkMaxPosInUnitChunks, MaxPointInUnitChunks);
}
```

Algorithm 2.9: Min/max intersecting chunk coordinates calculation based on intersection volume

```
SetupChunkCollisionKernel: BoneEntity, ToolEntity
Mat4 BoneToObjTransform = BoneEntity.GetTransformTo(ToolEntity)
For XYZ = ChunkMinPosInUnitChunks -> ChunkMaxPosInUnitChunks  {
        Index = XYZGridCoordsToIndex(XYZ, BoneEntity.VoxelGrid.GridDims)
        Chunk = BoneEntity.ChunkGrid.Chunks[I]
        If (!Chunk.Empty) {
                SetupChunkCDKernelArgs()
                EnqueueChunkCDKernel()
        }
}
```

Algorithm 2.10: Intersection chunk collision detection OpenCL kernel setup pseudocode

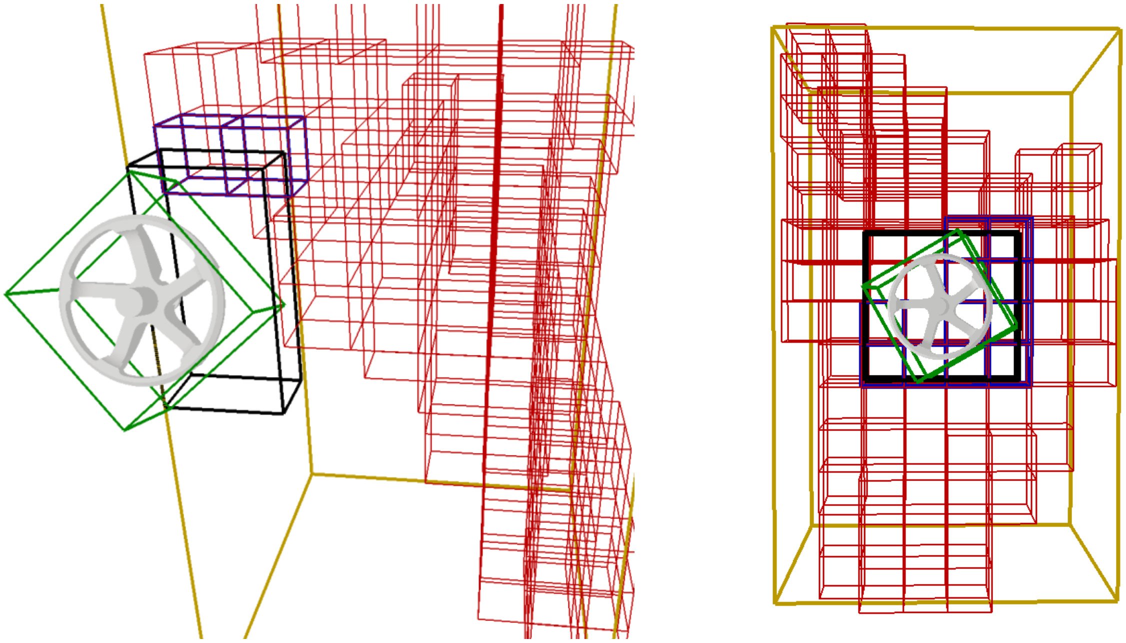


Figure 2.3: OOBB intersection test and intersecting chunks identification.

If the OOBBs intersect, then the material removal algorithms are executed (Alg. 2.8-2.10). Before that, however, the scapula chunks that are involved in the collision between the two entities must be identified and collected. To do so, an intersection bounding box is computed (Alg. 2.8). The intersection bounding box's volume encapsulates all the chunks that are involved in the collision event (Fig 2.3).

From the corners of the intersection volume, chunks are collected (Alg. 2.9), and then their OpenCL voxel array buffers are run through an OpenCL kernel (Alg. 2.10). This kernel identifies colliding voxels and removes them by setting the voxel's density to 0. The chunk material removal kernel is based on the algorithm found here [114]. The algorithm was adapted for chunks. Where the original method would iterate though the sparse voxel buffer of the tool entity, the chunk variant iterates though all the voxels of the chunk's voxel buffer. Essentially, it is the reverse operation. The voxel's in the chunk buffer have their values updated if the algorithm determines an overlap between the transformed position of the tool voxel and the chunk voxel. This change is then immediately reflected in rendering as the render process uses the Chunk's OpenCL/OpenGL shared voxel buffer to draw the chunk (see Chapter 3, 3.3).

2.6 The render loop

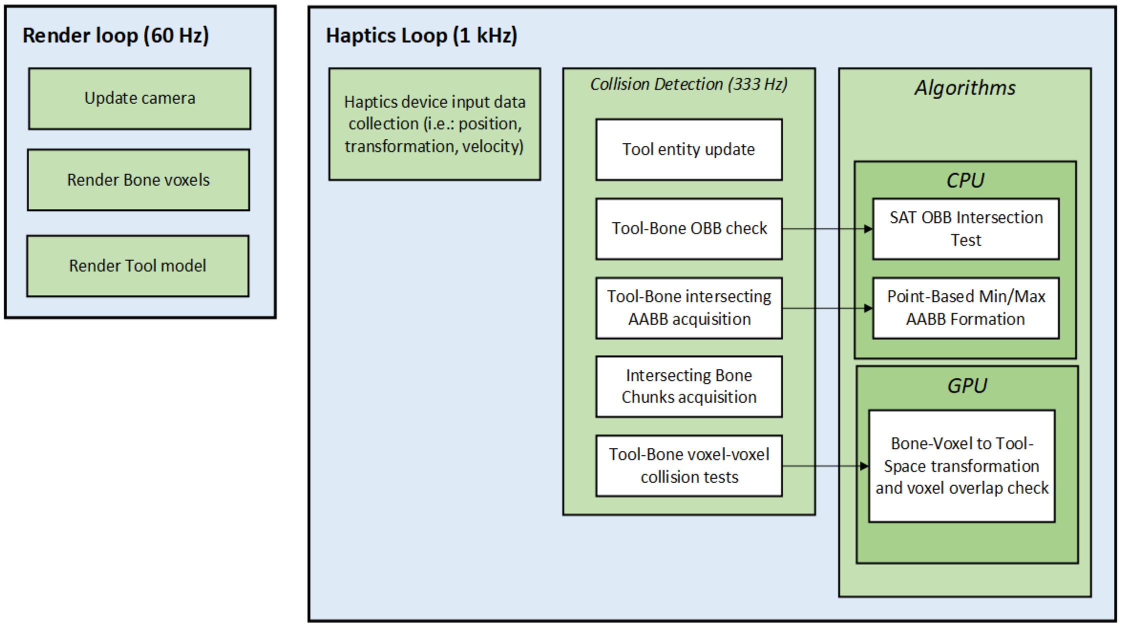

Render loop (60 Hz)
Update camera
Render Bone voxels
Render Tool model
Haptics Loop (1 kHz)
Haptics device input data collection (i.e.: position, transformation, velocity)
Collision Detection (333 Hz)
Tool entity update
Tool-Bone OBB check
Tool-Bone intersecting AABB acquisition
Intersecting Bone Chunks acquisition
Tool-Bone voxel-voxel collision tests
Algorithms
CPU
SAT OBB Intersection Test
Point-Based Min/Max AABB Formation
GPU
Bone-Voxel to Tool-Space transformation and voxel overlap check

Figure 2.4: Flowchart of the engine's render and haptics loops.

The render loop for the engine is straightforward, considering that only the surgical tool models and chunks of the scapula are rendered (Fig 2.4). At the start of the loop, the camera's view and projection matrices and uniform buffer are updated based on user input. Next the geometry framebuffer (GBuffer) is bound. Currently, the GBuffer contains a R32 depth 2D texture attachment (for vertex position extraction), and 2 RGBA32 color 2D texture attachments (1 for vertex normals, and the other for vertex colors). With the GBuffer being bound, the entities in the simulation are then rendered.

The scapula is rendered chunk-by-chunk using an adapted GL_POINT based voxel rendering technique [112]. The technique enables voxels in a buffer to be directly rendered without the need to create a triangle mesh, greatly increasing framerate performance (see

Chapter 3.3). When rendering a chunk, the chunk's OpenCL/OpenGL shared voxel array buffer is bound to the context via an associated single attribute VAO (i.e.: `glVertexAttribIFormat(0, 1, GL_SHORT, 0)).` A uniform buffer is then bound and updated with the current chunk's voxel grid position in voxel units. This allows proper calculation of the chunk's voxel 3D world position. After calling `glDrawArrays(GL_POINTS, 0, 32 * 32 * 32)`, the voxel vertex and fragment shaders, taken and adapted from [112], are invoked. The changes from the original shaders are minor. Inside the vertex shader, the position of the voxel that is used for the projection algorithm is computed as follows:

```
VoxPos = IndexTo3DGridCoords(gl_VertexID, ChunkDims) + ChunkPositionInUnitVoxels
```

Furthermore, the model matrix used for the projection function is the scapula's OOBB model matrix. Finally, in the fragment shader, the computed normal and voxel color are copied to the GBuffer's normal and color 2D texture attachments, respectfully. The color of bone voxels is #e3dac9. Bone voxels with densities less than 120 are discarded by setting gl_Position to vec4(-1).

To render the surgical tools, their triangle meshes are run through a simple shader where each vertex is transformed by the model view projection matrix made from the camera matrices and the tool's model matrix. In the fragment shader, the vertex normals and colors are transferred to the correct GBuffer 2D texture attachments. With all entities rendered to the GBuffer, the GBuffer is unbound, its 2D texture attachments are bound, the screen's framebuffer is cleared, and a full-screen triangle is rendered. Position, normal, and color data are extracted from the bound texture attachments and basic directional lighting is applied, giving the final image on the screen [110, 111].

Chapter 3

The chunk system

3.1 Memory optimizations

3.1.1 Against single buffer/texture data structures

The chunk system addresses two concerns simulation engines face when attempting to achieve optimal runtime performance. The first is minimal memory overhead when loading high resolution CT scans. The second is mitigation of high-volume computations performed during the collision testing and material removal stages. The chunk system confronts these challenges through a grid partitioning scheme, where units, a.k.a. chunks, of said grid act independently of one another. Through this discrete property, chunks allow “empty” (i.e.: inactive) sections of a model’s representative voxel grid to be omitted for space efficiency. Additionally, chunks enable collision processing tasks to query, and work on, specific segments of the model as needed, which bypasses numerous unnecessary voxel collision participation checks.

Regarding memory management, it is important to note several observations that are apparent when converting a CT scanned object to a representative 1D voxel buffer organized as a 3D matrix (i.e.: where buffer indices are convertible to object-relative 3D grid positions). To start, CT scans are text files that contain a list of data entries, each of which hold two pieces of information, a 3D float vector position and an integer density. These entries are, for all intents and purposes, object voxels that were generated by a real-world CT scanner, and are of a size pre-set by said machine. Generating virtual voxels from CT scans is thus a matter of reorganizing voxels into a 3D matrix (see Chapter 2, 2.4). With voxel query improvements, and 12 bytes of memory being saved per voxel, 3D matrices hold properties that make them process efficient, especially when compared to the list format of CT scan files. However, the problem with this approach is that extra memory is needed to create a 3D matrix that completely encompasses a CT scan object’s bounds. Essentially, when representing areas of empty space, the associated buffer locations must be filled with voxels containing “uninteresting” data.

A voxel that is considered to contain data of interest (i.e.: one that is “active”) is one that has a non-zero density value indicating the presence of material. Inactive voxels, therefore, don’t hold any useful information other than to denote that there is empty space at their coordinates. However, these voxels still take up memory by being present in the 3D matrix.

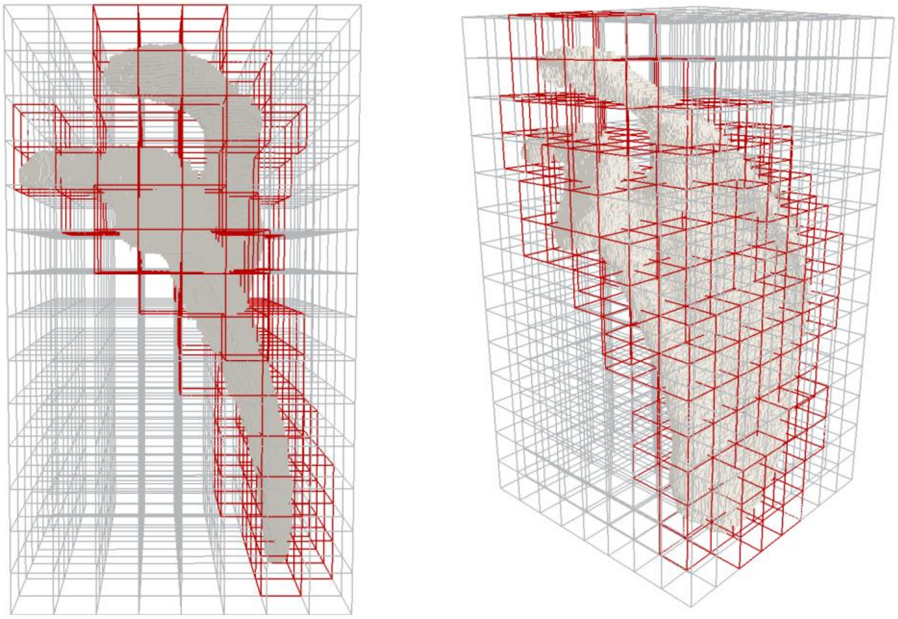


Figure 3.1: Visual representation of active (red) vs. inactive (grey) chunks in generated Scapula chunk grid.

When creating a 3D matrix from a CT scanned object, there will often be a lot of memory that describes empty space due to the object’s mass being unevenly distributed, or its geometry having an irregular shape. For a single 3D matrix, there is no optimization possible to avoid consuming memory for this empty space, as you cannot create a 3D rectangular matrix that tightly wraps around the surface of the object. To accurately represent an object, the 3D matrix must stretch across the object’s XYZ bounds. However, if you were to have multiple 3D matrices, arranged in a grid-like fashion, to represent the object, then it’s likely that certain matrices would contain nothing but empty space. Those matrices could then be freed from memory without affecting the others, thus saving memory. To indicate that the matrix in the grid doesn’t have a backing voxel buffer, a simple flag could be added at that grid position. This is logic behind how the chunk system operates compared to a single encompassing voxel buffer or 3D texture approach (Fig. 3.1).

Table 3.1: General voxel grid characteristics of the CT scanned objects used for testing

Model	*Voxel size*	*Voxel grid dimensions*	*Voxel Count*	*Active Memory (2B/Voxel)*	*Text file size on disk*
Cellular foam 1	32µm isotropic	314 × 320 × 604	60,689,920	**121.4 MB**	764.1 MB
Trabecular 2	32µm Isotropic	314 × 316 × 360	35,720,640	**71.4 MB**	1.4 GB
Trabecular 1	32µm isotropic	156 × 156 × 313	7,617,168	**15.2 MB**	101.4 MB
Scapula	(0.47, 0.47, 1) mm	256 × 234 × 201	12,040,704	**24.1 MB**	34.3 MB
Glenoid	(0.47, 0.47, 1) mm	49 × 81 × 41	162,729	**0.3 MB**	3.7 MB

Table 3.2: Chunk grid characteristics of the CT scanned objects used for testing

Model	*Voxel grid dims after chunk gen*	*Chunk grid dimensions*	*Chunk count*	*Active chunk count*	*Raw memory (2B/Voxel)*	*Active memory (2B/Voxel)*
Cellular foam	320 × 320 × 608	10 × 10 × 19	1900	1669	124.5 MB	**109.4 MB**
Trabecular 2	320 × 320 × 384	10 × 10 × 12	1200	1060	78.6 MB	**69.5 MB**
Trabecular 1	160 × 160 × 320	5 × 5 × 10	250	250	16.4 MB	**16.4 MB**
Scapula	256 × 256 × 448	8 × 8 × 14	896	202	58.7 MB	**13.2 MB**
Glenoid	64 × 96 × 96	2 × 3 × 3	18	18	1.2 MB	**1.2 MB**

Tables 3.1-3.2 show various characteristics of the voxel grids generated for the 5 CT scanned objects used for testing and analysis (Fig. 1.4). Of importance are the “Voxel grid

dimensions” and “active memory” categories. The voxel grid dimensions of Table 3.1 describe the computed minimum lengths of a voxel grid that could be generated to completely encompass the bounds of the associated CT scanned object. These would be the dimensions of the 3D matrix of the single buffer voxel representation techniques, and from these values, it possible to calculate the amount of memory occupied by the buffer, assuming each voxel consumes 2 bytes to store material density. Table 3.2 shows the same 5 CT scanned objects, expect with chunks as their backing voxel representation instead. Note that here, voxel dimensions are ceiled to the nearest chunk to avoid partial chunks within the grid, thus being larger compared to the values present in Table 3.1. Although there are indeed filler layers of voxels added to accommodate the chunk system, it is evident that memory is still being saved, as in most instances, the number of active chunks (i.e.: chunks with at least 1 active voxel present) is, sometimes, dramatically less than the total chunk count.

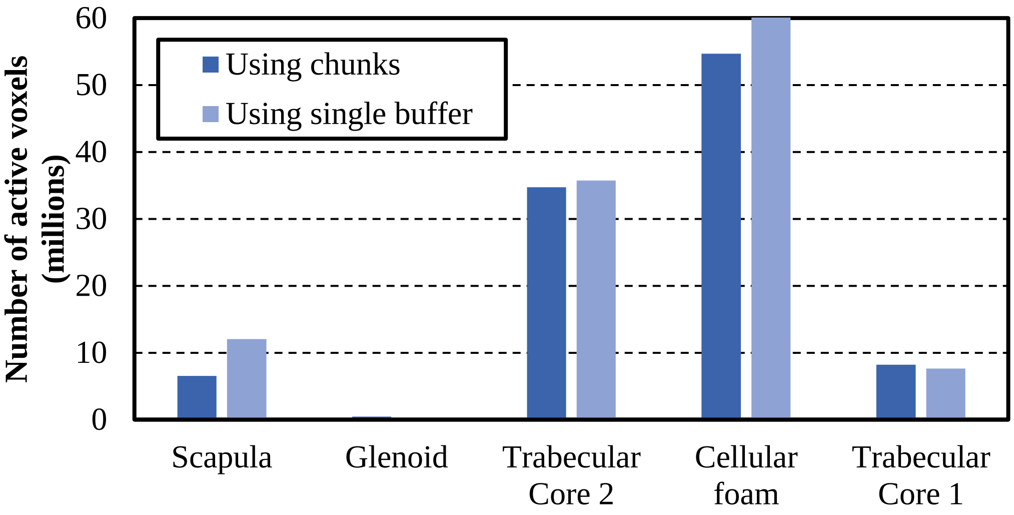


Figure 3.2: Active voxels stored in memory across test models and data structures.

When analysing the memory consumption of the chunk system against a traditional single buffer arrangement to store an CT scanned object’s voxel representation, the chunk system wins in 3 of the 5 cases, excelling when the geometry of the CT scan is non-uniform or has uneven mass distribution. Under this condition, memory savings can be up to 50% (ex: Scapula) over using a traditional single buffer or 3D texture (Tables 3.1 - 3.2, Fig 3.2). Interestingly, a reduction in memory is not always guaranteed when using the chunk system, as is the case with the glenoid and trabecular core 1 (Table 3.1 - 3.2, Fig. 3.2). The reason for this has to do with the geometries of these objects. With these objects being cubic and uniform in mass (Fig. 1.4), a 3D matrix tightly encompassing object bounds could be created. As a result, when adapting these objects to the chunk system, the lack of empty chunks (Table 3.2), and the extra layers generated by the chunk grid creation algorithm would cause the chunk system to occupy more memory.

3.2 Material removal

3.2.1 Collision testing with chunks

Real-time collision detection and material removal were the forefront features considered when designing and implementing the chunk system. The idea was that, if the bone model to be worked upon was partitioned into uniform sectors, it would then be possible to identify and segregate only sections that are involved in a collision event. Work could then be done on those tool-intersecting volumes, which may, theoretically, lead to fewer voxels being processed, and, in turn, reduce computation time. Considering that the time allocated for collision updates is extremely short due to the 1 kHz requirement by the haptics device (where 1 update refresh must be equal to or less than 1 ms), any boost in performance compared to the current collision algorithms would be considered a success. Alternatively, achieving parity would also be welcome, as it would demonstrate that the chunk system is a viable candidate for effective collision testing that is also deserving of further investigation.

There are two stages of any collision detection mechanism (see Chapter 2, 2.5). The first stage is the identification of the volume where the virtual tool and bone models intersect. The second stage is the intersection testing of individual bone and tool voxels, and the updating of colliding bone voxels to signify a change (i.e.: setting density to 0 to denote voxel is now empty). The bottleneck of this procedure is the processing of individual voxel intersections, and thus the algorithm to be used for this testing must be performant enough such that collision events are able to be completed in real-time. While the chunk system allows for narrowing down the scope of the intersection volume, the algorithm used to compute voxel intersections is adapted from a method developed by Reza et. al [114]. In the chunk variant, active voxels from the intersecting bone chunks are iterated through, and their grid positions are transformed, first relative to the origin by subtracting the bone model's OOBB extents, and then using a bone-to-tool transformation matrix. These transformed coordinates are then compared to the tool's OOBB extents. If they are inside the extents, they are then converted into an index relative to the tool's voxel grid. If the tool voxel at that index is active (i.e.: has density of 1), the bone voxel's density is set to 0. This effectively removes voxels from the scene, which visually shows material removal.

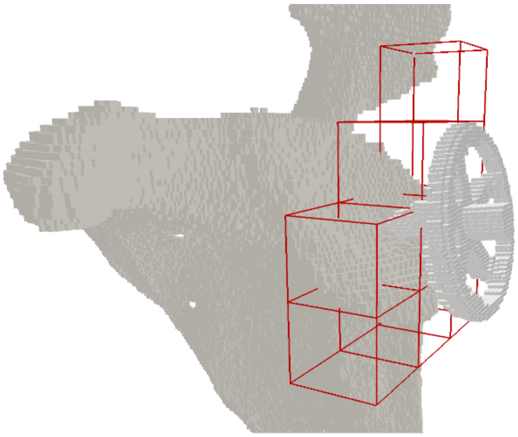


Figure 3.3: Visualization of the chunk system collision testing mechanism. Chunk voxels (voxels in red WF) are iterated and compared against tool voxels

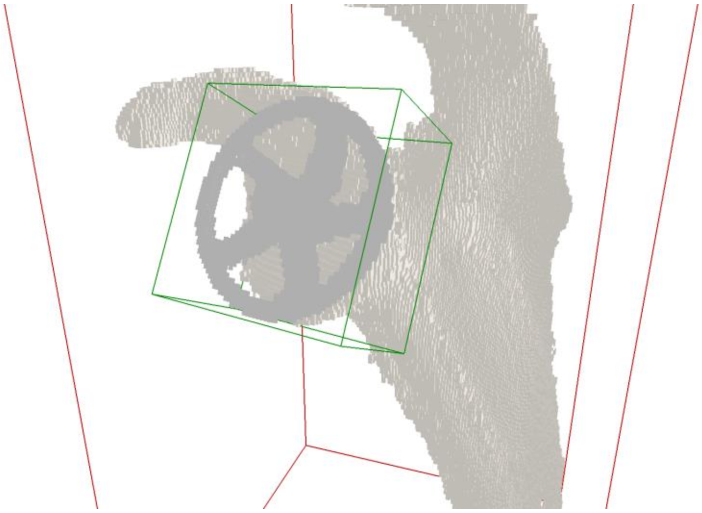


Figure 3.4: Visualization of tool-to-bone voxel collision testing. Tool voxels (green WF) are iterated through, transformed, and collision checked against bone voxels (red WF)

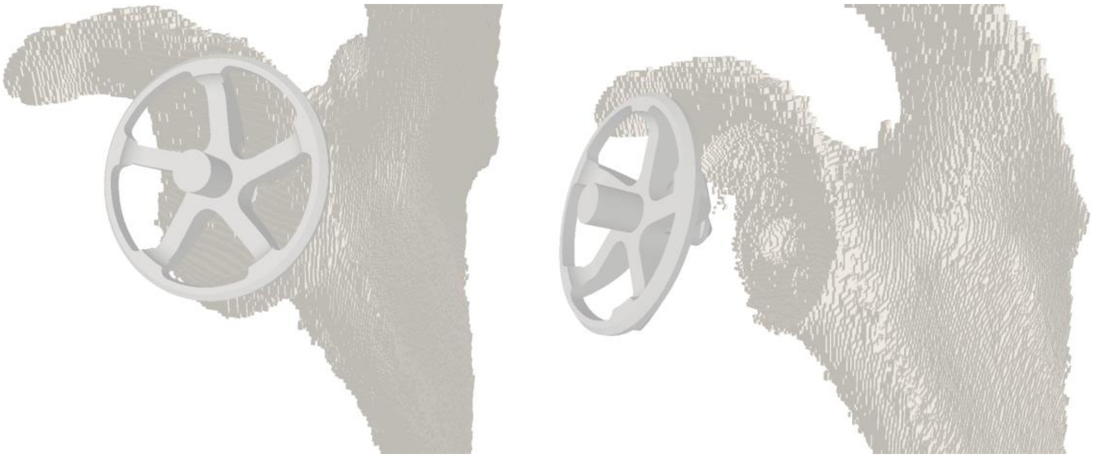


Figure 3.5. Demonstration of the real time collision and material system.

The differences between the chunk variant and the algorithm developed by Reza et. al are minimal, as they follow the same steps for voxel intersection testing. The exception is that, rather than iterate and transform bone voxels (bone-to-tool), Reza et. al's method does the opposite, and runs through the tool voxels instead (tool-to-bone) [114] (Fig 3.4). Regardless, the developments achieved by the engine demonstrate a proof-of-concept that a partitioning system can achieve real-time collision detection and material removal (Fig 3.5).

3.2.2 Baseline chunk collision performance with various tools and sizes

Before comparing the performance of the chunk system against known voxel-based collision detection algorithms, it is important to set the standard for which the chunk system will be judged upon. Essentially, for any one collision event, there will be a set number of chunks that participate in collision testing. Since each of these chunks contain 32^3 voxels, the time required to process a chunk remains relatively consistent between chunks, and across collision events. Therefore, statistics on individual chunk performance, and how computation time changes as more chunks are processed, should be investigated. Of importance is the set size of individual chunks, as the number of voxels worked on per chunk determines the computation time. Setting chunks to 32^3 was a deliberate choice based on several collision based-tests, where chunk sizes were varied during a static collision event (Fig 3.6).

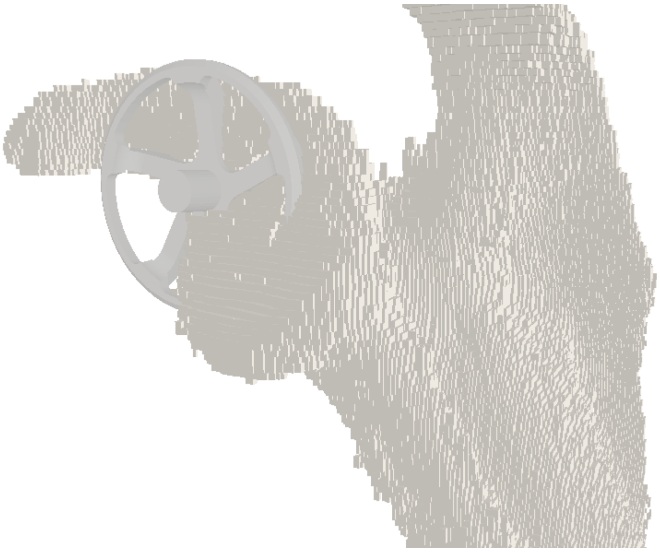


Figure 3.6: Static collision event using Reamer tool for testing effect on performance of chunk sizes during collision testing.

Of the dimensions tested (16^3, 32^3, and 64^3), 32^3 was shown to be a sweat-spot in general cases, followed closely by 16^3 under certain circumstances. On the other hand, sizes of 64^3, and likely larger, have noticeable drawbacks (Table 3.3).

Table 3.3: Static collision event timings for various chunk dimensions using the reamer

Chunk Dimensions	16^3	32^3	64^3
Chunks Collided	84	18	5
Processing time per chunk (ms)	0.012	0.055	0.405
Total processing time (ms)	1.024	0.970	2.023

For chunks with dimensions of 64^3, it is easy to understand why those would take longer when taking into consideration that the intersection surface between the tool and the bone does not change between tests (Fig. 3.6). Essentially, 64^3 chunks are less subdivided than 32^3 and 16^3, meaning that, although less chunks are hit during a collision event, the larger encompassing volume of these individual chunks causes significantly more voxels to be parsed (since chunks are parsed entirely or not at all based on the algorithm). Thus, the amount of computations increases, resulting in more time needed to parse a chunk. Interestingly, the performance between 16^3 chunks and 32^3 chunks is very similar despite more chunks being processed per collision event in the case of 16^3 chunks (Table 3.3). Even more interesting, however, is that the choice of tool during a collision event also plays a part in computation time.

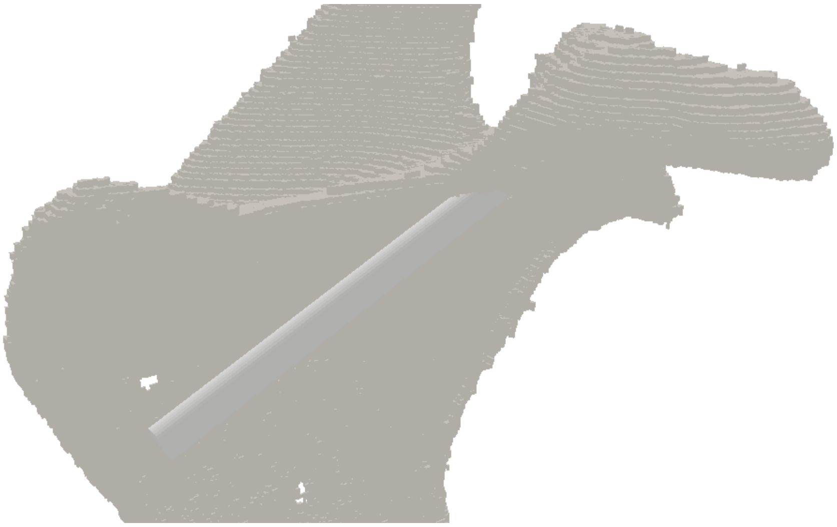


Figure 3.7: Static collision event using peg-hole cylinder tool for testing effect on performance of chunk sizes during collision testing.

When this test was repeated with the peg-hole cylinder tool (Fig. 3.7), while the results confirmed that 64^3 is the least performant of the chunk sizes, it showed that the 16^3 sized chunks performed better overall (Table 3.4).

Table 3.4: Static collision event timings for various chunk dimensions using peg-hole cylinder tool

Chunk Dimensions	16^3	32^3	64^3
Chunks Collided	80	18	8
Processing time per chunk (ms)	0.012	0.065	0.405
Total processing time (ms)	0.964	1.172	3.234

The discrepancy between the two tools is difficult to explain. Logically, since the algorithm iterates through the bone voxels to make its comparisons, the tool's voxel grid should not influence the intersection testing process. Indeed, that is one of the benefits of using the chunk system, since it is possible to vary properties of the tool's voxel grid without affecting the performance. This is further confirmed when varying the voxel-size (and by extension the voxel resolution) of the Reamer tool during a static collision event (Fig 3.6, Table 3.5). With this benefit, it is possible to increase the resolution of tool's voxel grid and increase the precision of the cutting tool without any performance loss.

Table 3.5: Static collision event timings at chunk dimensions of 32^3 using the Reamer tool while varying tool voxel-sizes

Tool Voxel sizes	1.0^3	0.75^3	0.5^3
Chunks Collided	18	18	18
Processing time per chunk (ms)	0.982	0.975	0.967
Total processing time (ms)	0.0545	0.0542	0.0538
Number of voxels	33212	75000	232286

However, when switching tools, from reamer to peg-hole cylinder, it appears that the average processing time for 32^3 chunks goes from 0.055 to 0.065 (Table 3.3 - 3.4). This shouldn’t occur, but it does regardless. One possible theory is that branch prediction in the collision kernel and the tool geometry are responsible. In the chunk collision kernel, there is a check to determine whether a tool voxel is active at a certain position. Perhaps, with branch prediction, there is an optimization that assumes that this check is strongly taken, meaning that it will succeed more than it fails. As such, the code inside the check is executed before the check occurs. If the check evaluates to true, then time is saved since the code has already been executed. If not, the hardware must correct the mistake. In the case of the peg-hole cylinder, due to its small surface area and elongated shape, maybe this check fails more often than it succeeds compared to when the reamer is used. Ultimately, what this means is that if working with tools with small surface areas, chunk sizes of 16^3 should be used. Otherwise, for larger tools, 32^3 chunks are more appropriate, but no larger. On a final note, only power of 2 sizes were investigated, as these lengths enable index-to-3D grid position conversion optimizations (i.e.: using bit shifts rather than modulo to calculate x, y, z coordinates).

Having decided on a chunk size of 32^3 for general purpose use, the chunk collision system was put under load and run through various Windows 10 PC configurations. The complete specifications for the PCs tested are found in Table 3.6 below.

Table 3.6: Specifications for the Windows 10 PCs used over the course of engine testing

CPU		**GPU**				**RAM**
Processor	Base/Boost freq. (GHz)	GPU	Base/Boost freq. (MHz)	Cores	VRAM (GDDR5)	
Intel Core i5 8600	3.1 / 4.3	Nvidia GTX 1070	1506 / 1683	1920	8	8 GB
Intel Core i7 6700K	4.0 / 4.2	Nvidia GTX 970	1050 / 1178	1664	4	16 GB
Intel Core i7 8700K	3.47 / 4.7	Radeon RX 580	1257 / 1340	2304	4	16 GB
Intel Core i5 6300u	2.4 / 3.0	Intel HD Graphics 520	300 / 1000	N/A	N/A	8 GB

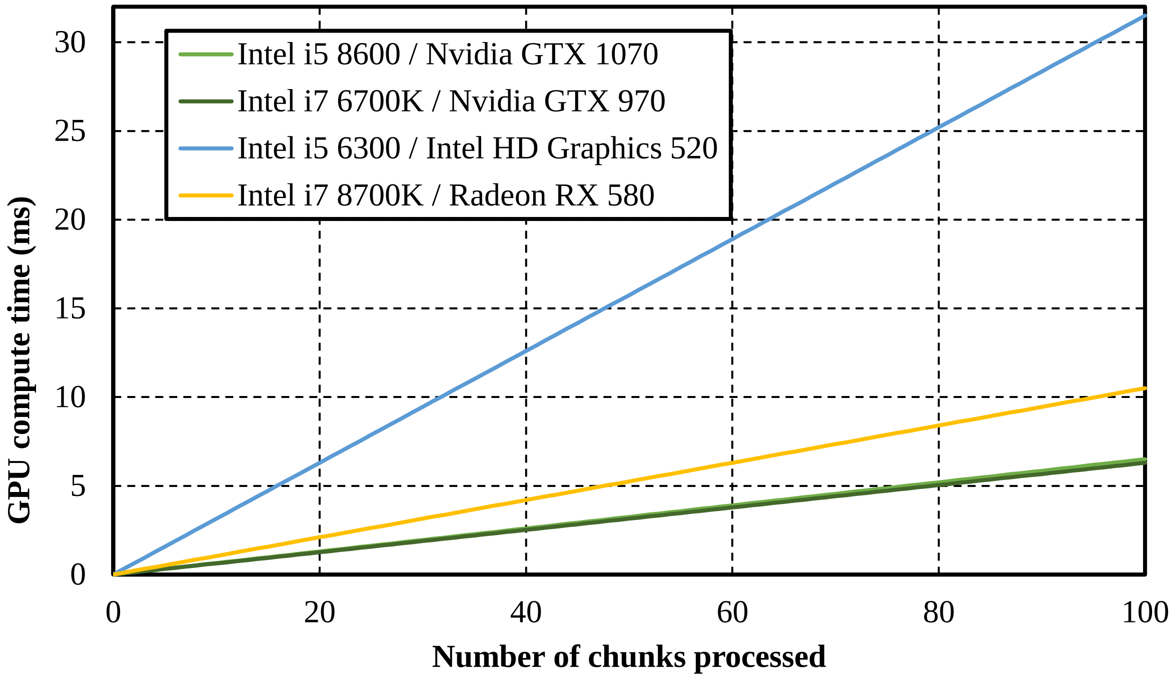


Figure 3.8: Chunk collision task computation time using the peg-hole cylinder tool with varying PC configurations

Several conclusions can be extrapolated from the results of figure 3.8. Firstly, discrete GPUs clearly outperform integrated graphics units (Intel HD Graphics 520), signifying that dedicated graphics cards are necessary for peak performance. Secondly, the GPU, and not the CPU, is what determines the relative processing performance of the chunk system, which is to be expected considering all voxel-related tasks are OpenCL accelerated (i.e.: ran on the GPU). As such, the PC equipped with an Intel Core i7 8700K (i.e.: the strongest tested CPU), did not have the best compute times because its GPU, the Radeon RX 580, was not as powerful as the Nvidia GTX 1070 and Nvidia GTX 970. Unexpectedly, the Nvidia GTX 1070 and Nvidia GTX 970 had almost identical compute performance, despite the 1070 being 970’s successor. Being only one generation apart, perhaps the hardware did not evolve enough to drastically impact raw compute power.

3.2.3 Chunk system collision testing performance compared to single-buffer-based algorithms

The chunk system is an approach that was designed to address a common problem in collision testing for voxel-based simulations, which is high-volume computation strain. The sheer number of voxels present during any given collision event requires that the algorithms used to perform intersection checks and removals between individual voxels must perform at sub millisecond speeds, especially when taking haptics input at 1 kHz. While the chunk system is capable of meeting target speeds at low chunk numbers (Fig. 3.8), it is one of many algorithms that have been created to address this problem. Other algorithms exist that have offered solutions to OpenCL accelerated voxel-based collision testing. These algorithms differ from the chunk system in that they iterate through, and work on, tool voxels contained in single sparse buffers. For these algorithms, the bone voxels are also contained in a single buffer, although organized as a 3D matrix that covers the bone's object bounds [75, 114, 115].

To assess the relative performance of the chunk system, collision test timings were compared against those generated by the methods developed by Zheng et. al, Yau et. al, and Reza et. al [75, 114, 115]. When comparing the performance of chunks in terms of sheer voxels processed per millisecond, the chunk system struggles to reach parity. For instance, at raw voxel processing counts of 64^3, if there is a collision event between the reamer and the bone (Fig 3.6), the chunk system runs at 0.44 ms, while the leading algorithm, Reza et. al's, runs at 0.19 ms. Performance is substantially worse when processing voxel counts of more than 128^3 (Fig. 3.9).

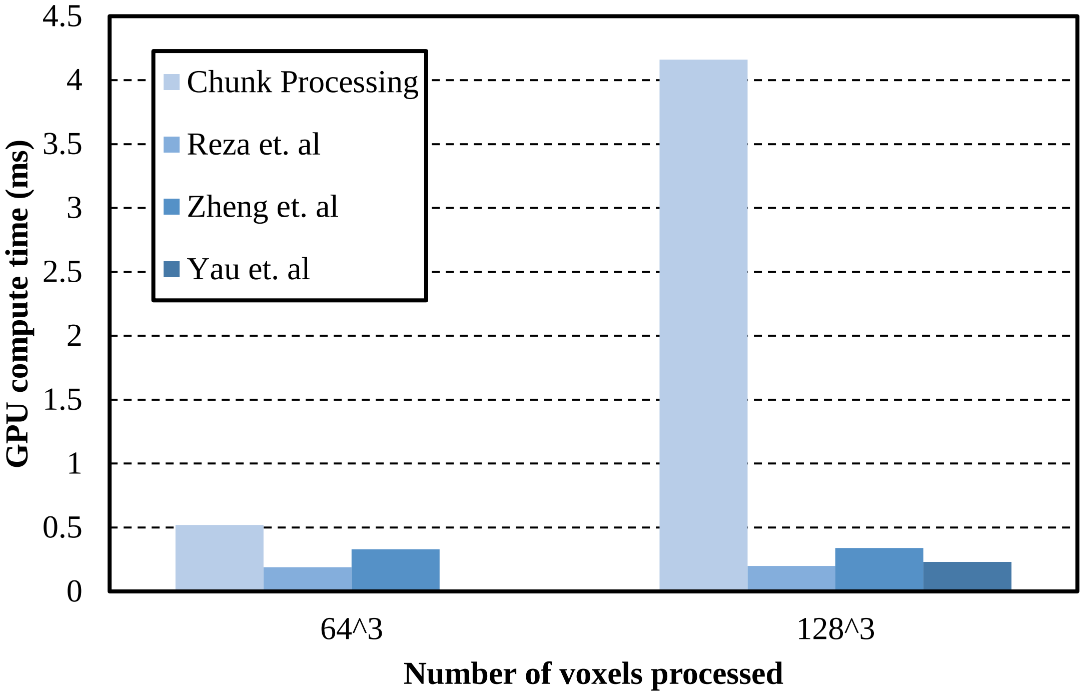


Figure 3.9: Voxel processing time of various collision algorithms on Windows 10, Intel i5 8600, Nvidia GTX 1070 PC. Chunk collision was performed with reamer.

The explanation for this discrepancy is due to the fundamental way the chunk system handles collisions, in that the chunk system processes voxels from the bone voxels to tool voxels (Fig 3.3), rather than the reverse, which is the case for the alternative algorithms (Fig 3.4) [75, 114, 115]. In both methods, tool voxels are stored as a single continuous array of elements, although in this engine, tool voxels are arranged as a 3D matrix, meaning that there are empty voxels present. The alternatives utilize a sparse voxel buffer, which is a linear array of only active voxels that aren't necessarily in spatial order. Sparse buffers have the advantage of removing inactive voxels, thereby minimizing the number of voxels to process during collision testing. These algorithms can store tool voxels this way because they iterate through the tool voxels during collision events. It is not possible to have a sparse array for tool voxels in a bone chunk system since the world position of the tool voxel is no longer tied to a buffer index that can be used to calculate a coordinate in a grid space that is relative to the tool. Determining the tool voxels that are involved in a collision intersection volume would thus require additional metadata per voxel to house a 3D world position. It would also necessitate iterating through the tool's sparse voxel buffer during collision detection since these voxels aren't in a predictable order (i.e.: would vary from tool to tool), which would be extremely costly in terms of computation. In these types of algorithms, a sparse voxel buffer is allowed for only one of the two colliding entities (the one who's voxels are being iterated through to check for collisions against the other), but not both since grid positional information is lost when using a sparse organisation. Now, understanding the nature of sparse buffers is important because, although the algorithms claim to process 64^3 voxels, those numbers instead indicate the volume represented by the buffers. The total number of voxels processed is much lower since most of the void space present in the tool objects was stripped away during the voxelization process. On top of that, the voxelization algorithm used only considers the surface of the object, leading to fewer voxels being generated to represent the tool [75, 114, 115]. Chunks, on the other hand, derive their data from volumetric sources (i.e.: CT scans), meaning that, overall, they contain more voxels, which results in more computation time required to parse multiple chunks.

However, rather than raw voxel processing, it is better to compare the chunk system on a chunk-by-chunk basis. Since it is possible to have a variable number of chunks during collision events, it is better to analyze how the chunk system scales as more chunks are processed and note the conditions in which the chunk system wins, achieves parity, and plateaus compared with the alternatives. Starting with the method developed by Reza et. al, it is clear that Reza et. al's method is dependent on the voxel resolutions of both the tool and bone objects being used (Fig 3.10) [114].

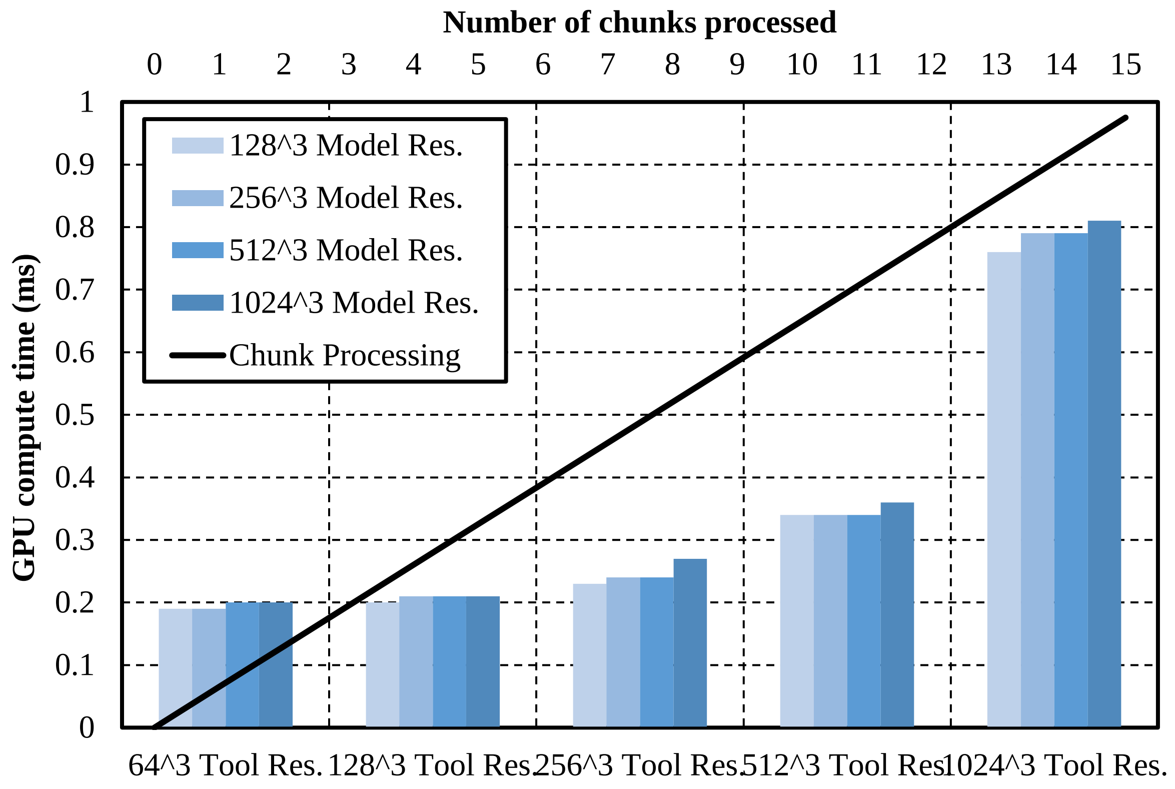


Figure 3.10: Chunk system collision timings using peg-hole cylinder tool vs Reza et. al's timings at varying tool and model voxel resolutions.

The takeaway of figure 3.10 is that the chunk system matches or beats Reza's algorithm only under certain conditions, which are dictated by however many chunks are being processed at the time of a collision event, and the resolutions of the virtual objects. For instance, at the highest tool and model (bone) resolutions measured (i.e.: 1024^3), the chunk system performs better if 12 or fewer chunks are being processed (Fig 3.10). Otherwise, Reza et. al's algorithm wins since it takes less time to complete a collision event. In comparing the chunk system against Zheng et. al and Yau et. al, a similar trend is observed (Fig 3.11-3.12). The chunk system does not necessarily outperform any existing method, but rather, there are instances where it matches performance, and where it surpasses.

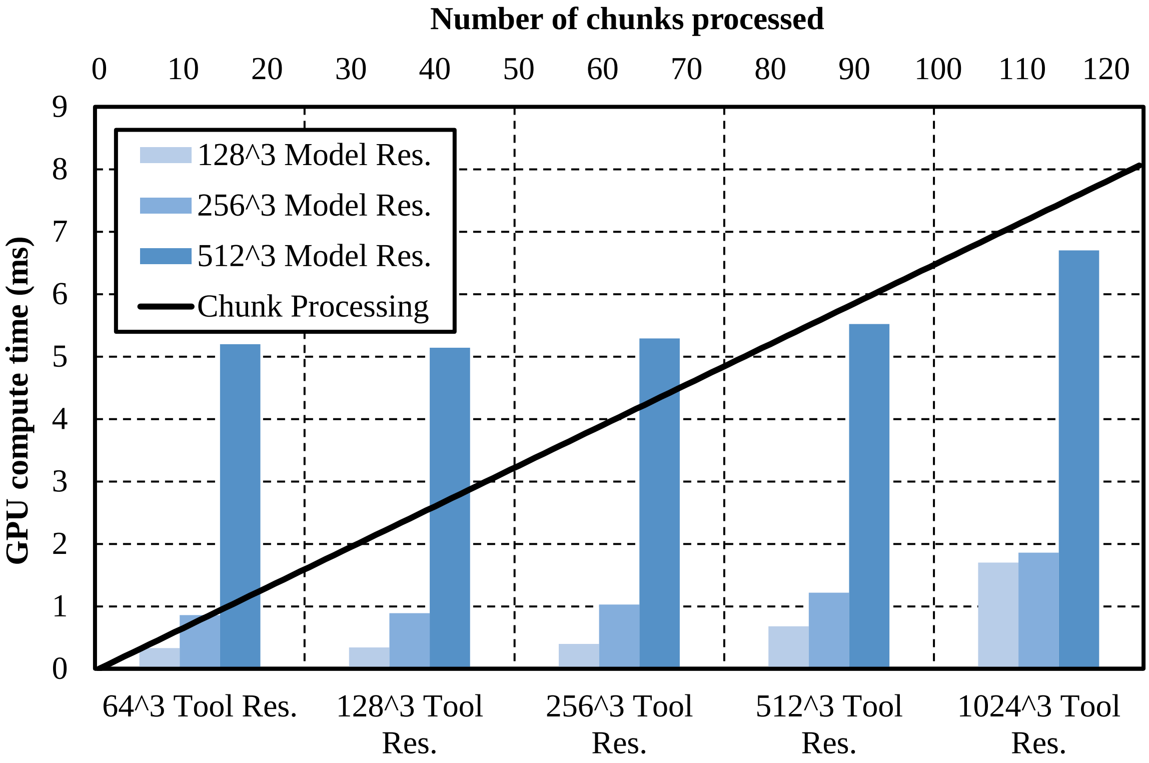


Figure 3.11: Chunk system collision timings using peg-hole cylinder tool against those measured when using Zheng et. al's method

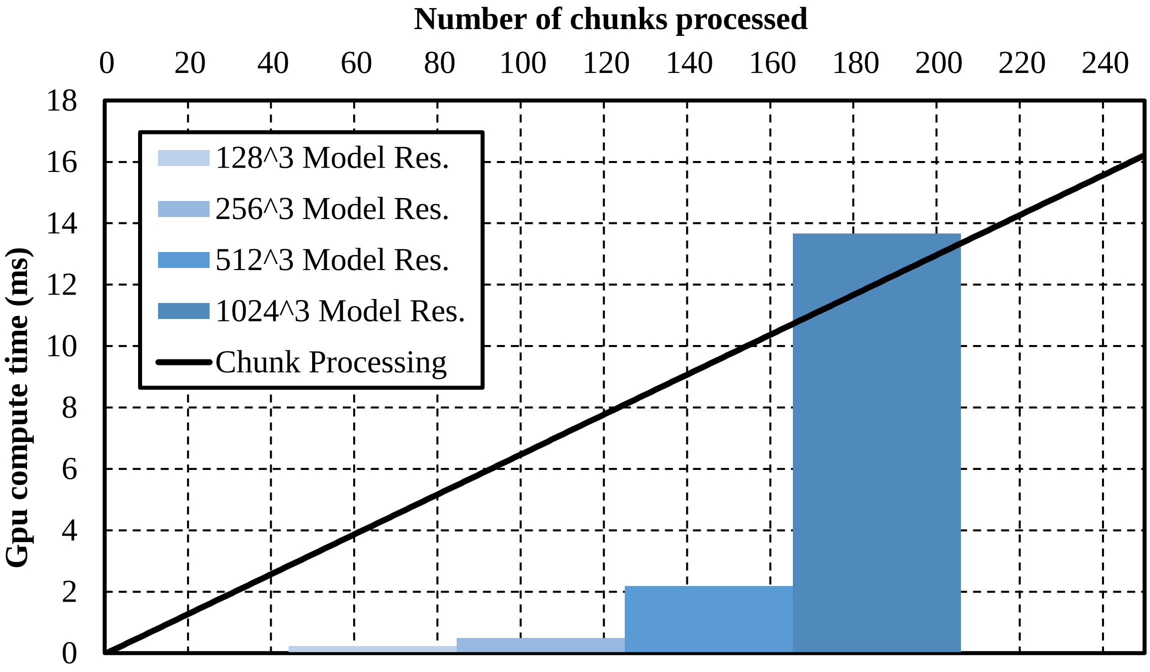


Figure 3.12: Chunk system collision timings using peg-hole cylinder tool against those measured when using Yau et. al's method

In practicality, the chunk system achieves performance that is comparable to the single-buffer based algorithms. The collisions that occurred during testing were located on the periphery of the bone object, meaning that the bone-tool intersection volume remained relatively small. Given this, the number of chunks that were processed during collision testing were normally between 1 – 20 (Table 3.4). At these numbers, the time required to compute a collision event varied between 0.065 ms and 1.17 ms, which is within the acceptable range of the target time allotted for 1 kHz haptics updates. While the chunk system isn’t a perfect solution, it can achieve real-time material removal with results that are comparable with the alternatives. In the end, the chunk system performs best if the number of chunks to be processed is low.

3.2.4 Chunk system complexity

The complexity of the collision system in the context of chunks is O(n * m) where n is the number of voxels within a chunk (all of which must be processed per chunk collision test) and m is the number chunks that need to be processed per collision event. Although the size of the chunk is constant and is set at compilation, since it can be varied ahead of time, it is still variable. Now, it is important to take into consideration the complexity of an algorithm to obtain a rough estimate of the computation power required to perform a task. Looking at the single buffer approach, the complexity is O(n), were n is the number of tool voxels being processed per collision event. What is key is how the growth of the chunk system compares with that of the single buffer approach. As shown in figures 3.10 – 3.12, assuming the highest voxel resolution of both tool and bone objects, the chunk system grows much more slowly in terms of computing power required compared to the single buffer approaches, providing better performance despite being more complex. Finally, the chunk system offers efficiency over a naïve approach of checking every possible combination of intersecting bone-to-tool voxels. Doing so would increase the computation cost exponentially ($O(n^m)$) and wouldn’t be feasible regardless of hardware power.

3.2.5 Chunk system vs. octrees

The chunk system is an alternative to the single-buffer based approach of storing voxels. However, it shares similarities with another data structure known as octrees [29]. Octrees are a tree data structure where each node contains either no child nodes or exactly 8. Essentially, octrees subdivide voxel space into regions of 8 units. Because nodes can have no children, inevitably, there will exist sections in the octree that are more subdivided than others. The reason for this subdivision is due to the metadata of the voxels within that region. Voxels of equal or similar metadata are grouped under one node while others are pushed into subdivisions until the node is homogenous in content [29]. This is done as a memory optimization wherein large groups of voxels can be represented by a few nodes, which, individually, do not occupy a lot of memory. Like the chunk system, octrees partition voxels into sections, and the chunk system can even be thought of as an octree with a fixed height of 1 and unlimited nodes. Therein lies the key difference. The chunk system does not subdivide voxels any further. In practicality, this means several things. Lookups in the chunk system are O(1) since there is no tree traversal to find a voxel anywhere in the voxel representation of the model. This, however, comes at a cost of memory storage as chunks do not subdivide as optimality as the octree structure to merge similar voxels. Additionally, chunks do not restructure themselves. An octree has the option to refresh nodes based on current state of the voxel representation after a collision event. This intermediary step could prove beneficial in later collision events as the traversal of the tree may end up being shorter with more and more voxels being removed. However, this intermediary step would still cost computing time, time which is saved using the chunk system. Overall, the chunk system was used as the voxel data structure for this engine for its fast lookup times despite a potential increase in memory usage.

3.2.6 Independency from tool voxel resolution

A benefit of the chunk system, compared to the alternatives, is that the performance of collision testing is independent of the voxel resolution of the tool object (Table 3.5). As the chunk system’s collision algorithm goes from bone chunk voxels to tool voxels, the time required to perform a complete collision event is entirely dependent on the number of bone chunks being processed. Increasing the tool’s voxel resolution does not affect performance, as the physical size of the tool object remains the same. Thus, the intersection volume, along with the number of colliding chunks, remains constant (Table 3.5). However, whether this property enables more accurate collision results with a finer grained tool voxel representation is untested. Still, contrast this property to the alternatives, which show a gradual increase in the computation time required to perform a collision event as the tool’s voxel resolution increases (Fig 3.10-3.12).

3.3 Chunk voxel rendering

3.3.1 Neighbor-aware triangle mesh generation

Early in development, triangle meshes were created, and refreshed, for each chunk to act as their visual representation in virtual space. The chunk triangle mesh creation process culminated in a mesh that appeared to be composed of a series of cubes, wherein each cube represented one voxel (Fig 1.3). As a performance enhancement, a preprocessing step would occur were cube faces were culled from the final mesh if they were obscured by an adjacent, opaque voxel face. The algorithm worked on a chunk-by-chunk basis and would create a seamless mesh across chunks by considering the voxels of neighbouring chunks (Fig 2.2). Essentially, for voxels that are on the external layers of the query (i.e.: central) chunk, the algorithm would look to the first layer of voxels in the neighbouring chunk immediately adjacent to the current voxel and decide whether a quad should be added into the chunk's triangle mesh vertex buffer based on the presence, or absence, of the neighbour voxel. For quick access, each chunk would have an array of 6 pointers to its neighbours. However, if a neighbouring chunk was not present (i.e.: the query chunk is located on the outer layers of the chunk grid), then the pointer would point to a global "empty" chunk, a specialized read-only chunk where all voxel densities are zero. This enabled the algorithm to work as expected. Neighbor chunk pointers would be setup during chunk grid creation in the CT scan data transfer task. Each chunk triangle mesh would then stored with its respective chunk and would be updated when the chunk's voxel contents were changed due to a collision event.

The triangle mesh creation process is performed entirely on the GPU through OpenCL and is as follows. Voxel data from the query chunk and its neighbors are transferred to a 34 × 34 × 34, R16, 3D texture called the "SuperChunk". The SuperChunk has extended dimensions to add the 6 immediately adjacent neighbouring chunk layers, and to avoid unnecessary conditional branches when performing neighbour voxel lookups to determine whether a quad face should be added to the mesh. When copying data to the SuperChunk, all the voxels from the query chunk are transferred, starting from index 1 × 1 × 1 to 32 × 32 × 32. Then, voxel data from neighbouring chunks are added. Essentially, one 32 × 32 face layer of voxel data from each neighbouring chunk is added to the SuperChunk. The layer of chunk data added depends on the face of the neighbour chunk that is directly adjacent to query chunk. In the end, the SuperChunk contains all the voxel data needed to go through voxels 1 × 1 × 1 to 32 × 32 × 32, while being neighbor aware.

After writing to the SuperChunk, a parallel prefix sum and stream compaction operation is performed. Generation of the triangle mesh is performed on the GPU to avoid unnecessary transfer of vertex data to the CPU and back OpenGL vertex buffers. To reduce workload and be able to properly index into the output vertex buffer, active voxel (i.e.: voxels that have at least one visible face, meaning a neighbour with < 120 density) and active face (to determine the number of quads needed to generate per active voxel) indices are identified in one kernel, summed in another kernel, and then stream compacted to buffers in a third kernel. The active voxel identification, parallel prefix sum, stream compaction algorithms are taken from the Marching Cubes sample in Nvidia's OpenCL SDK, with the difference being that active voxels and face indices are identified based on the absence of active neighbouring voxels. Once the compacted active voxel and face indices arrays have been computed, if there are any active voxels, the buffers are iterated through in a final OpenCL kernel to generate the triangle mesh. For every active voxel, the 6 neighbors of the voxels are checked, and for each neighbor that is empty, the vertex coordinates of the face are computed at the offset of the current compacted face index + 4 * the index of the face currently being checked. In the end, a triangle mesh of cubes is generated. A final step is to transfer the OpenCL buffer output to the Chunk's OpenGL mesh buffer.

3.3.2 Ray-box intersection voxel rendering

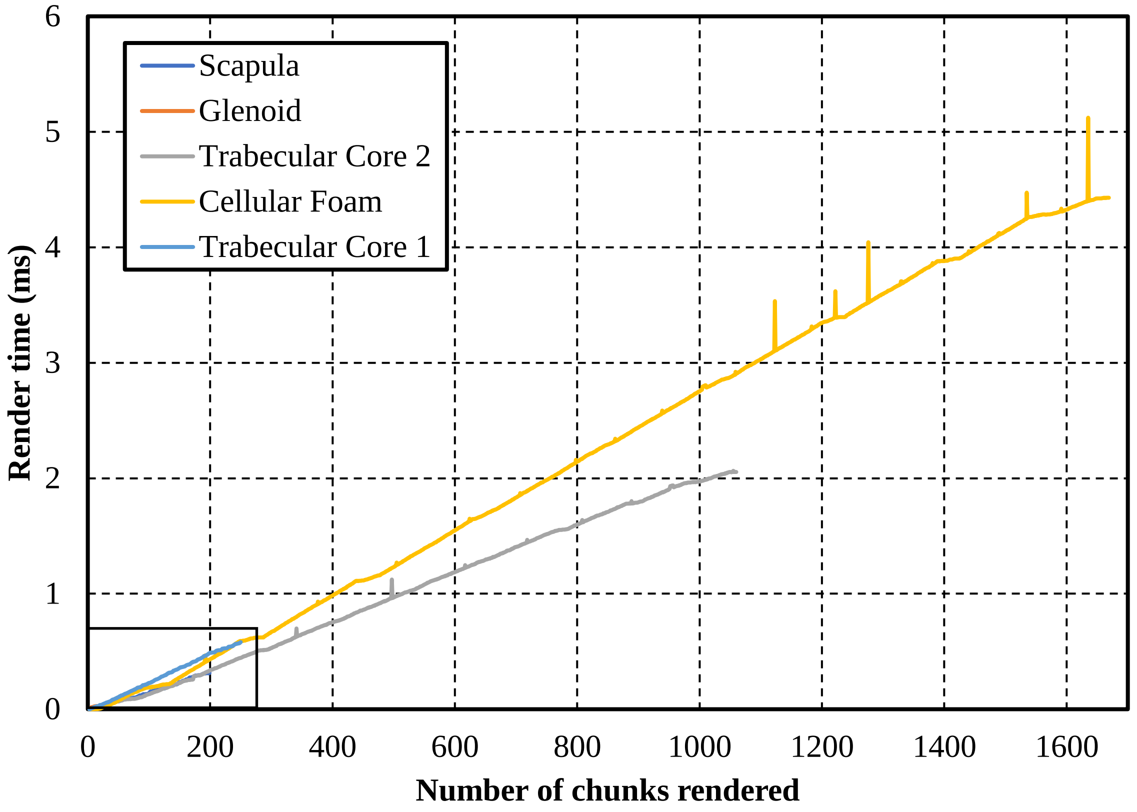
Recently, a novel method has been released by Majercik et. al that offers an alternative to the mesh-based approach of rendering voxel-based models [112]. The algorithm gives the same visual result as the triangle-mesh method, although the performance differs heavily, especially when viewed in conjunction with the added computational stress of collision testing. To summarise the technique, a bounding box is rasterized from a GL_POINT axis-aligned square for each model voxel in a vertex shader, and then a ray-box intersection test is performed on every pixel within that box in the accompanying fragment shader. If the test passes, the pixel is shaded according color of the voxel and bounding-box face hit. If the test fails, the pixel is discarded, giving the final image. To accommodate this technique for the chunk system, several changes had to be made. First, in terms of the shaders, the vertex and fragment shader were changed to be called per chunk, and each voxel had to have its position computed based on the chunk's relative grid offset and the current model matrix of the voxelized object. Second, the CT data used for the chunks had to be modified to be compatible with the algorithm. The GL_POINT based technique assumes that every voxel has equal dimensions, given that OpenGL's GL_POINTS and GL_POINT_SIZE functions manipulate square points, which can only represent cubes. However, the Scapula and Glenoid CT scans used for engine testing were generated with an irregular voxel size (0.47 mm × 0.47 mm × 1.0 mm). Given the restriction, one would have either alter the CT scan data, or the algorithm itself. The solution chosen was to introduce "filler" voxels along with the CT scan data, essentially subdividing the existing voxels, but occupying the same volume (see Chapter 2, 2.4, Fig 2.2). The triangle-mesh approach does not have this problem since the triangle vertices can be scaled to the voxel size of the CT scanned object in the vertex shader. Interestingly, this method does not incur any additional memory cost over the triangle mesh method, since instead of memory being occupied by filler voxels, memory is instead taken up by the triangle mesh buffer (Table 3.7).

Table 3.7: Various statistics of the voxel rendering methods used.

	Scapula (Triangle mesh)	***Scapula (GL_POINTS)***
Voxel Size	0.47 × 0.47 × 1.0) mm	(0.47 × 0.47 × 0.47) mm
Number of chunks	448	896
Number of Active chunks	99	199
Active voxel memory (2B/voxel)	6.48 MB	13.04 MB
Triangle mesh memory	6.93 MB	0.00 MB
Total memory	13.41 MB	13.04 MB

3.3.3 Comparison of the voxel rendering methods

The rendering algorithms were tested with 5 CT scanned objects (Fig 1.4), and timing data was collected when the engine was at idle, and when it experienced a collision event. Additionally, the rendering times of chunks were recorded in increments of 1 to determine the base time required to render a set number of chunks using each method. The results are shown in figures 3.13 - 3.14. It was found that the triangle-mesh method had lower rendering times (up to 6x) per chunk than the GL_POINT based approach. Perhaps the reason for this is due with the varying number of vertex elements processed between the methods. In the GL_POINT method, each voxel in a chunk is processed in the shaders (32,768 voxels for chunk sizes of 32^3), meaning that every chunk rendered has consistent vertex shader invocations. On the other hand, a triangle-mesh is much more likely to contain fewer elements to process, given that only the surface of the voxelized object is being drawn. Inner voxels that are obscured by neighbouring voxels have their quads omitted, while all voxels in a Chunk are processed in the GL_POINT method. The shaders used for the triangle mesh are also simpler, requiring fewer matrix transforms, and no ray-tracing in the fragment shader.



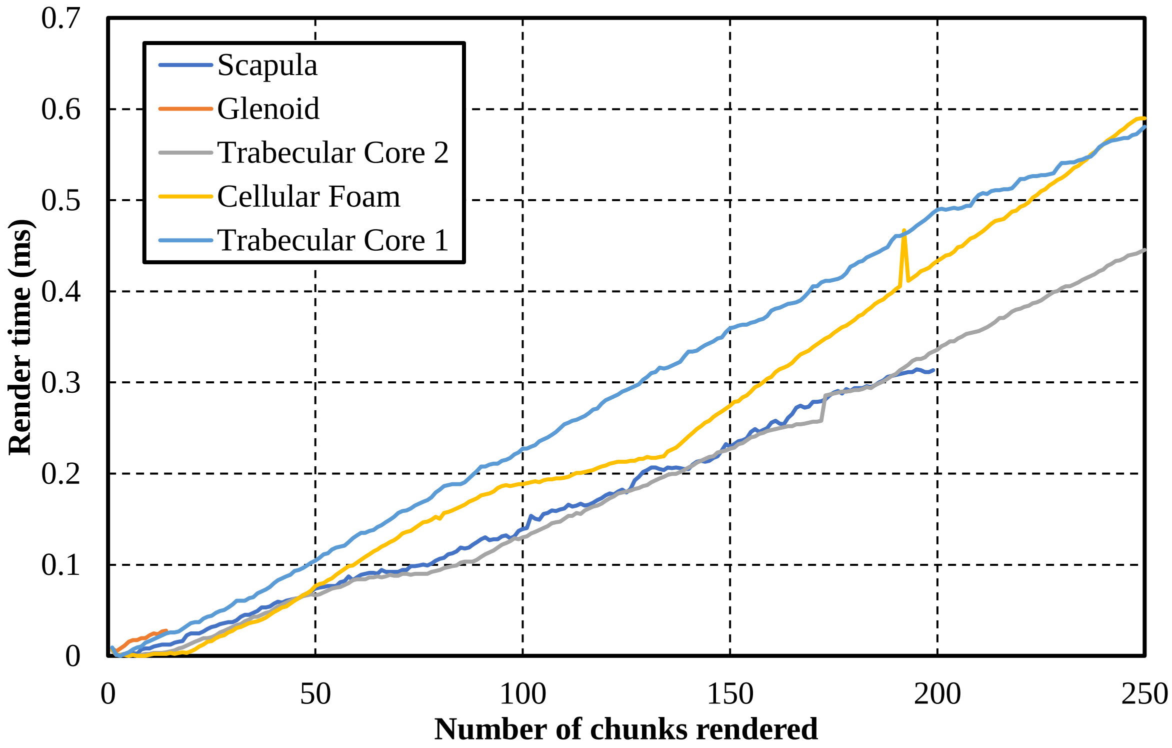


Figure 3.13: 720p render timings for triangle mesh voxel rendering of 32^3 chunks across tested models on Windows 10, Intel i5 8600, Nvidia GTX 1070 PC

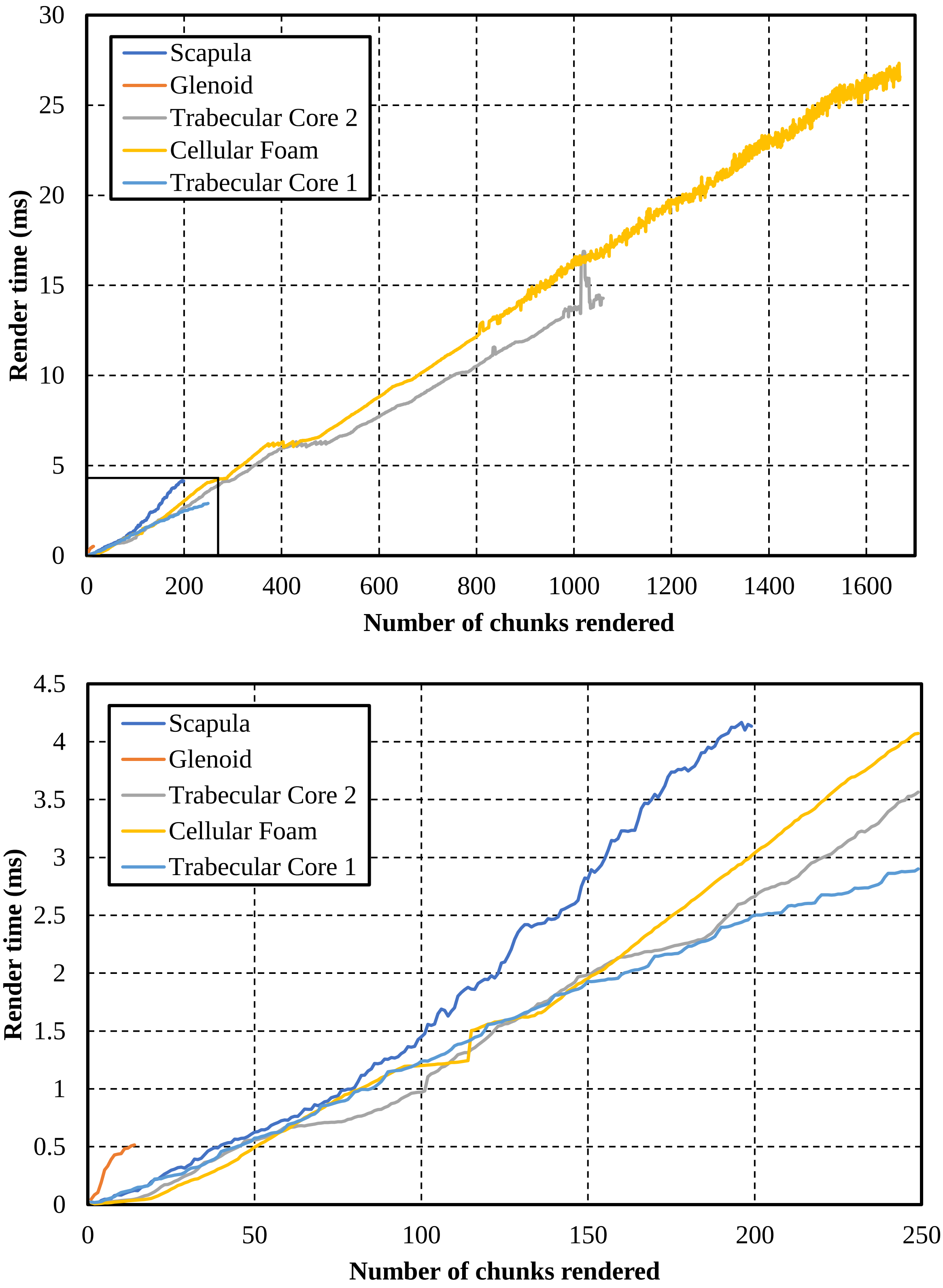


Figure 3.14: 720p render timings for GL_POINT based voxel rendering of 32^3 chunks across tested models on Windows 10, Intel i5 8600, Nvidia GTX 1070 PC

On figure 3.13, there appears to be performance spikes that occur periodically when rendering a high number of chunks. These spikes are consistent across several runs measuring the rendering timings of the triangle-mesh algorithms. V-Sync does not appear to affect these spikes. The current hypothesis is that the spikes are caused by cache misses. However, figure 3.13 is more of a visual aid to get an estimate of the expected time required to render a set number of chunks using the triangle-mesh method. The same for figure 3.12 with the GL_POINT based approach.

Before continuing, a note must be made on why chunks have different render times between the CT scanned objects tested, despite each chunk having the same number of voxels (i.e.: 32^2). In the shaders, an optimization is made to skip over inactive voxels, and only continue if the voxel is active.

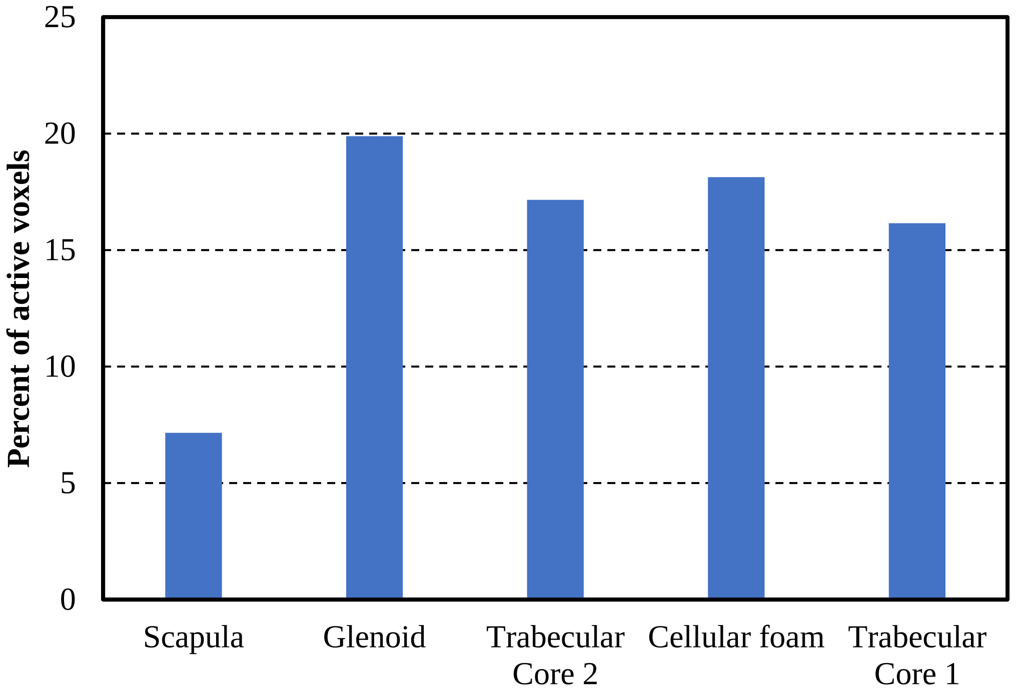


Figure 3.15: Ratio of active to inactive voxels across tested models

The more active voxels a voxelized object has, the longer it will take to render chunks of that object. The ratio of active-to-inactive voxels plays a role in the render times (Fig

3.15). This, along with the variability of the active voxel ratios across the CT scanned objects, would explain the trends observed in figures 3.13-3.14.

Given how the triangle mesh method performs, one would expect it to be the logical choice when implementing a rendering scheme in a simulation engine. However, figures 3.13 - 3.14 show render timings at idle. The more important comparison is how both methods perform when the engine undergoes collision testing. The key difference here is that the triangle-mesh method must regenerate chunk meshes to accurately reflect the changes made during a collision update, while the GL_POINT based method does not have any additional intermediary steps. Effectively, this means that for every unique chunk whose data has been changed due to a collision event, the triangle-mesh generation algorithm must be re-executed, which comes at a cost (Table 3.8).

Table 3.8: Breakdown of the runtime for different phases of chunk triangle mesh generation

Task	*GPU compute time (ms)*
Copy chunk cluster data to padded Chunk	0.077047
Classify active voxels in padded chunk	0.067482
Generate sparse active voxels buffer with parallel prefix scan	0.081673
Generate triangle mesh and transfer to global mesh buffer	0.018723
Copy global mesh buffer data to chunk mesh buffer	0.007131
Total triangle mesh computation time	0.252056

For every 4 chunks that require an update after a collision event, ~1 ms is required to refresh the triangle meshes of those chunks. Recall that the target visual refresh rate is 60 Hz and the target haptics update rate is 1 kHz. Since the mesh regeneration process is too costly to be done in the update loop, it is performed in the render loop instead. With an allocation of ~16 ms per frame, minus the amount of time required to render the tool models, allowing room for future graphics enhancements, and using the Scapula model as an example (0.58 ms to render all the chunks), roughly 56 (14 ms * 4 chunk per ms) total regeneration events can occur before performance degrades. However, this cost is on top of the time needed for the update loop, and since anywhere from 1 – 20 unique chunks are affected per collision event (Table 3.3), the number of refreshes allowed is too low. More importantly, this method is not scalable if using tools that cover larger surfaces that would increase the number of collided chunks to process. Given these restraints, the GL_POINT base method was chosen as the default voxel rendering technique for the engine.

Having chosen the GL_POINT method as the voxel rendering method of choice, timing data was collected to assess the performance of the algorithm across various PC configurations (Table 3.6).

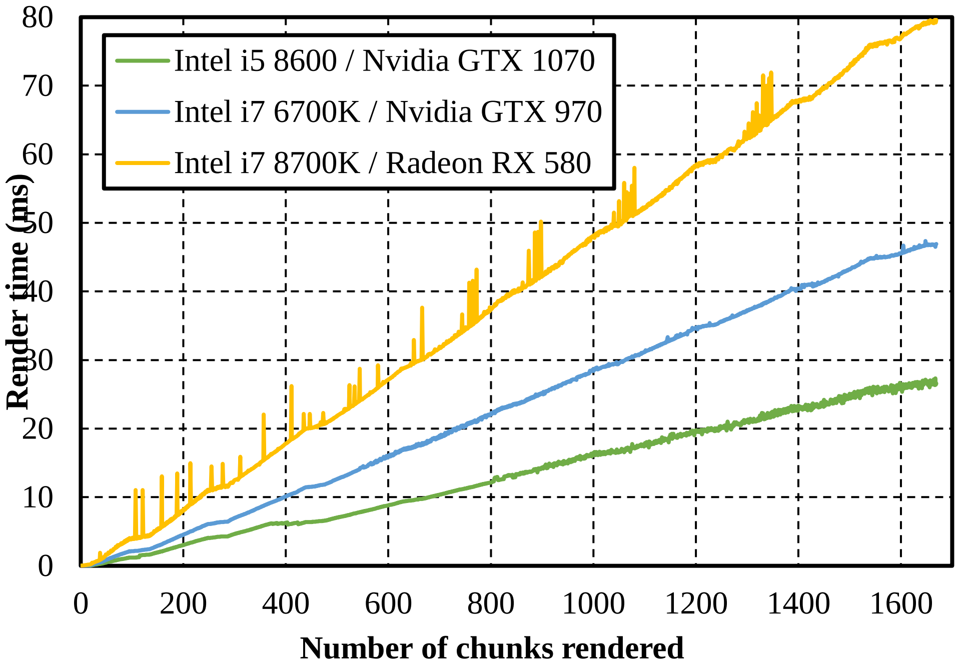


Figure 3.16: 720p render timings for GL_POINT based voxel rendering for "Cellular Foam" model across varying Windows 10 PC configurations

Again, the determining factor for performance was the GPU, with the Nvidia GTX 1070 performing the best, the Nvidia GTX 970 coming in second, and the Radeon RX 580 ending in 3rd (Fig 3.16). The cause for the spikes on the Radeon GPU is unknown.

On a final note, it is worth mentioning that there do exist isosurface extraction methods (ex: Marching Cubes) that can generate smooth triangle meshes out of voxels [120]. However, the fact remains that rebuilding meshes will cost some amount of compute time regardless of the number of refreshes required. Ultimately, the GL_POINT based method gives the same visual result without an intermediary step. Volume rendering is used in medical applications to rendering voxels without meshes, but the technique wasn't explored, as there were difficulties adapting the process to be compatible with the chunk system [113].

Chapter 4

Thesis closure

4.1 Summary

The focus of this project was to develop a simulation engine platform whose code and techniques may be integrated into current and future simulators for the benefit of the medical community. Over the course of the engine's development, a partitioning scheme, known as the "Chunk" system, had been introduced to address common bottlenecks when attempting to simulate the complex interactions between objects of arbitrary voxel resolutions, those being memory consumption and high-volume workloads. The chunk system has successfully resolved the issue of memory management, given that, on average, there was a reduction in the amount of memory allocated by voxels, sometimes up to 50 % (Tables 3.1-3.2). As for the performance of collision testing, the results of the chunk system, when compared to alternative methods, varied depending on the size of the workload. To summarize, the chunk system is not the ideal solution that will outperform all methods in all conditions. Rather it is more of a proof-of-concept that a voxel partitioning scheme can match the performance of single buffer algorithms at various settings. In most cases, the chunk system will perform adequately given that most surgical scenarios involve surface remodeling and small intersection volumes between the cutting tool and the bone. As such, the system is unlikely to be processing high amounts of chunks at any one time, and thus the simulator will run at the expected real-time refresh rates of 60 Hz visual, and 1 kHz haptic update.

4.2 Future work

Performance enhancements are a continual focus for simulation engine development. Key areas are reducing the number of collision detection tests to a minimum and implementing more efficient rendering techniques. In terms object-to-object interaction, a switch from the current broad-phase collision detection scheme to a more refined means collision detection would greatly speedup the haptics/update loop [121]. The issue is that the computed bounding box intersection volume sometimes overestimates the number of chunks that collide with the reamer, causing an unnecessary amount of chunk collision test executions. Using capsule colliders or convex hull colliders, objects that more tightly wrap around models than bounding boxes, would be worth investigating to alleviate the problem. The best rendering optimization is to reduce the number of objects rendering at any time. Frustum culling could be introduced so that chunk’s that are outside the camera’s frustum are not rendered [110]. However, another form of culling, occlusion culling, can also be implemented [122]. Occlusion culling with software rasterization can identify chunks that are obscured by other chunks, or otherwise hidden from the camera’s view, and prevent those chunks from rendering. These chunks most likely include those that are on the inside an object, or on the side of the object opposite of the camera. Finally, as a feature enhancement, adding VR headset (ex: HTC Vive) compatibility to the engine would greatly improve the user experience, as the user would be able to physically re-enact the surgery in 3D virtual space.

4.3 Conclusion

The simulation engine described here has achieved its goal of replicating the complex interaction between an arbitrary cutting tool and a CT scanned object in real-time. We believe the algorithms developed will contribute to the foundation of current upcoming surgical simulators, accelerating the progression of these training modules.

Bibliography

1. Halm, E.A., C. Lee, and M.R. Chassin, *Is volume related to outcome in health care? A systematic review and methodologic critique of the literature.* Annals of internal medicine, 2002. **137**(6): p. 511-520.
2. Luft, H.S., S.S. Hunt, and S.C. Maerki, *The volume-outcome relationship: practice-makes-perfect or selective-referral patterns?* Health services research, 1987. **22**(2): p. 157.
3. Bell Jr, R.H., et al., *Operative experience of residents in US general surgery programs: a gap between expectation and experience.* Annals of surgery, 2009. **249**(5): p. 719-724.
4. Flood, A.B., W.R. Scott, and W. Ewy, *Does practice make perfect? Part I: The relation between hospital volume and outcomes for selected diagnostic categories.* Medical care, 1984: p. 98-114.
5. Flood, A.B., W.R. Scott, and W. Ewy, *Does practice make perfect? Part II: The relation between volume and and outcomes and other hospital characteristics.* Medical care, 1984: p. 115-125.
6. Luft, H.S., *The relation between surgical volume and mortality: an exploration of causal factors and alternative models.* Medical care, 1980: p. 940-959.
7. Birkmeyer, J.D., et al., *Hospital volume and surgical mortality in the United States.* New England Journal of Medicine, 2002. **346**(15): p. 1128-1137.
8. Taylor, H.D., D.A. Dennis, and H.S. Crane, *Relationship between mortality rates and hospital patient volume for Medicare patients undergoing major orthopaedic surgery of the hip, knee, spine, and femur.* The Journal of arthroplasty, 1997. **12**(3): p. 235-242.

9. Luft, H.S., J.P. Bunker, and A.C. Enthoven, *Should operations be regionalized? The empirical relation between surgical volume and mortality.* New England Journal of Medicine, 1979. **301**(25): p. 1364-1369.

10. Brennan, T.A., et al., *Incidence of adverse events and negligence in hospitalized patients: results of the Harvard Medical Practice Study I.* New England journal of medicine, 1991. **324**(6): p. 370-376.

11. Leape, L.L., et al., *The nature of adverse events in hospitalized patients: results of the Harvard Medical Practice Study II.* New England journal of medicine, 1991. **324**(6): p. 377-384.

12. Ericsson, K.A., *Deliberate practice and the acquisition and maintenance of expert performance in medicine and related domains.* Academic medicine, 2004. **79**(10): p. S70-S81.

13. Duvivier, R.J., et al., *The role of deliberate practice in the acquisition of clinical skills.* BMC Medical Education, 2011. **11**(1): p. 101.

14. Thomas, G.W., et al., *A review of the role of simulation in developing and assessing orthopaedic surgical skills.* The Iowa orthopaedic journal, 2014. **34**: p. 181.

15. McGaghie, W.C., et al., *Does simulation-based medical education with deliberate practice yield better results than traditional clinical education? A meta-analytic comparative review of the evidence.* Academic medicine: journal of the Association of American Medical Colleges, 2011. **86**(6): p. 706.

16. Kunkler, K., *The role of medical simulation: an overview.* The International Journal of Medical Robotics and Computer Assisted Surgery, 2006. **2**(3): p. 203-210.

17. Froelich, J.M., et al., *Surgical simulators and hip fractures: a role in residency training?* Journal of surgical education, 2011. **68**(4): p. 298-302.

18. Issenberg, S.B., et al., *Simulation technology for health care professional skills training and assessment.* Jama, 1999. **282**(9): p. 861-866.

19. Lam, C.K., K. Sundaraj, and M.N. Sulaiman, *Computer-based virtual reality simulator for phacoemulsification cataract surgery training.* Virtual Reality, 2014. **18**(4): p. 281-293.

20. Chan, S., et al., *High-fidelity haptic and visual rendering for patient-specific simulation of temporal bone surgery.* Computer Assisted Surgery, 2016. **21**(1): p. 85-101.

21. Kühnapfel, U., H.K. Cakmak, and H. Maaß, *Endoscopic surgery training using virtual reality and deformable tissue simulation.* Computers & graphics, 2000. **24**(5): p. 671-682.

22. Ho, A.K., et al., *Virtual reality myringotomy simulation with real-time deformation: Development and validity testing.* The Laryngoscope, 2012. **122**(8): p. 1844-1851.

23. Kusumoto, N., et al., *Application of virtual reality force feedback haptic device for oral implant surgery.* Clinical oral implants research, 2006. **17**(6): p. 708-713.

24. Hsieh, M.-S., M.-D. Tsai, and Y.-D. Yeh, *An amputation simulator with bone sawing haptic interaction.* Biomedical Engineering: Applications, Basis and Communications, 2006. **18**(05): p. 229-236.

25. Gordon, M.S., et al., *"Harvey," the cardiology patient simulator: pilot studies on teaching effectiveness.* American Journal of Cardiology, 1980. **45**(4): p. 791-796.

26. Seymour, N.E., et al., *Virtual reality training improves operating room performance: results of a randomized, double-blinded study.* Annals of surgery, 2002. **236**(4): p. 458.

27. Lee, T.-Y., C.-H. Lin, and H.-Y. Lin, *Computer-aided prototype system for nose surgery.* IEEE Transactions on Information Technology in Biomedicine, 2001. **5**(4): p. 271-278.

28. Razavi, M., et al., *A GPU-implemented physics-based haptic simulator of tooth drilling.* The International Journal of Medical Robotics and Computer Assisted Surgery, 2015. **11**(4): p. 476-485.

29. Yau, H., L. Tsou, and M. Tsai, *Octree-based virtual dental training system with a haptic device.* Computer-Aided Design and Applications, 2006. **3**(1-4): p. 415-424.

30. Wang, D., et al., *Cutting on triangle mesh: local model-based haptic display for dental preparation surgery simulation.* IEEE Transactions on Visualization and Computer Graphics, 2005. **11**(6): p. 671-683.

31. Wu, J., et al., *Toward stable and realistic haptic interaction for tooth preparation simulation.* Journal of Computing and Information Science in Engineering, 2010. **10**(2): p. 021007.

32. Bowyer, M.W. and R.B. Fransman, *Simulation in General Surgery*, in *Comprehensive Healthcare Simulation: Surgery and Surgical Subspecialties*. 2019, Springer. p. 171-183.

33. Vaughan, N., et al., *A review of virtual reality based training simulators for orthopaedic surgery.* Medical engineering & physics, 2016. **38**(2): p. 59-71.

34. Peters, J.H., et al., *Development and validation of a comprehensive program of education and assessment of the basic fundamentals of laparoscopic surgery.* Surgery, 2004. **135**(1): p. 21-27.

35. Neyret, F., R. Heiss, and F. Sénégas, *Realistic rendering of an organ surface in real-time for laparoscopic surgery simulation.* The Visual Computer, 2002. **18**(3): p. 135-149.

36. Wilson, M., et al., *MIST VR: a virtual reality trainer for laparoscopic surgery assesses performance.* Annals of the Royal College of Surgeons of England, 1997. **79**(6): p. 403.

37. Kothari, S.N., et al., *Training in laparoscopic suturing skills using a new computer-based virtual reality simulator (MIST-VR) provides results comparable to those with an established pelvic trainer system.* Journal of laparoendoscopic & advanced surgical techniques, 2002. **12**(3): p. 167-173.

38. Vassiliou, M., et al., *The MISTELS program to measure technical skill in laparoscopic surgery.* Surgical Endoscopy And Other Interventional Techniques, 2006. **20**(5): p. 744-747.

39. Sperling, J.W., R.H. Cofield, and C.M. Rowland, *Minimum fifteen-year follow-up of Neer hemiarthroplasty and total shoulder arthroplasty in patients aged fifty years or younger.* Journal of shoulder and elbow surgery, 2004. **13**(6): p. 604-613.

40. Mabrey, J.D., K.D. Reinig, and W.D. Cannon, *Virtual reality in orthopaedics: is it a reality?* Clinical Orthopaedics and Related Research®, 2010. **468**(10): p. 2586-2591.

41. Wiet, G.J., et al., *Virtual temporal bone dissection: an interactive surgical simulator.* Otolaryngology—Head and Neck Surgery, 2002. **127**(1): p. 79-83.

42. Sui, J., et al., *Mechanistic modeling of bone-drilling process with experimental validation.* Journal of Materials Processing Technology, 2014. **214**(4): p. 1018-1026.

43. Lin, Y., et al., *Development and validation of a surgical training simulator with haptic feedback for learning bone-sawing skill.* Journal of biomedical informatics, 2014. **48**: p. 122-129.

44. Peng, X., et al. *Bone surgery simulation with virtual reality*. in *ASME 2003 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. 2003. American Society of Mechanical Engineers.

45. Morris, D., et al. *A collaborative virtual environment for the simulation of temporal bone surgery*. in *International conference on medical image computing and computer-assisted intervention*. 2004. Springer.

46. Shantz, J.A.S., et al., *The internal validity of arthroscopic simulators and their effectiveness in arthroscopic education.* Knee Surgery, Sports Traumatology, Arthroscopy, 2014. **22**(1): p. 33-40.

47. Mabrey, J.D., et al., *Virtual reality simulation of arthroscopy of the knee.* Arthroscopy: the journal of arthroscopic & related surgery, 2002. **18**(6): p. 1-7.

48. Heng, P.-A., et al., *Application to anatomic visualization and orthopaedics training.* Clinical Orthopaedics and Related Research®, 2006. **442**: p. 5-12.

49. Vankipuram, M., et al., *A virtual reality simulator for orthopedic basic skills: a design and validation study.* Journal of biomedical informatics, 2010. **43**(5): p. 661-668.

50. *Health indicators, annual estimates*. Table 13-10-0451-01 2010-2014 2019-03-03; Available from: https://www150.statcan.gc.ca/t1/tbl1/en/tv.action?pid=1310045101&pickMembers%5B0%5D=1.1&pickMembers%5B1%5D=2.1&pickMembers%5B2%5D=3.1.

51. Kim, S.H., et al., *Increasing incidence of shoulder arthroplasty in the United States.* JBJS, 2011. **93**(24): p. 2249-2254.

52. Tsai, M.-D., M.-S. Hsieh, and C.-H. Tsai, *Bone drilling haptic interaction for orthopedic surgical simulator.* Computers in Biology and Medicine, 2007. **37**(12): p. 1709-1718.

53. Kusins, J.R., et al., *Development of a vibration haptic simulator for shoulder arthroplasty.* International journal of computer assisted radiology and surgery, 2018: p. 1-14.

54. Karelse, A., et al., *A glenoid reaming study: how accurate are current reaming techniques?* Journal of shoulder and elbow surgery, 2014. **23**(8): p. 1120-1127.

55. Weishaupt, D., et al., *Posterior glenoid rim deficiency in recurrent (atraumatic) posterior shoulder instability.* Skeletal radiology, 2000. **29**(4): p. 204-210.

56. Matsen III, F.A., et al., *Glenoid component failure in total shoulder arthroplasty.* JBJS, 2008. **90**(4): p. 885-896.

57. Radnay, C.S., et al., *Total shoulder replacement compared with humeral head replacement for the treatment of primary glenohumeral osteoarthritis: a systematic review.* Journal of shoulder and elbow surgery, 2007. **16**(4): p. 396-402.

58. Wang, W., Y. Ouyang, and C.K. Poh, *Orthopaedic implant technology: biomaterials from past to future.* Annals of the Academy of Medicine-Singapore, 2011. **40**(5): p. 237.

59. Saltzman, M.D., et al., *Shoulder hemiarthroplasty with concentric glenoid reaming in patients 55 years old or less.* Journal of shoulder and elbow surgery, 2011. **20**(4): p. 609-615.

60. Bohsali, K.I., M.A. Wirth, and C.A. Rockwood Jr, *Complications of total shoulder arthroplasty.* JBJS, 2006. **88**(10): p. 2279-2292.

61. Nuss, K.M. and B. von Rechenberg, *Biocompatibility issues with modern implants in bone-a review for clinical orthopedics.* The open orthopaedics journal, 2008. **2**: p. 66.

62. Williams Jr, G.R., et al., *The effect of articular malposition after total shoulder arthroplasty on glenohumeral translations, range of motion, and subacromial impingement.* Journal of shoulder and elbow surgery, 2001. **10**(5): p. 399-409.

63. Szabo, I. and G. Walch, *Factors affecting cemented glenoid component loosening in total shoulder arthroplasty.* International Journal of Shoulder Surgery, 2007. **1**(1): p. 23.

64. Walch, G., et al., *Patterns of loosening of polyethylene keeled glenoid components after shoulder arthroplasty for primary osteoarthritis: results of a multicenter study with more than five years of follow-up.* JBJS, 2012. **94**(2): p. 145-150.

65. Brewer, B.J., R. Wubben, and G. Carrera, *Excessive retroversion of the glenoid cavity. A cause of non-traumatic posterior instability of the shoulder.* JBJS, 1986. **68**(5): p. 724-731.

66. Pritchett, J.W., *Shoulder Resurfacing.*

67. Torchia, M.E., R.H. Cofield, and C.R. Settergren, *Total shoulder arthroplasty with the Neer prosthesis: long-term results.* Journal of Shoulder and Elbow Surgery, 1997. **6**(6): p. 495-505.

68. Nguyen, D., et al., *Improved accuracy of computer assisted glenoid implantation in total shoulder arthroplasty: an in-vitro randomized controlled trial.* Journal of shoulder and elbow surgery, 2009. **18**(6): p. 907-914.

69. Lewis, G.S. and A.D. Armstrong, *Glenoid spherical orientation and version.* Journal of shoulder and elbow surgery, 2011. **20**(1): p. 3-11.

70. Chin, P.Y., et al., *Complications of total shoulder arthroplasty: are they fewer or different?* Journal of shoulder and elbow surgery, 2006. **15**(1): p. 19-22.

71. Wirth, M.A. and C.A. Rockwood Jr, *Complications of total shoulder-replacement arthroplasty.* JBJS, 1996. **78**(4): p. 603-616.

72. Danda, A., M.A. Kuttolamadom, and B.L. Tai, *A mechanistic force model for simulating haptics of hand-held bone burring operations.* Medical engineering & physics, 2017. **49**: p. 7-13.

73. Forsslund, J., E.-L. Sallnas, and K.-J. Palmerius. *A user-centered designed FOSS implementation of bone surgery simulations*. in *EuroHaptics conference, 2009 and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems. World Haptics 2009. Third Joint*. 2009. IEEE.

74. Arbabtafti, M., et al., *Physics-based haptic simulation of bone machining.* IEEE Transactions on Haptics, 2011. **4**(1): p. 39-50.

75. Zheng, F., et al., *An analytical drilling force model and GPU-accelerated haptics-based simulation framework of the pilot drilling procedure for micro-implants surgery training.* Computer methods and programs in biomedicine, 2012. **108**(3): p. 1170-1184.

76. Pourkand, A., N. Zamani, and D. Grow, *Mechanical model of orthopaedic drilling for augmented-haptics-based training.* Computers in biology and medicine, 2017. **89**: p. 256-263.

77. Ghasemloonia, A., et al., *Evaluation of haptic interfaces for simulation of drill vibration in virtual temporal bone surgery.* Computers in biology and medicine, 2016. **78**: p. 9-17.

78. Petersik, A., et al., *Realistic haptic interaction in volume sculpting for surgery simulation*, in *Surgery Simulation and Soft Tissue Modeling*. 2003, Springer. p. 194-202.

79. Agus, M., et al., *Real-time haptic and visual simulation of bone dissection.* Presence: Teleoperators & Virtual Environments, 2003. **12**(1): p. 110-122.

80. Wang, Q., et al., *Real-time mandibular angle reduction surgical simulation with haptic rendering*. 2012, Chinese University of Hong Kong.

81. Morris, D., et al., *Visuohaptic simulation of bone surgery for training and evaluation.* IEEE Computer Graphics and Applications, 2006. **26**(6).

82. Sagardia, M., et al. *Improvements of the voxmap-pointshell algorithm-fast generation of haptic data-structures*. in *53rd IWK-Internationales Wissenschaftliches Kolloquium, Ilmenau, Germany*. 2008.

83. Engine, U., *"What is Unreal Engine 4.* Unity3D Engine, 2016.

84. Wu, J., et al. *Voxel-based interactive haptic simulation of dental drilling*. in *ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. 2009. American Society of Mechanical Engineers.

85. Renz, M., et al. *Stable haptic interaction with virtual environments using an adapted voxmap-pointshell algorithm*. in *In Proc. Eurohaptics*. 2001. Citeseer.

86. Ramezanzadehkoldeh, M. and B.H. Skallerud, *MicroCT-based finite element models as a tool for virtual testing of cortical bone.* Medical engineering & physics, 2017. **46**: p. 12-20.

87. Mor, A., S. Gibson, and J. Samosky. *Interacting with 3-dimensional medical data: Haptic feedback for surgical simulation*. in *Proceedings of phantom user group workshop*. 1996. Citeseer.

88. Tsai, M.-D. and M.-S. Hsieh, *Accurate visual and haptic burring surgery simulation based on a volumetric model.* Journal of X-ray Science and Technology, 2010. **18**(1): p. 69-85.

89. Molino, N., Z. Bao, and R. Fedkiw. *A virtual node algorithm for changing mesh topology during simulation*. in *ACM Transactions on Graphics (TOG)*. 2004. ACM.

90. Seiler, M., et al., *Robust interactive cutting based on an adaptive octree simulation mesh.* The Visual Computer, 2011. **27**(6-8): p. 519-529.

91. Baker, T.J., *Mesh generation: Art or science?* Progress in Aerospace Sciences, 2005. **41**(1): p. 29-63.

92. Torre, J., et al., *WebGL-based Visualization of Voxelized Brain Models.* 2012.

93. Eisemann, E. and X. Décoret. *Single-pass GPU solid voxelization for real-time applications*. in *Proceedings of graphics interface 2008*. 2008. Canadian Information Processing Society.

94. Eisemann, E. and X. Décoret. *Fast scene voxelization and applications*. in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. 2006. ACM.

95. Itkowitz, B., J. Handley, and W. Zhu. *The OpenHaptics/spl trade/toolkit: a library for adding 3D Touch/spl trade/navigation and haptics to graphics applications*. in *First Joint Eurohaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems. World Haptics Conference*. 2005. IEEE.

96. Ruthenbeck, G.S. and K.J. Reynolds, *Virtual reality surgical simulator software development tools.* Journal of Simulation, 2013. **7**(2): p. 101-108.

97. Chan, L.S.-H. and K.-S. Choi. *Integrating PhysX and OpenHaptics: Efficient force feedback generation using physics engine and haptic devices*. in *Pervasive Computing (JCPC), 2009 Joint Conferences on*. 2009. IEEE.

98. Scarpino, M., *OpenCL in action: how to accelerate graphics and computations.* 2011.

99. Pratx, G. and L. Xing, *GPU computing in medical physics: A review.* Medical physics, 2011. **38**(5): p. 2685-2697.

100. Taylor, Z.A., M. Cheng, and S. Ourselin, *High-speed nonlinear finite element analysis for surgical simulation using graphics processing units.* IEEE transactions on medical imaging, 2008. **27**(5): p. 650-663.

101. Fang, S. and H. Chen. *Hardware accelerated voxelisation*. in *Volume Graphics*. 2000. Springer.

102. Dick, C., J. Georgii, and R. Westermann, *A real-time multigrid finite hexahedra method for elasticity simulation using CUDA.* Simulation Modelling Practice and Theory, 2011. **19**(2): p. 801-816.

103. Zheng, F., et al., *Graphic processing units (GPUs)-based haptic simulator for dental implant surgery.* Journal of Computing and Information Science in Engineering, 2013. **13**(4): p. 041005.

104. Courtecuisse, H., et al., *GPU-based real-time soft tissue deformation with cutting and haptic feedback.* Progress in biophysics and molecular biology, 2010. **103**(2-3): p. 159-168.

105. Li, W., et al., *GPU-Based flow simulation with complex boundaries.* GPU Gems, 2003. **2**: p. 747-764.

106. Comas, O., et al. *Efficient nonlinear FEM for soft tissue modelling and its GPU implementation within the open source framework SOFA*. in *International Symposium on Biomedical Simulation*. 2008. Springer.

107. Neylon, J., et al., *A GPU based high-resolution multilevel biomechanical head and neck model for validating deformable image registration.* Medical physics, 2015. **42**(1): p. 232-243.

108. Faieghi, M., O.R. Tutunea-Fatan, and R. Eagleson, *Fast and cross-vendor OpenCL-based implementation for voxelization of triangular mesh models.* Computer-Aided Design and Applications, 2018. **15**(6): p. 852-862.

109. Akenine-Möllser, T., *Fast 3D triangle-box overlap testing.* Journal of graphics tools, 2001. **6**(1): p. 29-33.

110. Wright Jr, R.S., et al., *OpenGL SuperBible: comprehensive tutorial and reference*. 2010: Pearson Education.

111. Shreiner, D. and B.T.K.O.A.W. Group, *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. 2009: Pearson Education.

112. Majercik, A., et al., *A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering.* Journal of Computer Graphics Techniques Vol, 2018. **7**(3).

113. Rubin, G.D., et al., *Perspective volume rendering of CT and MR images: applications for endoscopic imaging.* Radiology, 1996. **199**(2): p. 321-330.

114. Faieghi, M., *Virtual Reality Simulation of Glenoid Reaming Procedure.* 2018.

115. Yau, H.T., L.S. Tsou, and Y.C. Tong, *Adaptive NC simulation for multi-axis solid machining.* Computer-Aided Design and Applications, 2005. **2**(1-4): p. 95-104.

116. Ericson, C., *Real-time collision detection*. 2004: CRC Press.

117. Schulze, T., et al., *Open asset import library (assimp), January 2012.* Computer Software, URL: https://github. com/assimp/assimp.

118. McNeely, W.A., K.D. Puterbaugh, and J.J. Troy, *Voxel-based 6-dof haptic rendering improvements.* 2006.

119. McNeely, W.A., K.D. Puterbaugh, and J.J. Troy. *Six degree-of-freedom haptic rendering using voxel sampling*. in *ACM SIGGRAPH 2005 Courses*. 2005. ACM.

120. Cirne, M.V.M. and H. Pedrini, *Marching cubes technique for volumetric visualization accelerated with graphics processing units.* Journal of the Brazilian Computer Society, 2013. **19**(3): p. 223.

121. Mirtich, B., *Efficient algorithms for two-phase collision detection.* Practical motion planning in robotics: current approaches and future directions, 1997: p. 203-223.

122. Coorg, S. and S. Teller. *Real-time occlusion culling for models with large occluders*. in *Proceedings of the 1997 symposium on Interactive 3D graphics*. 1997. ACM.

Curriculum Vitae

Name:	Vlad Popa
Post-secondary Education and Degrees:	The University of Western Ontario London, Ontario, Canada 2012-2016 Bachelor of Medical Sciences The University of Western Ontario London, Ontario, Canada 2017-2019 Masters in Engineering Science
Honours and Awards:	Province of Ontario Graduate Scholarship 2012
Related Work Experience	Teaching Assistant The University of Western Ontario 2017-2019

Publications:

Vlad Popa, Danielle A. Trecroce, Robert G. McAllister, and Lars Konermann. (2016). *“Collision-Induced Dissociation of Electrosprayed Protein Complexes: An All-Atom Molecular Dynamics Model with Mobile Protons.”* J. Phys. Chem. B. 120(23): 5114–5124.

Haidy Metwally, Robert G. McAllister, Vlad Popa, and Lars Konermann. (2016). *“Mechanism of Protein Supercharging by Sulfolane and m-NBA: Molecular Dynamics Simulations of the Electrospray Process.*” Anal. Chem. 88(10): 5345–5354.

Tilo D. Schachel, Haidy Metwally, Vlad Popa, Lars Konermann. (2016). *“Collision-Induced Dissociation of Electrosprayed NaCl Clusters: Using Molecular Dynamics Simulations to Visualize Reaction Cascades in the Gas Phase.”* J. Am. Soc. Mass Spectrom. 27(11): 1846–1854.

Vlad Popa, Danielle A. Trecroce, Robert G. McAllister, and Lars Konermann. (2016). *"Mobile-Proton MD Simulations for Modeling the Dissociation of Electrosprayed Protein Complexes."* Proceedings of 64TH ASMS Conference on Mass Spectrometry and Allied Topics. 64TH ASMS Conference on Mass Spectrometry and Allied Topics, Conference Date: 2016/6

Lars Konermann, Haidy Metwally, Robert G. McAllister, Vlad Popa. (2018). *"How to Run Molecular Dynamics Simulations On Electrospray Droplets and Gas Phase Proteins: Basic Guidelines and Selected Applications."* Methods. 114: 102–112