8-19-2016 12:00 AM

WebGL-Based Simulation of Bone Removal in Surgical Orthopeadic Procedures

Arezoo Tony
The University of Western Ontario

Supervisor
Dr. Remus Tutunea-Fatan
The University of Western Ontario Joint Supervisor
Dr. Roy Eagleson
The University of Western Ontario

Graduate Program in Electrical and Computer Engineering
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Engineering Science
© Arezoo Tony 2016

Abstract

The effective role of virtual reality simulators in surgical operations has been demonstrated during the last decades. The proposed work has been done to give a perspective of the actual orthopeadic surgeries such as a total shoulder arthroplasty with low incidence and visibility of the operation to the surgeon. The research in this thesis is focused on the design and implementation of a web-based graphical feedback for a total shoulder arthroplasty (TSA) surgery. For portability of the simulation and powerful 3D programming features, WebGL is being applied. To simulate the reaming process of the shoulder bone, multiple steps has been passed to be able to remove the volumetric amount of bone which was touched by the reamer tool. A fast and accurate collision detection algorithm utilizing Möller –Trumbore ray-triangle method was implemented to detect the first collision of the bone and tool mesh objects in order to accelerate the computations for the bone removal process. Once the collision detected, a mesh Boolean operation using CSG method is being invoked to calculate the volumetric amount of bone which is intersected with the tool to be removed. This work involves the user interaction to transform the tool in a Three.js scene for the simulated operation.

Keywords

Total shoulder arthroplasty, WebGL, Three.js, virtual reality, surgical simulators, CSG, Möller –Trumbore, mesh Boolean operation, Collision detection

Acknowledgments

I would like to express the deepest appreciation to my supervisors Dr. Remus Tutunea-Fatan and Dr. Roy Eagleson. During my Masters, they have always provided constructive and invaluable ideas to support and encourage me to progress in my research. I would also like to thank them for dedicating their time for having individual weekly meetings, their great comments and contribution in my research. They always provided constructive feedback in every single step of my work and helped me in various ways to solve the issues. I would like to thank them for their time that spent every single day from the beginning of my research and being a wonderful supervisor all the time supporting me, to provide me a great study atmosphere with great office mates to help and learn together. I consider myself very fortunate to have had the support and friendship of the members from both Dr. Remus Tutunea-Fatan's and Dr. Roy Eagleson's, namely: Jing Jin, Reza Faieghi, Trinette Wright and Kevin Brightwell. At the end, I would like to thank my mom and dad from the bottom of my heart for their love and support all time who dedicated their lives to their children. You are my inspiration and my motivation.

Table of Contents

List of Tables

List of Figures

Chapter 1

1 Introduction

1.1 Overview

This introductory chapter reviews shoulder anatomy and the Total Shoulder Arthroplasty procedure. This chapter also contains the challenges of the Total Shoulder Arthroplasty and the reason a virtual reality simulator is being developed using WebGL which could be a solution to all types of all bone removal procedures.

1.2 Joints and Implants

Joints are the area of the body at which the bones are attached in purpose of body movements. A joint is usually formed by connective tissues and cartilage. Joints are mainly classified structurally which determines how the bones connect to each other and functionally in which the degree of the movement between the bones is being focused. Functionally, joints can be classified into three categories; 1) synarthrosis joints with limited or no mobility, 2) amphiarthrosis joints with slight mobility and 3) diarthrosis joints with freely movements such as shoulder, elbow, hip and knee which are more likely to be injured (Whiting and Rugg 2006).

An orthopedic or prosthesis implant is an artificial medical device for a missing joint or bone replacement to restore function and range of motion in a diseased joint(Wang et al. 2011). However, the implant fixation reduces pain and restores function; it can loosen, wear, and break in place. It is north worthy to mention that correct placement and fixation of the implant (Figure 1-1) into the cavity of the bone is all-important to bring back the functional joint (Nuss and von Rechenberg 2008).

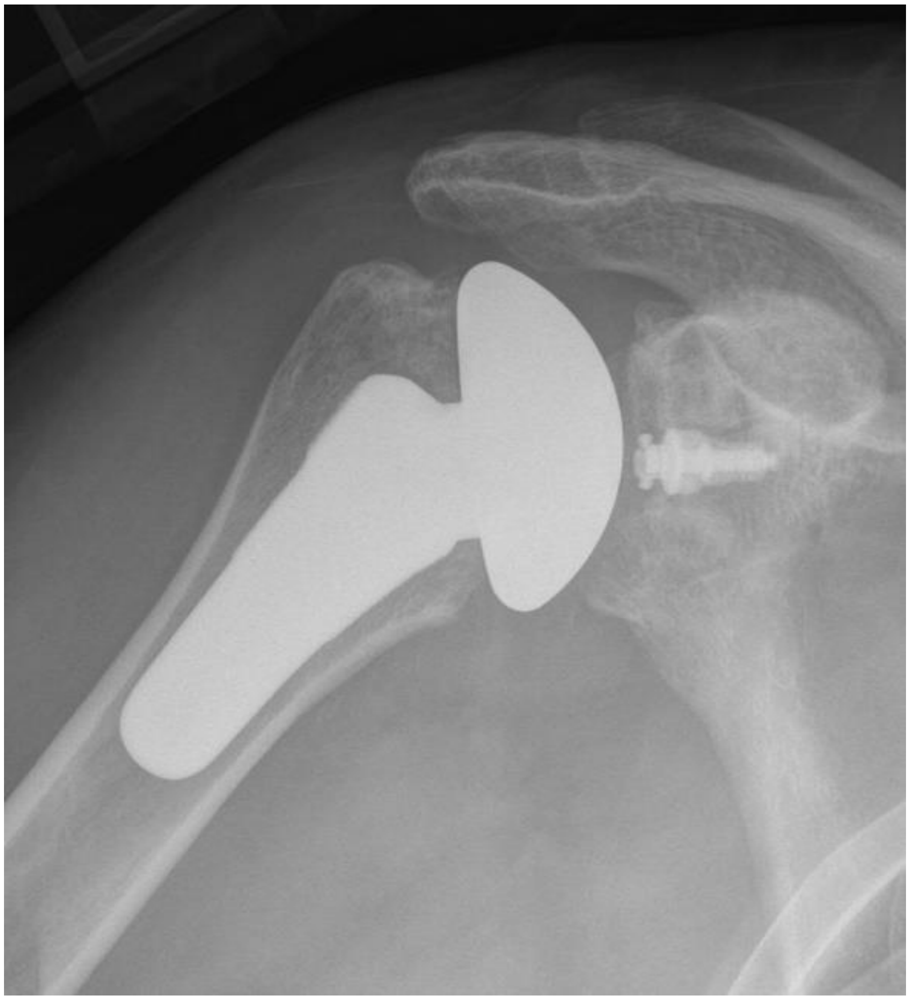


Figure 1-1 Implant fixation in Total Shoulder Arthroplasty

Adapted from [http://www.mayoclinic.org]

1.3 Shoulder

1.3.1 Anatomy

The shoulder is made up of three bones: the upper bone (humerus), the shoulder blade (scapula) and the collarbone (clavicle) as well as muscles, ligaments and tendons (Figure

1-2). The shoulder is a ball and socket joint in which the ball is the humeral head and the socket is called glenoid (Karelse et al. 2014) where the head of the upper arm bone fits into it. The smooth cartilage surface of the shoulder allows the ball and the socket, one to move smoothly along the other and helps to maintain the stability of the shoulder.

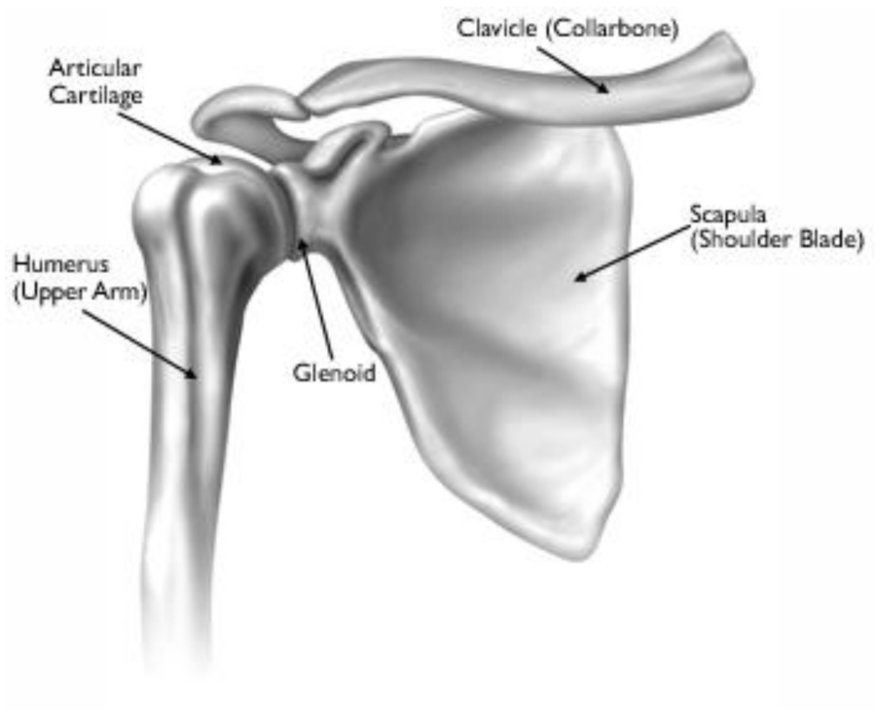


Figure 1-2: The anatomy of the shoulder

1.4 Total Shoulder Replacement Arthroplasty

Total Shoulder Replacement (TSR) also known as Total Shoulder Arthroplasty (TSA), is a well-established and successful procedure which is commonly performed to restore motion, strength, function and eliminate pain in patients with severe glenohumeral arthritis(Karelse et al. 2014). The first total shoulder resurfacing surgery was performed in 1958 (Pritchett 2016).

Shoulder arthritis occurs when the smooth cartilage surface of the shoulder disappears which results in a “bone on bone” joint which is painful. Thus new surfaces can be a robust and reliable solution to restore comfort.

1.4.1 Surgical Techniques

Compared to the previous similar procedure (drilling), reaming has great advantages. Drilling is a procedure in which a hole is being made out of the bone removal using the drill tool in the scapula for the fixation of the implant, while reaming resurfaces the bone. The volumetric amount of the bone removal is not comparable to the reaming procedure which removes the maximum volume of 2mm of surface of the bone to make the smooth congruent surface.

1.4.2 Glenoid Reaming Procedure

Glenoid reaming is a task in Total Shoulder Arthroplasty in which the glenoid cavity is resurfaced in order to provide a congruent, smooth surface for fixation of implant (Saltzman et al. 2011).

When the cartilage surface of the shoulder disappears, the joint becomes stiff which results in a bone on bone joint and reducing the range of motion and limiting activities. To restore comfort, resurfacing the bone (Figure 1-3) would be a solution using reaming procedure.

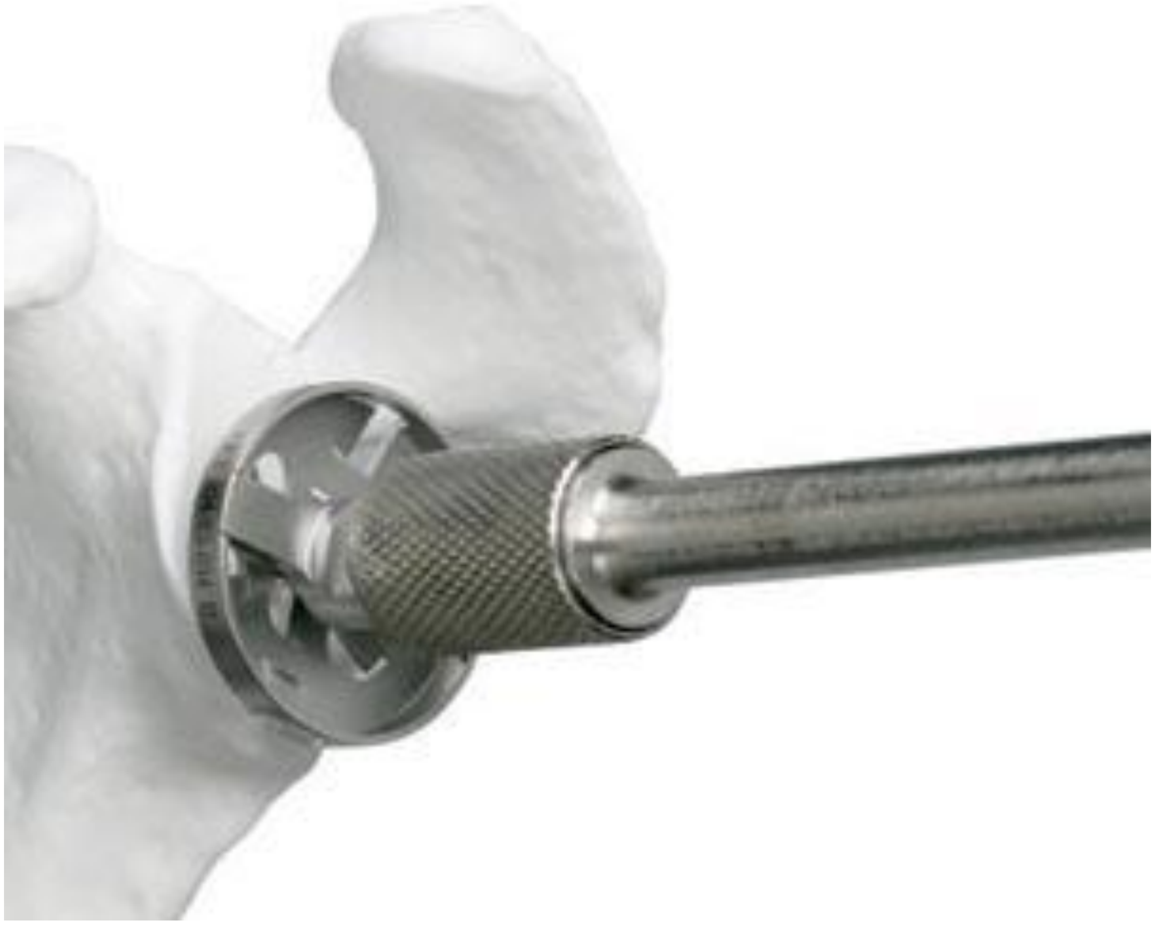


Figure 1-3 Glenoid reaming process

Adapted from [http://faculty.washington.edu]

1.4.3 TSA Challenges

Glenoid replacement is still a challenging and complicated surgery in Total Shoulder Arthroplasty. Resurfacing is a complex task in terms of visualization which is essentially lacking due to the interposed position of the reamer as well as the limited bone volume (Nguyen et al. 2009). Correct positioning and fixation of the implant is difficult depend on the surgeon, the implant, the anatomic variation of the glenoid in patient (Karelse et al. 2014) and improper position and orientation can cause painful glenoid erosion, infection, implant loosening and instability(Aldinger et al. 2010).

1.5 Virtual Reality Simulators for Surgical Training

The effectiveness of virtual reality based training simulators for orthopaedic surgery has been extensively investigated and demonstrated over the last two decades(Vaughan et al. 2016). However, there are always more complications and less control precision in surgical procedures for novice trainees. Additionally, the majority of simulators tend to be focused on lower limb, whereas upper limb procedures are equally or could be regarded as even more challenging due to their lower incidence (Sperling, Cofield, and Rowland 2004; Vaughan et al. 2016) in the population that in turn translates into limited training and practice opportunities. Among upper limb procedures, total shoulder replacement involving glenoid resurfacing (Figure 1-4) pose significant challenges, particularly since cutting tool/bone interface cannot be directly visualized in real-time during the glenoid reaming operation.

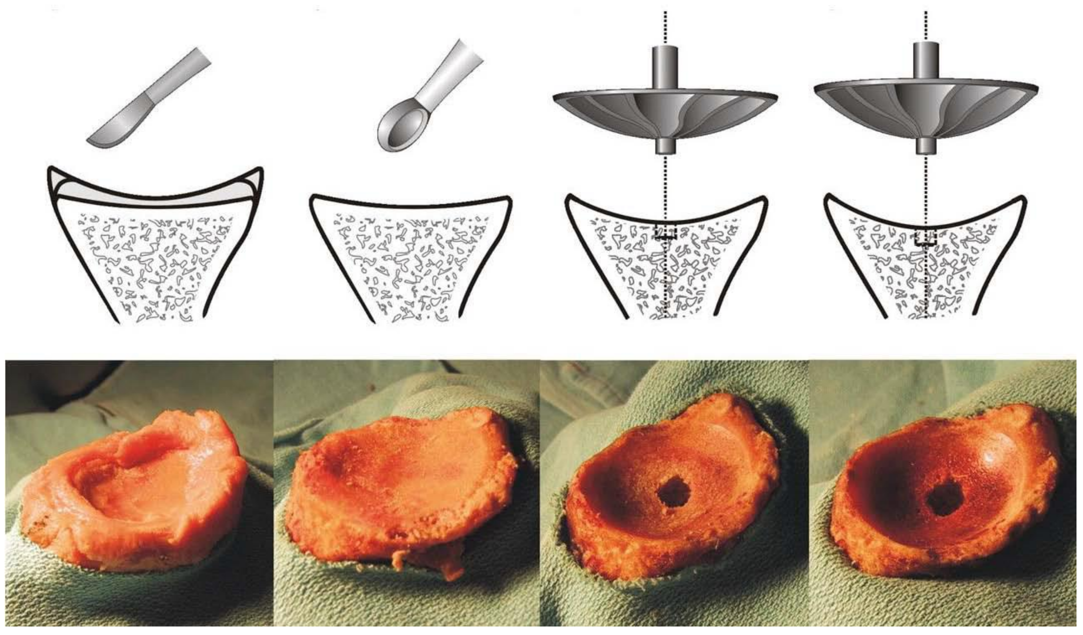
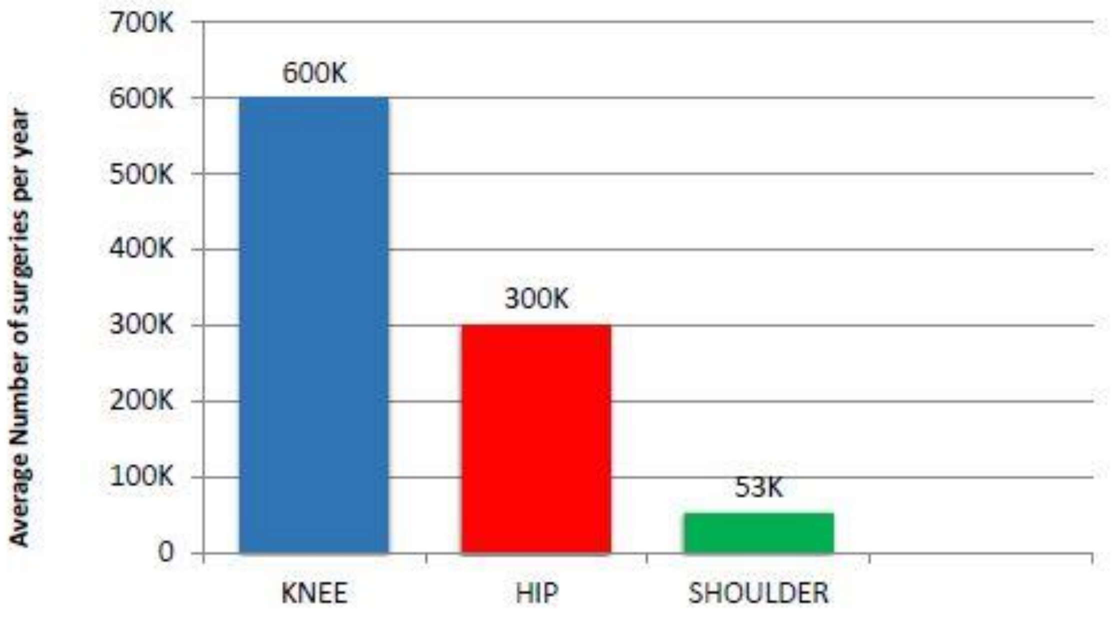


Figure 1-4: Glenoid resurfacing using reamer

Adapted from [http://shoulderarthritis.blogspot.ca/2011_05_01_archive.html]

As can be seen in Table 1-1, the shoulder surgeries happens around 53,000 per year in US which is comparing to the number of more than 90,000 for hip and knee replacement surgeries (“Shoulder Joint Replacement-OrthoInfo - AAOS” 2016). While the scope of a virtual orthopaedic simulator is broad and it includes several modules addressing different types of augmented feedback to guide the actions of the surgical trainee(Figure 1-5), the present study was solely focused on the development of the graphical rendering module. The core of this graphics module – aiming to deliver explicit visual feedback on the location of the bone removed as a function of the reamer posture (i.e., position and orientation) – can be reduced to a conventional Boolean operation between two objects.

Table 1-1 Incidence rate of most common joint surgeries



Surgical training has been performed in the operating rooms (OR) directly before the real surgery and under supervision of senior surgeons.

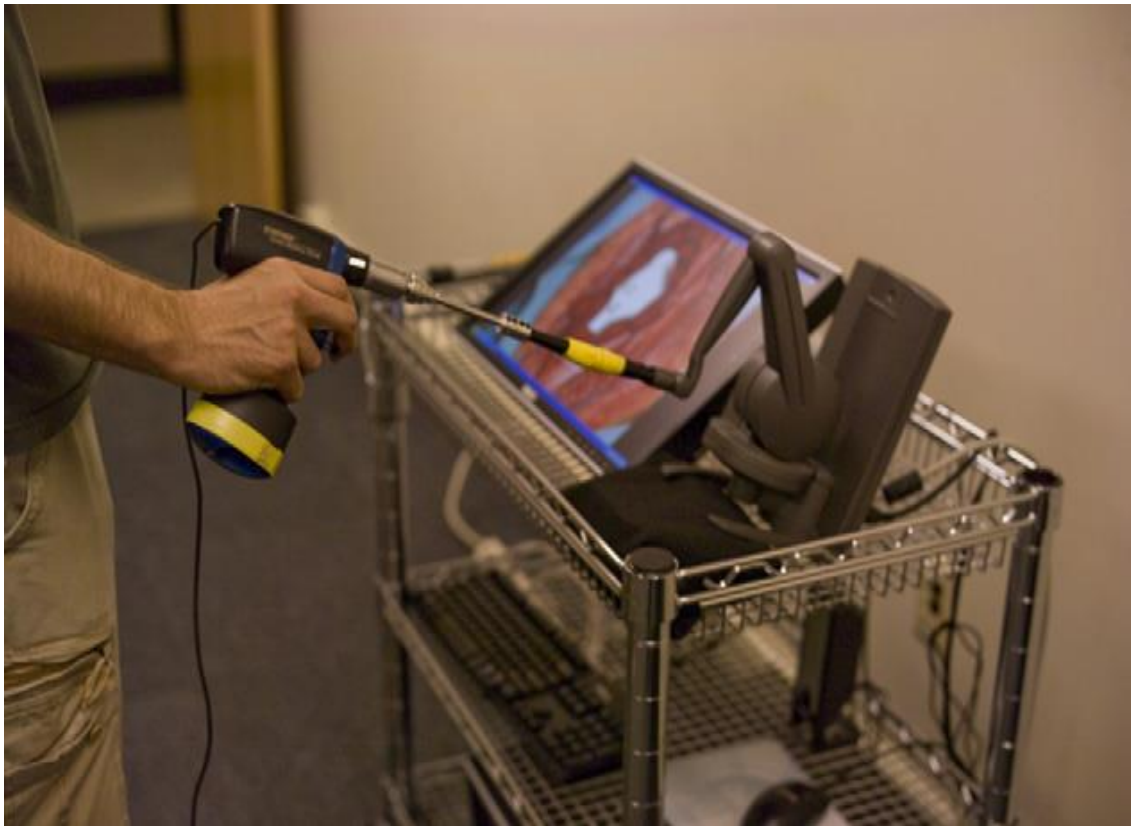


Figure 1-5 Orthopedic haptic drilling simulator by Vankipuram et al

1.5.1 WebGL: Interactive 3D Computer Graphics API

This work constitutes the design of the Graphical Interface using WebGL to provide visual information of the surgical operation and propose the high performance of graphics on the web. WebGL is a cross-platform, standard for 3D graphics Application Program Interface (API) based on OpenGL ES 2.0, integrates to the Document Object Model (DOM) through the HTML5 Canvas element (“WebGL - OpenGL ES 2.0 for the Web” 2016). This standard defines a low level graphic API between software applications and hardware. WebGL is a shader based API using GLSL -The OpenGL Shading Language- which allows the web browsers to have access to the Graphical

Processor Unit (GPU). Besides, previous surveys in Medical Image rendering show the robustness of WebGL in handling large geometries(Cantor-Rivera, Bartha, and Peters 2011).

WebGL provides a real-time interactive 3D web based environment which allows the trainees and surgeons to overcome the classic desktop software tools and access the applications through a desired web browser exclusive of a specific operating system, without extra plug-ins or particular third party components (Chen and Xu 2011).

In this work, a real-time simulator of a tool-bone interaction during glenoid reaming is proposed to demonstrate the use of collision between the two main objects (bone and reamer) as well as the total shoulder resurfacing procedure while the process of reaming in case of more complex geometries to determine the amount of volumetric interference between the two penetrating objects. The simulator is also constituted by a navigation tool for the user interaction to have the better and more precise training performance.

1.5.2 Three.js Library

Three.js was created by Ricardo Cabello, aka Mr.Doob, and has been on gitHub since 2010. Three.js is the most widely-used and powerful, light-weight WebGL library in 3D graphics programming with very low level of complexity. This is a well-designed framework which is fairly easy to learn, understand and use. It is easily possible to create a scene, cameras, objects and use lighting models in Three.js using the built-in functions and settings which results to reduce the amount of work compared to pure WebGL (Danchilla 2012). There are also advanced built-in algorithms and functions in Three.js library which is functional for the current project including import/export various mesh files. Having loaders and exporters in the directory of Three.js allows to respectively import and export types of files containing JSON, OBJ, STL and MTL.

1.6 Rationale

1.6.1 Motivation

The growing evolution of surgical simulator industry has provided a drastic development of even more powerful virtual reality simulators that are capable of real-time interactive environment between the trainees and the various feedbacks more specifically graphical module.

The motivation of this design is to ease the preoperative task required while replacing the shoulder joint with the prosthesis. The virtual reality simulator for the Total Shoulder Arthroplasty (TSA) would help the trainees and the surgeons to have a better tangible perspective of the real surgery. While it is true that there is a growing body of research on the orthopeadic simulators, existing solutions are typically expensive and slow, considering that the simulator is the type of surgical and should be real-time and interactive to achieve the best results out of the training.

Existing research on the virtual reality simulator of the TSA focusing on bone removal is the first time being explored during reaming procedure for training and education purposes.

1.6.2 Objectives and Hypothesis

The final goal of this thesis is to investigate not only the possibility of developing real-time interactive surgical simulators with the new techniques in Total Shoulder Arthroplasty, but also to investigate the effective results of using the simulator to train the residents and surgeons before the real surgery. To accomplish this task, a Web-based graphical interface has been developed of a tool-bone interaction during glenoid reaming process to demonstrate the use of collision detection between the two objects as well as the bone resurfacing in case of more complex geometries to determine the amount of volumetric interface intersected in the bone out and from the reamer tool.

A fundamental purpose of the proposed research is to provide the trainees and surgeons with a realistic, three-dimensional representation of each step of the given procedure. The developed method will permit a precise bone removal procedure with less amount of damage to the glenoid and more accurate replacement of the implant into the bone.

To address this purpose and goal, the objectives of this work are:

1) To develop a web-based graphical interface of the surgery to enable user interaction of the tool while bone removal procedure.

2) To develop a fast algorithm in purpose of finding the first collision between the two intersecting objects.

3) To develop a computer-aided method capable to resurface the bone and regenerate the new mesh out of the bone-tool intersection in each collision.

1.6.3 Contribution

The major contribution of the proposed research is the development of a virtual reality simulator of a TSA. To accomplish this, several algorithms and methods within the scope of this study aimed to represent the information of the tool-bone intersection including the minimum distance, the first intersected points of the collision, the intersection part - the amount of penetration between two objects- and reconstruct the bone once the two objects collide.

This work is the first attempt of WebGL development in the context of simulating the real-time bone removal while reaming procedure in orthopeadic surgeries. By utilizing the developed techniques, surgeons can assess the volumetric amount of bone removed in the graphical module while using the haptic device, vibration and sound feedback. Utilizing WebGL, surgeons are able to run the simulation on portable platform and on a desired web browser either on a computer or a mobile platform exclusive of the hardware and software.

1.7 Structure of the Thesis

This document will discuss about a web-based surgical simulator using WebGL which contains the graphics and the computations in the following chapters.

Chapter 2 outlines the main scenario of the framework including the elements. This chapter also contains the architecture of the framework and complications of the simulator implementation.

Chapter 3 contains the main features of the graphics in the simulator including the main WebGL API and its popular Three.js library, their functions, all included JavaScript libraries, the graphic features such as GUI and utilized to implement a powerful graphics and visual experience of the real procedure in the surgery.

Chapter 4 outlines the main numerical algorithms of the proposed research including all the developed functions and libraries as well as the main procedure of the simulator to resurface the bone and regenerate the mesh object using Möller -Trumbore method to implement the collision detection algorithm in order to find the minimum distance between the two objects and the vertex candidates for the first collision and also the CSG method to implement the mesh Boolean operation between the bone and the surgical tool.

Chapter 5 describes the performance evaluation of the developed surgical simulator and algorithms. It contains the main results of the proposed research using the new techniques and algorithms.

Chapter 6 provides the conclusion and the future related works of the thesis.

Chapter 2

2 Simulator Framework

2.1 Overview

This chapter contains the main scenario of the framework including the elements; trainee who will perform the tasks and respond to the events in the scenario of the implemented simulator and events or actions which the trainee try to perform. This chapter also contains the architecture of the framework and complications of the simulator implementation.

2.2 Framework Architecture

When the user (in this case would be the trainee or the surgeon), tries to perform a task, the input device starts to get involved in the process flow. User can start interacting with the geometrical module through the input deviceto have access to all the information of the objects (such as vertices and faces). Generally, input devices can be from a variety kind of devices including mouse or keyboard. Once the graphical module has been called, the computational module will start invoking the related functions regarding user’s request. Thus, the calculated results through the computational module would be rendered through the rendering module to be displayed on the screen (output device). For each single task and user interaction this process will be executed iteratively. Consequently, the modules of this framework consisting graphics and computations are the main elements of the following work which will be discussed more in details in the next chapters. Besides, the input/output device and the various kinds used in this simulator will be explained in the Graphical Module chapter.

Figure 2-1 illustrates the main diagram of the architecture of the framework related to the proposed work.

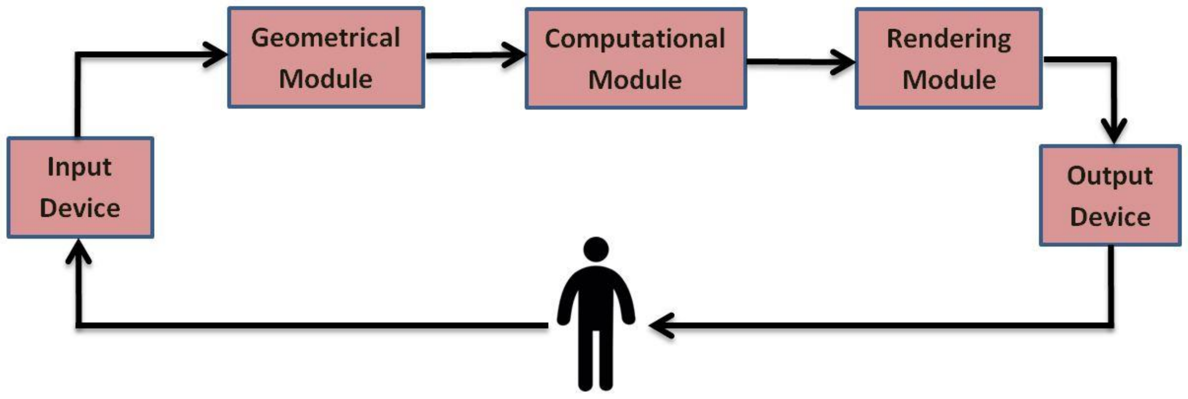


Figure 2-1 The Framework Architecture Diagram

2.2.1 Input device

Input device is a piece of computer hardware that allows the user to interact with the graphics. Input devices can be varied based on the purpose of the software applications including keyboard and mouse for most of the applications and computer games as the basic input devices, Leap Motion sensor device for virtual reality simulators and haptic devices for surgical simulators consisting force computations. Input devices can be also classified based on modality or the number of degree of freedom.

Chapter 3 contains more about input devices and the different types (Figure 2-2) in this work.

Figure 2-2 Different input devices

2.2.2 Graphical Module

The main module of the proposed work is the graphics which includes the main WebGL code containing 3D programming using Three.js. This module provides the classes consisting of user interface menu, imported objects, classes containing the main information of the computational function results, windows settings and more classes which will be explained in both graphical and computational module sections.

The graphical module will be discussed in the next chapter.

2.2.3 Computational Module

The graphics is the interface between the user input and the computational module which contains all the functions and the algorithms implemented in the proposed project.

It provides the implementation of a fast and exact collision detection algorithm between the two objects which calculates the minimum distance between the objects and also find the first vertex candidates in both objects which prone to have the first collision.

The computational module also provides an algorithm and a library in JavaScript which computes the volume of the intersected bone in each step of collision and removes the intersection volume from the geometry. It utilizes the Constructive Solid Geometry method for the use of Boolean operations on the mesh objects. The method, calculations and the library will be discussed in chapter 4.

2.2.4 Output Device: Display

Once the flow passes the computational modules and run the functions and algorithms by the user request received from the input device through the graphical module, it reaches the output device to display the results on the screen. The user then can view the results of own interaction to get a perspective of what is being executed during the reaming process.

The proposed project only contains the graphical module with the related computations while the main project and the final simulator also contains the Audio module, Vibration module and Haptic module which help the surgeon more about the reaming process.

Figure 2-3 illustrates some types of popular output devices in surgical simulators.

Figure 2-3 Output devices in surgical simulators

2.3 Scenario of the Simulator

One of the main goals of the framework is to allow any type of scenario to be executed, for instance any type of implant replacement surgical procedure scenarios including total shoulder/hip/knee arthroplasty or even non-surgical related scenarios such as entering data into a web-based game application.

A scenario in the context of the hierarchical-based framework contains various elements: a user, a sequence of events and a context. The user has the role to respond to the events in the scenario. The events contain all the actions performed by the user (surgeon). And the context explains the environment in which the scenario is being taken.

Events can contain the main actions the user can have during the surgical simulation. The main program has different types of events for translating the objects in 3D, rotating the object around an arbitrary axis, panning the whole scene in the HTML canvas and resizing the web browser based on the platform.

2.3.1 Users in the Scenario

In the described scenario, user is the trainee or the surgeon who will do the experiment to evaluate the performance of the surgery. A user participates in the flow of the events to interact with the graphical module in 3-dimensional environment.

2.4 Development Environment

The implemented web application is for use on any web browsers and operating systems. The programming language used for this application is JavaScript; a high-level, dynamic script language. Alongside HTML and CSS, JavaScript is one of the three core technologies of World Wide Web (www) supported by all modern browsers without the use of any plug-ins. JavaScript also supports the object-oriented concepts and functional programming style which is critical for the good design of large software projects as well as containing various libraries, platforms and technologies that allows the program using the specific powerful features of it. JavaScript also allows creating a professional design of Graphical User Interface for the Virtual Reality based simulators.

2.4.1 Modular design

The structure of the proposed system is designed modular which is subdivided into smaller reusable modules and functions to reduce the cost as well as increasing the flexibility of the system. Similar functions are grouped in one unit of programming section and separate functions are developed in separate sections so that the code can be reused multiple times and by other applications.

Object-oriented programming in JavaScript is compatible with the modular design and programming concept in order to be used in larger scales. Modular programming also enables multiple programmers to divide the whole project into smaller sections and be able to program and debug pieces of the code independently and shortly and the scope of variables can be easily controlled.

In the proposed project, modular programming in JavaScript allows the main project to be implemented into different modules and classes to reuse the same functionality in different part of the program. For instance, collision detection algorithm should be checked every time the user moves the reamer object along to the bone object to detect the first collision. Furthermore, once the collision happens, the Boolean mesh operation get involved. This class is one of the usable modules which will be invoked as many times as the user continue the process of the bone removal to get the desired, smooth surface of the scapula.

2.4.2 OpenGL Framework

As named by Khoronos, OpenGL (Open Graphics Library) is the most widely adopted 2D and 3D graphics API in industry. OpenGL has the latest and newest graphic hardware features so developers could enable high performance of graphics on the application markets such as gaming, entertainment and virtual reality to achieve hardware accelerated rendering. It has the direct access to the graphical processor unit as well as providing a robust platform such as WebGL.

2.4.2.1 OpenGL Libraries

OpenGL contains associated libraries which provide a simpler way to make use of OpenGL features and functions.

There is an OpenGL utility toolkit called GLUT which allows the OpenGL program for opening a window or managing inputs. GLUT has two versions: first is the free version which is called Freeglut and second is the original version of the GLUT which is not an updated version.

Another useful library is Glew that helps for loading and querying the OpenGL Extensions. Glew is responsible to check the OpenGL Extensions based on the specific platform.

There are a variety of OpenGL libraries regarding the purpose of the program. But the mentioned above libraries are among the most useful ones which are included in almost all the programs exclusive of the purpose.

2.4.2.2 OpenGL Versions

OpenGL is and evolving API with new versions and features released by Khronos Group. The first version, OpenGL 1.0 , was released in 1992(Shreiner et al. 2013). Since then, there are both many versions and libraries of OpenGL for simplifying the development of more complicated video games or medical software tools.

There is a newer version of OpenGL as OpenGL for Embedded SubSystems (OpenGL ES) which is still for 2D and 3D graphics with hardware acceleration feature but designed for embedded systems such as smartphones, tablets and video game consoles, televisions with color screen but limited resources. OpenGL ES was called to be the most widely deployed 3D graphics API in the history (SIGGRAPH 2015).

Several versions of the OpenGL ES have been released. The first, OpenGL ES 1.0 was released in 2003 with some differences compared to the OpenGL. Removing the brackets of OpenGL library calls for begin and end, adding fixed point data type for vertex coordinates, removing several technical drawing modes, eliminating display lists and feedbacks and many other extra features to make it a more lightweight API.

OpenGL ES 1.1 had quite the same features as the previous version except some small changes such as vertex buffer object and multi-texture support.

OpenGL ES 2.0 was released in 2008 which is based on OpenGL 2.0 with some modifications in the rendering pipeline. Rendering features of lighting and transformation are replaced by Shaders in GLSL so it became a shader-based rendering pipeline using vertex and fragment shaders. It can be supported by Android and IOS platforms and various other platforms. Most of all, it can be supported by web browsers using WebGL.

2.4.2.3 OpenGL’s Rendering Pipeline

OpenGL rendering pipeline is the sequence of steps for converting the data into the final OpenGL rendered image.

OpenGL starts with geometric data which contains the vertices and geometric primitives. It goes through a couple of stages such as vertex shader, tessellation and geometry shaders before starting for the rasterization stage. Rasterizer generates for the clipping section.

Vertex Specification:

For the very first stage, OpenGL needs all data to be stored in the buffer mostly with the glBufferData () command.

Next stage is vertex processing which contains three levels: 1) Vertex shader, 2) Tesselation, 3) Geometry shading.

Vertex Shading:

Vertex shader is a piece of program which receives each vertex as an input containing all its attributes, processes the vertex and produces an output vertex.

After buffer initialization, geometric primitives should be rendered with glDrawArrays () command. By drawing, OpenGL transfers data to the server. Vertex is a bundle of data which includes the position information.

This is a 1:1 input to output mapping stage.

Tessellation:

The previous stage vertex output will be this stage input to pull into primitives and being tessellated based on the logic and computing new vertex values in terms of color, position and texture and other vertex attributes. This is an optional stage so if no tessellation, the output from Vertex Shading stage goes directly to the next stage.

Geometry Shading:

A geometry shader is a piece of program which is also optional that governs the processing of primitives. This stage is not a 1:1 input to output stage so it receives a single primitive as an input and may have zero or more outputs.

Vertex Post-Processing:

The last stage output comes as an input for psot-processing stage mostly for Primitive Assembly and Rasterization stages. It contains two steps: 1) Transform Feedback which is for recording data from previous stage into buffer object to keep the post-transform rendering state and resubmit it multiple times. 2) Clipping which is primitive clipping of each vertex to the viewport- the permitted region of window to draw- by its clip space position. Different type of primitives can be clipped differently. For instance, points cannot be clipped. If points are outside the clipping view, they will be discarded. But if they are inside, they will be considered. For the lines, if they are partially outside of the volume, they will be clipped and new vertices will be generated. And for triangles, if they are partially inside the clipping view, new triangles will be generated with vertices on the boundary of the volume.

Primitive Assembly:

In this stage, vertex stream converts into the sequence of base primitives for the rasterization process.

Rasterization:

The updated primitives are sent to the rasterizer to generate fragments. A fragment can be in the buffer to store the position, while another can be rejected and never update its pixel location. The main processing of the fragments happens in the next two stages.

Fragment Shader:

This is the final stage to determine the color and depth value of a fragment although it can be still modified in the per-sample operation if needed.

The main difference between a vertex and a fragment shader is that a the location of a primitive is determined by the vertex shader while the fragment shader will have the information of the color of that fragment on the screen.

Per-Sample Operation:

Per-fragment operation or per-sample operation is the final stage where some modifications outside the fragment shader can be done. There are some tests in this stage such as Z-buffer or depth testing for the visibility of the fragment. There are also more tests including scissor test, stencil test, blending or logical operations.

If a fragment goes through all the tests successfully, it can be written in the frame buffer and update the color of the pixel.

Figure 2-4 illustrates all the described stages of the OpenGL rendering pipeline of an image on the screen (“Rendering Pipeline Overview - OpenGL.org” 2016).

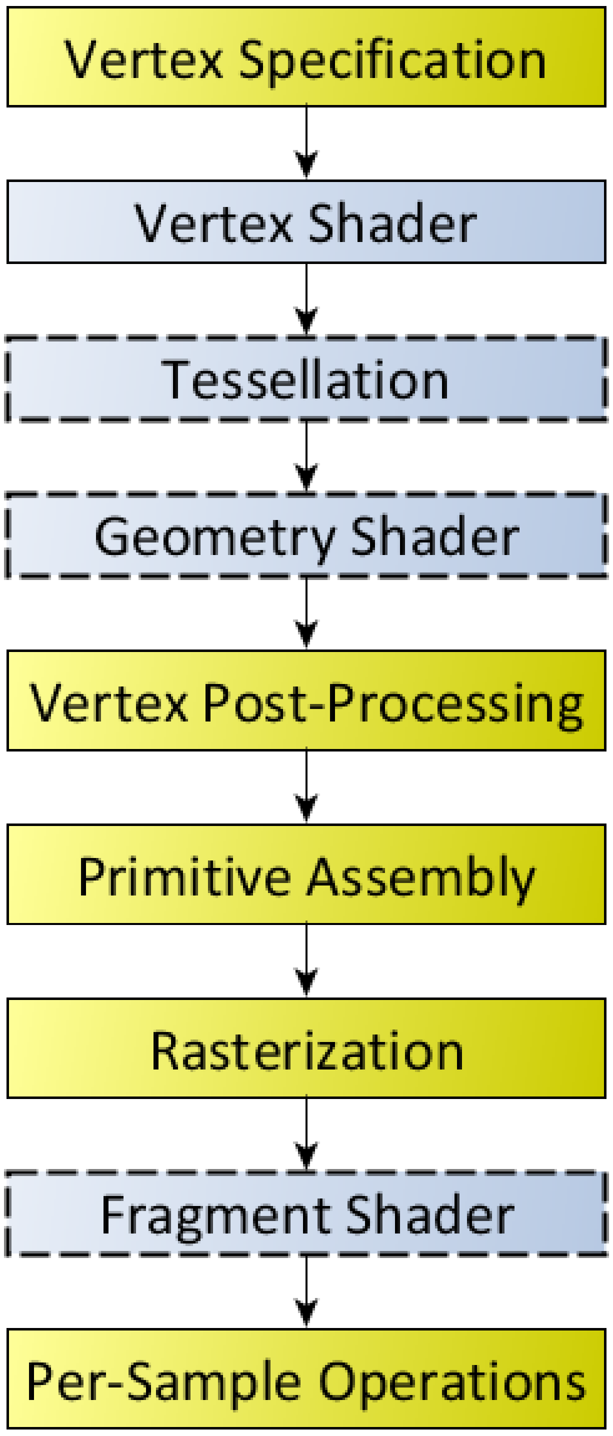


Figure 2-4 OpenGL Rendering Pipeline

2.4.3 WebGL: OpenGL ES 2.0 for the Web

Rendering the simulator view of this work is done using WebGL, a cross-platform, DOM API to create 3D graphics in Web Applications (https://www.khronos.org/webgl) for efficient rendering of 3D medical modeling (Cantor-Rivera, Bartha, and Peters 2011). WebGL is the JavaScript implementation of OpenGL ES 2.0 which is a low level API between software and hardware or software applications though HTML5 canvas element which needs a proficient knowledge of 3D programming and math. WebGL could bring the power of low-level programming of OpenGL to the web for the JavaScript developers (Leung and Salga 2010). What makes WebGL unique is that it allows the web browser to directly access the Graphical Processor Unit (GPU). Thus, surgical virtual reality simulators could make use of 3D web visualization in real-time.

WebGL has characteristics including integration to the Document Object Model through HTML canvas element.

Also, WebGL does not need to communicate with the server once the data and the algorithm reach GPU so it can go offline till the scene needs update.

Plus, WebGL has shown the robustness of handling large geometries in various projects such as the tests done by Cantor-Rivera(Cantor-Rivera, Bartha, and Peters 2011).

In terms of the rendering performance, it can be said that C++ is a compiled language and it could be faster compared to WebGL as a JavaScript API which is an interpreted language.

High-level JavaScript libraries such as Three.js would make it easier and more accessible for the web developers to deal with the 3D math less than the use of pure low-level languages such as GLSL or WebGL.

2.4.3.1 Shaders in WebGL

Shaders get the data and turn it into pixels on the screen. WebGL and OpenGL ES 2.0 are based on shader rendering requiring vertex and fragment shaders. Vertex shader is being processed on each corner of a triangle to transform the points, texture coordinates, get the normal of each triangle to compute the lightings and then pass the output to the fragment shader. Fragment shader runs on each pixel of the transformed triangle passed from vertex shader to receive the color, lighting and texture and output the pixel on the screen. Shaders will be defined in the scripts on top of the program in the HTML Code before the WebGL section.

2.5 Chapter Summary

This chapter included the framework architecture of the proposed work containing all the main elements as well as the OpenGL pipeline. We also included how WebGL works based on OpenGL and the shader programming language to access the GPU.

In the next chapter, the graphical module of the current research will be presented including all the graphical features of the WebGL and Three.js.

Chapter 3

3 Targeting Simulator: Graphical Module

3.1 Overview

The main goal of developing a targeting simulator was to design an environment that allows for simulation of a surgical operation and evaluation of the user performance in interaction with the surgical tools in complex procedures with low incidence and visualization. While such a program could be applied using existing software tools such as SolidWorks or Rhinoceros with predefined features of most of the graphics and computations, implementing a virtual reality simulator are aimed to help for the poor visualization of complicated surgeries such as Total Shoulder Arthroplasty utilizing a powerful graphics and visual experience of the real procedure. Web-based simulation using WebGL technology provides an environment exclusive of specific platform and extra plug-ins.

3.2 Scenario Design

The scenario was first created to evaluate the performance of the surgeons or trainees during the operation. It is designed to allow the surgeons to place the desired objects in the scene and configure the environment of the scene using the UI tools for changing the color of the scene, material of the objects and lighting types. Typically, the virtual reality simulators contain a set of objects by importing mesh files. Objects can be bound to various selectable input devices such as keyboard, mouse, Leap Motion or a haptic device based on the purpose of the program. Figure 3-1 can illustrate the use case diagram of the proposed work. It contains the main elements of the design including their features and configuration attributes such as color, material and position.

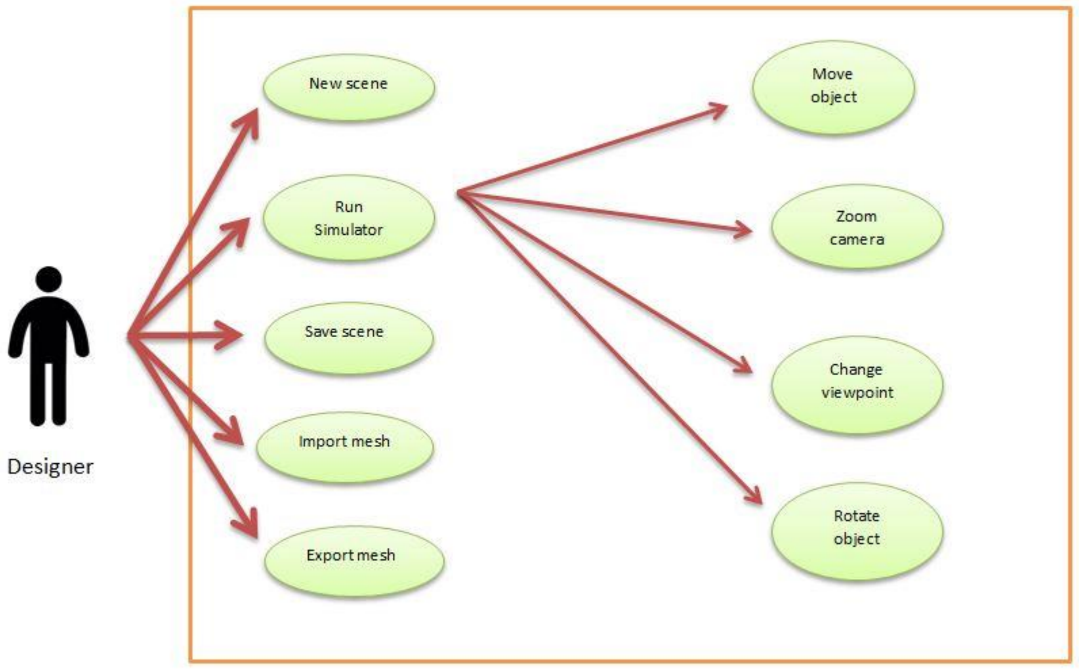


Figure 3-1 Scenario design use case diagram

3.3 Scene and Scene Objects

Firstly, a scene should be created to display the objects and user interface menu. The very first step to use Three.js is to include the Three.js library to the code and enable the Three.js syntax.

To display anything using Three.js, three components are needed including: A scene, a camera and a renderer to render the scene with the camera. Components of the scene also can be categorized into three groups based on functionality: Lights, Camera(s) and Objects. All types are capable of being represented to visualize in the scene of the virtual simulator. Lights, camera, objects and renderer will be discussed in more details in the following sections.

It is easy to set up a new scene, camera and renderer with the following piece of code using Three.js library:

```
var scene = new THREE.Scene();

var camera = new THREE.PerspectiveCamera( 75, window.innerWidth /
window.innerHeight, 0.1, 1000 );

var renderer = new THREE.WebGLRenderer();

renderer.setSize( window.innerWidth, window.innerHeight );

document.body.appendChild( renderer.domElement );
```

3.3.1 Components of the Scene in ThreeJs

3.3.2 Cameras in ThreeJs

There are two main types of cameras in Three.js: 1) camera with the perspective projection and 2) camera with the orthographic projection.

We also have more types of camera including the combined camera of perspective and orthographic or the CubeCamera () which creates 6 different cameras.

3.3.2.1 Orthographic Camera

Orthographic projection renders all the objects of the scene with the same size regardless of their distance to the camera. This is often used in 2D games.

For the Orthographic Camera, we need to define a volume in space and make it move and orient around the scene arbitrarily. The following piece of code will create the rectangular box of that volume.

```
viewSize = 900;

aspectRatio = window.innerWidth / window.innerHeight;

// orthographicCamera (left, right, top, bottom, near, far);

var camera = new THREE.OrthographicCamera( aspectRatio * viewSize
/ - 2, aspectRatio * viewSize / 2, aspectRatio * viewSize / 2,
aspectRatio * viewSize / - 2, -1000, 1000 );

scene.add( camera );
```

The mentioned above code and the Figure 3-2 and Figure 3-3 shows how the Orthographic Camera works.

The first step is to create the camera and define the boundaries of left, right, top and bottom. Near and far properties show the distance of the rendering from the camera. The aspect ratio describes how wide the view is compare to how height it is. Also for the convenience, we can set a view size which is how much vertical space fit in to the view. (*Learning Three.js: The JavaScript 3D Library for WebGL: Create and Animate Stunning 3D Graphics Using the Open Source Three.js JavaScript Library* 2013)

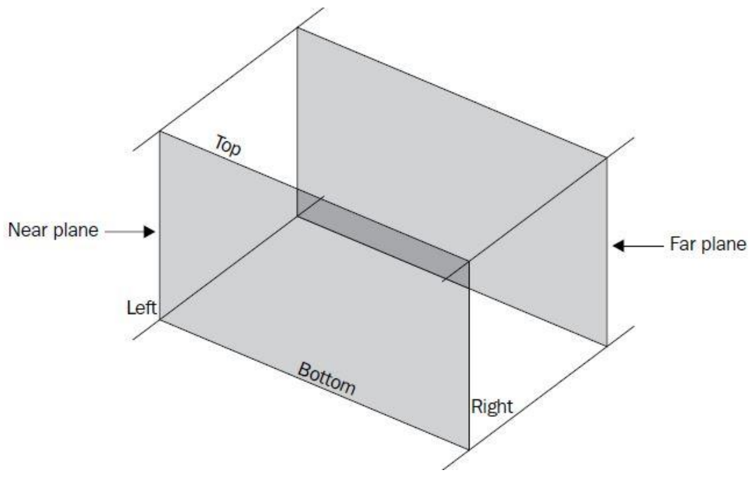


Figure 3-2 Orthographic Camera Properties

The following figure shows the properties of the Orthographic camera and how they can affect the view of the scene. As can be seen, all the properties have the same size regardless of the distance and perspective of the view.

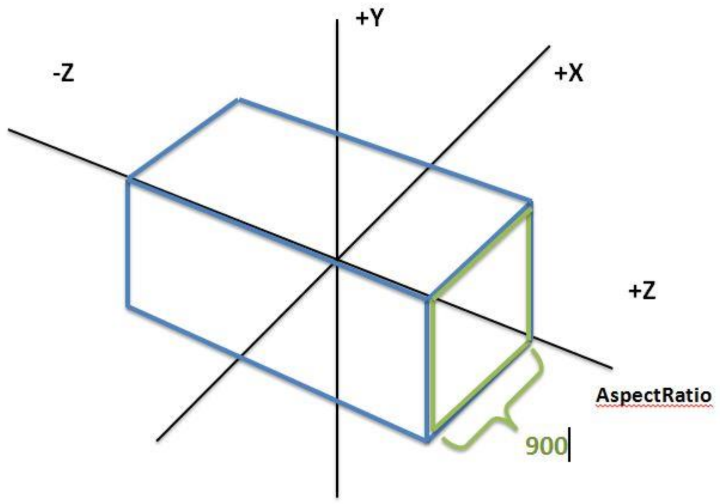


Figure 3-3 Three.js Orthographic Camera

3.3.2.2 Perspective Camera

Perspective camera is like a real world with objects in the distance becoming smaller. Perspective camera works same as the human eye. When the human eye views an object, objects in the distance are smaller which is called perspective. On the other hand, orthographic camera ignores this distance to get the accurate and exact measurement of the objects regardless of the distance.

The following code and Figure 3-4 and Figure 3-5 shows how the perspective camera works. Perspective camera takes fewer arguments than orthographic camera. Fov stands for Field Of View which can be seen from the perspective of the camera. Human's eye has 180 degree fov. For computer games and applications it is mostly between 60 and 90 degree.

Aspect has the same definition as in the Orthographic camera. This is the aspect Ratio between the horizontal and vertical size of the rendered view. Near and far properties also show the rendering distance from the camera.

```
// perspectiveCamera (fov, aspect, near, far);

var camera = new THREE.PerspectiveCamera( 45, width / height, 1,
1000 );

scene.add( camera );
```

Figure 3-5 shows the field of view and the near and far plane from the camera which can be set through the definition of the Perspective Camera in Three.js.

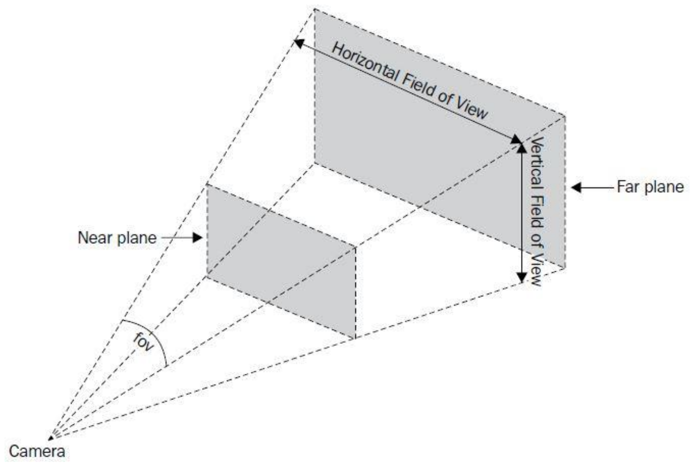


Figure 3-4 Perspective Camera Properties

Figure 3-5 shows the difference of the size of the object in terms of distance to the camera.

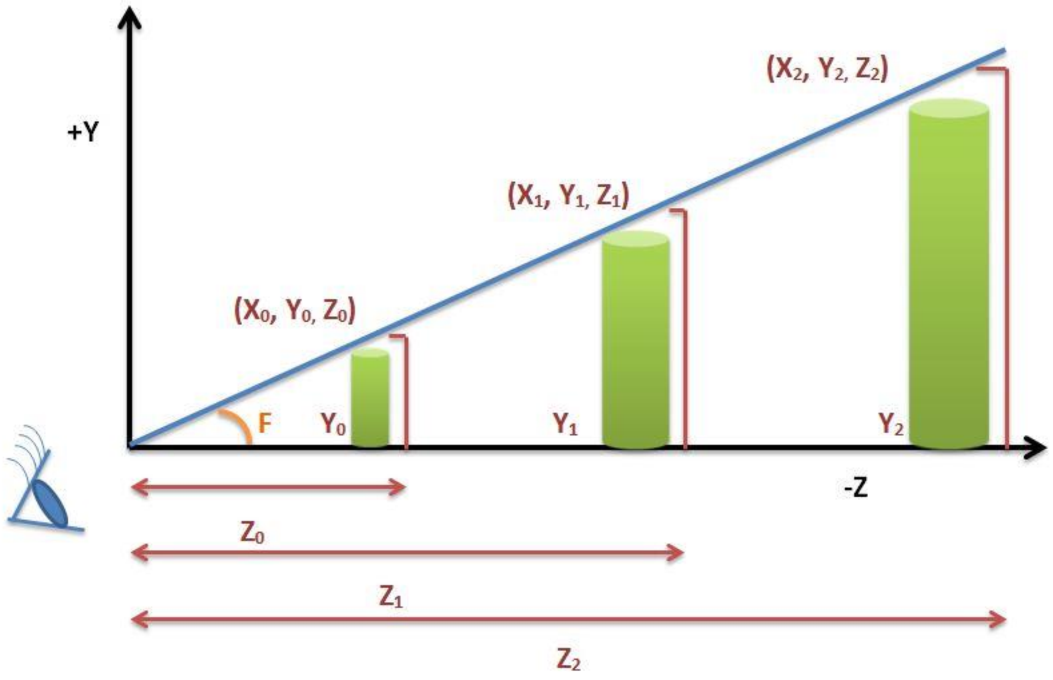


Figure 3-5 Three.js Perspective Camera

In the proposed work, an orthographic projection and camera is being used for more accurate results and view of the objects as well as having the real size of the objects while removing the bone surface in the reaming process. Using perspective camera can increase the level of mistakes and extra bone removal.

3.3.3 Renderers in ThreeJs

Three.js has several renderer classes to render with different targets. It includes a WebGL renderer and a canvas renderer. The canvas renderer translates 3D graphics into the 2D canvas API when WebGL is not available, but it doesn't implement many of the more interesting 3D features, and it is generally much slower than the WebGL renderer. In this module, a WebGLRenderer is being defined using the following piece of code:

THREE.WebGLRenderer is a renderer that renders using WebGL:

```
renderer = new THREE.WebGLRenderer ();

renderer.setClearColor (0xC9CFD1);

renderer.setPixelRatio (window.devicePixelRatio);

renderer.setSize (window.innerWidth, window.innerHeight);
```

The main target of a renderer is to render an image. For that, a camera, scene and an object is required. Also, a render function should render and update the camera, scene and object attributes.

3.3.4 Light Sources in ThreeJs

All the objects in the scene have a material which can be presented with a specific type of light source. Light is an abstract class of Three.js which has two properties to identify the color and intensity of the light. It can be easily created in Three.js with the following piece of code:

```
var light = new THREE.AmbientLight( 0x404040 );

 scene.add( light );
```

Different types of light sources in Three.js exist in terms of usage and behavior. Furthermore, a combination of different type of light sources can be applied. A brief description of each type will be given to demonstrate the utilized types in the proposed work.

AmbientLight: Using this type of light source will give the color to all the objects in the scene. This type could be an appropriate light source while using orthographic camera to have the realistic view and size of the objects in the scene. Figure 3-6 shows the scapula rendering using ambient light.

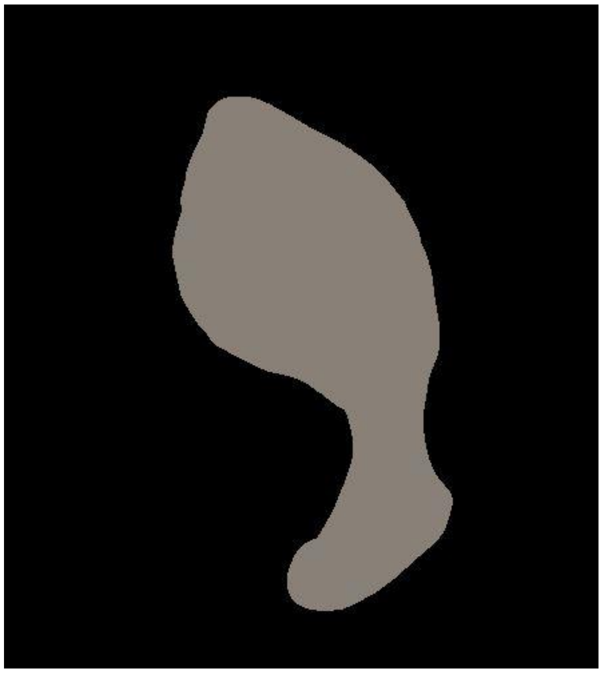


Figure 3-6 Rendering the scapula using ambient light

PointLight: This type of light source is a single point in space which affects objects with lambert and phong material. Figure 3-7 shows the scapula rendering using point light.

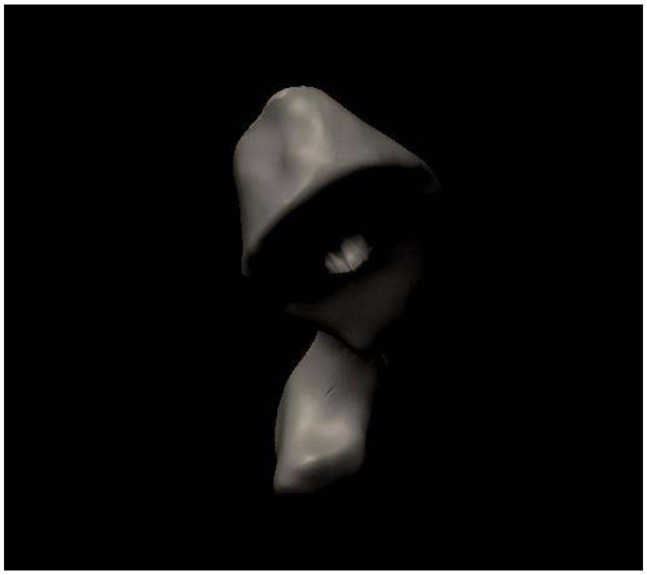


Figure 3-7 Rendering the scapula using point light

SpotLight: This is a point light which could cast a shadow in one direction on the objects with lambert and phong material.

DirectionalLight: This type creates a light from a specific direction and not a position. It behaves as it is infinitely far away from the object like the sun so the ray produced by this would be parallel with the same angle to all the objects. This is the most natural light which can be used for such surgical simulators. Figure 3-8 shows the scapula rendering using directional light.

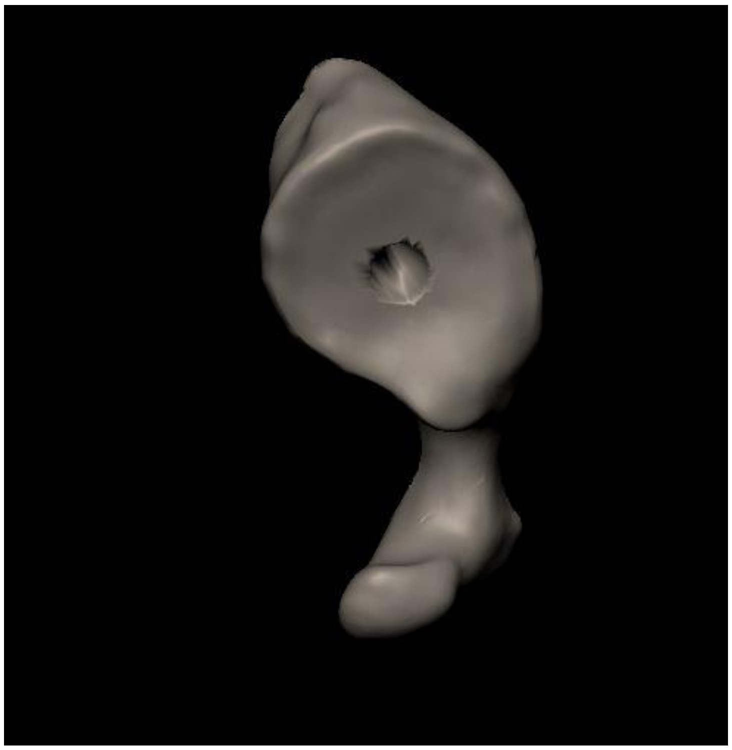


Figure 3-8 Rendering the scapula using directional light

For the proposed research, a combination of directional, ambient and point lights are applied to get a realistic object rendering in the Three.js scene (Figure 3-9).

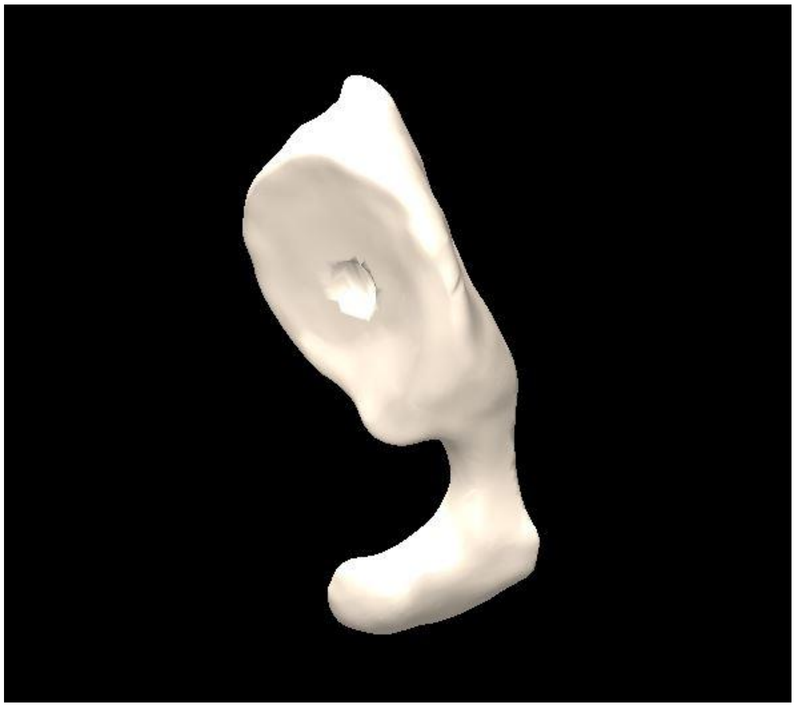


Figure 3-9 A combination of directional, ambient and point light for the scapula rendering

```
//lights

var AmbientLight = new THREE.AmbientLight (0xAB9E91);

AmbientLight.position.set (100,100,400).normalize();

scene.add(AmbientLight);

var Pointlight = new THREE.PointLight(0xAB9E91);

Pointlight.position.set(100,100,400);

scene.add(Pointlight);

var directionalLight = new THREE.DirectionalLight(0xAB9E91, 1);
```

```
directionalLight.position.set(100,100,500).normalize();

camera.add(directionalLight);
```

3.3.5 Mesh Input

3.3.5.1 Rendered Objects

The role of the scene in Three.js is to render the objects (bone and the reamer) containing all the information which is needed to simulate a virtual reality environment for the surgery.

There are two main objects in the proposed Three.js scene: the bone and the reamer. Both objects are the type of polygon mesh (or triangular surface mesh) objects which contain the information of the triangles which make the surface of the object. The main attributes of the mesh files are the triangles (faces), vertices coordinates which make the triangles, the material as well as the rotation, position and scale information of each.

Three.js has its predefined objects and geometries such as a cube or a sphere that can be rendered easily with Three.js syntax.

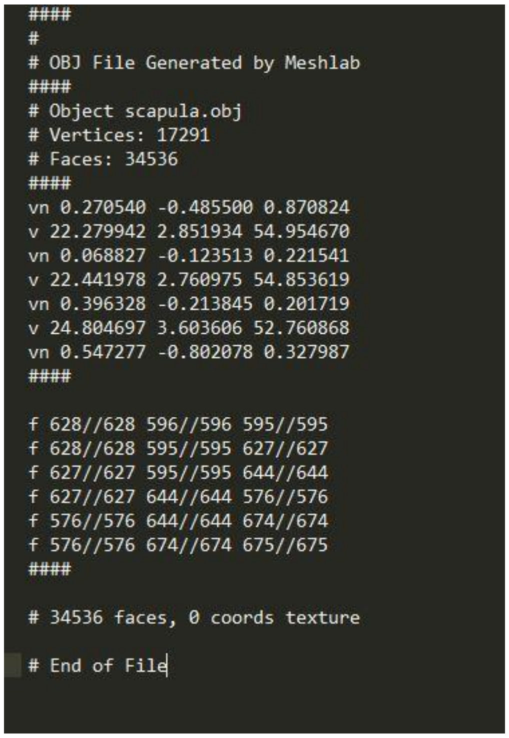
In the proposed research, the triangular mesh representation of the virtual reamer obtained from the reverse engineering methods and the virtual glenoid bone obtained from the CT data thus the bone and the reamer would be rendered in the scene through a type of loader based on type of the object.

3.3.5.2 OBJ File Format and Loader

There are many types of file format to save the information of the mesh objects in Three.js scene. Among all loaders such as OBJ, MTL, JSON and Collada, OBJ loader was chosen for the easy access to the geometry information (Dirksen 2014) which is a need for all the calculations and algorithms discussing in the next chapter.

OBJ is a file format developed by Wavefront Technologies. OBJ format is a simple type which contains almost all the required information such as position of the vertices, UV positions of the textures, faces and vertex normal that is typically modeled using programs such as Blender or Maya or Meshlab.

As can be seen from figure 3-10 every OBJ file contains the main information of the vertex position, vertex normal and at the very bottom of the file the faces which contain the vertex indices of the each face (triangle). Also this figure illustrates that the file was generated through meshlab program.



```
####
#
# OBJ File Generated by Meshlab
####
# Object scapula.obj
# Vertices: 17291
# Faces: 34536
####
vn 0.270540 -0.485500 0.870824
v 22.279942 2.851934 54.954670
vn 0.068827 -0.123513 0.221541
v 22.441978 2.760975 54.853619
vn 0.396328 -0.213845 0.201719
v 24.804697 3.603606 52.760868
vn 0.547277 -0.802078 0.327987
####

f 628//628 596//596 595//595
f 628//628 595//595 627//627
f 627//627 595//595 644//644
f 627//627 644//644 576//576
f 576//576 644//644 674//674
f 576//576 674//674 675//675
####

# 34536 faces, 0 coords texture

# End of File
```

Figure 3-10 A sample of an OBJ file of the bone

Vertex positions are defined with the prefix ***v*** with three components of x, y, z in Cartesian coordinate system.

Vertex Normals represent the normal vector at each vertex with the prefix ***vn*** with the same x, y, z components which may be or may not be normalized that should be considered for the rendering and calculations.

Lastly, faces with the prefix ***f,*** represent the triangles of the mesh. Each face is comprised of three vertices and each one in the obj file contains three vertex indices which made each triangle usually with counter clock wise order. The order of the vertices in each face is important to distinguish the back of the triangle from the front. It will be discussed more in the collision detection algorithm using Möller -Trumber method in chapter 4.

Three.js has OBJLoader to load the .obj file format. The recent versions of OBJLoaders in Three.js load the buffer geometry which contains the general information of the object to increase the memory usage and decrease the cost of passing all the parameters to the GPU but it does not contain the required information for the calculation of the bone removal algorithm. So, in this work, an older version of OBJLoader (version 68) is being utilized for the purpose of loading the geometry directly and to avoid the conversion of buffer geometry to geometry to update the object matrix after each rotation, translation.

Figure 3-11 shows the rendering of scapula bone and virtual reamer representation in Three.js scene.

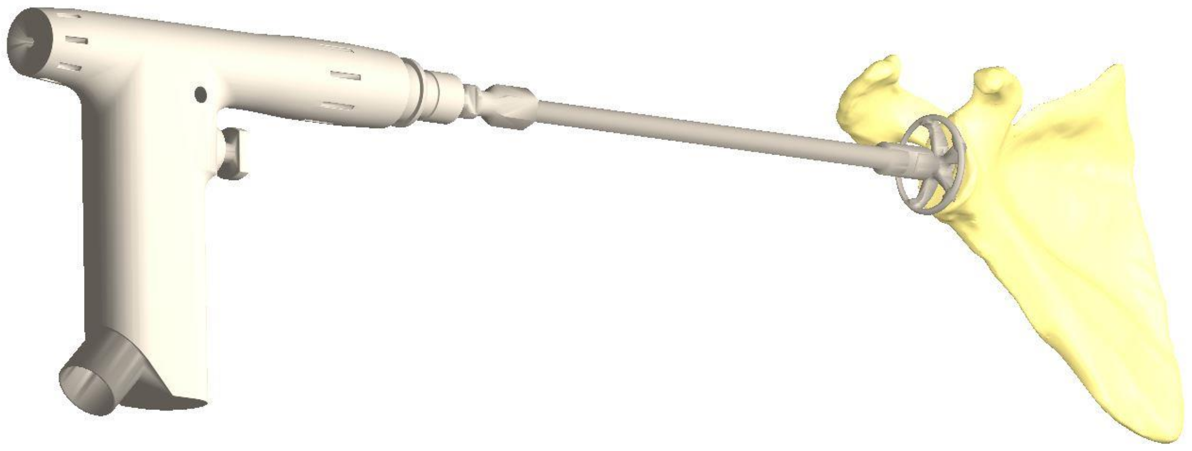


Figure 3-11 Scapula bone and virtual reamer

3.4 Simulation Components

This simulation has a couple of components in order to keep track of the objects and their attributes such as position.

3.4.1 Time Keeper Component

Time keeper is to keep track of time spent on each task in the scenario. It measures the exact amount of time to find the first collision between the objects as well as the run time once each mesh subtraction occurs plus the rendering time of the new generated bone after subtraction in each step. This could verify that all the computations are fast enough to run the program close to real-time.

3.4.2 Parameter Tracking Component

This component is to keep track of the objects and their attributes including the position of the object and each vertex, the orientation of the reamer and other important elements which needs to be computed in the collision detection and Boolean operation algorithms.

3.4.3 Logging Component

Logging component contains the most useful information of the calculations such as the minimum distance between the two objects before the collision occurs, or the rotation axis and angle of the reamer after each user interaction to demonstrate the reamer condition towards the scapula bone.

3.5 User Interaction

In information technology, user interface (UI) is designed to allow the user interact with the system through any device such as a display screen, mouse, keyboard, leap motion in games, haptic device in surgical operations and such devices from the human end.

In the proposed work, the user interacts with the application through mouse and keyboard to achieve the full control over the reamer in order to translate it along the scapula bone axis, to rotate the reamer to the desired orientation while reaming process to get the smooth surface of the bone beside having control over the Three.js scene to pan the whole scene to the left and right sides or to zoom the camera to the desired point for better visualization of the reaming process.

Control over the camera in Three.js could be accomplished using built-in camera controls. The most popular controls are TrackBallControls and OrbitControls.

TrackBallControls is the most used control to utilize the mouse or the trackball in order to easily allow the user to pan, zoom and rotate the camera around the scene. Table 3-1 shows how the TrackBall controls work using mouse.

Table 3-1 TrackBall controls and actions in Three.js

Control	**Action**
Left mouse button and move	Rotate the camera around the scene
Scroll wheel	Zoom in and zoom out
Middle mouse button and move	Zoom in and zoom out
Right mouse button and move	Pan around the scene

The first step is to define the control after the camera definition and update the controls in the render function loop. This type of control works really well with perspective camera.

The functionality of the Trackball control could be modified in terms of speed or enable/disable the elements.

```
//Trackball controls

trackballControls = new THREE.TrackballControls
(camera,renderer.domElement);

controls.rotateSpeed = 0.3;

controls.zoomSpeed = 0.5;

controls.panSpeed = 0.8;

controls.enableZoom = true;

controls.enablePan = true;

controls.enableDamping = true;

controls.dampingFactor = 0.3;

controls.addEventListener('change', render );
```

OrbitControls is another popular type of control which simulates a satellite in orbit around the Three.js scene to pan and rotate around an object in the scene. Table 3-2 illustrates how the orbit control works.

Table 3-2 Orbit controls in Three.js

Control	**Action**
Left mouse click + move	Rotate the camera around the center of the scene
Scroll wheel or Middle mouse click + move	Zoom in and zoom out
Right mouse click + move	Pan around the scene
Left, right, up, and down arrows	Pan around the scene

Same as the TrackBall control, camera should be defined first and then the Orbit control should be declared. Also, we need to update the controls in the render function loop.

```
orbitControls = new THREE.OrbitControls(camera);

//Render function

function render () {

     requestAnimationFrame(animate);

     orbitControls.update(camera);

     renderer.render (scene, camera); }
```

3.5.1 Matrix Transformation

Three.js has its built-in rotation and translation properties which can be used easily to move and rotate the object around the scene. Meanwhile, Three.js is using the matrix transformation behind the scenes for each modification of the object in terms of position and rotation. It is also possible to modify the matrix transformation directly to the geometry or the mesh object.

3.5.1.1 Orientation and Translation

Creating own transformation is simple and straight forward. All required is to instantiate the matrix 4 of the transformation and then apply it either on the mesh or the geometry (Dirksen 2015).

This is how it works using the built-in transformation matrix in Three.js which moves the object 2 steps in the +X, +Y, +Z.

```
THREE.Matrix4().makeTranslation(2,2,2);
```

And this is how to make own transformation matrix in Three.js by instantiating the matrix and then apply it on the object:

```
var translationMatrix =  new THREE.Matrix4 (
      1, 0, 0, controls.x
      0, 1, 0, controls.y
      0, 0, 1, controls.z
      0, 0, 0, 1 );
```

After instantiating the transformation matrix, it could be applied on the mesh:

```
cube.applyMatrix( translationMatrix);
```

Or it could be applied on the geometry. But it needs to update the vertices positions of the geometry manually as it is not updated automatically.

```
cube.geometry.applyMatrix(translatoinMatrix);

cube.geometry.verticesNeedUpdate = true;
```

In this work, the scapula bone is fixed and could not be moved or rotated while the reamer can be transformed in any axis and distance. So, transformation matrix has been applied on the reamer geometry to enable translation and rotation on the mentioned object. The vertices positions need update each time the reamer moves or rotates. This update is applying both in the render function and inside the setKeyControl() function which is responsible to compute the rotation matrix and apply the translation matrix once the user press the related keys on the keyboard.

```
//example of a rotation around X axis with Theta degree

case 65:    // key a to rotate around X by rotationAngle amount
of radian

ReamerGeometry.applyMatrix (new THREE.Matrix4 ().makeTranslation
(-reamerCenter.x, -reamerCenter.y, -reamerCenter.z));

ReamerGeometry.applyMatrix (new THREE.Matrix4 ().makeRotationX
(rotationAngle));

ReamerGeometry.applyMatrix (new THREE.Matrix4 ().makeTranslation
(reamerCenter.x, reamerCenter.y, reamerCenter.z));

// example of a translation to the left

case 37: //left arrow key

ReamerGeometry.applyMatrix (new THREE.Matrix4 ().makeTranslation
(-1, 0, 0));

break;
```

3.5.2 Mouse and Keyboard

To make an advanced interaction in a game or a surgical simulator, various keys on the keyboard might be used in order to control the elements in the scene. The events could be handled through the HTML JavaScript event handler.

In this program, the setKeyControl function handles the events once a key on a keyboard is being pressed.

Arrow keys are utilized to move the reamer along the X and Y axis. Also ***x*** and ***z*** keys are being used to move the object along the Z axis as the simulator is being implemented in 3 dimensional environment.

Furthermore, ***a*** and ***s*** keys enable the reamer to rotate around +X and –X, ***r*** and ***e*** around +Y and –Y as well as ***d*** and ***f*** around +Z and –Z axis respectively by the defined amount of angle.

Mouse movements enables the TrackBallControls as discussed previously to rotate around the scene, pan the scene or zoom in or out to the specified point of the scene.

3.5.3 Leap Motion

The other input tool which is being applied in the next step of the implementation is the Leap Motion which is a sensor device that captures the hand and finger motion (Weichert et al. 2013) to recognize the rotational and angular reamer movements (Motion 2016).

The Leap Motion controller introduces a new gesture and position tracking system with sub-millimeter accuracy compared to the previous sensor input devices. The accuracy and robustness of the tool is still being analyzed.

To use the leap motion in Three.js scene, a new leap control should be replaced to the TrackBallControls. Various controls exist in Three.js for the leap motion to enable different functionalities such as rotation, pan and zoom or add the controls to allow the leap in order to move the object exclusive of keyboard and mouse interactions.

3.5.4 Haptic Device

Haptics is the science of applying tactile sensation to human interaction with computers; a sensation produced by pressure or touch. A haptic device involves physical contact between the computer and the user through an input/output device, such as a joystick or a surgical haptic device, which senses the body's movements and computes the amount of force and pressure in order to transform the object in the scene. By using haptic devices, the user could sends information to the computer as well as receiving other information from the computer in the form of a sensation feeling on some part of the body such as a vibration on the hands while holding the tool or hearing the sound. In a virtual reality environment such as a surgical simulator, a user can translate and rotate the reamer around the scene and along the scapula bone. The computer senses the movement and moves the virtual reamer on the display. However, because of the nature of a haptic interface, the user will feel the reamer movements in his hand through tactile sensations that the computer sends through the surgical haptic device.

3.6 Import and Export Objects

Three.js has various import and export libraries in terms of file type such as obj, blender, collada, vtk based on the purpose of the simulator. In this work, obj file format is being imported to get the required in formation of the vertices, faces and normals as discussed in the OBJ File Format and Loader section in the current chapter. Thus, the same type of exporter is being utilized to get the obj file format of the scapula bone in each step of Boolean mesh operation in order to save all the modified sections of the bone.

3.7 Chapter Summary

This chapter included all the graphics components of the proposed work as well as the WebGL graphical features such as object types and loaders, light sources, renderers, cameras and all the user interface components in the WebGL GUI. The graphics is being updated every time the user interacts with the simulation and passes the computational module to access the algorithms and update the results in the graphical module.

In the next chapter, all computations for the proposed work will be described as well as the implemented algorithms using CSG and Möller -Trumbore methods.

Chapter 4

4 Targeting Simulator: Computational Module

4.1 Overview

This chapter contains the main computations of the proposed work including the main numerical algorithms of all the developed functions and libraries as well as the main procedure of the simulator to resurface the bone and regenerate the mesh object using Möller -Trumbore method to implement the collision detection algorithm in order to find the minimum distance between the two objects and the vertex candidates for the first collision and also the CSG method to implement the mesh Boolean operation between the bone and the virtual reamer. Besides, this chapter outlines some critical concepts and methods in programming and algorithms.

4.2 Divide and Conquer Algorithm

In computer science, divide and conquer (D&C) is a general paradigm for algorithm design. It works recursively by breaking down the main problem into sum-problems of the same types till it becomes easy to solve those sub-problems and then combine to give the solution of the original problem(Brassard and Bratley 1996).

Thus, divide and conquer algorithm divides into three step process(Cormen et al. 2001):

1. To divide the main problem into sub- problems
2. To conquer by solving each sub- problem
3. To combine the solutions and results of all sub- problems

This technique is the basis of efficient algorithms for solving all types of problems including sorting (merge sort), finding the closest pair of point and such big problems that could not be solved easily and straightforward.

Utilizing the divide and conquer technique could be challenging as requires a good understanding of the underlying main problem.

Divide and conquer method could have various advantages(Levitin and Mukherjee 2003) that are listed below:

- Solving tough and difficult problems

This is a very efficient and powerful way of solving tough and big problems by dividing into sub-problems. In some cases, decrease and conquer method also could a solution such as in Tower of Hanoi as being solved by decreasing the height to n-1 instead of n. This also could be applied on the proposed work problem of Boolean mesh operation algorithm to divide into smaller problems which will be discussed soon. Furthermore, the decrease and conquer technique applied on the collision detection algorithm to reduce the number of the candidates prone to have the first collision.

- Parallelism

Divide and conquer method can be adapted for execution on multi-processor machines. Thus, each sub-problem could be executed on a different processor to accelerate the whole execution time.

- Efficiency

The divide and conquer method helps to discover the efficient algorithms and reduce the cost of tough and popular challenges such as merge sort, quick sort, matrix multiplication and fast Fourier transforms(Dasgupta, Papadimitriou, and Vazirani 2016).

- Memory Access

Divide and conquer algorithms make efficient use of memory access because the main problem is divided into small sub-problems that could be solved within the cache instead of the main memory. It also could be designed to use for important algorithms listed above for the optimal cache oblivious algorithms(Frigo et al. 1999).

4.3 Parallelism in JavaScript

Parallel computing is the use of multiple sources simultaneously to solve a computational problem. In this case, using divide and conquer algorithm, the main and principle problem would divide into small discrete parts that could be executed on different processors both on CPU and GPU at the same time. Today, all computers are able for parallel computation from a hardware perspective.

This could be applied for multiple reasons such as saving time and cost, use of concurrency, better use of the hardware features of parallelism which is widely used in science and engineering(“Introduction to Parallel Computing” 2016).

There is a tiny library for multi core-processing in JavaScript called Parallel.js. Although JavaScript is fast enough, it has lack of parallelism due to its single-threated computing model. Parallel.js(“Parallel.js: Parallel Computing with Javascript” 2016) could be a solution with high level access to the multicore processing using web workers.

4.4 Computational Framework

Referred to the framework architecture and diagram in chapter 2, once the input comes out of the graphical module to start the computations, it goes to another set of tasks for the calculation of bone removal process. Figure 4-1 can illustrate the main tasks in this module.

When the two main objects are loaded in Three.js scene, it receives all the information of the vertices and faces from the obj files and starts to check for the collision between the objects. The very fast way to detect the first intersection is to apply the bounding box collision detection algorithm which bounds the two objects around a 3D box and check for the collision of the boxes. As soon as the first collision occurs between the bounding volumes of the bone and the reamer, a fast and exact collision detection algorithm using Möller -Trumbore method is being invoked. Bounding box algorithm would reduce the cost of running Möller -Trumbore algorithm with lots of calculations and multiplications as no collision would happen before the bounding boxes collide. Möller -Trumbore starts running continuously for the first intersection of the reamer and the bone while moving the virtual reamer along to the bone axis in Three.js scene. After the detection of first intersection between the scapula bone and reamer, the Boolean operation algorithm using CSG method would start calculating the volumetric amount of intersected bone with the reamer to be removed and generate a new surface of bone by reconnecting the new vertices of the object.

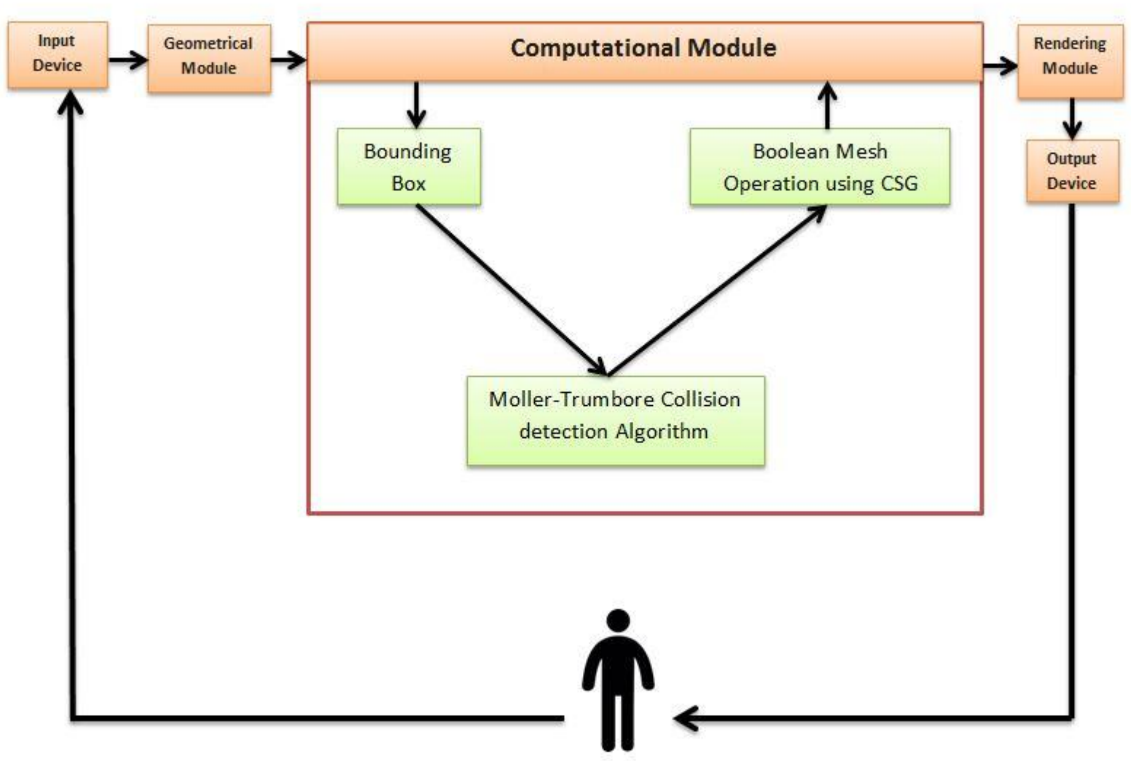


Figure 4-1 Computational module framework diagram

For all the computations of the collision and mesh Boolean subtraction between the two objects, a triangulated form of surface mesh files are being utilized.

Figure 4-2 shows the wireframe material of both the scapula bone and virtual reamer including all the vertices and the faces.

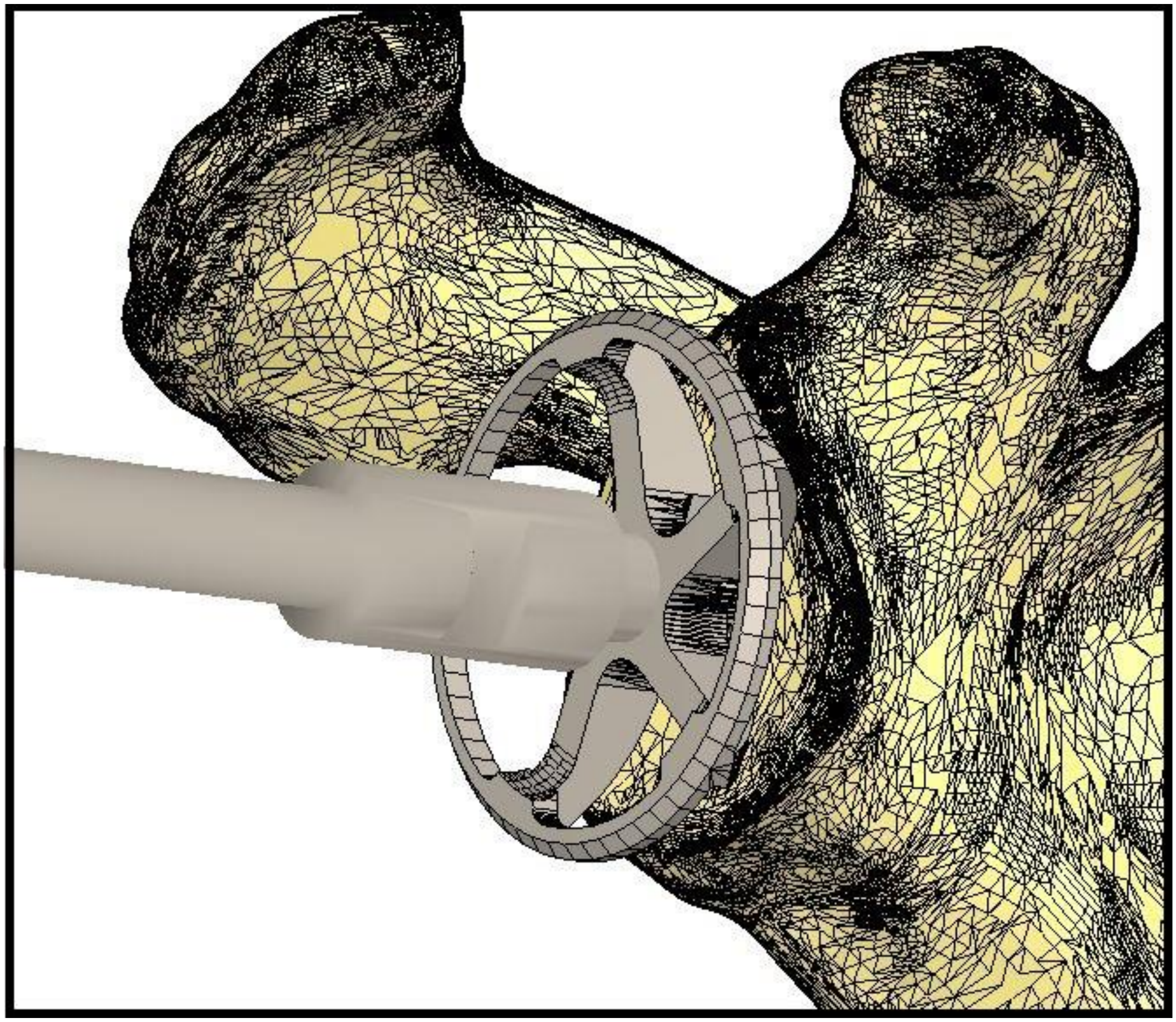


Figure 4-2 Wireframe mesh representation of the bone and reamer

4.5 Collision Detection

Here is an introduction to various algorithms and methods to detect intersection and collision in either 2D or 3D games and simulations depend on the shape and complexity of the objects. The computational cost of collision detection depends on both the complexity of the interference and the number of times the algorithm is being executed (Jiménez, Thomas, and Torras 2001).

Collision detection in generic simple objects is straightforward as having a few numbers of vertices and faces. What is being researched in this work is to detect the intersection in complex triangle mesh objects that can consist of thousands of faces and vertices. There are a majority of algorithms and methods to solve this problem because there is still not a perfect and fast solution which takes care of everything in real-time (Bäckman 2011).

Here is the list of the most common and popular techniques for collision detection:

4.5.1 Axis-Aligned Bounding Box

One of the simple forms of collision detection is between two boxes that are axis aligned and by axis-aligned it means no rotation. The algorithm works by checking the gap between the faces of the boxes in 3D and when the gap exists, no collision detects.

This is the most popular collision detection algorithm as the basis of all complex methods to find the intersection between objects faster.

In this work, bounding box method is utilized to accelerate the computations for the first intersection of the scapula bone and reamer.

Three.js has its predefined bounding box method to find the minimum box bounded around the objects considering the greatest and lowest X, Y and Z in 3D. We update the bounding box of the objects as the bone is being regenerated after each intersection and also the position changes while user is moving the reamer.

```
//boundingBox Helper

helper = new THREE.BoundingBoxHelper(object, 0x000000);

helper.update();

// If you want a visible bounding box

scene.add(helper);
```

Figure 4-3 shows the bounding box of the reamer and scapula using for the first collision detection step.

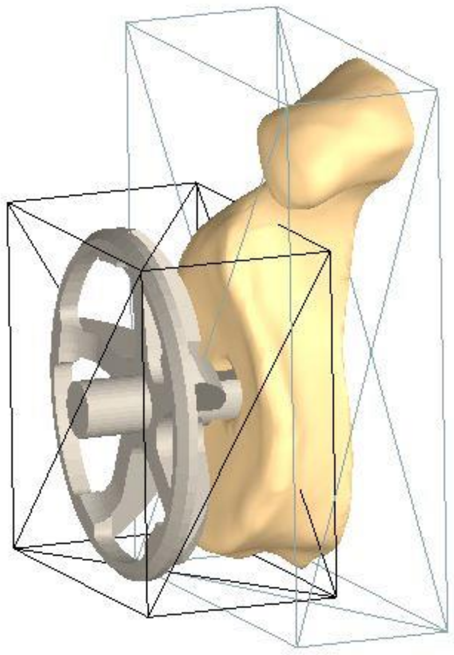


Figure 4-3 Bounding Box representation of the bone and reamer

4.5.2 Separating Axis Theorem

This is a collision algorithm that can detect a collision between any two convex polygons. A shape is considered convex for any line drawn from two points on the shape, it only crosses twice. It's more complicated to implement than the above methods but is more powerful. This theorem could not be practical for the proposed work as objects loaded in this simulator are not convex. SAT states that: “If two convex objects are not penetrating, there exists an axis for which the projection of the objects will not overlap.”

```
// loop over the axes
for (i = 0; i < axes.length; i++) {
  Axis axis = axes[i];
  // project both shapes onto the axis
  Projection p1 = shape1.project(axis);
  Projection p2 = shape2.project(axis);
  // do the projections overlap?
  if (!p1.overlap(p2)) {
    // then we can guarantee that the shapes do not overlap
    return false;
  }
}
```

4.5.3 Circle Collision

A simple shape for collision detection in 2D is between two circles. This algorithm uses the center points of two circles and check for the distance between the centers points to be always less than the addition of both circles’ radius.

```
var dx = circle1.x - circle2.x;

var dy = circle1.y - circle2.y;

var distance = Math.sqrt(dx * dx + dy * dy);

if (distance < circle1.radius + circle2.radius) {

    // collision detected!

}
```

4.6 Collision Performance

Although some collision detection algorithms could be simple and straightforward, it could increase the cost to consider all the possibilities between each two entity. So, games and virtual reality simulations based on collision detection algorithms split in to two phases, broad and narrow.

4.6.1 Broad Phase

Broad phase give the list of entities that could be the candidates to collide which is usually implemented with a spatial data structure to accelerate the process but difficult to handle dynamic scenes. Spatial data structures could be such as Quad Tree, R-Tree, Spatial Hashmap(Luque, Comba, and Freitas 2005) .

4.6.2 Narrow Phase

Narrow phase in collision detection is being applied when there is a small list of entities that could be all checked to provide a certain and exact result whether collision happens or not. Narrow phase can sometimes returns a Boolean value as a result (Mirtich 1997).

4.6.3 Möller -Trumbore Algorithm

The Möller–Trumbore ray-triangle intersection algorithm, named after its inventors Tomas Möller and Ben Trumbore, as a fast method to calculate the intersection of a ray and a triangle in 3D which can be used in computer graphics to implement ray tracing of triangle mesh objects.

4.6.3.1 Algorithm

The Möller-Trumbore algorithm is a fast ray-triangle intersection algorithm which was introduced in 1997 by Tomas Möller and Ben Trumbore in a paper titled "Fast, Minimum Storage Ray/Triangle Intersection". Today, Möller -Trumbore algorithm is being considered as a fast and exact method to find the intersection of two objects. This method uses the parameterization of the intersection point (p) in barycentric coordinate system which needs to only store the vertices of the triangles and does not need any preprocessing (Möller and Trumbore 2005).

Barycentric coordinates are particularly important and practical in computer graphics in the context of triangles. It can express the location of any point inside a triangle with three scalars in a unique consequence of three vertices of that triangle. Thus, any point inside a triangle in barycentric coordinates (Figure 4-4) could be written in the following form (equation 1)

$$P = wA + uB + vC$$

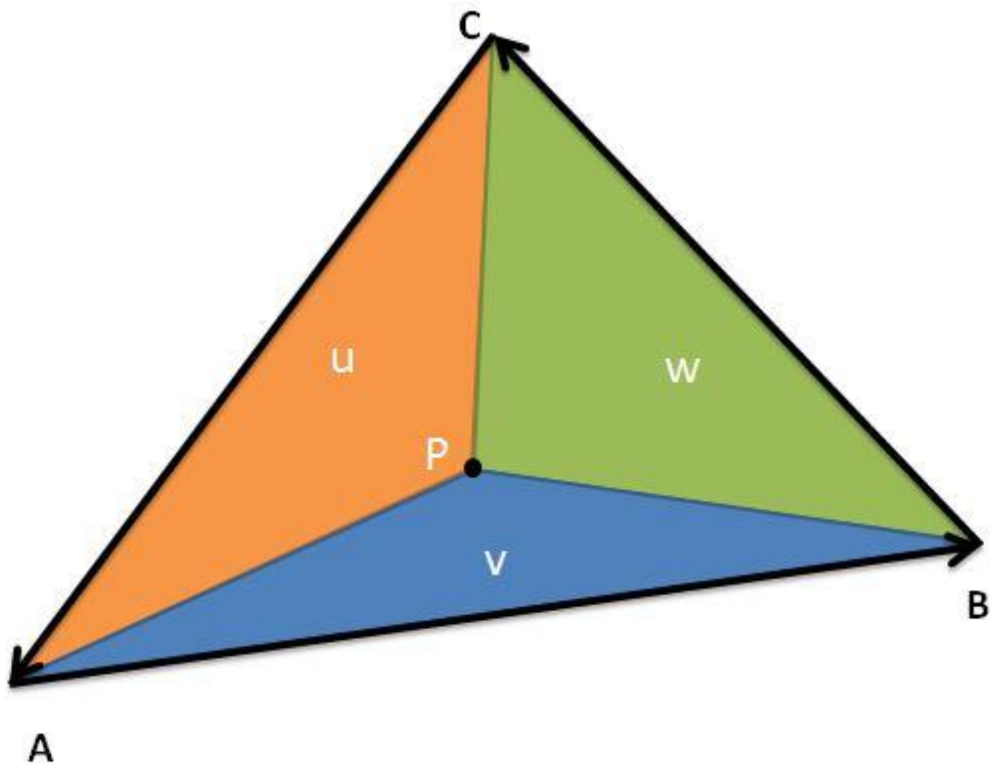


Figure 4-4 Intersection point inside a triangle in barycentric coordinates

A, B and C are the vertices of the triangle and u, v and w are the scalars in barycentric coordinate system that u + v + w = 1 so w = 1 – u – v and we can write:

```
P = (1 − u − v) A + uB + vC
```

If we develop:

```
P= A − uA − vA + uB + vC = A + u(B−A) + v(C−A)
```

(B−A) and (C−A) are the edges AB and AC of the triangle ABC. The intersection ***P*** can be written using the ray's parametric equation (equation 2):

$P = O + tD$

Which ***P*** is the intersection point, ***O*** is the origin and ***D*** is the normalized direction.

In equation 2 ***t*** is the distance from the ray's origin to the intersection ***P***. With replacement of ***P*** in equation 1 in the main parametric equation, we get:

$O + tD = A + u(B-A) + v(C-A)$

$O - A = -tD + u(B-A) + v(C-A)$

On the left side of the equal sign, there are three unknowns (t, u, v) multiplied to three known terms (B−A, C−A, D).

$$\begin{bmatrix} -D & (B-A) & (C-A) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - A$$

The left side rearranged into a row-column vector multiplication which is the simplest form of matrix multiplication to take the first element of the row matrix and multiply by the first element of the column vector.

The position of P could be written in t, u, v space. ***t*** indicates the distance from ***P*** to the ray origin (O) which is parallel to the t axis.

Imagine a point ***P*** in the above triangle ABC. By any type of transformation of the triangle such as translation, rotation or scaling, the position of the point ***P*** would change. What MT algorithm does is to take advantage of creating a new coordinate system (Barycentric Coordinate system) in which any transformation would not affect the position of the point ***P*** in terms of ***u*** and ***v*** instead of ***x, y*** and ***z*** **(Figure 4-5)**.

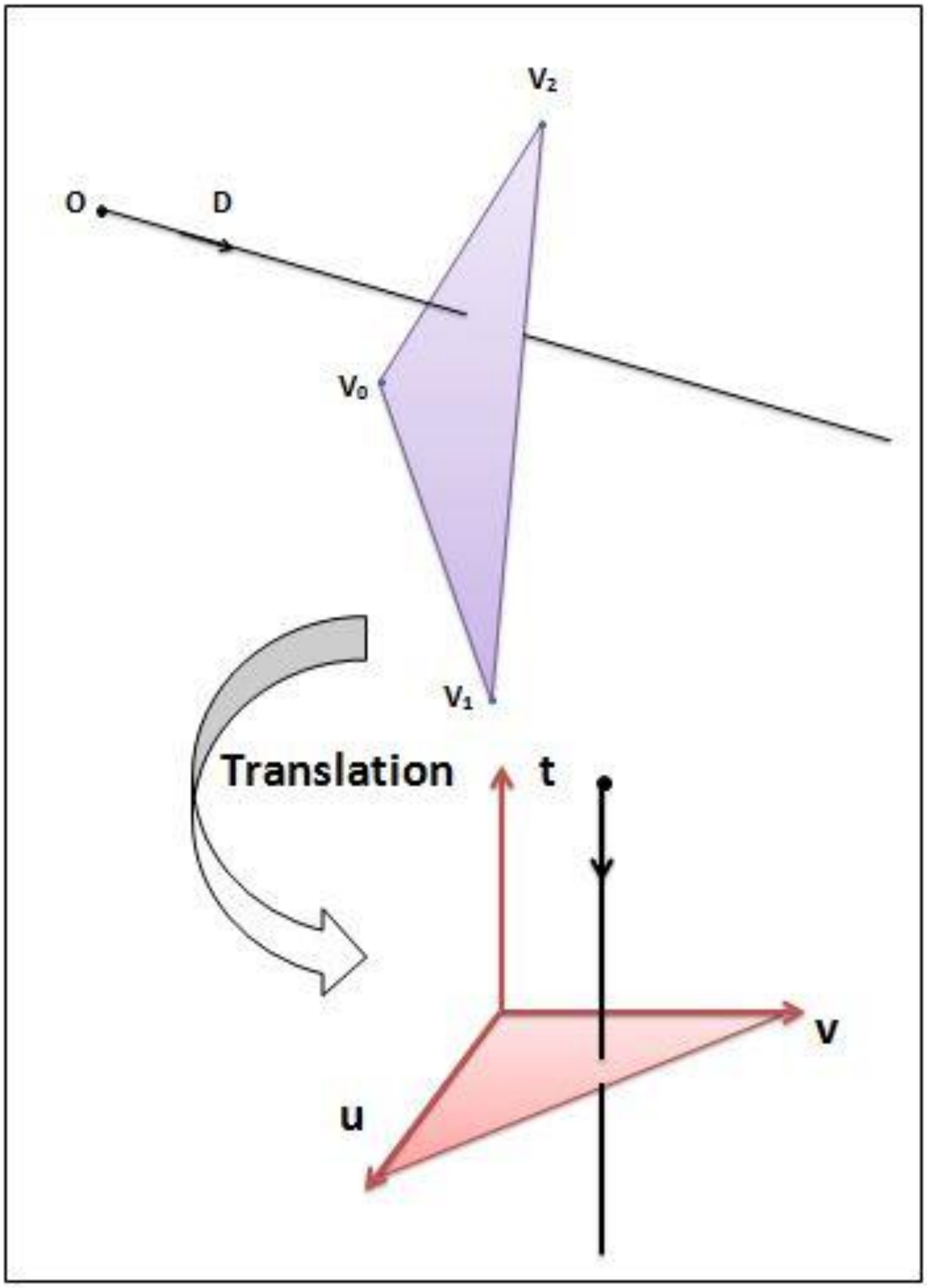


Figure 4-5 Cartesian to barycentric coordinate system conversion

Thus, the three-dimensional ***x, y, z*** position of point ***P*** converts to ***uv*** space inside a unit triangle.

Now there is a coordinate system defined by three axes, ***t, u*** and ***v***. In fact, ***t*** expresses the distance from the ray origin (***O***) to the intersection point (***P***).

The term (O − A) is the transformation of moving the triangle from the original of world space position to the origin. The other side of the equation transforms the intersection point from ***x, y, z*** space to ***tuv*** space.

Using the Cramer's rule to find ***t, u*** and ***v***, equation 5 would be generated:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{[|-D \quad E1 \quad E2\,|]} \begin{bmatrix} |T \quad E1 \quad E2| \\ |-D \quad T \quad E2| \\ |-D \quad E1 \quad T| \end{bmatrix}$$

$|ABC| = -(A \times C) \cdot B = -(C \times B) \cdot A$

The determinant is a scalar triple product of combination of a cross and a dot product.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E2) \cdot E1} \begin{bmatrix} (T \times E1) \cdot E2 \\ (D \times E2) \cdot T \\ (T \times E1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E1} \begin{bmatrix} Q \cdot E2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}$$

Where P = (D×E2) and Q = (T×E1). So, ***t, u*** and ***v*** can computed with cross and dot products between known variables (vertices of the triangle, the origin and the ray direction).

4.6.3.2 Implementation of the MTalgorithm

Implementing the MT algorithm is straightforward. The scalar triple product (AxB).C is the same as A.(BxC). So, the denominator (DxE2).E1 in equation 5 is the same as D.(E1xE2). (E1xE2) calculates the normal of the triangle. If the dot product of the ray direction D and the triangle normal is 0, the triangle and the ray are parallel, therefore there is no intersection. It is also possible to discard back-facing triangles. If the triangle is front-facing the determinant is positive otherwise it is negative. In the code, determinant is computing first. If the result is negative and the triangle is single sided or close to 0, there is no intersection. On the other hand, if the triangle is double sided, the determinant should be checked whether it is close to 0 or not.

```
AB.subVectors(Point2, Point1);

AC.subVectors(Point3, Point1);

var pVector = new THREE.Vector3();

pVector.crossVectors(rayDirection, AC);

var determinant = AB.dot(pVector);

if (! (determinant < 0 && Math.abs (determinant) < 0)) {

     var invDet = 1 / determinant;

     var tVec = new THREE.Vector3();

     tVec.subVectors(rayOrigin, Point1);

     var tempU = tVec.dot(pVector);

     //calculating u

     u = tempU * invDet;
```

For less computations and checking all the possible triangles (in this work, each ray from each vertex of the reamer to each triangle of the scapula bone), we compute u first to reject if it is lower than 0 or greater than 1. If it passed this step, v could be computed with the same tests and also check for the u+v not to be greater than 1. After passing these two steps, there should be an intersection with the triangle to compute t.

```
if (u >= 0 && u <= 1 ) {

      //calculating v

      var qVec = new THREE.Vector3();

      qVec.crossVectors(tVec, AB);

      var tempV = rayDirection.dot(qVec);

      var v = tempV * invDet;


      if (v >= 0 && u + v <= 1) {

            //calculating t

            var tempT = AC.dot(qVec);

            var t = tempT * invDet;

      }
```

When the ray and the normal of the triangle are facing each other so they go in opposite directions and the determinant is positive or greater than 0. On the other hand, when the ray hits the triangle from back so the ray and the normal points are in the same direction and the determinant is negative.

When culling is active, rays intersecting the triangles from back will be discarded. This can easily be checked for, using the sign of the determinant. If culling is active and the determinant is negative, the ray doesn't intersect the triangle.

When culling is off and the determinant is negative, it is important to normalize u by multiplying it by the inverse of the determinant. Because if u is greater than 0 and lower than 1, when u is negative and the determinant is also negative, the sign of u when normalized is inverted and becomes positive.

It is worthwhile to mention that the order of the vertices in each triangle is very important for all the computations based on the vectors created by those vertices. Therefore, the very first step to run the ray-triangle intersection algorithm is to sort the points of the triangles on the scapula bone in an counter clockwise order. For this purpose, the first point (A) would be the one on the face with the maximum Z, the second point (B) would be the vertex with the minimum Y and the left would be the third point (C) of the face in the order of the algorithm.

As the normal vector of each triangle should be in the positive Z axis, the normal vector should be calculated after ordering the points to make sure that they are sorted correctly. If the z coordinate of the calculated normal vector is positive, the order is correct. But, if the z coordinate is negative, it means that the order of points 2 and 3 should be swapped to get the positive normal vector in the z axis.

For this purpose, following piece of code is implemented:

```
if (ScapulaVertices[ScapulaFaces[j].a].z >
ScapulaVertices[ScapulaFaces[j].b].z &&
ScapulaVertices[ScapulaFaces[j].a].z >
ScapulaVertices[ScapulaFaces[j].c].z ) {

     Point1 = ScapulaVertices[ScapulaFaces[j].a];



     if (ScapulaVertices[ScapulaFaces[j].b].y <
     ScapulaVertices[ScapulaFaces[j].c].y){

          Point2 = ScapulaVertices[ScapulaFaces[j].b];

          Point3 = ScapulaVertices[ScapulaFaces[j].c];

          }

     else {

          Point2 = ScapulaVertices[ScapulaFaces[j].c];

          Point3 = ScapulaVertices[ScapulaFaces[j].b];

          }

     }

//b is the first point

if (ScapulaVertices[ScapulaFaces[j].b].z >
ScapulaVertices[ScapulaFaces[j].a].z &&
ScapulaVertices[ScapulaFaces[j].b].z >
ScapulaVertices[ScapulaFaces[j].c].z ) {

     Point1 = ScapulaVertices[ScapulaFaces[j].b];

     if (ScapulaVertices[ScapulaFaces[j].a].y <
     ScapulaVertices[ScapulaFaces[j].c].y){

     Point2 = ScapulaVertices[ScapulaFaces[j].a];
```

```
          Point3 = ScapulaVertices[ScapulaFaces[j].c];

          }



          else {

               Point2 = ScapulaVertices[ScapulaFaces[j].c];

               Point3 = ScapulaVertices[ScapulaFaces[j].a];

               }

          }

     //c is the first point

     if (ScapulaVertices[ScapulaFaces[j].c].z >
     ScapulaVertices[ScapulaFaces[j].a].z &&
     ScapulaVertices[ScapulaFaces[j].c].z >
     ScapulaVertices[ScapulaFaces[j].b].z ) {

          Point1 = ScapulaVertices[ScapulaFaces[j].c];

          if (ScapulaVertices[ScapulaFaces[j].a].y <
          ScapulaVertices[ScapulaFaces[j].b].y){

               Point2 = ScapulaVertices[ScapulaFaces[j].a];

               Point3 = ScapulaVertices[ScapulaFaces[j].b];

          }

          else {

               Point2 = ScapulaVertices[ScapulaFaces[j].b];

               Point3 = ScapulaVertices[ScapulaFaces[j].a];

               }

          }
```

And to check the order of points B and C, the normal vector will be calculated:

```
var normalVector = new THREE.Vector3 ();

normalVector.crossVectors(AB, AC);

//if the normal is negative swap point 2 and 3 to get the
positive normal

if (normalVector.x < 0){

     var temp = new THREE.Vector3();

     temp = Point2;

     Point2 = Point3;

     Point3 = temp;

     AB.subVectors(Point2, Point1);

     AC.subVectors(Point3, Point1);

     normalVector.crossVectors(AB, AC);

} // end of normal calculation
```

4.7 Mesh Boolean Operation

Boolean operations are mathematical operations which has the results true or false. Boolean operations on polygons are the set of the Boolean operations on a set of polygons in computer graphics or CAD.

A Boolean operation is made of three steps:

1. Process mesh 1 with regard to mesh 2
2. Process mesh 2 with regard to mesh 1

3. Combine the results

Boolean mesh operation is a practical method which is included in a wide range of applications such as computer vision, medicine, games, virtual reality simulations and movie special effects.

With all the long history of the mesh operation issues and challenges, Constructive Solid Geometry (CSG) is still a practical choice for most 3D modeling systems (Bernstein and Fussell 2009)

Constructive Solid Geometry (CSG) is a technique to create complex geometries by combining objects and applying Boolean operations on those objects which is mostly used in computer graphics and CAD. CSG method has limitations based on different applications and different purposes.

Every solid object is constructed from primitives such as cubes, spheres, cones and cylinders that allow the Boolean operations on the set of primitives of original objects. Boolean operations consist of union, intersection ad subtraction.

Union of two objects is the collection of all sets of elements of those objects. Intersection of two objects is the sets that contain all the elements of the first object which belongs to the second object and no other elements. And finally, subtraction or difference of one object with respect to another object would be the set of elements of the first object but not in the second object. This is the target Boolean operation of this work which can be applied to the scapula bone each time the reamer touches and intersects the surface of the bone. Figure 4-6 shows the above mentioned Boolean operations.

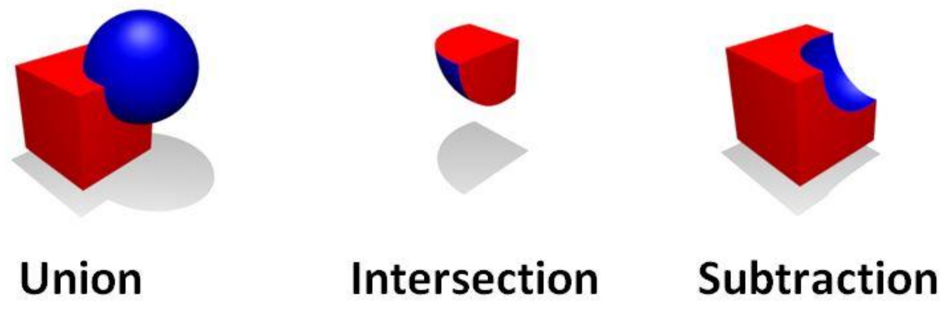


Figure 4-6 Boolean Operations

4.7.1 CSG Method: Constructive Solid Geometry

Constructive Solid Geometry (CSG) or Computational Binary Solid Geometry is a technique to apply Boolean operations on objects which is a well-studied problem in computer graphics and CAD/CAM applications (Gopi and Krishnan 2002).

Multiple techniques exist in CAD packages and computer graphics applications and software (Segura, Stine, and Yang 2013) such as voxel modeling, boundary representation (B-Rep) of the mesh objects but one of the most common-used and functional techniques is CSG that could be modified easily for specified purposes.

CSG has a variety of advantages over the computational geometry methods. The main one could be to perform Boolean operations on simple models and create a complex and accurate object as a result. It could also mention that CSG detects various geometric characteristic of the 3D models such as collision detection which was described in the previous section and current chapter, and water-tightness which means that water cannot leak out of the object if it is filled with water. In other words, water-tight mesh objects have no holes, cracks or missing features of the mesh. Collision detection can be a very expensive computational process and using CSG reduces the cost of the computations by finding the intersection or collision between the objects before applying the Boolean operations.

Another advantage of CSG is the visibility of models relative to each other with a changing viewpoint to render the front objects and ignore the back ones which is facilitated by binary space partitioning algorithm. Figure 4-7 illustrates an example of a CSG tree containing different Boolean operations.

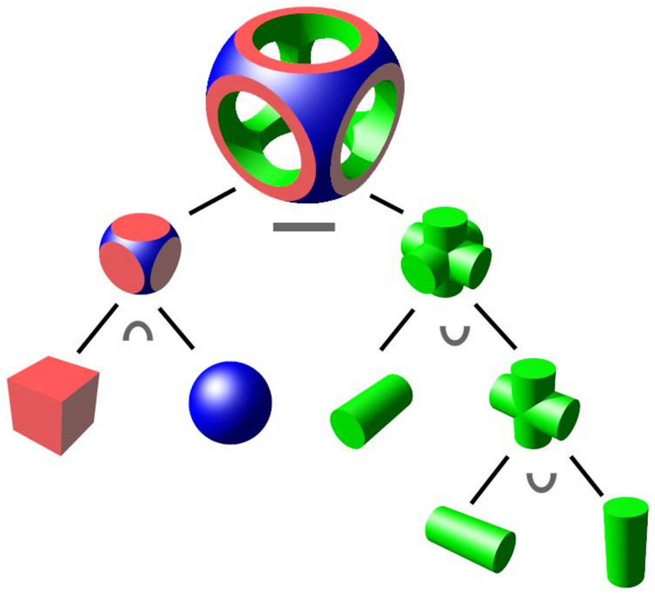


Figure 4-7 Example of a CSG tree adopted from https://en.wikipedia.org/wiki/Constructive_solid_geometry

It can be challenging to determine the visibility of the objects relative to each other in complex geometries and rendering several models. To solve this issue, CSG uses a spatial data structure called BSP tree representation of the objects. BSP tree would recursively divide the scene into two sides (back and front) using a partitioning plane that can orient in any directions. BSP tree has the most flexibility in partitioning the scene compared to the similar data structures.

CSG method includes several main stages to apply the Boolean operation and generate the result of the desired operation.

4.7.1.1 Create BSP- Tree

In computer science, Binary Space Partitioning (BSP) is a method to subdivide a space recursively into multiple sections and sets by a plane which gives a representation of the objects in a BSP-Tree data structure. In this structure, BSP-tree uses spatial information about the objects including the ordering from front-to-back with respect to a fixed view. This method is being performed in several methods and applications including CSG method, collision detection, CAD, 3D video games and ray tracing.

All is required for this stage of CSG implementation is to specify an implicit plane which divides all points of the object in one side of the plane as P^+ that creates a function of $f_1(P^+) > 0$ and the rest of the points as P^- to create the function of $f_2(P^-) < 0$. Also, a first triangle could be defined as the plane with respect to the rest of the triangles to divide them into two subdivisions.

f_1 (p) is the implicit function of the plane created by triangles with counter clockwise order of vertices a, b and c.

$$f_1(p) = ((b-a) \times (c-a)) \cdot (p-a) = 0$$

It could be also faster by using the following format of the equation with taking the cross product:

$$Ax + By + Cz + D = 0$$

Which D is equal to:

$$D = -n.a$$

Figure 4-8 illustrates the BSP-tree creation and representation.

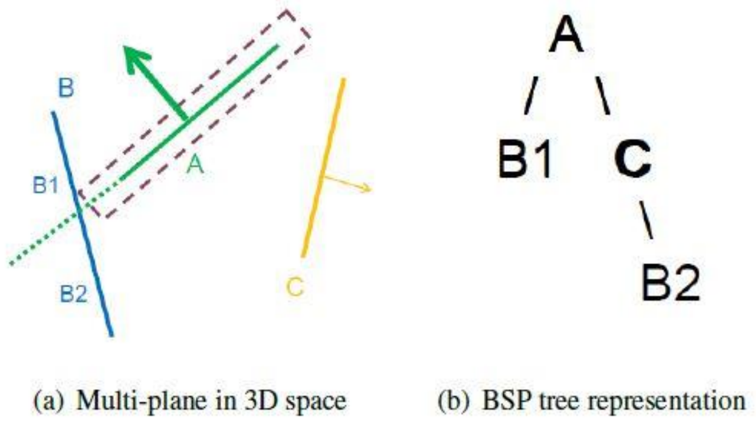


Figure 4-8 BSP-Tree Creation

Adopted from (Segura, Stine, and Yang 2013)

In the above mentioned method, only triangles were considered with either all vertices at the front (to put as the front-subtree) or all the vertices as the back (to put in the back-subtree). For those triangles with both positive and negative vertex functions, the triangle splits into three new triangles (figure 4-9).

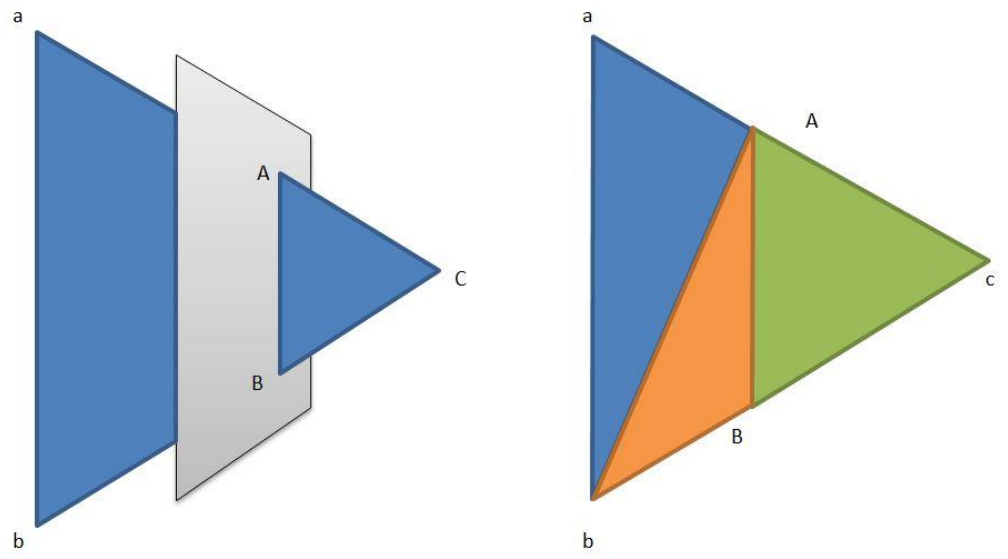


Figure 4-9 Split a Triangle in BSP-Tree

The last component of the BSP-Tree creation is to calculate A and B:

$p(t) = a + t(c - a)$

$n .(a + t(c - a)) + D = 0$

$t = \frac{n.(a+D)}{n.(c-a)}$

$A = a + t(c - a)$

And now the BSP-Tree creation is complete.

4.7.1.2 Merge Two Trees

To start this stage of the CSG method, two separate BSP-Trees for each object should be created initially (one for the scapula bone and the other for the virtual reamer).

For this purpose, to create the BSP-Trees, polygons on the surface of the model would be chosen as the implicit plane to divide the object faces into back and front trees in order to create an efficient data structure to traverse boundaries of complex models. Then the two boundary representations could be merged by traversing one of the trees to get the surface boundary representation of that model. Next, we can push this boundary representation of the first model (Scapula bone) through the tree of the second object's tree (virtual reamer) and vice versa. If a point is at the back of the plane, and there are no other back-subtree planes, that triangle would be considered inside the object. After inserting all the triangles, a list of inside/outside triangles of the first model with respect to the second model will be created for both objects to combine and get the Boolean operation results of the reamer and scapula bone.

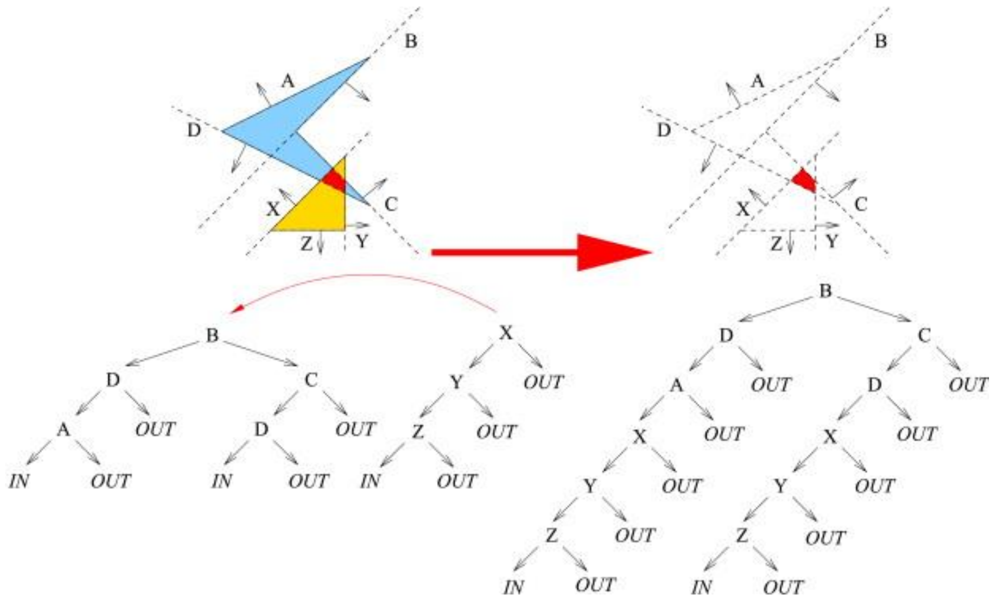


Figure 4-10 Merge the two BSP-Trees

Figure 4-10 shows how the BSP-Trees will be merged to create the list of inside/outside triangles with reference to the other object.

Also the following suede code shows the main algorithm.

```
for i = 2-> N do
 insert(tree$_B$:root, A$_{triangles}$[i])
end for
function INSERT(Node node, triangle T)
     f$_a$ = f$_{(a)}$
     f$_b$ = f$_{(b)}$
     f$_c$ = f$_{(c)}$
     if |f$_a$| < $\varepsilon$ then
          f a = 0
     end if
     if |f$_b$|< $\varepsilon$ then
          f$_b$ = 0
     end if
     if |f$_c$|< $\varepsilon$ then
          f$_c$ = 0
     end if
     if f$_a$ <= 0 && f$_b$ <= 0 && f$_c$ <= 0 then
     if back-subtree does not exist then
          inside-model.add(T)

     else
          insert(node.back-subtree, T)
     end if
     else if f$_a$ >= 0 && f$_b$ >= 0 && f$_c$ >= 0 then
     if front-subtree does not exist then
          outside-model.add(T)
     else
```

```
            insert(node.front-subtree, T)
        end if
        else
            cut the triangle // figure 4-8
        end if
        compute A
        compute B
        T1 = (a;b;A)
        T2 = (b;B;A)
        T3 = (A;B;c)
        if $f_c$ >= 0 then
            insert(node.back-subtree, T1)
            insert(node.back-subtree, T2)
            insert(node.front-subtree, T3)



        else
            insert(node.front-subtree, T1)
            insert(node.front-subtree, T2)
            insert(node.back-subtree, T3)
        end if
end function
```

Three.js also takes advantage of the CSG method to apply the Boolean operations on triangulated mesh objects.

4.7.1.3 Optimization in CSG Method

To accelerate the algorithm, both reamer and scapula are divided into two parts; one for the computations and the other for the rendering process as not getting involved in the calculations. For the virtual reamer, the umbrella surface that cut the bone would be considered for the calculations and for the scapula bone only a small volumetric amount of bone which would be affected in the bone removal process while reaming is selected (Figure 4-11). The rest of the object would not be in the collision detection and mesh Boolean operation computations.

Also, decreasing the resolution of the rendering in Three.js scene could be an optimization for the CSG method.

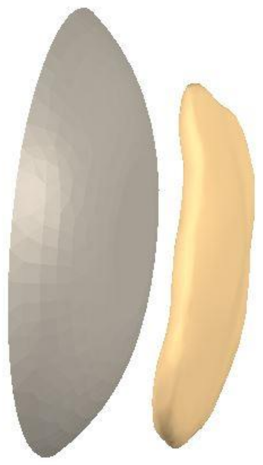


Figure 4-11 Reamer and Scapula Computational Volume

When both objects convert to CSG mode after creating both BSP-Trees for objects, Boolean operation applies on the new lists of BSP-Tree nodes. Lastly, the result should be converted back from the BSP-Tree nodes to triangle mesh objects.

Below, a piece of code utilizing this section is provided:

```
reamerCSG = new ThreeBSP(ReamerGeometry);

scapulaCSG = new ThreeBSP(ScapulaGeometry);

var subtractedNew = scapulaCSG.subtract(reamerCSG);
```

```
result = subtractedNew.toMesh(new THREE.MeshPhongMaterial ({
color : 0xE35D9C }));

result.geometry.mergeVertices();

result.geometry.computeVertexNormals();

scene.add(result);

ScapulaGeometry = result.geometry;
```

As the reaming process is a continuous procedure, each time the CSG algorithm applied on the bone to get the new surface as an output, the new bone object would be considered as the input for the next round of reaming (ScapulaGeometry = result.geometry;).

4.8 Chapter Summary

This chapter contained the main computations and algorithms of this work. For the bone removal procedure, a bounding box collision detection algorithm checks for the first collision between the 3D boxes bounded around the two objects. Once the bounding boxes collide, the ray-triangle intersection algorithm based on Möller-Trumbore runs to find the first intersection of the reamer vertex on the bone face. Lastly, Boolean mesh operation based on CSG method would be applied to compute the amount of the intersected bone, remove that and generate the new resurfaced mesh.

In the next chapter, results of the collision detection and CSG mesh subtraction will be presented.

Chapter 5

5 Performance Evaluation

5.1 Overview

In the two previous chapters, both computational and graphical modules of the proposed work were presented. This chapter contains the final results of the applied algorithms and methods on both objects in WebGL and Three.js scene.

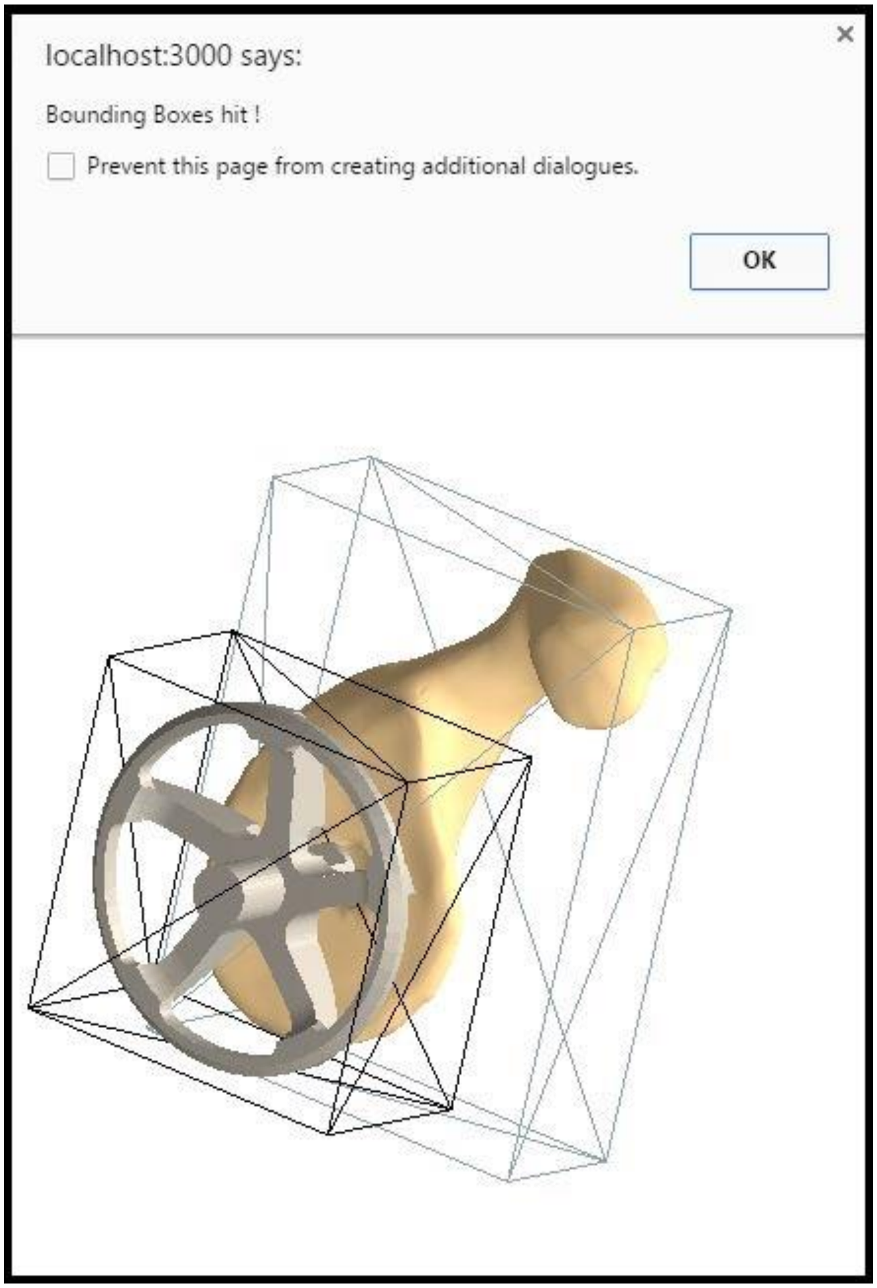
5.2 Objectives

The main goal of the proposed work is to implement a web-based simulator for orthopeadic surgeries in real-time for training and education purposes. To approach this, all computations and graphics of the bone reaming process have been done and were discussed in the previous chapters. In this section, visual results and execution times of the implemented methods and algorithms will be presented.

5.3 Collision Detection

In chapter 4, implemented collision detection algorithm was described in details. For this, we utilized the Möller-Trumbore method of ray- triangle intersection. Compared to other methods, this is still one of the fastest and exact methods as a basis of a number of implemented collision detection algorithms. Besides, ray-triangle intersection was selected for the purpose of more accurate results of intersection points and faster execution time in comparison with other similar implementations such as triangle-triangle intersection method.

The process of finding the intersected volume of the scapula bone to be removed includes several steps. The first step is to detect the collision between the bounding boxes of the virtual reamer and the scapula bone (Figure 5-1).

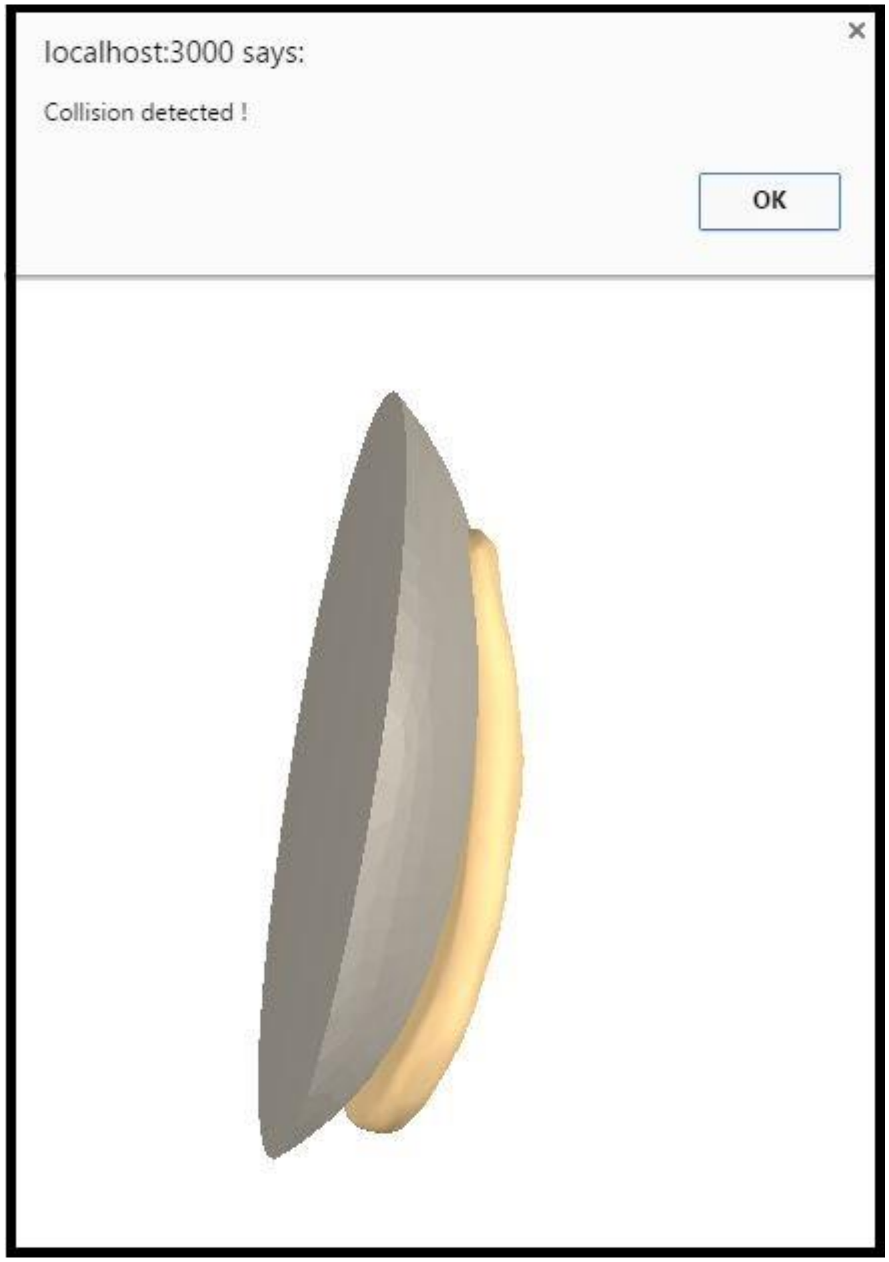

localhost:3000 says:

Bounding Boxes hit !

Prevent this page from creating additional dialogues.

Figure 5-1 Bounding Box Collision Alert

Once the bounding volumes collide, the ray-triangle collision detection using Möller-Trumbore method will be invoked to check for the minimum distance between the objects and the first intersection point of the virtual reamer vertex on the scapula bone face.


localhost:3000 says:

Collision detected !

Figure 5-2 Collision detection Alert

5.4 Mesh Boolean Operation

As discussed in the previous section, the reaming process of the scapula bone contains several steps to reach the mesh Boolean operation. The reason is to prevent the algorithm to create the BSP-Tree data structures of objects and check for the intersection each time the virtual reamer is being transformed (rotation and translation) before colliding with the bone.

As mentioned in the previous chapters, for less computation on the objects, we reduced the number of vertices and faces to apply in the algorithms.

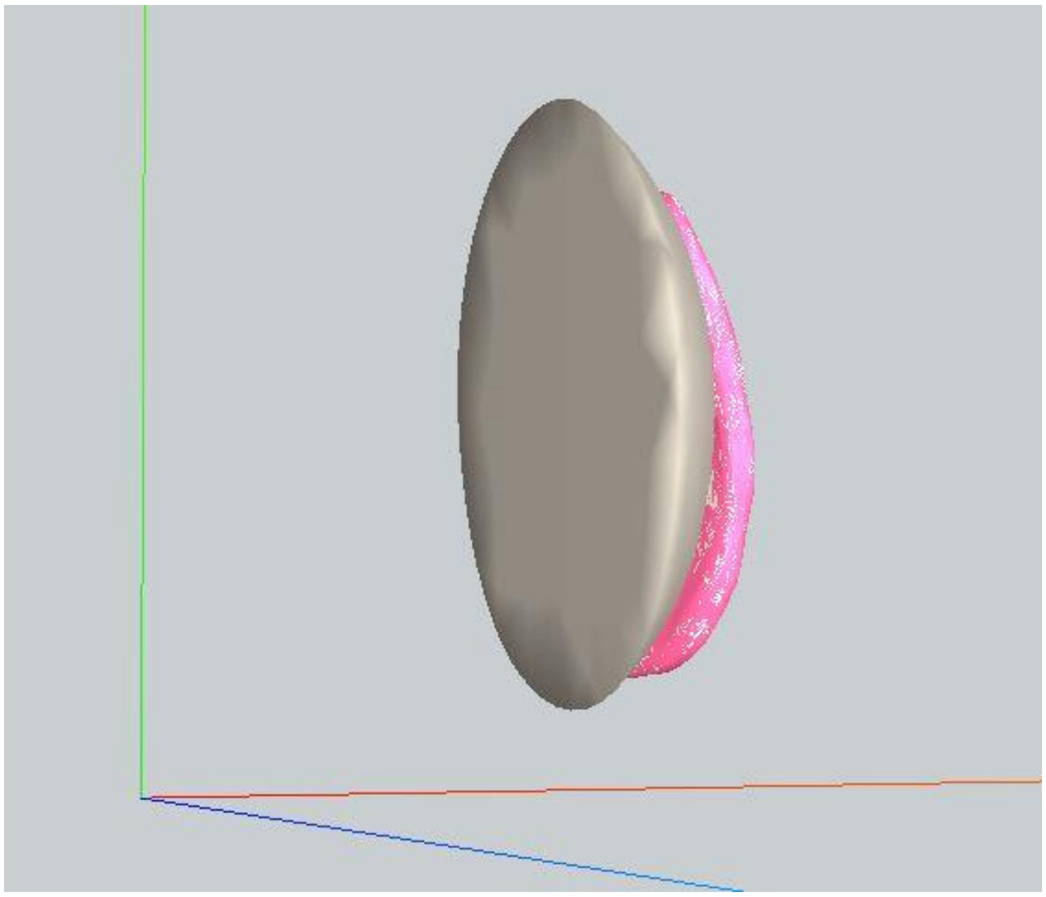


Figure 5-3 Mesh Boolean Subtraction of the Reamer on the Bone

Figure 5-3 shows the first mesh Boolean subtraction between the scapula bone and the reamer. The subtracted bone (the new generated bone) is colored in pink.

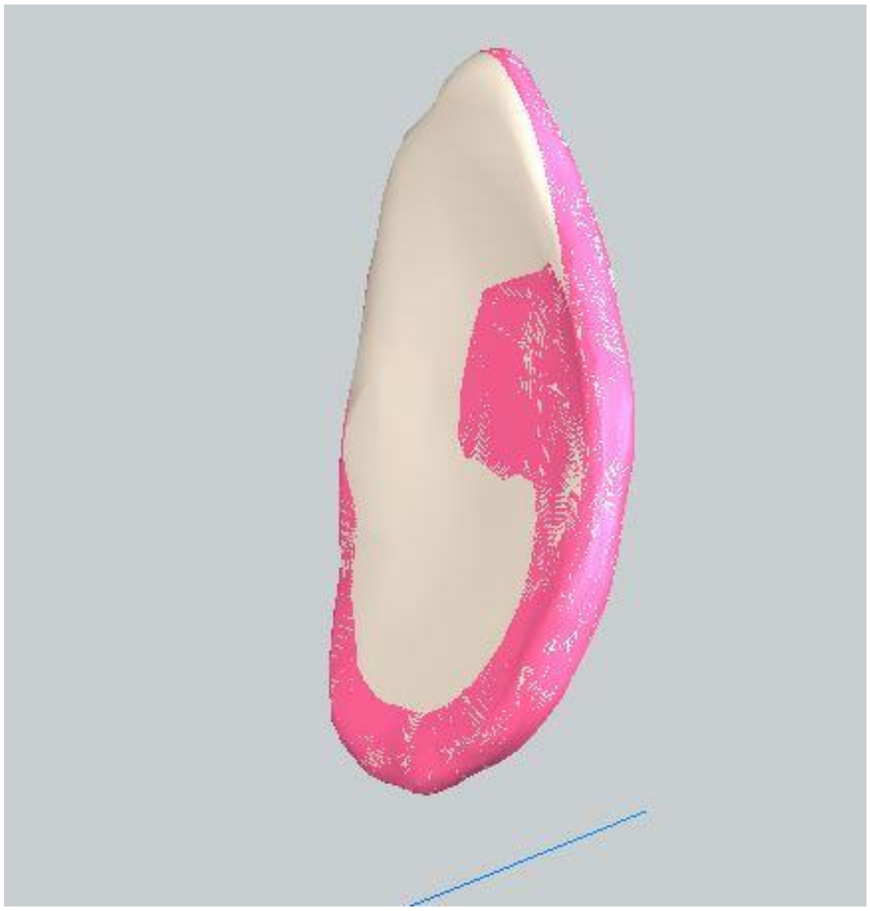


Figure 5-4 Subtraction on the Scapula Bone

Figure 5-4 illustrates the bone after the first subtraction. The white bone is the original scapula bone before subtraction and the pink bone is the one after the first Boolean operation.

Figure 5-5 shows the original bone (left side in white color) and the subtracted bone (right side with pink color).

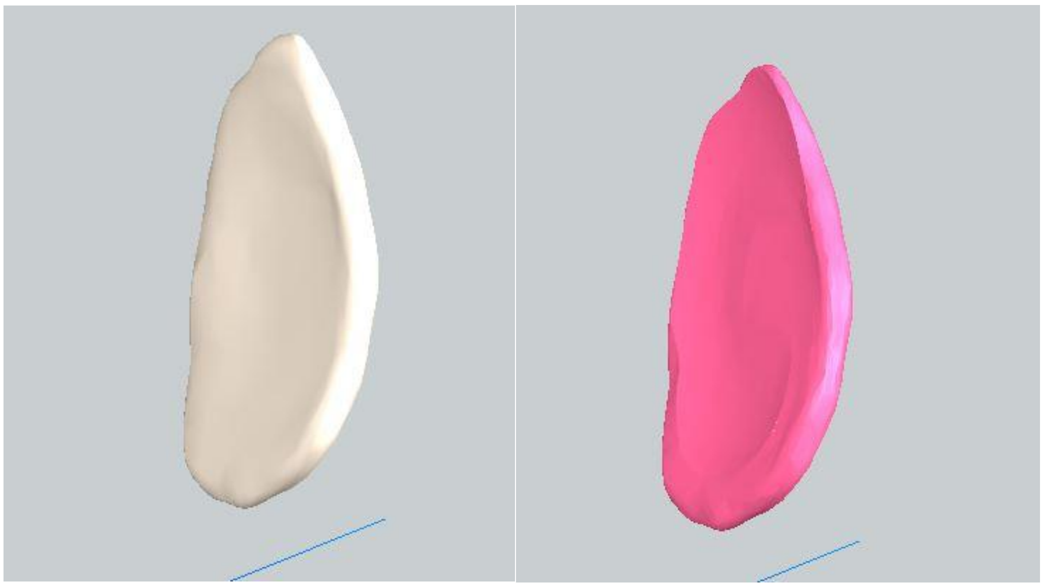


Figure 5-5 Scapula Bone Before and After Subtraction

Figure 5-6 shows the wireframe mesh representation of the original bone.

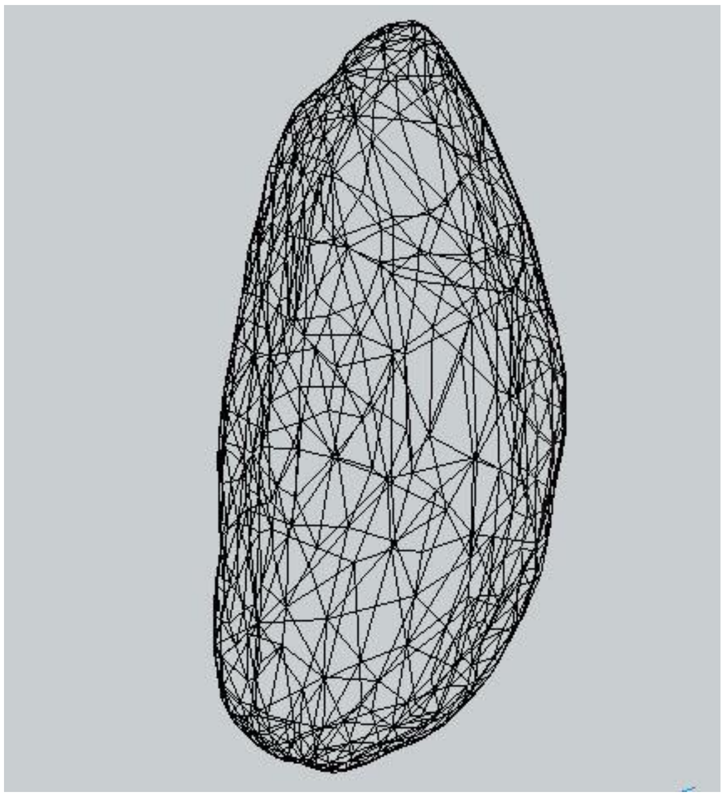


Figure 5-6 Original Bone Wireframe Representation

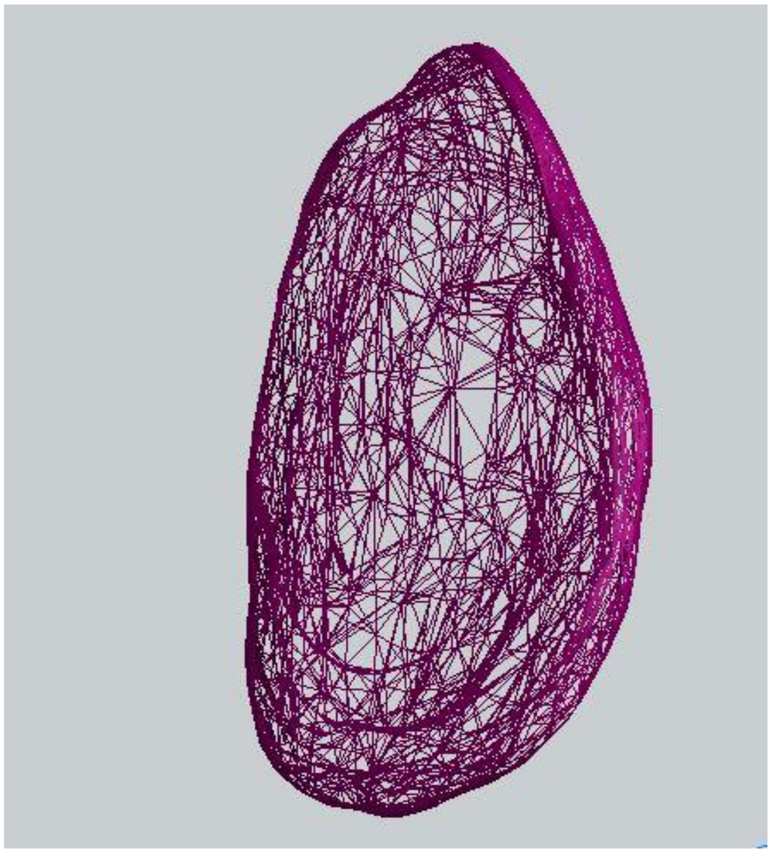


Figure 5-7 Subtracted Bone Wireframe Representation

Figure 5-7 shows the wireframe representation of the bone after the first subtraction. The new generated object using CSG method is a more complex geometry with more vertices and faces compared to the original bone (Figure 5-6 and 5-7). As discussed in the CSG method, to recreate triangles, each triangle converts to three more triangles. Thus, the object becomes bigger and heavier for the next subtraction process and it results an increasing execution time for the next steps as the reaming process is a continuous operation during the surgery.

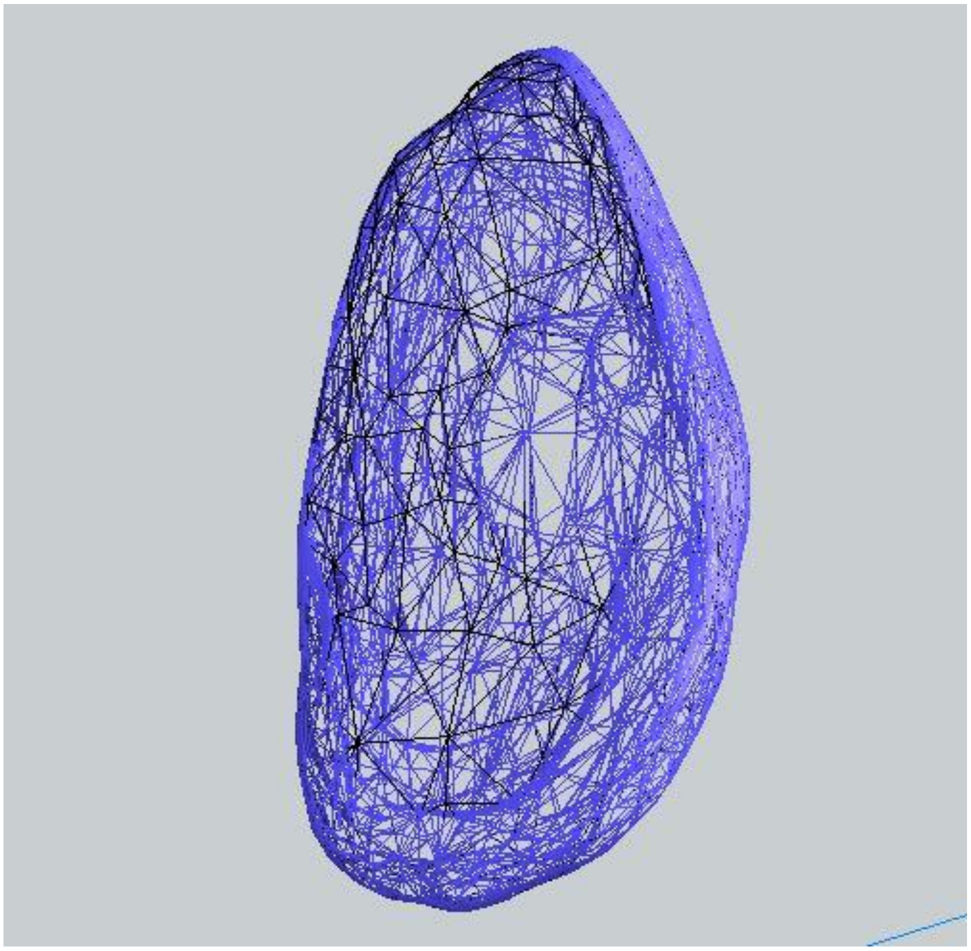


Figure 5-8 Overlapping of Both Original and Subtracted Bone

Figure 5-8 could compare the bone before and after the subtraction process in wireframe. In this figure, the original bone wireframe is colored in black and the subtracted bone wireframe in blue. As can be seen from this figure, the original bone has less and bigger triangles.

5.5 Evolution of Bone Geometry during Reaming Procedure

The following virtual experiment was performed on the bone geometry to measure the volume of the removed part of the bone after applying each mesh Boolean operation. For this experiment, the virtual reamer is translating along the X axis to the bone geometry to evaluate the results of the subtraction of the bone in multiple increments.

Table 5-1 shows the number of the vertices and faces of the bone after each subtraction.

Table 5-1 Comparison of the bone vertices and faces in 4 experiments

Experiment	Number of Vertices	Number of Faces
Original bone	502	1000
After first subtraction	6682	12045
After second subtraction	19741	36635
After third subtraction	34106	64264
After forth subtraction	55536	106091

5.6 Dependence of Hardware/Software Platform

In the next step, multiple tests of mesh Boolean operation will be presented on different platforms with various CPU configurations and different web browsers (Table 5-2).

Table 5-2 Tests on different browsers and different computers

Configuration / Browser	Windows10 Laptop (ms)	Windows10 PC (ms)
Google Chrome	233	259
Mozilla Firefox	535	599
Internet Explorer	249	292
Edge	272	266

Different browsers have different JavaScript engines to run the HTML code. The JavaScript engine for Chrome is called V8.Google claims that this engine is faster than Firefox and Safari.

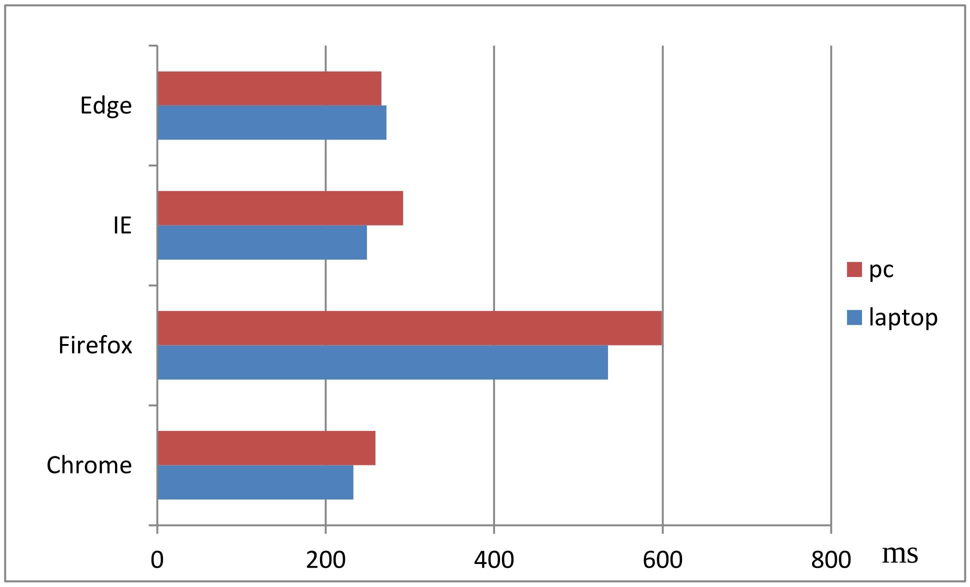
The run time of the algorithm on the same machine and situation is based on several factors such as OS, graphics card and, most importantly, the test we run.

Results on Windows show similar performance that chrome is still faster. But for example on Linux, Firefox did not use full OpenGL capabilities of the card for sometimes. And the same goes with GPU. Apparently, NVidia cards do better job and offer similar performance in both browsers, with Chrome slightly ahead.

That brings to the benchmark. Browsers do not only pass the code to the GPU. A part of code needs to be processed by main program. It is widely known that Chrome's JavaScript engine is much faster in some specific tasks, and if benchmark relies on that, it will run much faster in Chrome. Lastly is environment management. Chrome separates tabs into their own processes while Firefox does not do that.

Table 5-3 illustrates the bar chart for the comparison of the execution time of mesh Boolean operation on different browsers and two different computers.

Table 5-3 Run time on different browsers and computers



The laptop configuration is windows 10 professional x64, Intel core i5 CPU 2.40 GHz and 4.00 GB installed memory (RAM) which shows the blue line in the bar chart.

The pc is configured with windows 10 professional x64 and Intel core 2 Quad CPU 2.66 GHz with 4.00 GB RAM which is illustrated in red color.

5.7 Chapter Summary

Chapter 5 included all the visual results of the methods discussed in chapter 4 including the bounding box collision detection, ray-triangle intersection and finally the mesh Boolean operation based on CSG method. We demonstrated the comparison of the bone object before and after the first subtraction. Also, the reason of the changes in the new generated bone was discussed.

In the next chapter, conclusion and future works will be explained.

Chapter 6

6 Closing Remarks

6.1 Discussion and Conclusion

This thesis began with the background of the shoulder anatomy, the main reasons of why it needs a surgery and the solution to it with total shoulder arthroplasty operation plus the literature review of the previous simulations and related works for virtual reality implementations. This provides answers to why a surgical simulator in virtual reality environment helps for training and education purposes and also why WebGL could be efficient platform for this work. Next, the main framework was proposed to present the architecture of this work including all its components and break those components down to find a solution for smaller and simpler tasks with respect to the main complex task. Chapter 3 had the main graphics implemented for this work using WebGL API and its popular Three.js library for 3D programming. It contains all the scene's components comprising cameras, light sources, objects with different types and loaders and how they all setup together to create the current surgical simulator scene. Furthermore, the user interaction involved to complete the process of the simulation implementation. The next chapter described all the computations require for the process of reaming the scapula bone in the virtual reality simulator to get the smooth congruent surface of the bone while it is being received and visualized by the output devices (monitor, google glasses or haptic device). Lastly, the results of the computations of bone removal process through the VR simulation were demonstrated to show how fast and accurate this simulation is currently working.

6.2 Implementation Challenges and Lessons Learned

This work has been performed using WebGL API to allow the users to run the simulation on the desired web browser regardless of the hardware/software.

WebGL has various advantages such as the light libraries, being fast, cross-platform and having complicated built-in functions.

On the other hand, WebGL has its own disadvantages including the little support and undocumented libraries (such as Three.js) as WebGL is a new API. Also, in some cases, it becomes slow with no clear reasons so it is difficult to solve. It is also hard sometimes to accelerate the functions in WebGL without rewriting and modifying the basic libraries. For Instance, the latest OBJLoader library loads obj files and extract the buffer geometry instead of the geometry. So, to access the vertices and faces and all the related features, we included an older version of the loader to have access to geometry but rewrite the library to be adapted with the new Three.js library versions. Another example could be the controls and cameras in Three.js. To utilize the Trackball control using orthographic camera, we required to rewrite the Trackball control library and add the scroll feature separately. Furthermore, most developer libraries are still in an early stage so it makes WebGL to utilize different game engines and 3D libraries.

6.3 Future Direction

Throughout the duration of this thesis, a number of challenges of both the surgery and the simulation have been mentioned. Currently, the framework is developed in JavaScript and WebGL that works with the browser. Presently, all the computations are running and processing through the CPU. Shader programming in WebGL using GLSL based on OpenGL was discussed in this thesis. Consequently, adding GPU programming through shaders to accelerate the process of mesh Boolean subtraction as well as the object rendering time is greatly desired for the future directions.

Another possible future extension of the current work would involve the extension of the developed algorithms to other optimal design geometries and mesh subtraction algorithm to remove limitations of objects water-tightness and manifoldness.

Moreover, improving the current mesh Boolean operation into an optimal algorithm using CSG method to create less number of vertices and faces as a result could be a desired future work. It prevents the increasing execution time of the next Boolean operations because of the large number of vertices and faces of the input object.

Lastly, rendering and processing complex and big geometries could lead to an ideal and fast, practical algorithm to be employed in various simulations exclusive of the type and the process.

Bibliography

Aldinger, Peter R., Patric Raiss, Markus Rickert, and Markus Loew. 2010. “Complications in Shoulder Arthroplasty: An Analysis of 485 Cases.” *International Orthopaedics* 34 (4): 517–24. doi:10.1007/s00264-009-0780-7.

Bäckman, Nils. 2011. “Collision Detection of TriangleMeshes Using GPU.” http://www.diva-portal.org/smash/record.jsf?pid=diva2:403566.

Bernstein, Gilbert, and Don Fussell. 2009. “Fast, Exact, Linear Booleans.” In *Computer Graphics Forum*, 28:1269–1278. Wiley Online Library.

Brassard, Gilles, and Paul Bratley. 1996. *Fundamnetals of Algorithms*. Prentice-Hall, Inc.

Cantor-Rivera, Diego, Robert Bartha, and Terry Peters. 2011. “Efficient 3D Rendering for Web-Based Medical Imaging Software: A Proof of Concept.” In , edited by Kenneth H. Wong and David R. Holmes III, 79643A. doi:10.1117/12.878814.

Chen, Bijin, and Zhiqi Xu. 2011. “A Framework for Browser-Based Multiplayer Online Games Using WebGL and WebSocket.” In *Multimedia Technology (ICMT), 2011 International Conference on*, 471–474. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6001673.

Cormen, Thomas H., Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms*. Vol. 6. MIT press Cambridge. http://sites.google.com/site/ingridteles02/Introduction.to.Algorithms.pdf.

Danchilla, Brian. 2012. *Beginning WebGL for HTML5*. The Expert’s Voice in Web Development. [Berkeley, Calif.] : New York: Apress ; Distributed to the book trade worldwide by Springer Science+Business Media.

Dasgupta, Sanjoy, Christos H. Papadimitriou, and Umesh Virkumar Vazirani. 2016. “Algorithms.” http://lib.agu.edu.vn:8180/collection/handle/123456789/3592.

Dirksen, Jos. 2014. *Three.js Essentials*. Packet Publishing Ltd.

———. 2015. *Three.js Cookbook*. Packet Publishing Ltd.

Frigo, Matteo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. “Cache-Oblivious Algorithms.” In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, 285–297. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=814600.

Gopi, M, and Shankar Krishnan. 2002. “A Fast and Efficient Projection-Based Approach for Surface Reconstruction.” In *Computer Graphics and Image Processing, 2002. Proceedings. XV Brazilian Symposium on*, 179–186. IEEE.

“Introduction to Parallel Computing.” 2016. Accessed July 25. https://computing.llnl.gov/tutorials/parallel_comp/.

Jiménez, Pablo, Federico Thomas, and Carme Torras. 2001. “3D Collision Detection: A Survey.” *Computers & Graphics* 25 (2): 269–285.

Karelse, Anne, Steven Leuridan, Alexander Van Tongel, Iwein M. Piepers, Philippe Debeer, and Lieven F. De Wilde. 2014. “A Glenoid Reaming Study: How Accurate Are Current Reaming Techniques?” *Journal of Shoulder and Elbow Surgery* 23 (8): 1120–27. doi:10.1016/j.jse.2013.11.023.

Learning Three.Js: The JavaScript 3D Library for WebGL: Create and Animate Stunning 3D Graphics Using the Open Source Three.js JavaScript Library. 2013. Birmingham: Packt Publishing Limited.

Leung, Catherine, and Andor Salga. 2010. “Enabling Webgl.” In *Proceedings of the 19th International Conference on World Wide Web*, 1369–1370. ACM. http://dl.acm.org/citation.cfm?id=1772933.

Levitin, Anany, and Soumen Mukherjee. 2003. *Introduction to Design Ad Analysis of Algorithms*. Addison-Wesley Reading, MA.

Luque, Rodrigo G, João LD Comba, and Carla MDS Freitas. 2005. “Broad-Phase Collision Detection Using Semi-Adjusting BSP-Trees.” In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, 179–186. ACM.

Mirtich, Brian. 1997. “Efficient Algorithms for Two-Phase Collision Detection.” *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, 203–223.

Möller, Tomas, and Ben Trumbore. 2005. “Fast, Minimum Storage Ray/triangle Intersection.” In *ACM SIGGRAPH 2005 Courses*, 7. ACM.

Motion, Leap. 2016. “Leap Motion.” Accessed July 25. https://www.leapmotion.com/.

Nguyen, Duong, Louis M. Ferreira, James R. Brownhill, Graham J.W. King, Darren S. Drosdowech, Kenneth J. Faber, and James A. Johnson. 2009. “Improved Accuracy of Computer Assisted Glenoid Implantation in Total Shoulder Arthroplasty: An in-Vitro Randomized Controlled Trial.” *Journal of Shoulder and Elbow Surgery* 18 (6): 907–14. doi:10.1016/j.jse.2009.02.022.

Nuss, K. M., and Brigitte von Rechenberg. 2008. “Biocompatibility Issues with Modern Implants in Bone-a Review for Clinical Orthopedics.” *Open Orthop J* 2: 66–78.

"Parallel.js: Parallel Computing with Javascript." 2016. Accessed July 25. https://adambom.github.io/parallel.js/.

Pritchett, James W. 2016. "Shoulder Resurfacing." Accessed June 2. https://www.researchgate.net/profile/David_Marker/publication/24398038_Shoulder_Resurfacing/links/0912f50bbfd60b8ad8000000.pdf.

"Rendering Pipeline Overview - OpenGL.org." 2016. Accessed July 25. https://www.opengl.org/wiki/Rendering_Pipeline_Overview.

Saltzman, Matthew D., Aaron M. Chamberlain, Deana M. Mercer, Winston J. Warme, Alexander L. Bertelsen, and Frederick A. Matsen. 2011. "Shoulder Hemiarthroplasty with Concentric Glenoid Reaming in Patients 55 Years Old or Less." *Journal of Shoulder and Elbow Surgery* 20 (4): 609–15. doi:10.1016/j.jse.2010.08.027.

Segura, Christian, Taylor Stine, and Jackie Yang. 2013. *Constructive Solid Geometry Using BSP Tree*. http://www.andrew.cmu.edu/user/jackiey/resources/CSG/CSG_report.pdf.

"Shoulder Joint Replacement-OrthoInfo - AAOS." 2016. Accessed July 25. http://orthoinfo.aaos.org/topic.cfm?topic=A00094.

Shreiner, Dave, Graham Sellers, John M. Kessenich, Bill Licea-Kane, and Khronos OpenGL ARB Working Group, eds. 2013. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Eighth edition. Upper Saddle River, NJ: Addison-Wesley.

Sperling, John W., Robert H. Cofield, and Charles M. Rowland. 2004. "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* 13 (6): 604–13. doi:10.1016/j.jse.2004.03.013.

Vaughan, Neil, Venketesh N. Dubey, Thomas W. Wainwright, and Robert G. Middleton. 2016. "A Review of Virtual Reality Based Training Simulators for Orthopaedic Surgery." *Medical Engineering & Physics* 38 (2): 59–71. doi:10.1016/j.medengphy.2015.11.021.

Wang, Wilson, Youheng Ouyang, Chye Khoon Poh, and others. 2011. "Orthopaedic Implant Technology: Biomaterials from Past to Future." *Annals of the Academy of Medicine-Singapore* 40 (5): 237.

"WebGL - OpenGL ES 2.0 for the Web." 2016. *The Khronos Group*. Accessed July 25. https://www.khronos.org/api/webgl/.

Weichert, Frank, Daniel Bachmann, Bartholomäus Rudak, and Denis Fisseler. 2013. "Analysis of the Accuracy and Robustness of the Leap Motion Controller." *Sensors* 13 (5): 6380–93. doi:10.3390/s130506380.

Whiting, William Charles, and Stuart Rugg. 2006. *Dynamic Human Anatomy*. Vol. 10.

Curriculum Vitae

Name:	Arezoo Tony
Post-secondary Education and Degrees:	Azad University of Tehran Tehran, Iran B.Sc, Software Engineering 2006-2011
	University of Western Ontario London, Ontario, Canada M.E.Sc, Computer and Electrical Engineering 2014-2016
Related Work Experience	Teaching Assistant The University of Western Ontario 2014-2016