- RASP -
Robotics and Animation Simulation Platform

by
Gene S. Lee

Technical Report 94-25
October 1994

- RASP -

Robotics and Animation Simulation Platform

by

Gene S. Lee

S.B. (Computer Science and Engineering)

Massachusettes Institute of Technology, 1989

A Thesis Submitted in Partial Fulfillment of

the Requirements for the Degree of

Masters of Science

in the Faculty of Graduate Studies

Department of Computer Science

We accept this thesis as conforming

to the required standard

.....

.....

THE UNIVERSITY OF BRITISH COLUMBIA

January, 1994

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1Z1

Date:

Abstract

A basic problem associated with the development of new techniques in the fields of computer animation, robotics, and simulation is that many researchers utilize dissimilar constructs to represent common structures. Attempts to combine various models into one coherent system can often be painstakingly difficult. This forces the users to expend valuable time re-inventing previously written code.

To resolve this problem, this thesis presents the RASP (Robotic and Animation Simulation Platform) toolkit - an extensible collection of primitives, functions, and essential abstractions for the creation of reusable time-varying simulations. Based on object-oriented principles, modern patterns of communications, and various simulation techniques, the toolkit defines a common architecture and set of conventions for researchers to follow when developing simulations. Through these building blocks, users will be able to borrow, without considerable need for modifications, code segments and tools from previously developed RASP projects.

The RASP toolkit is highlighted by the following set of features: (a) *IMVCD* - a framework for the construction of time-varying systems; (b) *Connection Paradigm* - a "port"-based approach to data communication; (c) *Hierarchical Temporal Modeling* - a top down approach to temporal management based upon multiple world views and first-class temporal primitives; and (d) *Hybrid Object Construction* - a clear design for the development and visualization of complex objects.

Table of Contents

List of Tables

List of Figures

ACKNOWLEDGMENTS

My foremost thanks go to my supervisors, Dr. David Forsey and Dr. Dinesh Pai. If it were not for their inspiration, support, and guidance, this project would never have been accomplished. I appreciate the freedom they granted me to choose my own direction, to adjust continually my time lines, and to take leave during the summer.

I would also like to thank the following list of people. Their valuable assistance and contribution to the development of this thesis is greatly appreciated.

...My sister Gia Lee for spending her valuable time to proofread.
...Dr. Jack Snoeyink for supporting my research and for discussions on graduate school.
...Larry Palazzi for being my student reader (although he was no longer a student) and for his help in the layout and formatting of this thesis.
...Pierre Poulin for xfig-diagrams, valuable discussions, and being a good friend.
...My friends at Apple Computer, Inc. for an extremely educational and exciting summer.
...Grace Wolkosky for fighting to extend the deadline of my thesis until completion.
...Mandeep Dhami and Sameer Mulye for valuable discussions on system protocols, architectures, and terminology.
...Donald Acton for discussions on co-routines and RAVEN.
...Bill Gates for potato chips, pizza money, and 7-11 Big Gulps.
...Larissa McWhinney and Olivier Tardiff for being good friends, providing help in assembling my thoughts and supplying good quotation books.

Finally, I would like to thank my family. Their love and support has inspired me to excel and dream.

To

my parents,

sister,

and friends.

CHAPTER 1

INTRODUCTION

'Where shall I begin, please your Majesty?' he asked.
'Begin at the beginning' the King said, gravely,
'and go on till you come to the end: then stop.'
- Lewis Carroll, Alice's Adventures in Wonderland, Ch. 11

Computer simulation serves to reproduce or represent test conditions likely to occur in real situations. In addition to supplying behavioral patterns, collections of statistics, and good estimates, simulations prove important industrial tools. Due to their off-line nature[1] and their ability to predict the behavior of the systems they emulate, simulations are used to reduce the costs, hazards, and design schedules of real world applications. In certain instances, the value of simulations actually exceeds that of natural observations, providing otherwise imperceptible information such as the internal stresses within materials.

Productivity and the power of simulation can be greatly enhanced by reusing the components of various simulations. However, the melding of two or more simulations into one coherent application proves quite difficult. For example, simulations based upon alternative methodologies of specifying changes to a system can be arduous to combine. Incompatible designs force users to redesign the plans of their original simulations before the simulations can be combined. Apart from prolonging the length of time it takes to construct a simulation, the redesign process can introduce errors not established in the original designs.

There are three principal features in an ideal reusable simulation environment. First, it should have the ability to incorporate algorithms, elements, and interfaces from other simulation designs. Users should not need to design new models from scratch every time a new objective is encountered. Second, it should be quickly modifiable. Extensive re-modeling to provide elementary changes defeats the purpose of reusability. Third, the domain of the simulation

[1]Off-line development does not require the actual objects being studied to be used during the simulation process.

should not be limited in scope. This does not imply that every possible feature of every simulation should be enforced in one design. Rather, it should be possible to modify the simulation to imitate a large set of multifarious functions and behaviors.

Most attempts to create reusable simulation environments fail to satisfy one or more of the above criteria. The majority of approaches to simulation are too specialized and often too complicated to extend. This leads to tools unusable for future research. For graphical systems, this can mean unavoidable revisions of the entire modeling environment. Reusable simulations should not constrain the user's ability to incorporate other models into already existing ones.

Attempting to create a completely reusable set of tools for a wide variety of simulations is an enormous task. An all-purpose toolkit would require the development of many new ideas and concepts concerning the theory of modeling and simulation. A smaller task, yet still of great size, is the development of a reusable library for the creation of computer animations and robotics applications. The necessity for such a set of building blocks is sensed in both the academic and production community. A common development base would encourage researchers from a wide variety of disciplines to share their workspace, thereby, enhancing each other's facilities.

1.1 Characteristics of Simulation Tools

The task of creating reusable tools for the development of computer simulations is not an unfamiliar endeavor. The literature abounds with research papers written by simulationists delineating the features of numerous simulation libraries, languages, and systems. The major trait that distinguishes one approach from another is the extent to which each tool imposes on the design and structure of simulations. Basic tools, such as simulation libraries, provide users with collections of components and functions. Free to use these tools in the manner that befits them the best, users are not restricted to construct simulations that adhere to a general design. Although this freedom promotes the general use of these types of tools, it severely limits the ability of users to reuse the components from multiple simulations. Models which adhere to differing modeling conventions are often difficult to interchange and redefine. Intermediate tools, such as simulation languages, provide users with sets of high-level expressions.

Users assemble expressions into meaningful phrases to define the components and dynamic interactions of a simulation. Language grammars define the rules for simulation construction. However, rules are usually limited to tools usage, not simulation design. Modeling precepts are not supplied that define a general framework for simulation development. Advanced tools, such as simulation systems, provide users with a complete simulation modeling environment. Users are supplied with sets of components, functions, high-level expressions, and simulation design rules that conform to a general plan. This plan promotes the creation of simulations with high reuse potential. Although these types of tools are powerful, their strengths contribute to their weaknesses. The modeling environments formed by these tools are usually difficult to extend and modify. In some cases, the architecture of the modeling environment is unalterable. This limitation restricts the number and variety of simulations which can be constructed with these types of tools.

Simulation tools can also be distinguished by three additional characteristic traits. First, they can be characterized by the "projected range" of applications they wish to accommodate. Some tools attempt to serve a wide variety of applications while others cater to the needs of a select few. Second, the tools can be described by the "level of abstraction" of its components. Advanced tools usually enable users to construct from a high-level of abstraction while intermediate and basic tools force users to construct from lower levels of abstraction. Third, simulation tools can be characterized by their "design focus". Distinct sets of tools that attempt to solve similar problems do not always provide users with exactly the same collections of components and abstractions. Some tools will focus their designs on certain elements of a simulation while others will focus their designs on alternative elements. A tool's focus is dependent upon the requirements of the audience it serves.

The composition of a set of simulation tools is directly affected by the projected range of applications it attempts to model. Tools intended to model a gamut of applications are generally composed of sets of generic components and functions. They provide users with a minimal amount of support to create a wide variety of scenarios. Components and functions are developed to meet the demands of a broad range of users' objectives. Tools intended to model a small range of applications provide users with specialized components. For instance, tools

generated from the computer animation community are typically very specific in nature. Most collections are usually utilized to attack only a subset of problems, such as *object modeling*, *motion specification*, and *image synthesis*, encountered in the field of graphics.

The *model specification* technique, such as entity construction methods[35, 83, 66], environmental planning procedures[55, 17, 3], and protocols of temporal control[70, 27, 6, 40], employed by a set of simulation tools dictates the level of abstraction and define the methodology used by users to describe their simulations. These techniques are classified as *programming*-based or *scripting*-based. Programming-based techniques provide users with collections of data structures and data types and an enormous amount of control. However, it requires users to develop simulations from an extremely low-level of abstraction and to possess proficient knowledge of programming techniques to accurately express their designs. Scripting-based techniques provide users with special expressions, grammars, and interpreters. Adhering to the grammars, users organize expressions into meaningful phrases for interpreters to transform into low-level descriptions. Although these techniques empower users to define simulations from a high level of abstraction, they limit users from making detailed modifications to their simulations.

The design focus of a set of simulation tools is governed by the needs of its group of users. Distinct groups, possessing dissimilar design goals and attempting to solve analogous problems, generally emphasize the advancement of different aspects of the simulation modeling process. For instance, the drive to alter the attributes of geometrical figures over time has introduced a variety of animation and robotics systems. Animation systems tend to stress the development of constructs for user-scripted changes while robotics systems stress the importance of defining relationships between physical bodies and applying control algorithms to them. In turn, neither of these two approaches address the creation of general *transitional structures*, the foundation of many simulation languages, that serve to define, organize, and execute the passage of model variables from state to state.

1.2 The Thesis

In the past two decades, the field of computer science has witnessed an enormous growth in the variety of projects studied by researchers. A general problem associated with an outgrowth of

new ideas is the difficulty of combining the benefits of several research fields into one coherent design. For example, recent trends in robotics research[36] has seen the need to visualize and plan the actions of robotic elements in a complex environment. Although computer animation systems have addressed the problem of placing automated figures in configurable workspaces for some time, few constructs from the field of computer animation have been incorporated into robotics research. The lack of a prevailing set of tools to create time-varying simulations limits users from borrowing components and structures from simulations outside their domain. However, this problem is not limited to users across multiple disciplines. Great hardships are even experienced by users who attempt to merge the designs of various simulations within their respective fields. Simulations which share similar goals are not always easily united. It is often difficult to reconcile the differences between equivalent simulation components represented by dissimilar structures.

The failure of simulation tools to gain widespread acceptance is attributed to the absence of firm theories and general principles concerning software reusability. The lack of a clear understanding of reusability has festered a general mood of apprehension and dread toward the usage of reusable tools[5, 38]. The lack of firm tradeoffs between generality and specialization exacerbates users' tendencies to neglect the usage of simulation tools. Users experience frustration when their tools are too specific or too vague. Explicit guidelines and definite structures limit a tool's total applicability while obscure rules of usage and ambiguous structures limit a tool's purpose.

This thesis aims to present the computer graphics, robotics, and simulation community with a set of tools for the development of time-varying simulations. The research presented here synthesizes knowledge from each of these fields to determine the appropriate abstractions and integrates the results with existing reusable technologies. Through this careful examination, a collection of building blocks and abstractions are constructed to provide programmers, animators, and researchers with a foundation for application development. The toolkit's design especially attempts to provide users with a non-constraining environment that readily supports their particular designs and enables them to borrow ideas and segments of code from previously developed applications. Extensible data structures, modern patterns of communications, and

variable modes of control help to facilitate the design of new concepts and algorithms.

1.3 Thesis Contributions

This thesis presents RASP (Robotics and Animation Simulation Platform), an object-oriented collection of primitives and abstractions for the development of time-varying simulations. Providing research scientists with a common platform to construct, manipulate, and visualize their temporally-based applications, RASP's reusability results from its employment of object-oriented strategies, hierarchical schemes, and extensible designs. One may envisage RASP as providing the role UNIX[2] has with respect to general application programming. While UNIX provides users with a consistent interface to peripheral devices, file systems, and multi-language support, RASP provides users with a clear framework for the development and visualization of complex objects, an extensible approach to simulation modeling, and a simple scripting convention to manipulate temporal data. The RASP toolkit is highlighted by the following set of features.

Uniform Terminology

The RASP toolkit's terminology derives from the literature related to the development of a toolkit for time-varying simulations. These terms reflect the various types of simulation techniques from different fields which inform the overall design of the RASP toolkit. The creation of a uniform set of tools enhances both RASP's flexibility and reusability in a large variety of modeling simulations.

I-M-V-C-D Framework

Serving as a framework for time-varying systems, IMVCD (Informer-Model-Viewer-Controller-Delegator) informally defines the divisions and rules of interaction between the various elements of a RASP constructed application. Influenced by the concepts from the MVC user-interface modeling paradigm[45], this object-oriented framework provides users with a relatively simple modeling pattern for the development of reusable applications. Simulations devised from this

[2]UNIX is a registered trademark of AT&T

reference model enhance their ability to be examined, understood, and modified by future users. It strengthens their potential for reusability.

Connection Paradigm

The means of data communication within a simulation system affects the way information is transferred between the individual elements of a simulation. Patterns of communication that embed their rules of interaction in solitary structures promote unnecessarily the design of overly complex models. Concentrated patterns enforce interacting components to accommodate additional constructs and plans towards the maintenance and formation of data links. A superior plan apportions the duties and responsibilities across a number of modeling elements. The RASP toolkit employs a distributed[3] pattern of communication based on the connection paradigm[55]. Using unidirectional data ports and active data links, informational pathways are maintained and constructed by elements not directly influenced by the data transferal process. External elements ensure the transportation of information between the ports of compatible models. Models are constructed to react to events raised on their ports, not to establish data links. They are never involved in the data transferal process and are always oblivious to the identity of the partners to which they exchange information. Apart from reducing the complexity of models, this design enhances the development of reusable components and strategies.

Hierarchical Temporal Modeling

The behavior of a time-varying simulation is determined by the nature of its user-defined state changes and the techniques employed to regulate the progression of time. A simulation can not express or imitate behaviors that are not explicitly or implicitly defined in its model specifications. Hence, a simulation's validity is compromised if it is not possible for users to specify specific temporally-dependent state changes. In RASP, simulation developers express time-based state changes with the assistance of a collection of temporal modeling tools. Conforming to the prescribed rules of the connection paradigm, these temporal primitives enable users to

[3]In this context, the keyword "distributed" is not be identified with distributed system or parallel architectures.

specify the starting times, durations, and granularities of state altering actions. A clear relationship between time and state is generated from the natural hierarchical arrangement of these temporal building blocks. In addition, the temporal primitives offer users the ability to incorporate multiple temporal progression techniques or world views into one simulation model. This *multiple interface* approach towards simulation modeling permits users to select the technique that provides the most flexibility to their modeling needs.

Hybrid Model Construction

In simulation modeling, the intrinsic design of the elements in a system imposes a strict set of constraints on the transmission of information "to" and "from" the models they describe. An model's internal architecture defines how data is stored and accessed by other system components. In RASP, an model's internal organization is governed by its "feature" ports. Adhering to the connection paradigm, these special ports encourage the delegation of model responsibilities and the hierarchical organization of information. Deemed as a hybrid model, this approach fosters the construction of odels which responds to messages and an architecture which supports rendering operations.

1.4 Organization of the Thesis

This thesis consists of four parts. The first part, chapters 1 to 5, previews the motivation towards and background concepts of the creation of a toolkit for the development of time-varying simulations. Discussion encompasses previous work in computer graphics toolkit design, computer animation system development, simulation designs, and temporal manipulation techniques. Part two, chapters 6 to 10, defines the design features of the RASP toolkit. While chapters 6 and 8 present design goals, the IMVCD framework, the connection paradigm, and the multiple interface approach to discrete-event modeling, chapters 9 and 10 elucidate the relationship between time and state, and outline the hybrid model construction methodology. The third part, chapters 11, discusses the details of the toolkit's implementation. The final part, comprised of the conclusion and appendices, provides an analysis of the toolkit's design and implementation, suggests possible future modifications and enhancements.

CHAPTER 2

GRAPHICS TOOLKITS

Give us the tools, and we will finish the job.
- Sir Winston Leonard Spencer Churchill, 9 Feb 1941

Computer graphics toolkits provide users with a basic set of tools and definitions for the creation and manipulation of three-dimensional geometric models. In addition, they provide users with the ability to share resources, to build device-independent interfaces to multiple platforms, and to export standard database metafiles.[1]

The first standard in three-dimensional graphics was called *3D Core Graphics System*[28]. Intended as a baseline specification in computer graphics, this standard lead to the development of several graphics packages, such as GKS-3D[35] (the Graphical Kernel System) and PHIGS+[89] (Programmer's Hierarchical Interactive Graphics System). Each standard defined a set of methods and structures for the modeling and displaying of three-dimensional data. The primary emphasis of these two graphics packages was the construction of geometric models. Utilizing a display list architecture, users associated physical and user-defined attributes with geometric primitives to define and render computer-generated images. Apart from a minimalistic set of dynamic features, such as scaling, rotation, and translation, there were no mechanisms for the specification of general dynamic movements. Graphics researchers were forced to define their own mechanisms for the specification of physical movement.

The introduction of faster hardware, improved programming models and languages, and the demand for higher levels of abstraction spurred the construction of larger toolkits with greater capabilities. Enhanced features included paradigms for direct manipulation, object-based construction and modeling, and improved modularity of toolkit components.

[1]Metafiles are data files containing collections of low-level device-independent descriptions. Although they are usually large in size, they provide users with a standard method to describe a picture or scene.

The remainder of this chapter describes and analyzes several of the latest developments in computer graphics toolkit design. Readers already familiar with these packages may jump to this chapter's summary without loss of continuity.

2.1 Survey of Graphics Toolkits

2.1.1 DORE

DORE[46], developed by Kubota Pacific Computer, is a semi-object-oriented photo-realistic three-dimensional graphics library. It supports various geometric primitives, surface property tools, scene manipulation elements, numerous rendering representations, and a wide variety of graphics database editing functions.

Written in C and FORTRAN, DORE places an object-oriented framework on top of the *display list* approach of traditional 3D graphics systems. Sequences of drawing commands, which are sequentially parsed to alter the state of the rendering environment, are encapsulated into objects. Although these objects can not be treated as first-class items, this representation scheme significantly improves the display list approach to computer graphics development. DORE supports three types of objects:

- **primitives**: These objects represent the basic set of geometric shapes supported by DORE. Users may define additional primitives if the pre-defined basic set does not match their geometrical or behavioral needs. All new primitives must define private variables, identification and initialization routines, and a basic set of editing and querying operations. In addition, since all DORE renderers do not render the same set of primitives, it is the responsibility of the user to empower the new primitive to decompose itself into alternate representations.

- **primitive attributes**: The appearance of geometric primitives is effected by these types of objects. They effect the display representation, material properties, and shading style of primitive objects.

- **geometric transformations**: These objects affect the shape and position of 3D shapes. Effects such as scale, translate, rotate, and shear are defined by these objects.

DORE utilizes reference counts to administer the process of garage collection. Each object has a reference count that is incremented by one every time it is added to an organizational object. Similarly, object reference counts are decreased by one as they are removed from organizational objects. Once an object's reference count reaches zero, it is removed from the system. The memory deallocation[2] process is overridden by locking objects. This forces the system to retain the locked object in main memory until its reference equals zero whereupon it is unlocked and deallocated.

In DORE, a group is an ordered listed of object handles. Groups contain references to primitive objects, primitive attribute objects, geometric transformation objects, labels, and other groups. The position of elements within a group is important. Only elements with higher precedence (lower list index) in the ordered list effect those in the remainder of the list. An object's appearance is not affected by attribute references possessing lower precedence.

A group references another group in one of two manners: as a subroutine or as a macro. In the subroutine case, the referenced group's attributes can not affect the appearance of the parent group. In the macro (in-line) case, a child group's attributes produces display changes in the objects of its parent group. In-line groups are often utilized to rapidly change dynamic attribute values.

Callback objects invert execution of the DORE database. When activated, these objects pass user specific data to user-written functions. Callback objects initiate three special functions during their active lifetimes: force re-execution of the current database method, terminate execution of the database traversal process, or prune the current execution path.

[2]Deallocation is the opposite operation to allocation. Allocated memory is freed by deallocation procedures.

2.1.2 INVENTOR

INVENTOR[83] is an object-oriented toolkit for 3D graphics applications. Heavily relying on SGI's GL graphics library[3], this toolkit enables users to created interactive programs. Uncommon to previous 3D libraries, INVENTOR supports the direct manipulation (picking) of 3D objects and regards objects as geometric, physical entities.

The scene database is the foundation of the INVENTOR toolkit. The dynamic representation of scenes are stored as a composition of objects, called nodes, in a hierarchical graph structure. Each node in a graph represents a geometrical shape, physical property, database traversal behavior, or composite group. The group nodes define the framework and method of interpretation for each graph. Individual nodes are connected as they associate with groups. The type of group node defines how children are traversed and how properties are inherited. Some group nodes have the ability to cache the traversal state, while others dynamically prune the tree traversal path.

INVENTOR's database provides a set of basic actions that are applied to entire scenes or segments of scenes. Fundamental operations, such as rendering, picking, calculating bounding boxes, event handling, and scene storing, are defined as action objects. Encapsulating actions into objects enables users to define new database traversal tasks.

In addition to the basic set of actions, the toolkit supports sensors and callbacks. Sensors are special objects that enable users to build simple animations. They are utilized to detect changes in groups of nodes or to continuously trigger changes to the scene database. Callback objects are defined to invoke user-defined functions. These nodes enable users to create their own application-specific mechanisms.

2.1.3 CONDOR

CONDOR (Constraints Dynamics Objects and Relationships)[41], written by Micheal Kass of Apple Computer, Inc., is an interactive dataflow programming environment for computer graphics. Most properly viewed as a next-generation math compiler, it supports constraints,

[3]GL is a registered trademark of Silicon Graphics, Inc.

dynamics, and various other computational models. Every CONDOR dataflow element performs derivative evaluation and interval arithmetic. Utilizing an interactive graphics interface, users compose new functions by linking vector (or scalar) inputs to vector (or scalar) outputs. An efficient C++ "generating" compiler and a special set of optimization operations enhance system performance by producing efficient C++ code segments.

CONDOR relies on the dynamic composition of compiled functions to configure quickly complex systems. As users form links between data ports, the environment associates efficiently compiled modules together. Error-free functional units are generated by CONDOR's compiler. Written in Lisp, the compiler utilizes the Mathematica symbolic math package to generate streamlined C++ code. It is important to note that only C++ code is called during application run-time. The Lisp compiler and Mathematica package are utilized only to generate C++ code. Essentially, the CONDOR expression tree evaluation process consists of a series of calls to compiled functions.

2.1.4 GRAMS

GRAMS[18], developed by Parris Egbert of the University of Illinois, is an object-oriented system for 3D computer graphics applications. Using a multi-layer paradigm, this system separates the modeling and rendering aspects of traditional graphics systems into separate entities. This approach to graphical support allows users to define applications at high levels of abstraction. The extendibility of this model is attributed to its object-oriented design and structured scheme to the image synthesis process.

The three main components in GRAMS are the application, graphics, and rendering layers. Each layer is responsible for a separate phase of the image generation and application development process.

- **Application**: All user's applications reside in this layer. This layer separates the application architecture from the graphics sub-system. Application data may be stored and manipulated by each program in any form that is most convenient. At image generation time, vital rendering data, such as object coordinate transformations, materials, and geometries, are extracted from this level and passed to the graphics layer.

- **Graphics**: This layer performs as an intermediary between the Application and Rendering layers. It is responsible for transforming the high-level data objects from the application layer into a suitable format for the rendering layer. This process is handled by well-defined paths of communication and an internal translation mechanism.

- **Rendering**: The actual rendering process is performed at this level. Given information from the Graphics layer, this layer generates a static image. It is important to note that this layer defines the type of information that will be accepted from the Graphics layer. The format and quantity of information may vary from application to application.

GRAMS' greatest contribution to the design of an extensible graphic toolkit is the concept of independent construction. Allowing users to focus attention on separate aspects of the application design process enables them to build a variety of 3D programs. Useful ideas, such as the independent construction of renderers and geometric objects, are valuable concepts to the development of reusable components.

2.2 SUMMARY

This chapter has presented a brief summary of a variety of three-dimensional computer graphics toolkits. Each toolkit provides users with a set of constructs to associate information with geometric models and to manipulate their physical structures. Toolkits differ in the manner they store, interpret, and access a model's information. Some toolkits emphasize the interaction of the geometric models with their users and the simulation environments while others emphasize the interaction of the models with toolkit image renderers.

A careful review of computer graphics toolkits permits the creation of a general feature list.

- **scenario modeling tools**: Basic operations are provided to facilitate the development of the simulation environment. The global behavior and organization of models are regulated by these tools.

- **geometric primitives**: A common set of geometric objects, such as spheres, cylinders, cones, etc., is defined for user-convenience. Primitives serve as building blocks for physical design.

- **geometric composition**: The hierarchical or flat[4] construction of geometric primitives enhances users's abilities to build complex geometric entities. Complex geometries are formed using a variety of composition techniques.

- **transformations**: Associating transformation, such as rotation, translation, scaling, etc., with primitives enables users to alter the shape and position of their models, lights, and cameras. Transformations are linked with temporal information to define simple animations.

- **geometric attributes**: Attribute techniques allow users to fasten physical properties and application dependent information to geometric primitives and user-created objects.

- **camera primitives**: Camera types allow users to alter the viewing configuration of image without manipulating the attributes of an image renderer. Viewing parameters, such as field of view, aspect ratio, and point of view, are controlled by these tools.

- **illumination primitives**: A basic set of illumination devices, such as directional lights, spot lights, point lights, and area lights, is defined for user-convenience.

- **direct manipulation**: Direct support of user interface events provides users with simple methods to perform picking and highlighting operations.

- **callbacks & sensors**: The placement of user-defined routines into toolkit structures facilitates the construction of complex applications. Callbacks enable users to invoke user-dependent operations while sensors enable users to test the values of specific variables.

- **object-oriented design**: The incorporation of object-oriented principles in graphic libraries enhances the coherency of toolkit constructs, improves the reusability of the package, and assists in the design of independent components.

Although the strength of a toolkit can be documented by its features, a toolkit should also be assessed by its extensibility. Supporting user-defined structures and components enhances

[4] A flat composition of multiple objects places every entity on common ground. No object has precedence or advantage over another.

the usability of a package. A well-planned extension process must be defined if new constructs are to be added to any system. If a well-defined set of rules is not established, the extension process will meaninglessly clutter the contents of any toolkit.

CHAPTER 3

COMPUTER ANIMATION SYSTEMS

For tribal man space was the uncontrollable mystery.
For technological man it is time that occupies the same role.
- Marshall McLuhan, The Mechanical Bride, "Magic that Changes mood"

Animation is generated from the rapid display of images. The slight alteration of successive images imparts the illusion of motion. A computer animation system aims to provide users with powerful, but easy to use, mechanisms to coordinate the motion of animated objects. The strength of an animation system can be partially judged by its ability to separate its internal control constructs from its user interface. An unencumbering interface allows users to concentrate on the design of sequences of animation without interacting with the low-level system architecture.

3.1 CONTROL MODES

Three-dimensional animation systems can be classified according to the methods they use to describe the behavior of animated objects[100]. The three primary methods or control modes are labeled as *guiding*, *animator level*, and *task-level*. In a guiding system, the motion of animated objects must be defined explicitly. Guiding systems, such as BBOP[82]and TWIXT[27], require users to specify the details of motion. In an animator-level system, users are allowed to specify the behavior of objects algorithmically. Typical animator-level systems, such as GRAMPS[62], ASAS[72], and MIRA[52], support adaptive motion and abstraction. In a task-level system, the animation of objects is described in terms of events and relationships. For example, in Zeltzer's knowledge-based animation system[99], animation is specified using broad outlines of movements.

3.2 Motion Specification

Every animation system employs a methodology to specify motion. The simplest of these techniques, key-framing, emulates the steps used in traditional hand drawn animation. Users specify the values of particular variables at key points in time while the computer fills the temporal gaps with intermediate values. Although very powerful, this technique requires users to manipulate an extraordinary large number of variables. For complex sequences of animation, key-framing is arduous and unwieldy.[1]

In the most advanced computer animation systems, motion is specified with a scripting language. Containing many special mechanisms for animation, a scripting language provides users with a notation to describe the dynamic changes in sequences of animation. Essentially, a layer of abstraction is created between users and the intricate detailing of the parameters of motion. A number of dissimilar approaches have been developed which attempt to simplify the complexity of this abstraction and yet still maintain a powerful scripting system. These include new animation languages[19, 72, 11, 98], extensions to existing programming languages (by adding constructs for graphics and animation)[52, 40], and object-oriented designs[21, 25].

The remainder of this chapter discusses the design of several computer animation systems developed in academic environments.[2] Emphasis is placed on the methodology used to specify motion and temporal progression. The systems are presented in chronological order to elucidate the relative changes appearing in animation research during the last two decades.[3] Readers already familiar with the design of these systems may jump to this chapter's summary without loss of continuity.

3.2.1 ANIM8

ANIM8[93] (designed as an education tool) utilizes a block diagram notation very similar to a data flow graph. The flexible block diagram syntax facilitates the interpretation of data paths

[1] Although quite burdensome, key-framing systems prove extremely popular in the consumer field of animation. Most non-key-framing animation systems have not reached a state for general use by the public.

[2] The general lack of literature about "production" systems precludes their discussion in this chapter

[3] Subsection titles refer to the names of systems being described. However, when a system name is unknown, the author's name has been used as a surrogate title.

for animation. It supports three ways to specify motion: algorithmic descriptions, tabular descriptions, and real-time data.

3.2.2 GRAMPS

GRAMPS[62] (developed at the National Resource for Computation in Chemistry, Berkeley Laboratory) utilizes framed object data types and delayed update functions to produce animation. The framed objects permit dynamic variations of an object's coordinate data, while the delayed update functions provide for the variations to the transformations.

3.2.3 DIAL

DIAL[19] (developed at Brown University) utilizes an action specification scheme very similar to musical notation. Each line of a stave describes when actions are to occur and how long they will last. DIAL notation is separated into two distinct parts: the definitions and the execution lines. In the definition phase, particular actions are assigned to animated objects. Execution lines formalize the length and time of occurrences of these actions. The DIAL system is actually only a pre-processor. It functions to convert all of its notation into event definitions, event executions, and timing actions.

3.2.4 ASAS

ASAS[72] (developed at the Architecture Machine Group at MIT) is an object-oriented system that utilizes several types of objects to produce animation sequences:

- *Actors* are the main driving forces behind the ASAS system. One may think of an Actor as "an independent computing process in a non-hierarchical system with synchronized activation and able to communicate with other actors by message passing."[72]. Actors can be initialized, activated, or terminated by scripts, themselves, or other actors. Unlike previous scripting systems, ASAS Actors can be defined to respond to external state changes. This enables users to describe an Actor's behavior in terms of its relationship with its environment.

- *Newtons* represent animated numbers. Their values are automatically updated every time step according to a predefined sequence (chain of cubic piecewise continuous curves).

3.2.5 MIRA

MIRA[52] (developed at the University of Montreal) is a procedural based system utilizing the concept of language extensions. With regards to scripting, MIRA's extended language supports animated basic types and *actors*. Programmers describe how the traits of many standard types, such as integers, floats, and vectors, vary with time. They can define the starting and ending times, the starting and ending values, and the evolutionary law[53] or function that governs each basic type's value. An actor data type is a time constrained abstract graphical type. It is only valid within a specific interval of time. An actor is constructed from a *time range* and a graphical representation.

3.2.6 TWIXT

TWIXT[27] (developed at Ohio State University) is a multi-track event driven animation system. Events on every track (stored as events lists) indicate the transitional behavior of display parameters. At each frame, the system evaluates the activity of every track of every object. Higher levels of functionality are provided through transformations, such as coping, shifting, and scaling, on individual tracks.

3.2.7 AVENUE

AVENUE[17] (developed in Japan) is a rule-based motion system that generates animation automatically based on environmental information and user-specified criterion. Utilizing an implicit representation, animation is presented as a collection of events. Each event refers to the changes in objects and their environments with respect to place and time. Special events called *rules* enable users to specify prescribed guides for conduct or action. Every event (and rule) is represented by the following tuple $< \mathbf{L}, \mathbf{R}, (\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \ldots, \mathbf{x_n}), \mathbf{F} >$, where $\mathbf{L}$ is a time-space location, $\mathbf{R}$ is n-array relation (logical expressions), $\mathbf{x_i}$'s are individuals, and $\mathbf{F}$ is a boolean flag indicating whether the event is to be labeled as "true" or "false".

The system generates motion data from the analysis of events and rules. At each point in time, the system selects an appropriate rule (from a rule set) and determines its validity. If valid, the rule is applied and if necessary, new events or rules are created. The system repeats itself until all rules (or events) are exhausted.

3.2.8 FIUME

Fiume[21] (when at the University of Geneva) created an object-oriented language for expressing the temporal co-ordination of animated objects. Every expression denotes a temporal relationship between instances of animated objects. Temporal operators allow users to specify the *chronological sequencing*, *repetition*, *asynchronous* and *nondeterministic execution*, *temporal overlap*, *conditional triggering* and *simultaneous activation and termination* of multiple objects. Operators also exist to pause or delay the activation of objects relative to others. Since all objects are defined over a continuous time domain, the scaling speed of animations can be enforced by reducing the sampling rate.

To enhance reusability, all object displacements, for example, trajectories, are encapsulated as *motion objects*. Motion objects have duration and temporal properties, and provide users with the ability to script complex motion patterns.

The system scheduler utilizes multiple *binary expression trees* to generate animation. Every expression creates an independent tree in which the nodes represent the displacement in *ticks* between the left and right subtrees. Each scene in an animation is composed of a forest of expression trees - in particular, one synchronous tree linked with multiple active asynchronous trees. Each tick from the scheduler enables time to percolate down each tree, causing the formation of messages to affected objects.

3.2.9 SOLAR

SOLAR[11] (developed at the Institute of Systems Science) is an object-oriented three-pass interpreted animation language utilizing abstraction, adaptive motion, and controlled environmental access to create complex scenarios. Utilizing a master clock to synchronize all operations, all statements are either synchronous or asynchronous. At every clock cycle, asynchronous

statements are checked and, if necessary, executed before synchronous ones.

3.2.10 CLOCKWORKS

CLOCKWORKS[25] (developed at the Rensselaer Design Research Center at RPI) is an object-oriented animation system embracing a variety of image synthesis, modeling, and simulation capabilities. CLOCKWORKS' scripting system CORY[56] utilizes a two tiered approach to isolate the data structures and data manipulation from the user interface. All animation sequences are broken down hierarchically into sets of cues and scenes. A cue provides the starting and ending times for particular actions, while a scene represents the consolidation of a set of cues that are interrelated.

Scenes serve to limit the time that a set of common cues remain active. To enhance top down design, all cue frame times are *local*; their starting times are relative to the start of the scene in which they are contained. Only scenes utilize the global clock. In CORY, all scenes are deemed to be independent of the others and non-overlapping in time.

3.2.11 PINOCCHIO

PINOCCHIO[54] (developed at the Politecnico di Milano, Italy) utilizes a motion database (movement dictionary) and an object-oriented mechanism to animate a sequence of actions. Motions in the movement database are classified according to a *movement grammar*. Movements (verbs) are classified as either *transitional*, *locomotional*, *environmental*, or *communicational*. Additional action parameters include *space* and *time* attributes, *object* and *position* attributes, and *qualitative aspects*.

The system is composed of four special subclasses of the class *object*. These include the *director*, *person*, *motion*, and *camera*. It is the responsibility of the *director* to control the general execution and coordination of scenes by defining an *animation script* and by associating motions to all the active objects. Motions are related through a set of temporal operators which include rules for *sequential execution*, *parallel execution*, *repeated execution*, *time delay*, and *grouping of motions*. The director also associates with every motion object a set of *initial* and *final constraints*. Constraints specify *motion timing*, *spatial location*, and *coordinations* with other

motion objects.

A *person* object contains the geometric description of an entity and a script to coordinate its behavior by controlling the various *motion* objects associated with it. It is the responsibility of a *person* and its *motion* objects to coordinate themselves once the director has specified the constraints on motion performance. In other words, the purpose of the director is only to issue general instructions to actors. It is the burden of the actors to determine how they should perform.

Finally, the *camera* is utilized to manipulate the viewing parameters of each animation sequence. It also follows an animation script given to it by the director.

3.2.12 ZELEZNIK

Zeleznik et. al[98] (developed at Brown University) created an object-oriented animation system that utilizes a hierarchical delegation architecture to dynamically change the attributes of all its objects. Objects send and receive time dependent messages indicating how it and its influencees are to change. Changes can be specified using scripted, gestural, or behavioral specifications. This novel interactive modeling and animation system provides users with an environment where both time and behavior are modeled as first-class objects.

The systems's flexibility can be attributed to it use of *time-varying messages*, *lazy evaluation schemes*, *caching*, *time-varying delegation hierarchies*, and *multiple controllers*. Time-varying messages extend modeling tools to support animation. Lazy evaluation and caching exploit inter- and intra-frame coherency, while multiple controllers enable users to specify complex animations. The time-varying delegation scheme enables objects to alter their prototypical behaviors. Unlike most delegation-based systems, object hierarchies can be altered, they are not "static".

3.2.13 KALRA

Kalra[40] (developed at Caltech) utilizes a time primitive, called an *event unit*, to create complex time sequences or event systems with discontinuous behaviors. The organization of event units provides users with a *time programming* language to develop hierarchical schemes for motion

sequences. *Event units* are specified as triplets, $S : (B_i(X), L(X), B_{i+1}(X))$, where X is the state of the system, $B_i(X)$ is the behavioral rules of the system before the event, $L(X)$ is the logical condition signifying the event, and $B_{i+1}(X)$ is the behavioral rules of the system after the event. General behavioral rules include known functions of time, differential equations of motion, constraints, zero time behaviors, and initializations.

Event systems are constructed from the manual linking of *event units*. Directed graphs with event systems as nodes and edges representing the connections between the event units can be utilized to connect event systems. General composition techniques include the following:

- *time line*: a linear arrangement of event systems. Each event system can be entered from and lead to only one other event system, and only one event can occur in every event system.
- *time tree*: allows multiple connections (without loops) between event units. Behaviors may enter from or lead to more than one event system.
- *time graph*: same as time trees, except for loops. The system may visit events that has it been to before.

3.3 SUMMARY

This chapter has presented a brief survey of computer animation systems from past to present. Each system is designed to provide users with an alternative and powerful interface to control the behavior of animated objects. Animation systems are distinguished according to their degree of abstraction and technique of motion specification. Variable control modes enable animators to design complex scenarios at a variety of different abstract levels. Scripting languages provide users (those with programming experience) a special notation to specify time-varying actions.

The continual push to provide users with advanced modeling features, alternative control modes, and new animation techniques has produced a variety of innovative features in computer animation systems. Although new tools have emerged from a variety of domains, three key research fields have been dominant contributors. They are as follows:

- **Motion Patterns**: To facilitate the reuse of previously defined motions, several animation systems use some variation of a *motion patterns*[53, 21, 11, 54, 40]. Also called *controllers*[98], motion patterns are used to assign pre-defined movements or state changes to user-defined entities. Ideally, an animator's chore becomes easier as he or she accumulates a personal collection of motion controllers.[4]

- **Adaptive Motions**: The necessity of modeling discontinuous behaviors has guided the development of *adaptive motions*[100]. These structures enable animators to define an object's state in terms of its relationship with its environment. Objects can be instructed to observe its surroundings and respond to particular stimuli. Adaptive specifications are integral elements of rule-based[11, 17, 40] and goal-directed systems. From a simulationist's viewpoint, adaptive techniques are produced with the *discrete-event activity scanning* model (described in section 5.1.3).

- **Temporal Reasoning**: The unwieldy nature of organizing large collections of parallel actions in complex models has spurred the induction of *temporal reasoning* abstractions into computer animation systems[21, 25, 54]. Providing users with the ability to define relationships between sets of actions enables them to coordinate the behaviors of objects at a high level of abstraction.

Recent research has shifted to accommodate new lines of thought toward the creation of computer animations. The need to develop realistic models, to control all modeling attributes, and to use multiple simulation techniques has induced several new fields of research. Modern topics include physically-based modeling[86], first-class temporal representations[98], and discrete-event techniques[20, 40].

[4]This has yet to be proven true.

CHAPTER 4

SIMULATION

Who controls the past controls the future.
Who controls the present controls the past.
- George Orwell, Nineteen Eighty-Four

All computer animations are simulations of objects in motion. Using a pre-defined set of laws and objectives, (most) animators attempt to design computers animations that impart the illusion of life.[1] The influence of physical laws, realistic models of motion, and patterns of interaction have narrowed the gap between simulation and animation. Although the development of time-varying simulations has become an integral part of computer animation research, very few researchers have incorporated simulation constructs or languages. Most users emphasize the accomplishments of their research without concern for the methodology they use. The lack of a common foundation between various simulation frameworks has hindered the development of general animation systems supporting a variety of simulation models. The inclusion of modeling concepts and tools from simulation languages is essential to the construction of a powerful, yet flexible, computer animation system.

Although computer animation scripting languages and simulation languages may have common goals, they do not attempt to solve similar problems in the same fashion. These incongruous views of problem solving are a result of the differing design philosophies of each approach. In a script-based model, a central description defines all the actions within a system. Individual entities do not usually control their own actions. The system follows the strict script of behaviors. Scripting languages provide many different ways to specify change: interpolation schemes, sets of behaviors, and time-dependent variables. Unlike script-based systems, simulation languages do not usually support the concept of a central database controlling the evolution of change within a simulation. Simulation languages localize the control within the components of

[1] Some animators exaggerate the motions of their objects to induce greater dramatic effect.

the simulation. After connections are created between system modules, a simulation proceeds. A central control mechanism is defined only to ensure the passage of messages between modules and to guarantee the uniform passage of time. Scripting languages are generally used to exhibit a particular behavior while simulation languages are commonly utilized to discover the behavior of a system over time.

The next two sections of this chapter provide a brief synopsis of simulation languages and environments. Each section enumerates the distinguishing features of various simulation building tools. The chapter terminates with a discussion of the development of simulators using standard programming languages. Particular attention is given to the influence of object-oriented principles on their designs.

4.1 General Simulation Languages

Most simulation systems allow users to describe their models using a prescribed descriptive language. Descriptive specifications range from a straightforward sequential style to the extreme *general network* type. The system translates (or sorts) the descriptive statements into a formal description that is carried out by a sequential program. For parallel models, the order of the statements does not usually make a difference. Each statement describes independent, yet interacting, actions or processes[96]. This type of design provides a strong foundation to utilize object-oriented constructs and principles.

Simulation languages can be divided into two major groups. They can be classified as either *scenario* or *procedural.* In scenario languages, active transactions execute descriptive scenarios, typically in the form of block diagrams, to model simulations. The languages SLAM II[70], SIMAN[67], and GPSS[77]fall into this category. Although procedural languages do not support as many simulation constructs as scenario languages, their strength comes from the inclusion of general-purpose programming devices with simulation-specific techniques. Their power and flexibility enable users to attack a wider range of problems. Languages falling into this category include HSL[74], SIMULA[6], SIMSCRIPT II.5[42], CSIM[79], and GPSS/H[78].

4.1.1 Scenario Languages

There are four major disadvantages in using scenario languages. First, implementing a hierarchical stepwise refinement scheme[2] for modeling is arduous. The scenario block network approach to modeling does not facilitate the use of hierarchies. All the components of a simulation are designed at one level of abstraction. Hence, the introduction of additional detail to any simulation component may require a complete replacement of the component with a component of higher detail. Second, high level modularity is not supported. There is no separation between "control information" and "model actions" statements. The static and dynamic characteristics of a system are defined according to the experimental conditions under which it is run.[3] Third, models not readily supported by scenario constructs require complex implementation. Fourth, the limited use and length of user-defined identifiers inhibit program readability.

However, scenario languages do provide one good feature. Their modeling construction methods are conducive to graphical specification techniques. This advantage can enhance the ability of users to comprehend the characteristics and properties of any system[74].

4.1.2 Procedural Languages

Unlike scenario languages, procedural languages provide stepwise refinement schemes, high-levels of modularity, full complements of structured control statements, and long meaningful variable names. For example, SIMSCRIPT, an event-oriented language, is organized in a five-level hierarchy: three levels of a general purpose programming language, one level of entity manipulation, and one level of additional simulation features, such as time manipulation techniques. Its pseudo self-documenting code helps bridge the gap between modeling and programming.

[2]Stepwise refinement means building components from abstract elements and then refining those elements deemed to be important into sub-elements to introduce additional detail.

[3]Ziegler[96] stresses that a proper simulation system must make a distinction between the models of a simulation and the experimental frame under which the models are run. Altering the experimental frame should not require the models to be altered.

4.2 Simulation Environments

Despite the large pool of simulation languages, many users do not embrace their usage to construct their simulations. Many users view the strict interface and design methodologies imposed by simulations languages as detrimental qualities. Consequently, many users spend considerable effort developing their own simulation environments. Each new system accomodates the needs of users in a specialized field. Unfortunately, the range of requirements for this multitude of users precludes the incorporation of generic constructs into many new languages.[4] This loss of generality has prohibited the widespread usage of many exceptional systems. Although simulation environment lack large audiences, the abundance of features established within each new framework supplies designers of new simulation languages with an exhaustive set of useful suggestions, innovative ideas, and novel concepts.

Every simulation environment partitions the simulation modeling process. According to [69], an ideal distribution which promotes greatest reusability is achieved when distinctions are made among the *physical*, *informational*, and *control/decision* elements of a simulation. Physical objects represent tangible things found in the real world such as parts, machines, and robots. Informational objects may also be tangible,[5] but most often they represent facts or pieces of data. For example, constraints or series of operations are represented by these type of objects. Control/decision objects represent the creative intelligence of a simulation. Their primary function is to evaluate the state of the system, exercise logic algorithms, and incite appropriate actions when required. Basically, they provide the interconnections between physical and informational objects.

The following sections discuss the frameworks of various simulation environments. Each system exemplifies an alternative approach toward the creation of physical simulations. Careful attention has been made to emphasize each environment's division of the simulation modeling process.

[4] Given enough time and resources, many researchers would probably choose to increase the viability of their systems. However, the realities of life preclude this type of activity from occurring.

[5] An tangible object whose information content is of primary importance may be classified as an informational object. e.g. bills of materials.

4.2.1 SIMLAB

SIMLAB[66] is a software environment for creating reusable physical systems. Although narrow in scope, it introduces an interesting alternative view concerning the production of simulations. Unlike most simulation tools, SIMLAB does not require users to build simulators using a conventional programming language. Simulations are automatically created from high-level expressive descriptions defined by users. Users are not required to define data structures, combine numerical packages, implement visualization routines, or implement algorithms via a programming language. All these steps are performed by SIMLAB without user interaction.

To create a simulation with SIMLAB, users must define two pieces of information: the *physics* model and the *global formulation.* A SIMLAB physics model is very simple. Each instance contains definitions of *primitives*, *connections*, *quantities* and *constraints.* Primitives represent the basic entities in the model while the connections serve to specify the interactions between primitives. Quantities and constraints represent any primitive's state or constrained behavior. Every physics model is interpreted by the global formulation. This formulation specifies how SIMLAB creates a set of equations from the primitives, connections, quantities, and constraints.

SIMLAB's power comes from its unusual interface. The simulator allows users to concentrate on the problem of modeling without worrying about writing complex programming data structures and algorithms. Users design physics models and formulations while the system handles the creation and operation of the simulation. While limited to the construction of less complex simulations, this method does have it merits. Users can create simulations from a high level of abstraction.

4.2.2 INEFFABELLE

INEFFABELLE[64] is a simulation environment (written in LISP) for the development of reusable robotic models and programs. Central to INEFFABELLE's design is the common body of information and family of functions found in all robot simulation programs. For example, all robotic applications require some set of methods to delineate the geometric and kinematic

parameters of a robot. With INEFFABELLE, users create model robots and workcells[6] through a simple and clear set of rules and procedures. Application specific properties can be assigned to these models using built-in functions.

INEFFABELLE functions can be categorized into three distinct groups: *entity modeling*, *entity manipulation*, and *display*. Robot models and their environments are designed using entity modeling functions. Standard entities found in INEFFABELLE's library include joints, work cells, coordinate frames, links, and sensors. Users can create new entities or alter the properties of existing models through INEFFABELLE's flexible modeling mechanisms. The motion of robots, grasping of objects, and other common robot related tasks are performed using entity manipulation functions. Display and entity manipulation function work together to provide users with computer animations of their animated models.

4.2.3 WADE

WADE (A Workcell Application Design Environment)[36], a process-oriented system written in AML/X,[7] was developed to meet the needs of simulationists designing workcell applications. Before WADE, most robotic systems focused primarily on the aspects of robot programming and simulation while ignoring issues introduced by the broad range of industrial equipment typically found in workcells. Not enough attention had been directed to building tools for creating scenes with interacting components. WADE's designers envisioned a system with tools that would provide users with important information, useful methodologies, and multiple representation schemes during the various stages of workcell development.

WADE can be decomposed into three basic constituents: *modeling*, *simulation*, and *user-interaction*. The modeling component supplies tools to create and manipulate the relevant characteristics of abstract entities (robots, sensors, etc.). The simulation component visualizes the dynamic behaviors of these abstract entities. The user-interaction component provides users with a highly interactive and user-friendly interface.

[6] A workcell is a collection of interconnected pieces of industrial equipment, such as robots, cooperating on a single manufacturing task[36].

[7] AML/X (A Manufacturing Language/eXtended) is a multi-layered programming languages designed primarily for manufacturing applications

4.3 Object-Oriented Simulation Design

Many well known simulation languages such as GPSS, SIMAN, and SIMULA, are based upon object-oriented principles. Each language assists in reducing program development time and enhancing model understandability by providing users with high level simulation constructs. However, most users do not utilize these special programming languages to create their simulations. Most simulations are written in general purpose language such as ADA, FORTRAN, C, and PASCAL. For many users, every new application is constructed from scratch. It is not common for users to generate their own simulation libraries. The failing of simulation languages to gain widespread acceptance can be attributed to the fact that most users lack experience with a simulation language. Compounded with additional learning time, the limitations of simulation languages have hindered users from incorporating them into their arsenal of programming tools. The major disadvantage with simulations designed with general purpose program languages is that users spend too much time specifying the state changes in the simulation system. Valuable developmental time is lost in designing the characteristics of the simulated targets.

Fortunately, the growing popularity of object-oriented programming and the demand for reusable tools have fostered the creation of flexible simulation constructs and environments using popular programming languages. Users can profit from the benefits of simulation tools without expending the time to gain the understanding of another programming language. Tools for simulation development can be roughly divided into two categories: *clients* and *servers*. Clients tools assist in the creation, manipulation, and destruction of the simulation target entities. Targets, labeled as clients, usually represent physical or informational objects. Servers attend to service the clients of a simulation. They perform duties "for" and "on" clients. A client's state and behavior is controlled by sets of servers. The power behind this division is that it enable users to construct independently the characteristics of their clients and the details of their servers. Clients can be constructed without much foreknowledge of their usage, and servers can be built without concern for the internal architecture of the clients they serve.

The following sections present the highlights of several object-oriented simulation and modeling environments constructed from commonly used programming languages.

4.3.1 DOSE

DOSE[55] is a discrete-event C++ simulation environment based on the *connection* paradigm. The entire system is structured as a set of components interconnected through their "input" and "output" ports. A component is defined as an object with its own internal state and collection of handlers. Each handler is a response to a set of internal and external events. Component behaviors are specified with respect to their ports. Once a connection is established, output ports notify its connected input ports, via an event, whenever a change or update occurs. An output port can be utilized to signal an event or to multicast[8] an internal state or variable. An input port can be linked to an internal variable or attached to an event handler. The dynamic attachment and detachment of ports provide users with a flexible simulation mechanism.

The simulator object `Sim` provides users with an interface to the run time system support. It is assisted by three run time support objects: a *component manager* for the handling of components, a *connection manager* for the creation and maintenance of connections, and an *event scheduler* for the planning and sequencing of internal and external events during the simulation.

4.3.2 PRISM

PRISM[90] is a generic event-processing simulator written in C++. Communications is supported with events and (if desire, hierarchically constructed) simulation units or models. The execution of events, called `SimEvents`, is the driving force behind this simulation system. Events are posted to a simulation engine (a `Simulator`) by simulation units (`SimUnits`) to cause future computations. The two most important member functions declared in `SimEvent` is `doEvent` and `cancelEvent`.

Rather than offering one global simulation engine, PRISM allows users to define multiple simulation engines (instances of the class `Simulator`). Each engine manages an event queue and deals directly with the system model (instance of the class `Model`). Members functions declared in `Simulator` interactively control time, post events, cancel events, and obtain information. Although multiple engines can be defined, there seems to be no method to design

[8] To (simultaneously) issue data to multiple ports.

two separate, yet interdependent engines. No procedures exist to regulate the progression of mutually dependent engines. Users must define their own methods to balance and control the operation of multiple engines.

4.3.3 DESAda

DESAda[48] is a simulation template based on tasking in Ada. The system is composed of a combination of servers that provide service to clients and clients which acquire services from the servers. Users are given several "off the shelf" modules to handle the transitions occurring within data entities. The simulation system is divided into five major components.

- **Clients** (*user-implemented*): Implemented as objects, clients are composed of sequential descriptions of their life cycles. A client becomes *active* the instant it is declared or allocated. Each client posts a (time) **signal** to the task controller as to when it wants to be notified. Specific client requirements or needs are sent to the task controller via the parameters of the signal.

- **Task Controller** (*built-in*): This unit is responsible for maintaining the simulation clock and processing all the *signal* requests from every client. After a particular client completes its own logic (after being notified to proceed), the controller continues to process the special needs of the client. Clients may make a particular request such as "get service from a server", "schedule a future event", or "execute one more signal".

- **Event List** (*built-in*): Each node in this unit represents an event to be activated at a future time. Nodes contain two pieces of information not commonly found in standard implementations of event lists. They contain direct links to their clients, and each event node has no knowledge of the effect of the event on the client. Unlike many traditional system where events control the execution threads of the simulation, this system delegates the process of a clients' life cycle to itself. The system merely narrates the timing of events.

- **Servers** (*built-in* or *user-implemented*): Servers are dynamic objects which provide service to clients and other servers. There are two types of servers: *built-in* and *user-implemented.*

Built-in servers are simple functional objects utilized to serve clients one by one. These type of servers usually maintain waiting queues for multiple "service requesting" clients. To handle more complex server sharing, queueing and allocation strategies, users can implement their own server packages.

- **Data Collection** (*user-implemented*): Data collection is performed by each of its separate entities. Upon request, each object can pass along its collected (summarized) data to a user-defined system collector.

4.4 SUMMARY

This chapter has provided a brief overview of simulation languages, systems, and modeling environments. Each approach provides users with special constructs to facilitate the specification of "transitions". Transitional elements reduce users' torment in forming their own methods to alter the states of the system. In addition, all three models support methods to form relationships or communication pathways among collections of objects. Links empower objects with the ability to react to influential forces and external stimuli. Apart from the primary benefits, both mechanisms enable users to focus their attention toward the improvement of their models. Users are not compelled to continually redefine sets of structures common to all their applications.

Although tools for simulation come in a variety of forms, each tool can be gauged according to the strength of its underlying framework. A superior design is distinguished by its ease of use and reusable potential. Providing users with the ability to easily reuse segments of previously defined simulations enhances their productivity. A simulation tool's potential can be measured according to three important qualities.

- It must be able to support modular design. Modularity enables users to construct complex simulations from the amalgamation of various well-defined components. Modules can also serve as building blocks for the creation of multifarious components. In an ideal modular environment, the task of modifying or creating new models is reduced to replacing old objects with new ones.

- Simulation tools must provide users with appropriate abstractions to refine hierarchically their models. Alternative viewpoints empower users to manufacture simulations at disparate perspectives. High-level expressions[66] and stepwise refinement techniques[6, 42] are common abstraction building mechanisms.

- A division of the simulation modeling process must be apparent. A highly evolved framework provides users with a clear path for application development. *Model creation*, *entity manipulation*, *temporal management*, and *display techniques* are commonly defined partitions in many simulation environments. Although a clear division has not been solidified by the simulation community, an observable consensus can be extracted from recent research. Modern developments have advocated a separation of decision making algorithms from the models they are controlling. An environment's reusable potential decreases when it disperses and embeds control logic into its models. A strong division enhances a user's ability to alter continually a model's interaction with its surroundings.

The increasing demand for general simulation mechanisms as well as the rise of object-oriented methodologies have influenced the design of a variety of new simulation environments. Using an assortment of common object-oriented languages, several simulationists have objectified the simulation modeling process. The drive to maintain object-oriented principles, such as encapsulation and modularity, has introduced additional perspectives on the development of simulation environments. Design philosophies, such as first-class events[90] and first-class object interfaces[55], introduce new architectures and issues not commonly found in standard simulation systems.

CHAPTER 5

TEMPORAL MANAGEMENT

What then is time? If no one asks me, I know what it is.
If I wish to explain it to him who asks, I do not know.
-St. Augustine, Confessions

In time-varying simulations,[1] a variety of techniques are employed to control and manipulate the flow of time. An assortment of methodologies provide different techniques to advance time, to structure time, and to define the logic and sequence of events. The capabilities of each approach is directly proportional to its complexity. In general, systems and languages possessing an advanced set of features require users to define and specify a large set of simulation variables. Therefore, in many cases, users may opt to choose simpler tools. The needs and requirements of users vary from individual to individual. Therefore, to obtain a large following, an optimal tool for simulation should define a simple interface with a large collection of features.

This chapter provides a brief introduction to a variety of important topics related to the manipulation of time. A quick analysis of temporal management techniques is followed by a discussion of the advantages and difficulties associated with the design and development of a process-oriented simulation.

5.1 TEMPORAL ADVANCEMENT

Underlying every time-varying simulation resides a methodology to regulate the progression of time. It the responsibility of this simulation engine to ensure that every module in one system efficiently dwells in the same time frame. For most simulations, it is essential that simultaneous system actions are performed at the same point in time. The progression of concurrent actions and activation of mutual interactions are indispensable qualities of a sound temporal based

[1] In a time varying simulation, time enters explicitly as an argument of the rules of interaction.

simulation. Time must not progress faster than the quickest acceptable rate of any element in a simulation.[2]

5.1.1 CLASSIFICATION

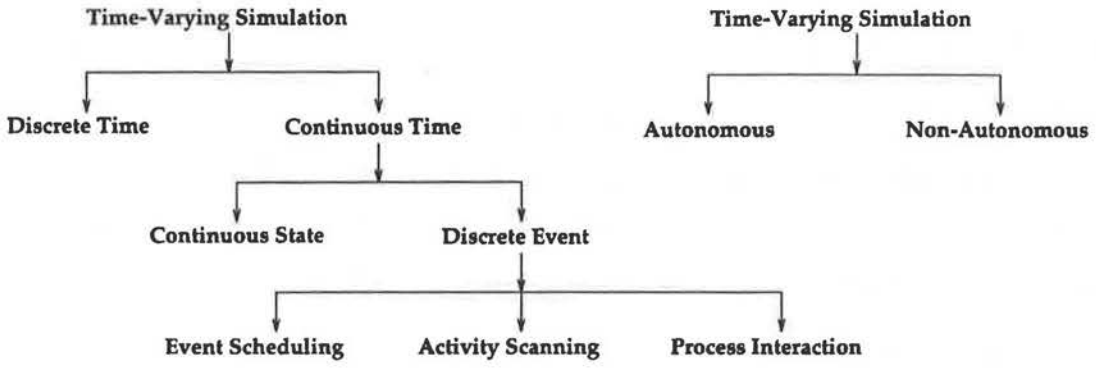

Figure 5.1: Classification of Time-Varying Simulations

All time-varying simulations can be classified as either *continuous* or *discrete* time approaches. In the former case, time flows continuously[3], while in the latter, time advances in pre-defined periodic jumps. Most animation research is discrete. The clock continually moves forward in discrete time intervals, while the simulation's descriptive variables assume a discrete set of values.

The continuous time approach can be further divided into the *continuous state* and the *discrete event.* In the continuous state approach, state changes are continuous and the system's time derivatives are governed by its differential equations. The discrete event approach is characterized by state changes occurring in discontinuous jumps and events arbitrarily separated from each other.

Time-varying simulations can also be classified according to their interaction with their

[2]In real-time simulations, time may advance faster than the quickest desired rate of any system module. It is the obligation of every system element to compensate whenever the pace of the simulation exceeds its ideal rate. However, for this thesis, real-time demands are not in effect. This is an issue for future work.

[3]Because most simulations are performed on digital computers, time does not truly flow continuously. State variables within continuous systems are usually described by deterministic differential (or algebraic) equations which are solved using standard step-by-step methods.[58]

environment[96]. If the simulation is not influenced by its environment, it is labeled as *autonomous*. Conversely, a *nonautonomous* simulation is directly influenced by the events occurring in its environment.[4] The general structure of time-varying simulations is shown in Figure 5.1.

5.1.2 Discrete Time

Discrete time simulations are generally sequential and iterative. They continually repeat the same set of steps until a terminating condition is met. There are usually no constructs for the scheduling, creation, or deletion of events and no methods to control the passage of time. All discrete time simulation are variations of the following prototypical procedure[96].

1	Initialize state variables.
2	Initialize the clock to a starting time.
3	Apply the rules of interaction to the contents of the state variables to produce new values.
4	Advance the system clock.
5	Check if the clock value exceeds the termination time. If yes, stop. If no, go to step 3.

Because most of the interactions of a simulation's components are not sequential, the discrete time approach is limited to a small subset of the possible simulations. The "parallel" nature of most simulations require that the simulation engine handle many simultaneous actions. A good simulation kernel must be able to coordinate, control, and execute concurrent actions in their proper time sequence[96].

5.1.3 Discrete Event

The discrete event philosophy frees the simulator from fixed time step intervals. The system is driven by an event list containing the sequential ordering of "next clock" times when components are scheduled to alter their state. Scheduled events times are known as *hatching times*. The simulator advances the clock to the closest hatching time on its list and executes all component actions prescribed for that time. Since the system is not confined to a constant time step, the

[4] In the field of dynamical systems, simulations are defined to be autonomous or nonautonomous if they are dependent on time. This nomenclature is ignored to comply with the terminology used in simulation literature.

simulator ignores the intervals between clock jumps where no actions are known to occur[96]. It is only recently that research[20] has addressed the usage of discrete event modeling in computer animation.

The discrete event simulator relies on two basic presumptions. First, the predictable hatching times of some events are a direct result of (the hatching of) other events. When an event's hatching time is predictable, it can be scheduled. Second, unless (or until) the state change of a prescheduled event causes a model to alter its state, the model will not undergo any modification in its condition. The validity of any simulation using the discrete event philosophy fails if either of these two presumptions are violated.

Conflicts arise in the discrete event philosophy when two or more events are triggered at the same time. Since computers are inherently sequential processors, concurrent events can not be resolved in the same instant of time. Because only one event can be processed at a time, several different tie-breaking schemes have been developed. The three most basic methods are as follows:

- Select events as they are found. If A was found before B, process A before B.
- Select an event at random.
- Specify a tie-breaking rules that select the most "imminent" event.

The third approach is the most widely utilized method. Its simplicity and automatic sequencing of individual events is ideal for many situations[96].

In discrete event simulations, special attention must be paid to external events that effect the state of the system. Controlling the effects of external events should not necessarily be a large responsibility of the autonomous system rules. Only internal events should be controlled by the autonomous rules, while external events are controlled by *special* rules[96].

5.2 Discrete Event Strategies

Discrete-event simulations can be separated into three related categories: **event scheduling**, **activity scanning**, and **process interaction**. The third being a combination of the first two. In all three cases, actions are executed at specific event times. In this section, we briefly go over

the main structure and features of each classification using an informal description[5] developed by Zeigler[96]. The description is divided into three major parts: *components*, *descriptive variables*, and *component interactions*. The components are the elements from which the simulation is constructed. The descriptive variables serve to characterize the range of states each component can achieve, and each component's role in the simulation. The component interactions are the rules that govern the behavior of the simulation. They define how components interact other components.

5.2.1 Event Scheduling

In an event-oriented scheduling approach, every event is prescheduled. Each event contains a reference to a point in time when it is to be executed. An event is not triggered until its reaches its time of activation. The scheduling of events is controlled by an *event list*.[6] This list sorts every event awaiting activation by its hatching time. Events with earlier activation times are situated near the head of the list. As a simulation progresses, events are placed, executed, and removed from the event list. The event scheduling approach requires all users to design their simulations from a global viewpoint. A complete description of all the changes to the entire system must be given for each event occurrence. Additionally, only explicitly designated state changes can alter the behavior of the system. It is not possible to test the state of any system component to invoke state transitions. The event scheduling methodology is structured as follows:

1. **Components:**

   In the event scheduling approach, the set of components, $D = \{\alpha_1, \alpha_2, \ldots, \alpha_N\}$, is divided into ACTIVE and PASSIVE types. ACTIVE components invoke changes in a system, while PASSIVE types retain their state indefinitely unless acted upon by other components.

[5]The description is informal because it is open to certain intrinsic problems, such as incompleteness, inconsistency, and ambiguity. However, it is very useful because it communicates the essential nature of a simulation strategy.

[6]Although the word 'list' implies a linear data structure, other possibilities exist, such as indexed lists and heaps. Therefore, *event queues* is a more appropriate term than *event list*. However, event list is used to conform to the standard terminology used in simulation literature.

2. **Descriptive Variables:**

   Every ACTIVE type is described by its state, time to activation, and set of influences. At any moment during a simulation, an ACTIVE component is defined by its value and the time that remains before it influences the behaviors of other components in the simulation.

		STATE-OF-α	TIME-LEFT-IN-α	INFLUENCES-OF-α
ACTIVE-α	Range:	S_α (a set)	$R_\alpha = \{0, \infty\}$	D
	Value:	s_α	σ_α	$\{\beta_1, \beta_2, \ldots, \beta_M\}$
PASSIVE-α	Range:	S_α		
	Value:	s_α		

3. **Component Interaction:**

   For each ACTIVE α a local transition function[7] $\{\delta_\alpha\}$ is specified. This function simply maps the set of state assignments to the INFLUENCES-OF-α.

$$\underbrace{((s'_{\beta_1}, \sigma'_{\beta_1}), \ldots, (s'_{\beta_A}, \sigma' {}_{\beta_A})}_{ACTIVE}, \underbrace{s'_{\beta_{A+1}}, \ldots, s'_{\beta_M}}_{PASSIVE}) = \delta_\alpha((s_{\beta_1}, \sigma_{\beta_1}), \ldots, (s_{\beta_A}, \sigma_{\beta_A}), s_{\beta_{A+1}}, \ldots, s_{\beta_M})$$

   The transition function σ_α is split into m distinct functions $\{\sigma^i_\alpha\}$, where m represents the number of states that S_α can assume. Each function describes the activity of component α when it is started in one of its possible states. For many simulations, each σ^i_α is coded separately as a program or routine. This design facilitates the use of object-oriented programming since every function can be represented as a separate object or member function.

Next Event Simulation

Simulation systems providing event scheduling operations are often called *next event* simulations. Apart from the development of a temporal metric to schedule the activity of events, all *next-event* modeling systems require the creation of an NEXT-EVENTS-LIST and a SELECT function. The NEXT-EVENTS-LIST is used to dynamically sort pairs of the form [event, time], where each pair defines the activation time of a specific action. Actions associated with earlier

[7] Given a list of values of the state variables of a model at time t_i, a *state transition function* produces a list of values for the model's state variables at time t_{i+1}.

activation times are placed at the head of the list. Given a set of active types from the NEXT-EVENTS-LIST, the SELECT function singles out an individual pair. For many simulations, the behavior of the system is decided from the choices determined by the selection function. To ensure the selection of certain events over others, some simulations incorporate a priority ordering.

All next-event simulation are subtle variations of the prototypical procedure shown in Figure 5.2. Developed by Zeigler[96], this algorithm advances time from event to event. As events are executed, the states of influenced events are adjusted. This usually entails the reordering of future events in the event-list.

Initialization	1	Set CLOCK to initial simulation time t_0
	2	Set variables $\overline{S}_{\alpha_1}, \ldots, \overline{S}_{\alpha_n}$ to hold the initial values of s_α's.
	3	For every ACTIVE α, place the pair (EVENT-s_α, $t_0 + \sigma_\alpha$) on the NEXT-EVENT-LIST. (order the list by low time)
Time Advance	4	Advance the CLOCK to the time of the first pair on the NEXT-EVENT-LIST. Call the new time, t, the NEW-EVENT-TIME.
Tie Breaking	5	Apply SELECT to all components with events scheduled at NEW-EVENT-TIME. Let $\overline{\alpha}$ denote the winning component.
State Transition	6	Remove (EVENT-$s_{\overline{\alpha}}$, $t_{\overline{\alpha}}$) from the NEXT-EVENTS-LIST.
	7	Invoke routines for EVENT-$s_{\overline{\alpha}}$ (a) Check if each ACTIVE-INFLUENCEE-β-OF-$\overline{\alpha}$ is a member of a pair on the NEXT-EVENT-LIST. If yes, remove it from the list. $\sigma_{\beta_i} = \begin{cases} t_{\beta_i} - t & \text{if (EVENTS-}s_{\beta_i}\text{-OF-}\beta_i, t_{\beta_i}\text{) was} \\ & \text{removed from the NEXT-EVENT-LIST)} \\ \infty & \text{otherwise} \end{cases}$ (b) Adjust the state for every INFLUENCEE-β-OF-$\overline{\alpha}$ (set $\overline{S}_\beta$ to s'_β). For every ACTIVE-β with $\sigma'_\beta < \infty$, place the pair (EVENT-s'_β-OF-β, NEW-EVENT-TIME $+\sigma'_\beta$) in its proper place on the NEXT-EVENT-LIST.
Any events left?	8	If CLOCK and the time of the first pair on the NEXT-EVENT-LIST are equal, jump to 5.
Termination Test	9	If NEW-EVENT-TIME exceeds termination time, STOP. Else goto 4.

Figure 5.2: Next-Event Prototype

5.2.2 ACTIVITY SCANNING

The activity scanning approach is an augmented event-oriented system. Apart from allowing the explicit prescheduling of component activation times, contingency tests allow the conditional activation of state changes. Every satisfied test implicitly schedules the execution of a collection of events. This enhanced approach, unlike the event oriented, enables components to possess negative "time to activation" times. At any time, there may be many components in the "ready" (α-TIME-LEFT $= 0$) or "due" (α-TIME-LEFT < 0) condition. A component is defined to be due if is ready to be triggered and its activation is being precluded by the absence of an external influence. Only when the external influence obtains a certain state will the component be activated. Therefore, the ordering of waiting components is controlled by a *conditions list* or *activities list*, not an event list. The activity scanning methodology is structured as follows:

1. **Components**:

   The set of components, $D = \{\alpha_1, \alpha_2, \ldots, \alpha_N\}$, is divided into ACTIVE and PASSIVE types. ACTIVE components impose changes to a system, while PASSIVE types retain their state indefinitely unless acted upon by others.

2. **Descriptive Variables**:

   Activity scanning ACTIVE types are characterized by the same set of descriptive variables that describe next-event ACTIVE types. Each type is distinguished by its value and capability to alter the states of other components in a simulation. However, activity scanning ACTIVE types are also distinguished by a set of influential components which define when and how ACTIVE types exercise its authority over others.

	ACTIVE-α		PASSIVE-α	
	Range	Value	Range	Value
STATE-OF-α	S_α	s_α	S_α	s_α
TIME-LEFT-IN-STATE-α	$R_\alpha = \{-\infty, \infty\}$	σ_α		
INFLUENCEES-OF-α	D	$\{\beta_1, \ldots, \beta_M\}$		
INFLUENCERS-OF-α	D	$\{\overline{\beta}_1, \ldots, \overline{\beta}_M\}$		

3. **Component Interaction:**

   • For each ACTIVE-α a local transition function $\{\delta_\alpha\}$ is specified. Given the union of the current values for the INFLUENCERS-OF-α and the INFLUENCEES-OF-α, this function simply produces a new list of values for the INFLUENCEES-OF-α. Assigning $\overline{s}$ to represent this union, $\delta_\alpha(\overline{s})$ defines the value of the INFLUENCEES-OF-α immediately after α is tested and activated.

   • Associated with every INFLUENCERS-OF-α is a boolean predicate. This logical assertion represents the condition that determines if α's state is to be altered. Immediately after the condition is deemed to be true, a set of actions is performed on the INFLUENCEES-OF-α. Given that C_α represents the boolean predicate on the state assignments to the INFLUENCERS-OF-α and f_α[8] defines the action performed by α on the INFLUENCEES-OF-α, the transition function δ_α is implemented as follows:

$$\delta_\alpha(\overbrace{(s_{\beta_1},\sigma_{\beta_1}),\ldots,s_{\beta_m}}^{\text{INFLUENCEES}},\overbrace{(s_{\overline{\beta}_1},\sigma_{\overline{\beta}_1}),\ldots,s_{\overline{\beta}_m}}^{\text{INFLUENCERS}}) =$$

$$\begin{cases} f_\alpha((s_{\beta_1},\sigma_{\beta_1}),\ldots,s_{\beta_m}(s_{\overline{\beta}_1},\sigma_{\overline{\beta}_1}),\ldots,s_{\overline{\beta}_m}) & \text{if } C_\alpha((s_{\overline{\beta}_1},\sigma_{\overline{\beta}_1}),\ldots,s_{\overline{\beta}_m}) = \text{ TRUE} \\ ((s_{\beta_1},\sigma_{\beta_1}-t(s)),\ldots,s_{\beta_m}) & \text{otherwise} \end{cases}$$

   If C_α is TRUE, then apply f_α to obtain the new states of the INFLUENCEES-OF-α. Otherwise, perform no actions except for clock updates.

Activity Scanning Simulation

In the activity-oriented approach, the actions of the simulator are partitioned into segments called *activities*. Every activity, defined as the state of a model over an interval, is delimited by two successive events. Each activity is associated with a boolean condition set to true or false depending on the state of the system. As the simulation progresses from event to event, the simulator scans the status of all the activities in the model. Every activity satisfying its contingency test is immediately scheduled for execution. The activity scanning approach is more attractive than the event oriented approach when the number of events in a simulation

[8]The function f_α has the same domain and range as δ_α.

grows to great size. However, this approach has difficulties when used to accurately model continuously changing operations. Continually varying variables must be discretized into several distinct states if they are to be manipulated in an activity scanning approach. Apart from the introduction of errors by poor apportionment, this approach requires the user to specify a discretization algorithm.

In the activity scanning approach, the CONDITIONS-LIST dynamically orders a list of activities according to a priority-based metric. Activities with superior rank are situated at the top of the list. Each activity is usually stored as a triplet of the form (ACTIVE-α, f_α, C_α), where f_α represents CONDITIONS-ROUTINE-FOR-α (boolean predicate on state assignments) and C_α identifies the ACTIVITY-ROUTINE-FOR-α (state altering action). After every event, activities in the list are scanned from top to bottom. Although it is possible to develop an autonomous SELECT function, the descending scan eliminates the need of such a routine. Activities are selected by a SCAN pointer, as shown in Figure 5.3, according to their location in the list.

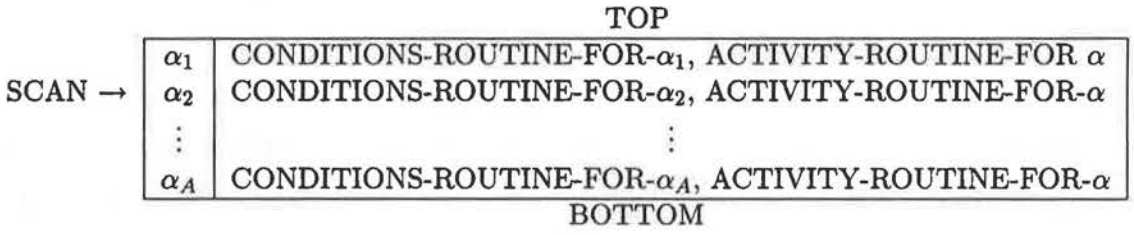

Figure 5.3: Conditions List

All activity scanning simulations are subtle variations of the prototypical procedure shown in Figure 5.4. This routine, developed by [96], advances time from one event to another. As events are triggered, the algorithm scans its conditions list to determine if any conditional events need to be activated. The simulation languages CSL[9] and SIMON[31] are based upon this approach. Although it may seem redundant to continually scan the *conditions-list* after every event, it is a required operation. If events occur between successive scans, it is possible for the scan to miss a state change. The diagram in Figure 5.5 illustrates this problem.

Initialization	1	Set CLOCK to initial simulation time t_0
	2	Set state variables $\overline{S_\alpha}$'s to initial values of s_α's.
	3	Initialize CONDITIONS-LIST.
	4	Set time cells T_α's to initial values $\sigma'_\alpha s$
Activity Scanning	5	Move the SCAN to TOP of CONDITIONS-LIST.
	6	SCAN down until the first ACTIVE-α is found ($t_\alpha \leq t$) and its CONDITION-ROUTINE-FOR-α returns TRUE when applied to the INFLUENCERS-OF-α.
State Transition	7	Execute the ACTIVITY-ROUTINE-FOR-$\overline{\alpha}$
Test for End of Scanning	8	If SCAN has not reached the BOTTOM of the CONDITIONS-LIST then goto 5.
Time Advance	9	Advance the CLOCK to the time of the next event (the minimum $T_\alpha \geq$ the current CLOCK value)
Termination Test	10	If CLOCK exceeds termination time, then STOP! else goto 5.

Figure 5.4: Activity Scanning Prototype

5.2.3 Process interaction

The process interaction approach is a combined event scheduling - activity scanning system. In addition to a list of scheduled events, this approach maintains a list of conditional activities. As planned events are executed, the contingency tests of each activity is scanned. Unlike event scheduling and activity scanning, the process interaction method stresses the interaction between the entities of the system. Model component descriptions are amalgamated into units called *processes* rather than unstructured collections of unconnected events and activity routines. The behavior of a system is described by the flow of its processes through time. Users specify the behaviors of processes, while the system implicitly handles the detection and activation of events. This type of programming construct provides greater control over the actual structure of the system they are simulating. The languages GPSS[77] and SIMULA[6] use this approach to simulation modeling.

Unlike an event or activity, a process's routines are explicitly described in terms of time flow. A process's behavior may be interrupted at any point in time. It may be forced into an inoperative state when it comes into conflict with an another process or while it is awaiting the

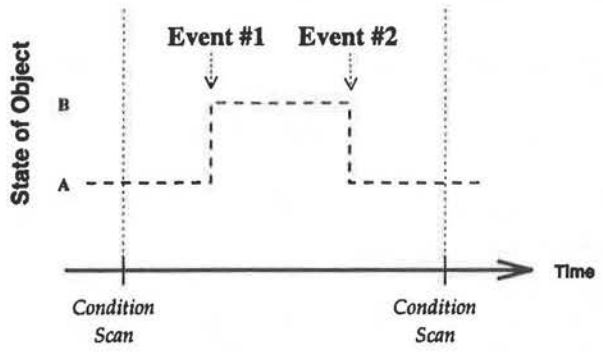

Figure 5.5: The scan does not note the change in the object's state

arrival of a future event. Multiple entry points, called *reactivation* points, within individual processes enables users to define a variety of conditions to reinstate the activity of any halted or delayed process. The process interaction methodology is structured as follows:

1. **Components:**

   The set of components, $D = \{\alpha_1, \alpha_2, \ldots, \alpha_N\}$, is divided into ACTIVE and PASSIVE types. ACTIVE components modify the behavior of the system. Their conduct is regulated by time or conditional predicates. PASSIVE types maintain their state values unless they are altered by external stimuli. Although they do not have the ability to directly control system execution, they can influence the behavior of ACTIVE components. An ACTIVE type's behavior may be suspended if it can not obtain essential information from an engaged PASSIVE component.

2. **Descriptive Variables:**

   Process interaction ACTIVE types are qualified by a set of features similar to those that characterize activity scanning ACTIVE types. Every component's state, time to activation, and sphere of influence is described by its descriptive variables. Unlike the next-event and activity scanning approaches, the process interaction method decomposes a component's state description into two distinct elements. An ACTIVE type's state is

defined by the values of its local variables and the status of its control instructions.

$$\text{STATE-OF-}\alpha = \text{CONTROL-OF-}\alpha \times \text{MEMORY-OF-}\alpha$$

	ACTIVE-α		PASSIVE-α	
	Range	Value	Range	Value
CONTROL-OF-α	L_α	$\{0, 1, 2, \ldots, M\}$	L_α	$\{0, 1, 2, \ldots, M\}$
MEMORY-OF-α	V_α	v_α	V_α	v_α
TIME-LEFT-IN-STATE-α	R_∞	σ_α		
INFLUENCEES-OF-α	D	$\{\beta_1, \beta_2, \ldots, \beta_M\}$		
INFLUENCERS-OF-α	D	$\{\overline{\beta}_1, \overline{\beta}_2, \ldots, \overline{\beta}_M\}$		

3. **Component Interaction:**

• In the activity scanning approach, the transition functions of ACTIVE components were characterized by two parts, the conditions predicate C_α and the action function f_α. In the process interaction approach, each part is broken down into several segments. Each segment is associated with a substate corresponding to the control state of the program implementing δ_α. Thus, C_α and f_α are decomposed into sets of $\{C_\alpha^l\}$ and $\{f_\alpha^l\}$. Employing this design, the transition function δ_α is expressed as follows:

Control State	
Control 0	If C_α^0 is TRUE $\rightarrow$ apply f_α^0 else nothing.
Control 1	If C_α^1 is TRUE $\rightarrow$ apply f_α^1 else nothing.
⋮	
Control M	If C_α^M is TRUE $\rightarrow$ apply f_α^M else nothing.

If α's control algorithm is in state 1, only the condition C_α^1 and f_α^1 are used to construct the local transition function. This design enables every ACTIVE component to dynamically alter its transition function according to its needs. Unlike the activity scanning approach, a component's transition function is not static.

• If should be noted that if α does not possess the ability to alter its own behavior ($\alpha \notin$ INFLUENCEES-OF-α), then every C_α^l will be equivalent to C_α and f_α^l will be identical to f_α for all $l \in L_\alpha$. However, for many scenarios this situation will not arise. Most α's will influence their own behavior. For these α's, their C_α^l and f_α^l are defined as

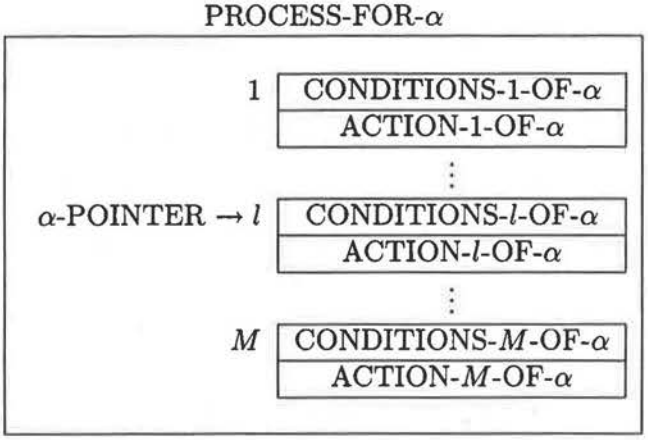

Figure 5.6: Process Definition

follows:

$$C_\alpha^l((v_\alpha,\sigma_\alpha),(l_{\overline{\beta}_2},v_{\overline{\beta}_2},\sigma_{\overline{\beta}_2}),\cdots,v_{\overline{\beta}_{\overline{m}}}) = C_\alpha(\overbrace{(l,v_\alpha,\sigma_\alpha)}^{SUBSTATE},\overbrace{(l_{\overline{\beta}_2},v_{\overline{\beta}_2},\sigma_{\overline{\beta}_2}),\cdots,v_{\overline{\beta}_{\overline{m}}}}^{INFLUENCERS})$$

$$f_\alpha^l((v_\alpha,\sigma_\alpha),\ldots,v_{\overline{\beta}_{\overline{m}}}) = f_\alpha((l,v_\alpha,\sigma_\alpha),\ldots,v_{\overline{\beta}_{\overline{m}}})$$

These two equations are derived by fixing the control component of α to value l in C_α.

Process Interaction Simulation

In a process-oriented simulation, every active component is represented as a *process*. A process is configured as a sequence of statements divided into M distinct segments. Each segment corresponds to one of the substates of the transition function δ_α. Every segment is composed of two divisions. Each division corresponds to one of the elements of $\{C_\alpha^l\}$ and $\{f_\alpha^l\}$. The diagram in Figure 5.6 portrays the basic structure of a process.

The first component of every segment represents the *activation* position of a process. The execution of a process always begins at an activation point. As the process's control algorithm sequentially progresses from segment to segment, it examines the conditional predicate of every division. If a segment's conditional predicate evaluates to true, its associated "action" routine is executed. Unlike the components of the previous strategies, processes retain their state during their periods on inactivity. A reactivated process continues processing from the point of last

abandonment. The temporary withdrawal of a process does not force the process to re-initialize its state.

The process-oriented simulator schedules the operation of all the processes in a simulation. Each process is given the opportunity to execute their actions within the given time frame. For a single processor simulator, this operation is accomplished with a quasi-parallel algorithm. The illusion of parallelism is imparted by a piecemeal execution of the statements associated with individual processes.

A typical process interaction simulation, as shown in Figure 5.7, employs a *future-activations* list and *current-activations* list. Storing triplets of the form $(\alpha, l_\alpha, t_\alpha)$, each list reflects the states and activation times of individual processes. Functionally equivalent to a *next-event* list, the future-activations list maintains a list of processes waiting to be activated. The forward progression of time is determined by the activation times of the members in this list. The current-activations list contains references to processes whose scheduled time has just arrived and to processes waiting for their activation conditions to become true. As active processes are activated, the conditional tests of suspended processes are examined. Although the current-activations list is functionally similar to the *conditions-list* of the activity scanning approach, each of the lists serves a slightly different purpose. Items in the current-activation list maintain references to running processes while members of a conditions-list represent processes waiting to be executed. Components waiting for future activation do not reside in the current-activations-list.

5.3 Processes Coordination

In a process-oriented simulation, the primary mechanism of computation is a *process*. A process is an "independent" program or procedure that uses the resources of a system to fulfill its goals. This approach to simulation facilitates the definition of parallel activities. After users define, describe, and initiate a collection of processes, the underlying simulation architecture controls their concurrent execution. The simplicity of this design places a greater burden on the simulation engine. Apart from ensuring the proper activation and deactivation of processes, the engine must protect individual processes from becoming *deadlocked*. A process is defined

Initialization	1	Set CLOCK to initial simulation time t_0
	2	Set $\overline{V}_\alpha$'s to the initial values v_α's of MEMORY-OF-α variables.
	3	For each ACTIVE α: $\rightarrow l_\alpha$ = initial value of CONTROL-OF-α $\rightarrow$ Place $(\alpha, l_\alpha, t_0 + \sigma_\alpha)$ in $\begin{cases} \text{FUTURE-ACTS-LIST} & \text{if } \sigma_\alpha > 0 \\ \text{CURRENT-ACTS-LIST} & \text{if } \sigma_\alpha \leq 0 \end{cases}$
Scanning Phase	4	Set SCAN to top of CURRENT-ACTS-LIST
	5	(a) Move SCAN down until the first α is found whose scanned triplet $(\alpha, l_\alpha, t_\alpha)$ returns TRUE when the CONDITIONS-OF-α is executed. Denote the winning α as $\overline{\alpha}$. (b) Remove $(\overline{\alpha}, l_{\overline{\alpha}}, t_{\overline{\alpha}})$ from the CURRENT-ACTS-LIST.
State Transition	6	Execute ACTION-$l_{\overline{\alpha}}$-OF-$\overline{\alpha}$ associated with activation point $l_{\overline{\alpha}}$): $\rightarrow$ execute $f_{\overline{\alpha}}^{l_{\overline{\alpha}}}$ for each $\beta \in$ INFLUENCEES-OF-$\overline{\alpha}$. The action function will yield new state $(v'_\beta, l'_\beta, \sigma'_\beta)$ for every β. $\rightarrow$ if β is an ACTIVE type, remove it from the FUTURE or the CURRENT-ACTS-LIST. $\rightarrow$ Insert $(\beta, l'_\beta, t_\beta + \sigma'_\beta)$ in $\begin{cases} \text{FUTURE-ACTS-LIST} & \text{if } \sigma'_\beta > 0 \\ \text{CURRENT-ACTS-LIST} & \text{if } \sigma'_\beta \leq 0 \end{cases}$
Scan Done?	7	If SCAN has not reached bottom of CURRENT-ACTS-LIST goto 5.
Time Advance	8	Advance the CLOCK to the time of the first triplet on the FUTURE-ACTS-LIST. Let NEXT-TIME-EVENT denote the new CLOCK value.
Update lists	9	Remove imminent activations from the FUTURE-ACTS-LIST and insert them into the CURRENT-ACTS-LIST.
Termination Test	10	If NEXT-TIME-EVENT exceeds termination time, then STOP! else goto 5.

Figure 5.7: Process Interaction

to be deadlocked if it is inactive and none of its reactivation conditions can ever become true.

Because a process-oriented simulation is decomposed into a collection of distinct events, users do not design simulations in terms of events. The design philosophies of an event-based and a process-based simulation are not equivalent. In an event-oriented simulation, users define events and event processing subroutines. The logic associated with every event is developed after the events have been created. In a process-oriented simulation, users define interacting processes. Focus is on the creation of entities and the descriptions of behaviors. Unlike the event-orient approach, it is not necessary to define the logic associated with processes. The simplicity of this approach to simulation is derived from pre-defined logic associated with every process-oriented statement.

In process-oriented simulations, processes can simulate the resources of a system or act as the active entities of a system. Processes of the former type are labeled as *resource* processes, while processes of the latter type are labeled *transaction* processes. Neither approach has a distinct advantage over the other. The nature of a simulation dictates which process type serves as a better tool for a given situation.

Process-oriented languages contain collections of mechanisms to coordinate the communication and synchronization of concurrent processes. These constructs facilitate the transferral of information from process to process and from process to system resource. In addition, these structures attempt to properly handle a variety of conflicts that may arise between processes. If a process requests the use of an already busy system resource, the structures must execute resolving actions. Typical responses include: the invocation of resource allocation schemes, process suspension strategies, and data locking tactics.

The remainder of this chapter examines three paradigms of process coordination. The first section covers traditional approaches, while the last two designs offer alternative proposals. Readers wishing to further explore coordination algorithms are directed to examine the plethora of articles found in the fields of concurrent systems[32], parallel languages, and simulation[4].

5.3.1 Common Schemes

Common process synchronization methods include Hoare monitors, Kessels monitors, Robert & Verjus control modules, and Campbell & Habermann path expressions. Each of these techniques may be combined in an hierarchical fashion to create elaborate synchronization schemes. A monitor is basically a shared data structure accessible by only one process at a time. Most monitors contain special constructs to manage waiting queues of processes. Control modules separate pure synchronization instructions from the description of the process and are mainly composed of sets of methods and synchronization rules. These rules dictate authorization for processes to execute particular methods. Path expressions also emphasize the separation between the scheduling of and functionality of operations.

5.3.2 LINDA

LINDA[10], developed by Nicholas Carriero and David Gelernter, is an approach to process creation and coordination that enables users to organize and control the execution of multiple threads. Utilizing tuple-space operations, this approach is based on generative communication. Data is never exchanged between processes through messages or shared variables. Instead, processes place and receive persistent data objects (called tuples) from a region called tuple space. There is no direct communication between individual processes. A tuple can be an inactive data object or a "live" computing process.

It is important to note that LINDA is not a language by itself. It is a conceptual model that must be embedded into a base "computing" language. Currently, LINDA has been successively combined with C, FORTRAN, and LISP.

5.3.3 MANIFOLD

MANIFOLD[3] is a co-ordination language. Its primary purpose is to describe and manage the complex interconnections between independent concurrent processes. Based upon a cheap lightweight process paradigm, MANIFOLD enables users to explicitly define the parallel execution of various computational modules. The interaction and communication of autonomous

active agents are controlled by addressless messages[9] and the activation of global flags. The language's primary focus is on how processes are dynamically interconnected during the lifetime of a system. The design of reusable components is enhanced by separating the communication issues from the functionality of the component modules in a concurrent system. This separation enables users to control the operations of co-operating processes at a high level of abstraction.

In MANIFOLD, processes are black box elements with sets of well-defined ports. Ports are joined via connections to facilitate the transferral of information from one process to another. Each process is oblivious to the identity of any other process with which it exchanges information. The sequential flow of information between process ports is represented by a stream. A stream can be dynamically constructed by the sender or receiver of information, or by any third party MANIFOLD process. The additive nature of stream definitions enable single ports to be simultaneously connected to many other ports. The flows of information in streams are replicated or merged at port junctions.

It is important to note that ports within MANIFOLD processes are separate structures. This enables ports to maintain input and output queues, and store tables of connectivity information. In addition, ports can be associated with filters that change, combine, and split units of information that pass through them.

In MANIFOLD, the primary mechanism of control is the event. Events are atomic pieces of information that define upcoming state changes in the system. As events are broadcast into the environment, individual processes select, interpret, and react to each event. It is important to note that all events are observed[10] asynchronously. Once an event is raised by an external source, the process generally continues with its own processing. Every event propagates through the environment independently from its source.

Unlike LINDA, MANIFOLD is a separate language for defining processes. It allows any process to directly influence the execution of other processes. Communication is not restricted to a single tuple-space environment.

[9] Addressless message are not directed to be sent to specific places. They roam freely.

[10] Because events are not directly sent to processes, they are observed, not received by processes.

5.4 SUMMARY

This chapter has provided a brief overview of several temporal management strategies. Each strategy was evaluated according to its method of temporal advancement and its implicit approach to world modeling. The temporal advancement methodology of a strategy dictates how the progression of time is regulated and how the activity of concurrent action is governed within a simulation. The world modeling approach of a strategy establishes the perspective from which users construct their simulations.

The table in figure 5.4 compares, in their "purest" form,[11] four of the most popular strategies in temporal management. Each strategy uses a different set of structures and primitives to provide users with an alternative approach to simulation modeling.

	Discrete-Time	*Event Scheduling*	*Activity Scanning*	*Process Interaction*
Primary	NA	Event	Activity	Process
Control Structure	NA	Event list	Conditions List	Process List
Temporal Jumps	Constant	Variable	Variable	Variable
Coordination Schemes	NA	None	None	Many
Concurrency Control	No	Yes	Yes	Yes
Ease of Use	Easy	Easy	Moderate	Complex
State Transition Types	None	None	None	None

Table 5.1: Temporal Management Methodologies

The discrete-time strategy is the simplest to use and easiest to implement. As time flows uniformly forward, users apply rules of interaction to the state variables of a simulation. However, the simplicity of this strategy contributes to its failure to handle complex scenarios. Users are not provided with structures to control the passage of time or to schedule the operations of state altering actions.

Discrete-event strategies vary in complexity: from the easy, event scheduling, to the complex, process interaction. The complexity of a strategy is directly linked to the world view it implicitly embodies. Since not one world view is intrinsically the best, no one discrete-event strategy

[11] The pure form is identified as the first historically attempted strategy of its type.

dominants. One world view can neither naturally express nor efficiently process all the forms of model description. Unlike the discrete-time strategy, discrete-event strategies have the ability to manipulate the passage of time and to schedule the creation and deletion of state altering actions. Time may advance non-uniformly and simultaneous actions may occur.

Although temporal management strategies provide users with an attractive base, they do not guarantee the production of reusable simulations. Rules of interaction and simulation structures are only supplied to control the passage of time and to schedule the activation of state changes. Users are not provided with any structures to facilitate the definition of state transitions. This shortcoming, which compels users to define their own types of state changes, limits the reusability of a simulation. Distinct simulations which utilize dissimilar structures to alter the values of state variables are difficult to combine.

An ideal reusable temporal management strategy would provide the following benefits:

- **State Transition Times**: A set of methods to assist users in specifying the execution times, durations, and frequencies of state changes.

- **State Transition Types**: A set of extensible structures that define the type and variety of state transitions which can occur in a simulation.

- **Multiple World Views**: Simulation languages and systems which embody multiple world views allow users to construct simulations from multiple points of view. Users intermix the various approaches to world modeling to create simulations which naturally express and efficiently process their designs.

- **Coordination Schemes**: A set of rules and structures to assist users in coordinating the actions of concurrent operations. The co-ordination rules will manage the complex interconnections between simulation models and regulate the transfer of data between them.

CHAPTER 6

RASP: THE DESIGN GOALS

'The time has come,' the Walrus said,
'To talk of many things:
Of shoes - and ships - and sealing-wax -
Of cabbages - and kings -
And why the sea is boiling hot -
And whether pigs have wings.'
- Lewis Carroll, Through the Looking-Glass, Ch. 4

It is the aim of the RASP toolkit to provide computer graphics researchers, simulationists, and robotic scientists with a common set of tools to build applications within their respective domains. Providing these users with basic structures enhances their ability to reuse components and ideas from previously defined applications. Components of applications from multiple domains can also be freely exchanged without sizeable modifications promoting the development of new and innovative simulations. Users are not forced to build from scratch each time a new application is constructed.

This chapter discusses the four major goals embraced by the RASP toolkit. RASP solutions to each of these goals are described in subsequent chapters.

6.1 SIMULATION FRAMEWORK

6.1.1 RULES OF INTERACTION

A set of tools is ineffectual unless accompanied by a set of rules of interaction. Labeled as a *framework* (see section B.2.3), these rules abstractly define how users organize relationships between the various components of their simulations. The precepts of a framework define how the state and behavioral patterns of simulation components are regulated.

6.1.2 Decomposability

A framework is extremely useful when decomposable. A partitioned framework enables users to control effectively the overall design of their simulations. An organized separation of a framework provides users with a standard set of sub-goals. Each sub-goal is used as a measuring device for the individual segments of an application. Any segment that fails to meet the requirements of a sub-goal fails the regulations of the complete framework.

6.1.3 Communication Architecture

The communication pathways between framework components establish the behavior of an object-based system. The transfer of information from one object to another defines the performance and capabilities of a system. A favorable communication architecture promotes the design of reusable components, endorses the formation of dynamic connections (links), and facilitates the creation of objects directly involved in the communication process.

6.2 Multiple Temporal Strategies

The temporal management strategy employed by a simulation system dictates the point of view from which users see the system they are modeling. It influences greatly the structure and manner in which models and their interactions are specified. The inability of individual strategies to express naturally and process efficiently all forms of model description hampers the construction of simulations which are generally easy to manipulate, decipher, and reuse. A better foundation of simulation development supports the use of a wide variety of formalisms and multiple temporal management strategies[97]. Users select the strategy that provides the most flexibility to their modeling needs and enables them to construct simulations which closely emulate the behaviors of real-world systems. Recent trends which have seen the emergence of simulation languages and systems which employ multiple strategies reveal the importance of such a design.

6.3 Time and State

6.3.1 Definitional Uniformity

A basic set of definitions is needed to distinguish the time and state relationship. A clear distinction is critical to the design of reusable simulations and the realization of simulation foundations. A muddled understanding impedes the communication between model developers and model users, contributes to cost overruns, and aggravates model disutility[13]. A definitional uniformity is necessary to clarify simulation concepts, unify simulation structures, and facilitate the portability of models[57].

6.3.2 Hierarchical Temporal Modeling Tools

The behavior of a time-varying simulation is guided by the nature of its state changes and the techniques employed to regulate the progression of time. Therefore, it is important for the toolkit to provide an extensible set of temporal modeling tools which standardizes the specification of state changes and the employment of temporal management methodologies. A standardization adhering to the connection paradigm and conforming to a uniform set of definitions promotes the creation of simulations that are easy to interpret, alter, and reuse.

The need to alter rapidly the behavioral specifications of a simulation at various levels of detail requires a natural hierarchical relationship to exist among the set of temporal modeling tools. Tools at the highest level administer global changes while those at the lowest level administer local changes. A natural hierarchy contributes to stepwise modeling refinement and program readability.

6.3.3 Temporal Granuality

A simulation is heavily influenced by the granularity of time between system events. In many cases, the magnitude of the temporal step size can effect the accuracy of a model. Large intervals reduce the precision of many computations. To reduce this problem, many systems let users set the size of the maximum interval. Although this approach reduces the possibility of error, it can introduce one major side effect. Unless proper structures are defined, it will confine all

the operations of a simulation to a single minimum temporal interval. This scheme is extremely inefficient for parallel computations soliciting disproportionate step sizes. An improved design allows dissimilar computations to operate at disparate step sizes. Although this enhanced model burdens the system with the task of organizing unevenly computing operations, it affords users with the ability to define optimized simulations.

6.3.4 Minimal Kernel

At the heart of every computer simulation resides a *simulation kernel.* The driving engine of a simulation, the kernel employs a temporal advancement methodology (see Chapter 5) to control the progression of time, to ensure that every module in a system is aware of the global state, to manage the execution of concurrent activities, and to coordinate the activation of simultaneous actions. A kernel's design profoundly affects the operation and structure of a simulation. To function properly, all the components of a system must adhere to a single organizational pattern. Apart from a possible reduction in a kernel's efficiency, multiple formulations can induce undesirable behaviors.

A kernel's performance and understandability is enhanced when its collection of responsibilities are confined to a small set. Kernels serving many roles are frequently difficult to manage and reuse. Although many designers are aware of this fact, it is not uncommon for them to add an assortment of miscellaneous operations within their simulation kernels. A system's readability and tractability is routinely swapped for system optimizations. For example, some kernels are directly responsible for controlling the interaction between user-interface devices and the models of a simulation. Although these additions may enhance a system's outward performance, this scheme hampers a system's versatility. Operations and functions that are incorporated into a kernel are usually difficult to modify or control.

The diagrams in figure 6.1 illustrate the difference between a minimal and expanded kernel design. In the minimal model, the kernel controls only the elements of the system. Each element governs a distinct duty. In the expanded model, the kernel fulfills all the expectations of the minimal model plus more. This design impedes a system's versatility because it does not provide users with methods to alter the behavior of the kernel's additional operations. Although it is

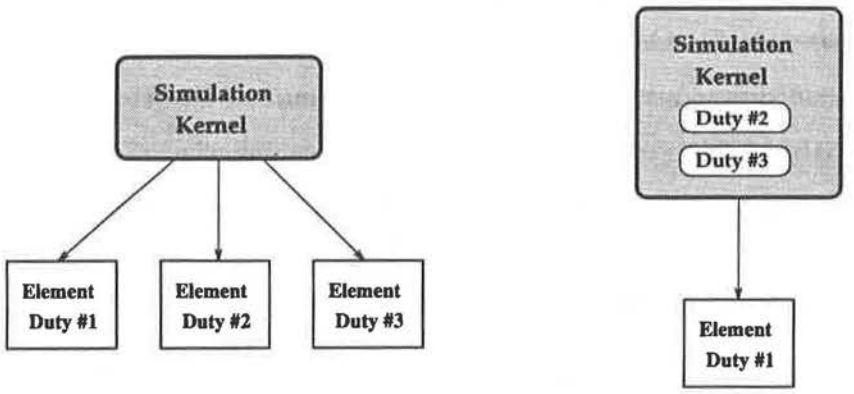

Figure 6.1: Simulation Kernel Designs: Items within the grey boxes are internal to the kernels and inaccessible to simulation components.

possible to define constructs to control the extra behaviors, this is not recommended because of two unfavorable side effects. First, unpredictable configurations may corrupt the operation of the kernel, causing the entire system to crash or generate inaccurate solutions. Second, a kernel exhibiting multiple behaviors will induce additional constraints on a users' designs. Users must ensure that their designs accommodate various kernel configurations.

6.4 Geometric Model Construction

6.4.1 Model Creation Methodology

The intrinsic designs of the models in a simulation impose a strict set of constraints on the specification of state changes and the transmission of information between system components. Similarly, the external interfaces employed by models restrict the range of operations supportable by the models. Therefore, it is necessary to establish a standard approach to model creation which promotes a manageable and extensible internal architecture and a versatile external interface.

To permit alterations of model traits at various levels of detail, the internal architecture must support a natural hierarchical arrangement of information. Major changes are induced by

altering information at top of the hierarchy while minor ones are induced by altering information near the lower end of the hierarchy.

To enable models to respond to state changes during runtime, the model's external interface must permit the toolkit's temporal modeling tools to access the value of internal state variable and to alter the relationships formed between models and their environments.

6.4.2 Rendering Supportive Architecture

Graphical views of (geometric) models in action are valuable elements of many simulations, especially computer animations. They ease the verification and augment the validity of simulation designs. Therefore, there is a great need to construct models and image synthesizers (renderers) which facilitate the translation of model descriptions into visual images. In addition to the design goals previously mentioned, a model's internal architecture must support readily the interaction between models and renderers. However, this interaction must not lead to the development of models and renderers which strongly depend upon each other. A clear division must be established to promote independent and reusable design.

6.4.3 Complete Control

The design of a powerful toolkit enables users to control dynamically every attribute of a model. Providing users with the authority to manipulate many of the individual features of a model enhances their ability to create complex systems. The value of every state variable associated with a model must be alterable. After state variables have been initialized, operators should be available to revise their values during the lifetime of the simulation.

CHAPTER 7

RASP: THE FRAMEWORK

Only connect!
- Edward Morgan Forster, Howards End, Epigraph

Recent trends in the field of simulation have demonstrated the necessity to divide the simulation modeling process into distinct components (see section 4.4). Adhering to a well-defined framework, a clear division enables the development of the components of a simulation one at a time. After all the individual components have been developed, they can be assembled together to form an operative simulation.

This chapter describes RASP's simulation framework. Discussion entails an examination of the *patterns of change*, a description of the *IMVCD Pentad*, and a definition of the *connection paradigm.*

7.1 PATTERNS OF CHANGE

The essence of every temporal simulation or computer animation involves the continual evolution of state variables through time. As time progresses forward, a pre-defined set of laws or rules modify state variables. These "controlling" rules may influence the literal values of these state variables in one of two ways: *explicitly* or *implicitly.* Explicit rules precisely dictate the values for state variables while implicit rules define the behavioral response pattern for state variables. For example (see Figure 7.1), if a rule informs an object to move to a particular position, an explicit rule determines the object's position. However, if the object is told to move away from its closest associate, an implicit rule influences the object's position. Implicit rule are commonly used in activity scanning simulations. For example, in the animation system developed by Kalra[40], state variables are controlled by behavioral rules. Whenever a specific rule is triggered, it alters the state of the system. In either case, if a state variable changes value, it is always possible to deduce some motivating factor that is forcing it to change.

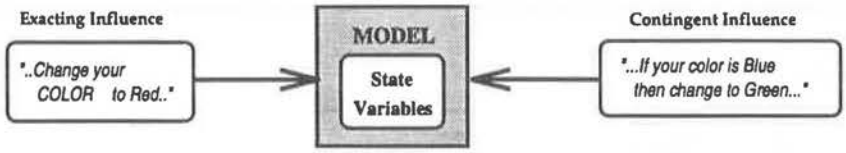

Figure 7.1: Explicit vs. Implicit Rule

Controlling rules can be also classified according to their association to the variables of influence and the ways in which they define a change to a variable. If embedded into the design of a specific variable, then the controlling influence of change is defined to be *internal.* However, if an influence is not an integral part of a variable, it is labeled to be *external.* In contrast to external influences, the former tightly bind variables to sets of behaviors. The same rules of conduct always govern a variable's reaction to foreign stimuli. A variable's behavior is usually unalterable. External influences do not elicit unalterable behaviors because they regulate a variable's behavior from afar. This loose binding enables users to alter a variable's behavior by simply changing its controlling influence. In an object-based environment, one may envision an internal motivation as a sub-unit of one monolithic entity while an external influence is an entirely separate entity. Figure 7.2 illustrates this concept.

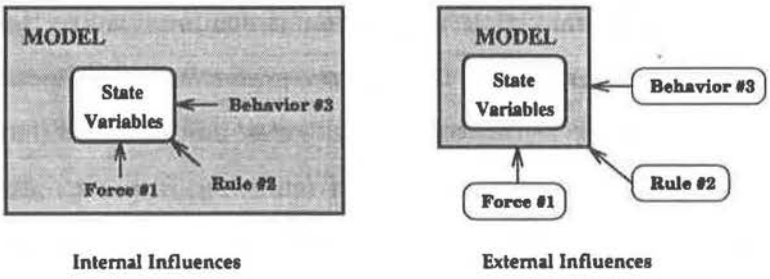

Figure 7.2: Internal vs. External Rule

Examples of internal rule are commonly found in many animation systems which support "animated basic types". In ASAS[72] and MIRA[52], special variables are defined that automatically update themselves according to an "evolutionary law". Although ideal for some

situations, this is not a suitable control mechanism. Directing an animated type to follow dynamically a variety of evolutionary laws can be difficult or impossible. The inability to remove or modify an internal rule from any variable type hampers the variable's reusability.

External rules provide a better device for general control than internal rules because they separate a variable from its behavioral patterns. External rules enable users to build independent models of change without directly influencing the variables they are manipulating. Exemplary usage of external rules is found in the PINOCCHIO animation system[54] and in the motion objects of Fiume's temporal system[21]. In both works, external controlling rules are associated with state variables to produce sequences of animations.

7.2 The I-M-V-C-D Pentad

RASP's simulation framework, known as IMVCD (Informer-Model-Viewer-Controller-Delegator), is based upon the development and usage of external controlling rules. Individual models are not allowed to make changes to their own state. Only external controllers can induce modifications to the state variables of a model. Therefore, it is duty of the models of a simulation to interpret and execute changes issued by controllers.

The IMVCD framework is divided into five abstract components, each representing a different aspect of the simulation modeling process. Dividing the process into separate sections promotes modular design and enhances the reusability of RASP-built applications. IMVCD's object-oriented design provides users with a common architecture and organizational plan to build their applications. IMVCD's components, as shown in figure 7.3, include the following:

- **Informer**: These elements define the physical traits and immaterial characteristics of various models in a simulation. Informational traits are affixed to a common substructure to create complex models. Constituents of this group control a model's shape, material attributes, associations, properties, and qualities. For example, Figure 7.4 shows a model constructed with four **Informer** objects.

- **Model**: All application objects are representative elements of this grouping. Models define the physical or active elements of a simulation. External rules place constraints on

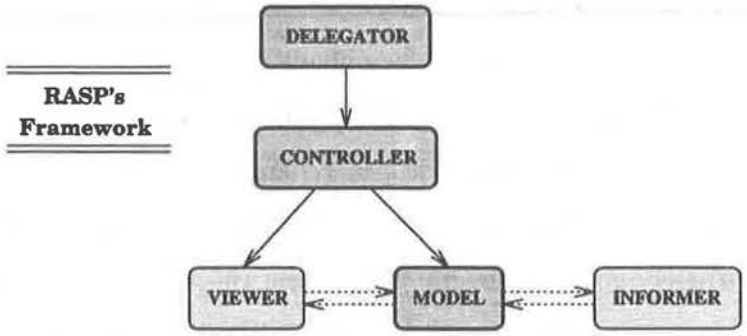

Figure 7.3: The IMVCD Framework

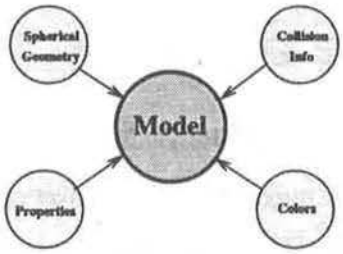

Figure 7.4: Model with Informers

a model's interface to control its interaction with other models. Models are not directly responsible for their own behavior. Every model must organize sets of **Informer** elements to regulate its appearance and to react to external influences. A model interprets its externally situated constraints to control the flow of information between the external environment and its **Informer** objects.

- **Viewer**: Responsible for the production of static images, these objects interpret data from the physical models of the systems to form visual displays of a simulation.

- **Controller**: The foundation of RASP's bi-level hierarchy of simulation control, these objects are "indirectly" responsible for modifications to the attributes and state variables of all the models. Controllers do not form direct links with the variables they are altering. Rather, they establish constraints on the links that bond the controller and an object's interface to induce modifications. This form of *external* influence promotes the design of

Figure 7.5: From Model to Viewer to Image

"independent" controlling objects. A controller induces a change to its interface, not to its controlling objects. State modifications are produced by the propagation of changes from a controller's interface to a model's interface. The nature of this scheme permits users to construct *non-adaptive* and *adaptive* influential links for complex simulations.

- **Delegator**: Members from this class of objects serve to control the interactions between the **Controllers** and **Models** in a time-varying simulation. Forming the upper layer of RASP's bi-level hierarchy, these objects administer the linking of component interfaces. They delegate to individual **Controllers** "when" and "how" they are to direct the behaviors of the models in a simulation.

The communication pathways established by the interactions of these five components constitute an integral element of RASP's IMVCD framework. A strict set of rules for data transmission regulates the flow of information between components. In many cases, data is examined by a variety of objects called *ports*. Ports ensure that legitimate data is being sent from component to component. In addition, some ports can signal the occurrence of state changes. Ports are discussed in section 7.3.3.

7.3 Connection Paradigm

The IMVCD framework communication architecture is based upon the *connection paradigm*[55]. Based upon first-class links and first-class interfaces, the connection paradigm structures a system as a set of components interconnected through unidirectional ports. Component behaviors are specified with respect to their ports, not their bindings with other components. Bindings between components are formed by elements not directly influenced by the data transferal

process. The separation of binding information from behavioral specifications promotes the development of reusable components.

The following three subsection describe the advantages and minor drawbacks of systems that adhere to the connection paradigm. Discussion entails the benefits of indirect communication, first-class links, and first-class interfaces.

7.3.1 Direct vs. Indirect Communication

Communication techniques are distinguished according to their means of establishing data links. If a system relegates the duty to the components of the system, it uses a "direct" approach. If the components, such as those following the connection paradigm, are not directly involved in the establishment of their data paths, the system uses an "indirect" approach.

In the direct approach, all the components of a system define their own links. They ensure that all the communication protocols between themselves and their partners are correct. Although simple and easy to implement, this plan has several drawbacks. First, every component must have explicit knowledge of its partner's identity and interface if it wishes to exchange successfully information. Second, it can be very difficult to alter an component's partner. Unless the new partner has an identical interface to the previous partner, a change may require a modification to the repertoire between the two components. Figure 7.6a illustrates the direct approach to component communication.

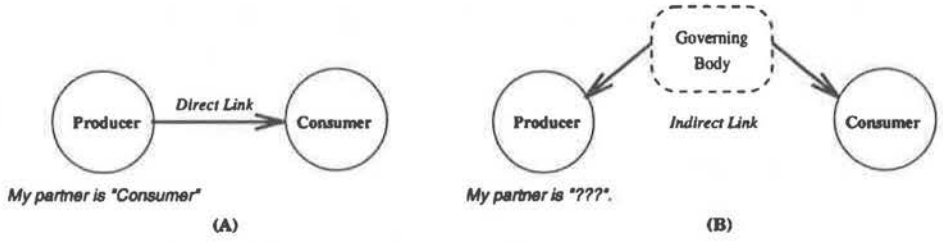

Figure 7.6: Direct vs. Indirect Communication

The indirect approach transcends these drawbacks. In the indirect approach, an external governing body establishes all the links between the various components of a system. The

governing body obtains datum from a "producer" component and delivers it to a "consumer" component. This process is illustrated in Figure 7.6b. Individual component do not generate their own data links. They are always oblivious to the identity of their partners. Although the additional level of indirectness created by this approach is not as efficient as a direct communicational link, it affords three important qualities to reusable simulation design.

- Indirect communication enhances the independent construction of complex components. Dependencies are extracted from all components. This differs from the direct approach which may unnecessarily force users to incorporate dependency information into object designs.

- Indirect connections provide an attractive base for the maintenance of temporally dependent informational pathways because dynamic links are formed and destroyed by external sources. This design enables objects to be constructed without any time-varying constructs. A component's dynamic behavior is not controlled by the component itself.

- Multiplex links, as seen in Figure 7.7, are easy to form. Many paths may meet at or originate from a single source. Sources and consumers involved in multiplex links are not required to manage all their connections. For example, a source with multiple links is not required to send repeatedly one piece of datum to multiple objects.

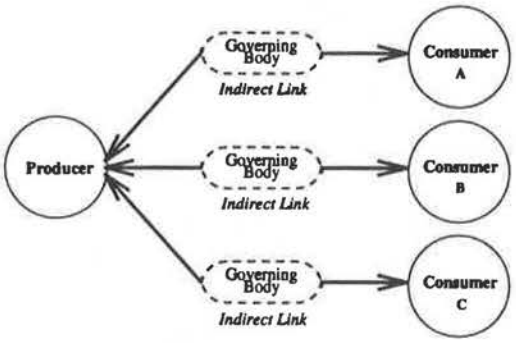

Figure 7.7: Multiplex Connection

7.3.2 First-Class Links

The *indirect* approach to component communication requires an external source to define the data pathways between individual components. To accomplish this task, the governing source may establish *passive* or *active* links. With passive links, the external source must ensure that proper bonds are formed between compatible components. If a consumer component requires a double-precision value, it should not be linked to a producer component that generates only "string" values. Unlike *passive* links, *active* links are first-class objects. They are well-defined entities with structure. Performing as moderators between components, they ensure the transferal of identical "types" of information from one entity to another. In addition to "type-checking", *active* links can create couplings between incongruous objects. After receiving data from a source component, the link can filter or modify the data to an appropriate form.

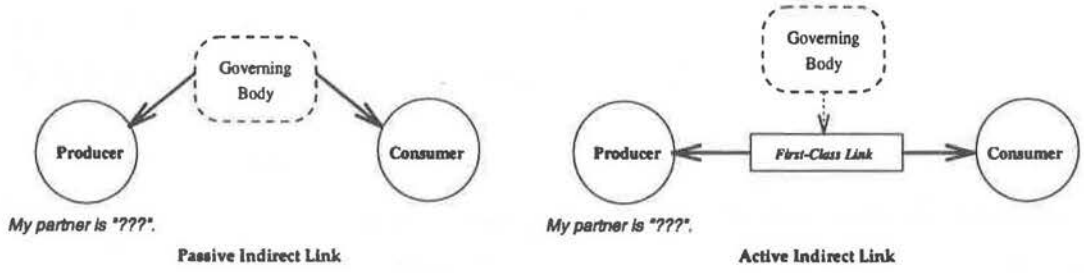

Figure 7.8: Indirect Link Types

The diagram in Figure 7.8 and the following set of pseudo-code illustrate the difference between the two approaches of component linking.

```
procedure passive_indirect_foo( obj producer, obj consumer )
{
    a = producer.get();             /* get data from source */
    if (type(a) == type(consumer)) /* type-check data with consumer */
        consumer.send(a);           /* pass data to consumer */
}

procedure active_indirect_foo( obj producer, obj consumer ) {
    link bar( producer, consumer ); /* create indirect link */
    bar.execute()                    /* pass info along link */
}
```

7.3.3 First-Class Interface

The structure and definition of *passive* or *active* links in any system depends upon the interfaces of the components to be joined. Each interface asserts the type and quantity of information produced or consumed by a component. In addition, the interface specifies all the prerequisites for a component. In common object-oriented languages, such as C++, an component's interface consists of a collection of *second-class* procedures. These routines, called member functions, are the interface to a component's internal assembly. Passive elements of a component's framework, they serve to ensure the proper transferal of information from the source to the environment, and vice-versa. Because they do not retain state or react to messages, member functions do not usually manage other types of behaviors or functions.

Although prevalent in object-oriented languages this type of component interface provides limited support for simulation models based on links. The inability of links to obtain information, such as data requirements or state changes, from member functions hampers their flexibility. An interface composed of first-class objects provides a superior foundation for linking-based systems. Behaving as member functions, these objects, called *ports*, provide users with powerful mechanism to control the flow of information "in" and "out" of an object. Although ports are similar to member functions, they are not easily interchangeable. Their differing designs and capabilities induce alternative approach to simulation design.

Member functions and ports contrast in the following ways:

	Member Functions	Ports
Direct access to data members?	Yes	No
Dataflow	Bidirectional	Unidirectional
Can be Queried?	No	Yes
Component composition?	No	Yes
Attach Information?	No	Yes
Ease of development	Easy	Moderate

Table 7.2: Member functions vs. Ports

- All ports are unidirectional. They are defined as either "in" or "out", but not both. They can not alter their directional behavior at any time.

- The arity of port routines is at most one. This prohibits all "out" ports from defining a list of arguments. Parameter values must be set by an associated list of "in" ports. For example, the following member function can not be expressed by a single "out" port.

```
int PROCEDURE poof( argument1 );  /* get arg, run poof & return data */
```

The procedure `poof` must be partitioned into two separate "port" functions as follows:

```
InPort  arg1( argument1 );        /* get argument and store for poof */
OutPort poof();                   /* run poof and return data */
```

Although appearing to be a major drawback, this constraint actually enhances the design of independent components. Functions that simultaneously edit (consume data) and query (produce data) an component's state are not encapsulated into one command. The separation of functions enables multiple links to query concurrently the state of the component whenever a change occurs. A single function that performs editing and querying operations may delay the activation of vital "state" monitoring actions.[1]

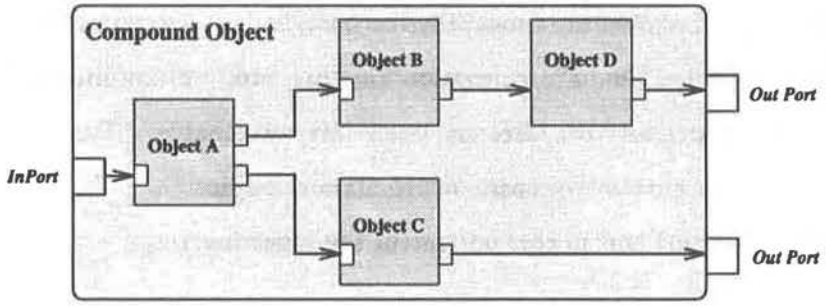

Figure 7.9: Amalgamated Component

- A port-based system aids the construction of large "black-box" components from numerous smaller components. Users simply link the "out" ports of components to the "in" ports of other components to create amalgamated components. Diagram 7.9 depicts an example construction. Without a pre-defined organizational plan, the creation of amalgamated components is not possible in member function-based systems.

[1] A state monitoring action is a special function that is immediately triggered whenever a component reaches a particular state.

- Ports can accept queries and additional arguments not directly related to its purpose. For example, a port can be informed to invoke a specific operation when it receives new information or observes changes in its associated component's state.
- Ports are generally harder to implement than standard member functions. Users must define additional parameters and links to ensure their proper usage. However, this difficulty may vanish with future language support.

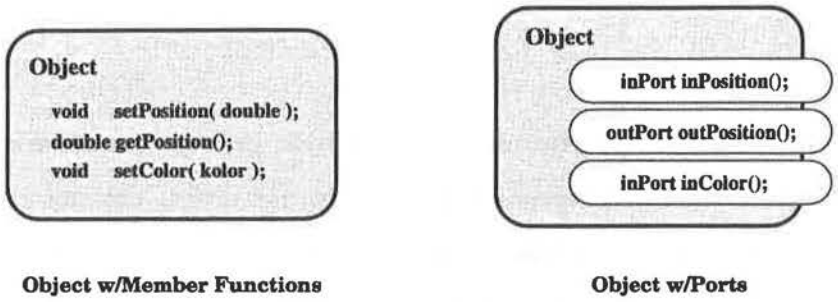

Figure 7.10: Members vs. Ports

The diagram in Figure 7.10 illustrates the difference between a component created with standard member functions and an component created with ports.

7.3.4 Connections vs. Dataflow

The first-class links formed by the connection paradigm are similar to the links found in dataflow networks[30, 15]. Each interconnects the components via port-like structures to form a working system. However, several key differences exist between the connection paradigm and dataflow architecture. First, the nodes in a dataflow network usually perform an operation roughly equivalent to an assembly instruction while nodes in a connection paradigm system can have arbitrary computational power. Second, the topology of a dataflow network is static while that of a connection paradigm system is dynamic. State transition can reorganize the networks of connection paradigm systems at run time. This advantageous scheme fosters the development of complex systems which closely resemble the real-world systems they model.

7.4 IMVCD vs. MVC

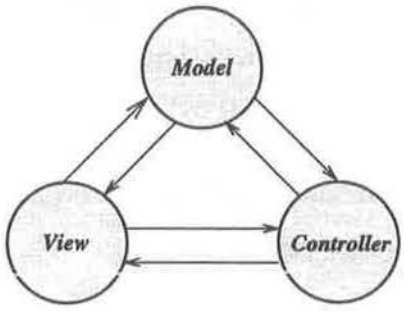

Figure 7.11: MVC Framework

The IMVCD and MVC[45] (see figure 7.11) object-oriented frameworks share many common traits. Each architecture decomposes an application into several abstract components and each establishes communication pathways between the individual components. In some ways, IMVCD may be viewed as an augmented or enhanced MVC framework. However, IMVCD differs from MVC in many other ways. IMVCD defines an alternate modularization of a system. Some of IMVCD's modules generalize certain MVC components, while others partition them. IMVCD incorporates temporal information and defines a communication protocol between components. The difference between the two frameworks can be contributed to their differing design goals. MVC supplies an architecture to design reusable user-interface application while IMVCD supplies an architecture to design reusable time-varying simulations.

Models

MVC *models* and IMVCD **Models** are similar in that both represent the active entities of a simulation, and both are distinguished by their attribute traits. However, each component differs in the way it is constructed. In MVC, *models* are constructed to follow only a rudimentary set of modeling constraints. Users are free to build all the models within a simulation differently. In IMVCD, **Models** are constructed solely to organize sets of **Informers** objects. No model is permitted to deviate from this design.

IMVCD enforces its design methodology by requiring all user-defined **Models** to inherit properties and procedures from "model templates". Templates (OOP base classes) define basic operations, which **Models** (subclasses) use to organize **Informers** and to control the exchange of information between **Informers** and the environment.

MVC does not support model templates[2] because it would be detrimental to its framework. It is MVC's belief that "..the model is completely application-dependent and must therefore be implemented by the application programmer."[92] Although this method of model development can be viewed as an acceptable manufacturing technique, it does not promote the design of reusable models. The lack of a structured design methodology endorses the creation of a wide variety of non-reusable model configurations.

Views

MVC *views* and IMVCD **Viewers** are similar in that both objects produce visual representations of the models in a simulation. However, each differs in the number of duties it performs. MVC *views* embrace many responsibilities. They interpret information from models, form links between controllers and models, and react to the changes occurring in a simulation. The problem with this design is that it promotes the creation of *views* of monolithic size and it requires *views* to have direct knowledge of the models they are manipulating. These two impediments seriously undermine an object's reusable potential.

IMVCD **Viewers** avoid these difficulties because they embrace only one responsibility. They interpret information obtained from the models of a simulation to generate synthetic images. Additional tasks associated with MVC *views* are distributed to other components. For example, link formation duties are relegated to **Delegators**, while observation tasks are dispatched to **Controllers** and **Models**.

Controllers

IMVCD **Controllers** differ from those of MVC's in that the former are "general" manipulators, are oblivious to the identity of the models they control and the operations which the models

[2]MVC supports templates for its *views* and *controllers*, but not for its *models*.

provide, and are not required to inherit properties and data from an abstract controller (class). These three properties enhance their reusability:

Controllers are general because they influence all the states and behaviors exhibited by models (or other controllers). They are not restricted to handle only one responsibility, such as the one delegated to MVC *controllers* - to manage the interaction between models and user-interface devices.

Controllers are oblivious to the identity of the objects govern because they do not actually interact directly with them. All modifications are relayed from **Controllers** to models via **Delegators** and ports. This level of indirection, not found in the relationship between MVC *controllers* and *models*, promotes the construction of controllers which are not constrained to interact with only a few types of models.

Unlike MVC, IMVCD does not require controllers to inherit information from one abstract class. The expanded functionality of IMVCD **Controllers** restricts the creation of such a class. One abstract class would hamper the construction of a wide variety of controller types. However, it should be noted that this latitude may change in the future. Abstract or base classes may need to be created to promote greater reusability.

CHAPTER 8

RASP: DISCRETE-EVENT MODELING

To choose time is to save time
- Francis Bacon, 1st Baron Verulam, Viscount St. Albans
Essays, "Of Dispatch"

Many computer animation systems and robotic applications are based upon a *discrete* time approach. The variables of the system change discretely at specific times. Although easy to implement and use, this approach does not provide a robust foundation for time-varying simulations. The lack of event scheduling constructs limits the user's ability to control the passage of time. In addition, the approach's inability to describe the execution of concurrent actions precludes it from coordinating the parallel nature of many simulations. The discrete-event approach provides a better foundation for building time-varying simulations because its variable time step philosophy permits the definition of future actions and events are not constrained to occur at pre-defined intervals.

Temporal systems based upon the discrete-event philosophy can be separated into three world views: **event scheduling**, **activity scanning**, and **process-interaction** (see section 5.2). Events can be explicitly pre-defined, activated by environmental factors, or initiated by running processes. Historically, most simulation systems supported only one world view. However, recent trends have seen the emergence of simulation systems which support multiple world views. For instance, SIMSCRIPT II.5[42], SLAM[71], and SIMAN[67] offer users the choice to create simulations using events and processes.

This chapter introduces RASP's technique to simulation modeling. Labeled as a multiple interface approach, this scheme enables users to design simulations through any of three world views in union or separately.

8.1 Multiple Interface Approach

All three views toward discrete-event modeling are based upon the creation and activation of events. Although each view supports a different design philosophy, their common foundation suggests the development of an approach encompassing all three. A *multiple interface approach* to discrete-event modeling enables users to develop simulation systems with pre-defined events, conditional events, and processes.

It is important to distinguish the difference between a multiple interface approach and a "combined" approach. The process-interaction approach to discrete-event modeling is a combined approach.[1] It enables users to create pre-defined events and conditional activities. However, it is not a multiple interface approach. Users are relegated to define a process interaction simulation with only processes. All events and conditional-events are defined in terms of processes. Although one may view this approach to modeling as a positive quality, it does not facilitate the creation of simple events and conditional activities. A multiple interface approach enables users to create events, activities, or processes using an event, activity, or process interface.

8.2 Activity Scanning Modeling

An aggregate interface approach to modeling places a heavy burden on a simulation system. As events, conditional activities, and processes are defined (using three different interfaces), the system must translate them into one unified event-based framework. This task is simplified by shifting some of the event handling responsibility from the central processor (of the simulation loop) to the data components of the system. Aside from reducing the quantity of computations performed by the system kernel, this shift improves the maintainability of a model. It is usually easier for users to maintain a system composed of components with minimal responsibilities.

The activity scanning approach to discrete-event modeling necessitates the inspection of contingency tests for conditional actions. Because the activity scanning approach dispatches

[1] Although any simulation approach that supports continuous and discrete time can be defined as a combined approach, this is not the definition proposed in this chapter.

this service to the simulation kernel, the kernel must maintain a dynamically ordered *conditions-list.* After the completion of every event, the kernel scans the list to determine if any activity's condition has been satisfied. Apart from imposing an additional burden on the simulation kernel, this approach introduces redundant operations. The continual review of unvarying conditions is inefficient. If the elements of a condition do not change their state, it should not be necessary to evaluate the condition's state. An alternative approach to activity scanning transfers the responsibility of contingency condition testing from the simulation kernel to the state variables of the system. As variables have their state's altered, they inform all conditional tests of their new values. This scheme initiates contingency testing only when it is required. Querying operations are performed only after relevant state changes occur.

Although this alternative scheme has a higher computational overhead, it affords three additional beneficial qualities. First, it eliminates the need for a conditions-list. The simulation kernel is freed from continually performing tests of conditional activation. Second, it promotes the design of reusable first-class conditionals. Developed as an object with "ports", a first-class condition possesses state and is viewed as a simulating entity. States variables are linked with conditionals during the progression of a simulation. A conditional's predicate is evaluated each time it receives a new value from one of its external linkages. Third, it contributes to a modular approach to simulation development. It coerces users to balance the distribution of conditional tests throughout a system.

8.2.1 First-Class Conditionals

In RASP, conditional predicates are constructed as first-class entities with sets of "input" and "output" ports. Input ports values are evaluated by internally-defined predicates to produce values for output ports. The values of output ports are altered each time a conditional object perceives a change in its input ports. Adhering to the connection paradigm, this scheme enables users to construct temporally dependent conditional tests. One conditional object can be used to evaluate the states of several variables at specific times. For example, the diagram in Figure 8.1 illustrates two configurations of the same conditional entity during different stages of a simulation. At time t_1, the states of objects A and B are being tested, while at time t_2, the

values of objects A and C are being evaluated. It is important to note that the conditional is never aware of the variables supplying its input ports with state values. Its behavior is strictly dependent on the values of its "input" ports.

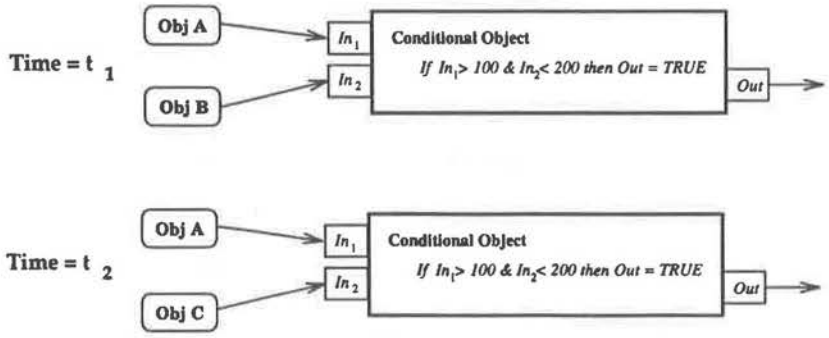

Figure 8.1: First Class Conditional

An activity scanning action is constructed in three steps. First, a first-class conditional with n number of input ports is declared. The cardinality of the input ports depends upon the arity[2] of the conditional's predicate. Second, all state variables involved in the contingency test are informed to monitor their states. Any noted state modifications are transmitted to the conditional. In the last step, the conditional observes the value of its predicate test. If a state change is perceived, the conditional immediately schedule the activation of an event or activity. The following segment of pseudo-code illustrates the commands users would issue to define the configuration in Figure 8.1.

```
FROM t={0,100}    if (object A changes state) then
                     ''pass values from A and B to CONDITIONAL''.

FROM t={0,100}    if (object B changes state) then
                     ''pass values from A and B to CONDITIONAL''.

FROM t={101,200}  if (object A changes state) then
                     ''pass values from A and C to CONDITIONAL''.

FROM t={101,200}  if (object B changes state) then
                     ''pass values from A and C to CONDITIONAL''.

FROM t={0,200}    if (CONDITIONAL predicate changes state) then
                     ''schedule new ACTIVITY''
```

[2]Computer terminology defining the number of arguments an expression requires.

The first two statements apply to the period from $t = 0$ to $t = 100$. During this temporal interval, the values of objects A and B are observed. The following two statements alter the observed variable list to objects A and C. This observation period is established to occur from $t = 100$ to $t = 200$. Sustained from $t = 0$ to $t = 200$, the last statement orders the activation of a new activity if the conditional's predicate evaluates as true.

8.3 Process Modeling

A process is a powerful simulation modeling abstraction. It enables a modeler to group sets of activities, events, and conditions into one functional unit. Without this mechanism, it is relatively difficult to construct relations between otherwise unrelated events and conditional activities. One might view a process to be a complex activity with dynamic qualities that can alter its behavior according to the presence or absence of specific external or internal variables.

Theoretically, all process-oriented simulation models can be emulated by a system supporting only events or activities. However, requiring the definition of complex processes using only simple temporal management tools can prove an arduous task. Users risk generating errors every time they utilize their own "process-type" constructs. Providing users with "process" abstraction tools reduces their time of implementation, enhances their ability to focus on their problem domains, and increases the reusability of their designs.

8.3.1 Process Requirements

A robust process-oriented simulation system must provide users with three important features. First, it must support multiple mechanisms for creating communication pathways between processes. This includes structures for synchronous and asynchronous communication. Second, it must provide constructs for the resolution of conflicting requests of exclusive-use resources. It should not allow for competing processes to access simultaneously restricted sources of information. Third, structures for processes to receive and to send information to their environment must be available. Processes must coordinate their activities with other processes and resources to accomplish their aims.

8.3.2 Process States

In a process-oriented simulation, every process is an individual entity. It interacts with other processes and uses the resources of a system to fulfill its immediate goals. The runtime environment for a process-oriented simulator must manage the interactions between processes and regulate the behaviors of processes according to their states. At any time during its lifetime, a process is in only one of three states. It may be *idle*, *holding*, or *waiting*. A process awaiting activation (or reactivation) is deemed to be idle. Idle processes neither invoke operations nor affect the state of other processes in a system. A process enters a holding state when awaiting simulated time to pass. Viewed as an interrupted process, a holding process resumes execution after its waiting time expires. A process is in a waiting state when it is accessing an unavailable data server (resource). A process withdraws from a waiting state immediately after the data resource becomes available.

Process-oriented simulators implemented on single processor machines require a scheduler to govern adequately the actions of the concurrent processes. The emulation of a genuine parallel management system is accomplished with an interleaving technique, such as those used in multi-tasking operating system. As the simulator leaps from process to process, it executes enough code for each process until it advances it to its next state. This incremental scheme ensures that every process progresses through time at the same rate.

8.3.3 Process Definition

All process-oriented languages provide developers with special constructs or commands to construct processes. Based upon their approach to process assembly, process-oriented languages can be classified as either *program-based* or *object-based.*

In program-based languages, a process is defined by a collection of routines or procedures. Each time a process is required to interpret information or change its state, one of these routines is invoked. The major disadvantage of this approach is that it does do not allow multiple instantiations of a single process. Users must duplicate code segments or define "state" caching statements to initiate repeated instances of an individual process. Notable program-based languages include GPSS[77], SLAM[71], and CSIM[79].

In object-based languages, processes are instances of *process templates.* Each template is composed of a collection of routines and data members that defines the logic of a single process. Every instance of a template defines a new process. This approach to process assembly offers three distinct advantages. First, processes are first-class objects and are manipulated as basic elements of the simulation language. They may be used in the following ways: (a) as parameter values to functions or other processes; (b) as return values of functions; (c) as arguments of equality or inequality tests; or (d) in assignment to process variables. Second, processes may be instantiated more than once. The use of templates facilitates the creation of similar processes. Third, it is possible to embed protocols of interaction into a process' design. Processes can be supplied with rules to regulate their employment during a simulation. For example, a process may be specified to govern selectively the visibility of its operations according to its state. This mechanism can ensure that the process is properly manipulated by external influences. Notable object-based languages include SIMULA[6], HSL[74], and Concurrent-C++[24].

8.4 RASP Process

The RASP toolkit uses an *object-based* approach to process modeling. Processes are created from general process templates. To enhance a process' efficacy, the toolkit provides users with additional constructs to control their progression through time. Commands are furnished to initiate, suspend, and terminate a running process. These tools facilitate the placement of processes into the hierarchy of temporal tools. To promote greater reusability, the toolkit advocates a set of design rules for the development of processes. These guidelines enforce users to construct processes as an enclosed entities interacting with their environment only through unidirectional ports. Conforming to the connection paradigm, this design methodology is consistent with the regulations administered by the IMVCD framework.

8.4.1 Process States

A process' period of activity can be confined to a closed temporal interval.[3] Not commonly used in process-oriented languages,[4] this scheme enhances users' abilities to construct variably operating processes. At any point during a simulation, a process may be coerced into an active or inactive state; thus the five[5] behavioral states are defined as follows:

- **idle**: Processes are defined to be idle if they are not actively attempting to alter one or more of the state variables of a simulation. Inactive processes do not process information or wait for time to pass. Processes are declared idle before their first summons to activation and immediately after they have been deactivated.

- **active**: A process is declared active if engaged in processing information. A process will enter an active state: (1) immediately after initializing; (2) directly after it has waited for time to pass; (3) as soon as a locked resource has been freed; or (4) after it subsides from a suspended state.

- **holding**: A process waiting for time to pass is defined to be in a holding state. A holding process can not query the environment or react to external events. It is (usually) the responsibility of a process to define the length of time it is to remain in a holding state. After a process' holding time has expired, it immediately passes into an active state.

- **waiting**: Some processes require direct interaction with their surroundings. Often, they will attempt to use or alter the resources of a system. However, not all resources may be readily available when they are requested for use by a process. When this situation occurs, a process can respond in two ways. First, it can alter its conduct to avoid utilization of the unavailable resource. This behavior does not (usually) require the process to forebear its processing of information or to suspend itself temporarily. Second, it can wait indefinitely for the resource to become obtainable. If a process chooses to pursue this second option,

[3]The association of temporal periods with processes is described in section 9.2.4.
[4]Except for languages using light-weight threads.
[5]RASP processes exhibit two additional states. However, this may change in the future.

it will be placed in a waiting state. A waiting or "blocked" process instantly becomes an active process the moment the previously unavailable resource becomes available.

- **suspended**: A suspended process is temporarily inoperative. It does not engage in any information processing tasks, and is not waiting for external events or holding times to expire. Only processes defined to be in an active, holding, or waiting state can be suspended.

8.4.2 Process Design

The reusability of an object's definition in a programming environment is directly influenced by the measure and type of constraints incorporated into the object's design. Design constraints establish the domain of an object's usage. An object's domain is easily expanded or altered when it is easy to alter its embodied constraints. Therefore, to enhance an object's usage, it should be clear how to identify and modify the object's constraints. In section 7.3.1, it was shown that *indirect* links coupled with a first-class interface affords important qualities to the design of reusable components. Objects purposely using *indirect* links reduce the exactness and highlight the identity of their constraints. Viewed and defined as objects in the RASP toolkit, processes can also utilize *indirect* links. The usage of indirect links, which alleviates an object from knowing the exact identity of its interacting companion, enhances the design of reusable processes.

Influenced by Manifold[3], the connection paradigm, and Kerridge's port language (CPP)[44], RASP processes are designed to interact with the environment through first-class "ports", unidirectional data entry or egress gateways. A process obtains or transmits data through its set of ports. Datum transfers occur only through ports. Although process languages, such as SIMULA, CSIM, and Concurrent C, support port-like structures,[6] each language's port devices apply only to incoming information. This scheme empowers processes to be oblivious of their data sender's identity but requires them to be aware of their data receiver's identity. To transmit data to its environment, a process must notify every data receiver of the impending transaction. This difficulty does not arise when processes issue data through *output* ports. After

[6]CSIM's "mailboxes" and Concurrent C's "transactions" are similar to RASP process ports.

a process transfers information to its ports, the system is responsible for advancing the data from the ports to the appropriate receivers.

This process design scheme separates communication issues from the functionality of process modules. Processes are developed independently of a context. The communication pathways between collections of processes and data entities are specified by users after a process' design has been completed. In RASP, all pathways are constructed with the assistance of a set of temporal management tools (see section 9.2). These tools establish links between separate objects (processes) contingent upon the value of time or the observation of state changes. Unlike well-known process organization paradigms, such as Communicating Sequential Processes[32], the communicational pattern between distinct processes is not fixed at compile time. The topology of the communication network and the potential connectivity of individual processes is established and alterable during run-time. The ability to change dynamically the communicational patterns of a running simulation can better emulate the properties of a real world system.

8.4.3 Message Passing

In process-based systems, processes interact via message passing. The transfer of information occurs when one process transmits a message to another. Message passing schemes are classified as either *synchronous* or *asynchronous*. In the synchronous model, distinct processes synchronize their behaviors to accommodate the transfer of information. This scheme requires one or both of the processes to suspend (block) their behavior until the delivery of information is completed. In asynchronous (non-blocking) message passing, processes do not block to transmit or receive information. Communicating processes are not enforced to synchronize their behaviors to exchange data. Synchronization methods are replaced with message buffers and sizable message controllers. As data flows from process to process, it must pass through a message buffer to enter or exit a process. "Bufferless" schemes are used when the loss of information between processes is deemed as acceptable.

RASP processes support three types of message passing schemes: *asynchronous send*, *asynchronous receive*, and *synchronous receive*. Although a general plan to permit "synchronous send" is not defined, it does not imply that it is not possible to formulate. This commonly used

message passing scheme is consigned under the topic of future work.

Asynchronous Message Send

A process sends messages asynchronously by setting the value of one or more of its "output" ports, whereupon it continues to execute. The simulation kernel directs the data from the process' outport ports to the "input" ports of other objects or processes. It should be noted that RASP ports do not buffer information. Immediately after new data arrives, old information is overwritten. Thus objects or processes loose old values unless they choose to cache it themselves. The impact of buffering ports into RASP design is consigned for future work.

Asynchronous Message Receive

A process asynchronously receives a message by polling one of its "input" ports for a value. If a new value is not available, the process suspends its progress for an explicit period of time, after which the process is free to re-examine its input port or execute other procedures. This temporary suspension mechanism is vital to process-oriented simulations implemented on single-processor machines. A process continually polling its ports for values without repose hinders the forward progression of a simulation. An enormous amount of processor time would be devoured by this non-stop polling process.

Synchronous Message Receive

To receive messages synchronously, a process examines one of "input" ports for a value. Unlike the asynchronous method, the failure to obtain a valid value does not force the process to re-examine its ports at a later time. A failure induces the invalid "input" port to block the process' progression and wait for the arrival of information. Upon obtaining a valid value, the port notifies the blocked process to restart its state of operation. The diagram in Figure 8.2 shows a few lines of pseudo-code delineating the steps involved in the reception of a synchronous message.

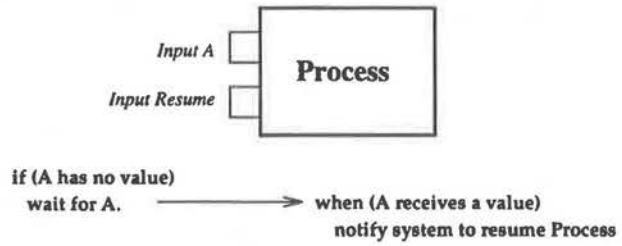

Figure 8.2: Synchronous Receive

8.4.4 Benefits

The construction of independent processes is enhanced by alleviating each process from acknowledging the identity of its data partners. Eliminating the "naming" restriction allows processes to be developed without concern for the demands and requests of its surrounding environment. In addition, port-based processes afford four important qualities to simulation design. They are as follows:

- **Parallelism**: Although the current toolkit is not devised to take advantage of parallel architectures, its design can be modified for concurrency. The influence of Manifold, Concurrent C, and Kerridge's CPP language, to RASP's design advocates the development of a parallel toolkit.

- **Reusability**: The ability to develop RASP processes with little regard for the environment where it interacts contributes to the design of reusable processes. Liberating a process from its external constraints enables it to be used in a wide variety of applications.

- **Composition**: The toolkit's black-box approach to process development encourages the creation of composite processes. Consolidated processes are formed by an orchestration of user-defined connections. All unconnected ports serve as the ports for the new composite process. The diagram is Figure 8.3 depicts a complex process created from a collection of smaller processes.

- **Proper Usage**: Process ports are useful to ensure a process' protocol for usage is observed. As a port is provided with a value, it can signal its governing process to disregard

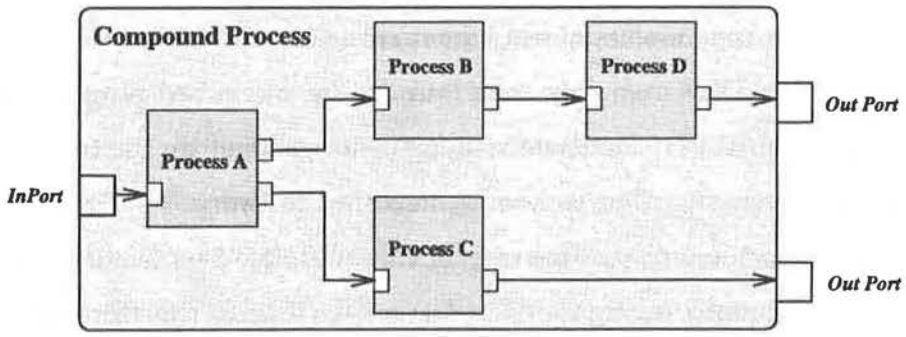

Figure 8.3: Composite Process

information from other ports. This scheme safeguards the process from attaining improper values or ascertaining a corrupt state.

8.4.5 Coroutines

In a process-oriented simulation, all active processes execute in parallel. Each process may be viewed as a self-governing entity. Unless specifically defined, there are no naturally occurring hierarchical relationship existing between separate processes. Therefore, a process-based simulator must be able to guide fairly the concurrent actions of multiple processes. Unfortunately, this is not truly possible to accomplish on machines with only one central processing unit (CPU). However, the quasi-parallel execution of processes may be achieved by making each process temporarily suspend its thread of execution after it has performed a subset of its operations. This technique enables multiple processes to carry out their patterns of behavior at approximately uniform rates.

The traditional procedure-based approach to programming does not permit routines to temporarily suspend themselves. Once they exit their train of execution, their procedure instances are terminated. This problem can be alleviated by utilizing special routines called "coroutines". These routines enable suspended routines to become reactivated after having been dormant for indefinite periods of time. Reactivation automatically restores the pre-suspension state of a routine. All variables and registers are reset to their previously active values. Most importantly, reactivation will begin execution from the point immediately following the statement

which suspended the process.

Despite their apparent usefulness, coroutines are not directly supported by the popular high-level languages. They have only been provided by specialized programming languages, such as SIMULA and BLISS. Fortunately, it is possible to emulate the functionality of coroutines within basic languages. This can be accomplished in two ways. First, assembly code can be developed to store and restore the registers of the stack. This would enable users to explicitly suspend the execution of any routine. The major drawback to this method is that one must write a different set of assembly code for each machine used. Second, the explicit features of a programming language can be manipulated to store states of routines and bypass segments of code. Although this method preserves the portability of code, it requires users to incorporate additional constructs into their programs. This drawback can be eliminated via the development of a language pre-processor and/or definable macro expansions.

The RASP toolkit employs the second method. It requires users to incorporate basic structures into their code. This enables standard procedures to emulate coroutines.

8.4.6 Unresolved Issues

The RASP toolkit provides users with a common framework to define processes and describe their interactions. Although this scheme provides enough guidance for the incorporation of processes in a simulation model, it does not attempt to resolve every issue associated with process-oriented simulations. Common design problems, such as deadlocking, process priorities, and port visibilities are not addressed.

8.5 Informal Description

Using Zeigler's informal description, RASP's multiple interface approach to simulation modeling is structured as follows:

1. **Components**:

   The set of components is partitioned into two groups: $D = \{\alpha_1, \alpha_2, \ldots, \alpha_M\}$ and $E = \{\phi_1, \phi_2, \ldots, \phi_N\}$. Each partition is divided into ACTIVE and PASSIVE types. ACTIVE

components impose changes to a system, while PASSIVE types retain their state unless acted upon by others.

2. **Descriptive Variables:**

Every ACTIVE-D is described by its state, time to activation, set of influencees, and set of influencers. At any moment in time, an ACTIVE-D is defined by its value, its capability to alter the states of other components (D or E), and its ability to be altered by others components (D or E). ACTIVE-E's are described by the same set of characteristics, except that their states are defined by the values of their local variables and the status of their control instructions.

	ACTIVE-α		ACTIVE-ϕ	
	Range	Value	Range	Value
STATE-OF-	S_α	s_α		
CONTROL-OF-			L_ϕ	$\{0, 1, 2, \ldots, M\}$
MEMORY-OF-			V_ϕ	v_ϕ
TIME-LEFT-	$\pm\infty$	σ_α	$\pm\infty$	σ_ϕ
INFLUENCEES-	D, E	$\{\beta_{D_1}, \ldots, \beta_{D_M}, \beta_{E_1}, \ldots, \beta_{E_N}\}$	D, E	$\{\beta_{D_1}, \ldots, \beta_{D_M}, \beta_{E_1}, \ldots, \beta_{E_N}\}$
INFLUENCERS-	D, E	$\{\overline{\beta}_{D_1}, \ldots, \overline{\beta}_{D_M}, \overline{\beta}_{E_1}, \ldots, \overline{\beta}_{E_N}\}$	D, E	$\{\overline{\beta}_{D_1}, \ldots, \overline{\beta}_{D_M}, \overline{\beta}_{E_1}, \ldots, \overline{\beta}_{E_N}\}$

3. **Component Interaction:**

• For each ACTIVE-α a local transition function $\{\delta_\alpha\}$ is specified. Given the union of the current values for the INFLUENCEES-α and INFLUENCERS-α, the function simply produces a new list of values for INFLUENCEES-α[7].

$$Influencees_\alpha = \overbrace{(s_{\beta_{D_1}}, \sigma_{\beta_{D_1}}), \ldots, s_{\beta_{D_m}}}^{\text{Active \& Passive}}, (\overbrace{(l, v_\phi, \sigma_\phi)}^{\text{Substate}}) \overbrace{(l_{\beta_{E_2}}, v_{\beta_{E_2}}, \sigma_{\beta_{E_2}}), \ldots, v_{\beta_{E_n}}}^{\text{Active \& Passive}}$$

$$Influencers_\alpha = (s_{\overline{\beta}_{D_1}}, \sigma_{\overline{\beta}_{D_1}}), \ldots, s_{\overline{\beta}_{D_m}}, ((l, v_\phi, \sigma_\phi)(l_{\overline{\beta}_{E_2}}, v_{\overline{\beta}_{E_2}}, \sigma_{\overline{\beta}_{E_2}}), \ldots, v_{\overline{\beta}_{E_n}})$$

$$\delta_\alpha[Influencees_\alpha, Influencers_\alpha] =$$

$$\begin{cases} f_\alpha(Influencees_\alpha, Influencers_\alpha) & \text{if } C_\alpha(Influencers_\alpha) = \text{TRUE} \\ ((s_{\beta_{D_1}}, \sigma_{\beta_{D_1}} - t(s)), \ldots, s_{\beta_{D_m}}, & \\ \quad ((l, v_\phi, \sigma_\phi)(l_{\beta_{E_2}}, v_{\beta_{E_2}}, \sigma_{\beta_{E_2}} - t(s)), \ldots, v_{\beta_{E_n}}) & \text{otherwise} \end{cases}$$

[7]Except for the inclusion of variables associated with ϕ, this local transition function is similar to the one described section 5.2.2.

- For each substate of ACTIVE-ϕ a local transition functions $\{\delta^l_\phi\}$ is specified[8].

$$Influencees^l_\phi = (s_{\beta_{D_1}}, \sigma_{\beta_{D_1}}), \ldots, s_{\beta_{D_m}}, ((v_\phi, \sigma_\phi)(l_{\beta_{E_2}}, v_{\beta_{E_2}}, \sigma_{\beta_{E_2}}), \ldots, v_{\beta_{E_n}})$$

$$Influencers^l_\phi = (s_{\overline{\beta}_{D_1}}, \sigma_{\overline{\beta}_{D_1}}), \ldots, s_{\overline{\beta}_{D_m}}, ((v_\phi, \sigma_\phi)(l_{\overline{\beta}_{E_2}}, v_{\overline{\beta}_{E_2}}, \sigma_{\overline{\beta}_{E_2}}), \ldots, v_{\overline{\beta}_{E_n}})$$

$$\delta^l_\phi[Influencees^l_\phi, Influencers^l_\phi] =$$

$$\begin{cases} f^l_\phi(Influencees^l_\phi, Influencers_\phi) & \text{if } C^l_\phi(Influencers^l_\phi) = \text{TRUE} \\ ((s_{\beta_{D_1}}, \sigma_{\beta_{D_1}} - t(s)), \ldots, s_{\beta_{D_m}}, & \\ \quad ((v_\phi, \sigma_\phi)(l_{\beta_{E_2}}, v_{\beta_{E_2}}, \sigma_{\beta_{E_2}} - t(s)), \ldots, v_{\beta_{E_n}}) & \text{otherwise} \end{cases}$$

To integrate this approach with the toolkit's set of temporal tools (section 9.2), a description of a prototype multiple interface simulation is withheld until section 9.3.

[8]Except for the inclusion of variables associated with α, this local transition function is similar to the one described section 5.2.3.

CHAPTER 9

RASP: TIME AND STATE

We must use time as a tool, not as a couch.
- John Fitzgerald Kennedy,
The Observer, "Saying of the Week", Dec 10, 1961

A simulation specifies how a system changes over time. The validity of a model is compromised if it lacks descriptive declarations of important state changes. Before users can specify vital state changing information, they must have a clear understanding of the relationship between time and state. The association of time and state imposed by the structure of a simulation language severely effects a user's comprehension. Therefore, it is essential to provide users with a precise set of "implementation-free"[1] definitions which carefully characterize the state and time relationship.

This chapter describes the relationship between time and state in the RASP toolkit. It is "implementation-free" design. Basic definitions, important classifications and descriptive labels are discussed in detail. It should be noted that the following deliberately attempts to conform the terminology in this section to the descriptions used in Nance's theory of time and state[57] which has provided an excellent foundation for the development of the RASP toolkit. The chapter concludes with an informal description of RASP's multiple interface to simulation and a description of RASP's simulation kernel.

[1]Implementation-free ideas or objects were not designed to conform to any specific programming language.

9.1 Time Representation

9.1.1 Time Structure

An important design of any simulation language is the specification of a *time structure*. The structure of time defines the unit of measurement for a temporal system. Time can be mapped to the set of rational numbers, floating point numbers, or integer numbers. Although an integer valued time axis is used in many simulation languages, the RASP toolkit uses floating point numbers. Real numbers provide a better foundation for continuous time simulations. The inability to specify actions at non-integer times constrains all time-varying simulations to a discrete-time foundation.

9.1.2 Central Clock

Every simulator maintains an internal clock whose values represent the passage of time. Since the definition of the system's state is often a direct function of time, the behavior of the simulation clock is important. All simulations created with the RASP toolkit observe one global clock. The state of this clock represents the "absolute" time of a simulation. Although there is only one timepiece, the toolkit does not constrain users to reference continually the global clock as the only source of a "time" value. The nature of RASP's temporal management tools enables users to design a variety of modeling situations relative to "local" time frames.

9.2 Temporal management tools

The RASP toolkit provide users with a set of temporal management primitives to produce scripted animations or self-governing simulations. These primitives are divided into two groups: *action* types and *governor* types. The two primary action types, *events* and *processes*, serve to alter the values of state variables within a system. The two primary governor types, *activities* and *processions*, serve to manage the behavior of action types and to dictate when action types become active. The well-formed relationships (shown in Figure 9.1) formed between the primitives facilitates the construction of complex simulations.

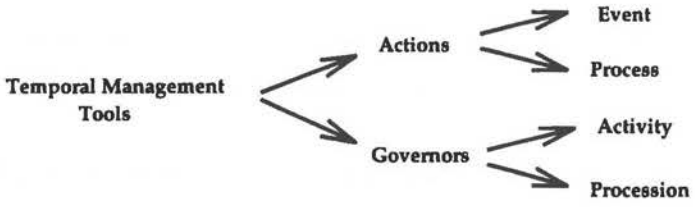

Figure 9.1: Breakdown of Temporal Tools

9.2.1 EVENTS

Representing non-decomposable elemental actions, an *event* promotes a modification to the state or structure of a system. These modifications include function calls, simple data transactions, and link declarations. All events produce instantaneous changes to a system. It is important to note that events do not possess any temporally based information. It is not an event's responsibility to regulate its conduct through time. The exclusion of temporal information enables users to concentrate on the development of system changes without concern for when they are to occur.

Although all event-driven languages support the concept of an "event", most of them do not define structures for the creation of an "event". The pervasive ideology of most simulation languages is that all events should be created by users. Users create events for the system to perform. The lack of common event types contributes to the hardship of joining two independently constructed models. A common foundation does not always guarantee an effortless process of model assembly. In addition to facilitating the integration of two models, a classification of event types enhances the language's reusability and ability to create complex simulations.

The RASP toolkit introduces a set of event "templates". Developed to take advantage of the connectionist structure imposed by the toolkit's design, each event type executes an important state changing operation. Events are defined as follows:

- **CallEvent**: This is the simplest type of event. It performs only one function during actuation. It simply activates (executes) a *target* port. There is no transfer of data or analysis of port state.

- **Event**: An instance of this type of event performs one of two actions. When triggered by the simulation kernel, it may transfer data from a *source* port to a *target* port, or it may execute any procedure requiring no arguments.

- **TimeEvent**: This type of event is similar to an **Event**. It supports two types of actions and can be specified during run-time. Only the functional requirements of the two events differ. A TimeEvent does not require a *source* port and it can only execute a procedure requiring one argument of type "double".[2] When this event is triggered, a *system time* value is transmitted to either the *target* port or the single argument procedure.

- **StateEvent**: This event type is unlike any of the three other events. Never explicitly activated by the simulation kernel, its actuation is entirely dependent upon the state of an associated *source* port. If the port changes state, this event type will immediately trigger the occurrence of another event or *activity*. Events of this type facilitate the building of any models requiring an *activity scanning* world view. Once a StateEvent is activated, it remains active until disabled.

- **DisableEvent**: Given a *source* port, this event type clears the port of all previously associated StateEvents.

- **ChainEvent**: In many simulations, several sets of events occur simultaneously or execute in tandem during a single instance of time. Requiring the user to continually pass the individual elements (events) of each set can become burdensome. This onerous task is alleviated by using **ChainEvents**. This collection type (class) enables instances of **Events**, **TimeEvents**, **StateEvents**, and **CallEvents** to be grouped into one single multi-action event.

9.2.2 Event Activation

When an Event or CallEvent is triggered, each event type executes one or more of its associated set of actions. Target ports are activated or data is transferred between two locations. The

[2]Time values are of type "double" to accommodate the floating point time axis established in section 9.1.1.

result of activating a TimeEvent or StateEvent is not as simple. The activation of a TimeEvent provides a port (or procedure) a temporal value. This value is elicited from either the "global" system time or the "local" lifetime of an activity. It is the responsibility of the user to specify the time frame from which the temporal value is deduced. This is usually specified when an event is created.

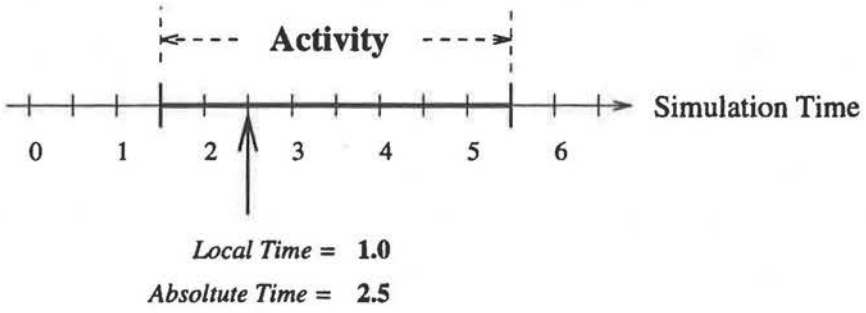

Figure 9.2: Absolute vs. Relative Time

The diagram in figure 9.2 illustrates the difference between the "global" and "local" time frames. One explicit instance of time is labeled twice. The first label indicates the "local" value, while the second label indicates the corresponding "global" value. In the "local" frame, a "zero" time value is defined to be the beginning of an activity. Zero time in the global time frame is defined by the beginning of the simulation.

The behavior of a StateEvent is dissimilar to the three other event types. Its activation may or may not produce any noticeable results. When activated by an activity, this event type notifies an associated port to signal the occurrence of any change in the port's state. Immediately after receiving a positive state change affirmation from the port, the event will direct the appearance of its associated action or activity. It should be noted that the state inspection request does not endure forever. The query is recalled the moment the event's activity is deactivated.

9.2.3 ACTIVITIES

Activities[3] are used to associate temporal information with events. They provide meaning (purpose) to collections of otherwise independent events. Only when an activity obtains an "active" state can its corresponding set of events become "active". The systematic activation of events defines an *occurrence.* An occurrence represents a continual action occurring over a finite length of time.[4] Although every activity delimits the duration its existence, no activity can trigger itself. Actuation notification must come from an external source. The inclusion of self-triggering mechanisms would unnecessarily clutter the structure and operation of an activity.

Timing information governs the state of every activity. Defined as an interval, this span establishes two important rules of conduct. First, it defines a set of conditions to transform a "passive" activity into an "active" one and vice-versa. Conditions delimit specific instances of time or identify prerequisite conditions to signal the beginning and termination of the activity. Second, it defines the temporal rate at which the activity is to progress. An activity with a high rate executes its events more often than a low rate activity. This important feature allows concurrent activities to advance time using different increments.

Every activity partitions its events into one of three categories. Each classification defines a different frequency and timing patterns for its set of events. Categories are as follows:

- **Initial Event**: Events placed in this category are instantly activated when its governing activity is assigned an "active" state. These events will be triggered only once during the lifetime of the activity.

- **Acting Event**: Every event assigned to this group will be continually activated for the entire duration of its ruling activity's "active" existence. Events will be activated during the initialization, advancement, and termination of the activity. It is important to note that all "initial events" trigger before all "acting events" at the beginning of the

[3]In the activity scanning approach (described in section 5.2.2), activities are delimited by two successive events and are defined to represent the state of an entity over an interval of time. RASP activities differ in that the delimiting events need not be successive and the state of the entity may vary. Allowing events to occur in between enables users to model simple continuous actions.

[4]This is not entirely true. An activity may be defined to endure for an infinite length of time.

activity's life span. Similarly, all "finish events" execute after all "acting events" during the completion of the activity.

- **Finish Event**: Events set in this category are immediately activated when their governing activity is completing. Nothing occurs between the activation of final events and the final moment of the activity's existence.

Events are placed into one or more categories. No regulations exist as to the quantity or type of events linked with each category. Therefore, an event may simultaneously belong to more than one category and to several activities.

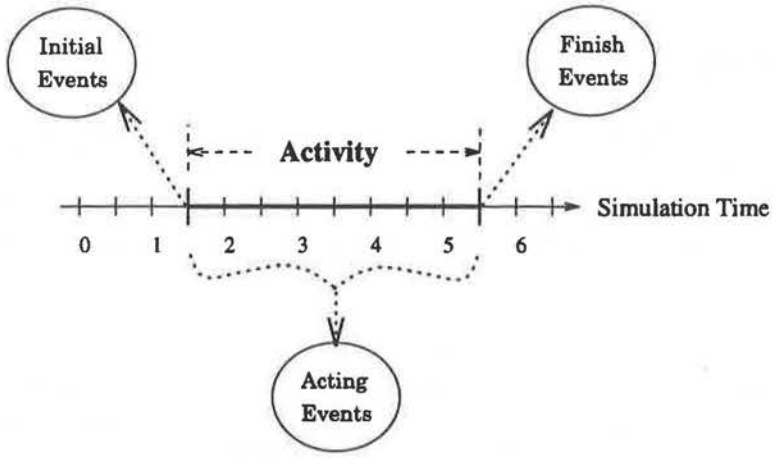

Figure 9.3: Activity Event Partitions

9.2.4 Processes

In the RASP hierarchy of temporal tools, processes are ranked at a level equivalent to events. This ranking does not imply that processes are identical to events. It implies that both temporal primitives are manipulated in a similar fashion. Each requires the assistance of an activity to initiate their actions and manage their temporal existence. A process' activity determines the exact time when the process is triggered. Unlike events, once a process is activated, it can be designated to exist for an indefinite length of time. Until terminated, the process will

continually accept information and alter the state variables of the system as time advances forward. A governing activity can not alter the progression of an indefinite process after it has been initiated. Regardless of the designated time span of a process, RASP activities do not have the ability to alter the internal behavior of any process.

A process can be terminated in one of three ways. First, it can intentionally discontinue its state of activation. When a process completes it last operation, it will not require reactivation. Second, it can receive a termination message from an external source, such as other processes, routines, events, etc. Third, the process' governing activity can expire. Unless explicitly designated to exist for an indefinite length of time, a process can endure only as long as the activity which initiated it.

9.2.5 PROCESSIONS

A governing entity for collection of activities, a *procession* organizes sets of activities, activates them in chronological order, and controls their behaviors. Through a procession, activities are placed into a common *event-list.*[5] This list determines the order in which activities are processed.

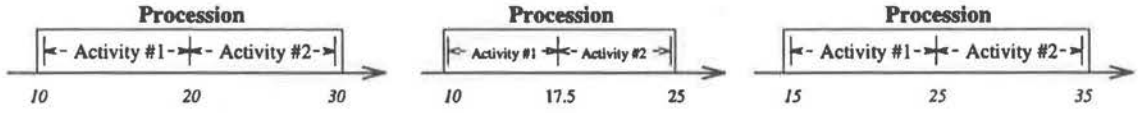

Figure 9.4: Variational Timing of Processions

Each procession defines a "local" timeframe. Through specifying the placement of activities within the timeframe of a procession, users form temporal relationships among collections of activities without references to the global clock. A procession's placement in absolute time defines the activation times for its set of activities. In addition, this design makes it easier to shift, contract, or expand the timing patterns of a set of activities. Figure 9.4 illustrates the timing intervals of three processions containing identical activities. The interval in the middle

[5]Every *Procession* utilizes two *event-lists*. One list maintains a set of "waiting" activities, and the other list contains a collection of "active" activities.

has been temporally contracted while the right one has been shifted forward five units in time.

9.2.6 Hierarchical Structure

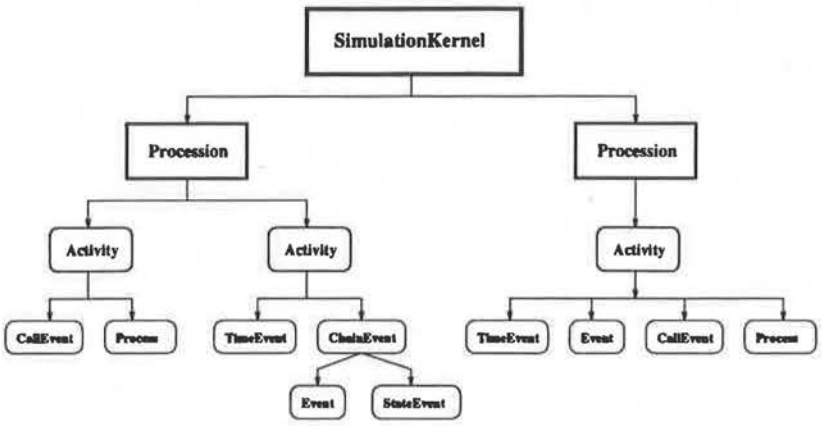

Figure 9.5: Simulation Hierarchy

The relationship between RASP events, activities, processes, and processions forms a natural hierarchy, as shown in Figure 9.5. It provides users with a well-defined conceptual framework for the modeling of time-varying simulations. The responsibilities of each node at any level in the tree are clearly delineated. The effects of modifications applied to the parameters of any of the temporal primitives is localized.

9.3 RASP's Kernel

RASP's kernel is designed to take advantage of the toolkit's multiple interface approach to discrete-event modeling and its hierarchy of temporal tools. Simulations are defined using a combination of events, contingent activities, and processes. Using an object-oriented approach, the kernel minimizes the size of its control algorithm to a simple set of steps by apportioning duties to the set of temporal tools, such the activation of events and the maintenance of process activation points,

RASP's kernel simply stores and manages the **processions** of a simulation. Through examining of each procession, the kernel advances time forward. The kernel does not maintain

any type of sorted list and it is unaware of the existence of events, activities, or processes. The kernel's purpose is to manage the progression of time.

A procession stores and manages the **activities** of a simulation. Activities are partitioning into two groups according to the value of simulated time. Those waiting for activation are placed into a "waiting activity" list while those already activated are placed into an "active activity" list. Both lists are sorted by activation times to facilitate the rapid retrieval of activities with earliest activation times. A procession serves to organize activities (for the simulation kernel).

An activity stores and manages the **events** of a simulation. Events are partitioned into three groups according to their frequency, as described in section 9.2.3. An activity serves to organize events and define the temporal granularity between events (for the simulation kernel).

The diagram in Figure 9.6 illustrates the interactions which occur between the kernel and the toolkit's temporal tools. The direction of the arrows indicate the flow of information between the components.

Multiple Interface Simulation

The diagram in Figure 9.7 shows a complete expansion of the algorithm driving RASP's kernel.[6] The algorithm's object-oriented design, shown in Figure 9.6, is omitted to emphasize its overall structure. The statement in line 9 enables users to restrict the temporal jump size of a simulation to a maximum value. This restriction allows users to design discrete-time simulation without specifying enormous numbers of events. Future enhancements, which further decompose the RASP kernel, may decide to consign this restriction to a kernel subclass.

[6]Variables used in the diagram are described in section 8.5.

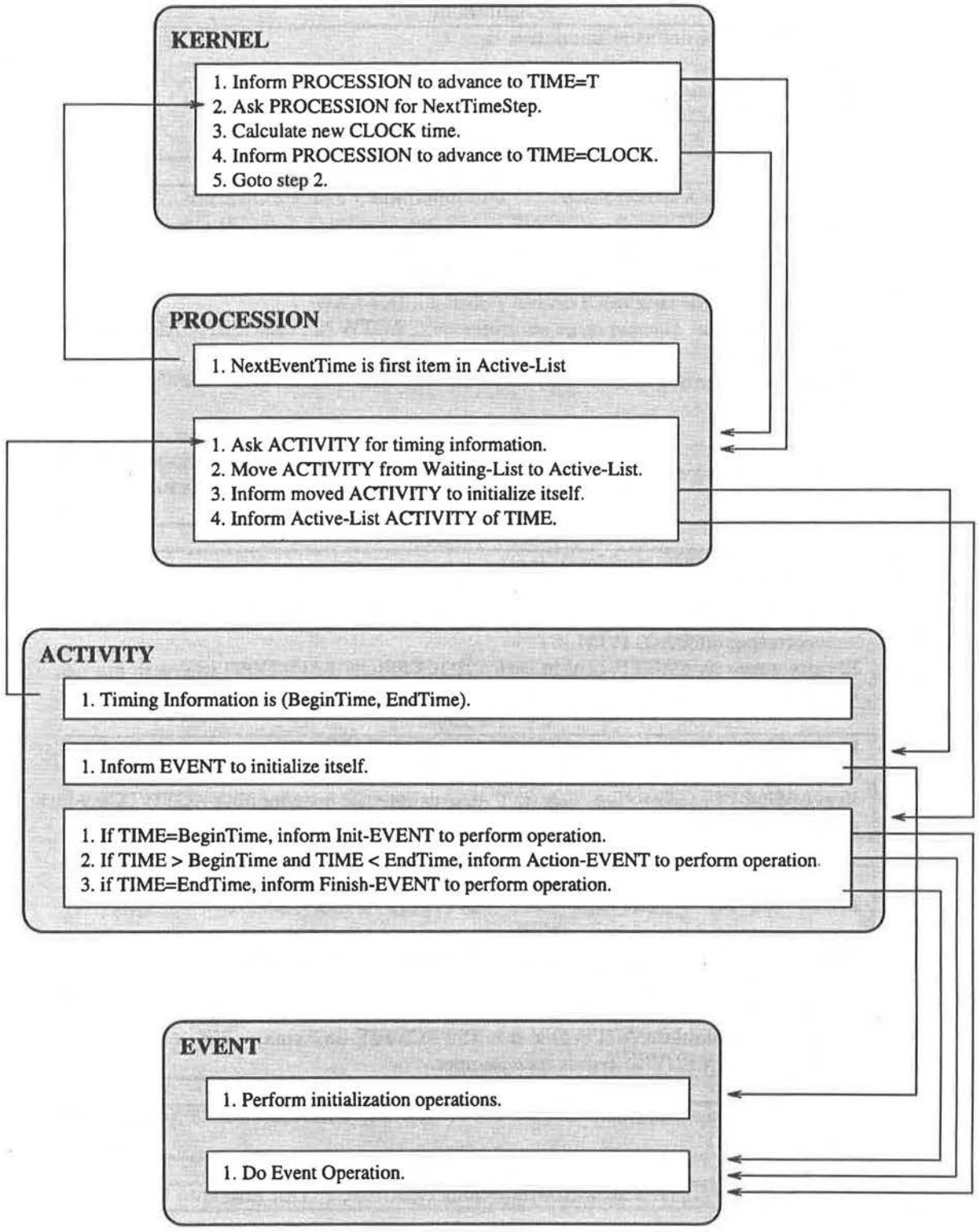

Figure 9.6: Object Kernel Design

Initialization

1 Set CLOCK to initial simulation time t_0
2 Set $\Delta_{MAX-TIME}$ to initial value Δ_0
3 Set state variables $\overline{S_\alpha}$'s to initial values of s_α's.
4 Place every ACTIVITY in a PROCESSION's WAITING-LIST
5 Place every PROCESSION in the PROCESSION-LIST.

Scanning Phase

6 (a) Scan the PROCESSION-LIST to determine which PROCESSION's
→ PASSIVE-LIST has the ACTIVITY with the smallest δ. Let $\overline{\delta}$ denote the winner.
→ ACTIVE-LIST has the ACTIVITY with the smallest ω. Let $\overline{\omega}$ denote the winner.
(b) Set NEXT-EVENT-TIME to the smaller of $\overline{\delta}$ and $\overline{\omega}$.
7 Set $\Delta_{NEXT-TIME}$ to (NEXT-EVENT-TIME - CLOCK).
8 (a) Examine the $\Delta_{TIME-INC}$ of every ACTIVITY in each PROCESSION'S ACTIVE-LIST.
(b) Let $\Delta_{REQ-TIME}$ denote the temporal step that satisfies every ACTIVITY's requirements.
9 Set $\Delta_{NEW-TIME} = \begin{cases} \Delta_{MAX-TIME} & if(\Delta_{MAX-TIME} < \Delta_{REQ-TIME}) \\ & \text{and } (\Delta_{MAX-TIME} < \Delta_{NEXT-TIME}) \\ \Delta_{REQ-TIME} & if(\Delta_{REQ-TIME} < \Delta_{NEXT-TIME}) \\ \Delta_{NEXT-TIME} & \text{otherwise} \end{cases}$

Time Advance

10 Advance the CLOCK by $\Delta_{NEW-TIME}$.

List Update

11 Move every ACTIVITY in each PROCESSION's PASSIVE-LIST with $\sigma < 0$ to its corresponding ACTIVE-LIST.
12 Remove every ACTIVITY in each PROCESSION's ACTIVE-LIST with $\sigma = \infty$.
13 Mark every ACTIVITY in each PROCESSION's ACTIVE-LIST as unselected.

Tie Breaking

14 Apply SELECT_1 to every PROCESSION's ACTIVE-LIST. Let Λ_{Act} denote the list containing the winning ACTIVITY of each PROCESSION.
15 Apply SELECT_2 to the Λ_{Act}. Let $\overline{ACT}$ denote the final winning ACTIVITY. Mark $\overline{ACT}$ as selected.

State Transition

16 Invoke $\overline{ACT}'s$ $\begin{cases} \text{EVENTS}_{Init} \text{ and EVENTS}_{Acts} & \text{if CLOCK} = \overline{ACT}_{time-begin} \\ \text{EVENTS}_{Acts} \text{ and EVENTS}_{Fini} & \text{if CLOCK} = \overline{ACT}_{time-end} \\ \text{EVENTS}_{Acts} & \text{otherwise} \end{cases}$
17 Adjust the state for every INFLUENCEE-β-OF-$\overline{ACT}$-EVENTS_{ANY}.
(a) For every ACTIVITY-β with:
→ $\sigma_\beta < \infty$, place ACTIVITY-β in the PASSIVE-liST containing $\overline{ACT}$.
→ $\sigma_\beta \leq 0$, place ACTIVITY-β in the ACTIVE-liST containing $\overline{ACT}$.
(b) For every EVENT-β, invoke its operation.

Any Activities Remaining?

18 If there exists an unmarked ACTIVITY in any PROCESSION's ACTIVE-LIST, then goto 11.

Termination Test

19 If NEW-EVENT-TIME exceeds termination time, STOP. Else goto 6.

Figure 9.7: RASP Multiple Interface Kernel

CHAPTER 10

RASP: GRAPHICAL MODELS

People see only what they are prepared to see.
- Ralph Waldo Emerson, Journals, 1863
A place for everything and everything in its place.
- Samuel Smiles, Thrift

Visualizing the attributes, behaviors, and actions of a simulation enhances the transfer of information. Graphical views serve as valuable tools for the development, explication, and augmentation of complex simulations. The visualization process is especially important to the production of computer animations. Visual representations of time-varying models aid in verifying the validity of any simulation. The visualization process can be segmented into two phases, *model creation* and *data-to-image translation.* In the model creation phase, the components of the simulation are linked with visual attributes. Common attributes include geometric shapes, physical properties, and material characteristics. Data-to-image translation entails the generation of computer images from the data of the first phase. Component attributes are interpreted for rendering engines to form resplendent pictures.

This chapter presents RASP's approach to model creation and data-to-image translation. Discussion includes an analysis of previous approaches to the visualization process and a description of a new design which improves upon these previous approaches.

10.1 Model Creation

The primary purpose of the *model creation* phase is to create "visual" entities with "informational" characteristics. Properties associated with models define their appearance and shape. For time-varying systems, the ability to dynamically manipulate all the characteristics of an model proves essential.

There are two traditional approaches to *model creation*. Popularized by computer graphics toolkits[35, 89, 92, 46, 83], the first approach forms models by amassing physical, material, and viewing primitives into one large ordered list. Users edit the elements of the list to produce a visual change to an model's appearance. The diagram in figure 10.1A demonstrates the usage of the *display-list* approach to model creation. A red cube is placed along the x-axis. The benefits of this approach include the simple incorporation of new attributes and a straightforward protocol to hierarchical modeling.[1] Although simple to use and implement, the *display-list* approach has two major drawbacks. First, models are *first-class* entities. The lack of a formal object-oriented interface prevents users from directing queries or creating local changes to a model. In addition, users are not separated from an model's internal representation. Although previous works by [92, 46, 83] layer an object-oriented interface on top of the display-list structure, these approaches do not attempt to conceal the display-list implementation from users. Model procedures (data members) are primarily used to edit the display list structure. Second, to alter a specific trait of a model, the attribute's position in the display-list must be known. This requirement usually entails the creation of "tags" or tables of "index" or "path" values.[2]

The second approach to model creation uses *geometric primitives* to represent basic geometric shapes, such as a cube, square, etc. As shown in Figure 10.1B, a primitive's appearance is defined by its associated set of "visual" characteristics which are attached to them using a pre-defined set of procedures or member functions, A model's capability is expanded by adding new functions to its inventory of operations. There are several benefits to this design. Inheritance and polymorphism assist in the creation of compatible model interfaces. For example, in GRAMS[18], all primitives are sub-classes of the abstract class "GraphicObject" that defines

[1] In Phigs[89], attributes not defined in a sublist are inherited from the sublist's parent.
[2] The position of an element in a hierarchical data structure can be represented by a "path".

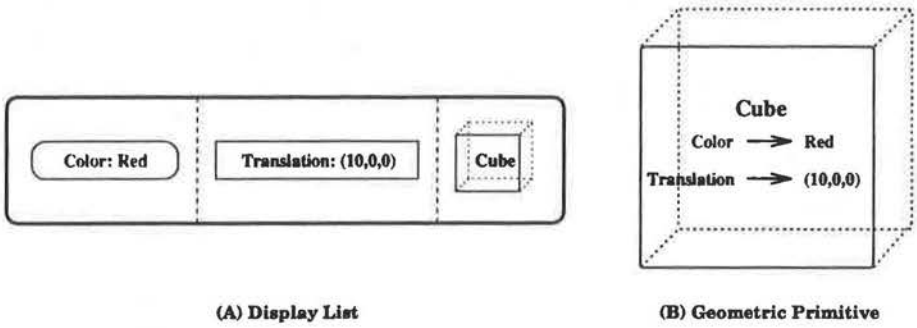

Figure 10.1: Display-list vs.Geometric Primitive

the common attributes and member functions for all primitives. In addition, "pure" virtual declarations require the creation of a collection of routines by all of its inheritors. Model are first-class entities and sets of operations that may be performed by them are publicly known. Models are defined by their operations, not their internal implementation. Lastly, complex geometric figures are created by hierarchically combining collections of primitives.

Although this approach to model creation is powerful, its strengths are also its weaknesses. First, models may become difficult to use or manage as repeated addition of new operations produce "enormous" models that use more memory and are not easy to modify or extend. Second, geometries are not usually modifiable. Most definitions enable the primitive to change its representation but not its overall shape.[3] In a complex simulation, an model's entire appearance may need to change. Third, a model's interface that does not provide access to specific encapsulated data structures or data members may hinder an model's usefulness in time-varying environments. A model may be unusable if it is not possible to animate all of its features.

The discrepancies between the display-list and geometric primitive approaches can be attributed to their dissimilar aims. Each model emphasizes a different aspect of object modeling. In the display-list approach, importance is placed on the hierarchical ordering of geometrical shapes and the attributes needed for image rendering. The content and quality of an image are defined by a one-pass traversal of the elements in the display-list because traversed elements

[3] A sphere may be represented by a collection of quadrilaterals or one large triangular mesh. The representation is different, but its external shape is still spherical.

change the state of the rendering process. Although the geometric primitive approach provides rendering support, the primary emphasis is placed on unifying the interface between users and models. The definition of first-class models with clearly defined member functions enables users to manipulate models as "physical" objects. The disparate placement of importance is highlighted in figure 10.2.

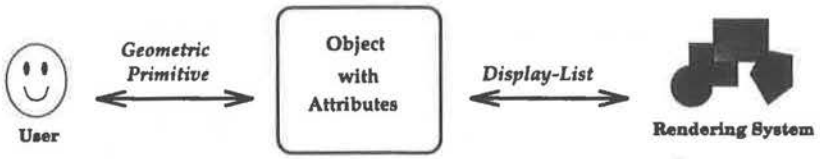

Figure 10.2: Object-User vs. Object-Render

10.1.1 The Hybrid Model

The RASP method of model creation combines the two approaches. Based on an object-oriented design, this *hybrid model* uses a unified user interface and rendering architecture. Models are first-class entities while visual characteristics are referenced in a display-list fashion. *Slots* and *ports* provides a simple method to manage and manipulate the internal structures of any model during a simulation.

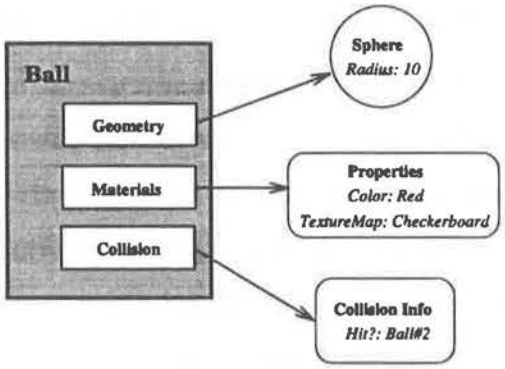

Figure 10.3: Object "Ball" with three *features*

In the hybrid model, an model is defined as a collection of "unordered" slots.[4] Each slot contains a reference to a *feature object* that controls the primary function or the regulation of a set of attributes. For example, in Figure 10.3, the model "Ball" has three slots: geometrical information, material attributes, and collision data.[5] To add additional feature objects to a model, users simply create new slots.

The hybrid model approach discourages the creation of massive models by delegating additional duties to other feature objects. Hybrid models do not define new operations and are not in themselves, a geometrical primitive. An model's shape is defined by its geometrical feature object. There are two primary benefits to this design. First, an model's shape can be easily manipulated because geometry is a not a part of an model's definition. It is a separate feature. Second, object features allow the geometry of an model to change without affecting its additional characteristic attributes. For example, if model "Ball" in figure 10.3 alters its shape, it retains its material attributes and collision data.[6]

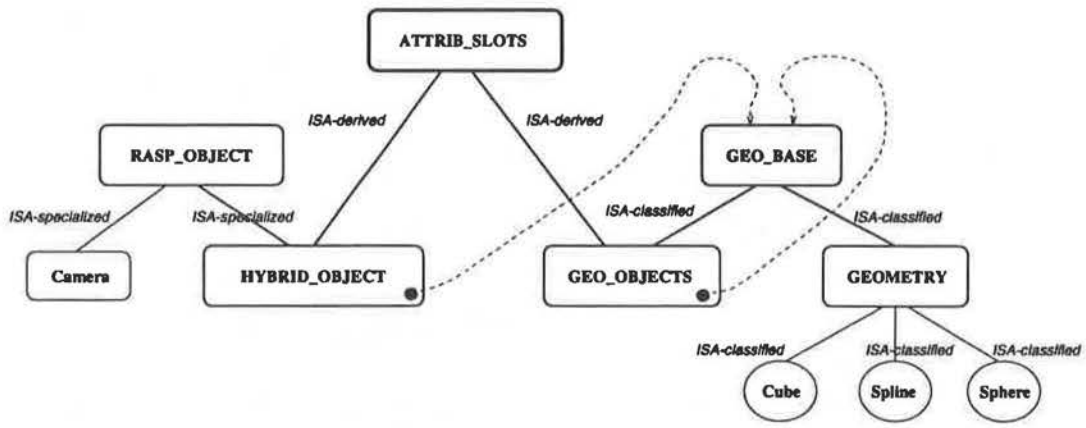

Figure 10.4: Hybrid Model Inheritance Tree

The diagram in Figure 10.4 shows the relationships of components (classes) of a hybrid model. The solid lines represent direct inheritance links, the dashed lines identify component

[4]Because slots are unordered, tags and tables are not required to access specific slots.
[5]Collision data is associated with models to facilitate collision detection operations.
[6]Collision data will probably need to be recalculated.

employments,[7] and the solid circles define object slots. A HYBRID_MODEL represents a hybrid model, while the descendants of the class GEO_BASE describe geometrical shapes. A GEO_BASE type object is a feature object in each instance of a HYBRID_MODEL. Classes derived from the base class GEOMETRY define geometric classes, while GEO_OBJECTS store collections of GEO_BASEs. This organization facilitates the hierarchical construction of composite geometries. The classes, GEO_OBJECTS and HYBRID_MODELS, inherit the ability to store feature objects in slots from the class ATTRIB_SLOTS which enables hybrid models and collections of geometrical objects to define their own feature objects. Although sets of geometrical objects are not required to define supplementary features objects, this property provides added flexibility in the design of complex hybrid models. For example, the illustration in Figure 10.5 portrays the visual and internal representation of a hybrid model composed of two spherical shapes of differing colors.

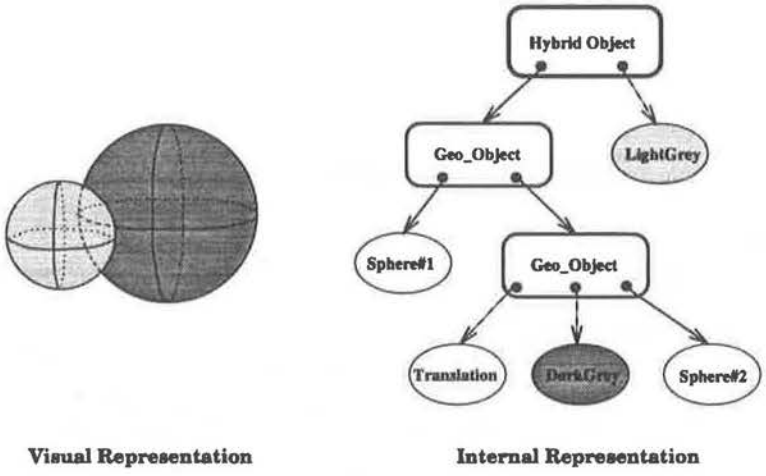

Figure 10.5: Complex Hybrid Model: Dark circles represent slots, while the arrows define the contents of the slots. Solid arrows emphasize the location of geometric slots.

Note that every slot established within ATTRIB_SLOTS does not necessarily control a single attribute or function of a model. Often, groups of similar features are clustered into one slot. For example, one of the primary slots defined within ATTRIB_SLOTS refers to a "material properties" feature object. This object manages the attributes defining a model's

[7] A class employs another class if it utilizes that class within its internal representation.

"visual" appearance. Coupling features allow changes to a model's attribute set to occur at two levels of detail. A single feature may be manipulated as an individual object or as a part of a larger set. Coupling also provides a simple organizational pattern for the management of feature objects. The hierarchical grouping of attributes reduces the complexity of managing an extensive list of features.

10.1.2 "Feature" Ports

In the hybrid model, all slots are governed by *feature ports.* Each port regulates the contents of its associated slot during the lifetime of a simulation. Adhering to the *connection paradigm* of section 7.3, these ports enable users to induce changes to slots according to a well-defined script or as a consequence of the activation of a series of indeterminate events. For example, the following set of pseudo-code changes an model's color at the halfway point of the simulation.

```
1       MaterialObject  colorR = red, colorB = blue;
2       Sphere          sph;
3
4       send sph.attribPort = colorR at time t=1;
5       send sph.attribPort = colorB at time t=15;
6
7       do simulation from t=1 to t=30;
```

Feature ports are not simple variables or member functions as the preceding lines of pseudo-code may indicate. They are first-class entities with their own set of routines that act as administrators for slots. Data is sent to a port; ports do not point to data. Additionally, the statement in lines four and five do not actually send any information to the port `attribPort`. They are actions to be performed at a particular time in the simulation. This additional layer of indirection between an action and an model's internal components structures any changes made to a model's features and ensures that slots receive the correct type of information. Figure 10.6 shows an model with its associated ports.

10.2 Data-to-image translation

The *data-to-image translation* phase uses information from the model creation phase to generate visual images by directly translating or interpreting information from the models of a scene.

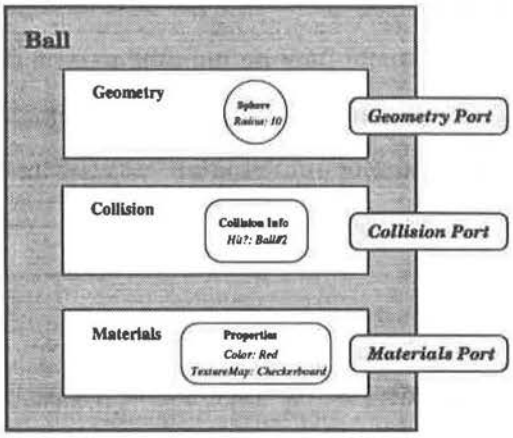

Figure 10.6: A model with its "feature" ports

As simple as this task may seem, no general approach has achieved widespread acceptance in the computer graphics community. The diverse set of requirements of rendering and geometry groups hampers the design of a common extensible interface for the two disciplines. Common difficulties include: the inability of renderers to support all model shapes and characteristics, an model's failure to produce alternative representations, and the inability to introduce new functionality and algorithms into the data-to-image translation process.

A variety of techniques have been used to administer the *data-to-image translation* phase of the visualization process. The following list, provided by [18], provides a brief summary of an assortment of methods.

- **Multiple geometries, multiple renderers**: In this approach, every renderer is knowledgeable of all types of geometries. Though simple, this method requires the development of complex renderers. In addition, extending the capabilities of the model can be arduous because of the many dependencies between the geometries and renders.

- **Conversion to common primitive(s)**: Advocated in systems developed by [88, 95, 46], this technique requires geometrical objects to be decomposed or translated into a single or common set of primitives before being forwarded to any renderer. The reduced set of primitives permits the development of moderately sized renderers. In this approach,

geometries may become overly complex because of the necessity of providing a set of methods to decompose themselves into all primitives. In addition, a renderer that can efficiently handle a larger set of primitives will not be used to its fullest capability.

- **Common interface**: In this approach, a common interface is established between the modeling and rendering components. The interface defines a standard set of routines, data formats, and data communication mechanisms that must be supported by both geometries and renderers. Renderers do not need to be familiar with geometrical types and the independent construction of geometrical objects is enhanced. Research systems described in [29, 87, 68, 63] follow this approach. The additional complexity of geometrical objects constitutes the major drawback of this method.

- **Single primitive**: This method restricts all modeling to one geometric primitive. Although this design reduces the complexity of the rendering components of a system, it unduly burdens users - they must spend considerable time editing and combining the single primitive to generate their geometric objects - and it may be difficult or impossible to produce certain geometries. Modeling testbeds described in [22, 61] adhere to this approach.

10.2.1 Multiple Geometries, Primitives, and Renderers

The RASP toolkit's data-to-image translation technique represents as a combination of the "multiple geometries, multiple renderers" and "conversion to common primitive" approaches. Each renderer specifies the type of geometric representations it can render, and every geometrical object identifies the type of geometric representations it can form. The union of these two lists defines the form used to render a geometric object. Based upon work by [18], this method promotes the independent construction of image renderers and geometric objects. New geometries can be created without the need to update existing renderers, and similarly, new renderers can be produced without knowledge of the existing geometries.

In the combined model, every renderer is a first-class entity composed of a set of rendering routines and two distinct lists. Each list is used for the data-to-image translation process. The

first list, the *primitive list*, contains the primitive geometric types that the renderer can use. Values are assigned to this list immediately after a renderer is instantiated. The second table, called the *geometry list*, stores a similar list of types, but unlike the first list, every entry in this list corresponds to a different geometric type and defines the method of data exchange between the renderer and all of its known geometric types. To render a geometric object, the renderer examines its geometry list and notifies the object of what representation type it must produce. Not containing static values, each entry of the geometry list is updated during the lifetime of a simulation. Usually performed at the beginning, every object informs the set of renderers of the type of representations they can supply. Renderers cross-reference this information against its *primitive* list to generate values for its *geometry list*. The caricature in Figure 10.7 depicts the formation of both lists for a rendering object called **Renderer**.

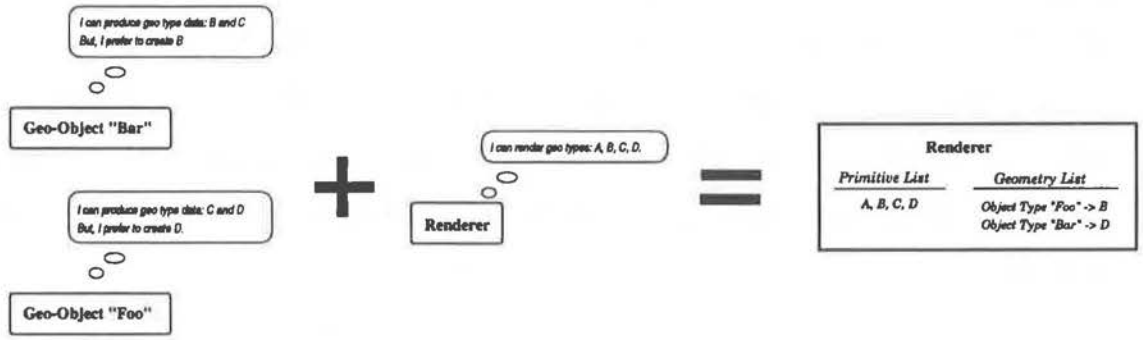

Figure 10.7: Renderer Object List Formation

Although permitting the independent construction of geometries and renderers, this dual-list scheme requires all geometric objects to support more than half the members of a common set of representation types. Any object that fails this requirement is not guaranteed to be renderable. For example, if a geometric object can not produce a particular representation type for a single-type renderer, it will not be possible to produce an image. However, if both objects are constrained to support a minimalistic set of representations, it will always be possible to create an image.

10.2.2 Image Creation

During rendering, a renderer object is passed to every geometric object in a scene. Each geometric object passes geometrical information of an appropriate type to the renderer. To determine the suitable type, the object notifies the renderer of its type and identity. Using this information, the renderer searches its *geometric list* to locate the appropriate form of information it should receive from the geometric object. Once the geometric object is told what graphic representation to produce, it generates the proper information and passes it to the renderer - this includes a complete set of material attributes. Since all this information is encapsulated within the object's material attributes slot, the material feature object is the only item that needs to be passed to the renderer. The following lines of pseudo-code identify the basic set of function calls required to generate a single image.

```
PROCEDURE drawObject() {
1     Object   ball;          /* create an object */
2     Sphere     sph;           /* create a spherical geometry */
3     Renderer   rend;          /* create a renderer */
4
5     ball.addGeometry( sph ); /* set the object's geometry */
6
7     rend.addObject( ball );  /* let render know about object, so it generate
8                                 an entry for it in its ''geometry'' list. */
9     ball.render( rend ); }
```

The following lines of pseudo-code detail the operations of `ball.render` on line 9 above.

```
MEMBER FUNCTION Object::render( Renderer rend )
{
1       /* ask renderer for appropriate data type */
2       type = rend.getType( geometryIdentity, ME );
3
4       /* generate the appropriate data */
5       representation = makeData( type );
6
7       /* send material info the renderer */
8       rend.sendMaterial( materialFeatureObject );
9
10      /* send data to renderer */
11      rend.sendData( representation );
}
```

Currently, renderers require only the geometric representation and material attributes of any model. However, since future renderers may request supplementary information, every

model passes its identity to the current renderer. This operation is exemplified in line 2 above. The value "ME", representing the model being rendered, provides the renderer with a reference to use in requesting additional data.

Chapter 11

RASP: The Implementation

When we build, let us think that we build forever.
- John Ruskin, The Seven Lamps of Architecture

This chapter discusses the implementation of the RASP toolkit. It is intended to exemplify the power and flexibility of the toolkit's design. Although the library is exclusively developed in C++, it does not preclude the development of a similar toolkit written in another programming language. The selection of C++ was driven by three motivating factors:

Abstraction capabilities To properly realize the toolkit's design, it was essential to be able to construct high-level abstractions that support the object-oriented ideology. The ability to create objects and to define relationships between various types of objects was of vital importance.

Platform availability Practicability constraints limited language selection to its availability on various hardware platforms. The importance of image synthesis to the toolkit's design encouraged the selection of a language supported by machines specializing in computer graphics application development.

Popularity To entice users to develop applications with the toolkit, it was imperative to select a language with widespread acceptance. Many users are easily discouraged from using libraries and tools constructed with uncommon languages.

C++ was the only language that satisfied all three design criteria.

11.1 Class Design

In C++, the main abstraction mechanism is a *class*. A class enables users to define their own data types. Classes serve as templates from which objects (data types) are created. Every class consists of a set of data members and member functions. Data members represent the state variables of an object while member functions represent operations that are applied to data members. Users invoke member functions to alter the state of an object. A class' *interface* is defined by the number and type of member functions it possesses.

11.1.1 Member Function Classification

Although a class' usefulness is primarily judged by its design and functionality, a class is deemed to be useless if it is unreadable. If users can not rapidly correlate an association between a class' member functions and its functional objectives, the class will be difficult to use. Obscure or abstract public interfaces obfuscate the intended meaning of a class. To clarify a class' implementation, every header file[1] in the RASP toolkit segments each class' set of member function into six distinct categories. This scheme enables users to identify member functions with commons aims and to determine swiftly the general purpose of individual member functions.

Influenced by [51]'s classification plan, class member functions are categorized as follows:

- **manager**: The construction and destruction of class instances are governed by manager member functions. Management activities, such as initialization, assignment, memory management, and type conversion, are performed by these of functions.
- **interface**: Explicit duplication and equality testing operations are defined as interface members. These member functions provide users with a variety of methods to copy a single object or test the equality of two distinct objects.
- **access**: Any function that enables users to access private data members are incorporated into this category. Functions of this type are usually preceded with the prefix "set" or "get". Predicate operating functions are also included in this set.

[1] In C++, header files are distinguished by ".h" endings.

- **implementor**: Functions that serve to invoke the capabilities associated with a class' abstraction are placed in this category. In general, the behavior of a class object is altered when functions of this type are activated.

- **helper**: All protected or private member functions are defined to be helpers. These functions are not intended to be invoked by users. They serve to perform hidden auxiliary tasks.

- **operator**: Any function that operates to work on instantiated objects of a class are placed in this last category. All testing functions, pseudo-math like functions, and logical operation functions are listed under this category.

- **port**: Port member functions are unlike all the previous member functions because they do not induce state changes or return the values of data members. They return ports ("in" and "out") which are similar to member functions. Ports offer users an alternative class interface more appropriate for simulation.

The result of applying this classification scheme to a class' member functions is illustrated in the following class definition. The general aim of each member function is quickly determined from its classification grouping.

```
class Rectangle {
    long x,y, x2, y2;              /* data members */

  public:                          /* member functions */

    /* manager functions */
    Rectangle();
    Rectangle( long, long, long, long );

    /* interface functions */
    Rectangle* copy() const;

    /* access functions */
    long       left() const                          { return x; }
    long       height() const                        { return y2-y; }

    /* implementor functions */
    void       translate( const long, const long );
    void       scaleFromCenter( const );
```

```
    /* operator functions */
    Rectangle& operator =( const Rectangle& );
    Rectangle  operator *( const long );

    /* port functions */
    OPort*     outX();
    IPort*     inX();

  private:
    /* helper functions */
    void       initialize();
};
```

11.1.2 IdentifyInfo Class

Many regular classes and abstract classes in the RASP toolkit inherit information and properties from the base class **IdentifyInfo**. This base class supplies its inheritors with data members and member functions to manage their identity. For clarity, most references to the base class **IdentifyInfo** are omitted from many of the diagrams and discussions in this chapter. Readers are advised to examine Figure D.1 for a complete diagram of the toolkit's inheritance tree.

11.1.3 RogueWave Classes

To alleviate the creation of many common data structures, the RASP toolkit uses various classes from the RogueWave class libraries. The RogueWave library set[73], developed by Rogue Wave Associated, is composed of Tool.h++, Math.h++, and Matrix.h++. Although the header files and documentation for these libraries are slightly obscure, their usage is highly recommended. Further references to RogueWave classes and data structures are omitted from this chapter for clarity.

11.2 World Modeling

This section describes the basic concepts and essential programming constructs found in all RASP built simulations. It provides a brief overview of the four required components of a visual simulation: **Setting**, **Cameras**, **Renderers**, and **HybridModels**. A fully-functional simulation can not be created without at least one instance of each these components.

11.2.1 The Setting

The heart of every RASP simulation, the **Setting** describes the global environment, initiates the progression of time, and invokes rendering routines. Users specify the parameters and contents of the setting. By analogy, the setting is the stage on which all the models of a simulation perform. A stage is empty unless the director places objects, lights, and actors on it.

Users must specify at least one **Renderer**, **Camera**, and **Window** to synthesize static images of the simulation. The camera holds the viewing parameters of the rendering. One may visualize the parameters of the camera as defining the location where the camera sits relative to a stage where actors are performing.[2] The window defines the size and type of user-interface window to be displayed on users' screens and it delimits the size of viewing screens for cameras. This latter feature is important because **Cameras** do not define the size of their own viewing screen. Extracting the size of the viewing screen from a camera's definition permits users to construct independently new camera types and window types.

The following sample program, *redsphere.c*, creates a simple image of one red sphere. It explicitly renders the sphere once, then quits.

```
main()
{
  /* Create a setting and a renderer */
  RaspSetting  world;
  GLRenderer3D glRend;

  /* create a user interface window */
  fRect      windRect( 0, 0, 200, 200 );
  GLWindow3D wind( windRect, "RedSphere Example" );
  wind.open_window();
```

[2]RASP cameras may be place anywhere, even on the stage

```
  /* create a camera */
  Camera kamera( "Example Camera" );
  kamera.setView( Point3(20,35,110), Point3(0,0,0) );
  kamera.associateWindow( wind );
  kamera.associateRenderer( glRend );
  world.addObject( kamera );

  /* create a red sphere */
  Sphere         sph1( 10. );
  HybridModel    obj1( Point3(0,0,0), sph1 );
  world.addObject( obj1, BASIC_RED );

  world.renderAll();
}
```

The first line of the program initializes the object variable `world` to an instance of a `RaspSetting`. The second line creates an instance, called `glRend`, of a `GLRenderer3D`. This object represents a renderer that utilizes three-dimensional GL-library[3] calls to synthesize its images.

The next group of lines creates a `GLWindow3D` window. The argument `windRect` represents the size and location of the window. The following set of lines creates a `Camera` named "Example Camera". Along with its instantiation, the camera's viewing parameters, associated window and renderer have been set. Notice in the final line that the `kamera` is added to the `world`. Without this statement, the `world` would not be able to draw anything.

The last set of statements creates a red sphere. Just like the camera, the ball is added to the `world` because all objects must be placed into the setting if they are to be recognized. However, objects need not be placed into the setting at the beginning of a simulation. They may be added at any time during the course of a simulation. The final statement informs the setting to render everything.

11.2.2 HybridModels

In the last example, two separate statements were used to create one spherical object. Unlike standard object-oriented graphics libraries, RASP distinguishes between objects and geometries:

- Attributes of **HybridModels** represent those aspects of an object that are independent of its geometrical configuration. Properties such as surface color, position in world space,

[3]GL (Graphics Library) is a registered trademark of Silicon Graphics, Inc.

texture, and identity are examples of non-geometric attributes.

- Geometrical classes, such as **Sphere**, only contain information directly associated with its geometric shape because direct incorporation of non-geometric attributes only increases the complexity of a class' design. For example, in RASP, all **Geometry**-based classes are constructed within a local reference frame. Their position in world space is not defined in their class designs.

In the RASP environment, instances of either class can not stand alone. Only via the combination of both classes can an object be useful. An object without geometry does not have form while a geometry without an associated object does not have real world properties. The many benefits of this separation are:

- independent construction of (complex) geometrical classes.
- animation of an object's geometry allowing an object to changes its form and shape during run-time.
- multiple objects with (pointers to) the same geometrical configuration. Animating the common geometrical form will change all the shapes of the relating entities.
- an efficient hierarchical construction of geometric objects. Non-geometrical informational attributes are not repeatedly stored within the multiple levels of the hierarchy.

The following example illustrates the creation of two geometric entities: a cube and a cubic spline. For clarity, all statements concerned with the construction and control of cameras, windows, and renderer have been omitted.

```
main()
{
  /* Create a setting */
  RaspSetting    world;

  /* create a cube-oid object (5x5x5) at the global origin */
  Cube           cube( 0, 0, 0, 5, 5, 5 );
  HybridModel    obj1( Point3(0,0,0), cube );

  /* set color to RED and add to world */
  obj1.setColor( BASIC_RED );
```

```
  world.addObject( obj1 );

  /* create a cubic spline at location (10,10,0) in global space */
  Basis          sBasis( CUBIC_BASIS );
  Spline         spl1( 10, sBasis, FALSE );
  HybridModel    obj2( Point3(10,10,0), spl1 );

  /* set color to BLUE and add to world */
  world.addObject( obj2, BASIC_BLUE );

  world.renderAll();
}
```

After creating the `world`, the first group of statements creates a cube of length, width, and height of five units, assigns this geometry to `obj1`, and places it at the origin in global space. The following statements assign a RED color to `obj1` and adds it to the setting.

To create a spline-based object, `spl1` a basis function (class), the ten control vertices spline is assigned to `obj2`, which is placed at location $(10, 10, 0)$ in global space. Finally, `obj2` is assigned a BLUE color while it is being added to the world.

Note the difference between the two `addObject` statements. In the first call, no color argument is given. The object is added to the setting without declaring a color. This small example illustrates an important feature found throughout the RASP library - there are many ways (function-calls) to produce matching results. Multiple methods permits the design and development of many diverse programs.

11.2.3 Multiple Views

The RASP architecture does not limit the number of **Renderers**, **Windows**, **Cameras**, or **HybridModels** present in a setting. It does not even place a limit on the number of **RaspSettings**. However, the current toolkit does not support the parallel execution of multiple settings. Multiple setting can only be executed in tandem. Therefore, the usage of numerous setting is not advised at this point.

The following example illustrates a simple model with two cameras, two renderers and two windows. Please note the appearance of several member function calls not shown in the previous examples. These additional functions illustrate the high degree of control users possess in altering and defining the parameters of a model.

```
void main()
{
  /* create a setting and two renderers */
  RaspSetting   world;
  GLRenderer3D  glRend;
  OptikRenderer opRend;

  /* camera attributes */
  fVector viewup (-1.0, -1.0, 0. );
  dAngle fovx = 90., fovy = 90.;

  /* create a 3-Dimensional GLWindow */
  fRect       w( 100., 100., 300., 200 );
  GLWindow3D *wind = new GLWindow3D( w, "Test", TRUE );
  wind->open_window();
  wind->setColor( DARK_GREY );
  wind->clear_window();

  /* create a basic window */
  fRect   w2( 0, 0, 100, 100 );
  Window *wind2 = new Window( w2, "Test" );
  wind2->setColor( DARK_GREY );

  /* create a camera */
  Camera camera( "Main Camera" );
  camera.setView( Point3(20,35,110), Point3(0,0,0), viewup, fovx, fovy );
  camera.setClipPlanes( .001, 3500. );
  camera.associateWindow( wind );
  camera.associateRenderer( glRend );
  camera.wind_Set_OrthRt( .5 );

  /* create another camera */
  Camera camera2( "Other Camera");
  camera2.setView( Point3(0,30,100), Point3(0,0,0), viewup, fovx, fovy );
  camera2.setClipPlanes( .1, 1200. );
  camera2.associateWindow( wind3 );
  camera2.associateRenderer( opRend );
  camera2.wind_Set_OrthRt( .5 );

  /* add cameras to the setting */
  world.addObject( camera );
  world.addObject( camera2 );

  /* create a spherical object */
  Sphere         sph1( 10. );
  HybridModel    obj1( Point3(0,0,0), sph1 );
  world.addObject( obj1, BASIC_RED );

  world.renderAll();
}
```

11.3 Port Classes

This section discusses the design of RASP's port classes. Unlike many of the classes found in the toolkit, port classes are used only to assist in the development of other classes. They serve as tools to construct objects that adhere to the connection paradigm.

11.3.1 Inheritance Tree

As described in section 7.3.3, RASP inports and outports are similar to standard class member functions. Both regulate the value of data members and direct actions performed by the class. However, ports differ in that they are first-class, unidirectional, and able to respond to queries. Functionally, inports and outports are equivalent. Both ports access data members, associate conditional tests with data members, and invoke class actions. They differ only in direction. This commonality permits the construction of a basic **Port** class (see Figure 11.1) from which both ports inherit data and operations.

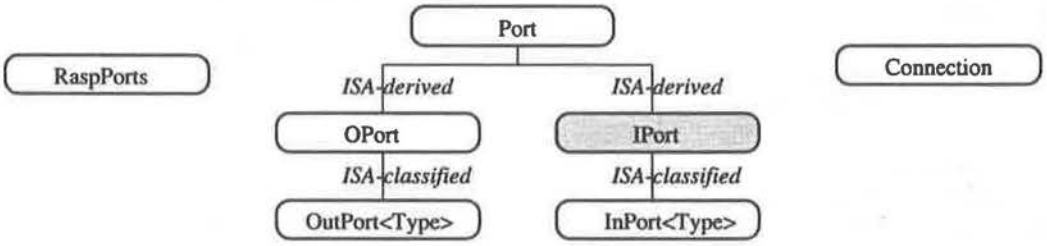

Figure 11.1: Port Hierarchy

The classes `OutPort<Type>` and `InPort<Type>` in Figure 11.1 are parameterized types.[4] They represent special port classes formulated especially for C++. Each class facilitates the construction of ports which maintain references (pointers) to class member functions. Each class is parameterized because C++ does not permit the construction of simple generic member function references. All reference declarations must state explicitly the class type from which the member function it references is defined.

[4]See section B.2.2.

11.3.2 Point Class

The following header file for the class **Point** exemplifies the definition of a class with ports. The data members `outPort` and `InPort<Point>` facilitate the creation of outports and inports. The constants declared in both `typedef` statements supply ports with simple identifiers. The port function `outThis()` returns a port which references the class itself.

```
/*****************************************************************************
  class POINT definition
 *****************************************************************************/
class Point {
 private:
  typedef enum {
    OP_X,
    OP_THIS
    }; /* outports identifiers */

  typedef enum {
    IP_X
  }; /* inports identifiers */

  RaspPorts      outPort;
  InPort<Point> *inPort;

 protected:
  double         x, y;

 public:
  /* manager functions */
  Point( const double, const double );

  /* access functions */
  void    setX( const double );
  double  getX( void ) const;

  /* port functions */
  OPort*  outX();
  OPort*  outThis();
  IPort*  inX();
};
```

The following segment of code implements the constructor function for the class defined above. In addition to setting the values of the data members, the constructor creates the class' ports, associates data members and member functions with individual ports, and identifies the data type managed by each port.

```
Point::Point( const double xVal, const double yVal ): x( xVal ), y( yVal )
```

```
{
  outPort.setNumOutPorts( 1 );
  outPort[OP_X]->setVar( &x, RASP_DOUBLE );

  inPort = new InPort<Point>[1];
  inPort[IP_X].setVarId( RASP_DOUBLE );
  inPort[IP_X].setHandler( this, &Point::setX );
}
```

It should be noted that the similarities between ports and member functions are not limited to functionality alone. Both structures are subject to the rules of inheritance. Subclasses can alter the purpose and design of all inherited ports.

11.4 Temporal Tools

This section describes the implementation of RASP's set of temporal primitives, which were discussed in section 9.2. All primitives, except for processes, are described in detail. A discussion about RASP processes is relegated to section 11.6.

11.4.1 Events

Every RASP *event* type is a subclass of the base class **EventBase** (Figure 11.2). This class defines three important functions which all event types must support: *stateEvent*, *endEvent*, and *doEvent*. The first function initiates actions to initialize an event. Exemplary *startEvent* actions include testing port availabilities, comparing ports types between links, and establishing links between ports. The second function simply cancels all stateEvent actions, such as eliminating link formations between ports. The third function executes the event's objective. Exemplary *doEvents* include passing temporal values to ports (**TimeEvent**), transferring values from one port to another (**Event**), and informing ports to test their states (*StateEvent*).

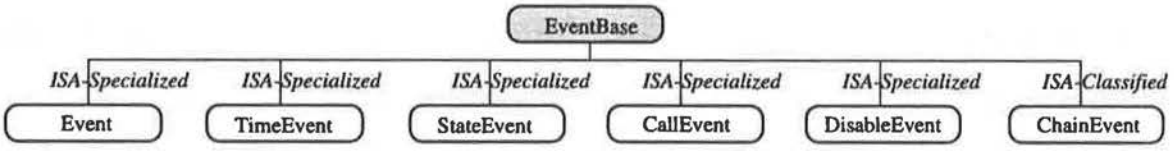

Figure 11.2: Event Hierarchy

The following segment of code illustrates the usage of a variety of RASP event types. The number and type of parameters an event receives is dependent upon its objective.

```
// create event to call target port with no arguments
CallEvent  ev1( col1->inRun() );

// 2a -> create data transaction event
// 2b -> create event to a call a procedure with no arguments
Event      ev2a( spl1->outMaxParam(), evol->inFinishVal() );
Event      ev2b( proc );

// 3a -> create event to pass time value to target port
// 3b -> create event to a call a procedure with one arguments
TimeEvent  ev3a( evol->inCalcValue(), RS_REL_TIME );
```

```
TimeEvent  ev3b( proc2, RS_ABS_TIME );

// create event to call activity when source port target alters state
StateEvent ev4( obj1->outCollision(), act );

// create a chain event composed of
ChainEvent
```

Conceptually, all RASP event types could have been constructed as one universal event. Its exact functionality would be extracted from an examination of its *argument list* during instantiation. However, there are three major arguments against the development of one universal event. First, one event type occupies more memory than any single-purpose event type. For large simulations requiring many events, an excessive use of memory can impede system performance. Second, augmenting the functionality of an universal event becomes an onerous task. Users would need to manipulate unnecessarily complex structures simply to extend the capabilities of the event. Third, the usage of one event type hinders the rapid analysis of a simulation. Users must examine the argument list of every event to determine their purposes.

11.4.2 Activities & Processions

RASP's **Activity** class and **Procession** class inherit data members and member functions relating to temporal actions from the base class **Timing** (Figure 11.3). This base class provides operations to establish and compare the timing information (temporal interval and granularity) of individual activities and processions. Because the toolkit's current design defines only one type of an activity and one type of procession, the class **Activity** and class **Procession** are not abstract. However, this does not preclude the development of abstract classes for activities and processions. Future enhancement to the toolkit's design may compel the creation of such classes.

The following example illustrates the creation of an two activities and one procession.

```
/* create an activity with time span from 5 to 10 */
Activity act( ''Example Activity'', 5, 15 );
act.addInitEvent( evt1 );
act.addActEvent(  evt2 );
act.addFiniEvent( evt3 );

Activity act2( ''Example Activity#2'', 10, 25 );
```

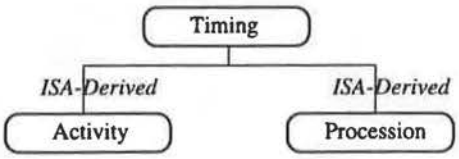

Figure 11.3: Activity Hierarchy

```
act.addActEvent( evt2 );
act.addActEvent( evt4 );

/* add activities to procession */
Procession seq1( ''Example Procession'' );
seq1.addActivity( act );
seq1.addActivity( act2 );
```

It is important to note that the toolkit does not restrict the quantity or type of events associated with each activities. Therefore, any event may simultaneously belong to more than one category and to several activities.

11.4.3 Examples

This section concludes with two examples to exemplify the usage of events, activities, and processions. Each example defines the motion path of a spherical object. Both paths are illustrated in Figure 11.4. The route in the first example is defined by a linear interpolation of three points. In the second example, the parametric values of a spline object generate the path for the object in motion.

Linear Interpolating Path

This example is divided into two routines. The first routine, `main`, is responsible for creating a setting and running the simulation, while the second procedure, `initWorld`, is assigned to create all the temporal actions of the model.

```
main()
{
  /* create a setting */
  RaspSetting world;
```

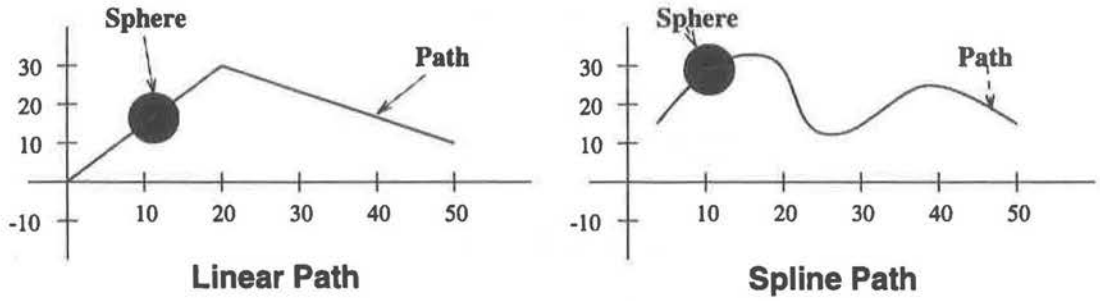

Figure 11.4: Motion Paths

```
  /* create objects, events, activities, and processions */
  initWorld( world );

  /* run the simulation */
  world.run();
}

void initWorld( RaspSetting *world )
{
  /* create a light blue sphere */
  Sphere         sph( 10. );
  HybridModel    *obj = new HybridModel( Point3(0,0,0), sph );
  world.addObject( obj, LIGHT_BLUE );

  /* create an Point3 evolution object */
  Point3   pt1( 0, 0, 0 ), pt2( 20, 30, 40 ), pt3( 40, -10, 30 );
  ptEvolve evol( pt1, pt2, 10 );

  /* create timeEvent and data transfer event */
  TimeEvent evt1( evol.inCalcValue() );
  Event     evt2( evol.outCurVal(), obj->inSetPosition() );

  /* create an event to alter the evolution start and end values */
  ChainEvent evt3;
  evt3.addEvent( pt2.outThis(), evol.inBeginVal() );
  evt3.addEvent( pt3.outThis(), evol.inFinishVal() );

  /* create an activity, and add both events */
  Activity move1( 5., 15. );
  move1.addActEvent( evt1 );
  move1.addActEvent( evt2 );

  Activity move2( 15., 25. );
  move2.addInitEvent( evt3 );
  move2.addActEvent( evt1 );
  move2.addActEvent( evt2 );
```

```
  /* create a procession, add the activity, and add procession to setting */
  Procession seq1( ''Move Ball'' );
  seq1.addActivity( move1 );
  seq1.addActivity( move2 );

  world.addProcession( seq1 );
}
```

The first set of statements in `initWorld` defines a blue spherical object. The next set of statements defines three distinct points and an evolutionary object of type `ptEvolve`.[5] Given an initial and final value, `ptEvolve` uses a linear interpolation scheme to generate a set of intermediate values. Its last argument represents the number of interval values that are to be calculated.

The next set of declarations defines two distinct events. The first statement creates an event to pass time to `evol`. The omission of a second argument to `TimeEvent` indicates that `evol` requires "local" (not "absolute") time values. The "local" time will be determined from the event's (yet to be defined) associated activity. The second statement creates an event to transfer data from `evol` to `obj`. This transaction will set the spatial location of the spherical object to equal the value produced by the evolutionary object.

After composing events `evt1` and `evt2`, the next set of statements creates a `ChainEvent`. This event is composed of two separate `Events`. When triggered, the tandem events will alter `evol`'s initial and terminal value. It is important to note that no explicit `Event` statements were required to generate the individual events. The ability to overload functions in C++ enables the member function "addEvent" to accept explicit events or special sets of arguments. In this case, "addEvent" automatically generates an `Event` from the combination of the two ports. The circumvention of basic declarations enables users to design rapidly simulations and reduce code size. For example, the following two statements are equivalent:

```
1=>   evt3.addEvent( pt2->outThis(), evol->inBeginVal() );
      Event aaa( pt2->outThis(), evol->inBeginVal() ));
2=>   evt3.addEvent( aaa );
```

The first statement does not require the user to define explicitly an `Event`. Users may utilize

[5] `ptEvolve` is a type definition of the template class `Evolve` with `Point3` passed as the parameterization argument.

either method to add events to a `ChainEvent`. Programmers may wish to utilize the second method if they require to reference the basic event more than once.

Following the event definitions is the activity declarations. The activities `move1` and `move2` are very similar. Each endures for ten temporal units and activates identical "acting" events. However, only `move2` defines an "initial" event. This one time only activation event will alter `evol`'s interpolation range from (`pt1`,`pt2`) to (`pt2`,`pt3`) .

The final declaration set performs three tasks. It defines a procession, insert activities into the procession, and adds the procession to the world.

Spline-Based Path

The following example defines an alternative motion path for the spherical object. It creates one activity, composed of several events, to move the object along a spline-based route. The structure of this code is very similar to the one utilized in the previous example.

```
main()
{
  /* create a setting */
  RaspSetting world;

  /* create objects, events, activities, and processions */
  initWorld( world );

  /* run the simulation */
  world.run();
}

void initWorld( RaspSetting *world )
{
  /* create a light blue sphere */
  Sphere          sph( 10. );
  HybridModel    *obj1 = new HybridModel( Point3(0,0,0), sph );
  world->addObject( obj1, LIGHT_BLUE );

  /* create a dark green spline with 10 CVs */
  Basis sBasis( CUBIC_BASIS );
  Spline *spl = new Spline ( 10, sBasis );
  HybridModel  obj2( origin, spl );
  world->addObject( obj2, DARK_GREEN );

  /* create a ''double'' evolution object */
  dEvolve  *evol = new dEvolve( 0, 0, 20 );

  /* set evolution finish value to spline's maximum parametric value */
```

```
  Event     evt1( spl->outMaxParam(), evol->inFinishVal() );

  /* create a chain event */
  ChainEvent evt3;

  /* send time to evolution object, then set spline's parametric value to
     value of evolution, then set spherical object's position to equal
     spline's parametric position */
  evt3.addEvent( TimeEvent( evol->inCalcValue() ));
  evt3.addEvent( evol->outCurVal(), spl->inParamVal() );
  evt3.addEvent( spl->outParamPos(), obj1->inSetPosition() );

  /* create an activity, and add both events */
  Activity move( 2., 22. );
  move.addInitEvent( evt1 );
  move.addActEvent( evt3 );

  /* create a procession, add the activity, and add procession to setting */
  Procession seq1( ''Move Sphere'' );
  seq1.addActivity( move );

  world->addProcession( seq1 );
}
```

The first set of statements in `initWorld` create one spherical and one spline-based object. Each entity is placed at the origin, given a distinct color, and added to the `world`. The next statement defines a pointer to an object of class `dEvolve`. This referenced interpolation object is responsible for generating successive values (of type "double") from zero to a currently unspecified number.

The next collection of declarations defines two events. The first event, `evt1`, defines an action to modify the terminal or maximum value of `evol`. The second event, `evt3`, is a multiple action event. When triggered, it will execute, in succession, three events. A visual illustration of the action performed by each event is portrayed in Figure 11.5.

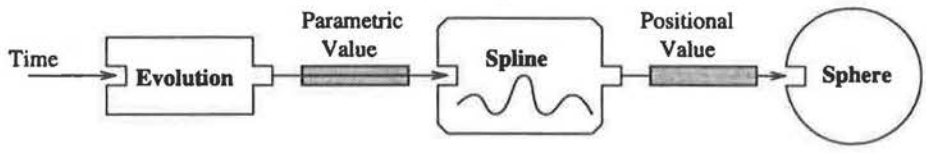

Figure 11.5: Spline Path Events

1. A temporal value is transmitted to the evolution object.

2. A value from the evolution object to transferred to the spline object. This value represents an index into the spline's parametric space. Give this index, the spline will generate a position in three-space.

3. The spline's three-space value is transferred to the sphere. This value will be utilized to set the sphere's spatial location in the setting.

After creating all the events, one activity, entitled `move`, is declared. It is characterized by one initializing event, one active event, and a duration of twenty temporal units.

The last set of statements defines a procession and associates it with the setting. Labeled "Move Sphere", this procession is declared to have one activity and no special temporal attributes.

Direct and Indirect Referencing

In the last two examples, some objects were referenced indirectly (via pointers). Without these references, both examples would have failed to produce a valid simulation. Most likely, the programs would have crashed or generated obscure results. The source of this problem is lodged in the manner that the C++ language defines and controls the memory associated with an object. Objects having "local" scope - directly referenced objects - fail to sustain their allocation of memory when their local environment disappears. Therefore, any reference to an object beyond the object's scope proves "undefined". Undefined references arise in RASP simulations if events attempt to reference ports of extinct objects. It is important to remember that all ports are directly linked with the objects they serve, and objects and their ports expire together. Therefore, when defining variables, the following general rule should be followed:

> *Pointers references should be generated for any objects having one or more of their "in" or "out" ports referenced in any type of an event.*

11.5 CHRONOS

This section discusses the design of RASP's simulation kernel, **Chronos**. This class interacts with the **Processions** of a simulation to control the flow of time. Currently, the class **Chronos** is instantiated and employed by the class **Setting**. Therefore, unless users explicitly wish to alter the behavior or functionality of the simulation kernel, most users will never directly use or invoke operations of the kernel. However, future modifications and enhancements to RASP's design may increase the interaction between users and the kernel.

The following C++ header file defines all the member functions of the class **Chronos**. Functions enable users to reset the simulated clock time, adjust the number of **Processions** the kernel controls, and initiate the start of the simulation.

```
/*************************************************************************
  class CHRONOS definition
 *************************************************************************/
class Chronos {
 protected:
  double             globalClock;
  double             globalTimeStep;
  GSlist(Procession) pList;

 public:
  /* manager functions */
  Chronos( double=0., double=1.0 );

  /* access functions */
  double       getWorldTime() const   { return globalClock; }
  void         resetClock( double );
  void         adjustStep( double );

  void         addProcession( Procession* );
  Procession*  getProcession( char* );
  void         removeProcession( Procession* );
  void         removeAllProcessesions();

  /* implementor functions */
  void         run( RaspSetting* );

 private:
  /* helper functions */
  double       getNextProcessionStep();
  void         advanceTime( double );
};
```

The following segment of code shows **Chronos**' member function **run**. The function iterates

a simple set of steps to advance the simulated time of a simulation. The list, **eventCameraList**, contains references to all the cameras in the simulation which are required to synthesize images at every new time step. Although this scheme increases the functionality of the kernel, it optimizes the system's performance and minimizes users' modeling efforts. The widespread usage of cameras which update at every time step justifies the insertion of this data structure into the routine.

```
void Chronos::run( RaspSetting *world )
{
  GSlist(Camera) *eventCameraList;
  double nextTime, advanceVal;
  Bool bLoop;

  eventCameraList = world->eventCamerasOnly();       /* get event cameras */
  advanceTime( 0 );

  while( TRUE ) {
    /* get the next value of time to procession towards */
    if ((nextTime = getNextProcessionStep()) == STOPTIME)
      break;

    /* don't advance time too fast */
    if ((advanceVal = nextTime) > globalTimeStep)
      advanceVal = globalTimeStep;

    bLoop = TRUE;
    while (bLoop == TRUE) {
      if (globalClock + advanceVal >= nextTime) {
        bLoop = FALSE;
        advanceVal = nextTime - globalClock;
      }
      globalClock += advanceVal;
      advanceTime( globalClock );

      /* tell update event cameras to render */
      for(int i=0; i<eventCameraList->entries(); i++)
        eventCameraList->at(i)->doSnapShot( globalClock );
    }
  }
}
```

11.6 PROCESSES

This section outlines the design of RASP processes. Readers are forewarned that process creation in C++ is not an easy task. Since C++ does not support co-routines,[6] users are required to incorporate mandatory "support" constructs into their process designs. Fortunately, users are not required to develop these additional constructs. All processes are developed with the assistance of previously defined structures and a set of essential guidelines. Adherence to these rules is mandatory.

The rules of process creation are very precise. They require users to utilize a basic set of structures and commands when developing processes. They do not restrict the behavioral development of a process. Only the structure of a process is confined to a standard set of operations. Processes are not obligated to alter the state variables or operations of any model. Attempting to stray from these prescribed regulations is not recommended. Irregularly developed processes may produce unwanted consequences.

11.6.1 ABSTRACT CLASS

Every process of a RASP simulation model must be an instantiation of a process-type class. This class must be a descendant of the abstract class **Process**. The class **Process** provides all process-type classes with a collection of important routines and data members. This class also defines a list of member functions which must be supported by all derived classes. Referred to as *pure virtual functions*, these routines can not be left undefined. The absence of one or more of these member functions will evoke errors during program compilation.

The abstract class **Process** declares that it is essential for all process-type classes to provide definitions for three particular argument-free member functions. They are as follows:

```
/* pure virtual implementor functions */
virtual void initialize() = 0;
virtual int  body() = 0;
virtual void finish() = 0;
```

[6] The usage of P-Threads *may* enable users to utilize structures which are functionally equivalent to co-routines.

VOID INITIALIZE()

This routine is called upon by the RASP kernel when a process is to be activated. It is not invoked during process instantiation. Any class variables explicitly defined to control a process' behavior is to be initialized within this routine. All data members which support the interface between the process and simulation kernel is to be initialized within the process' class constructor.

INT BODY()

This routine is the most important member function for a process. It defines the behavior of the process from start to finish. Once this routine terminates, the process will end. Conceptually, this routine need only be invoked once. Whenever a process must suspend its thread of operation, it should use coroutines. If C++ inherently supported coroutines, a sample body routine could have been written as follows:

```
void MyProcess::body()
{
  cout << "Can't" << endl;
  hold( 3 );
  cout << "touch" << endl;
  hold( 6 );
  cout << "this" << endl;
}
```

However, since coroutines are not directly supported by the C++ language, this routine will be invoked multiple times. Immediately after each invocation, the flow of execution will leap to the line following the line that suspended it. This is accomplished with the assistance of a large `switch` statement containing multiple `goto` statements. However, for this scheme to work, users must set a particular variable to a value representing the line after the suspension statement.

```
JumpTo MyProcess::body()
{
  /* this switch statement is mandatory */
  switch( jumpLine ) {
  case JUMP_1:
    goto JMP_1;
    break;
```

```
  case JUMP_2:
    goto JMP_2;
    break;
  };

  cout << "Can't" << endl;
  hold( 3 );
  return( JUMP_1 );

 JMP_1: cout << "touch" << endl;
  hold( 6 );
  return( JUMP_2 );

 JMP_2: cout << "this" << endl;
  return( NO_JUMP );
}
```

In this example, **body** contains two "hold" statements. These statements define the duration of simulation time that must pass before the next statement after each "hold" command is to be executed. A **return** statement must immediately follow each "hold" statement. This statement enables the process to suspend temporarily its thread of execution. Each **return** statement must be provided one argument of type **JumpTo**. This enumerated type defines all the possible values this argument can receive. The predefined enumeration is defined in the base class **Process**. This argument notifies the routine as to where it is to continue execution when it is restored.

The switch statement at the beginning of this routine is used to leap over segments of code. It enables this routine to believe that all of its "hold" statements are coroutines. It springs over fragments of code according to the value of the variable **jumpLine**. This variable is redefined every time a **return** statement is invoked.

VOID FINISH()

This routine will be invoked when a process is to be terminated or extinguished. It should contain all declarations ending a process' behavior. It is important to note the difference between a terminated and an extinguished process. A terminated process can restart its thread of execution after re-initialization while an extinguished process can not. Any memory associated with an extinguished process has been freed up. All references to extinguished processes are considered to be *dangling*.

11.6.2 Process Ports

There are two ways a process can communicate with its environment. It can explicitly reference external sources or it can make use of its ports. Although external references provide direct links with environmental sources, they hinder the development of independent processes. The formation of dependent links, created by external calls, constrains a process to make strong assumptions about the environment and the sources it references. These problems are not evident in processes communicating only through ports. Ports assist in the reduction of dependent links and direct reference calls. Ports are also very helpful is suspending and restoring the state of a process.

It is important to note that each method of communication has its own set of relative merits. There does not exist a straightforward procedure to choose one method over the other. The selection process is entirely context dependent. As a general guideline, ports are most useful when a process does not need to be concerned with the source of its external information. When the value of information is more important than the entity generating it, the port communication mechanism should be used. However, if the source of information is also very important, direct referencing may be the better choice.

The ports of a process and the ports of any other object within the toolkit are created in an identical fashion. Every "in" port requires the address of a class data member and every "out" port requires the service of an internal class member function. However, port processes and non-process ports are not completely equivalent. The ports of a process possess extra functionality not inherent in standard ports. For example, process ports provide additional query functions. Without these additional routines, it would not be possible to employ all the capabilities of a process.

Many process-oriented modeling situations require a process to temporarily halt its progress until it has obtained a particular resource. The process will resume its thread of execution immediately after the resource is freed or found to be available. The following routine exemplifies a process waiting for a valid value from one of its input ports.

```
int WaitProcess::body()
{
  /* this switch statement is mandatory */
```

```
  switch( jumpLine ) {
  case JUMP_1:
    goto JMP_1;
    break;
  }

  cout << " Waiting... " << endl;

 JMP_1: /* waiting for value from port */ ;

  if ( inPort[IP_VAL].getValue() ) {
    cout << " Got It! " << endl;
  }
  else {
    hold();
    return( JUMP_1 );
  }

  return NO_JUMP;
}
```

This process will declare itself to be "Waiting...", until it obtains a value for the port `inPort[IP_VAL]`. Once a value has been obtained, the suspended process will produce the statement "Got It!". The key feature to note in this example is the `if-else` clause. The `if` expression queries the port to determine if it has obtained a value. If the expression evaluates true, the process will continue its execution. If not, it will invoke the `else` clause. This clause will suspend the process until re-activation. In this case, a re-activation signal will originate from the port `inPort[IP_VAL]`. This port is automatically notified that it is responsible for resuming the process when it receives the `getValue` query and returns a value of "FALSE". It is important to note that it is not the responsibility of the process to query continually the port to ascertain if a valid value has been obtained. The process will remain in a waiting state until notified by the queried port.

11.6.3 Relationship with Activities

The following fragment of code illustrates the relationship between processes and activities.

```
Process  pr1, pr2;                  /* create two processes */
Activity move1( 5, 10 );            /* activity with duration of 5 units */
move1.addProcess( pr1 );
Activity move2( 7 );                /* instantaneous activity */
move2.addProcess( pr1 );
```

Processes `pr1` and `pr2` will terminate differently. Assuming the absence of intervention from external sources, `pr1` will expire conjunctively with activity `move1` and `pr2` will terminate after it has administered its last command. Note the difference in the instantiations of `move1` and `move2`. The activity `move2` has been defined to be "instantaneous". An instantaneous activity does not have the ability to summon the termination of the processes it governs.

Please note that the previous segment of code is not completely correct. Any attempt to compile the code segment would produce a variety of compiler errors. An intentional mistake appears in the first line. It is not actually possible to instantiate a variable of class **Process**. The class **Process** serves as an abstract class from which "real" processes are to be built. All legitimate processes must be derived from the class **Process**. For example,

```
class MoveProcess: public Process { ... };
class CollisionProcess: public Process { ... };
```

Processes **MoveProcess** and **CollisionProcess** are derivations of the class **Process**. The erroneous code segment is presented to define the relationship existing between activities and processes. It should not be directly incorporated into genuine simulation models.

11.6.4 EXAMPLE

This chapter concludes with an extensive example illustrating the power of process-based simulation. The goal of this example is to create a process for "mouse-based" user interaction. Unlike many simulations which separate the user-interface queries from the simulation model, this example incorporates the query task directly into the simulation. Users can control the number and the frequency of user-interaction queries.

The following segment of code is the header file for the process class **GLEventLoop**. This process must be provided an active setting and single-argument procedure to perform correctly. The one argument procedure will be invoked whenever the process receives a LEFTBUTTON message from the GL event scheduler.

```
class GLEventLoop: public Process {
  GL_Loop loop;
  short val;
  long event;
  Bool bLoop, (*leftBut)(short);
```

```
 public:
  /* manager functions */
  GLEventLoop( RaspSetting* );

  /* access functions */
  void setLeftButton( Bool (*)( short ));

  /* implementor functions */
  void   initialize();
  void   finish();
  int    body();
};
```

This next segment of code defines the three required routines every user-defined process must declare. The first two routines execute the initial and final GL device driver routines. These routines inform the GL event scheduler as to when to initiate and terminate LEFTMOUSE events. Close examination of the routine body reveals that this process will perform one mouse inquiry every five units of simulation time. It is important to remember that the process will make no inquires until activated by an external source.

```
void GLEventLoop::initialize()
{
  /* GL calls */
  qdevice( LEFTMOUSE );

  /* insert procedures into event loop */
  evt.insert( LEFTMOUSE, leftBut );
}

void GLEventLoop::finish()
{
  /* GL calls */
  unqdevice( LEFTMOUSE );
}

int GLEventLoop::body()
{
  /* this switch statement is mandatory */
  switch( jumpLine ) {
  case JUMP_1:
    goto JMP_1;
    break;
  };

  /* event loop */
  while( bLoop ) {
  JMP_1:if (qtest()) {
      event = qread( &val );
```

```
      bLoop = evt.event( event, val );
      world->renderAll();
    }
    hold( 5 );
    return( JUMP_1 );
  }

  return NO_JUMP;
}
```

This last segment of code define a simple simulation with one user-interaction process. The mouse query process is activated when the simulation time reaches thirty-three. Since the process has no terminating mechanisms, the simulation has been explicitly declared to stop at time 100.

```
Bool do_leftmouse( short )
{
  cout << "MOUSE " << endl;
  return TRUE;
}

main()
{
  /* create a setting */
  RaspSetting world;

  /* create a mouse inquire process */
  GLEventLoop *loop = new GLEventLoop( &world );
  loop->setLeftButton( do_leftmouse );

  /* create an activity */
  Activity *loopAct = new Activity( 33 );
  loopAct->addProcess( loop );

  /* create a procession */
  Procession *seq1 = new Procession( "GL-EVENTS" );
  seq1->addActivity( loopAct );
  world->addProcession( seq1 );

  /* run the simulation */
  world->endTime( 100 );
  world->run();
}
```

11.7 RENDERERS

Every *data-to-image translation* object or *image renderer* is derived from the base class **Renderer**. This class determines a collection of "implementor" operations that each derived class must define. In addition to these "abstract" member functions, **Renderer** provides a set of operations to maintain a *primitive* and *geometry* list. These "base" operations alleviate each renderer subclass from asserting their own set of "list" functions.

Renderer's abstract "implementor" functions are divided into two basic sets. Applied to alter a renderer's state, the first group of routines inform a renderer when to begin and terminate the synthesis of a single image. The second set of routines declares an assortment of commands pertinent to image synthesis. For every geometric primitive there is a corresponding image construction command. While each new renderer type must re-define every state altering routine, they are not ordained to redefine every primitive drawing command. They are mandated to redefine only those commands corresponding to the geometric primitives which they support. A renderer must redefine at least one command for each of the items in its *primitive* list.

Geometric Primitives

Point	Particle	Line	Polyline	Plane	Polygon
Triangle	Quadrilateral	Disk	Cone	Cube	Cylinder
Sphere	Square	Superquadric	Torus	Bezier-Patch	Elliptic-Cone

Table 11.3: Primitive List

Renderer's "base" operations regulate the *primitive* and *geometric* lists for each derived renderer subclass. It is the duty of every derived renderer to register every primitive it can support in its associated primitive list. This task is to be accomplished immediately after the renderer has been initialized. After the primitive list has been completely constructed, users are free to manipulate its contents at any point during a simulation. This enables users to produce sequences of animation with frames of differing quality. The values of the geometric list are not defined by renderer developers. Users are not required to affix any statements

to their derived renderers to manipulate its contents. Renderer objects inherit commands to adjust automatically the contents of its geometric list as it receives notification messages from the toolkit's scene modeling tools.

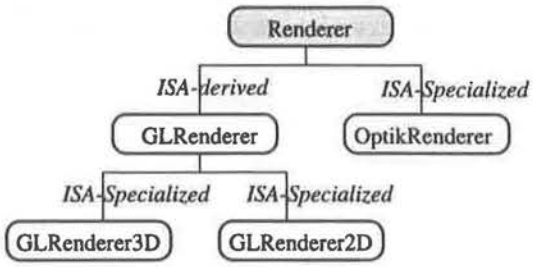

Figure 11.6: Renderer Hierarchy

The current toolkit supports three image translator objects: two GL-based renderers and one Optik-based renderer. The diagram in Figure 11.6 portrays the toolkit's rendering hierarchy. The GL-renderers, one for three-dimensions and the other for two-dimensions, are *on-line* translators. They directly convert primitive object definitions into SGI's graphics language commands. If available, GL-renderers can be directed to harness all of SGI's hardware enhancements. Although these renderers provide users with instantaneous feedback, they do not have the ability to produce high quality images. The Optik-renderer is a high quality *off-line* renderer. Primitive object definitions are used to produce a scene description file. Each file is fed to Optik, a ray-tracing image synthesizer, to produce vivid images. It should be noted that the toolkit's *data-to-image* translation technique does not impede the development of an *on-line* Optik-based renderer. This task is consigned for future work.

11.8 Geometry

In the toolkit, there are two types of geometric classes. Geometric form classes, which manage the properties of a simple physical shapes, and geometric assembly classes, which manage the construction of complex physical shape. Both geometric types are derived from the abstract class **GeoBase** (see Figure 11.8). Serving as an administrator of abstract operations, this abstract class ensures that all geometric classes are able to interface with rendering classes. Base operations are also provided by **GeoBase** to handle basic functions, such as the creation of ports common to all geometries and the management of important data members.

Geometric form classes derive additional information from the base class *Geometry*. This base class defines a few operations relevant to simple geometric shapes. In addition, the class **Geometry** serves as a classification device to distinguish geometric form classes from geometric assembly classes.

11.9 Utility Classes

The toolkit defines a large collection of additional classes which serve to facilitate the creation of simulations. Classes organize attributes, create windows (see Figure 11.7), manage bounding box information, represent points, vectors, arrays, heaps, lights, and colors, provide user-interface controls, and assist in caching data. Users wishing to further examine these classes are directed to obtain a copy of the toolkit and peruse its contents.

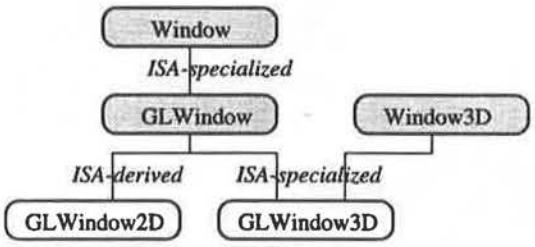

Figure 11.7: Window Hierarchy

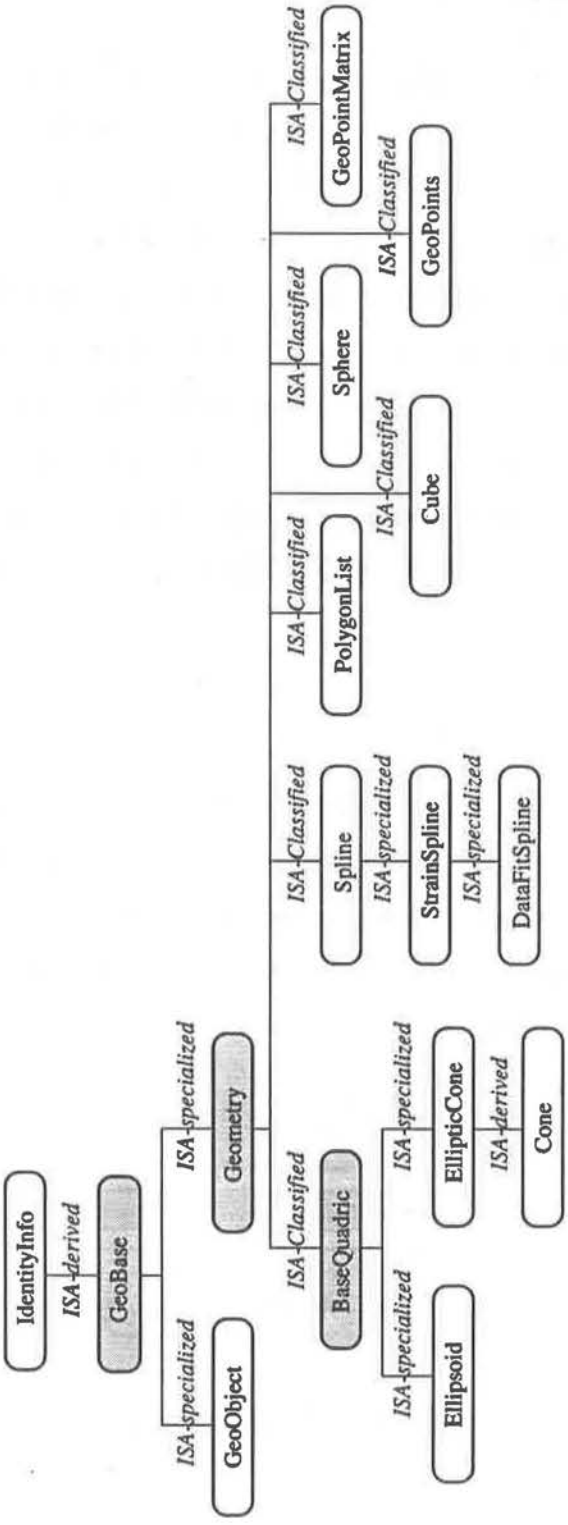

Figure 11.8: Geometry Hierarchy

CHAPTER 12
CONCLUSION

I have seen the future and it works.

- (Joseph) Lincoln Steffens, Autobiography, Ch. 18

Simulations are valuable industrial tools. They enable users to emulate and test the behaviors of real world applications and to improve the design of previously existing models and systems. Users' productivity is enhanced when they can reuse and intermix components from various simulations. Unfortunately, most simulation components are not naturally reusable. Many are hard to extend or modify. Although many tools, such as simulation languages, environments, and systems, have been developed for simulation, their usage has not cultivated the creation of genuinely reusable components. The failure of these tools to partition clearly the simulation modeling process, incorporate reusable technologies, and provide well-defined temporal modeling mechanisms, hampers the construction of simulations that are easy to reuse, decipher and decompose.

An attempt to provide users with a clear foundation towards the development of reusable simulations has been presented in this thesis in the form of the RASP toolkit. Emphasizing the needs of the computer graphics and robotics community, this toolkit uses object-orient principles and modern patterns of communication to promote the development of reusable simulations. The extensibility and tractability of the toolkit's design empowers users to construct a wide variety of models and systems. The toolkit is highlighted by *IMVCD* - a modeling framework, the *Connection Paradigm* - a communication architecture based upon ports, *Hierarchical Temporal Modeling Tools* - a set of primitives which establish a clear relationship between time and state, and *Hybrid Object Construction* - a design methodology to create graphical models.

IMVCD defines a set of design rules for users to observe when designing RASP applications. Focusing on the architecture of programs, as opposed to the reuse of implementation, the framework ameliorates users' potentials to create reusable simulations. The simulation

modeling process is informally partitioned to highlight the communication paths between system components: **M**odels, constructed with **I**nformer objects, are visualized by **V**iewers; and **C**ontrollers, regulated by **D**elegators, govern the behaviors of **M**odels.

The **Connection Paradigm** defines a communication architecture which delegates the responsibilities of creating and maintaining data pathways to a variety of modeling elements. Based upon unidirectional ports and first-class connections, this plan eliminates the need for models to be concerned with the identity of its communication partners and to be aware of the nature of environment in which they interact. Models react to events triggered on their *ports*, not to messages originating from other models or system components.

Hierarchical Temporal Modeling Tools enable users to describe and organize hierarchically the state changes occurring within a simulation. State changes are associated with temporal values to produce scripted animations and adaptive simulations. Constructed as first-class objects, the tools define a clear relationship between time and state: *Processions* regulate the activation of *Activities* while *Activities* regulate the activation of *Events* and *Processes*. In addition, the tools endorse the creation of a minimalistic simulation kernel and permit the development of simulations which incorporate multiple world views.

Hybrid Object Construction delineates a model design methodology which combines a unified user interface with a rendering architecture. Constructed as first-class entities, models organize collections of *feature ports* and *slots*. Adhering to the connection paradigm, feature ports regulate the contents of slots while slots contain references to *feature objects*. Feature objects manage primary functions and regulate model attributes.

12.1 Assessment of RASP

This section assesses the toolkit ability to meet the design goals of Chapter 6 and identifies the aspects of the toolkit which establish its uniqueness.

12.1.1 Goals

Framework

RASP's framework delineates a strong foundation for the development of reusable simulations. It leads users to develop simulations that adhere to a common design. The generality of the rules of interaction, the clear partitioning of the design process, and the clarity of the communication architecture permits the development of a wide variety of models and simulations. However, the framework still requires further enhancement. As yet, the rules of interaction neither provide regulations that manage the interaction of multiple controllers (see section 12.2) and nor describe the framework's relationship with the underlying system architecture. Rules are needed to clarify the role of the simulation kernel and its influence on the framework components. It is not clear yet whether the framework's current decomposition is adequate or complete. Sub-frameworks may be needed to promote greater reusable development, and new frameworks may be devised to handle alternative modeling designs.

Multiple Temporal Strategies

RASP's multiple interface to discrete-event modeling permits the creation of simulations via events, conditional activities, and processes. Users intermix all three types to form simulations that are relatively easy to build, manage, and interpret. The main problem with this approach is the manner in which conditional activities are specified and examined. To reduce the complexity of the simulation kernel, contingency tests normally associated with activities are relegated to the state variables of the simulation. This action burdens variables attached to contingency tests to signal state change events. It is not clear if this behavior is truly desirable. Future designs that attempt to allay this burden may require language support.

Time and State

RASP's set of temporal modeling tools standardizes the specification of state changes and the employment of temporal management methodologies, establishes a definitional uniformity of simulation terms, permits the creation of uneven (variable rate) computing operations, and

minimizes the responsibilities of the simulation kernel.

Three major issues of importance relate to these tools. First, it is not clear whether the tools are adequately extensible. For example, there is currently only one type of "activity" provided by the toolkit. Further testing must be performed to determine if another type of activity is supportable and if the tools need to be redesigned. Second, firm guidelines must be established to aid in the decomposition and reuse of the simulation kernel. Although the kernel possesses minimal responsibilities, users may wish to produce variations of the kernel to optimize the operations of their simulations[1] or to vary the duties and purpose of the kernel. Third, a clear distinction does not exist between control information and model actions. The tools must be altered or augmented to permit the specification of model behaviors independent of the experimental frame under which the models are run.

Geometric Model Construction

RASP's model construction methodology promotes a manageable model architecture by delegating the handling of model attributes and responsibilities to feature objects. Models manage feature objects via slots and ports to promote an external interface that adheres to the connection paradigm and permits temporal modeling tools to influence a model's state. Delegation is extended to feature objects to form a hierarchical ordering of model attributes and duties. Alterations to model traits are generated at various levels of detail by modifying the hierarchical data structure at different ranks.

RASP's model construction methodology supports the interaction between models and renderers by defining a communicational protocol for data-to-image translation. Models and renderers support interfaces which enable them to communicate and maintain independent knowledge of their needs and capabilities.

Although C++'s inability to support delegation limited the examination of RASP's model architecture, two important issues arose during testing. First, the hierarchical ordering of feature objects and feature ports increased the complexity of simulation models. To support

[1] Although it was noted in section 6.3.4 that optimizations to the simulation kernel reduces its versatility, this fact will not stop users from altering the operation of the kernel to enhance the performance of their simulations. Therefore, it is important to provide rules that guide any modifications made to the kernel.

RASP's multiple interface approach to discrete-event modeling, they needed to supply operations that transmitted information to and obtained information from its immediate predecessors and successors. Future enhancement need to determine if these operations can be eliminated via language support or new design. Second, rules must be established to coordinate the communication between two or more feature objects. It is not clear whether information is to be exchanged via ports, direct message passing, or delegation.

12.1.2 Discussion

Apart from obvious differences, the RASP toolkit differs from previously designed toolkits in four major ways. First, object-oriented principles were applied to the toolkit design from its inception to its current implementation. Objects were created to construct models, coordinate communication between components, and form simulation frameworks. RASP's construction process did not encounter problems, such as design specifications being far removed from implementation constructs, normally associated with software projects that attempt to objectify their systems only during the implementation phase of software development. As noted in [16, 60], the application of object-oriented principles to software design, analysis, and implementation imposes good software engineering practices, fosters reusability, and promotes structural continuity.[2] Second, the toolkit defines a framework for simulation development that emphasizes the reuse of design, not just implementation. The relationships and dependencies between groups of objects are delineated to promote a general architecture for component reuse. Third, the toolkit endorses a separation of communication and co-ordination from functionality and computation. Serving only one purpose, tools orchestrate the the interaction among sets of models or manipulate the attributes or properties of a model. Fourth, the toolkit endorses the development of models which maintain a loose connection to their environments. Models form relationships via ports and connections, not via direct associations. This design fosters greater model reusability and interpretability.

[2]Classes, objects, properties, and relations described in the design phrase show up in the implementation phase.

12.2 Future Work

More experience is needed to evaluate the potential and test the adequacy of the RASP toolkit. Only through experimentation is it possible to determine if the toolkit's constructs and abstractions provide enough leeway for users to create a wide variety of simulations. Nevertheless, it is clear that certain modifications and extensions to the toolkit will prove beneficial. The following itemization presents a few areas of possible enhancements.

Alternate Language

Although the RASP toolkit was initially designed independent of an implementation, continual enhancements and modification to the toolkit's design were strongly influenced by its C++ implementation. The inability of C++ to support delegation, garbage collection, and generic pointers to class members, and run-time type identification required the toolkit to supply users with many data structures and constants that enabled them to emulate these missing properties. Apart from cluttering the toolkit with extraneous structures, these additional tools attenuated the simplicity and power of the toolkit's design.

An alternate implementation of the toolkit using another language, such as RAVEN[1], Objective-C[14], or SELF[33], may provide insights concerning the correctness and applicability of the toolkit's design. The incorporation of new language features may improve the toolkit's design and its approach to reusability.

Temporal Relations

In the real-world, users do not always define the activation of events and activities in terms of explicit units of time. Activation times are often expressed in relation to other activation times. Therefore, it would be extremely beneficial for RASP to allow users to coordinate the activation of state changes without explicit references to specific points in time. For example, it would not be extremely difficult for RASP's design to provide operators which define many the *reference intervals* described in [2] and shown in Table 12.4.

By using these reference intervals, it would be possible to represent the relationship between

Relation	Symbol	Inverse Symbol	Pictorial Example
X *before* Y	<	>	XXX YYY
X *equal* Y	=	=	XXX YYY
X *meets* Y	m	mi	XXXYYY
X *overlaps* Y	o	oi	XXX YYY
X *during* Y	d	di	XXX YYYYYY
X *starts* Y	s	si	XXX YYYYY
X *finishes* Y	f	fi	XXX YYYYYY

Table 12.4: Temporal Interval Relations

multiple temporal intervals using constraint propagation, and explicitly handle problems dealing with relative temporal knowledge.

Interface Applications

Usage of the RASP toolkit requires programming experience. Users must be familiar with programming languages, software engineering, and object-oriented principles to construct properly reusable simulations. To cater to a wider audience and to improve greatly the user-friendliness of the toolkit, the construction of front-end programs that allow users to build RASP simulation via a graphical interface or high-level scripting language would be worthwhile. Such programs could directly interpret the simulation designed by users, or they may yield segments of source code that compile into RASP simulations.

Controller Interaction

The IMVCD framework dictates that only controllers are to control the dynamic properties of a simulation. Although this design enhances the development of reusable components, it is far from complete. The framework fails to provide rules which govern the conduct of multiple controller acting in unison or conflict. Rules must be defined to resolve conflicts among

discordant controllers, to assign semantic relationship between coupled controllers, and to classify the bindings between controllers and the entities which they control. Previous works by [43, 47, 98, 65] support the need for such rules.

Temporal Regression

Some simulations, especially computer animations, need to run simulations backwards. Reversing the direction of simulated time enables users to test the operation and validity of their simulations and to examine the effects induced by altering the nature of previously transpired state changes.

Parallelism & Distributed Principles

The benefits derived from distributing computations across multiple platforms and conducting operations in parallel are numerous. Apart from increasing the efficiency and productivity of simulations, they enable users to devise architectures which closely resemble real-world scenarios. For example, simulations which try to emulate and test the interactions between robotic devices may benefit from a design which distributes each device to a separate computer. Therefore, it is of interest to determine RASP capabilities to incorporate plans for distributed and parallel computing. The similarities between RASP and various co-ordination languages, such as MANIFOLD, suggests that the toolkit's design is amenable to such improvements.

APPENDIX A

OBJECT ORIENTED LANGUAGES

...objects are classified scientifically into three major categories - those that don't work, those that break down, and those that get lost. -Ibid.

A.1 DEFINITION

According to [91], a language is deemed to be object-oriented if it supports three important elements. First, it supports some notion of "objects". Every object must have a set of operations, called *methods*, and a *state* that is affected by these operations. Second, every object belongs to a *class*. Objects of the same class have uniform behavior and equivalent operations. Third, some form of an *inheritance* mechanism is used to define *class hierarchies*. Inheritance enables construction of new objects by extending, reducing, or modifying the functionality of existing objects.

According to [7], Wegner's definition of OOP is too restrictive and unsatisfactory. He proposes an alternative definition based on the properties of object-oriented systems as opposed to their implementation mechanics. For a system to be considered object-oriented, it must possess the concepts of encapsulation,[1] set-based abstraction, and polymorphism.

Encapsulation Objects are encapsulated if their data structures and procedures are enclosed within unpenetrable tight boundaries. Only through well-defined interfaces can an object's internal structures be accessed.

Set-based Abstraction Set-based abstraction requires all entities to belong to sets. Sets enable clients to abstract common properties among a collection of objects. All sets are capable of intersection and union. Therefore, it is possible for any object to be part

[1]It is not universally accepted that encapsulation provides greater programming constructs for OOP. Some object-oriented languages, such as SIMULA, allow direct access to internal variables. Others, like HSL, support weak encapsulation structures. Some researchers believe encapsulation reduces the communication abilities between various objects.

of more than one set. Languages which support inheritance and conformance[2] support set-based abstractions.

Polymorphism Polymorphism is the ability of procedures or functions to operate on more than one data type. Stringent type-checking methods have proven to be too restrictive to program development.

A.2 Class Hierarchies

A.2.1 Inheritance

Classes that inherit operations from *superclasses* and allow their operations to be acquired from *subclasses* exhibit inheritance. The ISA relationship, which is formed between a class and its superclass, can be taxonomised as follows:[85]

- **ISA-derived**: If it possible to apply a derivation rule to class A to form a subclass B, then B is defined to be derived from A. Although derivation rules come in many forms, they usually impose restrictions to a class' design. For example, the class TENNIS_BALL is ISA-derived from BALL by applying the predicate condition *type=tennis* to BALL. ISA-derived classes do not store additional data structures. They utilize the data structures of their parent classes to reserve their data.

- **ISA-specialized**: A subclass that subsumes the properties of its parent class is defined to be a specialized class. Besides exhibiting all the qualities of its parent class, an ISA-specialized class demonstrates supplementary traits. For instance, class TEAM_CAPTAIN is a specialized class of TEAM_PLAYER. Apart from performing the role of a typical player, a team captain behaves as a leader and instructor. A specialized class stores only distinctive information within its structures. General information, inherited from its genitor, is preserved in the parent class.

- **ISA-classified**: A subclass is ISA-classified from its parent class if its inheritance link is used purely as a classification device. No additional derived or functional relationship

[2]Conformance is related to the concept of *subtyping*. The reader is directed toward [8] for detailed discussions on *subtyping*.

exists between the subclass and its parent. ISA-classified classes do not rely on their parent classes to maintain data structures. Descriptive data is stored within the subclass' personal structures.

A.2.2 DELEGATION

Three problems are associated with the class/inheritance approach. First, it is not a precise method to describe resource sharing in class hierarchies. Second, object-oriented mechanisms are overloaded in the sense that inheritance is used to implement type checking, binding, and behavior sharing. Third, classes are used to describe abstract behaviors and attach functionality to them. Although one might consider the last two problems to be strengths as opposed to weaknesses, they limit the *degrees of freedom* implementors need to build sophisticated objects.

Delegation is a better class independent term for dynamic hierarchical resource sharing. By inheriting state and behavior, delegation makes it possible to change the behavior of object dynamically. In a delegation-based system, every object behaves as a "prototype" for the creation of a new object[7]. For example, in inheritance-based languages, such as Smalltalk[26, 34], when an object receives a message, it searches for method in its class. If a method can not be found, the search is expanded to the class's superclass, and the superclass's superclass, and so on. This hierarchical look-up mechanism results in subclasses inheriting methods from their superclasses. However, in delegation-based languages, like SELF[33], methods are stored in objects, and there are no concepts of classes. After receiving a message, if an object can not find a method within itself, it delegates responsibility to another object to find one. The delegated object can also choose to delegate responsibility to another object if it proves unable to find a suitable method. In the delegation hierarchy, objects inherit methods to which it delegates messages[38]. The following list enumerates the advantages of delegation over inheritance.

- Simplifies the programming model.
- Eliminates the complexity associated with metaclasses without removing their power.
- Easier to implement one-of-a-kind objects.
- Easier to the change the behavior of objects.

- Powerful enough to simulate inheritance[49].

Although delegation provide users with a different methodology than inheritance, there seems to be little difference between them in terms of implementation. Recent evidence by [81] seems to indicate that inheritance and delegation can provide exactly the same facilities.

A.3 LANGUAGES

Object-oriented languages come in a variety of forms: commercial products, research projects, standalone languages, language derivatives, etc. Some languages even support concurrency and distributed principles. The remainder of this section discusses two important OOP languages, C++ and SELF. Readers wishing to further explore the area of object-oriented languages are directed to start with [75].

A.3.1 C++

Created by Bjarne Stroustrup of AT&T Bell Laboratories, C++ was introduced in the early 1980s as an extension to the C programming language developed by Dennis Ritchie. While remaining compatible and comparable to C in terms of syntax, performance, and portability, C++ provides data abstraction and object-oriented programming facilities. Besides increasing the amount of static type checking, C++'s inheritance-based design enables users to define user-defined types that obey identical scope, allocation, and naming rules as built-in types[39].

As powerful as C++ may be, it is not perfect. Currently, C++ does not support accurately *meta-classes*, *exception handling* facilities, *typedef identification*, run-time creation of new types, *concurrency* mechanisms, and *persistence* - the placement of objects on secondary storage so that they can exist across multiple platforms and applications.

A.3.2 SELF

SELF[33] is a delegation-based language. Using neither classes nor variables, SELF utilizes a prototype metaphor for object creation. SELF objects do not obtain state information in the same manner as objects defined in most other object-oriented languages. They send messages

to "self", the receiver of the current message, to attain their current condition. SELF's great expressive power comes from its inability to distinguish state from behavior and its uniform capability to access different types of stored and computer data.

The following table provided by [33] compares SELF with class-based systems, such as SmallTalk.

	class-based systems	**SELF: no classes**
inheritance relationships	instance of	inherits from
creation metaphor	build according to plan	clone an object
initialization	executing a "plan"	cloning an example
one-of-a-kind	need extra object for class	no extra object needed
infinite regress	class of class of class of ...	none required

Table A.5: Class Systems vs. SELF

Appendix B

Software Reusability

It is widely believed that software reusability enhances the productivity and development of quality software. A library of reusable components provides users with a basic set of well-defined tools to create a wide variety of software applications. A reusable library raises the level of abstraction to allow users to concentrate on the problem domain. Although great strides have been accomplished in the field of software engineering, the promise of reusability has not yet been fulfilled. The lack of a standard method of software development and the multifarious needs of application users have attributed to the limited success of reusable software libraries. Common problems associated with reusable component development include:

- **inability to respond to changing user needs**: A component that may be appropriate during the early stages of development may be unsuitable for the final product.
- **excessively specialized components**: Any component that is difficult to use or modify deters users from utilizing it. The software development process may become severely encumbered if users must spend valuable time deciphering the components of any library.
- **missing components**: The absence of vital components hinders users from developing applications with the software library. If continually forced to define important constructs, users will reject the usage of a set of extraneous components.
- **implementation dependence**: It is difficult to define a module that client modules can rely on without knowledge of the module's implementation. Some users may experience apprehension when an important component's implementation details are hidden from them.
- **operational problems**: Without a clear organizational structure, the effective potential of a component library will diminish as it grows in size and functionality. Components

that are difficult to find or troublesome to interpret effect the reusability of any library.

- **poor early investment**: Creating a viable collection of reusable components is a time consuming task. Careful effort must be made to ensure that each module is reliable and general enough to suit the needs of users.

B.1 Reusability Technologies

There are two main approaches to reusable technology[5]. The first major group, *composition technology*, is characterized by the idea that all components of a reusable library are atomic units. Each module behaves as an independent entity. Although components may be modified for individual needs, this approach emphasizes that components do not need to be changed to be reused. Using this technology, new applications are developed by combining individual units into larger components. The UNIX pipe mechanism exemplifies this approach to reusability.

The second major approach, *generation technologies*, is based upon the idea that reusable components are the patterns of a generator program. Components are not concrete modules. They define the underlying structure and body of target programs. Reuse is not a matter of compositing components together. It is a matter of the execution of component generators. The Draco[59] approach to reusable component design illustrates this approach.

B.2 Object-Oriented Approach

The introduction of *object oriented programming* (OOP) has had a profound effect on computer software construction. There are many advantages of OOP over traditional (procedural) programming languages. Because programming is always structured and modular, the ability to design, construct, and maintain large scale systems is enhanced. Other benefits include increased software re-usability, improved team project coordination schemes, and enriched facilities for hierarchical design. Based on the theory of data abstraction, this technique introduces new ideas and constructs not commonly found in modular languages. Unlike classical functional design, object-oriented design does not base the modular decomposition of a software application on the functions the system performs. Decomposition is generated from the classes

of objects the system manipulates. Applications are decomposed into systems of interacting objects, not interacting functions.

Independent component design is a fundamental aspect of many object-based systems. Users build complex systems from the composition of various library modules. However, whenever a particular module (class) does not explicitly fit users' designs, they may define submodules (subclasses) that define new operations or redefine old operations to suit their needs. This process is effortless in object-oriented languages that support inheritance. Besides facilitating specialization, inheritance can be used for type checking and component classification. However, excessive use of inheritance has proven to be deleterious[94, 23, 12]. Apart from reducing the runtime efficiency of object oriented applications, large inheritance trees form interdependencies among large sets of objects. This condition makes it very difficult to transfer useful submodules between projects without transferring a large segment of the inheritance tree.

The importance of generation technologies in the development of reusable modules has guided the development of several key constructs in object-oriented languages. The introduction of *abstract data types*, *type parameterization*, and *frameworks* have greatly enhanced the design of reusable libraries.[1]

B.2.1 Abstract Data Type

An abstract data type (ADT) is a class of objects characterized by its external properties. The ADT is described by the abstract features of its associated operations. For some languages, such as C++[84], it is impossible to generate an instance of an ADT because at least one or more of its operations are always left unimplemented. Users must completely define all the operations of an ADT before using it. From a hierarchical standpoint, the ADT represents the superclass (see Appendix A) of user-defined classes. The ADT proves an important construct for developing numerous objects with the same protocol but vastly different implementations.

The abstract data type has three kinds of operations[37]:

- **abstract operations**: These types of operations are not implemented within the body of

[1] Although the concepts of abstract and parameterized data types existed before the development of object-oriented design, few languages provided mechanisms to implement these ideas. The introduction of object-oriented languages provided users with concrete mechanisms to express these reusable technologies

an ADT but must be defined. In a class-based system, abstract operations are declared in the superclass, but their implementation is to be defined in a specific subclass. Abstract operations are part of the specification for all subclasses of the ADT. A variety of languages use different techniques to ensure that the syntactic part of an ADT's specification is correctly followed. Some languages, such as C++, inspect for definitions of abstract operations during compile time while other languages, like Smalltalk, delay checking for definitions until the operations are actually used.

- **template operations**[2]: An abstract algorithm that is defined in terms of one or more abstract operations is called a template operation. Operations of this type can be interpreted as being partially implemented. The user of the ADT defines the abstract operations within the body of each template operation. The following segment of C++ code illustrates a simple template operation.

```
Foo::write( Device io )
{
    char buffer[255];
    this->read(buffer ); /* abstract operation */
    io->write( buffer );
}
```

  In this example, the algorithm "write" has been defined for the abstract class "Foo". Within this function, the abstract operation "read" is being called. The function "read" must be defined by users of "Foo" if they wish to use the "write" operation.

- **base operations**: A fully implemented operation of an ADT is defined as a base operation. Although users may redefine these operations, they are not required to do so.

B.2.2 Type Parameterization

Languages that support type parameterization enable users to define a type in terms of another, unspecified type. This feature is especially useful for creating general container types, such as list and array, where the supported element type is defined by a parameter. Used in conjunction

[2]Template operations are not needed in some object-oriented languages, such as SMALLTALK

with abstract data types, parameterized types are also useful for defining generic functions, such as "min" and "max", for a family of types. Languages such as C++ commonly refer to parameterized types as templates.[3]

B.2.3 FRAMEWORK

A framework is a set of design rules for a collection of collaborating objects. It defines how a system is divided into components and how functions are divided among each of the individual groups of objects. This technique of high-level design focuses on the architecture of programs, as opposed to the reuse of implementation. Frameworks emphasize the communicational paths between objects, not the dataflow between them. User-interface design packages, such as Model/View/Controller (MVC)[45], Interviews from Stanford[50], and MacApp[76], exemplify the popularity and necessity for frameworks.

Currently, frameworks prove difficult to define. Models of interaction and the definition of control flow among sets of objects are generally not easy to explain or visualize. Ideally, a framework would be best described in terms of operational constraints placed on objects in a system. This scheme facilitates the reuse of code and description of interacting objects. However, the formal specifications of object-oriented systems have not evolved to a mature enough state to express these ideas. This limitation, however, does not imply that it is impossible to define successful frameworks. Many popular frameworks, such as the ones mentioned above, utilize well-defined informal methods to express their designs.

[3]Parameterized "templates" are unrelated to the "template" operations previously defined above.

APPENDIX C
EXAMPLES

This appendix presents two simulations created with the RASP toolkit. The first simulation produces a small animation of a spherical object moving along a spline path and then bouncing freely within a cube. The second simulation produces an application with two concurrently operating processes. One process manages user-interface events from a "mouse", while the other simply waits to consume data.

Each simulation was developed on Silicon Graphics machines using SGI's C++ 3.0 compiler. Although several GL-based classes and routines are found in both simulations, neither simulation is GL dependent. Equivalent classes and routines, based on other architectures, may be freely used as substitutes.

It should be noted that many header files, extraneous data type definitions, and constant declarations in both simulations were omitted intentionally to reduce clutter. In addition, several local variables have been defined as global variables only to facilitate the decomposition of large routines into smaller procedures.

C.1 Bouncing Ball

In this simulation, a small sphere, which is confined within a large rectangular box, travels along a path determined by a spline from time $t = 5$ to $t = 15$. During this interval, the sphere's position is updated every 2 seconds. From $t = 16$ to $t = 100$, the sphere departs from the spline-based path and alters its position every second according to the "laws of motion". The sphere's direction changes each time it collides with one of the walls of the enclosing rectangular box.

Images of this animation are drawn to a display device using a GL-based renderer and stored as "optik-script" files using a Optik-based renderer. Images drawn to the display screen are shown in two windows. One window updates its view each time an event transpires while the other updates its view every other second during the interval from $t = 10$ to $t = 45$.

C.1.1 Main

The global environment is defined by the following list of variables, which represent the setting, three windows, eight models, and eight geometries. For reasons described in section C.1.3, the enclosing rectangular box is decomposed into six geometric faces.

```
/* the setting */
RaspSetting  *world;

/* windows */
GLWindow     *wind, *wind2;
Window       *wind3;

/* hybrid model objects */
HybridModel  *obj1, *obj2, *obj[6];

/* geometries */
Spline       *spl1;
Sphere       *sph1;
Cube         *cube[6];
```

The procedure `main` initializes a setting; invokes functions to create windows, cameras, and models; calls operations to establish dynamics actions; and informs the simulation when to begin.

```
void PROCEDURE main()
```

```
{
  /* create a setting */
  world = new RaspSetting();

  /* add objects to the world */
  create_windows()
  create_cameras()
  create_models();

  /* set up dynamic qualities of simulation */
  script_animation();
  create_collision_checker();

  world->run();
}
```

C.1.2 Creating Windows & Cameras

The procedure `create_windwos` creates three windows. The first two, `wind` and `wind2`, open windows on a graphical display. The third, `wind3`, is unlike the first two because it does not cause a window to be drawn anywhere. It is a "generic" window. It is used only to set the viewing window for the Optik off-line renderer.

```
void PROCEDURE create_windows()
{
  /* first window */
  fRect w( XMAXSCREEN/2.,100.,(float) XMAXSCREEN, (float) YMAXSCREEN/2.);
  wind = new GLWindow3D( w, "Test", TRUE );
  wind->open_window();
  wind->setColor( DARK_GREY );
  wind->clear_window();

  /* second window */
  fRect w2( XMAXSCREEN/2., (float)YMAXSCREEN*2/3., (float) XMAXSCREEN,
   (float) YMAXSCREEN );
  wind2 = new GLWindow3D( w2, "Test2", TRUE );
  wind2->open_window();
  wind2->setColor( DARK_GREY );
  wind2->clear_window();

  /* third window */
  fRect w3( 0, 0, 100, 100 );
  wind3 = new Window( w3, ''Test'' );
  wind3->setColor( DARK_GREY );
}
```

The procedure `create_cameras` creates one camera for each window defined in the previous procedure `create_windows`. However, this does not imply that every RASP simulation must

create a separate camera for every window it utilizes. One camera could have been used for all three windows. Separate cameras are created for this simulation so that the viewing parameters and update times for each of the three windows may vary.

```
void PROCEDURE create_cameras()
{
  /* create renderers */
  GLRenderer3D *glRend = new GLRenderer3D();
  OptikRenderer *opRend = new OptikRenderer();

  /* camera parameter values */
  fVector viewup (-1.0, -1.0, 0. );
  dAngle fovx = 90., fovy = 90.;

  /* set first camera parameters */
  Camera *camera = new Camera( "MyCamera" );
  camera->setView( Point3(20,35,110), ORIGIN, viewup, fovx, fovy );
  camera->setClipPlanes( .001, 3500. );
  camera->associateWindow( wind );
  camera->associateRenderer( glRend );
  camera->wind_Set_OrthRt( .5 );

  /* set second camera parameters */
  Camera *camera2 = new Camera( "SecondCamera" );
  camera2->setView( Point3(150,0,0), ORIGIN, viewup, fovx, fovy );
  camera2->setClipPlanes( .001, 3500. );
  camera2->associateWindow( wind2 );
  camera2->associateRenderer( glRend );
  camera2->wind_Set_OrthRt( .5 );
  camera2->setUpdateEvent( FALSE );

  /* set third camera's parameters */
  Camera *camera3 = new Camera( "OptikCamera" );
  camera3->setView( Point3(0,30,100), ORIGIN, viewup, fovx, fovy );
  camera3->setClipPlanes( .1, 1200. );
  camera3->associateWindow( wind3 );
  camera3->associateRenderer( opRend );
  camera3->wind_Set_OrthRt( .5 );

  /* add cameras to the world */
  world->addObject( camera );
  world->addObject( camera2 );
  world->addObject( camera3 );
}
```

C.1.3 Creating Models

The procedure **create_models** creates a variety of hybrid models and one directional light.[1] Models are associated with geometries (**Informer** objects, section 7.2) to create one spherical object, one spline object, and six flat plates (cubes). The flat plates form the sides of the box enclosing the moving sphere. One large box could have been constructed to replace the six plates but such a scheme would have made it difficult to determine collisions between the walls of the box and the sphere.

```
void PROCEDURE create_models()
{
  /* create spline with cubic basis */
  Basis *sBasis = new Basis( CUBIC_BASIS );
  spl1 = new Spline( 10, sBasis, FALSE );

  /* set the spline's control point positions */
  (*spl1)(0) = Point3( -100, 0, 0 );
  for(int i=1; i<10; i++)
    (*spl1)(i) = (*spl1)(i-1) + Point3(20,Random(50)-25,2);

  /* create some spherical geometries */
  sph1 = new Sphere( 10. );
  sph1->setName( "redBall" );

  /* create a light */
  DirectLight *dLight = new DirectLight( dVector(1,1,1), BASIC_WHITE );

  /* create cubes */
  cube[0] = new Cube( Point3(-120, -52, -50), Point3( 120, -50, 50 ));
  cube[1] = new Cube( Point3(-120, 52, -50), Point3( 120, 50, 50 ));
  cube[2] = new Cube( Point3(-122, -50, -50), Point3( -120, 50, 50 ));
  cube[3] = new Cube( Point3( 122, -50, -50), Point3( 120, 50, 50 ));
  cube[4] = new Cube( Point3( -120, -50, -50), Point3( 120, 50, -52 ));
  cube[5] = new Cube( Point3( -120, -50, 50), Point3( 120, 50, 52 ));

  /* create Hybrid objects */
  obj1 = new HybridModel( ORIGIN, sph1, RED_BALL );
  obj2 = new HybridModel( ORIGIN, spl1, 002 );
  obj1->set_velocity( dVector( .5, 1, 0 ) );

  char name[255];
  for(i=0; i<6; i++) {
    sprintf( name, "Cube #%d", i );
    cube[i]->setName( name );
    obj[i] = new HybridModel( ORIGIN, cube[i], i+2010 );
  }
```

[1] A directional light source emanates rays of light in only one direction. Directional lights are usually used to represent light sources which are infinitely distant from a scene, such as the sun.

```
  /* add models and lights to the world */
  world->addObject( obj1, BASIC_RED );
  world->addObject( obj2, BASIC_BLUE );
  world->addObject( dLight );

  /* add boxes with different colors */
  world->addObject( obj[0], BASIC_GREEN );
  world->addObject( obj[1], BASIC_GREEN );
  world->addObject( obj[2], BASIC_ORANGE  );
  world->addObject( obj[3], BASIC_CYAN );
  world->addObject( obj[4], BASIC_MAGENTA );
  world->addObject( obj[5], BASIC_MAGENTA );
}
```

C.1.4 Scripting Animation

The procedure `script_animation` defines three `Activitys`. The first activity, `move1`, propels a spherical object along a spline-based path for 10 units of time. This action is accomplished by passing the maximum parametric value of the spline to the **Controller** `evol`, delegating `evol` to generate a range of numerical values (parametric positions) for `spl1`, and delegating `spl1`[2] to set the position of the sphere `obj1`. The second activity, `move2`, enables `obj1`'s motion to be governed by Newton's laws of motion[80] from $t = 16$ to $t = 25$. This action is manufactured by creating a **Controller** (`Motion`) which is aware of Newton's law of motion and delegating it to control the sphere's position. The third activity, `move3`, simply informs the camera "Second Camera" to generate a new image every other unit of time during the interval $t = 10$ to $t = 45$.

```
void PROCEDURE script_animation()
{
  Procession *seq1 = new Procession( "Balls", 0, 50. );

  /** sphere moving behavior from t=5 to t=15 **/
  Activity *move1 = new Activity( "SpMoving", 5., 15., 2 );

      /* Pass the spline's maximum parametric value to the evolution object. */
      dEvolve *evol = new dEvolve( 0, 0, 10 );
      Event act0( spl1->outMaxParam(), evol->inFinishVal() );
      move1->addInitEvent( act0 );

      /* (a) pass temporal value to evolution.
         (b) get value from evolution and pass to spline.
         (c) get value from spline and pass to object.
```

[2]Although a spline is inherently a geometric entity, it may also control the behavior of other objects. In RASP, any object may be delegated to be a **Controller**.

```
        */
       ChainEvent act123;
       TimeEvent act1( evol->inCalcValue() );
       act123.addEvent( act1 );
       act123.addEvent( evol->outCurVal(), spl1->inParamVal() );
       act123.addEvent( spl1->outParamPos(), obj1->inSetPosition() );
       move1->addActEvent( act123 );

  /** Sphere object follows laws of physics **/
  Activity *move2 = new Activity( "Free Motion", 16., 25. );

       /* (a) pass position, velocity, and acceleration from object to motion
          (b) send time to motion.
          (c) send new position from motion to object.
        */
       Motion *motion = new Motion();
       Event act4( obj1->outPosition(), motion->inPosition() );
       Event act5( obj1->outVelocity(), motion->inVelocity() );
       Event act6( obj1->outAcceleration(), motion->inAcceleration() );
       TimeEvent *act7 = new TimeEvent( motion->inDTime() );
       Event act8( motion->outPosition(), obj1->inSetPosition() );

       move2->addInitEvent( act4 );
       move2->addInitEvent( act5 );
       move2->addInitEvent( act6 );
       move2->addActEvent( act7 );
       move2->addActEvent( act8 );

  /** Update second camera every 2 second from t=10 to t=45 **/
  Activity *camUpdate = new Activity( "Update Camera", 10, 45., 2 );
       Camera *camera = (Camera*) world->getCamera( "SecondCamera" );
       TimeEvent camAct( camera->inSnapShot() );
       camUpdate->addActEvent( camAct );

  /** add activities to procession **/
  seq1->addActivity( move1 );
  seq1->addActivity( move2 );
  seq1->addActivity( camUpdate );

  /** add procession to the world **/
  world->addProcession( seq1 );
}
```

C.1.5 Collision Checker

The procedure `create_collision_checker` induces two primary collision checking activities. The first activity `outMove` alters the color of any wall struck by the moving sphere, `obj1` during the interval $t = 1$ to $t = 40$. This act is accomplished by affixing the **StateEvent** `state` to `obj1`'s `outCollision` port. This action forces the **Activity** `condAct` to become active

if `outCollision` ever changes value. The second activity `collCheck` alters the direction of the moving sphere each time it strikes a wall. Collision checking is performed by an collision checking object of type `Collide` and the procedure `alter_sphere_direction`.

Note: The two collision checking activities could have been amalgamated into one large activity. However, this was not performed so that the usage of various RASP operations and objects could be shown.

```
void PROCEDURE create_collision_checker()
{
  /** call on Collision reaction tester function **/
  Activity *condAct = new Activity( "Collision Reaction" );
      Event condEv( alter_wall_color );
      condAct->addInitEvent( condEv );
      condAct->assocProcession( seq1, 0.0 );

  /** if object's collision state changes during the interval t=1 to t=40,
      then initiate condAct.  **/
  Activity *outMove = new Activity( "Conditional Activation", 1., 40., 1 );

      GeoBase *geoInfo = obj1->getGeoBase();
      CollisionInfo *collInfo = geoInfo->getCollisionInfo();
      StateEvent state( collInfo->outCollision(), condAct );
      DisableEvent state2( collInfo->outCollision(), condAct );

      outMove->addInitEvent( state );
      outMove->addFiniEvent( state2 );

  /** set collision checking parameters **/
  Collide *collider = new Collide();
  HybridModel *srcObj = world->getObject( RED_BALL );
  collider->addSource( srcObj );
  for(i=0; i<6; i++)
    collider->addTarget( obj[i] );

  /** perform Collision checking from t=10 to t=100. **/
  Activity *collCheck = new Activity( "CollisionChecking", 10, 100., 1 );
      CallEvent collEvent( collider->inRun() );
      collCheck->addActEvent( collEvent );
      collCheck->addActEvent( alter_sphere_direction );

  /* add new activities to the world */
  Procession *seq1 = world->getProcession( "Balls" );
  seq1->addActivity( collCheck );
  seq1->addActivity( outMove );
}
```

The procedure `alter_wall_color` is invoked when the sphere's collision state changes. If the sphere is determined to be in collision with another object (wall), this procedure will alter

the color of the colliding object to light yellow.

```
void PROCEDURE alter_wall_color()
{
  static GeoBase *hitObj = NULL;
  GeoBase *newHitObj, *ball = world->getObjectGeometry( RED_BALL );
  CollisionInfo *collInfo = ball->getCollisionInfo();

  if (collInfo->getCollisionState()) {
    newHitObj = collInfo->getCollisionObjAt( 0 );
    if (newHitObj != hitObj) {
      Properties *prop = newHitObj->getProperties();
      Material *mat = (Material*) prop->getProperty(ATTRIB_MATERIAL_PROPERTY);
      ColorBase *col = mat->getDiffuse();
      newHitObj->setColor( LIGHT_YELLOW );
      hitObj = newHitObj;
    }
  }
}
```

The procedure `alter_sphere_direction` determines which wall a sphere strikes and alters the sphere's forward direction accordingly.

```
void PROCEDURE alter_sphere_direction()
{
  HybridModel  *box, *sphere;
  GeoBase      *wall;
  dVector      vel;
  static int   lastWall = -1;

  GeoBase *ball = world->getObjectGeometry( RED_BALL );
  CollisionInfo *collInfo = ball->getCollisionInfo();

  if (collInfo->getCollisionState()) {

    for(i=0; i<6; i++) {
      box = world->getObject( i+2010 );
      wall = box->get_geometry();

      if (collInfo->collisionInfo( wall ) && i != lastWall ) {
        sphere = ball->getOwner();
        vel = sphere->get_velocity();

        switch( i ) {
          case 0, 1:
            vel.J() = -vel.J();
            break;
          case 2, 3:
            vel.I() = -vel.I();
            break;
          case 4, 5:
            vel.K() = -vel.K();
```

```
            break;
        }

        sphere->set_velocity( vel );

        Procession *seq1 = world->getProcession( "Balls" );
        Activity *act = seq1->getActivity( "Free Motion" );
        Activity *newAct = new Activity( *act );
        seq1->removeActivity( "Free Motion" );

        double time = world->getWorldTime();
        newAct->setTiming( time, time+20 );
        seq1->addActivity( newAct );

        lastWall = i;
        break;
        }
      }
    }
}
```

C.1.6 IMAGES

The following set of images (Figures C.1 to C.6) show the views of `camera1` and `camera2`. The view on the left is updated every time an event occurs, while the the view on the right is updated every other unit of time from the period $t = 10$ to $t = 45$.

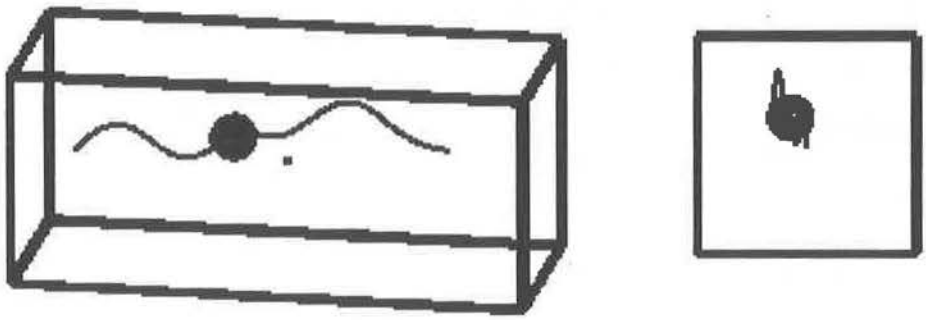

Figure C.1: Frame at $t = 10$

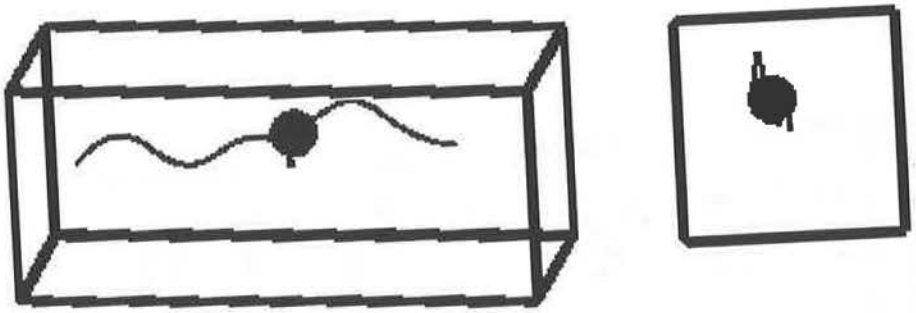

Figure C.2: Frame at $t = 12$

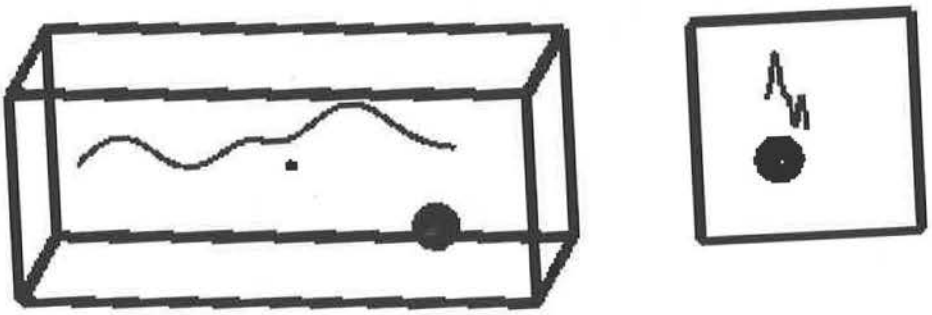

Figure C.3: Frame at $t = 30$

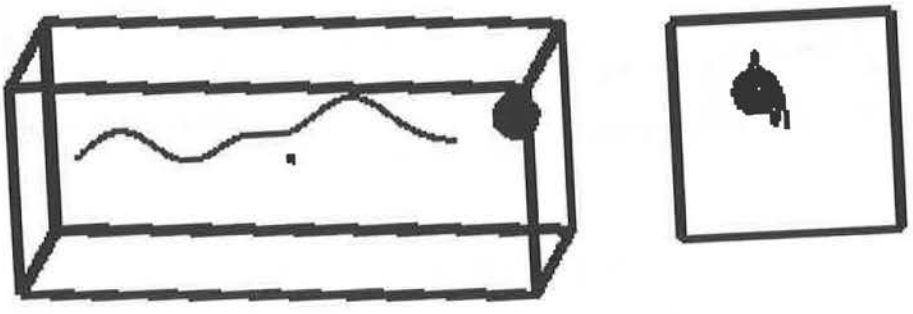

Figure C.4: Frame at $t = 47$

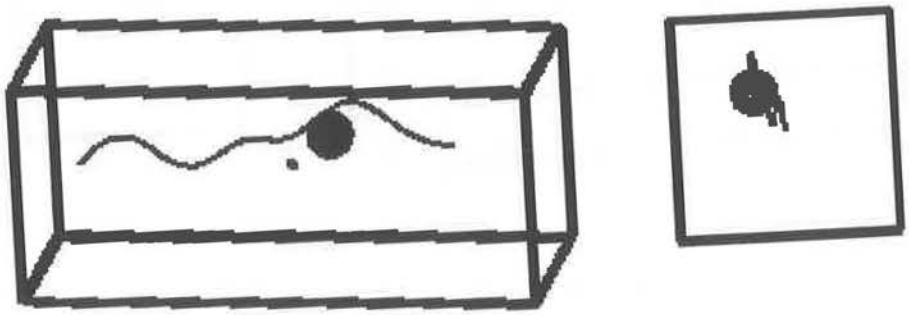

Figure C.5: Frame at $t = 75$

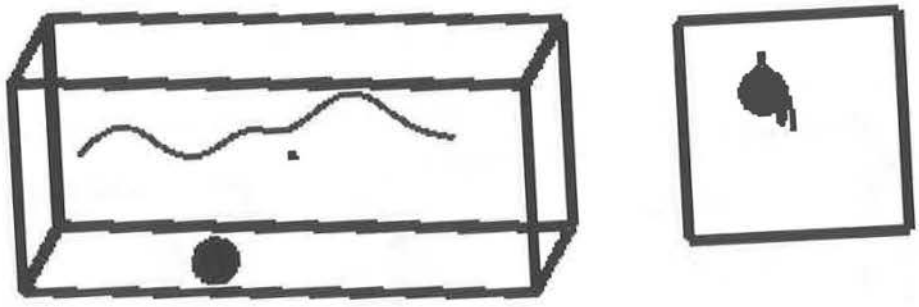

Figure C.6: Frame at $t = 89$

C.2 Two Processes

In this simulation, two concurrent processes operate independently. While one process manages an event loop for a user-interface mouse, the other waits to consume data. Two geometric models (spline with control points) are rendered using a two-dimensional renderer to further illustrate the creation of various models and renderers.

C.2.1 Main

The procedure `main` initializes a setting; invokes a function to establish the contents of the world; calls a procedure to compose dynamic actions; and informs the simulation when to begin.

```
/* process definition header files */
#include "loop.h"
#include "wait.h"

 RaspSetting *world;
 GLWindow    *wind;

PUBLIC void PROCEDURE main()
{
  /* create a setting */
  world  = new RaspSetting();

  /* setup the world and script process behaviors */
  init_world();
  script_processes();

  /* run simulation */
  world->run();
}
```

C.2.2 Creating Windows, Cameras, Models

The procedure `init_world` sets up a 2D renderer, window, and camera; creates two models; and adds all of them to the world. The second geometry is created to visualize the control vertices of the spline. Because both geometries reference the same control point data structure, a modification to one geometry will automatically alter both.

```
void PROCEDURE init_world()
{
```

```
  /* create a renderer */
  GLRenderer2D *glRend = new GLRenderer2D();

  /* create a window */
  fRect w( XMAXSCREEN/2.,100.,(float) XMAXSCREEN, (float) YMAXSCREEN/2.);
  wind = new GLWindow2D( w, "Test", TRUE );
  wind->open_window();
  wind->setColor( DARK_GREY );
  wind->clear_window();

  /* create a camera */
  Camera *camera = new Camera( "MyCamera" );
  camera->associateWindow( wind );
  camera->associateRenderer( glRend );
  camera->wind_Set_OrthRt( .5 );

  /* create spline model */
  Basis *sBasis = new Basis( CUBIC_BASIS );
  Spline *spl1 = new Spline( 10, sBasis, FALSE );

  (*spl1)(0) = Point3(-50.,0,0);
  for(int i=1; i<10; i++)
    (*spl1)(i) = (*spl1)(i-1) + Point3(15,Random(60)-30,2);

  /* create points from spline's control vertices */
  GeoPoints *pts = new GeoPoints( 10 );
  pts->setNewPoints( spl1->getCVs() );

  HybridModel *obj1 = new HybridModel( ORIGIN, spl1, MY_SPLINE );
  HybridModel *obj2 = new HybridModel( ORIGIN, pts, MY_CVS );

  /* add objects to the world */
  world->addObject( camera );
  world->addObject( obj1, BASIC_BLUE );
  world->addObject( obj2, BASIC_YELLOW );
}
```

C.2.3 Script Processes

The procedure `script_process` creates two activities. One activity to activate `loop` and `wait` and a second activity to send data to `wait` during the interval $t = 20$ to $t = 25$. The process `loop`, which administers the operation of a "mouse", is provided two procedures, `do_leftmouse` and `do_rightmouse`. The former outputs the position of the mouse while the latter terminates the simulation.

```
void PROCEDURE script_process()
{
  /* create a procession */
```

```
  Procession *seq1 = new Procession( "GL-EVENTS", 0, 50. );

  /** intiate two processes **/
  Activity *loopAct = new Activity( "Loop", 1, 1 );

     /* create a mouse process */
     GLEventLoop *loop = new GLEventLoop( world );
     loop->setLeftButton( do_leftmouse );
     loop->setRightButton( do_rightmouse );

     /* create a waiting process */
     WaitProcess *wait = new WaitProcess( world );

     loopAct->addProcess( loop );
     loopAct->addProcess( wait );

  /** send time value to waiting process during interval t=20 to t=25 **/
  Activity *valProcess = new Activity( "Give Value to Process", 20, 25 );

     TimeEvent evt( wait->inValue() );
     valProcess->addActEvent( evt );

  /* add activities to procession */
  seq1->addActivity( loopAct );
  seq1->addActivity( valProcess );

  world->addProcession( seq1 );
}

Bool PROCEDURE do_leftmouse( short val )
{
  /* get mouse coordinates and print */
  fPoint2 pos = wind->getMouseWindCoords();
  cout << "MOUSE " << pos << NL;
  return TRUE;
}

Bool PROCEDURE do_rightmouse( short val )
{
  gexit();     /* quit program */
  return FALSE;
}
```

C.2.4 Process Definitions

The following two segments of code show the class definitions for processes `GLEventLoop` and `WaitProcess`. Each process is a subclass of the abstract class `Process`.

GLEventLoop

Header File

```
#include <process.h>
#include <gl_utils.h>

/*****************************************************************************
  process class GL-EVENT-LOOP definition
 *****************************************************************************/
class GLEventLoop: public Process {

  GL_Loop     loop;
  short       val;
  long        event;
  Bool        bLoop;
  Bool        (*leftBut)(short), (*middleBut)(short), (*rightBut)(short);

 public:
  /* manager functions */
  GLEventLoop( RaspSetting* );

  /* access functions */
  void setLeftButton( Bool (*left)(short) )      { leftBut = left; }
  void setMiddleButton( Bool (*middle)(short) )  { middleBut = middle; }
  void setRightButton( Bool (*right)(short) )    { rightBut = right; }

  /* implementor functions */
  void   initialize();
  void   finish();
  int    body();
};
```

Member Function Definitions

```
#include <device.h>
#include <rw/defs.h>
#include "loop.h"
#include "setting.h"

/*************************************************************************
  process class GL-EVENT-LOOP member functions
 *************************************************************************/
GLEventLoop::GLEventLoop( RaspSetting *environ ): Process( environ )
{
  leftBut = middleBut = rightBut = NULL;
  bLoop = TRUE;
}

void GLEventLoop::initialize()
{
  qdevice( LEFTMOUSE );          loop.insert( LEFTMOUSE, leftBut );
  qdevice( MIDDLEMOUSE );        loop.insert( MIDDLEMOUSE, middleBut );
  qdevice( RIGHTMOUSE );         loop.insert( RIGHTMOUSE, rightBut );
}

void GLEventLoop::finish()
{
  unqdevice( LEFTMOUSE );
  unqdevice( MIDDLEMOUSE );
  unqdevice( RIGHTMOUSE );
}

int GLEventLoop::body()
{
  world->renderAll();

  JUMPS2();            /* this jump statement is mandatory */

  while( bLoop ) {
    if (qtest()) {
      event = qread( &val );
      bLoop = loop.event( event, val );
      world->renderAll();
    }

    if (bLoop) {
      PR_HOLD( 5, JUMP_1 );
    JMP_1:
      PR_HOLD( 4, JUMP_2 );
    JMP_2:
    }

  }
  return NO_JUMP;
}
```

WaitProcess

Header File

```
#include <process.h>

/*** Forward declaration ***/
class IPrPort_Wait;

/*****************************************************************************
  process class WAIT-PROCESS definition
 *****************************************************************************/
class WaitProcess: public Process {
  typedef enum {
    IP_VAL
  }; /* inPorts */

  IPrPort_Wait *inPort;
  Bool          bLoop;
  double        value;

 public:
  /* manager functions */
  WaitProcess( RaspSetting* );
  ~WaitProcess();

  /* implementor functions */
  void   initialize() {}
  void   finish() {}
  int    body();

  /* inports */
  IPort* inValue();

 private:
  void inValue( void* );
};

#include "wait.port"

#endif
```

Member Function Definitions

```
#include <rw/defs.h>
#include "wait.h"
#include "setting.h"

/*************************************************************************
  process class GL-EVENT-LOOP member functions
 *************************************************************************/
WaitProcess::WaitProcess( RaspSetting *environ ): Process( environ )
{
  bLoop = TRUE;
  inPort = new IPrPort_Wait[1];
  inPort[IP_VAL].setVarId( RS_DOUBLE );
  inPort[IP_VAL].setHandler( this, &WaitProcess::inValue );
}

 WaitProcess::~WaitProcess()
{
  delete [] inPort;
}

int WaitProcess::body()
{
  JUMPS2();              /* this jump statement is mandatory */

  while( bLoop ) {

    JMP_1: /* jump position */ ;
      if ( inPort[IP_VAL].getValue() ) {    /* received value */
        PR_HOLD( 10, JUMP_2 );
      }
      else {
        hold();
        return( JUMP_1 );
      }

    JMP_2:
  }

  return NO_JUMP;
}

IPort* WaitProcess::inValue()
{
  return (IPort*) &inPort[IP_VAL];
}

void WaitProcess::inValue( void *arg )
{
  value = *((double*) arg );
}
```

APPENDIX D
RASP CLASS LIBRARY

This appendix presents the complete list of classes found in the RASP toolkit. Classes are organized into tables according to functionality and purpose. A visual presentation of the toolkit's inheritance tree is shown in Figure D.1.

D.1 CLASS ORGANIZATION

Abstract	Class Name	Subtype of	Comments
	IdentityInfo		Name and NumId
	RaspSetting	IdentityInfo	
	RaspObject	IdentityInfo	
	HybridModel	RaspObject	
	Camera	RaspObject	
Yes	Window	IdentityInfo	
	Window3D		
	GLWindow	Window	
	GLWindow3D	Window3D, GLWindow	
	GLWindow2D	GLWindow	

Table D.6: Environmental Classes

Abstract	Class Name	Subtype of	Comments
Yes	Port		
	OPort	Port	
	OutPort<Type>	OPort	
	IPort	Port	
	InPort<Type>	IPort	
	RaspPorts		
	Connection		

Table D.7: Port Classes

Abstract	Class Name	Subtype of	Comments
	Chronos		Non-User Class
	Timing		
	Activity	Timing, IdentityInfo	
Yes	EventBase		
	Event	EventBase	
	TimeEvent	EventBase	
	ChainEvent	EventBase	
	StateEvent	EventBase	
	CallEvent	EventBase	
	Process	IdentityInfo	
	Procession	Timing, IdentityInfo	

Table D.8: Temporal Tools Classes

Abstract	Class Name	Subtype of	Comments
	KnotSequence		
	Basis		
Yes	GeoBase		
Yes	Geometry	GeoBase	
	GeoObject	GeoBase	Composite Geometry
	Sphere	Geometry	
	Cube	Geometry	
	GeoPoint	Geometry	
	GeoPoints	Geometry	
	GeoPointMatrix	Geometry	
	PolygonList	Geometry	
	Spline	Geometry	
	strainSpline	Spline	
	dataFitSpline	strainSpline	
Yes	BaseQuadric	Geometry	
	Ellipsoid	BaseQuadric	
	EllipticCone	BaseQuadric	
	Cone	EllipticCone	

Table D.9: Geometric Classes

Abstract	Class Name	Subtype of	Comments
Yes	PopupMenu	IdentityInfo	
	GLPopupMenu	PopupMenu	
	GLDrawMenu	PopupMenu	
	GLLinearMenu	PopupMenu	
	GLCircularMenu	GLDrawMenu	
	GL_Loop		

Table D.10: User Interface Classes

Abstract	Class Name	Subtype of	Comments
Yes	ColorBase		
	RGBColor	ColorBase	
	HSVColor	ColorBase	
	HSLColor	ColorBase	
Yes	Light	RaspObject	
	PointLight	Light	
	SpotLight	PointLight	
	LinearLight	Light	
	DirectLight	Light	
	AreaLight	Light	
Yes	Renderer		
	GLRenderer	Renderer	
	GLRenderer3D	GLRenderer	
	GLRenderer2D	GLRenderer	
	OptikRenderer	Renderer	

Table D.11: Rendering Classes

Abstract	Class Name	Subtype of	Comments
	Evolution<Type>		
Yes	RefClock		
	Motion	RefClock	
	ParameterizedData		
	Trace		Debugging Class
	OneTrace	Trace	Debugging Class

Table D.12: Specialized Classes

Abstract	Class Name	Subtype of	Comments
	dAngle		
	Point2		
	Point3		
	Point3List		
	DerivPoint3List	Point3List	
	Vector<Type>		
	Array<Type>		
	MinMaxArray<Type>	Array<Type>	
	MultiArray<Type>		
	CollectionSet<Type>	Array<Type>	
Yes	CacheTiming		
	ValueCache<Type>	CacheTiming	
	ArrayCache<Type>	CacheTiming, Array<Type>	
	MultiCache<Type>	CacheTiming	
Yes	HeapBase		
	Heap<Type>	HeapBase	
	MinHeap<Type>	Heap<Type>	
	MinHeapQueue<Type>	MinHeap<Type>	
	PtrHeap	HeapBase	
	MinPtrHeap<Type>	PtrHeap<Type>	
	MinPtrHeapQueue<Type>	MinPtrHeap<Type>	
	Orientation		
	Quaternion		
Yes	QueueBase<Type>		
	FifoQueue<Type>	QueueBase<Type>	
	LifoQueue<Type>	QueueBase<Type>	
	Rectangle		
	BinaryTree<Type>		
	N_Ary_Tree<Type>		
	Value<Type>		

Table D.13: Utility Classes

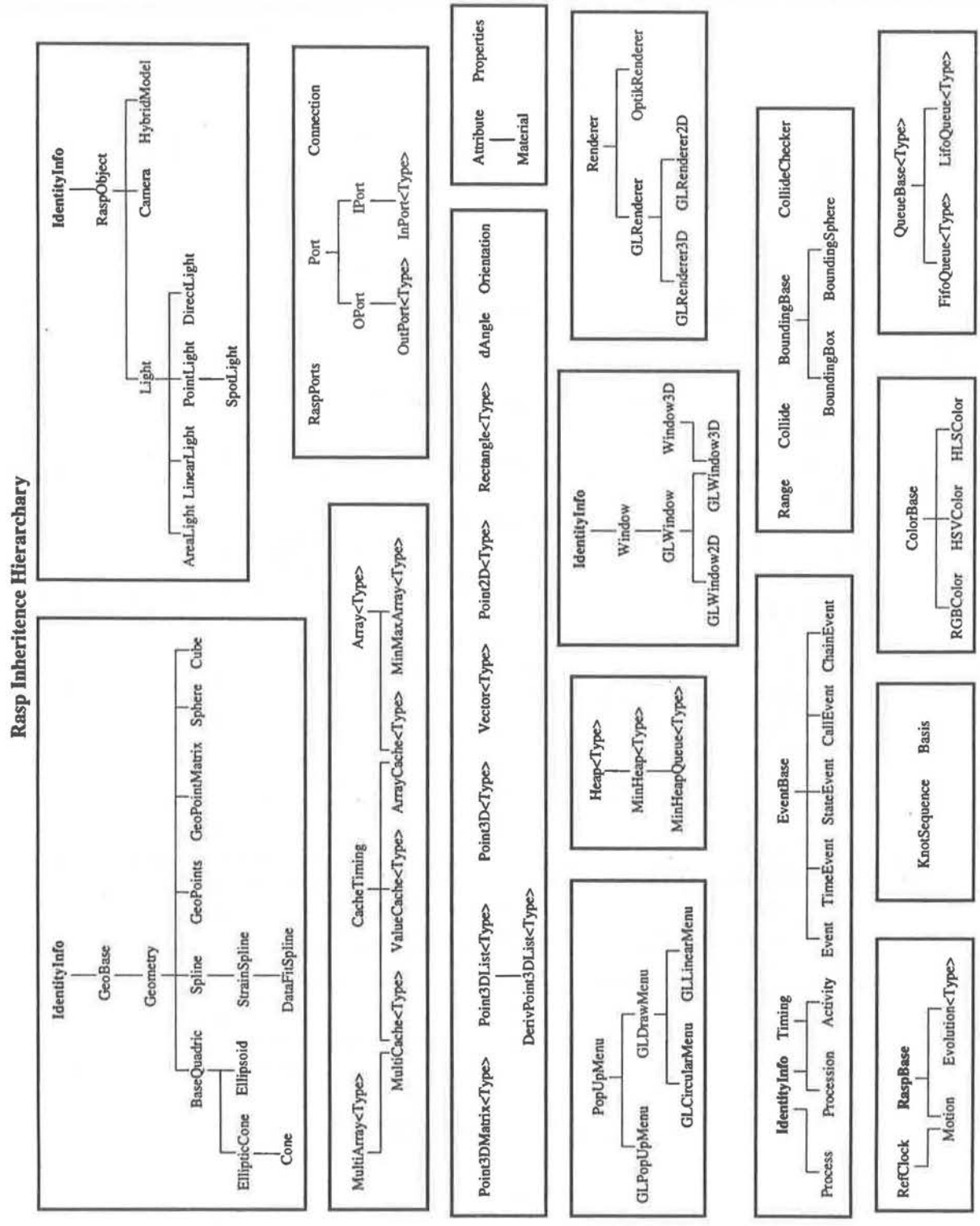

Figure D.1: Rasp Inheritance Tree

BIBLIOGRAPHY

[1] ACTON, D., COATTA, T., AND NEUFELD, G. The Raven System. Tech. Rep. 92-15, University of British Columbia, Vancouver, British Columbia, August 1992.

[2] ALLEN, J. F. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM 26*, 11 (November 1983), 832–843.

[3] ARBARB, F., HERMAN, I., AND SPILLING, P. An Overview of Manifold and Its Implementation. *Concurrency: Practice and Experience 5*, 1 (February 1993), 23–70.

[4] BEZIVIN, J. Some Experiments In Object-Oriented Simulation. In *OOPSLA '87* (1987), Association of Computer Machinery, pp. 394–405.

[5] BIGGERSTAFF, T., AND RICHTER, C. Reusability Framework, Assessment, and Directions. *IEEE Software 4*, 2 (March 1987), 41–49.

[6] BIRTWHISTLE, G. M., DAHL, O. J., MYHRHAUG, B., AND NYGAARD, K. *Simula Begin.* Petrocelli/Charter, New York, 1975.

[7] BLAIR, G. S., GALLAGHER, J. J., AND MALIK, J. Genericity vs Inheritence vs Delegation vs Conformance vs... *Journal of Object Oriented Programming* (Sept/Oct 1989), 11–17.

[8] BRUCE, K. M., AND WEGNER, P. An Algebraic Model of Subtypes in Object-Oriented Langauges. *SIGPLAN Notices 21*, 10 (October 1986).

[9] BUXTON, J. N. Writing Simulations in CSL. *Computer Journal 9*, 2 (1966), 137–143.

[10] CARRIERO, N., AND GELERNTER, D. Linda in Context. *Communications of the ACM 32*, 4 (April 1989), 444–458.

[11] CHUA, T.-S., WONG, W.-H., AND CHU, K.-C. Design and Implementation of the Animation Language SOLAR. In *New Trends in Computer Graphics: Proceedings of CG International '88* (Berlin, 1988), N. Magnenat-Thalmann and D. Thalmann, Eds., Springer-Verlag, pp. 15–26.

[12] COHEN, B., HAHN, D., AND SOIFFER, N. Pragmatic Issues in the Implementation of Flexible Libraries for C++. In *C++ Conference* (Washington, DC, April 1991), USENIX Association, pp. 193–202.

[13] COMPTROLLER GENERAL OF THE U.S. Ways to Improve Management of Federally Funded Computerized Models. Tech. Rep. LCD-75-111, General Services Administration, August 1976.

[14] Cox, B. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, MA, 1987.

[15] Davis, A. L., and Keller, R. M. Data Flow Program Graphs. *IEEE Computer 15*, 2 (February 1982), 26–41.

[16] de Champeaux, D., Lea, D., and Faure, P. The Process of Object-Oriented Design. In *OOPSLA'92 Conference Proceedings* (Vancouver, BC, 1992), The Association for Computing Machinery, pp. 45–62.

[17] Doi, A., Aono, M., Urano, N., and Uno, S. AVENUE: An Integrated 3-D Animation System. *Computer Graphics Forum 7* (1988), 27–43.

[18] Egbert, P. K. An Object-Oriented Approach to Graphical Application Support. Tech. Rep. UIUCDCS-R-92-1755, University of Illinois at Urbana-Champaign, Urbana, IL, 1992.

[19] Feiner, S., Salesin, D., and Banchoff, T. Dial: A Diagrammatic Animation Language. *IEEE Computer Graphics and Applications 2*, 7 (September 1982), 43–54.

[20] Fishwick, P. A., and Porr, H.-O. A. Using Discrete Event Modeling for Effective Computer Animation control. Tech. Rep. TR-005, University of Florida, Gainesville, FL, 1991.

[21] Fiume, E., Tsichritzis, D., and Dami, L. A Temporal Scripting Language for Object-Oriented Animation. In *Eurographics '87* (North-Holland, 1987), Eurographics Association, Elsevier Science Publishers B. V., pp. 283–294.

[22] Fleischer, K., and Witken, A. A Modeling Testbed. In *Graphics Interface '88* (1988), Graphics Interface, pp. 127–137.

[23] Fontana, M., and Neath, M. Checked Out and Long Overdue: Experiences in the Design of a C++ Class Library. In *C++ Conference* (Washington, DC, April 1991), USENIX Association, pp. 179–191.

[24] Gehani, N. H., and Roome, W. D. Concurrent C++: Concurrent Programming with Class(es). *Software - Practice and Experience 18*, 12 (December 1988), 1157–1177.

[25] Getto, P., and Breen, D. An Object-Oriented Architecture for a Computer Animation System. *The Visual Computer 6*, 2 (March 1990), 79–92.

[26] Goldberg, A. Introducing the Smalltalk-80 System. *Byte Magazine* (August 1981), 14–26.

[27] Gomez, J. E. Twixt: A 3D Animation System. In *Eurographics '84* (North-Holland, 1984), Eurographics Association, Elsevier Science Publishers B. V., pp. 121–133.

[28] Graphics Standards Planning Committee. Status Report of the Graphics Standards Planning Committee. *Computer Graphics 13*, 3 (August 1979).

[29] HALL, R. A., AND GREENBERG, D. P. A Testbed for Realistic Image Synthesis. *IEEE Computer Graphics and Applications 3*, 11 (November 1983), 10–20.

[30] HERATH, J., SAIKO, N., AND YUBA, T. Dataflow Computing Models. *IEEE Transactions on Software Engineering 14* (1988), 1805–1828.

[31] HILLS, P. R. SIMON - A Computer Simulation Language in ANGOL. In *Digital Simulation in Operations Research*, S. H. Hollingdale, Ed. Elseview North Holland, New York, 1967.

[32] HOARE, C. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, New Jersey, 1985.

[33] HOLZLE, U. The SELF Papers. Tech. Rep. CIS 209, Stanford University, Palo Alto, California, 1987.

[34] INGALLS, D. H. H. Design Principles Behind Smalltalk. *Byte Magazine* (August 1981), 286–298.

[35] INTERNATIONAL STANDARDS ORGANIZATION. International Standard Infromation Processing Systems - Computer Graphics - Graphical Kernel System for Three Dimensions (GKS-3D) Functional Description. Tech. Rep. ISO Document Number 8805:1988(E), American National Standards Institute, New York, 1988.

[36] JAYARAMAN, R., AND LEVAS, A. A Workcell Application Design Environment (WADE). In *CAD Based Programming for Sensory Robots*, B. Ravani, Ed. Springer-Verlag, Berlin, 1968, pp. 91–120.

[37] JOHNSON, R. E., AND RUSSO, V. F. Reusing Object-Oriented Designs. Tech. Rep. UIUCDCS-R-91-1696, University of Illinois, Urbana-Champaign, Illinois, May 1991.

[38] JOHNSON, R. E., AND ZWEIG, J. M. Delegation in C++. *Journal of Object Oriented Programming* (Nov/Dec 1991), 31–34.

[39] JORDAN, D. Implementation Benefits of C++ Language Mechanisms. *Communications of the ACM 33*, 9 (September 1990), 61–64.

[40] KALRA, D., AND BARR, A. H. Modeling with Time and Events in Computer Animation. In *Eurographics '92* (Cambridge, England, August 1992), Eurographics Association, Blackwell Publishers, pp. 45–58.

[41] KASS, M. CONDOR: Constraint-Based Dataflow. In *Computer Graphics* (Chicago, IL, July 1992), SIGGRAPH, Association of Computing Machinery, Inc., pp. 321–330.

[42] KAVIAT, P. J., MARKOWITZ, H. M., AND VILLANUEVA, R. *SIMSCRIPT II.5 Programming Language.* CACI, Los Angeles, 1983.

[43] KAZMAN, R. HIDRA: An Architecture for Highly Dynamic Physically Based Multi-Agent Simulations. *International Journal in Computer Simulation* (1993). (To appear).

[44] KERRIDGE, J., AND SIMPSON, D. Communicating Parallel Processes. *Software - Practice and Experience 16*, 1 (January 1986), 63–86.

[45] KRASNER, G. E., AND POPE, S. T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming 1*, 3 (August/September 1988), 26–49.

[46] KUBOTA PACIFIC COMPUTER, INC. *Dore Programmer's Guide*, 5 ed., Sept 1993. Dore Graphics Library.

[47] LALONDE, W. R., WHITE, P., AND MCGUIRE, K. Coordinators: A Mechanism for Monitoring and Controlling Interactions Between Groups of Objects. Tech. Rep. SCS-TR-190, Carleton University, Ottawa, Ontario, April 1991.

[48] LI, Q., AND MACKEY, W. Object-Oriented Discrete Event Simulation using Ada. In *Object-Oriented Simulation 1991* (San Diego, CA, 1991), R. K. Ege, Ed., The Society of Computer Simulation (SCS), pp. 51–56.

[49] LIEBERMAN, H. Using Prototypical Objects to Implement Shared Behavior in Object Orient Systems. In *OOPSLA '86 Proceeedings* (September 1986), ACM, pp. 214–248.

[50] LINTON, M. A., VLISSIDES, J. M., AND CALDER, P. R. Composing User Interfaces with InterViews. *Computer 22*, 2 (February 1989), 8–22.

[51] LIPPMAN, S. B. *C++ Primer*, 2 ed. Addison-Wesley Publishing Company, Reading, Mass, 1991.

[52] MAGNENAT-THALMANN, N., AND THALMANN, D. Use of High-level 3-D Graphical Types in the MIRA Animation System. *IEEE Computer Graphics and Applications 3*, 9 (December 1983), 9–16.

[53] MAGNENAT-THALMANN, N., AND THALMANN, D. Three-Dimensional Computer Animation: More an Evolution Than a Motion Problem. *IEEE Computer Graphics and Applications 5*, 10 (October 1985), 47–57.

[54] MAIOCCHI, R., AND PERNICI, B. Directing an Animated Scene with Autonomous Actors. *The Visual Computer*, 6 (Decemeber 1990), 359–371.

[55] MAK, V. W. DOSE: A Modular and Reusable Object-Oriented Simulation Environment. In *Object-Oriented Simulation 1991* (San Diego, CA, 1991), R. K. Ege, Ed., The Society of Computer Simulation (SCS), pp. 3–11.

[56] MCLACHLAN, D. R. CORY: An Animation Scripting System. Tech. Rep. 85006, Rensselaer Polytechnic Institute, 1985. Master's Thesis, Rensselaer Design Research Center.

[57] NANCE, R. E. The Time and State Relationship in Simulation Modeling. *Communications of the ACM 24*, 4 (April 1981), 173–179.

[58] NEELAMKAVIL, F. *Computer Simulation and Modeling.* John Wiley & Sons, Chichester, 1987.

[59] NEIGHBORS, J. M. The Draco Approach to Constructing Software from Reusable Components. *IEEE Trans. Software Engineering* (Sept 1984), 564–574.

[60] NERSON, J.-M. Applying Object-Oriented Analysis and Design. *Communications of the ACM 35*, 9 (September 1992), 63–74.

[61] NEWELL, W. M. *The Utilization of Procedure Models in Digital Image Synthesis.* PhD thesis, University of Utah, 1975. Department of Computer Science.

[62] O'DONNEL, T., AND OLSON, A. GRAMPS - A Graphics Language Interpreter for Real-time, Interactive, 3-Dimensional Picture Editing and Animation. In *Computer Graphics* (Dallas, Texas, August 1981), SIGGRAPH, Association of Computing Machinery, Inc., pp. 133–142.

[63] PAETH, A. W., AND BOOTH, K. S. Design and Experience with a Generalized Raster Toolkit. In *Graphics Interface '86* (1986), Graphics Interface, pp. 91–97.

[64] PAI, D., AND LEU, M. C. Ineffabelle - An Environment for Interactive Computer Graphic Simulation of Robot Applications. *Proceedings of IEEE International Conference on Robotics and Automation* (1986), 897–903.

[65] PAI, D. K. Least Constraint: A Framework for the Control of Complex Mechanical Systems. In *Proceedings of the American Control Conference* (1991), pp. 1615 – 1621.

[66] PALMER, R. S., AND CREMER, J. F. SIMLAB: Automatically Creating Physical Systems Simulators. Tech. Rep. TR91-1246, Cornell University, Ithica, NY, November 1991.

[67] PEGDEN, C. D. Introduction to SIMAN. In *The 1986 Winter Simulation Conference* (Washington, D.C., Dec 8-10 1986), J. R. Wilson, J. O. Henrikson, and S. D. Roberts, Eds., Institute of Electrical and Electronic Engineers, pp. 95–103.

[68] POTMESIL, M., AND HOFFERT, E. M. FRAMES: Software Tools for Modeling, Rendering, and Animation of 3D Scenes. In *Computer Graphics* (1987), SIGGRAPH, Association of Computing Machinery, Inc., pp. 85–93.

[69] PRATT, D. B., FARRINGTON, P. A., BASNET, C. B., BHUSKUTE, H. C., KAMATH, M., AND MIZE, J. H. A Framework for Highly Reusable Simulation Modeling: Separating Physical, Information, and Control Elements. In *The 24th Annual Simulation Symposium* (New Orleans, Louisiana, 1991), IEEE Computer Society Press, pp. 254–261.

[70] PRITSKER, A. A. B. *Introduction to Simulation and SLAM II.* Systems Publishing Corp., West Lafayetter, IN, 1986.

[71] PRITSKER, A. A. B., AND PEGDEN, C. D. *Introduction to Simulation and SLAM.* Halstead Press, New York, 1979.

[72] REYNOLDS, C. Computer Animation with Scripts and Actors. In *Computer Graphics* (Boston, Mass, July 1982), SIGGRAPH, Association of Computing Machinery, Inc., pp. 289–296.

[73] ROGUEWAVE. *Roguewave.* RogueWave Associates, Inc., XXX, 1900.

[74] SANDERSON, D. P., SHARMA, R., ROZIN, R., AND TREU, S. The Hierarchical Simulation Language HSL: A Versatile Tool for Process-Oriented Simulation. *ACM Transactions on Modeling and Computer Simulation 1*, 2 (April 1991), 113–153.

[75] SAUNDERS, J. H. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming 1*, 5 (March/April 1989), 5–11.

[76] SCHMUCKER, K. J. *Object-Oriented Programming for the Macintosh.* Hayden Book Company, 1986.

[77] SCHRIBER, T. J. *Simulation Using GPSS.* Wiley, New York, 1974.

[78] SCHRIBER, T. J. *An Introduction to Simulation Using GPSS/H.* Wiley, New York, 1990.

[79] SCHWETMAN, H. CSIM: A C-based, Process-Oriented Simulation Language. In *The 1986 Winter Simulation Conference* (Washington, D.C., Dec 8-10 1986), J. R. Wilson, J. O. Henrikson, and S. D. Roberts, Eds., Institute of Electrical and Electronic Engineers, pp. 387–396.

[80] SEARS, F. W., ZERMANSKY, M. W., AND YOUNG, H. D. *University Physics*, 6th ed. Addison-Wesley Publishing Company, Reading, Mass, 1982.

[81] STEIN, A. L. Delegation Is Inheritance. In *OOPSLA '87 Proceeedings* (October 1987), ACM, pp. 138–146.

[82] STERN, G. BBOP: A System for 3D Keyframe Figure Animation. In *Siggraph '83 Tutorial Notes on Computer Animation.* Association for Computing Machinery, 1983, pp. 240–243. Vol. 7.

[83] STRAUSS, P. S., AND CAREY, R. An Object-Oriented 3D Graphics Toolkit. In *Computer Graphics* (Chicago, IL, July 1992), SIGGRAPH, Association of Computing Machinery, Inc., pp. 341–349. SGI Inventor Toolkit.

[84] STROUSTRUP, B. Parameterized Types for C++. In *Proc. of C++ Conference* (Denver, CO, October 1988), USENIX Association, pp. 1–18.

[85] TEO, P. K. An Entity-Relationship Based Object-Oriented Data Model. Master's thesis, University of Singapore, 1992.

[86] TERZOPOULOS, D., AND WITKIN, A. Physically-based models with rigid and deformable components. In *Graphics Interface '88* (June 1988), pp. 146–154.

[87] TRUMBORE, B., LYTLE, W., AND GREENBERG, D. P. A Testbed for Image Synthesis. In *Eurographics '91* (North Holland, 1991), Eurographics Association, Elsevier Science Publishers, pp. 467–480.

[88] UPSTILL, S. *The RenderMan Companion.* Addison-Wesley, Redwood City, Ca, 1990.

[89] VAN DAM, A. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics 22*, 3 (July 1988), 125–218.

[90] VAUGHAN, P. W., NEWTON, D. E., AND JOHNS, R. P. PRISM: An Object-Oriented System Modeling Environment in C++. In *Object-Oriented Simulation 1991* (San Diego, CA, 1991), R. K. Ege, Ed., The Society of Computer Simulation (SCS), pp. 32–39.

[91] WEGNER, P. Dimensions of Object-Based Language Design. *SIGPLAN Notices 22*, 12 (October 1987), 168–182.

[92] WISSKIRCHEN, P. *Object-oriented graphics : from GKS and PHIGS to object-oriented systems.* Springer-Verlag, Berlin, 1990.

[93] WOODSIDE, C. M., AND CAVERS, J. Block Diagram Computer Language for Educational Animations. *IEEE Trans Educ 19*, 4 (November 1976), 133–139.

[94] WYBOLT, N. Experiences with C++ and Object-Oriented Software Development. In *C++ Conference* (San Fransisco, CA, April 1990), USENIX Association, pp. 1–9.

[95] WYVILL, B., MCPHEETERS, C., AND NOVACEK, M. Specifying Stochastic Objects in a Hierarchical Graphics System. In *Graphics Interface '85* (1985), Graphics Interface, pp. 17–20.

[96] ZEIGLER, B. P. *Theory of Modelling and Simulation.* John Wiley & Sons, New York, 1976.

[97] ZEIGLER, B. P. Multifaceted, Multiparadigm Modeling Perspectives: Tools for the 90's. In *The 1986 Winter Simulation Conference* (???, Dec 8-10 1986), J. Wilson and ???, Eds., Association of Computer Machinery, Association of Computer Machinery, pp. 708–712.

[98] ZELEZNIK, R. C., CONNER, D. B., WLOKA, M. M., ALIAGA, D. G., HUANG, N. T., HUBBARD, P. M., KNEP, B., KAUFMAN, H., HUGHES, J. F., AND VAN DAM, A. An Object-Oriented Framework for the Integration of Interactive Animation Techniques. In *Computer Graphics* (Las Vegas, Nevada, July 28 - August 2 1991), SIGGRAPH, Association of Computing Machinery, Inc., pp. 105–111.

[99] ZELTZER, D. Knowledge-based Animation. In *SIGART Interdisciplinary Workshop on Motion* (Toronto, 1983), ACM Siggraph, pp. 187–192.

[100] ZELTZER, D. Towards An Integrated View of 3-D Computer Character Animation. In *Graphics Interface '85* (1985), Graphics Interface, pp. 105–115.

GLOSSARY

Abstract Class - a class that serves a prototype for its subclasses. It is not usually possible to create an instance of an abstract class.

Activity - the state of an model over an interval. Delimited by two successive events, an activity represents a period of inactivity or period of static actions.

Activity Scanning - a discrete event methodology where the actions of a simulation are partitioned into activities. Activities, maintained by a conditions-list, are executed when their associated contingency tests are satisfied.

Base Class - a class that adds properties and functionalities to its subclasses. Base classes do not serve as prototypes, are always possible to instantiate, and are usually situated at the top of an inheritance tree.

C++ Class - a prototype for user-independent data types. Classes consist of a set of data members and member functions. Class instantiations are first-class objects.

C++ Method function - operation applied to a class' data members to induce changes to an object's state.

Connection Paradigm - a design approach based upon the idea of structuring models and systems as sets of interconnected components. Components are joined via links and ports.

Control Mode - a method used to describe the behavior of animated objects. Computer animation are labeled as guiding, animator level, or task-level according to their control mode(s).

Controller - a thing that controls or regulates the attributes and behaviors of another entity. In turn, controllers may also be governed by other controllers.

Data to Image Translation - the act of interpreting geometric data possessing physical attributes into a form usable for the generation of computer images.

Delegation - a term for dynamic hierarchical resource sharing. Delegation is the act of empowering or giving authority to others to perform some operation or function.

Discrete Event - a temporal progression technique characterized by state changes occurring in discontinuous jumps and events arbitrarily separated from each other. All actions within a simulation are executed at specific event times.

Display List - a hierarchical data structure used in many computer graphics toolkits to associate physical information with geometric primitives. Its organizational structure accommodates the design image renderers.

Event - an important happening that instantaneously alters the state of a simulation. When an event is activated, time is suspended. Time is resumed immediately after the event induces it state altering modifications.

Event List - an ordered data structure used in next event simulation to organize the activation of events. Events, ordered according to their activation times, are dynamically added and removed from the event list during the course of a simulation.

Event Scheduling - a discrete event methodology where the scheduling of events is controlled by an event list. As a simulation progresses, events are placed, executed, and removed from the event list.

Feature Ports - first class entities which govern the slots of a hybrid model.

First-Class Conditional - contingent predicate constructed as first-class object with sets of "input" and "output" ports. The values of output ports are altered each time the conditional object perceives a change in its input ports.

First-Class Object - an object which retains state and can react to messages.

Global State - the condition of the elements in a simulation which define the environment. A simulation's global state is defined by the values of its environmental state variables.

Hybrid Model - model design based on slots and ports that promotes an unified user-interface and a rendering supportive architecture.

Interface - a communication technique used to bring two or more things together in an association. An interface describes the type, quality, and nature of the patterns of communication between distinct entities.

IMVCD Pentad - an object-oriented framework for the construction of time-varying simulations. Pentad components include model, informer, viewer, controller, and delegator.

Key-framing - a computer animation technique in which users specify the values of variables at key points in time while the computer fills the temporal gaps with intermediate values.

Link - anything that joins together or passes information between two connection points. In RASP, links are used to transfer information from "out" ports to "in" ports.

Model - a thing which imitates an entity worth copying. A simulation is comprised of a set of interacting models.

Model Specification - the act of specifying the detailed description of a model in a simulation. The type and behavioral patterns of models are defined during this act.

Modeling - the act of creating the various parts of a simulation. This action entails users to describe the models of a simulation and to characterize each model's behavior.

Motion Specification - the act of specifying a change in movement of physical things. Key-framing and scripting techniques are common paradigms of motion specification.

Multiple Interface - a communication technique involving many interfaces. Communication between various entities is accomplished in a variety of manners.

Next Event Simulation - a simulation using event scheduling to control the activation of state changes and the progression of time.

Object Modeling - the act of defining the attributes, characteristic traits, and behaviors of simulation models.

Object-Oriented Framework - a set of design rules for a collection of collaborating objects. If defines how a system is divided into components and the manner that functions are divided among each of the individual groups of objects.

Object-Oriented Programming - a programming technique characterized by the development of first-class objects and object-oriented systems.

Object-Oriented System - a set of first-class objects working together to form a whole. In an object-oriented system, objects communicate via message passing and exhibit polymorphism.

Occurrence - an action occurring over a finite length of time. The continual activation of a single event over a finite length of time defines an occurrence.

Port - unidirectional data monitors which observe and regulate the flow of information "in" and "out" of a host entity.

Process - an "independent" program or procedure that uses the resources of a system to fulfill its goals. A process' routine can be explicitly describe in terms of time flow.

Process Interaction - a discrete event methodology which stresses the interaction between the entities of a simulation. The behavior of a simulation is governed by the flow of processes through time.

Reusable code - segments of programming code which may be used again. The use of reusable code facilitates the rapid development of new computer programs.

Robotics Application - a computer program simulating the motion of articulated figures. Robotics applications are generally developed to design and test new articulated configurations before they are actually physically built.

Scenario Modeling - the act of defining the environment of a simulation.

Scripting Language - a special vocabulary or notation used to describe the dynamic changes occurring in a computer animation.

Second-Class Object - an object which does not retain state and can not react to messages. Objects of this type usually perform one function and are hard to modify.

Simulation - a device to reproduce or represent test conditions likely to occur in real situations. A simulation emulates and analyzes the behaviors of its simulated conditions.

(Simulation) Environment - all the conditions and surrounding influences that affect the behavior and development of a simulation. The models of a simulation exist and interact in an environment.

Simulation Kernel - a central part of a simulation. It controls the progression of time and ensures that every model is aware of the global state.

Simulation Language - a set of high-level expressions which facilitate the specification of a computer simulation. Expressions, assembled into meaningful phrases by users, are interpreted (by a computer's compiler) to construct, coordinate, and manage the dynamic changes occurring in a simulation.

Simulationist - an individual who creates simulation tools, such as toolkits, languages, and simulators. Users utilize the tools created by simulationists to create simulations.

Simulator - a thing, such as a computer, that simulates. Simulations are devised and performed on simulators.

State - the condition of a model or thing at a certain time. The state of a model is defined by the value of its state variables.

State Variable - descriptive thing or quality that characterizes the range of states and types of behaviors a model can achieve.

Time Structure - the unit of measurement for a temporal system. Time can be mapped to the set of rational number, floating point numbers, or integer numbers.

Time-varying Simulation - a simulation whose behavior is strongly dependent upon the value of simulated time. References to explicit temporal values are integral elements of the simulation.

Transitional Structures - programming constructs that enable users to define runtime changes to an entity's state. Events, activities, and processes are exemplary transitional structures.

World View - the viewpoint from which users develop simulations. The world view embodied by a simulation strategy establishes the methodology users use to specify the components of a simulation and their interactions.