ViC : virtual cadaver - a prototype extendible system for building photorealistic interactive visualizations of human anatomy using game development technology

Alexander Yakobovich
Ryerson University

ViC: VIRTUAL CADAVER – A PROTOTYPE EXTENDIBLE SYSTEM FOR BUILDING PHOTOREALISTIC INTERACTIVE VISUALIZATIONS OF HUMAN ANATOMY USING GAME DEVELOPMENT TECHNOLOGY

by

Alexander Yakobovich, BSc, Ryerson University, Toronto, Ontario, 2009

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2011

ViC: Virtual Cadaver - A prototype extendible system for the photorealistic interactive visualization of human anatomy using game development technology

Alexander Yakobovich

MSc, Computer Science, Ryerson University, 2011

Abstract

This thesis presents a Virtual Cadaver system, ViC, which allows users to interactively "cut open" and visualize a highly realistic representation of 3D human anatomy, including skin, muscles, bones, arteries and many other anatomical systems and structures. ViC's interaction and visualization functionality, coupled with a multi-touch interface, provide real-time cutaway operations using simple and familiar gestures. To support the cutaway feature, ViC's 3D human anatomy dataset is preprocessed, breaking up the large anatomy system data meshes into smaller, more manageable mesh "fragments". The shape of each fragment is crafted to support semantically meaningful cutaways while the "granularity" of the fragments supports interaction efficiency. ViC uses game technology to enable highly realistic rendering of human anatomy. Furthermore, ViC's performance and cutaway capabilities were evaluated on consumer-grade hardware to confirm that real-time interaction and visualization with highly-responsive multi-touch input actions can be achieved.

ACKNOWLEDGEMENTS

I wish to thank my supervisor Dr. Tim McInerney for the strong dedication to the work documented by my thesis. His passion was reflected in the countless hours spent on guiding me through the unfamiliar territory of developing a Master's thesis. The effort and attention that he invested in this work is unparalleled by anyone in my entire university career.

A big thank you to all my close friends for understanding the importance of my work and being patient with me while I was sacrificing the many promises I made in order to finish it.

I heartfelt thank you to my mother, Irina Yakubovich, for her continuous support. Having encouraged me every step of the way to only learn from the mistakes of the past and look forward to the best of the future, she kept me going at times when it didn't seem possible.

TABLE OF CONTENTS

LIST OF FIGURES

List of Algorithms

Chapter 1: Introduction

Modern-day medical imaging and computer simulation allow medical professionals to both view and manipulate highly detailed medical data. For example, the entire field of radiology is heavily dependent on the ability to visualize and measure data gathered from x-ray, ultrasound and Magnetic Resonance Imaging, to name a few. With the advent of powerful graphics processors and advanced computer algorithms, surgeons can now plan and even simulate surgical procedures on virtual representations of patients [1]. Furthermore, visualizing and interacting with real or hand-crafted 3D medical data enables medical educators and students to perform tasks that would otherwise be inefficient, difficult or impossible to replicate on a real cadaver.

Despite the availability of powerful hardware and sophisticated visualization algorithms, simply rendering highly detailed human anatomy data is only part of the medical professionals' (users') needs. Even the most realistic renderings must be accompanied by flexible controls for interactively generating meaningful contextual views of the data. Contextual data views de-emphasize or cutaway part of an *occluding* anatomical object in order for the user to be able to see a hidden (*occluded*) object while still retaining some visual cue of the occluding object shape. Realizing this semantic view generation capability is especially challenging due to the inherently difficult nature of interacting with 3D data that is projected onto a 2D plane for display. If the user is not able to navigate through the visualization and select regions of interest, or if there is limited ability to generate spatially local semantic views from the often huge 3D datasets, their examination of the data remains at a global anatomy system level and is therefore often less insightful. For example, certain anatomy "browsers" [2] provide the ability to view whole anatomy systems[1], such as the arterial, skeletal, and muscular systems, separately or in combination. However, if the user wishes to visualize the spatial relationship of a specific artery segment with respect to the neighboring bone and muscles, then these global anatomy system views are often ineffective.

A complete visualization system is therefore one that provides not only powerful rendering capabilities but also simple and intuitive interaction controls. The controls must be easily useable and/or quickly learnable by users who are not computer graphics experts. This can be achieved by utilizing automated, or pre-authored, controls that directly generate semantic views of the data [3], rather than changing parameter values that indirectly affect the data view. That is, the controls must encapsulate the ability to not just interact with the 3D data, but interact with it in such a way that what the user sees and does is meaningful to them. The prime example is the already mentioned issue of removing or deemphasizing an occluding anatomy object in order to not only see the target object but also get a sense of the visual relationship between the two objects.

Meaningful visualization and interaction capabilities are not the only challenges when developing an anatomy visualization system. Given that the system must support photorealistic rendering of the data as well as highly-responsive interaction controls, challenges of achieving high performance when rendering and performing geometric computations arise. In addition, the computer hardware that the system runs on must be affordable and accessible to its intended users. This means that the system must be designed to effectively harness all the rendering and compute power that is available on modern personal computers, such as Intel-based PCs.

[1] An *anatomy system* or *body system* is a group of organs that work together, usually for a single purpose

The final challenge that must be considered in the design of an interactive anatomy visualization program is that of program longevity through extendibility. Programs that are designed for one purpose with only a single set of capabilities that cannot be extended beyond the original intentions of the developer are best described as *applications*. However, a program that can be modified beyond its default capabilities by design, either through an Application Programming Interface (API), in-application editing or source code that can accommodate expansion, can be described as a *system*. An extendible anatomy exploration system should allow the intended user, whether it's the end user, a third-party developer, or a medical researcher, to expand the tool beyond the given capabilities in order to add new functionality, modify existing functionality or to completely redefine the ultimate purpose of the system.

This thesis presents a prototype virtual cadaver system, known as ViC, for the photorealistic interactive visualization of human anatomy using game development technology. The thesis presents solutions to all of the aforementioned issues and challenges. The following section documents these solutions as contributions.

1.1 Contributions

This thesis presents a number of contributions delivered in a prototype system, known as 'ViC'. ViC is a real-time, visually compelling, high performance visualization system that features interactivity elements allowing the user to perform a number of operations on a 3D dataset of human anatomy that mimic real-world actions. ViC's game engine based renderer uses photorealistic rendering capabilities to generate close to real-world visuals. In addition, ViC is designed to be extendible through the use of a component-based software model. The following subsections present each thesis contribution in more detail.

AN INTERACTIVE VIRTUAL CADAVER

ViC is a system that is not only a browser of human anatomy but an interactive virtual cadaver. The distinction between an anatomy browser and an interactive virtual cadaver is the ability of the latter to manipulate 3D data at a fine scale, and not just the ability to navigate through it. A browser typically allows the user to view, emphasize/deemphasize, isolate, and combine "global" anatomy systems, such as the arterial system or the skeletal system, or perhaps also individual anatomy objects. These viewing capabilities are often controlled using simple sliders. An interactive virtual cadaver, on the other hand, encompasses these browsing capabilities as well as the manipulation of regions or parts of individual anatomy objects, mimicking tissue cutting and peeling operations on a real cadaver. That is, while a browser allows showing, hiding, isolating[2] and viewing objects, a virtual cadaver allows direct manipulation of object regions and parts.

[2]*Isolation* of a 3D object involves removing, hiding or deemphasizing (e.g. through the use of transparency) all other objects in a 3D scene in order to make the target object the only (or most) visible object in the 3D scene

ViC uses a gesture-based operation in order to unwrap or peel away patches of skin or individual muscles etc. to reveal the anatomy objects or systems that they occlude. The key feature of the cutaway portion of ViC is the use of partitioned anatomical surface meshes, such as a muscle or bone or artery, that are broken up into smaller pieces known as "fragments". These fragments can be progressively cut away with a simple sweep gesture of a finger on a touch screen, allowing the user to expose underlying anatomy. The fragmentation of the anatomy meshes was performed in a preprocessing step using a semi-automatic authoring system that subdivided the meshes into logical pieces. For example, a tubular artery branch was fragmented into smaller tubes. This "semantic fragmentation" constrains the interactive cutaway operation such that it makes sense to the user. Constraining the cutaway in this manner can be contrasted to systems that allow a user to cut away arbitrarily shaped regions of a surface mesh. Giving the user more or full control over the cutaway inherently adds significantly more complexity to the process, including the increase in the number of controls required to perform the operation, a dramatic increase in the computational expense, and the addition of an intermediate step to interactively define the cutaway region. The constrained cutaway operation in ViC, on the other hand, requires minimal input from the user. Simple touch and drag gestures result in a progressive cutaway of neighboring mesh fragments. For example, the skin fragments of the upper arm are defined as patches that wrap around the arm, following the arm's natural cylindrical shape. The constrained cutaway operation *unwraps* some or all of these patches, as many as the user desires, to expose the muscles, veins, etc., underneath. For this anatomy part, the unwrapping is well understood by the user and allows the user to focus on the exploration of the anatomy rather than on constantly thinking about how to define a region of an occluding mesh that will expose the occluded anatomy of interest.

PHOTOREALISTIC REAL-TIME VISUALIZATION USING GAME ENGINE TECHNOLOGY

ViC takes advantage of game-engine[3] based GPU-accelerated real-time rendering and shading. It features photorealistic visualizations of human anatomy not just for the purpose of aesthetics, but also for the purpose of realism and accuracy. All rendering, shading, lighting, post-processing and other techniques that contribute to the realism of ViC are computed at run-time. The resultant frame rate of the system is guaranteed to be no less than the refresh rate of a modern LCD display – 60 frames per second – in order to create a smooth and realistic user experience.

TOUCH-BASED NATURAL CONTROLS

ViC features a very compact set of controls that are used for navigation and interaction using a multi-touch based interface. A touch interface was chosen over a mouse/keyboard combination as it offers a more natural interaction [4]. The controls are gesture-based with the gestures ranging from a single finger swipe to multi-finger taps. The gestures are complimented with on-screen notifications as well as selectable options.

EFFICIENT EXECUTION ON MAINSTREAM COMPUTER HARDWARE

ViC is built on top of a software platform that has been designed for consumer-level hardware. This means that ViC does not require any specialized or overly expensive hardware that cannot be purchased off-the-shelf at a local computer store. The hardware on which ViC was evaluated is considered as a mid-high range gaming PC. The touch display on which ViC was evaluated is a *30*-point touch interface. However, ViC has been designed to run on a touch display that supports just *2* touch points.

[3] A game engine is a system (often along with a development environment) that facilitates the creation of computer games.

HARNESSES STATE-OF-THE-ART GAME DEVELOPMENT TECHNOLOGY

ViC uses a game development platform and a game engine that were created exclusively for game development and deployment. This contribution serves as proof that a scientific development project is not limited solely to scientific-grade packages/platforms. ViC uses the Microsoft XNA framework [5] and the SunBurn [6] game engine for scene rendering, lighting and post-processing. The photorealistic and efficient rendering of human anatomy achieved using a game engine, with a minimal amount of programming, is unmatched by scientific visualization platforms such as VTK (see Chapter 2). Scientific platforms currently do not support a sufficient set of features presented at a high enough level of interface abstraction to easily achieve similar results.

EXTENDIBLE SOFTWARE SYSTEM

ViC is designed to be modular and extendible *system* rather than an *application*. This creates opportunities for other students or professionals to continually increase the functionality of ViC towards a more complete representation of a real cadaver, dissectible with virtual surgical tools. ViC's modularity and extendibility is due to its use of a component based software model. ViC is constructed using Microsoft's free and publically available XNA game development framework (see Chapter 3).

1.2 Thesis Outline

The remainder of this thesis is organized into the following chapters.

Chapter 2 is split into two separate but equally important sections. The first section is a survey of numerous existing applications, platforms and techniques that focus on the interactive visualization of 3D datasets of human anatomy. The approaches surveyed are unique in their technical implementations, and each achieves some of the goals presented in the thesis contributions. These existing approaches utilize different rendering techniques, such as volumetric rendering or web graphics based rendering, as well as different interaction techniques such as specialized surgical interfaces. Specific features and shortcomings of these approaches are examined.

The second section is a survey of general visualization platforms, ranging from game engines to modeling applications to even more generic kits such as Visualization Toolkit (VTK). These platforms themselves do not offer sufficient "out of the box" support for the requirements of a virtual cadaver system outlined in the contributions of this thesis. However, they do potentially offer the foundation for creating one, albeit with a great deal more effort. These platforms were considered as candidates for creating ViC.

Chapter 3 begins by explaining why the Microsoft XNA framework was chosen as the foundation of ViC. The remainder of the chapter dissects ViC to explain how it works, how to use it and how to extend it. The purpose of XNA as the foundation of ViC is explained, as well as how its component based model contributes to ViC's modularity, making it highly extendible. The fragmentation preprocessing of the 3D human anatomy dataset that ViC utilizes to provide its cutaway operation is also covered.

Chapter 4 presents a number of experiments to demonstrate that ViC is a true interactive virtual cadaver as opposed to just an anatomy browser. Extendibility of ViC is also put to the test by expanding its capabilities beyond those of the prototype[4] presented. The chapter concludes with performance evaluation both before and after system extendibility.

Chapter 5 summarizes the achievements presented through the prototype, ViC, and suggests further improvements to the system through possible future work.

[4] The term *prototype* is used to refer to ViC's stock functionality and capabilities throughout this thesis.

Chapter 2: Literature Survey

In the pursuit of a balanced system that offers powerful interaction capabilities, longevity through extendibility, realistic rendering capabilities and high performance, this chapter covers a number of categories of existing work. These include similar human anatomy visualization systems presented in the research literature or free/commercial software, as well as platforms that could serve as foundation for ViC. The following sections describe a wide range of such existing techniques, tools and frameworks.

2.1 Existing Approaches

Google Body

Google Body (Body Browser) [2] is a 3D model viewer of the human body that features navigation and *peel back* of anatomical layers (i.e. systems such as the skeletal system) through transparency control. The browser allows the user to select anatomical parts to identify them, or search for them by name. The principal input device, a mouse, is used to move the camera from side to side or to rotate the camera around the upright human body. A single mouse button click is used to select anatomy parts. A double-click is used to set the selected part as the view target by isolating it.

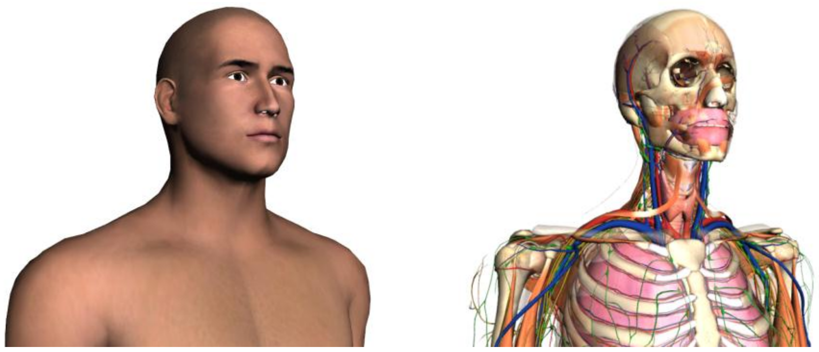


Figure 1: Opaque skin and fully transparent skin with underlying anatomy systems captured from Google Body [2]

Google Body is a showcase of WebGL (Web-based Graphics Library) technology that allows a WebGL-compatible web browser to render 3D graphics by executing graphics code directly on the client's graphics processor (Figure 1). This means that the entire Google Body client runs inside a web browser.

While Google Body does not concentrate on realism in visualization or performance, its key feature is in browsing through the *peel back* operation that is achieved through transparency. The browser offers on-screen slider-based controls that allow the user to select the level of transparency of the individual anatomy systems. The two modes of control for this collection of sliders are simply called *one slider* (Figure 2) and *many sliders* (Figure 3).

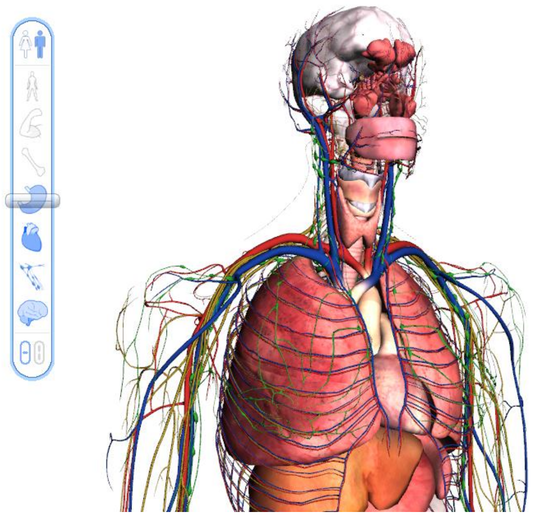


Figure 2: Google Body *one slider* control [2]

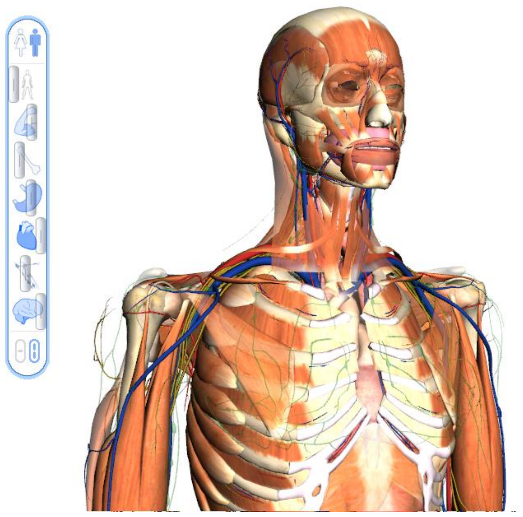


Figure 3: Google Body *many sliders* control [2]

The *one slider* control allows the user to navigate through the layers of anatomy systems. This is done by sliding the transparency control starting from the inner most parts (organs) all the way up to the outer most system (skin). The *many sliders* control allows the user to set the transparency of each anatomy system directly. As the user progresses through the slider in either mode, various parts of each system are automatically removed once their alpha[5] reaches near full transparency.

Google Body's approach to browsing the 3D dataset is limited. While the use of transparency has been shown to be effective under numerous conditions [7] [8], it has the inherit nature of making complex images far too noisy to properly retain their visual context, especially if more than one layer is partially transparent. In the case of the complex shapes and complex spatial interrelationships inherent in human anatomy, transparency alone severely limits the navigation of Google Body. *Globally* reducing the alpha of an anatomy system does not guarantee the contextual exposure of the *local* system region/part. The browser does not give the user direct control over what parts the user can fade away using transparency, let alone the granularity of such control. The browser itself is a web application and is not offered with an API (Application Platform Interface) or extendibility capabilities of any kind, making changes to the above limitations impossible.

VOXEL-MAN

Voxel-Man [9] takes another approach to human anatomy visualization and interactivity. Voxel-Man is the general name given to numerous computer programs that use 3D volume models of human anatomy derived from CT (Computed Tomography) scans, MRI (Magnetic Resonance Imaging) scans or photography. The purposes of these individual programs include dental trainers [10], anatomical atlases [11] and surgery simulators [12].

Voxel-Man uses volumetric rendering of 3D volume images as opposed to surface rendering of 3D model meshes. Surface meshes consist of connected triangles and a triangle is the

[5]*Alpha* is the numerical value of a transparency parameter, ranging between 0% (transparent) and 100% (opaque), denoted by 0.0 and 1.0.

fundamental rendering element. Computer graphics hardware has evolved to optimize the rendering of triangles. Volume images, on the other hand, can be thought of as a stack of images and the individual rendering element is a voxel, which can be thought of as a tiny cube or 3D pixel. In a nutshell, volume rendering can be visualized as the “firing” or casting of rays from each pixel on a screen window through the volume image. Where a ray intersects voxels, a cumulative color and opacity value is calculated based on voxel intensity values and user-settable mapping functions. This color and opacity value is used to set the ray’s origin pixel on the screen. Volume rendering (Figure 4) allows both the interior material of an anatomical structure and the surface boundary between structures to be shaded [13].

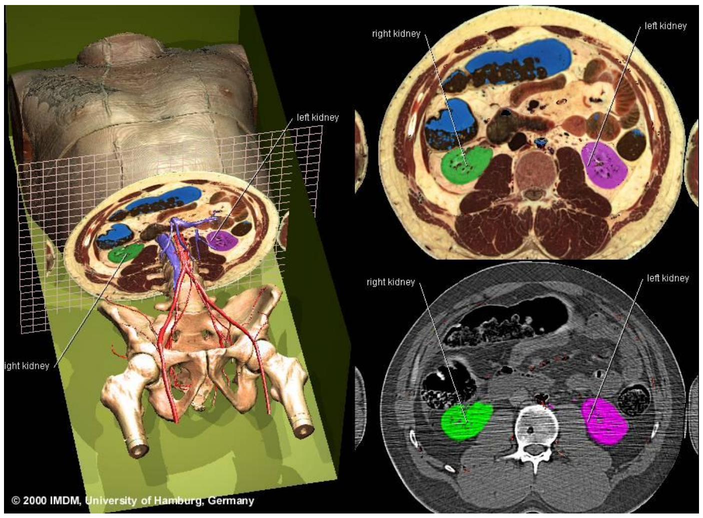


Figure 4: Volumetric rendering in Voxel-Man [14]

The Voxel-Man 3D-Navigator series is a collection of anatomical atlases that can be used to interactively explore inner organs, brain and skull and the upper limb [15]. These systems are aimed at trainee radiologists and allow for the interactive removal of entire anatomy objects. Voxel-man’s volumetric rendering capabilities allow for realistic visualizations. However, volume rendering is well-known to be much more computationally expensive than surface rendering and requires either specialized graphics hardware or GPU acceleration. Voxel-man's programs are not extendible.

INTERACTIVE CUTAWAY SYSTEM

Li et al [16] presented a fairly well balanced approach to general interactive visualization that has been applied to human anatomy data. Similar to ViC, this approach uses a two part system including a preprocessing component that is used to "rig" a 3D dataset in order to constrain interactive cutaway operations on it, and an interactive visualization component. This approach allows meaningful and contextual interactive cutaway operations to be performed on the dataset without any specialized interfaces or pre-set scenarios. The system also uses Non-Photorealistic (NPR [17], Figure 5) shading based conventions in order to further visually augment the cutaway operations. For example, *edge shading* is used to provide a visual cue about the surface orientation.

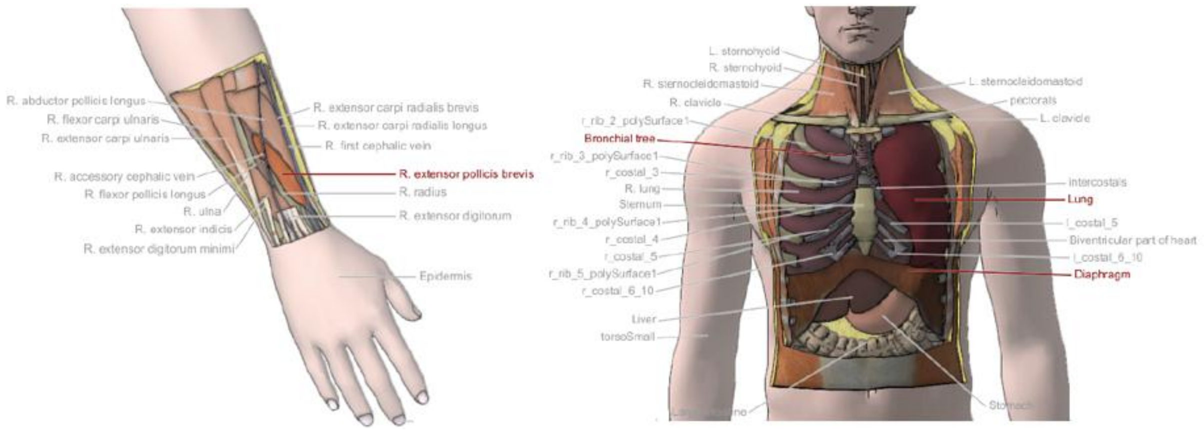


Figure 5: Non-Photorealistic cutaway illustrations of human anatomy [16]

The system uses the GPU-accelerated Open CSG package [18] to support cutaway operations. Constructive solid geometry (CSG) implements Boolean operations, such as union and intersection, on two meshes. While this package supports efficient and robust cutaway operations, the disadvantage of using this package is the inability to photo-realistically shade the output mesh. The reliance on non-photorealistic visualization makes the system better suited for illustration-based visualization that is commonly found in an anatomy textbook. While NPR may be advantageous for creating a simplified and understandable visualization, a true virtual cadaver system requires photorealism to better mimic the dissection of a real cadaver. Furthermore, since ViC is not dependent on an external package such as Open CSG, ViC's rendering style is changeable and can readily accommodate NPR techniques.

Finally, despite powerful interaction capabilities, this system does not introduce any formal extendibility capabilities that can be used to extend either of the two system components. The reliance on Open CSG also restricts extendibility. For example, it may not be possible to add a non-rigid physics component to the system to allow for elastic tissue simulation. ViC, on the other hand, was designed to accommodate additional custom authored components or existing components, such as a physics engine.

CYBER-ANATOMY

Cyber-Anatomy Med [19] is a real-time 3D anatomy explorer for education and illustrative purposes. It is aimed at both professionals as well as medical and even high-school students. In addition to standard navigation operations such as zooming, rotation and emphasis/de-emphasis, it supports interactive *peeling* of muscles (Figure 6).

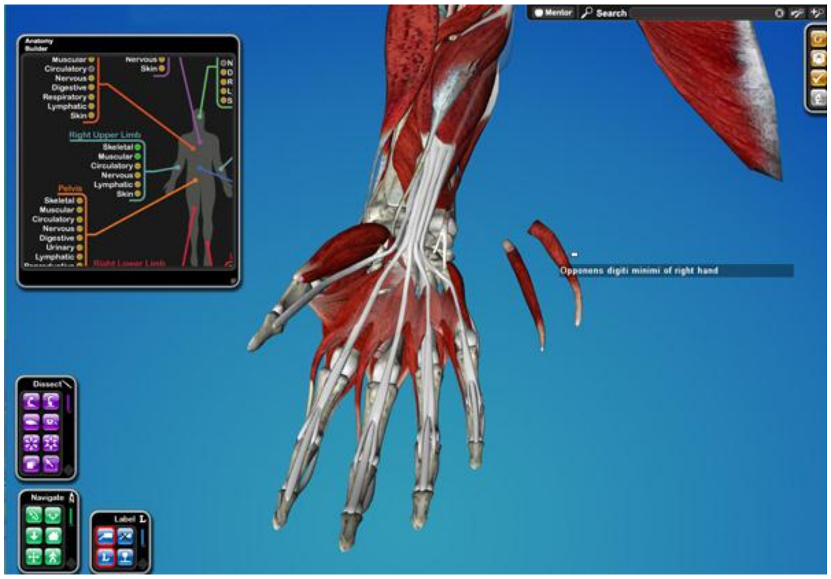


Figure 6: Interaction in Cyber Anatomy [19]

The peeling operation allows for the removal of layers such as individual muscle objects one by one. Upon activation of the operation, the user can select a muscle to remove and slide it anywhere within the 3D model space. The direction of removal of the peeled muscle depends on the angle view of the camera. The option of resetting the position of the muscles automatically is available to return them to their original location.

While the peeling away of individual muscles is useful for exposing other occluded anatomical systems and structures, Cyber Anatomy Med provides this operation only for the muscular system and not for other anatomy systems. This restriction limits the ability to build fully meaningful contextual views. In addition, removal of an entire muscle object is also restrictive. It may often be desirable to peel or cut away only a part of a muscle, or a part or region of other types of anatomy, to better retain visualization context. The science and art of anatomy illustration is very old and illustrators have discovered optimal ways of visualizing anatomy objects, including the liberal use of partial cutaway of objects.

2.2 Candidate Software Platforms

This section is a survey of current mainstream software platforms that were considered as the foundation of ViC. In order to achieve the goals outlined in the contributions (Section 1.1), candidacy for the foundation of ViC was constrained to the following requirements:

- The platform must provide facilities for the representation and manipulation of 3D data. Whether by established standards and/or technologies or by exclusively internal capabilities, the technology must provide extensibility, to allow the end-user to customize the parameters of the visualization and perform meaningful operations on it. The most important operation is algorithmic, real-time, and/or procedural manipulation of 3D data.
- The platform must be affordable by the general public. More specifically, the cost (if any) must be within a reasonable level (i.e. not requiring government/enterprise level funding).
- The platform must be compatible with mainstream hardware. Mainstream hardware is defined as a standard Intel-based x86 processor[6] and a mainstream or workstation-class graphics processor unit. Hardware should not require any additional interface software aside from the manufacturer supplied drivers, which should be included in the hardware cost.
- The platform must be readily available. The product must not be exclusive or require special status – in any context – to obtain. Any individual of the scientific research, medical field or the general public should be able to obtain the technology.

[6] Despite their affordability, ARM-based SoC (System On Chip) solutions were not considered on the grounds that as of the time of writing they still did not provide sufficient capabilities for running ViC.

Since the documented platforms were originally designed for a specific purpose, they are placed into common categories. The four main categories are:

- **Game Engines**: Systems that were originally designed to facilitate the creation and development of video games.
- **Frameworks** (in some cases Toolkits): Systems that were originally designed to provide (or automate) common low-level/boilerplate functionality and provide a high-level interface to utilize this functionality.
- **Graphics Modeling Packages**: Packages that were originally designed to generate, manipulate, animate, deform and render complex geometric models.
- **Packages**: Ready-to-use systems/applications that offer sufficient visualization and data manipulation capabilities, typically via a graphical user interface and/or scripting language.

Some platforms have either a relationship to, or dependency on, other platforms that are described. In such cases, the platform was an eligible candidate only if the dependency did not incur additional cost and/or unobtainable requirements.

The documented platforms in each category were chosen such that they are representative of that category. That is, other, often numerous, platforms within the same category share similar features and capabilities and were therefore omitted.

2.2.1 Game Engines

UNREAL ENGINE

The Unreal Engine [20] made its debut in 1998 in the first-person shooter (FPS) game titled *Unreal*. Although originally designed as an FPS engine, throughout the years Unreal Engine was used in other game genres ranging from action-adventure to MMORPGs (Massively multiplayer online role-playing games). Figure 7 showcases rendering capabilities of the engine. The engine was used for the development of the game *America's Army*, which the United States military admits, serves as a cost effective recruitment tool and as a propaganda device [21].

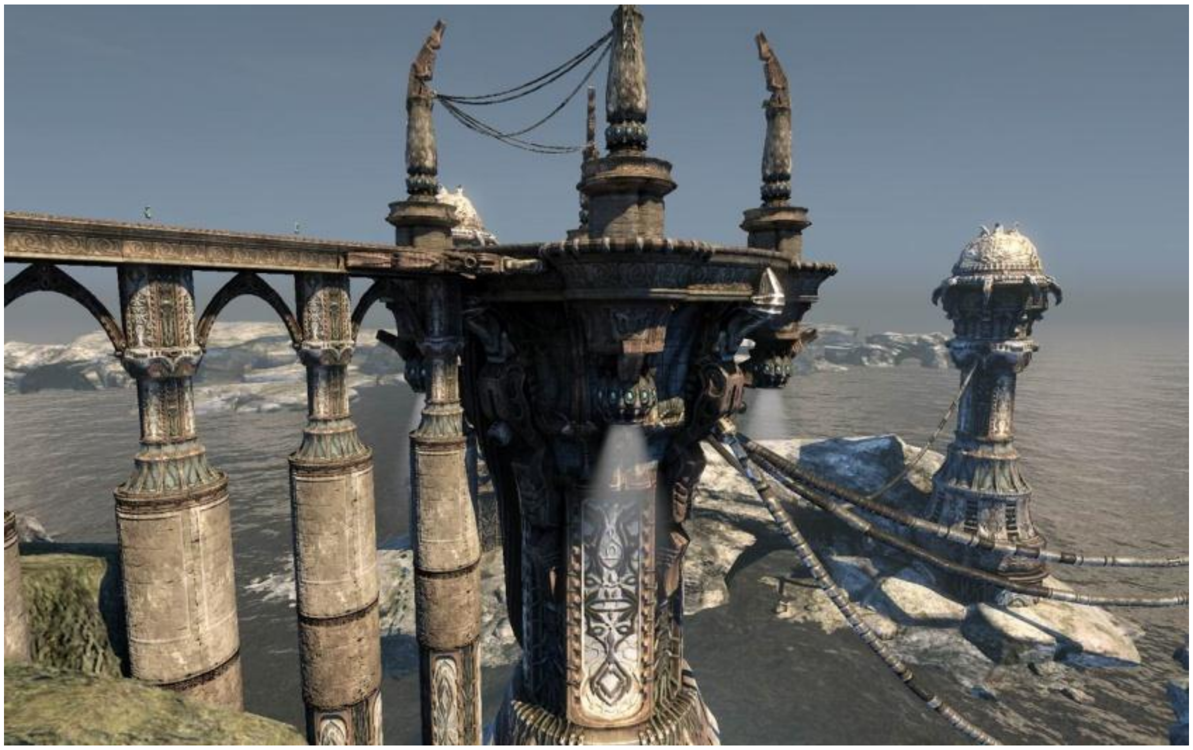


Figure 7: Unreal Engine 3 scene [20]

The Unreal Engine is also being used in the Serious Games[7] realm. Digital content creator Virtual Heroes developed an Emergency Medical Services (EMS) training simulator *Zero Hour: America's Medic* using Unreal Engine 3 [22]. In recent years, Unreal Engine's purpose went

[7] A *serious game* is a computer/video game that serves any purpose other than entertainment. The primary purpose is often education. ViC can be classified as a serious game.

beyond gaming. A children's television show *LazyTown* uses the engine as a cost-effective computer-generated imagery (CGI) tool.

The Unreal Development Kit (UDK) is the third and current iteration of the engine. Engine creator Epic Games brands UDK as a complete professional development framework that includes all the tools required to create games and other advanced visualizations and/or detailed 3D simulations.

The Unreal Engine is written in C++, making it portable across various desktop platforms including Windows, Linux and Mac OS. The third iteration (Unreal Engine 3) is designed around DirectX[8] for Microsoft Windows. It is one of few game engines in the industry to support both 32-bit and 64-bit systems. Currently, UDK is available for Windows only.

Although the engine itself is written in C++, much of the application code can be written in UnrealScript, C++ and Java based scripting language, used to author application flow and events. Use of the UDK for non-commercial purposes is free of charge. Commercial usage is subject to fees and royalties depending on profits and nature of business at Epic Games' discretion. UDK is closed source.

CryEngine

CryEngine (CryENGINE) [23] was originally developed as a technology showcase for GPU manufacturer NVIDIA, but was later used for the first-person shooter title *FarCry*. CryEngine 2 was used in the award winning title *Crysis*. CryEngine 3 features include multi-core CPU

[8] *DirectX* is a set of media API that include functionality for graphics, audio, networking, input and more.

support, deferred lighting[9], soft body physics and more. Figure 8 showcases the combination of the many advanced features used to achieve high level of realism with the engine.

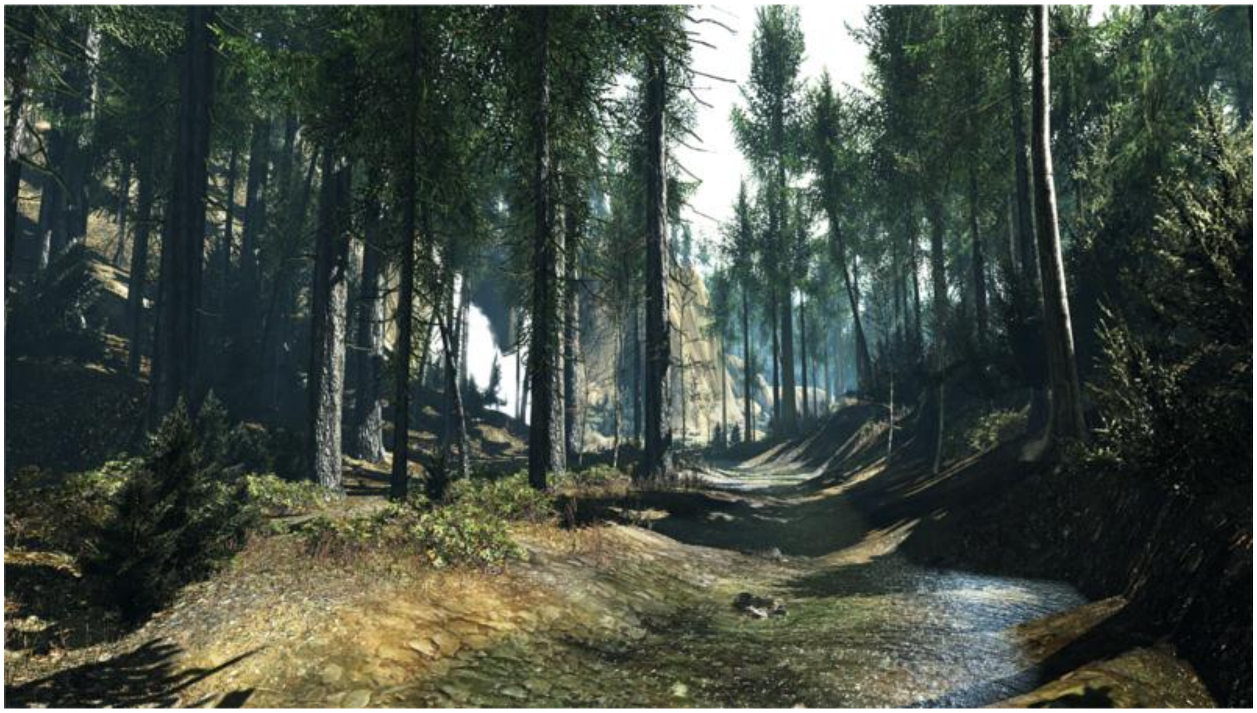


Figure 8: An example CryEngine 3 scene [23]

The engine is available for Windows supporting both DirectX 9 and DirectX 10. The engine developer (Crytek) advertises the engine as a tool for game development, education, simulation and visualization, with each also determining the engine license type. While educational licensing is free, most applications fall under a commercial license. Medical imaging, for instance, is categorized as "Simulation". Pricing depends on the nature of engine application and is determined by Crytek. Binary (closed source) and Source (open source) are both available as separate license types.

OGRE/AXIOM

[9] A rendering technique where the lighting information is applied after the scene is rendered in order to increase performance.

OGRE (Object-Oriented Graphics Rendering Engine) [24] is a 3D graphics rendering engine which was conceived in 1999 and finally released in 2005. OGRE is a scene-oriented engine used to provide general solutions for graphics rendering. OGRE is used only for rendering purposes since it does not provide support for audio, input, physics and other libraries thus giving the developer freedom to choose their own supporting libraries. OGRE is designed in C++ while Axiom is a C# port. They are both open source and together cover Windows, Linux and Mac OS X. OGRE has an Object-Oriented interface which follows a plug-in architecture, thus minimizing the cost of rendering 3D scenes. Its support for both Direct3D (Microsoft's proprietary graphics API) and OpenGL (an open standard cross-platform graphics API) makes it independent of any 3D implementation. In Windows it builds on Visual C++ [25] and Code Blocks [26], in Linux it uses GCC 3+ and XCode in Mac OS X.

Some of the key features of OGRE are the automated tools for spatial culling (determining what's visible in order to increase rendering efficiency), transparency and render state (the state in which the graphics card should be in to perform a specific rendering operation) management. It supports shaders with vertex and fragment programs written in GLSL (OpenGL Shading Language used to write code for the graphics processor in OpenGL), HLSL (High-Level Shader Language used for graphics processor code with DirectX), and other languages. Some other notable operations include multi-pass blending, texture coordinate generation and modification and multi texturing. Developers may design with multiple material techniques having alternative effects as OGRE automatically uses the best one supported.

OGRE accepts several data types and provides a sophisticated skeleton animation support. A compositing manager, complete with a scripting language and post-processing interface for numerous post-processing effects, is also included. Interoperability of OGRE is extended with the availability of exporting tools for 3D modeling tools including 3Ds Max, Maya and Blender amongst others.

Axiom, which is written in C#, is often referred to as the rendering middleware due to its flexibility. It also supports both OpenGL and DirectX with tools for shader editing and other rendering tools featured in OGRE. Figure 9 shows a test scene from the ORGE demo collection.

Figure 9: A test scene from the OGRE demo collection [24]

OGRE and AXIOM are both open source available under the MIT license free of cost.

2.2.2 Frameworks and Toolkits

XNA

Microsoft XNA [5] is a framework consisting of a set of tools with a managed runtime environment that facilitates primarily computer game development. The post-preview build was released to the public at the end of 2007. XNA's primary goals are the reduction of repetitive boilerplate code as well as the merger of various aspects of game development into a single system that can be used on multiple platforms.

XNA was introduced in 2006 as a high level framework of the DirectX API. XNA uses the .NET framework[10] and the Compact .NET framework (a smaller, less functional version of the .NET framework) on the Xbox 360 and Windows Phone 7 devices.

Despite the fact that XNA is aimed at game development, the framework does not contain any game-specific components such as level editors, scripting or pre-built content editing, and is therefore not branded as a game engine. Instead, it provides a vast high-level wrapper for DirectX, allowing for visualization and interaction capabilities that are not exclusive to gaming (e.g. Volumetric Rendering (Figure 10)).

[10] The *.NET framework* is a large general software framework for the Windows operating system that encompasses functionality for user interface, web development, database connectivity, networking, algorithms and more.

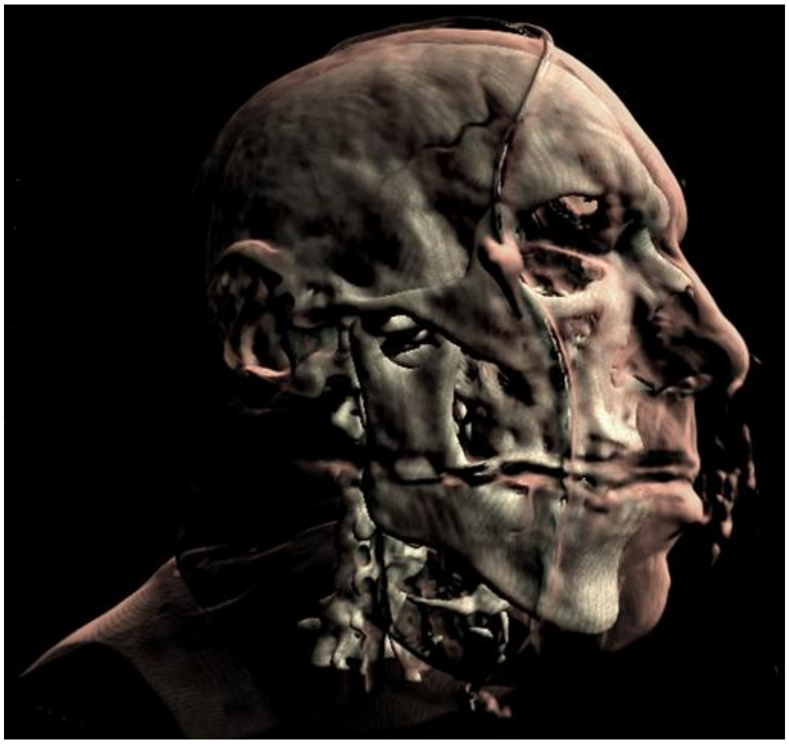


Figure 10: Volumetric Rendering in XNA [27]

XNA framework runs entirely in a managed environment. It supports C# as the principal programming language, and High-Level Shader Language (HLSL) for graphics code.

XNA requires DirectX version 9.0c and fully supports Shader Model version 3.0. XNA development binaries, toolset, and redistributables are free of charge. XNA is officially closed source, although in some circumstances some of its code is released to the public for educational purposes only.

VTK

Visualization Toolkit (VTK) [28] is a free, open-source 3D system aimed at 3D visualization and image processing. Internally, VTK is a C++ class library as well as a collection of interface layers. It was created in 1993 as complementary software to the book "The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics" [29]. It is fairly exclusive to scientific and medical applications and is neither branded nor used as a general visualization platform supporting other applications such as computer gaming or cinematic rendering. However, given its capabilities and extensibility it could, in theory, be used for any visualization application. Figure 11 show an example of VTK's rendering capabilities.

VTK is the core of numerous medical visualization and/or interaction applications including Slicer and ParaView (which was created by Kitware, the developer of VTK).

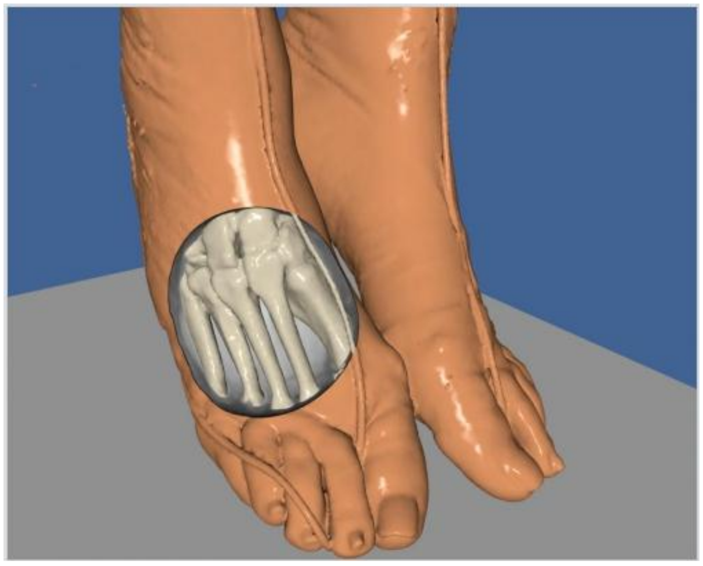


Figure 11: Original visualization author Bill Lorensen [30]

ParaView can be described as the front-end of VTK, where VTK is a collection of classes that facilitates the development of visualization applications and ParaView is an example of such implementation. ParaView also supports client–server architecture to facilitate remote visualization of central data.

VTK runs on Linux, Windows, Mac and Unix-based platforms. VTK consists of nearly 1000 C++ classes and over 1,000,000 lines of C++ code. Surface and volume rendering are both supported by VTK. VTK is open source and is free of charge regardless of usage.

2.2.3 Graphics Modeling Tools

BLENDER

Blender [31] is a high-end 3D tool used for modeling, shading, animation, rendering, imaging and real time 3D game creation. Its applications involve visual effects for animated films and video games. Blender includes extensive and advanced simulation tools with its consistent and flexible GUI with scripting available in the Python language.

Blender was authored and developed by a company founded by Ton Roosendal in June 1998. Since then Blender has exchanged a few creditors and is now developed under the supervision of *The Blender Foundation*, making it a well-established product.

Blender runs on Windows (2000, XP, Vista), Mac OS X, Linux, Sun Solaris, FreeBSD and SGI Irix 6.5. Unofficial ports are also available for AmigaOS 4, BeOS, MorphOS, Pocket PC and SkyOS. It is written primarily in C, C++ and Python. One of the key features of Blender is its support of Python scripting in the user interface, allowing custom tools, prototyping and tool creation.

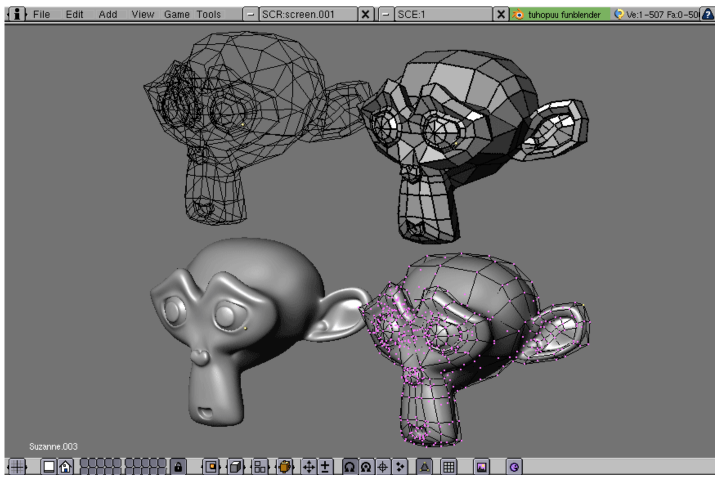


Figure 12: 3D modeling in Blender using the node compositor [31]

Its rendering capabilities expand its usability and versatility owing to a fast built-in ray tracer[11]. It allows layering (separation of objects; Figure 12) and passes along with object-to-object baking (merging two mesh objects). Various texture maps are supported, including animated mapping (which animate the surface of the object) and reflection mapping (which map a reflection onto the object).

Game Blender, a free and open sourced game engine is a component of Blender, which offers interoperable features such as collision detection, dynamics engine, and programmable logic. It may also be used as a base for independent, real-time applications ranging from architectural visualization suites to video game construction.

Blender is free and open source, available under the GNU General Public License. Blender has simple requirements with a small installation kit despite its key-characteristics of high-end modeling and imaging. This is in contrast with major modeling tools such as 3ds Max, which has a *significantly* larger size and system requirements.

3DS MAX

3ds Max [32] is a design and modeling tool used to provide powerful 3D visualization, animation and rendering. It supports only Windows (XP, Vista, 7) and is based on plug-in architecture making it extendible. Its built-in scripting language, *MAXScript*, can be used for a variety of purposes including the development of new tools, interfaces, plug-in modules and combining existing functionalities.

3ds Max is a highly interoperable tool and having its own scripting language allows it to create custom import/export tools, write procedural controllers and build batch-processing tools

[11] *Ray tracing* is a technique that traces the path of light with the screen pixel as the origin. The resultant image is the effect caused by encountering the destination (the 3D object).

(automated tools), thus making it flexible and extensible. Aside from using MAXScript, it may also be extended and customized through its SDK.

Figure 13 shows the modeling results along with various texture maps.

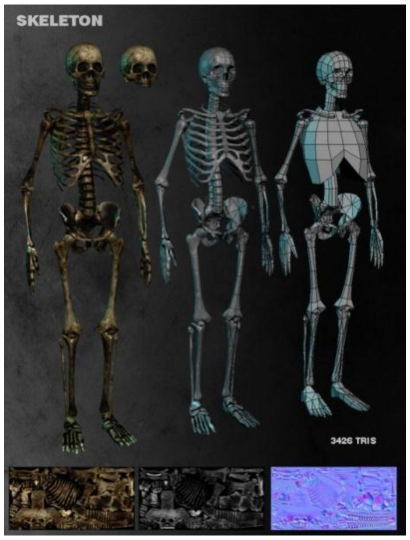


Figure 13: Biped skeletons modeled using 3Ds Max [33]

Inverse Kinematics (skeletons) tools are used to animate characters which include expressions, scripts, wiring and list controllers. It features four plug-in tools namely, history dependent solver, history independent solver, spline IK solver and limb solver, each of which aid in smooth blending, better control and time efficiency. *Character Studio* is a key plug-in tool used for human animation. The *XView Mesh Analyzer* aids polygonal manipulation helping optimize poly models, topology, loop tools. Freeform capabilities allow users to create or modify meshes with complete freedom.

3Ds Max is distributed under a propriety license and is priced at several thousand dollars for commercial use and several hundred for academic use.

2.2.4 Visualization Packages

MATLAB

MATLAB (MATrix LABoratory) [34] is a technical computing language with an interactive environment which is used for data visualization and mathematical models. MATLAB is a proprietary product of MathWorks, which was created by Cleve Moler in the 1970s.

MATLAB has varying capabilities in 3D visualization ranging from simple mesh creation, transparency mapping, volume rendering, multifaceted viewing and more. MATLAB's interpretation of data is formatted as matrices and vectors, which could define surface plots, grid plots and mesh data.

MATLAB interfacing can be done in programming languages such as C, C++ and FORTRAN although programming in MATLAB is primarily done in MATLAB code (m-code). MATLAB runs on Windows (XP, Vista, 7), Linux and Mac OS X.

Figure 14: Newell Teapot demonstrating MATLAB’s visualization capabilities [34]

MATLAB’s 3D visualization capabilities (Figure 14), although not fast or advanced, provide a flexible and easy to use interface for fairly large amounts of data. Most notably, MATLAB’s integration with VTK gives it the power of VTK’s visualization without the need for explicit conversation of data from its native format to VTK’s format. matVTK, a name given to the framework integrating VTK and MATLAB, is an open source VTK wrapper which provides the functionality of VTK directly in MATLAB’s toolbox set. The data flow design of matVTK follows a *pipe and filter* architecture which helps compute multiple steps at once and time stamps them to allow for faster re-computation. MATLAB is extended using a C, C++ and Fortran API for MATLAB.

The matVTK framework is based on three major building-blocks namely, the *Pipeline Handle*, the *configuration interface* and the *graphics primitives*. The Pipeline Handle is a VTK handle used to keep track of the data sets used in a current scene. These may be used implicitly or explicitly for multiple scenes. The configuration interface provides a tight knit control over parameters such as filters, scene components and global settings. Graphics primitives refer to MATLAB's plotting functions on various VTK primitives.

IMAGEVIS3D

ImageVis3D [35] is a volume rendering application developed by the NIH/NCRR Center for Integrative Biomedical Computing (CIBC). ImageVis3D has extensive volume visualization capabilities featuring slice-based and GPU ray casting tools to visualize 3D data. ImageVis3D was developed and initiated in 2007 and is publicly released under the MIT license. ImageVis3D is currently funded and accessible through the Scientific Computing and Imaging Institute (SCI) at the University of Utah.

ImageVis3D runs on Windows (XP, Vista, 7), Linux and Mac OS X. It is written in C++, and wraps both OpenGL and DirectX. Various ImageVis3D tools provide flexibility and extensibility in importing/exporting data. Multiple modules allow developers to build on them separately. The scalability of the application is extended owing to its ability to run on relatively old hardware.

Along with Voxel-Man, ImageVis3D supports stereo rendering (stereoscopic 3D effects) through anaglyph imaging[12]. It also has a fairly unique high-level feature, *ClearView*, that is designed to explore specific areas in the data while preserving context information and removing visual clutter at the same time.

The scalability in ImageVis3D comes from its ability to function out of core thus allowing it to perform not only on the newest generation of graphics but also on common or older hardware despite supporting fairly large data sets. The component design allows developers to not only

[12] A technique where the rendered frame contains encoded 3D (stereoscopic) information that is viewable through red/cyan glasses.

extend it but also reuse parts of it such including the kernel. ImageVis3D is open-source and is free of charge.

Chapter 3: Methodology and Implementation

The second section of the previous chapter lists numerous visualization platforms that could potentially serve as the foundation of ViC. Although many of the candidate platforms excel in one or more of the requirements documented in Section 2.2, very few offer a balance between all of the requirements. The non-managed game engine kits from the ***Game Engines*** category were ruled out primarily due to questionable licensing costs and terms and notoriously complex development environments with a steep learning curve. Although they presented all of the technical capabilities for implementing a virtual cadaver, the aforementioned problems completely outweighed the balance of all requirements. ***Graphics Modeling Tools***, despite their very nature of geometry creation, manipulation and visualization, do not provide the out of the box facilities for creating interactive, real-time visualizations. They are primarily targeted toward digital artists. Many ***Packages*** also suffered from limited support for complex interaction customization. MATLAB, for instance, offers only primitive 3D mesh visualization techniques that are nowhere close to the realism required by a virtual cadaver.

In consideration for the requirements in Section 2.2 with particular focus on performance, cost, feature set and support for longevity with respect to not only *developing* ViC, but also to *extending* and *running* it, the list was narrowed down to three candidates: Axiom, VTK and XNA. The XNA framework was ultimately chosen. XNA is an unrestricted framework for the creation of 3D visualizations with support for advanced visualization and interaction. XNA and its development environment as well as distribution binaries are free of charge and are publicly available. XNA-based applications run on a standard x86 processor and a mainstream graphics card that supports Shader Model 2.0, making it backwards compatible with low-end hardware.

Furthermore, XNA offers a high-level *managed* solution to building DirectX-based real-time visualizations. With C# as the principal development language and .NET as the development environment, XNA automatically inherits the many advantages of managed code execution [36]. XNA has been optimized well enough to run even on less powerful embedded .NET Compact Framework based devices such as ARM-based mobile devices. It is also compliant with DirectX's shader language, HLSL[13], allowing XNA to take advantage of any DirectX graphics code.

The remainder of this chapter describes ViC's implementation under the XNA framework. The usage, capabilities and extendibility of ViC are also discussed.

[13] High Level Shader Language is a C-like language that is used to write graphics programs (shaders) that run on the graphics processor as opposed to the CPU.

3.1 An XNA-based System

Microsoft XNA is a high-level game development framework that wraps the DirectX API and uses the Microsoft .NET framework. It sits between the XNA-based system (for example, ViC) or application and the DirectX API in the execution hierarchy (Figure 15).

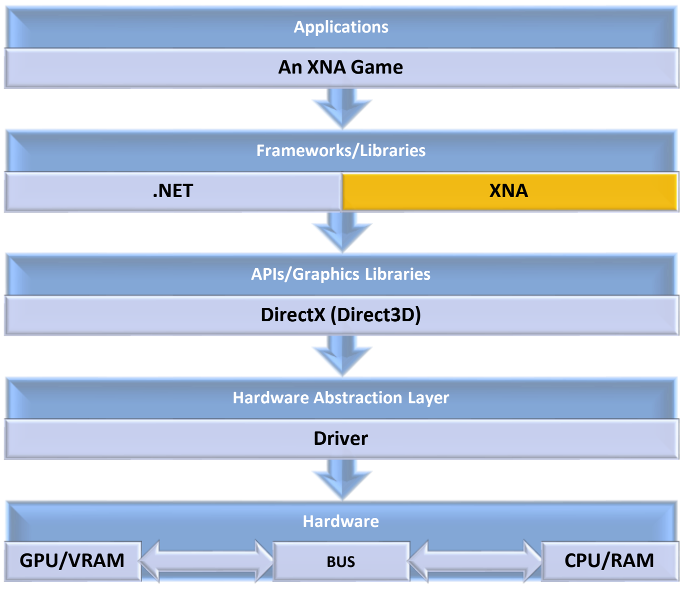


Figure 15: XNA's placement in the execution hierarchy of a game application or system

XNA itself can be broken down into two fundamental parts, each with specific roles:

Content Pipeline

XNA's content pipeline is used to import and process numerous types of data including 3D models, images and audio. Multiple standard data formats are supported, but the pipeline is fully extendible to support any standard or custom format. The data that goes through the pipeline is ultimately transformed (processed) into XNA's native format (.XNB) for runtime "consumption". The pipeline code is executed at build-time, which takes place when the project is built prior to runtime.

LOADING AND GAME LOOP

This part begins with game logic initialization and the loading of content assets[14], followed by the continuous execution of two key methods – *Update* and *Draw*. The looped execution of the Update and Draw methods constitutes the concept of a *game loop*.

XNA's *Game* class is the base class that defines the above execution model. ViC extends XNA's *Game* class, inheriting the aforementioned behavior (Figure 16).

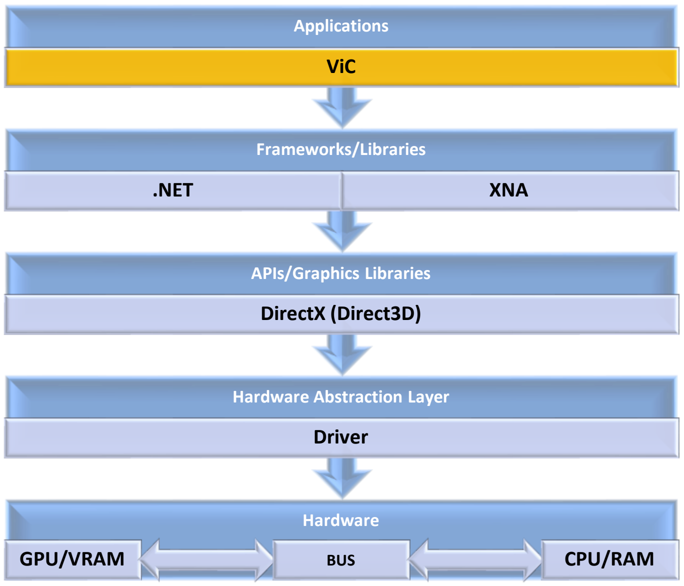


Figure 16: ViC's placement in the execution hierarchy

Figure 17 provides an abstract overview of ViC's software architecture giving a high-level snapshot of its stock functionality and capabilities.

[14] An example of a *content asset* would be the model file where an anatomy system or the complete dataset is located.

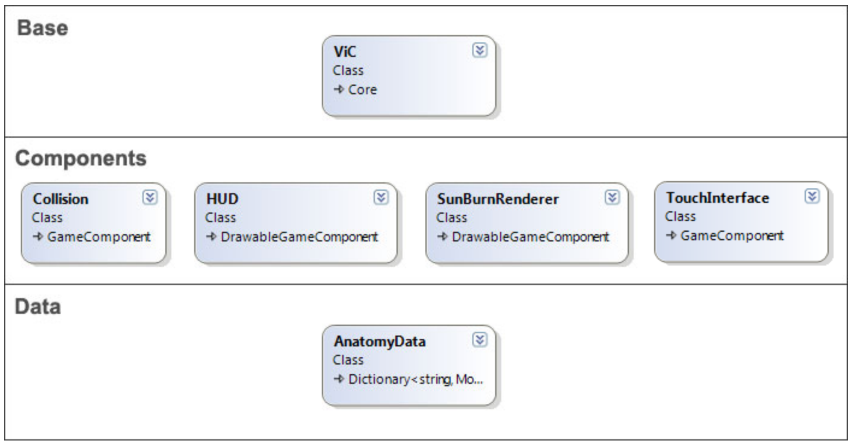


Figure 17: High-level overview of ViC.

ViC's modularity, and ultimately its extendibility, is accomplished through the inheritance of XNA's *game component* model, which is implemented through two key classes, *GameComponent* and *DrawableGameComponent*. The *GameComponent* class exposes the Update method to custom update logic, and the *DrawableGameComponent* class exposes both the Update and Draw methods to offer rendering[15]. Classes that extend either one of these component classes follow the same execution flow as XNA's base *game* class. Game components are added to a game's *component collection* and the execution of their code is managed by XNA. Furthermore, individual game components are not required to be dependent on the game that owns the component collection, making them self-contained modules. Sections 3.1.4 to 3.1.7 explain the key components of ViC. Furthermore, Section 3.3 explains how other components can be added to extend ViC's functionality.

[15] DrawableGameComponent's Update method is used to update logic that will be used for rendering in the Draw method.

3.1.1 Preprocessing ViC's dataset

Among other formats, XNA's content pipeline fully supports the .FBX standard model format and the well-known .JPG/.PNG image formats. ViC's 3D anatomy dataset [37] is kept in .FBX files along with .JPG/.PNG material files that are imported and processed for use by ViC at runtime. The .FBX files contain geometric descriptions of the anatomy meshes that make up the dataset. Geometric descriptions typically define, among other properties, the 3D vertex positions of the polygons (triangles) making up an anatomy mesh. The FBX files also contain anatomy mesh material descriptions and these descriptions typically refer to the .JPG/.PNG material files. A material description defines attributes such as mesh color and the material files contain texture images that are ultimately "wrapped" around the geometry of each mesh, supporting photorealistic rendering. The dataset meshes represent the skin, cardiovascular, circulatory, digestive, lymphatic, muscular, nervous, respiratory and skeletal systems. The total size of the dataset is approximately 5.4 million polygons that comprise a total of 190 meshes each representing an anatomy part. The meshes are complemented with 10 high resolution sets of textures that include diffuse, normal and specular UV maps sized at 4096 by 4096 pixels.

As mentioned in the introduction, to give ViC the feel of a virtual cadaver the individual anatomy meshes must be fragmented and labeled. This fragmentation is performed as a preprocessing step in a custom mesh "authoring" program. The fragmented meshes are then stored in the .FBX files.

The fragmentation layout of an anatomy object mesh depends on the natural shape of the object, the original digital artist's layout of the rectangles and triangles making up the mesh, as well as the type of the anatomy object. For example, most muscle meshes are tubular in shape and a user would most often want to cut out tubular sections from the muscle in order to expose the bone and arteries underneath. Consequently, ViC's muscle meshes (Figure 18), were fragmented into tubular sections[16] (Figure 19), allowing the user to slide a finger along the length of a muscle and remove a series of tubular fragments. Since real muscle is a solid rather than a surface, each tubular section also contains two "end caps". The skin (Figure 20) on the other hand, is a thin sheet of tissue. It was fragmented into patches (Figure 21) to more closely resemble a skin graft or surgical peeling or "unwrapping" of skin.

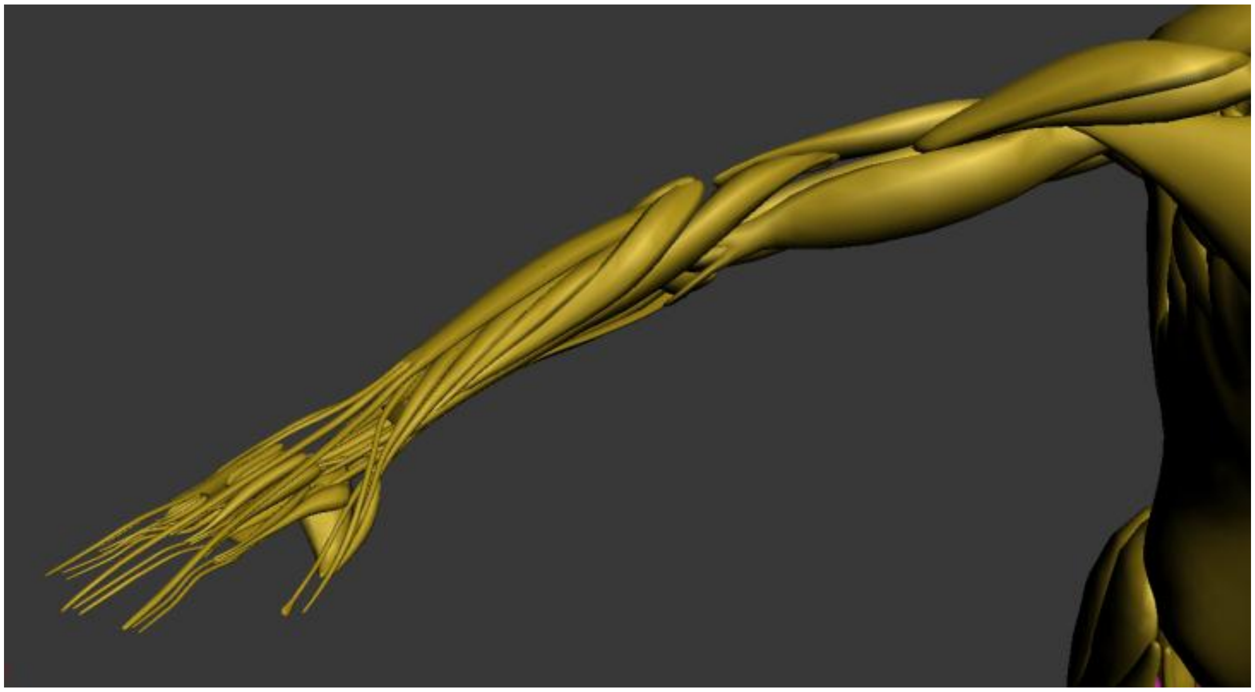


Figure 18: Model of arm muscles prior to any fragmentation

[16] Currently only the right arm has been fragmented in the prototype.

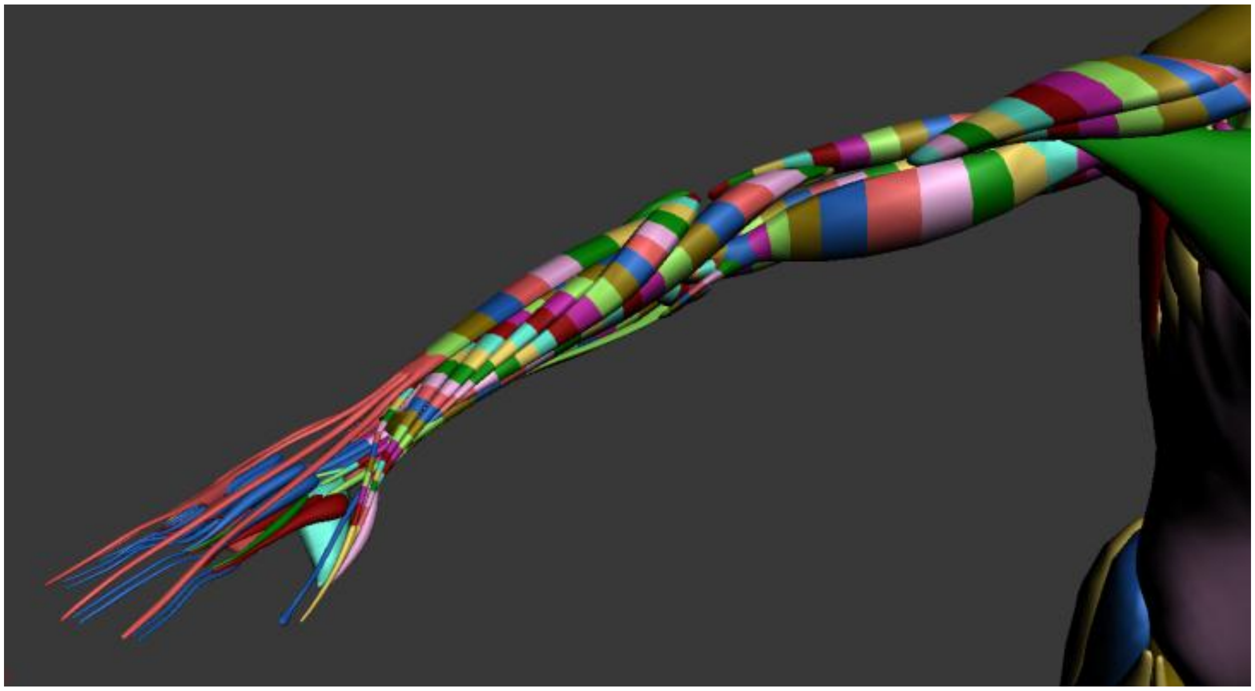


Figure 19: A color coded representation of the fragmented muscles consisting of a series of tubular fragments

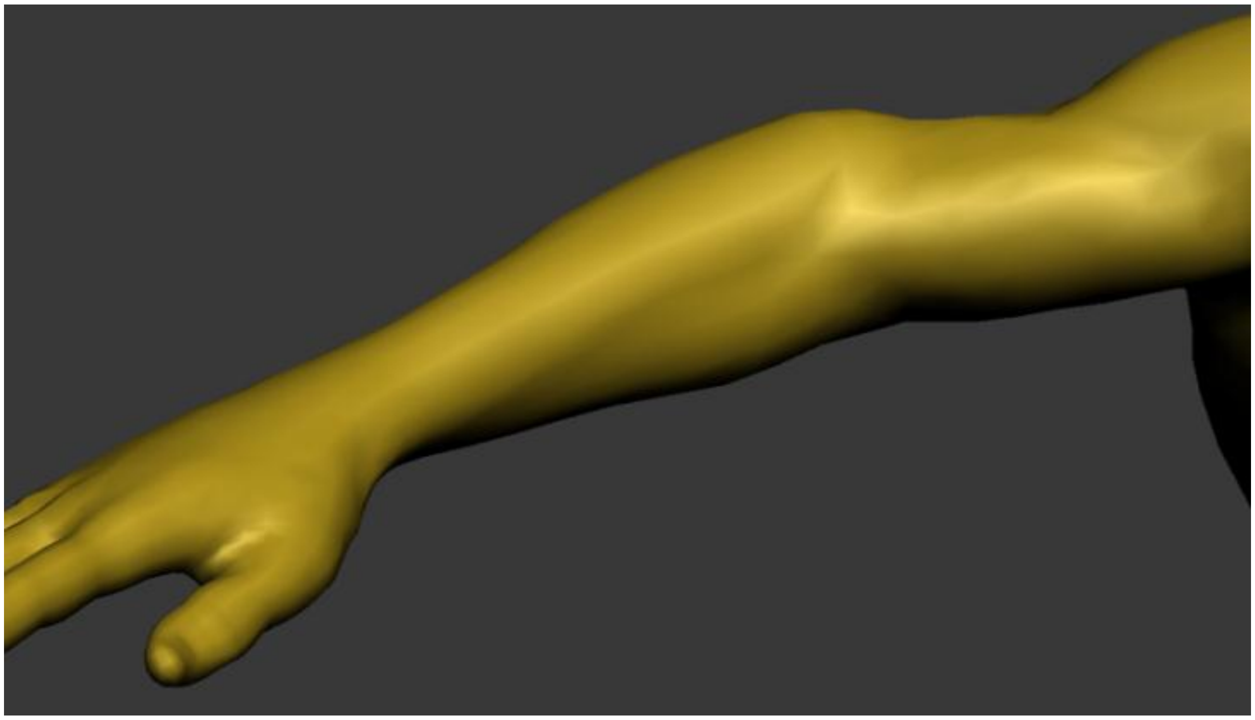


Figure 20: Model of skin of the arm prior to any fragmentation

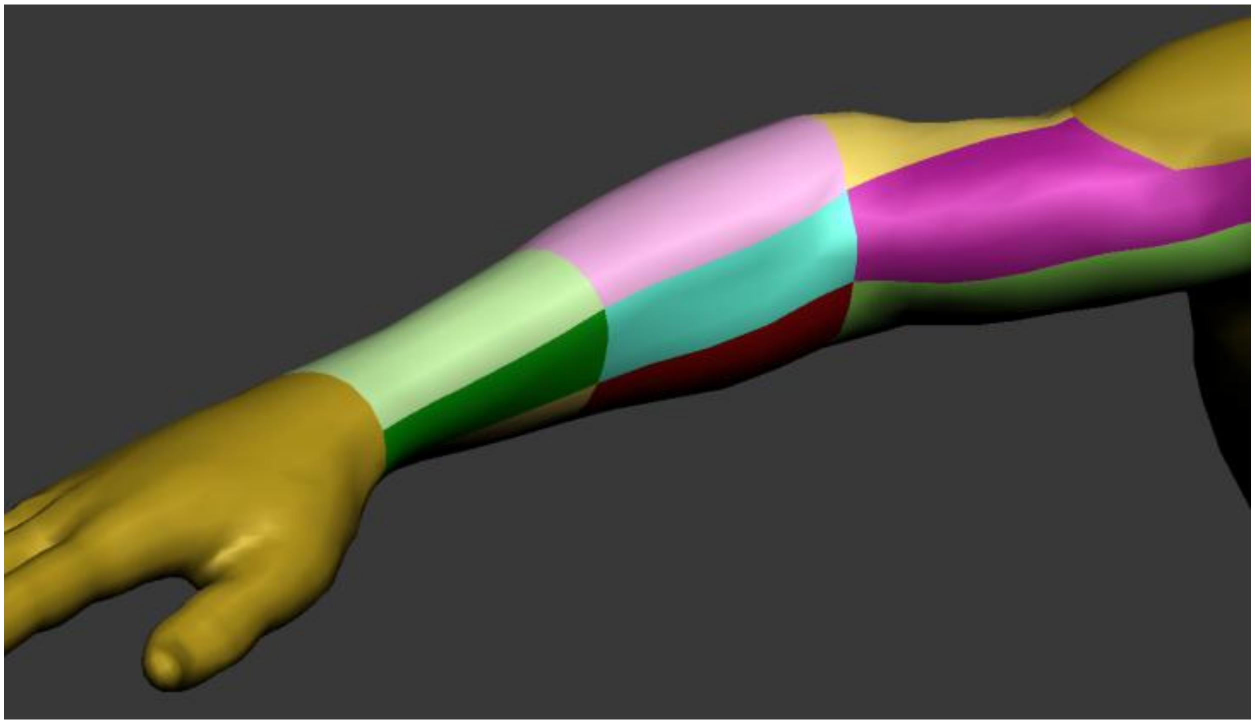


Figure 21: A color coded representation of the fragmented "patch"-like layout of skin

From a rendering perspective, labeled mesh fragments in the .FBX model file are no different than regular meshes. The sole difference is in the naming convention. This convention and how meshes that follow it become unique is explained in section 3.1.2.

3.1.2 ViC's Content Pipeline

XNA's content pipeline is a black box that takes a standardized file format, such as .FBX, and processes it into XNA's native .XNB file format. This allows the pipeline to take any standardized or custom file type and transform into a format that XNA understands. ViC inherits XNA's pipeline and its functionality. However, further customization is needed in order to support the prototype system's interaction capabilities.

As mentioned in section 3.1.1, ViC's 3D meshes are preprocessed in order to fragment them. The fragmentation layout of each anatomy object is carefully considered such that the final interactive cutaway operation is simplified, is well understood by the user, and makes sense in the context of real-world cadaver dissection. Other factors that are taken into consideration or constrain the fragmentation layout are the existing mesh rectangle layout as authored by the digital artist and the degree of "granularity" of the fragments that balances the computational and rendering expense against the ability of the user to generate a desired view of occluded anatomy.

As far as XNA is concerned, what is loaded from these models is nothing more than a collection of meshes. However, ViC takes advantage of the pipeline's extensibility to define the concept of a *mesh fragment*. ViC's content pipeline extension was customized to do two things when building and loading a dataset that contains mesh fragments:

Detect a fragment: Mesh fragmentation involves a naming convention to notify ViC's pipeline that a mesh is a fragment of a larger mesh. Two parts are added to the beginning of the name of an anatomy mesh when it is fragmented. The first is the addition of the letter 'f', which tells the pipeline that this mesh is a fragment. The second is a numerical value indicating an index into the sequence of fragments that together make up the original anatomy object.

Extraction of collision data: When a cutaway operation is executed, for example when the user sweeps a finger along the touch screen over a fragmented mesh, a collision check against the fragments is performed in order to determine which fragments the user is selecting for removal. Section 3.1.5 explains this runtime process in more detail. However, it also requires collision data to be extracted during system *build-time*. ViC's content pipeline extracts the vertices of all meshes that qualify as fragments and passes the information on to the collision component (Section 3.1.5) which ultimately performs collision checking with triangular precision.

3.1.3 Loading and Runtime

Before ViC enters the game loop, it must first go through a loading process. The loading process performs initialization of program logic, such as setting up the virtual cadaver scene and loading of the human anatomy dataset.

At this stage, the dataset has already gone through ViC's content pipeline and has been prepared for rendering and interaction. When XNA loads 3D models along with their material files, all of the data is sent directly to the GPU (Graphics Processor Unit) and a reference to that data is retained. However, many of ViC's operations require looking up meshes and mesh fragments by name, making reference-based lookup insufficient.

In order to make the lookup of anatomy data efficient for ViC's purposes, the data is not just loaded into XNA's default reference-based model containers, but is also loaded into an *AnatomyData* class. This class uses a dictionary structure[17] to store and lookup models and their meshes by name. The class itself extracts the name of each mesh and uses it as a dictionary key, while the actual data reference is submitted as a *scene entity* to the ViC's rendering component (further explained in Section 3.1.4), which becomes the dictionary value. The *AnatomyData* class provides ViC's components with the mesh data of any body part or system on demand simply by a name lookup.

In addition to initialization of the base logic and data loading, during loading ViC also adds its game components. Game components that are added to the ViC's component collection undergo the same process of initialization and loading as ViC's base class prior to entering the game loop.

[17] .NET framework's *dictionary* is a specialized data structure to keeps a collection of keys and their associated values. In order to find a value, the associated key can be looked up in the collection.

Once the components have gone through the loading process, they enter the game loop just like the base class. Game loop management of ViC's components is auto-managed by XNA.

3.1.4 The Rendering Component: SunBurn Game Engine

ViC's 3D rendering is done by an XNA-powered game engine – SunBurn. The SunBurn game engine itself is a standalone entity that is self-sufficient to be the foundation of an XNA application. For ViC's purposes, only its renderer is used as the rendering game component.

As previously mentioned, when the dataset is loaded by ViC, the 3D data is placed into the *AnatomyData* class. The *AnatomyData* class uses a dictionary with mesh names as the keys, and an instance of a SunBurn *SceneEntity*[18] class for each mesh as the value. When the SunBurn renderer enters the game loop, it iterates over all scene entities in the *AnatomyData* class and renders them with their associated materials.

In order to create the most realistic rendering of the 3D human anatomy set possible, SunBurn performs the following operations in real-time.

3.1.4.1 Maps and Materials

Human anatomy systems in the dataset that require the highest level of detail in order to look as realistic as possible are represented not only by *high poly*[19] meshes but also by supplementary UV maps[20]. These maps include diffuse lighting maps, mesh vertex normal displacement maps,

[18] SunBurn uses an instance of the *SceneEntity* class to represent each 3D scene object such as a mesh.
[19] An informal term used in the field of computer graphics to indicate that a mesh has a high level of detail achieved through a high polygon count.
[20] A UV map is 2D representation (texture) of extra detail that is applied to 3D geometry. The letters U and V refer to the coordinates on the texture, whereas X, Y and Z are used to describe the 3D vertices of the geometry.

and specular lighting maps, each adding specific details to the surface of a mesh when it is rendered.

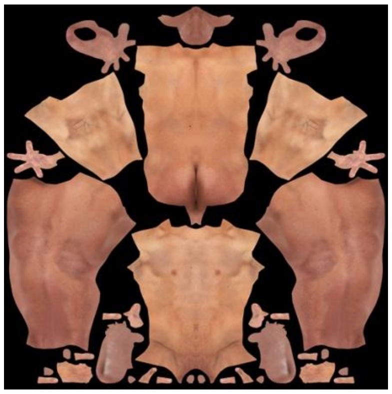


Figure 22: Diffuse map of skin [37]

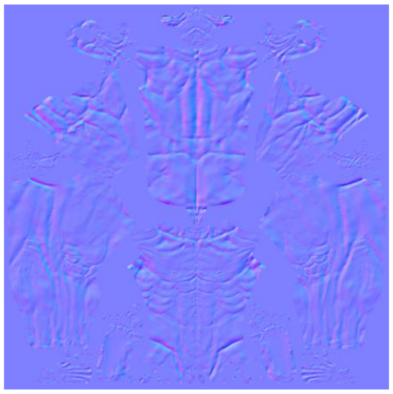


Figure 23: Normal map of skin [37]

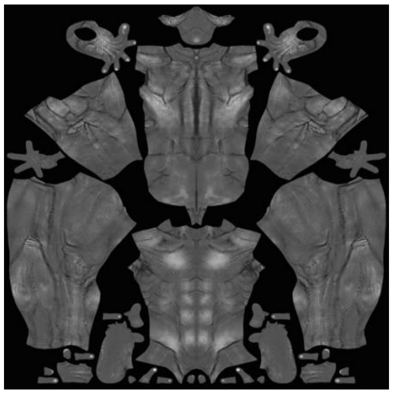


Figure 24: Specular map of skin [37]

Diffuse Map (Figure 22): This texture map provides the color representation of a 3D object. The color information of the diffuse map is constant and does not account for any lighting conditions.

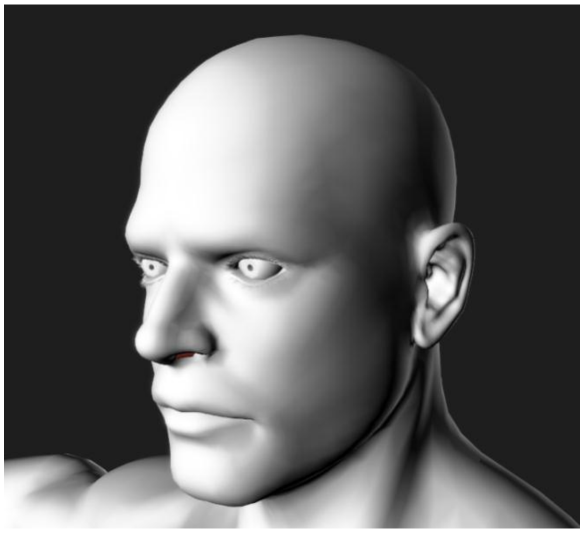
Normal Map (Figure 23): This texture map carries additional 3D coordinates for surface normals in order to create fake bumps and dents that increase the detail level of a mesh.

Specular Map (Figure 24): This texture map provides information for surface's shininess and highlight color. Higher (lighter) values in this map indicate shinier surfaces.

The SunBurn renderer applies these maps to a mesh to give it the most detailed appearance possible. The captures from Figure 25 to Figure 30 show rendering results after applying each of the UV maps to the mesh of the head:

Resultant mesh view:

UV map application:



No UV maps applied

Figure 25: The head mesh without any UV maps

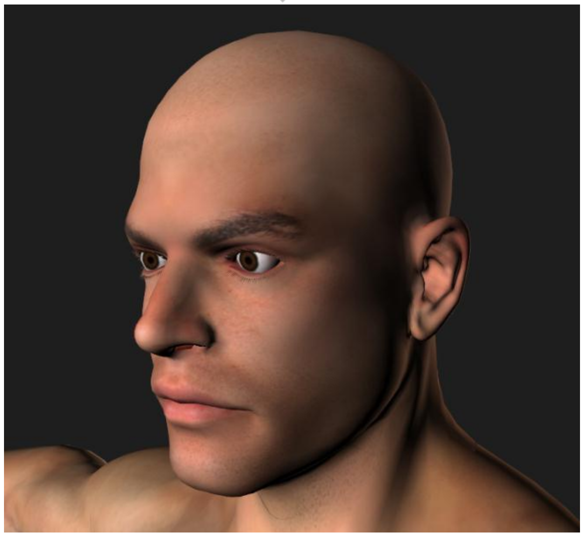


Figure 26: The head mesh with the diffuse map applied

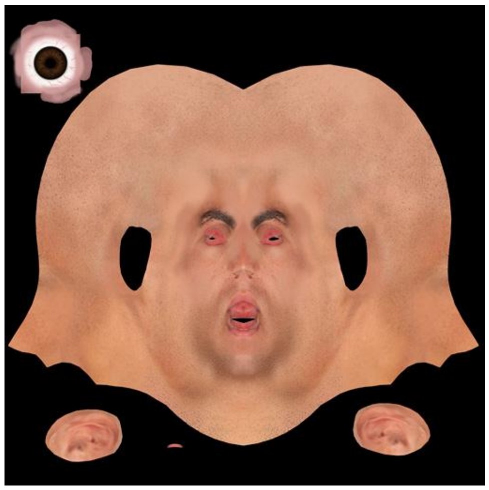


Figure 27: The diffuse map of the head is applied

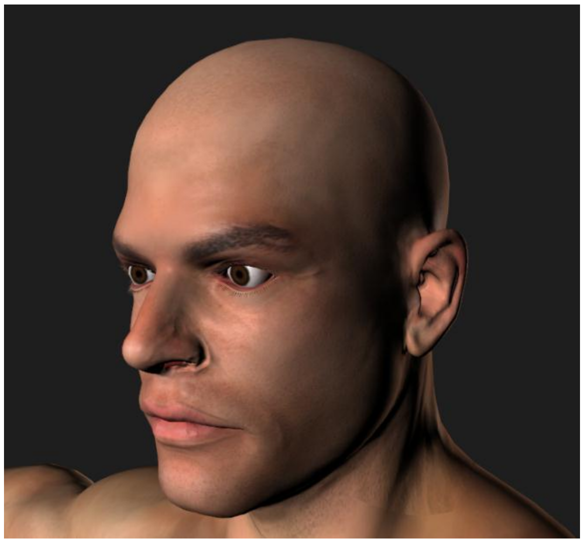


Figure 28: The head mesh with diffuse and normal maps applied

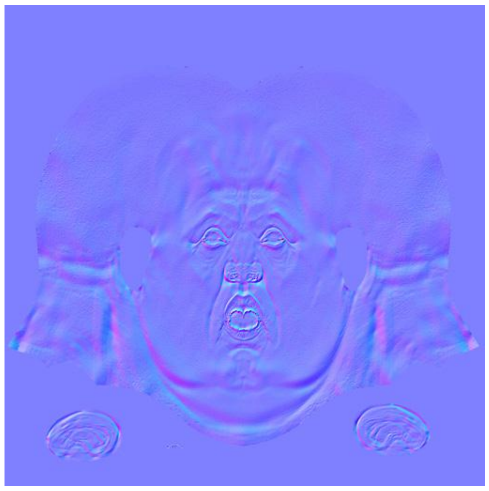


Figure 29: The normal map of the head is applied

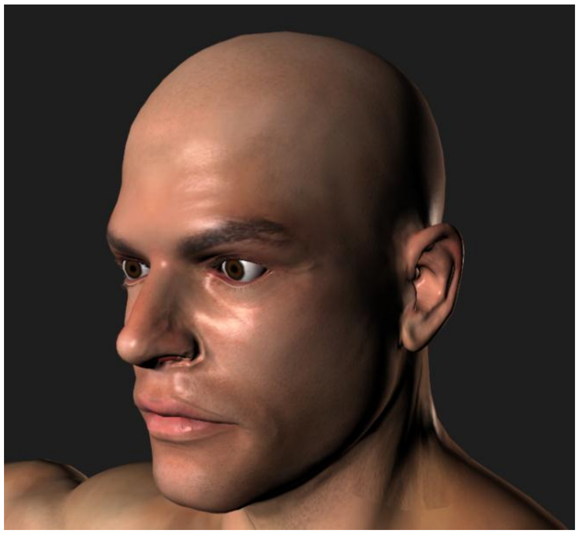


Figure 30: The head mesh with diffuse, normal and specular maps applied

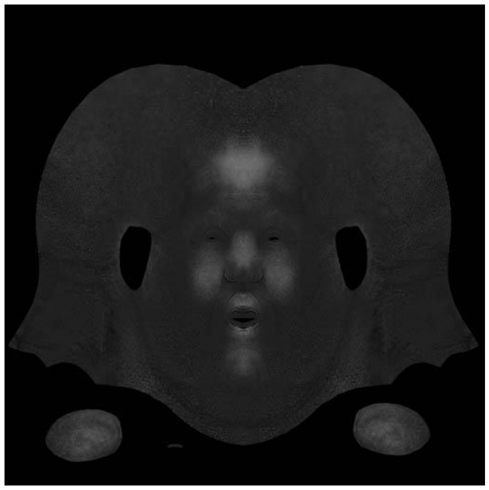


Figure 31: The specular map of the head is applied

The combination of the above mentioned maps along with additional custom color information (such as additional emissive color) constitutes the *material* of the mesh. Once the material is applied to a mesh, the renderer proceeds to apply lighting and shadowing information (Section 3.1.4.2).

3.1.4.2 Lights and Shadows

To create an even more realistic scene, each mesh in the scene must be properly lit and generate shadows that correspond to scene lights. To achieve this, SunBurn offers two types of lighting modes:

Dynamic Lighting: A lighting mode where each light in the scene is calculated in real-time. If the light properties (such as direction, position and intensity) change or if the object or camera properties change, then the lighting of the scene is affected in real-time just like in the real world. This mode offers realism at the expensive of performance.

Static (Baked) Lighting: A lighting mode where the lighting information never changes and is therefore "baked" into an additional map called the *light map*, which is applied to the mesh just like the material UV maps. This mode offers performance at the expense of realism.

Because ViC is a dynamic, interactive and extendible system, the dynamic lighting mode is chosen to provide real-time lighting. There are two *directional* lights setup in the prototype with one lighting the front of the cadaver, and the second lighting the back. An additional two *spot* lights are setup around the interactive area of the prototype to provide additional lighting to the currently fragmented area of the data.

Despite its contribution to realism, only limited shadowing is used. Global shadowing, where each of the *SceneEntity* instances casts a shadow, is not used as it limits the visibility of the interactive portion of ViC. However, to ensure that the element of realism is retained, *self*-

shadowing is used. Self-shadowing enables *SceneEntity* instances to casts shadows onto themselves, but not onto other scene objects (Figure 32).

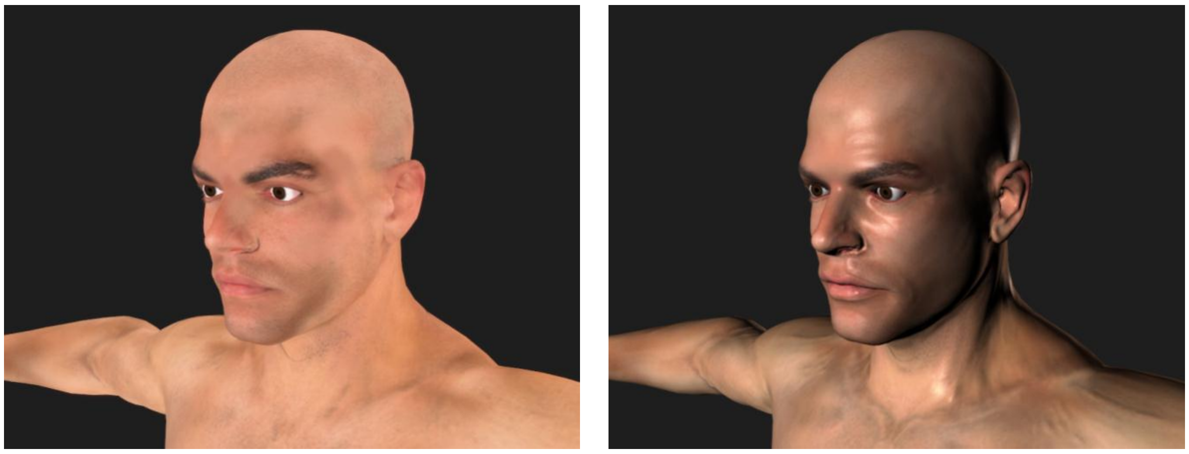


Figure 32: A comparison of a scene with only ambient lighting and no self-shadowing to a scene with a four-light setup and self-shadowing enabled

As a consequence of dynamic lighting and shadowing, the number of polygons rendered in ViC's final scene is higher than the dataset's 5.5 million polygons. Dynamic lighting and shadowing require multiple passes at ViC's dataset geometry resulting in approximately 8 million polygons rendered when the entire cadaver is visible.

3.1.4.3 Post processing

Post-processing is image processing that takes place after the final scene frame is constructed. In both film/video production and real-time 3D rendering such as video games, post-processing is used to improve the quality of the frame or add additional realistic or cinematic effects.

ViC uses two post-processors to increase the realism of the cadaver:

Bloom: Strictly speaking, the human eye never sees the bloom effect. Instead, bloom simulates the fact that in the real world, camera lenses almost never focus perfectly. As a result the image they capture often contains imaging artifacts where light around the brightest object in an image obscures some parts of the object. This effect is better described as a *cinematic* effect.

SSAO: Screen Space Ambient Occlusion is an implementation of the *ambient occlusion* technique that is suitable for real-time rendering. Ambient Occlusion is a shading technique that adds realism to a 3D scene by accounting for the way light affects all surfaces in the real world, including those that are considered non-reflective [38]. SSAO adds dark shadow-like areas that simulate the attenuation of light due to occlusion.

To ensure that the scene is not “drowned” in a massive amount of post-processing that would take away from the main interest and goals of ViC, both of these post-processors are set to the most subtle level possible.

3.1.5 The Collision Component: Picking with Triangle Precision

As stressed previously, ViC is not just a browser but an *interactive* virtual cadaver with cutaway capabilities. To offer interactivity in 3D, ViC must detect what part of the anatomy the user is (virtually) contacting. Cutaway operation in the virtual cadaver is achieved through a touch display. ViC fires a ray from the location of the user’s finger on the touch display into the scene when the cutaway gesture is performed. This ray is used for an intersection test with the meshes. Section 3.1.6 gives a more in-depth look into how input is received from the touch panel and how it is translated into gestures to provide high-level input information.

ViC’s collision component performs a ray-to-triangle intersection test against all interactive objects in 3D space in order to detect selected fragments with triangular precision (Algorithm 1).

```
1. For each fragment
    a. Perform a Ray vs. BoundingSphere test (Figure 33)
    b. Record the names of all fragments that passed the above test
2. For each fragment that passed the Ray vs. BoundingSphere test
    a. Perform ray-to-triangle intersection test (Figure 34)
    b. Record the names of all fragments that passed the above test
```

Algorithm 1: Collision checking algorithm

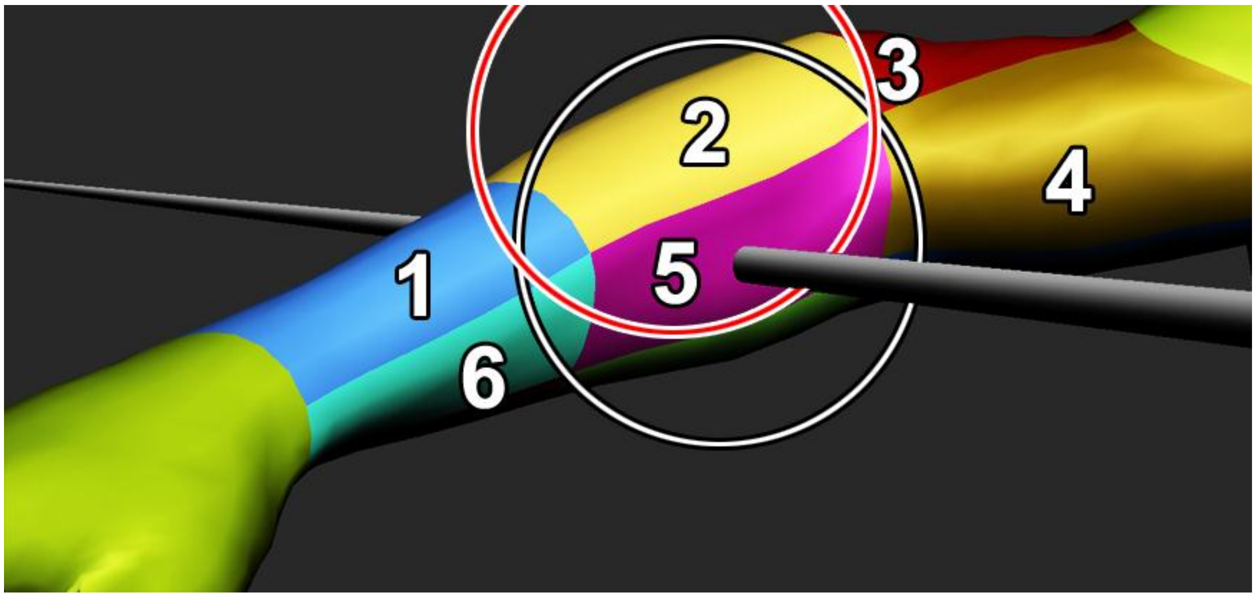


Figure 33: A Ray[21] vs BoundingSphere test. Sample ray intersects the bounding spheres of fragment 2 and 5, requiring a triangular collision test to determine which fragments were actually collided with.

[21] The ray representation in these images is for the purpose of visualizing the collision relationship. Fundamentally, a ray does not have physical dimensions.

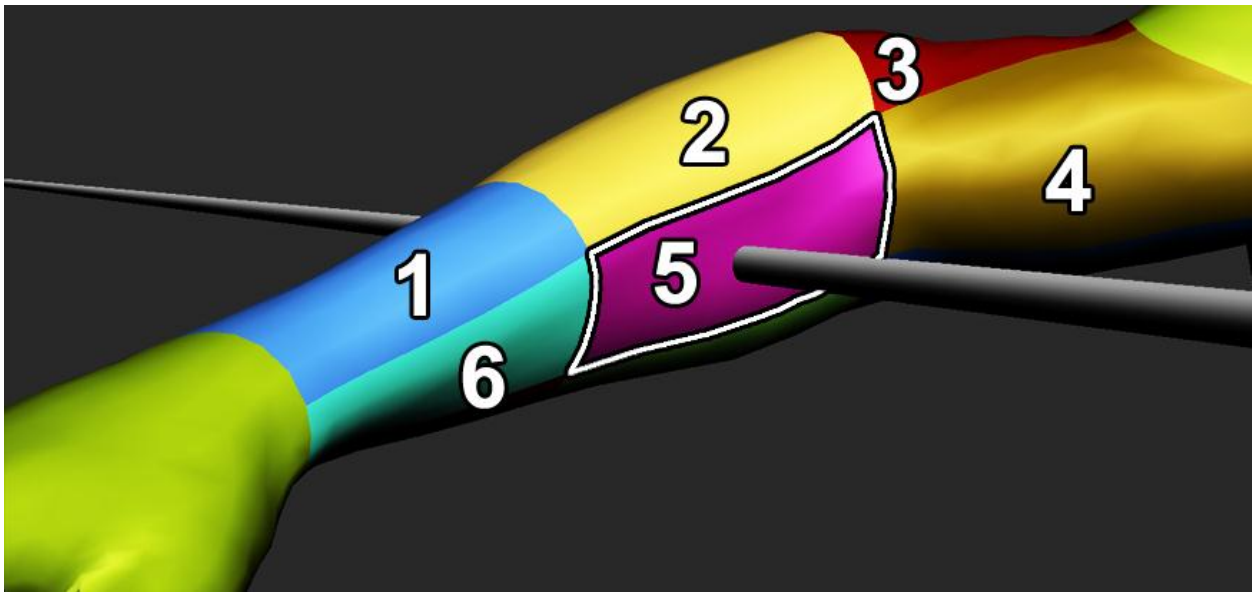


Figure 34: A Ray[21] vs Mesh Fragment test. On the triangular level the ray only collided with fragment 5.

The triangular intersection test [39] is a fast, CPU-side test that determines whether or not a ray intersects one or more triangles of a fragment. While this test is fast enough for a real-time application, given the size in polygons of ViC's dataset, millions of polygons per frame would have to be evaluated. This would make ViC's collision component a bottleneck[22]. The solution is to rule out all geometry that cannot be intersected by the ray through a much faster test, the *bounding sphere* test. The bounding sphere test records the names of all fragments that contain the ray in their bounding spheres. To determine whether or not their actual geometry intersects the ray, the *ray vs. triangle* test is then performed only on the fragments that passed the bounding sphere test.

Algorithm 1 runs on all meshes when the user makes the initial touch to determine which anatomy part the user made contact with. Once the anatomy part represented by the selected

[22] A *bottleneck* is a situation where the performance of a system is slowed down by a slower component.

mesh is determined, the cutaway process is locked to that part only. As a result, Algorithm 1 needs to run on the mesh fragments of the selected body part only during the cutaway process.

3.1.6 The Touch Interface Component: Input from gestures

Interaction with the virtual cadaver is achieved through a multi-touch display. To receive touch data from the display and pass it on to ViC, the Windows Touch API [40] is used. The API exposes three methods that record rich[23] input data through the *touch down*, *move* and *release* events. This data is recorded by ViC and the combination of these events is used to build *gestures*[24].

Gestures that involve interaction with the fragmented areas of the virtual cadaver also pass the location of the finger(s) to the collision component (Section 3.1.5) that constructs a *ray*[25]to be used for intersection tests. ViC uses the following *core* (required) gestures for interaction with the virtual cadaver:

Touch and move: This gesture involves the user touching the touch panel for a half a second (500 milliseconds), which enables camera control. Subsequent movement of the finger along the touch panel without lifting is used to rotate the camera in that direction. Horizontal movement of the finger is used to rotate the camera about the Y-axis. Vertical movement is used to rotate the camera about the X-axis. The gesture ends when the user lifts the finger off the touch panel.

[23] The touch data includes the location of the touch, the timestamp, the size of the touch area, whether or not the touch is a finger or a palm and more.
[24] A gesture is a high-level encapsulation of the way the user touches the touch panel.
[25] A *ray* is an imaginary line in 3D space that originates at the location of the touch and continues perpendicularly to the plane of the touch display.

Tap: This gesture involves the user tapping (touching and instantly releasing) the touch panel. This gesture is contextual. If this gesture is performed on a fragment, it becomes the object of interest. The camera pans towards the object of interest to make it centered on the screen. Additionally, if the camera zoom level is less than a certain threshold[26], the camera automatically zooms in on the object of interest. If this gesture is performed on a UI element, the option of that element is executed. Section 3.1.7 covers the UI elements in-depth.

Pinch: This gesture involves the user touching the touch panel with 2 spaced-out fingers and moving them towards each other or apart to zoom-in or zoom-out the camera view. The gesture is registered only if the fingers are spaced apart. If the two fingers are touching when they make contact with the touch display, a different (unused) gesture is fired.

Touch and move/swipe: This gesture involves the user touching and instantly proceeding to either move or swipe along the touch panel. This gesture is contextual. If the user touches and moves/swipes along an empty or non-interactive area of the virtual cadaver, no action is taken. If the user touches and moves/swipes along a fragmented area, the collision component (Section 3.1.5) determines which fragments the touch made contact with, and performs the cutaway operation. The cutaway removes the fragments according to the direction of the move/swipe and the layout of the fragments. For example, if the user touches and swipes along a patch of skin, the fragments are faded away.

In addition to the above required gestures, ViC also contains the following optional gestures that are available on a touch panel that supports more than 2 touch points.

[26] ViC's default zoom threshold for objects of interest is 2.85X.

Three-Finger Tap: This gesture involves the user tapping the touch panel with three fingers, which resets all fragment cutaways that were performed on the virtual cadaver.

Four-Finger Tap: This gesture involves the user tapping the panel with four fingers, which resets the camera view location and zoom level to make the center of the cadaver as the object of interest.

3.1.7 The UI Component

The UI component complements ViC's 3D scene with a 2D interface, which draws 2D elements on top of the 3D scene. This interface is broken down into 2 parts:

Menu System: The extendible menu system is found on top of the screen, although it can be configured to be on one on more of the 4 sides of the viewport. It contains a collection of icon-based menu options that offer the toggling of the visibility of the various human anatomy systems.

Notification system: The notification system displays notifications about ViC's various events. For example, if the user touches and holds the touch panel, the user is notified that the camera control has been enabled as per the registered gestures of the touch interface component (Section 3.1.6).

3.2 Using ViC

3.2.1 Camera Control

ViC uses a *Third Person Camera*, which has an unrestricted horizontal movement but a restricted vertical movement to a 180-degree view about the X-axis. This camera model mimics how human beings turn their head to see around them and tend to only look up high enough to see what’s directly above them, or only low enough to see their feet.

A user presses a finger and holds anywhere on the touch screen to activate the camera control. Once the camera control is activated, the user moves a finger up/down the touch panel to move the camera about the X-axis (Figure 35 and Figure 36) or left/right to move it sideways. The user then lifts their finger off the touch panel to deactivate the camera control.

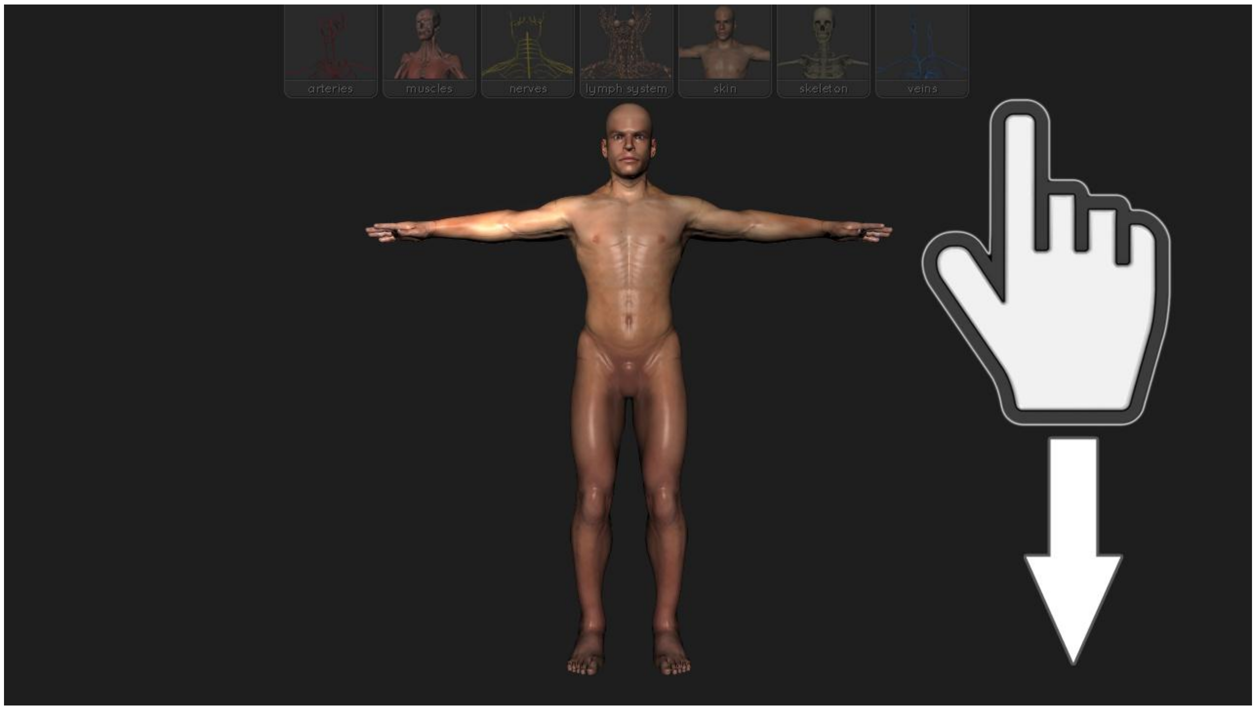


Figure 35: One finger being move down along the touch panel to move the camera view up

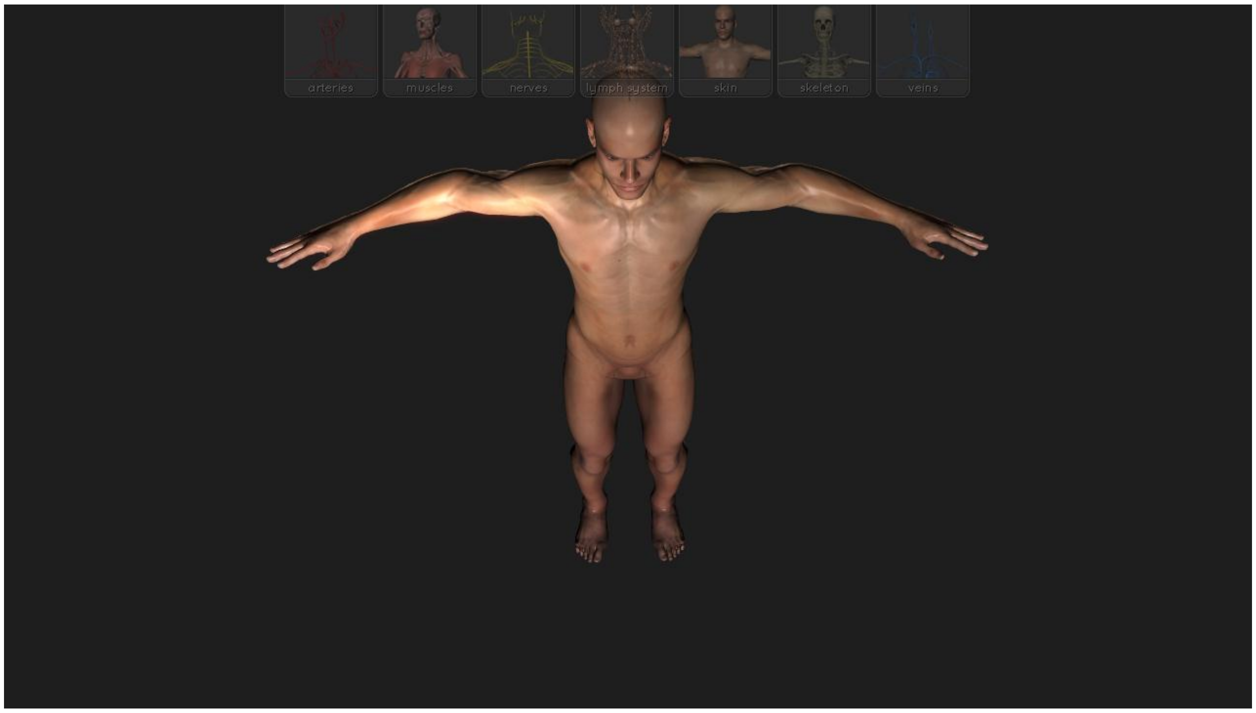


Figure 36: Camera view after the completion of the touch and move down gesture.

3.2.2 Cutaway

The removal or cutting away of part of an anatomy object in ViC is based on a common natural gesture that uses the index finger to slide things out of the way. To understand this gesture, picture a piece of paper the size of a post-it note that is placed on top of a book which is in turn placed on a table. The move that involves the least effort to expose the part of the book that is covered by the paper is to slide the paper off the book. Since the paper is the size of a post-it note, only one finger is necessary for this gesture.

This natural sliding or swiping gesture is used to cutaway fragments in ViC. Moving/swiping with a finger over a sequence of fragments removes them in real-time. In Figure 37, a cutaway on an upper arm muscle commences with a touch followed by a swiping gesture in the direction of the arrow (Figure 38).

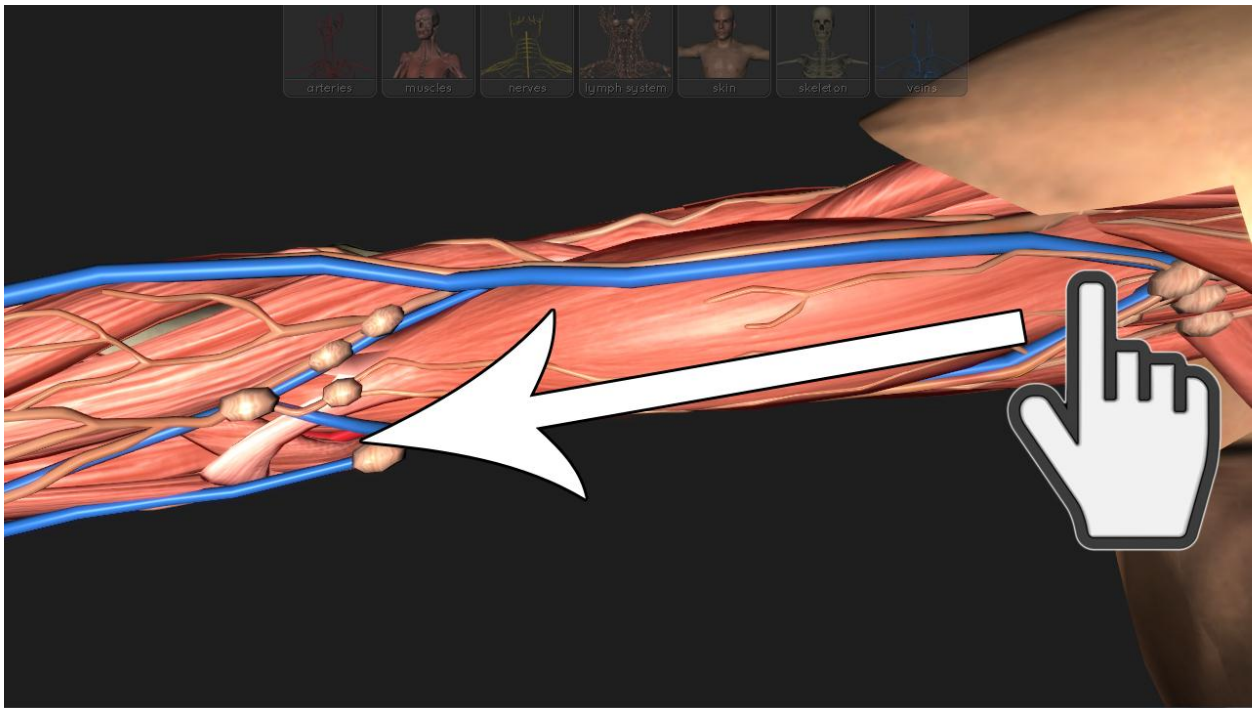


Figure 37: A cutaway action on an upper arm muscle commence at the touch point indicated

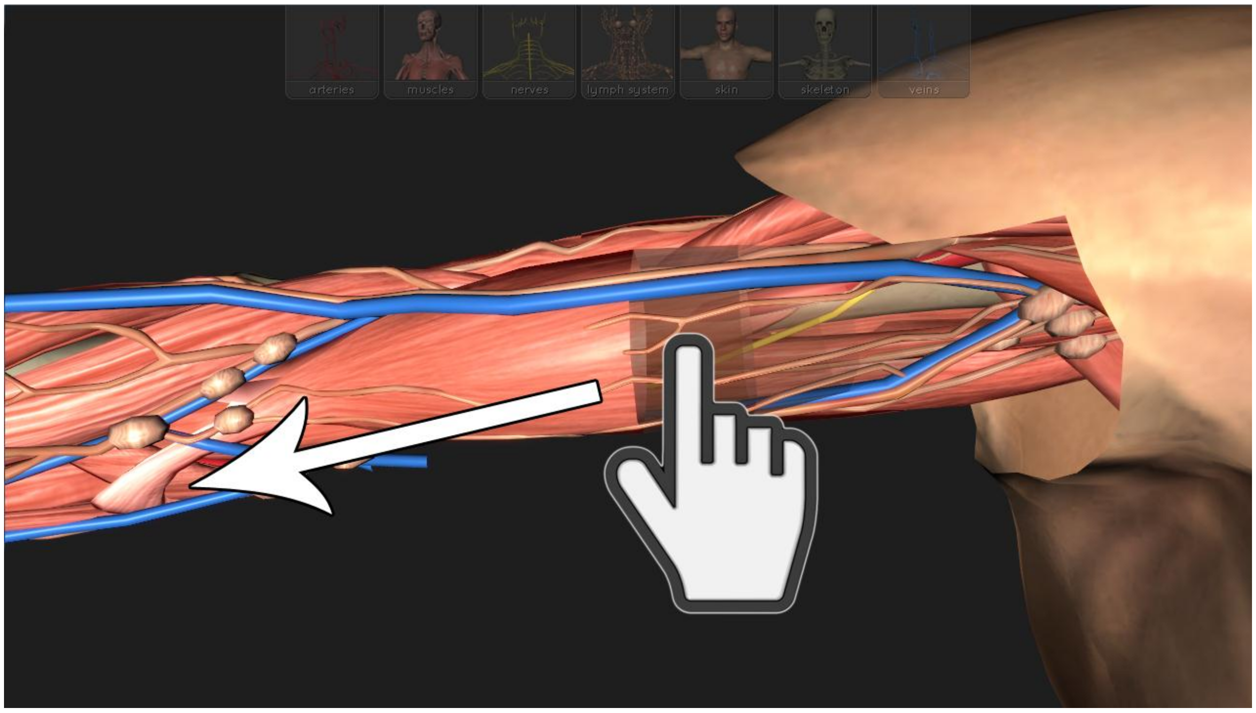


Figure 38: A quick swiping gesture along the length of the muscle removes the underlying fragments

During the course of the cutaway gesture, the anatomy object on which the cutaway action began remains the only object on which it can continue. The cutaway is complete when the user lifts the finger or when the object has been completely removed (Figure 39).

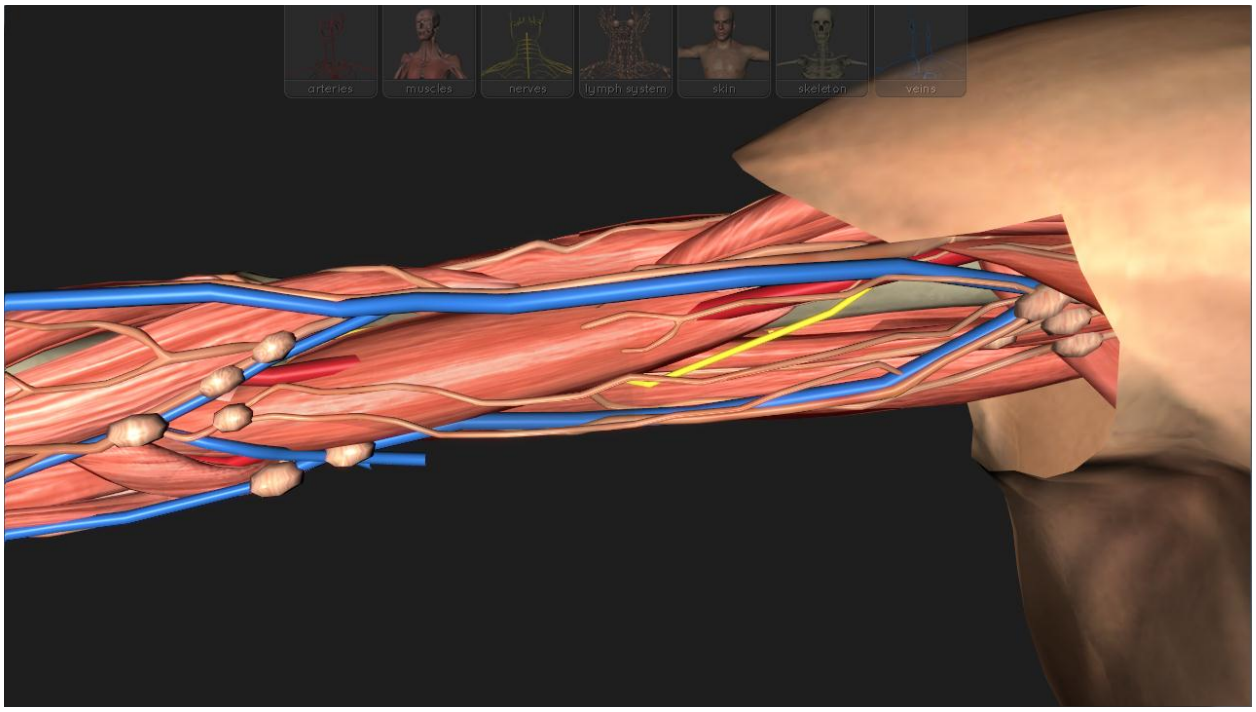


Figure 39: A cutaway of the entire muscle has been completed

3.2.3 Input Combinations

Some of the gestures outlined in Section 3.1.6 can be used together in meaningful combinations.

CAMERA PAN + ZOOM

If the camera control has been activated, the camera can be rotated with the finger that is currently in contact with the touch panel. However, if the second finger makes contact with the touch panel while the first finger is still down, the zoom control is activated *without* deactivating the camera control. Once the second finger is lifted, zooming of the camera ceases and the camera rotating control continues.

CUTAWAY PRECISION

Once the cutaway process commences, ViC will continue to remove fragments that the finger is making contact with. To skip a fragment while cutting, it is possible to navigate the finger around the fragment then return to the object to continue the removal of the remaining fragments. An alternative technique involves making contact with the touch panel with a second finger to "halt" the cutaway process. The cutaway finger can continue moving along its intended path without removing fragments that it is making contact with. To resume the cutaway, the second finger is lifted. It is important to note that this combination is nearly identical to the ***camera pan + zoom*** combination. The only difference between the two combinations is context. In the ***camera pan + zoom*** combination, the situation involves the activation of the camera control prior to the second finger making contact with the touch panel.

3.2.4 UI

The default menu system offers selectable options that toggle the visibility of the individual anatomy systems (Figure 40). Tapping a highlighted option removes the corresponding anatomy system. Tapping a dim option brings back the view of the anatomy system along with any cutaway operations that may have been applied to it.

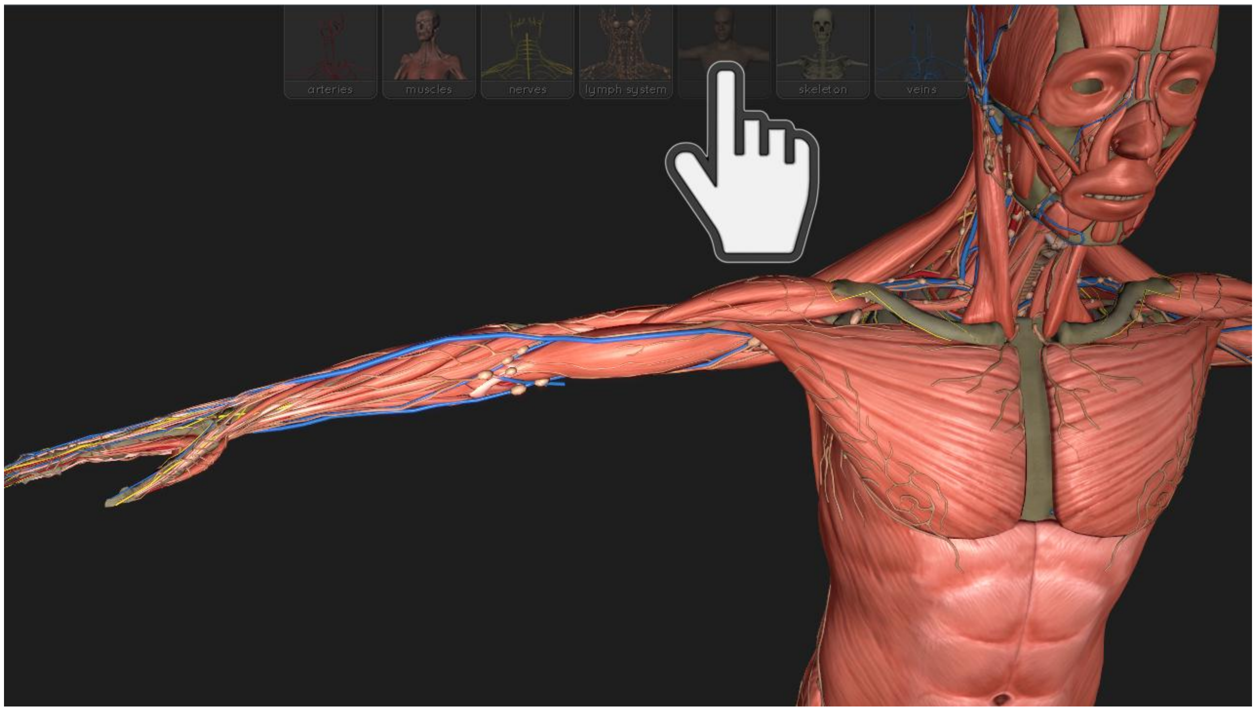


Figure 40: The skin option has been tapped. The UI option dimmed indicating that the skin anatomy system is no longer visible and the skin has been hidden. Tapping it again will make the skin visible.

3.3 Extendibility

The components that provide ViC with various unique capabilities are loosely coupled, meaning they are modular enough to have little or no dependency on other components or ViC's core class [41]. The *AnatomyData* container is a completely independent class that can be accessed by both the core of the system, its components or any other class. Furthermore, the *AnatomyData* container is a read-only collection of data. Once ViC loads *AnatomyData* with the dataset and builds the look-up dictionary, the entire class becomes a read-only container from which data references can be retrieved through name-based look-up but nothing can be modified. This ensures that independent components, which have no knowledge of other components or how they interact with the data, do not modify the dataset and cause inadvertent effects on other components that use it.

As mentioned in Section 3.1, ViC's components extend XNA's *GameComponent* and *DrawableGameComponent* classes. These classes automatically subscribe the components that extend them to XNA's execution model. This makes the execution flow of any component identical to the execution flow of the XNA-derived application (ViC), allowing XNA to safely manage the execution flow of the component.

In order to extend ViC's functionality through XNA's component model, an extended component class organizes its logic into two fundamental sections.

INITIALIZATION

This part of the execution flow requires the component to match the initialization and loading to XNA's two key methods – *Initialize* and *LoadContent.* The *Initialize* method requires the component to perform all of its logic initialization prior to content loading or any graphics operations. The *LoadContent* method requires the component to perform all of its loading of content assets.

RUNTIME

This part of the execution flow requires the component to match the game loop through XNA's two key methods – *Update* and *Draw*[27].

Despite the advantages of component-based modularity, extendibility of ViC is not restricted to the component model. Some functionality is inefficient or outright impossible to be implemented in a self-contained module. For this reason, any functionality that must reside in an independent class that does not conform to XNA's component model can be self-managed without any complications. Section 4.2.2 demonstrates and evaluates such an example.

[27]*Draw* method is required if the component derives *DrawableGameComponent* and intends to render to screen.

Chapter 4: Evaluation and Results

In the preceding chapters it has been discussed that a virtual cadaver must offer not only human anatomy browsing at the *global* anatomy system level but also interaction capabilities on a *local* anatomy object region level. The visualization of human anatomy must allow the user to see meaningful 3D views both with and without any interactive cutaway operations on the dataset. The interaction on the dataset must be meaningful in order to allow the user to perform actions that resembles real-world actions. The input actions must be intuitive and natural in order for them to be easy to learn and easy to understand their effect. Additionally, in order to qualify as a system, a program which provides these interaction capabilities must also be readily extendible. Furthermore, all such achievements must not come at the expense of real-time performance; the user's visual feedback and interaction lag cannot become choppy or unbearably slow[28].

To evaluate the interaction, extendibility, and performance capabilities of ViC, the following sections document a number of experiments and performance measurements with respect to the above requirements.

[28] In real-time rendering, the term *slow* usually refers to a low frame-rate.

4.1 Interaction: Exposing a target anatomy object/system in context

This scenario involves the gradual removal of occluding geometry to expose a hidden target anatomy part/system while retaining visualization context. The target anatomy part for this experiment was the right *humerus*, the upper arm bone from the shoulder to the elbow [42]. The goal of this experiment was to expose the humerus enough to see its location relative to the muscle system.

4.1.1 Exposing the humerus through browsing operations only

To demonstrate the difference between browsing and the cutaway ability of ViC, this experiment was first performed with browsing operations only.

STAGE 1:

To navigate to the general area of the right humerus – the upper section of the right arm – a point on the skin in that general area was tapped (Figure 41).

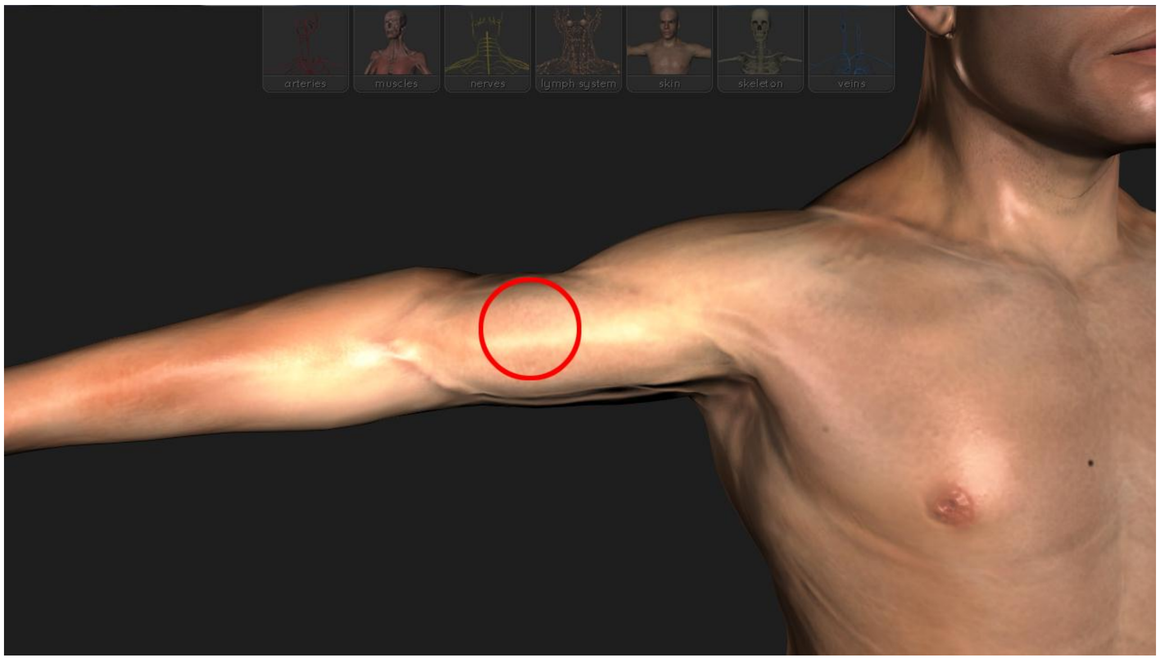


Figure 41: The upper right arm becomes the focus of the camera after the area enclosed by the circle was tapped

STAGE 2:

The visibility of the skin was turned off with the goal to possibly expose some of the visibility of the skeletal system (Figure 42).

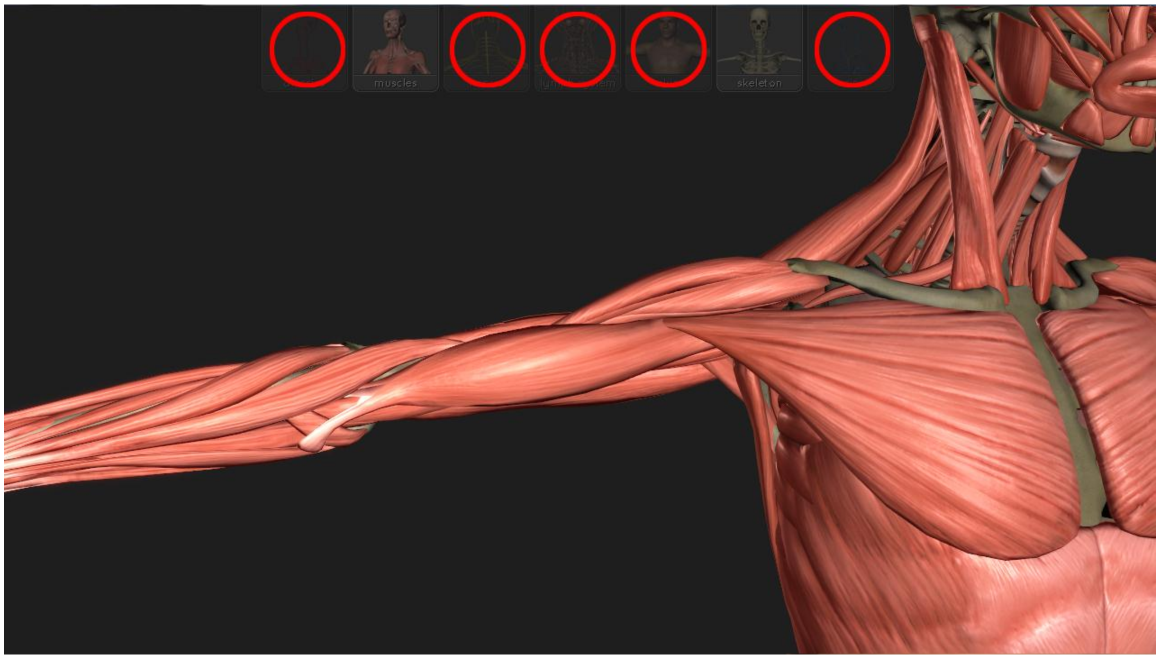


Figure 42: The skin visibility turned off. Veins, arteries, nerves and the lymph system were not needed for this experiment, so they were hidden as well.

STAGE 3:

At this point, the humerus was still not visible. The only browsing operation left at this point was to turn off the visibility of the muscles (Figure 43).

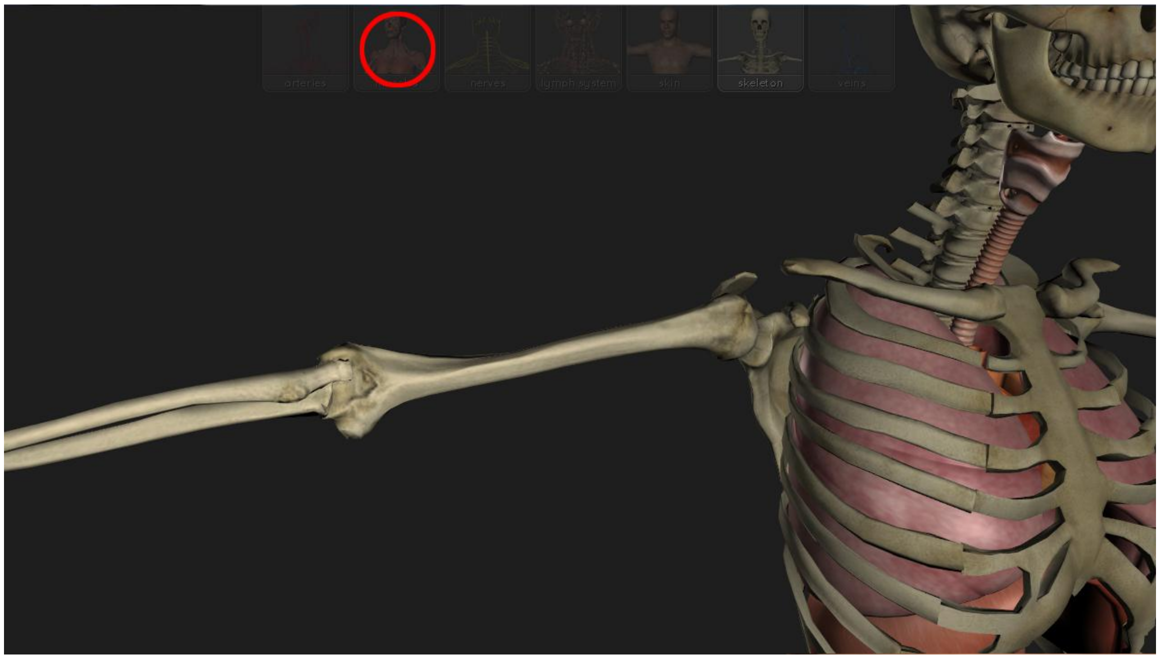


Figure 43: Muscle system visibility turned off exposing the humerus bone in the arm

The humerus was finally exposed in Figure 43. However, by this point the experiment failed. While the humerus was clearly visible, the intended context – exposing the humerus in its place relative to the muscles – has been completely destroyed.

4.1.2 Exposing the humerus through fine grained cutaway

The experiment in Section 4.1.1 failed at ***Stage 3*** due to the fact that exposure of the humerus meant the complete destruction of the context by removing the entire muscle system or retaining a noisy context through the use of partial transparency. This experiment was repeated with the replacement of ***Stage 3*** with ViC's fragment cutaway approach. The *biceps brachii* [42] muscle was cutaway (Figure 44) in order to remove only enough muscle tissue in an attempt to expose the humerus without completely destroying the context.

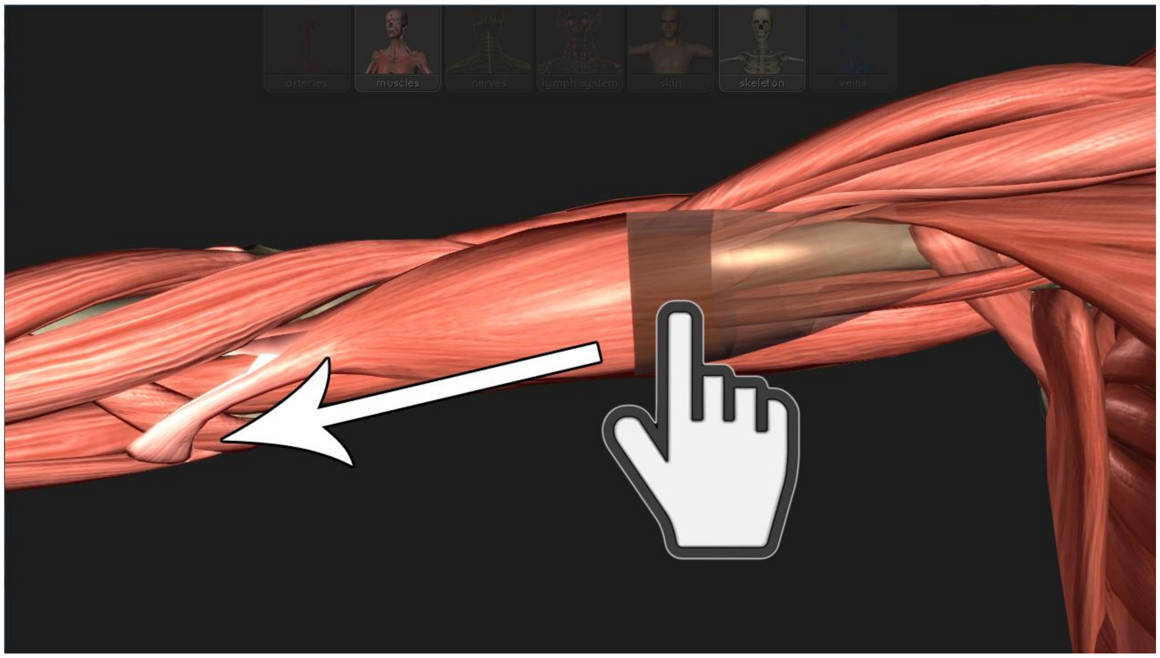


Figure 44: The *biceps brachii* muscle being progressively removed in order to expose the humerus.

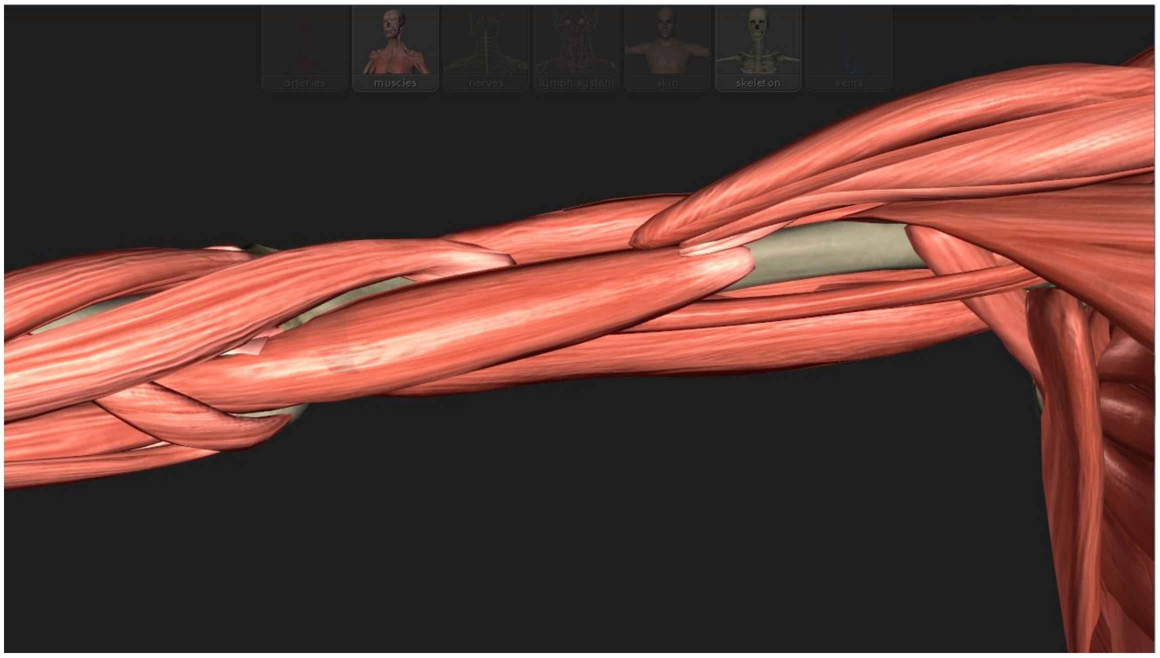


Figure 45: Result of completion of the biceps muscle cutaway.

Figure 45 shows the humerus partially visible in its appropriate context. The cutaway process was repeated for the second occluding muscle – *brachialis anticus* [42] – in order to fully expose the humerus (Figure 46).

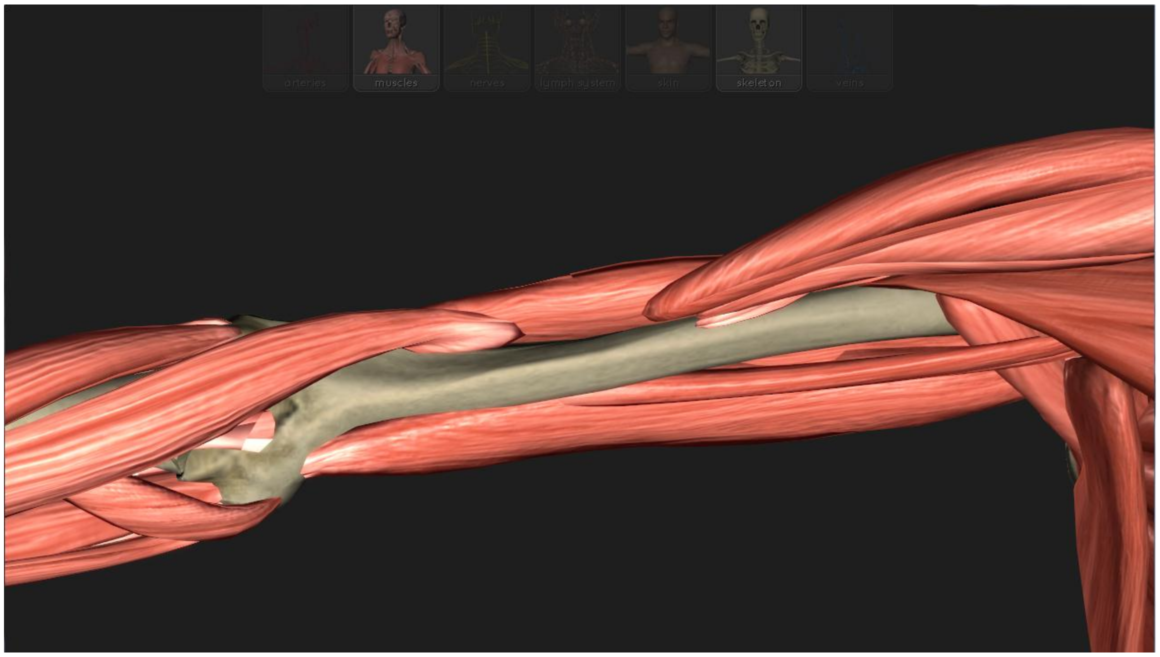


Figure 46: Two muscles were cutaway in order to fully expose the humerus.

Figure 46 shows the humerus fully visible, with only two occluding muscles cutaway. The visualization context – the humerus location relative to the muscular system – remains in-place.

4.1.3 Exposing the humerus in full context

In the previous experiment, the humerus was exposed in the context of the muscular system with the cutaway of the two occluding muscles. However, this view is not sufficient if the user's goal is to expose the humerus in its *full* context.

For this experiment, all of the omitted systems from the previous experiment were set to visible again (Figure 47).

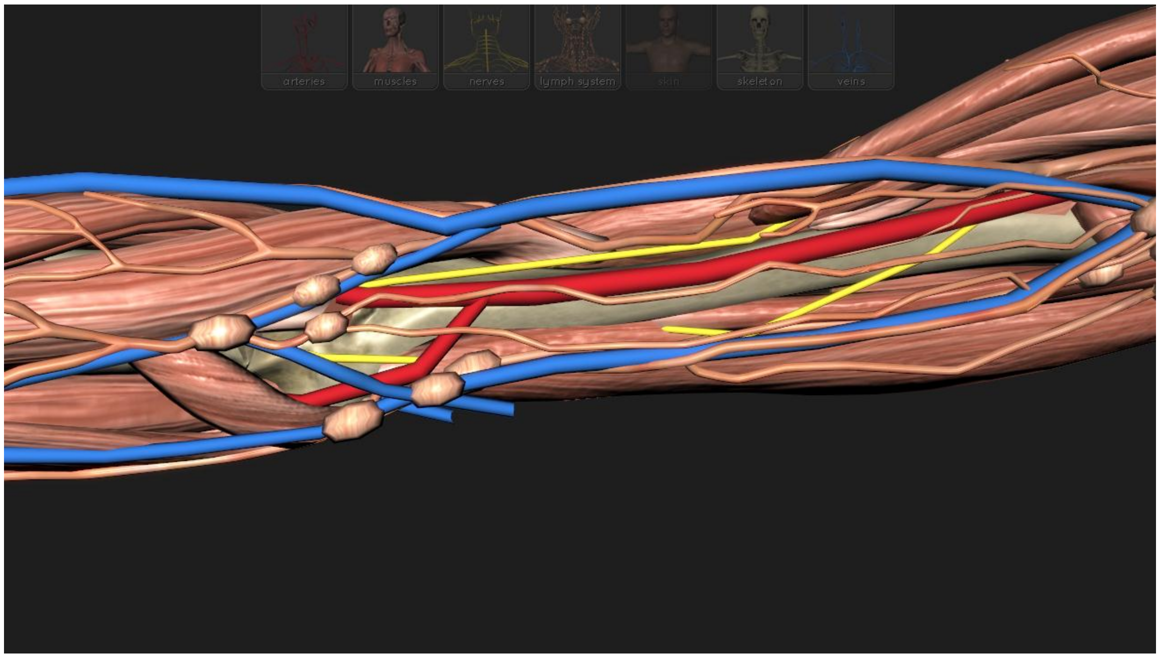


Figure 47: The humerus is once again occluded by the veins, nerves, arteries and the lymph systems.

In order to once again expose the humerus, the occluding parts had to be removed. However, this time, entire systems were occluding the humerus, as opposed to just single anatomical structures such as the brachialis anticus and biceps brachii muscles.

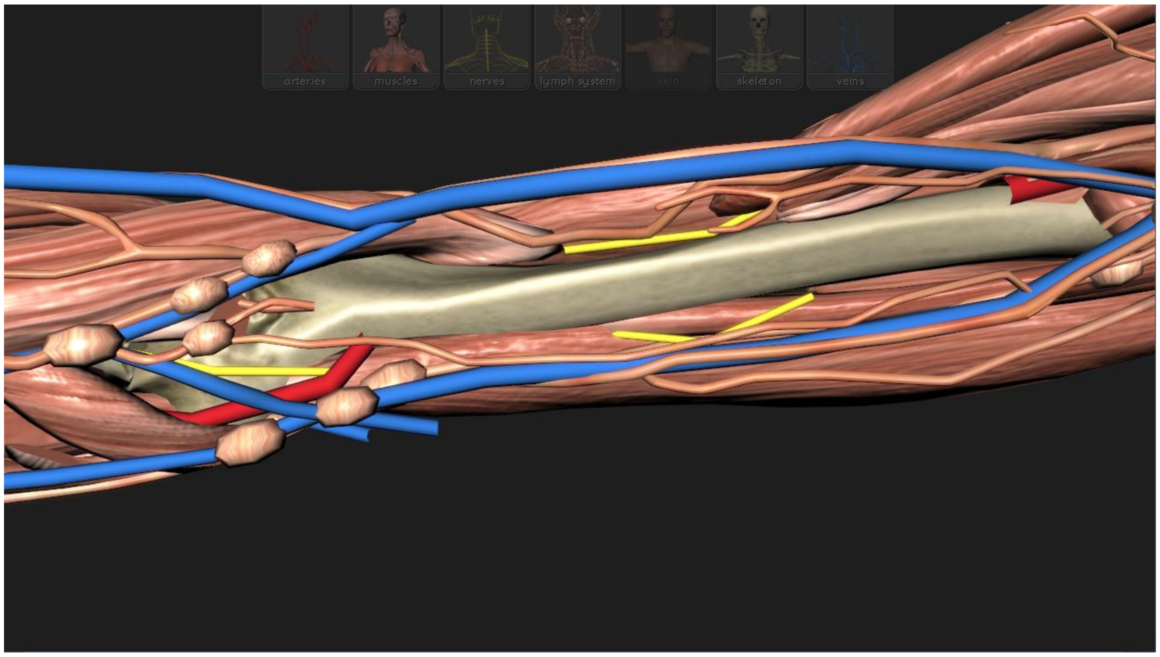


Figure 48: Cutaways performed on sections of entire anatomy systems to expose the humerus

Cutaway of sections of the veins, arteries, nerves and the lymph system were performed in order to remove the occluding portions. The humerus was once again exposed in context with not only the muscular system but all of the anatomical systems (Figure 48).

4.1.4 Performing a complete cutaway of the right arm

In the previous two experiments, cutaways were performed in order to expose a target anatomical structure. Another operation that a user might be interested in performing is the general exploration of the right arm area through cutaway operations. As already shown in the first experiment, browser capabilities are limited to anatomical system level when it comes to general exploration. If the user wishes to explore further into the anatomical systems themselves, full control to cutaway all systems must be provided.

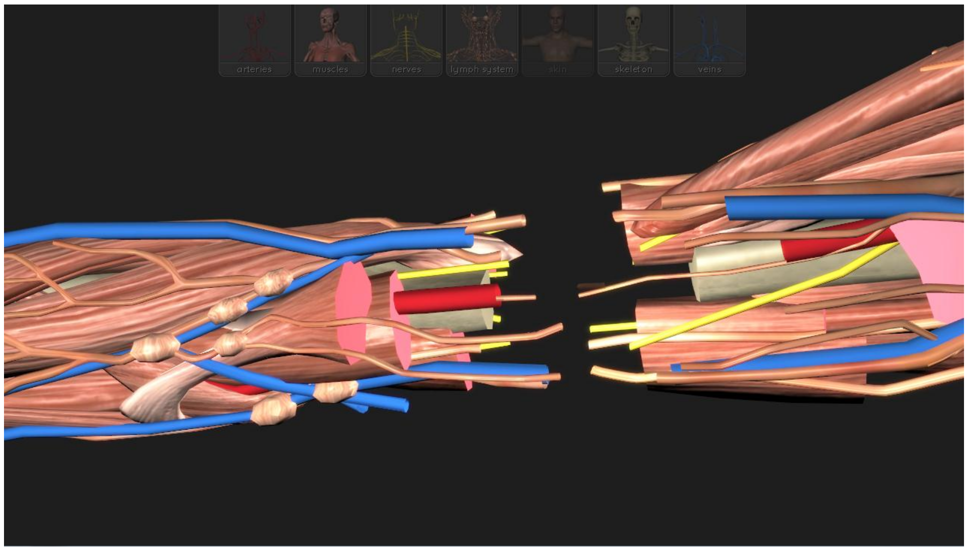


Figure 49: A full cutaway of all anatomical systems of the right arm

Figure 49 shows how a number of cutaways were performed an all of the anatomical systems of the right arm in order to completely cut through it. All of the spatial relationships have been retained and all anatomical systems are still visible in their contexts relative to each other.

4.2 Extendibility: Extending ViC beyond the prototype capabilities

Section 3.3 provides an in-depth look into the component extendibility and how components comply with XNA's execution flow in order to become fully managed by XNA. This section evaluates the extendibility capabilities by extending ViC's functionality.

4.2.1 Adding a component

Section 4.3 evaluates ViC's performance to determine whether or not it runs at a minimum of 60 frames per second as per the feature-contribution listed in Section 1.1. To determine and benchmark the frame-rate of ViC, a *profiler* component was added to the component collection. A *profiler* is a tool that can be used to determine and benchmark the performance of a real-time rendering application. ViC's profiler component will simply measure and show the *frame-rate*.

The following algorithm [43] provides an accurate way to measure the frame-rate of an XNA application:

```
1.  Initialize the frame counter and current elapsed time to zero
2.  When rendering
        a.  Increment the frame counter
3.  When updating
        a.  Increment the current elapsed time
        b.  Check if the current elapsed time is greater than or equal the target time (1 second)
        c.  If b. is true
                i.   Subtract one second from the current elapsed time
                ii.  Record the value of frame counter[29]
                iii. Restart the frame counter
        d.  Otherwise carry on until the next update call
```

Algorithm 2: Frame-rate measuring algorithm.

[29] This is technically the *return value* of the algorithm. It is not "returned" but instead accessed when the frame rate value is being displayed.

Note that the above approach maps perfectly to XNA's execution flow and therefore easily becomes a self-contained component. The first step resides in the component's *Initialize* method while the third step belongs in the component's *Update* method. In order to count a rendered frame, this approach requires the current frame counter to be incremented inside the *Draw* method, subsequently placing the third step into the *Draw* method. As a result, this requires the profiler to be a *DrawableGameComponent* despite the fact that it does not technically need to render anything. The profiler passes the resultant frame-rate to its parent (the core *ViC* class) which in turn displays the frame-rate in the title bar of ViC's window. The profiler component was added to ViC's component collection. Since this component is just a frame counting benchmark tool, it does not require access to the *AnatomyData* container.

To verify the accuracy of the profiler component, its results were compared against *fraps* – a commercial frame-rate measuring tool that is known to be accurate.

The results of both the profiler and fraps were compared through two simple experiments. In the first experiment, the frame-rate of ViC under normal load was noted. Under normal load, a properly functioning and well-performing XNA application should run at the refresh rate of the monitor[30] with *vertical synchronization*[31] on. Figure 50 shows the frame-rate measured by ViC's profiler component (in the title bar) matching that of which was measured by fraps (just under the title bar).

[30] A modern day LCD refresh rate is 60 hertz.
[31] Vertical synchronization (or V-sync) forces the completion of a frame rendering, prior to clearing the screen buffer and rendering the next frame. This is used to prevent *tearing*, a quality-impacting condition in which a mishmash of both frames appears on the screen.

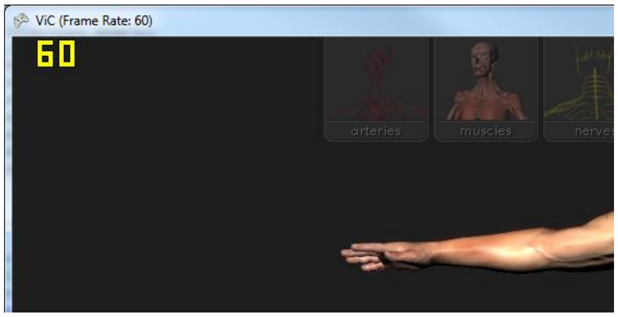


Figure 50: Frame-rate under normal load.

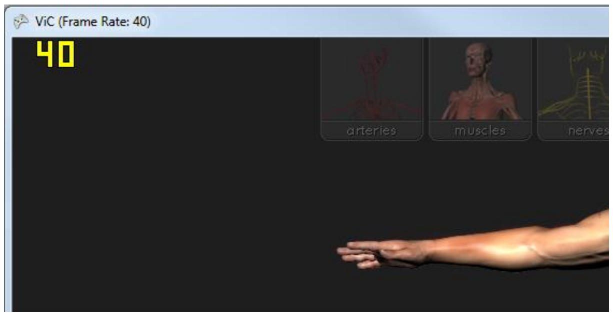


Figure 51: Frame-rate under heavy load.

To force the frame-rate to drop without any practical purpose, ViC's core update method was populated with junk code executed in a large loop. The consequence of the increased load was the reduction of the frame-rate by 20 frames per second. Figure 51 shows that both ViC's profiler component and the control (fraps) detected and agreed on this figure, verifying the profiler's accuracy.

4.2.2 Adding further non-component based functionality

Despite the advantages of the component model, ViC's extendibility is not bound to it. Further functionality and capabilities can be encapsulated by any class that does not necessarily extend XNA's game component classes. Calls to these classes' methods can be made directly from ViC's core class or through other components. A common instance of such scenario would be a third party library that has its own mechanism that is not modeled around XNA's execution flow.

As an example, ViC was further extended with a third-party concurrency library. *ParallelTasks* [44] is an XNA library that acts as a task scheduler to distribute the application load between all available processor cores. It is designed to mimic the Microsoft Task Parallel Library. *ParallelTasks* does not follow XNA's execution flow. In fact, the library has only one method that must be executed in the game loop in order to execute all of the currently scheduled tasks.

The functionality of the entire library is encapsulated in its static *Parallel* class, methods of which can be called on demand. *ParallelTasks* was imported into ViC as a separately built .NET-based library and its namespaces[32] were exposed in ViC's core class. *ParallelTasks*' *RunCallbacks* method call was placed in the core class *Update* method.

Section 3.1.5 explains how the collision component uses a fast triangle collision test to determine which of the fragments are selected by a gesture. The test iterates over all fragments that passed the bounding sphere test. Although this is a fast test, it is a brute-force (exhaustive search) approach and would therefore not scale with a sharp increase in the number of fragments. To solve this problem, the second step of the collision component's collision checking algorithm was modified to include the functionality of the *ParallelTasks* library:

```
1. For each SceneEntity in AnatomyData that is a fragment (has collision data)
    a. Perform a Ray vs. BoundingSphere test (Figure 33)
    b. Record the names of all fragments that passed the above test
2. Use ParallelTasks' parallel ForEach method to create a task for each of the fragments that passed the Ray vs. BoundingSphere test
    a. Perform ray-to-triangle intersection test (Figure 34)
    b. Record the names of all fragments that passed the above test
```

Algorithm 3: Multithreaded version of Algorithm 1.

ParallelTasks automatically scales the number of tests in step 2 of the collision checking algorithm to *n* processor cores. The results of the collisions are returned when ViC's core class *Update* method calls *ParallelTasks' RunCallbacks* method.

Section 4.3 documents the evaluation of the resultant performance gain from the above extension to confirm that parallelization was functioning properly and had a positive effect on ViC's performance.

[32] A namespace is a categorization mechanism for a collection of related classes.

4.3 Performance: Evaluating ViC's performance before and after extendibility

In order to qualify as a real-time system, ViC must run at a minimum frame-rate of 60 frames per second. The release[33] of ViC is automatically locked to a frame-rate of 60 frames per second, since vertical synchronization is turned on for the final build. However, in order to benchmark the true performance of ViC, determining just the fact of whether or not ViC is running at 60 frames per second is not enough. A more meaningful measure is required in order to conclude whether or not the prototype can handle further extendibility. Furthermore, to determine whether or not the functionality that was added in Section 4.2.2 truly improved the performance of ViC, the real (and not just *vsync locked*) frame-rate has to be measured. As a result, a much more thorough benchmarking mechanism than the one presented in Section 4.2.1 is required.

For the purpose of an accurate benchmark, a *profiler build* of ViC is configured. This build is identical to the release build, with the only difference being that vertical synchronization is turned off. This does not require the GPU rasterizer to wait for one frame to be drawn on screen before starting to draw the next. As a result, this enables ViC to run at its maximum frame-rate as opposed to the maximum vsync locked frame-rate of 60 frames per second. To record meaningful data with these changes, the profiler component from Section 4.2.1 is further modified to record a snapshot of the frame-rate figure every second for the duration of the benchmark.

To evaluate ViC's performance, three benchmarks were setup: *idle*, *single-threaded cutaway*, and *multi-threaded cutaway*.

[33] A release build of a .NET application is one that has been stripped of all debug symbols and optimized for usage as a final product.

IDLE

This benchmark involved ViC, in its state *before* the addition of the component in Section 4.2.2, running without any user intervention. The purpose of this benchmark was to determine the stock performance of ViC without any input.

SINGLE-THREADED CUTAWAY

This benchmark involved ViC, in its state *before* the addition of the component in Section 4.2.2, running while cutaway operations were continuously performed on the most heavily fragmented of the human anatomy systems – the muscles. The purpose of this benchmark was to determine ViC's stock performance while the prototype's principal operation – cutaway – was utilized.

MULTI-THREADED CUTAWAY

This benchmark evaluated ViC, in its state *after* the modifications in Section 4.2.2, to determine whether or not the non-component based extendibility of *ParallelTasks* had an intended (positive) effect on ViC's performance. As with ***single-threaded cutaway***, cutaway operations are performed continuously on the muscle system.

The three benchmarks were run for 10 seconds each, capturing 10 snapshots of the frame-rate. The benchmarks were performed on a PC with the following specifications: Intel i7 930 CPU, 6GB of RAM, GeForce GTX 480 GPU, and Windows 7 64-bit OS.

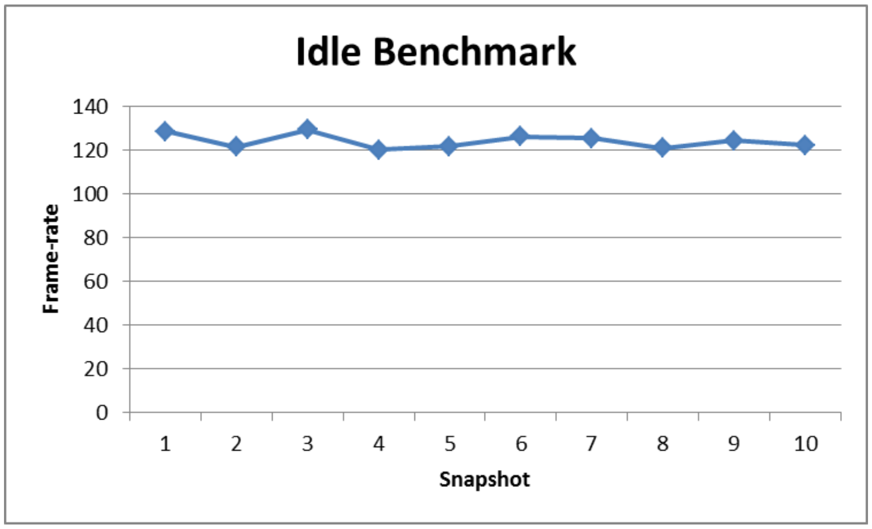


Figure 52: Interval data of the *idle* benchmark

With an average of 124.07 frames per second, the *idle* benchmark indicates that ViC was running at more than twice the locked frame-rate. This proves that ViC's stock performance is well beyond the required minimum in order to run at 60 frames per second. However, this test does not address the question of how much room for extendibility remains as the principal functionality, cutaway, is not utilized.

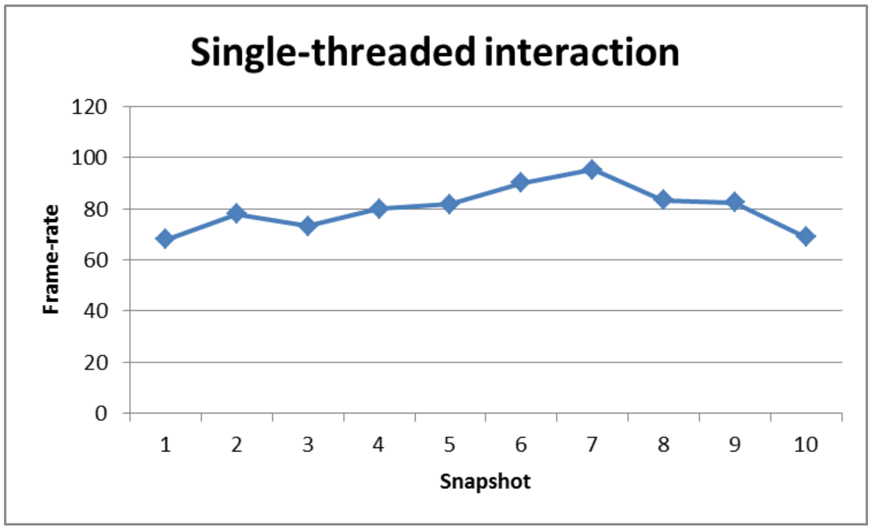


Figure 53: Interval data of the *single-threaded cutaway* benchmark

With cutaway in action, the *single-threaded cutaway* benchmark highlighted a drop in performance. While the average frame-rate in this benchmark was still higher than the minimum required, a loss of an average of over 40 frames per second with during continuous cutaway, indicates that the collision component was consuming a significant amount of CPU time. Furthermore, this means that ViC's performance is likely to degrade even more with a sharp increase in the fragment count or granularity.

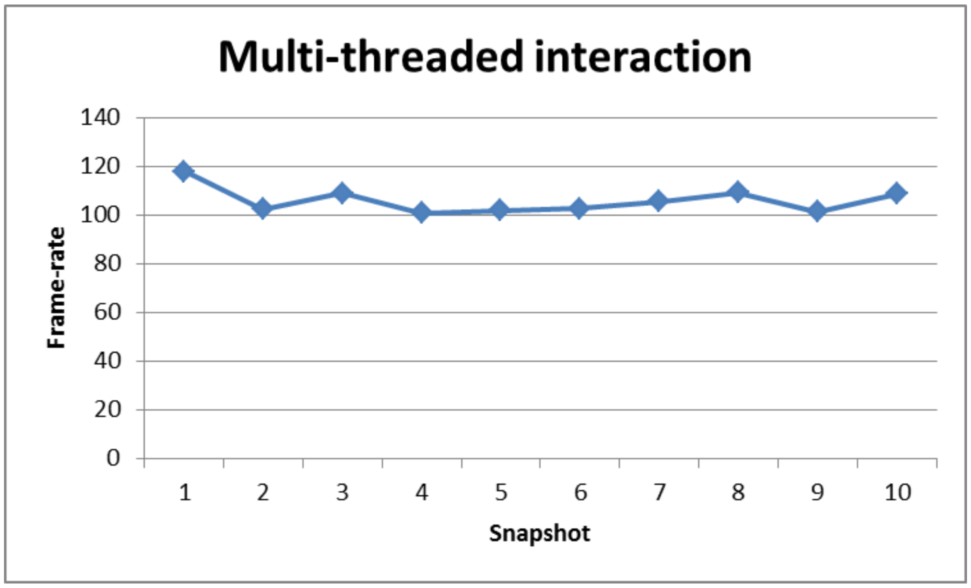


Figure 54: Interval data of the *multi-threaded cutaway* benchmark

Utilizing the extended functionality documented in Section 4.2.2, the *multi-threaded cutaway* benchmark shows that parallelization of the collision checking algorithm from Section 3.1.5, increased ViC's performance by an average of over 25 frames per second. With the final average frame-rate of 106.1 frames per second, the extended version of ViC was performing well above the minimum required performance.

Chapter 5: Summary, Conclusion and Future Work

With the advent of powerful graphics processors, medical professionals are now able to view and manipulate highly detailed renderings of human anatomy data. Whether it is for the purpose of medical education or surgical simulation, interactive medical visualization systems can provide capabilities that are impractical or expensive on a real cadaver.

As discussed in Chapter 1, just rendering human anatomy data is only part of medical professionals' needs. The visualization of 3D data must be accompanied by powerful contextual view capabilities and interaction controls. The controls must be constrained, effectively simplifying what would otherwise require complex user input actions. The combination of these powerful controls and photorealistic rendering of the data should create meaningful visualizations and operations that are analogous to real-world procedures.

In addition to the aforementioned technical requirements of a virtual cadaver system, such a system must also be designed in such a way that nothing beyond affordable mainstream hardware would be required to run it. With real-time performance a key element in the user experience, the system must effectively harness the readily available compute and rendering power of mainstream hardware.

Finally, for a system to offer longevity even beyond its original intended purpose, it must also be extendible. Whether by exposing an API, or offering a feature of in-application expandability or even exposing its source code, the term *system* is reserved only for a program that offers such capabilities.

5.1 Conclusion

This thesis presented ViC, a real-time, photorealistic, high performance human anatomy visualization system that offers interaction capabilities designed to mimic operations on a real cadaver. In addition to advanced, touch screen based interaction using natural gestures, ViC also features highly realistic data rendering capability by using state-of-the-art game engine technology. ViC is designed to be fully extendible through two different approaches. By following its component-based model, additional functionality can be placed into a self-contained module that follows the execution flow of ViC's foundation, XNA. This has the added advantage of complete automation of such component's code execution. Secondly, in situations where following XNA's execution model is not possible, additional functionality can be placed into a class of any design which can access data and call ViC's stock functionality on-demand. Performance of ViC both before and after extendibility was evaluated to ensure that the system runs at a real-time rate of a minimum of 60 frames per second.

5.2 Future Work

Currently ViC's interactive area is limited to the right arm. All meshes in the arm have been fragmented, including bones, muscles, arteries, veins, nerves, and the lymphatic system. Once the entire data set has been fragmented, ViC's cutaway functionality must be verified for all fragmented meshes. In addition, the end cap regions of a solid mesh fragment (such as the muscle or bone) are currently rendered using a solid color. These end caps must be more realistically rendered using a texture image and shading effects such as edge shading.

Despite the very nature of ViC as an extendible system, a number of improvements to the current prototype can be made particularly in its interaction capabilities. The collision component currently performs at high-speed but its exhaustive search collision testing should be replaced with a more efficient component, such as a physics engine. Physics engines offer more complex capabilities including high-performance collision checking through spatial subdivision optimizations such as quad-trees [45], as well as more realistic features such as soft-body simulations. This improvement will become even more important once the entire data set is fragmented.

Taking advantage of more advanced physics capabilities can also potentially provide further useful interaction operations that mimic interactions with a real cadaver. For example, the incorporation of soft body physics would allow for an interaction operation such as a pull with a medical instrument to elastically move tissue out of the way in order to see hidden tissue.

Finally, the preprocessing program itself should be incorporated into ViC's content pipeline or even the runtime of ViC. This merged system could then expedite or even inspire the development of new types of operations directly in ViC's rendering context.

REFERENCES

[1] Ravindran, S. (2011, June 21). *Stanford students use new virtual dissection table to study anatomy*. Retrieved July 3, 2011, from Contra Costa Times: http://www.contracostatimes.com/science/ci_18319433?nclick_check=1

[2] Google Labs. (2010). *Google Body*. (Google) Retrieved July 14, 2011, from http://bodybrowser.googlelabs.com/

[3] Dyck, J., Pinelle, D., Brown, B., & Gutwin, C. (2003). Learning from Games: HCI Design Innovations in Entertainment Software. *Proceedings of Graphics Interface*, (pp. 237-246). Halifax, Nova Scotia.

[4] Forlines, C., Wigdor, D., Shen, C., & Balakrishnan, R. (2007). Direct-touch vs. mouse input for tabletop displays. *Proceedings of the SIGCHI conference on Human factors in computing systems*, (pp. 647-656). New York, NY.

[5] Microsoft Corporation. (n.d.). *XNA Game Studio 4.0* . (Microsoft) Retrieved August 26, 2011, from http://create.msdn.com/en-us/resources/downloads

[6] Synapse Gaming LLC. (n.d.). *XNA Game Development and Engine Technology*. (Synapse Gaming) Retrieved August 26, 2011, from http://www.synapsegaming.com/

[7] Čmolík, L. (2008). Relational Transparency Model for Interactive Technical Illustration. *Proceedings of the 9th international symposium on Smart Graphics*, (pp. 263-270). Berlin, Heidelberg.

[8] Diepstraten, J., Weiskopf, D., & Ertl, T. (2002). Transparency in Interactive Technical Illustrations. *Computer Graphics Forum*, *21(3)*, pp. 317-325. Oxford.

[9] *VOXEL-MAN - Surgery Simulators and Virtual Body Models*. (n.d.). (University Medical Center Hamburg-Eppendorf) Retrieved August 26, 2011, from http://voxel-man.de/

[10] Pohlenz, P., Gröbe, A., Petersik, A., Von Sternberg, N., Pflesser, B., & Pommert, A. (2010). Virtual dental surgery as a new educational tool in dental school. *Journal of Cranio-Maxillofacial Surgery, 38* (8), 560-564.

[11] Hacker, S., & Handels, H. (2009). A framework for representation and visualization of 3D shape variability of organs in an interactive anatomical atlas. *Methods of information in medicine, 48* (3), 272-281.

[12] Reddy-Kolanu, G., & Alderson, D. (2011). Evaluating the effectiveness of the Voxel-Man TempoSurg virtual reality simulator in facilitating learning mastoid surgery. *Annals of the Royal College of Surgeons of England, 93* (3), 205-208.

[13] Drebin, R. A., Carpenter, L., & Hanrahan, P. (1988). Volume rendering. *SIGGRAPH Comput. Graph., 22* (4), 65-74.

[14] *VOXEL-MAN Gallery*. (n.d.). (University Medical Center Hamburg-Eppendorf) Retrieved August 26, 2011, from http://www.voxel-man.de/gallery/

[15] *VOXEL-MAN - Virtual Body Models*. (n.d.). (Voxel-Man) Retrieved August 26, 2011, from http://www.voxel-man.de/3d-navigator/

[16] Li, W., Ritter, L., Agrawala, M., Curless, B., & Salesin, D. (2007). Interactive cutaway illustrations of complex 3D models. *ACM Transactions on Graphics, 26* (3), 31-40.

[17] Gooch, B., Gooch, P., Shirley, P., & Cohen, E. (1998). A non-photorealistic lighting model for automatic technical illustration. *Proceedings of ACM SIGGRAPH 98*, (pp. 447-452). Orlando, FL, USA.

[18] Kirsch, F. (n.d.). *OpenCSG - The CSG rendering library*. Retrieved August 26, 2011, from http://opencsg.org/

[19] *Cyber Anatomy Med*. (n.d.). (Cyber Anatomy) Retrieved August 26, 2011, from http://www.cyber-anatomy.com/product_CAHA.php

[20] *Unreal Technology*. (n.d.). (Epic Games, Inc) Retrieved August 26, 2011, from http://www.unrealengine.com/

[21] Morris, C. (2002, June 3). *CNN Money - Your tax dollars at play*. (CC) Retrieved April 5, 2011, from http://money.cnn.com/2002/05/31/commentary/game_over/column_gaming/

[22] *Zero Hour: America's Medic*. (n.d.). (The George Washington University and The George Washington University Medical Center) Retrieved August 26, 2011, from http://zerohour.nemspi.org/

[23] *Crytek | MyCryENGINE*. (n.d.). (Crytek) Retrieved August 26, 2011, from http://mycryengine.com/

[24] *OGRE – Open Source 3D Graphics Engine*. (n.d.). Retrieved August 26, 2011, from http://www.ogre3d.org/

[25] *Visual C++*. (n.d.). (Microsoft) Retrieved August 26, 2011, from http://msdn.microsoft.com/en-us/library/60k1461a.aspx

[26] *Code::Blocks*. (n.d.). Retrieved August 26, 2011, from http://www.codeblocks.org/

[27] Hayward, K. (n.d.). *Volume Rendering 102: Transfer Functions*. Retrieved September 9, 2011, from Graphics Runner: http://graphicsrunner.blogspot.com/2009_01_01_archive.html

[28] *VTK - The Visualization Toolkit*. (n.d.). (Kitware) Retrieved August 26, 2011, from http://www.vtk.org/

[29] Schroeder, W., Martin, K., & Lorensen, B. (1993). *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics.* Kitware.

[30] *VTK - The Visualization Toolkit*. (n.d.). Retrieved September 4, 2011, from http://www.vtk.org/VTK/project/imagegallery.php

[31] *Blender*. (n.d.). (Blender Foundation) Retrieved August 26, 2011, from http://www.blender.org/

[32] *3ds Max - 3D Modeling, Animation, and Rendering Software*. (n.d.). (Autodesk) Retrieved August 26, 2011, from http://usa.autodesk.com/3ds-max/

[33] Calo, T. F. (n.d.). Retrieved September 9, 2011, from Tucho online portfolio: 3dart_wanted_gallery: http://artbytuchoweb.blogspot.com/2009/11/3dartwantedgallery.html

[34] *MATLAB - The Language Of Technical Computing*. (n.d.). (MathWorks, Inc) Retrieved August 26, 2011, from http://www.mathworks.com/products/matlab/index.html

[35] *ImageVis3D*. (n.d.). (NIH/NCRR Center for Integrative Biomedical Computing) Retrieved August 26, 2011, from http://www.sci.utah.edu/cibc/software/41-imagevis3d.html

[36] Gough, J. (2005). Virtual machines, managed code and component technology. *Proceedings of the Australian Software Engineering Conference, ASWEC, 2005*, pp. 5-12. Washington, DC, USA.

[37] cgshape. (n.d.). *Human Male Anatomy - Body, Muscles, Skeleton, Internal Organs and Lymphatic*. Retrieved September 8, 2011, from TurboSquid: http://www.turbosquid.com/3d-models/human-male-anatomy---3d-model/584511

[38] Mittring, M. (2007). Finding next gen: CryEngine 2. *ACM SIGGRAPH 2007 courses* (pp. 97-121). San Diego: ACM.

[39] Möller, T., & Trumbore, B. (1997). Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools, 2* (1), 21-28.

[40] *Windows Touch: Developer Resources*. (n.d.). (Microsoft Corporation) Retrieved September 4, 2011, from MSDN: http://archive.msdn.microsoft.com/WindowsTouch

[41] Schmidt , H. W., Crnkovic , I., Heineman , G. T., & Stafford, J. A. (2005). A Hybrid Component-Based System Development Process. *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, (pp. 152-159). Washington, DC, USA.

[42] Taj Books Ltd. (2004). *Atlas of Anatomy*. Burnaby: Giunti Editorial Group.

[43] Hargreaves, S. (2007, June 8). *Displaying the framerate*. Retrieved August 2, 2011, from Shawn Hargreaves Blog - Game programming with the XNA Framework: http://blogs.msdn.com/b/shawnhar/archive/2007/06/08/displaying-the-framerate.aspx

[44] *ParallelTasks Threading Library for Windows and Xbox360*. (n.d.). Retrieved September 9, 2011, from CodePlex - Open Source Community: http://paralleltasks.codeplex.com/

[45] Tang, B., & Miao, L. (2010). Real-Time Rendering for 3D Game Terrain with GPU Optimization. *Proceedings of the 2010 Second International Conference on Computer Modeling and Simulation*, *Vol. 1*, pp. 198-201. Washington, DC, USA.