ANIMATED VISUALISATION IN A COMPONENT-BASED SIMULATION ENVIRONMENT

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

November 2000

By
M. Nedim Alpdemir
Department of Computer Science

Contents

List of Figures

Abstract

0.1 Abstract

The animated visualization of virtual prototypes together with a virtual environment they are anticipated to be used in, is most likely to require the integration of discrete event simulation packages, behaviour simulation engines and visualization packages possibly through a network if distributed processing is desirable. This, clearly, can involve many problems such as, integration of and synchronization between discrete and continuous simulators, maintaining an acceptable frame rate for real time animation with a reasonable level of detail and rendering quality, designing an effective user interface for the manipulation of and interaction with the virtual environment, reducing the network latency for distributed simulation environments etc., each of which deserves a separate research effort. A fundamental problem, however, in systems involving such level of complexity is that of finding a well-defined, structured way of building software systems making them operational.

In this thesis a software engineering point of view is taken to this set of potential problems, highlighting the problem of constructing virtual prototyping systems in an easy and intuitive way which ensures maintainability and re-usability. Component Oriented Programming approach, a recent development in software engineering discipline, is proposed as a candidate solution leading to the concept of component-based simulation environments, and a particular area, the interaction between a behaviour simulator component and an animator component, is taken as a case study. A new animation component, AniComp, is designed and implemented as an example tool as well as an illustration of the ideas that has developed during this research, intending to make a contribution towards the realization of component-based simulation software construction. The usage of AniComp is exemplified in MISAF (MInimal Simulation Animation Framework);

a new component framework that aims to provide a loose yet flexible mechanism for regulating (simulator-animator) component interactions in a simulation environment. Finally AniComp and MISAF are used to build an example application, the Virtual Lab Application, for dynamically creating virtual laboratories containing virtual instruments with their behaviours implemented as simulator components, in a web enabled environment. Two instruments, namely the Ball and Beam and Inverted Pendulum, are modeled both geometrically and functionally as examples for building control engineering labs.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

Anything good in this work is by the grace of God, our cherisher and sustainer, to whom is due all praise.

The author is very grateful to his supervisor Dr. Richard N. Zobel for his support, frendliness, encouragement and patience. He is also grateful to Kirikkale University of Turkey who supported his education financially. He directs special thanks to Dr. Hüsamettin Baki and Dr. Bülent Küçük for their extensive support, and valuable comments. He appreciates the kindness and friendly environment provided by the other members of the Synthetic Environments Group, especially Dr. Jafar Habibi and Pichaya Tandayya.

The last, but not least, he deeply appreciates the extensive encouragement, understanding and patience of his mother, Hafize, his wife, Rabia, and lovely children, Sükran, Ibrahim and Yusuf. They sacrificed a lot to allow this work to go on.

Chapter 1

Introduction

1.1 Simulation, Animation and Virtual Prototyping

Generating virtual models of objects before they exist is desirable. Simulating the functionality of these objects, making them behave as if they really existed, is even more desirable and promising, since it enables us to use these models for purposes ranging from marketing presentations to detailed physical or operational analysis.

Creating 2D digital representations of products prior to their manufacturing has been possible by the use of CAD (Computer Aided Design) software packages for more than two decades. However utilizing these representations in creating functional virtual prototypes by using a simulation engine to animate the geometric model, is a more recent development. Although *computer animation* has been around from 1960's on, thanks to the Sketchpad system designed by Sutherland [Mye96, Sut63], it has mainly found its usage in the entertainment and the advertisement industry where artistic impression can be more important than physical realism. Hence, the majority of the traditional animation packages do not address the needs of software constructors who want to implement animated representations of digital products, where animated behaviour of the product should be generated by a simulation model. This must be one of the reasons why a separate thread of research and a branch of software development industry has been formed, namely *virtual prototyping* [Tse98, Ben98, Pot99]. The animated visualization of virtual prototypes together with the virtual environments they are anticipated to be used in, is most likely to require the integration of discrete event simulation packages, behaviour simulation engines and visualization packages possibly through a network if distributed processing is desirable. This, clearly, can involve many problems such as, integration of and synchronization between discrete and continuous simulators, maintaining an acceptable frame rate for real-time animation with a reasonable level of detail and rendering quality, designing an effective user interface for the manipulation of and interaction with the virtual environment, reducing the network latency for distributed simulation environments etc. Each of these challenges deserve a separate research effort. A fundamental problem, however, in systems involving such levels of complexity, is that of establishing a well-defined, structured way of building and integrating their software components to facilitate their construction and maintenance.

In this thesis, a software engineering point of view is taken to this set of potential problems, highlighting the problem of constructing virtual prototyping systems in an easy and intuitive way which ensures maintainability and re-usability. The **Component Oriented Programming** approach, a recent development in the software engineering discipline, is proposed as a candidate solution leading to the concept of component-based simulation environments. Within component-based simulation, a particular area, the interaction between a behaviour simulator component and an animator[1]component, is taken as a case study. A new *Animator Component* is designed and implemented as an example tool as well as an illustration of the ideas that have developed during this research, intending to make a contribution towards the realization of component-based simulation software construction.

A more detailed description of the specific issues underlying the motivation for this thesis is given in the following section. This is followed by an introduction to the component-oriented approach to software construction and then by a section establishing the objective of this research. The chapter concludes with a section on how this thesis is organized.

[1]The word *animator* is mostly used when referring to a human animator who creates the animation sequences by interactive editing. Here, however, it refers to the piece of software that creates the animations. This meaning implies the algorithmic (or procedural) generation of motion suggesting a conceptual shift, which, I believe, is also consistent with the general context of this thesis where simulation engines (i.e. software modules, not a human) are perceived to be the primary behaviour generators. The word is used with this meaning throughout the thesis. It is prefixed with the word human where needed.

1.2 Motivation

The primary intent of this research has been to investigate and improve the ways of enabling simulators (perceived as motion generators) to render the behaviour of their corresponding physical objects in a 3D virtual environment, by adhering to the component-based software development principles. During the initial phase of the research some experience was gained with a visualization tool called VISTA[CAS95, Tan98, Hab98], a system capable of receiving position and orientation updates from external simulators through a run-time interface. Based on this experience and literature survey, two alternative paths had emerged. 1 - Modifying an existing visualization tool and wrapping it with an interface conforming to one of the existing component wiring standards[2] (eg.COM [Wil94b], CORBA [Vin97], JavaBeans [Fla97]) 2 - Designing and implementing a new animation and visualization component. For reasons that are closely related to the properties and capabilities of the existing visualization software, the second option has been followed [Alp99]. These characteristics and capabilites are listed below:

- Conventional animation systems focus on interactive editing techniques for creating animations. These techniques typically require intensive labour of the human animator and, in fact, they are not necessary at all where the motion can be generated procedurally by a simulation engine. Although more recent systems enable users to attach dynamic properties to objects to simulate their motion under user-defined physical forces, these are too general and simple for the simulation of most engineering systems. Mechanisms provided for integration with the external simulation engines for more complex cases are poor, since the main intent is a self-contained framework for interactive editing of animation sequences[3].

- The existing virtual prototyping systems, on the other hand, enable the users to create advanced simulations of the engineering systems. However they impose a tight framework for the definition of the simulation

[2]Component wiring standards provide (and impose) rules and guidelines for supporting composition of components by providing mechanisms for introspection, event handling, dynamic linking etc. This issue and the properties of several component wiring standards are discussed in more detail in Chapter 2.

[3]Features of a typical commercial mid-range geometric modeling and animation system, TrueSpace, can be found at the web pages of the Caligari Inc. [Tru00]. The author has generated some of the VRML models used for the test cases in this thesis using this package

models/engines, which prevents effective re-use of other simulation models/engines that might have been defined previously, using another (and may be a more convenient) modeling and simulation tool. This is most likely to be an inhibiting factor in incorporating them into a heterogeneous simulation environment, which may typically require the integration and simultaneous operation of multiple simulation programs of different timing and event properties.

- There are visualization tools which provide interfaces to external simulators but these interfaces are usually API's operating at relatively low levels and they demand a considerable knowledge of the underlying graphics system[4].

- The majority of the recent simulation packages involve animation modules (or toolboxes) which enable the user to generate animated visualization of the objects and processes simulated. However, since these modules are extensions to the core simulation software they are tightly coupled with the simulation code (and hence with a specific simulation package) and sometimes limited in capabilities.

Similar points of criticism have been raised by other researchers [Bal98, Nub98], and there are ongoing research activities exploring the ways of producing better systems.

In addition to the above list, one problem which is most emphasized in this work and thus constitutes one of the essential parts of this thesis is the lack of a well-defined framework for composing different components of a simulation environment at binary level, whether the component be a functionality simulator based on a mathematical model or a visualization component, or a random number generator for a discrete event simulator. The vision adopted in this thesis is that the manufacturers specialized in the production of physical components could also produce the simulators for these physical components by conforming to one of the well-established component wiring standards, and the software constructors who wish to build simulators of more complex products could use these component simulators off-the-shelf, by plugging them in together; just as the complex physical products are manufactured in the real world. Of course this is the ideal case and many problems such as the specification of re-usable

[4]VISTA, for example, provided a limited set of functions for continuous simulation engines. To extend its functionality was very difficult because of its highly monolithic architecture

interfaces for simulators, resolving interface mismatches, the need for adaptation of the interfaces with incompatibilities etc [Kuc98] are to be faced. The next section presents a brief introduction to component oriented software construction to familiarize the reader with the concepts involved.

1.3 Component-Oriented Software Construction

It is a common practice to describe, model and build complex systems in a modular and hierarchical way. A system usually involves some subsystems which may themselves be composed of lower level subsystems, and so on. Modularity has always been indispensable in software engineering, too, leading to the development of various methods for defining building blocks for more complex software systems. The apparent advantage of structuring a system into modules is division of the computation (and complexity) into intellectually manageable (comprehensible) pieces. Development and maintenance also become easier, because changes can be made locally without having unknown effects on the whole system as long as the interface to the module remains intact; errors have a more chance of being confined to a module; and separate analysis of modules is possible. Modular software has a higher degree of scalability and adaptability than that of monolithic software: A modular application can be extended by adding new modules, and adapted by replacing appropriate ones, or removing unneeded ones. A further significant opportunity is to be able to distribute the modules among various engineers to allow parallel development and maintenance.

Component-oriented (or component-based) programming [Szy98a, Nie95] builds on the modular programming approach and investigates the principles and techniques to enhance it by separating the development and deployment of software modules. The usual case in modular software development is that a typical product consists mainly of tailor-made modules, developed from scratch and bound together by the same group of people. These modules, particularly the domain-specific ones, are hardly ever re-used in other projects [Sam97]. On the contrary, a component- oriented product, in theory, consists of ready-made components, and is only assembled (or composed) by application developers. Indeed, the fact that many software systems contain parts with similar or even identical behaviour makes it possible to consider development of large software systems by assembling pre-fabricated components. The components can be produced in-house or can constitute independently-made off-the-shelf products. Ideally, if any required components are not available, they are built out of lower level components. Finally, when even low-level components are not available, they are implemented in some programming language. Thus, coding is largely replaced by composition. Software development by re-using existing components and/or using off-the-shelf components has a positive impact on software quality and productivity [Sam97]. Reliability and performance is increased using well-tested and optimised components, developed by experts in the associated domain. Much of the details of computation is embedded in components which reduces the size of code and documentation that has to be written, and the amount of expertise to be possessed by the application assembler. These make analysis, design and implementation less laborious and less error prone than doing everything from scratch. Maintenance also becomes easier, as fewer defects can be expected to occur when proven components are in use and less code is being maintained. Of course, these benefits mentioned are not obtained without effort and investment. Component-oriented software development requires having a wide variety of high-quality components, proper classification and retrieval mechanisms, sufficient and proper documentation of components, and a flexible means for combining components.

1.4 Objective and the Scope of the Research

1.4.1 The objective

Aiming to bring the benefits of component oriented software engineering to simulation software construction, the objective of the research is set as implementing an example animator component based on the investigation of possible patterns of component interactions in a simulation environment; particularly the interactions between a visualization component and its clients. By doing this it is hoped that this work will contribute to the realization of digital objects [Fis98] providing a visibility interface, pluggable to the functional representations of those objects.

A possible configuration of a component-based simulation environment is depicted in Figure 1.1. The diagram illustrates the simulation components accessing each others' interfaces and notifying their clients when certain events are triggered. The illustration does not cover all the components that could exist in a simulation environment. Components that might be responsible for simulation model database access or for the management of communication across the network are not included since these topics are not central to this research.

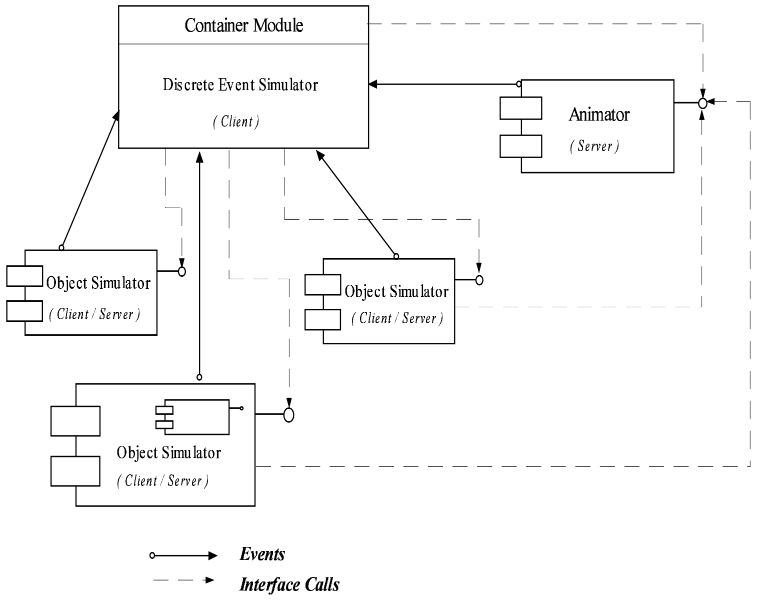


Figure 1.1: The Proposed Component Based Simulation Environment

1.4.2 Issues Researched

There are three main set of questions that are attempted to be answered:

- What could be the modes of interaction between a simulator component and a visualization component? Is it possible to design an animator component with a sufficiently flexible interface, to address the needs of different interaction modes?
- How black-box simulator components can make themselves visible (present) in the same virtual environment? How are they notified of events that are relevant to them? What type of mechanisms are required for managing the virtual environment that contains a number of simulated objects?
- To what extent is it possible to decouple the visualization module from the simulator module to be able to hide the complexity of the underlying graphics system and present a high level interface to the clients of the

animator component. Consequently, how to approach to the challenge of increasing the re-usability of the Animator component.

In the following sections these issues are explained in more detail and the approaches adopted to answer the questions are introduced.

The Nature of Interactions Between a Behaviour Simulator and the Animator

An interaction occurs when the simulator attempts to modify the static or dynamic state of a geometric object in the virtual environment, and conversely, when the virtual environment responds to these state changes by triggering events. In this research work, the classification of the interactions is made from the inherent structural complexity point of view, ranging from *atomic*, causing a single, simple change in the state, to *aggregate*, causing an array of atomic state changes with a time dimension attached to them. One particular concern of the research regarding the interactions has been towards determining the pattern of changes in the nature of the interactions during the lifetime of a certain simulation exercise. This has lead to the objective of discovering whether certain types of simulation models tend to produce only certain modes of interaction consistently; or whether random mode changes appears to be a common pattern within the same modeling paradigm, suggesting a weak correlation between the type of the simulation and the resulting interactions.

Note that the word interaction is used in the context of *component interactions* referring to the message interchanges which arise during interface invocation. There are other paradigms in software engineering which place slightly different meaning to interaction. Interaction-based programming, a term used by Lee [Lee98], for instance, defines an interaction as a separate construct and organizes a program into components and interactions. This approach, however, mandates a run-time evaluator, in addition to a standard programming language compiler, to analyze and verify the program's interactions, which inevitably incurs a certain run-time overhead and would probably require support during the software development phase. Other researchers [Nie95] use the term component interaction when referring to low-level communication patterns which enable the components to be compatible to each other and make them cooperate and integrate seamlessly by adhering to certain protocols. In this thesis, this term is used to underline the component interactions which take place at a higher operational level and which directly exploit the computational content of the interacting parties.

Managing the 3D World

The existence of multiple active (i.e. moving) objects in a Virtual Environment (VE) necessitates the employment of a number of mechanisms to handle situations such as object loading, object identification and manipulation, collision detection and collision response, view (camera) management and user interaction. The term *virtual environment* usually implies a user interaction metaphor based on the immersion of the user into the computer generated environment through the use of special hardware such as a head mounted display, head or body tracking devices, data gloves and 3D sound systems [Fur95]. Although the Animator component implemented during this study, presents a 3D virtual environment, the emphasis is on generating the behaviour of the objects which populate the environment, rather than the type of user interaction. Acknowledging the fact that user interaction through immersive interfaces is considered to be important [Bry95, Gig93], the Animator component's environment should be classified as a non-immersive VE relying on the classic WIMP (Windows, Icons, Mouse and Pointer) user interaction paradigm.

The Animator component employs a number of internal classes that undertake the task of managing the 3D environment. The main classes are `CAnimate`, `CSimEnvironment`, `CCollisionManager`, `CViewManager`, `CRigidBody` and `CBehaviour`. A detailed discussion of these classes; their roles and responsibilities and their interrelations are presented in Chapter 5.

To populate the virtual environment, object simulators should keep a generic pointer to the environment they are expected to render their existence in; a pointer which should be passed to the simulator by the container application during the loading and initialization phase. The approach taken to handle these requirements has been to implement a component framework which imposes a number of rules regulated by a weak agreement. The agreement is realized in the form of an *interface contract* and enforced by the framework to ensure valid interactions between the simulator components and the animator component. The details of the mechanisms involved are discussed in Chapter 6.

Reducing the Coupling Between the Simulation Module and the Graphics System

The notion of coupling, borrowed from structured design approach, is defined as "the measure of the strength of association established by a connection from one module to another" [Boo94]. An interesting counter example to good coupling is mentioned in Booch's book on Object Oriented Analysis an Design [Boo94] describing a modular stereo system in which the power supply is located in one of the speaker cabinets.

The component based development approach implies and promotes weak coupling by its very nature, as mentioned earlier. Furthermore, a broad examination of the characteristics of a typical simulation engine and a 3D visualization system reveals a clear conceptual distinction between the two; a natural basis for decoupling. Simulation engines implement symbolic calculations based on a continuous system (mathematical) model or on a discrete event model, whereas a visualization and animation system manipulates geometric and color data to change the position, orientation and appearance of the geometric objects. It seems natural to think of a situation where the data produced by the simulators are simply translated into the visualization system and cause the 3D objects to change their states. This conceptual distinction, however, tends to blur in practice, since a simulator based on a physical model, for example, generates position and orientation information which can directly modify the state of the geometric objects. Moreover the interaction of an object with other objects (or with the user) in the virtual environment may require its behaviour simulator to immediately access low level geometric data such as 3D vertex information, to calculate a realistic response; both of which suggest tight coupling. This creates a tension in deciding whether to tightly couple the behaviour simulators to the graphics system endangering re-usability and ease of development, or to decouple them at the expense of less flexibility on the simulators side.

The approach adopted in this research work is to incorporate the tasks requiring tight coupling into the graphics module (i.e. the animator component) as much as possible. To achieve this, a simple rigid body simulator and mechanisms for creating and re-using pre-defined motion sequences are incorporated into the animator component. On the other hand, the animator component's interface is based on name based invocation and ensures that the internal details of the underlying graphics system remains hidden to the simulation programmer. Both of these decisions lead to weak coupling. However, to support the needs of the simulators that require frequent interaction with the graphics system, a set of low level functions have been provided to facilitate atomic interactions.

1.4.3 The Scope

The following points have been effective in defining the boundaries of this research work:

- The Animator component does not support a multi-user distributed virtual environment. Although the underlying component wiring technology used in the implementation (COM/DCOM) enables remote access to the component's interface, and some of the functionality provided in the Animator's interface such as dead reckoning and re-usable behaviours, promise an improved performance in a distributed environment, distributed simulation is not specifically addressed in this research. Distributed computation requires carefully designed, robust mechanisms to handle multi-user/multi-process interaction across the network, which should, in itself, constitute a separate research effort. This, therefore, is left as a future enhancement to the component.

- The animation techniques employed in the research are restricted to rigid-body animation. The behaviour of deformable bodies and animation types such as liquid, cloth, fire or smoke animation, which require significantly different approaches to geometric (and animation) modeling from that of rigid-bodies, are left out of the scope.

- The user interaction with the 3D environment is only of limited interest in this study. Advanced interaction interfaces and techniques that are usual to find in many 3D virtual worlds, such as head mounted display, head and body tracking devices, data gloves etc., are not supported by the Animator component. The reason for this is mainly an accessibility problem: Advanced interaction interfaces require sophisticated (and expensive) hardware and software; a resource which wasn't available to the author.

- Although a considerable amount of this research effort has been devoted to the fields of simulation and computer animation, this study does not aim to produce a better or an improved simulation model of a certain system nor does it intend to introduce a novel animation or rendering technique. The primary aim is to apply a recent approach in *software engineering*, namely the component-based software development approach, to simulation software domain as a broader field, and in particular, to the visualization of digital objects' behaviours that are generated and coordinated by simulation models. Therefore the main focus of the research has been on the analysis of the simulation and the animation (and visualization) domains, seeking to extract the necessary knowledge for implementing a re-usable visualization component for component-based simulation environments.

1.5 Organization of the Thesis

The rest of this thesis can be divided into two main parts. The first part, consisting of Chapters 2,3 and 4, builds towards an understanding of the issues concerning the *animation in the context of component-based simulation*, and the second part, which includes Chapters 5 and 6, presents the design, implementation and the testing of a novel animation and visualization component.

The next chapter, Chapter 2, presents basic structural and functional features of software components. It also sketches an architectural view of the component wiring platforms, followed by a study on the meaning of the component interactions. Chapter 2, thus, aims to lay the foundation for the component-based development effort that is to appear later in the thesis.

Chapter 3, provides an introductory discussion of animation techniques and animation systems. The chapter classifies the animation techniques in such a way that the differences emanating from the method of creating animation sequences are highlighted. This emphasis aims to leverage the understanding of the animations generated by *procedures* (i.e. simulation engines) as opposed to interactive editing. This, in turn, provides a basis for identifying the requirements for an animation component capable of serving external simulators.

Chapter 4 attempts to sketch a picture of how the simulation software packages work and what do they produce as output, that can be related to the creation of the visual behaviour of the simulated entities. To this end, the chapter presents a study on the effect of different simulation modeling approaches to the simulation-animation interaction patterns and suggests a top level model for the interaction patterns stemming from the outputs of the simulation components.

Chapter 5 presents the design and implementation of an animation and visualization component, called AniComp. The chapter, first, introduces the adopted design methodology and the tools used to support this methodology. Then the interface design and the internal design of AniComp is explained. The chapter ends with the discussion of some implementational issues and an example implementation.

Chapter 6 introduces a light-weight component framework for regulating the component interactions in a component based simulation environment where AniComp is used as a visualization component. The chapter discusses advantages and disadvantages of the framework's design and demonstrates its usage through the implementation of an application for creating virtual laboratories. The chapter, also, presents the simulation models of two virtual instruments commonly used in control engineering laboratories, namely, the Ball and Beam and the Inverted Pendulum.

Chapter 7 presents the conclusions reached during this research work and lists the contributions made as a result of the work. It also offers some suggestions for further work.

Finally, five appendices are provided at the end of the thesis. Appendix A is a notation guide to the UML (Unified Modeling Language) modeling elements used in the thesis. Appendix B is a concise reference manual of AniComp, which lists the functions in its interface and provides brief explanations on how to use them. Appendix C contains the definitions of the C++ classes employed in the internal design of AniComp. Similarly, Appendix D contains the class definitions of the component framework implemented to test AniComp, as well as the class definitions of virtual laboratory application. The last appendix, Appendix E presents the publications produced during this research work.

Chapter 2

Component-Based Software Development

2.1 Introduction

The benefits of using Component Based Software Development (CBSD or CBD) have been briefly mentioned in the introduction chapter. It is interesting to observe that the ideal vision of component assembly as a way of software construction is becoming more and more popular[1]. Many experts from the software industry are stating that the aim of turning software development into something resembling an authentic engineering discipline (Software Engineering) seems to be nearer to becoming reality. [Mac99, Mey99, All99, Szy99]. Mactaggart[Mac99] argues that although underlying technical advances are an important cause for this trend, the shift is essentially business driven: New commercial models require organisations to understand and to react to global influences. The need for flexibility and rapid response to market conditions, and increasingly important issue of reliability, are the factors driving this particular change. Figure2.1 illustrates this new focus from a top level point of view. CBD, however, as this thesis provides an example, will inevitably have implications for any domain in which software is an important asset.

In this chapter a more detailed, more technical discussion of the concepts underlying the component based approach will be presented. Following a definition

[1]In recent months Component Based Development was the cover story of two popular journals: May 1999 issue of Software Development [Mey99], and Sept/Oct issue of Application Development Advisor [Mac99]

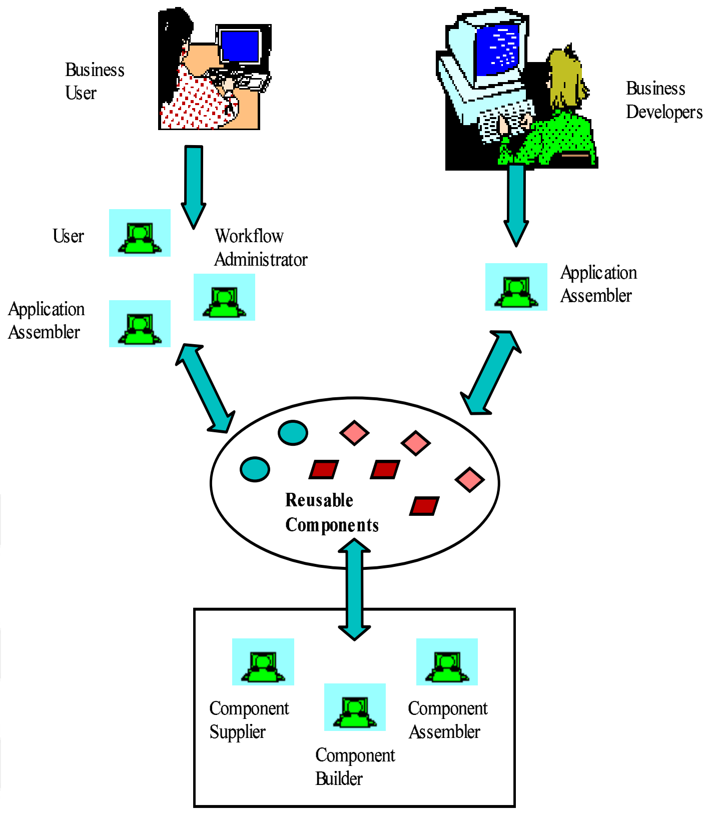


Figure 2.1: The Envisioned Scenario for Component Based Development

of the component, basic structural and functional features of the components will be discussed . Next, the existing approaches to implement component wiring standards will be introduced including their architectural view using software patterns. Following that, a discussion on how to study component interactions is presented as a prelude to Chapter 4 which discusses the interaction patterns between a simulator and an animator component in more detail. Finally another discussion is given on the issue of re-usability and extensibility of the components.

2.2 The Definition of a Component

There is not an agreed upon, universal definition of a component since different researchers from both academia and software industry, tend to take different perspectives and emphasize different properties.[2]. However there are also some

[2]For a discussion of what a component means between nine researchers see[Bro98]

common points in the proposed definitions such as the existence of well defined interfaces, being subject to composition, being able to import/export functionality via its interfaces in a well defined way. A definition developed in the first Workshop on Component-Oriented Programming is quoted here as a compact example which states that "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."[Szy97]. An even more compact definition comes from Nierstrasz and Dami stating that a component is *a static abstraction with plugs*[Nie95]. They continue to explain that "static" means that a software component is a long-lived entity that can be stored in a software base, independently of the applications in which it has been used. By "abstraction", they mean that a component puts a more or less opaque boundary around the software it encapsulates. And they finally explain that "with plugs" mean that there are well-defined ways to interact and communicate with the component(parameters, ports, messages etc.). This definition seems to capture the most important properties of a component as it is envisioned in the context of a more revolutionary usage of the term "component-based software development" as opposed to a more conventional context where software components are seen as any artefacts distinctly identifiable in a software system including source code fragments and documentation.

A software entity complying with the properties identified in the above definitions is technically a component, although its widespread acceptance demands further qualities. The success of a software module as a component is affected largely by the degree of separation between its development and usage, and the degree of its re-usability[Szy98a]. In the following sections, basic features of components, which contribute to achieving these fundamental measures of quality are explained. Using the features identified as criteria, it becomes possible to assess how suitable a particular software module is to be used as a component; that is, how easy it is to develop it as a separate system and to use it in various contexts.

2.3 Basic Features of Software Components

Basic features of the components can be classified into two broad categories: (1) Structural features (2) Functional features. *Well-defined interfaces*, *information hiding* and *effective interconnetion mechanisms* are the structural features, whereas *little context dependence*, *loose coupling* and *high cohesion* are functional features. The following sections examines these features in more details.

2.3.1 Structural Features

Well-Defined Interfaces

Components, by definition, are things that are connected together to form a system; and interfaces are the means by which components connect. A component that implements an interface constitutes a provider for the interface; and a component that uses (or calls) an interface constitutes a client. In other words, through an interface a component (provider) offers a service to another component (client). A component may implement multiple interfaces, each offering different services. Also, a particular type of interface may be supported by a number of different providers. Since components are developed independently, their successful interoperation demands a careful, rigorous definition of the interfaces they provide. An interface definition constitutes a contract between a client of the interface and a provider of its implementation. The contract states what the client needs to do, to use the interface. It also states what the provider has to implement to meet the services promised by the interface[3].

A typical interface consists of a set of operations. A complete definition for such an interface should include the syntax and semantics of the operations. A formal specification of the semantics includes the preconditions i.e. conditions which must exist for the operation to execute correctly, and postconditions i.e. conditions which will exist once the operation has executed correctly. Usually a simplified interface definition is used which ignores a formal specification of the semantics. A programming language or a special interface definition language (IDL) is used for syntactical (and informal textual) descriptions of operations. Each operation is described by a name, a signature and possibly a return type. The signature states the order, types and passing modes of parameters. The corresponding (simplified) contract demands that the clients invoke the interface operations with correct names, applying correct type of values or variables to the input (and in-out) parameters; and it also demands that the providers' implementations of the operations return correct types of values through the output

[3]Viewing interfaces as contracts may be useful in modeling the interactions between the components as well. This issue is further explored in Section 2.5.1.

(and in-out) parameters. The specification of a component usually involves a set of interfaces that it implements. The specification may also involve a set of interfaces that the component calls, without stating the destination of the call (i.e. the service provider), which will be determined at run-time. The former group are called the incoming interfaces of the component and the latter are called its outgoing interfaces[Szy98a]. Outgoing interfaces are particularly useful in event-based programming[Sha96]. The way the interfaces are defined, used and implemented depends on the types of the associated components, and the underlying computational platform(s).

Hidden Implementation Information

Usage of components only through their interfaces is an essential principle for separation of their development and deployment. It suggests that the interfaces associated with a component contain the information sufficient for its utilization; and other information (procedures and data) contained within the component are not needed by other components, therefore can (and should) be made inaccessible. Information hiding prevents the clients from manipulating the component's internal mechanism, which is very error-prone. Furthermore, the clients are kept independent of implementation details which are subject to modification by the developers, during the component's lifetime and maintenance. Components are classified according to the degree of information hiding they adopt. A *black-box* component is used without seeing, knowing and modifying any of its implementation aspects. The users get the information about what the component does, but not about how it is achieved. Therefore, the implementation can be changed without any effect on the users. An efficient way of encouraging black-box usage of components is compiling them individually, and offering them in binary form. This is a safe approach, as anything that is not hidden can potentially be used by the clients, and thus becomes part of the component's interface even if it is not specified explicitly in the interface definition. Binary components also possess the advantage of being ready to use, i.e. without compilation by users; and they protect the proprietary rights of providers, by preventing distribution of commercially sensitive source code. *White-box* components make the implementation fully visible to users, i.e. by providing the source code. The main point of this is to enable extension (or modification) of the component's capabilities by re-using the available code as much as possible. That is, white-box components are typically not re-used as they are, but by adaptation. The mechanisms of implementation inheritance and parameterized classes, widely used in object-oriented programming, are particularly convenient for this purpose. Being able to access the internal data makes extension easier, unfortunately at the expense of the benefits of information hiding. Consequently, small-size components with a high chance of user extension or adaptation are suitable candidates for white-box implementation. If only part of a component's implementation is disclosed for inspection and modification, the term *grey-box* re-use is also used[Bc97, Bc99].

Effective Interconnection Mechanism

The interconnection mechanisms, at least, facilitate exchange of information messages between components, as well as facilitating synchronization between active components running a parallel thread of execution. In more sophisticated cases, interconnection mechanisms are able to connect not only the components within a shared address space, but also the components located in separate address spaces, running as individual processes. There are significant benefits of address space separation: (1) Avoiding a shared address space makes it much easier to prevent components from damaging one-another. (2) Combined with a language-independent medium for communications, separation of address space also enables integrating components implemented in different programming languages, making them more re-usable. (3) They can even be running on different (incompatible) computers if the connectivity software supports distributed computing. Besides being built for an incompatible computer, remote operation of a component is also convenient when it demands substantial computational resources; and of course, the performance will improve if the distributed components can be made to run concurrently. Unfortunately, address space separation has a negative side: the high cost of communications. Cross-process message exchange is much more time-consuming than in-process message exchange, and cross-machine communications is even slower. Consequently, a component-oriented system cannot always rule out in-process (local) components in favor of out-of-process (remote) ones. Since being local or remote is actually an implementation issue, the connection mechanisms usually provide a technique for hiding this feature from other components as much as possible. These types of interconnection mechanisms are further explored in Section 2.4.4 during the discussion of broker and proxy patterns employed by existing component wiring standards.

Another interconnection mechanism is formed when a technique called late binding is applied. Modern programming paradigms require a dynamical (or late) binding of senders and receivers of messages. One important application area of dynamic binding is object-oriented programming. Another important application area is event propagation. In event-based programming, components push information (through an outgoing interface) to the recipients as it arises, rather than the more common case where a client pulls information from a server or provider as needed. This is called firing an event. There can be multiple targets for a single event fired, and the source of the event cannot statically know the interested targets. A structural pattern called observer (or publisher/subscriber) facilitates the implementation of event-based programming and is included in existing component standards. This issue is discussed in more detail in Section 2.4.4 during the discussion of the observer pattern.

2.3.2 Basic Functional Features of Software Components

Little Context Dependence

A component is dependent on another component if it explicitly uses a service provided by that component. One negative side of dependence is the difficulty of using the dependent component if the server is not easily available. Another disadvantage is increased work for the people assembling these components. Two types of additional work are identified. One is connecting the component and server, which is not necessarily straightforward, particularly in case of interface mismatches. The second task is maintenance — the server will be visible to the assemblers, so they will have to deal with the future modifications in the server's interface (e.g. due to version updates). One way of decreasing a component's dependence is, making it swallow the component that it depends on. To be specific, a component that needs a service provided by another component may either use the server's interface explicitly, thus becoming dependent on it, or may encapsulate and hide the server, making the server part of its implementation. The latter option leads to less context dependent, coarse grain components. Excessive usage of this method causes massive components, challenging the benefits of modularity. On the other hand, the former option, (i.e. avoiding encapsulation of secondary services in components) leads to lean, more re-usable, but more dependent components. Component designers have to strive for a balance[Szy98a].

Loose Coupling

Coupling is a measure of interaction among components. It is proportional to the complexity of the interfaces, the complexity of the interaction protocol, and the complexity of the data that passes across the interfaces. Minimizing the coupling between components should be a major objective in component-oriented programming, since tight coupling has negative effects in various ways. Excessive coupling means complicated interfaces that are hard to standardize and thus hard to implement independently. Another issue is related to efficiency. In case of tight coupling, inter-component communications happen at much higher frequencies compared to loose coupling. This leads to penalties in execution speed, since inter-component message exchange is expensive. Furthermore, communications in case of tight interaction are much more synchronous; the caller of a service often has to wait for the result before it can continue. Maintenance is also more problematic, because component interfaces are harder to understand; modifications in component implementations are more likely to affect their interfaces; and separate analysis of modules is made more difficult. Coupling can be kept low by careful design of the system-level architecture. An example of this is using a coordinator (or mediator). The coordinator manages the connections between the components, and performs the computations that require an overall view of the system. The components only interact with the coordinator, and are independent of the others. Without a coordinator, the operations that require simultaneous information from several components can only be performed by complex and excessive interactions between the components. Less coupling can also be achieved by careful arrangement of component capabilities.

Sometimes loosening the coupling of a component adds complexity to its usage. For instance, reducing the frequency of information exchange is possible by exchanging blocks of data rather than individual data items. But an interface that accepts blocks of data is harder to use, as the client needs to implement a buffering mechanism. The Distributed Interactive Simulation standard [IEE93] offers another solution for lowering communication requirements, which has the same side effect: components try to estimate the status of the peer components as long as possible, rather than expecting frequent status messages. The simulation can proceed with much fewer messages exchanged, but the cost is the added complexity of status estimation.

High Cohesion

Cohesion refers to unity. A cohesive component does just one thing, or thinking in an object-oriented way, corresponds to just one entity at a particular level of abstraction. Embedding multiple, different kinds of behaviour in one component contradicts with the principle of modularity. Alternatively, distributing the implementation of a particular concept among two or more components leads to strong dependence between components, and complicated component interfaces. This is because the implementation of a single concept usually consists of multiple small-size elements tightly coupled to one another to form a conceptual unity. Apart from being necessary for little dependence and loose coupling, cohesiveness helps a component's re-usability, simply by making it a meaningful entity. Note finally, that cohesiveness does not imply being fine-grain. That is, it is possible for a cohesive component to contain several sub-components, which together implement a single, higher level concept.

2.4 Component Technology Standards

Component standards aim to provide a set of rules and mechanisms to define platforms for development and composition of components. Currently there are three major standards in the software industry namely CORBA (Common Object Request Broker Architecture), COM/DCOM (Component object Model) and JavaBeans/Java RMI (Java Remote Method Invocation. Since all of these standards aim to ensure location transparency (i.e. mechanisms for object method invocations should work consistently irrespective of whether the objects reside on the same host/in the same process, or in the same host/in a different process, or on a different host across the network) they are sometimes called Distributed Component Platforms. It is difficult, if not impossible, to compare these platforms and prefer one to another. There are several reasons for this. First, the market is highly competitive and grows rapidly. New features are added and announced continuously, which makes the situation hard to follow and may render a certain evaluation incomplete or obsolete after a short period of time. Second, most of the reviews coming from the industry seem to be biased, since support for a wrong choice may prove to be commercially fatal. And third, researchers are reluctant to provide a thorough analysis of the standards probably for reasons connected to first and second items. One exception to the third item, however, is Szyperski's relatively recent book. In his book[Szy98a] devotes a whole chapter to the comparison of the emerging standards. He identifies three fundamental issues that component standards need to address:[Szy98b]

- **Wiring.** Components entering a system need to be ”connected” to components that are already present.
- **Meta-Infrastructure.** Components need to be able to find about services provided by other components and they need to be able to locate and , in general, load and dynamically link components into a running system.
- **Semantic Interface Standards.** Being able to locate and call components is clearly necessary. However, without an agreement on what it means to invoke a particular operation, to post a particular event, or to handle a particular message, the entire approach remains meaningless.

In the remainder of this section some basic and common mechanisms of component standards are introduced. First, their approach to define component interfaces is introduced; second, basic mechanisms for implementing method invocation are introduced; third, basic issues related to naming and location of services they offer are briefly examined and finally a more detailed architectural view is drawn using software patterns.

Before continuing with this section, it is worth noting that although the choice made for the work described in this thesis is the COM/DCOM platform, this choice is not based on a detailed technical evaluation, therefore does not imply that the chosen system is technically superior to others. The COM/DCOM platform is chosen because it was easily available and the author is more familiar with this particular technology. The type of component technology used is not fundamental to this research; it is *the use of component technology* rather than a particular component platform which counts in pursuit of the illustration of the ideas defended in this thesis.

2.4.1 Interface Definition Languages

As mentioned in section 2.3.1, components communicate through their interfaces. Whenever a client needs some service from a remote distributed object, it invokes a method implemented by the remote object. The service that the remote distributed object (Server) provides is encapsulated as an object and the remote object's interface is described in an Interface Definition Language (IDL). The interfaces specified in the IDL file serve as a contract between a remote object server and its clients. Clients can thus interact with these remote object servers by invoking methods defined in the IDL. All of the component wiring standards define a form of IDL that has to be incorporated into the component specification. But their approaches differ, Those based on traditional object models define a one-to-one relation between interfaces and objects (e.g. in CORBA and SOM). An object provides the state and implementation behind one interface. Other approaches associate many interfaces with a single object (JavaBeans) or many interfaces with many objects in a component (e.g. in COM).

2.4.2 Remote Method Invocations

The idea in remote method calls is to replace local callee's end and the remote caller's end by *stubs*. In reality, the caller calls a local stub that *marshals (linearizes)* the parameters and sends them to the remote end. At that end, another stub receives the parameters, *unmarshals (delinearizes)* them and calls the true callee. The callee procedure itself, just as the caller, follows local calling conventions and is unaware of being called remotely. The marshaling and unmarshaling are responsible for converting data values from their local representations to a network format and on to the remote representation. To invoke a remote method, the client makes a call to the client proxy. The client side proxy packs the call parameters into a request message and invokes a wire protocol, like IIOP(in CORBA) or ORPC(in DCOM) or JRMP(in Java/RMI), to ship the message to the server. At the server side, the wire protocol delivers the message to the server side stub. The server side stub then unpacks the message and calls the actual method on the object. In both CORBA and Java/RMI, the client stub is called the stub or proxy and the server stub is called the skeleton. In DCOM, the client stub is referred to as a proxy and the server stub is referred to as a stub.

2.4.3 Naming and Location of Services

There is not an agreed upon approach on how to name interfaces and on how to relate them to each other between existing standards. COM uses a form called Globally Unique Indentifiers (GUIDs). GUIDs are used to name uniquely a variety of entities, including interfaces (IIDs), groups of interfaces called categories (CATIDs, and classes (CLSIDs). OMG CORBA originially left unique naming to individual implementations, relying on language bindings to maintain program portability. In CORBA 2.0, globally unique Repository IDs have been introdunced. These can either be DCE[4]UUIDs or strings to the familiar universal resource locators (URLs) used in the World Wide Web.

Given a name, all services provide some sort of a repository to help in locating the corresponding service. On top of this directory-like function, all approaches offer some degree of *meta-information* on the available services. The minimum that is supported by all is the runtime test of the types of the offered interfaces, the runtime reflection of the interfaces, and the dynamic creation of new instances.

2.4.4 An Architectural View: Illustrated Using Design Patterns

In this section design patterns are utilized to understand the architecture of and basic strategies used by component wiring standards (or component middleware). Design Patterns are an effective way of expressing proven techniques and designs used in software systems. They were first introduced by a building architect Christopher Alexander[Ale77] and later drew the attention of software engineering community. A book[5] written by a group of researchers known as the Gang of Four or GOF (Erich Gamma, Richard Helm, Ralph Johnson, amd John Vilissides) publicized the topic in the software comunity further. They define design patterns as descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context[Gam95]. They state that a design pattern names, abstracts and identifies the key aspects of a common design structure that makes it useful for creating re-usable design. There will be more

[4]Distributed Computing Environment (DCE)is a standard of the Open Software Foundation (part of the Open Group) and aims to implement remote procedure calls across heterogeneous platforms

[5]The book provides a catalog of 23 design patterns, and includes a detailed discussion of their structure, usage areas, consequences of using them, and their implementation alongside with examples of their employment in existing software systems.

references to design patterns in Chapter 5, when discussing the design of AniComp and its internal architecture. The patterns discussed here are based on an article by F.Plasil and M.Stal[Pla91] which draws an architectural view of the three most common component standards mentioned above. All the patterns discussed are well-known and cataloged by pattern books[Gam95, Bus96].

Broker Pattern

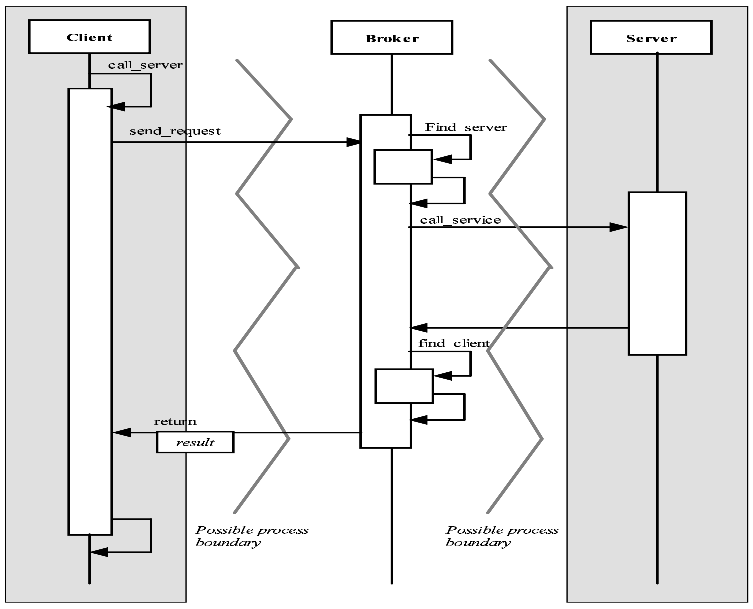


Figure 2.2: A Functional View of the Broker Pattern

The broker architectural pattern is used to structure distributed software systems with decoupled components that interact with remote service invocations [Bus96]. Figure 2.2 illustrates a simplified view of its functionality. As it can be seen, it reflects the classical client-server relationship implementation in a distributed environment. In most cases, broker collaborates with six types of participating components some of which are other patterns that will be discussed in the next sections. These participants are clients, servers, bridges, client-side proxies and server-side proxies. In a loop, the broker forwards requests and responses to the corresponding proxies. These requests and responses are encoded in messages by which the broker communicates with the proxies.

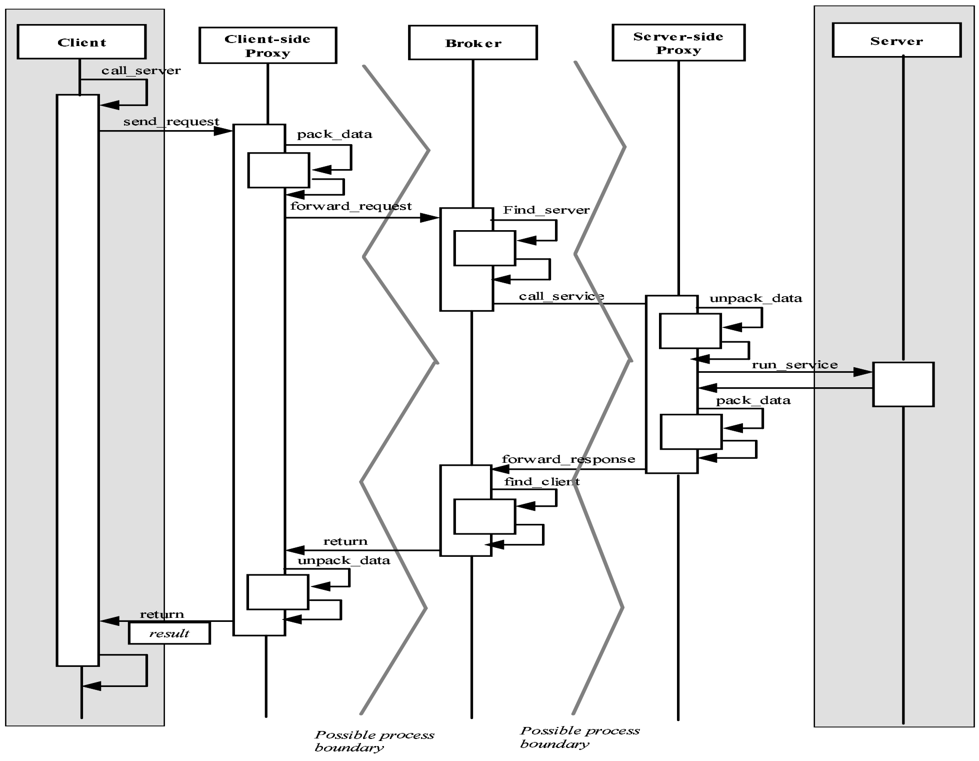


Figure 2.3: Broker and Proxy Patterns in collaboration

Proxy Pattern

The proxy pattern makes the clients of a component communicate with a representative, rather than to the component itself. Introducing such a place holder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorized access. Distributed component platforms employ proxies as wrapper-like objects to bridge the conceptual gap between the remote and local style of references both in the client and server code. Figure 2.3 shows the broker and the proxy patterns in collaboration during a typical scenario found in distributed component platforms. The client-side proxy and the corresponding server-side proxy communicate with each other to transmit requests and responses. The client-side proxy supports the same interface as the remote object does. Thus the client-side proxy acts as a local representative of the corresponding remote object by ensuring that calling a method `m` of the proxy achieves the same effect of calling the method `m` of the remote object. The role of the server-side proxy is similar. It delegates and transforms an incoming request into a local call form and inversely it transforms the result of the call to a form suitable for transmission back to the client-side proxy.

Abstract Factory Pattern

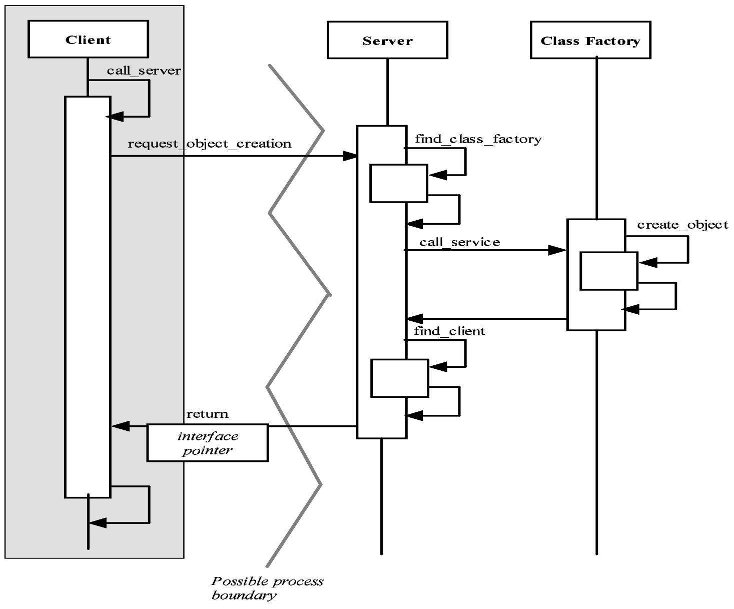


Figure 2.4: A Typical Scenario for Object Creation Using Class Factory

The abstract factory pattern provides an interface for creating families of related or independent objects without specifying their concrete classes[Gam95]. In distributed environments, factories are typically used on the server side to create new objects on a client's demand. The main reason for employing factories is that the usual language tools, like `new`, cannot be used in distributed environments. In a routine scenario the client issues a request to an object factory to create a new object; the object factory returns a remote reference to the newly created object. Figure 2.4 illustrates this scenario. Typically the created object resides in the same server as does the object factory involved in the request. In COM, for example, every object is an instance of a class and every class has a class factory which is used to create instances. The body of the code making up the class is called a COM server. In a server, each of the server COM object (component) classes is associated with an object factory instance residing in the server, which creates new server objects based on this particular class.

Observer (or Publisher/Subscriber) Pattern

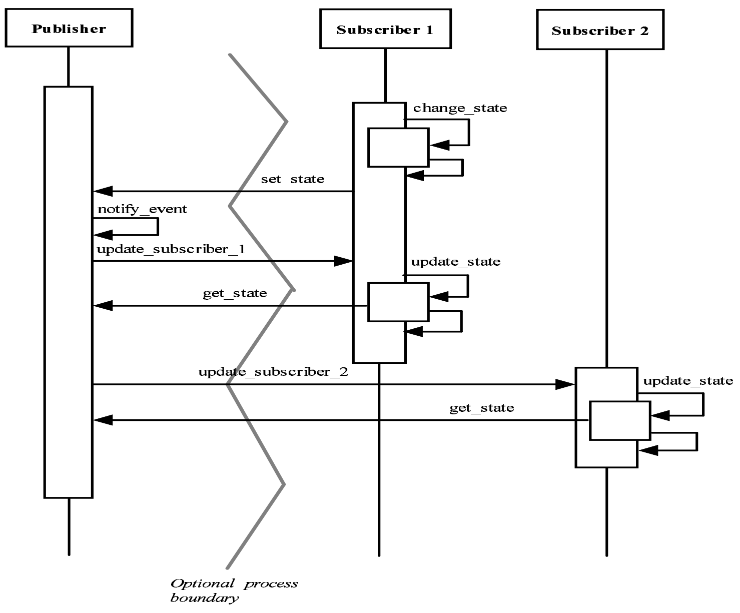


Figure 2.5: The Publisher-Subscriber Pattern

The Publisher-Subscriber pattern helps to keep the state of cooperating components synchronized. To achieve this it enables one-way propagation of changes: one publisher notifies any number of subscribers about changes to its state[Bus96]. This pattern facilitates the implementation of a programming technique know as reactive (or event-based) programming. The basic idea is that, as the definition of the pattern implies, a *listener* object (Subscriber) can subscribe to an event source (Publisher) to be notified about an event occurence. The Publisher announces what interfaces its listeners have to honor and what methods of these interfaces will be called to report an event. One of the key benefits of this approach is that the Publisher does not have to know its listeners at compile time; listeners subscribe dynamically. In Figure 2.5 a publisher updates the states of all of its subscribers on an event notification that interests them.

COM/DCOM provides *connection points* and *outgoing interfaces* to implement this pattern. Connection points and outgoing interfaces provide mechanisms through which a publisher announces a set of interfaces it will call to report on an event. Each of these "outgoing" interfaces has to be implemented by a separate connection-point object. A subscriber, *sink object*, implements some of the outgoing interfaces and subscribes with particular connection point(s). When asked by the publisher, the connection point notifies all of its subscribers by systematically calling the method that corresponds to the event being reported in all of its subscribers.

Reflection Pattern

The reflection pattern provides a mechanism for changing the structure and behaviour of software systems dynamically[Bus96]. Its execution is facilitated by the existence of meta-level information about selected parts of the system properties, which makes the system self-aware. This in turn requires the structuring of the system into two basic layers: Meta-level which consists of meta objects, and base-level which defines the application logic and uses meta objects to remain independent of those aspects that are likely to change. For example in a distributed system there may be meta objects that provide information about the physical location of base-level components.

Different component standards support reflection (or meta-programming) in different levels and flavors. COM does not fully support meta programming. However, it provides functionality for exposing and accessing type information about COM objects and their interfaces[6]. The information can be accessed to dynamically inquire about structural aspects of COM objects, and to create invocations of COM interfaces. A mechanism called *type library* is used by COM to provide run-time type information for all interfaces and classes described in that type library. Using the CLSID (a globally unique class identifier) of a class clients can query the COM registry for type information on that class. Available information includes the number and type of the parameters of a method, the categories to which a class belongs and the attributes of the interfaces that particular class defines.

2.5 Discussion: How to Study Component Interactions

In this thesis the issue of component interactions has a particular importance since one of the aims of this research is to attempt to understand the set of typical

[6]This mechanism has been used in this work, in the implementation of a component framework which contains a contract validation procedure. This is discussed in Chapter 6.

interactions between simulators and the animator component in a simulation framework, in order to achieve a successful interface design for AniComp. This section forms a preliminary discussion of that issue.

Component interactions are facilitated by properties and methods that are defined by a component and events that are triggered by it. As discussed earlier, properties and methods typically represent a component's callable API-the protocol used by external entities to access a component's services. Properties expose a component's public attribute data via accessor operations; methods inaugurate a component's behaviour. Events specify a component's response to external stimuli or internal conditions[Kri98]. Since interaction can only occur by the participation of at least two parties, specifying a component's interface alone does not suffice to understand the interaction sequences that a particular component is likely to go through. In fact for a successful design, the component developer needs to make reasonable assumptions about the interactions that the component's interface may be subject to. In object oriented design methodology, object interaction diagrams are used to capture the semantics of the possible scenarios for object communications. With the addition of scripts and focus of control notation, object interaction diagrams provide a powerful tool[Boo94]. This notation can be utilized to study component interactions. Component interaction diagrams were used throughout Section 2.4.4 to illustrate basic scenarios that key patterns employed by component standards take part in. A problem with using object interaction technique, however, is that their main purpose is to analyze the interaction between the objects of an existing system or during the design of a whole new system. For the case of off-the-shelf binary components it is not possible to exactly know which other components will interact with a particular component after its deployement. Off-the-shelf components, in principle, are developed independently of the system they will participate in. Thus the only way to reveal their expectation about possible interaction patterns that they will take part in is through their interfaces.

A useful approach to study component interactions based on their interfaces, is viewing component interfaces as *contracts* between a client of an interface and a provider of an implementation of an interface[Szy98a, Wec97a]. The next section explains this approach.

2.5.1 Component Interfaces as Contracts

The contract states what the provider has to implement to meet the services promised by the interface. On the level of an individual operation of an interface, there is a particularly popular contractual specification method: The two sides of the contract can be captured by specifying preconditions and postconditions for the operation. The client has to establish the precondition before calling the operation, and the provider can rely on the precondition being met whenever the operation is called. The provider has to establish the postcondition before returning to the client and the client can rely on the postcondition being met whenever the call to the operation returns. This leads to a triple specification consisting of precondition operation postcondition, for each operation in the interface. Although this approach introduces a limited semantics to the interface definition and gives some hints as to what type of interactions are expected between a provider and its client, it does not capture the interaction scenarios sufficiently. Furthermore the idea of contracts does not fit well in systems containing *callbacks*. Callbacks reverse the direction of the flow of the control and may reveal the intermediate state of the provider to the clients and cause it to change through direct and indirect calls. This may lead to cases where the contracts are broken.

Koen at.al. extends the idea of *interfaces as contracts* into *re-use contracts* [Hon97]. They define a re-use contract to be essentially an interface description for a set of *collaborating* participant components. A re-use contract reveals the participants that play a role in the re-use contract, their interfaces, their acquaintance relations, and the interaction structure between acquaintances. Koen at.al. state that since the interaction structure between a component and its acquaintances is crucial to get a good understanding of the component architecture, and to ensure correct composition, re-use contracts not only provide the interface of a component, but they also document what interface a component requires from its acquaintances and what interaction structure is required for correct inter-component behavior.

Re-use Contracts

A re-use contract documents the assumptions each participant makes about its acquaintances. When a component developer builds a component, he can rely on these assumptions to implement the component according to the participant descriptions. However, requesting that a component is fully compliant with the interface and interaction structure descriptions, would make re-use contracts too constraining, and consequently too impractical to use. Instead, components may deviate from the re-use contract, but the component developer has to document how they deviate exactly, so that this information can be used later on to perform conflict checking. Therefore, re-use contracts are subject to so-called re-use operators, or modifiers, actions that adapt participants and the interaction structure between these participants. Typical re-use operators on re-use contracts are extension and refinement, and their inverse operations, cancellation and coarsening. These operators come in two flavors: one flavor handles the operations on a participant, while the other flavor handles the operation on the context of a re-use contract, being the set of participants and their acquaintance relationships. Koen at. al. develop a graphical notation to illustrate a re-use contract.

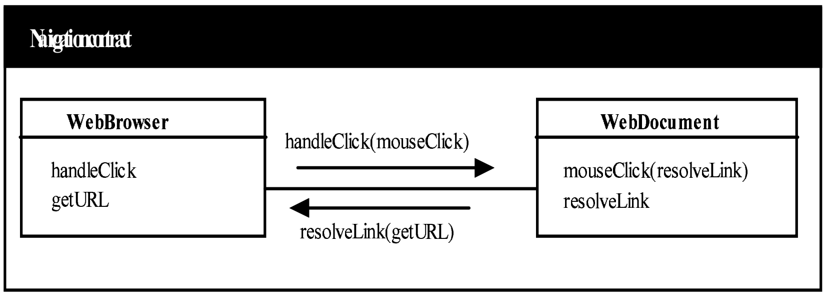


Figure 2.6: A Re-use Contract Illustrated by a Diagram

Just to illustrate the notation an example figure is taken from their paper[Hon97] and given in Figure2.6 where `handleClick` on `WebBrowser` invokes `mouseClick` on `WebDocument` and `WebDocument` invokes its resolveLink operation when the mouse was clicked on a link. `resolveLink` invokes `getURL` on `WebBrowser` in order to get the contents of the web page pointed to by the link.

Re-use contracts will be used in chapter 5 when analyzing the interaction of AniComp with its environment, as a more formal way of specifying the component's interface to facilitate its successful composition by third parties.

2.6 Discussion 2: Problems with Re-usability and extendibility of Components

Re-usability is a well known and highly publicized issue in software engineering. It is not only used in the context of component re-use as introduced previously but to denote a broader meaning. Sommerville mentions four types of re-use including (1) Application system re-use, (2) Sub-system re-use (3) Module or object re-use and (4) Function re-use [Som96]. He also distinguishes software development *with* re-use, meaning developing software by using existing re-usable software components, from software development *for* re-use, referring to the development of the software components with their re-usability in mind (as a primary goal). In traditional practice, however, its application has either been in the form of re-using procedural (binary or source code) libraries or class libraries in source code form.

The Component-Based approach to software development, in its most radical form, promotes *black-box re-use*, referring to binary components being re-used off-the-shelf. This implies, as discussed earlier, that the components are developed independently of each other by different vendors. Since the only way the binary components can agree to communicate, is through their interfaces, the issue of designing interfaces capable of entering into valid contracts with other possible partner components, presents itself as a problem. Obviously it is impossible to foresee every possible interaction that a component can take part in by its designer: hence the problem of interface mismatches. One answer to this problem is to enforce a framework, specific to an application domain, which regulates the component collaborations by enforcing certain rules. Domain specific component frameworks, however, possibly based on more basic component wiring standards, may not totally eliminate interface mismatches, since an independently developed component may not support the framework completely but still need to be composed into a system based on the framework. An approach known as *interface adaptation* is used to resolve the problem of this sort (and other cases of interface mismatches). Approaches to interface adaptation include: (1) Automated techniques which are based on interface processing to automatically create adapter components, (2) approaches which suggest layering the internal structure of the component to allow the integration of the adapters as outer layers, and (3) techniques which suggest the use of white-box customizable adapter components [Kuc98].

Another important issue is how to incorporate extension into binary components. Designing extensible systems at source code level is relatively well studied and, in fact, some architectural patterns have been identified that accomodate extensibility into a software system. The *Microkernel*, which suggests that more stable functions of a system should be structured as a core part and functions that are more likely to evolve should be implemented as layers of internal and external servers coupled with adapters where necessary, is such a pattern. One other pattern is *Reflection* which was discussed in Section 2.4.4. Although these architectural patterns can be employed in the design of binary components, still, the issue of extending independently developed binary off-the-shelf components is relatively under-researched. Weck introduces the term "independently extensible component frameworks" and identifies some principles for realizing them [Wec97b].

He defines an extensible system as a system that allows the addition of functionality at run time. Only the functionality currently used is loaded into the computer and may consume resources. Further functionality can be added when needed. The software to be added can even be retrieved via the network. A system is called independently extensible if extensions can be developed by different people in complete ignorance of each other. Still, extensions are expected to cooperate where appropriate; at least, they must not to interfere with each other. He further states that independently extensible systems can change the way software is marketed. Instead of monoliths, designed to fit all, small software components can be offered. The customers select the functionality they want, buy the appropriate components, and compose their system from them. In this thesis a structure called *Flexi-Switch* has been introduced which along with facilitating dynamic contract validation, can incorporate run-time switching capability from one component to another as the need to change the functionality (behaviour) of a certain part of the component arises. The structure is flexible in the sense that the range of options for the alternative behaviours can also be extended at run time. A good example for a possible usage of this structure is the use of different collision detection algorithms that can be used in the animator component. Collision detection algorithms can evolve in time. As components from different vendors come to the market they can be connected to the collision detector switch as long as their interface is in agreement with the animator client.

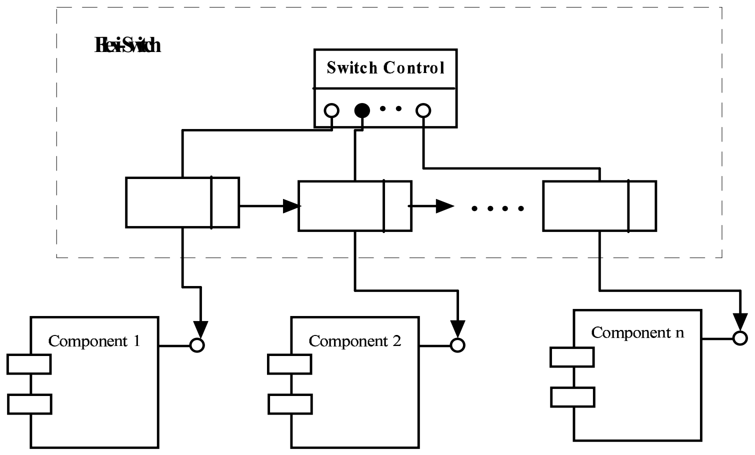


Figure 2.7: The Flexi-Switch structure

Figure 2.7 shows an illustration of the structure. The black circle in the switch control denotes the active component. The design and implementation of the Flexi-Switch will be discussed in more detail in Chapter 6.

2.7 Summary

This chapter presented a detailed account of issues related to component based software development, including the definition and various features of individual components, basic patterns describing the architecture of distributed component platforms and the contractual nature of component interfaces. The first discussion at the end of the chapter aimed to introduce re-use contracts as a way of specifying components interfaces and modeling interactions between collaborating components. The second discussion underlined the importance of run-time extensibility for component frameworks.

Chapter 3

Animation Techniques and Systems

3.1 Introduction

This chapter provides an introductory discussion of animation techniques and animation systems. First, most popular animation techniques are introduced based on a classification which aims to highlight differences emanating from the method of creating animation sequences. Then, the approaches to the construction of animation systems are introduced, aiming to sketch an overall structural view and also to gain some insight into the internal mechanisms. Virtual environments is the last topic discussed as a type of system where animation finds its usage in a different context compared to more conventional animation systems.

3.2 Classification of Animation Techniques

Animation techniques can be classified based on different criteria. They can, for instance, be classified based on whether the animated objects are rigid or soft objects. Alternatively, the classification can be made considering whether the animation is generated in real time or produced frame by frame, recorded and later played back at a rate between 25 to 30 frame per second. In this thesis the primary concern is to study simulation models as the originator of the animation sequences. Therefore the classification is based on whether an animation is created by a model and/or procedure, or whether it is generated by interactive editing performed by a human animator. By examining the animation techniques with this distinction in mind, it is anticipated that a foundation can be established for understanding the underlying logic of the simulator-animator interactions.

3.3 Interactive Animation Techniques

3.3.1 Keyframing

Keyframe systems take their name from the traditional hierarchical production system first developed by Walt Disney. In this system first a master (skilled) animator would draw the most important, or "key" frames (called **keyframes**) of an animation sequence, then a number of less experienced animators would draw the **i**n-betweens, or frames that fell between keyframes. The emulation of this technique by the computer, whereby the interpolation replaces the in-between artist, producing the in-betweens automatically, was one of the first computer animation tools to be developed [Wat92]. This technique was quickly generalized to allow for the interpolation of any parameter affecting the motion, thereby providing a higher level of control than traditional keyframing.

The simplest form of keyframed animation involves changing basic parameters of three forms of transformation. A 3D object have 9 values that characterizes its transformational state. These are *translation(x,y,z)*, *rotation(x,y,z)* and *scale(x,y,z)*; that is, its translation, rotation and scaling along three coordinate axis. In an interactive animation system, the human animator specifies the key values for these nine parameters, and the system calculates the intermediate values by interpolation. The parameter to be interpolated, however, may also be, for example, a joint angle of an appendage of a robot, or the transparency attribute of an object, or any other parameter used in the display or manipulation of the graphics entities. One simple form of interpolation is the *linear interpolation* in which the rate of change of the value of the interpolant remains the same over the entire length of the interpolation. Linear interpolation, can result in abrupt change between consecutive frames, characterized by a sudden change in the slope of the line, as illustrated in Figure3.1 (a). This is an unwanted effect as it distorts the smoothness of the motion. *Spline interpolation* eliminates this problem, where key frames constitutes the control points of the interpolating spline, resulting in a smooth, continuous change in the rate at which the interpolant is changing.

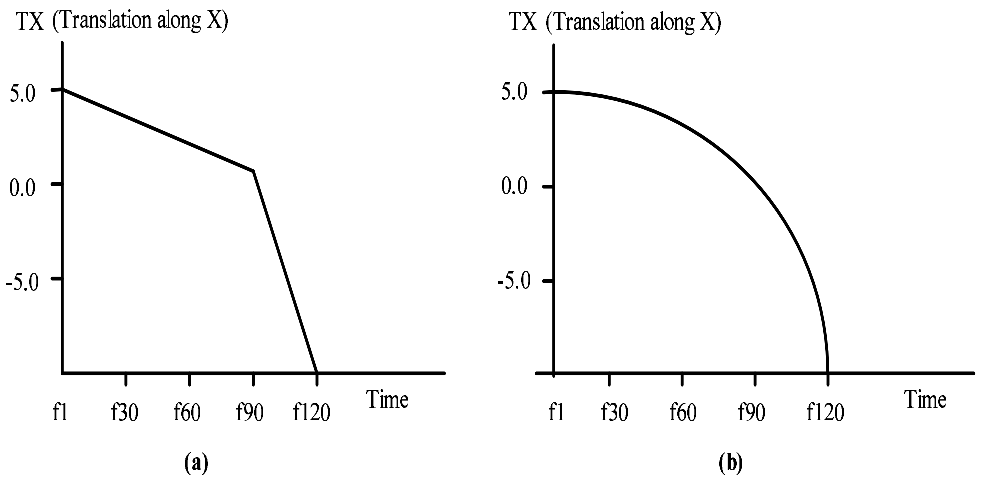


Figure 3.1: Motion of An Object Along An Animation Path and Corresponding Keyframes

Figure 3.1 (b) illustrates spline interpolation.

Splines can be used to directly manipulate the parameters that control the animation. In that approach the spline becomes a direct and immediate representation of the animation. If the graph of the spline is changed, then the animation changes accordingly. Many 3D software packages allow the human animator to modify, or edit, these parameter graphs and thereby to directly re-define, or edit, the animation itself. The next two sections are examples of this approach.

3.3.2 Motion paths (Space Curves) and Velocity Curves

Motion paths are used to move an object along a path in space, specified by a spline, say $Q(u)$, over a given number of frames. If the object is required to move along the path with a uniform speed, then the computer's task is to find a set of points along the spline such that the distance traveled along the curve between consecutive frames is a constant. The object can then be positioned at these points frame for frame. Finding these points requires a procedure called *arclength parameterization*, which is not a trivial task since the relationship between the parametrizing variable and the arclength is very nonlinear. Watt [Wat92] gives a detailed discussion of how this process works. The velocity curve is a two dimensional spline of distance traveled, or arclength, against time. Figure 3.2 illustrates a 3D torus object placed at different positions along a spline path to

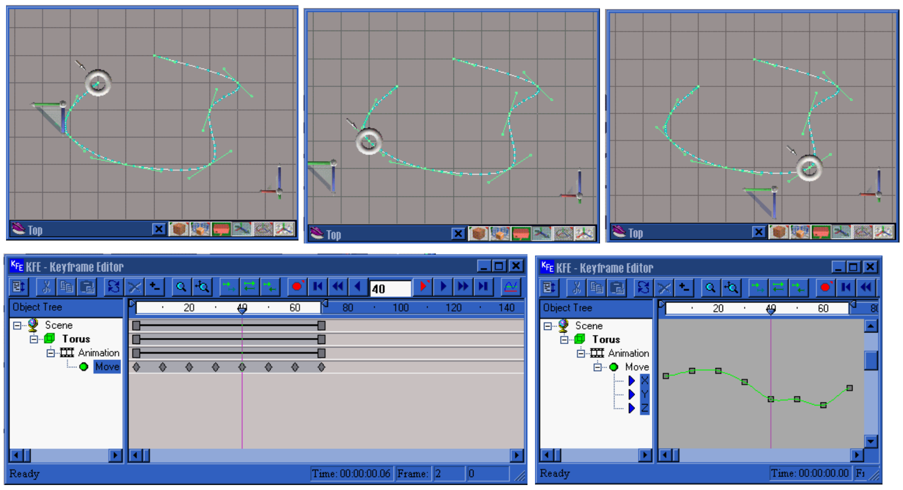


Figure 3.2: Frame Transition in Linear and Spline Interpolation

specify key frames. A key frame interpolator would then create a smooth motion of the torus along the path.

The space curve and velocity curve, whether interpolated or not, are specified independently. Different velocity curves can be tried on the same space curve and vice versa. Modification of the motion can be made without affecting the space curve. Working with velocity curves is intuitive. Useful curves can be stored by the human animator and used across a wide variety of applications. One of the well known and widely used curves that can be part of such a library is ease-in/ease-out curves.

3.3.3 Animation of Articulated Figures: Kinematics

An articulated figure is a structure that consists of a series of rigid links connected by joints. Articulated figure animation has become more popular especially due to the growing interest in simulating synthetic actors (humans or other creatures) in 3D animation environments. The direct (or forward) kinematics problem consists in finding the position of end point positions (e.g. hand, foot) with respect to a fixed-reference coordinate system as a function of time, without regarding the forces or the moments that cause the motion. In the case of forward kinematics the human animator has more and more transformations to control, which while lending more freedom to achieve a more expressive animation, may prove to be extremely complicated and difficult to achieve in practice. Once created, however, a library of pre-specified animations can be used for well defined, repeated tasks.

The inverse kinematics permits direct specification of end point positions. Joint angles are automatically determined. Inverse kinematics animation is sometimes called "goal directed motion" since a human animator can set up the animation, using high-level, goal-directed motion commands which are satisfied by an inverse kinematics engine. Unfortunately, the transformation of position from Cartesian to joint coordinates generally does not have a closed-form solution. Watt [Wat92] presents a detailed discussion on the articulated body animation, including techniques for representing the articulated structure, and detailed steps in calculating the joint values once the end effector's position is given.

3.4 Automatic Motion generation

Current technology is not capable of generating motion automatically for arbitrary objects. However, algorithms for specific types of motion can be built. These techniques are called procedural methods because a computer procedurally follows the steps in an algorithm to generate the motion. Procedural methods have two main advantages over keyframing techniques: (1) they make it easy to generate a family of similar motions, (2) they can be used for systems that would be too complex to animate by hand, such as particle systems or flexible surfaces.

Simulation imply automatic motion generation by its nature. Considering their internal mechanisms for creating motion, simulated objects (or systems) can be divided into two broad categories: passive and active. Passive systems have no internal energy source and move only when an external force acts on them. They are well suited to physically based simulation because once the physical laws have been encoded correctly and the initial conditions of the animation have been specified, the method is ready for use. Pools of water, clothing, hair, and leaves have been animated using passive simulations. Active systems do have an internal source of energy and can move of their own volition. People, animals, and robots are examples of active systems. These systems are more difficult to model because in addition to implementing the physical laws, the behavior of the simulated muscles or motors must be specified. An additional algorithm, a control system, must be designed to allow the model to walk, run, or perform other actions.

Procedural methods can also be used to generate motion for groups of objects that move together. Flocks of birds, schools of fish, herds of animals, or crowds of people are all situations where algorithms for group behaviors can be used. The following section presents various established procedural techniques.

3.4.1 Physically-Based Modeling and Dynamic Simulation

Physically based simulation refers to a class of procedural methods that makes use of the laws of physics, or an approximation to those laws, to generate motion. On the other hand, physically-based modeling incorporates physical characteristics into models, allowing numerical simulation of their behavior. Common elements in physically based-modeling are classical dynamics (motion based on forces, mass, inertia, etc.) with rigid or flexible bodies; inter-body interaction; and constraint-based control. Physically-based models are founded on mathematical equations and often involve numerically intensive computation to simulate their behaviour. The emerging behaviour (i.e. simulated motion) is inherently realistic and for many applications that is an advantage. Unfortunately, building a new simulation is sometimes a difficult process requiring an in-depth understanding of the relevant physical laws. Once a simulation has been designed, however, a human animator may use it without knowing how the internals of the simulation function.

Physically-based modelling and simulation is a relatively recent approach in computer animation compared to more classical key-framing techniques. Being computationally expensive its widespread use has been hampered by the limits of computer hardware until recently. However, with the rapid improvement of the computers' processing power and development of high performance 3D graphics cards even for desktop PCs, this approach has become more and more viable. Increasingly, recent real-time 3D computer game developers are announcing the incorporation of physics engines into their games for a more realistic virtual world experience.

Dynamic simulation, based on physical models, offers two potential advantages over other sources of motion for synthetic characters in virtual environments. First, simulation generates physically realistic motion that may be difficult to create using keyframing. While not all environments need or even benefit from physical realism, a growing set of applications like sports training, task training, and team- oriented games require it. Second, because their motion is computed on the fly, dynamically simulated characters offer a more precise form of interactivity than characters animated with a fixed library of precomputed or recorded motion. For example, in football video games, the motion resulting from a collision between opposing players is a function of the magnitude and direction of their velocities as well as their body configurations at the time of impact. Because of the very large number of initial conditions, its difficult to model this interaction accurately with a library of fixed motions.

It is worth mentioning that a considerable amount of research effort have been devoted to devise a formal, structured approach to develop physically-based models of objects. These efforts are particularly important because they help us to develop a generic model for incorporating physical realism into the objects' behaviour thereby enabling us to clarify the general rules that govern simulation-animation interaction and integration. Some of these approaches are summarized below: Hegron et.al. [Heg96] mentions a structured approach adopted from Luciani et.al. [Luc91] which classifies the physical models in the following way:

- The continuous distributed model:This is the model proposed by the classical analytical physics which describes object behaviour with a set of differential equations. The solutions are accurate but time consuming. The articulated rigid object model falls into this class.

- The continuous localized constants model: This model deals with a structural discretization of the object. For each component, a continuous distributed model is used to express its mechanical behaviour. Some analytical equations link the different components to provide motion equations for the entire mechanical system. The finite elements method lies in this family.

- The discrete localized constants model: This model also contains a structural discretization of the object but implicitly defines a computation algorithm for obtaining the emergent mechanical behaviour of the object. A particle system belongs to this class.

Hegron also sketches a computational scheme similar to one in Figure 3.3 where, based on the physical model and mechanical properties of a system, a main loop numerically evaluates and computes the motion equations, thereby producing the motion parameters that can be fed into an animation system. This scheme

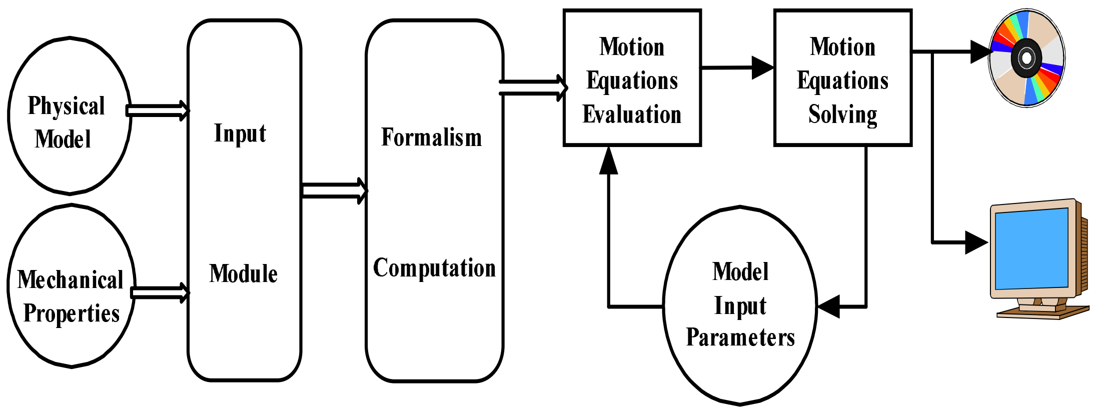


Figure 3.3: The Computational Scheme for Dynamic Simulation and Animation

provides a general framework that delineates simulation-animation interaction for the case of continuous simulation and will be utilized in designing the AniComp's interfaces later in the thesis.

In a notable work by Barzel [Bar92] an attempt is made to devise yet another structured approach to the definition and implementation of physically based models. His attempt is particularly important since it has the potential to facilitate component-based construction of behavioral simulators for virtual objects. Based on an analysis of a general modeling activity, Barzel identifies the Abstraction-Representation-Implementation (ARI) paradigm from which he derives a canonical structure for physically-based modeling. The structure is called CMP where C stands for *conceptual model*, M for *mathematical model* and P for *posed problems*. A well-defined CMP structure, he states, helps a model to be robust, re-usable, and extensible; and helps us to understand, build, and debug the model.

In another notable work, Witkin et.al. [Wit97] devise an architecture for simulating the motion of rigid bodies under unconstrained as well as constrained conditions. They gradually develop terms, concepts and equations required to build a rigid body simulator eventually providing a set of algorithms and structures for their implementation.

Interaction in a Dynamics Simulation

A particular area of interest in simulation and computer graphics has been the use of dynamic simulation or related optimization techniques as a means of geometric interaction, allowing users to directly manipulate simulated objects subject to constraints. This approach has been applied to animation as well as areas like free-form surface modeling, mechanical design and interactive molecular simulation. Other interesting application areas involve training and education, where systems that support realistic motion within the geometric constraint of a layout, allow users to experiment with and practice strategies for assembling and disassembling equipment and mechanisms. Another application area is robot path-planning and learning, where contact and collision dynamics of robots and their surrounding environments are simulated quickly. Hence, a significant amount of research has been devoted to this area. Barzel [Bar88] developed a modeling system based on dynamic constrains. Later research works introduced interaction into the dynamic simulation environments [Wit90]. Baraff [Bar95] implemented an interactive system for simulating rigid bodies with contact and friction for moderately complex mechanical systems. Harada et.al. [Har95] extended the physically-based interaction paradigm to encompass the exploration of discrete as well as continuous problem space, to allow objects to spontaneously undergo some discrete re-arrangement with the least perceived disruption in the overall arrangement. Mirtich and Canny [Mir94, Mir95] introduced impulse- based simulation as opposed to constraint-based simulation of rigid bodies, which according to the authors, unifies all types contact (colliding, rolling, sliding, and resting) under a single framework, and provides a simpler and faster method compared to constraint-based methods, making it even more suitable for interactive simulation environments.

Despite its benefits and promises, however, the computational cost it incurs is, still, one disadvantage of dynamic simulation for complex applications. Its computational complexity makes it unsuitable for real-time animation especially if the animation environment contains many dynamically simulated objects (or actors). Looking from the character animation point of view Thalmann [Tha96] argues that dynamic simulation also imposes some limitations on the behavior of synthetic characters. He states that simulated characters are less maneuverable than those modeled as point-mass systems and those that move along paths specified by the human animators. For example, although a point-mass model can change direction instantaneously, a legged system can change direction only with a foot planted on the ground. If the desired direction of travel changes abruptly, the legged system may lose its balance and fall. These limitations, although physically realistic may not always be desirable or visually plausible. This problem is addressed by Barzel [Bar96] who introduces the idea of plausible motion: motion that could happen, given what is (un)known about the system and uses dynamic simulation in a different framework. He states that recasting the animation simulation problem domain to be one of plausible rather than accurate motion opens up a variety of interesting and promising directions for investigation.

3.4.2 Particle Systems

Particles are used to simulate two major types of phenomena:

1. Mass and spring systems such as cloth or non-rigid structures
2. Natural phenomena such as fire, cloud, smoke etc.

Although the fundamental definition of a particle remains mostly similar in both uses, there are some differences which may lead to substantial variations in the overall behaviour of the simulated system. The following sections briefly explain the two approach.

Mass and Spring Systems

The most simple and intuitive way of designing a mechanical simulation system is to consider the object as being discretized into a set of vertices that interact with each other through elastic forces. A time discretization process then updates numerically the position and speed of each vertex and yields the evolution of the system. By opposition to continuous systems, particle systems work on explicit discretizations of the simulated objects. Based on this simple idea, a huge category of particle system based simulation techniques have been worked out, which mainly differ from one another by the way the forces between the particles are computed. The simplest models, called spring-mass models, consider a triangular mesh where the vertices are masses and the edges are springs with constant rigidity and optional viscosity. These models yield very simple computations, but are

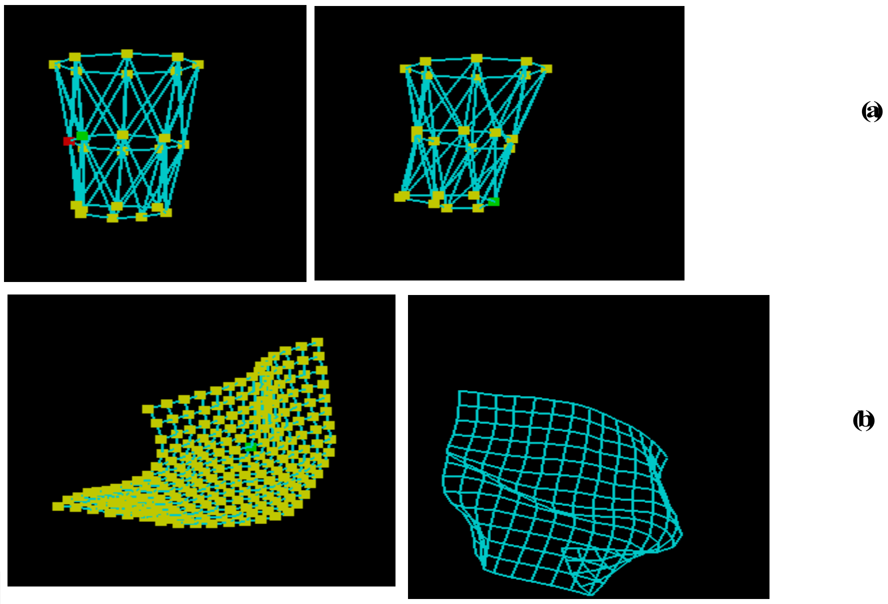


Figure 3.4: Particle Systems Consisting of Mass and Springs: (a) Two Poses of A Hooked Hoop (b) Two Poses of A Cloth

not very accurate for simulating deformable surfaces, since an array of springs cannot represent exactly the elastic behavior of a plain elastic surface.

Mass and spring systems are modeled as particles that have mass, position and velocity, and entities that apply forces to particles. Witkin [Wit97] groups the forces that act on the particles into three broad categories:

- Unary forces, such as gravity and drag, that act independently on each particle, either exerting a constant force, or one that depends on one or more of particle positions, particle velocity, and time. The effect of drag is to resist motion, making a particle gradually come to rest in the absence of other influences.

- n-ary forces, such as springs that apply forces to a fixed set of particles.

- forces of spatial interaction, such as attraction and repulsion, that may act on any or all pairs of particles, depending on their positions. Figure 3.4 illustrates the simulation of a cloth and a hoop using mass and spring model[1].

[1]These screenshots were produced using two programs provided by March 1999 and May 1999 issues of Game Developers magazine(http://www.gdmag.com)

Natural Phenomena

The use of particles in that sense goes back to Reeves's work in 1983 [Ree83]. He defined a particle system as a large collection which, taken together, represent a fuzzy object. Because there is typically a large number of particles, simplifying assumptions are used in rendering them and in calculating their motions. Different authors make different simplifying assumptions. Some of the common assumptions made are: Particles do not collide with other particles; they do not cast shadows, except in an aggregate sense; and they do not reflect light - they are each modeled as point light sources.

Particles are often modeled as having a finite life-span so that during an animation there may be hundreds of thousands of particles used, but only thousands are active at any one time. In order to avoid excessive orderliness, randomness is introduced into most of the modeling and processing of particles. In computing a frame of motion for a particle system, the following steps are taken:

- Any new particles which are born during this frame are generated,
- each new particle is assigned attributes,
- any particles which have exceeded their allocated life-span are terminated,
- the remaining particles are animated and their shading parameters are changed according to the controlling processes
- the particles are rendered

Particles are generated according to a controlled stochastic process. One method is to, at each frame, generate a random number using some distribution distribution centered at the desired average number of particles per frame with a desired variance. The attributes of a particle determine it's motion status, its appearance, and its life in the particle system. Typical attributes are its position, velocity, shape parameters, color, transparency and lifetime. In Figure 3.5 (a), for example, the bonfire image is generated by particles with a shiny color and a level of transparency[2]. Each of the attributes are initialized when the particle is created. Again, to avoid uniformity, values are typically randomized in a controlled way. The position and velocity are updated according to the objects motion. The shape

(a)

(b)

Figure 3.5: Particles Used in Simulation of Natural Phenomena: (a) Exploding Fireworks (b) Volcanic Eruption

parameters, color, and transparency control the objects appearance. The lifetime attribute is a count of how many frames the particle will exist. At each new frame, each particles lifetime attribute is decremented by one. When the attribute reaches zero, the particle is removed from the system. Typically, each active particle in a particle system is animated throughout its life. This includes, not only its position and velocity, but also its display attributes: shape, color, transparency. To animate the particle's position in space, the velocity is updated, average velocity is computed and used to update the particle's position, and the particle's velocity is updated. Gravity, other global force fields (e.g. wind), local force fields (vortices), and collisions with objects in the environment are typically used to update the particle's velocity. The particle's color and transparency can be a function of global time, its own life-span remaining, its height, etc. The particle's shape can be a function of its speed.

[2]These screenshots were obtained using example programs included in the Microsoft's DirectX 7 Software Development Kit(SDK.)

3.4.3 Behavioral Animation

In behavioral animation an autonomous character determines its own actions, at least to a certain extent. This gives the character some ability to improvise, and frees the animator from the need to specify each detail of every character's motion. However, while in some limited sense autonomous characters have a mind, their simplistic behavioral controllers are more closely related to the field of artificial life than to artificial intelligence.

Like particle systems, behavioral animation is used to control the motion of many objects automatically. The primary difference is in the objects being animated. Instead of simply procedurally controlling the position of tiny primitives, motion is generated for ”actors” with orientations, ”current state”, and interactions.

Behavioral animation has been used to animate flocks, schools, herds, and crowds. All of these require interaction between a large number of ”characters” with relatively simple, rule-based motion. Fabric can also be simulated using behavioral techniques.

The animation techniques discussed up to now, aimed to provide a general understanding of how 3D objects can be animated. The next section introduces various approaches to the construction of animation software systems.

3.5 Approaches to the Construction of Animation Systems

It is desirable for an animation system to support the application of animation techniques discussed above as extensively as possible. However it is difficult to include all of them in a single package, since different animation techniques operate based on distinct paradigms making it difficult to integrate them in a common framework. Although there are efforts to create such frameworks [Pre96, Bar97] their coverage is still limited to a class of techniques. Many animation systems therefore consists of a core module centered around a keyframing system, other paradigms being implemented as add-ins (or plug-ins). This is especially true for commercial animation packages since they are mostly aimed at entertainment industry where interactive keyframe animation is still a central technique of creating non-real-time animation sequences. 3D Studio Max is a good example of such type of extensible systems [Stu98]. It is useful to identify the central paradigm that a particular system relies on, to determine the basic capabilities and the primary approach of that system. Applying this criteria one can classify animation systems as actor-based systems, scripting systems, object-oriented systems, constraint-based systems and real-time animation systems or virtual environments. However one should note that these categories are not necessarily exclusive of one another and they may (and usually do) overlap, especially in the case of commercial systems which are increasingly rich in paradigms they support. With the exception of the last category all of the named systems have a form of key-framing system at their core and allow a form of interaction for the human animator in editing the motion sequences. Virtual environments, however, are not animation systems in the usual sense. Animation is a tool used to create simulated behaviours of the objects in the virtual environment aiming to add a sense of realism to it. Interaction, in that case, refers to the interaction of the user with the environment and other inhabitants (if any), rather than the interaction of the human animator with the animation mechanisms in the system. The issue of virtual environments is diverse (and distinct) enough to deserve a separate section and will be further explored in Section 3.6.

3.5.1 Actor-Based and Scripted Systems

To continue with the discussion on animation systems, two systems can be mentioned as the pioneers of actor-based and scripting systems. ASAS (Actor Script Animation System) [Rey82] was the first system to introduce the concept of an actor. An actor in ASAS holds its own internal animation specification embedded in its definition and is able to communicate with other actors by sending messages. An actor is responsible for a visible element in an animation sequence holding a chunk of code which contains all values and computations specific to that visible element and gets executed once every frame. ASAS uses a LISP like scripting language. MIRA [Tha85] is another scripting system. It contains an artist oriented interface called

MIRANIM, and uses a scripting language called CINEMIRA which is an extension of Pascal programming language. CINEMIRA contains a modeling system and an artist-oriented editor which has eight operational modes to allow human animators to manipulate scene background, lights, cameras; create and modify objects and actors and finally animate them. Although scripting systems can provide powerful ways of specifying motion, it is reported that they have decreased in popularity with the advent of interaction in production animation [Wat92].

3.5.2 Object-Oriented Systems

The popularity of object-oriented methods in software development have been reflected to the design of animation software resulting in object-oriented animation systems. ASAS and MIRA mentioned in the previous section, are early examples which essentially carry object oriented ideas [Erk95]. Clockworks [Bre87] is a system with integrated capabilities such as modeling, image synthesis, animation and simulation, internally based on object-oriented concepts such as inheritance, class instantiation and message passing. Erkan and Ozguc present a system where they create and control animation sequences using levels of motion abstractions based on a recursive and hierarchical mechanism [Erk95]. REALISM is another animation system by Palmer and Grimsdale [Pal96] designed to allow explicit or implicit control of animated objects to enable a sequence of desired events to be visualized. To this end it contains a hierarchical control scheme consisting of a script, a scene and an actor. Each of these major classes encapsulate enough data and processes to govern its own behaviour and to interact with other classes through message passing in classic O-O style. REALISM facilitates the dynamic behaviour of objects by two distinct mechanism: rules and constraints. Rules are defined (and applied) on objects to model their behaviour, subject to the conditions imposed by constraints. Both of these mechanisms are themselves implemented as dynamic objects. A more recent system developed by Dollner and Hinrichs as an object-oriented toolkit, called MAM/VRS (Modeling and Animation Machine / Virtual Rendering System) [Dol97], consists of two main layers: the rendering layer and the graphics layer. The rendering layer is closer to the hardware (rendering libraries) and graphics layer rests on top of it, holding more abstract structures such as geometry nodes, behaviour nodes and 3D widgets. In MAM/VRS a 3D application is organized as geometry nodes and behaviour nodes that at the end form two separate directed acyclic graphs and which can be processed independently. These nodes must be able to understand a set of semantic specific communication protocols. Animation and interaction is specified by nodes which apply constraints to graphics objects which are associated with geometry and behaviour nodes. Constraints, having two distinct types called one-way and multi-way constraints, play a central role in specifying animation sequences. The authors also claim that separating behaviour nodes as first class objects, enable the design of re-usable behaviour components and facilitates the modeling of complex animations and interactions.

The Object-oriented paradigm, when applied to animation system design, promises an intuitive, relatively easy to use system by capturing the logical separation present in the related domains and reflecting this into the internal architecture of the system. Figure 3.6 illustrates a very general outline of a typical object oriented system at design level, pointing out the main logical components of such a system.

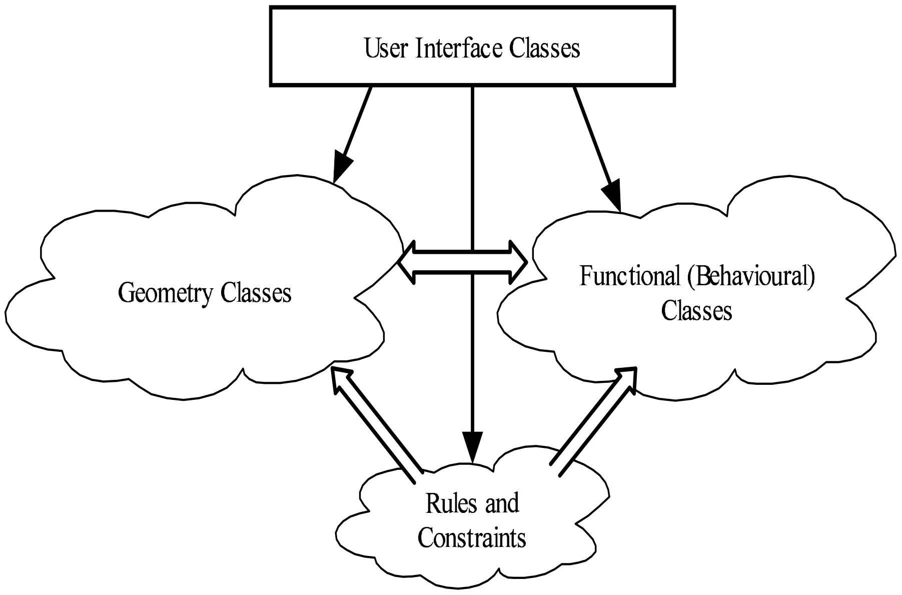


Figure 3.6: A General Picture of Object-Oriented Animation Systems

The toolkit used in the implementation of AniComp, namely Cosmo3D, is also an object-oriented class library. Cosmo3D is essentially a scene tree management toolkit but also contains classes to incorporate nodes for time and event processing into the system. The philosophy on which this processing relies is greatly effected by the internal execution model of VRML 2.0 (Virtual Reality Modeling Language version 2.0) specification [vrm98]. It, however, lacks a sound mechanism to accomodate constraints and rules to govern the interaction between objects and time/event management.

3.6 Real Time Animation and Virtual Environments

3.6.1 Understanding Real-Time Animation

Real-time animation implies that the images are being generated at a fast enough rate to produce the perception of persistence of motion, which is, on average, 24 frames per second. To understand the process by which the real-time animation is realized, it is important to distinguish **motion control** from **rendering**. Motion control refers to the generation of the motion sequences by some method (simulation, scripting, keyframing and interpolation etc.). Rendering refers to the process of producing a two dimensional picture (onto the computer display) from three dimensional data combined with other elements such as camera, lights, surface color and characteristics. In one possible scenario, the motion control might be sophisticated simulations of physical processes that require high performance, special hardware. But at the end of the calculations, a series of transformations for objects over multiple times is produced, which the display hardware can read in and render in real time. At the other extreme, simple motion control such as linear interpolation may be used in conjunction with a software ray tracer for the rendering. In this case, the motion calculations may be able to run at real-time rates but the renderer can't produce images in real-time, because ray tracing requires intense calculations despite the photo-realistic images it creates. Both scenarios limits the possibility of real-time animation. For real-time animation to be possible, it is necessary to guarantee a reasonably short amount of time for motion computations as well as a reasonable level both for total polygon count in the scene and the quality of the rendering. These factors however are determined by the capabilities of the underlying hardware and the methods used for software algorithms.

The most well known examples of real-time animation applications are 3D games. Virtual environments are other prominent example where real-time 3D animation is a must to give the impression of a lively environment. The next section discusses virtual environments further.

3.6.2 Virtual Environments

Virtual Environments are computer generated 3D environments which provide the effect of *immersion* and where objects have spatial presence. The feeling of immersion, and non-conventional modes of user-interaction are considered to be important distinguishing factors of virtual environments compared to conventional 3D computer graphics applications, by many researchers [Bry95, Gig93, Fur95, Ell95]. Gigante [Gig93] lists seven technologies that he considers to be enabling technologies for virtual environments. These are:

1. Real-time 3D computer graphics
2. Wide-angle stereoscopic displays
3. Viewer (head) tracking
4. Hand and gesture tracking
5. Binaural sound
6. Haptic feedback
7. Voice input/output

He states that the first three of these are mandatory, the fourth is conventionally used but under some circumstances may not be necessary, and items five and six are becoming increasingly important to the researchers in the field. The last item, he continues to state, while of immense utility, is probably not a key technology that characterizes Virtual Reality. The reader is referred to [Gig93] for a detailed discussion of the listed items.

A somewhat cautious comment on virtual environments comes from Bryson [Bry95]. He states that it has become accepted wisdom that the expected success of virtual reality applications has, with a few notable exceptions, largely failed to materialize. Being cautious he establishes some important guidelines for designing successful VR (Virtual Reality) systems. He underlines the critical role of constructing a good metaphor for the virtual environment in an application, as this metaphor would determine the appearance and behaviour of the environment as well as the way user interacts with that environment. He focuses on three levels of metaphors:

- Overall environment metaphor: The metaphor which determines the overall appearance of the environment including the types of application objects which appear in the environment. This metaphor will also impact the types of behaviours in the environment.
- Information presentation metaphor: The metaphor for how information about the environment is presented to the user.
- Interaction metaphor: The metaphor for how the user interacts with the environment.

This approach suggests that every VR application may have distinct metaphors intrinsic to it and if this is the case *these* metaphors should be used in the implementation rather than applying some standard metaphors. Thus, for example, an application dedicated to training for a real-world task should mimic the real-world environment of that task as closely as possible, whereas an environment for information visualization should use the language of the field from which that information derives.

Animation in Virtual Environments

Virtual environments are populated by many objects. While some of these objects are static, such as walls, windows, streets and other terrain, many others have to exhibit certain behaviours, if a sense of reality is to be achieved. This requires animation. Conventional animation techniques, such as keyframing or to some extent kinematics, however, are not, always suitable in a highly interactive environment. As the user interacts with objects, or objects interact with each other it is most likely that unpredictable situations will arise. Therefore pre-computed motion sequences may result in very unrealistic animations such as objects moving through each other or an object coming to rest at a location where it has no visible support. The use of dynamics provides an answer to this type of problems. Dynamics produces physically realistic motion and allows for user and object interactions to be modeled without worrying about the time and location of a particular event. It is however a computationally expensive method as mentioned in Section 3.4.1. The computational needs for a complex environment containing many simulated objects may quickly overwhelm the processing capacity of many hardware platforms. Although there are promising attempts to develop mechanisms for managing growing complexity, such as that of Chenney and Forsyth's [Che97] *dynamics culling* approach, their maturity is yet to be seen.

3.6.3 Distributed Virtual Environments

Virtual Environments(VEs) can in addition be multi-user, supporting multiple interacting users, and distributed, running on several computers connected by a network. It has become common to refer to a VE with both these additional properties as a DVE (Distributed Virtual Environment). Greenhalgh [Gre96] states that VR systems which are capable of bringing together geographically dispersed users in a single virtual world must necessarily be distributed systems. Some VR systems are distributed, principally to take advantage of parallel computation. VR emphasizes interactivity, both between multiple users and between users and the elements of the virtual world. This interactivity, and the usefulness of highly populated virtual worlds, make VR an ideal driving application for developing and investigating large, multi-user, highly interactive distributed systems.

The first large-scale DVE was the military simulation system SIMNET [Cal93] which networked various simulators (mainly tanks) using dedicated networks. The success of SIMNET triggered a standards push in the form of Distributed Interactive Simulation [IEE93]. In spite of its success, problems with DIS, such as its limited scalability and the rigidity caused by embedding its application in its architecture, have made the military community feel the need to move on. To support its wider goals for modelling and simulation, the US Department of Defense started the High-Level Architecture [HLA] initiative, to define an open framework for the interoperability of heterogeneous simulations [Mil96].

Some other systems worthy of note are DIVE (Distributed Interactive Virtual Environment) from the Swedish Institute of Computer Science [Car93], Spline from Mitsubishi Electric Research Laboratories [Wat97], and AVIARY [Wes93]. Both DIVE and AVIARY utilise a layered approach to DVE support, DIVE being originally based on a distributed environment called ISIS and Spline on its own Interactive Sharing Transfer Protocol. AVIARY concerns itself more with the distribution of data and computation in a heterogeneous network.

3.7 Summary

This chapter provided a survey of animation techniques highlighting the differences between the interactive techniques and the procedural ones, and focusing more on automatic motion generation. A review of real-time animation in the context of virtual environments has also been provided. By doing that, this chapter aimed to clarify the underlying concepts regarding the generation of animations via external procedures, and thus to bring about an understanding of the requirements for integrating an animation module to simulation engines that model behaviour of the geometric objects involved.

Chapter 4

Integrating Simulation and Animation

4.1 Introduction

The use of computer simulation is increasing because of the benefits it provides. As a model evaluation tool it is more powerful and more compelling than any other tool available to date [Ker97]. For the real (physical) system which does not exist, simulations can be performed to predict its behaviour and help us to decide whether the system can be built or not, or to modify it before it is built. Furthermore, the system can be studied using different time frames such as compressed time to speed up a study, or expanded time to observe the details of a study. Computer simulations have the ability to work on manageable time and spatial scales, where the time scale in the real experiment is too long or too short (e.g. movement of planets, galaxies and fast chemical or nuclear reactions). In addition, by using simulation it is possible to create a model and accompanying animation that truly represents an existing or proposed facility. The use of animation will help to communicate ideas, sell projects and build confidence in recommendations.

Given these benefits and ongoing improvements in both hardware and software technologies, it is natural to expect the use simulation software to be even more widespread. Recalling the issues introduced in Chapter 3, the advance of virtual reality applications suggests the need for the integration of simulation software to animation and visualization software to create high fidelity virtual environments. From a system architectural point of view, however, there is much yet to be done to construct a well defined framework which enables seamless integration of simulation engines into graphics and animation software. To remind to the reader, this thesis aims to contribute to the realization of such a framework by attempting to identify interaction patterns occurring between the simulation and visualization software modules. Building towards this aim this chapter attempts to sketch a picture of how the simulation software packages work, and what do they produce as output, that can be related to the creation of the visual behaviour of the simulated entities. Before examining the internals of the simulation modeling approaches, an overview of the efforts for developing component-oriented simulation software is provided. This is followed by a very brief history of the evolution of the use of animation in conjunction with simulation activities, which is then followed by a discussion of simulation modeling approaches aiming to clarify the effect of different simulation modeling techniques to the simulation-animation interaction patterns. Finally a top level model is developed to capture the main interaction patterns stemming from the inputs and outputs of the simulation components.

4.2 Component-Based Simulation

The basic elements of a modelling and simulation environment include a (visual) modelling tool with a graphical model editor, a simulator, model databases for storing model libraries [Lee97], and tools for output analysis, visualization and animation. See [Ada98] and [Hes95] for efforts that aim to establish general interfaces for implementation of these elements as components.

4.2.1 Progress Towards Simulation with Components

The importance of modular model construction is well known in the simulation community. Most of the modeling and simulation languages have special program modules for describing subsystems (called submodels, blocks, segments, etc.) to be used as building blocks of more complex system models. Thus, libraries of frequently used submodels are built for many different application domains, by means of which model construction becomes easier and less error prone. Modern languages also allow hierarchical modeling, i.e. submodels can contain smaller submodels. Some issues in simulation where further improvement is needed are highly related to the component-oriented programming research. Specifically, these include being able to simulate complex systems with parts from different producers, and being able to exchange models between different tools. Since there are no well-established standards for component implementation and interoperation, the simulation components produced (mostly submodel libraries) are confined to a particular modeling language or environment. This situation restricts the re-usability of components and obstructs the realization of a software market of independently produced simulation components. Despite the availability of a great number of simulation tools (specialized for different application areas, such as electronic or logical circuits, mechanical or hydraulic systems, chemical processes, control systems, etc.), in most cases it is not possible to exchange models between them. So, it is hardly possible to change to another simulator without generating all the custom libraries again, or to use a model with different tools. Nor is it possible to easily perform mixed-discipline simulation. It is extremely difficult to use different special-purpose tools to model different parts of a complex system and then to merge the parts to make a model of the total system [Mat93].

The simulation community is aware of the problems, and indeed, there are efforts for development of a standard platform for component implementation and interconnection. One approach suggests standardization of modeling languages to serve as an exchange medium between various simulation environments. Models and submodels whose interfaces and behaviour are defined in a standard, uniform modelling language can be used in all the environments that support the standard. Recently two domain-independent modeling languages, namely Modelica [Mat98] and VHDL-AMS [VHD], are being designed in parallel, with standardization in mind. One drawback of this approach can be seen easily: the model definitions have to be transformed from the standard language to the native language of an environment for usage. So, if a translator is not available, the components cannot be used. Furthermore, since the components are provided in a high-level modeling language, their providers cannot contribute to the implementation of the component's binary form, i.e. they cannot place model-specific code (related to integration, equation solution, parallelism, etc.) in the implementation, whereas being able to do so could improve the speed significantly. Another major draw back is that, the components provided this way are at source-level, thus are open to a white-box usage, the risks of which have been stated in Chapter 2. An alternative standardization effort, namely the DSblock approach [Ott98], uses general-purpose programming languages such as FORTRAN, C or C++ to define standard interfaces for models or submodels. Implementations of the interfaces are called DSblocks. DSblocks, either at source-level (in the same language as its interface), or possibly compiled to a binary form, can be linked together for simulation. Apparently these components do not need transformation. Modeling and simulation environments are expected to be able to produce the DSblock equivalents of the models or submodels written in their custom language. Of course, it is also possible to code a DSblock-compatible model directly in a given programming language. There are other types of binary components, for example the segments of ESL [Hay92] and S-functions of SIMULINK. These differ from DSblocks in that, in this case the interfaces are mapped to a modeling language, so the components are treated as submodels at the modeling level; and their implementations are linked to the code generated by the modeling and simulation environment. A third standardization effort of interest is the High Level Architecture (HLA), introduced earlier in Chapter 3 Section 3.6.3. HLA aims at distributed simulation with programming language independent binary components. A major advantage of components being binary is the enforcement of a black-box usage. Also, there is no need for their re-compilation, and the component providers can optimize the implementations for the best performance.

Parallel developments can also be observed in the discrete-event simulation community. The most notable effort to develop a standard approach to discrete-event modeling which addresses the needs of a component-oriented philosophy, is that of DEVS formalism [Zei90]. DEVS is a set-theoretic formalism [Ahn94] within which one must specify 1) the basic models, from which larger ones are built, and 2) how these models are connected together in hierarchical manner. The basic models are called *atomic* models and compound (complex) models are called *coupled* models and they tell how to couple (connect) several component models together to form a new model. Every DEVS model is associated with an abstract processor which is capable of interpreting the dynamics of the model [Kim94]. The job of an abstract simulator (processor) for an atomic model is to schedule the next event time and request the associated atomic model to execute its transition functions and output function timely. The processors for coupled DEVS models are called coordinators and they are responsible for synchronizing the component simulators for scheduling the next event time and to route external event messages to component simulators. The overall simulation is managed by a root-coordinator.

These efforts help to analyse the simulation domain from a component-based software construction point of view [Alp98]. The next section provides an introduction to the issue of integrating simulation with animation and visualization.

4.3 A Brief History of Integration Between Simulation and Animation

In the early 70s simulation and animation first came together [Ear94], and the simulated process has been presented on computer screens. Before animation was available, simulation study results were only a list of numbers that had to be graphed and interpreted by the analyst. This was hardly illuminating. Program code and output were virtually unintelligible to management. A major improvement came from custom output reports: results of the simulation could be sent to the computer screen or to a data file. However, the simulation itself retained a black box aura. Now, however, with animation, the end-user can "see" what is happening. Each time the simulation is re-run with different data, a different animation will take place.

In the last ten years, all major simulation systems have contained animation modules or have been coupled with animation systems. As a result, computer animation is now an integral part of many simulation projects. The animated dynamic displays representing the system under investigation provide an important common interface between client, analyst, designer and modeler. Animation plays an important role in the process of creating and using simulation models. Teo [Teo93] identifies the principal parts of a simulation project where animation offers assistance:

- model verification and validation.
- communication and marketing.
- improvement in system understanding.
- presentation of model results.
- education and training.
- better model debugging.

- credibility of model.
- bottleneck analysis.

Given these potential benefits it is more likely to observe a growing interest in integrating animation techniques into simulation exercises. The method of combining animation and simulation can take a number of forms, which are introduced in the next section.

4.3.1 Use of Animation in Simulation Exercises

The use of animation in simulation systems has been experienced mainly in three ways:

1. In a relatively old technique, computer animations are created, through the use of a file generated by the simulation program only when the simulation is finished. This file, called the trace file, is then used to drive a layout file to produce the animation; an approach called post-processing animation. Post-processing animation requires animation packages that are capable of analyzing and understanding the trace file. This technique was mainly used (and is still being used) to generate iconic animation of simulation entities super-imposed on a 2.5 dimensional layout, in discrete-event simulation software. It is however possible to use the trace files to create high quality three dimensional animation using a suitable animation package.

2. The person creating the animation basically uses the current simulation data, presenting the animation while the simulation is taking place. This approach is called concurrent animation. As the simulator runs it creates the data that can be mapped to the motion of the geometric objects representing the entities in the simulation in real time. There is no interaction of the user with the animation or the simulation during the simulation run. Depending on the scene complexity and also on the complexity of the individual geometric object high or medium quality 3D animation can be generated.

3. The third technique, called interactive visual simulation, allows interactions during the execution of a model and modification of the model, in

order to display the effects caused by these changes immediately. An extension to this technique is to allow the interaction of the user not only with the underlying simulation model but also with the environment created by the simulation model as a direct inhabitant of the simulated (virtual) environment. This concept leads to the fusion of the visions of the virtual reality community and simulation community suggesting virtual environments with meaningful scenarios and being inhabited with more intelligent objects(actors).

Table 4.1: A classification of animation systems in relation to their use with simulation software

Animation Environment	Graphics Complexity	Level of involvement (Immersion)	Level of Interaction
Layout and Iconic Animation	2D or 2.5 D objs. Low complexity	None	None/Limited
Recorded 3D video	3D Objects high complexity	Limited	None
Interactive 3D Animation	3D Objects/ Medium/High Complexity	Limited	Limited/High
Virtual Environments	3D Objects Medium Complexity	high	high

These alternative ways of using animation in simulation applications, listed above, are summarized in Table 4.3.1. The first item in the list is represented in the first and second rows of the table; the second item in the list, correspond to the second row of the table; and the third item in the list describes the second and third rows of the table.

4.4 Approaches to Simulation Modeling and Their Impact on Simulator-Animator Interaction

Simulation modeling can broadly be classified as *continuous time*, *discrete event* and *combined simulation* approaches. It is desirable to have a general understanding of the modeling principles employed by these approaches as well as the execution logic of the corresponding simulation software to identify the possible interaction patterns that can potentially arise during the integration of simulation and animation components. The following sections introduce the three basic approach mentioned above. First, most common methods of modeling continuous-time models are discussed, which are also employed in this thesis during the modeling of example systems: the Ball and Beam apparatus and the Inverted Pendulum apparatus (these are discussed in Chapter 6). Then basic mechanisms of a discrete-event simulation system are described. Finally the combined simulation approach is described. Particular attention is devoted to the identification of when these simulation models are likely to enter into an interaction with a visualization component and in what form.

4.4.1 Continuous Simulation Modeling

Simulation models involving mathematical descriptions based on differential equations are called **continuous-variable** models. One of the most fundamental divisions in continuous-variable models, in terms of mathematics, is between **linear** and **non-linear** system models. In a linear system model responses are additive in their effect and the output is directly proportional to the input whereas this is not true for a non-linear model. **Time invariance** is another important property which can provide a basis for the classification of models of systems. A time-invariant system is one in which the observed performance of the system is independent of the times at which the observations are made[Mur95]. Models which are both linear and time-invariant are particularly important and receive considerable attention in the fields such as systems engineering, automatic control, electrical circuit analysis and applied mechanics. This is because many practical systems do have properties which may be approximated in a satisfactory way by linear time-invariant descriptions, eliminating the difficulties presented by non-linear or time-varying system descriptions.

Model Description

Many different forms of mathematical description are possible for the representation of a given dynamic system. Two of the most widely used are the *reduced form* and the *state-variable form*. A well-known example[Mur95, War96, Phi00] of a simple mechanical system is illustrated here to form a basis for an understanding

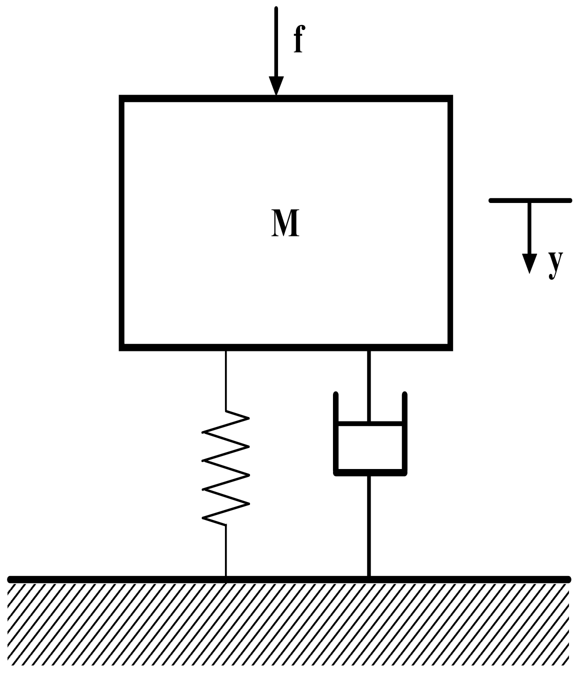


Figure 4.1: A Mechanical System Consisting of Mass, Spring and Viscous Resistance Elements

of the output of a continuous time simulator, in relation to simulator-animator interaction. First the reduced form representation is given, then state-variable form is shown. The example mechanical system is illustrated in Figure 4.1 and the corresponding mathematical model is given in equation 4.1.

$$M\ddot{y} + R\dot{y} + Ky = f(t) \tag{4.1}$$

Equation 4.1 is the differential equation involving the displacement, $y(t)$, of a mass M suspended by means of a linear spring of stiffness K and damping element having viscous resistance R. The mass is subjected to an external force $f(t)$. The corresponding description in state-variable form involves two first order equations in place of the single second order equation.

$$\dot{x}_1 = x_2 \tag{4.2}$$

$$\dot{x}_2 = -\frac{K}{M}x_1 - \frac{R}{M}x_2 + \frac{f(t)}{M} \tag{4.3}$$

In equations 4.2 and 4.2, x_1 is the displacement, $y(t)$, and x_2 is the velocity, dy/dt. These new variables are known as the state variables of the system.[1]. These two equations can be rewritten as a single-vector matrix equation as follows:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{K}{M} & -\frac{R}{M} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{M} \end{bmatrix} f(t) \tag{4.4}$$

Or more concisely:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{b}u \tag{4.5}$$

Where $\mathbf{x}$ is the state vector, $\mathbf{A}$ is a 2x2 square matrix and $\mathbf{b}$ is a two-element column vector. This equation relates the rate of change of the state to the present state and the input. It is a form which is found to be very convenient for simulation since the numerical solution can be obtained for each equation within the state-space model simply by a process of integration. Another form of continuous

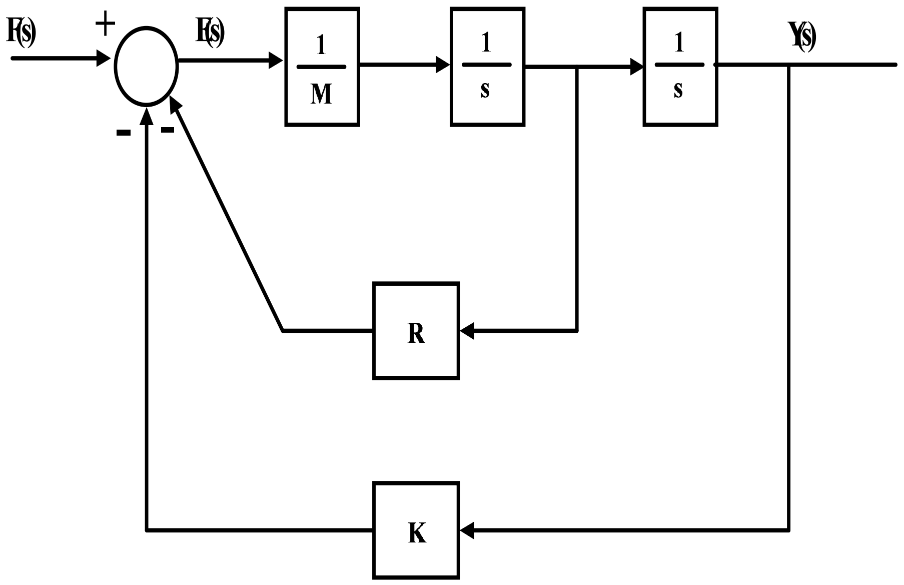


Figure 4.2: The Block Diagram of the Mechanical System Consisting of Mass, Spring and Viscous Resistance Elements

time system representation is the transfer functions, defined as the ratio of the

[1]In general an nth order description in reduced form is equivalent to a set of n first order ordinary differential equations in state-variable form. See[Mur95] and [Phi00] for a description of how this conversion can be done.

Laplace transform[Jor98] of the model output variable to the Laplace transform of the input when all initial conditions are zero[Mur95]. This form of description is particularly widely used in analysis and design of engineering control systems. It is also a convenient description for the derivation of *block diagrams* which are widely used by many continuous time simulation packages. As an example, the transfer function for the simple mechanical system illustrated in Figure 4.1 is given as:

$$\frac{Y(s)}{F(s)} = \frac{1}{Ms^2 + Rs + K} \tag{4.6}$$

This equation is obtained from equation 4.1 by assuming that all initial conditions are zero and replacing d/dt by s, d^2/dt^2 by s^2 and so on for higher derivatives, to get an algebraic form in Laplace variable s as follows:

$$Ms^2Y(s) + RsY(s) + KY(s) = F(s) \tag{4.7}$$

Manipulating this equation algebraically and re-arranging gives equation 4.6. The block diagram representation of this transfer function can then be derived as in Figure 4.2.

Impact on the Simulation-Animation Interaction

From the animation point of view, the simulation described by the block diagram given in Figure 4.2 produces the amount of incremental translation of the mass M in time, along the direction of force applied on it. Feeding this output parameter to the animation component from within a simulation loop generates the visual behaviour of the body M. This simple case can be extended for more complex models. If the system, hence the mathematical model, is more complex, then the number of output parameters can be higher. If, for example, the mechanical system involves several rigid bodies connected by different types of joints, then the resulting mathematical description would consist of many equations that have to be solved simultaneously. The output of the simulation of such a system might produce position and orientation information for each of the bodies depending on the number of degrees of freedom allowed by the overall configuration. This physical information can then be dispatched to a graphics module to generate the desired motion of the geometric objects. This is summarized in Figure 4.3

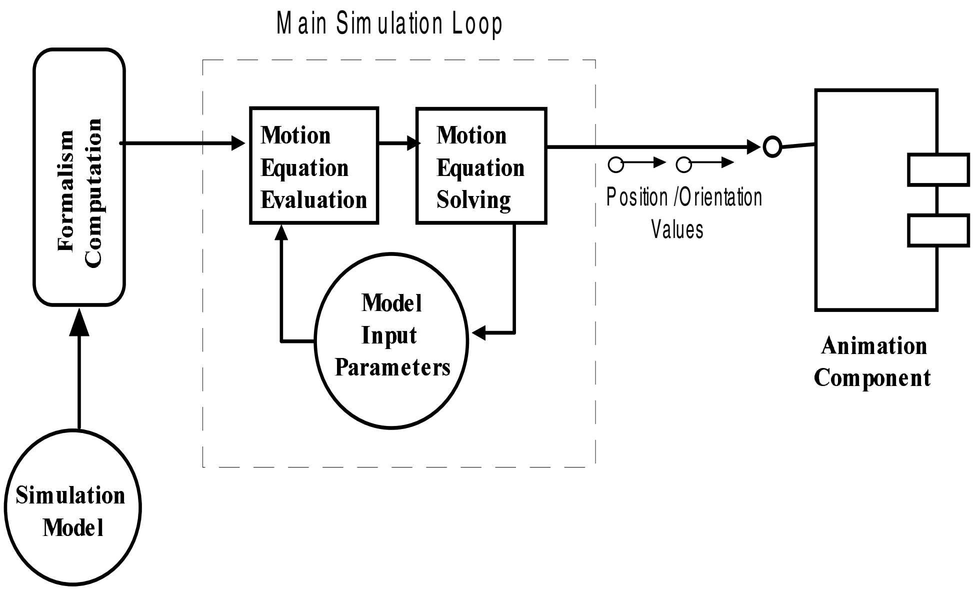


Figure 4.3: The Main Execution Logic of a Continuous Simulation Module for Motion Computation

where the main simulation loop involves three main tasks of evaluating the motion equations, solving the motion equations and incorporating the model input parameters to these computations. If this loop is a purely numerical loop, the equation evaluation task must be performed once every cycle of the loop, whereas in a symbolic computation scheme this task is performed only once to produce a multi-tree representation whose nodes are arithmetical operators and leaves are the parameters and constants. For a detailed example of the realization of such an architecture applied to a complex multi-body simulation problem, see[Heg96].

4.4.2 Discrete Simulation Modeling

The goal of a discrete simulation model is to portray the *activities* in which the *entities* engage and thereby learn something about dynamic behaviour and performance potential of the system. The problem areas that discrete event simulation deals with can generically be described as *queuing* problems. These problems have one thing in common: they are concerned with the delays experienced by objects while waiting to be processed within a system. Another goal of this kind of analysis is to try to asses the degree of utilization of expensive resources and facilities that carry out the processing within the system. The objects or components of a discrete system model are usually known as *entities*.

There are several different world views that a discrete simulation system can be based on. Most researchers agree on three distinct modeling views: Event-Oriented View, Process-Interaction Oriented View and Activity Scanning view[Ben98, Ban98, Pri95, Pri98]. Each of these are explained below briefly:

1. **Process-Interaction View:** According to this view the computer program should emulate the flow of an object through the system. The *entity* moves as far as possible in the system until it is delayed, becomes engaged in an activity, or exits from the system. When the entity's movement is halted, the clock advances to the time of the next movement of any entity. During their movements, all changes, delays, waiting and working times incurred by these entities are recorded.

2. **Event Oriented View:** In the event oriented view, a system is modeled by defining the changes that occur at event times. The task of the modeler is to determine the events that can change the state of the system and then to develop the logic associated with each event type. A simulation of the system is produced by executing the logic associated with each event in a time-ordered sequence.

3. **Activity Scanning View:** In this approach the modeler describes the activities in which the entities in the system engage and prescribes the conditions which cause an activity to start or end. The events which start or end the activity are not scheduled by the modeler, but are initiated from the conditions specified for the activity. As simulated time is advanced, the conditions for either starting or ending an activity are scanned. To ensure that each activity is accounted for, it is necessary to scan the entire set of activities at each time advance, which leads to a relative inefficiency compared to other approaches mentioned above.

Whatever the approach, the conceptual framework of a discrete event model seems to be sketched by the relationships between the notions of activity, event, process and entity. From the point of view of this thesis, it is important to capture these relationships and to have a general understanding of the processing logic of a discrete event simulator in order to identify possible points where it is most likely to trigger the activation of a motion sequence. These points are also natural candidates for initiating an interaction with the animation component in a component-based simulation environment. Figure 4.4 shows the timing

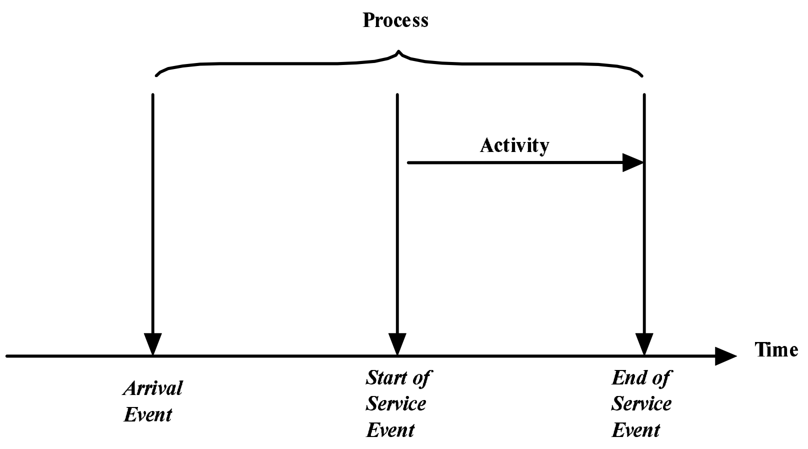


Figure 4.4: Relationship between the notions of activity, event, and process in a Discrete Event Model

relationships between the concepts of activity, event and process. As it can be seen from the figure, an activity spans a period of the simulation time delimited by a starting and an ending event. An event takes place at an isolated point in time. A process is a time ordered sequence of events and may encompass several activities. Figure 4.5 (see next page) illustrates the general execution logic of a discrete-event simulation software. The flow chart shows a single replication in a simulation exercise. Among the four main phases shown, the *entity movement phase* and the *clock update phase* are of particular interest here since their repetitive execution constitute the main loop where the behaviour of the system is specified. In the *entity movement phase* all qualifying entities take whatever action they can at the current simulated time. Some of these actions can directly be mapped to a motion sequence in the visualization module to simulate the movement of the geometric representation of the corresponding entity in a virtual environment. Once it is decided that an action of the entity must be visualized as an animation then a convenient animation technique can be applied to the geometric representation of the entity. In Section 4.4.4 a number of suggestions are made for establishing guidelines for selecting an appropriate animation technique.

The purpose of the *clock update phase*, on the other hand, is to set the clock

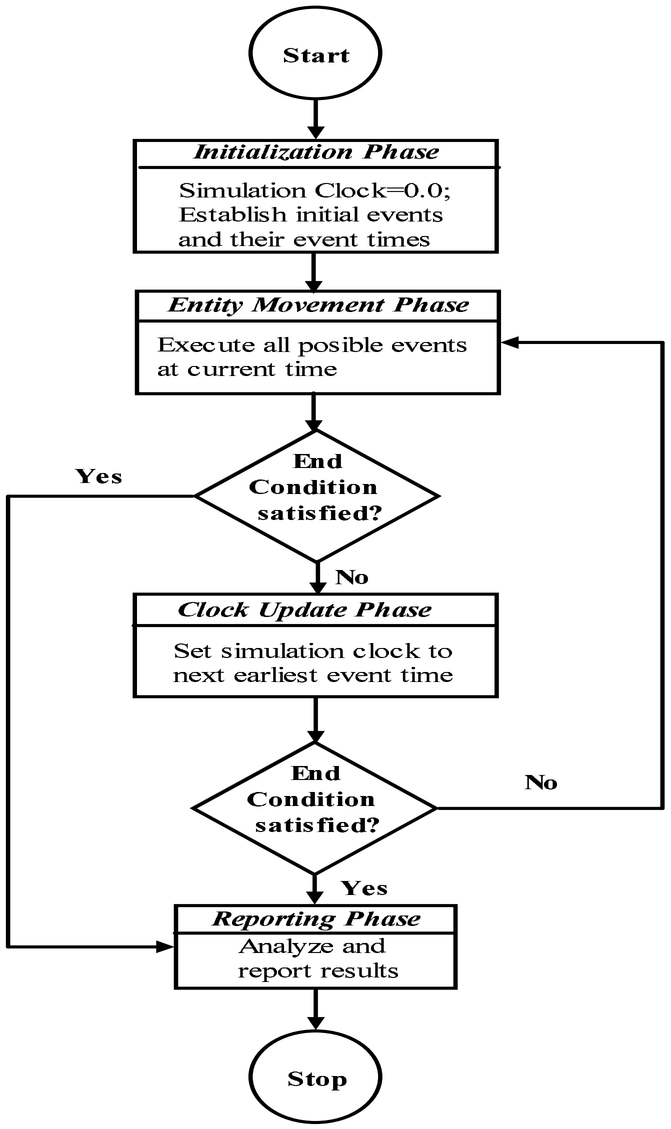

Start
Initialization Phase
Simulation Clock=0.0; Establish initial events and their event times
Entity Movement Phase
Execute all posible events at current time
End Condition satisfied?
Yes
No
Clock Update Phase
Set simulation clock to next earliest event time
End Condition satisfied?
No
Yes
Reporting Phase
Analyze and report results
Stop

Figure 4.5: A Typical Discrete Event Simulation Loop

to the next earliest simulated time at which one or more actions have been scheduled. This *next earliest simulated time* might be the current simulated time, depending on whether two or more entities have been scheduled to act at the same simulated time *and* depending also on the implementation choices made by language designers. Alternatively, this next earliest simulated time might be a *later* point in simulated time. Within the main loop described in Figure 4.5

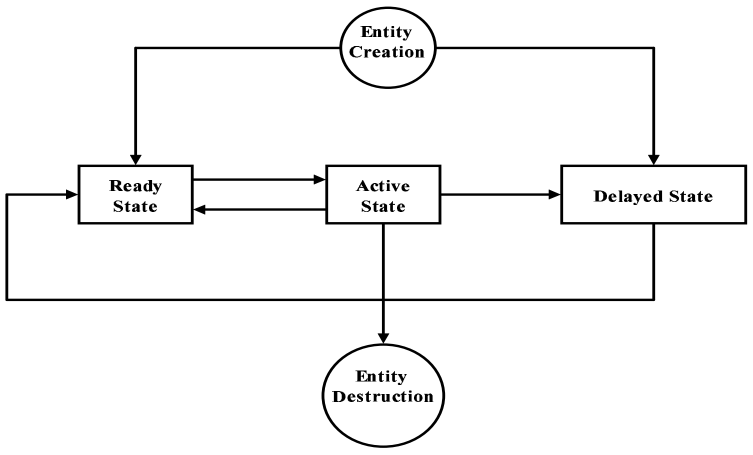


Figure 4.6: The Possible States of An Entity During Its Lifetime

entities of the system go through several states. During its life cycle an entity migrates from state to state, usually passing through these states several times before it is destroyed. There are three main state that an entity can engage. These are *ready state, active state* and *delayed state.* A state transition diagram of possible transitions between these states is given in Figure 4.6.

The Impact on Simulation-Animation Interaction

There are several important points to be considered when determining the communication patterns between a discrete-event simulator and an animation component.

- In a discrete-event simulation when entities are in an active state they are not considered to consume simulation time. Simulation time advances only when entities are being served by other entities (usually called resource entities) which puts them into the delayed state. For the animation, however, this is not realistic. Animation time is continuous. In a 3D animation environment, entities, whether they are serving or being served, may need to be animated even if the simulation time remains fixed. As an example, think of a customer (i.e an entity) in a bank who is waiting in a queue. When a teller becomes idle, the customer becomes active to seize the teller. This instantaneous process consumes no time in the discrete-event simulation. But in a 3D virtual environment it is not realistic to *fly* the customer to the till. Instead the customer should *walk* a certain amount to reach to the till, which would typically require a combination of kinematics and key-framing techniques to animate the walk and would also take some animation time. This has to be accounted for when controlling the animations with a discrete-event simulator.

- The fact that more than one entity can be scheduled to be active in the same simulated time raises some interesting situations concerning the timing of the corresponding animations. As it can be seen from Figure 4.7, in a

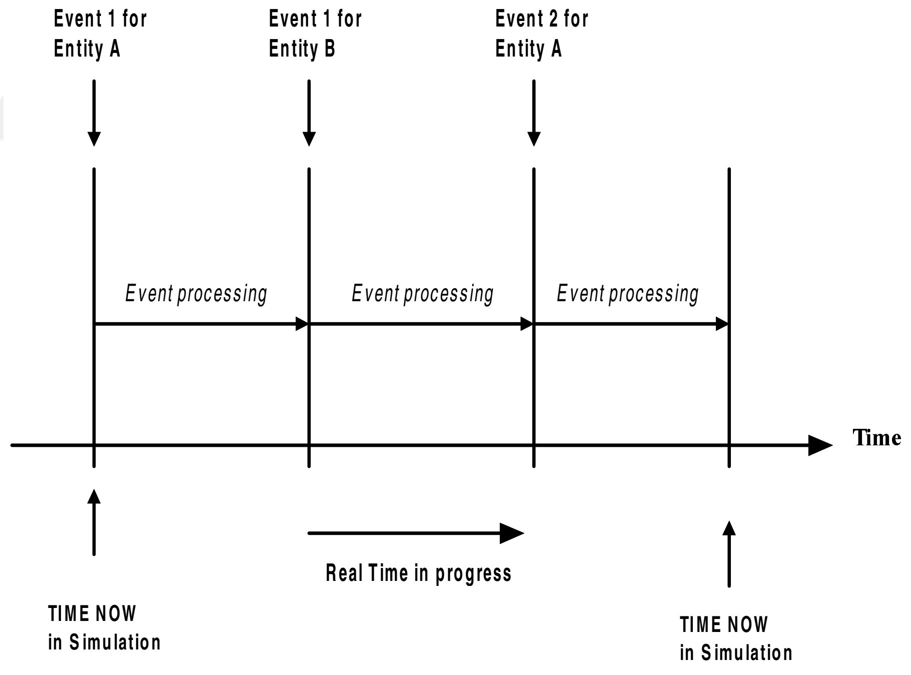


Figure 4.7: Events Occuring At the Same Simulated Time

period where more then one event are handled, although the real time continues to advance, the simulated time remains fixed. If the events of such nature are related to the same entity *and both* require the visualization of distinct motion sequences, then this would result in discontinuities in or abnormal termination of their motion. This is because the initiation of the next motion sequence at the same time, may prevent the previous animation from ending gracefully. This problem can be resolved by either modeling the scenario in such a way that simultaneous events are replaced by *activities* which ensures that they consume some simulation time, or by obeying to a protocol which guarantees that the motion corresponding to the next simultaneous-consequitive event is not started before receiving a notification of completion of the previous one.

- If the animation is being used as a tool for model validation and verification rather than as a sole presentation medium, then it is necessary to make a transformation of the discrete-simulation time into a pseudo-continuous time. Hill [Hil96] states that as the time scale taken for the simulation allows time intervals during which no graphics event occurs, it is fundamental to effect timing with the current time scale when nothing has to be animated. He gives the case of a manufacturing workshop as an example and argues that if a shuttle is inactive for a quarter of an hour, the animator must correctly time this phenomenon and must not jump to the next date of occurrence of an animation-triggering event. This is indeed important as it enables the observer to notice that perhaps the system is over-dimensioned, or other anomalies in the model. Despite the advantage of reflecting the simulation time scale to the animation in such cases, however, this would be impractical in situations where the simulation study covers a long periods of time, such as days, weeks or even more. The simulation-animation interaction scheme, therefore, should facilitate the implementation of both strategies.

4.4.3 Combined Modeling Approach

In combined discrete-continuous models, dependent variables may change both disceretly and continuously. The world view of a combined model specifies that the system can be described in terms of entities, their associated attributes, and state variables. The behaviour of the system is simulated by computing the values of the state variables at all time steps and by computing the values of the attributes of entities at event times.

Three fundamental interactions can occur between discretely and continuously changing variables [Pri98]. First a discrete change in value may be made to a continuous variable(e.g. completion of a maintenance operation that instantaneously increases the rate of processing by machines in a system). Second a continuous state variable achieving a threshold value may cause an event to occur or to be scheduled(e.g. the arrival of a material handler to a prescribed position initiates an unloading process). Third the functional description of continuous variables may be changed at discrete time instants (e.g. a human entering into the vicinity of a crane changes the equations governing the acceleration of the crane).

Combined discrete-continuous modeling constitutes a significant advance in the field of simulation[2]. A large number of problems are in reality a combination of discrete and continuous phenomena and should be modeled using a combination of the two approaches. An area where combined modeling approach is proven to be effective is the simulation of the material handling equipments. The modeling of cranes, conveyors and automated guided vehicles involves continuous-time concepts such as acceleration and velocity changes and also discrete requirements associated with loading, unloading, picking up material, moving over network segments etc.

Modeling with combined discrete-continuous approach, however, is not straightforward. Care should be taken when determining interaction and interfaces of the components of the system with different properties. Pritsker [Pri98] suggests that in modeling combined systems, the continuous aspects of the problem should be considered first followed by the development of discrete aspects of the model. And finally the interfaces between discrete and continuous variables should be approached. The issue of modeling systems with the combined approach and studying the interfaces between system components with different modeling properties is not considered in detail in this thesis. It is the impact of such a modeling approach on the simulation/animation interaction which is examined. Therefore, only the issues related to the necessary mechanisms needed to implement the animated visualization of a simulation exercise involving both discrete and continuous properties are considered. The next section looks into this issue.

[2]There are distinct groups within the simulation community for discrete-event and continuous simulation. Traditionally the Summer Computer Simulation Conference is for continuous simulation modelers and the Winter Simulation Conference for discrete-event simulationists (SCS San Diego); only recently have the two groups started to mix.

The Impact on the Simulation-Animation Interaction

The world view of a simulation model involving combined modeling principles translates into the animation domain in the form of a world view where the behaviour of virtual objects are mainly generated by continuous models whereas the transition between different behaviours and the interrelation of the objects are governed by discrete-event models. In a typical simulation exercise, the discrete event model would act as a scenario manager which generates the events based on statistical models (random number generators), or sets of input stimuli or messages, to direct the flow of the entities within the system. The continuous models then, would provide a basis for realistic simulation of the behaviours of the entities of various types. In such a scheme, the bulk of the message traffic concerning the coordination of different simulator components, is likely to be between the discrete event simulator and the continuous simulator engines implementing the objects' behaviour. The animation component, being decoupled from the simulators as much as possible would receive position and orientation updates from the (continuous) behaviour simulators for the entities being animated by these simulators. Additionally, the discrete event simulator might initiate some simple animations by invoking appropriate functions provided by the animator interface. One important interaction between the animator component and the simulators might occur when two animated objects collide. Since the geometric information has to be kept local to the animator component, it is the most suitable place for detecting collisions between objects. Therefore collision events have to be detected and reported to the coordinator simulator (discrete-event simulator) as well as the colliding objects' simulators (continuous simulators) if any.

4.4.4 Developing A Model for Simulation-Animation Interaction

To establish a meaningful timing relationship between simulation and animation components it is important to understand the notion of time in both.

As indicated above in section 4.4.2, in a typical discrete-event simulation program processes evolve in time as a series of events separated by time intervals. An event is a collection of actions which are considered occur instantaneously. Between two events of a process, events of other processes may occur. Parallelism is simulated by executing events of all processes in sequential order. Time is

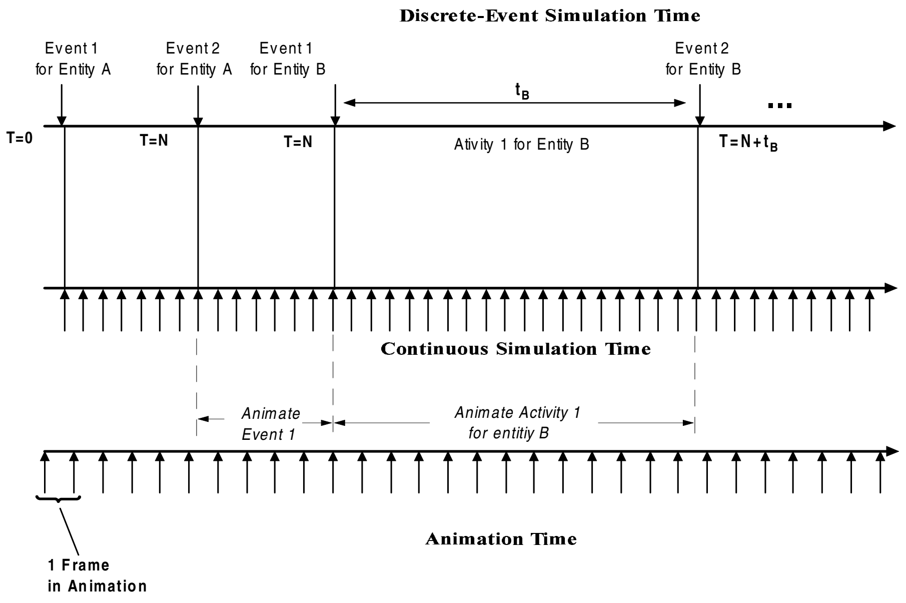


Figure 4.8: Timing Relationships Between a Discrete-Event Simulator, Continuous Time Simulator and An Animator

simulated by advancing it from one event to the next. The simulation is, thus, event driven.

A continuous simulation program, as discussed in section 4.4.1, has no great common points with event-driven simulation programs. Typically it consists of a program to solve simultaneous differential equations to obtain values of state variables of any value of time; continuous simulation is time-driven.

In an animation system, objects are controlled by state variables (position, orientation, color etc.) that drive their motion. Values of the state variables are calculated at each frame, and the animation is frame driven. As frames are equally time-spaced (1/NIPS second; NIPS being the number of frames per second), this can be considered as a special case of time-driven simulation, with a continuous time approach using a time resolution of 1/NIPS second. Animation is said to be *frame-driven.* A pictorial representation of these differences is given in Figure 4.8. The effect of these differences are reflected in the anatomy of the interaction patterns. In the case of a continuous time simulation, the simulation engine generates the position and orientation values as dependent on the sampled time variable. This is naturally the result of numerically integrating the differential equations of motion with a certain step size. This results in high frequency

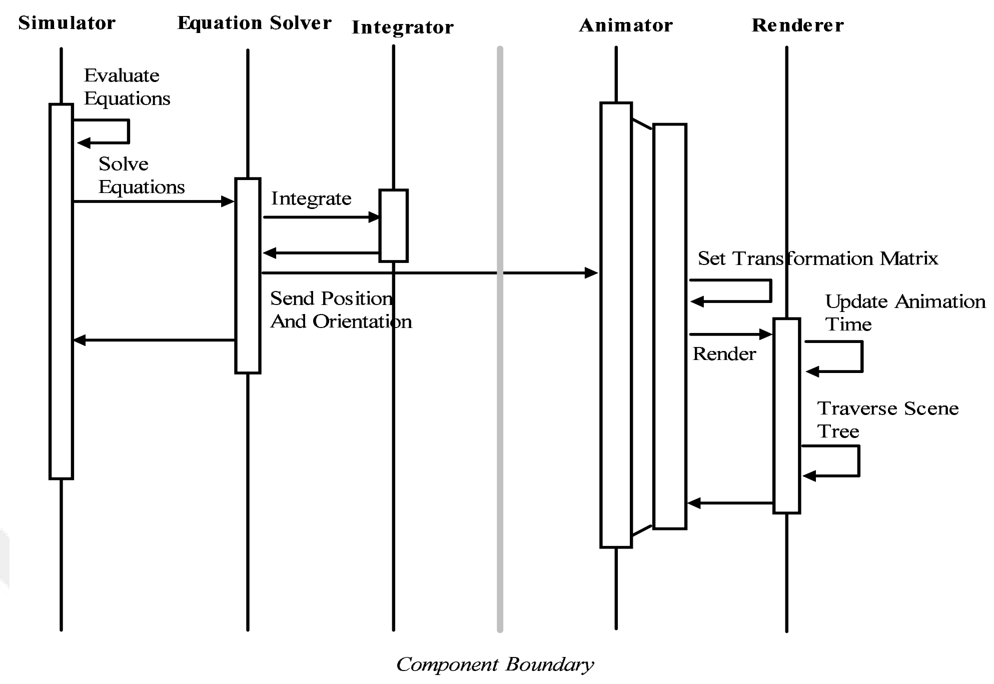


Figure 4.9: Simulator-Animator Interaction for Continuous Time Simulators

calls (frequency depending on the integration step size) to the Animator's interface functions. In this type of interaction, Animator's interface functions at a low level, typically receiving position and orientation vectors, or a transformation matrix containing both. This interaction scheme is illustrated in Figure 4.9. If, however, the object is sufficiently simple and its motion can be defined by its velocity or acceleration or forces that apply to its center of mass, only changing at discrete times, then Animator can employ an internal differential equation solver (integrator) to reduce the number of calls. In this case the simulator would only specify the time at which the quantities such as velocity or acceleration need to change and then call an appropriate interface function which sets values in the data structures internal to the Animator to reflect the change. In fact, a similar technique is employed by DIS (Distributed Interactive Simulation) packages and is known as *dead reckoning.* It is mainly used to reduce the network traffic in a distributed simulation environment. It is, however, useful to reduce the coupling between the simulation and animation modules in a non-distributed environment as well. In the case of a discrete-event simulator, an appropriate technique for the animation of the entities is to associate a set of pre-defined motion sequences to each entity and replay them when the entity takes the relevant action. In that

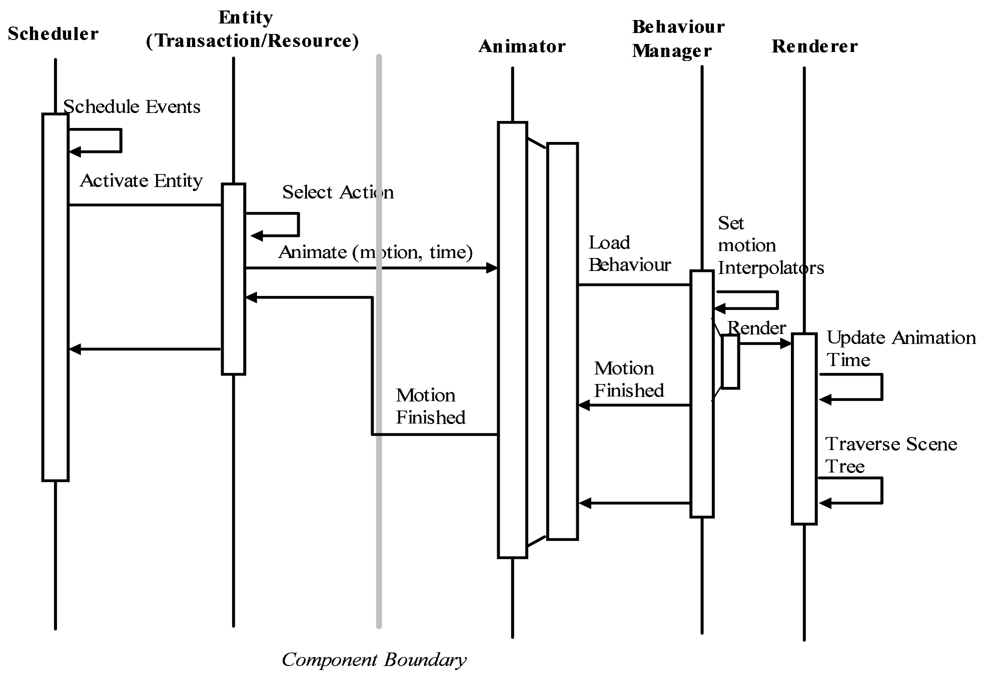


Figure 4.10: Simulator-Animator Interaction for Discrete-Event Simulators

case the motion sequences composing a particular behaviour can be created only once by an underlying dynamic simulation model, or by a human animator using a variety of interactive animation techniques, and recorded. Such recorded behaviours may then form an indexed behaviour library (or repository) from which behaviours can be called and replayed when the entity (or actor) is scheduled to exhibit a particular behaviour. A typical interaction in this case would manifest the pattern depicted in Figure 4.10. This technique is being adopted in many real time games to simulate the behaviours of the actors in the game. It has the potential of significantly reducing the number of interface function calls, and can help to decouple the simulation and the animation modules. It, however, has two major disadvantages:

1. The behaviours must be created and recorded beforehand in a format recognizable by the animator which requires a separate coding effort or use of additional tools.

2. The pre-recorded motion can not respond to unpredictable interaction. If the type of interaction is predictable the response can be created as another distinct behaviour and called from the behaviour library after the

interaction but this may necessitate many behaviours to be created; a process which may cause burden that quickly overwhelm the benefits of the technique. Even if the response is pre-definable it may be hard to compute it realistically as it might be impossible to predict the exact nature of the interaction. Furthermore, for some types of interaction, such as realistic collisions in a virtual environment, this technique is simply impossible to apply since it is not conceivable to know the colliding objects' properties such as mass, colliding vertex, the normal of collision etc. before the event, whereas these are necessary to compute a realistic response.

The first disadvantage is fairly ignorable as it is possible to reduce the effort needed to a minimum by implementing a framework to relieve the user from most of the burden. This can be achieved by defining easy to use, integrated tools for constructing key-frame interpolators, and facilitate loading, saving, composition and activation of the created behaviours. The second disadvantage, on the other hand, limits the use of technique, especially for highly interactive environments. The nature of user interaction with the objects and the inter-object interaction is mostly unpredictable and therefore it is nearly impossible to use a predefined library of behaviours. Despite its limitations, however, this technique is still worth considering for three reasons:

1. First, even in an interactive environment there might be objects which give fixed responses to some type of interactions. For example, a two wing door opens if a human touches a button or approaches the door. In a more complicated case a fixed robot arm, when activated, can pick an object from a specified location and place it to a specified location moving along pre-computed trajectories with pre-computed joint values; and such cases are common especially in discrete event simulation scenarios.

2. Second, it significantly reduces the number of calls required to create a complex motion sequence. This in turn helps to minimize the coupling between the simulation and animation modules.

3. Third, important benefits can be obtained if a re-usable framework can be established to create and re-use the pre-defined behaviours. The producers of a certain engineering product can create a library of animations for a set of behaviours that are thought to characterize the functionality of that

particular product. The developers who wish to involve that product in the virtual environment they are constructing, can then use the behaviour library to invoke a particular behaviour on occurrence of a triggering event. Since invoking a behaviour from the library can be as simple as specifying the object name and the name (or number) of the desired behaviour, this method is extremely convenient. Although this may not be useful for dynamic analysis, it is certainly very beneficial for presentation purposes. In fact, this approach is adopted by many recent real-time 3D game engines. These game engines are usually distributed with a library of pre-created characters which are accompanied with a set of pre-defined distinct behaviours such as a single walking or a running cycle, falling to the ground, jumping, attacking the enemy etc. The game developer then can either load these characters and invoke their behaviours as the user (player) triggers them or can create his/her own library of characters with their customized behaviours to be used in his/her game.

The combined simulation modeling approach integrates continuous and discrete models. The result of this approach from the animation point of view, in its ideal form, is the *realistic behaviour of virtual objects governed by a meaningful and controllable scenario.* In a combined simulation exercise, the behaviours of the individual objects are most likely to be generated by continuous simulation model. The discrete event simulator, then, would act as a scheduler which generates the events based on statistical models, to direct the flow of the entities within the system. However, it is possible to mix and match the technique of reusing pre-defined motion libraries with continuous time simulation for generating the behaviours of the entities in the simulation. The simulation programmer can designate continuous simulation engines for the motion generation of certain entities if their behaviour needs to be more realistic; and he/she can invoke pre-defined motions for other entities where appropriate, leading to a hybrid interaction scheme. A simplified illustration of possible interactions for this type of scenarios, is given in Figure 4.11.

4.5 Summary

Along with summarizing the history of integrating animation with simulation and the trends for component oriented simulation, this chapter provided a study of

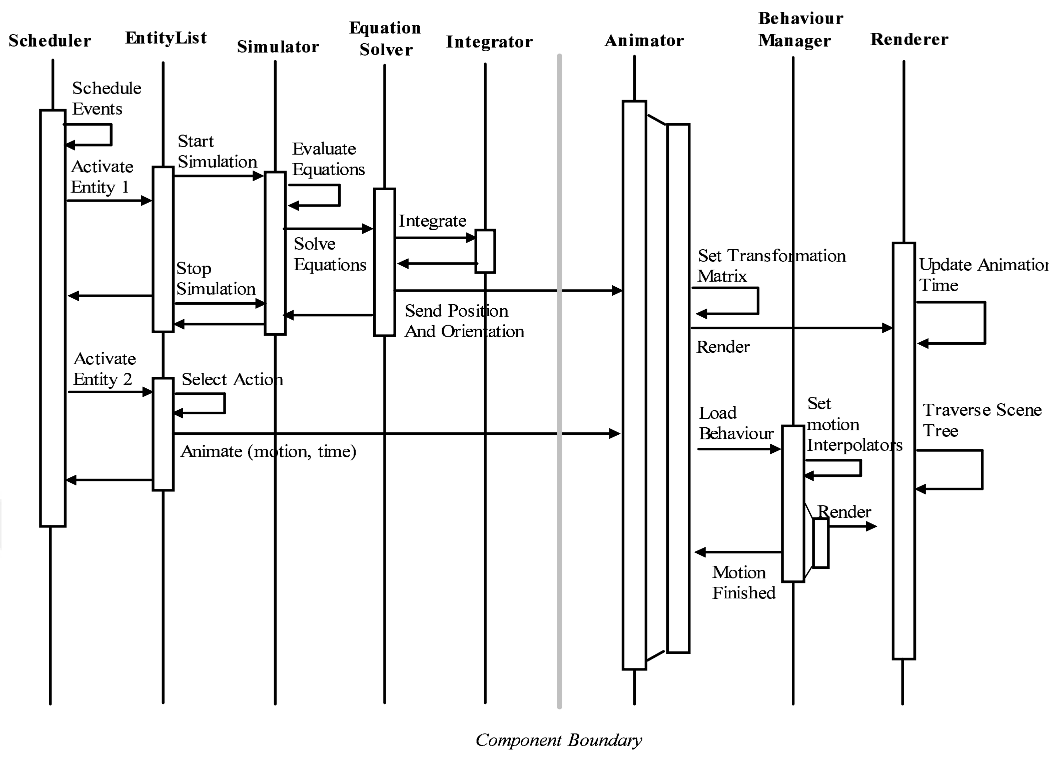


Figure 4.11: Simulator-Animator Interaction for Combined Discrete-Event Continuous Time Simulators

simulation modeling approaches emphasizing their impact on the generation of simulation-animation interaction patterns. An attempt has been made to devise a model for identifying these patterns. This chapter, thus, aimed to supply significant amount of domain knowledge to the design process of the animation component's interface functions.

Chapter 5

Design and Implementation of AniComp

5.1 Introduction

The design process of an off-the-shelf binary component should exhibit some differences compared to the design process of a complete software system. The most important difference is that in a complete system design, the emphasis is typically on the end user's requirements and that all the components of the system are identified and are specifically known at design time. During a component design, however, the emphasis is on the requirements of *other components* that are assumed to collaborate with the component being designed. The component is designed to implement an interface specification possibly published by clients and the designer is not able to exactly know the context of deployment. The conventional software development life cycle has several phases which start with requirement definition and continue with system and software design, implementation and unit testing, integration and system testing and finally operation and maintenance [Som96]. This approach was first formulated as the water-fall model and later evolved to several other different methodologies, notably to full object-oriented analysis, design and implementation [Wil94a]. Although the principles established by the water-fall approach or by other variants are still influential in the component-based development process some new concepts have also emerged. In one of the relatively more recent efforts Sametinger [Sam97] re-formulates the software engineering life cycle around the idea of component-based software development. He introduces the idea of *application engineering* referring to the process of finding, assessing, adapting and finally assembling the existing re-usable components to build a new application. He also introduces the notion of *component engineering* meaning software development *for* re-use in other contexts than the one it was initially developed for. In another effort Short [Sho97] identifies and outlines the steps involved in the development of an off-the-shelf binary re-usable component.

Despite these efforts, however, there is not a well-established, widely accepted methodology that promises successful development of off-the-shelf binary components that are assured to deliver the benefits of component-based development implied in its theoretical framework. Of particular concern is the development of component-based systems which exhibit required system-wide properties such as reliability, availability, maintainability, security, responsiveness, manageability, and scaleability [Szy98c]. Another challenge is to design component interfaces that are capable of meeting the requirements of a reasonably large set of different contexts to support meaningful collaborations with other related components in construction of various systems.

This chapter presents the design process of AniComp, based on the techniques and approaches provided in the literature mentioned above. First, the adopted design methodology is introduced followed by the discussion of the tools used to support this methodology. Then the outcomes of applying this methodology are presented. It should be noted that the design solutions demonstrated here exhibit the latest state of an incremental and iterative process, which is not very mature and complete.

After the discussion of design issues the chapter continues with the discussion of a number of issues related to the implementation of AniComp, including the tools used, performance and efficiency considerations, internal logic of animation generation and name based object manipulation. Finally, the creation and the use of re-usable behaviours are exemplified by the modeling and implementation of a six degree of freedom (6DOF) robot arm.

5.2 The Design Methodology

To start with, recalling the issues introduced in Chapter 2 it can be said that in general, a component has three facets: 1 - a specification, which declares the component's interface and describes the semantics. The semantics explains what the component does, and how a client should use it. 2 - an implementation design, which describes how an implementer has chosen to design and construct the software and the data stored to meet the intended specification. 3 - an executable which delivers the component's capability on a designated platform. Once an interface for a component is specified and published, it can serve as a reference point for the implementers of the component and also as a reference point for the component assemblers who wish to compose that component with others to build an application. It is important to recognize that the specification model for

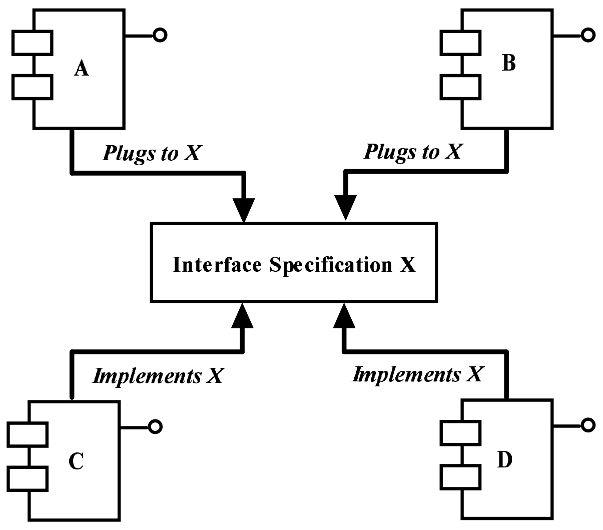


Figure 5.1: The Relationships Between Components and Interface specifications

an interface ***is not*** an implementation design, it is totally independent of any specific technology. In fact, the interface does not expose how the data, which is managed by the component, is organized or stored. An interface succinctly summarizes a parcel of behavior and responsibilities that any component will have to uphold if it is to participate as part of an assembly which represents some collaboration in the application domain. Figure 5.1 shows the relationships between the implementers and clients of an interface specification. Interface X, having a presumably useful set of services, is re-used by both the components A and B. From the perspective of the clients, there are two choices for the suppliers of X services - either component C or D, since both implement the same specification. Perhaps team A could choose component C (it may be cheaper), but later substitute it with component D (it may perform better). For an interface specification to successfully address the requirements of the possible collaborations with different

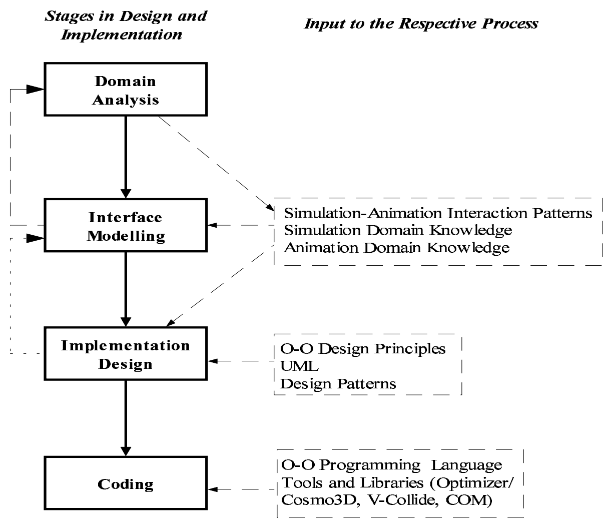

Stages in Design and Implementation
Input to the Respective Process
Domain Analysis
Interface Modelling
Implementation Design
Coding
Simulation-Animation Interaction Patterns
Simulation Domain Knowledge
Animation Domain Knowledge
O-O Design Principles
UML
Design Patterns
O-O Programming Language
Tools and Libraries (Optimizer/ Cosmo3D, V-Collide, COM)

Figure 5.2: The Stages in Design and Implementation of AniComp

components in an application domain it is necessary to have a good understanding of the nature of component interactions in that domain. Although it is difficult (if not impossible) to know all the possible interactions of a component in an application domain before its deployment, a domain analysis study can help to understand and determine the basic forms and types of such interactions. Then an interface design can be proposed. In fact such an analysis effort can contribute to regulate and formalize the structure of component interactions in applications in a certain domain which can in turn lead to the construction of component frameworks. The stages in the design of an interface and the component which implements that interface are shown in Figure 5.2. Domain analysis, is followed by interface modeling which is followed by implementation design. The dashed back reference lines indicate that these three phases can be repeated to ensure a more accurate design. If, however, the interface modeler and the implementation designer are different persons (or organizations) the second feedback line may not exist or may be indirect. This is indicated by a faint dashed line. These stages are adopted from a preliminary form proposed by Short [Sho97]. They outline a methodological approach to design and implementation of an off-the-shelf binary component. The diagram in Figure 5.1 also illustrates the input to the stages in terms of tools and materials some of which are particular to the application domain studied during this research work.

The following section gives a brief explanation of the tools and approaches utilized in applying this methodology.

5.2.1 Tools Used in the Interface and Implementation Design of AniComp

To apply the design methodology introduced in the previous section several tools were utilized. The information obtained from the domain analysis is used to model the interface of the AniComp in the form of **re-use contracts**. The inter-component interactions and intra-component interactions (interactions of the objects inside the component) resulting from them are modeled using **collaboration diagrams**. Several **design patterns** are used in the internal design of the component. The **Unified Modeling Language (UML)** is used to represent the classes, objects, interaction and collaboration diagrams, as a tool and notation guide to object-oriented design of the internal structures of AniComp. These tools are further explained below.

Re-use Contracts

Re-use contracts were introduced in Chapter 2 Section 2.5.1. A re-use contract is defined as an interface description for a set of collaborating participant components. It states the participants that play a role in the re-use contract, their interfaces, their acquaintance relations, and the interaction structure between acquaintances.

It is important that information on the interaction structure should be part of the interface specification of a component, so that it can be used to make the architecture clear, to help developers in adapting components to particular needs, and to verify component composition based on their interfaces instead of auxiliary (and perhaps informal) documentation. It is also important to emphasize the interface specification's contractual nature. Since the component and its clients are developed in mutual ignorance, the standardized contract must form a common ground for successful interaction [Szy99]. Re-use contracts address these requirements at the interface specification level.

Interaction Diagrams

A pattern of interaction among instances is shown on an interaction diagram. Interaction diagrams come in two forms based on the same underlying information, specified by an interaction, but each form emphasizing a particular aspect of it. The two forms are: sequence diagrams and collaboration diagrams. Sequence diagrams show the explicit sequence of stimuli and are better for real-time specifications and for complex scenarios. Collaboration diagrams show the relationships among instances and are better for understanding all of the effects on a given instance and for procedural design. A sequence diagram shows the interactions arranged in temporal sequence. In particular, it shows the instances participating in the interaction by their “lifelines” and the stimuli they exchange ordered in time. It does not show the associations among the objects. A Collaboration defines a set of participants and relationships that are meaningful for a given set of purposes.

Design Patterns

Design patterns were first introduced in Chapter 2 for the structural analysis of component wiring platforms. They are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. A design pattern names, abstracts and identifies the key aspects of a common design structure that makes it useful for creating a re-usable design. The most notable work which presents a catalog of classified design patterns which are directly applicable is that of the Gang of Four [Gam95], as was mentioned in Chapter 2. Three of these patterns are used in the internal design of AniComp, namely the *Mediator, Delegator and Facade* patterns. The mediator pattern uses one object to coordinate the communication of the state changes between other objects. Putting the logic in one object to manage the state changes of other objects, instead of distributing the logic over the objects, results in a more cohesive implementation of the logic and decreases the coupling between the other objects. The Facade pattern simplifies access to a related set of objects by providing one object that all objects outside the set use to communicate with the set [Gra98].

Object-Oriented Design and UML

Booch defines object-oriented design as a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design [Boo94].

The Unified Modeling Language (UML) [UML99] is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML is claimed to represent a collection of the best engineering practices that have proven successful in the modeling of large and complex systems. The UML constructs used in this work are class diagrams, object diagrams, collaboration diagrams and sequence diagrams. A notation guide to these constructs is provided in Appendix A.

5.3 The Design Process

5.3.1 Domain Analysis: Determining Simulation-Animation Interaction Patterns

The application domain in the study presented here is simulation combined with animation. Of particular interest is to animate the behaviour of a simulated entity or visualise many entities in a simulation scenario. There are two main tasks involved in specifying the requirements of an animation and visualization server (component) connectable to simulation components: 1 - To identify the execution logic of simulation software packages based on the simulation modeling approach they are based on. 2 - To identify the internal mechanisms of a graphics system capable of loading, displaying and manipulating (animating) a geometric model. Chapter 4 presented the major simulation modeling approaches and their impact on simulation software execution logic as well as on the generation of simulation-animation interaction patterns. An interaction model was also developed in section 4.4.4 of chapter 4. Figure 5.3 (a) summarizes the execution logic of discrete event simulation software and its communication with an animation component. Figure 5.3 (b) summarizes the logic of a typical continuous simulation loop and its output to an animation component. Chapter 3 provided a discussion of animation techniques in general, with an emphasis on automatic

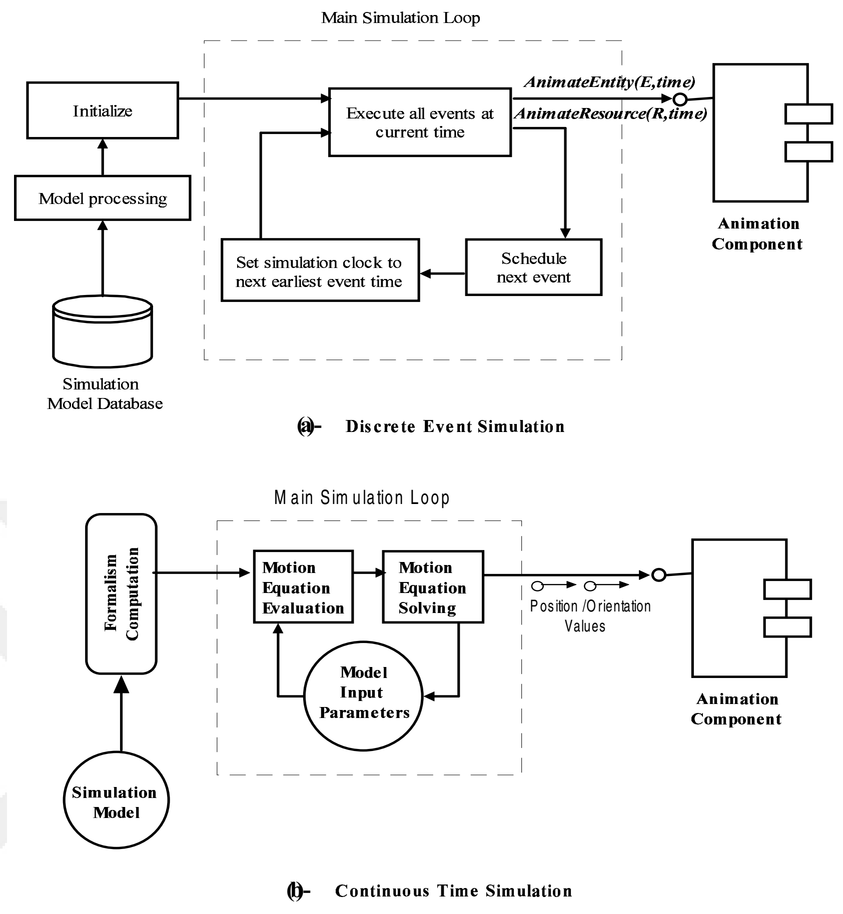


Figure 5.3: The Main Simulation Loop in a Discrete Event Simulation and its Output to the Animation Component

motion generation techniques. Procedural animation is particularly important from this thesis point of view since simulation engines are conceived to calculate the motion of geometric objects. Given an external procedure responsible for generating the animation of geometric objects, there can be three ways of imposing the calculated motion on the geometric objects:

1. The procedure can calculate successive position and orientation values at suitable time steps to generate a smooth motion of geometric objects. The object can be as complex as an articulated structure containing joints and many components (a multi-body system) in which case the procedure has to take into account the constraints imposed by the joints (e.g. a robot arm). Alternatively the object may represent a very simple system in which case the procedure is significantly simpler generating only single position and

orientation values at each time step.

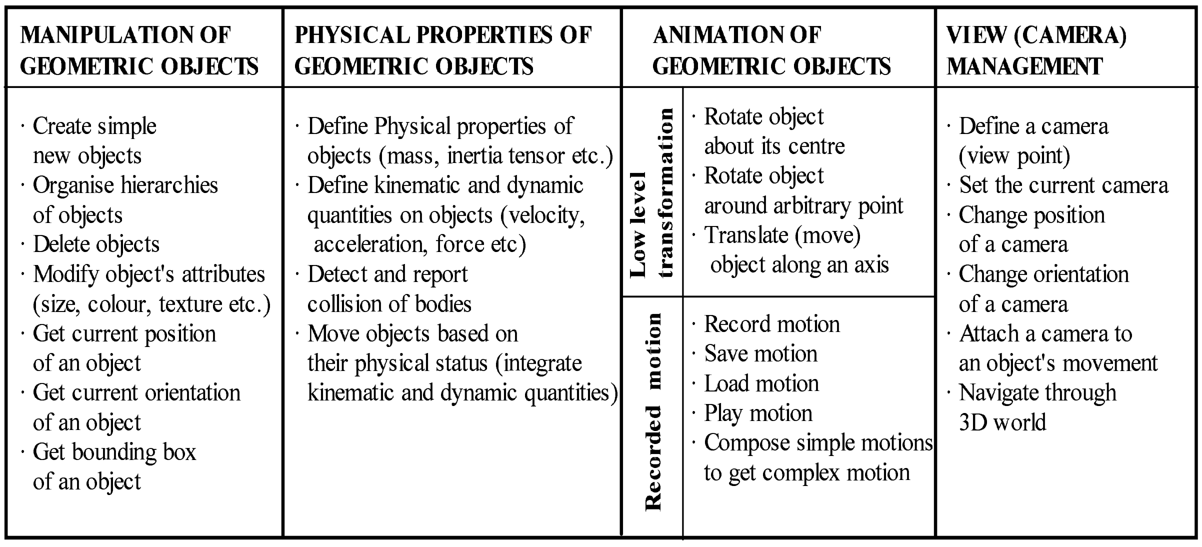
2. The procedure can calculate higher level physical values such as velocity, angular velocity or force on objects, which are likely to change at a much lower frequency than successive position and orientation values. These values can then be passed to yet another procedure (which should be local to the animation module) to be integrated to generate lower level motion quantities. This approach is particularly desirable in distributed environments to reduce the number of calls and thus to reduce the network traffic. It also serves to reduce the coupling between the module containing the procedure (i.e. the simulation component) and the module responsible for moving the 3D geometry (i.e. the animation module).

3. The main responsibility of the procedure can be to orchestrate the motion of multiple objects in which case the motion of the individual objects can be generated by one of the following two methods:

   (a) The motion of an object can be computed and recorded beforehand to be invoked (or activated) when the object needs to present a particular behaviour depicted by that motion sequence.

   (b) The main procedure can call yet another procedure to compute the motion of an individual object in real-time.

The scheme given above, which was also discussed in Chapter 4, summarizes the findings of the domain study presented in Chapters 2 and 3, and forms a basis for determining the requirements for AniComp's interfaces. These requirements are presented in a tabular form in Figure 5.4 (see next page). Based on the domain analysis performed and the interaction scheme given above, a top level architectural view has been devised as illustrated in Figure 5.5. The animation component is designed to have three *conceptual* layers. The top layer is responsible for view management, motion generation and management, and time management. The view management tasks involve window and camera generation, assignment of views to windows etc. The motion generation and management functions include recording, saving, loading and indexing (naming) motion sequences to be invoked (activated) later as discussed in the third item of the interaction scheme described above. The time management functions consist of setting and activating the global animation clock and setting the timing of motion sequences



MANIPULATION OF GEOMETRIC OBJECTS	PHYSICAL PROPERTIES OF GEOMETRIC OBJECTS	ANIMATION OF GEOMETRIC OBJECTS		VIEW (CAMERA) MANAGEMENT
· Create simple new objects · Organise hierarchies of objects · Delete objects · Modify object's attributes (size, colour, texture etc.) · Get current position of an object · Get current orientation of an object · Get bounding box of an object	· Define Physical properties of objects (mass, inertia tensor etc.) · Define kinematic and dynamic quantities on objects (velocity, acceleration, force etc) · Detect and report collision of bodies · Move objects based on their physical status (integrate kinematic and dynamic quantities)	**Low level transformation**	· Rotate object about its centre · Rotate object around arbitrary point · Translate (move) object along an axis	· Define a camera (view point) · Set the current camera · Change position of a camera · Change orientation of a camera · Attach a camera to an object's movement · Navigate through 3D world
		Recorded motion	· Record motion · Save motion · Load motion · Play motion · Compose simple motions to get complex motion	

Figure 5.4: Summary of Requirements for An Animation Module

for recorded behaviours. The collision management functions are responsible for detecting collisions between rigid bodies moving in the environment. They can also be used to handle the realistic navigation of the user in the virtual world by applying non-penetration constraints between an avatar representing the user and the environment. The Local Physics Simulator extrapolates the incoming physical values of velocity or acceleration to obtain successive displacement or rotation values. The Scene Graph Management module facilitates the manipulation of the geometric objects after they are loaded into the memory in the form of a directed acyclic graph. The Geometric Model Serialization module deals with loading and saving of the geometric models from and to the model files. The Rendering and Animation module is used to display the geometric objects on the computer screen by calling the underlying graphics system's functions and managing the frame buffers.

The User Interaction Metaphor shown in Figure 5.5 refers to the way user chooses to interact with the virtual environment in which the simulation is taking place. This can be a conventional 2D window with buttons, sliders or other gadgets; or it can be based on direct manipulation of 3D objects within the virtual environment by using advanced interaction devices such as 3D space balls, data

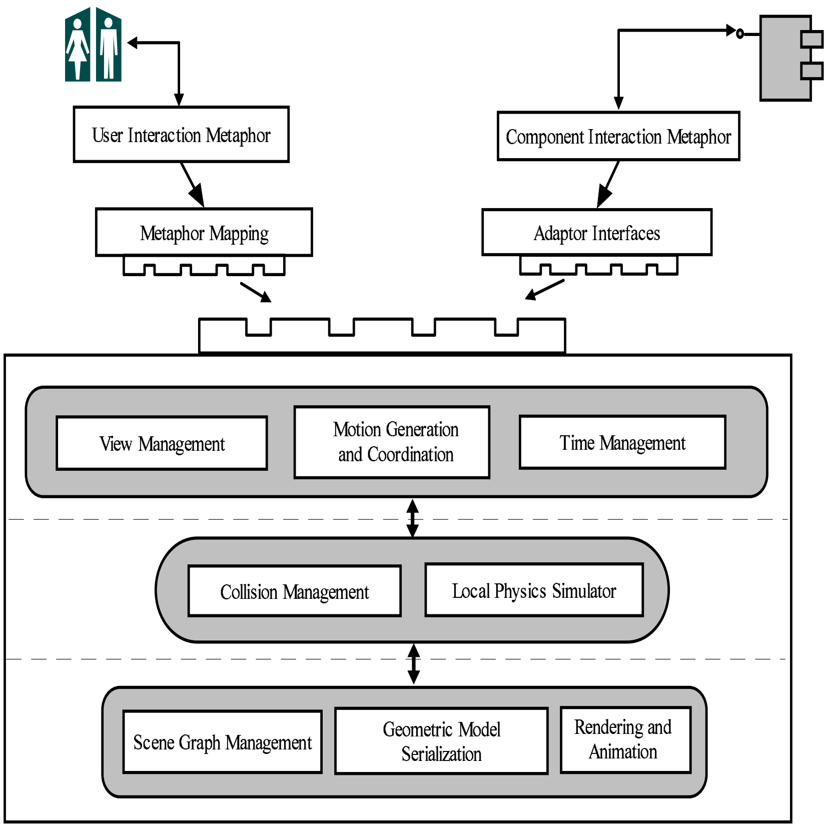


Figure 5.5: A Top Level Architecture for Animation Component

gloves and hand and gesture tracking devices. It is important to design an animator component to allow the user to choose from different interaction paradigms which capture a particular application's need in a most suitable manner. To enable this, a general interface can be provided for 3D object interaction and manipulation purposes and the mapping of a particular interaction metaphor can then be left as a relatively less challenging programming task for the user. The Metaphor Mapping component is meant to refer to this coding task. Similarly in the case of interface mismatches with the simulator engines which wish to use the animator's interfaces, an adaptor component may need to be developed for resolving the interface differences. The Adaptor Component is meant to address this requirement.

5.3.2 Interface Design and Specification

The design and the specification of the Anicomp's interfaces is accomplished by using *re-use contracts*. Re-use contracts were introduced in Chapter 2 Section 2.5.1 and an example figure was given to familiarize the reader with its notation. During their use in the specification of the AniComp's interfaces a slight extension has to be made to the notation. To show the exact protocol of interaction between the participant components it is essential to explicitly indicate the order of function calls. This however may result in many function calls to be repeated in a re-use contract diagram, which reduce its re-usability. To eliminate this a new clause has been added to the notation called *preceded by:* which specifies that a contract is preceded by an ordered list of other contracts. Each contract must have a well defined name and all the contracts preceding a certain contract in the interaction protocol are shown in a bulleted list, in the middle of the main diagram box. Figure 5.6 shows the first step in a typical interaction of a sim-

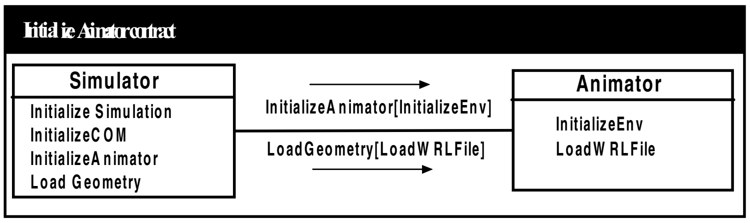


Figure 5.6: Re-use Contract for Initializing the Animator Component

ulator component with the animator component. It indicates that a simulator component, after its own initialization phase, must initialize the COM package and then initialize AniComp. This should be followed by loading a geometric model to populate the 3D virtual environment. In Figure 5.7 the added no-

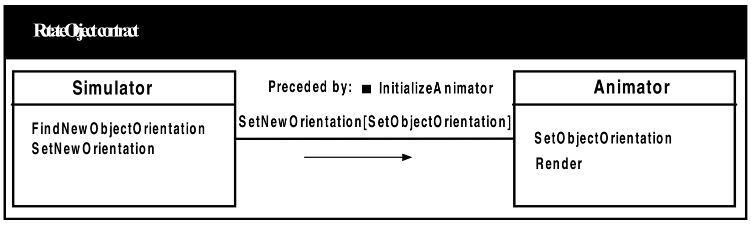


Figure 5.7: Re-use Contract for Setting the Orientation of An Object

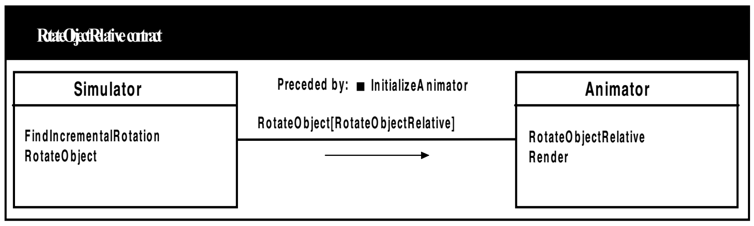

RotateObjectRelative contract
Simulator
FindIncrementalRotation
RotateObject
Preceded by: ■ InitializeAnimator
RotateObject[RotateObjectRelative]
Animator
RotateObjectRelative
Render

Figure 5.8: Re-use Contract for Rotating an Object Relative to its Current Orientation

tation appears indicating that before being able to rotate an object, first the InitializeAnimator contract must be applied. Otherwise the interaction defined by this contract would not be valid. It also shows that the animator component expects the simulator component to compute an absolute orientation to be able to use this interaction. This is not the case for the RotateObjectRelative contract in Figure 5.8, however, because this time the animator component requires an *incremental* rotation to be computed. Both contracts make it clear that the animator component would render the effect of the rotations immediately, therefore no other call is required for rendering. It can be seen that re-use contracts

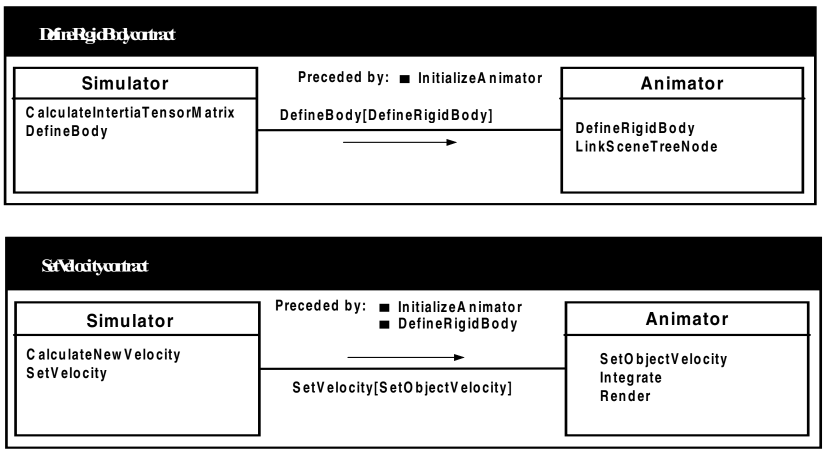


Figure 5.9: Re-use Contract for Defining and Moving a Rigid Body With Velocity

can convey a significant amount of information to the user, in a semi-formal and highly readable way. This is important because currently component interfaces are mostly described in the form of IDL (Interface Definition Language) files, which only contain a list of functions in the component's interfaces with their parameters. This, however, is not expressive enough to specify the protocol of

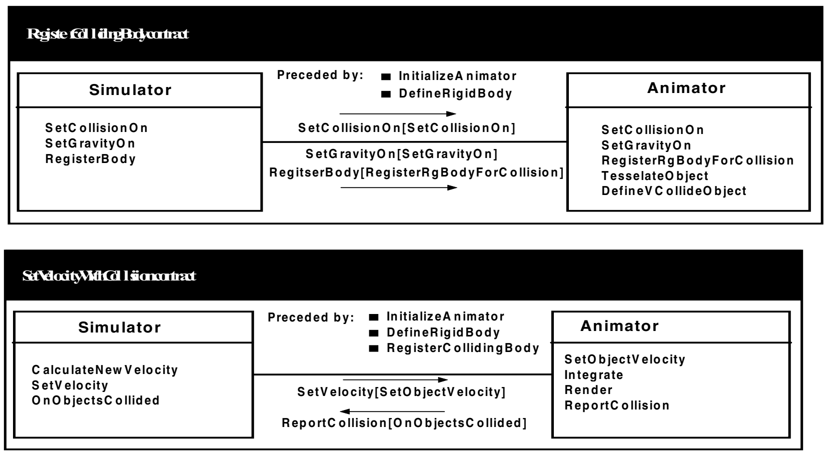


Figure 5.10: Re-use Contract for Moving a Rigid Body With Collision Detection Enabled

interaction (the order of function calls) and it also does not give any indication of the component's expectations from its acquaintances.

As the interactions become more complicated the importance of such information becomes even more apparent. Figure 5.9 shows an example of such a case. The example shows that to move an object with a specified velocity, first the object must be defined as a rigid body because SetVelocity contract is preceded by DefineRigidBody contract in a valid interaction. The user also realizes that to define a rigid body, the inertia tensor matrix have to be calculated. An even more notable case is the one illustrated in Figure 5.10. Here, the SetVelocityWithCollision contract shows that if the moving objects are to be checked against collision with other objects, then, before setting their velocity, the RegisterCollidingBody contract must be applied. This contract imposes that first, collision detection must be enabled and the rigid body must be registered for collision detection. More importantly the SetVelocityWithCollision contract shows that the animator component sends the collision events to the simulator, and that the simulator component must implement a call-back function (OnObjectsCollided) to handle the collision responses.

5.3.3 Implementation Design

In this section, first, a top level class diagram is given to illustrate the overall structure of the animator component. Following that, the collaboration of these classes at the time of interactions with simulator components are presented to illustrate a dynamic view of the system. Since it is not possible to discuss all the probable interactions that can arise by calling all the functions in AniComp's interfaces (a total of 65 functions) in this thesis, only some representative cases are selected. These cases are formulated to capture the most important aspects of a certain group of functions in the AniComp's interface. A list of all the functions in the interface, accompanied by their description and usage information, is given in the concise reference manual, in Appendix B.

The General Picture

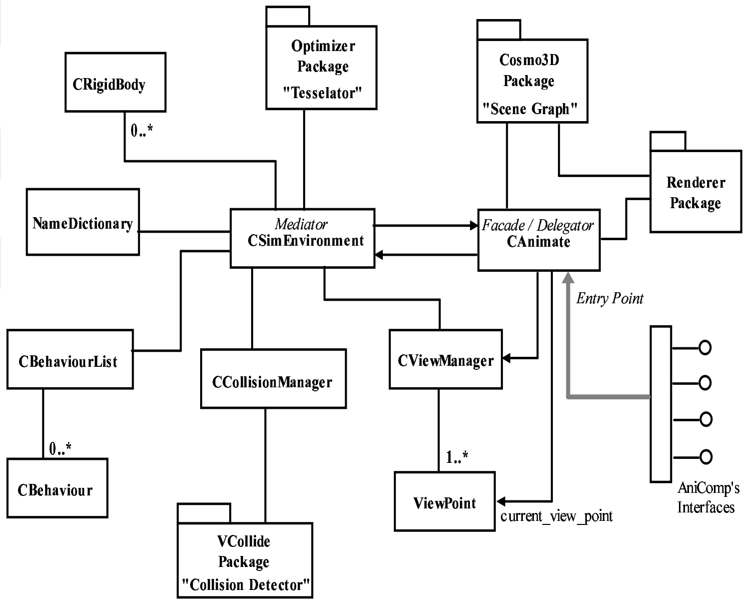


Figure 5.11: A Top Level Class Diagram of AniComp's Design

The internal design of AniComp utilizes well-known design patterns. Design patterns are an effective way of expressing proven techniques and designs used in software systems. They name, abstract and identify the key aspects of a common design structure that makes it useful for creating re-usable designs [Gam95]. Three such patterns are Facade, Delegator and Mediator. The CAnimate class shown in Figure 5.11 acts as an entry point to the whole system. It therefore exemplifies the Facade pattern. It also *delegates* incoming requests from the clients to objects of classes which are designed to handle them. So it is also a delegator. The CSimEnvironment class acts as a central class handling the communication between many other classes. Assigning a class to mediate between many other classes reduces the complexity of the mutual dependency network between classes. CSimEnvironment, therefore, is a good example of the Mediator pattern. The CRigidBody class implements a simple rigid body simulator local to the animator. It extrapolates physical quantities such as velocity or acceleration using its integrator to generate the motion of the objects. This is useful in reducing the number of function calls between simulator and animator if the motion of the object can be determined by such physical quantities. The CBehahviourList and CBehaviour classes hold named motion sequences that are loaded from a motion library and activated on demand. Motion sequences are keyframe animations that are implemented as position and orientation interpolators. NameDictionary is a hash table which in each entry, keeps the name of a 3D object and a pointer to the object's transformation node in the scene graph. The CollisionManager class checks whether the geometric objects in the environment are subject to an interference at every animation frame. If so, then it reports collisions, which is then delegated to the CSimEnvironment object. AniComp utilizes the OpenGL Optimizer/Cosmo3D class library [Cos00] for scene management, rendering and animation. It also employs the V-Collide [Hud97] collision detection library for detecting collisions between geometric objects. Brief explanations of these systems will be given during the discussions about the implementation details, later in this chapter.

Interactions During the Initialization Phase

Every simulation client must call AniComp's initialization function once at the beginning of a group of interactions. The initialization call is typically followed by a call for loading a geometric model. In Figure 5.12 these two calls are shown with their respective order represented by a sequence number. Both calls trigger a number of other interactions among the objects of the animator. In the diagram[1]arrows show the direction of the call. Each function call is preceded by a sequence number. Once invoked, the initialization function causes the CAnimate object first to create and show a dialog window for displaying the hierarchy of the geometry objects, then sets and starts the global clock. It then starts

[1]To remind the reader; the notation used for the collaboration diagrams is the UML notation and a detailed description of the UML structures used in this thesis is given in Appendix A.

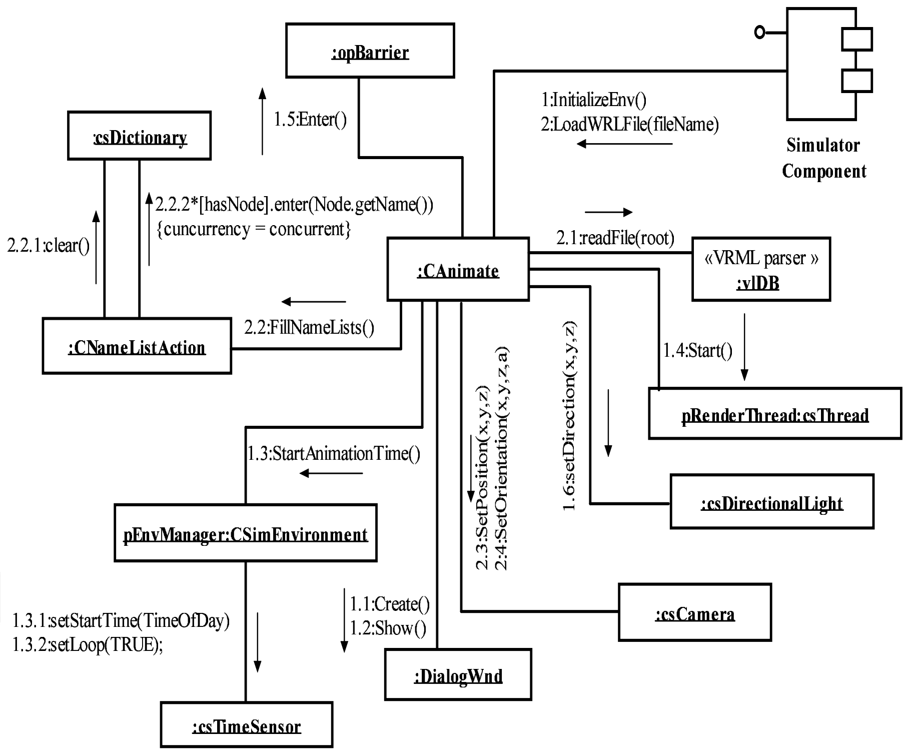


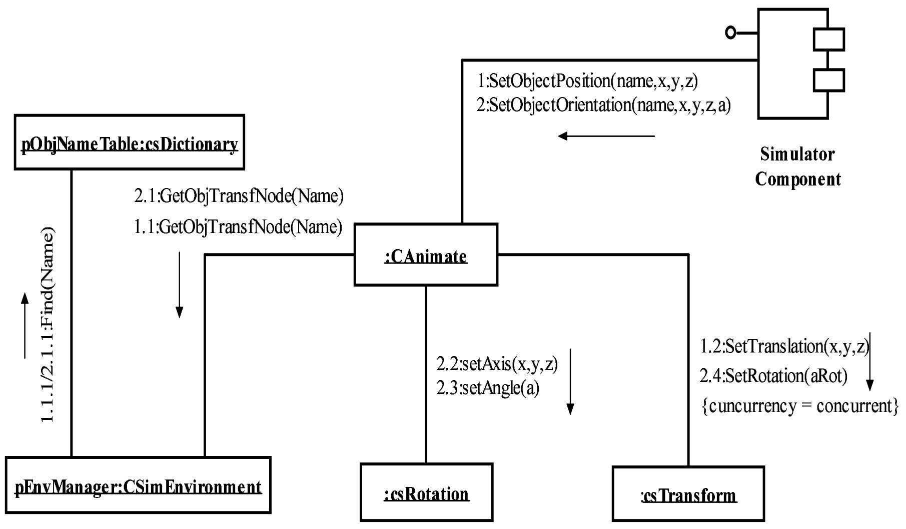
Figure 5.12: Collaboration Diagram for Initialization Phase

a new thread for rendering and gets blocked on a synchronization barrier (a semaphore-like thread synchronization mechanism) to prevent the current thread from modifying the scene tree before the new (rendering) thread finishes some initialization tasks related to the same scene tree. After a very short while the current thread becomes free to receive incoming interface calls once again. The fact that objects in the scene tree are shared between separate threads requires the use of thread synchronization mechanisms. This is reflected in the figure, next to the method call with sequence number 2.2.2. The statement `concurrency = concurrent` indicates that two threads are not allowed to access the same data simultaneously.

Similarly the second call, the LoadWRLFile() function, causes the CAnimate object to call the VRML file parser object, then to call the CNameListAction object which recursively traverses the scene tree and fills in a name dictionary. This name dictionary is used to facilitate name-based manipulation of 3D geometric objects in the virtual environment. After that, a camera object is initialized to make the geometric model visible to the user. After these two calls the animation component is ready to receive other calls for animating the geometric objects.

Interactions During Low-level Transformation Calls

The low level transformation calls mostly originate from a continuous simulation component. They may involve the specification of an absolute position, an incremental translation, an absolute orientation, an incremental rotation or some of these combined in a transformation matrix. As it can be seen in the collabora-



1:SetObjectPosition(name,x,y,z)
2:SetObjectOrientation(name,x,y,z,a)
Simulator Component
pObjNameTable:csDictionary
2.1:GetObjTransfNode(Name)
1.1:GetObjTransfNode(Name)
:CAnimate
1.1.1/2.1.1:Find(Name)
2.2:setAxis(x,y,z)
2.3:setAngle(a)
1.2:SetTranslation(x,y,z)
2.4:SetRotation(aRot)
{cuncurrency = concurrent}
pEnvManager:CSimEnvironment
:csRotation
:csTransform

Figure 5.13: Object Collaborations During Low Level Transformation Calls

tion diagram given in Figure 5.13, once initiated, these calls cause CAnimate to delegate a request to the pEnvManager object, for a name lookup for the object to be transformed. If the object exists, then the lookup operation returns the transformation node of the geometric object. This node in the scene tree is then updated with the new incoming values. The concurrency statement implies that the modifications made to the scene tree are synchronized with the rendering thread.

Interactions During Rigid Body Simulation

The interactions during rigid body manipulation are more complex compared to the low level transformation functions. First, in order to be regarded as a rigid body, a geometric object must be associated with a number of physical attributes.

This requires a definition procedure. This requirement was made explicit in Section 5.3.2, during the interface specification phase, by presenting the relevant re-use contract. Figure 5.14 illustrates the interactions internal to AniComp dur-

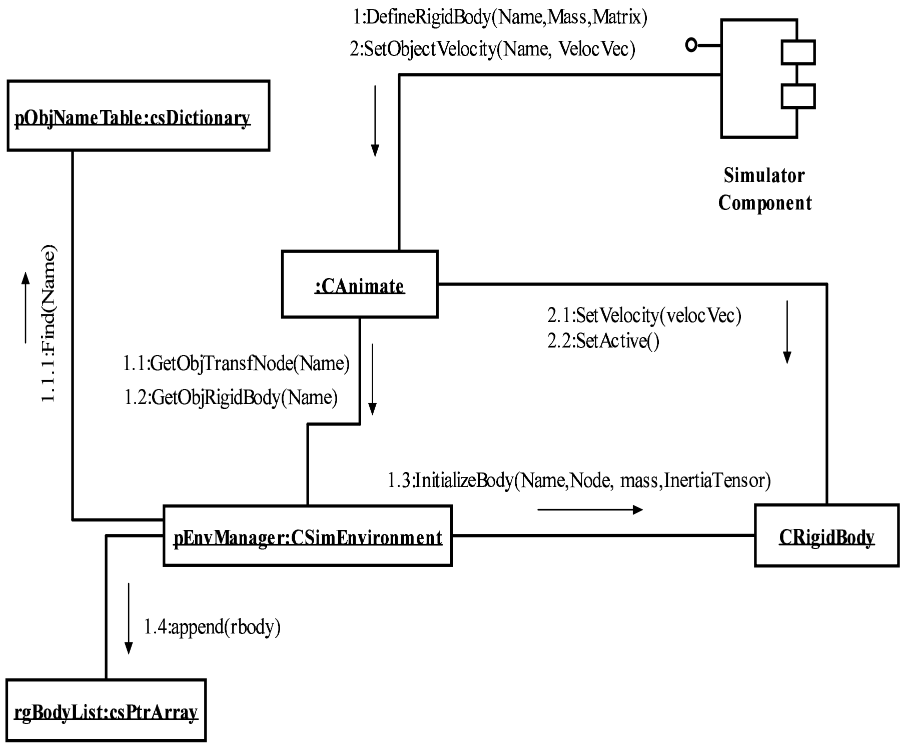

1:DefineRigidBody(Name,Mass,Matrix)
2:SetObjectVelocity(Name, VelocVec)
Simulator Component
pObjNameTable:csDictionary
1.1.1:Find(Name)
:CAnimate
1.1:GetObjTransfNode(Name)
1.2:GetObjRigidBody(Name)
2.1:SetVelocity(velocVec)
2.2:SetActive()
1.3:InitializeBody(Name,Node, mass,InertiaTensor)
pEnvManager:CSimEnvironment
CRigidBody
1.4:append(rbody)
rgBodyList:csPtrArray

Figure 5.14: Object Collaborations During Rigid Body Definition

ing this definition call. The environment manager initializes a rigid body object with the given physical properties as parameters in the initialization routine and then adds the new body to the list of rigid bodies in the environment. Following this call, the velocity of that body can be set. This procedure is shown by the calls with sequence numbers 2.1 and 2.2. What happens after that, however, requires more explanation and a separate diagram, which is given in Figure 5.15. The collaboration diagram in Figure 5.15 shows the relevant interactions during a typical rendering loop which gets called for every animation frame. The CAnimate object delegates the animation request of the rigid bodies to pEnvManager. The pEnvManager object then finds the number of rigid bodies in the environment and for each body in the list integrates its physical quantities for one step, renders their new positions and delegates a check-for-collision request to a CCollisionManager object. This process is repeated continuously. For the collision detection

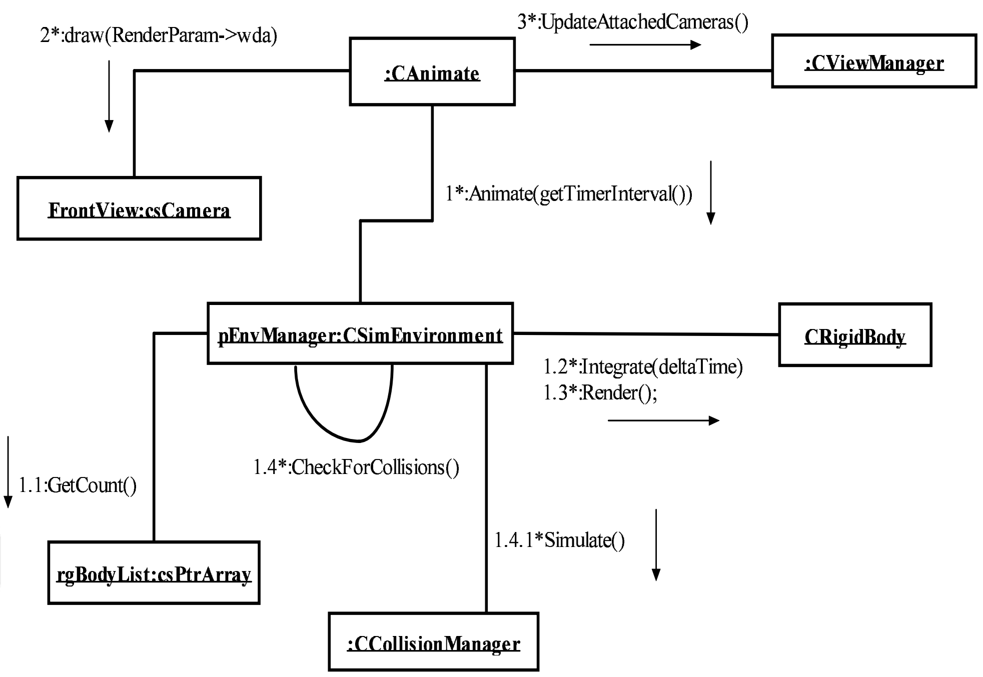

2*:draw(RenderParam->wda)
:CAnimate
3*:UpdateAttachedCameras()
:CViewManager
FrontView:csCamera
1*:Animate(getTimerInterval())
pEnvManager:CSimEnvironment
CRigidBody
1.2*:Integrate(deltaTime)
1.3*:Render();
1.1:GetCount()
1.4*:CheckForCollisions()
1.4.1*Simulate()
rgBodyList:csPtrArray
:CCollisionManager

Figure 5.15: Object Collaborations During the Movement of Rigid Bodies

to be done correctly, however, the bodies must be registered for collision. The process for registering an object for collision triggers a number of other object interactions. Figure 5.16 shows these interactions. Since the V-Collide collision detection package used in the implementation requires objects to be defined as a list of triangles, every geometric object must be triangulated (i.e. tesselated). This requires finding the sub-components of an object recursively if any. In the end, the new tesselated object is registered with the collision manager as a new V-Collide object.

Interactions During Behaviour Generation and Activation

The term *behaviour* is used to refer to pre-recorded key-framed motion sequences. There are two ways of creating motion sequences:

1. They can be defined and edited by using a VRML 2.0 authoring package. Advanced packages such as 3DMax Studio allow the users to export model files in VRML 2.0 format. If some of the objects are animated, these animations are also saved in the exported VRML files because VRML 2.0 has defined nodes for behaviours [vrm98]. AniComp can import and recognize

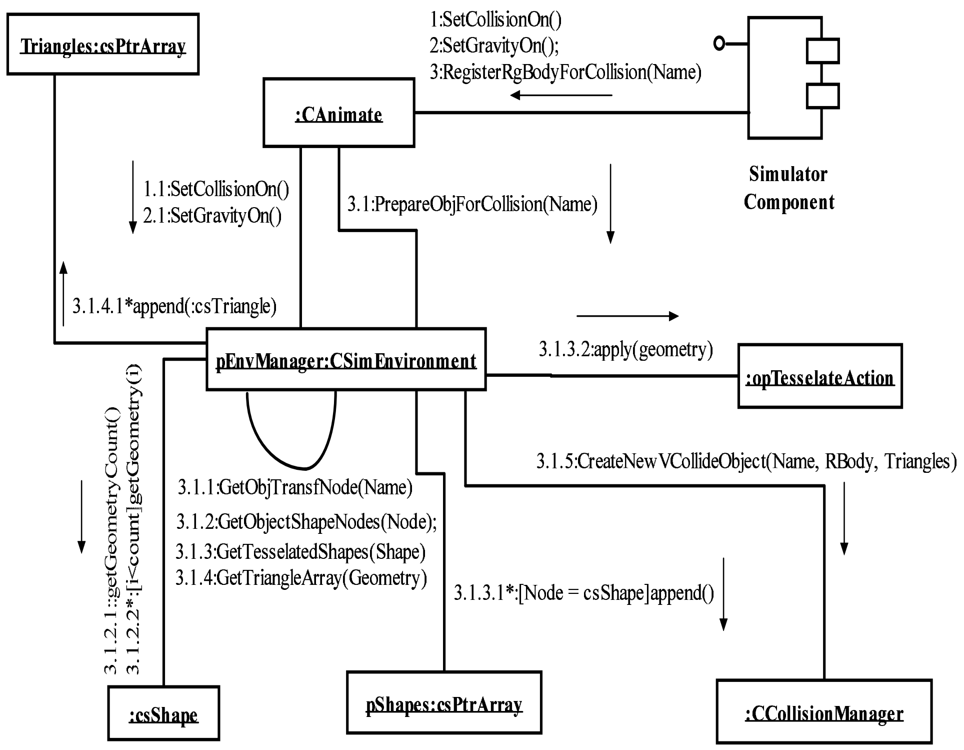

Triangles:csPtrArray
1:SetCollisionOn()
2:SetGravityOn();
3:RegisterRgBodyForCollision(Name)
:CAnimate
Simulator Component
1.1:SetCollisionOn()
2.1:SetGravityOn()
3.1:PrepareObjForCollision(Name)
3.1.4.1*append(:csTriangle)
pEnvManager:CSimEnvironment
3.1.3.2:apply(geometry)
:opTesselateAction
3.1.5:CreateNewVCollideObject(Name, RBody, Triangles)
3.1.2.1::getGeometryCount()
3.1.2.2*:[i<count]getGeometry(i)
3.1.1:GetObjTransfNode(Name)
3.1.2:GetObjectShapeNodes(Node);
3.1.3:GetTesselatedShapes(Shape)
3.1.4:GetTriangleArray(Geometry)
3.1.3.1*:[Node = csShape]append()
:csShape
pShapes:csPtrArray
:CCollisionManager

Figure 5.16: Object Collaborations on Registering an Object for Collision

those nodes but they need to be named and activated explicitly.

2. AniComp provides a programming interface to generate (or record) a motion sequence and name it as a behaviour for later activation. This is especially useful if a well-defined behaviour is to be generated by dynamic simulation or an inverse kinematics engine. Within the main simulation loop of such simulation engines successive position and orientation values can be saved in key-framed interpolators by using AniComp's interface functions.

A behaviour must first be defined; that is it has to be named and attached to the corresponding object. Figure 5.17 illustrates object collaborations during this initial phase. The simulator specifies the name of the behaviour and the name of the object for which the behaviour is defined. The CAnimate object delegates a name lookup request to the environment manager which then gets the new behaviour object and appends it to the list of behaviours. Figure 5.18 shows a later stage in the process of defining behaviours. The simulator calls AddItemToPosInterpolator function within a loop (indicated by a star after the sequence number).

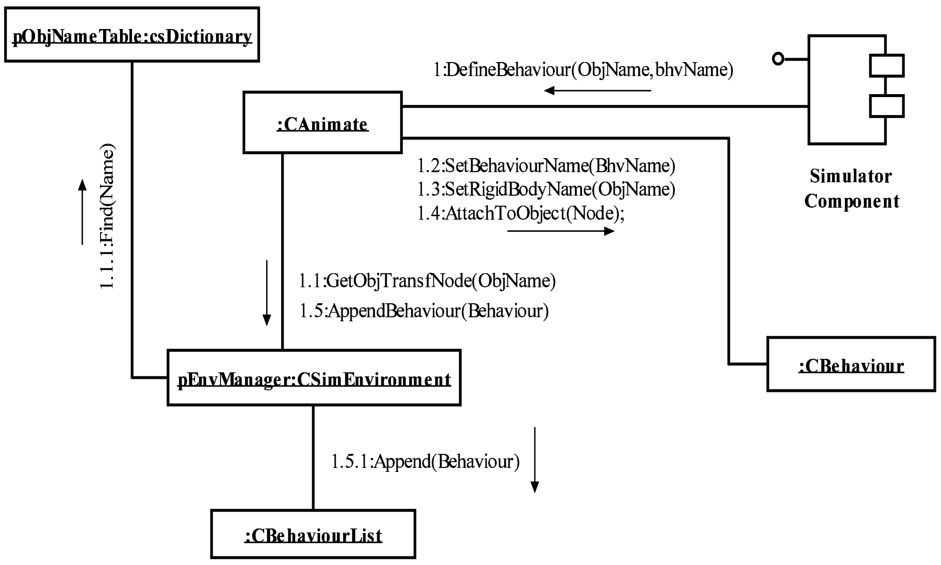

pObjNameTable:csDictionary
1:DefineBehaviour(ObjName,bhvName)
:CAnimate
1.2:SetBehaviourName(BhvName)
1.3:SetRigidBodyName(ObjName)
1.4:AttachToObject(Node);
Simulator Component
1.1.1:Find(Name)
1.1:GetObjTransfNode(ObjName)
1.5:AppendBehaviour(Behaviour)
pEnvManager:CSimEnvironment
:CBehaviour
1.5.1:Append(Behaviour)
:CBehaviourList

Figure 5.17: Object Collaborations on Defining a New Behaviour for An Object

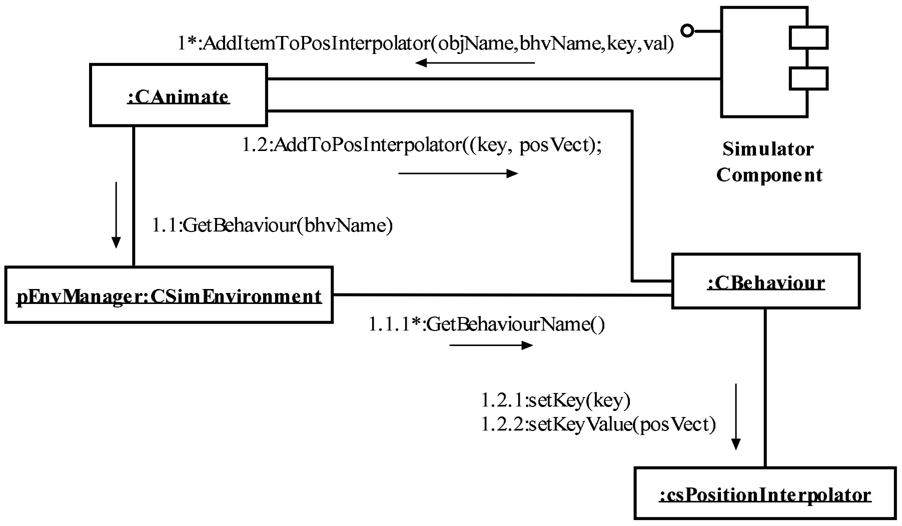

1*:AddItemToPosInterpolator(objName,bhvName,key,val)
:CAnimate
1.2:AddToPosInterpolator((key, posVect);
Simulator Component
1.1:GetBehaviour(bhvName)
pEnvManager:CSimEnvironment
:CBehaviour
1.1.1*:GetBehaviourName()
1.2.1:setKey(key)
1.2.2:setKeyValue(posVect)
:csPositionInterpolator

Figure 5.18: Object Collaborations When Adding Values to Interpolators

This function's parameters includes the names of the behaviour and the object as well as a key and its corresponding position value (a vector). The (key, keyvalue) tuples get recorded continuously until the end of the motion sequence to form a key-frame interpolator. The same procedure is valid for recording orientation values, which is not shown in the figure. The activation of the behaviours involves a

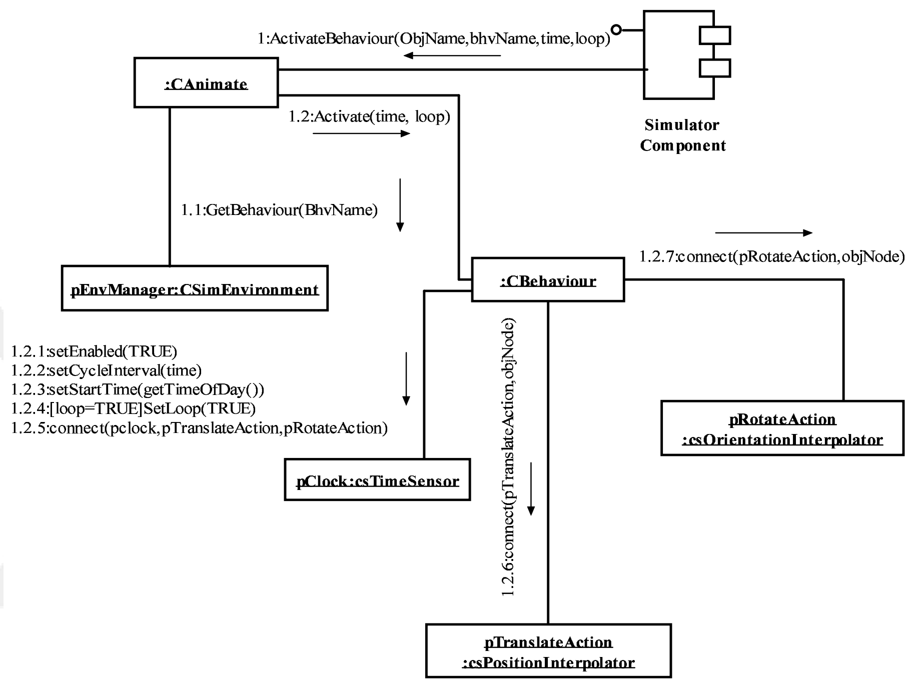

1:ActivateBehaviour(ObjName,bhvName,time,loop)
:CAnimate
1.2:Activate(time, loop)
Simulator Component
1.1:GetBehaviour(BhvName)
1.2.7:connect(pRotateAction,objNode)
pEnvManager:CSimEnvironment
:CBehaviour
1.2.1:setEnabled(TRUE)
1.2.2:setCycleInterval(time)
1.2.3:setStartTime(getTimeOfDay())
1.2.4:[loop=TRUE]SetLoop(TRUE)
1.2.5:connect(pclock,pTranslateAction,pRotateAction)
pRotateAction :csOrientationInterpolator
pClock:csTimeSensor
1.2.6:connect(pTranslateAction,objNode)
pTranslateAction :csPositionInterpolator

Figure 5.19: Object Collaborations on Activating the Behaviour of An Object

number of tasks such as initializing clocks, connecting clock ticks to interpolators and connecting inperpolators' outputs to the transformation nodes of the objects to be animated. This procedure is shown in Figure 5.19. The CBehaviour object is responsible for much of these tasks. The mechanisms involved are essentially based on the VRML 2.0 internal execution model which is further explored during the discussion of implementational details later in this chapter.

5.4 The Implementation of AniComp

5.4.1 Tools Used

Visual C++ Development Environment with MFC and ATL

The development language selected was C++, due to several reasons including the performance requirements of the application domain, the compatibility issues of the tools used, and author's familiarity with the language. The Microsoft Visual C++ development environment has a number of advantages over other PC based C++ environments regarding the particular programming tasks involved in this work [Hor97, Bla97]. First, it has extensive support for the component technology selected for this study: COM (Component Object Model). Second, it integrates frameworks such as MFC (Microsoft Foundation Classes) and ATL (Active Template Library)[2] [Bla97] which are very helpful for creating user interfaces and component interfaces, respectively. Third, ATL uses advanced optimization techniques to assure very efficient execution of the COM components [Gri98b]. Although they bring about platform dependence, these frameworks reduce the development time significantly. Recalling that (from Chapter 2) COM/DCOM, by definition, supports platform independence, the issue of platform dependence caused is restricted to the use of user interface framework of choice (i.e. MFC).[3].

COM/DCOM As the Component Platform

As introduced in Chapter 2, the Component Object Model (COM) is a component software architecture that allows applications and systems to be built from components supplied by different software vendors. COM is the underlying architecture that forms the foundation for higher-level software services [Wil94b]. COM provides a component software architecture that:

- Defines a binary standard for component inter-operability
- Is programming language-independent

[2]COM, MFC and ATL are trademarks of Microsoft Corporation

[3]MFC provides many classes for other purposes, such as data structures, algorithms and many more, but care has been taken to use standard C++ constructs for such purposes to reduce the portability problems

- Is provided on multiple platforms (Microsoft Windows, Microsoft Windows NT, Apple Macintosh, UNIX)
- Provides for robust evolution of component-based applications and systems
- Is extensible. In addition, COM provides mechanisms for the following:
    1. Communications between components, including communications across process and network boundaries
    2. Shared memory management between components
    3. Error and status reporting
    4. Dynamic loading of components

VRML 2.0 As A Geometric Modeling Format

The Virtual Reality Modeling Language (VRML) is a file format for describing 3D interactive worlds and objects. It may be used in conjunction with the World Wide Web. It may be used to create three-dimensional representations of complex scenes such as illustrations, product visualizations and virtual reality presentations.

VRML is becoming a widespread 3D exchange format. Good-quality scene authoring software is available, and the method for dynamic update of 3D scenes over the Internet is built into the standard. These properties will help realize the re-use of detailed CAD models for many purposes [Sti97]. VRML is also rapidly gaining acceptance as a mechanism to visualize geometric representations of a variety of manufacturing related entities [Res97].

OpenGL Optimizer/Cosmo3D As the Rendering and Animation Framework

Cosmo3D is a scene graph API that brings 3D graphics programming to desktop applications. It allows applications to use a higher-level interface than the low-level OpenGL graphics library that it is based on. Developers interact with C++ objects that are arranged in an object hierarchy. A scene graph is a directed acyclical graph of nodes, which embodies the semantics of what is to be drawn, but not how it is to be drawn [Cos00]. Developers interacting with a scene graph are interested in achieving a result, usually seeing a model on the screen and manipulating it. It is the scene graph APIs such as Cosmo3D which usually undertake the task of achieving this result in the most efficient way.

With its scene graph architecture and features such as culling, level of detail (LOD), 2D texture mapping, Cosmo3D enables the programmer to develop complex graphic applications. A Cosmo3D scene graph consists of objects that inherit appropriate methods and attributes from the Cosmo3D classes. Conceptually, there are four kinds of classes:

- Base classes: csObject, csField, csContainer and csNode. These classes are never instantiated directly. Instead, applications create subclasses that inherit certain functionality from the base classes.
- Scene graph construction classes: csGroup, csShape, csGeometry, and csAppearance determine appearance in a general way.
- Specific appearance classes: csContext, csDrawTraversal, csEnvironment and some of their subclasses determine how things are drawn, for example, whether lights or fog are applied.
- Geometry classes: such as csSphere or csCylinder, are the building blocks of the model itself.

There are classes that work with scene graphs, but are not nodes; they cannot be part of the scene graph, but they serve the vital function of enabling animation. csEngine is such a class. These classes are examined in more detail in Section 5.4.3 when discussing the internal logic of animating 3D objects in AniComp.

V-Collide As the Collision Detection Library

The V-Collide [Hud97] collision detection package is based on two other previous packages called I-Collide [Coh95] and RAPID [Got96]. V-Collide unifies the framework of the I-Collide and RAPID systems. The n-body "Sweep and Prune" algorithm for filtering collisions among large numbers of objects (I-Collide) sits at the top level of the collision detection routines, with an oriented bounding box (OBB) hierarchy providing pairwise exact contact determination for objects underneath (RAPID). RAPID assumes that inputs are triangulated polygonal models. Rather than extending RAPID's OBB hierarchy to handle VRML's cones, spheres, and cylinders, V-Collide requires the library caller to tessellate the solids within the (user-specified) tolerance desired for collision detection.

V-Collide was chosen as the collision detection library for this PhD work because it has two features which make it particularly suitable for general purpose, geometry-based collision detection:

- It is fast, therefore suitable for dynamic environments.
- It does not enforce the user to make any assumptions about the objects' movement (bounds on velocity, pre-defined trajectories), their geometry (solidity, convexity etc) and correctness or richness of the data structures involved. Therefore it is general purpose.

5.4.2 Performance and Efficiency Issues

One concern about using an underlying component technology is whether it brings a considerable computational overhead to hinder the performance of the software. As for COM, if the components are not communicating across the network this overhead is negligible. Once a client establishes a connection to a component, calls

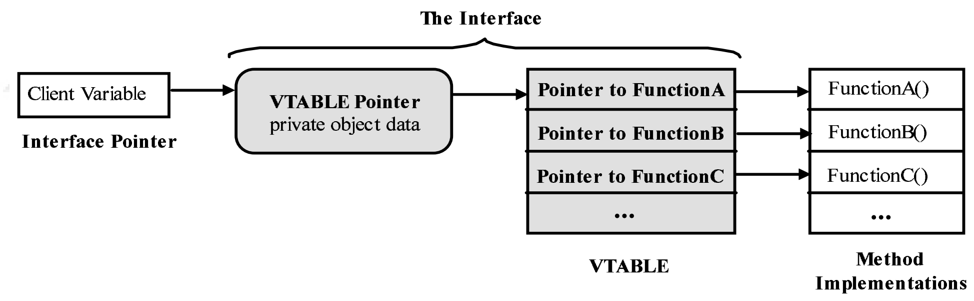


Figure 5.20: Accessing COM Interfaces Through a Virtual Table

to that component's services (interface functions) are simply indirect function calls through two memory pointers [Gri98a]. This is illustrated by a diagram in Figure 5.20. As a result, the performance overhead of interacting with an in-process COM object (an object that is in the same address space) as the calling code is negligible. Calls between COM components in the same process are only a handful of processor instructions slower than a standard direct function call and no slower than a compile-time bound C++ object invocation. In addition, using multiple interfaces per object is efficient because the cost of negotiating interfaces (via QueryInterface) is done in groups of functions instead of one function at a time.

Performance is an important issue in the simulation and animation fields. AniComp is primarily intended for in-process use, and therefore the use of COM does not incur significant overhead. Nevertheless other precautions have been taken to improve the efficiency of the procedures involved. These are as follows:

- In some cases the simulation client needs to pass many numerical values to the animator in an array. Array handling in COM, however, can be very tedious. Since the user cannot just pass a pointer to a C++ array (because of the language independence), a data type called SAFEARRAY has to be defined specifically, with the number of elements and their data type explicitly identified. This is necessary to facilitate a procedure called marshaling and un-marshaling of the function parameters. Constructing the arrays as SAFEARRAYs causes a reverse procedure at the receiving end, which involves identifying the base type, extracting the data and freeing the memory allocated to the SAFEARRAY. This obviously takes time. Therefore in AniComp's interfaces multiple versions of the functions containing array (matrix) parameters have been provided as options for the user. The user can choose the most efficient version if he/she calls the interface function frequently, e.g. in a loop.

  ```
  SetTransformMatrix16(BSTR objName, VARIANT MatArray16);
  SetTransformMatrixElem16(BSTR objName, FLOAT m00, FLOAT m01, FLOAT
  m02, FLOAT m03, FLOAT m10, FLOAT m11, FLOAT m12, FLOAT m13, FLOAT
  m20, FLOAT m21, FLOAT m22, FLOAT m23, FLOAT m30, FLOAT m31, FLOAT
  m32, FLOAT m33);
  ```

  The example above illustrates two versions of the function for sending a transformation matrix to the animator. The first one has a VARIANT type which contains a SAFEARRAY, whereas the second one sends the 4x4 matrix as 16 numerical values a of simple data type. The first one is more convenient if the local matrix to be passed has already been constructed with some other procedure and the efficiency is not a primary concern. The local C++ matrix is converted to SAFEARRAY (utility functions are provided for this purpose) and passed straight away. The second one, on the other hand, despite requiring the individual elements to be passed one by one, is more efficient.

- All the rigid bodies in the environment are visited and their physical values are integrated at every animation frame. This is computationally quite expensive. For some objects in the environment however this may not be necessary at all. Therefore in AniComp, for a geometric object to be regarded as a rigid body it has to be explicitly defined. This ensures that 3D objects are treated as physical objects only if it is necessary, which reduces the computational load.

- Collision detection causes further computational load. The V-Collide library used in AniComp's implementation cannot directly operate on Cosmo3D's data structures. It can only operate on its own primitive data type which is the triangle (a three point data structure). 3D objects imported from VRML files however are not guaranteed to be triangulated. They may contain higher-order polygons. This requires the objects to be tesselated (triangulated) and fed into V-Collide's own data structures. In addition, collision detection requires all the rigid bodies to be checked against a non-penetration constraint at every frame of the animation. This is a significant amount of overhead. In AniComp's interfaces, therefore, functions have been provided to register (and unregister) only selected rigid bodies for collision. For example if the scenario involves rigid body activities in a large environment and the user loses interest in checking some of the objects against collision based on a condition (e.g. objects being too far), then he/she can unregister them dynamically (at runtime) and relieve the animator from the associated computational load.

5.4.3 The Internal Logic of Creating Animations

In AniComp, the internal logic for modifying the states of the geometric objects is based on Cosmo3D's internal architecture. Cosmo3D's architecture, in turn, is based on the VRML internal execution model[4]. This execution model involves *event-in* and *event-out* fields associated with each node, some *routes* to connect these fields, and sensors for triggering events. Some VRML nodes generate events in response to environmental changes or user interaction. Event routing gives authors a mechanism, separate from the scene graph hierarchy, through which these

[4]This is quite natural because initial the VRML specification (version 1.0) was based on the Open Inventor specification from SGI (Silicon Graphics Inc) and Cosmo3D is a product of the same company.

events can be propagated to effect changes in other nodes. Once generated, events are sent to their routed destinations in time order and processed by the receiving node. This processing can change the state of the node, generate additional events, or change the structure of the scene graph.

Cosmo3D provides the csEngine class and its subclasses (interpolators) for the purpose of generating values to be fed to the event-in fields of other nodes. csEngine has input and output fields. It updates its output fields according to the function carried out by the csEngine. A csEngine object can be connected to other nodes or engines to create an animation, to cause a chain reaction of motions, or to cycle through a set of attributes, such as color cycling. A method called csContainer::connect() is used to make the connections. Among the 10 engines defined, 2 of them are particularly important from the animation point of view: csOrientationInterpolator and csPositionInterpolator. A csOrientationInterpolator interpolates between two rotations by computing the shortest path on the unit sphere between the two rotations. csPositionInterpolator linearly interpolates between sets of values in position vectors. This is appropriate for interpolating a translation. The vectors are interpreted as absolute positions in local space. Figure 5.21 illustrates these mechanisms. Based on these structures AniComp uses three techniques to generate animations:

1. Direct application of successive position and orientation updates to named objects. This is achieved by directly setting the translation and rotation fields of the transformation node of the object's geometry. The transformation node must be the immediate parent of the object's geometry node.

2. Animation of objects by rigid body simulation. If an object is declared to be a rigid body then it is associated with some physical values and its movement is governed by these physical quantities. On declaration a rigid body's position and orientation vectors are connected to the corresponding values (fields) in the transformation node of the object's geometry. At every integration step these values get updated automatically, which cause the objects to move accordingly.

3. Animation of objects by re-usable motion sequences. AniComp uses position and orientation interpolators as a basis for recording behaviours. It has internal classes (CBehaviour and CbehaviourList) which handle defining, recording, activating, saving and loading of behaviours. Each behaviour

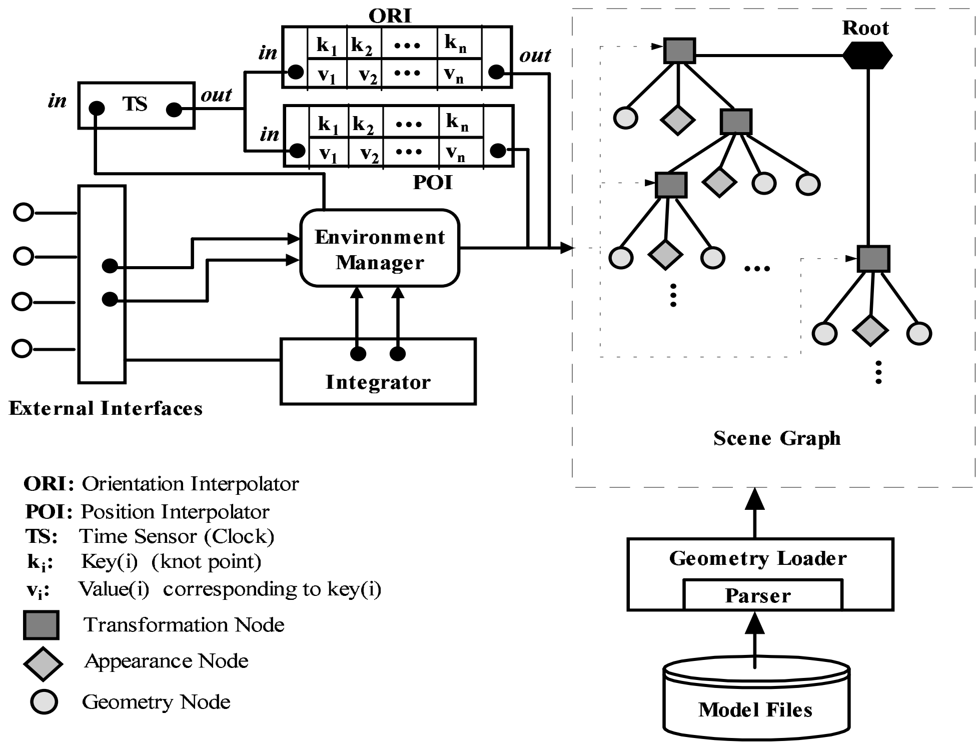


Figure 5.21: Motion Generation Mechanisms Based on VRML's Internal Execution Logic

has its set of interpolators, dedicated clocks and is attached to the object for which it is defined.

5.4.4 Name-Based Object Manipulation

AniComp provides a name-based interface for manipulating 3D objects. That is, to refer to an object, only its name is sufficient. The user does not have to know about the structure of the scene tree in detail. A broad knowledge of the conceptual hierarchy is enough to identify the correct object. This however requires the objects to be named uniquely. The name-based object manipulation brings about significant simplicity in dealing with animation. The standard VRML browsers provide an external authoring interface (EAI) for the manipulation of objects in a VRML world but this interface is quite difficult to use. The VRML 2.0 Specification sought to create a node structure that would facilitate optimization for traversal and added behavioral mechanics such as routes, Script nodes, and an external authoring interface that a VRML viewer is to export to the Web browser. Regardless of the method that authors use to add behaviors to their scenes, they must structure and understand their scene graph. Authors must either create routes between the appropriate fields in the scene, or use a supported scripting language to retrieve the appropriate fields from the nodes of interest. Unfortunately, most VRML files require very deep hierarchies to obtain the placement, articulation, inline, and level of detail hierarchies needed to develop complex scenes. Compounding this fact with the many parameters required to describe geometries, it is difficult to examine a VRML file and understand the relevant structure. If the author has created a human figure that he wishes to articulate, he may be interested only in the rotation fields of the transforms representing joints. Here, the many other fields and values in the scene are merely noise [Bee].

The complexity is magnified by the fact that compelling VRML content requires a wide range of skills; thus a VRML world is frequently the product of a team. Worlds are often created by a group of individuals who specialize in modeling, code, images, or sound. If one member of a team is responsible for the code governing behaviors and another member of the team modifies the scene structure, the code can be made obsolete. Because the current Java and JavaScript access methods to the scene graph are based on fetching nodes and the fields of those nodes, any changes to the scene may break the code accessing that scene. Analogies exist for other specialists interactions, each pointing to the fact that the behavior of complex worlds is precariously dependent on the scene graph structure.

Because of these considerations a simple name-based interface has been designed which hides the complexity of the scene graph from the user.

```
void CBallBeamSet::ObtainInitialPositions() {
    Vec3 posVec, EllipsePos, BeamPos;
    VARIANT posVar, EllPosVar, BeamPosVar;
    posVar = m_pAnimator->GetObjectPosition(CString("Ball"));
    EllPosVar = m_pAnimator->GetObjectPosition(CString("Ellipse"));
    BeamPosVar = m_pAnimator->GetObjectPosition(CString("Beam"));

}
```

```
void CBallBeamSet::Simulate(float time) {

    // do the calculations to determine position and orientation
    // of the individual objects to be animated
    m_pAnimator->RotateObjectRelative(CString("Ball"),0,1.0,0,w);
    m_pAnimator->SetObjectPosition(CString("Ball"),x,y,z);
    m_pAnimator->RotateAroundPoint(CString("Ellipse"), Ex,Ey,Ez,
                                   tVecVar,DeltaAng);
    m_pAnimator->RotateObjectRelative(CString("Gear"),0,0,1.0,
                                      deltaGearAngle);
}
```

The code fragment given in the example above shows this simplicity. These two functions are taken from a real implementation of a simulation engine (to be presented in chapter 6) which uses AniComp for displaying the animated behaviour of the objects it is simulating. The first function obtains the initial positions of three objects with names Ball, Ellipse and Beam. The second function is the main simulation loop in which successive updates are calculated and applied to the objects with the same names.

Name-based object manipulation entails a certain amount of overhead because every time the object is referred to, its name has to be looked up in a table which keeps a pair of a name and its associated pointer to the actual node in the scene tree. However this is not a significant problem as the lookup procedure is not an inefficient sequential search. A hash table is used to implement the name dictionary and this is a proven, fast method for name lookup tasks [Nyh88, Kru91].

5.4.5 Creating and Using Behaviours

Basic steps in creating and using behaviours (motion sequences) were introduced in section 5.3.3. To exemplify the use of this technique, the simulation model of a 6DOF (Degree of Freedom) robot arm was implemented. This section first presents the robot model and then explains how it was used to create re-usable motion sequences.

The Model of the Robot Arm

The 6DOF robot arm is a sufficiently complex example for demonstrating the definition and activation of named behaviours. The robot modeled here consists of 5 physical links. The sixth degree of freedom was assigned to the last link, making it capable of moving with two degrees of freedom. All joint movements are assumed to be rotational. The relations between the links were modeled using

theta	d	a	α
*	0	0	0
*	0.56	0	90
*	0	0.48	-90
*	0	0.35	90
*	0	0.20	-90
*	0	0	90

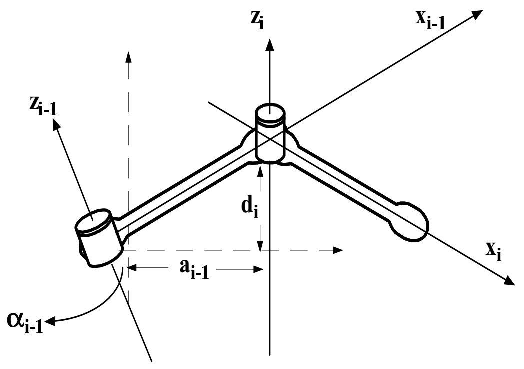


Figure 5.22: DH Parameters for 6DOF Robot Arm and Their Representation

the Denavit-Hartenberg(DH) notation [Wat92]. This notation is widely used in the field of robotics. The table in Figure 5.22 shows the values assigned to DH parameters for this specific robot arm as well as a diagrammatic interpretation of the parameters. The parameters have the following meaning:

- a_i is the distance from z_i to z_{i+1} measured along x_i (the length of the link).
- α_i is the angle between z_i and z_{i+1} measured about x_i (the twist of the link).
- d_i id the distance between the x_{i-1} and x_i axes measured along z_i (the distance between links).
- θ_i is the angle between x_{i-1} and x_i measured about z_i

Using forward kinematics on a structure defined by the DH parameters, 4 transformations need to be applied to relate each coordinate frame assigned to a joint to its neighbouring frames. The concatenation of these four transformations yields the following matrix:

$$T_i = \begin{bmatrix} \cos\theta_i & \sin\theta_i \cos\alpha_{i-1} & \sin\theta_i \sin\alpha_{i-1} & 0 \\ -\sin\theta & \cos\theta\cos\alpha_{i-1} & \cos\theta\sin\alpha_{i-1} & 0 \\ 0 & -\sin\alpha_{i-1} & \cos\alpha_{i-1} & 0 \\ a_{i-1} & -d\sin\alpha_{i-1} & d\cos\alpha_{i-1} & 1 \end{bmatrix} \tag{5.1}$$

All frame-to-frame transformations can be concatenated to form a single transformation matrix that links frame 0 to frame N:

$$T_N^0 = T_1^0 T_2^1 ... T_N^{n-1} \tag{5.2}$$

This transformation is a function of all the joint variables and will give the Cartesian position and orientation of the last link.

Creating Re-usable Behaviours for the Robot Arm

The robot arm simulator reads the DH parameters from a file and builds the necessary internal data structures including frame matrices. As an example, to create a ”pick object” behaviour for the robot, first each link needs to be assigned a name. These names must be the same as the names given when building the geometric model. For each link a behaviour must be defined to record the changes in position and orientation values during the course of a specified movement. The following function, taken from the member functions of the CRobotArm class, illustrates how behaviour definition is performed by calling the appropriate function of the AniComp's interfaces.

```
void CRobotArm::DefineLinkBehaviours() {
    // for each link define a named behaviour
    string ObNm, BhNm;
    for (int i=1; i<NR_LINKS; i++)
    {
        ObNm = m_ListOfLinks[i-1].GetName();
        BhNm = m_ListOfLinks[i-1].GetBehaviourName();
        m_pAnimator->DefineBehaviour(ObNm.c_str(),
                     BhNm.c_str());
    }
}
```

After defining the behaviours successive changes in the joint values need to be recorded. This can be done by sending the transformations for each joint to the animator, each time the frame matrices are calculated. The following function gets executed every time a joint value changes. This function couples the current values (elements) of the frame matrices with a key which represents the relative position of the these values within the whole recording process. It then passes these (key, matrix) pairs to the animator.

```
void CRobotArm::RecordLinkMovements() {
    float Rot[4], Trans[3], Key = -1.0;
    string ObNm, BhNm;
    float Row1[4], Row2[4], Row3[4], Row4[4] ;
    VARIANT r1,r2,r3,r4;

    for (int i=1; i<NR_LINKS; i++)
    {
        // [code described in the following two lines was omitted ...]
        // for each link, assign  its frame matrix
        // to row vectors r1,r2,r3 and r4

        ObNm = m_ListOfLinks[i-1].GetName();
        BhNm = m_ListOfLinks[i-1].GetBehaviourName();

        // use r1,r2,r3,r4 as parameters to the following function
        // from AniComp's interface. This function extracts
        // the translation and rotation values  and adds an
        // item to position and orientation interpolators
        // for each key value given as the other parameter

        m_pAnimator->TransfMatrixToInterpolators(ObNm.c_str(),
                     BhNm.c_str(), Key, r1, r2, r3, r4);
    }

}
```

The user may chose to set the key values to -1 in which case two utility functions called RearrangePosInterpKeys and RearrangeOrientInterpKeys, can be used to generate the keys automatically as equally spaced fractional values ranging from 0 to 1. This is illustrated below:

```
void CRobotArm::ConfigureRobotBehaviour() {
    string ObNm, BhNm;
    CStringList aList;
    VARIANT vRStrL;
    for (int i=1; i<NR_LINKS; i++)
    {
        ObNm = m_ListOfLinks[i-1].GetName();
        BhNm = m_ListOfLinks[i-1].GetBehaviourName();
        m_pAnimator->RearrangePosInterpKeys(ObNm.c_str(),
                     BhNm.c_str());
        m_pAnimator->RearrangeOrientInterpKeys(ObNm.c_str(),
                     BhNm.c_str());
        aList.AddTail(BhNm.c_str());
    }
    StrListToVARIANT(&vRStrL, &aList);

    // synchronize the movement of all the links by assigning
    // the same time sensor as their clock, and setting the
    // same time interval for their corresponding motion sequence

    m_pAnimator->SynchronizeBehaviours(vRStrL,
                 CString("RoboClock"), 15.0);

}
```

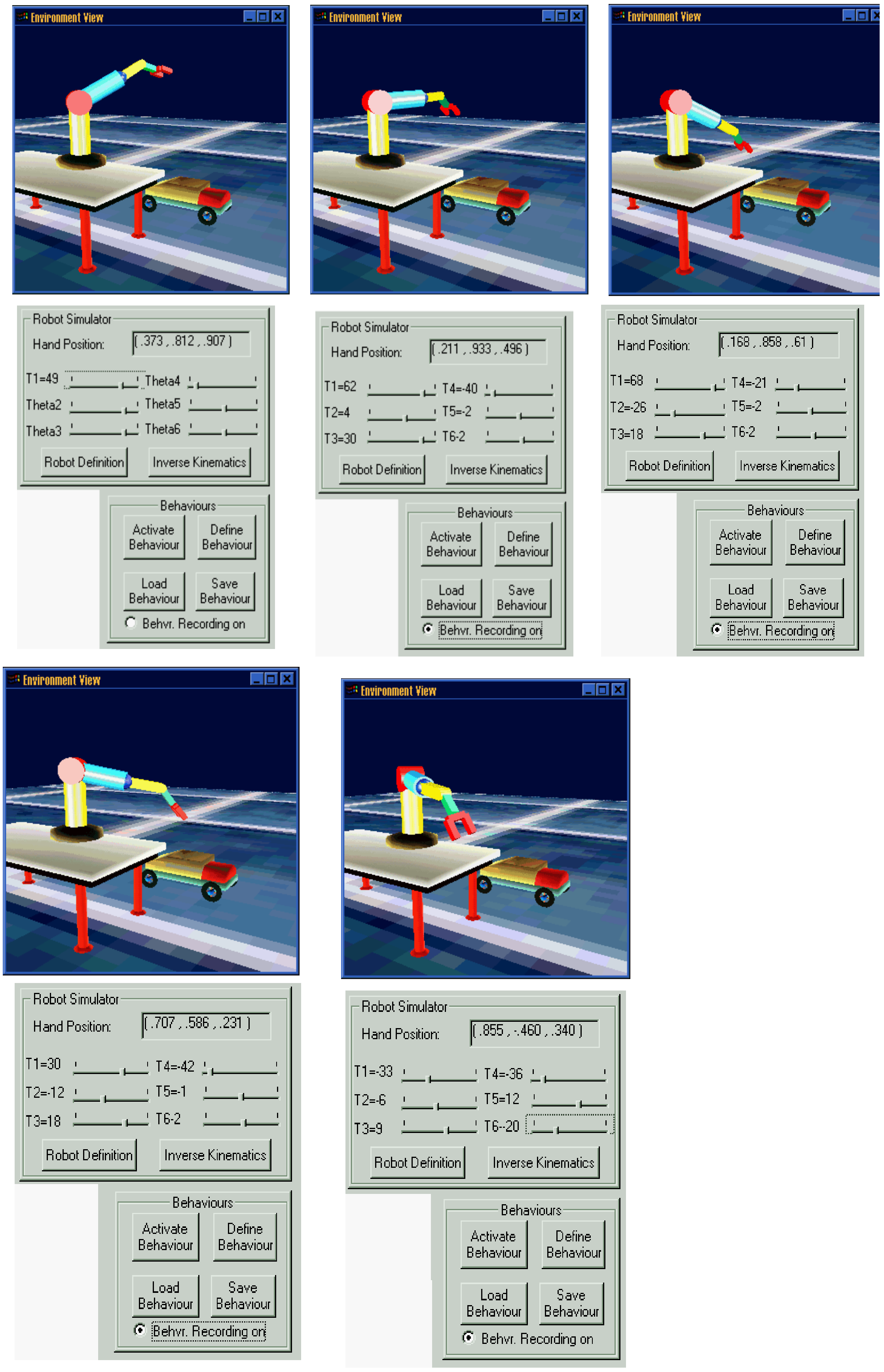
The recorded behaviour could then be activated by calling the ActivateBehaviour function of the AniComp interface. This is shown in the following code fragment:

```
void CRobotArm::ActivateRobotBehaviour() {
    // activate the behaviour of each link
    string ObNm, BhNm;
    for (int i=1; i<NR_LINKS; i++)
    {
        ObNm = m_ListOfLinks[i-1].GetName();
        BhNm = m_ListOfLinks[i-1].GetBehaviourName();
        m_pAnimator->ActivateBehaviour(ObNm.c_str(),
                        BhNm.c_str());
    }
}
```

The process of recording a "pick object" behaviour is visually illustrated in Figure 5.23. As the links are moved by controlling the joint angles using the sliders provided by the user interface, the transformation matrices for each link gets calculated and passed to the Animator. Note that at each frame the theta (θ) values (joint angles) are different and the overall configuration of the links changes accordingly.

5.5 Summary

In this chapter, first, a design methodology was devised and applied to the specification of the AniComp's interfaces. The internal design of AniComp was also presented starting from a top-level architecture diagram detailing down to the collaboration of the internal objects, aiming to sketch both a static and a dynamic view. This was followed by the discussion of some of the implementational issues, and the routes taken were explained. In the last section, a detailed implementation example of using AniComp's behaviour interface was illustrated. The next chapter presents a more complete application which uses AniComp in a component framework specifically designed to support AniComp's usage in component based simulation environments.



Hand Position: [.373 , .812 , .907]
T1=49 Theta4
Theta2 Theta5
Theta3 Theta6

Hand Position: [.211 , .933 , .496]
T1=62 T4=-40
T2=4 T5=-2
T3=30 T6-2

Hand Position: [.168 , .858 , .61]
T1=68 T4=-21
T2=-26 T5=-2
T3=18 T6-2

Hand Position: [.707 , .586 , .231]
T1=30 T4=-42
T2=-12 T5=-1
T3=18 T6-2

Hand Position: [.855 , -.460 , .340]
T1=-33 T4=-36
T2=-6 T5=12
T3=9 T6-20

Figure 5.23: Recording Robot Behaviour by Forward Kinematics

Chapter 6

Testing AniComp in a Framework

6.1 Introduction

A component framework is defined as a software entity that supports conforming to certain standards and allows instances of these components to be 'plugged' into the component framework [Szy98a]. Szyperski argues that to avoid the immediate

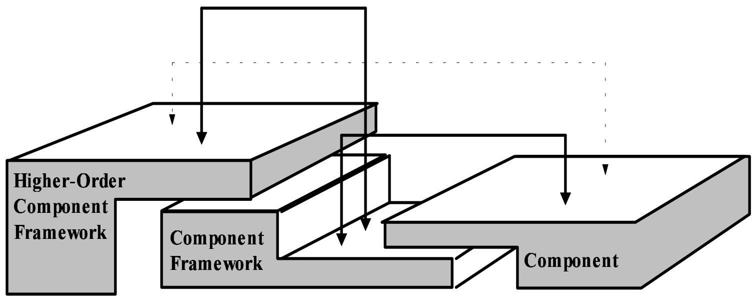


Figure 6.1: The multi-tier Component Framework Architecture

trap of frameworks insisting on overall control, the architecture can be extended recursively: a component framework itself can slot into a higher tier framework that regulates interaction [Szy98c]. He suggests that component frameworks can themselves be implemented as components that collaborate with other frameworks in a multi-tier architecture, as illustrated in Figure 6.1.

At this point, it is important to note the difference between class frameworks, or white-box frameworks, and component frameworks, or black-box frameworks. A class framework is represented by a set of abstract classes and the way their instances interact [Joh97]. Class frameworks are re-used at source code level, normally through sub-classing (inheritance) [Fay97]. A component framework, on the other hand, is represented by a set of collaborating binary components. By construction, a component framework accepts dynamic insertion of component instances at run-time. It also supports the partial enforcement of architectural principles by forcing the component instances to perform certain tasks.

This chapter presents a light-weight component framework that has the functionality of run-time contract validation, and can impose a minimal set of interactions between simulation components. The framework regulates the interactions of the simulation components with the animation component, AniComp, design of which was discussed in Chapter 5. In the remainder of the chapter, the design and the implementation of the MInimal Simulation Animation Framework (MISAF) is introduced. Then, its usage is exemplified in an application: The Virtual Lab. The MISAF is itself implemented as a component. Therefore, it can be plugged into higher order frameworks. In this respect, it aims to provide a contribution to the realization of the idea of multi-tier component frameworks mentioned above.

6.2 The MInimal Simulation Animation Framework (MISAF)

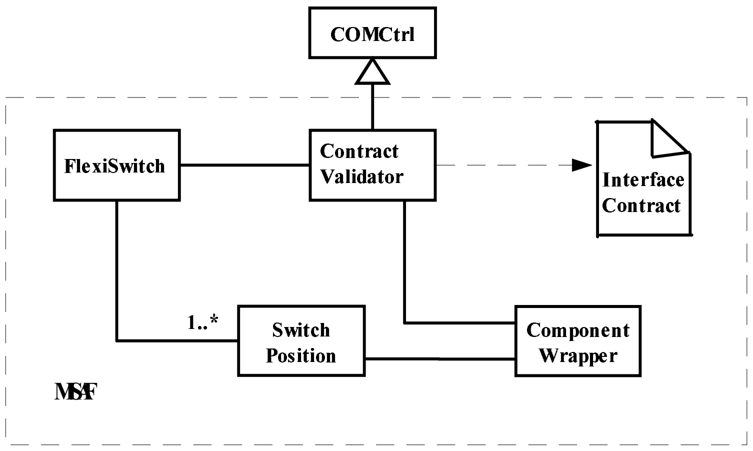


Figure 6.2: The class diagram of the MInimial Simulation Animation Framework

The MISAF consists of two main structures: a run-time contract validator and a flexi-switch. The run-time contract validator contains a minimal set of interface function specifications hard coded into the ComponentWrapper class. A class diagram of the structures involved is given in Figure 6.2. The ContractValidator uses the meta-information contained in the type libraries of the COM components to understand whether a particular component has an interface implementing the functions specified in the contract. This procedure involves testing the individual parameters and parameter types that are supposed to exist in the argument list of the specified interface functions. The list of functions in the interface contract is given below:

```
BOOL InitializeComponentEnv(LPDISPATCH pAnimDisp, LPDISPATCH pFlexiSwitch);
BOOL LoadVirtInstrumentModel(float x, float y, float z);
BOOL StartIndependentSimulation( );
BOOL StartSimulation( );
BOOL StopSimulation( );
BOOL DoCommand(CString cmdStr,short paramNum, CString paramTypeStr, CString
paramValueStr);
```

After testing a component's interface, the framework performs an initialization procedure to ensure that further interactions with the component are valid. It calls the InitializeComponentEnv interface function with two parameters. The first parameter is the interface pointer of the animation component (AniComp) and the second parameter is a pointer to the frameworks's flexi-switch interface. This second pointer enables the components to access other components' interfaces through the framework. The second function call is made to load the geometric model of the object being simulated. The last function, DoCommand, is a generic call to the simulator's behaviour interface. It contains a command name, the number of the parameters to be passed to the command, the types of the parameters in a string, concatenated and delimited by commas; and the parameter values themselves, again packed into a string and delimited by commas in the same order as the types. This enables the simulator components to call other simulators' functions specific to those simulators, and for which the framework makes no assumptions. For example, if a component simulating a robot arm has a number of behaviours exposed in the form of functions such as Pick-Object, DropObject, GoToHomePosition etc., other participating components (in the run-time environment governed by the framework) would not be able to access these functions through the framework since no assumptions are made about the participants' interfaces except the minimal contract presented above. Each component, then, can implement a DoCommand interface function which parses the string passed as the parameters to that function, identifies the request and calls its internal function which implements the corresponding behaviour. This provides a loose, yet, flexible mechanism for components to trigger one another's behaviours, if required. A typical scenario where this can be desirable, is that of manufacturing systems. An automated guided vehicle may want to trigger a robot arm's pick behaviour on its arrival to the robot arm's operating point.

The second structure, the flexi-switch, is a class which provides mechanisms for run-time management of the binary components being accepted to the framework. It implements a dynamic list of component wrappers that can grow and shrink. It can make a name based search for a requested component and deliver a reference to it. Participating components can make use of this facility to access each other's interfaces.

6.2.1 Discussion: The Simplicity of the Framework

The MISAF is deliberately built as a simple framework both in terms of the mechanisms involved and the complexity of the interface contract being validated. This has some advantages as well as disadvantages. The disadvantages mainly stem from the fact that imposing only a minimal set of regulations may hinder the completeness and robustness of the overall framework. The framework could be based on one (or more) of the established simulation modeling standards to regulate the component interactions to a finer level of detail which would then ensure a consistent and robust collaboration scheme for the components in a given simulation modeling framework.

As for the advantages; being simple, the framework makes a minimal set of assumptions about simulators' internal architecture. One assumption is that they must interface to AniComp as a visibility tool *if* they want to render themselves (with their behaviour) in the same virtual environment as the other participants. The StartIndependentSimulation function in the interface, drops even this requirement, but the components, then, have to handle initialization, rendering and user-interaction in an isolated environment. The other assumption is that if the simulation requires user interaction and control then it must provide a user-interface for that and must open it in the initialization phase. And finally, *if* the component wants to expose its behaviour interface to other components, it has to implement the DoCommand function. Being a loose agreement, the contract leads to a flexible framework. As it will be seen later in this chapter during the discussion of the Virtual Lab application, a monolithic package (MATLAB/Simulink) was easily integrated into the framework by writing a simple adapter, and it was used alongside another simulation component with completely different internal characteristics.

6.3 The Virtual Lab Application: Testing AniComp within MISAF

The Virtual Lab application was developed to exemplify the usage of AniComp and MISAF by illustrating the collaboration of simulation and animation components in a component framework. Using the application, the user can dynamically add rooms and corridors to the virtual environment, controlled by a layout manager internally. It is then possible to insert instruments into rooms. Each instrument is coupled by a component which simulates and controls its functionality and animates it using AniComp. The simulation components are required to conform to the interface contract mentioned above. Figure 6.3 shows the class

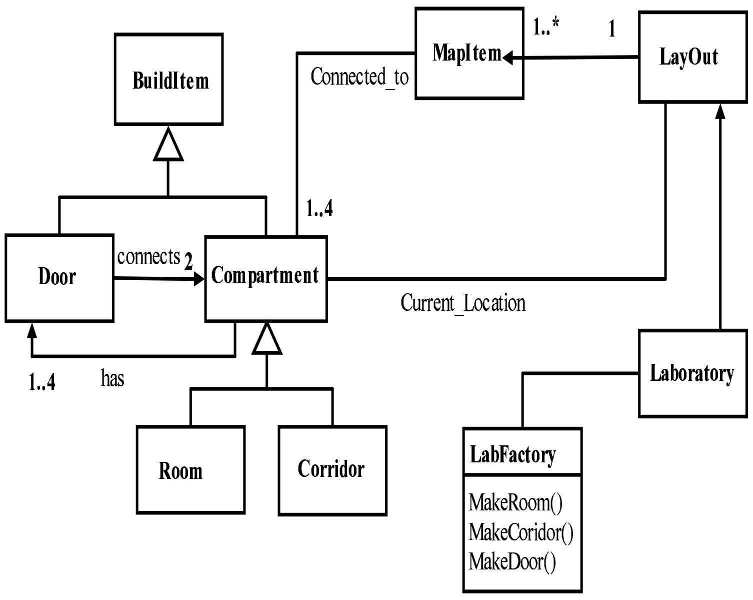


Figure 6.3: A Top Level Class Diagram of Virtual Lab Application

diagram representing the overall design of the program. The LayOut class maintains a map of the laboratory. The Laboratory class utilizes the LabFactory class to create instances of BuildItem class which might either be a room, a corridor or a door.

The Virtual Lab application employs MISAF to handle the dynamic plugging (containment) of the incoming simulation components. Each time the geometric model of an instrument is added to the laboratory, the component that simulates its behaviour is also integrated to the environment. The framework is itself a binary component and presents the following interface to its acquaintances, as it appears in its COM IDL file:

```
dispinterface _DMISAFFramework {
    methods:
    // NOTE - ClassWizard will maintain method information here.
    //    Use extreme caution when editing this section.
    //{{AFX_ODL_METHOD(CMISAFFrameworkCtrl)
    [id(1)] short GetComponentCount();
    [id(2)] VARIANT GetSysRegisteredComponents();
    [id(3)] VARIANT GetComponentListInFramework();
    [id(4)] void InitializeComponentEnv(BSTR ComponentName,
            IDispatch* pAniComp);
    [id(5)] void LoadObjGeometricModel(BSTR ComponentName,
            float xPos, float yPos, float zPos);
    [id(6)] void StartSimulationComponent(BSTR ComponentName);
    [id(7)] void StopSimulationComponent(BSTR ComponentName);
    [id(8)] short ComponentDoCommand(BSTR ComponentName,
            BSTR cmdStr, long paramNum, BSTR paramTypeStr,
            BSTR paramValueStr);
    [id(9)] boolean TestMISAContract(BSTR ComponentName,
            IDispatch* pAniComp);
    [id(9)] LPDISPATCH GetPointerToComponent(BSTR ComponentName);
    //}}AFX_ODL_METHOD
    [id(DISPID_ABOUTBOX)] void AboutBox();
};
```

Each function in the interface contains a parameter (ComponentName) to indicate which component is targeted for a specific operation. This name is (must be) a unique name which represents the component in the system (Windows) registry. The GetSysRegisteredComponents function is a utility function which finds all the registered components in a system by searching the Windows registry and returns this list. The GetPointerToComponent function returns a dispatch interface pointer of a component whose name is specified. This is useful if components wish to access other components' interfaces for such tasks other than the ones specified by the framework. The interface returned is the *dispatch interface* defined by COM belonging to the specified component. This interface allows other components to find out (query) which methods the component's interface supports at run-time [Gri98b].

It is worth noting that the architecture of the framework can potentially be generalized to handle the management of component interactions based on other contracts as well. This can be achieved by allowing the framework to import any interface contract rather than hard coding it into the framework. This, however, requires a standardized way of defining the contracts and also the initial interaction protocol that has to take place. Some preliminary considerations on how this can be achieved are made in chapter 7 as part of the suggestions made for future work. Figure 6.4 illustrates the usage of MISAF within the Virtual Lab

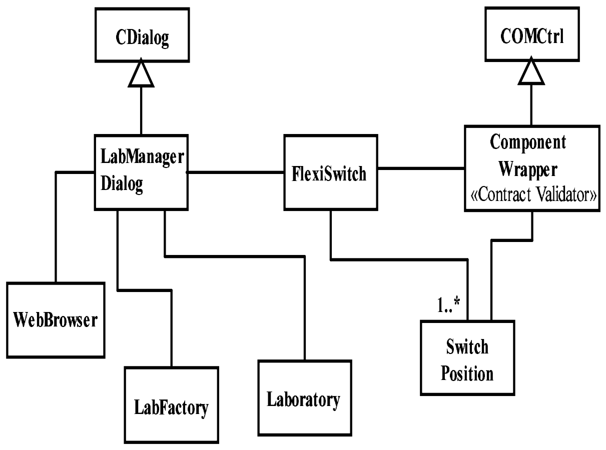


Figure 6.4: The class diagram for FlexiSwitch structure

application. Run-time contract validator and flexi-switch are used to implement the addition of instrument components to the virtual laboratory at run-time.

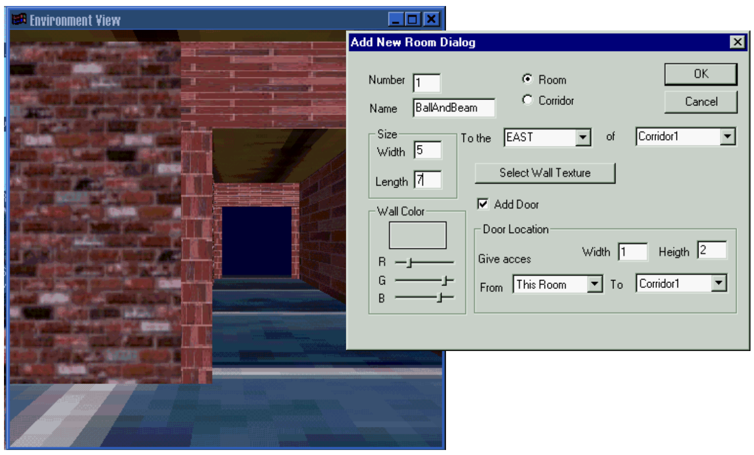
6.3.1 Using the Virtual Lab in a Web-Enabled Environment

The Virtual Lab application allows the user to integrate instrument components from anywhere on the internet. This achieved by an integrated web browser which appears in the dialog window responsible for selecting the instrument components. For this, the component must be defined as follows:

```
<object ID="BallAndBeam"
   CLASSID="clsid:48D7784E-A1A6-11D3-B8F7-81C1C89C090F"
   CODEBASE="http://www.alpdemir.fsnet.co.uk/BallBeamTest1OCX.ocx
   #version=1,0,0,0">
</object>
```

The above definition must be embedded in an HTML file and must be browsed by a browser which supports the DHTML (Dynamic HTML) standard. The integrated browser is compatible with DHTML.

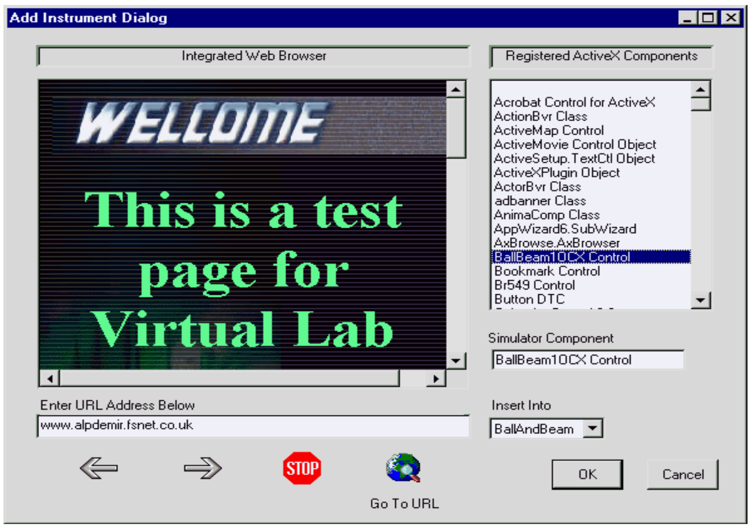
To demonstrate the various steps of building a virtual lab in a web-enabled environment, several screen shots are included below, coupled with an explanation of the processes involved. Figure 6.5 shows the initial view of the lab and the



Number: 1
Name: BallAndBeam
[✓ Room] Corridor
Size — Width: 5, Length: 7
To the: EAST of: Corridor1
[✓ Add Door]
Door Location — Give acces — Width: 1, Heigth: 2
From: This Room To: Corridor1

Figure 6.5: The Initial View of the Lab And the Dialog For Adding New Room

dialog window for adding a new room. It is possible to specify attributes of the room such as size, color, texture of the walls, size and position of a door. Figure 6.6 illustrates the "add instrument dialog window" which contains an integrated web browser. The dialog, when first opened, presents the user with a list of all



Acrobat Control for ActiveX
ActionBvr Class
ActiveMap Control
ActiveMovie Control Object
ActiveSetup.TextCtl Object
ActiveXPlugin Object
ActorBvr Class
adbanner Class
AnimaComp Class
AppWizard6.SubWizard
AxBrowse.AxBrowser
BallBeam1OCX Control
Bookmark Control
Br549 Control
Button DTC

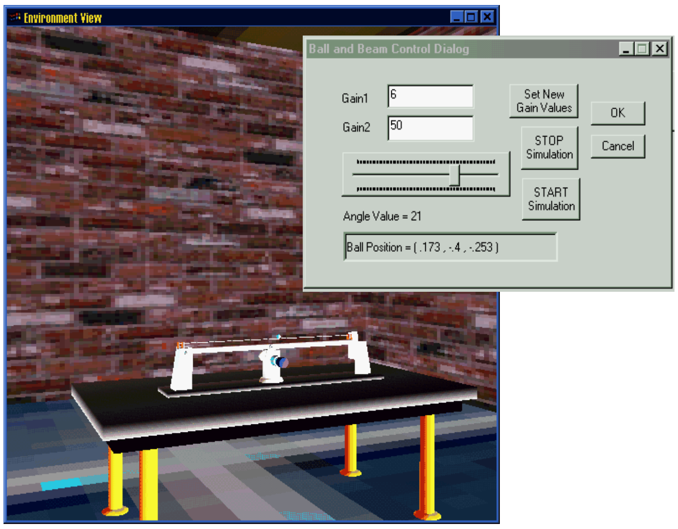
Simulator Component: BallBeam1OCX Control

Enter URL Address Below: www.alpdemir.fsnet.co.uk

Insert Into: BallAndBeam

Figure 6.6: The Dialog For Adding A New Instrument Component. This Dialog Contains An Integrated Web Browser.

registered components in the system so that the user can select the instrument component easily. If the desired component is not available locally, then the user can submit an URL address that contains the component, to the integrated web browser. The browser then accesses the given page, locates the component; downloads, installs and registers it automatically. It then refreshes the list of registered components in the system so that the new component is now selectable. The user then selects the components and specifies the room to be inserted in. Upon the selection the framework checks the component interface against the hard-coded contract. If satisfied the component gracefully integrates into the environment and presents its own instrument control dialog to the user. Figure 6.7 shows the ball and beam instrument inserted into a room, ready to be experimented with. The component simulating the behaviour of the virtual ball and beam instrument was implemented as an ActiveX [Arm97] component in C++. The simulation and control modeling of the ball and beam instrument will be presented later in this chapter. Figure 6.8 shows the second instrument, the virtual inverted pendulum, inside another room. The simulation process of the inverted pendulum is different from that of ball and beam. The simulation component, which is again an ActiveX component implemented in Visual Basic, does not actually hold the simulation code. Instead it simply connects Matlab/Simulink engine to the application by acting as an *adaptor component* [Kuc98] eliminating the



Gain1 6
Gain2 50
Angle Value = 21
Ball Position = (.173 , -.4 , -.253)

Figure 6.7: The Virtual Ball And Beam Instrument in A Room

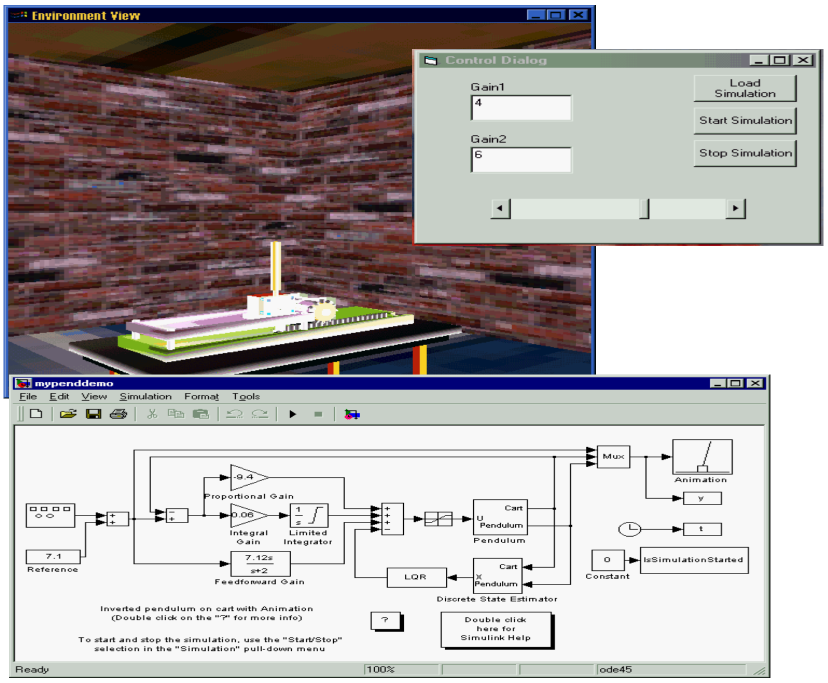
incompatibility between matlab's automation interface and the MISAF.

6.4 Modelling the Instruments

6.4.1 The Ball and Beam Instrument

A ball and beam apparatus is a simple mechanical system but demonstrates a difficult control problem. It consists of a rigid beam, which is rotated around a center pivot, and a solid ball rolling between two ends of the beam. The control problem can be described as positioning the ball at a desired point on the beam using the angle of beam as an input. The nature of this problem can be given by assuming that a mass is sliding on a frictionless parallel to the plane, which has a certain angle as shown in Figure 6.9.

The physical model of the ball end beam can be given using variational methods [Wel78]. The beam angle α is controlled by a dc-motor and an ellipsoid prism rotating around a point different from its origin. Therefore, different positions of ellipse changes the angle α. Since the system contains mechanical and electrical components, the use of Lagrange's equation provides a systematic unified approach for handling this system. The general form of Lagrange's equation can be given as follows:



Inverted pendulum on cart with Animation
(Double click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop"
selection in the "Simulation" pull-down menu

Figure 6.8: The Virtual Inverted Pendulum Instrument in Another Room

$$\frac{d}{dt}\left(\frac{\delta L}{\delta \dot{q}_1}\right) - \frac{\delta L}{\delta q_i} + \frac{\delta D}{\delta \dot{q}_i} = Q_i, \;\; i = 1, 2, 3, ..., n \tag{6.1}$$

where n represents the number of independent coordinates or degree of freedom in the system. L is defines as

$$L = U^* - T \tag{6.2}$$

where U^* is the total kinetic co-energy of system, T is the total potential energy of system, D represents co-content related to the energy, which is dissipated as heat (produced by friction in mechanical systems and by resistance by electrical systems.) Q_i corresponds to the force functions applied to system. In the mechanical system, Q_i can be forces and torques applied externally in the direction of generalized coordinates, and in the electrical systems, it appears as voltage and current sources.

In the ball and beam apparatus, the mechanical and electrical components can be decoupled easily. The transfer function between the voltage $\theta(s)$ controlling

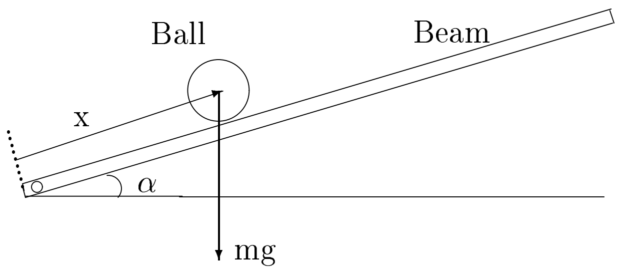


Figure 6.9: Simple illustration of ball and beam system

the field current of the dc-motor and the angle of the ellipsoidal prism can be given as follows:

$$\frac{\theta(s)}{V_f(s)} = \frac{K_m}{s(Js+f)(L_f s+R_f)} = \frac{K_m/fR_f}{s(\frac{J}{f}s+1)(\frac{L_f}{f}s+1)} \tag{6.3}$$

where J is the inertia of the ellipsoidal prism, f is the friction and K_m is the motor constant.

In the experiment, two control loops are used; one for controlling the angle of the motor, and one for controlling the position of the ball. In the first loop, the angular position of the ellipsoidal prism is set to a reference value using a simple position control mechanism. The main disturbance for the angle $\theta(s)$ is the effect of the disturbance torque. This closed loop is shown in the following diagram in Figure 6.10. The controller is a simple PI (Position-Integral) control.

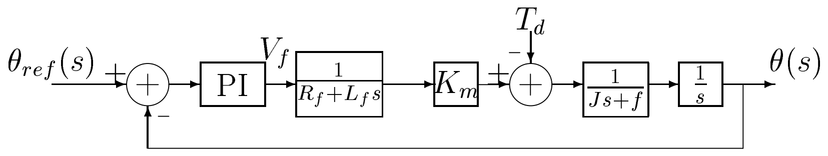


Figure 6.10: Block diagram of inner loop control for $\theta(s)$

The model of the dc-motor part of the apparatus can simply be found using the basic relationships between electrical and mechanical components of the dc-motor. In the ball and beam part, two independent coordinates; namely the position of the ball and the angle of beam are chosen in order to use variational methods. Here q_1 and q_2 can independently determine the position of the system. Furthermore, the ball position and the beam angle are not geometrically constrained. So, the pair $(q_1\ q_2)$ are a complete independent set of variational coordinates.

The kinetic co-energy of the system U^*, which is related to ball mass m, the ball inertia I_b and the beam inertia I_a can be given as follows:

$$U^* = \frac{1}{2}mv^2 + \frac{1}{2}I_b w^2 + \frac{1}{2}I_a \dot{q}_2^2 \tag{6.4}$$

where v is the translational velocity of the ball and w is the rotational velocity of the ball. v has two components; namely, the derivation of x and multiplication of x with the derivation of *alpha*, since the ball is both moving in the direction of x and on the arc calculated by $x\alpha$ at the same time. Therefore, this speed can be expressed as follows:

$$\begin{aligned} v &= \sqrt{\dot{q}_1^2 + (q_1\dot{q}_2)^2} \\ w &= v/r + \dot{\alpha} \end{aligned} \tag{6.5}$$

where r is the radius of the circle related to contact point of the ball in the beam.

In fact, $q_2 = \alpha$ is extremely constrained by the external input, so $\delta q_2 = 0$. Therefore, Lagrangian of the system can given as follows:

$$L = \frac{1}{2}[m(\dot{x})^2 + (x\dot{\alpha})^2) + I_b(\frac{\dot{x}}{r} + \dot{\alpha})^2 + I_a(\dot{\alpha})^2] \tag{6.6}$$

and Lagrange's equations of motions are

$$\begin{aligned} \frac{d}{dt}(\frac{\delta L}{\delta \dot{q}_1}) - \frac{\delta L}{\delta q_1} + \frac{\delta J}{\delta \dot{q}_1} &= F_1(t) \\ \frac{d}{dt}(\frac{\delta L}{\delta \dot{q}_2}) - \frac{\delta L}{\delta q_2} + \frac{\delta J}{\delta \dot{q}_2} &= \tau_2(t) \end{aligned} \tag{6.7}$$

where J is the system's co-content, $F_1(t)$ is the generalized force in the direction of $q_1 = x$, and $\tau_2(t)$ is the generalized torque in the direction of $q_2 = \alpha$.

$F_1(t)$ is equal to the component of the ball weights in the direction of q_1. The generalized torque is composed of the components of torque contributed by the external forces acting about the beam pivot. The first component is due to the dc-motor torque, and the second force is due to gravitational force on the ball

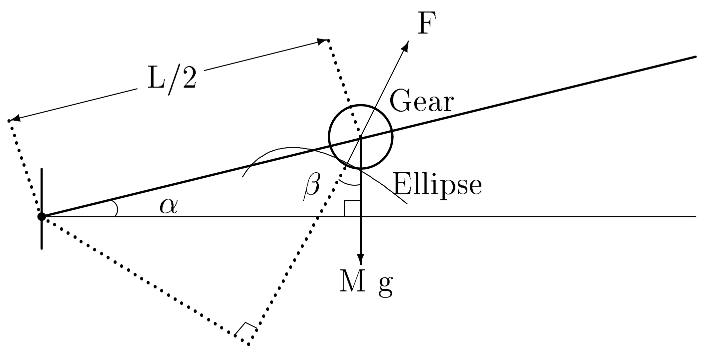


Figure 6.11: Evaluation of the generalized torque

and beam. Indeed, the ball weight can be neglected beside the weight of beam.

The generalized torque can be calculated using the diagram given in Figure 6.11. In this figure, two forces acting on the contact point of ellipsoidal prism and the small gear cause a torque around Point A. Therefore, the expression for the external force and torque can be calculated as follows:

$$\begin{aligned} F_1(t) &= mg\sin(\alpha) \\ \tau_2(t) &= F((L/2)\sin(90-\alpha-\beta) - Mg(L/2)\cos(\alpha) \end{aligned} \tag{6.8}$$

where F is the contact force, which is produced by the torque of the dc-motor, and M is the mass of beam (the mass of ball is neglected).

Since there is no energy storage element and friction the effort storage energy $T = 0$ (no string) and $f = 0$ (no friction). Hence, by solving Equations in 6.7, the model of the system is given as follows:

$$\begin{aligned} &(m + \frac{I_b}{r^2})\ddot{x} + \frac{I_b}{r^2}\ddot{\alpha} - mx(\dot{\alpha})^2 = F_1(t) \\ &(mx^2 + I_b + I_a)\ddot{\alpha} + (2mx\dot{x} + bl^2)\dot{\alpha} = \tau_2(t) \end{aligned} \tag{6.9}$$

In this model, the influence of $\ddot{\alpha}$ and $\dot{\alpha}$ can be neglected, and I_b and m can be neglected beside I_a. For small α, $\sin(\alpha) = \alpha$ and $\cos(\alpha) = 1$. Hence, the simplified linear model can be given as follows:

$$\begin{aligned} &(m + \frac{I_b}{r^2})\ddot{x} = mg\alpha \\ &I_a\ddot{\alpha} = (L/2)[F\sin(\pi/2-\alpha-\beta) - Mg] \end{aligned} \tag{6.10}$$

This linear model will be used for the both animation and control purposes. In the next section, the relationship between β and the dc-motor angle will be found, and the animation of the model will be carried out accordingly.

Creating a Virtual Apparatus

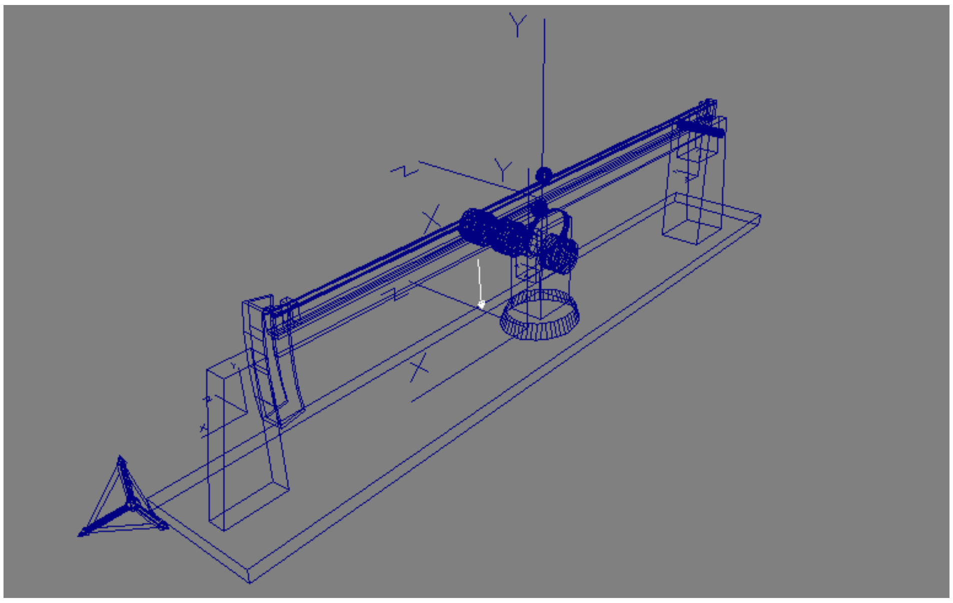


Figure 6.12: The Wire Frame Model of Ball and Beam Instrument

The components of the virtual apparatus have been prepared in the **"TrueSpace"** environment, as shown in Figure 6.12, and saved in a **"VRML"** file to be used by the animation component. The simulation component based on the simulation model discussed here uses AniComp's low level transformation functions to translate or rotate the objects in 3D by supplying position, velocity or acceleration vectors.

In the first control loop, the angular position of ellipsoidal prism can be controlled very accurately. In the simulator component, the value of this angular position is assumed the input of virtual model, and the rotation and translation of other objects are calculated according to this input value. An instant of the ellipsoidal prism, gear and beam are shown in Figure 6.13.

In order to find the relationship between θ and the position change in the

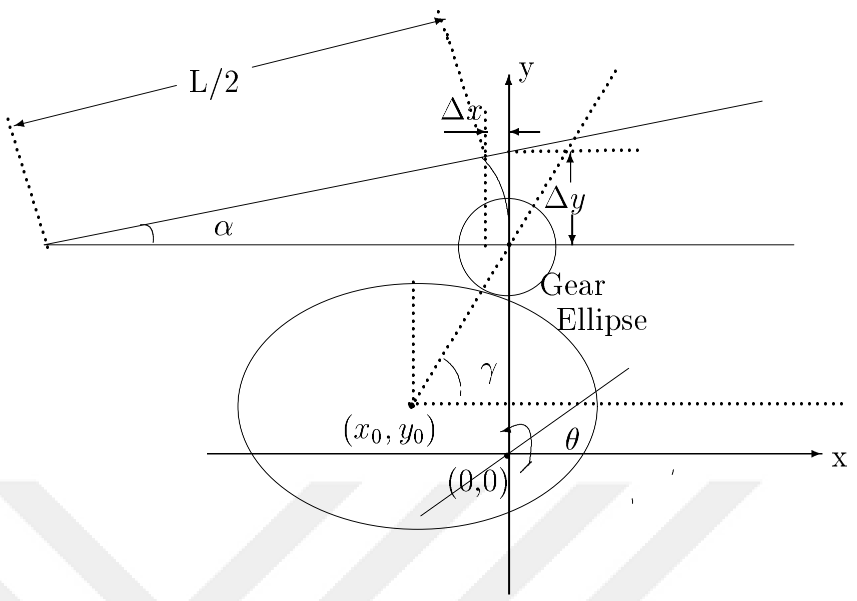


Figure 6.13: Calculating the change of position of the gear

gear, a small rotation around the origin can be given as follows:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_0 + (a + r_0)\cos\gamma \\ y_0 + (b + r_0)\sin\gamma \end{bmatrix} \tag{6.11}$$

where x_1 and y_1 is the new position of the gear, a and b are the inverse of semi-axis of ellipse, r_0 is the radius of the gear. Now, set $x_1 = x_{02}$, then the first equation becomes as follows:

$$x_0\cos\theta + (a + r_0)\cos\gamma\cos\theta - \sin\theta y_0 - (b + r_0)\sin\theta\cos\gamma = 0 \tag{6.12}$$

If this equation is solved with respect to γ using a symbolic tool, such as Mathematica [Wol96], the following equation is obtained:

$$\gamma = \pm ArcSec(\frac{t_1 \mp \sqrt{t_2^2(1 - t_1^2 + t_2^2)}}{t_1^2 - t_2^2}) \tag{6.13}$$

where

$$\begin{aligned} t_1 &= \frac{y_0 \sin\theta - x_0 \cos\theta + x_{02}}{(a + r_0)\cos\theta} \\ t_2 &= \frac{b + r_0}{a + r_0} \tan\theta \end{aligned} \tag{6.14}$$

There are four values of θ which give solution to the Equation 6.11. Only one of these solutions is valid for four quadrants of the ellipsoidal prism. Using an appropriate value of θ, y_1 is found as follows:

$$y_1 = \sin\theta(x_0 + (a + r_0)\cos\gamma) + \cos\theta(y_0 + (b + r_0)\sin\gamma) \tag{6.15}$$

Therefore, the general algorithm to animate the ellipsoidal prism, gear and the beam can be formulated as follows:

- Initialize x_1 and y_1, the x and y coordinates of the center of the gear, respectively.
- Enter the value of $\Delta\theta$ (rotation of dc-motor)
- Calculate Δy using Equation 6.15.
- Calculate $\Delta\alpha$ using $\Delta\alpha = Arcsin(2\Delta y/L)$
- calculate $\Delta x = (L/2)(1 - \cos\Delta\alpha$
- Go to second item and repeat the process.

Controller Design

In order to keep the ball at a desired position a controller is essential since the system is open-loop unstable due to double integrator characteristic. A standard pole assignment by state-feedback has been chosen for this purpose. Although nonlinear equations of the ball and beam apparatus have been used in the simulation and animation of the apparatus, a linear model has been chosen to design these two controllers. The system has two state-variables; namely, the position of the ball x, and the speed of the ball v. The control input is the angle of the ellipsoidal prism. The dynamic relationships between this angle and the position of the ball can be approximated by a gain and the linear model can be given as follows.

$$\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ b \end{bmatrix} u(t) \tag{6.16}$$

A state-feedback control law [Sø98, Sø99] can be given as follows

$$u(t) = -kx(t) + r(t) \tag{6.17}$$

where $r(t)$ is the reference ball position and $k = [k_1, k_2]$ is the gain vector to be chosen in order to put the closed-loop poles in appropriate places.

In a real system, the velocity of the ball and beam apparatus is not generally available, but can be obtained by a simple observation. In this virtual experiment set, this value can be taken from the simulator component as if it is being measured. Another way of observing it to integrate the actuator input (ellipse turn angle) such that the resulting signal will be proportional to the speed of the ball. The closed-loop transfer function of the system can be calculated as follows:

$$x(s) = \frac{b}{s^2 + sk_2 + k_1 b} r(s) \tag{6.18}$$

The state-feedback controller can be illustrated in the Figure 6.14.

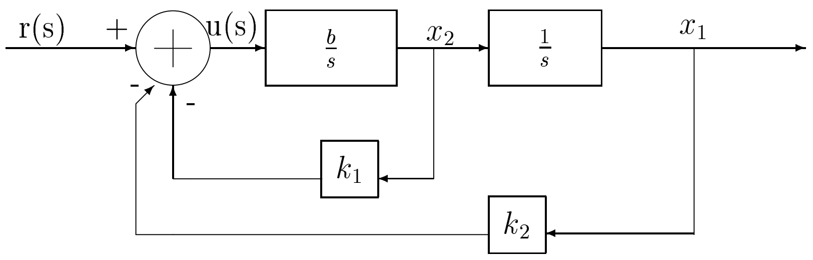


Figure 6.14: The state-feedback controller

By choosing the position of the closed-loop poles, the dynamic behaviour of the ball can be specified according to user's desire.

6.4.2 Modelling the Inverted Pendulum Instrument

Nearly all Feedback Control System Engineers have studied the "Inverted Pendulum" problem. It is often used as an example in many college and university engineering courses on dynamic systems, feedback control systems, and linear systems. The problem involves a dynamic system that consists of a cart with a stick hinged to its top. The system is obviously unstable - the stick will not remain upright without an input force. The objective of the problem is to design a control system that successfully maintains the stick in the upright position by applying a horizontal force to the cart.

The model for the inverted pendulum on a moving cart results in a fourth order nonlinear dynamical system. This model does not capture motor dynamics or pendulum flexing. The only system input is the force applied to the cart. The cart with an inverted pendulum, shown in Figure 6.15, is "bumped" with an impulse force, F. First the dynamic equations of motion for the system are developed, and linearized about the pendulum's angle, $\theta = 0$. Summing the

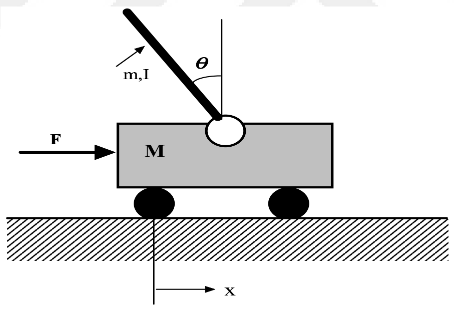


Figure 6.15: A diagram representation of inverted pendulum

forces in the Free Body Diagram of the cart in the horizontal direction, we get the following equation of motion:

$$M\ddot{\mathbf{x}} + \mathbf{b}\dot{\mathbf{x}} + \mathbf{N} = \mathbf{F} \tag{6.19}$$

Next, summing the forces in the Free Body Diagram of the pendulum in the horizontal direction, we can get an equation for N:

$$N = m\ddot{x} + ml\ddot{\theta}cos\theta - ml\dot{\theta}^2\sin(\theta) \tag{6.20}$$

If we substitute this equation into the first equation, we get the first equation of

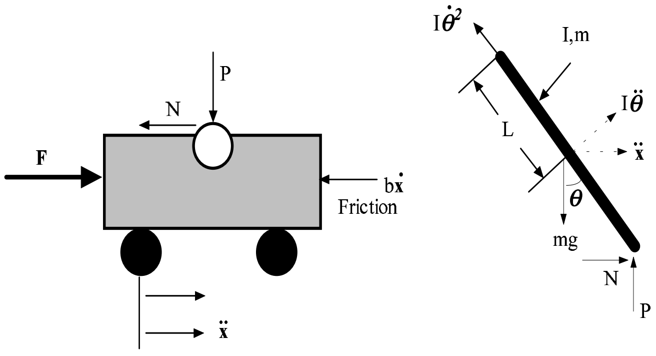


Figure 6.16: Forces effective on the cart and the pendulum

motion for this system:

$$(M+m)\ddot{x}+b\dot{x}+ml\ddot{\theta}\cos(\theta)-ml\dot{\theta}^2\sin(\theta)=F \tag{6.21}$$

where x denotes the position of the cart, θ denotes the angle which the pendulum makes with the vertical and is measured in radians, F denotes the force applied to the cart measured in Newtons, m denotes the mass of the pendulum measured in kilograms and l denotes the length of the pendulum measured in meters. To get the second equation of motion, we sum the forces perpendicular to the pendulum. Solving the system along this axis we get the following equation:

$$P\sin(\theta)+N\cos(\theta)-mg\sin(\theta)=ml\ddot{\theta}+m\ddot{x}\cos(\theta) \tag{6.22}$$

Summing the moments around the centroid of the pendulum to get rid of the P and N terms in the equation above, give us the following equation:

$$Pl\sin(\theta)-Nl\cos(\theta)=ml\ddot{\theta}+I\ddot{\theta} \tag{6.23}$$

Combining these last two equations, we get the second dynamic equation:

$$(I+ml^2)\ddot{\phi}+mgl\sin(\theta)=-ml\ddot{x}\cos(\theta) \tag{6.24}$$

We need to linearize about $\theta=\pi$. We assume that $\theta=\pi+\phi$ (ϕ represents a small angle from the vertical upward direction). Therefore, $\cos(\theta)=-1$, $\sin(\theta)=-\phi$, and $(d\theta/dt)^2=0$. After linearization the two equations of motion become (where u represents the input):

$$(I + ml^2)\ddot{\phi} - mgl\phi = ml\ddot{x} \tag{6.25}$$

$$(M + m)\ddot{x} + b\dot{x} - ml\ddot{\phi} = u \tag{6.26}$$

Although the system is unstable, it can be proven that the system is completely controllable [MT00]. There exists a feedback control law in the form [Phi00]:

$$F = K1 * X1 + K2 * X2 + K3 * X3 + K4 * X4 \tag{6.27}$$

Building the Virtual Model

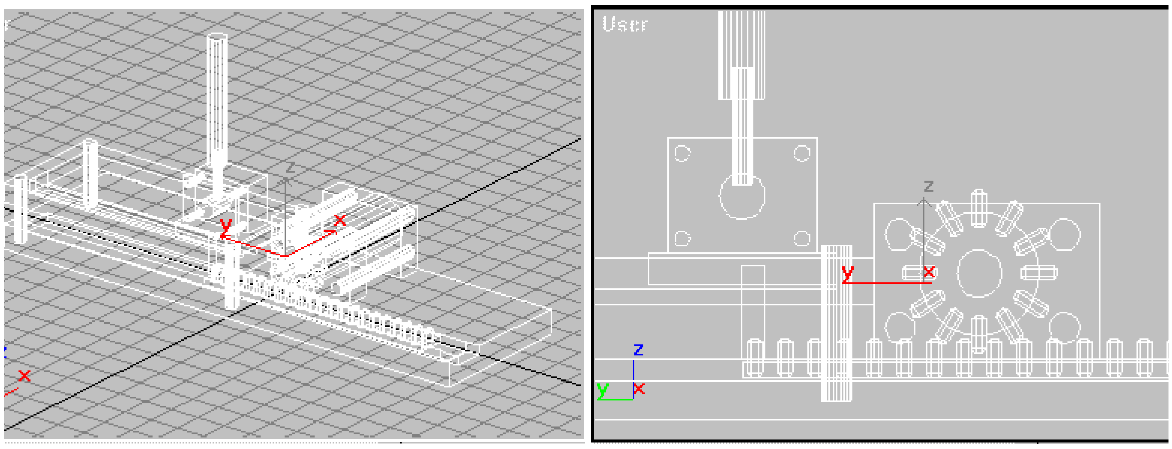


Figure 6.17: The Wire Frame Model of Inverted Pendulum Instrument

The geometric model of the instrument was developed using 3D Max package [Kin00]. Figure 6.17 shows a wire frame representation of the designed model. The window on the left shows the whole instrument. The window on the right illustrates the gear system used to transfer the electric motor's movement to the cart. The circular gear drives a linear gear which is attached to the cart at the other end. The simulation model is a Simulink model. The diagram in Figure 6.18 shows the simulation block diagram developed using simulink. The Inverted Pendulum model used in this experiment is one of the demo models that comes with Simulink package. The model was adapted for use with AniComp by modifying the animation function that comes with the model to contain some global

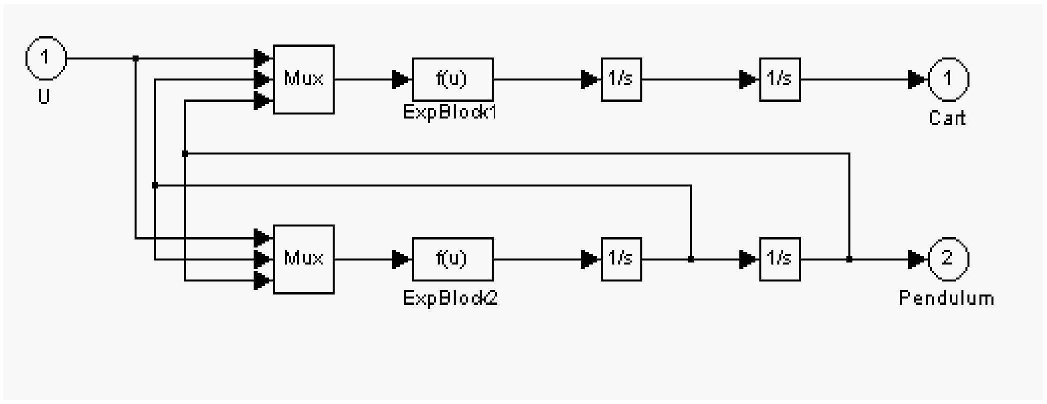


Figure 6.18: The Simulation Diagram of Inverted Pendulum in Simulink

variables that holds the incoming values for cart position and pendulum angle. These variables are, then, continuously sampled by the Visual Basic component that was implemented for that purpose, and their values are used to animate the system's behaviour. An additional variable called IsSimulationStarted is also defined which indicates whether the simulation exercise has been started by the user. This variable is polled by the VisualBasic component to synchronize with Simulink. In the diagram (Figure 6.18), the expression block (ExpBlock1) that produces input for the integrators which at the end leads to cart position contains the following equation:

$$\frac{(\frac{u}{m} - g\sin(\theta)\cos(\theta) + l\dot{\theta}^2\sin(\theta))}{(\frac{M}{m} + \sin(\theta)^2)} \tag{6.28}$$

The expression block defined for the pendulum angle output (ExpBlock2), on the other hand, contains the following equation:

$$\frac{-u\frac{\cos(\theta)}{m} + (M+m)g\frac{\sin(\theta)}{m} - l\dot{\theta}^2\sin(\theta)\cos(\theta)}{l(\frac{M}{m} + \sin(\theta)^2)} \tag{6.29}$$

In both equations θ denotes the pendulum angle and u is the input to the block.

The overall controller design is represented in Figure 6.19. Please refer to the tutorial given in [Mes00], for a step by step explanation of modeling an inverted pendulum system and its controller using Simulink package.

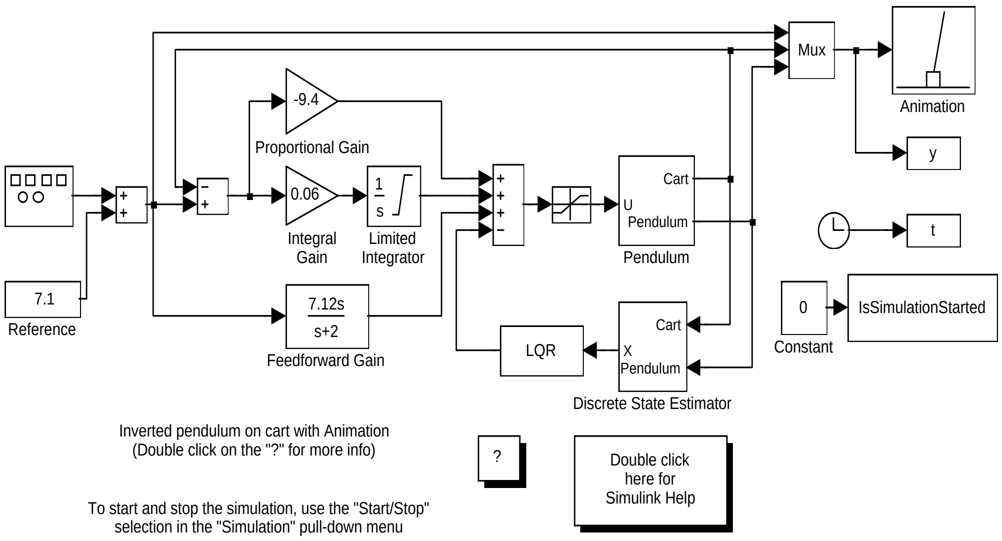

Mux
Animation
y
t
-9.4
Proportional Gain
0.06
Integral Gain
1/s
Limited Integrator
Cart
U
Pendulum
Pendulum
7.1
Reference
7.12s / s+2
Feedforward Gain
LQR
Cart
X
Pendulum
Discrete State Estimator
0
Constant
IsSimulationStarted
Inverted pendulum on cart with Animation (Double click on the "?" for more info)
?
Double click here for Simulink Help
To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

Figure 6.19: The Controller and Simulator Diagram in Simulink

6.5 Summary

This chapter exemplified the deployment of AniComp as part of a component framework which regulated the component interactions by imposing a weak agreement. The framework and AniComp were used to implement an application designed for building a web-enabled virtual laboratory environment. Issues such as dynamic component containment, run-time contract validation, incorporation of coarse grain, dissimilar simulation packages into a common framework were discussed. The geometric and mathematical models of two virtual instruments were also presented to support the discussions with concrete examples.

Chapter 7

Conclusions

The study presented in this thesis has focused on three main research topics:

1. The study of software components, in general, and interaction patterns between a simulation component and a visualization component in component based simulation environments.

2. Specification of an interface for an animation and visualization server component that aims to meet the requirements of its simulator clients. Additionally, the design of the internal architecture of such an animation component.

3. The design and implementation of a novel animation component capable of serving as a re-usable, off-the-shelf binary component for simulation applications. In addition, the implementation of a basic component framework providing an environment for integrating simulation components with the animation component implemented during this work.

A summary of the conclusions reached for these areas is given below. Their outcome have been demonstrated by a number of contributions which are listed in section 7.4.

7.1 Simulator-Animator Interactions

Having surveyed the basic simulation modeling approaches as well as the well-known animation techniques, the nature of possible interaction patterns between a simulation module and an animation module has been identified. The three basic simulation modeling approaches, namely, continuous-time, discrete event, and combined simulation, have been studied. It has been found out that the simulation programs based on continuous-time models that generate the behaviour of objects, tend to produce successive position and orientation values, which results in high frequency transformation calls to the graphics software. This is because the continuous-time models that can directly be related to the animation of virtual objects are models of the physical characteristics of those objects, usually consisting of a number of differential equations. Solving these equations results in values that can directly be mapped to physical motion specification.

The discrete-event simulation approach, on the other hand, is more concerned with exercising scenarios where many entities are involved and evolve in time, going through many state changes in relation with many resources they use (or consume). It can be said that these state changes are not necessarily always translatable into *motion*, nor can the entities (or resources) always be mapped to real world (virtual) objects. However, it has been observed that in most cases the scenario involves entities and resources that have equivalent observable objects, and their *motion* is of particular interest to the simulationist for reasons such as model validation and verification or presentation. In discrete-event simulation, the primary focus concerning the visualization of the scenario is the smooth operation (animation) of entities and resources in a way that is not only visually plausible but also reflects the time scale of the exercise, rather than *how* a particular entity's motion is generated. Therefore, it has been concluded that the interface calls emanating from a discrete-event simulation module (to the animation module), can be translated into calls to a pre-recorded motion (behaviour) library, irrespective of how those motion sequences were created. Thus, a scheme has been suggested that facilitates creation, serialization, and activation of re-usable motion sequences for that purpose.

The combined simulation modeling approach integrates continuous and discrete models. The result of this approach from the animation point of view, in its ideal form, is the *realistic behaviour of virtual objects governed by a meaningful and controllable scenario*. Although one of the main issues in combined simulation is to determine the interactions and interfaces between the simulation components with different properties, this is not considered as a central issue in this thesis. The focus has been on the impact of this modeling approach on the simulation-animation interaction. It has been proposed that, the world view of a simulation model involving combined modeling principles translates into the animation domain in the form of a world view where the behaviour of virtual objects are mainly generated by continuous models whereas the transition between different behaviours and the interrelation of the objects are governed by discrete-event models. In a typical simulation exercise, the discrete event model would act as a scenario manager which generates the events based on statistical models (random number generators) to direct the flow of the entities within the system. The continuous models then, would provide a basis for realistic simulation of the behaviours of the entities of various types.

The findings of the study of simulation modeling approaches with particular emphasis on the possible interaction patterns they cause, have been detailed in Chapter 4. These findings formed one of the bases for determining the requirements for the interface of an animation and visualization component.

7.2 The Interface Specification and Internal Design of an Animation Server Component

There are some important aspects of the work carried out for the interface design and the internal design of the animation component implemented during this research work:

- The difference between analysis and design of a complete software system and design of an independently deployable (off-the-shelf) binary component has been highlighted. It has been pointed out that during a complete system design, the emphasis is typically on the end user's requirements and that all the components of the system are identified and are explicitly known at design time. During a component design, however, the emphasis is on the requirements of *other software components* that are assumed to collaborate with the component being designed. Devising a model for expected component interactions is therefore an important part of the design process. For this reason, an interaction model for animation and simulation components has been devised based on the domain analysis performed, as indicated above (in Section 7.1).

- A distinction has been made between the interface design and the implementation design of a binary component. An interface specifies a parcel of behavior and responsibilities that any component will have to uphold if it is to participate as part of an assembly which represents some collaboration in the application domain. It has been emphasized that the interface design does not necessarily have to be performed by the people who set out to design and implement the component which implements that interface.

- In the design of the animation component's (AniComp) interfaces, a recent tool, *re-use contracts*, has been used for publishing the interface specification. Re-use contracts are based on the view that a component's interfaces constitute a *contract* with its acquaintances. The technique presents these contracts in the form of interactions aiming to sketch a more complete view of the component's expectations from other components. As part of the design work presented in this thesis, the notation of re-use contracts has been extended further to explicitly indicate the order of interactions in a concise way.

- In the internal design of AniComp a number of *design patterns* have been used. It has been an interesting experience to observe that design patterns do actually deliver their promises. AniComp's implementation consists of approximately nine thousand lines of code (excluding the externally included packages). The structural design patterns, Facade, Mediator, and Delegator, have helped to end up with a more manageable and maintainable code of this size.

- The VRML (Virtual Reality Modeling Language) internal execution model has been found to be an effective way of creating real-time animations especially when driven by external procedures. Reflecting the logical hierarchy of geometric objects (in a model file) into the internal representation of those objects (as a scene tree) and exposing the transformation nodes of these objects to manipulation via well-defined mechanisms provides a robust environment. Therefore, this model has been adopted as the foundation for mapping the functional model of a virtual object (i.e. its simulation model) to its geometric representation.

7.3 Using AniComp in a Component Framework: MISAF

An important challenge in developing an independently deployable component is to ensure that the component is re-used correctly. That is, the component assembler should be made aware of the component's expectations from other interacting components. One move towards this end is to document the component's interfaces in an expressive way. Re-use contracts have been used as a medium for achieving such expressiveness. As has been discussed in Chapter 6, component frameworks promise to be even more effective in this respect, since they impose an interaction scheme on collaborating components. The MInimal Simulation Animation Framework was implemented as a test framework to, if possible, verify this claim. It has been observed that a component framework can indeed regulate component interactions thereby securing the application of a valid interaction protocol to a certain extent. However, there is a trade-off to be made, when deciding whether to impose more rules for interacting components and hence secure a higher degree of cohesiveness and robustness, or to let the framework offer a looser contractual agreement and hence achieve a higher degree of flexibility. The route taken in this work is the latter. This is partly because imposing a tight framework requires the possession of a significant amount of experience with the candidate simulation modeling standards and the development of software components based on the operational principles of those standards. Without such experience the imposed framework may prove to be limiting and premature. Another reason for taking this route, is that viewing the simulation components as coarse grain, self-contained modules leads to less assumptions to be made about the internal architecture of the components which, in turn, facilitates easier integration of the existing simulation software into the framework. This was exemplified by integrating MATLAB/Simulink simulation package into an application through MISAF.

7.4 Contributions and Claims to Originality

As a result of the research work presented in this thesis, a number of contributions have been made. To the best knowledge of the author these contributions are original. These are presented in the following list:

- Component-based development has been applied to the simulation domain, in particular to the animated visualization of simulated entities.

- A model has been developed for simulation-animation interaction in a component-based simulation environment. For that purpose, simulation modeling approaches have been studied from a particular point of view, i.e. considering their impact on the visualization of the entities they are simulating.

- Re-use contracts have been applied to the specification of the interfaces for a binary component.

- The notation of re-use contracts has been extended for indicating the explicit order of interactions between two collaborating components.

- A novel animation and visualization component (AniComp) has been designed and implemented conforming to one of the established component wiring standards (COM), to demonstrate the validity of the work.

- Mechanisms for low level 3D object transformation, rigid body simulation, collision detection and high level re-usable motion sequences have all been encapsulated in a single, independently deployable visualization component.

- An easy-to-use interface has been designed for AniComp, which employs name based geometric object manipulation. This hides the complexity of the underlying graphics system and internal organization of the geometric objects, and enables geometric object manipulation through a more abstract layer.

- A run-time interface contract validation mechanism has been designed and implemented which searches for all the registered (COM) components in the system, queries the meta information in a selected component's type library and validates a hard coded interface contract. This is coupled by a component wrapper class and the flexi-switch structure to manage the dynamic incorporation of binary components at run-time.

- The mechanisms that were implemented for contract validation and run-time component management have been combined to implement a component framework (MISAF) for supporting AniComp's use in component-based simulation environments.

- The use of AniComp and MISAF are exemplified in the implementation of an application which can be used to dynamically build a virtual control laboratory involving two virtual instruments, namely the Ball and Beam and the Inverted Pendulum.

7.5 Future Work

The work presented in this thesis can be extended and improved in a number of ways. There are several points that are regarded to be candidates of further improvements.

- **Extending MISAF.** The component framework implemented here can be generalized by allowing any interface specification to be used as a contract. The contract then should be read in, rather than being hard-coded into the framework. This requires the contract being defined in a widely accepted format accompanied by a well-defined way of defining a minimal set of interaction protocols. Importing the interface specification to be used as the contract is relatively simple. IDL files are natural candidates for parsing and identifying the functions in the interface. However specifying the interaction protocol is not an easy task. It would be an interesting research effort to devise a way of directing the framework to behave according to a specified interaction protocol without prior knowledge of that particular protocol.

- **Advanced Virtual Reality Interaction Techniques.** Currently the AniComp's user interface does not support advanced interaction techniques. This was, as pointed out in the introductory chapter, partly due to the unavailability of advanced interaction devices and their supporting software, to the author. An immediate enhancement to the animation component would be to incorporate the use of such interaction techniques. Immersion of the user into the virtual environment would then be possible.

- **Support for Distributed Simulation Environments.** Although the underlying component technology (COM/DCOM) supports distributed computation and some mechanisms used in AniComp's implementation such as rigid body simulation (a form of dead reckoning), and the use of pre-defined motion sequences potentially reduce the network traffic, AniComp has not been tested in distributed simulation scenarios. A possible scenario would be to visualize the simulated entities that are distributed to different physical locations, in the same shared virtual environment.

Bibliography

[Ada98] Adamski, D. and Miller, M. CORBA in simulation tasks. In K. Juslin, editor, *Proc. Eurosim Simulation Congress (EUROSIM'98)*, pages 14–19, Espoo, Finland, 1998. Federation of European Simulation Societies.

[Ahn94] Ahn, Myung Soo and Kim, Tag Gon . DEVSim++: C++ Based Modeling/Simulation of Hierarchical Modular DEVS Models. *http://www-ais.ece.arizona.edu/*, 1994.

[Ale77] Alexander, C. A. *A Pattern Language.* Oxford University Press, New York, 1977.

[All99] Allen, P. . Doing bussiness with component based development. *Aplication Development Advisor*, 3(1):36–44, Sept/Oct 1999.

[Alp98] Alpdemir, M.N. and Zobel, R.N. A Component-Based Animation Framework for DEVS-Based Simulation Environments. In *Proceedings of the 12th European Simulation Multiconference (ESM'98)*, pages 79–83, Manchester, UK, June 1998.

[Alp99] Alpdemir, M.N. and Zobel, R.N. Component-Based Simulation Environments. In *Proceedings of the 1st MiddleEast Simulation Multiconference (MESM'99)*, pages 30–34, Amman, Jordan, March 1999.

[Arm97] Armstrong, T. *Designing and Using ActiveX Controls.* M&T Books, New York, 1997.

[Bal98] Balet, O. and Duchon, J. PROVIS: An interactive system for virtual cooperative prototyping. In *Proceedings of the 5th International Workshop on Simulation for European Space Programmes (SESP98)*, pages 161–164, ESTEC, Noordwijk, The Netherlands, 3-5 November 1998.

[Ban98] Banks, Jerry. Principles of Simulation. In Jerry Banks, editor, *Handbook of Simulation*, pages 3–30. John Wiley & Sons Inc., 1998.

[Bar88] Barzel, Ronen and Barr, Alan H. A Modeling System Based on Dynamic Constraints. *Computer Graphics*, 22(15):179–7188, 1988.

[Bar92] Barzel, Ronen. *Physically-Based Modeling for Computer Graphics: A Structured Approach.* Academic Press Inc., 1992.

[Bar95] Baraff, David. Interactive Simulation of Rigid Bodies. *IEEE Computer Graphics and Applications*, (15):63–75, May 1995.

[Bar96] Barzel, Ronen and Hughes, John F. and Wood, Daniel N. Plausible Motion Simulation for Computer Graphics Animation. In R. Boulic and G. Hgron, editors, *Computer Animation and Simulation '96 (Proceedings of the Eurographics Workshop)*, pages 183–197, New York, 1996. Springer Wien.

[Bar97] Baraff, David and Witkin, Andrew. Partitioned Dynamics. Technical Report CMU-RI-TR-97-33, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.

[Bc97] Bchi, Martin and Weck, Wolfgang. A plea for grey-box components. Technical Report, ISBN 952-12-0047-2 122, TUCS, 1997.

[Bc99] Bchi, Martin and Weck, Wolfgang. The greybox approach: When blackbox specifications hide too much. Technical Report, ISBN 952-12-0508-3 297, TUCS, 1999.

[Bee] Beeson, C. *An Object-Oriented Approach To VRML Development.* Silicon Graphics homepage, http://reality.sgi.com/curtisb_engr/.

[Ben98] Bennet, B. N. . *Essential COM.* Addison Wesley Longman Inc., 2.0 edition, 1998.

[Bla97] Blaszczak, M. *Professional MFC with Visual C++.* Wrox Press, 1997.

[Boo94] Booch, Grady. *Obbject-Oriented Design and Analysis With Applications.* the Benjamin/Cummings Publishing Company, Inc., 2.0 edition, 1994.

[Bre87] Breen, D.E. and Getto, P.H. and Apocada, A.A. and Schmidt, D.G. and Saracan, B.D. The Clockworks: An Object-Oriented Computer Animation System. In *Eurographiccs'87*, pages 275–282, North Holland, 1987.

[Bro98] Broy, M. et al. . What characterizes a (software) component? . *Software - Concepts & Tools*, 19(1):49–56, 1998.

[Bry95] Bryson, Steve . Approaches to the succesful design and implementation of VR Applications. In J.A. Vince R.A. Earnshaw and H. Jones, editors, *Virtual Reality Applications*, pages 3–15. Academic Press Inc., 1995.

[Bus96] Buschmann, F. et. al. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[Cal93] Calvin, J. and others. The SIMNET Virtual World Architecture. In *Proceedings of the IEEE VRAIS 93 Conference*, pages 450–455, 1993.

[Car93] Carlsson, C. and Hagsand, O. DIVE - a Multi-User Virtual Reality System. In *Proceedings of the IEEE VRAIS 93 Conference*, pages 394–400, 1993.

[CAS95] CASA, Division Espace. *VISTA Executive Summary.* 1995.

[Che97] Chenney, Stephen and Forsyth, David . View-Dependent Culling of Dynamic Systems in Virtual Environments. In *1997 Symposium on Interactive 3D Graphics*, Providence, RI, April 27-30 1997.

[Coh95] Cohen, J. and Lin, M. and Manocha, D. and Ponamgi, K. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scaled Environments. In Jean-Paul Laumond and M. Overmars, editors, *Proceedings of ACM Int. 3D Graphics Conference*, pages 189–196, 1995.

[Cos00] Cosmo3D. *SGI Optimizer Home Page.* http://www.sgi.com/software/optimizer/, 2000.

[Dol97] Dollner, Jurgen and Hinrichs, Klaus . Object-Oriented 3D Modelling, Animation and Interaction. *The Journal of Visualization and Computer Animation*, 8:33–64, 1997.

[Ear94] Earle, N. J. et. al. Proof Animation, Reaching Hights in Animation . In *Proc. of the 1994 Winter Simulation Conference*, pages 509–516, Florida, USA, 1994.

[Ell95] Ellis, Stephen R. Origins and Elements of Virtual Environments. In Woodrow Barfield and Thomas A. Furness III, editors, *Virtual Environments and Advanced Interface Design*, pages 14–57. Oxford University Press, 1995.

[Erk95] Erkan, Bilge and Ozguc, Bulent. Object-Oriented Motion Abstraction. *The Journal of Visualization and Computer Animation*, 6:49–65, 1995.

[Fay97] Fayad, E.M. and Schmidt, C.D. . Object-Oriented Application Frameworks . *Communications of the ACM*, 40(10):33–38, October 1997.

[Fis98] Fishwick, P. A. Issues with Web-Publishable Digital Objects. In *SPIE Aerosense Conference*, Orlando Florida, USA, April 1998.

[Fla97] Flanagan, D. *Java in a Nutshell.* O'Reilly & Associates, Sebastopol, USA, second edition, 1997.

[Fur95] Furness III, Thomas A. and Barfield, Woodrow. Introduction to Virtual Environments and Advanced Interface Design. In Woodrow Barfield and Thomas A. Furness III, editors, *Virtual Environments and Advanced Interface Design*, pages 3–13. Oxford University Press, 1995.

[Gam95] Gamma, E. and Helm, R. and Johnson, R and Vlissides, J . *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gig93] Gigante, Michael A. Virtual Reality: Enabling Technologies. In M.A. Gigante R.A. Earnshaw and H.Jones, editors, *Virtual Reality Systems*, pages 15–25. Academic Press Ltd., 1993.

[Got96] Gottschalk, S. and Lin, M. and Manocha,D. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. In Jean-Paul Laumond and M. Overmars, editors, *Proc. of ACM Siggraph'96*, 1996.

[Gra98] Grand, M. *Patterns in Java, Volume 1.* John Wiley & Sons Inc., 1998.

[Gre96] Greenhalgh, Chris. Approaches to distributing virtual reality. Technical report, nottcs-tr-96-5, Department of Computer Science, The Univerity of Nottingham, August 29 1996.

[Gri98a] Grimes, R. *Professional ATL COM Programming.* Wrox Press, 1998.

[Gri98b] Grimes, R. and Stockton, A. and Reilly, G. and Templeman, J. *Beginning ATL COM Programming.* Wrox Press, 1998.

[Hab98] Habibi, J. and Alpdemir, M.N. and Zobel, R.N. Implementation of Functional Simulation and Animated Visualisation on Heterogeneous Parallel Architectures. In *Proceedings of the EUROSIM'98 Simulation Congress*, pages 7–13, Finland, April 1998.

[Har95] Harada, Mikako and Witkin, Andrew and Baraff, David. Interactive Physically-Based Manipulation. In *Computer Graphics Proceedings, SIGGRAPGH 95*, pages 199–208, Los Angeles, August 6-11 1995.

[Hay92] Hay, J. and Pearce, J. and Crosbie, R. . *The ESL User Manual.* iSiM Simulation, Salford Univ. Business Service Ltd, 1992.

[Heg96] Hegron, G. and Arnaldi, B. and Lecerf, C. Dynamic Simulation and Animation. In N.M. Thalmann and D. Thalmann, editors, *Interactive Computer Animation*, pages 71–99. Prentice Hall, 1996.

[Hes95] Hessel, E. Model exchange – illusion or future reality. In F. Breitenecker and I. Husinsky, editors, *Proc. EUROSIM Conference EUROSIM '95*, pages 469–474. Society for Computer Simulation International., Elsevier Science B.V., 1995.

[Hil96] Hill, David R.C. *Object-Oriented Analysis and Simulation.* Addison-Wesley, first edition, 1996.

[Hon97] Hondt, Koen De and Lucas, Carine and Steyaert, Patrick . Reuse contracts as component interface descriptions. In Clemens Szyperski Wolfgang Weck, Jan Bosch, editor, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, number 5 in TUCS General Publications, pages 43–50. Turku Centre for Computer Science, Sep. 1997.

[Hor97] Horton, I. *Beginning MFC Programming.* Wrox Press, 1997.

[Hud97] Hudson, T.C. and Ming, C.L. and Cohen, J. and Gottshalk, S. and Manocha, D. V-Collide: Accelerated Collision Detection for VRML. In *Proceedings of VRML'97*, 1997.

[IEE93] IEEE,Institute of Electrical and Electronics Engineers. Standard for information technology - protocols for distributed interactive simulation applications (international standard ansi/ieee std 1278-1993). March 1993.

[Joh97] Johnson, Ralph E. Frameworks = Components + Paterns . *Communications of the ACM*, 40(10):39–42, October 1997.

[Jor98] Jordan, D.W. and Smith, P. *Mathematical Techniques.* Oxford University Press, 2 edition, 1998.

[Ker97] Kerckhoffs, E. and Lehmanm, A. and Pierrval, H. and Zobel, R., editor. *Modelling and Simulation of Complex Systems Concepts, Methods and Tools.* SCS, Ghent Budapest, first edition, 1997.

[Kim94] Kim, Tag Gon. DEVSim++ User's Manual. Korea Advanced Institute of Science and Technology, 1994. Computer Engineering Research Lab.

[Kin00] Kinetix. *3D Studio Max Modeling Animation Package.* Kinetix Homepage, http://www2.discreet.com/products/, 2000.

[Kri98] Krieger, David and Adler, Richard M. The Emergence of Distributed Component Platforms. *IEEE Computer*, pages 43–53, March 1998.

[Kru91] Kruse, R.L. and Leung, B.P. and Tondo, C.L. *Data Structures and Program Design in C.* Prentice Hall, New Jersey, 1991.

[Kuc98] Kucuk, B. and Alpdemir, M.N. and Zobel, R.N. Customizable adapters for blackbox components. In Clemens Szyperski Wolfgang Weck, Jan Bosch, editor, *Proceedings of the Workshop for Component Oriented Programming (WCOP'98)*, TUCS General Publications, pages 53–59, Brussels, Belgium, July 1998. Turku Centre for Computer Science.

[Lee97] Lee, C.H. and Kucuk, B. and Zobel, R.N. A simulation model library using OODBMS for product prototyping. In J. W. Wallace et al., editor, *Proc. Object-Oriented Simulation Conference (OOS'97)*, pages 155–160, Phoenix, Arizona, 1997. Society for Computer Simulation International.

[Lee98] Lee, G.S. Reusable Interactions for Animation. In *In Proceedings of the Fifth International Conference on Software Reuse*, pages 320–329, Victoria, British Columbia, Canada, 2-5 June 1998. IEEE Computer Society Press.

[Luc91] Luciani, A. and et. al. An Unified View of Multitude Behaviour, Flexibility, Plasticity and Fractures, Balls, Bubbles and Agglomerates. In *Modelling in Computer Graphics, IFIP Working Conference 91(TC 5/WG5.10)*, pages 55–74. Springer-Verlag, 1991.

[Mac99] Mactaggart, M. Cbd, components, and class libraries. *Aplication Development Advisor*, 3(1):14–17, Sept/Oct 1999.

[Mat93] Mattsson, S.E. and Andersson, M. and Astrom, K.J. Object-oriented modeling and simulation. In D. Linkens, editor, *CAD for Control Systems, chapter 2*, pages 31–69, New York, 1993. Marcel Dekker Inc.

[Mat98] Mattsson, S.E. and Elmqvist, H. An overview of the modeling language modelica. In K. Juslin, editor, *Proc. Eurosim Simulation Congress (EUROSIM'98)*, pages 182–186, Espoo, Finland, 1998. Federation of European Simulation Societies.

[Mes00] Messner, Bill and Tilbury, Dawn . *Control Tutorials for Matlab and Simulink.* http://kopernik.eos.uoguelph.ca/ zelek/matlab/ctms/index.htm, 2000.

[Mey99] Meyer, Bertrand. Rules for component builders. *Software Development*, 7(5):26–30, May 1999.

[Mil96] Miller, Duncan C. . The dod high level architecture and the next generation of dis. In *Proceedings of 14th DIS Workshop*, 1996.

[Mir94] Mirtich, Brian and Canny, John . Impulse-based dynamic simulation. In J.C. Latombe K. Goldberg, D. Halperin and R. Wilson, editors, *Proceedings of the Workshop on the Algorithmic Foundations of Robotics*, Boston M.A., Februaury 1994. A.K. Peters.

[Mir95] Mirtich, Brian and Canny, John . Impulse-based simulation of rigid bodies. In *Proceedings of 1995 Symposium on Interactive 3D Graphics*, April 1995.

[MT00] Inc. Meixler Technologies. *Inverted Pendulum Lab Manual.* http://www.meixler-tech.com/ip.htm, 2000.

[Mur95] Murray-Smith, D.J. *Continuous System Simulation.* Chapman & Hall, 1995.

[Mye96] Myers, Brad A. A brief history of human computer interaction technology. Technical Report CMU-CS-96-163, Carnegie Mellon University School of Computer Science, December 1996.

[Nie95] Nierstrasz, O. and Dami, L. . Component-oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages chapter 1, 3–28. Prentice Hall, 1995.

[Nub98] Nuber, C. and Kueke, R. and Groten, L. DBView - a modular virtual reality extension for simulators. In *Proceedings of the 5th International Workshop on Simulation for European Space Programmes (SESP98)*, pages 177–181, ESTEC, Noordwijk, The Netherlands, 3-5 November 1998.

[Nyh88] Nyhoff, L. and Leestma, S. *Advanced Programming in Pascal with Data Structures.* Macmillan Publishing Company, New York, 1988.

[Ott98] Otter, M. and Mosterman, P. The DSblock model interface, version 4.0 . The Modelica World Wide Web site (http://www.Dynasim.se/Modelica), 1998.

[Pal96] Palmer, Ian J. and Grimsdale, Richard L. Object-Oriented Animation in the REALISM System. In Peter Wisskirchen, editor, *Object-Oriented and Mixed Programming Paradigms*, pages 87–98. EUROGRAPHICS, Springer-Verlag, 1996.

[Phi00] Phillips, Charles L. and Harbor, Royce D. *Feedback Control Systems.* Prentice Hall, 4 edition, 2000.

[Pla91] Plasil, F. and Stal, M. Simulation Modeling of Flexible Manufacturing System Using Activity Cycle Diagrams. *Operational Research Society Ltd*, 45(9):1011–1023, 1991.

[Pot99] Potter, C.D. Sim-Factory. *Computer Graphics World*, pages 42–46, Februaury 1999.

[Pre96] Preston, Martin. A Motion Generator Framework. In *Proceedings of Computer Animation.* IEEE Computer Society Press, 1996.

[Pri95] Pritsker, A. and Alan, B. *Introduction to Simulation and SLAM II.* Systems Publishing Corporation, West Lafayette, Indiana, fourth edition, 1995.

[Pri98] Pritsker, A. Alan B. . Principles of Simulation Modeling. In Jerry Banks, editor, *Handbook of Simulation*, pages 31–51. John Wiley & Sons Inc., 1998.

[Ree83] Reeves, W. T. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graphics*, 2:91–108, April 1983.

[Res97] Ressler, S. and et.al. Using VRML to Access Manufacturing Data. In *Proceedings of the Second Symposium on Virtual Reality Modeling Language*, pages 109–116, Monterey, CA USA, February 24 - 26 1997.

[Rey82] Reynolds, Craig W. . Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics (SIGGRAPH '82 Proceedings)*, 16(3):289–296, 1982.

[Sam97] Sametinger, J. *Software Engineering with Reusable Components.* Springer-Verlag, 1997.

[Sha96] Shaw, Mary and Garlan, David. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, Upper Saddle River, New Jersey 07458, 1996.

[Sho97] Short, K. *Component Based Development and Object Modeling.* http://www.cool.sterling.com/cbd/whitepaper/coverpg.htm, February 1997.

[Sø98] Søylemez, M. T. *Pole Assignment for Uncertain Systems.* PhD thesis, Control Systems Centre, UMIST, 1998.

[Sø99] Søylemez, M. T. . *Pole Assignment for Uncertain Systems.* UMIST Control System Centre Series. Research Studies Press, London, 1999. ISBN: 0-863-80246-X.

[Som96] Sommerville, I. *Software Engineering.* Addison Wesley, fifth edition, 1996.

[Sti97] Stiles, R. and Tewari, S. and Mehta, M. Adapting VRML 2.0 for Immersive Use. In *Proceedings of the Second Symposium on Virtual Reality Modeling Language*, pages 75–81, Monterey, CA USA, February 24 - 26 1997.

[Stu98] Sturmann, David . The State of Animation . *ACM SIGGRAPH Newsletter*, 32(1), February 1998.

[Sut63] Sutherland, I.E. SketchPad: A Man-Machine Graphical Communication System. In *AFIPS Spring Joint Computer Conference*, pages 329–346, 1963.

[Szy97] Szyperski, C. and Pfister, C. Workshop on Component-Oriented Programming, Summary. In M. Muhlhauser, editor, *Special Issues in Object-Oriented Programming — ECOOP'96 Workshop Reader*, Heidelberg, 1997. Dpunkt Verlag.

[Szy98a] Szyperski, C. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley Longman Ltd., 1998.

[Szy98b] Szyperski, C. Emerging component software technologies – a strategic comparison. *Software – Concepts & Tools*, 19(1):2–10, 1998.

[Szy98c] Szyperski, C. and Vernik, R. Establishing System-Wide Properties of Component-Based Systems: A Case for Tiered Component Frameworks. In *Workshop on Compositional Software Architectures*, Monterey, California, 6-8 January 1998.

[Szy99] Szyperski, Clemens. Components and objects together. *Software Development*, 7(5):33–41, May 1999.

[Tan98] Tandayya, P. and Zobel, R. N. VISTA Real-time Animation Software in a Distributed Spacecraft Simulation System. In *Proceedings of the 5th International Workshop on Simulation for European Space Programmes(SESP98)*, pages 183–190, ESTEC, Noordwijk, The Netherlands, 3-5 November 1998.

[Teo93] Teo, Y. M. PortSim: Simulation and Animation of Container Port . In J. Schoen, editor, *Proc. of the 1993 Summer Computer Simulation Conference*, pages 143–147, Boston, Massachusetts, July 19-21 1993. SCS.

[Tha85] Thalmann-Magnenat, N. and Thalmann, D. *Computer Animation: Theory and Practice.* Springer, Tokyo, 1985.

[Tha96] Thalmann-Magnenat, Nadia and Thalmann, Daniel. Computer Animation. In *Handbook of Computer Science*, pages 1300–1318. CRC Press, 1996.

[Tru00] TrueSpace. *Calgari TrueSpace Homepage.* http://www.caligari.com, 2000.

[Tse98] Tseng, M.M. and Jiao, J. and Su, C. Virtual Prototyping for Customized Product Development. *Integrated Manufacturing Systems*, 9(6):334–343, 1998.

[UML99] *OMG Unified Modeling Language Specification.* Object Management Group, UML Documentation Resources, http://www.rational.com/uml/resources /documentation /index.jtmpl, June 1999.

[VHD] Vhdl-ams world wide web site. (http://vhdl.org/vi/analog).

[Vin97] Vinovski, S. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

[vrm98] VRML 2.0 (VRML97) Specification. *Online document*, http://www.web3d.org/Specifications/, 1998.

[War96] Warwick, K. *An Introduction to Control Systems.* World Scientific Publishing Co., 2 edition, 1996.

[Wat92] Watt, Alan and Watt, Mark . *Advanced Animation and Rendering Techniques: Theory and Practice.* ACM Press, New York, 1992.

[Wat97] Waters, R. et.al. Diamond Park and Spline. *Presence*, 6(4):461–481, 1997.

[Wec97a] Weck, Wolfgang . Inheritance using contracts and object composition. In Clemens Szyperski Wolfgang Weck, Jan Bosch, editor, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, number 5 in TUCS General Publications, pages 105–112. Turku Centre for Computer Science, September 1997.

[Wec97b] Weck, Wolfgang. Independently Extensible Component Frameworks. In M. Mhlhuser, editor, *Special Issues in Object-Oriented Programming*, pages 177–183, Heidelberg, 1997. dpunkt Verlag.

[Wel78] Wellstead, P. E. and Crimes, V. The ball and beam control experiment. *Int. Journal of Elect. Eng. Education*, 13:21–39, 1978.

[Wes93] West, A.J. and Howard, T.L.J. and Hubbold, R.J. and Murta, A.D. and Snowdon, D.N. and Butler, D.A. AVIARY-A Generic Virtual Reality Interface for Real Applications. In M.A. Gigante R.A. Earnshaw and H.Jones, editors, *Virtual Reality Systems*, pages 213–236. Academic Press Ltd., 1993.

[Wil94a] Wilkie, G. *Object-Oriented Software Engineering.* Addison-Wesley Publishing Co., 1994.

[Wil94b] Willliams, S. and Kindel, C. *The Component Object Model: A Technical Overview.* Microsoft Corporation homepage, http://msdn.microsoft.com/ library/default.asp, October, 1994.

[Wit90] Witkin, A. and Gleicher, M. and Welch, W. Interactive Dynamics. In *Computer Graphics*, volume 24, pages 11–21, March 1990.

[Wit97] Witkin, Andrew and Baraff, David. *An Introduction to Physically Based Modeling.* SIGGRAPH 97 course, 1997.

[Wol96] Wolfram, S. *The Mathematica Book.* Wolfram Media and Cambridge University Press, 3rd edition, 1996.

[Zei90] Zeigler, B.P. *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems.* Academic Press, 1990.

Appendix A

Notation Guide for UML

A.1 Introduction

This appendix does not considers all the UML modeling elements. Only the ones which have been used in the thesis are presented. These are class diagrams, object diagrams, collaboration diagrams and component diagrams. The features of these modeling elements discussed here are limited to the features used in this thesis only. Therefore, this document should not be seen as a comprehensive guide to UML notation. The following sections presents the constructs mentioned above.

A.2 Class Diagrams and Object Diagrams

A class diagram is a graph of classifier elements connected by their various static relationships. A class diagram may also contain interfaces, packages, relationships, and even instances, such as objects and links. A class diagram illustrates a graphic view of the static structural model. It is a collection of (static) declarative model elements, such as classes, interfaces, and their relationships, connected as a graph to each other and to their contents. Class diagrams may be organized into packages either with their underlying models or as separate packages that build upon the underlying model packages. The most basic element of a class diagram is a *class*. Classes are drawn as rectangles. The rectangles can be divided into three compartments. Figure A.1 shows the same class presented at three different level of detail. In the most detailed one the top compartment contains the name of the class. The middle compartment lists the class's variables. The bottom compartment lists the class's methods. The symbols that precedes each variable

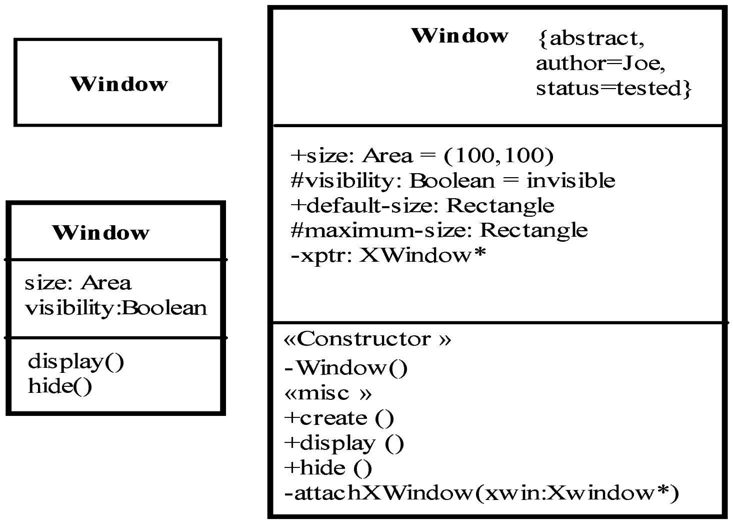


Figure A.1: Class Notation: Details Suppressed, Analysis-level Details, Implementation-level Details

and method are *visibility indicators*. There are three different types of visibility indicators:

```
+ : Public
# : Protected
- : Private
```

The visibility marker may be suppressed. The absence of a visibility marker indicates that the visibility is not shown (not that it is undefined or public).

In a UML drawing a word in guillemets is called a *stereotype*. A stereotype is used like an adjective to modify what comes after it, The `Constructor` stereotype indicates that the methods that follow it are constructors.

A.2.1 Showing the Relationships Between Classes

The relationships between classes are indicated by lines connecting the classes. There are several types of relationships that can be shown. These are *inheritance, association* and *aggregation*. A solid line with a closed arrowhead indicates the inheritance relationship. This is illustrated in Figure A.2. Th figure shows that the Polygon, Ellipse and Spline classes inherit from the Shape class. If a certain class is an abstract class, this is indicated by an italicized name, as in Figure A.2.

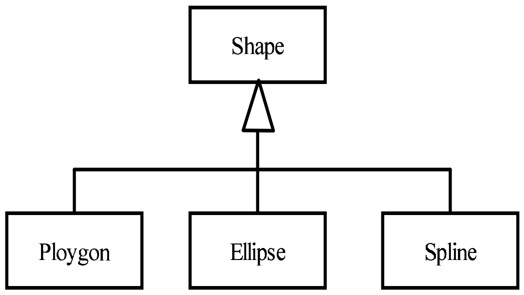


Figure A.2: Inheritance relationship

A binary association relationship is shown by a solid path connecting two classifier symbols (both ends may be connected to the same classifier, but the two ends are distinct). The path may consist of one or more connected segments. There are a number of things that can appear with an association that provide information about the nature of an association. First, there could be an *association*

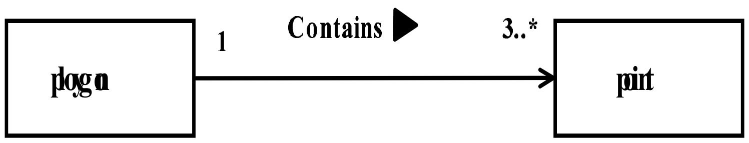


Figure A.3: An Association Relationship Between Classes

name. An example to that is the word `Contains` which is shown in Figure A.3. There may be a black triangle at one end of the name. This triangle suggests the direction of in which the association is read. Second there could be a *navigation arrow* at the end of the association line. navigation arrows indicate the direction in which an association is navigated. In Figure A.3, the navigation arrow implies that, `polygon` objects will have a reference to allow them to access `point` object, but not the other way around. Third there could be a multiplicity indicator. This indicates how many instances of each class participate in an occurrence of an association. A multiplicity indicator may appear at each end of association. It can be a simple number like 1 or 0, or it can be a range of numbers indicated as follows:

```
0..2
```

An asterisk used as the high value of a range means an unlimited number of occurrences. The multiplicity indicator 1..*, for example means at least 1 instance;

0..* means any number of instances. The multiplicity indicator given in Figure A.3, 3..*, means that a polygon must at least have three points but there is no an upper limit on the number of points in a polygon.

A.2.2 Object Diagrams

An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. The use of object diagrams is fairly limited, mainly to show examples of data structures. Class diagrams can contain objects, so a class diagram with objects and no classes is an object diagram. Objects are indicated by a preceding colon (:) before the name of the

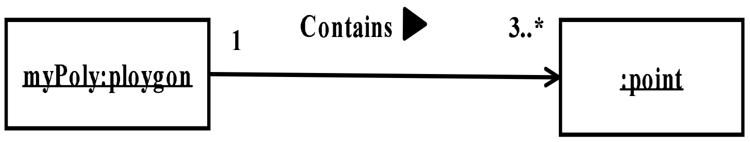


Figure A.4: A simple Object Diagram

class they instantiated from. The object name must be underlined. There may be another name before the colon, which would then identify that particular object. In Figure A.4 `myPoly` is a distinct object name which is an instance of the class `polygon`. The object `point` is any instance of the class `point`.

A.3 Collaboration Diagrams

Collaboration diagrams show objects, the links that connect them, an the interactions that occur over each link. They also show the sequence and concurrency requirements for each interaction. Any number of interactions can be associated with a link. Each interaction involves a method call. Next to each interaction or group of interactions is an arrow that points to the object whose method is called by the interaction. The entire set of interactions shown in a collaboration diagram is collectively called a *collaboration*. Each interaction starts with a sequence number. Sequence numbers indicates the order in which method calls occur. Multilevel sequence number consist of two or more numbers separated by a period. Multilevel sequence numbers correspond to multilevel method calls.

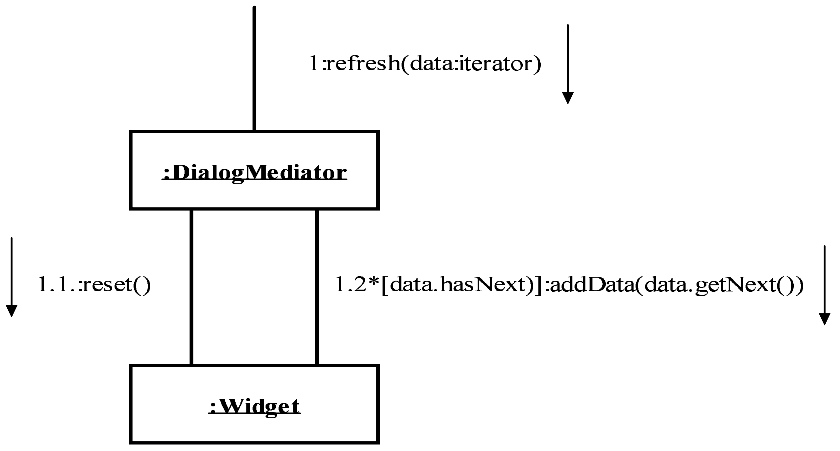


Figure A.5: An example of a Collaboration Diagram

Interactions numbered with a multilevel sequence number occur during another interaction's method call. So the method calls of interactions numbered 1.1 and 1.2 are made during the method call of interaction 1. This is indicated in Figure A.5. An asterisk after a sequence number indicates a repeated interaction. UML allows a condition to be associated with a repetitive interaction by putting it after asterisk inside square brackets. Figure A.5 shows an example of a conditional repetitive interaction where the `Iterator` object is passed to a `DialogMediator` object's `refresh` method. Its `refresh` method, in turn, calls its `addData` method, while the `Iterator` object's `hasNext` method returns true.

When dealing with multiple threads, something that often requires specification about methods is what happens when two threads try to call the same method at the same time. UML specifies this by placing one of the following constructs after a method:

```
{concurrency = sequential}
{concurrency = quarded}

{concurrency = concurrent}
```

If the concurrency is `sequential`, then only one thread at a time calls a method. No guarantee is made about the correctness of the method's behaviour if the method is called with multiple threads a time. If the concurrency if `guarded` then if multiple threads call a method at the same time, they all execute it concurrently and correctly. If the concurrency is `concurrent` then if multiple

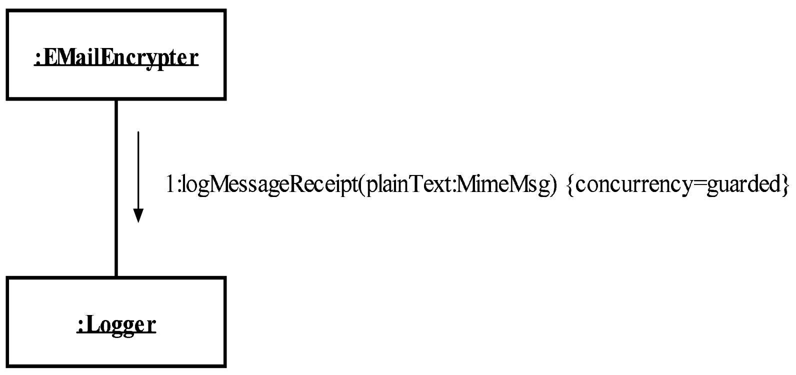

:EMailEncrypter
1:logMessageReceipt(plainText:MimeMsg) {concurrency=guarded}
:Logger

Figure A.6: An Example of Showing Concurrent Method Calls

threads call a method at the same time, only one thread at a time is allowed to execute the method. Figure A.6 shows an example of a synchronized method.

A.4 Sequence Diagrams

A sequence diagram shows an interaction arranged in time sequence. In particular, it shows the instances participating in the interaction by their lifelines and the stimuli they exchange arranged in time sequence. It does not show the associations among the objects.

Sequence diagrams come in several slightly different formats intended for different purposes, like focusing on execution control, concurrency etc. A sequence diagram can exist in a generic form (describes all the possible sequences) and in an instance form (describes one actual sequence consistent with the generic form).

A sequence diagram has two dimensions: 1) the vertical dimension represents time and 2) the horizontal dimension represents different objects. Normally time proceeds down the page. (The dimensions may be reversed, if desired.) Usually only time sequences are important, but in real-time applications the time axis could be an actual metric.

The horizontal ordering of the lifelines is arbitrary. Often call arrows are arranged to proceed in one direction across the page; however, this is not always possible and the ordering does not convey information. Figure A.7 shows an example sequence diagram. vertical dashed lines in the diagram are called the lifeline. The lifeline represents the existence of the Object at a particular time.

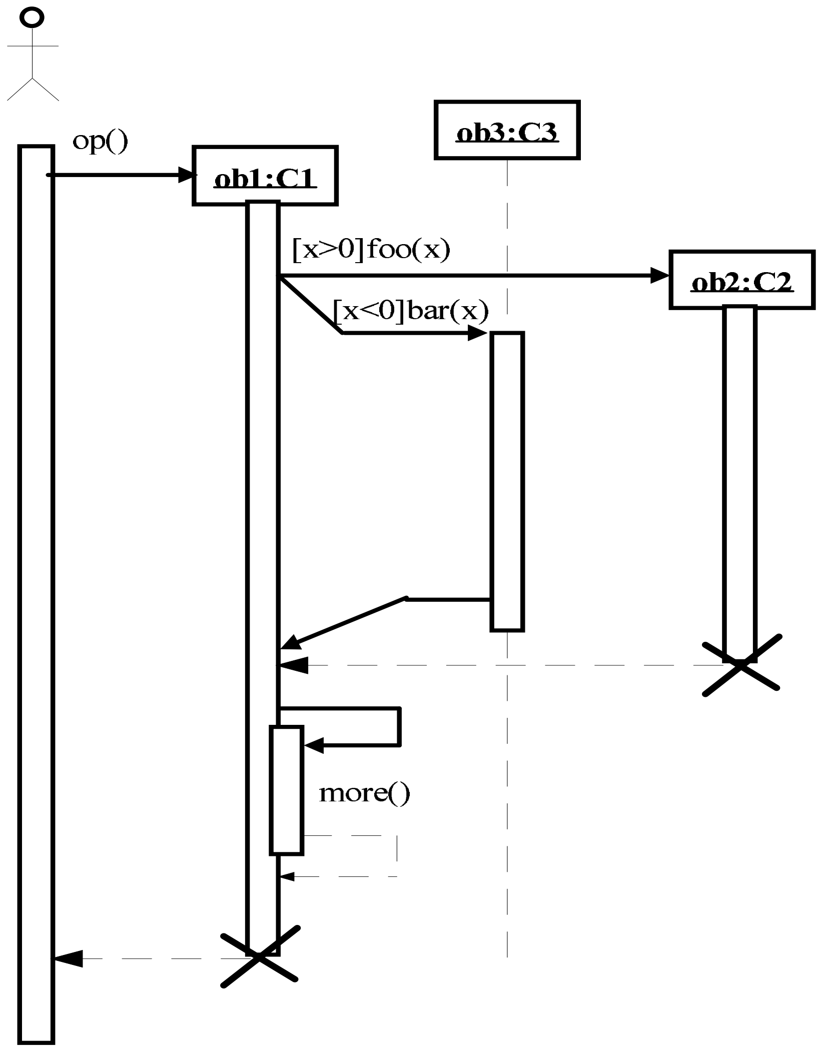


Figure A.7: An Example of Showing Concurrent Method Calls

If the Object is created or destroyed during the period of time shown on the diagram, then its lifeline starts or stops at the appropriate point; otherwise, it goes from the top to the bottom of the diagram. An object symbol is drawn at the head of the lifeline. If the Object is created during the diagram, then the arrow, which maps onto the stimulus that creates the object, is drawn with its arrowhead on the object symbol. If the object is destroyed during the diagram, then its destruction is marked by a large X, either at the arrow mapping to the Stimulus that causes the destruction or (in the case of self-destruction) at the final return arrow from the destroyed Object. An Object that exists when the transaction starts is shown at the top of the diagram (above the first arrow), while an Object that exists when the transaction finishes has its lifeline continue beyond the final arrow.

The tall thin rectangle in the diagram in Figure A.7 is called an activation. An activation rectangle's top is aligned with its initiation time and whose bottom is aligned with its completion time. In the case of concurrent Objects each with their own threads of control, an activation shows the duration when each Object is performing an Operation. Operations by other Objects are not relevant. If the distinction between direct computation and indirect computation (by a nested procedure) is unimportant, the entire lifeline may be shown as an activation. In the case of a recursive call to an Object with an existing activation, the second activation symbol is drawn slightly to the right of the first one, so that they appear to stack up visually. (Recursive calls may be nested to an arbitrary depth.)

In a sequence diagram a Stimulus is shown as a horizontal solid arrow from the lifeline of one Object to the lifeline of another Object. In case of a Stimulus from an Object to itself, the arrow may start and finish on the same Object lifeline. The arrow is labeled with the name of the Stimulus (Operation or Signal) and its argument values or argument expressions.

The function `op()`, in Figure A.7, creates the object `ob1` which is an instance of class `C1`. The object creation is indicated by the stimulating arrow's head ending directly on the object symbol. The object `ob3` for example is not created by object `ob1`, but it is accessed by `ob1`. A cross (X) sign at the end of the life line of an object indicates the termination of that object. Object `ob2` for example, after performing some task, terminates and returns to its caller. The two lines forking from the object `ob1` in Figure A.7, shows a conditional execution. If x is bigger that 0 than the method `foo(x)` gets called, if the reverse is true than the method `bar(x)` gets called.

A.5 Component Diagrams

A component diagram shows the dependencies among software components, including source code components, binary code components, and executable components. Component Diagrams show the structure of actual software components that will be used to implement the system. UML treats the concept of software components in the broad sense to include business procedures and documents as well. A software module may be represented as a component stereotype. Some components exist at compile time, some exist at link time, some exist at run time. The interfaces of a component is shown as a small line with a small, hollow circle

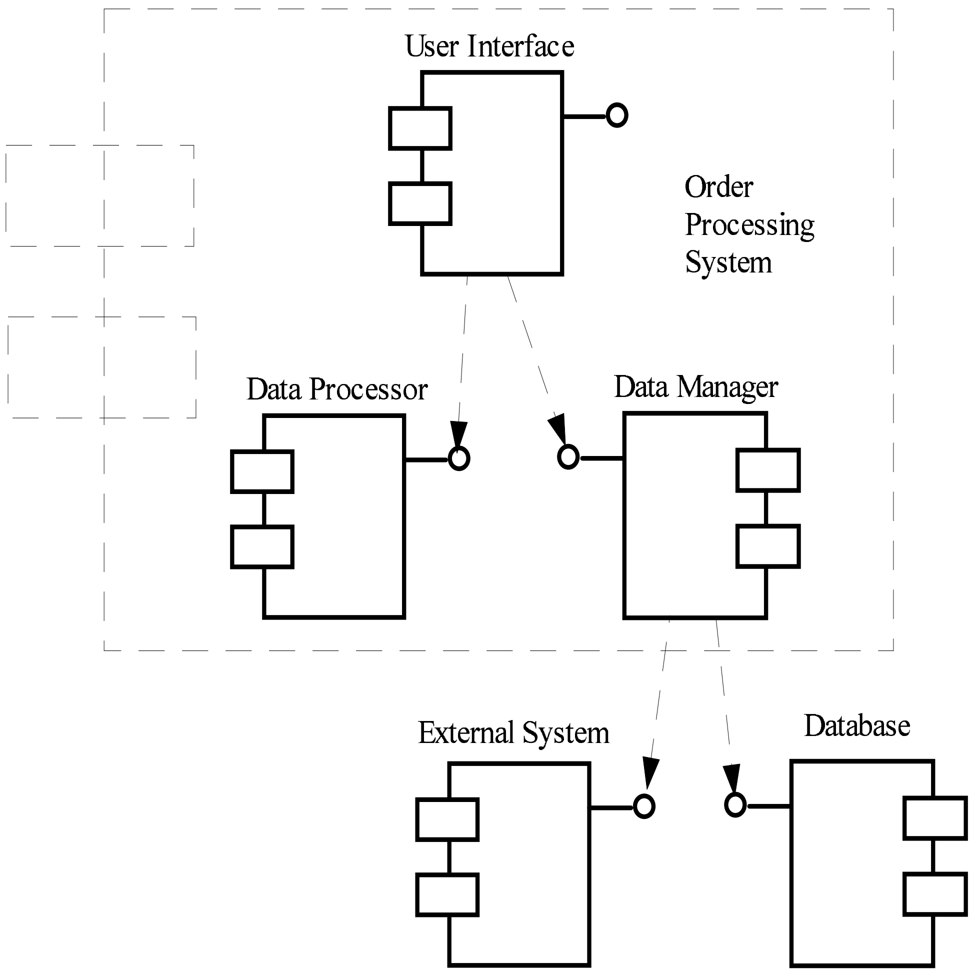


Figure A.8: An Example Component Diagram for an Order Processing System

at its end, attached to one side of the rectangle representing the component. This is illustrated in Figure A.8. A call to the interface of a component is indicated by a dashed line with an arrow pointing to the interface being invoked. Components may also be connected to components by physical containment representing composition relationships. In Figure A.8, `User Interface`, `Data Processor` and `Data Manager` components are contained in the `Order Processing System` component.

Appendix B

AniComp's Concise Reference Manual

B.1 Initialization Functions

```
InitializeEnv();
```

This function initializes the internal structures of AniComp. It must be called as the first function in any interaction with AniComp.

```
LoadWRLFile(\textit{BSTR Filename });
```

This function causes the geometric model file named Filename to be loaded into the environment. The file must be a VRML97 file.

```
LoadGeomObject(BSTR fileName);
```

This function inserts the geometric object defined in filename into the current virtual environment. A node is created which become the parent node of the new object (or collection of objects). This new node is then added to the root node in the scene tree. The name dictionary is updated. The name list and the node tree represented in the dialog window also get updated.

B.2 Object Manipulation Functions

```
CreateSphere(\textit{BSTR objName, FLOAT radius, BSTR textureFile});
```

This function creates a new sphere object whose radius is defined by radius parameter. The object is placed under a new transformation node. This transformation node is then placed under the root node. The textureFile parameter supplies the name of the image file to be wrapped around the object. The image file can be in one of the popular image formats such as .JPG, .TIF. BMP etc. OGLOptimizer uses a library called IFL (Image Format Library)

```
CreateCylinder(BSTR objName, FLOAT bottomRadius, FLOAT height,
BSTR textureFile);
```

This function creates a new cylinder object. The radius of the bottom circle is given by bottomRadius, the height of the cylinder is given by height. The object is placed under a new transformation node. This transformation node is then placed under the root node. The textureFile parameter supplies the name of the image file to be wrapped around the object. The image file can be in one of the popular image formats such as .JPG, .TIF. BMP etc. OGLOptimizer uses a library called IFL (Image Format Library)

```
CreateBox(BSTR Name, VARIANT Size, VARIANT Center,  VARIANT color,
FLOAT transparency, BSTR textureFile);
```

This function creates a new rectangular prism object. Its length, width and height is given by a 3 element vector called Size. The Center parameter defines the position of its center in the environment. The object is placed under a new transformation node. This transformation node is then placed under the root node. The textureFile parameter supplies the name of the image file to be wrapped around the object. The image file can be in one of the popular image formats such as .JPG, .TIF. BMP etc. OGLOptimizer uses a library called IFL (Image Format Library). Utility functions are provided for converting C++ float arrays of 3 elements to the VARIANT type. Include HelperFuncs.h and use Vec3ToVARIANT (VARIANT* pVar, Vec3 Vector); function to do the conversion.

```
GroupObjectsUnder(BSTR groupName, VARIANT objList);
```

This function is used to group existing objects under a new parent node. A new group node with the name groupName is created and the objects whose names are given in the objList are added as children of this new group node. All the grouped objects are deleted from their original place in the scene tree hierarchy.

```
ReplaceObjectWith(BSTR objNameOld, BSTR objNameNew);
```

This function deletes the object named objNameOld and places the object named objNameNew in the place of deleted object. Both object must exist before the operation.

```
DeleteObject(BSTR objName);
```

This function deletes the object named objName.

```
ModifyBox(BSTR Name, VARIANT Size, VARIANT Center, VARIANT color,
FLOAT transparency, BSTR textureFile);
```

This function modifies the properties of an existing box object. Size, Center and color parameters are vectors of three elements. Utility functions are provided for converting C++ float arrays of 3 elements to the VARIANT type. Include HelperFuncs.h and use Vec3ToVARIANT (VARIANT* pVar, Vec3 Vector); function to do the conversion.

B.3 Low Level Transformation Functions

```
SetTransformMatrixElem16(BSTR objName, FLOAT m00, FLOAT m01,
FLOAT m02, FLOAT m03, FLOAT m10, FLOAT m11, FLOAT m12, FLOAT m13,
FLOAT m20, FLOAT m21, FLOAT m22, FLOAT m23, FLOAT m30,
FLOAT m31, FLOAT m32, FLOAT m33);
```

This function sends a 4x4 transformation matrix to the transformation node of the object named objName as a list of 16 (single precision) float values. This is more efficient that packing the matrix as a VARIANT.

```
SendTransformMatrix(BSTR objName,  VARIANT Row1Vec4,
VARIANT Row2Vec4, VARIANT Row3Vec4, VARIANT Row4Vec4);
```

This function is similar to the two above except that it sends the transformation matrix as 4 separate row vectors of 4 elements. Include HelperFuncs.h and use Vector4ToVARIANT(VARIANT* pVar, float* Vector4) function to convert a C++ array of 4 float values to VARIANT.

```
SetTransformMatrix16(BSTR objName, VARIANT MatArray16);
```

This function sends a 4x4 transformation matrix to the transformation node of the object named objName as an array of 16 values packed as a VARIANT. This may be more convenient to use for some cases. A Utility function is provided for converting a C++ float array of 16 elements to the VARIANT type. Include HelperFuncs.h and use Vector16ToVARIANT (VARIANT* pVar, float* Vector16) function to do the conversion.

```
RotateAroundPoint(BSTR objName, FLOAT x, FLOAT y, FLOAT z,
VARIANT rotAxis, FLOAT angle);
```

This function rotates an object around an arbitrary point. The point specified by (x,y,z) is set as the center of the rotation. rotAxis is a vector of three elements which determines the axis of rotation. The angle parameter specifies the angle by which the object is to be rotated.

```
RotateObjectRelative(BSTR objName, FLOAT x, FLOAT y, FLOAT z,
FLOAT angle);
```

This function rotates an object relative to its current orientation. The given axis-angle rotation is multiplied by its existing orientation using quaternion multiplication.

```
TranslateObjectRelative(BSTR objName, FLOAT x, FLOAT y,
FLOAT z);
```

This function apply an instantaneous translation to an object, named objName, relative to its current position.

```
RotateRelativeToInitialOrientation(BSTR objName, FLOAT x, FLOAT y,
FLOAT z, FLOAT angle);
```

This function rotate an object, named objName, relative to its initial orientation. The initial orientation is the one which is defined when the object was first loaded into the environment. This initial orientation is stored internally and can be used later if needed.

```
SetObjectTransCenter(BSTR objName, FLOAT cx, FLOAT cy,  FLOAT cz);
```

This function sets the center of transformation for an object, named objName, to be (cx,cy,cz). All the transformations applied after this command operate relative to this new center point.

```
SetObjectOrientation(BSTR objName, FLOAT x, FLOAT y, FLOAT z,
FLOAT angle);
```

This function applies an absolute rotation to an object specified by the axis (x,y,z) and angle of angle. All the previous orientation information is lost.

```
SetObjectPosition(BSTR objName, FLOAT x, FLOAT y, FLOAT z);
```

This function locates an object to an absolute position in 3D environment given by the point (x,y,z).

B.4 Rigid Body Simulation Functions

```
SetTorqueOnObject(BSTR objName, VARIANT torqueVec3);
```

This function sets a torque value on an object. torqueVec3 is vector of 3 elements (of type float). Include HelperFuncs.h and use Vec3ToVARIANT (VARIANT* pVar, Vec3 Vector); function to do the conversion. Not working properly, yet!

```
SetObjectAngularAccel(BSTR objName, VARIANT angAccelVec3);
```

This function sets an angular acceleration value on an object. angAccelVec3 is a vector containing 3 elements (of type float). Include HelperFuncs.h and use Vec3ToVARIANT (VARIANT* pVar, Vec3 Vector); function to do the conversion from a float vector to VARIANT.

```
SetObjectAngularVeloc(BSTR objName, VARIANT angVelocVec3);
```

This function sets angular velocity value on an object. angVelocVec3 is a vector containing 3 elements (of type float). Include HelperFuncs.h and use Vec3ToVARIANT (VARIANT* pVar, Vec3 Vector); function to do the conversion from a float vector to VARIANT.

```
SetForceOnObject(BSTR objName, VARIANT forceVec3);
```

This function defines a force vector on an object. forceVec3 is a vector containing 3 elements (of type float). Include HelperFuncs.h and use Vec3ToVARIANT (VARIANT* pVar, Vec3 Vector); function to do the conversion from a float vector to VARIANT.

```
SetObjectAcceleration(BSTR objName, VARIANT accelVec3);
```

This function defines an acceleration vector on an object. accelVec3 is a vector containing 3 elements (of type float). Animator component multiplies this vector by the mass of the object and sets its force vector. Include HelperFuncs.h and use Vec3ToVARIANT (VARIANT* pVar, Vec3 Vector); function to do the conversion from a float vector to VARIANT.

```
SetObjectVelocity(BSTR objName, VARIANT velocVec3);
```

This function defines a linear velocity vector on an object. velocVec3 is a vector containing 3 elements (of type float). Include HelperFuncs.h and use Vec3ToVARIANT (VARIANT* pVar, Vec3 Vector); function to do the conversion.

```
SetCollisionOff();
```

This function turns the collision detection off. Internally a global variable is set to false so that non of the collision detection computations are performed.

```
SetCollisionOn();
```

This function turns the collision detection on. Internally a global variable is set to true. Collision detection becomes effective.

```
RegisterRgBodyForCollision(BSTR objName);
```

This function registers an object named objName for collision detection. The object must have been defined as a rigid body. This function causes the object geometry to be tesselated (triangulated). Its triangle set are then fed into the collision detection engine. Once an object is registered for collision an event is fired if it is involved in a collision. The calling party must implement a call-back function to handle the event.

```
UnRegisterRgBodyFromCollision(BSTR objName);
```

This function removes an object from the list of objects that are checked against collision.

```
SetGravityOff();
```

This function sets the gravity off.

```
SetGravityOn();
```

This function sets the gravity on.

```
DefineFloor(BSTR objName,BSTR texFileName, FLOAT width,
FLOAT depth);
```

This function defines a floor. If the object with objName already exists then it is defined as a rigid body (if not already defined) and it is registered for collision. This is useful if the geometric model file does not contain a flat floor.

```
DefineRigidBody(BSTR objName, FLOAT mass, VARIANT inerTensRow1Vec3,
VARIANT inerTensRow2Vec3, VARIANT inerTensRow3Vec3);
```

This function

```
StopObject(BSTR objName);
```

This stops a moving rigid body by assigning a zero vector to its physical quantities.

B.5 High level motion generation Functions

```
DefineBehaviour(BSTR behaviourName, BSTR objName);
```

This function defines a behaviour and attaches it to an object named objName. A new behaviour object is defined and initialized. One position interpolator, one orientation interpolator and one dedicated clock are also defined and initialized as data members of the behaviour object. After this function is called the defined behaviour is ready to accept incremental position and orientation values to define the motion sequence.

```
AddItemToPosInterpolator(BSTR objName, BSTR behaviourName,
FLOAT key, VARIANT posVec3);
```

This function adds a key and a key value to the position interpolator of a defined behaviour. The keys should be generated as fractional values within the range of [0,1]. The first key being 0 and the last one being 1. For each key, there should exist one and only one key value (position vector). If you are not comfortable with the idea of creating proper keys, enter a -1 for all the keys and after finishing creating all the position values, call RearrangePosInterpKeys to force AniComp to generate equally spaced keys in a correct way. posVec3 is a 3 element vector specifying the absolute position of the object corresponding to a key. Include HelperFuncs.h and use Vec3ToVARIANT (VARIANT* pVar, Vec3 Vector); function to do the conversion from a float vector to VARIANT.

```
AddItemToOrientInterpolator(BSTR objName, BSTR behaviourName,
FLOAT key, VARIANT RotVec4);
```

This function adds a key and a key value to the orientation interpolator of a defined behaviour. The keys should be generated as fractional values within the range of [0,1]. The first key being 0 and the last one being 1. For each key, there should exist one and only one key value (orientation vector). If you are not comfortable with the idea of creating proper keys, enter a -1 for all the keys and after finishing creating all the position values, call RearrangePosInterpKeys to force AniComp to generate equally spaced keys in a correct way. RotVec4 is a 4 element vector in the form of (x,y,z,a) where (x,y,z) is the normalized rotation axis and a is the an angle (in radian) which indicates the amount of incrimental rotation. Include HelperFuncs.h and use Vector4ToVARIANT (VARIANT* pVar, float* Vector4) function to do the conversion from a float vector to VARIANT.

```
RearrangeOrientInterpKeys(BSTR ObjName, BSTR BehaviourName);
```

```
RearrangePosInterpKeys(BSTR ObjName, BSTR BehaviourName);
```

The two functions above are utility functions that force AniComp to re-generate keys for orientation and position interpolators of a given behaviour. This is useful if you are not sure how to create the keys for successive position and orientation values. The internal CBehaviour object finds the number of values in the interpolators and calculates equally spaced keys between 0 and 1.

```
SaveBehaviours(BSTR fileName);
```

This function saves all the behaviours defined into the file name given by fileName. The format is binary. That is the objects containing the behaviour information are serialized to the given file. The function also outputs a text file which contains the behaviour names and intepolator values for convenience. But this text file is not highly readable.

```
LoadBehaviours(BSTR fileName);
```

This function loads the previously defined and saved behaviours form the file whose name is given by fileName. The file should be a valid file. The binary content of the file is serialized back to the internal objects. Behaviours are attached to their objects and become ready for activation.

```
ActivateBehaviour(BSTR objName, BSTR behaviourName);
```

This function causes a behaviour to be activated. Activation process involves starting the clocks and connecting the interpolators to the transformation node of the relevant object. The behaviour must be defined or loaded before this function can operate.

```
SynchronizeBehaviours(  VARIANT BehaviourList, BSTR ClockName,
FLOAT timeInterv);
```

This function synchronizes the clocks of different behaviours so that the time range of the behaviour completion is equalized. This is useful if a number of behaviours were defined to last for different amount of time ranges but you need to make them start and finish at exactly the same time.

```
ComposeObjBehaviours(BSTR objName, BSTR newBehavName,
VARIANT behaviourList);
```

This function combines two or more behaviours and makes them one single behaviour. It is the user's responsibility to make sure that the motion sequences of consecutive behaviours follow each other with a smooth transition. The new behaviour can then be activated to visualize the whole sequence.

```
GetObjectBehaviours(BSTR objName, /*[out, retval]*/ VARIANT* behavList);
```

Not implemented yet!

```
GetAllBehaviourList(/*[out, retval]*/ VARIANT* behavList);
```

Returns the list of all the behaviours defined in the environment. The behavList is a string list packed as a VARIANT. Include HelperFuncs.h and use VARIANTToStringList (VARIANT Var, CStringList* pStrList) function to do the conversion from a VARIANT to CStringList class. This class is defined by MFC. It is much easier to manage.

```
TransfMatrixToInterpolators(BSTR objName, BSTR behavName,
FLOAT key, VARIANT Row1V4, VARIANT Row2V4,
VARIANT Row3V4,  VARIANT Row4V4);
```

This function sends a 4x4 transformation matrix as four row vectors, to a behaviour object to fill in its interpolators. It does the job of RearrangeOrientInterpKeys and RearrangePosInterpKeys functions at once. The position and orientation values are extracted from the matrix and entered into the interpolators.

B.6 Utility functions

```
GetObjNameList(/*[out,retval]*/VARIANT* nameList);
```

This function returns the names of all the geometric objects in the virtual environment as a string list. It does this by consulting the internal name dictionary of AniComp. The type of nameList is a VARIANT. Include HelperFuncs.h and use VARIANTToStringList (VARIANT Var, CStringList* pStrList) function to do the conversion from a VARIANT to CStringList class. This class is defined by MFC. It is much easier to manage.

```
CompileObject(BSTR objName);
```

This function is an optimization function. It causes a part of the scene graph to be converted to OpenGL display lists. OpenGL display lists optimizes the speed of the graphics operations. However the objects must be stable. If they are modified they should be re-compiled. objName parameter defines the transformation node of an object. All the children of that node are optimized. Use this if you are sure that the object is in its final form.

```
WriteText(BSTR textObjName, BSTR text, VARIANT posVec3,
VARIANT colorVec3RGB);
```

Not implemented yet!

```
DoesObjectExists(BSTR objName,  /*[out,retval]*/INT* exists);
```

Returns 1 if the object exists, 0 otherwise.

```
GetBoundingBox(BSTR objName,  VARIANT* sizeVec3);
```

This function returns the bounding box of a given object as a vector of three elements (sizeVec3). Include HelperFuncs.h and use VARIANTToVec3 (VARIANT Var, Vec3* pVec) function to do the conversion from a VARIANT to a vector of three floats. The elements of the returned vector give the dimensions of the bounding box are x,y,z axes respectively.

```
MoveCurrentViewPoint(  FLOAT x, FLOAT y, FLOAT z);
```

This function moves current camera by (x,y,z) relative to its current position.

```
AddPointLightToEnv(BSTR lightName, FLOAT x, FLOAT y,
FLOAT z, FLOAT intensity);
```

This function adds a point light to the environment. Point (x,y,z) is the absolute position of the new light. The intensity parameter adjusts the brightness of the light.

```
GetObjectPosition(BSTR objName, /*[out, retval]*/ VARIANT* posVec3);
```

This function returns the current absolute position of the object whose name is given by objName. The returnes value is a vector of three elements packed as a VARIANT. Include HelperFuncs.h and use VARIANTToVec3 (VARIANT Var, Vec3* pVec) function to do the conversion from a VARIANT to a vector of three floats.

```
DefineEnvironmentClock(BSTR clockName, SHORT isLooping,
FLOAT timeInterv);
```

This function defines a global clock to be used for synchronizing behaviours. This clock can then be passed as a parameter to SynchronizeBehaviours function. If isLooping is 1 the clock loops, causing the behaviours to be replayed continuously. If it is 0 then the behaviours are re-played only during the time specified by timeInterv parameter. This parameter is defined in terms of seconds.

```
AttachViewPtToObject(BSTR camName, BSTR objName,
VARIANT orientAxis, FLOAT orientAng,  VARIANT distance);
```

This function attaches a camera to an object. As the object moves the camera follows it from a given relative position (distance) and orientation. This is useful if you want to follow the movement of an object (e.g. a human, or a car).

```
ZoomInToObject(BSTR objName, BSTR camName);
```

Not implemented yet!

```
ChangeViewPoint(BSTR winName, BSTR camName);
```

This function changes the current view point. The viewing camera becomes the one whose name given by camName, which should have been created before.

```
CreateLight(BSTR lightName,  BSTR objName, VARIANT color,
FLOAT intensity, FLOAT posX, FLOAT posY,  FLOAT posZ);
```

This function creates a directional light. Its position, color, and intensity can be defined. A directional light is different from a point light, it simulates the sun light.

```
CreateNewWindow(BSTR winName);
```

Not implemented yet!

```
CreateNewCamera(BSTR camName, VARIANT posVec3,
VARIANT orientVec3, FLOAT orntAngle);
```

This function defines a new camera. Its position and orientation (look at point) can be specified. After a new camera is created the ChangeViewPoint function should be called to make it the current view point.

Appendix C

AniComp's Main Internal Classes

```
/*****************************************************/
/* Animate.h - Include file for CAnimate class         */
/* Author   : M.N Alpdemir                             */
/* Date     : 8/3/2000                                 */
/*****************************************************/
class csPerspCamera;
#ifndef NO_AUDIO
class csMicrophone;
class csSoundAction;
#endif
class csEnvironment;
class csDrawAction;
class csTransform;
class csPtrArray;
class csContext;
class csGroup;
class csStringDict;
class csStringArray;
class csTransformAction;
class CNameListAction;
class CRobotArm;
class CViewManager;

#include "resource.h"       // main symbols
```

```
#include "DialogWnd.h"
#include "SceneWindow.h"
#include "RigidBody.h"
#include "AnimatorCP.h"

typedef enum {X_AXIS,Y_AXIS,Z_AXIS,Y_AXIS_OPS,OBJECT_VIEW} ViewAxis;

class ATL_NO_VTABLE CAnimate :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CAnimate, &CLSID_Animate>,
   public IConnectionPointContainerImpl<CAnimate>,
   public IDispatchImpl<IAnimate, &IID_IAnimate, &LIBID_ANIMATORLib>,
   public IProvideClassInfo2Impl<&CLSID_Animate, &DIID__IAnimateEvents>,
   public CProxy_IAnimateEvents< CAnimate >,
   public IDispatchImpl<IActorProxy, &IID_IActorProxy,
          &LIBID_ANIMATORLib>
{

protected:

//Internal Attributes

float  m_MaxSceneRange; //The max scene width to make sure that
// everything is visible
csTransform  *m_pRoomBoundary;
CNameListAction  *m_pNameListAction;
csTransformAction *m_pObjXfAction;
CRobotArm  *m_pRobotArm;
csPerspCamera  *m_pZCamera;
csPerspCamera  *m_pXCamera;
csPerspCamera  *m_pYCamera;
csPerspCamera    *m_pOpposYCamera;
csPerspCamera  *m_pAttachedCamera;
ViewAxis  m_CurrentViewAxis;
csPtrArray  m_EnvLights;
```

```
CListBox  m_NameList;
CTreeCtrl        m_SceneTree;
CListBox  m_MessgBox;
    CWnd  *m_pThisWnd;
DialogWnd  *m_pDialgWnd;
csNode  *m_OldModel;
int              m_ind;
csMatrix4f  m_RotZTheta,
 m_RotXAlpha,
 m_TrsZd,
 m_TrsXa;
bool  m_FromY;
bool  m_InitialLoad;


public: //methods for internal usage

void ShowNameList(csStringArray *pNameList);
void ShowMessage(CString messg);
void MakeDialogWnd();
void LclMoveWithVelocity(csString* objName,
     csVec3f velocVec, float time);
void LclDefineRigidBody(csString* objName);
void SetRotTheta(float theta);
void SetRotAlpha(float alpha);
void SetTrsZd(float d);
void SetTrsXa(float a);
void MakeRoomBoundaryVisibile(float x, float y, float z);
void SendCollisionEvent(csString obj1 ,csString obj2);
public:
CAnimate();
~CAnimate();
```

```
DECLARE_REGISTRY_RESOURCEID(IDR_ANIMATE)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CAnimate)
COM_INTERFACE_ENTRY(IAnimate)
//DEL  COM_INTERFACE_ENTRY(IDispatch)
COM_INTERFACE_ENTRY(IConnectionPointContainer)
COM_INTERFACE_ENTRY(IProvideClassInfo)
COM_INTERFACE_ENTRY(IProvideClassInfo2)
COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer)
COM_INTERFACE_ENTRY2(IDispatch, IAnimate)
COM_INTERFACE_ENTRY(IActorProxy)
END_COM_MAP()

BEGIN_CONNECTION_POINT_MAP(CAnimate)
CONNECTION_POINT_ENTRY(DIID__IAnimateEvents)
END_CONNECTION_POINT_MAP()

// IAnimate
public:
STDMETHOD(CompileObject)(/*[in]*/ BSTR objName);
STDMETHOD(SetTransformMatrixElem16)(/*[in]*/ BSTR objName,
/*[in]*/ FLOAT m00, /*[in]*/ FLOAT m01, /*[in]*/ FLOAT m02,
/*[in]*/ FLOAT m03, /*[in]*/ FLOAT m10, /*[in]*/ FLOAT m11,
/*[in]*/ FLOAT m12, /*[in]*/ FLOAT m13, /*[in]*/ FLOAT m20,
/*[in]*/ FLOAT m21, /*[in]*/ FLOAT m22, /*[in]*/ FLOAT m23,
/*[in]*/ FLOAT m30, /*[in]*/ FLOAT m31, /*[in]*/ FLOAT m32,
/*[in]*/ FLOAT m33);
STDMETHOD(SetTransformMatrix16)(/*[in]*/ BSTR objName,
/*[in]*/ VARIANT MatArray16);
STDMETHOD(MoveCurrentViewPoint)(/*[in]*/ FLOAT x, /*[in]*/
FLOAT y, /*[in]*/ FLOAT z);
STDMETHOD(RotateRelativeToInitialOrientation)(/*[in]*/
BSTR objName, /*[in]*/ FLOAT x, /*[in]*/ FLOAT y,
/*[in]*/ FLOAT z, /*[in]*/ FLOAT angle);
STDMETHOD(SetObjectTransCenter)(/*[in]*/ BSTR objName,
/*[in]*/ FLOAT cx, /*[in]*/ FLOAT cy,  /*[in]*/ FLOAT cz);
STDMETHOD(ModifyBox)(/*[in]*/ BSTR Name, /*[in]*/ VARIANT Size,
/*[in]*/ VARIANT Center, /*[in]*/ VARIANT color,
/*[in]*/ FLOAT transparency, /*[in]*/ BSTR textureFile);
STDMETHOD(ReplaceObjectWith)(/*[in]*/ BSTR objNameOld,
/*[in]*/ BSTR objNameNew);
STDMETHOD(DeleteObject)(/*[in]*/ BSTR objName);
STDMETHOD(CreateLight)(/*[in]*/ BSTR lightName, /*[in]*/
BSTR objName, /*[in]*/ VARIANT color, /*[in]*/ FLOAT intensity,
/*[in]*/ FLOAT posX, /*[in]*/ FLOAT posY,/*[in]*/ FLOAT posZ);
STDMETHOD(GroupObjectsUnder)(/*[in]*/ BSTR groupName,
/*[in]*/ VARIANT objList);
STDMETHOD(CreateSphere)(/*[in]*/ BSTR objName, /*[in]*/
FLOAT radius, /*[in]*/ BSTR textureFile);
STDMETHOD(CreateCylinder)(/*[in]*/ BSTR objName, /*[in]*/
FLOAT bottomRadius, FLOAT height, /*[in]*/ BSTR textureFile);
STDMETHOD(RotateAroundPoint)(/*[in]*/ BSTR objName, /*[in]*/
FLOAT x, /*[in]*/ FLOAT y, /*[in]*/ FLOAT z, /*[in]*/
VARIANT rotAxis, /*[in]*/ FLOAT angle);
STDMETHOD(WriteText)(/*[in]*/ BSTR textObjName, /*[in]*/
BSTR text, /*[in]*/ VARIANT posVec3, /*[in]*/ VARIANT colorVec3RGB);
STDMETHOD(AddPointLightToEnv)(/*[in]*/ BSTR lightName,
/*[in]*/ FLOAT x, /*[in]*/ FLOAT y, /*[in]*/ FLOAT z,
/*[in]*/ FLOAT intensity);
STDMETHOD(GetObjectPosition)(/*[in]*/ BSTR objName,
/*[out, retval]*/ VARIANT* posVec3);
STDMETHOD(SetObjectOrientation)(/*[in]*/ BSTR objName,
/*[in]*/ FLOAT x, /*[in]*/ FLOAT y, /*[in]*/ FLOAT z,
/*[in]*/ FLOAT angle);
STDMETHOD(SetObjectPosition)(/*[in]*/ BSTR objName,
/*[in]*/ FLOAT x, /*[in]*/ FLOAT y, /*[in]*/ FLOAT z);
STDMETHOD(SetGravityOff)();
STDMETHOD(SetGravityOn)();
STDMETHOD(SetWallDistance)(/*[in]*/ FLOAT widthX,
/*[in]*/ FLOAT heightY, /*[in]*/ FLOAT depthZ);
STDMETHOD(DefineFloor)(/*[in]*/ BSTR objName,/*[in]*/
BSTR texFileName, /*[in]*/ FLOAT width, /*[in]*/ FLOAT depth);
STDMETHOD(TriangulateObject)(/*[in]*/ BSTR objName);
STDMETHOD(SetCollisionOff)();
STDMETHOD(SetCollisionOn)();
STDMETHOD(UnRegisterRgBodyFromCollision)(/*[in]*/ BSTR objName);
STDMETHOD(RegisterRgBodyForCollision)(/*[in]*/ BSTR objName);
STDMETHOD(GetObjectBehaviours)(/*[in]*/ BSTR objName,
/*[out, retval]*/ VARIANT* behavList);
STDMETHOD(GetAllBehaviourList)(/*[out, retval]*/
VARIANT* behavList);
STDMETHOD(TransfMatrixToInterpolators)(/*[in]*/ BSTR objName,
/*[in]*/ BSTR behavName,/*[in]*/FLOAT key, /*[in]*/
VARIANT Row1V4, /*[in]*/ VARIANT Row2V4, /*[in]*/
VARIANT Row3V4, /*[in]*/  VARIANT Row4V4);
STDMETHOD(DefineEnvironmentClock)(/*[in]*/ BSTR clockName,
/*[in]*/ SHORT isLooping, /*[in]*/ FLOAT timeInterv);
STDMETHOD(SynchronizeBehaviours)(/*[in]*/ VARIANT BehaviourList,
/*[in]*/ BSTR ClockName, /*[in]*/ FLOAT timeInterv);
STDMETHOD(RearrangeOrientInterpKeys)(/*[in]*/ BSTR ObjName,
/*[in]*/ BSTR BehaviourName);
STDMETHOD(RearrangePosInterpKeys)(/*[in]*/ BSTR ObjName,
/*[in]*/ BSTR BehaviourName);
STDMETHOD(SendTransformMatrix)(/*[in]*/ BSTR objName,
/*[in]*/ VARIANT Row1Vec4, /*[in]*/ VARIANT Row2Vec4,
/*[in]*/ VARIANT Row3Vec4 , /*[in]*/ VARIANT Row4Vec4);
STDMETHOD(DefineBehaviour)(/*[in]*/ BSTR behaviouName,
/*[in]*/ BSTR objName);
STDMETHOD(SaveBehaviours)(/*[in]*/ BSTR fileName);
STDMETHOD(DefineRigidBody)(/*[in]*/ BSTR objName,
/*[in]*/ FLOAT mass, /*[in]*/ VARIANT inerTensRow1Vec3,
/*[in]*/ VARIANT inerTensRow2Vec3,
/*[in]*/ VARIANT inerTensRow3Vec3);
```

```
STDMETHOD(GetObjNameList)(/*[out,retval]*/ VARIANT* nameList);
STDMETHOD(ComposeObjBehaviours)(/*[in]*/ BSTR objName,
/*[in]*/ BSTR newBehavName,/*[in]*/ VARIANT behaviourList);
STDMETHOD(ActivateBehaviour)(/*[in]*/ BSTR objName,
/*[in]*/ BSTR behaviourName);
STDMETHOD(LoadBehaviours)(/*[in]*/ BSTR fileName);
STDMETHOD(ZoomInToObject)(/*[in]*/ BSTR objName,
/*[in]*/ BSTR camName);
STDMETHOD(AttachViewPtToObject)(/*[in]*/ BSTR camName,
/*[in]*/ BSTR objName, /*[in]*/VARIANT orientAxis,
/*[in]*/FLOAT orientAng,/*[in]*/ VARIANT distance);
STDMETHOD(SetTorqueOnObject)(/*[in]*/ BSTR objName,
/*[in]*/ VARIANT torqueVec3);
STDMETHOD(SetObjectAngularAccel)(/*[in]*/ BSTR objName,
/*[in]*/ VARIANT angAccelVec3);
STDMETHOD(SetObjectAngularVeloc)(/*[in]*/ BSTR objName,
/*[in]*/ VARIANT angVelocVec3);
STDMETHOD(ChangeViewPoint)(/*[in]*/ BSTR winName,
/*[in]*/ BSTR camName);
STDMETHOD(CreateNewWindow)(/*[in]*/ BSTR winName);
STDMETHOD(CreateNewCamera)(/*[in]*/ BSTR camName,
/*[in]*/ VARIANT posVec3, VARIANT orientVec3,
/*[in]*/ FLOAT orntAngle);
STDMETHOD(StopObject)(/*[in]*/ BSTR objName);
STDMETHOD(SetForceOnObject)(/*[in]*/ BSTR objName,
/*[in]*/ VARIANT forceVec3);
STDMETHOD(SetObjectAcceleration)(/*[in]*/ BSTR objName,
/*[in]*/ VARIANT accelVec3);
STDMETHOD(SetObjectVelocity)(/*[in]*/ BSTR objName,
/*[in]*/ VARIANT velocVec3);
STDMETHOD(AddItemToPosInterpolator)(/*[in]*/ BSTR objName,
/*[in]*/ BSTR behaviourName,  /*[in]*/ FLOAT key,
/*[in]*/ VARIANT posVec3);
STDMETHOD(AddItemToOrientInterpolator)(/*[in]*/ BSTR objName,
/*[in]*/ BSTR behaviourName,  /*[in]*/ FLOAT key,
/*[in]*/ VARIANT RotVec4);
STDMETHOD(DoesObjectExists)(/*[in]*/ BSTR objName,
/*[out,retval]*/ INT* exists);
STDMETHOD(GetBoundingBox)(/*[in]*/ BSTR objName,
/*[out,retval]*/ VARIANT* sizeVec3);
STDMETHOD(RotateObjectRelative)(/*[in]*/ BSTR objName,
/*[in]*/ FLOAT x, /*[in]*/ FLOAT y, /*[in]*/ FLOAT z,
/*[in]*/ FLOAT angle);
STDMETHOD(TranslateObjectRelative)(/*[in]*/ BSTR objName,
/*[in]*/ FLOAT x, /*[in]*/ FLOAT y, /*[in]*/ FLOAT z);
STDMETHOD(LoadGeomObject)(/*[in]*/ BSTR fileName);
STDMETHOD(CreateBox)(/*[in]*/ BSTR Name, /*[in]*/ VARIANT Size,
/*[in]*/ VARIANT Center, /*[in]*/VARIANT color,
/*[in]*/FLOAT transparency, /*[in]*/ BSTR textureFile);
STDMETHOD(InitializeEnv)();
STDMETHOD(LoadWRLFile)(/*[in, string]*/ BSTR Filename );
// IActorProxy
};

/*****************************************************/
/* Behaviours.h - Include file for CBehaviour class    */
/*                and CBehaviouList class              */
/* Author   : M.N Alpdemir                             */
/* Date     : 16/7/1999 (last date modified)           */
/*****************************************************/

#ifndef Behaviours_h
#define Behaviours_h



#include <Cosmo3D/csTransform.h>
#include <Cosmo3D/csTimeSensor.h>
#include <Cosmo3D/csSphereSensor.h>
#include <Cosmo3D/csOrientationInterpolator.h>
#include <Cosmo3D/csPositionInterpolator.h>
```

```
#include <Cosmo3D/csArrays.h>
#include <Cosmo3D/csBasic.h>
#include "Stdafx.h"

typedef enum {SIMPLE, COMPOSED} BehaviourType;

class CBehaviour {
protected:
csString m_BehaviourName;
// belongs to this RBody
csString m_RigidBodyName;
csPositionInterpolator    *m_pTranslateAction;
csOrientationInterpolator  *m_pRotateAction;
csTimeSensor    *m_pDedicatedClock;
BOOL m_Synchronized;
BehaviourType m_BehaviourMode;
csPtrArray    *m_pSimpleBehavList;
// used for serialization
CStringList m_SimpBehavNameList;
csTimeSensor    *m_pSynchroClock;
csTransform    *m_pConnectedTo;
unsigned int m_PoIKeyIndex;
unsigned int m_OrIKeyIndex;


//Public member functions
public:
CBehaviour();
~CBehaviour();
void   SetTimeInterval();
void   AddToPosInterpolator(float key, csVec3f pos);
void   AddToOrientInterpolator(float key, csRotation rot);
void   Activate(float timeInt, bool loop);
void   Activate();
void   Deactivate();
void   WriteASCII(CArchive& ar);
void   WriteBinary(CArchive& ar);
void   ReadBinary(CArchive& ar);
csPositionInterpolator* GetTranslateAction()
{ return m_pTranslateAction;}
csOrientationInterpolator* GetRotateAction()
{ return m_pRotateAction;}
csString   GetBehaviourName() { return m_BehaviourName;}
void   SetBehaviourName(csString& str)
{ m_BehaviourName = str;}
unsigned int  GetPosIntKeyIndex(){ return m_PoIKeyIndex; }
unsigned int  GetOrientIntKeyIndex(){ return m_OrIKeyIndex; }
csString   GetRigidBodyName() { return m_RigidBodyName;}
void   SetRigidBodyName(csString& str)
{ m_RigidBodyName = str;}
void   AttachToRigidBody(csTransform* xf)
{ m_pConnectedTo = xf;}
void   AttachToObject(csTransform* xf)
{ m_pConnectedTo = xf;}
csTimeSensor* GetClock() { return m_pDedicatedClock;}
void   SetSynchroClock(csTimeSensor* pTS)
{ m_pSynchroClock = pTS;}
csTimeSensor* GetSynchroClock() { return m_pSynchroClock;}
void   SetSynchronizedOn() { m_Synchronized = TRUE; }
void   SetSynchronizedOff() { m_Synchronized = FALSE; }
void   SetBehaviourMode(int mode)
{ m_BehaviourMode = mode ? COMPOSED : SIMPLE;}
BehaviourType GetBehaviourMode() { return m_BehaviourMode; }
void   SynchronizeSimpleBehavList(csTimeSensor* pTS);
void   MakeSimpBehavNameList();
void   AppendToNameList(CString name);
void   InitSimpBehavList()
{ m_pSimpleBehavList = new csPtrArray; }
void   AppendToSimpBehavList(CBehaviour* pBhv)
{ m_pSimpleBehavList->append((void*)pBhv);}
```

```
CStringList*  GetSimpBehavNameList()
{return &m_SimpBehavNameList; }
void   UpdateClock()
{ m_pDedicatedClock->updateSensors();}
void   SetTimeInterval(double time);
void   SetTimeLoopFlag(bool loop);
void   RearrangePosIntKeys();
void   RearrangeOrientIntKeys();
// void   SetStartTime


};


class CBehaviourList {
public:
//constructor
CBehaviourList() { };
//public data members
csPtrArray m_Behaviours;
public:
void WriteOutASCII(CString FName);
void WriteOutBinary(CString FName, CFile* pFile);
void ReadInBinary(CString FName, CFile* pFile);
void UpdateClocks();
int  GetNumberOfBehaviours()
{ return m_Behaviours.getCount();}
csPtrArray* GetBehaviourList()
{ return &m_Behaviours;}
BOOL SynchronizeBehaviourClocks(csTimeSensor* pTS,
csString behavName);

};

#endif
```

```
/*********************************************************/
/* Collision.h - Include file for CCollisionmanager class  */
/*                and CPolyObject class                     */
/* Author   : M.N Alpdemir                                  */
/* Date     : 27/10/1999 (last date modified)               */
/*********************************************************/

#ifndef Collision_h
#define Collision_h

#include "VCollide.h"
#include "Quaternion.h"
#include "RigidBody.h"
#include <vector>// STL header file for dynamic array
using namespace std; // for STL vector template

#include <Optimizer/Optimizer.h>
#include <Optimizer/opInit.h>
#include <Optimizer/opTriangle.h>
#include "CommonTypeDefs.h"
class CSimEnvironment;

typedef float r;

class CPolyObject {

protected:

csString m_Name;
int m_Id;
// an Array of opTriangle objects
csPtrArray     m_Triangles;
// a pointer to transf. node representing this object
```

```
CRigidBody     *m_pRigidBody;
Quaternion     *m_Orientation;
csVec3f *m_Position;
//the delta change in velocity per frame.
csVec3f *m_Veloc;
//the delta change in orientation per frame.
csVec3f *m_Omega;
vector<csVec3f>m_BoundingVertices;
int m_CollVertInd;

public:
void SetName(csString name) { m_Name = name; }
csString GetName() { return m_Name; }
void AppendTriangle(opTriangle* tri)
{m_Triangles.append((void*)tri);}
opTriangle* GetTriangle(int i)
{return (opTriangle*)m_Triangles.get(i);}
void CalculateBoundingVertices(csVec3f c);

CPolyObject();
~CPolyObject() { };

friend class CCollisionManager;
};

class CCollisionManager {

// number of triangulated objects
int m_NumOfPolyObjects;
VCollide m_Vc;
csPtrArray m_PolyObjArray; // array of CPolyObject's
CSimEnvironment *m_pEnvManager;

float WorldXLength, WorldYLength, WorldZLength;
enum { NumberOfWalls = 6 };
```

```
    struct wall
    {
        // define wall by plane equation
        csVec3f Normal;  // inward pointing
        float d;         // ax + by + cz + d = 0
    } aWalls[NumberOfWalls];

public:
int GetNumOfObjects() { return m_NumOfPolyObjects; }
void ComputeResponse(csVec3f normal,
CPolyObject* pPolyObj, csVec3f collVertex);
CPolyObject* GetObjectWithId(int id);
int GetIDofObject(csString name);
void CollisionTestReportAndRespond(void);
bool ActivateObjectPair(csString nm1, csString nm2);
void ActivateAll();
CollisionState UpdateOneStep(CPolyObject* pPoly,
double trsMat[16], bool fillMatrix);
void UpdatePolytope(int id, double trans[4][4]);
CollisionState Simulate();
void CreateNewVCollideObject(csString nm,
CRigidBody* pRgdBody, csPtrArray* pTriArray,csVec3f* v,
csVec3f* omega);
void SetWorldSize(float x, float y, float z);
void CalculateWallDistances();
CollisionState  CheckForWallCollisions();
CollisionState  UpdateOneStepForWallColl(CPolyObject* pPoly);
CollisionState  HandleWallCollision();

CCollisionManager(CSimEnvironment* pEnvMan);
~CCollisionManager() { };
};

inline double RAD2DEG(double x)
  {
    return (180.0*(x) / M_PI);
  }

#endif


/****************************************************************/
/* NameListAction.h - Include file for CNameListAction class  */
/*                                                            */
/* Author   : M.N Alpdemir                                    */
/* Date     : 5/3/1999 (last date modified)                   */
/****************************************************************/


#ifndef NameListAction_h
#define NameListAction_h
#include "Stdafx.h"

class csAction;
class csStringArray;

class CNameListAction {

public:
CNameListAction(CTreeCtrl* treeCtrl)
{ m_pTreeCtrl = treeCtrl; };
~CNameListAction() { };
void  FillNameLists(csNode *node,
HTREEITEM item, csStringDict* NameDict);
csStringArray* getNameList()
{ return &m_StrList; }

private:
csStringArray  m_StrList;
CTreeCtrl*     m_pTreeCtrl;
```

```
};

#endif


/*******************************************************/
/* RigidBody.h - Include file for CRigidBody class      */
/*                                                      */
/* Author   : M.N Alpdemir (adapted from Chris Hecker,  */
/*                          Game Developer Magazine)    */
/* Date     : 14/9/1999 (last date modified)            */
/*******************************************************/

#ifndef RigidBody_h
#define RigidBody_h

#include "Quaternion.h"
#include "Behaviours.h"

class csString;
class csVec3fArray;
class csMatrix4f;
class csVec3f;
class csTransform;

class CRigidBody {
public:
csString Name;
float OneOverMass;
    csMatrix4f InverseBodyInertiaTensor;
csRotation InitialOrient;
    float CoefficientOfRestitution;
int GravityActive;
csVec3f Gravity;
```

```
    csVec3fArray aBodyBoundingVertices;
int SourceConfigurationIndex;
    int TargetConfigurationIndex;

    enum { NumberOfConfigurations = 2 };

    struct configuration
    {
        // primary quantities
        csVec3f CMPosition;
        Quaternion Orientation;

        csVec3f CMVelocity;
        csVec3f AngularMomentum;

        csVec3f CMForce;
        csVec3f Torque;

        // auxiliary quantities
        csMatrix4f InverseWorldInertiaTensor;
        csVec3f AngularVelocity;

    } aConfigurations[NumberOfConfigurations];
csTransform *pGeomObject;
private:
int IsActive;
CBehaviourList *pBehaviours;
public:
CRigidBody(csString *name, csTransform* geom,
float mass, csMatrix4f *InertTens);
CRigidBody() { };
~CRigidBody() { };
void InitializeBody(csString* name, csTransform* geom,
float mass, csMatrix4f *InertTens);
void InitializeForces();
void ComputeForces( int ConfigurationIndex );
void dummy(int x, float y);
void Integrate( float DeltaTime );
void ToggleConfigIndexes();
void SetGravityOn(){ GravityActive = 1;}
void SetGravityOff(){ GravityActive = 0;}
int  IsGravityActive(){return GravityActive;}
void Render();
int IsBodyActive() { return IsActive;};
void SetActive() { IsActive = 1;};
void SetInActive() { IsActive = 0;};
csVec3f* GetPosition(int ConfIndex)
{ return &(aConfigurations[ConfIndex].CMPosition);}
Quaternion* GetOrientation(int ConfIndex)
{ return &(aConfigurations[ConfIndex].Orientation);}
csRotation  GetInitialOrientation()
{ return InitialOrient; }
csVec3f* GetLinearVelocity(int ConfIndex)
{ return &(aConfigurations[ConfIndex].CMVelocity);}
csVec3f* GetAngularVelocity(int ConfIndex)
{ return &(aConfigurations[ConfIndex].AngularVelocity);}
};


inline csVec3f operator*(csMatrix4f &mat, csVec3f &vec)
{
//assumes that the last row and last column of mat
// are 0,0,0,1 and multiplies top 3x3 matrix with vec

csVec3f res, temp;
mat.getRow(0,temp);
res.set(0, vec.dot(temp));
mat.getRow(1,temp);
res.set(1, vec.dot(temp));
mat.getRow(2,temp);
```

```
res.set(2, vec.dot(temp));

return res;
}

#endif

/**********************************************************/
/* SimulationEnv.h - Include file for CSimEnvironment class */
/*                                                          */
/* Author   : M.N Alpdemir                                  */
/* Date     : 10/8/1999 (last date modified)                */
/**********************************************************/


#ifndef SimulationEnv_h
#define SimulationEnv_h

#include "RigidBody.h"
#include "Behaviours.h"
#include "stdafx.h"
#include <Cosmo3D/csArrays.h>
#include <Cosmo3D/csTime.h>
#include <Cosmo3D/csTimeSensor.h>
#include <Cosmo3D/csTransform.h>
#include <Cosmo3D/csBox.h>
#include <Cosmo3D/csShape.h>
#include <Cosmo3D/csSphere.h>
#include <Cosmo3D/csCylinder.h>
#include <Cosmo3D/csCone.h>
#include <Cosmo3D/csIndexSet.h>
#include <Optimizer/opTessParaSurfaceAction.h>
#include <Optimizer/opTessCurve3dAction.h>
#include <Optimizer/opTessCuboidAction.h>
#include <Optimizer/opTorus.h>
```

```
#include <Optimizer/opSphere.h>
#include <Optimizer/opCylinder.h>
#include <Optimizer/opCone.h>
#include <Optimizer/opRuled.h>
#include <Optimizer/opTriangle.h>
#include <Optimizer/opGeoBuilder.h>
#include "CommonTypeDefs.h"
#include "Collision.h"

class CAnimate;

class CSimEnvironment {
public:
//constructor
CSimEnvironment(CRITICAL_SECTION* pCriticalSection,
CAnimate* pAnim);
~CSimEnvironment();
//public data members
csPtrArray RgBodyList;
csPtrArray m_ClockList;
CBehaviourList  *m_pBehaviourList;
CRigidBody      *m_pActiveBody;
csStringDict *m_pObjNameTable;
csTimeSensor    *AnimationTime;
CListBox *m_pMessgBox;
CString m_FileName;
CFile *m_pFile;
CAnimate *m_pAnimator;
CRITICAL_SECTION* m_pSceneAccessSection;

protected:
opTessParaSurfaceAction *m_pParaTess;
opTessCuboidAction *m_pCuboTess;
opTessCurve3dAction *m_pCurveTess;
CCollisionManager       *m_pCollManager;
```

```
csPtrArray *m_pShapes;
csPtrArray *m_pGeometryArray;
csPtrArray *m_pRepArray;
csPtrArray m_AttachedViewPoints;
bool m_CollisionOn;
CollisionState m_CollisionState;
float m_tolerance;
float m_CollDeltaTime;
bool m_IsGravityOn;


public:
//public member functions
void StartAnimationTime();
csTime StopAnimationTime();
void RestartAnimationTime();
void DefineClock(csString clkName,
short loop, float timeInterv);
void AppendClock(csTimeSensor* pTS);
void Animate(float deltaTime);
void ShowMessage(CString messg);
csTransform* GetObjTransfNode(csString* objName);
CRigidBody* GetObjRigidBody(csString* objName);
CBehaviour* GetBehaviour(csString* behavName);
// Scan list of behaviours and link behaviours to their corresponding
// rigid bodies
void AttachBehavioursToBodies();
void ConstructCompositeBehavioursAfterLoading();
void ComposeBehaviours(csString objName,
csString bhvName, CStringList* pStrList);
void AppendBehaviour(CBehaviour* aBehaviour);
void SaveBehaviourList(CString fileName);
void LoadBehaviourList(CString fileName);
// create a common time sensor and set local time sensor
// of all thebehaviours to that sensor to synchronize behaviours
```

```
void SynchronizeBehaviours(CStringList*pNameList,
csString clockName);
inline double GetTimeDbl()
{ return (AnimationTime->getTime()).get();};
inline csTime GetTime()
{ return AnimationTime->getTime();};
    inline float GetTimeFraction()
{ return AnimationTime->getFractionChanged();};
void GetObjectShapeNodes(csTransform* pXf);
void GetTesselatedShapes(csShape* pShape);
bool GetTriangleArray(csTriStripSet* pGeometry,
csPtrArray* pTriangleArray);
void SetCollisionOn() { m_CollisionOn = true ;}
void SetCollisionOff() { m_CollisionOn = false; }
void SetGravityOn();
void SetGravityOff();
bool IsCollisionOn() { return m_CollisionOn; }
bool PrepareObjForCollision(csString objName);
bool CancelObjCollisionProperty(csString objName);
void CheckForCollisions();
void CheckForWallCollisions();
void HandleCollision();
void FireCollisionEvent(csString obj1, csString obj2);
void DefineRoomBoundaries(float x, float y, float z)
{ m_pCollManager->SetWorldSize(x, y, z);}

};

#endif

/********************************************************/
/* ViewManager.h - Include file for CVIewManager class    */
/*                                                      */
/* Author   : M.N Alpdemir                              */
/* Date     : 8/8/1999 (last date modified)             */
```

```
/*******************************************************/



#ifndef ViewManager_h
#define ViewManager_h

#include <Cosmo3D/csPerspCamera.h>
#include <Cosmo3D/csTransform.h>
#include <Cosmo3D/csArrays.h>
#include <Cosmo3D/csBasic.h>
#include <Cosmo3D/csLinMath.h>
#include "SimulationEnv.h"

typedef struct {
csPerspCamera *pCamera;
csTransform *pAttachedNode;
csVec3f relativePos;
csRotation relativeOrnt;
bool is_Current;
bool is_Attached;
} ViewPoint;

class CViewManager {

public:
CSimEnvironment *m_pEnvManager;

protected:
csPtrArray m_ViewPoints;
//csPtrArray m_AttachedViewPoints;

public:
void DefineCamera(csString camName,
csRotation ornt, csVec3f pos);
ViewPoint* FindCamera(csString camName);
ViewPoint* AttachCameraToObject(csString camName,
csString objName, csRotation orient, csVec3f relativePos);
void UpdateAttachedCameras();
//void setAttachedCamera(csPerspCamera* pCam);

CViewManager(CSimEnvironment* pEnvMan);
~CViewManager() { }
};

#endif
```

Appendix D

Class Definitions for Virtual Lab and MISAF

D.1 Class Definitions for Virtual Lab Application

```
/*******************************************************/
/* Items.h - Include file for Compartment,Room,Door,    */
/*           BuildItem, Corridor,ItemSize classes       */
/* Author   : M.N Alpdemir                              */
/* Date     : 23/11/1999 (date last modified)           */
/*******************************************************/

#ifndef Items_h
#define Items_h

#include "Stdafx.h"
#include <afxtempl.h>
#include "GlobalDefs.h"

class IAnimate;
class MapItem;

class ItemSize {
```

```
public:
float width;
float length;
float height;
public:
ItemSize(float w, float l, float h);
ItemSize() { }
~ItemSize() { }

ItemSize& operator = (ItemSize aSize);
};

inline ItemSize operator + (ItemSize s1, ItemSize s2)
{
ItemSize size;
size.width  = s1.width  + s2.width;
size.length = s1.length + s2.length;
size.height = s1.height + s2.height;

return size;
}

inline ItemSize operator - (ItemSize s1, ItemSize s2)
{
ItemSize size;
size.width  = s1.width  - s2.width;
size.length = s1.length - s2.length;
size.height = s1.height - s2.height;

return size;
}

class MapItem;

class BuildItem : public CObject {
```

```
protected:
CString m_Name;
ItemSize m_Size;
RGBColor m_Color;
CPoint m_Center;
CString m_TextureFileName;
IAnimate* m_pAniComp;

public:

BuildItem(CString name, ItemSize sz, RGBColor col,
         CString texFName);
CString  GetName()  { return m_Name;}
ItemSize  GetSize()  { return m_Size;}
RGBColor  GetColor() { return m_Color;}
void  SetColor(RGBColor col)
{ m_Color.R = col.R;m_Color.G = col.G;
  m_Color.B = col.B; }
void  SetName(CString nm){ m_Name = nm; }
void  SetVisibilityInterface(IAnimate* pVisInt)
{ m_pAniComp = pVisInt; }
virtual void Draw3D(MapItem* pLoc) { }
~BuildItem() { }

};
class Compartment;
class Door : public BuildItem {
protected:
Compartment* m_Connected1;
Compartment* m_Connected2;
BOOL m_IsLocked;
BOOL m_IsOpen;

public:
Door(Compartment* comp1, Compartment* comp2, CString name,
    ItemSize sz, RGBColor col, CString texFName);
~Door() { }

void Draw3D(MapItem* pLoc);
void Open()  { m_IsOpen = TRUE; }
void Close() { m_IsOpen = FALSE; }
void Lock()  { m_IsLocked = TRUE; }
BOOL IsOpen() { return m_IsOpen; }
BOOL IsLocked(){ return m_IsLocked; }
Compartment* GetConnectsFrom() { return m_Connected1;}
Compartment* GetConnectsTo() { return m_Connected2;}

Door& operator = (Door aDoor);
//BOOL operator < (Door aDoor);
BOOL operator == (Door aDoor);

};

typedef CTypedPtrList<CObList, Door*> DoorListType;

class Compartment : public BuildItem {
protected:
DoorListType m_DoorList;
public:
Compartment(CString name, ItemSize sz, RGBColor col,
            CString texFName):
BuildItem(name, sz, col, texFName)
{ }
void AddDoor(Door* pDoor);
void RemoveDoor(Door* pDoor) { }
~Compartment() { }
};

class Room : public Compartment {
protected:
```

```
int m_RoomNo;

public:
Room(int no, CString name, ItemSize sz,
     RGBColor col, CString texFName);
void Draw3D(MapItem* pLoc);
CString DrawDooredWall(CString wall, Door* pDoor);
CString     DrawPlainWall(CString wallDir);
~Room() { }
};

class Corridor : public Compartment {
protected:
int m_CorridorNo;

public:
Corridor(int no, CString name, ItemSize sz,
         RGBColor col, CString texFName);
void Draw3D(MapItem* pLoc);
CString DrawWall(CString wallDir);
CString DrawWallWithDoor(CString wallDir);
~Corridor() { }
};
#endif

/*********************************************************/
/* Factories.h - Include file for LabFactory class      */
/*                                                       */
/* Author   : M.N Alpdemir                               */
/* Date     : 23/11/1999 (date last modified)            */
/*********************************************************/

#include "Items.h"

class LabFactory {
```

```
public:
//Public Methods:
Room* MakeRoom(int rno, CString nm, ItemSize sz,
    RGBColor cl, CString texFName)
{return new Room(rno, nm, sz, cl, texFName);}
Corridor* MakeCorridor(int cno, CString nm, ItemSize sz,
            RGBColor cl, CString texFName)
{return new Corridor(cno, nm, sz, cl, texFName);}
Door* MakeDoor(Compartment* cmp1, Compartment* cmp2,
            CString nm, ItemSize sz, RGBColor cl,
CString texFName)
{return new Door(cmp1,cmp2, nm, sz, cl, texFName);}

LabFactory() { }
~LabFactory() { }
};

#endif

/*******************************************************/
/* LabMap.h - Include file for MapItem class          */
/*                                                     */
/* Author   : M.N Alpdemir                             */
/* Date     : 19/11/1999 (date last modified)          */
/*******************************************************/

#ifndef LabMap_h
#define LabMap_h

#include "Items.h"
#include "GlobalDefs.h"

typedef struct {
Compartment* ConnectedTo;
```

```
Door* pAccessDoor;
Directions AccessDirection;
} MapItemType;

class MapItem : public CObject{
protected:

Compartment* m_Compartment;
MapItemType m_Connections[4];

public:
void SetConnection(Directions dir,
                Compartment* conItem, Door* aDoor);
MapItemType GetConnection(Directions dir)
                { return m_Connections[dir];}
CString GetCompartmentName()
                { return m_Compartment->GetName(); }
Compartment* GetCompartment(){ return m_Compartment; }
void SetCompartment(Compartment* pComp)
                { m_Compartment = pComp; }
MapItem& operator = (MapItem item);
//BOOL operator < (MapItem item);
BOOL operator == (MapItem item);
MapItem(Compartment* item);
MapItem();
~MapItem() { }
};

/*********************************************************/
/* Laboratory.h - Include file for LayOut and           */
/*                Laboratory class                       */
/*                                                       */
/* Author   : M.N Alpdemir                              */
/* Date     : 19/11/1999 (date last modified)          */
/*********************************************************/
```

```
#ifndef Laboratory_h
#define Laboratory_h

#include <afxtempl.h>
#include "Items.h"
#include "LabMap.h"
#include "GlobalDefs.h"
class LabFactory;
class IAnimate;

typedef CTypedPtrList<CObList, MapItem*> LabMapType;

class LayOut {

LabMapType m_LabMap;
Compartment* m_CurrentLocation;

public:
void AddLocation(MapItem* item);
Compartment* GetCurrentLocation()
                { return m_CurrentLocation; }
void AddLocationToMap(Compartment* pItem);
void RemoveLocationFromMap(Compartment* pItem);
Compartment* FindLocation(CString compName);
void Draw();
};

class Laboratory : public CObject {
public:
LayOut* m_LabLayout;
LabFactory* m_LabFactory;
IAnimate*   m_pVRInterface;

public:
```

```
Compartment* GetCompartmentFromName(CString compName);
void DrawLocation(MapItem* pLoc);
void SetVRInterface(IAnimate* pAnimator)
                {m_pVRInterface = pAnimator;}
void Initialize();
Laboratory(LabFactory* factory);
Laboratory() { }
~Laboratory();
};
#endif
```

D.2 Class Definitions for MInimal Simulation Animation Framework

```
/*******************************************************/
/* MISAFFramework.odl - COM Interface definition file   */
/*               for MISAF Framework                     */
/*                                                       */
/* Author   : M.N Alpdemir                               */
/* Date     : 9/01/2000                                  */
/*******************************************************/
#include <olectl.h>
#include <idispids.h>

[ uuid(E5ED2E73-2917-11D4-B8F9-444553540000), version(1.0),
  helpfile("MISAFFramework.hlp"),
  helpstring("MISAFFramework ActiveX Control module"),
  control ]
library MISAFFRAMEWORKLib
{
importlib(STDOLE_TLB);
importlib(STDTYPE_TLB);

//  Primary dispatch interface for CMISAFFrameworkCtrl
```

```
[ uuid(E5ED2E74-2917-11D4-B8F9-444553540000),
  helpstring("Dispatch interface for MISAFFramework Control"), hidden ]
dispinterface _DMISAFFramework
{
methods:
// NOTE - ClassWizard will maintain method information here.
//    Use extreme caution when editing this section.
//{{AFX_ODL_METHOD(CMISAFFrameworkCtrl)
[id(1)] short GetComponentCount();
[id(2)] VARIANT GetSysRegisteredComponents();
[id(3)] VARIANT GetComponentListInFramework();
[id(4)] void InitializeComponentEnv(BSTR ComponentName,
        IDispatch* pAniComp);
[id(5)] void LoadObjGeometricModel(BSTR ComponentName,
        float xPos, float yPos, float zPos);
[id(6)] void StartSimulationComponent(BSTR ComponentName);
[id(7)] void StopSimulationComponent(BSTR ComponentName);
[id(8)] short ComponentDoCommand(BSTR ComponentName,
        BSTR cmdStr, long paramNum, BSTR paramTypeStr,
BSTR paramValueStr);
[id(9)] boolean TestMISAContract(BSTR ComponentName,
        IDispatch* pAniComp);
//}}AFX_ODL_METHOD

[id(DISPID_ABOUTBOX)] void AboutBox();
};

//  Class information for CMISAFFrameworkCtrl

[ uuid(E5ED2E76-2917-11D4-B8F9-444553540000),
  helpstring("MISAFFramework Control"), control ]
coclass MISAFFramework
{
[default] dispinterface _DMISAFFramework;
[default, source] dispinterface _DMISAFFrameworkEvents;
};
};

/*****************************************************/
/* CtrlWrap.h - Include file for CCtrlWrapper and  file */
/*                CEventInfo classes                  */
/*                                                    */
/* Author   : M.N Alpdemir                            */
/* Date     : 10/01/2000                              */
/*****************************************************/
class CEventInfo : public CObject
{
public:
CEventInfo(UINT nParams, BSTR* pNames, ELEMDESC typeRet,
          ELEMDESC* typesParam,
      INVOKEKIND invkind = INVOKE_FUNC);
~CEventInfo();

VARTYPE m_vtFuncType;
CString m_strName;
INVOKEKIND m_invkind;

UINT m_nParams;
CString* m_pParamNames;
VARTYPE* m_pParamTypes;
};

class CCtrlWrapper : public CWnd
{
// Construction
public:
CCtrlWrapper();

// Attributes
public:
```

```
IDispatch FAR*  m_pComponent;
// Operations
public:
CEventInfo* GetInfo(DISPID dispid);

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CCtrlWrapper)
//}}AFX_VIRTUAL

// Implementation
public:
virtual ~CCtrlWrapper();
BOOL GetBrowseInfo();
void LoadEventTypes(LPTYPEINFO lpTypeInfo);
void LoadPropMethTypes(LPTYPEINFO lpTypeInfo);
void SetDispatchInterface(IDispatch FAR* pDisp)
{ m_pComponent = pDisp; }
void ShowErrorMessage(HRESULT hres);

// Methods in the Lab-Instrument Interface Interaction Contract
BOOL InitializeComponentEnv(LPDISPATCH pAnimDisp, bool IsTest);
BOOL LoadVirtInstrumentModel(float x, float y,
     float z, bool IsTest);
BOOL StartIndependentSimulation(bool IsTest);
BOOL StartSimulation(bool IsTest);
BOOL StopSimulation(bool IsTest);
BOOL DoCommand(BSTR cmdStr, short paramNum,
     BSTR paramTypeStr, BSTR paramValueStr, bool IsTest);

BOOL TestContract(LPDISPATCH pAnim);

void LoadControl(CArchive& ar);
void WriteControl(CArchive& ar);
void Serialize(CArchive& ar);

UINT m_nEvents;
CMapWordToOb m_EventInfoMap;
CObList m_PropertyList;
CObList m_MethodList;
};

/*********************************************************/
/* FlexiHotSwitch.h - Include file for SwitchPosition   */
/*                 and FlexiHotSwitch classes           */
/*                                                       */
/* Author   : M.N Alpdemir                               */
/* Date     : 13/01/2000                                 */
/*********************************************************/

#ifndef FlexiHotSwitch_h
#define FlexiHotSwitch_h

#include "stdafx.h"
#include <afxtempl.h>
class CCtrlWrapper;

class SwitchPosition : public CObject{
public:
int m_CtrlId;
CString m_CompName;
CString m_CLSID;
CCtrlWrapper* m_pCtrlWrap;

public:

SwitchPosition& operator = (SwitchPosition aSwPos);
BOOL operator == (SwitchPosition aSwPos);
void SetControlWrapper(CCtrlWrapper* pCtrlWrap)
```

```
{ m_pCtrlWrap = pCtrlWrap; }
SwitchPosition();
SwitchPosition(int id, CString Name, CString CLSID);
~SwitchPosition() { }
};

typedef CTypedPtrList<CObList, SwitchPosition*> ComponentListType;

class FlexiHotSwitch {

protected:
short m_MaxSlotNum;
POSITION m_CurrentPos;
ComponentListType m_ComponentList;
SwitchPosition* m_pActiveComponent;

public:
void AddComponent(int CtlId, CString CompName,
                         CString aCLSID);
void AddComponent(SwitchPosition* pSwPos);
int GetComponentCount()
            {return m_ComponentList.GetCount(); }
SwitchPosition* GetActiveComponent()
            { return m_pActiveComponent; }
void SwitchToComponent(IDispatch FAR* pNewActiveComp);
void DeleteComponent(IDispatch FAR* pComp);
void SwitchToComponent(CString NewActiveCompName);
void DeleteComponent(CString CompName);
LPDISPATCH  FindComponent(CString compName);
CtrlWrapper* GetComponentWrapper(CString CompName);

FlexiHotSwitch();
~FlexiHotSwitch();
};
#endif
```

Appendix E

Published Papers

Four papers have been published during this research work [Alp99, Alp98, Kuc98, Hab98]. Their copies are provided in the following pages.