2008

A LAYERED FRAMEWORK FOR SURGICAL SIMULATION DEVELOPMENT

Timothy James Hayes
Western University

A LAYERED FRAMEWORK FOR SURGICAL SIMULATION DEVELOPMENT

(Spine title: A Layered Framework for Surgical Simulation Development)

(Thesis format: Monograph)

by

Timothy James Hayes

Graduate Program in Engineering Science

Department of Electrical and Computer Engineering

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Engineering Science

School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Joint-Supervisor	Examiners
Dr. Rajni V. Patel	Dr. Mike Katchabaw
Joint-Supervisor	Dr. Hamada H. Ghenniwa
Dr. Luiz F. Capretz	Dr. Hanif Ladak

The thesis by

Timothy James Hayes

entitled:

A Layered Framework for Surgical Simulation Development

is accepted in partial fulfillment of the
requirements for the degree of

Master of Engineering Science

Date____________________

Chair of the Thesis Examination Board
Dr. Vijay Parsa

Abstract

The field of surgical simulation is still in its infancy, and a number of projects are attempting to take the next step towards becoming the *de facto* standard for surgical simulation development, an ambition shared by the framework described here.

Dubbed AutoMan, this framework has four main goals: a) to provide a common interface to simulation subsystems, b) allow the replacement of these underlying technologies, c) encourage collaboration between independent research projects and, d) expand the on targeted user base of similar frameworks.

AutoMan's layered structure provides an abstraction from implementation details providing the common user interface. Being highly modular and built on SOFA, the framework is highly extensible allowing algorithms and modules to be replaced or modified easily. This extensibility encourages collaboration as newly developed modules can be incorporated allowing the framework itself to grow and evolve with the industry. Also, making the programming interface easy to use caters to casual developers who are likely to add functionality to the system.

Keywords: Surgical Simulation, Simulation Development Framework, Virtual Reality

Acknowledgements

First and foremost, I must acknowledge the guidance, and patience of my supervisors Dr. Rajni Patel and Dr. Luiz Fernando Capretz. Without the opportunities they provided and the assistance along the way, this thesis would never have seen the light of day.

To Mr. Kenn Sippell and Mr. Craig Follett who helped me get started on the right foot. And to Mr. David Allison for listening to constant ranting and providing much needed input along the way. And to my friends and family far and wide whose love and friendship kept me going through the years.

Everything I accomplished here is due to these people and the support they provided. I am indebted to you all.

Contents

List of Tables

List of Figures

Chapter 1 Introduction

The benefits of training simulations are truly undeniable. Some of the obvious benefits include the ease of restarting, the low cost of running, and the increased safety of both the user and the equipment. All have been well documented and there is little need for them to be presented again here. These benefits fueled the development of flight simulators from their infancy 100 years ago to such a level of fidelity and in turn, such a level of acceptance that the Federal Aviation Administration requires that all of its pilots pass a simulated flight test on an annual basis [1].

These same benefits are fueling similar developments and advancements in the area of surgical simulation. Unfortunately, the simulation of surgery is far more complicated than that of flying [1]. This high degree of complexity is one of the major challenges that surgical simulations need to surmount to become commercially viable and eventually become accepted and used as common practice. A development framework that provides implementations of these complex algorithms could really expedite the development of all types of surgical simulators. At the very least, this would reduce project duration and cost.

This is not a new idea, many research projects [2], [3], [4], [5] have attempted to take the first real step towards a *de facto* framework for developing high end simulations. Although each has provided a unique take on the problem, none have yet achieved the ultimate goal. Each has their own benefits and potential failings, but all have supplied valuable insight into how such a framework should be developed. The framework presented in this thesis does not claim to have reached this ultimate goal, and is more an attempt to improve upon some of the potential failings of these other projects. By improving on such failings as low usability, and the lack of virtual reality support, AutoMan attempts to help the field of surgical simulation development take one step closer to its holy grail.

This introductory chapter provides an overview of a simulation development framework dubbed AutoMan. AutoMan is named for Automatic simulation Manager since it uses dedicated managers to control the main parts of the simulation[1]. The following section presents the aims of the thesis and of AutoMan and is followed by some background information that will be helpful in understanding the rest of the thesis. Also highlighted in this chapter are a number of similar projects, this should provide some insight into the current state of the industry and what is available. It should also lay the foundations for many of the requirements and design decisions that are presented in the subsequent chapters. This chapter ends with an outline of the organization and the content of the thesis.

[1] AutoMan is built upon the SOFA framework, the pun was just too fitting to resist.

1.1 Thesis Aims

On the grand scale, there are four main goals of the AutoMan framework:

- Provide a common interface to a variety of libraries and engines for simulation development.
- Allow the addition and replacement of components as new and improved algorithms are developed.
- Encourage collaboration among different research projects and allow the incorporation of these projects.
- Expand the targeted user base of these systems.

Before these grand goals can be realized, a number of base requirements for the simulation development framework must be met. These base requirements will not necessarily make a simulation development framework successful, but their omission will undoubtedly result in failure. These include things like real-time performance, a method of input and output, and high fidelity model representations and interactions. These represent many of the specific requirements that AutoMan has been designed to provide. However, rather than attempting to build a single all-encompassing framework, it is wise to make use of a number of technologies that are dedicated to specific tasks. In general, this will increase the quality of the system [2].

To make simulation development easier for the user, AutoMan provides a common interface to these libraries and technologies. The developer of the simulation is only required to learn the AutoMan interfaces and does not need to worry about the implementation details of the specific libraries. This opens many new technologies to developers that may only have expertise in certain areas of simulation development.

However, providing an interface to common libraries raises a problem. The state of the surgical simulation industry is thriving. On a very regular basis, new algorithms are developed, each improving on its predecessor. If a framework is built on specific libraries, it will be become deprecated as soon as any improvement is made to its underlying technologies. Considering the regular rate of algorithm improvement, this is unacceptable for a development framework such as AutoMan. To try and counteract this tendency, AutoMan is designed to be highly extensible and modular, allowing new modules and algorithms to be added and old ones replaced. Furthermore, incorporating new algorithms does not require any change to the existing simulation code built within the framework. This can allow any simulation built within AutoMan to ride the wave of innovation; constantly on the leading edge of simulation technology.

By allowing the different underlying technologies to be replaced, AutoMan inherently encourages collaboration between different research projects. Instead of each project developing their own version of every component in the simulation, they can focus on only a small subset of them and use the components developed by other research endeavors. Focusing the efforts of a project to a smaller subset of goals tends to produce better results; do one thing well. When they are finished, the algorithms can be integrated into AutoMan and simulations built therein will benefit from the endeavors of all of the external research projects.

However, beyond targeting research projects, AutoMan attempts to also cater to wider audience. There is a lot to be said for the users who develop on their free time. This is an untapped resource in terms of surgical simulation development. Already there are many developers who write add-ons to their favorite video games and even create their own games from scratch. These kind of casual developers have a very wide range of skills and have the expertise to improve the world of surgical simulation. They provide a user base for the existing technology, which in turn fuels the continual research of new algorithms, but they also can directly help the industry by developing new algorithms themselves.

Furthermore, should an active user community grow from the AutoMan project, it can be self delegating. The community will continue to develop simulations in the framework providing examples of its use. It will continually not only develop new algorithms and modules for the framework, but also support them. The community will generate its own documentation for new modules, even tutorials on how to use it.

This sort of community driven development is the main reason that many open source initiatives are thriving. Projects like Ogre3D [6], a graphical rendering engine, has this same community of developers that continually provide new tutorials and examples of the engine's use as well as add-ons that increase the functionality of the engine (for example, OgreODE). Simulation development should not be left solely to the academic community, in fact academic projects may significantly benefit from these casual developers.

The AutoMan Framework has been designed with these main goals in mind. However, its purpose still remains to help developers to create high-end simulations, like those of surgery. To that end it has a number of more focused requirements to meet, such as:

- Loading model geometry.
- Representing various types of models, both physical and visual.
- Simulating realistic physics and object interactions.
- Displaying the simulated scene.
- Manipulating the simulated scene.

These are the basic requirements that every simulation will likely need. The AutoMan Framework strives to meet these base functional requirements yet also meet the constraints of extensibility, accessibility, and community as described above.

1.2 Related Work

There have been a number of similar endeavors whose goal is a framework for simulation development. This section of the thesis presents a few of these projects and highlights a number of potential benefits and failings of each. The projects described here are either used in AutoMan to provide some functionality, or at the very least have provided an example that helped at various phases of this project.

1.2.1 SOFA

SOFA aims to provide an extensible framework in which algorithmic modules can be exchanged between research groups [7]. This allows for distribution across a number of development teams and greatly improves the extensibility and scalability by allowing new and improved algorithms to be easily incorporated. This extensibility is one of the main reasons that SOFA was used as the backbone for the development of AutoMan.

SOFA also uses multiple representations of a single simulated object that allows different components to use the representation that best serves its purpose. For example, a high resolution polygonal surface would be used as the visual model while a mass-spring or FEM model would be used as the behavior model [7]. Furthermore, it provides a scene graph loader that will load all of the simulation's information via a single XML file [5].

SOFA provides a number of features that are common with the requirements of this thesis. This made SOFA a favorable choice on which to build AutoMan. On top of this, while designing this framework SOFA had been getting a lot of attention and it was thought that it could be the first to really take off as the basis for all surgical simulation endeavors thereafter.

1.2.2 VR Juggler

VR Juggler is not a surgical simulation framework; it is a "virtual platform for virtual reality application development" [8]. That is to say it provides everything that should be required to develop virtual reality (VR) applications. AutoMan uses the tools available with VR Juggler to provide a VR rendering option for any simulation developed in the framework. But beyond that, a number of cues in regard to the design of AutoMan have been extracted from the structure of the VR Juggler suite, which includes the application object concept for one.

Also, considering that VR Juggler is a very successful community oriented, open-source project, AutoMan has attempted to mimic a lot of the support structure provided with VR Juggler. However, in November 2003, Infiscape Corporation began offering commercial support and services for VR Juggler, and would later assume responsibility of the project altogether [8]. Having this commercial backing has greatly improved the quantity and quality of documentation and services available to its users. AutoMan does not have this backing, nor does it have the resources to provide the same caliber of support, but the design of AutoMan's support system has been greatly affected by that of VR Juggler and many other open source community projects.

1.2.3 Other Surgical Simulation Projects

This section describes a number of other simulation projects, SPRING, GiPSi, Hysteroscopy Simulator, and general game development engines. These frameworks have taken some important preliminary steps towards the development of a fully functional surgical simulation development framework. The intent of the AutoMan Framework is to serve as the next step towards this fully functional framework. Many cues have been extracted from these related works, which has heavily affected the design of AutoMan.

1.2.3.1 SPRING

SPRING [2], [9], [10] was one of the first concerted efforts to develop a general simulator. It strives, "to be a general simulator with a broad base of technological features and a broad range of potential applications, with the emphasis on real-time performance" [2]. SPRING provides a number of the basic modules required for high end simulation applications and they perform with a reasonable level of proficiency.

Actually, SPRING has implemented more than most other frameworks to date, including algorithms for piercing and cutting [9], [10]. However, there is a distinct lack of separation of components meaning low extensibility and makes it difficult to modify and maintain. Combined with a shortage of API documentation, and usage tutorials, development within this framework can be trying.

1.2.3.2 GiPSi

GiPSi, an open source/open architecture venture, focuses on "providing support for heterogeneous models of computation and defined API's for interfacing heterogeneous physical processes"[3]. This framework supports both mesh and point based geometry definitions and has placed a lot of focus on model representation. Generalized interfaces keep the representation independent of the modeling methods. However, GiPSi is not a complete simulation engine [3] and fails to support a number of features required by the AutoMan Framework.

1.2.3.3 HystSim

The Hysteroscopy Simulator (HystSim) is a highly modular framework that has placed great emphasis on parallelization and multi-threading. This improves scalability and extensibility while maintaining the real-time constraints [4]. This parallelism greatly affected the design of AutoMan and a few concepts in that regard have been incorporated into its design. HystSim also supports basic multi-representational models. However at the time that the AutoMan Framework was being developed, there was yet to be a stable release of this simulator.

1.2.3.4 Game Engines

Although not geared directly towards surgical simulation, video game engines in both the commercial realm and public domain were also considered. Many of these engines have served as development frameworks for some of the biggest and most advanced video games in the industry [6], [11], [12].

They do lack many of the key components required for surgical simulations, such as support for deformable models, and cutting simulation [4]. However, considering the widespread use of some of these frameworks, it would be foolish to not consider them. Their extensive documentation tends to draw a large user base and community which further increases the popularity. However, engines like Valve's Source [11] engine or ID's id Tech 4 [12] engine are not extensible. They are built to get the best results out of the current technology but it is difficult to incorporate new algorithms or libraries. This means that they need to be rebuilt every couple of years or so. The gaming industry has the resources to rebuild on a regular basis, but the world of surgical simulation does not have the same means. Medical companies are hesitant to spend money on developing training systems as they consider it "a cost without financial return" [1]. The field of surgical simulation needs a framework that can evolve with the technology; continually improving with each new advancement.

However, the main cues that have been extracted from video game engines refer to the architecture and interfacing used. These game engines are designed and built with usability in mind. This is achieved through things like incorporating concepts that map directly to the real world and thoroughly defining programming interfaces. On top of this, they are very well documented and have a healthy support system set up for their users. Since one of the main goals of AutoMan is to expand the user base of the surgical simulation technology, many of these concepts that increase the usability and support of the framework have been incorporated.

1.3 Outline of Thesis

This chapter has provided the reader with an introduction to the AutoMan Framework. It has also provided sufficient background information for the reader to fully understand the work that has gone into the development of the framework and the impact of its completion, all of which will become apparent in subsequent chapters.

The second chapter presents the foundations of the AutoMan framework by defining the targeted types of users and the overall functions and constraints on the system.

Chapter 3 presents the functional and quality of service requirements of the framework. This includes a definition of each module and a description of the system's operation.

Chapter 4 is a detailed look at the design and inner workings of each module within the framework. The presentation of this material is aided with UML diagrams and descriptions of the details therein. This will provide the reader an excellent understanding of the structure and of how many of the stated requirements have been met.

This is followed by a discussion of the implementation process that highlights some of the more interesting development issues and concerns. This chapter also presents the testing methodology and results, and showcases the framework in action.

The final chapter, Chapter 6, concludes the thesis by discussing the success of the project by reaffirming that the requirements set out at the beginning of the project have in fact been achieved. This chapter also presents a number of possible extensions to the framework.

Chapter 2 System Requirements

This chapter covers the main requirements of the AutoMan framework. It starts with a discussion of the types of users targeted, and moves onto a description of the functional and quality-of-service requirements. The information presented in this chapter starts to lay the foundation of the framework's design.

2.1 Targeted Users

AutoMan is primarily targeting software developers who are attempting to build simulations of complex activities, such as surgery. It is safe to assume that these users have some experience in developing simulations, and are generally familiar with the C++ programming language. It is also expected that they have experience using other 3D development engines whether it be other surgical simulation frameworks (such as Spring[2], or SOFA[5]) or video game and general rendering engines (such as Valve's Source Engine[11], or Ogre3D[6]). This type of user will utilize the framework to build specific simulation applications. It is expected that they will make little effort to advance the development of the framework itself. Their endeavors are primarily focused on the simulation being developed.

Although they are not the primary targets for the framework, there are two other sets of users who are still within the target market of the framework; casual users and advanced users.

Casual users include those who enjoy working with new technology, specifically the cutting edge of computer graphics and simulation. It is expected that they are knowledgeable in the C++ programming language and have a moderate understanding of real-time computer graphics. They have likely used a number of the other 3D development engines and are familiar with many of the common concepts. Furthermore, it can be assumed that they have a basic knowledge of Newtonian mechanics and the physics of rigid bodies; however it is unexpected that this expertise stretches into the areas such as the advanced physics of deformable models or fluid dynamics. To clarify, term "casual" is no indication of the expertise of the user, but is more an indication of the user's intent in using the framework. These users may be using the framework for entertainment purposes and may not be developing a specific simulation application. However, should these users really get interested in the software it is very likely that they will move into the advanced user category.

The set of advanced users are generally comprised of experienced real-time developers. They have extensive knowledge of simulation development and are experts in one or more facets therein, such as graphical rendering optimization, haptic feedback rendering, advanced physics simulation, etc. These users may not only use the framework to build simulations, but may add functionality to the framework itself. This may include developing new modules for it to use, or even modifying the framework core. Some example additions could include new rendering engines, deformable model representations, or separate tools that will aid development within the framework.

Table 2-1 summarizes the three types of users we expect to use the AutoMan framework. Throughout the rest of this text, the terms in this table will be used to denote the type of user in question.

Table 2-1 Types of Users

Type of User	Description
Normal User	A developer who has some simulation development experience and intends to create advanced simulations.
Casual User	A developer who may only have a basic understanding of the simulation development and wishes to use the framework for a more nominal application or entertainment purposes.
Advanced User	A developer with extensive knowledge in simulation development who will add to or modify the core components of the framework.

2.2 Main Functional Requirements

The next step was to define the specific operation and functionality requirements of the framework. This list of functional requirements was extracted from the features of similar development frameworks and from personal experience using these other frameworks. This list of requirements serves as a base on which the system can be designed and also serves as a method of validating the implementation once it is completed [13].

Figure 2-1 shows a business level use case diagram. Each use case represents a functional requirement or feature of the system. Please note that the functional requirements addressed in Figure 2-1 and throughout the rest of this chapter are only those that have a significant bearing on the architecture. For a complete list of the functional requirements and the full Use Case diagram, please refer to Appendix A.

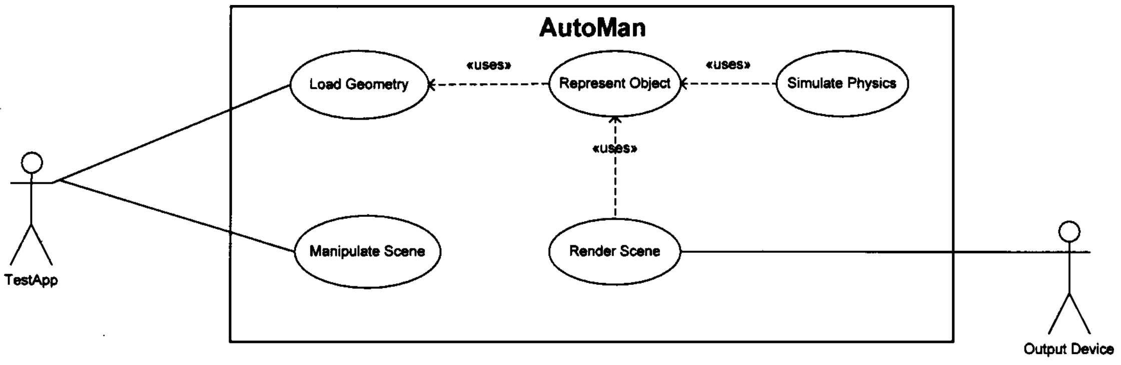


Figure 2-1 Business Use Case Diagram

2.2.1 Load Model Geometry

In order to create a simulation, there must be a method of defining the scene entities, especially geometrically. The easiest way for this to be achieved is by allowing the geometry to be created in another software package whose primary function is to generate 3D geometry, and then import this geometry into the simulation. Therefore the framework must have the ability to load CAD object models. The file format used for importing the object should be an available export option for most common CAD packages, including non-professional or open-source packages such as Blender [14].

Content creation is outside the scope of this thesis, nonetheless allowing users to create the geometric content in dedicated software packages will likely increase the quality of the content created. Furthermore, this facilitates the use of other methods of content creation, such as retrieving geometry from a physical model by the use of a 3D scanner or medical imaging equipment.

2.2.2 Model Representation

Model representation is a key component of AutoMan. The objects should be multi-representational models. This means that each acting object in the scene may have more than one representation. This may be a high resolution visual model for visualization, a lower resolution to help speed up the collision detection algorithms, and a physical model to define physical properties such as rigidity or elasticity.

Furthermore, considering that there is no standard algorithm that accounts for all deformable models, it must be possible to change the physical model's representation and solving algorithm. For instance, a Mass-Spring representation may be better suited for a purpose than an FEM representation.

2.2.3 Simulate Realistic Physics and Object Interactions

The physical models and algorithms should define how an object interacts within the simulation environment and with other objects within the scene. For instance, objects should fall due to gravity, collide with other objects, and adhere to all expected laws of physics.

This is a fundamental requirement for any simulation. In the area of medical simulation where complex physical models are required for simulating organs and tissue, meeting this requirement becomes difficult. But this is one of the main areas of simulation development that the AutoMan framework is intended to aid, therefore it must be able to handle this sort of model representation and calculations.

Also, as mentioned before, the use of the multi-representational models allows the collision and physical models to be separate from the visualization. This means that the physics algorithms have dedicated representations to help ease the computations.

2.2.4 Render Scene

First and foremost, the simulation must be able to give feedback to the user. Therefore the framework must supply at the very least a method of rendering the scene geometry to a regular monitor display. This visualization should constantly update as simulation state changes, thereby allowing the viewer to observe the scene's response to various forces and inputs.

On top of this, the system must also support the addition of other methods of rendering the simulation. If desired, an advanced user should have the ability to add rendering engines that use other devices, such as speakers for audio feedback, or a haptic device for force feedback. These users must have the ability to easily define a rendering engine that will instruct the framework as to how to use the device.

2.2.5 Manipulate Scene

There must also be a method for the user to manipulate the simulated scene via device inputs. This may include moving objects through the scene space, moving the camera through the scene, changing simulation options, and exiting the simulation.

If so desired, the developer must be able to define what actions are performed as an input device is activated. This allows the developer to define specific controls on a per simulation basis.

2.3 Non-Functional Requirements

The definition of non-functional requirements is the next improtant step required in developing a framework such as AutoMan. Non-functional requirements do not define the behavior of the system, but rather define quality of service [15]. They define the constraints under which the functionalities, as defined in the previous section must be performed. This includes the ease of use, scalability, performance, and accessibility of the functions.

This section describes these constraints and suggests methods of achieving the desired quality of service. Further details on these requirements may be found in Appendix A.

2.3.1 Usability

Usability primarily deals with the effort required to learn and operate a piece of software[13]. To ensure that AutoMan meets this requirement, a comprehensive API and tutorials should be supplied. This includes the programming interfaces, UML documentation, and coded examples. Verbose commenting of all code, especially the example applications, should be incorporated to aid the developer understand each feature's structure and function.

To also aid in this respect, the framework should use generic, common interfaces to the framework. Common coding standards and practices will round out the learning curve thus reducing overall development time.

2.3.2 Extensibility

As hinted at in the functional requirements, the framework should have an easy method upgrading and expanding. This defines the framework's level of extensibility. Should developers want to add features to the framework, a simple, easy method should be provided along with a method of distributing the addition to the other users of the framework. These interfaces should be well documented to encourage users to develop the additional modules.

2.3.3 Performance

There are a lot of factors that affect the performance of a simulation. However, the two main factors are the computational power of the system running the application, and the complexity of the simulation.

The first of the two cannot be attended to in software, but it is something that requires thought during the design of the framework. Simulations developed in the framework, should be able to run the sample applications at a real-time frame rate on most workstation-grade computers with a high-end graphics processing unit (GPU). The performance may be enhanced by running the simulation on more powerful hardware.

The second of the two major factors can be addressed in software, at least to an extent. As hinted at in the requirements section, the user must be allowed to define a lower quality model for use in the collision detection and response algorithms. These are algorithms with a high degree of computational complexity and really hinder the performance of a simulation. Allowing them to be run against simplified versions of the object models, reduces the computations required and ultimately increases the performance of the system.

2.3.4 Accessibility

To ensure that the framework has a high degree of accessibility, a web site devoted to the framework should be available. Here users will be able to download the software binaries for immediate use, as well as view code samples, tutorials, and API documentation.

Also, the site should provide a method for users to download the source code of the framework allowing them to directly modify the core of the framework. This is how the advanced users can make modifications and upgrades.

Chapter 3 The AutoMan Framework for SOFA

As stated in the introduction of this thesis, the main goals of this framework are to:

- Provide a common interface to a variety of libraries and engines for the medical community.
- Allow the addition and replacement of components as new and improved algorithms are developed.
- Encourage collaboration among different research projects and allow the incorporation of these projects.
- Expand the targeted user base of these systems.

But even beyond these grand goals, there are a number of more basic requirements that must be first met. These basic functionalities were described in the previous chapter. With this list of required functionality and quality of service, the analysis and design phase of software development lifecycle may progress. This chapter discusses this phase of the AutoMan project by describing the framework architecture and operation.

3.1 Framework Architecture

One of the primary reasons that the development of surgical simulations is so difficult is that many common features of a surgical simulation still require a great deal of expertise from the developer to implement, both in terms of mathematical theory and in terms of environment nuances. A framework for surgical simulation development should provide a layer of abstraction between the developer and the low level implementation details of the simulation. This not only unifies the development environment but allows users with limited expertise to develop complete simulations. Figure 3-1 shows an overview of this structure. Please keep in mind that this is not a technical diagram; it serves to show the layered organization of the framework.

The Simulation Logic block represents the code that the user of the framework will implement. This should contain all of the simulation specific information such as which models to load and where to find them, what kind of feedback the user wishes to receive, etc. It resides in the upper most layer, meaning that it only will have direct contact with the layer directly below; the AutoMan framework.

AutoMan functions as the middle layer of the structure. Aside from providing its own functionality, it provides the mapping between the action that the simulation code wishes to perform and the implementation in a software library.

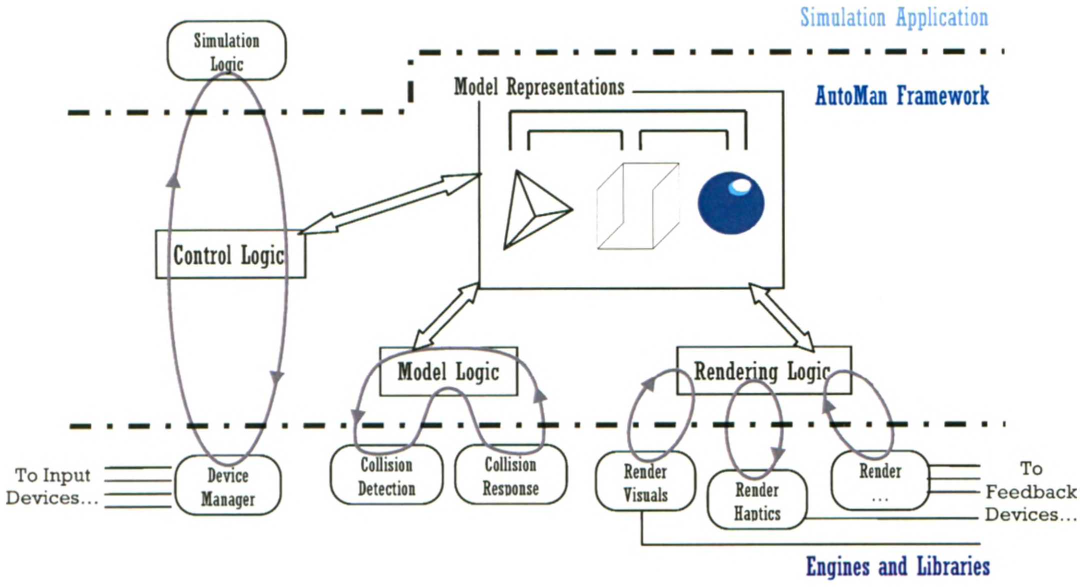


Figure 3-1 Overview of the AutoMan Framework

The lower most layer contains all of the implementations of the algorithms to be used during the simulation. Here specific software libraries and engines are called by the AutoMan Framework to perform a specific task.

Since the user's simulation code is contained in the upper layer, it does not have direct contact with any code in the lowest layer. This means that the implementation specific details of each engine or library are completely hidden from the user. AutoMan handles all of the environmental nuances of these engines, thus providing a unified development environment for the user.

This abstraction reduces development time and avoids some of the risk of incorrectly initialized libraries. Furthermore, the simulation developer does not need to be knowledgeable in using the specific implementation libraries; the code the user writes is completely independent of the library that will be performing the actions.

3.1.1 Internal Architecture

Beyond the layered structure AutoMan is separated into three different main modules. Figure 3-2 shows the overall layout of the different modules in the framework.

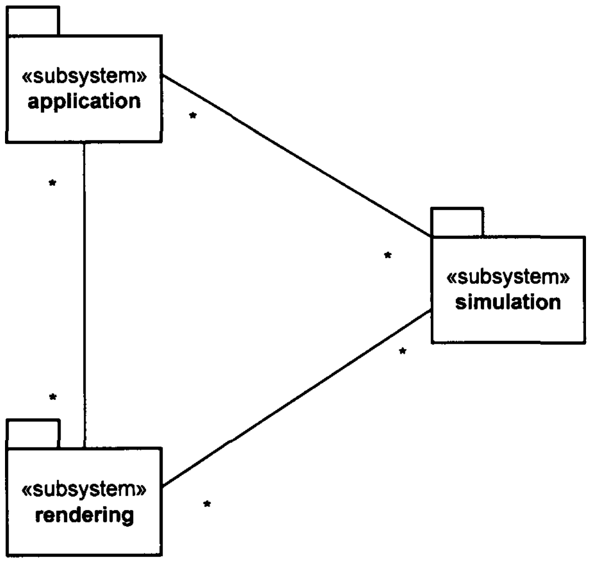


Figure 3-2 Package View

This overall structure takes after the Model-View-Controller software architectural model. The simulation package stores the state of the entire simulation and resembles the model archetype. The rendering package displays the state of the simulation and supplies it to the feedback devices, which is similar to the view archetype. The application module simulates the controller archetype in its role to manage the application as a whole.

Adhering to this multitier architectural model allows the framework to be easily modified and upgraded. It decouples the data from the application logic and the presentation allowing either of the modules to be modified without requiring changes to be made to any other [16].

3.1.1.1 Simulation Module

The simulation package stores the current state of the simulation. This includes all the information relating to the objects in the simulation. These include ambient objects that are only used for visualization and active objects that can be manipulated and follow the laws of physics.

It also encapsulates all of the processing logic of the simulation. All of the physics and collision detection / response algorithms are contained within this module.

3.1.1.2 Rendering Module

The main purpose of a simulation is to supply feedback to the user. Therefore it is important to provide a method for defining methods of doing so. This could include visualization, sound, networking interfaces, or force feedback for haptic devices. The term “rendering” is used to signify the act of sending specific information about the simulation state out through an interface. For instance a visualization rendering engine draws the objects of the scene to a display device, where an auditory rendering engine would provide auditory feedback in the simulation to the speakers. This allows the developers to choose between a few supplied rendering engines or develop their own for specific devices. For example, the developer may choose to make a rendering engine that renders the force exerted on an object to a haptic device.

This functionality is encapsulated in the rendering module. Everything that will send data out of the system will do so through this module.

3.1.1.3 Application Module

One part of simulation development that is often understated is definition of the application logic. Many libraries focus on the visualization component and leave the structure of the actual application up to the developer. Although this means that the developer has total freedom on how the application is structured, it often does not get the same attention. This may result in poorly designed applications that are difficult to modify and maintain. It also makes it nearly impossible for anyone except the original developer to make modifications, resulting in applications that go unmaintained, using deprecated technology and eventually going unused. A unified application structure eases this maintenance phase of the development cycle. Also, by supplying a sample application structure, the developer can focus on content and fidelity rather than spend valuable time on defining the application structure.

The application module provides this unified structure. It can be considered the control centre of the framework. It creates and initializes the entire application and starts the main loop execution. This ensures that the initialization and setup is performed correctly and efficiently. Also, all device input is handled in this module and propagated through the rest of the system, keeping with the controller archetype of the MVC architecture.

3.2 Framework Operation

This section outlines the different stages of execution during operation. This can be divided into the following stages:

- System initialization
- Main loop execution
- System termination

All applications follow these operation phases during the course of execution.

3.2.1 System Initialization

The system initialization stage consists of *initialization* and *scene creation*. These steps will be discussed in greater detail in Section 4.1.2.1 of this document.

During the *initialization* phase, each component is constructed and initialized. This includes each of the modules mentioned in the previous section. Once these are created and ready to perform their task, the scene to be simulated is generated.

The *scene creation* phase creates all of the scene objects that are to be simulated. This builds the scene that will be rendered to devices and modified throughout the course of the simulation's execution.

3.2.2 Main Loop Execution

The main loop runs the entire simulation. Here the state of the simulation is rendered through the defined rendering engines, input is handled, and the state of the simulation objects is updated. This is repeated until the simulation execution has ended. Again, this will be discussed in greater detail in Sections 4.1.3.2 and 4.1.3.3 of this document.

3.2.3 System Termination

This stage is invoked when the main loop is signaled to finish. The system termination phase destroys all the objects created, releases the allocated memory and terminates the execution threads. This ends the simulation application.

Chapter 4 Design and Documentation Framework

This chapter presents the design of the AutoMan framework. Included are many UML diagrams that will help to explain the structure and operation of the different modules within the system. Along the way, the rationale behind many of the design decisions should become apparent. The chapter ends with description of the supporting documentation available for users of the framework.

4.1 Module Design

The previous chapter covered many of the requirements for each module and hinted at many items that should be considered during the design. It was about this stage of the software development lifecycle that it was decided SOFA would serve as a solid backbone for the AutoMan. The reasons behind this decision will become apparent through the design presented in this section.

4.1.1 Simulation

Adhering to the MVC structures; the simulation package stores all information relating to the state of the objects in the simulation (geometry, position, orientation, velocity, rigidity, etc). Furthermore, it handles the model logic such as the physics algorithms and collision detection / response algorithms. Since SOFA already has a package that very closely mirrors these requirements, it was a logical step to base AutoMan's version on the existing SOFA module.

However, the SOFA simulation objects are not easy to work with. For a single entity in the simulation, many different classes needed to be instantiated and added to the scene. Figure 4-1 shows the object graph, each box represents a class that needs to be created to define two objects, a deformable cube and a static floor. With AutoMan striving to make simulation development easier, requiring the simulation developer to create and configure every one of these classes for a single entity in the simulation was unacceptable. Therefore, the concept of a simulation object was extracted and defined in AutoMan's simulation module. A simulation object instantiates the SOFA classes automatically; the user only needs to create the object as defined in AutoMan. To reiterate, the cube defined in Figure 4-1 corresponds to a single class in AutoMan, and the floor is another single class. This single class creates all of the SOFA classes upon its own creation, completely hiding them from the simulation developer.

Figure 4-2 shows the static structure of AutoMan's simulation module. For the most part the module consists of the simulation object class inheritance tree. At the top of this tree is the `SimulationNode` class. This embodies the concept of a frame of reference within the scene. Other objects can be added as children to this node, meaning that they exist within this frame of reference. For example, if the parent node was translated a certain value, all of the objects within this node would also be translated this value. For those that are familiar with developing in SOFA, this is an extension of the `GNode` class.

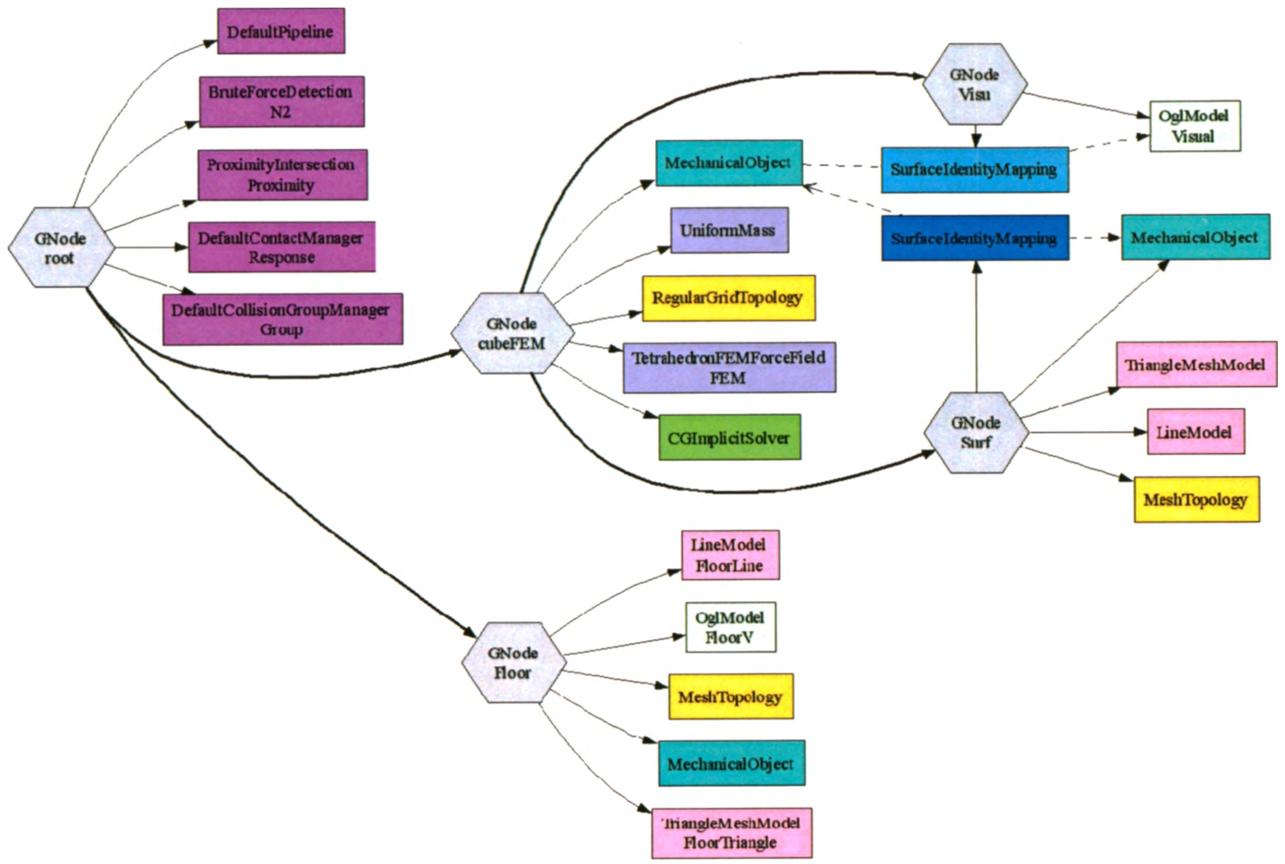


Figure 4-1 collisionTriangle.scn Object Graph

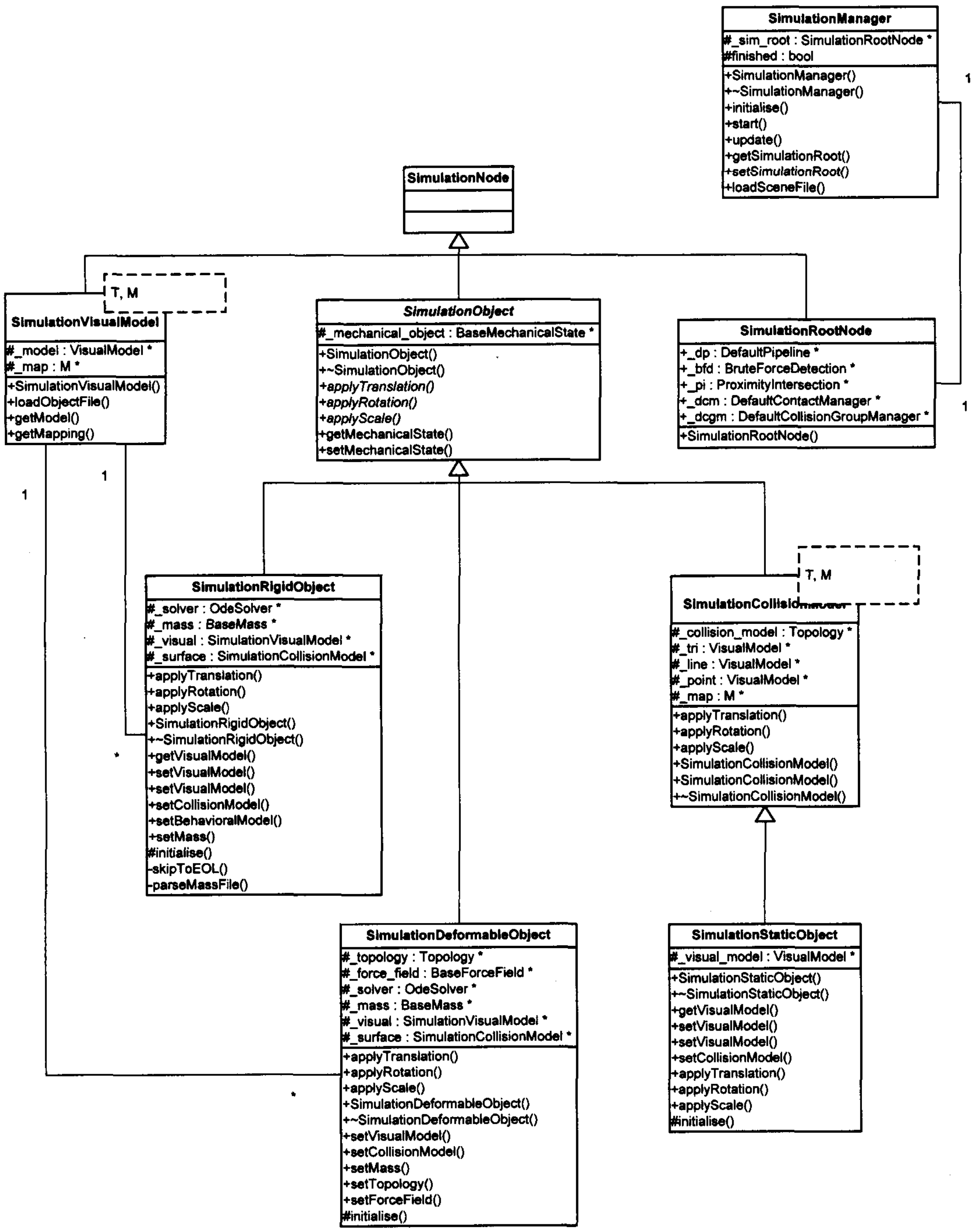


Figure 4-2 Simulation Package – Class Diagram

Inheriting from this `SimulationNode` class is the `SimulationRootNode` class. This class defines the root of the simulation and as such, it has a little more to do than a regular `SimulationNode`. This node automatically sets up the physics solvers for the scene, including gravity, and collision detection / response algorithms. Since it inherits from the `SimulationNode` class, gravity and other environmental physics set up here will affect all of the objects attached to this node. Considering this is the root node, the entire scene will be affected by the physics defined here.

The rest of the classes in this module are rather similar to each other so, for the sake of space and avoiding a rather dry reading chapter, the main function of each class is summarized in Table 4-1.

Along with this simulation object tree described above, there is also a manager that controls the state of the tree. This class is called the `SimulationManager`. Before every frame is rendered, the `SimulationManager` increments the simulation by a time step which updates the positions, orientations, velocities, etc., of all the objects in the scene. This is repeated every frame causing the scene to be animated.

Although at the time of the writing of this thesis, it has not been completed (please refer to section 5.1.2 for an explanation), a multithreaded version of the AutoMan framework is under development. In the multithreaded version, the

Table 4-1 Simulation Package – Object Tree Class Descriptions

Class Name	Description
`SimulationNode`	Scene node.
`SimulationRootNode`	Setup up scene physics and store simulation objects. Defines the root node of the scene.
`SimulationVisualModel`	Defines a visual model. It stores the geometry of an object that will be rendered to the screen. It does not store any physical properties so it is often mapped to a physical object.
`SimulationObject`	Defines a basic physical object. This class does not define any geometry, just a position and orientation in simulation space.
`SimulationCollisionModel`	Stores collision geometry to be used in collision detection algorithms. This geometry is not used for rendering and often is mapped to a visual model.
`SimulationStaticObject`	Defines an object that has visual geometry and physical geometry but cannot move within the scene. Other objects can collide and deform upon collision with this type of object, but this object will not change position or orientation as a result. Could be used as the floor of the scene.
`SimulationRigidObject`	Defines an object that has both visual and collision geometry and can move through scene space. It will be affected due to gravity and other object collisions.
`SimulationDeformableObject`	Defines an object that has the same properties as a `SimulationRigidObject`. However, the surface of this object will deform due to physical contact with another object or even under its own weight due to gravity.

`SimulationManager` will have a dedicated thread that will continually update the state of the simulation based on a specific time step instead of having to wait for each frame to be rendered before updating it again. This decoupling should improve the fidelity of the physics simulation by ensuring a constant time step that mimics real world movement rates [17]. If the system has trouble rendering the frame, the state of the simulation does not slow down with it.

4.1.2 Application

This module is considered the controller of the application following the MVC structure. It creates and initializes the entire application and starts the main loop execution. Furthermore, it provides a unified structure for applications using AutoMan. A few reasons for this were discussed in Section 3.1.1.3 of this document. Also discussed in a previous chapter (Section 2.1) were the types of users targeted by AutoMan and some of the features that each requires. This section discusses the design of this module.

Figure 4-3 shows the class diagram for the application module. It contains four classes but the current implementation only fully makes use of one, the `Application` class.

This class handles the entire system initialization phase of the program's execution. In a general program, the *main()* function defines the entry point [18]. This is true only to an extent in the case of an AutoMan application. The entry point is still the *main()*

function, but this method basically remains the same for any program made using AutoMan. Its sole purpose is to create the `Application` class and start the simulation.

The `Application` class is abstract. This means that it cannot be instantiated itself, because it is missing the definitions of one or more methods [19]. This class is created by *main()* and is an instance of a user created application. In Figure 4-3, the `TestApp_CollisionTriangle` class completes the implementation of the `Application` class by extending it. This means that it inherits the functionality of the parent class [19].

The two methods that `Application` does not provide definitions for are the *constructScene()* method, which loads the model geometry, and the *constructRenderingManager()* method, which tells the framework how to render the scene to a device. Reasons for this should be obvious since every simulation will have a different scene and will need to render that scene in some form or fashion. However, the default implementations for many of the other methods will not suffice for more than the simplest of simulations; it is expected that the user will override these methods and re-implement them.

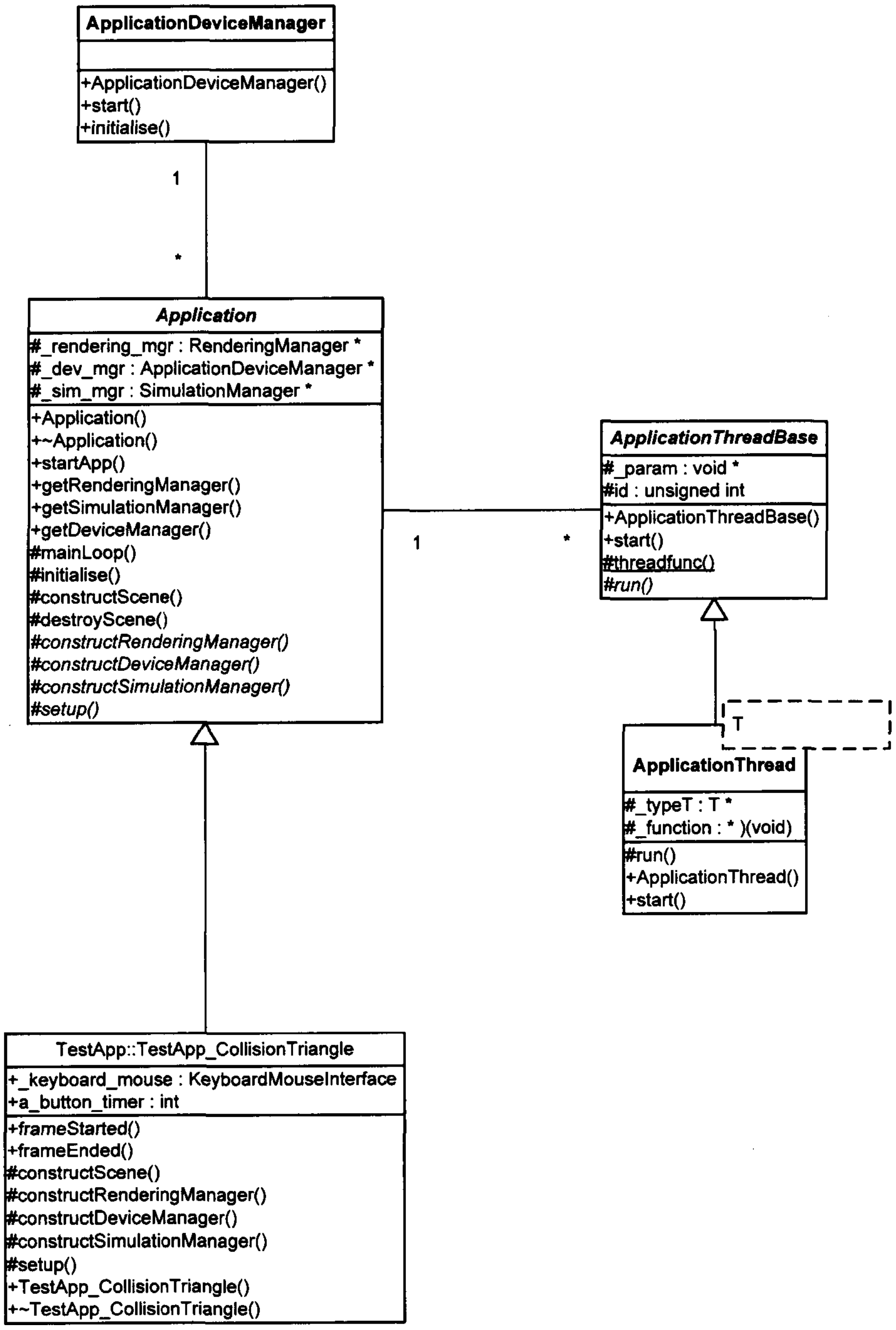


Figure 4-3 Application Package – Class Diagram

This allows casual users to develop simulations by only implementing the necessary functions, yet allows normal and advanced users to fully customize the framework for his or her specific simulation.

Looking at the other classes defined in this module, the `ApplicationDeviceManager` is empty at the time of writing this thesis. It is a placeholder class with only stubs of each method implemented. When completed, this class will provide a method of defining input devices in a similar manner to how the `RenderingManager` handles output devices.

Since there have been enormous advancements in tools for simulation, especially that of surgical simulation, the AutoMan framework should provide a method of allowing the advanced users to develop drivers for specific input devices. This should be similar to VR Juggler's Gadgeteer project [8].

The user will be able to choose from some supplied device drivers and add them to the `ApplicationDeviceManager`. The state of each device added will be monitored by the manager and it will perform the required actions as specified by the user.

This feature is a significant amount of work and it was deemed to be outside of the scope of this initial release of AutoMan. Currently, defining input devices is left to the simulation developer and can all be implemented within the user's code. So, although the AutoMan framework does not directly provide this feature, it only means that the user will need to implement it themselves. This way the user can implement all the input handling using whichever methods and tools they are familiar with.

The other two classes in this module provide a method of having a thread start execution on a member function of a class. The C++ process threads generally require a pointer to a global static method as the entry point. This would make multithreading a framework like AutoMan remarkably difficult. However, hiding this method inside a private abstract class, `ApplicationThreadBase`, and providing a template implementation, `ApplicationThread`, makes it possible [20]. This uses the concept of external polymorphism [21].

One need only create an instance of an `ApplicationThread` with a template parameter corresponding to the object that contains the method to be the thread's entry point. Then the *start()* method can be called when the thread is ready to be started.

These classes are the result of the cursory steps taken towards multithreading AutoMan and although they are unused within the framework at the time of writing, they are fully implemented and provide the user with the same threading functionality. Should the user want to manually multithread his or her simulation, a quick and efficient way to do so is supplied.

4.1.2.1 Application Initialization

The application module handles the system initialization phase of the program's operation. Figure 4-4 shows the method calling events during this phase.

The phase starts with a call to *startApp()*, which is a member function of the `Application` class. This starts the creation and initialization process of the application. The first task is to create a `SimulationManager` which stores the root node of the simulation. This root node will store all of the scene objects.

After the root node is created, the `RenderingManager` is created along with a `RenderingEngine`. In this case, the engine chosen is the `VRJugglerRenderingEngine` but this could be a `SofaQtRenderingEngine` or a user defined `RenderingEngine`, the operation is the same. A `RenderingEngineFrameListener` is added to the rendering engine. This will call a method in `TestApp` once the `RenderingEngine` starts rendering a frame, and again when it has finished. There will be more information on the `RenderingEngineFrameListener` class in the subsequent sections. Then the newly created `RenderingEngine` is added to the `RenderingManager`.

This is one of the virtual abstract methods of the Application class mentioned before. This must be implemented in the user's class. This method should load all of the scene geometry and set the initial conditions and parameters of the simulation.

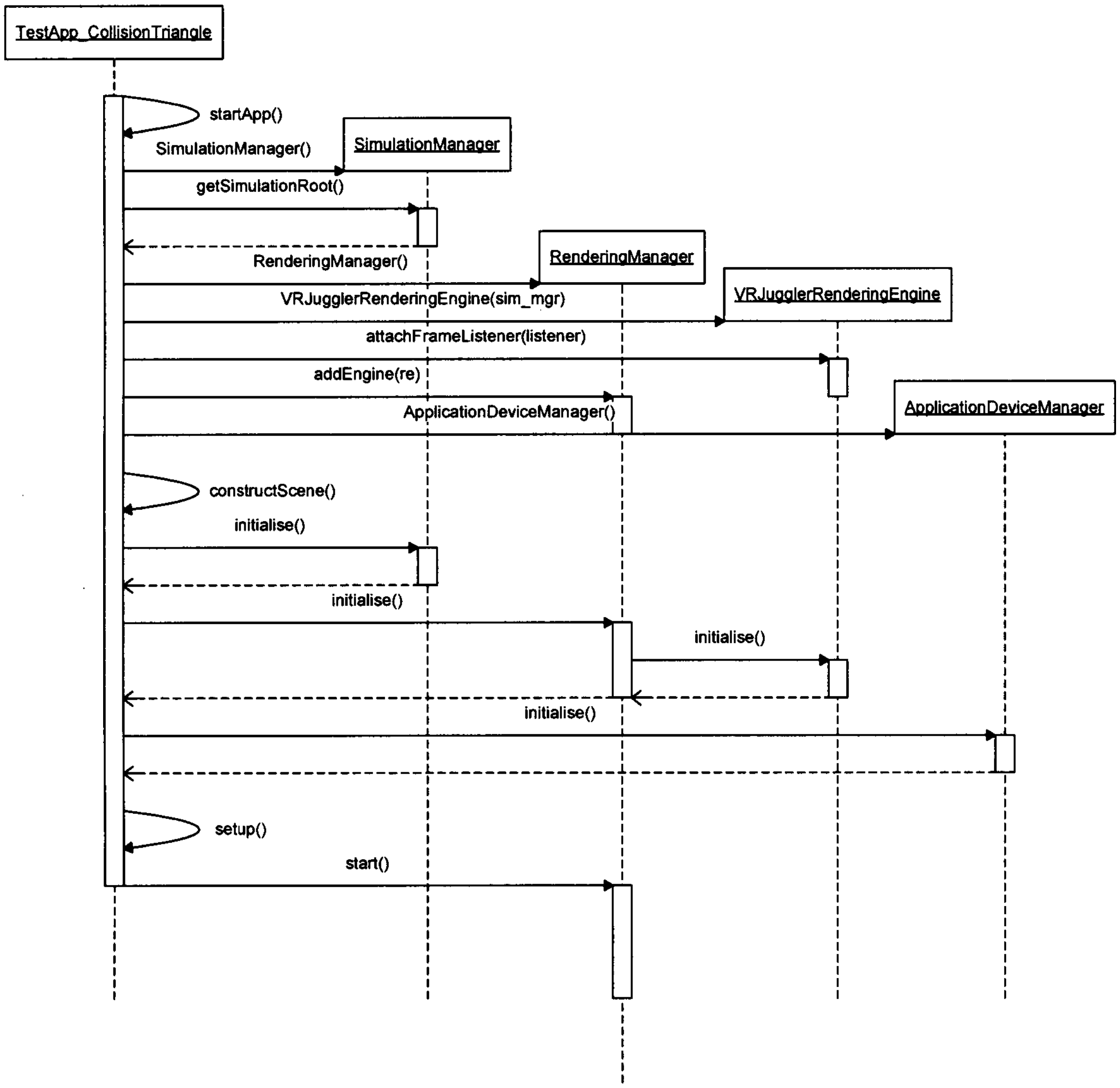


Figure 4-4 Application Module – System Initialization Sequence Diagram

Once the scene is created, all the managers are initialized by calling their respective *initialise()* methods. This step completes the system initialization phase. The only thing left to do is to start the simulation.

This is achieved with a call to the `RenderingManager`'s *start()* method. Only the `RenderingManager` needs to be started since the framework is not fully multithreaded at this point. In the multithreaded version, the *start()* method of all three managers would be called in different threads. Then each manager has its own dedicated thread and hence would not rely on any other manager to finish its operation before continuing. This would facilitate the complete parallelization of simulation updates, rendering of the scene, and input from devices.

Note that for simplification purposes, Figure 4-4 only shows the function calls from classes within the application module. For instance, the `SimulationManager`'s *initialise()* function performs a lot of operations and makes many different function calls. These were excluded to simplify the diagram.

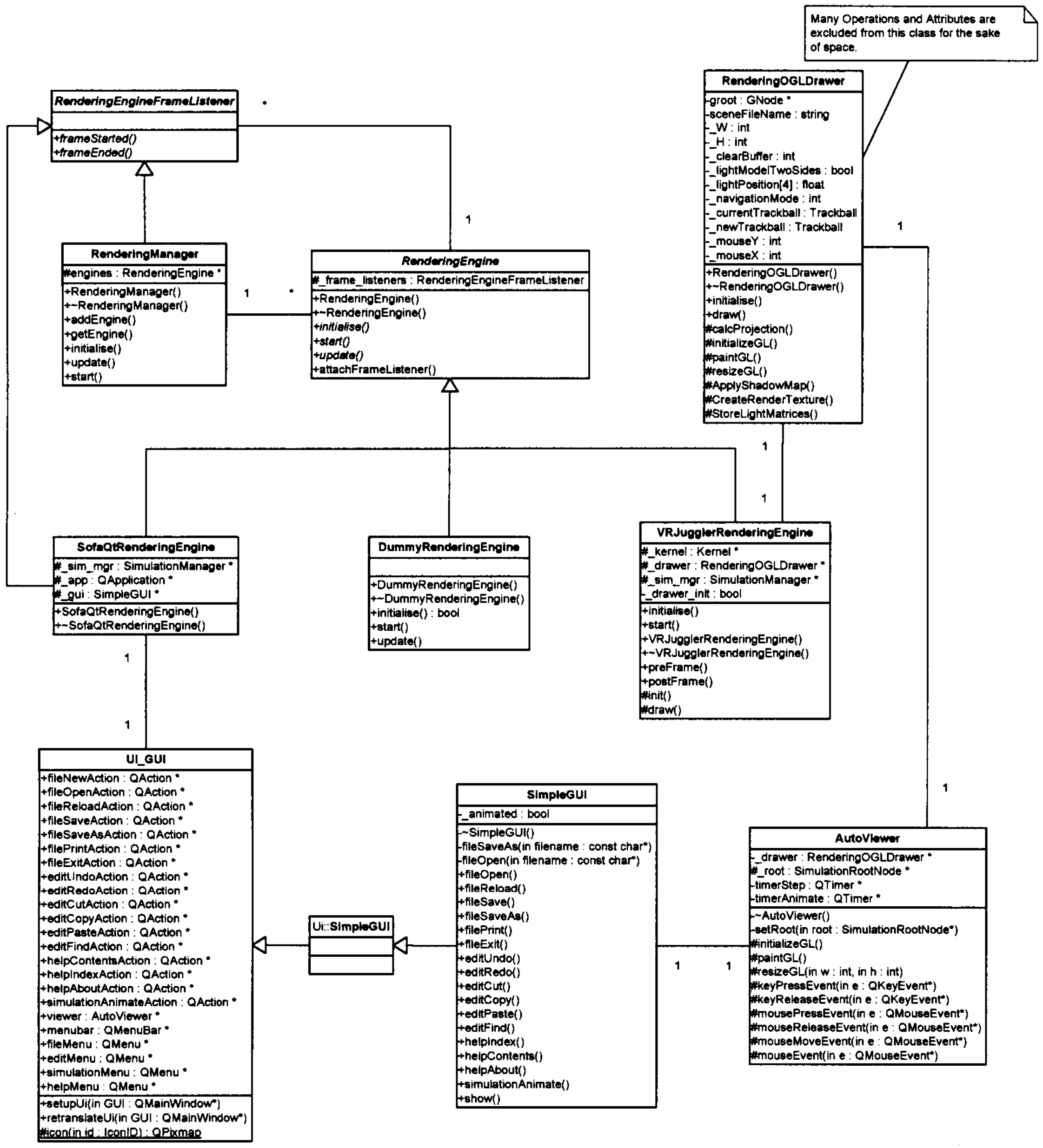
4.1.3 Rendering

Every simulation will need to perform some sort of feedback. In the AutoMan framework it is the rendering package that provides this functionality. Whether it is rendering geometry to a display, or haptic feedback to a device, this module should be able to handle it. This section of the thesis describes the design of the rendering package and shows its extensibility to output different information.

Figure 4-5, shows the static structure of this module. Technically it is only the upper four classes of the module that are considered part of this package. The rest of the classes in the figure make up different implementations of engines that will render the scene. This will become clearer shortly.

Like in the other two modules, there is a manager, `RenderingManager`. This class contains all of the references to the `RenderingEngines` to be used in the simulation. It ensures that all of them are initialized and that they get updated on every frame. In the multithreaded version that is under development, it would only need to start a thread for each of the rendering engines and then they could manage themselves.

The `RenderingEngine` class is an abstract class that defines an engine that will provide feedback to the user. Most commonly this will be via some sort of display. In that case, a `RenderingEngine` will draw the scene geometry and pass the information to the system's video card for output to the display. However, a `RenderingEngine` can be anything that provides the user with feedback. So this can include sending information to a console, or out a network interface. It could also include sending force feedback information to a haptic device. Anything that will send



Many Operations and Attributes are excluded from this class for the sake of space.
RenderingEngineFrameListener
+frameStarted()
+frameEnded()
RenderingOGLDrawer
-groot : GNode *
-sceneFileName : string
-_W : int
-_H : int
-_clearBuffer : int
-_lightModelTwoSides : bool
-_lightPosition[4] : float
-_navigationMode : int
-_currentTrackball : Trackball
-_newTrackball : Trackball
-_mouseY : int
-_mouseX : int
+RenderingOGLDrawer()
+~RenderingOGLDrawer()
+initialise()
+draw()
#calcProjection()
#initializeGL()
#paintGL()
#resizeGL()
#ApplyShadowMap()
#CreateRenderTexture()
#StoreLightMatrices()
RenderingManager
#engines : RenderingEngine *
+RenderingManager()
+~RenderingManager()
+addEngine()
+getEngine()
+initialise()
+update()
+start()
RenderingEngine
#_frame_listeners : RenderingEngineFrameListener
+RenderingEngine()
+~RenderingEngine()
+initialise()
+start()
+update()
+attachFrameListener()
SofaQtRenderingEngine
#_sim_mgr : SimulationManager *
#_app : QApplication *
#_gui : SimpleGUI *
+SofaQtRenderingEngine()
+~SofaQtRenderingEngine()
DummyRenderingEngine
+DummyRenderingEngine()
+~DummyRenderingEngine()
+initialise() : bool
+start()
+update()
VRJugglerRenderingEngine
#_kernel : Kernel *
#_drawer : RenderingOGLDrawer *
#_sim_mgr : SimulationManager *
-_drawer_init : bool
+initialise()
+start()
+VRJugglerRenderingEngine()
+~VRJugglerRenderingEngine()
+preFrame()
+postFrame()
#init()
#draw()
UI_GUI
+fileNewAction : QAction *
+fileOpenAction : QAction *
+fileReloadAction : QAction *
+fileSaveAction : QAction *
+fileSaveAsAction : QAction *
+filePrintAction : QAction *
+fileExitAction : QAction *
+editUndoAction : QAction *
+editRedoAction : QAction *
+editCutAction : QAction *
+editCopyAction : QAction *
+editPasteAction : QAction *
+editFindAction : QAction *
+helpContentsAction : QAction *
+helpIndexAction : QAction *
+helpAboutAction : QAction *
+simulationAnimateAction : QAction *
+viewer : AutoViewer *
+menubar : QMenuBar *
+fileMenu : QMenu *
+editMenu : QMenu *
+simulationMenu : QMenu *
+helpMenu : QMenu *
+setupUi(in GUI : QMainWindow*)
+retranslateUi(in GUI : QMainWindow*)
#icon(in id : IconID) : QPixmap
Ui::SimpleGUI
SimpleGUI
-_animated : bool
-~SimpleGUI()
-fileSaveAs(in filename : const char*)
-fileOpen(in filename : const char*)
+fileOpen()
+fileReload()
+fileSave()
+fileSaveAs()
+filePrint()
+fileExit()
+editUndo()
+editRedo()
+editCut()
+editCopy()
+editPaste()
+editFind()
+helpIndex()
+helpContents()
+helpAbout()
+simulationAnimate()
+show()
AutoViewer
-_drawer : RenderingOGLDrawer *
#_root : SimulationRootNode *
-timerStep : QTimer *
-timerAnimate : QTimer *
-~AutoViewer()
-setRoot(in root : SimulationRootNode*)
#initializeGL()
#paintGL()
#resizeGL(in w : int, in h : int)
#keyPressEvent(in e : QKeyEvent*)
#keyReleaseEvent(in e : QKeyEvent*)
#mousePressEvent(in e : QMouseEvent*)
#mouseReleaseEvent(in e : QMouseEvent*)
#mouseMoveEvent(in e : QMouseEvent*)
#mouseEvent(in e : QMouseEvent*)

Figure 4-5 Rendering Package – Class Diagram

information out of the system can be a `RenderingEngine`. Further detail will be presented throughout this section. Since this class is abstract, it cannot be instantiated; it must be extended by another class that implements the virtual methods. This is the same as with the `Application` class described above.

The `RenderingEngineFrameListener` is another abstract class. Its purpose is to provide a method for the other classes to be informed when a specific `RenderingEngine` has finished rendering a frame. A class that implements `RenderingEngineFrameListener` will have at least two methods, *frameStarted()*, and *frameEnded()*. Once this frame listener is attached to a `RenderingEngine`, before every frame is rendered, the *frameStarted()* function is called, and likewise after a frame is rendered, *frameEnded()* is called. This allows other classes to make per frame calculations and changes to the simulation.

A `RenderingEngine` can have more than one frame listener and each one will have their respective *frameStarted()* and *frameEnded()* methods called at the beginning and at the end of every frame the engine renders.

The final class in the rendering package is the `RenderingOGLDrawer`. This is a helper class for the `RenderingEngine`s that are going to render the scene geometry to a display. Simply by calling the *draw()* method, the geometry of the entire scene will be rendered. This class is heavily based on the SOFA's `QtViewer` class.

There are three `RenderingEngine` implementations included with AutoMan, they are: `DummyRenderingEngine`, `SofaQtRenderingEngine`, and `VRJugglerRenderingEngine`. These are provided within the rendering package of AutoMan, but there not exactly part of the framework itself. These implementations are provided but their use is completely optional.

The following sections describe each of these `RenderingEngine` implementations in detail. Also, although the rest of the framework is not dependent in any way on the implementation of the `RenderingEngine`, the internal workings of each are slightly different. As such, the operation of main execution loop operation phase will also be described for each of the following `RenderingEngines`.

4.1.3.1 DummyRenderingEngine

This engine does nothing except write one line to the console on every frame. It was used in the testing of the `RenderingManager`'s capability of handling more than one engine at a time.

It also serves as a bare bones template for advanced users to develop their own rendering engines. All the required methods are already implemented with stubs; the user only needs to fill in the methods to perform the required function.

4.1.3.2 SofaQtRenderingEngine

This `RenderingEngine` is similar to the graphical user interface (GUI) provided with SOFA. However, there have been minor changes to the layout, and drastic changes to the code. The layout of the original SOFA GUI provided more information than was actually needed for a simulation user, this has been removed. The engine itself still runs on Qt [22], but the viewer that was provided with SOFA, `QtViewer`, was not satisfactory for the purposes of AutoMan. Section 5.1.1 explains the reasoning behind this.

Figure 4-6 is the sequence diagram for the main execution phase of the framework when the `SofaQtRenderingEngine` is being used. It begins with a call to the *start()* method of the `RenderingManager`, which in turn starts the `RenderingEngine` by calling its *start()* method.

The `SofaQtRenderingEngine` displays the GUI but calling the *show()* method of the `SimpleGUI` class. After the program is displayed, the engine starts the `QApplication`, a Qt class, by calling the *exec()* method. Both the `SimpleGUI` and the `QApplication` objects were created during the initialization phase of the program.

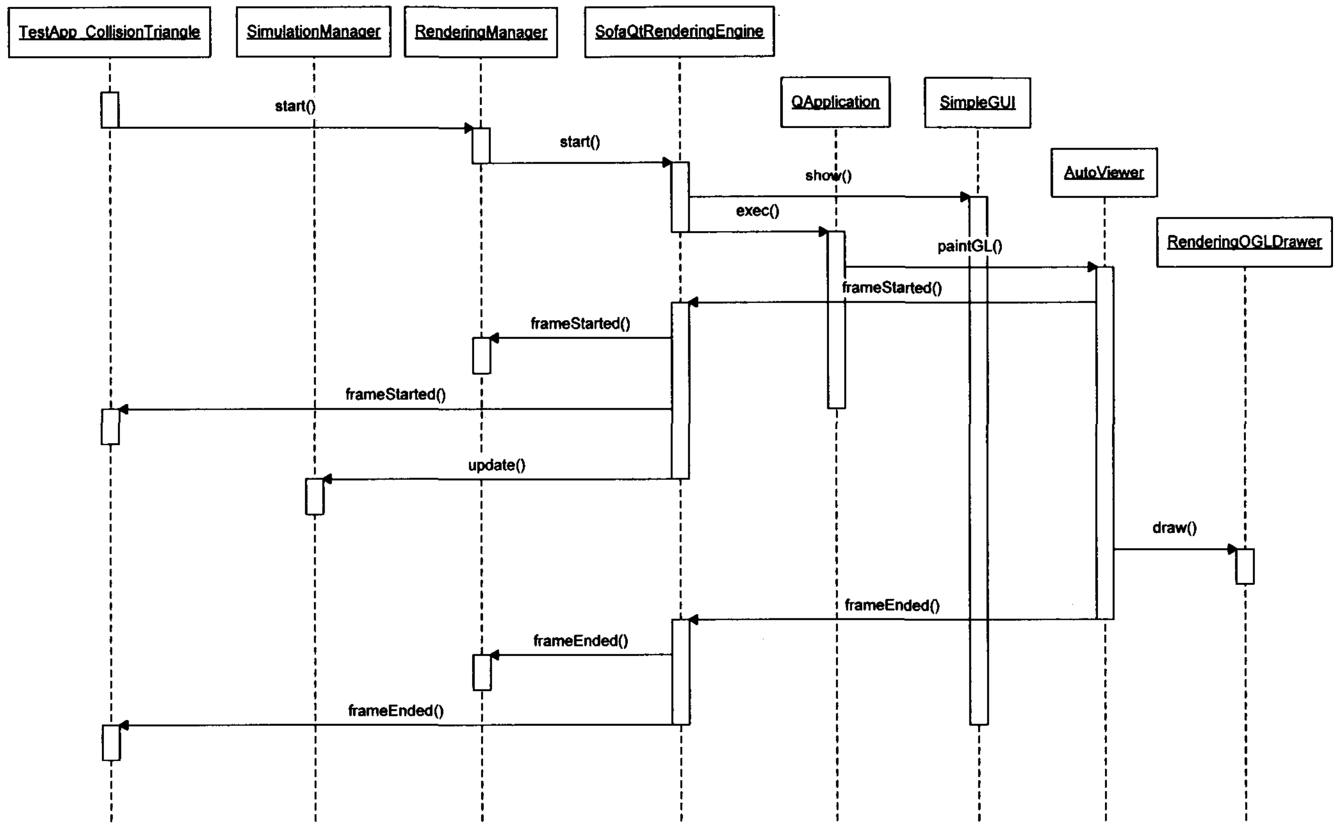


Figure 4-6 SofaQtRenderingEngine – Sequence Diagram

Starting the `QApplication` by calling the *exec()* method gives the main thread of execution over to the Qt library. Every frame, the *paintGL()* function of `AutoViewer` is called to draw the geometry to the window in the GUI. However, this is the only occasion where AutoMan has control of the program until the next frame. So, the `AutoViewer` informs the `SofaQtRenderingEngine` that it is about to start rendering the frame by calling the *frameStarted()* method.

This allows the rendering engine to make the calls to the attached frame listeners. The first is back to the `RenderingManager` so it can update the other `RenderingEngines`. The next is back to the user application for any processing that he or she may wish to perform at this time.

Then it steps the `SimulationManager` by calling the *update()* method. This causes the simulation to take one step in time allowing the simulation objects to move and react. This finishes the pre-frame calculations and the `AutoViewer` is free to render the scene by calling the *draw()* function of the `RenderingOGLDrawer`.

When the `AutoViewer` is finished, the post frame calculations need to be performed. So it calls the *frameEnded()* function of the `SofaQtRenderingEngine`, which makes all the required calls to its attached frame listeners respectfully. This ends the frame. When the `QApplication` makes the next *paintGL()* call, the whole process starts over for the next frame.

4.1.3.3 VRJugglerRenderingEngine

This rendering engine uses the VR Juggler library to display the scene geometry. This library not only allows the use of regular 2D displays, but also 3D stereoscopic displays. That means that a simulation built in the AutoMan framework that uses this engine can run the simulation on a stereoscopic display.

The following sequence diagram, Figure 4-7, describes the operation of this engine. Like with the `SofaQtRenderingEngine`, the process begins with calling the *start()* method of the `RenderingManager`, which in turn calls the start method of the `VRJugglerRenderingEngine`.

The `VRJugglerRenderingEngine` has created an instance of a VR Juggler `Kernel`, (`vrj::Kernel`) in the initialization phase of the program. The call to its *waitForKernelStop()* will give it control of the program's main execution thread like the call to *exec()* of the `QApplication` in the previous case.

However, the `vrj::Kernel` expects pre-frame calculations and makes a call back to the `VRJugglerRenderingEngine` just prior to starting the frame's rendering. This allows for the same pre-frame calculations to be done. This step is the same as the previous example, the `RenderingEngine` calls the *frameStarted()* method of the attached frame listeners and then steps the simulation by calling the *update()* method of the `SimulationManager`.

With that finished, the `vrj::Kernel` makes the *draw()* callback to the `VRJugglerRenderingEngine` which passes the instruction onto the `RenderingOGLDrawer`.

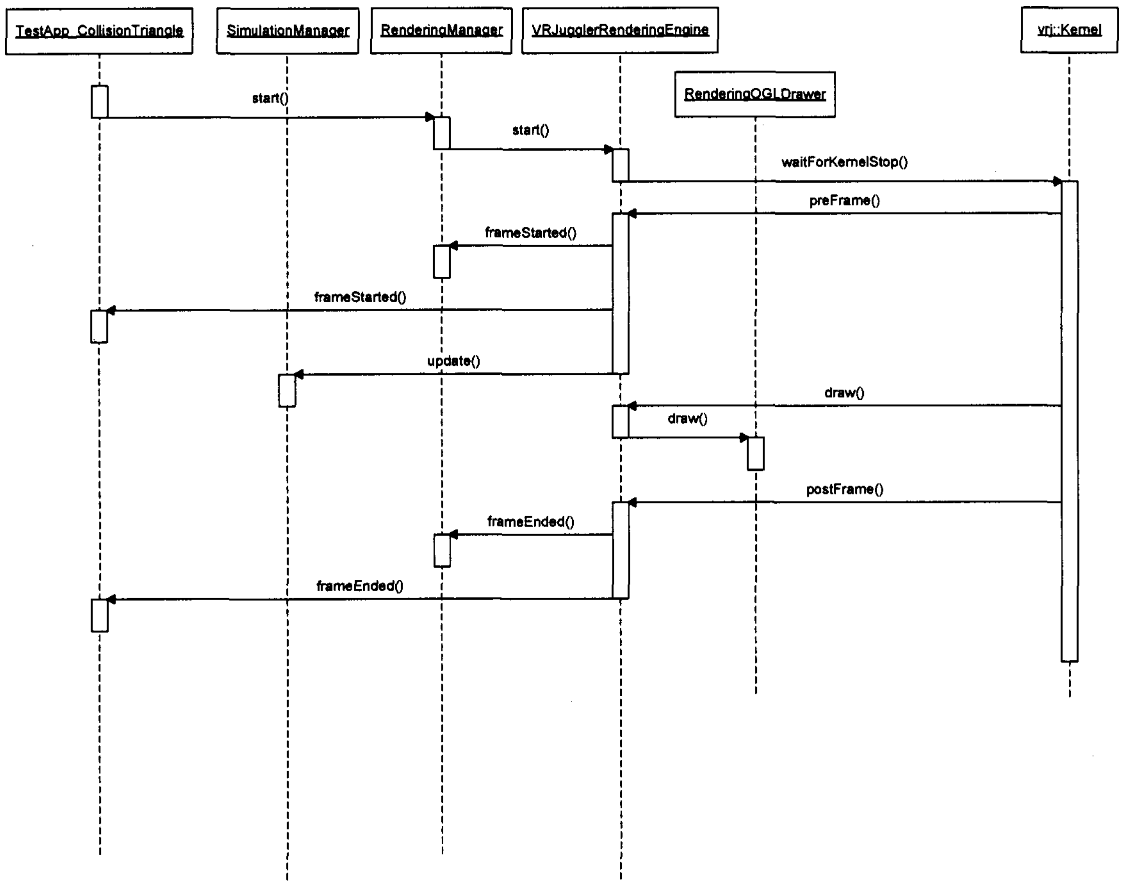


Figure 4-7 VRJugglerRenderingEngine – Sequence Diagram

When the frame has been rendered, `vrj::Kernel` calls the *postFrame()* method of the rendering engine allowing it to propagate the call though all of its frame listeners just as before. This ends the frame, and the `vrj::Kernel` makes another *preFrame()* call starting the process over again.

Since VR Juggler expects that pre- and post-frame calculations are probably necessary, the `VRJugglerRenderingEngine` is less complex than the

`SofaQtRenderingEngine`. This is expected as VR Juggler is designed for rendering graphics in real time applications, where Qt mainly used for windowed GUI's. Advanced users of the framework will find that writing their own rendering engines will be much simpler for libraries that are designed for this kind of graphical rendering.

4.2 Documentation Framework

One of the grand goals of this framework is to encourage collaboration between different institutions in the development of high caliber simulations, especially surgical simulations. Ideally as new algorithms and data representation models are developed, they will be added to the AutoMan framework, allowing everyone that has built simulations using this framework to benefit from the improved performance or fidelity.

In order to support the addition of new technology to AutoMan, a robust and thorough documentation framework must be set up. This should help new developers, regardless of their expertise, to familiarize themselves with the AutoMan structure and implementation. It should support all types of users of the framework, those that are just developing stand alone simulations and those who are attempting to add new functionality to or upgrading the framework itself.

This is realized through a number of different ways. Standard coding conventions ensure that the code is uniform throughout the framework reducing many possible bugs [23]. Extensive API documentation provides users with a look at specific interfaces and allows them to view the hierarchy structure of the classes. Also, user guides and how-to's will help aid developers with specific problems. Each is discussed further in the following sections.

4.2.1 Coding Standard Definition

The AutoMan framework has been coded to the standard as stated in the AutoMan C++ Coding Standard document which can be found in Appendix B.

This document defines the conventions followed during the implementation of the framework, including naming conventions for files and classes, proper commenting structure, and indenting and spacing rules.

These standards should allow the framework to:

- Have a consistent style throughout the environment.
- Be easy to read and understand for new developers.
- Be maintainable by other developers [23].

These are all important traits to have for the framework especially as it is expected that new developers will be adding to it. Advanced users who are developing new modules or modifying the framework code should be familiar with these standards and adhere to them whenever possible.

4.2.2 API and Tutorials

Since the code followed the above Coding Standard and has correctly commented each class with Doxygen [24] comments, the generation of an HTML based API is trivially easy to create. This API provides a method of viewing the structure and interface of each class.

A public website has been set up on the UWO Panther server[2] to host this API. This resource will be continually updated as major changes are committed to the AutoMan project. The public site also provides a number of tutorials that show the users of the framework how to complete specific tasks in the code. Currently this is managed internally, but ideally it will evolve into a community maintained endeavor like a Wiki

[2] The URL is as follows: http://publish.uwo.ca/~thayes4/

[25]. In this case, users of the framework can provide their own examples and create their own tutorials so that the rest of the user community can benefit from them.

4.2.3 Visual Studio .NET Project Wizard

In addition to the API documentation and tutorials, a helper tool is provided. The AutoMan Simulation Application Wizard generates the skeleton Visual Studio .Net project file. It automatically links all to the required library directories, saving the developer of the simulation a relatively lengthy project set up. The wizard prompts the user to enter a name for the project and a couple of parameters and automatically sets everything up, allowing him or her to start on the simulation content immediately.

Chapter 5 Implementation and Testing

This chapter starts with a discussion of a few of the major and therefore more interesting issues that arose during the implementation of AutoMan. Some of these issues were substantial enough to warrant a change in the design, or even the feature being dropped completely for the initial release of the software.

Following the discussion of the implementation issues is a description of the testing procedure carried out on the framework along with the ensuing results.

5.1 Implementation

This section covers the implementation issues encountered during the development of AutoMan. Although there were numerous issues, only a few of the major ones are described here. The first is in relation to the GUI supplied with SOFA and the many difficulties that arose from working with it. The second issue is in regard to multithreading the framework. The final issue that is discussed is in regard to the loading behavioral model information form files. This last discussion shows many of the difficulties in working with poorly documented software.

5.1.1 Interface

The original interface supplied with SOFA, `QtViewer`, did not fully meet the requirements of an AutoMan `RenderingEngine`. The following is a list of some of the shortcomings:

- Contained a sidebar that provided unnecessary options and information.
  - Included is the ability to show the wireframe, polygon normals, and mappings. Also, the ability to set the time step and take single animation steps, among other things.
- All of the OpenGL setup code was contained within the GUI class using private methods.
- No support of per-frame calculations.

The sidebar provided the SOFA GUI allowed the simulation user to do more than is likely needed in most simulations. There is little need for the simulation user to have access to the options provided by this panel, they are really helpful only in a debugging scenario. Most simulations should only require the scene to be loaded in a paused state, and the ability to start the simulation. Therefore, for the AutoMan version of this GUI, this sidebar was to be removed.

This required a complete rewriting of the GUI window. This was not too difficult as it was built in Qt, which is a very well supported development framework that is widely used for GUI applications, though not exclusively.

However, during this rewrite it was realized that the SOFA GUI was spread out over a few classes and much of the OpenGL setup was completed within one of them. Figure 5-1 shows a partial class diagram that illustrates the structure of the SOFA GUI. This was fine for the purposes of SOFA which was not designed to be used with different methods of rendering however, it was not sufficient for AutoMan's purposes.

This required a full re-write of one of the SOFA GUI classes, `QtViewer`. This class defined the Qt widget that served as the rendering window in the GUI. All of the OpenGL setup was completed here along with the calls to the simulation objects to render the geometry. It was decided that a new class should be implemented whose sole purpose was to handle all of the OpenGL calculations and method calls, it was called `RenderingOGLDrawer`.

The `RenderingOGLDrawer` class was briefly described in section 4.1.3. Most of this class is comprised of methods taken directly from SOFA's `QtViewer` class, but considering that `QtViewer` consists of roughly 2700 lines of code, it was no easy task determining what was required and what could be excluded. After the OpenGL code was separated, the rest of the functionality of `QtViewer` needed to be implemented to use the new class. This resulted in the `AutoViewer` class.

This separation of and re-implementation of the `QtViewer` class inherently solved the final shortcoming. `QtViewer` did not supply any method for the rest of the application to get control once the GUI had started the rendering loop. By using the AutoViewer class instead, this functionality was easily implemented. Every time that the GUI window asks this widget to refresh the scene, it made the correct calls to the `RenderingEngineFrameListener`'s attached to the parent `RenderingEngine`. This would not have been possible using the `QtViewer` supplied with SOFA.

The magnitude of this change to the GUI was not factored in during the design of AutoMan. This issue resulted in the addition of two new classes to the framework and required some rethinking of the operation of the rendering module altogether. Furthermore, it showed that although the addition of new `RenderingEngine` implementations to AutoMan may be easy, the act of developing them in the first place may not. However, this was no fault of AutoMan and is outside of its scope for the time being.

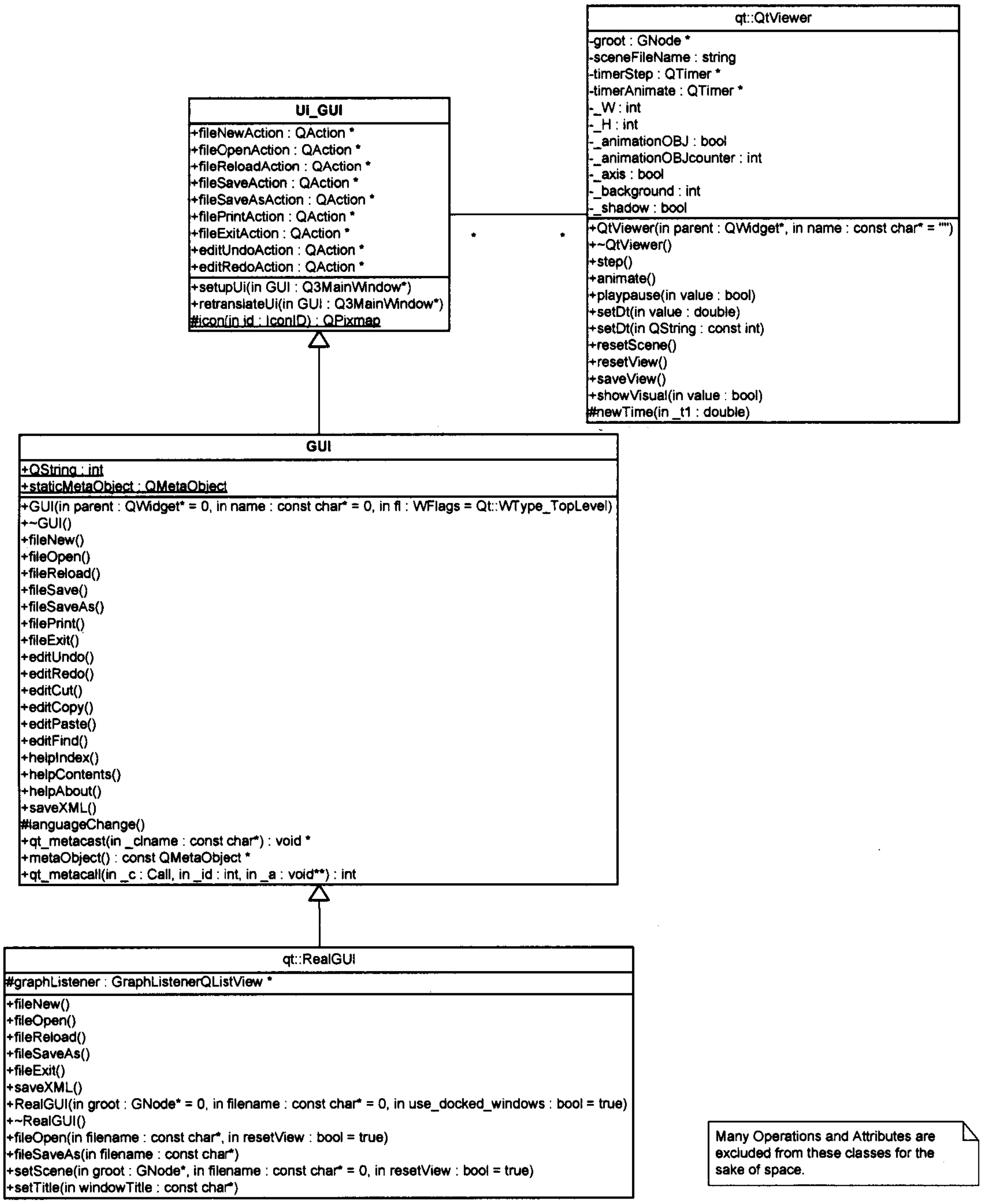

qt::QtViewer
-groot : GNode *
-sceneFileName : string
-timerStep : QTimer *
-timerAnimate : QTimer *
-_W : int
-_H : int
-_animationOBJ : bool
-_animationOBJcounter : int
-_axis : bool
-_background : int
-_shadow : bool
+QtViewer(in parent : QWidget*, in name : const char* = "")
+~QtViewer()
+step()
+animate()
+playpause(in value : bool)
+setDt(in value : double)
+setDt(in QString : const int)
+resetScene()
+resetView()
+saveView()
+showVisual(in value : bool)
#newTime(in _t1 : double)
Ui_GUI
+fileNewAction : QAction *
+fileOpenAction : QAction *
+fileReloadAction : QAction *
+fileSaveAction : QAction *
+fileSaveAsAction : QAction *
+filePrintAction : QAction *
+fileExitAction : QAction *
+editUndoAction : QAction *
+editRedoAction : QAction *
+setupUi(in GUI : Q3MainWindow*)
+retranslateUi(in GUI : Q3MainWindow*)
#icon(in id : IconID) : QPixmap
*
*
GUI
+QString : int
+staticMetaObject : QMetaObject
+GUI(in parent : QWidget* = 0, in name : const char* = 0, in fl : WFlags = Qt::WType_TopLevel)
+~GUI()
+fileNew()
+fileOpen()
+fileReload()
+fileSave()
+fileSaveAs()
+filePrint()
+fileExit()
+editUndo()
+editRedo()
+editCut()
+editCopy()
+editPaste()
+editFind()
+helpIndex()
+helpContents()
+helpAbout()
+saveXML()
#languageChange()
+qt_metacast(in _clname : const char*) : void *
+metaObject() : const QMetaObject *
+qt_metacall(in _c : Call, in _id : int, in _a : void**) : int
qt::RealGUI
#graphListener : GraphListenerQListView *
+fileNew()
+fileOpen()
+fileReload()
+fileSaveAs()
+fileExit()
+saveXML()
+RealGUI(in groot : GNode* = 0, in filename : const char* = 0, in use_docked_windows : bool = true)
+~RealGUI()
+fileOpen(in filename : const char*, in resetView : bool = true)
+fileSaveAs(in filename : const char*)
+setScene(in groot : GNode*, in filename : const char* = 0, in resetView : bool = true)
+setTitle(in windowTitle : const char*)
Many Operations and Attributes are excluded from these classes for the sake of space.

Figure 5-1 SOFA GUI – Partial Class Diagram

5.1.2 Threading

Initially, this release of AutoMan was supposed to be multithreaded. The fact that much of this feature is already accounted for in the design and is partially implemented should attest to that. Unfortunately an issue arose that drastically complicated the finalization of this feature.

The original design called for the *initialise()* method of each manager to create itself a dedicated thread. Then the `Application` class could start each manager's thread simply by calling the *start()* method of each manager. The `RenderingManager` would have initialized a thread for each `RenderingEngine` attached and each would be started during the `Application`'s call to *start()*. The problem arose when specific implementations of these `RenderingEngines` cannot work when they are started by anything but the main execution thread. VR Juggler for one.

`VRJugglerRenderingEngine` creates a `vrj::Kernel` which will not work properly unless is it run by the main execution thread. So, in an attempt to save the feature, the threading structure was reworked to allow the main execution thread to start all the other modules and then run the first `RenderingEngine` that was attached. This implies that when the user is attaching `RenderingEngines`, the main graphical display must be the first one attached. This is not the ideal solution, but it works as a "band-aid" fix.

However, the issues that ensued thereafter were irrecoverable. Running under this new multithreaded structure led to catastrophic failures in the system and it was realized that this feature required a lot more time and effort than was available. As a result, multithreading succumbed to functional triage in this first release of AutoMan. It has been pushed back to a later release of AutoMan to allow more time to work out this issue.

5.1.3 Behavioral Model File Loading

One feature that AutoMan strives to provide for its users is the ability to easily create the objects in the simulation in code instead of defining a scene file. As it turns out, this proved to be a bigger task than originally planned.

The SOFA `ObjectFactory` class is built after the factory model [26]. A factory is an abstraction of a constructor that ensures a class instantiates the correct object for its purpose [27]. It is this class that SOFA uses to create the simulated objects when loading a scene file. However, this is also the only way the factory can be used. The creation of an object from the factory requires a call to the *parse()* method of the object to be created. This method parses the `BaseObjectDescription` that is supplied as an argument, and sets the correct attribute values. This object description class is created while the scene file is being parsed. It is useless to attempt to create this object procedurally since it requires the exact line from the scene file that defines the object that you wish to create. This essentially means that the `ObjectFactory` cannot be to generate any new simulation objects from within the code.

In order to keep SOFA unaltered, AutoMan had to call the constructor methods of each specific SOFA object instead of using the factory class. Unfortunately this method falls short since some of the object member variables do not have accessor methods or the object itself did not have the ability to load information from a file. This was the case with the `UniformMass` object that defines the mass of a rigid object and is used in `SimulationRigidObject`.

Creating the `UniformMass` object was trivial if only a mass was required. All that was needed was to supply the number that corresponds to the mass of the object. The problem arose when mass information is stored in a behavioral file, such as with the case of the chain samples that come with SOFA.

In order to allow a `SimulationRigidObject` to load behavioral information from a file, it had to parse the file itself. This turned out to be easily implemented as the functionality was already present in SOFA, but was not accessible from the outside the class.

It had been a stipulation of using SOFA as the backbone of AutoMan that there would be no changes made to SOFA itself. This would have made it a lot more difficult to distribute AutoMan since it would also be required that the updated SOFA files also be distributed. Also, talking with the developers of SOFA, it seems that this functionality will be publically accessible in future releases of the framework.

This hidden functionality issue arose numerous times in the development of AutoMan. This is one of the shortcomings of using new technologies that are not fully implemented or documented. Unfortunately, this shortcoming ended up being a major cause of stress and delays in the development of AutoMan.

5.2 Testing

In order to test the full extent of the AutoMan functionality, a test plan was created and executed. The following sections describe this test plan and the results of the tests that were performed.

5.2.1 Test Plan

Considering the limited resources and time at hand, only coarse unit tests were performed to ensure that the implemented features worked as they should. This includes not only the framework itself, but the installation process, the VS .Net Project Wizard, and website. This is by no means a complete testing phase, but this is all that was possible under the resources and time at hand. Further testing will be performed for the next iteration of the framework; this release should still be considered an alpha build.

Appendix C is the test plan document that outlines the testing process and describes each test case in full.

5.3 Test Results

By executing this test plan, a Test Results document has been generated. It can be found in Appendix D. This document presents the results of each test case and comments on each.

To summarize the testing results, the implemented features of AutoMan work as expected. Each component of the AutoMan achieves the base functionality that it was designed to perform.

The one exception is the ability to rotate the simulated object. This is another case of hidden functionality that is not accessible from outside classes. That is to say that the SOFA objects can be rotated but, this functionality can only be invoked internally. The developer release of SOFA has remedied this issue, but the current version of AutoMan is built on the stable Beta 2 release of SOFA. In subsequent releases of AutoMan, this feature should work in full.

It is understood that there is a significant amount of testing that should still be performed, but these cursory results show that the main functionality of AutoMan is indeed present and working.

5.4 Case Study

To supplement the testing process and showcase some of the features of AutoMan a small application was developed as a case study.

The basic premise of this application was to retrieve a scene filename from the command line upon execution and run it using the `VRJugglerRenderingEngine`. This task was nearly trivial with the help of the VS .Net Project Wizard which generated the project and files and also implemented most of the required methods. In fact the only method that needed to be implemented was createScene(). All that was required here was to load the file specified as the command argument. This is a single call to the *loadSceneFile()* method of the `SimulationManager`.

The following figures are screen captures of this application running a number of the SOFA sample scene files. By modifying the VRJuggler configuration file used, many display options can be set including stereoscopic rendering, multiple viewports, and simulated 3d device input.

The first thing that should be noted about this exercise is the ease at which it was developed. Three lines of code needed to be added to the files generated by the project wizard to load the scene; two to store the filename from the command argument list, and one to load the file. The biggest task was adding the input handling code, but even then, this is around ten lines of code that are simple to produce assuming the developer is knowledgeable in the input system they are using. Furthermore, this whole process should be simplified when later releases of AutoMan provide input handling support.

However, there is a mild drop in performance (around 0.5 frames per second) when these scenes are run through AutoMan. This was expected since AutoMan has to do more per-frame processing than SOFA. However, most of this processing will be done in parallel as soon as the multithreaded version of the framework is complete.

Figure 5-2, Figure 5-3, and Figure 5-4 each show screen captures of this application running some SOFA sample simulations.

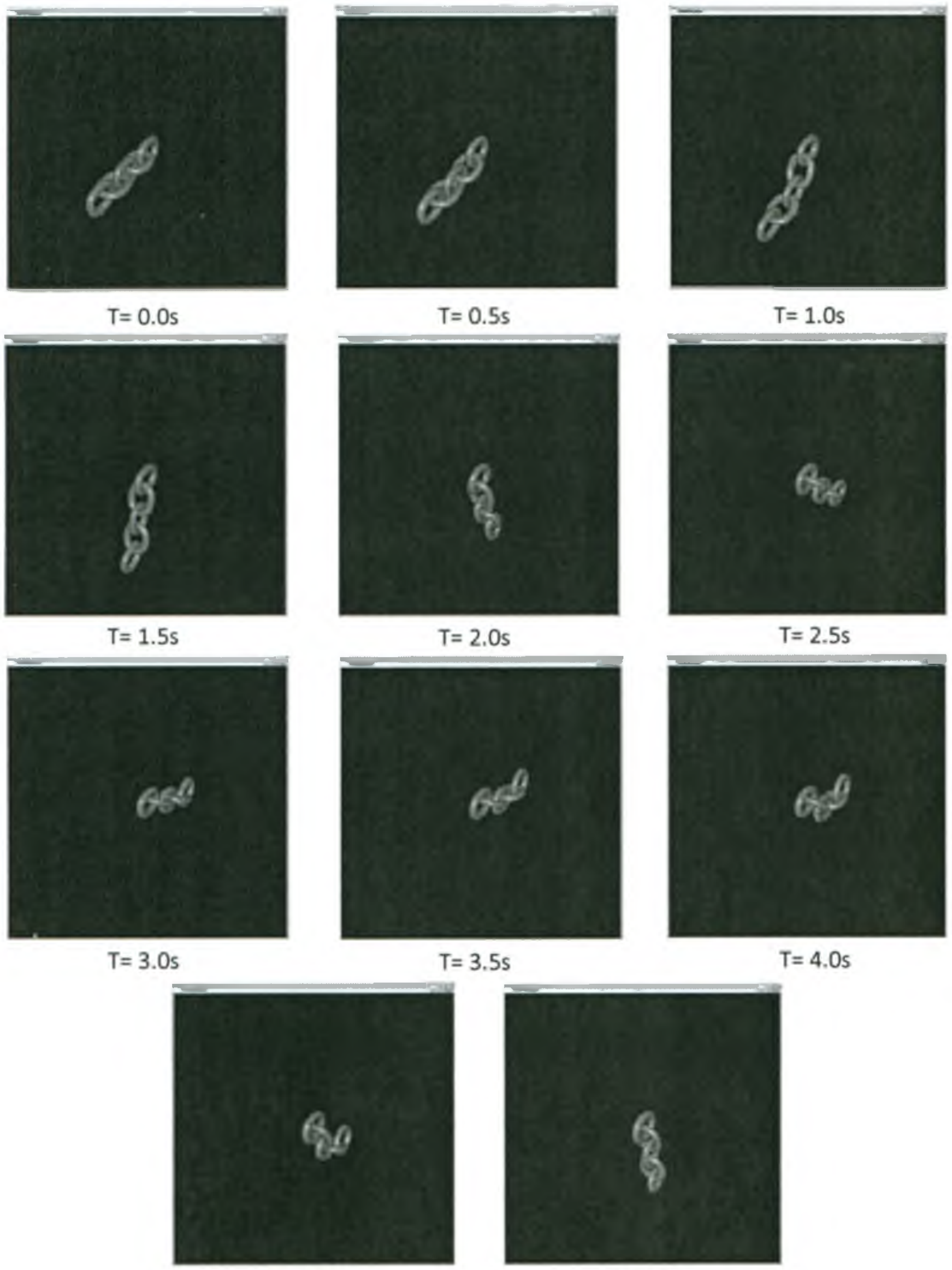


Figure 5-2 chainRigid.scn

A chain comprised of rigid links shown at various times (T) throughout its swing.

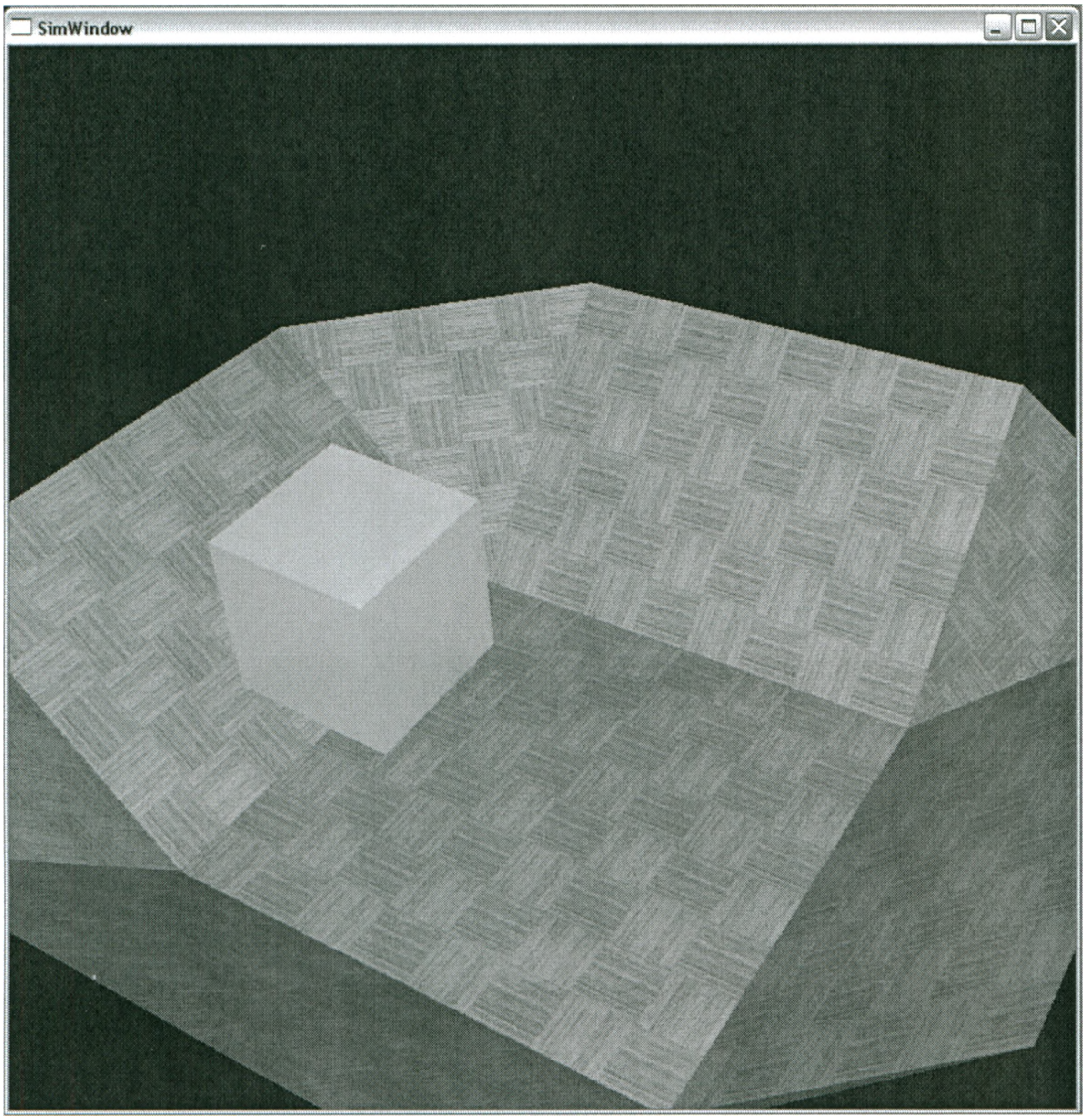


Figure 5-3 collisions.scn

A rigid box colliding with a textured floor model.

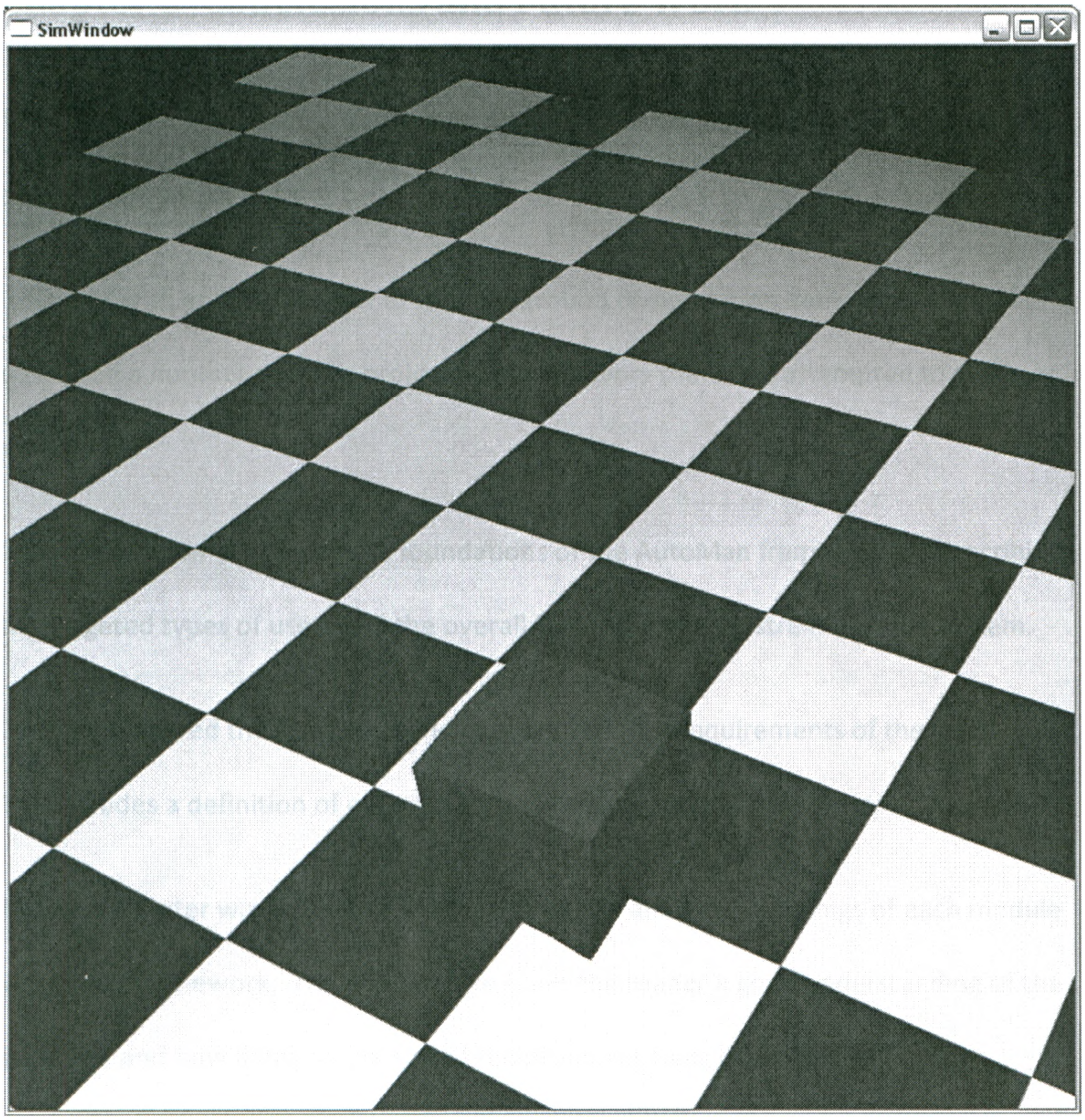


Figure 5-4 collisionSphere.scn

A deformable box falling due to gravity and colliding with a sphere.

Chapter 6 Conclusions and Future Work

This thesis has presented a framework named AutoMan that aids in the development of advanced simulations.

It started with an introduction to the background of simulation development and described a number of other projects and endeavors that have attempted to tackle a similar task.

The next chapter presented the foundations of the AutoMan framework by describing the targeted types of users and the overall functions and constraints on the system.

Chapter 3 offered the functional and quality of service requirements of the framework. This includes a definition of each module and a description of the system's operation

The next chapter was a detailed look at the design and inner workings of each module within the framework. This should have given the reader a good understanding of the structure and how many of the stated requirements have been met.

This was followed by a discussion of the implementation process that highlighted some of the more interesting development issues and concerns. Also presented was the testing methodology and results.

This chapter concludes the thesis by discussing the success of the project and reaffirms that the requirements set out at the beginning of the project have in fact been achieved. The chapter also presents a number of possible extensions to the framework in the Future Work section.

6.1 Conclusions

Eric S. Raymond's book, *The Cathedral and The Bazaar* [28], discusses open source development and many of the preconditions required for a successful open source project. The following is an excerpt,

> When you start community-building, what you need to be able to present is a *plausible promise*. Your program doesn't have to work particularly well. It can be crude, buggy, incomplete, and poorly documented. What it must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future.

The AutoMan project is still far from a level of completeness that would realize its use in commercial caliber applications. However, it presents a step in the right direction for surgical simulation development. According to Eric S. Raymond, this is all that is required to start a community based project, and AutoMan has exceeded this expectation.

As stated at the beginning of this thesis, the AutoMan framework had the following grand goals:

- Provide a common interface to a variety of libraries and engines for simulation development.
- Allow the addition and replacement of components as new and improved algorithms are developed.
- Encourage collaboration among different research projects and allow the incorporation of these projects.
- Expand the targeted user base of these systems.

Now that the reader is familiar with the AutoMan framework, this section will discuss the completion of these goals and by extension the success of the project itself.

AutoMan is built on top of many lower level libraries such as SOFA, and VR Juggler. However, since all the implementation details of these libraries are hidden from the user's application, the user has does not need any expertise in using these libraries. Furthermore, the structure of the framework allows the incorporation of new and improved implementations of these engines. This virtualization allows AutoMan to provide a common interface to developing simulations, regardless of the library implementation.

The highly modular design, supported with the use of SOFA as a backbone for AutoMan, the incorporation of the new and improved algorithms has been realized. This is partially demonstrated by the development of the two rendering engines that are supplied with this version of the project. These are two different rendering algorithms and can easily switch in and out and can both run the same simulation application.

Supplying a method to incorporate new algorithms, and providing a means of distributing and these new advancements inherently encourages independent research projects to work together. This community based structure has been used by the open source community to achieve the same goal, collaboration between many different developers. This is supported by the documentation framework which should allow developers to quickly become proficient in the development of simulations using AutoMan. The hard part is to get exposure and start building a user base which unfortunately has not been achieved at this point. Ideally, as the framework realizes new releases and the addition of new technologies, the exposure will come and by extension the user base.

In an effort to expand the user base, AutoMan has been developed to encourage non-professional / academic development. Facilitating casual users to develop in the framework provides a number of benefits. Getting people using the framework will serve as the beta testing phase that was not performed for the release of AutoMan for this thesis. Also, even though these casual users are not developing in a professional manner, many will have great expertise in the area and will develop additional modules and interfaces to other engines that are not currently supported.

To encourage the use of AutoMan by casual developers, the application module defines an easy to use method of creating simple simulations. The use of the provided helper tools such as the Visual Studio Project Wizard simplifies this process even further. However the most important part of encouraging casual users is the availability of thorough documentation. AutoMan does indeed provide a number of basic tutorials and how-to's and also provides a method of contacting the developers through a mailing list. At its current level, the documentation is enough to get people started. The content of the tutorials needs expansion to further meet the needs of the simulation developers.

AutoMan has met all of the grand goals it originally set out to achieve, meaning the project a success despite the fact that there is still a long way to go before it is ready for commercial use. Since there was not enough time to perform a full user testing phase, this claim is of course unverified. However, this thesis has shown that the framework has been designed and implemented to meet these goals. Verification will come as this release is employed and user feedback can be received. This initial release of AutoMan has provided the foundations that may lead to a *de facto* framework for surgical simulation. Subsequent releases will further step towards this end.

6.2 Future Work

There have been many hints to future advancements to the AutoMan framework throughout the body of this thesis. This section will formalize and discuss possible advancements that will require a more concerted effort to complete.

6.2.1 Virtualized Input Device Manager

Developing an easy to use method of defining input devices is a difficult task. However, the `ApplicationDeviceManager` is ready to support the implementation when it is completed. The idea here is that input devices can be defined and included the same way that output devices can be defined and used via the `RenderingEngine` interface.

With the large jump in the number of input devices developed for simulation purposes, and the staggering number of custom built input devices, it is important this interface is as general as possible.

In developing this manager, VR Juggler's Gadgeteer [8] project should be looked at. This package may already meet most of the requirements. This would mean that it need only be virtualized and wrapped to interface with the rest of the AutoMan framework.

6.2.2 Event System

Something that would ease the development of full size simulations would be an event system that would be supplied with AutoMan. This would allow the users to define and add events to objects within the scene and have the framework handle the processing instead of requiring them to develop it themselves.

Events could be based on time. For instance, a simulated surgery must be completed within a specific time limit or the attempt should be considered a failure. Or at a random time during the surgical simulation, a blood vessel bursts or something else goes awry.

Another example is an event based on object interactions. Keeping with the example of surgical simulation, an event could be thrown when a tool comes in contact with a particular object, perhaps a scalpel piercing the wrong organ.

This sort of event system would be a great addition to the AutoMan's functionalities and may alleviate a great deal of work for the simulation developer.

6.2.3 Multithreading

This is something that has already been seriously thought about and development started. The framework has been designed to reduce the coupling between packages which should allow the multithreading of the framework to be easy and beneficial.

Each of the three managers should have a dedicated thread that should continually update regardless of the state of the other managers. The thread devoted to the `ApplicationDeviceManager` would continually receive user input from devices and make the required changes to the state of the system. Meanwhile the `RenderingManager` thread could continually supply feedback of the simulation to the user via the added rendering engines. The `SimulationManager` thread would then continually update the state of the simulation based on the physical properties of each model and any affects caused by user input.

This setup would mean that neither of the managers has to wait for another to finish before they can do their required task. This should allow the simulation to realistically simulate the scene whether or not the rendering engine can keep up. As mentioned before, this would increase the realism and fidelity of the system since the physical simulation would not slow down if there is a lag in the input handling or the rendering. All in all, it should increase the fidelity of the simulation and improve performance if the system running the application has multiple processors.

6.2.4 Rendering Engines

At the time of writing, there are only two non-trivial implementations of a `RenderingEngine`: `SofaQtRenderingEngine`, and `VRJugglerRenderingEngine`. This was enough to show the proof-of-concept, but for the framework to be successful there should be a lot more provided. The following sections explain some rendering engines that should be investigated and developed.

6.2.4.1 Commercial Rendering Library

Although VR Juggler can be considered a commercial library, looking into developing a rendering engine that is built on a high-end graphics rendering should be considered. Something like Ogre3D [6] would provide higher quality rendering capabilities such as dynamic shadows, the use of procedural shaders, and many algorithms that can improve rendering performance [29], [30]. Ultimately this could greatly improve the look of the simulations created in the framework.

6.2.4.2 Ray tracing

With the increase in the power and number of processors in common computer systems, the use of ray-tracing in real time applications has become more and more a viable option [31]. There are many benefits that come with ray-tracing over the conventional rasterizing method of rendering. These include trivially simple parallelization, correctness, occlusion culling [32], and direct computation of global effects [33].

Since AutoMan has separated the scene geometry from the rendering, incorporating a ray tracing rendering engine should be as simple as incorporating a new rasterizing rendering engine like the others described here. Furthermore, open source libraries like OpenRT [34] have begun to show up that would certainly aid in this endeavor.

6.2.4.3 Non-graphical Rendering Engines

As was shown in previous chapters of this thesis, the rendering engine interface supports rendering other than graphical. Some examples of this should be provided with the framework. One such example would be an engine that provides auditory feedback to the user. A rendering engine based on OpenAL [35], an open source audio API, would be a welcome addition to the framework. Other examples include, haptic feedback rendering for common devices such as the Phantom, and the output of object positions to a physical robot.

6.2.5 Helper Tools

Along with the additions directly to the framework itself, the entire project would greatly benefit from a couple of helper tools. A helper tool is a separate application that aids the in the use of the AutoMan framework. These include tools like the VS .Net Project Wizard. The following sections describe some other tools that would be beneficial in aiding the users of AutoMan.

6.2.5.1 Scene File Creator / Editor

AutoMan supports the loading of SOFA scene (.scn) files. However, these files have to be manually written. It would be nice to have a GUI that would allow the user to input object models and place them in a scene.

The loaded models could then be scaled and oriented to the required position in the scene space. The user should also be able to set the global scene parameters such as gravity.

Once the scene has all of the objects positioned and oriented, the scene can be exported to a file, ready for loading into the simulation application.

Having a file that contains all the information of the scene's initial conditions trivializes the implementation of the *createScene()* method. In fact, the application built as the case study for this thesis did just that. No actual coding would be required if users had access to a scene file creator. This would open the use of the framework to users that have zero programming expertise whatsoever.

6.2.5.2 RenderingEngine Creator

Although it is expected that the users creating new `RenderingEngine` implementations will be advanced users, it would be nice to streamline the process. A method of streamlining rendering engine development would need to be designed and analyzed, but the benefits of this endeavor are obvious. The greater the variety of modules that is available for use with AutoMan, the greater the chances of drawing in more users. It is difficult to state what this tool would look like, but it would definitely be beneficial to find an easier way to develop new `RenderingEngine` implementations.

Bibliography

[1] Satava, R. M., "Accomplishments and challenges of surgical simulation." New York : Surgical Endoscopy, 2001, Issue 3, Vol. 15.

[2] Montgomery, K., et al., "Spring: A General Framework for Collaborative Real-time Surgical Simulation." Newport Beach, CA : Proceedings of Medicine Meets Virtual Reality (MMVR02), 2002.

[3] Goktekin, Tolga G. and Cavusoglu, M. C., "GiPSi: An Open Source/Open Architecture Software Development Framework for Surigcal Simulation." Newport Beach, CA : Proceedings of Medicine Meets Virtual Reality IIX (MMVR 2004), 2004.

[4] Tuchschmid, S., et al., "A Flexible Framework for Highly-Modular Surgical Simulation Systems." Zurich, Switzerland : Proceedings of Biomedical Simulation: Third International Symposium, ISBMS 2006, 2006.

[5] Allard, Jérémie, et al., "SOFA – an Open Source Framework for Medical Simulation." Newport Beach, CA : Medicine Meets Virtual Reality (MMVR), 2007.

[6] , Ogre 3D: Open Source Graphics Engine. [Online] [Cited: May 2, 2008.] www.ogre3d.org.

[7] Cotin, S. M., et al., "Collaborative Development of an Open Framework for Medical Simulation." s.l. : 2005 MICCAI Open-Source Workshop, 2005.

[8] , VR Juggler—Open Source Virtual Reality Tools. [Online] [Cited: April 28, 2008.] http://oldsite.vrjuggler.org/.

[9] Bruyns, Cynthia D. and Montgomery, Kevin., "Generalized Interactions Using Virtual Tools within the Spring Framework: Cutting." Newport Beach, CA : Medicine Meets Virtual Reality (MMVR02), 2001.

[10] Bruyns, Cynthia D. and Montgomery, Kevin., "Generalized Interactions Using Virtual Tools within the Spring Framework: Probing, Piercing, Cauterizing, and Ablating." Newport Beach, CA : Medicine Meets Virtual Reality (MMVR02), 2001.

[11] Valve Corporation., *Source Engine.* [Online] Valve Corporation. [Cited: May 2, 2008.] http://source.valvesoftware.com/.

[12] id Software., Technology Licensing: id Tech 4. [Online] id Software. [Cited: May 2, 2008.] http://www.idsoftware.com/business/idtech4/.

[13] Pressman, Roger S., *Software Engineering: A Practitioner's Approach, Sixth Edition.* New York : McGraw-Hill, 2005. 0-07-285318-2.

[14] , Blender. [Online] [Cited: May 2, 2008.] http://www.blender.org/.

[15] Sommerville, Ian., *Software Engineering.* New York, NY : Addison-Wesley, 2007.

[16] Krasner, Glenn E. and Pope, Stephen T., "A Cookbook for Using hte Model-View Controller User Interface Paradigm in Smalltalk-80." Journal of Object-Oriented Programming, Denville, NJ : SIGS Publications, 1988, Issue 3, Vol. I.

[17] Smith, Russell., "Open Dynamics Engine v0.5 User Guide." *Open Dynamics Engine.* [Online] February 23, 2006. [Cited: May 2, 2008.] http://www.ode.org/ode-latest-userguide.pdf.

[18] , "VR Juggler: The Programmer's Guide." [Online] [Cited: May 2, 2008.] http://developer.vrjuggler.org/docs/vrjuggler/2.2/programmer.guide/programmer.guide.html.

[19] McGregor, Robert W., *Practical C++.* [World Wide Web] Indianapolis, Ind. : Safari Books Online, 1999.

[20] Chan, Ben Chun Pong., CodeGuru: Delegate in Standard C++. *CodeGuru.* [Online] Janurary 2002. [Cited: April 21, 2008.] http://www.codeguru.com/cpp/cpp/cpp_mfc/article.php/c4119/.

[21] Cleeland, Chris, Schimdt, Douglas C. and Harrison, Timothy H., External Polymorphism - An Object Structural Pattern for Transparently Extending C++ Concrete Data Types. *Department of Computer Science and Engineering.* [Online] Washington

University in St. Louis, October 22, 1996. [Cited: April 21, 2008.] http://www.cs.wustl.edu/~cleeland/papers/External-Polymorphism/External-Polymorphism.html.

[22] Trolltech., Qt. [Online] Trolltech. [Cited: May 2, 2008.] http://trolltech.com/products/qt.

[23] Paoli, S., C++ Coding Standard Specification. *CERN Project Support Team.* [Online] CERN, Januray 5, 2000. [Cited: April 21, 2008.] http://pst.web.cern.ch/PST/HandBookWorkBook/Handbook/Programming/CodingStandard/c++standard.pdf.

[24] , Doxygen Source code documentation generator tool. [Online] [Cited: May 2, 2008.] www.doxygen.org.

[25] Mader, Stewart., *Wikipatterns.* Indianapolis, IN : Wiley, 2008.

[26] Tohen, Cal and Gil, Joseph., "Better Construction with Factories." s.l. : Journal of Object Technology, 2007, Issue 6, Vol. 6.

[27] Shalloway, Alan and Trott, James., *Design Patterns Explained: A New Perspective on Object-Oriented Design.* Boston, Mass. : Addison-Wesley, 2001.

[28] Raymond, Eric S., *The Cathedral and The Bazaar (1st ed.).* Beijing : O'Reilly, 1999.

[29] Fernando, Randima., *GPU Gems.* Boston, MA : Addison-Wesley, 2004.

[30] Nguyen, Hubert., *GPU Gems 3.* Upper Saddle River, NJ : Addison-Wesley, 2008.

[31] Benthin, Carsten, et al., "Ray Tracing on the Cell Processor." s.l. : 2006 IEEE Symposium on Interactive Ray Tracing, 2006.

[32] Wald, Ingo, et al., "Interactive Rendering with Coherent Ray Tracing." s.l. : Computer Graphics Forum, 2001. Eurographics.

[33] Woop, Sven, Schmittler, Jorg and Slusallek, Philipp., "RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing." Los Angeles, CA : ACM Siggraph 2005, 2005.

[34] , OpenRT Real-Time Ray Tracing Project. [Online] [Cited: May 3, 2008.] http://www.openrt.de/.

[35] OpenAL Cross-Platform 3D Audio. [Online] [Cited: May 3, 2008.] http://www.openal.org/.

[36] Montgomery, Kevin, et al., "Project Hydra - A New Paradigm of Internet-Based Surgical Simulation." Newport Beach, Ca : Medicine Meets Virtual Reality, 2006.

Appendix A Software Requirements Specification

A Layered Framework for Surgical Simulation Development

Software Requirements Specification

Version 2.1

Revision History

Date	Version	Description	Author
5/26/2006	1.0	Initial Creation.	Tim Hayes, Ryan Weiss
10/4/2007	1.1	Updated Sections 1-3.1	Tim Hayes
4/9/2008	2.0	Full document overhaul.	Tim Hayes
4/15/2008	2.1	Updated Supporting Information	Tim Hayes

A.1 Introduction

This Software Requirement Specification document describes in detail the requirements for a development framework for surgical simulators. This framework will be designed and developed by current and future students of Dr. Rajni Patel of The University of Western Ontario.

The framework will aide in the creation of surgical simulations for the purpose of training. This includes the display and manipulation of basic 3D models and volumetric based models. It will also handle soft tissue modeling and rigid body dynamics on the displayed objects to achieve a greater sense of realism in the simulation. A number of different interaction devices can be supported via a generic I/O interface. This will facilitate the use of tracking sensors, wands, gloves, haptic devices and even custom built devices.

A.1.1 Purpose

The SRS is designed to fully and unambiguously describe the development framework for surgical simulators – its features, its operating environment and its functional performance. For developers, technical readers, or anyone who requires detail, Section 3 of this document will list all functional and non-functional requirements along with a description for each. For those readers that do not require such detail, Section 2 outlines the main functionality providing a more immediate overview of this framework.

A.1.2 Scope

This document details the requirements for a development framework for surgical simulators. It outlines all of the functional requirements of the system as well as the non functional requirements. This document does not perform any analysis nor provide any design for the system. The details contained within this document should be used in the design of the system, but are not discussed here. This document should be referenced at the end of the development cycle to compare the finished project with the pre-design requirements. This should ensure that the final project is completed as originally planned and discussed before the start of the design and development phases.

A.1.3 References

1. Montgomery, K., Bruyns, C., Brown, J., Sorkin, S., Mazzella, F., Thonier, G., Tellier, A., Lerman, B., Menon, A.: *Spring: A General Framework for Collaborative*, Real-time Surgical Simulation, In: Westwood, J., et. al. (eds.): Medicine Meets Virtual Reality, IOS Press, Amsterdam, (2002).

2. R. M. Satava. "Accomplishments and challenges of surgical simulation", *Surgical Endoscopy, vol. 15, no. 3*: 232-241, Springer New York, March 2001

3. Cotin, Stephen; Delingette, Hervé; Ayache, Nicholas. "Real-Time Elastic Deformations of Soft Tissues for Surgical Simulations." *IEEE Transactions on*

Visualization and Computer Graphics, vol. 5, no. 1, March 1999: 62-73. IEEE Xplore. 25 May 2006. <http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=764872>

4. Vuskovic, V; Kauer, M.; Szekely, G; Reidy M. "Realistic force feedback for virtual reality based diagnostic surgery simulators." *IEEE International Conference on Robotics and Automation, 2000, vol 2*: 1592-1598. IEEE Xplore. 25 May 2006. <http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=844824>

A.1.4 Overview

This report contains the following sections:

1. **Introduction:** Provides a brief overview of the project and discusses the purpose and scope of this document.

2. **Overall Description:** Describes in general the requirements of the project. This includes the perspective, main functionality, and constraints.

3. **Specific Requirements:** Contains specific details on each requirement to a level sufficient enough to produce a complete design of the system. Each feature presented here should be found in the finished project.

4. **Supporting Information:** Includes information to help the reader further understand the project detailed in this document.

A.2 Overall Description

This section describes the general requirements of the project. This includes the perspective, main functionality, and constraints. Further description of the features presented here can be found in Section 3 of this document.

A.2.1 Product Perspective

The benefits of a computer-based surgical simulators are numerous and significant. For example, computer-based surgical simulators easily widen the training curriculum by facilitating anatomical variations (gender, size), pathologies (diseases, trauma), and operating environment conditions (emergency room, microgravity, battlefield)[1]. Furthermore, they provide the ability of objectively quantify the performance of a surgeon's skills[2] and a simulation can be repeated indefinitely without the cost of purchasing cadavers or animals for training. The benefits are undeniable; the problem is that surgical simulations are difficult and costly to produce.

The framework described in this document will provide a platform from which powerful simulation applications can be developed quickly and easily in a relatively short amount of time. This reduces development cost and will encourage organizations to undertake the development of surgical simulations for training. The use of this framework will allow the developers to focus more on content than worry about the technical difficulties.

A.2.2 Product Functions

This section presents a list of functions that the framework will perform. Details on each feature will be discussed in Section 3.

- Load model geometry.
    - Load common file formats.
    - Load files for each model representation.
    - Load scene file.
- Represent a Model.
    - Handle multiple representations: visual, collision, and behavioral.
    - Specify type of model: Ambient, Rigid, Deformable, etc
    - Specify type of representations: FEM, Mass-Spring, etc.
- Simulate realistic physics and object interactions.
    - Environment physics and Newtonian Mechanics.
    - Object-object collision detection and response.
    - Object deformation from acting force.
- Render scene
    - Specify methods of rendering the scene: visual, auditory, haptic, etc.
    - Ability to define new methods of rendering.
        - Supply at least one implementation.
- Manipulate scene

- Ability to use input devices to manipulate the scene.

A.2.3 User Characteristics

The primary users of the framework are software developers that should have a good understanding of the C++ programming language, and have some background in developing for real-time, graphical applications. It can also be assumed that the user has either been trained, or has is being guided by someone who knows how the surgery being simulated is to be performed. Ideally this would be a trained surgeon who has a lot of expertise in the specific surgery being simulated, and a basic knowledge of computer programming. These users will be deploying the framework to build specific applications to be used for surgical purposes.

A second set of users, although not specifically the prime target of the project should be addressed. These users are developers who enjoy working with new technology, specifically the cutting edge of computer graphics and simulation. It is expected that they are well versed in the C++ programming language and have a great understanding of real-time computer graphics. It can be assumed that they have a basic knowledge of Newtonian mechanics and the physics of rigid bodies; however it cannot be assumed that they have a good understanding of the physics of deformable models and fluids. These users may be using the framework for purposes not specifically surgical in nature.

A.2.4 Constraints

To aid the users of this framework, a comprehensive API and tutorials will be supplied. This will include the programming interfaces, UML documentation, and coded examples. Verbose commenting of all code, especially the example applications, should be incorporated to aid the developer understand the feature's structure and function.

To also aid in this respect, the framework should use generic, common interfaces to the framework. Common coding standards and practices add as sense of which will round out the learning curve thus reducing overall development time.

The framework should have an easy method upgrading and expanding. Should developers want to add features to the framework, a simple, easy method should be provided along with a method of distributing the addition to the other users of the framework.

A.3 Specific Requirements

This section contains specific details on each requirement to a level sufficient enough to produce a complete design of the system. Each feature presented here should be found in the finished project.

A.3.1 Functionality

This section describes the main features of the system. Refer to Section 4.1.2 – Use Case Diagram for supplementary information on the behavior described here.

A.3.1.1 Load geometry

Purpose: For visualization of simulation objects.

Input: The model file, with position and orientation in the virtual world.

Output: The model will be loaded for use later in the simulation

Exceptions: Invalid file formats or corrupt data will result in a failure to load the data and a notification displayed to the user.

A.3.1.2 Load common file formats

Purpose: Allows users to define models in common CAD packages and then import them into the simulation.

Input: The model file.

Output: The model will be loaded for use in the simulation

Exceptions: Invalid file formats or corrupt data will result in a failure to load the data and a notification displayed to the user.

A.3.1.1 Load files for each model representation

Purpose: Allows users to specify different models for each representation. For example, high poly version for rendering, low poly for collision, etc.

Input: A model file for each representation.

Output: The specified model will be loaded for use as its respective model representation.

Exceptions: Invalid file formats or corrupt data will result in a failure to load the data and a notification displayed to the user.

A.3.1.2 Load a scene file

Purpose: Allows the user to specify one file that defines the entire scene instead of manually loading and placing all the objects within the scene.

Input: A scene file.

Output: The specified scene will be loaded for use in the simulation.

Exceptions: Invalid file formats or corrupt data will result in a failure to load the data and a notification displayed to the user.

A.3.1.3 Represent a Model

Purpose: The system needs to understand that the loaded geometry defines a simulation object with physical properties.

Input: Object geometry and physical properties.

Output: None.

Exceptions: None.

A.3.1.4 Handle multiple representations

Purpose: To help reduce unnecessary calculations and increase the performance of the application.

Input: The user specifies different representations of the same model to be used for determining collision detection and response, physical behavior, and visual representation.

Output: The model will be represented using the visual model, but collision calculations will be processed on the collision model and physical behavior calculations will be performed on the physical representation.

Exceptions: If only the visual representation is specified, the bounding box will be used in collision calculations and the specified visual representation will be used for both visualization and physical behavior.

A.3.1.5 Specify type of model

Purpose: Allows users to define simulation objects as Ambient, Static, Rigid, and Deformable.

Input: Specified by the object constructor used.

Output:

- Ambient – purely visual, does not affect the simulation
- Static – affects the simulation but does not move
- Rigid – affects the simulation and moves but does not deform
- Deformable – affects the simulation, moves and is deformed due to applied forces

Exceptions: None.

A.3.1.6 Specify type of representations

Purpose: To allow more freedom in model definition. Certain objects will react as roughly expected using a mass-spring system, where others may require an FEM model representation.

Input: Determined via a class access method call.

Output: The model will be simulated using the specified system.

Exceptions: None.

A.3.1.7 Simulate realistic physics and object interactions

Purpose: To improve the realism of the simulation.

Input: The scene's initial condition and physical properties must be loaded before the simulation begins. Then the simulation must be started.

Output: The system will constantly calculate the current world condition (ie: object placement, shape, etc.) and update the visualization device as necessary.

Exceptions: None.

A.3.1.8 Simulate environment physics and Newtonian mechanics

Purpose: To improve the realism of the simulation.

Input: The scene's initial condition and physical properties must be loaded before the simulation begins. This includes the gravitational force and other worldly forces.

Output: The system will calculate the current simulation condition (ie: object placement, shape, etc.) based on the defined forces and update the visualization device as necessary.

Exceptions: None.

A.3.1.9 Simulate Object-Object collisions

Purpose: To improve the realism of the simulation.

Input: The scene's initial condition and physical properties of each model must be loaded before the simulation begins. Then two objects need to move and collide.

Output: The system will calculate the response to the collision and update the state of simulation (ie: object placement, shape, etc.) and update the visualization device as necessary.

Exceptions: None.

A.3.1.10 Simulate Object deformation

Purpose: To improve the realism of the simulation.

Input: The scene's initial condition and physical properties of each model must be loaded before the simulation begins. Then a force needs to be applied to a deformable model, this could be gravity or a collision with another object, etc.

Output: The system will calculate the response of the force on the deformable model's surface and update the visual representation.

Exceptions: None.

A.3.1.11 Render the scene

Purpose: Gives feedback of the simulation state to the user.

Input: The state of the simulation's scene.

Output: The simulation state will be rendered to a device.

Exceptions: None.

A.3.1.12 Specify rendering method

Purpose: Allows user to specify what feedback they wish to receive about the simulation state.

Input: The state of the simulation's scene and a type of feedback they wish to receive. This may include visual feedback to a monitor display, or auditory feedback to speakers.

Output: The simulation state will be rendered in the specified manner.

Exceptions: None.

A.3.1.13 Define new rendering method

Purpose: Allows user to define new methods of receiving feedback from the system.

Input: An implementation of the new rendering method.

Output: The simulation state will be rendered in the specified manner.

Exceptions: None.

A.3.1.14 Manipulate the scene

Purpose: Allows the user to use input devices to manipulate the scene.

Input: The user must perform an action on an input device that has been mapped to an action in the simulation.

Output: The mapped simulation action will be performed.

Exceptions: None.

A.3.2 Quality of Service

A.3.2.1 Usability

A comprehensive API and tutorials should be supplied. This should include the programming interfaces, UML documentation, and coded examples. Verbose commenting of all code, especially the example applications, will be incorporated to aid the developer understand the feature's structure and function.

The code will use generic, common interfaces to the framework. Common coding standards and practices should round out the learning curve thus reducing overall development time.

A.3.2.2 Scalability / Extensibility

The project should have an easy method upgrading and expanding. This increases both the scalability and extensibility of the framework. Supplementary to this, a method of distributing any addition to the other users of the framework should also be provided. These interfaces should be well documented to encourage users to develop the additional modules.

A.3.2.3 Performance

There are a lot of factors that affect the performance of a simulation. However, the two main factors are the computational power of the system running the application, and the complexity of the simulation.

A.3.2.4 Computational Power

Computational power of the system is not something that can be addressed in software, but it is something that requires thought during the design of the framework. Simulations developed in the framework, should be able to run the sample applications at a real time frame rate on most workstation-grade computers with a moderate to high end GPU. The performance may be enhanced by running the simulation on more powerful hardware.

A.3.2.5 Complexity of the simulation

To help reduce the complexity of calculations in the simulation, the user must be allowed to define a lower quality model for use in the collision detection and response algorithms. These algorithms with a high degree of computations really hinder the performance of a simulation. Allowing them to be run against simplified versions of the object models, reduces the computations required and ultimately increases the performance of the system.

A.3.2.6 Accessibility

A web site devoted to the framework should be created. It will link to downloads the software binaries for immediate use, as well as code samples, tutorials, and API documentation.

Also, the site should provide a method for users to download the source code of the framework allowing them to directly modify the core of the framework. This is how the advanced users can make modifications and upgrades and allows the framework to be scalable and extensible.

A.3.3 Design Constraints

A.3.3.1 Hardware Limitations

The framework's performance will be limited to the hardware specifications of the users' computer, depending on the processor speed, video rendering capabilities, and the amount of ram installed.

A.3.3.2 Software Language

The framework will be implemented in C++ and compiled using the Visual Studio .Net 2005 C++ compiler.

A.3.3.3 Software Process Requirement

The software process for this software will be based on the incremental model. However, the framework should be designed as a whole to ensure that the modules therein work together efficiently. The implementation of the software will be more incremental, adding features and then testing before the next feature is implemented.

A.3.4 On-line User Documentation and Help System Requirements

The framework will provide a web site that offers documentation and tutorials. This includes the source code API.

Included with the tutorials should be a User Guide that walks new users through the installation and setup of the software. Also, a General Programmer Guide should be provided to help users with the general setup and running of applications built using the framework. Aside from that, there should be a method for users to distribute their own tutorials on development specifics.

A.3.5 Purchased Components

Development of the framework includes the use of various applications, all of which would be installed on Windows XP based PCs. Visual Studio 2005 will be used for the coding and compiling of the software. Visio 2005 will be used to develop UML diagrams to design the structure of the software. Visual Studio 2005 will allow the software to be developed and complied under the same IDE. Additional software that will be used includes Office 2007 and Dreamweaver for documentation.

A.3.6 Legal, Copyright, and Other Notices

A.3.6.1 License

The project is to be released under the Lesser Gnu Public License. For a full description of the license, please refer to section 4.2 LGPL.

A.3.6.2 Disclaimer of Warranties

Yet to be discussed.

A.3.6.3 Logo

Yet to be discussed.

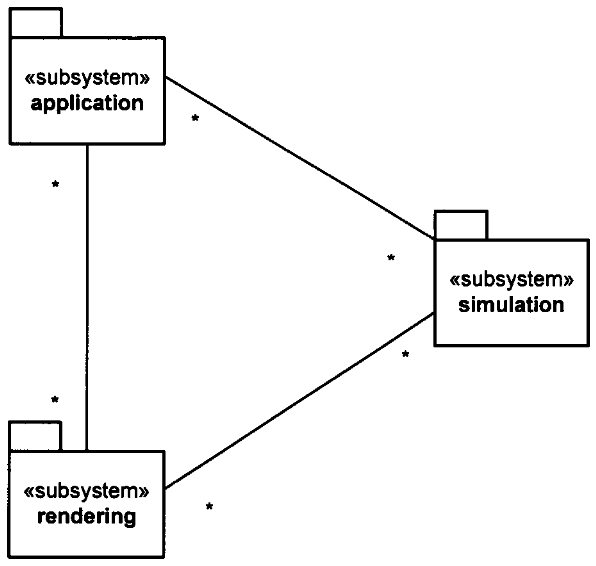
A.3.6.4 Restrictions on Use

Refer to the 4.2 LGPL.

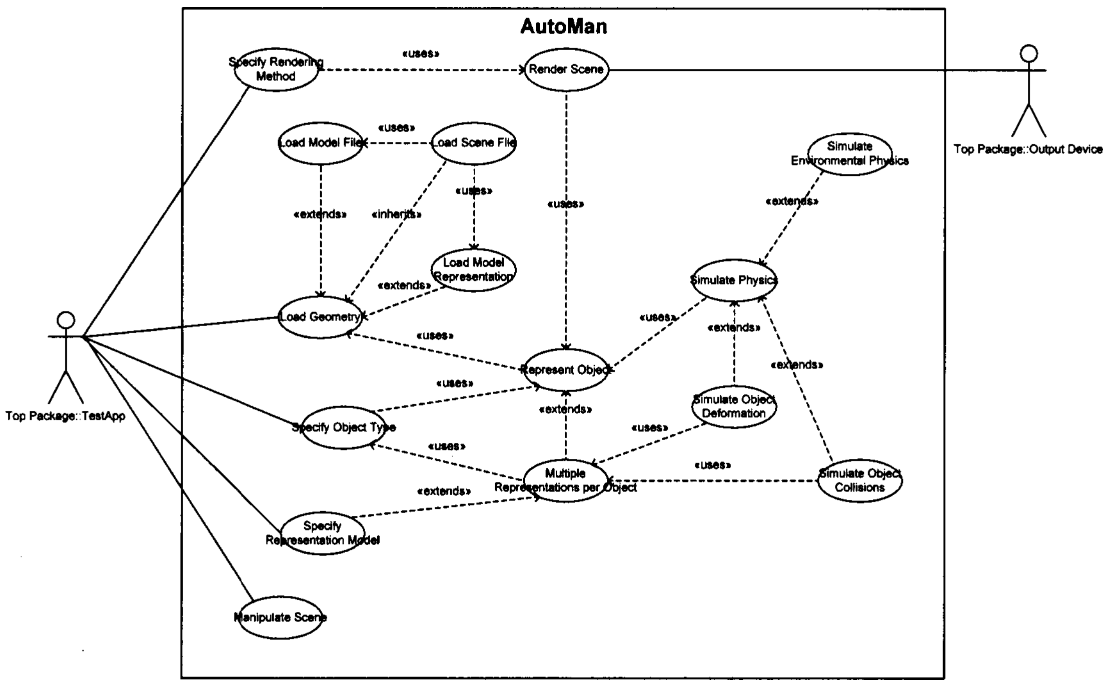
A.4 Supporting Information

A.4.1 Supporting Diagrams

A.4.1.1 Package Diagram



A.4.1.2 Use Case Diagram



A.4.2 GNU LGPL

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates

the terms and conditions of version 3 of the GNU General Public

License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the

function or data, the facility still operates, and performs
whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of
this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from
a header file that is part of the Library. You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the
Library is used in it and that the Library and its use are
covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license
document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form

suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may

Appendix B C++ Coding Standards Document

C++ Coding Standard

AutoMan Framework

Department of Electrical and Computer Engineering

The University of Western Ontario

London, Ontario, Canada

Revision History

Date	Version	Description	Author
10/1/2008	1.0	Initial Creation	T. Hayes

B.1 Introduction

This chapter describes purpose of this document and briefly examines the importance of coding standards. Then the intended audience is described, followed by a list of references. The chapter ends with an overview of the contents of this document.

B.1.1 Purpose

This document is intended to define a C++ coding standard to be followed during the development and maintenance of C++ applications. Standards such as the ones defined in the following sections aid in allowing the application's code to:

- Have a consistent style throughout the environment.
- Be easy to read and understand for new developers.
- Be maintainable by other developers [1].

All hand written code should follow the guidelines defined in this document. However, computer generated code is exempt from these guidelines as it was likely generated under a different set of standards.

If this document does not implicitly state a convention, the developer is free to user his or her own discretion.

B.1.2 Intended Audience

This document is specifically intended for current and future developers involved in the creation of AutoMan, from The University of Western Ontario (UWO) Department of Electrical and Computer Engineering and Canadian Surgical Technologies and Advanced Robotics (CSTAR).

B.1.3 References

1. http://pst.web.cern.ch/PST/HandBookWorkBook/Handbook/Programming/CodingStandard/c++standard.pdf
2. http://www.cs.northwestern.edu/academics/courses/311/html/coding-std.html#NameSensible
3. http://www.possibility.com/Cpp/CppCodingStandard.html#cuh
4. http://www.infospheres.caltech.edu/resources/code_standards/java_standard.html
5. http://geosoft.no/development/cppstyle.html

B.1.4 Organization of this Document

This document is organized in the following manner:

1. Introduction – this section
2. Naming – description of naming conventions
3. Documentation – description of documentation convention
4. Indenting and Spacing – description of indenting and spacing conventions
5. Code Examples – examples of code that follows the described conventions

B.2 Naming

This chapter describes the conventions that should be used for naming files and items within the code.

B.2.1 Naming Guidelines

The following are some guidelines that should considered when naming any element. This includes the naming of files, classes, methods and variables.

- Use pronounceable English names.
- The name should be descriptive of the element it represents.
- Abbreviations should be avoided except when widely accepted [1].

B.2.2 File Names

Header files should be named after the name of the class it defines appended with ".h". If more than one class is defined within the same header files, it should be named after the root of the class inheritance tree or the most prominent class defined therein.

Implementation files should follow the same pattern as the header files except with ".cpp" appended instead.

B.2.3 Class Names

Class names should be first letter capitalized and should describe the function of the class it represents.

If the class is within the framework, the name should be prefixed with the name of the package it is in.

Examples: `RenderingFrameListener`, `SimulationObject`.

B.2.4 Method Names

Methods should be lower-case and logical. If the name is more than one word, concatenate all the words and capitalize the first letter of every word excluding the first.

Examples: `start(),attachFrameListener(),createScene()`.

B.2.5 Variable Names

Variable name should be all lower case. If the name is more than one work, each word should be separated with an underscore, '_'.

Member variables should also be prefixed with an underscore, '_'.

If the variable has a very small scope, such as within a single method or a loop then it is not required to follow these conventions.

Examples: `mouse_evt, _frame_listeners, _sim_mgr`.

B.3 Formatting the Code

B.3.1 Tabs and Lines of Code

Every line of code should start with one tab for every nested block that the line is within. This makes the blocks more noticeable and easier to read.

No line should exceed 80 characters in length. This should avoid the need to side scroll on most IDE environments on most screen resolutions.

B.3.2 Braces

Follow all control primitives (if, else, while, for, do, switch, and case) with a block unless the contents is one instruction and can fit on the same line as the primitive.

Examples:

```
while ( condition )
{
     Statement;
     Statement;
}

if ( condition ) Statement;    // This is acceptable.

if ( condition )
     Statement;                // This is NOT acceptable.
```

B.4 Documentation

All comments should follow the DOxygen standard for documenting the code, found at http://www.stack.nl/~dimitri/doxygen/docblocks.html. This will make it easy to keep the API up to date.

B.5 Coded Examples

The following are the header and implementation file for the Application Class. This is shown as an example of correctly standardized code.

B.5.1 Application.h

```
#ifndef __APPLICATION_H__
#define __APPLICATION_H__

#include <exception>

#include "application/ApplicationDeviceManager.h"
#include "rendering/RenderingManager.h"
#include "simulation/SimulationManager.h"

/** \brief App Object.
 *
 *     This class defines an application.  User applications should
 *     extend this class and implement the virutal methods defined
 *     here.
 */
class Application
{
public:
/** \brief Constructor.
 *
 *  Default Constructor.
 */
      Application();
/** \brief Deconstructor.
 *
 *  Default Deconstructor.
 */
      ~Application();

/** \brief Starts the application.
 *
 *  This starts the initialisation, setup, and execution of
 *     the application.
 */
      void startApp();

/** \brief Gets the Rendering Manager.
 *
 *  Returns a pointer to the application's rendering manager.
 *     @return      a pointer to the rendering manager.
 */
      inline RenderingManager * getRenderingManager()
      { return _rendering_mgr; };

/** \brief Gets the Simulation Manager.
 *
 *  Returns a pointer to the application's simulation manager.
 *     @return      a pointer to the simulation manager.
 */
      inline SimulationManager * getSimulationManager()
      { return _sim_mgr; };

/** \brief Gets the Device Manager.
 *
 *  Returns a pointer to the application's device manager.
 *    @return      a pointer to the device manager.
 */
      inline ApplicationDeviceManager * getDeviceManager()
      { return _dev_mgr; };
protected:
/** \brief The main execution loop.
 *
 *  This is the main execution loop.
 */
      void mainLoop();

/** \brief Initialises the managers.
 *
 *  This method initialises the managers by calling each respective
 *     initialise method.
 *     @return True if initialisation was successful.
 */
      virtual bool initialise();
/** \brief Constructs the scene to be rendered.
 *
 *  This method constructs the scene to be rendered.  It should be
 *     overridden by the user application.  A default scene is defined here.
 */
      virtual void constructScene();
/** \brief Destroys the created scene.
 *
 *  This method destroys the scene.  It should be overridden by the user
 *     application.
 */
      virtual void destroyScene();

/** \brief Constructs the Rendering Manager.
 *
 *  Constructs the Rendering Manger.  It needs to be implemented in the user
 *     application.  This is a puerly virtual function.
 */
      virtual void constructRenderingManager() = 0;
/** \brief Constructs the Application Manager.
 *
 *  Constructs the Application Manger.  It needs to be implemented in the user
 *     application.  This is a puerly virtual function.
 */
      virtual void constructDeviceManager() = 0;
/** \brief Constructs the Simulation Manager.
 *
 *  Constructs the Simulation Manger.  It needs to be implemented in the user
 *     application.  This is a puerly virtual function.
 */
      virtual void constructSimulationManager() = 0;
/** \brief Method for any user application specific setup.
 *
 *  Method for any user application specific setup. It needs to be implemented
 *     in the user application.  This is a puerly virtual function.
 */
      virtual void setup() = 0;


      RenderingManager          * _rendering_mgr;
            /**< Pointer to the rendering manager. */
```

```
        ApplicationDeviceManager    * _dev_mgr;
                /**< Pointer to the device manager. */
        SimulationManager           * _sim_mgr;
                /**< Pointer to the simulation manager. */

#ifdef MULTITHREADED
        ApplicationThread<SimulationManager> * _sim_thread;
        ApplicationThread<ApplicationDeviceManager> * _dev_thread;
#endif
};

#endif //__APPLICATION_H__
```

B.5.2 Application.cpp

```
#include "Application.h"

Application::Application()
{};

Application::~Application()
{};

bool Application::initialise()
{
	//initialise the world manager
	if( !_sim_mgr->initialise() ) return false;

	// initialise the rendering engine
	if( !_rendering_mgr->initialise() ) return false;

	// initialise the application manager
	if( !_dev_mgr->initialise() ) return false;

	//any other initialisation

	return true;
};

void Application::constructScene()
{};

void Application::startApp()
{

	constructSimulationManager();
	_root = _sim_mgr->getSimulationRoot();

	constructRenderingManager();

	constructDeviceManager();

	constructScene();

	if(!initialise()) throw new std::exception();

	setup();

#ifndef MULTITHREADED
	_rendering_mgr->start();
#else
	_sim_thread = new ApplicationThread<SimulationManager>(_sim_mgr,
&SimulationManager::start);
	_app_thread = new ApplicationThread<ApplicationManager>(_app_mgr,
&ApplicationManager::start);

	_sim_thread->start();
	_app_thread->start();
	_rendering_mgr->start();
#endif
```

```
        destroyScene();
};

void Application::mainLoop()
{
        bool finished = false;

        while(!finished)
        {
                //finished = _app_mgr->update();
                //_rendering_mgr->update();
                //_rendering_mgr->start();
        }
};

void Application::destroyScene()
{
};
```

Appendix C Test Plan Document

Test Plan

AutoMan Framework

Department of Electrical and Computer Engineering

The University of Western Ontario

London, Ontario, Canada

Revision History

Date	Version	Description	Author
10/1/2008	1.0	Initial Creation	T. Hayes

C.1 Introduction

This document is intended to define a test plan for the quality assurance of the AutoMan project.

C.1.1 Supporting Documents

The following table lists documents that are required reading before reviewing this test plan. These documents provide much information such as the project scope, background, and operation instructions.

Title	Description	Filename
C++ Coding Standards	Describes the coding standards to be used during development.	AutoMan_CodingStandards
Software Requirement Specification	Details all requirements of the project.	AutoMan_SRS_v##
Installation Guide	Describes the software's installation process.	AutoMan_InstallGuide
Getting Started Guide	Walks through the development of a simple application in AutoMan	AutoMan_GettingStartedGuide
Project Wizard User Guide	Describes the installation and use of the AutoMan Simulation Application Project Wizard.	AutoManProjectWizard_UserGuide

C.2 Test Approach

Considering this project does not have the resources for a full thorough testing phase, only rough overall black box testing will be performed.

The items to be tested along with the specific cases are defined in subsequent sections of this document. These test cases should be performed internally by the developer.

The results of each test case should be tabulated into a Test Report.

C.3 Test Items

The following is a list of the modules and programs to be tested. The individual features that need to be tested within each will be addressed in subsequent sections.

- Installation Process
- VS .Net Project Wizard
- AutoMan – Application Module
- AutoMan – Rendering Module
- AutoMan – Simulation Module
- Website

C.4 Features to be Tested

This section details the specific features of each module / program that need to be tested. A set of tests should be devised for each feature listed here.

C.4.1 Installation Process

C.4.1.1 Installation Guide

Objective:	Explain the proper installation procedure for AutoMan.
Prerequisites:	A copy of the AutoMan Installation Guide document.
Test Process:	Follow steps as stated in Install Guide
Acceptance Criteria:	The system should have a valid installation of AutoMan.

C.4.2 VS .Net Project Wizard

C.4.2.1 Installation Guide

Objective:	Explain the proper installation procedure for the Visual Studio Project Wizard
Prerequisites:	A copy of the AutoMan Project Wizard User Guide document. Test Case C.4.1.1 passed.
Test Process:	Follow steps for installation as stated in the guide.
Acceptance Criteria:	The system should have a valid installation of the project wizard. This implies they should be able to select an 'AutoMan Simulation Project' from the list of project templates when they create a new project in Visual Studio 2005.

C.4.2.2 Generates project

Objective:	A Visual Studio 2005 project should be created.
Prerequisites:	Test Case C.4.2.1 passed.
Test Process:	Run the project wizard. Input "THayes" as the author's name. Input "TestClass" as the class name. Press the finish button.

Acceptance Criteria:	Visual Studio Project should be created.
	TestCase.h header file should be present.
	TestCase.cpp file should be present.
	Project should compile successfully.

C.4.2.3 FrameListener included [/ not included]

Objective:	Generated class should [/ should not] extend RenderingEngineFrameListener.
Prerequisites:	Test Case C.4.2.2 passed
Test Process:	Run the project wizard.
	Input "THayes" as the author's name.
	Input "TestClass" as the class name.
	Select [/ Deselect] "Include FrameListner" checkbox.
	Press the finish button.
Acceptance Criteria:	Visual Studio Project should be created.
	RenderingEngineFrameListener header should [/ should not] be included.
	TestCass class should [/ should not] extend RenderingEngineFrameListener
	Project should compile successfully.

C.4.2.4 VRJugglerRenderingEngine correctly included [/ not included]

Objective:	Generated class should [/ should not] use the VRJugglerRenderingEngine
Prerequisites:	Test Case C.4.2.2 passed.

Test Process: Run the project wizard.

Input “THayes” as the author’s name.

Input “TestClass” as the class name.

Select [/ Deselect] “VRJugglerRenderingEngine” radio option.

Press the finish button.

Acceptance Criteria: Visual Studio Project should be created.

VRJugglerRenderingEngine header should [/ should not] be included.

TestCass::constructRenderingManager() method should [/ should not] create an instance of VRJugglerRenderingEngine and attach it to the RenderingManager

Project should compile successfully.

C.4.2.5 SofaQtRenderingEngine correctly included [/ not included]

Objective: Generated class should [/ should not] use the SofaQtRenderingEngine

Prerequisites: Test Case C.4.2.2 passed.

Test Process: Run the project wizard.

Input “THayes” as the author’s name.

Input “TestClass” as the class name.

Select [/ Deselect] “SofaQtRenderingEngine” radio option.

Press the finish button.

Acceptance Criteria: Visual Studio Project should be created.

SofaQtRenderingEngine header should [/ should not] be included.

TestCass::constructRenderingManager() method should [/ should not] create an instance of SofaQtRenderingEngine and attach it to the RenderingManager

Project should compile successfully.

C.4.3 AutoMan - Application Module

C.4.3.1 Initialize and start the simulation

Objective: Should initialize and start the simulation.

Prerequisites: Test Case C.4.2.2 passed.

Test Process: Create a project using the project wizard.

In the *createScene()* method, create add the following line,

```
sim mgr->loadSceneFile("collisionTriangle.scn");
```

Compile and run the simulation.

Acceptance Criteria: The simulation should start running.

C.4.4 AutoMan - Rendering Module

C.4.4.1 Attach a RenderingEngine to the RenderingManager

Objective: Should be allowed to specify which `RenderingEngine` is to be used.

Prerequisites: Test Case C.4.2.2 passed.

Test Process: Create a project using the project wizard.

In the *createRenderingManager()* method, create a `VRJugglerRenderingEngine`.

Call the `RenderingManager` *addEngine()* method and specify the created `RenderingEngine` as the argument.

Compile and run the simulation.

Acceptance Criteria: A VRJuggler window should appear and render the scene.

C.4.4.2 Handle multiple RenderingEngines

Objective: Should be allowed to specify multiple `RenderingEngine`'s that are to be used.

Prerequisites: Test Case C.4.4.1 passed.

Test Process: Create a project using the project wizard.

In the *createRenderingManager()* method, create a `VRJugglerRenderingEngine`.

Call the `RenderingManager` *addEngine()* method and specify the created `RenderingEngine` as the argument.

Also in *createScene()*, create a `DummyRenderingEngine`.

Add this engine to the `RenderingManager` using the *addEngine()* method.

Compile and run the simulation.

Acceptance Criteria: A VRJuggler window should appear and render the scene.

The console should print "Dummy Rendering Engine – update method called."

C.4.4.3 RenderingEngineFrameListener makes callbacks

Objective: `RenderingEngine` should inform attached frame listener when it is about to start rendering a frame and when it has finished rendering a frame.

Prerequisites: Test Case C.4.4.1 passed.

Test Process: Create a project using the project wizard selecting to include a frame listener.

In the *createRenderingManager()* method, create a VRJugglerRenderingEngine.

Call the *attachFrameListener()* method of the `RenderingEngine` and specify `this` as the parameter.

Call the `RenderingManager` *addEngine()* method and specify the created `RenderingEngine` as the argument.

Create a member variable, `_frame_count` in the test application class and initialize it to 0 in the constructor.

In the *frameStarted()* method, add the line,

```
std::cout << "Frame number " << frame counter << " started..." << std::endl;
```

In the *frameEnded()* method, add the line,

```
std::cout << "Frame number " << frame counter++ << " ended." << std::endl;
```

Compile and run the simulation.

Acceptance Criteria: Output to the console should be look like this,

```
Frame number 0 started...
Frame number 0 ended.
Frame number 1 started...
Frame number 1 ended.
Frame number 2 started...
Frame number 2 ended.
...
```

C.4.4.4 Handle multiple RenderingEngineFrameListeners

Objective: `RenderingEngine` should accept and handle multiple listeners.

Prerequisites: Test Case C.4.4.3 passed.

Test Process: Create a project using the project wizard selecting to include a frame listener.

Set up application as in Test Case C.4.4.3

Create a second class called `TestFrameListener` that extends `RenderingEngineFrameListener`.

In the *frameStarted()* method, add the following line,

```
std::cout << "TestFrameListener::frameStarted()called…" << std::endl;
```

In the *frameEnded()* method, add the following line,

```
std::cout << "TestFrameListener::frameEnded()called." << std::endl;
```

Compile and run the simulation.

Acceptance Criteria: Output to the console should be look like this,

```
Frame number 0 started…
TestFrameListener::frameStarted()called…
Frame number 0 ended.
TestFrameListener::frameEnded()called.
Frame number 1 started…
TestFrameListener::frameStarted()called…
Frame number 1 ended.
TestFrameListener::frameEnded()called.
...
```

C.4.5 AutoMan – Simulation Module

C.4.5.1 Load scene file

Objective: Should be allowed to specify multiple `RenderingEngine`'s that are to be used.

Prerequisites: Test Case C.4.4.1 passed.

Test Process: Create a project using the project wizard.

In the *createScene()* method, add the following line,

```
sim mgr->loadSceneFile("collisionTriangle.scn");
```

Compile and run the simulation.

Acceptance Criteria: The collisionTriangle simulation should run.

C.4.5.2 Create Simulation Objects

Objective:	Should be able to create objects that are static, rigid or deformable in the scene.
Prerequisites:	Test Case C.4.4.1 passed.

<u>Test Process:</u> **Create a project using the project wizard.**

In the *createScene()* method, add the following lines,

```
SimulationRootNode * root = _sim_mgr->getSceneRoot();
SimulationNode * chain = new SimualationNode();
root->addChild( chain );

SimulationStaticObject * base_link = new SimulationStaticObject();
base_link->setName( "BaseLink" );
base_link->setVisualModel( "VisualModels/torus2.obj", "", "" );
base_link->setCollisionModel(
      "CollisionModels/torus2_for_collision.obj" );
chain->addChild( base_link ); //end base_link

SimulationDeformableObject * link1 = new
      SimulationDeformableObject();
link1->setName( "Link1" );
link1->setVisualModel( "VisualModels/torus.obj", "", "" );
link1->setCollisionModel(
      "CollisionModels/torus_for_collision.obj" );
link1->setMass(5.0);
sofa::component::forcefield::TetrahedronFEMForceField<
      sofa::defaulttype::Vec3Types> * force_field = new
      sofa::component::forcefield::TetrahedronFEMForceField<
      sofa::defaulttype::Vec3Types>;
force_field->setYoungModulus(1000);
force_field->setPoissonRatio(0.3);
force_field->setComputeGlobalMatrix(false);
force_field->setMethod(0); //1 = large displacements, 0 = small.
link1->setForceField(force_field);
sofa::component::topology::MeshTopology * topo = new
      sofa::component::topology::MeshTopology;
topo->load("Topology/torus_low_res.msh");
link1->setTopology(topo);
link1->applyTranslation( sofa::defaulttype::Vec3f(-2.5, 0.0, 0.0) );
chain->addChild( link1 ); //end link1

SimulationRigidObject * link2 = new SimulationRigidObject();
link2->setName( "Link2" );
link2->setVisualModel( "VisualModels/torus2.obj", "", "" );
link2->setCollisionModel(
      "CollisionModels/torus2_for_collision.obj" );
link2->setBehavioralModel( "BehaviorModels/torus2.rigid" );
link2->applyTranslation( sofa::defaulttype::Vec3f(-5.0, 0.0, 0.0) );
chain->addChild( link2 ); //end link2
```

Compile and run the simulation.

Acceptance Criteria: The rendered scene should consist of 3 links of a chain. The first is static, the second is deformable, and the third is rigid.

C.4.5.3 Objects can be translated, rotated, and scaled

Objective: Allows the objects to be placed and oriented correctly in the scene before the simulation starts.

Prerequisites: Test Case C.4.5.2 passed.

Test Process: Create a project using the project wizard.

In the *createScene()* method, create simulation objects as in Test Case C.4.5.2.

Translate, rotate, and scale each object.

Compile and run the simulation.

Acceptance Criteria: The rendered scene should display the objects at the new position, orientation, and scale.

C.4.5.4 Objects affected by gravity

Objective: Objects in the scene should be affected by gravity unless they are static objects.

Prerequisites: Test Case C.4.5.1 passed.

Test Process: Create a project using the project wizard selecting the SofaQtRenderingEngine.

In the *createScene()* method, add the following line,

```
sim mgr->loadSceneFile("collisionTriangle.scn");
```

Compile and run the simulation.

From the menu select Simulation->Animate.

Acceptance Criteria: The scene should start animating and the box should fall due to gravity.

The floor, being static, should remain in place.

C.4.5.5 Collisions detected and correct response performed

Objective: Objects in the scene should be affected by gravity unless they are static objects.

Prerequisites: Test Case C.4.5.4 passed.

Test Process: Create a project using the project wizard selecting the SofaQtRenderingEngine.

In the *createScene()* method, add the following line,

```
sim mgr->loadSceneFile("collisionTriangle.scn");
```

Compile and run the simulation.

From the menu select Simulation->Animate.

Wait for the box to fall until it touches the floor.

Acceptance Criteria: When the box touches the floor, it should start deforming due to the collision with a rigid surface.

C.4.6 Website

C.4.6.1 Publically accessible

Objective: Site should be publically available.

Prerequisites: None.

Test Process: On an computer that is not on the UWO network, browse to the site: http://publish.uwo.ca/~thayes4

Acceptance Criteria: The web site's home page should load.

C.4.6.2 Download Software – AutoMan, VS .Net Project Wizard

Objective: Site should allow the public to download AutoMan and the VS .Net Project Wizard.

Prerequisites: Test Case C.4.6.1 passed.

Test Process: Browse to the site: http://publish.uwo.ca/~thayes4

Navigate to the Downloads page.

Select software to download.

Acceptance Criteria: Selecting AutoMan should start downloading the AutoMan framework.

Selecting VS .Net Project Wizard should start downloading the project wizard.

C.4.6.3 View User Guides

Objective: Site should allow the public to view user guides.

Prerequisites: Test Case C.4.6.1 passed.

Test Process: Browse to the site: http://publish.uwo.ca/~thayes4

Navigate to the Docs page.

Select the user guide to view.

Acceptance Criteria: Selecting any user guide from the list on this page should open the document for viewing.

C.4.6.4 View API pages

Objective: Site should allow the public to view the API.

Prerequisites: Test Case C.4.6.1 passed.

Test Process: Browse to the site: http://publish.uwo.ca/~thayes4

Navigate to the Docs page.

Select the Doxygen API.

Acceptance Criteria: The HTML version of the API should load.

Appendix D Test Results Document

Test Report

AutoMan Framework

Department of Electrical and Computer Engineering

The University of Western Ontario

London, Ontario, Canada

Revision History

Date	Version	Description	Author
3/5/2008	1.0	Initial Creation	T. Hayes

D.1 Introduction

This is a report generated from the testing phase of the AutoMan project. It contains the results of the testing performed as described in the Test Plan document.

D.2 Test Results

D.2.1 Installation Process

D.2.1.1 Installation Guide

Test Case:	4.1.1	Prerequisites:	None:
Expected Result:	AutoMan successfully installed.		
Actual Result:	AutoMan successfully installed.		
Comments:	Guide is clear and easy to understand. An average user should be able to follow it to install the framework.		
Pass/Fail:	**Pass.**		

D.2.2 VS .Net Project Wizard

D.2.2.1 Installation Guide

Test Case:	4.2.1	Prerequisites:	4.1.1
Expected Result:	Project wizard successfully installed.		
Actual Result:	Project wizard successfully installed.		
Comments:	Guide is clear and easy to understand. An average user should be able to follow it to install the project wizard.		
Pass/Fail:	**Pass.**		

D.2.2.2 Generates Project

Test Case:	4.2.2	Prerequisites:	4.2.1
Expected Result:	Visual Studio Project should be created. TestCase.h header file should be present. TestCase.cpp file should be present. Project should compile successfully.		
Actual Result:	Project created. TestCase header and code files created. Project successfully compiled.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.2.3 Frame Listener included [/ not included]

Test Case:	4.2.3	Prerequisites:	4.2.2
Expected Result:	Visual Studio Project should be created. RenderingEngineFrameListener header should [/ should not] be included. TestCass class should [/ should not] extend RenderingEngineFrameListener Project should compile successfully.		
Actual Result:	When option is selected the project is created, header is included, class is extended and the project compiles When option is not selected the project is created, include is not present, no class extension, and project compiles.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.2.4 VRJugglerRenderingEngine correctly included [/not included]

Test Case:	4.2.4	Prerequisites:	4.2.2
Expected Result:	Visual Studio Project should be created. VRJugglerRenderingEngine header should [/ should not] be included. TestCass::constructRenderingManager() method should [/ should not] create an instance of VRJugglerRenderingEngine and attach it to the RenderingManager Project should compile successfully.		
Actual Result:	When option is selected the project is created, include is present, RenderingEngine is created and attached, and the project compiles. When the option is not selected, the project is created, the include is not present, no RenderingEngine is created and the project compiles.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.2.5 SofaQtRenderingEngine correctly included [/ not included]

Test Case:	4.2.5	Prerequisites:	4.2.2
Expected Result:	Visual Studio Project should be created. SofaQtRenderingEngine header should [/ should not] be included. TestCass::constructRenderingManager() method should [/ should not] create an instance of SofaQtRenderingEngine and attach it to the RenderingManager Project should compile successfully.		

Actual Result:	When option is selected the project is created, include is present, RenderingEngine is created and attached, and the project compiles. When the option is not selected, the project is created, the include is not present, no RenderingEngine is created and the project compiles.
Comments:	Works as expected.
Pass/Fail:	**Pass.**

D.2.3 AutoMan – Application Module

D.2.3.1 Initialize and start the simulation

Test Case:	4.3.1	Prerequisites:	4.2.2
Expected Result:	The simulation should start running.		
Actual Result:	Simulation starts running.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.4 AutoMan – Rendering Module

D.2.4.1 Attach a RenderingEngine to the RenderingManager

Test Case:	4.4.1	Prerequisites:	4.2.2
Expected Result:	A VRJuggler window should appear and render the scene.		
Actual Result:	Window displays the scene.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.4.2 Handle multiple RenderingEngines

Test Case:	4.4.2	Prerequisites:	4.4.1
Expected Result:	A VRJuggler window should appear and render the scene. The console should print "Dummy Rendering Engine – update method called."		
Actual Result:	Window displays the scene. Message printed to console.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.4.3 RenderingEngineFrameListener makes callbacks

Test Case:	4.4.3	Prerequisites:	4.4.1

Expected Result:	Console should print:
	`Frame number 0 started…`
	`Frame number 0 ended.`
	`Frame number 1 started…`
	`Frame number 1 ended.`
	`Frame number 2 started…`
	`Frame number 2 ended.`
	`...`		

Actual Result:	`Frame number 0 started…`
	`Frame number 0 ended.`
	`Frame number 1 started…`
	`Frame number 1 ended.`
	`Frame number 2 started…`
	`Frame number 2 ended.`
	`...`		

Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.4.4 Handle multiple RenderingEngineFrameListeners

Test Case:	4.4.4	Prerequisites:	4.4.3

Expected Result:	Console should print:
	`Frame number 0 started…`
	`TestFrameListener::frameStarted()called…`
	`Frame number 0 ended.`
	`TestFrameListener::frameEnded()called.`
	`Frame number 1 started…`
	`TestFrameListener::frameStarted()called…`
	`Frame number 1 ended.`
	`TestFrameListener::frameEnded()called.`
	`...`

Actual Result:	`Frame number 0 started…`
	`TestFrameListener::frameStarted()called…`
	`Frame number 0 ended.`
	`TestFrameListener::frameEnded()called.`
	`Frame number 1 started…`
	`TestFrameListener::frameStarted()called…`
	`Frame number 1 ended.`
	`TestFrameListener::frameEnded()called.`
	`...`

Comments:	Works as expected.
Pass/Fail:	**Pass.**

D.2.5 AutoMan – Simulation Module

D.2.5.1 Load scene file

Test Case:	4.5.1	Prerequisites:	4.4.1
Expected Result:	The collisionTriangle simulation should run.		
Actual Result:	The collisionTriangle simulation runs.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.5.2 Create simulation objects

Test Case:	4.5.2	Prerequisites:	4.4.1
Expected Result:	The rendered scene should consist of 3 links of a chain. The first is static, the second is deformable, and the third is rigid.		

Actual Result:	All 3 links are present.
Comments:	Works as expected.
Pass/Fail:	**Pass.**

D.2.5.3 Objects can be translated, rotated, and scaled

Test Case:	4.5.3	Prerequisites:	4.5.2
Expected Result:	The rendered scene should display the objects at the new position, orientation, and scale.		
Actual Result:	Objects are at the new positions.		
Comments:	Rotation does not work. SOFA doesn't support it yet. The methods are not publically accessible. SOFA site implies this will change soon.		
Pass/Fail:	***Fail.**		

D.2.5.4 Objects affected by gravity

Test Case:	4.5.4	Prerequisites:	4.5.1
Expected Result:	The scene should start animating and the box should fall due to gravity. The floor, being static, should remain in place.		
Actual Result:	The box falls and the floor remains in place.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.5.5 Collisions detected and correct response performed

Test Case:	4.5.5	Prerequisites:	4.5.4

Expected Result:	When the box touches the floor, it should start deforming due to the collision with a rigid surface.
Actual Result:	The box deforms on contact with the floor.
Comments:	Works as expected.
Pass/Fail:	**Pass.**

D.2.6 Website

D.2.6.1 Publically accessible

Test Case:	4.6.1	Prerequisites:	None.
Expected Result:	The web site's home page should load.		
Actual Result:	The web site's home page loads.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.6.2 Download software – AutoMan, VS .Net Project Wizard

Test Case:	4.6.2	Prerequisites:	4.6.1
Expected Result:	Selecting AutoMan should start downloading the AutoMan framework. Selecting VS .Net Project Wizard should start downloading the project wizard.		
Actual Result:	Both project archives can be downloaded.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.6.3 View User Guides

Test Case:	4.6.3	Prerequisites:	4.6.1
Expected Result:	Selecting any user guide from the list on this page should open the document for viewing.		
Actual Result:	Users guides can be opened for viewing.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		

D.2.6.4 View API pages

Test Case:	4.6.4	Prerequisites:	4.6.1
Expected Result:	The HTML version of the API should load.		
Actual Result:	API is present and available.		
Comments:	Works as expected.		
Pass/Fail:	**Pass.**		