Design & Implementation of a 3D Graphics System: An Interactive Hierarchical Modeling Approach

by

Nabeel Al-Mouslli

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

June, 1995

Design & Implementation of a 3D Graphics System
An Interactive Hierarchical Modeling Approach

BY

Nabeel Al-Mouslli

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

June 1995

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis written by

Nabeel Al-Mouslli

under the direction of his thesis advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Thesis Committee:

Dr. Subbarao Ghanta (Chairman)

Dr. Bassel Arafeh (Member)

Dr. Mohsen Guizani (Member)

Department Chairman

Dean, College of Graduate Studies

24/5/95

Date

To My Parents

Acknowledgments

All praise be to Allah for his limitless help and guidance. Peace and blessings of Allah be upon his prophet Muhammad.

Acknowledgment is due to King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for the generous help and support for this research.

I would like to express my profound gratitude and appreciation to my thesis chairman, Dr. Subbarao Ghanta, Associative Professor of Information and Computer Science, for his guidance and patience throughout this thesis. His continuous support and encouragement can never be forgotten. I would also like to thank Dr. Bassel Arafeh, Assistant Professor of Information and Computer Science, Dr. Mohsen Guizani, Associative Professor of Computer Engineering, and Dr. Muhammed Al-Mulhem, Assistant Professor and chairman of Information and Computer Science Department, for their consistent support and valuable suggestions.

I also wish to thank faculty, research assistants, and the staff members of the Information and Computer Science Department for their support. The encouragement and good wishes of my friends are also worthy of acknowledgment. Finally, special thanks must be given to my family for their encouragement and moral support.

Contents

List of Tables

List of Figures

Abstract

Name: Nabeel Mohammad Adnan Al-Mouslli

Title: Design & Implementation of a 3D Graphics System
An Interactive Hierarchical Modeling Approach

Major Field: Computer Science

Date of Degree: April, 1995

Graphics modeling is an essential part of computer graphics that plays a vital role in many applications. Existing graphics modeling systems don't offer sufficient flexibility to the end user (the designer) at different levels of abstraction. Our goal is to design a graphics modeling system that raises the level of interaction to the application level. To achieve this goal, we developed various techniques that enhance the modeling process. The object-orientation methodology was adopted and incorporated with a hierarchical modeling scheme that supports bottom-up construction and top-down decomposition. The modeling process was enhanced further by supporting the manipulation of constraints on object attributes which automates satisfying system requirements. Based on these techniques, a 3D graphics modeling system was designed. A subset of the design was implemented on the *NeXTStep* platform to help in testing and developing the introduced techniques.

Master of Science Degree

King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

ملخص

الاسم: نبيل محمد عدنان الموصللي

العنوان: تصميم و تنفيذ نظام رسومات ثلاثية الأبعاد
من خلال نمذجة هرمية تفاعلية

التخصص: علوم الكمبيوتر

تاريخ التخرج: نيسان، ١٩٩٥

تكوّن نمذجة الرسومات جزءاً هاماً من الرسم بواسطة الكمبيوتر وتلعب دوراً حيوياً في عدة تطبيقات. أنظمة الرسومات الموجودة حالياً لا تقدم للمستخدم (المصمم) مرونة كافية لمستويات متعددة من التجريد. الهدف من هذا العمل هو تصميم نظام نمذجة رسومات قادر على نقل مستوى التفاعل إلى مستوى التطبيق. ولهذا الغرض قمنا بتطوير عدة تقنيات لمعالجة عملية النمذجة، ابتداءً من تبني منهج غرضية-التوجه و دمجه مع نظام نمذجة هرمي يتميز بإمكانية البناء و التجزيء، إضافة إلى تحسين عملية النمذجة بجعلها قادرة على معالجة القيود على خصائص الأشياء، مما يسهل عملية الوصول إلى المتطلبات. و بناء على هذه التقنيات، قمنا بتصميم نظام نمذجة رسومات ثلاثية الأبعاد و بتنفيذ جزء من هذا التصميم على بيئة *NeXTStep* و ذلك للمساعدة في اختبار و تطوير التقنيات المدخلة.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول و المعادن

الظهران، المملكة العربية السعودية

Chapter 1

Introduction

Graphics modeling is an essential part of computer graphics that plays a vital role in many applications ranging from engineering and aerospace design to advertising and entertainment. The various disciplines of mechanical, electrical and civil engineering have all employed CAD systems extensively in the design process.

Over the past few years, computer graphics technology has improved dramatically. New algorithms produce near-photorealistic images, and new hardware accelerators lower the time required to run them. Much work has been devoted to improving image quality and reducing rendering time, but less work has been devoted to making graphics modeling easy to use in a broad set of applications.

Today, two types of graphics modeling systems exist. Systems of the first type are application-specific like the CAD systems that are now common in many industries. These systems can be useful only to the domain specialists. Systems of the other type are general graphics packages like the graphics packages based on PHIGS. The problem with these systems is that they expect the user to work at a low programming level to model his design. Our goal is to provide an easy to use, versatile graphics modeling framework that can be easily customized to a broad set of applications. The modeling framework is to have the ability to interact with the user at his application abstract level and doesn't require him to have programming skills.

In an application with graphics modeling, the user assembles, transforms, and manipulates data by interpreting portions of it as logically connected and interdependent. Thus, the user naturally encapsulates individual data groups as separate *objects.* Wether or not the data represents real-world phenomena, the user thinks of each data group as a distinct object. For example, a mechanical engineer designing a machine thinks in terms of components that build the machine, each as a separate object. This applies almost to all other applications. Therefore, object-orientation is a natural vehicle for supporting graphics modeling and, consequently, our work adopts the object-oriented paradigm as a basis for graphics modeling.

The target of our work is to design and implement a 3D graphics modeling

system. This Chapter presents the motivation behind this system, introduces the objectives of our work and the necessary background, discusses the drawbacks of the current systems, and concludes with an outline of the thesis.

1.1 Motivation

Modeling is a process which abstracts the real world into a smaller-scale representation called a *model*. Unnecessary details of the real world that are considered to be irrelevant to the modeling process are removed from the model. If the resulting model is too complex, further abstraction may be necessary, to reduce the problem to a manageable size. Figure 1.1 shows a graphical representation of the modeling process [TP91].

To complete the modeling process, the final model is analysed and manipulated to reach a feasible solution. Finally, the reached solution is turned into a real system. An architect, for example, designs a building (the real-world system) using a CAD system (the modeling tool). The building model ignores all irrelevant details, like the furniture to be installed in the building. Once finalized, the building model can be turned into a building on the ground.

In real world, numerous objects exist and, therefore, it is not realistic to develop

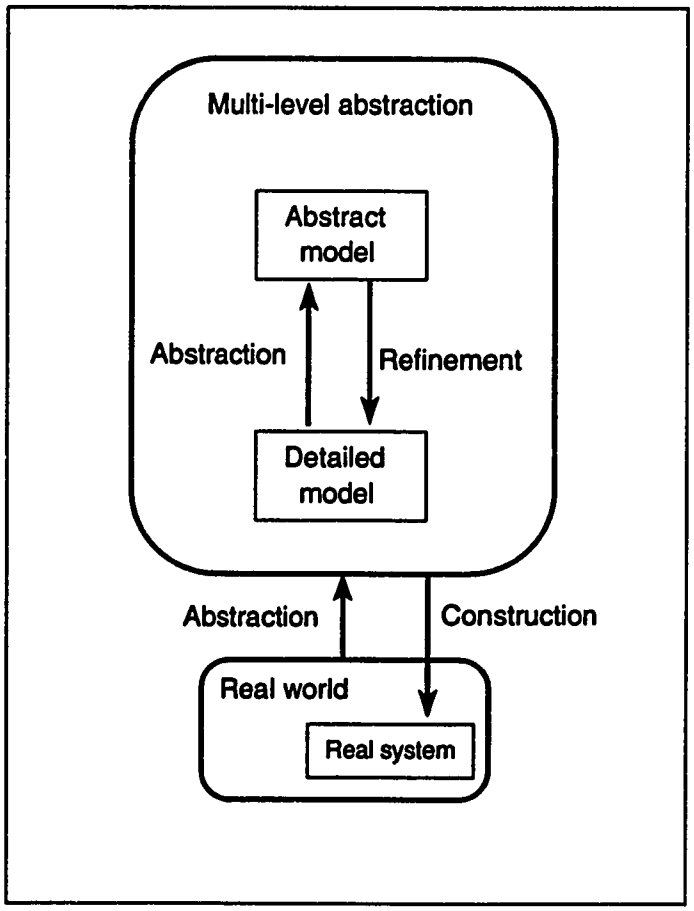

Figure 1.1: The modeling process

individual models for each of them. An alternative is to model some basic objects. Complex objects can then be modelled using the basic objects with some rigorous operations. This is what we refer to as hierarchical modeling. In hierarchical modeling the real world is abstracted as a hierarchy of objects, with each of these objects having domain attributes and constraints that makes it an integral part of the abstract model. Each of these objects in turn is a model of a subsystem.

A tool that supports hierarchical modeling and has the ability to facilitate the design interactively is needed.

1.2 Objectives

From the previous discussion it is evident that a good modeling environment should have:

- Support for defining design requirements: domain attributes and constraints.
- Support for hierarchical construction.
- Support for interaction that allows a designer to work at a suitable level of abstraction.

Keeping these in mind, we can now state the objectives of our work to be:

1. Extraction of requirements of a modeling system through simulated walk-throughs to identify the mechanisms which need to be realized for a modeling system.

2. Detailed consideration of the required mechanisms: attributes, constraints and hierarchical construction.

3. Design of an object-oriented modeling system to provide the required mechanisms.

4. Customization of the modeling system to 3D graphics.

5. Implementation of a subset of the 3D graphics system.

1.3 Background

In this Section the preliminary background information which is assumed for the subsequent chapters is presented. The proposed modeling system provides the user with an object-oriented view and its design is based on the object-oriented programming paradigm which is introduced next.

1.3.1 Fundamentals of Object-Oriented Programming

In the objected-oriented programming paradigm [Weg92], objects are the atomic units of encapsulation. Classes manage collections of objects, and inheritance structures collections of classes.

Objects

Objects partition the state of a computation into encapsulated chunks. Each object has an interface of operations that control access to an encapsulated state (Figure 1.2). The operations determine the object's behavior, while the state serves as a memory of past invocations that can influence future actions [Weg92]. An object named Point with state variables (instance variables) x, y and four operations for reading and changing them can be defined as follows:

Point: **object**

x:=0;y:=0;

read-x(): return x;

read-y(): return y;

change-x(dx): x:=x+dx;

change-y(dy): y:=y+dy;

By accepting messages only through read and change operations, the object Point

protects its instance variables x, y from direct manipulation by other objects. The operations of an object share its state. Therefore, state changes by one operation can be seen by subsequently executed operations. For example, $read-x$ and $change-x$ share the variable x which is non-local to these operations although x is local to the object.

Classes

Classes specify the behavior of collections of objects with common operations. For example, points with read and change operations may be specified by a class Point with two instance variables x, y and four explicit operations as follows:

Point: **class**

x:=0;y:=0;

read-x(): return x;

read-y(): return y;

change-x(dx): x:=x+dx;

change-y(dy): y:=y+dy;

create(): return aNewPointID;

Classes specify the set of messages accepted by objects of the class. They define an encapsulation interface. Classes also serve as templates for creating objects with

the specified interface and implementation behavior. For example, two instances $p1, p2$ of the class Point can be created as follows:

p1 := create Point;

p2 := create Point;

Each of the two point instances $p1, p2$ is an encapsulated chunk of state whose behavior is determined by the class operations.

Inheritance

Inheritance is a mechanism for sharing the code or behavior common to a collection of classes. It factors shared properties of classes into superclasses and reuses them in the definition of subclasses. Programmers can therefore specify incremental changes of class behavior in subclasses without modifying the already specified classes. Subclasses inherit the code of superclasses, to which they can add new operations and possibly new instance variables.

Figure 1.3 describes a class of mammals with persons and elephants as its subclasses. The class of persons has mammals as its superclass and students and males as its subclasses. Each instance John, Joan, Bill, Mark and Dumbo - has a unique base class. An instance's membership in more than one base class such as John's being both a student and a male, cannot be expressed.

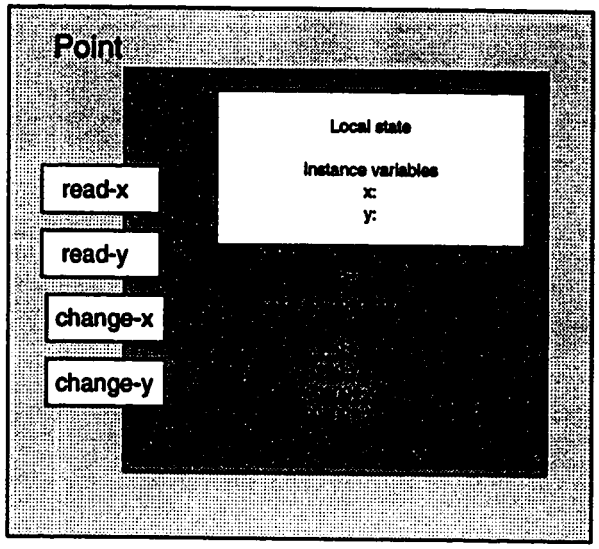

Figure 1.2: Point object

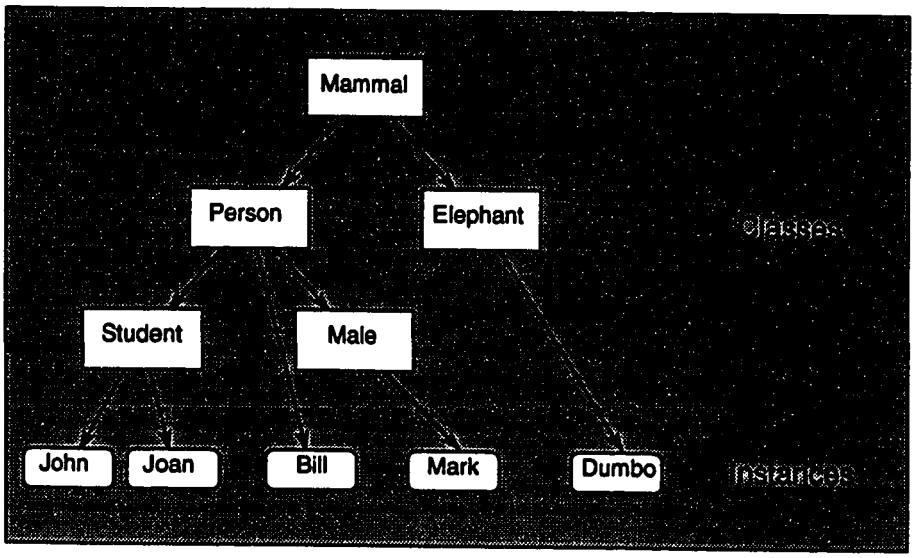

Figure 1.3: Example of an inheritance hierarchy

1.3.2 Advantages of The Object-Oriented Paradigm

The object-oriented paradigm is useful for both the system user and developer. It lets the system user abstract the real world entities into objects in a natural manner. When the system is developed based on object-orientation as well, the user view maps easily into the system developer view. The simplicity of this mapping avoids a lot of the redundant overhead on the part of the developer and lets him work at a higher level of abstraction.

Other advantages of the object-oriented paradigm include: encapsulating object data and interfaces, and extending the development environment through inheritance, subclassing and code reuse.

Many people have anticipated the benefits of combining graphics with object-orientation [EK92], and few researchers have designed systems towards this. Wisskirchen [Wis90] implemented the functionality of GKS and PHIGS in an objected-oriented fashion using Smalltalk. Other object-oriented systems include HOOPS [Kli89] and Dore [Kub91].

HOOPS (Hierarchical Objected-Oriented Picture System) allows application programmers to group primitives and attributes as objects. Once an object has been defined in this manner, it can then be used to form other objects providing a hier-

archical construction capability.

The drawback of these systems is that they require the user to work at a low level to specify his requirements rather than allowing him to work at an abstract level suitable to the designer.

1.3.3 3D Graphics Background

This Section briefly introduces the basics of 3D graphics namely, solid modeling and transformations. The interested reader is referred to [FV90] for a detailed treatment of the subject.

Solid Modeling

The building blocks that the proposed 3D graphics system uses are 3D solid components. These components are expected to be supplied as an input to the system, and so our work doesn't deal with solid modeling issues. For the sake of completeness it is discussed here briefly.

The most widely used solid representations are *constructive solid geometry* (CSG) and *boundary representation.* CSG allows an object to be defined as a boolean combination of other objects such as intersection and union [FV90]. A *solid* is

defined as a set of points in 3D space, with a surface separating points in the set from those not in the set. Given a number of solid objects (sets of points) in space, one can either combine them into composite objects (using the operation union), use only those points they have in common (set intersection) or use one solid to "carve out" another by subtraction (set difference).

In boundary representation a solid is defined as a set of surfaces that enclose a space. The surfaces must join together to completely enclose the space, providing a well-defined inside and outside such that every point in space is either inside or outside the solid.

To display the edges of a 3D solid on a computer display the boundary representation is much easier to handle, whereas CSG is much easier for the user to handle than the boundary representation [KW87]. So, many existing systems employ CSG method for inputting an object, and automatically transform the CSG representation into the boundary representation to store the object internally [Req80].

The developed 3D graphics tool, addressed in this thesis, uses RenderMan to represent the building blocks. RenderMan is a scene description methodology. RenderMan offers comprehensive support to describe objects, scenes, lights, and cameras so that a computer can create images from them [Ups90b] [Ups90a]. A user of the RenderMan interface writes, compiles and runs a program in a conventional

programming language (such as C), calling a set of RenderMan procedures to describe the scene. RenderMan uses the boundary representation to define solids as an assembly of surfaces.

Basic Geometric Transformations

Except for references to faces and edges, the only data that are stored in the boundary representation are the vertices in the 3D space, i.e., vectors of the form (x, y, z). In order to be able to view an object from different perspectives (angles) and to zoom in and out the particular object, one can apply three geometric transformations: translation, rotation and scaling. Transforming a geometric object is achieved by transforming all of its vertices.

A transformation of a vertex can be defined as a multiplication of the vertex with a corresponding transformation matrix. Then, we would be able to combine different transformation matrices by multiplying them. In order to represent a transformation as a matrix multiplication, a vertex is required to be stored as a four-element, rather than a three-element, vector [FV90]. Then vertex v_i is represented as

$$v_i = \begin{bmatrix} x_i, & y_i, & z_i, & 1 \end{bmatrix}.$$

The following matrix translates the vertex v_i a distance D_x along the $x-$axis, D_y along the $y-$axis, and D_z along the $z-$axis:

$$T(v_i) = \begin{bmatrix} x_i, & y_i, & z_i, & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ D_x & D_y & D_z & 1 \end{bmatrix} = \begin{bmatrix} x_i + D_x, & y_i + D_y, & z_i + D_z, & 1 \end{bmatrix}.$$

The following matrix scales the vertex v_i (as an endpoint) by S_x along the $x-$axis, S_y along the $y-$axis, and S_z along the $z-$axis:

$$S(v_i) = \begin{bmatrix} x_i, & y_i, & z_i, & 1 \end{bmatrix} * \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x_i * S_x, & y_i * S_y, & z_i * S_z, & 1 \end{bmatrix}.$$

The following matrix rotates the vertex v_i by the rotation angle θ about the $z-$axis:

$$\begin{bmatrix} x_i, & y_i, & z_i, & 1 \end{bmatrix} * \begin{bmatrix} cos\ \theta & sin\ \theta & 0 & 0 \\ -sin\ \theta & cos\ \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} x_i * cos\ \theta - y_i * sin\ \theta, & x_i * sin\ \theta + y_i * cos\ \theta, & z_i, & 1 \end{bmatrix}.$$

With suitable rotation matrices, one could achieve rotation of points (objects) around $x-$axis, $y-$axis or in general around a given line by a given angle. The

reader is referred to [FV90] for further details.

1.3.4 Role of User Interaction

As stated before, the problem with the current 3D graphics systems, like HOOPS and Dore, is that they don't interact with the user at his application level. The result is that applications must work at the renderers' level rather than at a convenient abstract level. Grams [EK92] is an object-oriented 3D graphics system that raises the abstraction level at which the application programmer and the end user interact with the system. The objective of our work is to design a modeling system that raises the level of abstraction even further to the application level. As a consequence, the user can work directly with the entities (e.g. the building blocks) of his application without caring about the low level representation.

Another main issue that raises the user's level of interaction is interactive constraint handling as it releases the user from worrying about satisfying the constraints manually. Most constraint specification techniques are text-oriented [Art89] and are not part of modeling systems as fundamental mechanisms. In the traditional constraint languages, constraints are specified in predicate calculus, mathematical or procedural format using text-based syntax. In our work, we consider text-based constraint specification only, though constraints can be specified graphically (i.e.,

using pictures or symbols).

Pizano [Art89] developed a language that uses pictures to specify constraints. The constraint pictures are translated into first-order predicate calculus describing the constraints specified. Gambit [Bag85] is an interactive tool that has support to define constraints interactively on a graphical representation of the database schema. Also, QBE [Zlo78] can, to some extent, be used to define integrity constraints interactively where a constraint definition involves completing blank tables. But, QBE allows only sum-of-products form of specifications based on the attributes of a relation, a form of propositional calculus.

1.4 Drawbacks of the Current Systems

The above discussion shows that the current 3D graphics systems suffer from one or more of the following drawbacks:

1. They are too low level for application users.
2. They are not user-interactive (excluding application-specific systems like CAD systems).
3. They do not handle constraints interactively.

1.5 Problem Statement

From the previous discussion it is evident that a good interactive modeling system should permit hierarchical development (refinement of the model of the system under consideration). It should be natural to spell out the domain-specific constraints, so that subsequent explorations will be restricted to the confines of such constraints, which is a highly desirable situation for a user (the designer). To promote large scale designs there should be adequate support for model reusability and the convenience of object orientation. In this thesis we propose the design and implementation of an object-oriented hierarchical modeling system and its customization to 3D graphics.

1.6 Outline of the Thesis

In this Chapter we introduced the problem, the motivation behind it and the necessary background. Some of the related tools and systems were reviewed and their limitations were addressed. In Chapter 2, we extract the requirements of a modeling system through simulated walk-throughs. Based on these requirements, Chapter 3 presents a design of an object-oriented hierarchical modeling system. Chapter 4 presents a customization of this design to 3D graphics modeling. In Chapter 5, we explain the implementation issues of the developed 3D graphics hierarchical mod-

eling system prototype. Finally, the Conclusion Chapter presents the achievements and possible future extensions.

Chapter 2

Modeling

Modeling is the process of transforming a real world system into an abstract model retaining the essence and side stepping irrelevant details. A modeling system allows its user (designer in some application) to build the required model of a real world system. Hence, it should have the capabilities to capture the relevant features of the real-world system.

The modeling system we propose in our work is object-oriented. In object-oriented modeling, a real world system is modeled as a set of objects and relationships between them. The modeling process starts with what is called object-oriented analysis where the modeler describes the significant objects in the (real world) system domain and the relationships among them [Flo91].

Objects are defined by identifying the distinctive entities in the application domain and their properties which are referred to as *attributes.* The relationships among the identified objects spell out specific system's requirements which we refer to as *constraints.*

This Chapter is mainly devoted for extracting and studying the required properties of attributes and constraints in hierarchical modeling, developing the mechanisms that need to be supported by any good modeling system. It concludes by customizing the developed mechanisms to 3D graphics modeling.

2.1 Objects

Objects are the abstract building blocks for modeling the overall properties of a real world system. Objects combine their state (represented by their attributes) and behavior (the operations to compute or alter the attributes) into abstract packages. When working on the attributes, the operations have to observe the constraints set on the involved attributes.

Objects can be simple or made of other objects which we refer to as aggregate objects. Existing objects can be assembled in a multitude of configuration to compose new aggregate objects.

2.2 Attributes

An attribute is a property of an object that can have values belonging to a specified domain. A graphical component, for example, can have a length attribute which is usually defined over the domain of real numbers. Aggregate objects can have attributes defined in terms of the attributes of the constituent objects.

Attributes are user-specified either as values from a domain or as computation formulae in terms of the attributes of other components. The name of an object can be a good example of a user-specified attribute for both primitive and aggregate objects where the user sets a name to an object independent of any other attributes of an object. Computed attributes can be based on the attributes of the lower level constituent objects (synthesized attributes), the higher level objects (inherited attributes) or both kinds of objects (hybrid attributes).

2.2.1 Synthesized Attributes

When an object is formed as an aggregate of lower level objects, new attributes can be defined for the formed object based on the attributes of the constituent objects. These attributes are referred to as *synthesized attributes* since the attributes of the constituent objects are synthesized to compute the attribute of the aggregate object.

As an example, let us consider constructive solid geometry where solid components are put together to form new solid components. There are different ways of composing components, like taking the union of the building components, or performing a subtraction of an intersection operation on them.

In some applications it is important to compute the volume of these solid components. If volume is an attribute for the building components, this attribute can be computed for the aggregate component. The computation of the volume attribute for an aggregate component clearly depends on the volume of the building components and the composition operation performed on them. So, if the operation was union, for example, the aggregate volume would be the sum of the volumes of the building components minus the volume of the intersection object. In other words when the constituent objects are pairwise disjoint, the aggregate component is expressed as the summation of the volume attributes for the constituent components.

2.2.2 Inherited Attributes

In contrast to the synthesized attributes of aggregate objects, the constituent lower level objects can have attributes whose values depend on the attributes of the aggregates they are part of. These attributes are referred to as *inherited attributes* since the attributes of the aggregate objects are inherited by the constituent objects.

For example, if a set of graphical objects are put together to form a new graphical object, then changing the color or any other graphical feature of the aggregate object would be reflected by changing the color of the constituent objects. So, the color attribute can depend on, or is inherited from, the color attribute of the aggregate object.

An interesting application for inherited attributes is hierarchical graphics modeling where basic building blocks can be put together to form aggregate components. Performing a transformation on an aggregate component, should transform the component as block independent of its composition whether it is a basic or an aggregate component. This means that the transformation done on the aggregate component should be inherited by its building components.

To achieve this requirement, the transformations done on a component are represented by a transformation matrix, while the overall resultant transformation (transformations done on the component and the inherited transformations) is represented by a composite transformation matrix.

The transformation matrix is a user-specified attribute which stores the transformations that the user performs on the component directly, like scaling the component or moving it (in its parent's coordinate system). On the other hand, the composite transformation matrix is a computed attribute obtained by computing the product

of the transformation matrix of the component with those of its ancestors in the building hierarchy.

2.2.3 Hybrid Attributes

The last type of computed attributes is a mix of synthesized and inherited attributes. Attributes of this type depend on the attributes of both the higher and lower level objects.

Let us reconsider hierarchical graphics modeling to study the bounding box attribute which illustrates this type of attribute. The bounding box of a component is the smallest box that can fit the component. In the case of the 3D graphics, it is the smallest box that can bound the component, whereas it is a rectangle in 2D graphics.

When a transformation is performed on a component, its constituent components inherit the transformation. Consequently, the inherited transformation changes the bounding boxes of the constituent components as a result of the transformation done on them. The transformation keeps propagating all the way to the lowest level transforming the components and changing their bounding boxes. So, the computation of the bounding box attribute of a component depends on the transformation matrix attribute of its ancestors.

Also, when a transformation is performed on a component, the transformation can change the component size or position within its aggregate component affecting the bounding box of the aggregate. Consequently, the bounding box of an aggregate component changes as a result of a transformation on one of its constituent components. This change propagates to the next higher level changing the bounding box of the ancestor, and the process continues all the way to the highest level. So, the computation of the bounding box attribute of a component depends on the transformation matrix attribute of its descendants.

2.3 Constraints and Their Role in Modeling

Constraints are restrictions on the sets of values attributes can take. Constraints may reflect the physical limitations or logical relationships among objects of the real world situation. For example, restricting a component to move in the horizontal direction is a constraint on its transformation matrix attribute. This constraint defines a valid set of transformation matrices for the transformation matrix attribute. In this case the attribute is user-specified as discussed before.

When an attribute is defined based on other objects' attributes, its domain becomes the product space of the domains of these attributes. Constraints restrict the set of values an attribute can have by defining a subset of valid values from the

product space. For example, restriction on the bounding box of a component to be less than a specific size is a constraint on the computed attribute bounding box.

When the user starts modeling a real world system, he would have system requirements which should be met by his model. At this stage, the user usually doesn't have a clear idea on how to meet these system requirements. Constraint specification and enforcement can help the user in specifying his requirements.

Without adequate support for constraint handling, the user will have to satisfy required relationships manually, which is an expensive and tedious process especially when the required relationship involves attributes of more than one object. In this case the user has to fix the attributes for each of the objects involved in the relationship separately and iteratively until the requirement is satisfied with final attribute settings. Besides, setting the attributes to satisfy a particular requirement can violate another already set requirement and thus the user has always to work with the whole system rather than concentrating on just part of it.

On the other hand, support for constraint specification and enforcement as part of a modeling and design tool can facilitate meeting the system requirements. The required relationships are specified as constraints and satisfied by enforcing these constraints. Finding valid attribute settings that satisfy the constraints is performed by the enforcement system automatically with minimal user intervention. This

allows the user to work on a requirement in a local abstraction context without having to worry about the resulting non-local consequences.

For example, setting an upper bound on the volume of component, say C_3, composed of components C_1 and C_2, can be a design requirement. To satisfy this requirement manually, the user has to experiment with various combinations of sizes of its constituent components C_1 and C_2 until the requirement is achieved. Experimenting with the sizes of C_1 and C_2 is not only a time consuming process, but can also violate already existing requirements like bounds on the sizes of C_1 and C_2 and so on. Automated support for constraint handling simplifies component design to the user, as constraint enforcement finds valid sizes for C_1 and C_2 that satisfy the upper bound requirement on C_3 without conflicting with existing constraints.

If C_1 and C_2 were themselves composed of other components, then finding valid sizes for C_1 and C_2 that satisfy the upper bound requirement on C_3 may require the adjustment of their constituent component sizes, and so on until we reach to the bottom of the hierarchy.

2.4 Constraint Specification

As stated earlier, a constraint is a form that expresses a desired relationship among one or more objects. In this generic definition, the objects are determined by the scope of the application defining the constraints. The constraint's form depends on the way the constraint is specified - the constraint-specification technique. There are three main constraint-specification techniques: mathematical, using predicate logic and procedural.

To satisfy a constraint, a constraint-satisfaction technique is required to find the values of parametric attributes that will make the relationships true. Constraint enforcement is the process of finding valid settings that satisfy a constraint. Enforcing a constraint on an attribute which is dependent on attributes of other objects in a particular hierarchy, requires propagating the task of finding valid attribute settings throughout the hierarchy in an attempt to satisfy the constraint.

The user of an application can define required constraints explicitly using the above specification forms or implicitly using a more user friendly method i.e. graphically. As an example, the use of a graphical editor can set the alignment unit to a particular value x. For the graphical editor this is translated to a mathematical constraint which states that any object movements should be an integral n of units x.

$ObjectMovement = n \times x$ where n is an integer

2.4.1 Mathematical Specification

In the mathematical form, constraints are specified as a system of mathematical equations. For example, in a CAD system one type of constraints can be to have object O_1 centered at (x_1, y_1) and object O_2 centered at (x_2, y_2) always on the same horizontal line. As one object moves, O_1 for instance, the other object O_2 should follow it so as to be always on the same horizontal line. In the mathematical form this is translated as follows:

$$y_2 = y_1$$

Constrains in the mathematical form can be modelled as graphs. For example, consider the constraint which represents the conversion between inches and centimeters:

$$IN \times 2.5 = CM$$

This constraint is modelled by the graph in Figure 2.1 [Lel88] where IN is the quantity in inches and CM is the quantity in centimeters. The square nodes represent

variables and the round nodes represent operators. The arguments to an operator are attached on one side and the result is attached on the other side. Constants are considered as operators with no arguments.

Several constraint-satisfaction techniques exist for mathematically specified constraints. This Section presents three of these techniques: local propagation, relaxation and equation solving. In the local propagation technique [SS79] known values are propagated along the arcs. When a node receives sufficient information from its arcs, it uses this information to compute the unknown variables. The newly computed values then propagate along the arcs, causing new nodes to fire and so on. So the constraint graph in Figure 2.1 represents the following rules:

$CM = 2.5 \times IN$	when IN is given
$IN = CM/2.5$	when CM is given

The major disadvantage of the local propagation technique is its inability to satisfy constraints that have cycles in their graphs as the following constraint which is modelled by the graph of Figure 2.2:

$$x + y = z$$

$$x - y = z$$

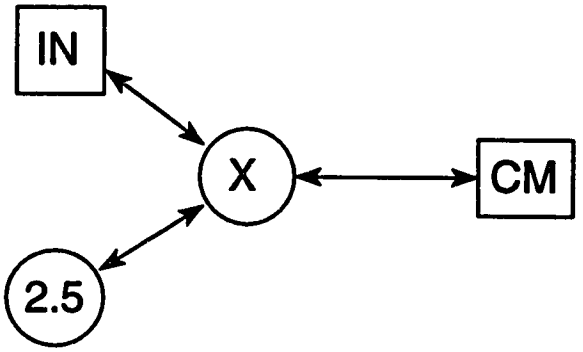

Figure 2.1: Graphical representation of the conversion constraint

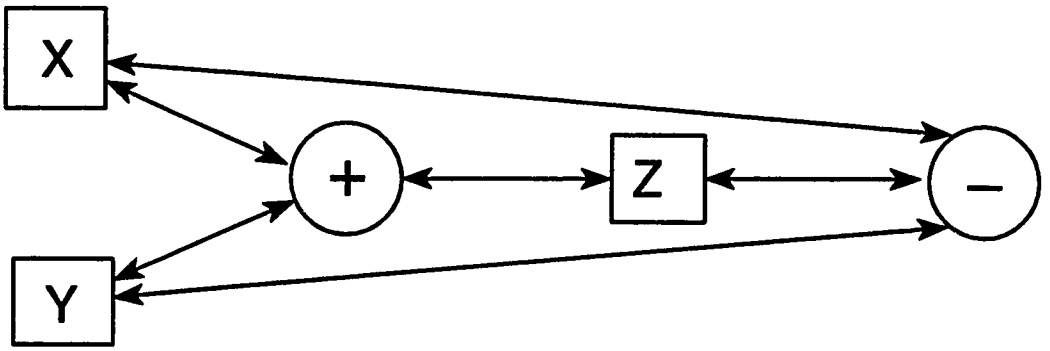

Figure 2.2: Constraint graph containing a cycle

If z is 1, this value can't propagate through either the plus node or the minus node to solve for either x or y because there is a cyclic dependency between x and y. So local propagation doesn't find a solution for this example, though there is one.

One classic method that is used to solve the above constraint is called relaxation [Sut63] which is an iterative numerical-approximation technique in which an initial guess at the unknown values is made, and then the errors that are caused by this guess are calculated. New guesses are then made based on these errors, and the process continues until the error is minimized to an acceptable level.

As an example, to solve the above constraint using this technique, let us guess x to be 3. Substituting this value in the first equation sets y to -2. Substituting the calculated value of y in the second equation estimates x to -1. A new guess for x is made by taking the average of its last two values $(-1+3)/2$ which turns out to be the solution.

The major disadvantage of the relaxation method is that it can be used with continuous numerical values and not on integer-valued objects. Practically, the relaxation method is used as a constraint-satisfaction technique when the local propagation fails, because it can be very slow in finding a solution with an acceptable level of error.

General equation solving techniques like the one used in symbolic algebraic sys-

tems could also be used for constraint-satisfaction. Gaussian elimination can be used to solve systems of linear simultaneous equations. These techniques like the relaxation technique are slow and are to be used after the local propagation has failed.

One desirable feature of constraint-satisfaction techniques is the ability to separate the constraint rules from the control mechanism. This allows the addition of new rules to the set of constraints without having to modify the control mechanism. Of the constraint-satisfaction techniques introduced above only the local propagation technique has this feature. Finally, none of these techniques are general-purpose, in the sense that they can be used to compute arbitrary computable functions.

2.4.2 Predicate Logic

Constraints can also be specified using predicate logic. A constraint is specified by a set of predicates each of which expresses one of the constraint's relationships. Let us reconsider the horizontal alignment example again, where $C_1(x_1, y_1)$ and $C_2(x_2, y_2)$ should always be on the same horizontal line, and try to express this constraint using predicate logic. This constraint would be specified as follows:

$Location(C, X, Y);$

$HrAlign(C_1, C_2)$

IF

$C_1 <> C_2$

$\&\ Location(C_1, x_1, y_1)$

$\&\ Location(C_2, x_2, y_2)$

$\&\ y_2 = y_1;$

In the above example Location and *HrAlign* are predicates. Each predicate expresses a relationship among one or more objects. The Location predicate expresses that the component is located at the (X, Y) position. The *HrAlign* predicate states that component C_1 and C_2 are on the same horizontal line, i.e., when C_1 is not equal C_2, if C_1 is located at (x_1, y_1), and C_2 is located at (x_2, y_2), it must be the case that $y_1 = y_2$. As can be seen from this example, predicates express the desired relationship using mathematical logic expressions. In the notation used above the equal sign (=) is used for equality and not for assignment.

Predicates express the relationship between objects which could be variables (having unknown values) or objects that are constants (have bound values). Conventionally, variables start with an upper case character and constants start with lower case (if not numeric). The following predicate states that the component "sphere1" is at (1,5):

$Location(\text{"}sphere1\text{"}, 1, 5)$

If the component "sphere1" was moved to (1,5) and the component "sphere2" is to be on the same horizontal line as "sphere1", then to satisfy the *HrAlign* constraint we need to satisfy the following goal:

$HrAlign(\text{"}sphere1\text{"}, \text{"}sphere2\text{"})?$

The process of trying to satisfy this goal, or inferring it, uses the inference capability of the system that implements a particular predicate logic language variant. For the above goal to be satisfied, this process uses the predicate $HrAlign(C_1, C_2)$ and binds C_1 to "sphere1" and C_2 to "sphere2". For $HrAlign(\text{"}sphere1\text{"}, \text{"}sphere2\text{"})$ to be true, all of its premises should be true which results in binding y_2 to 5. The inference process, however might not find a solution in which case there would be no solution satisfying the constraint.

An important advantage of specifying constraints using predicate logic is the convenience of ignoring the control mechanism (the inference process) which makes the programs expressing the constraints clearer and more declarative. This feature is clearly missing in the traditional procedural languages. Generally, the procedural technique is used for constraint specification when it is difficult to express the required constraints in any other form.

2.5 Construction and Constraints

Earlier in this Chapter, we introduced the concept of aggregation in a bottom-up fashion. Bottom-up construction is a hierarchical assembly scheme in which objects are put-together to assemble higher level objects. The newly assembled objects can themselves be used to build higher level objects and so on. Objects at the lowest level are primitives.

In contrast to the bottom-up construction scheme, top-down construction is a hierarchical decomposition scheme in which an object is decomposed in to lower level objects that can form it. Further, the decomposed objects in their turn can be decomposed into lower level objects that can form them. Objects at the lowest level, as in the bottom-up scheme, should be primitives which are assumed to be available.

In both schemes, objects can have attributes that are user-specified or computed based on the attributes of the higher and/or lower level objects, and constraints can be defined on these attributes. The difference between the two schemes is the order in which attributes and constraints are defined.

In the bottom-up scheme, attributes and constraints are defined starting at the lowest level objects up to the highest level objects. Whereas in the top-down scheme,

attributes and constraints are defined in exactly the opposite order starting from the highest level down to the lowest level. This could make a major difference in satisfying constraints. In the bottom-up scheme an aggregate object can have a constraint on a synthesized attribute that can't be satisfied as a result of the values of the lower level attributes.

For example, consider setting a constraint on the size of aggregate components. The bottom-up scheme can result in constructing components of invalid sizes because of the sizes of their constituent components. On the other hand, constructing components in a top-down scheme sets the size constraint on the aggregate before its lower level constituent components. Effectively, the aggregate component is spatially partitioned into partitions each of which corresponds to the dimensions of the lower level components. The lower level components in their turn are decomposed and spatially partitioned into even lower level components and so on.

Similarly, the top-down scheme can result into composing an object into objects that can't satisfy the desired constraints on some of their attributes. In the case of the size attribute, this applies when primitive components have constraints on their sizes instead of the aggregate components. An aggregate component, for example, can be decomposed into primitives of invalid sizes (don't satisfy the size constraint).

Combining the two schemes together and using the appropriate one as needed

would be the best strategy facilitating the design process for the user. An object can be constructed by decomposing (top-down) it into lower level objects. Each of the constituent lower level objects can be constructed by assembling it from lower level objects (bottom-up) or by decomposing it into lower level objects (top-down) and so on. Alternatively, objects constructed using either of the two schemes can be assembled together (bottom-up) to build a new high level object.

2.6 3D Graphics Construction Model

The above described model is a general model for construction. It has the ability to construct objects in bottom-up, top-down, or middle-out schemes according to user's convenience and taste, and provides powerful mechanisms for defining attributes and constraints. This Section applies this model to 3D Graphics construction. It describes in detail each of the construction schemes separately. While the general construction model has used the term object to refer to a construction entity, the 3D Graphics construction model uses the term component.

2.6.1 Bottom-up Construction

The bottom-up component construction process starts assembling a new component from the already existing components. So, if there has not been any created aggregate components, only primitive components would exist. The process proceeds as with the following steps which are illustrated in Figures 2.3 and 2.4:

1. The user starts the process by picking up each of the components that are needed to construct the desired aggregate component and places it in the world space. Each of these components has default values for its own attributes.

2. The user can change the default values as needed. For example, as a component is placed in the world space, the transformation matrix of this component is changed to reflect its current position. Recall that the transformation matrix is a user-specified attribute of the component. Color, shading and line thickness can be considered as other attributes.

3. The user asks the modeling system to assemble the selected components into an aggregate component. The system responds by defining a new component which has only a local coordinate system and redefines the transformations on the selected components to be with respect to this coordinate system. In other words the selected components' local coordinate systems become defined in the local coordinate system of the aggregate component instead of the

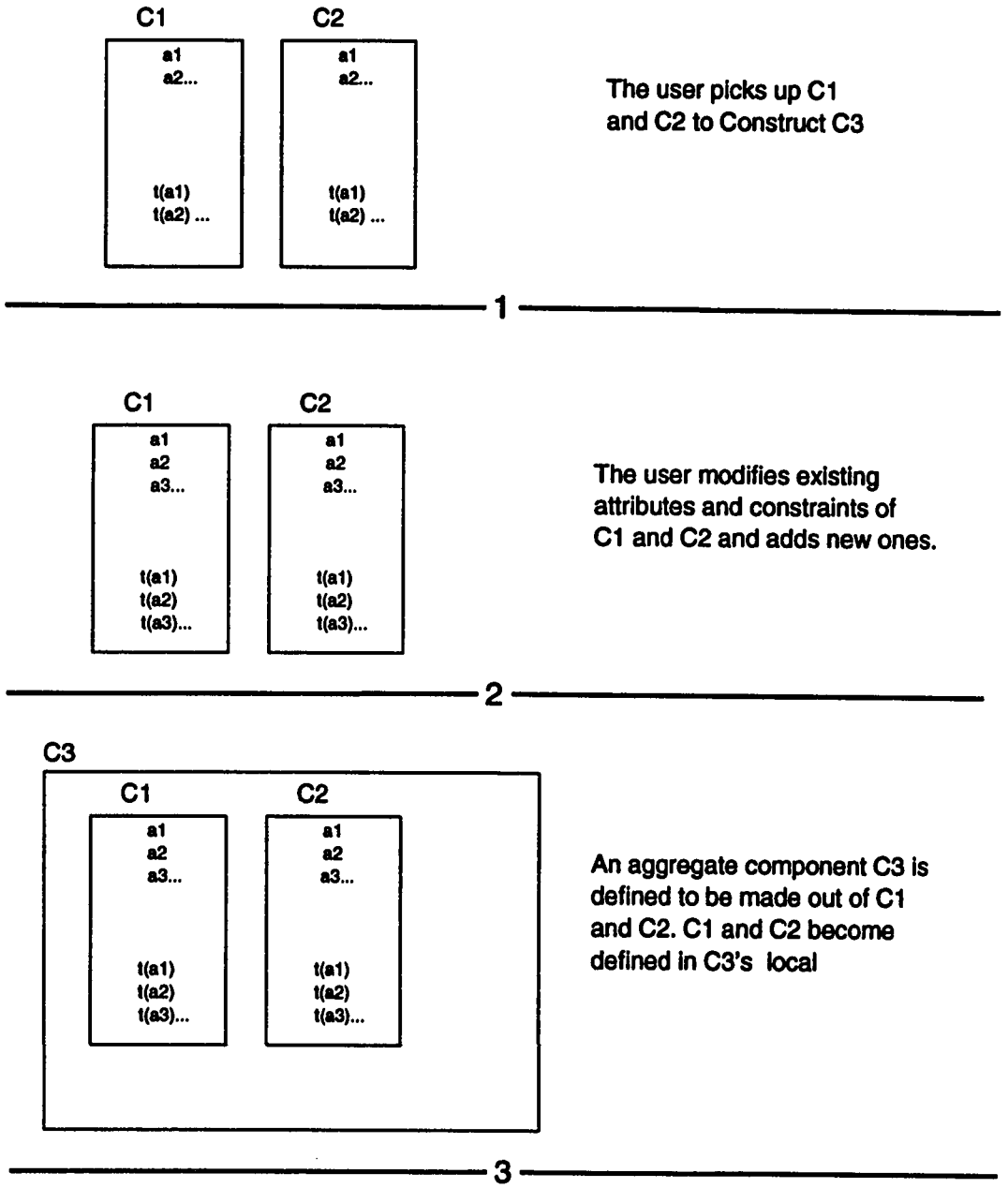

Figure 2.3: An example illustrating bottom-up construction in the 3D-graphics model

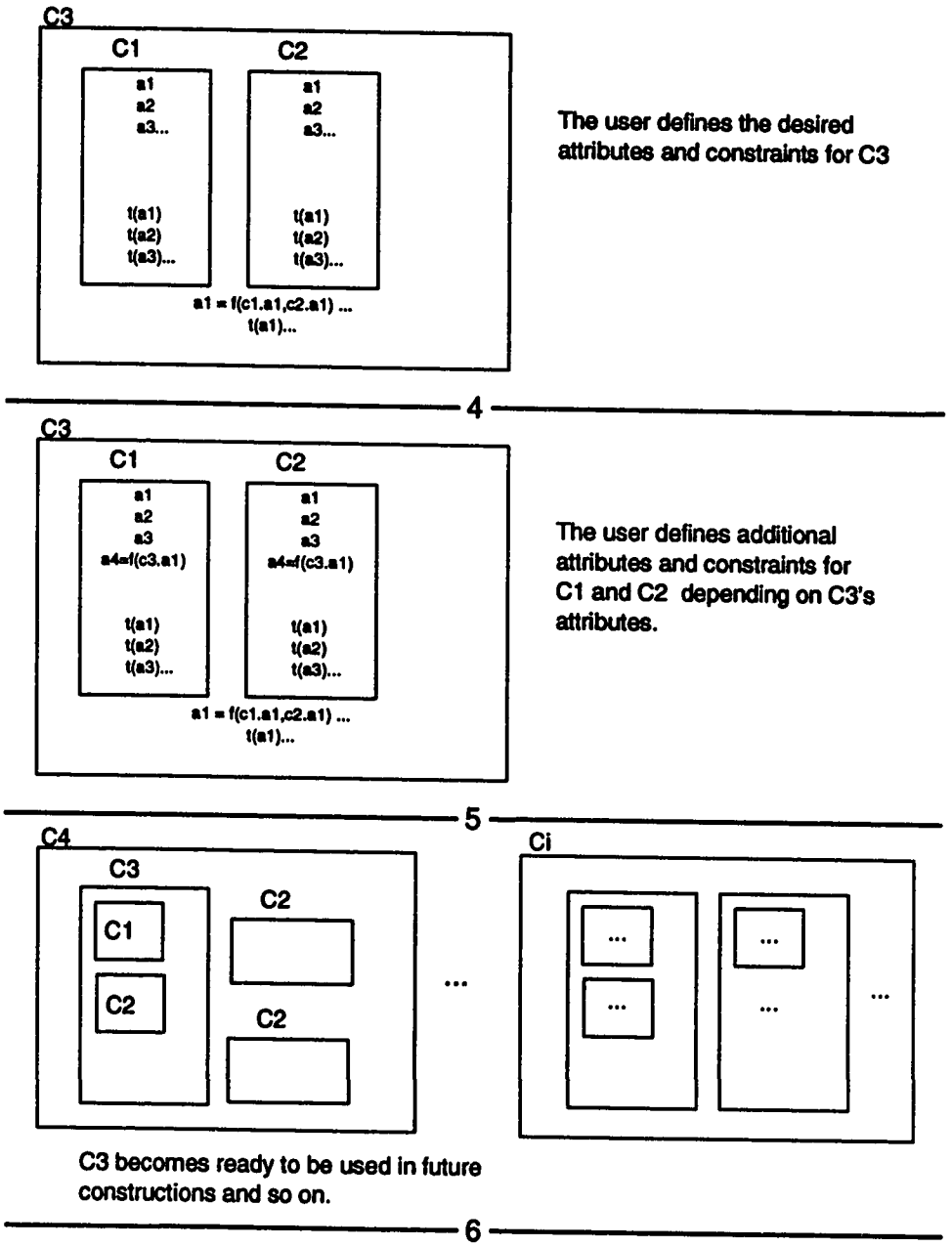

Figure 2.4: An example illustrating bottom-up construction in the 3D-graphics model (continuation)

world coordinate system. While, the local coordinate system of the aggregate component itself is defined in the world coordinate system.

4. The user defines the desired attributes and constraints for the newly created component.

5. The user defines additional attributes and constraints for the constituent components. These attributes typically depend on the attributes defined on the last step.

6. The component becomes ready to be added to the existing components and be used in construction, and so on.

2.6.2 Top-down Construction

The top-down component construction process starts by defining the top level component. The following steps which are illustrated in Figures 2.5 and 2.6 describe the process:

1. The user starts the process by asking the modeling system to create a new component. The system responds by inserting a new (top-level) component in the world space which has nothing but a local coordinate system. Let us refer to a component of this type which is merely a work space as an empty

component.

2. The user defines the desired attributes and constraints for the top-level component. An example of such an attribute is the bounding box attribute as discussed earlier. By defining the bounding box attribute, the user can attempt power to construct components of bounded sizes by setting a constraint on the size of the bounding box.

3. The user starts decomposing the top-level component by inserting new empty components in its space.

4. The user defines the attributes and constraints for each of the constituent components. For example, as a component is placed in the top-level component space, its transformation matrix is changed to reflect its position. Also, by defining the bounding box attribute for the constituent components, the top-level component is spatially partitioned into partitions each of which corresponds to the dimensions of the constituent components. The attributes for the constituent components are defined in such a way as to integrate the constituent components and form the required specifications of the top-level component out of them.

5. The user defines additional attributes and constraints for the top-level component. These attributes typically depend on the attributes defined for the components in the previous step.

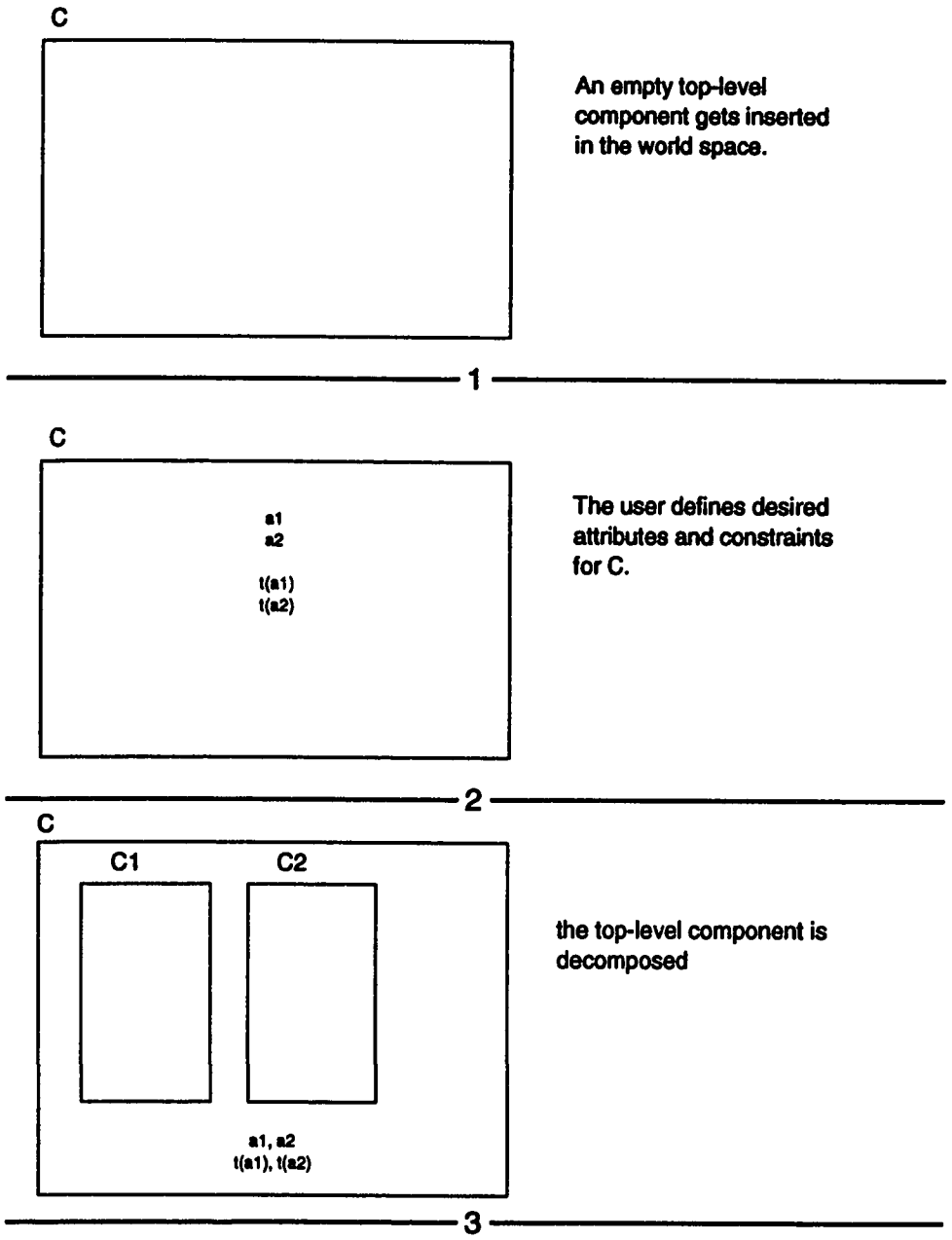

Figure 2.5: An example illustrating top-down construction in the 3D-graphics model

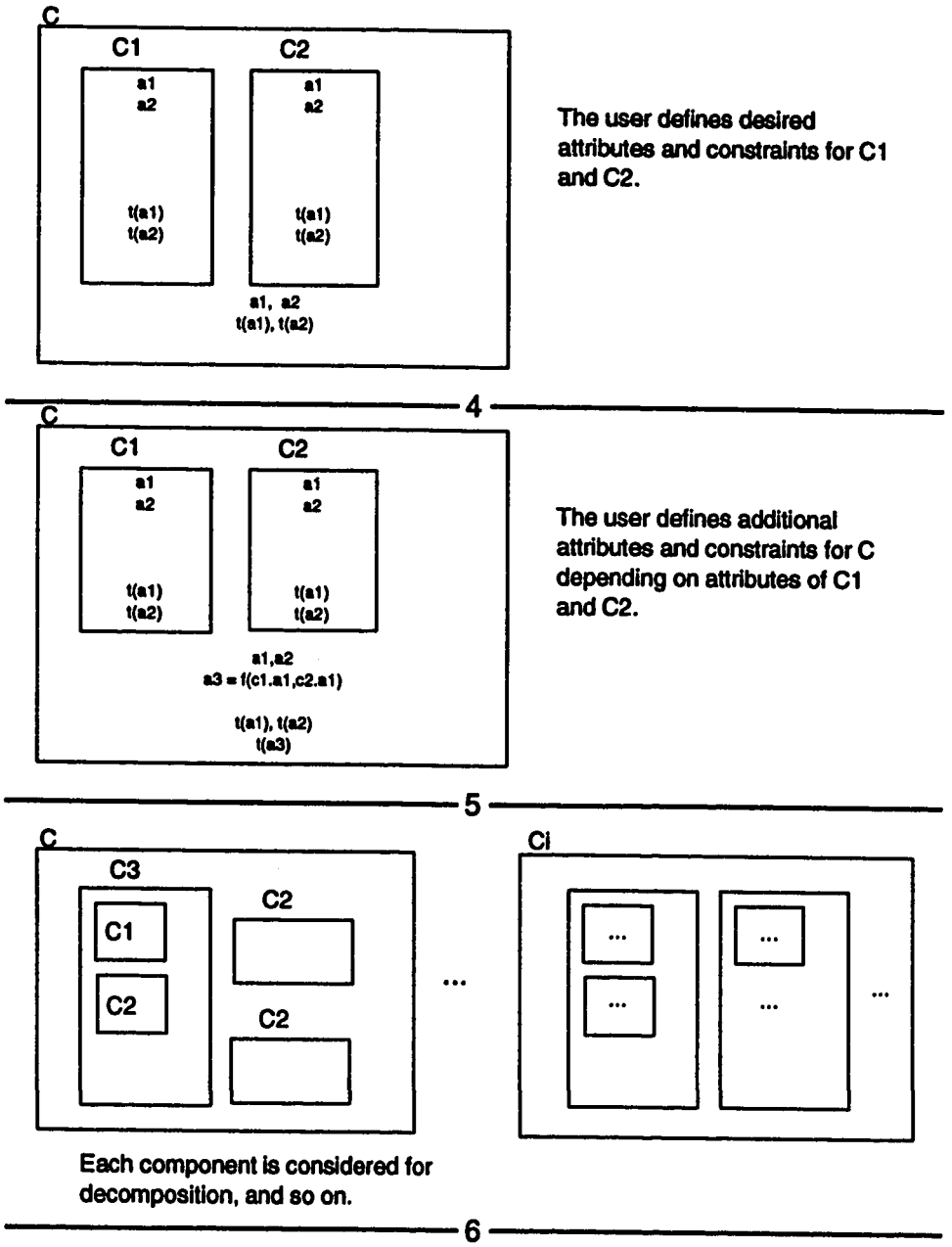

Figure 2.6: An example illustrating top-down construction in the 3D-graphics model (continuation)

6. Each of the constituent components can be either a primitive component or it is considered for further decomposition. For the components to be decomposed, the user repeats the last three steps where the considered component takes the role of the top-level component.

2.7 Summary

In this Chapter we presented a study of the issues that show up in typical modeling scenarios. General mechanisms that could be part of a modeling system were developed. Finally, the developed mechanisms were customized to support 3D graphics modeling systems.

Chapter 3

System Architecture

This Chapter presents a design of an object-oriented hierarchical modeling system based on the mechanisms developed in the previous Chapter. The system design is also object-oriented and decomposed into a collection of service providing managers (Figure 3.1).

The *Types Manager* is an object that provides the functionality of defining types for attributes. The *Attributes Manager* object provides the functionality of defining attributes for objects. The *Constraints Manager* object provides the functionality of defining constraints on attributes. Finally, the *Objects Manager* is an object that provides the functionality of defining user objects.

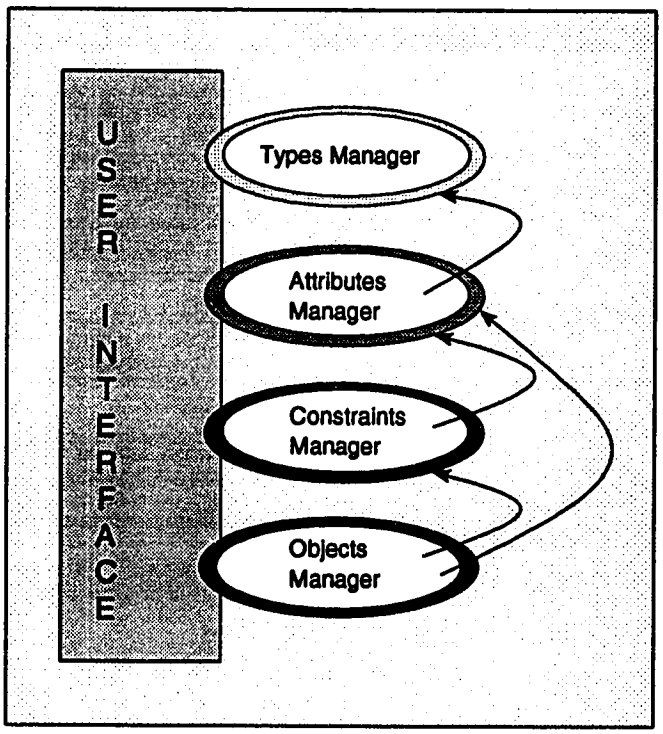

Figure 3.1: Service providing managers

The Objects Manager represents each user object by a system object. To distinguish between system objects and user objects, the term object will be used in this Chapter to refer to user objects, unless stated otherwise. The manager objects are referred to simply as managers.

Each of the managers provides its services through methods which define an interface to it. It stores its entities in a database and manages access to these entities, so that the other modules do not access its database directly but rather use the interface. In the diagram of Figure 3.1, the shadow of a manager represents its interface. The diagram shows the message flows among the managers and shows that the user interface provides access to the manager services through their interfaces as well. Finally, the entities are identified by IDs which refer to their locations in the database.

The rest of this Chapter explains the functionality of the managers in detail and presents their specifications. It concludes by giving a detailed example illustrating the usage of the managers.

3.1 Managers

3.1.1 Types Manager

The Types Manager provides the facility to the user to define types which are required to specify the domains of attributes. New types can be defined as in the Pascal language. The user is provided with the already defined types as a basis to define new types. Initially the defined types include only the primitive types (integer, floating-point, string and boolean).

Two techniques are provided to define new types. The user can define new types as subsets of existing types like defining the type hour to be a subset of the type integer (0..23). The other technique is to define new types as aggregates of existing types. Types can be aggregated, as in Pascal, as arrays or records of existing types. The type time, for example, is a record of three types: the type hours (a subset of integer 0..23), the type minutes (a subset of integer 0..59) and the type seconds (a subset of integer 0..59). The following methods are provided by the Types Manager:

TypeCreate: Creates a new user-defined type as discussed above. The new type gets stored in the types database.

TypeSearch: Given a type, this method searches for its existence in the types

database and returns its definition.

TypeDelete: Deletes a given type from the types database.

TypeStore: Stores a given type entry in the database.

TypeLoad: Loads a given type entry from the database into memory.

3.1.2 Attributes Manager

The Attributes Manager provides the facility to the user to define new attributes for objects. The attributes are stored in a database. Each entry in the database has three fields: attribute name, attribute domain (a type), and attribute value. The attribute value stores the user input if it is a user-specified attribute, or stores the attribute computation formula if it is a computed attribute. The following methods are provided by the Attributes Manager:

AttributeCreate: Allows the user to define a new attribute for an object and stores it in the attributes database.

AttributeDuplicate: Duplicates a given attribute by creating a new attribute entry and copying the given entry's contents into it.

AttributeSearch: Given an attribute, this method searches for its existence in the attributes database and returns its definition.

AttributeDelete: Deletes a given attribute from the attributes database.

AttributeModify: Modifies the value of a given attribute whether it is user-specified or computed.

AttributeEvaluate: Returns the value of the attribute. If the attribute is a computed attribute, its formula is evaluated and the resulting value is returned. An attribute gets evaluated whenever it is accessed. So, the evaluation of a computed attribute triggers the evaluation of the attributes referenced in its definition.

AttributeStore: Stores a given attribute entry in the database.

AttributeLoad: Loads a given attribute from the database into memory.

3.1.3 Constraints Manager

The Constraints Manager provides the facility to the user to define constraints on attributes. A constraint is either a primitive constraint, or a logical expression of existing constraints. While retaining the generality of constrain handling, in order to simplify the implementation, the scope of a constraint is restricted to one attribute only. Therefore, a constraint is defined on a single attribute. If there is a need to define a constraint on multiple attributes, a new computed attribute is defined

in terms of these attributes and the constraint is defined on the new attribute. Also, constraints can be combined using logical expressions to form new constraints. Constraints are stored in a database managed by the Constraints Manager. The following methods are provided by the Constraints Manager:

ConstraintCreate: Allows the user to define a new constraint and stores it in the constraints database.

ConstraintDuplicate: Duplicates a given constraint by creating a new constraint entry and copying the given entry's contents into it.

ConstraintSearch: Given a constraint, this method searches for its existence in the constraints database and returns its definition.

ConstraintDelete: Deletes a given constraint from the constraints database.

ConstraintModify: Modifies the definition of a given constraint.

ConstraintVerify: Verifies if in the current state of the design, a given constraint is satisfied or not. Uses the **AttributeEvaluate** method to evaluate the attribute on which the constraint is defined.

ConstraintEnforce: Forces the satisfaction of a constraint. Uses the **Attribute-Modify** method to modify the value of the attribute on which the constraint is to be enforced.

VerifyAll: Verifies if in the current state of the design, all the constraints of an object are satisfied simultaneously.

ConstraintStore: Stores a given constraint entry in the database.

ConstraintLoad: Loads a given constraint from the database into memory.

3.1.4 Objects Manager

The Objects Manager provides the facility to the user to define objects. It represents each user object by a system object, so each system object corresponds to a user object. The system objects created by the Objects Manager belong to a class that we refer to as the *Object* class. The Objects Manager is, in fact, the class object (the object that creates objects belonging to the class) of *the Object* class.

Each system object stores the following entities for the user object it represents: a list of attributes, a list of constraints, a list of constituent objects, and the parent object, if it exists. Constituent objects are also objects, and thus they are represented in exactly the same way. Primitive objects have empty lists of constituent objects. System objects have the following instance variables:

Attributes: A list of IDs of the object's attributes.

Constraints: A list of IDs of the object's constraints.

Children: A list of IDs of the object's constituent objects.

Parent: The ID of the object's parent.

System objects have also the following methods:

AddAttribute: Adds a given attribute ID to the object's list of attributes.

DeleteAttribute: Given an attribute ID, it deletes it from the object's list of attributes.

AddConstraint: Adds a given constraint ID to the object's list of constraints.

DeleteConstraint: Given a constraint ID, it deletes it from the object's list of constraints.

AddChild: Adds a given object ID to the list of constituent objects. The parent of the added object is set to point to the calling object.

DeleteChild: Given an object ID, it deletes it from the list of constituent objects.

The Objects Manager provides two methods to create new system objects along with other methods:

ObjectCreate: Defines a new system object and stores it in the objects database.

ObjectDuplicate: Duplicates a given system object by duplicating the given object's contents (attributes, constraints and constituent objects) and creating a new object to store the IDs of the duplicates. The source attributes, constraints are duplicated by calling the **AttributeDuplicate** and **ConstraintDuplicate** methods respectively. The process is applied recursively for the constituent objects.

ObjectDelete: Deletes a given system object from the objects database.

ObjectStore: Stores a given system object in the database. The object is stored by storing its attributes, constraints, parent ID, and constituent objects. Attributes and Constraints are stored by calling the **AttributeStore** and **ConstraintStore** methods respectively. The process is applied recursively for the constituent objects.

ObjectLoad: Loads a given system object from the database into memory. The object is loaded by loading its attributes, constraints, parent ID and constituent objects. Attributes and Constraints are loaded by calling the **AttributeLoad** and **ConstraintLoad** methods respectively. The process is applied recursively for the constituent objects.

3.2 Example

This Section illustrates the functionality of the managers through an example. The example illustrates building an aggregate object in several steps. Each step illustrates both the user view (the action performed and the results seen by the user) and the system view (the steps the system has to perform to implement the user's action). Though objects, attributes and constraints are identified by IDs, in this example we will refer to them by symbolic references which stand for the IDs.

In this example the user builds up a new object Object3 from two primitive objects O1 and O2 in a bottom-up fashion. Each of O1 and O2 has a single attribute, specifically an attribute which stores the object's name. Table 3.1 shows the attributes database with O1 and O2 attributes. The constraints database doesn't have any relevant entries since there are no constraints defined on the attributes of O1 and O2.

Table 3.1: O1 and O2 attributes at the beginning of the example

ID/Symbol	Attr. Name	Attr. Domain	Attr. Value
1 / O1.a1	Name	string	"cube"
2 / O2.a1	Name	string	"sphere"
:	:	:	:
:	:	:	:

O1 and O2 have the following contents:

O1:(Attributes:O1.a1)(Constraints:nil)(Children:nil)(Parent:nil)

O2:(Attributes:O2.a1)(Constraints:nil)(Children:nil)(Parent:nil)

The process proceeds as follows:

1. **User view:** The user instantiates the required objects O1 and O2.

   System view: The system performs this task by sending the **ObjectDuplicate** message to the Objects Manager:

   Object1 = [ObjectsManager ObjectDuplicate:O1]

   Object2 = [ObjectsManager ObjectDuplicate:O2]

   Note that a message has the following syntax which is similar to the syntax of the Objective C language:

   ReturnValue = [ReceivingObject Message:Parameter1:Parameter2...]

   The implementation of the first message above generates the following messages to the rest of the system:

   Object1 = [ObjectsManager ObjectCreate]

   Object1.a1 = [AttributesManager AttributeDuplicate:O1.a1]

   [Object1 AddAttribute:Object1.a1]

   Table 3.2 shows the attributes database at the end of this step:

   Object1:(Attributes:Object1.a1)(Constraints:nil)(Children:nil)(Parent:nil)

   Object2:(Attributes:Object2.a1)(Constraints:nil)(Children:nil)(Parent:nil)

Table 3.2: The attributes database after instantiating O1 and O2

ID/Symbol	Attr. Name	Attr. Domain	Attr. Value
1 / O1.a1	Name	string	"cube"
2 / O2.a1	Name	string	"sphere"
3 / Object1.a1	Name	string	"cube"
4 / Object2.a1	Name	string	"sphere"
:	:	:	:
:	:	:	:

2. **User view:** The user modifies the Name attribute of Object1 and Object2 and adds the Weight attribute to them.

   System view:

   [AttributesManager AttributeModify:Object1.a1::: "head"]

   [AttributesManager AttributeModify:Object2.a1::: "base"]

   Object1.a2 = [AttributesManager AttributeCreate:Weight:float:10]

   [Object1 AddAttribute:Object1.a2]

   Object2.a2 = [AttributesManager AttributeCreate:Weight:float:100]

   [Object2 AddAttribute:Object2.a2]

   This results in the following changes to the system objects and the attributes database (Table 3.3):

   Object1:(Attributes:Object1.a1,Object1.a2)(Constraints:nil)

   (Children:nil)(Parent:nil)

   Object2:(Attributes:Object2.a1,Object2.a2)(Constraints:nil)

   (Children:nil)(Parent:nil)

Table 3.3: The attributes database after Step 2

ID/Symbol	Attr. Name	Attr. Domain	Attr. Value
1 / O1.a1	Name	string	"cube"
2 / O2.a1	Name	string	"sphere"
3 / Object1.a1	Name	string	"head"
4 / Object2.a1	Name	string	"base"
5 / Object1.a2	Weight	float	10
6 / Object2.a2	Weight	float	100
:	:	:	:
:	:	:	:

3. **User view:** The user creates a new object Object3 as an aggregate of Object1 and Object2.

   System view: The system performs this task by creating a new system object Object3 and adding Object1 and Object2 to its list of constituent objects:

   Object3 = [ObjectsManager ObjectCreate]

   [Object3 AddChild:Object1]

   [Object3 AddChild:Object2]

   This results in the following:

   Object3:(Attributes:nil)(Constraints:nil)(Children:Object1,Object2)(Parent:nil)

   Object1:(Attributes:Object1.a1,Object1.a2)(Constraints:nil)

   (Children:nil)(Parent:Object3)

   Object2:(Attributes:Object2.a1,Object2.a2)(Constraints:nil)

   (Children:nil)(Parent:Object3)

4. **User view:** The user defines the Weight attribute for Object3 to be the sum of the Weight attribute values of Object1 and Object2.

   System view:

   Object3.a1 = [AttributesManager

   AttributeCreate:Weight:float:Object1.a2+Object2.a2]

   [Object3 AddAttribute:Object3.a1]

   Whenever the Weight attribute of Object3 is accessed, it is evaluated by the **AttributeEvaluate** method of the Attributes Manager which evaluates the formula of Object3.a1 and consequently evaluates Object1.a2 and Object2.a2. By the end of this step the attributes database looks as in Table 3.4:

Table 3.4: The attributes database after Step 4

ID/Symbol	Attr. Name	Attr. Domain	Attr. Value
1 / O1.a1	Name	string	"cube"
2 / O2.a1	Name	string	"sphere"
3 / Object1.a1	Name	string	"head"
4 / Object2.a1	Name	string	"base"
5 / Object1.a2	Weight	float	10
6 / Object2.a2	Weight	float	100
7 / Object3.a1	Weight	float	Object1.a2+Object2.a2
: :	: :	: :	: :

5. **User view:** The user sets an upper bound on the Weight attribute of Object3 by defining a constraint on it.

   System view: The system creates the constraint Object3.t(a1) on Object3.a1:

   Object3.t(a1) = [ConstraintsManager ConstraintCreate:Object3.a1:$<$ 200*]*

It checks its satisfaction:

Satisfied = [ConstraintsManager ConstraintVerify:Object3.t(a1)]

If the constraint is satisfied, which is the case, it gets added to Object3:

[Object3 AddConstraint:Object3.t(a1)]

If the constraint was not satisfied, the Constraints Manager would have not allowed its addition. In such a case, the user can try to enforce the constraint which is supported by the **ConstraintEnforce** method of the Constraints Manager. Table 3.5 shows the constraints database with the added constraint:

Table 3.5: The constraints database after Step 5

ID/Symbol	Attribute	Constraint Definition
1 / Object3.t(a1)	Object3.a1	< 200
:	:	:
:	:	:

Object3:(Attributes:Object3.a1)(Constraints:Object3.t(a1))

(Children:Object1,Object2)(Parent:nil)

Having the attributes and constraints of Object3 been defined in the last step, it becomes ready to be used like any of the existing objects. So, if the user instantiates Object3, the system responds by making a duplicate of it (Object3Dup). This duplicates the attributes, constraints and constituent objects of Object3 which results in the following changes (Tables 3.6 and 3.7)

Table 3.6: The attribute database after instantiating Object3

ID/Symbol	Attr. Name	Attr. Domain	Attr. Value
1 / O1.a1	Name	string	"cube"
2 / O2.a1	Name	string	"sphere"
3 / Object1.a1	Name	string	"head"
4 / Object2.a1	Name	string	"base"
5 / Object1.a2	Weight	float	10
6 / Object2.a2	Weight	float	100
7 / Object3.a1	Weight	float	Object1.a2+Object2.a2
8 / Object1Dup.a1	Name	string	"head"
9 / Object2Dup.a1	Name	string	"base"
10 / Object1Dup.a2	Weight	float	10
11 / Object2Dup.a2	Weight	float	100
12 / Object3Dup.a1	Weight	float	Object1Dup.a2+Object2Dup.a2
: :	: :	: :	: :

Table 3.7: The constraint database after instantiating Object3

ID/Symbol	Attribute	Constraint Definition
1 / Object3.t(a1)	Object3.a1	< 200
2 / Object3Dup.t(a1)	Object3Dup.a1	< 200
: :	: :	: :

Object3Dup:(Attributes:Object3Dup.a1)(Constraints:Object3Dup.t(a1))

(Children:Object1Dup,Object2Dup)(Parent:nil)

Object1Dup:(Attributes:Object1Dup.a1,Object1Dup.a2)(Constraints:nil)

(Children:nil)(Parent:Object3Dup)

Object2Dup:(Attributes:Object2Dup.a1,Object2Dup.a2)(Constraints:nil)

(Children:nil)(Parent:Object3Dup)

3.3 Summary

This Chapter presented an object-oriented design of a modeling system based on the mechanisms developed in the Chapter 2. The functionality of the system is decomposed into a collection of service providing managers. The functionality of each manager was specified through its methods. Finally, an example was given to illustrate the interaction among the managers.

Chapter 4

3D Graphics System Design

This Chapter presents a design for a 3D graphics modeling system by customizing the design presented in the previous Chapter. The system is to model 3D graphics components and to support all the construction schemes (top-down, bottom-up, and middle-out).

The entities of this system are 3D graphics components. Each component is represented by two objects: an *Icon* object that provides an iconic representation of the component and a *3D Graphics Component* object that represents the component itself.

The user interface of the tool handles the user input and provides access to the

system services hidden behind the user interface. These services are provided by the *Camera* object that provides the viewing mechanism, the *3D Graphics Component* objects, the Types, Attributes and Constraints managers introduced in the previous Chapter.

4.1 Camera Object

The *Camera* object provides the facility to view components placed in the world space. This requires mapping the three-dimensional world space into a two-dimensional image. The camera is placed in a specific position and has an orientation along which it projects a view of the world scene. Typical 3D Graphics systems use five different coordinate systems. These are shown in Figure 4.1.

The world space has a coordinate system called the *world coordinate system* or the *global coordinate system.* The world coordinate system is the principal frame of reference. It is the basis for defining and locating all objects in space including the camera's position. It serves as the master reference system in which all other coordinate systems are defined.

A *local coordinate system* is used to define the geometry of a component independent of the world system. Once the component is defined locally, it can be

placed in the world system by specifying the location and orientation of the local system within the world system which means a transformation is applied on the local coordinate system to transform it to the world coordinate system.

Once a component is defined in the global system, we can view it from any direction. To do this, we need to define the location and the direction of the line-of-sight of the camera. Define an eyepoint, P_E, and a viewpoint, P_V, by specifying their coordinates in the world coordinate system. P_V acts as the center of the scene viewed by the camera. P_E specifies the origin of the *camera coordinate system* and the vector from P_E to P_V specifies the line-of-sight of the camera. The z-axis of the camera coordinate system always points along the line of sight.

The *projection plane* is a plane on which the three-dimensional component is projected. It is perpendicular to the line of sight and usually located between the component and the camera. There are two types of projections: *perspective* and *parallel.* A perspective projection is produced when the lines of projection converge on the eyepoint. While, constructing the lines of projection to be parallel to the line of sight produces a parallel projection.

Finally, the projection plane is mapped to a window on the display referred to as the construction window since it views the process of constructing components.

The *Camera* object has the following instance variables:

CameraPosition: Specified by the eyepoint, P_E, and the viewpoint, P_V, of the camera.

ProjectionPlaneLocation: Specified as a distance between the $x-y$ plane of the camera and the projection plane.

ProjectionType: Specifies whether the projection is perspective or parallel.

The following methods are provided by the *Camera* object:

Display: Displays the components placed in the world space on the construction window as explained above.

Iconify: Produces an image of a constructed component with the size of an icon by mapping the projection plane into a view of the size of an icon. The image produced is used as an icon for the constructed component.

4.2 Icon Class

The *Icon* class provides the facility to have icon representation for the components to be used in construction. The icons of these components are displayed on a palette. When the user wants to instantiate a component, its icon is dragged from the palette to the construction window. If a newly created component is to be used in future

constructions, then an icon is created for it using the **Iconify** method of the *Camera* object and the icon gets added to the palette.

The *Icon* class has the following instance variables:

Name: Stores the name of the component the icon is representing.

Icon: A reference to a file where the icon representation is stored.

The following methods are provided by the *Icon* class:

Load: Loads the icon from its image file into the palette.

Highlight: Highlights the icon when it is clicked on.

Trigger: Triggers the instantiation of a component, when its icon is dragged from the palette to the construction window.

4.3 3D Graphics Component Class

The *3D Graphics Component* class is a customization of the *Object* class, introduced in the previous Chapter, to enable it to represent 3D graphics components. So, this class defines the attributes and constraints required to represent 3D graphics components and additional methods to handle 3D graphics operations.

The *Object* class models components in a hierarchical manner. A component is composed of other components — its children components — which in turn can be composed of other components and so on. Therefore, a component has a tree structure as Figure 4.2 illustrates by an example. Note that the leaf nodes represent primitive components, and vice versa, non-leaf nodes represent aggregates (non-primitives). The component illustrated in the figure is composed of two aggregate components: one of them is composed of two primitives, while the other is composed of two primitives and one aggregate.

4.3.1 Required Attributes and Constraints

Table 4.1 summarizes the attributes and constraints added to the *3D Graphics Component* class.

The *Name* attribute stores the name of the component.

The *Transform* attribute stores the transformation matrix of the component. As mentioned before, each component has a local coordinate system in which its geometry is defined. The transformation matrix of a component transforms the local coordinate system of the component to its ancestor's coordinate system. Since a component is located in its ancestor's coordinate system, any transformations performed on its ancestor's system are applied indirectly on it. Consequently, a

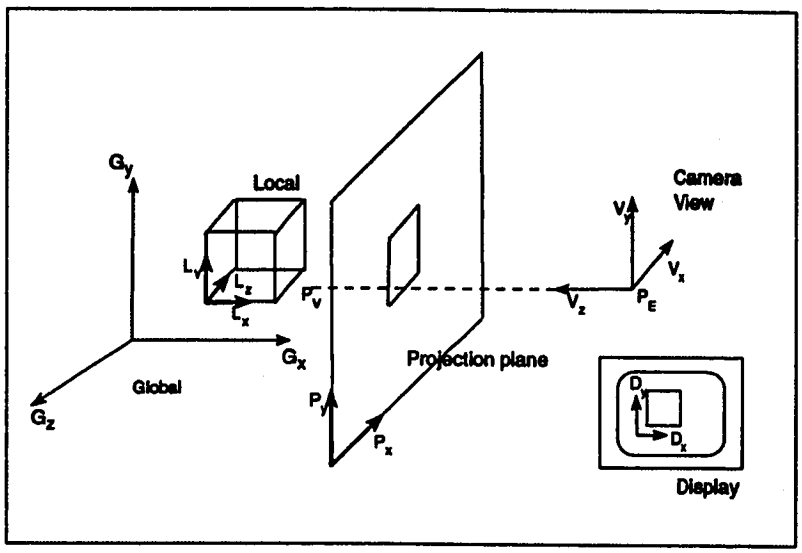

Figure 4.1: Coordinate systems used to construct a computer graphics display

Table 4.1: Required attributes and constraints to represent 3D graphics components

Attribute	Domain	Value	Constraint
Name	string	(... Name ...)	$Length(Name) < MaxLength$
Transform	array[4][4]	Transformation matrix	Valid transformation matrices
Geometry	string	3D graphics representation	
BoundingBox	array[3][2]	Bounding box	$x_2 < c_1, y_2 < c_2, z_2 < c_3$

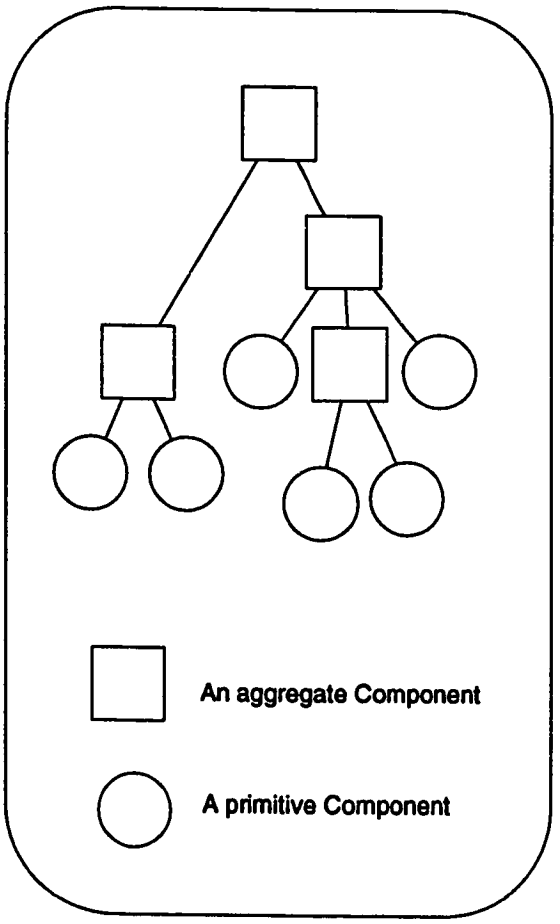

Figure 4.2: An example of a component structure

component inherits the transformation done on its ancestor and its ancestor's ancestors all the way to the top of the tree hierarchy. For the component at the top of a tree, the transformation matrix transforms its coordinate system to the world coordinate system. Finally, the Default value for the *Transform* attribute is the identity matrix.

The *Geometry* attribute stores the 3D graphics geometric representation of the component. The geometric representation is defined using a 3D graphics language like RenderMan mentioned in the first Chapter. The geometric representation of a component is stored in a file, and the *Geometry* attribute stores a reference to the file. Primitive components are assigned geometries that give them the ability to be used in constructing the required models. On the other hand, aggregate components have bounding box geometries, i.e., up-right wire-frame boxes that bound their composing components. The dimensions of the bounding box of an aggregate component is computed by its *BoundingBox* attribute.

The *BoundingBox* attribute stores the bounding box of the component. The bounding box is represented by three bounds, one in each dimension ($x_1 - x_2, y_1 - y_2, z_1 - z_2$). The bounds of a component, like its geometry, are defined in its local coordinate system. The intersection of the six planes ($x = x_1, x = x_2, y = y_1, y = y_2, z = z_1, z = z_2$) makes the bounding box. The bounding boxes of primitive components depend on their geometries and thus the bounds are set after the ge-

ometrics are defined. The bounding boxes of aggregate components are computed as the union of the bounding boxes of its composing components.

The *3D Graphics Component* class sets constraints on the defined attributes as illustrated by Table 4.1. There is a constraint on the length of the component's name, and another on the *Transform* attribute to ensure that only valid transformations can be performed. For example, transformations that scale a component too small or translate it too far are rejected. Also, assigning invalid transformations matrices to *Transform* — like a matrix with its diagonal elements other than ones — is not permitted by the constraint on *Transform*. The constraint on the *BoundingBox* attribute limits the values of its bounds, and thus limits the size of the bounding box. Likewise, if the user requires the component to be of a fixed size, additional constraints can be defined on *BoundingBox* ($|x_1 - x_2| = c_1, |y_1 - y_2| = c_2, |z_1 - z_2| = c_3$).

Since any component, whether it is a primitive or an aggregate, is an object of the *3D Graphics Component* class, it has the attributes and constraints added to the class. Primitive components are expected to be added to the system when it is configured to be ready to be used by the end-user (the modeler). To add a primitive component, an object of the *3D Graphics Component* class is instantiated and its *Name*, *Geometry*, and *BoundingBox* attributes are set. The default value of the *Transform* attribute, the identity matrix, is not changed.

4.3.2 Additional Methods

The following methods are added to the *3D Graphics Component* class to give it the capability to handle 3D graphics operations.

Translate Applies a given translation on the component. This is achieved by calling the **ModifyAttribute** method of the Attributes Manager to set a new value to the *Transform* attribute of the component. Next, the *Camera* object is sent a Display message to redisplay the world, so the translation is reflected.

Scale Applies a given scaling on the component. This is achieved by calling the **ModifyAttribute** method of the Attributes Manager to set a new value to the *Transform* attribute of the component. Next, the *Camera* object is sent a Display message to redisplay the world, so the scaling is reflected.

Rotate Applies a given rotation on the component. This is achieved by calling the **ModifyAttribute** method of the Attributes Manager to set a new value to the *Transform* attribute of the component. Next, the *Camera* object is sent a Display message to redisplay the world, so the rotation is reflected.

4.4 User Interface

The user interface handles the user input and provides access to the system services. It consists mainly of the construction window, the palette and the user interface controls required to express the definition of types, attributes and constraints graphically.

For example, to define an attribute for a component, the user browses through the existing components and selects the required one by a mouse click. Next, the user starts building the definition expression where the supplied user interface controls allow the user to select the required function or control structure (conditional and iterative control structure) graphically (i.e., clicking on a graphical symbol of a control structure, gets the structure inserted in the definition expression at the cursor position). The operands of the selected function or control structure can be constants, attributes (of the same or different component), functions or control structures and are selected graphically as well. This simplifies defining the attributes and ensures their correct syntax. Constraints and types are defined graphically in a similar fashion.

Also, the user interface provides interfaces to perform the following tasks:

- Selecting a component: a component is selected by clicking on it which gets it

highlighted by changing its color. The Attributes Manager is called to change the color attribute.

- Transforming the selected component: the selected component is sent a message to transform itself accordingly.

- Instantiating a component from the palette by calling the **ObjectDuplicate** method of the *3D Graphics Component* class.

- Constructing a component: calling the methods involved in construction as detailed in the example of previous Chapter. To support top-down construction scheme, an empty aggregate component, an aggregate with no children, is added to the list of existing components in the palette.

- Adding a component to the palette: calling the **Iconify** method of the *Camera* object to produce an icon that gets loaded in the palette by invoking its **Load** method.

4.5 Summary

A design for a 3D graphics system was presented. Various attributes related to graphics were introduced and the methods that affect them were explained. Finally, user interface issues were discussed.

Chapter 5

Prototype

A prototype modeling system was implemented to help the development and testing of the mechanisms introduced in the 3D model. The prototype is a simple 3D graphics hierarchical modeling system. It provides the ability to build reusable aggregate 3D graphics components from the existing ones. The prototype focuses on the 3D graphics attributes of components and defines them in the code. It is restricted in its functionality in the sense that it doesn't support the interactive definition of attributes and constraints.

The code of the prototype is listed in Appendix A. This Chapter explains the prototype code after introducing the platform on which it was implemented. It concludes by giving an example that illustrates the prototype. Appendix A includes

the classes of the graphical components required for this example.

5.1 Implementation Platform

The prototype was implemented on a *NeXT* machine running the *NeXTStep* operating system. This section introduces the *NeXT* platform (for more information refer to *NeXT* documentation). As illustrated in Figure 5.1, there are four levels of software between a *NeXT* application program and the hardware that executes it:

- *NeXT* development tools
- Object-oriented software kits
- *NeXT* Window Server and specialized C libraries
- Mach operating system

A brief description of these levels is presented NeXT.

5.1.1 Development Tools

The main development tools under *NeXTStep* [NeX92] are the *Interface Builder* and the *Project Builder*. Project Builder works like a control center which keeps track

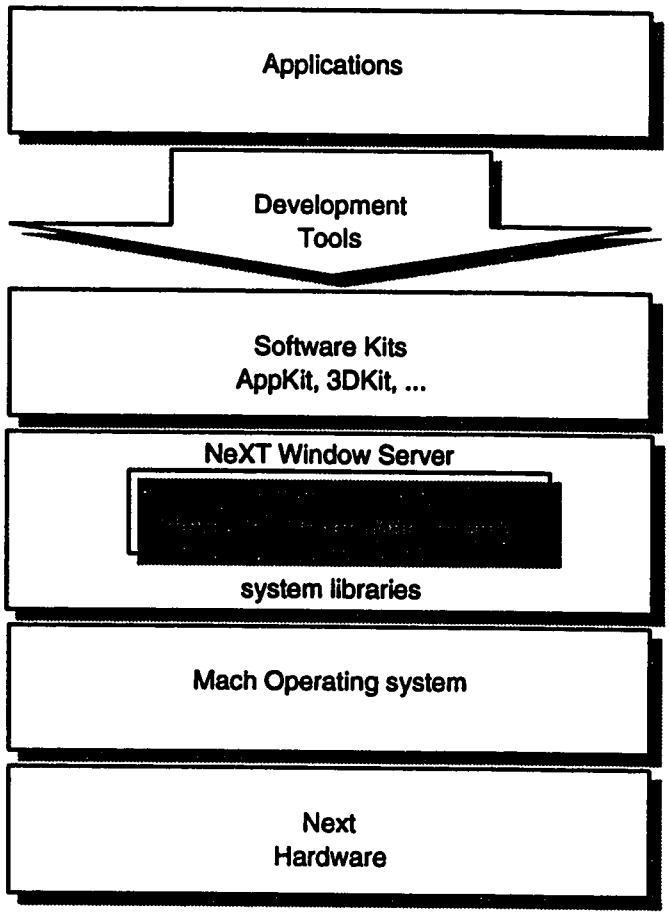

Figure 5.1: *NeXT* Platform

of the elements that make up the application and gives access to other development tools. So, the developer builds the interface, edits the source files, compiles and debugs the application while Project Builder is keeping track of the involved files with no interventions.

Interface Builder is a powerful tool that lets the developer graphically design an application's user interface. It also makes it easy to establish connections between user-interface objects and code (for example, the code to execute when a button on the screen is clicked).

5.1.2 Software Kits

NeXT application programs are written in Objective-C, an extension to C that adds object-oriented concepts to the language. The software kits define a number of classes, or object templates, that can be used in applications. The software kits that will be used by this project are the *Application Kit* and the *3D Graphics Kit.*

All applications use Application Kit regardless of their purpose and complexity. The buttons, sliders, and windows that are used to design an interface with Interface Builder are defined as classes in Application Kit. Also, as described in the next section, it is through this Kit that an application is able to draw on the screen and receive events from the user.

The Objective-C language and the software kits make it easy to create a new class of object. One of the features of the language is that it supports class inheritance; this means that one can create a class that inherits the attributes of another class. For example, we can create a class that inherits from the Application Kit's Button class. The new version of Button will be able to do everything that the Kit version can do, plus we can add to it the specialized functionality that our application requires.

3D Graphics Kit enables applications to model and render 3-dimensional scenes. While Application Kit has 2D graphics capabilities only, 3DKit provides 3D graphics capabilities. It models a 3D object by storing its graphical attributes (shape, transformation, lighting, shading and camera view). Based on the model, it generates *RenderMan Interface Bytestream* (RIB) code to describe the scene to a renderer.

Just as the *PostScript* language is frequently referred to as a page description language, the *RenderMan* language can be thought of as a scene description language. It provides graphic primitives, lighting specification, camera controls, and other features required for 3D scene description.

5.1.3 Window Server

NeXT Window Server is a low-level background process that creates and manip-

ulates windows on the screen. An application establishes a connection with the Window Server through the Application Kit and opens one or more windows. Windows provide a vehicle for communication between the user and the application. The Window Server manages this communication as it fulfills two functions:

- It draws images on the screen according to instructions sent from the application.
- It sends user events back to the user's application.

Both Application Kit and 3D Kit drawing capabilities are based on a client-server model, in which client applications send drawing code to the window server, which does the actual drawing. One difference in the implementations is in the code generated for drawing. For 2D drawing, a View (a class which provides a structure for drawing on the screen and for handling mouse and keyboard events) sends PostScript code to the window server's Display PostScript interpreter. For 3D drawing, a View sends RenderMan Interface Bytestream (RIB) code to the window server's Interactive RenderMan renderer.

5.1.4 Mach

Mach is a multitasking operating system developed at Carnegie Mellon University

which provides complete compatibility with UNIX 4.3BSD (Berkeley Software Distribution). It acts as an interface between the upper levels of software and *NeXT* hardware.

5.2 3D Graphics Kit

As introduced in the previous section, 3D Graphics Kit enables *NeXTStep* applications to model and render three-dimensional scenes. It is based on the RenderMan Interface which is a standard API for 3D scene description. One of the main features of RenderMan Interface is that it separates modeling from rendering. A modeling program stores data for objects in 3D scenes and generates RIB code to describe that scene to a renderer like the Interactive RenderMan renderer introduced before. 3D Kit supports shading, light source management and rendering of animated sequences of 3D images. This section introduces the main classes in 3D Kit, followed by a more detailed description on *N3DCamera* and *N3DShape* classes as they are the basis for implementing the proposed 3D Graphics tool.

N3DCamera This class provides the facility to view *N3DShapes* placed in the world system. An *N3DCamera* instance has a pointer to a single *N3DShape* -its world shape- which is at the top of the hierarchy of shapes viewed by it.

N3DShape Objects of this class are used to represent surfaces and transformations in a scene. The *N3DShape* class defines instance variables for managing a hierarchy of shapes. To represent complex 3D objects with *N3DShapes*, hierarchical relationships are defined among them. Subclasses of *N3DShape* can be implemented to draw any of the primitive surface types defined in the RenderMan interface.

N3DMovieCamera This subclass of *N3DCamera* supports the rendering of animated sequences of 3D images. Movies can be rendered on-screen by the interactive renderer in the MovieCamera, or off-screen by the photorealistic renderer as a set of TIFF or EPS streams.

N3DLight *N3DLight* provides light source management in the *N3DShape* hierarchy. *N3DLight* is a subclass of *N3DShape* -thus its instances can be placed in the world and connected to other shapes. Independently of its location in the Shape hierarchy, an *N3DLight* can be set to illuminate the entire scene or just the *N3DShape* object to which it is directly connected.

N3DShader The *N3DShader* class manages shader functions written in the RenderMan Shading Language. Every *N3DShape* can have one each of the six standard shaders types (Surface, Displacement, Lightsource, Imager, Volume and Transformation).

5.3 Implementation Details

In implementing the prototype, the *N3DCamera* and the *N3DShape* classes were used extensively. An instance of the *Camera* class which is defined as a subclass of *N3DCamera* views the model being constructed on a window -the construction window- and facilitates its interactive manipulation. A model is a set of components each structured as a tree of lower-level components as explained before.

This prototype supports the bottom-up construction scheme in which the components of a constructed model are put together to form a new higher-level component. To facilitate interactive construction, a palette which has a matrix of icons for the existing components is supplied. The user constructs a model by instantiating the desired components by double-clicking on their icons. Once the construction is done, the user can save the model or create a component out of it which gets added to the palette.

N3DShape class provides the facility to represent 3D graphics components. The *Shape* and the *ComShape* classes are defined subclass of *N3DShape* to support primitive and aggregate components respectively. Each primitive component is represented by a subclass of *Shape* that defines its geometric representation. The rest of this section explains the defined classes in detail.

5.3.1 Camera Class

Camera is a subclass of *N3DCamera*. *N3DCamera* provides the facility to view *N3DShapes* placed in the world system. An *N3DCamera* instance has a pointer to a single *N3DShape* -its world shape- which is at the top of the hierarchy of shapes viewed by it. Figure 5.2 shows the world and camera coordinates systems as defined by 3D Kit. Note that both of the coordinate systems are left-handed systems.

N3DCamera is a subclass of the View class which provides the facility to draw on the screen and handle mouse and keyboard events. So, the *N3DCamera* not only views the world shape but also handles the user interaction with this view. This is apparent through the inherited **mouseDown:** method. The *N3DCamera* class inherits the **mouseDown:** method which is invoked whenever the mouse is clicked and returns the coordinates of the mouse click and other details.

The **render** method defined by the *N3DCamera* class provides the basic functionality to display a camera view. It renders the world shape of the camera, and any shapes in the world shape's hierarchy. It is invoked by **drawSelf::** method to perform 3D rendering whenever the camera is displayed.

The *Camera* class is built on the *N3DCamera* class to provide the ability to view a model and allow its construction interactively. A model is constructed as a

hierarchy of shapes. A single instance of the *Camera* class is needed which has the same name Camera as well. The following are the main methods defined for the *Camera* class:

mouseDown: overrides the inherited **mouseDown:** method. It handles the mouse click, so if the mouse is double clicked it invokes the **selectShape:** method.

selectShape: selects the shape determined by the mouse click if any. The instance variable selectedShape of Camera stores the ID of the selected shape. The selected shape is flagged as selected and highlighted. A primitive shape is highlighted by displaying its bounding box. While, a complex shape is highlighted by changing its color to green since it is itself a bounding box. When, a complex shape is displayed it checks its selection flag. If the flag is set, it changes its color to green, otherwise it keeps it white as a default color. The **selectShape:** method ends by redisplaying the camera to show the changes (change of color or displaying a bounding box).

setBoundingBox: sets a box shape bounding the selected shape to highlight it, if it was primitive. Since only one shape is permitted to be selected at a time, a single bounding box shape is needed to highlight the selection. The instance variable theBoundingBox of Camera stores the ID of the bounding box shape. The bounding box shape has a box geometry with dimensions set by this method to the dimensions of the bounding box of the selected shape.

The bounding box shape is linked to the selected shape as a child so that any transformations done on the selected shape are inherited by the bounding box, effectively forcing the bounding box to follow the selected shape as it is transformed.

addToPal: creates a component, saves it and loads its icon in the palette. The component is created by inserting a *ComShape* as root to the shape hierarchy. Effectively, the world system is replaced by the inserted *ComShape*, so the transformations on the shape hierarchy become relative to the the *ComShape*. The component is assigned a name and saved into a file which has the same name and the extension "mdl" (which stands for model). Its icon is saved to a file with the same name as well, but with the extension "tiff".

icon: produces a tiff icon representation of the component to be created. It sets the frame size of the camera to the icon size and renders the world shape as tiff.

load: loads a component from a file and links it as a peer to the world shape. A component is loaded by this method when its icon is double clicked.

5.3.2 Shape Class

The *Shape* class is a subclass of the *N3DShape* class. *N3DShape* provides techniques for representing 3D transformations, for rendering the standard RenderMan surface primitives, and for creating and managing hierarchically organized structures. A shape hierarchy is made up by linking shapes in two kinds of relationships: descendant/ancestor and nextPeer/previousPeer. Figure 5.3 shows an example of a shape hierarchy. Each shape has a transformation matrix that it applies when it is rendered. A shape inherits the transformation of its ancestor.

A descendant and its peers share the same ancestor, and thus the same inherited attributes. The term descendants applies to a shape's descendant, all descendants below the descendant, and all peers of all descendants. Each peer descended from a common ancestor can apply its own attributes, independent of other members of its peer group.

Mapping a model hierarchy to an *N3DShape* hierarchy is simple. Figure 5.4 illustrates an example of this mapping. In an *N3DShape* hierarchy, a shape has only one descendant link, the other descendants are linked as peers to the descendant. The *Shape* class defines methods to save and load a shape hierarchy:

save: Saves a shape hierarchy to a file which is passed as an argument. Figure

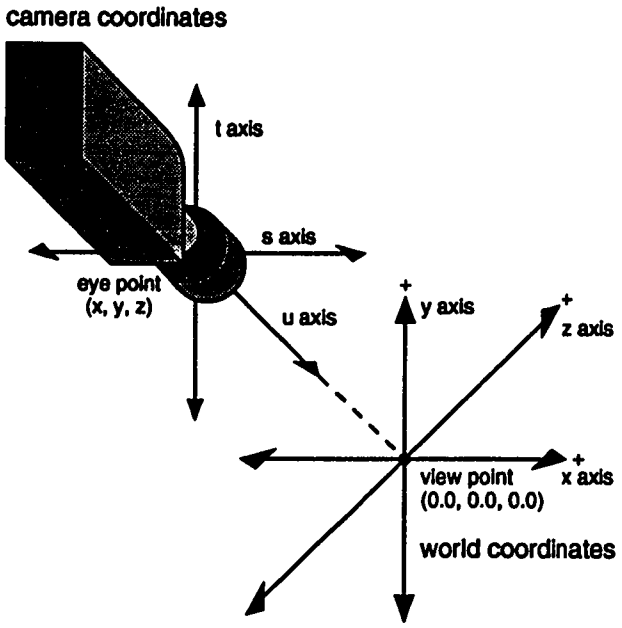

Figure 5.2: World and Camera coordinate systems

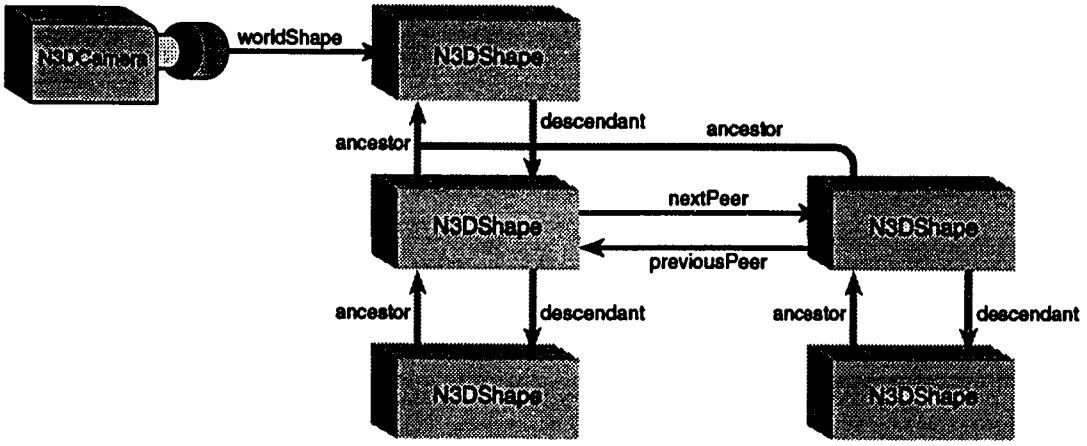

Figure 5.3: A Shape Hierarchy

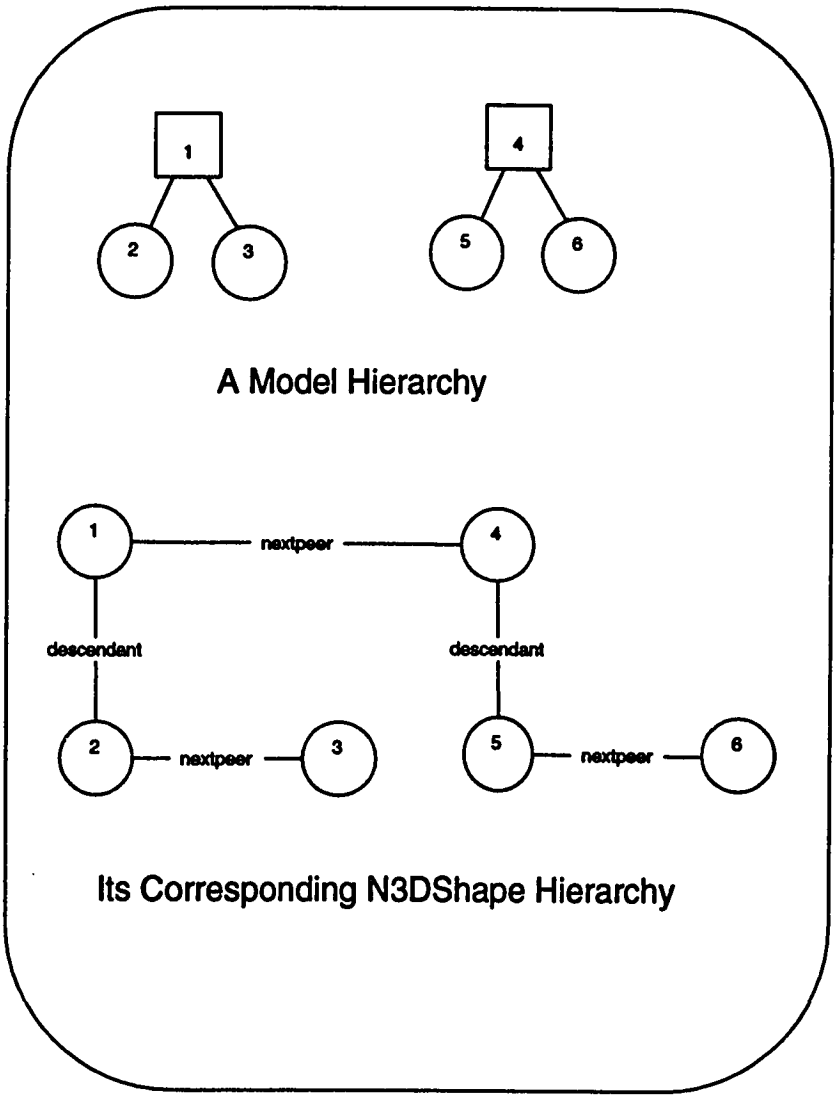

Figure 5.4: A Model hierarchy and its corresponding *N3DShape* hierarchy

5.6 shows the algorithm as a pseudo-code for saving a shape hierarchy by recursively traversing it. When, a shape is traversed, it is saved to the file by writing its class and transformation matrix. The descendant function returns the descendant of the passed shape, or nil if the shape doesn't have a descendant. Similarly, nextPeer returns the next peer of the passed shape, or nil if the shape doesn't have a next peer. The descendantLink, nextpeerLink and ancestorLink are of enumeration type.

load: Loads a shape hierarchy from a file saved by the **save:** method. Figure 5.5 shows the algorithm as a pseudo-code for loading a shape hierarchy by recursively reading its shape specifications from the passed file. A shape is created from a read specification (class and transformation matrix) by instantiating a shape of the read class and setting its transformation matrix to the read transformation matrix. This method returns the ID of the top shape of the read shape hierarchy. The setDescendant and setNextpeer functions set respectively the descendant and the next peer of a shape passed in the first argument to the shape passed in the second argument.

```
{... Declare shape, link ...}
load(file)
Begin
        read(class,transform,file)
        shape = instance(class)
        setTransformationMatrix(shape,transform)
        read(link)
        If link = descendantLink
            setDescendant(shape,load(file))
        Elseif link = nextpeerLink
            setNextpeer(shape,load(file))
        Elseif link = ancestorLink
            read(link)
        Endif
        return(shape)
End
```

Figure 5.5: Loading a shape hierarchy

```
save(shape,file)
Begin
        write(shapeClass,shapeTransform,file)
        If descendant(shape)
            write(descendantLink,file)
            save(descendat(shape),file)
        Endif
        If nextPeer(shape)
            write(nextpeerLink,file)
            save(nextPeer(shape),file)
        Else
            write(ancestorLink,file)
        Endif
        return
End
```

Figure 5.6: Saving a shape hierarchy

5.3.3 ComShape Class

ComShape is a subclass of *Shape* that implements an up-right wire-frame bounding box geometry for aggregate components. The bounding box of a *ComShape* is always up-right and dynamically changes its size to bound its composing components as they are transformed. The bounding box is represented by three bounds, one in each dimension ($x_1 - x_2, y_1 - y_2, z_1 - z_2$). The intersection of the six planes ($x = x_1, x = x_2, y = y_1, y = y_2, z = z_1, z = z_2$) makes the bounding box. The *ComShape* class defines the following methods:

renderSelf: renders a *ComShape* as an up-right wire-frame bounding box. The bounding box is computed by the **getBoundingBox:** method which returns the bounds of the bounding box. To keep the bounding box always up-right, the transformation inherited by the *ComShape* should not change its orientation. Therefore, the transformations on a *ComShape* are inverted and applied on its bounds since the bounds will always define an up-right bounding box. Transformations are applied on the bounds by finding the eight vertices of the bounding box they make, applying the transformations on the vertices and getting the new bounds by taking the minimum and maximum of the vertices' coordinates.

getBoundingBox: computes the bounding box of a *ComShape* and returns its bounds. This method overrides the **getBoundingBox:** method inherited from the *N3DShape* class. It computes the bounding box of a *ComShape* by taking the union of the bounding boxes of its composing components. This results in recursively invoking this method, if one of the composing components is an aggregate. For primitive components, the **getBoundingBox:** method returns the bounds stored in the boundingBox instance variable which is set depending on the geometry of the primitive.

5.3.4 Supporting Classes

The *Transform* class handles transformations on the selected shape. It allows translating, scaling, and rotating the selected shape and consequently transforming its descendants. This results in updating the transformation matrix of the selected shape. Also, Transform gives the ability to change the eye point and view point of the camera.

The *Palette* class implements the palette of the system. The palette views the icons of the existing components so that the user can get the desired component in the construction window by double clicking on it. The palette is structured as a matrix of cells each representing an icon of a component. The Palette provides

methods to implement the following:

- Load the palette at startup time by reading the file which stores the names of the existing components ("default.pal") and loading the icons of each of these components in the palette.
- Add the icon of a new component into the palette.
- Trigger the instantiation of a component when the user double-clicked on its icon.

  The *Icon* class provides the ability to draw the icon to be loaded in the palette. It supplies the **drawInside::** method to draw the icon as a combination of the image representation of the desired component stored in the tiff file and its name.

5.4 Example

To illustrate the developed tool, consider an example of building a shuffle architecture from the domain of optical architectures. In this example we build a shuffle optical architecture [McA91] that is composed of four inter-connected shuffle networks. The needed primitive components for this optical architecture are: lenses, semi-lenses and prisms. So, the *Lens* class, the *SemiLens* class and the *Prism* class

are defined as subclasses of the Shape class. Each of these classes defines the geometry of its component by overwriting the **renderSelf:** method.

The goal of this example is to build a shuffle network like the one in Figure 5.7 and use it to build an optical architecture composed of four networks. The process starts by building the shuffle network out of the primitive components as follows:

- A prism is instantiated by double-clicking on its icon in the palette. As a result, a prism component is placed at the origin of the world system as illustrated in Figure 5.8. The Figure shows the instantiated prism in the construction window. Note that the camera is put at an angle to show an oblique view, rather than a side view.

- The prism is selected by double-clicking on it. As a result, the prism gets highlighted by displaying its bounding box.

- The prism is transformed as needed (scaled, rotated, and translated to its final position) as in Figure 5.9.

- The previous three steps are repeated for the other constituent components of the shuffle network. Each constituent component is instantiated, selected and transformed to form the shuffle network in Figure 5.10.

- A new component is created of the formed shuffle network. Its icon gets added

to the palette.

- The shuffle network component is used to build a shuffle optical architecture composed of four inter-connected shuffle networks by instantiating four instances of the shuffle network component and transforming them as required. Figure 5.11 illustrates the created shuffle architecture. The Figure shows the shuffle network component icon displayed in the palette along with the other component icons.

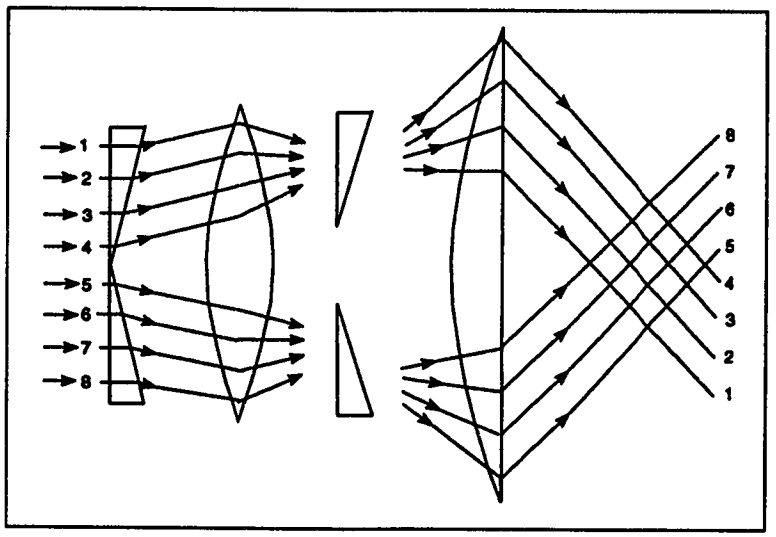

Figure 5.7: A shuffle optical architecture

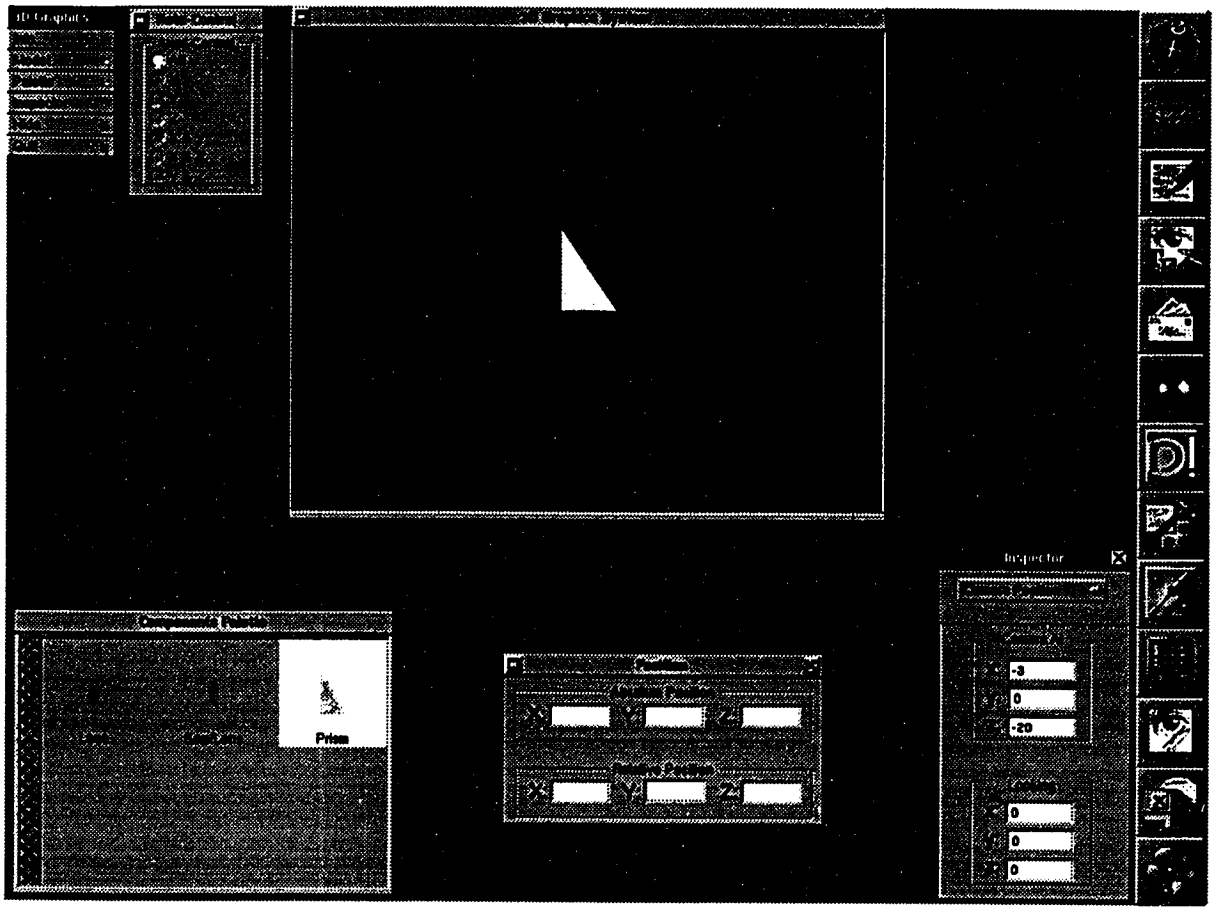

Figure 5.8: An instantiated prism

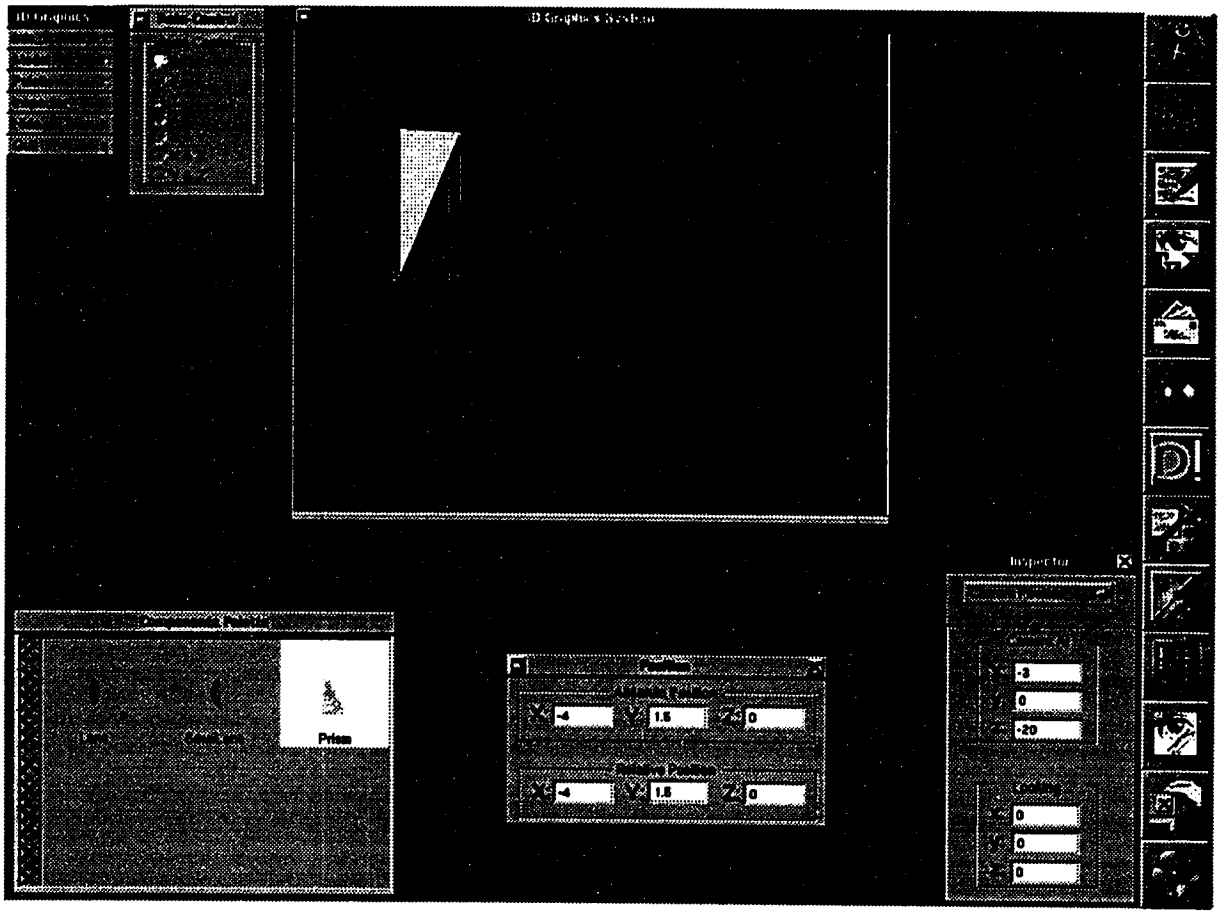

Figure 5.9: The prism selected and transformed

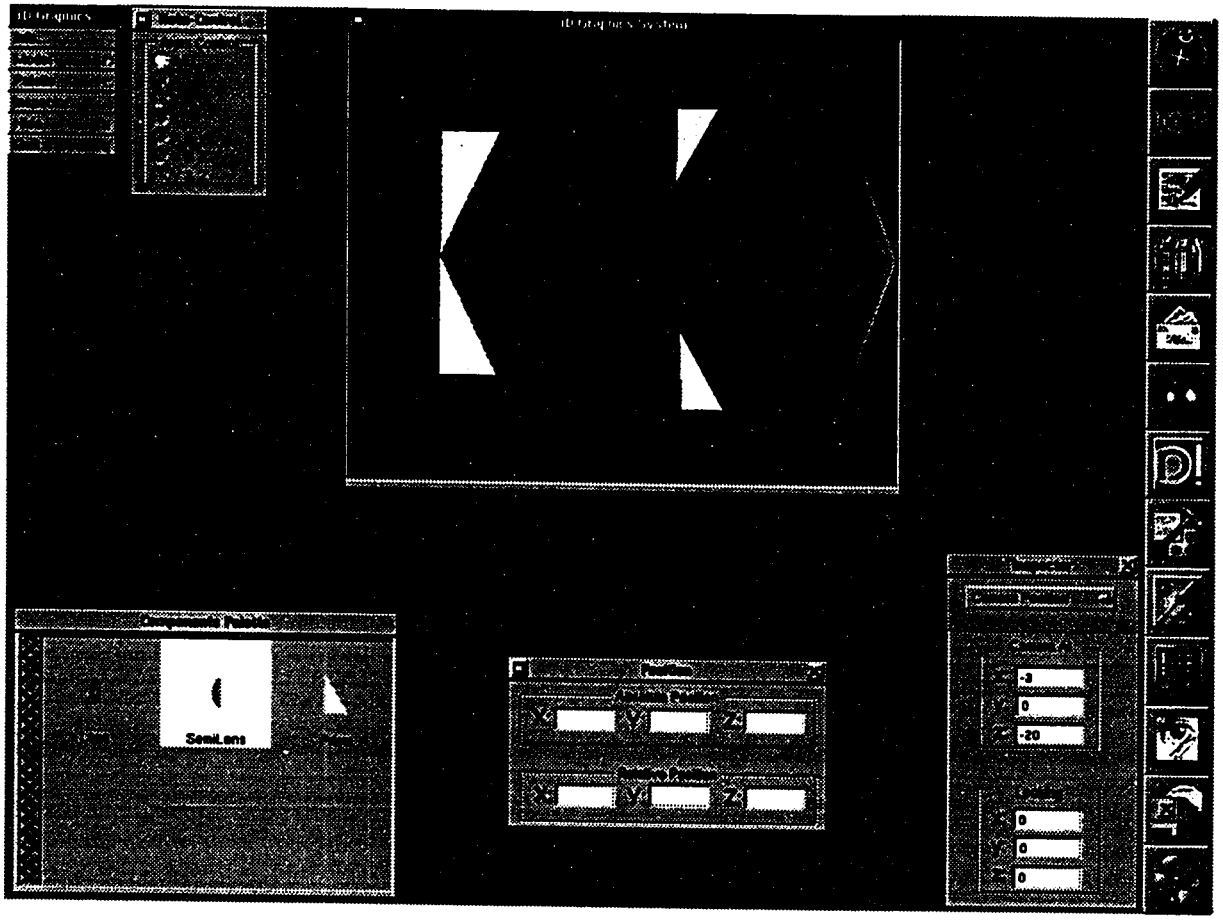

Figure 5.10: A shuffle network

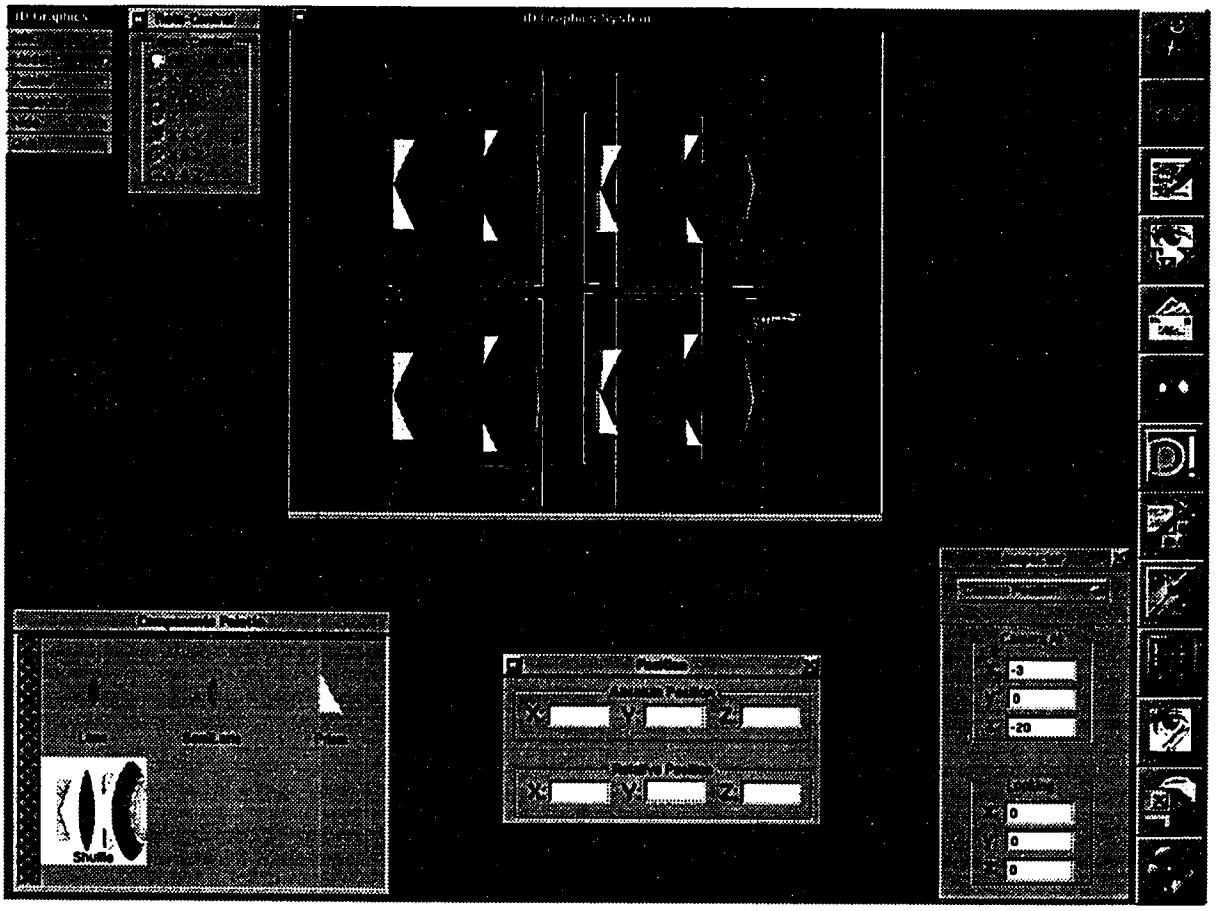

Figure 5.11: A shuffle optical architecture

Chapter 6

Conclusion

Current graphics modeling systems don't provide the user adequate support to model his real-world system at a convenient level of abstraction. Through our work, we considered a number of ideas that raise the level of interaction between the user and the modeling system. Object-orientation gives the user the ability to work at his application natural level. Interactive constraint handling releases the user from worrying about satisfying the required constraints on his design manually. Finally, hierarchical development provided by a construction model that supports bottom-up, top-down and middle-out construction schemes gives the user power and convenience in developing his design.

6.1 Objectives

The basic objective of our work is to find and explore the requirements that need to be built in a modeling system which gives the user the ability to work at a convenient level of abstraction without worrying about the representation details. Based on the considered requirements, a design for a general modeling system is to be developed.

Then, the general design is to be customized to support 3D graphics modeling. A subset of the design is to be implemented to help in developing and testing the mechanisms introduced by the design. The system design is to be implemented on the *NeXTStep* platform and make use of its supplied 3D graphics API (3DKit) capabilities. Finally, the implemented tool is to be illustrated by customizing it to support designing optical architectures.

6.2 Accomplishments

Our work helps in developing various techniques that enhance modeling environments. We adopted the object-orientation methodology in the modeling process and incorporated it with a hierarchical modeling scheme that supports bottom-up construction, top-down decomposition, and their combination. The modeling process was enhanced further by supporting the manipulation of constraints on object

attributes which automates satisfying system requirements.

Based on the developed techniques, a modeling system was designed. The system design is objected-oriented, employing a collection of service providing objects. To support hierarchical modeling, the design restricts attributes to be inherited or synthesized and constraints to be defined on one attribute only. The design was specialized to support 3D graphics by introducing the Camera and Icon classes and adding the required 3D graphics attributes to the objects: transformation matrix, 3D graphics representation, and bounding box.

A subset of the design was implemented as a prototype to help in developing the introduced techniques. The implementation tested these techniques and the feedback was used in refining the design. To control the complexity, only few constraints were considered and the definitions of the attributes and constraints were implemented in the code rather than providing the facility to define them interactively. Finally, the prototype was illustrated as a design tool for optical architectures.

6.3 Future Work

Our work introduced a number of ideas that enhance the modeling process. These ideas can be enhanced further. Specifically, constraint enforcement strategies and

evaluation techniques need a through study to further automate satisfying system requirements. Also, constraint-specification can be simplified to the end user by employing graphical constraint-specification techniques. Finally, the developed design can be fully implemented and customized to specific domains.

Appendix A

Prototype Code

This appendix lists the code of the classes introduced in the Prototype Chapter (Chapter 5) followed by the listing of the *make* file.

3DGraphics_main.m

```
/* Generated by the NeXT Project Builder
   NOTE: Do NOT change this file -- Project Builder maintains it.
*/

#import <appkit/Application.h>

void main(int argc, char *argv[]) {

    [Application new];
    if ([NXApp loadNibSection:"3DGraphics.nib" owner:NXApp withNames:NO])
            [NXApp run];

    [NXApp free];
    exit(0);
}
```

Camera.h

```
#import <3Dkit/3Dkit.h>
#import "Shape.h"
#import "ComShape.h"
#import "Palette.h"

#define TO_CAMERA 0
#define TO_WORLD  1

@interface Camera:N3DCamera
{
  id  shapeTransform;
  Shape *selectedShape;
  Shape *theBoundingBox;
  BOOL  renderImage;
}

- selectedShape;
- setSelectedShape:shape;
- selectShape:(NXEvent *)theEvent;
- setTheBoundingBox;
- icon;
- (char *)save:shape;
- load:(char *)filename;
- addToPal:sender;
- newModel:sender;
- openModel:sender;
- saveModel:sender;
- saveTIFF:sender;
- (BOOL) renderImage;
- setRenderImage:(BOOL)flag;
@end
```

Camera.m

```
#import <appkit/appkit.h>
#import "Camera.h"
#import "Transform.h"

@implementation Camera
id load (NXTypedStream *ts);

- initFrame:(const NXRect *) theRect
{
  RtPoint fromP = {0,0,-20.0}, toP = {0,0,0};
  id aShader;
  [super initFrame:theRect];

  // initialize camera and put it at (0,0,-20.0) looking at the origin (0,0,0)
  // roll specifies the roll angle of the camera...
  [self setEyeAt:fromP toward:toP roll:0.0];

  // initialize the highlighting bounding box
  theBoundingBox = [[[Shape alloc] init]setDrawAsBox:YES];

  worldShape = nil;
  [self setSurfaceTypeForAll:N3D_SmoothSolids chooseHider:YES];
  [theBoundingBox setSurfaceType:N3D_WireFrame andDescendants:NO];


   aShader=[[N3DShader alloc] init];
   [aShader setUseColor:YES];
   [aShader setColor:NX_COLORGREEN];
   [(N3DShader*) aShader setShader:"matte"];
   [theBoundingBox setShader:aShader];

  return self;
}

- mouseDown:(NXEvent *)theEvent
{
  if (!worldShape)
      return self;
  if (theEvent->data.mouse.click == 2)        /* double-click */
      [self selectShape:theEvent];
  else
      if (selectedShape)
      [shapeTransform rotateShape:theEvent];

  [shapeTransform loadPosition];
  return self;
}

- selectShape:(NXEvent *)theEvent
{
  Shape    *shape;
  NXRect   rect = {{0.0,0.0},{20.0,20.0}};

  rect.origin = theEvent->location;
  rect.origin.x -= 10;
  rect.origin.y -= 10;

  [theBoundingBox unlink];
  shape = selectedShape;
  [self setSelectedShape:
         [[self selectShapesIn:&rect] lastObject]];
```

```
  if (selectedShape && ![selectedShape selected]) {
      [selectedShape  setSelected:YES];
      [self setTheBoundingBox];
      if (shape != selectedShape)
          [shape setSelected:NO];
  }
  else  {
      [shape setSelected:NO];                                          70
      [self setSelectedShape:nil];
  }

  [self display];
  return self;
}

- selectedShape
{
   return selectedShape;                                               80
}

- setSelectedShape:shape
{
  selectedShape = shape;
  return self;
}

- setTheBoundingBox
{                                                                     90
  RtBound     box;
  id          shape;

  if ([selectedShape class] == [ComShape class])
      return self;

  [selectedShape  getBoundingBox:&box];
  [theBoundingBox setBoundingBox:box];

  shape = [selectedShape descendant];                                100
  if (shape)
      [shape linkPeer:theBoundingBox];
  else
      [selectedShape linkDescendant:theBoundingBox];

  return self;
}


- addToPal:sender                                                    110
{
  char *s;
  NXStream *ts;
  id aShape;

  [theBoundingBox unlink];
  aShape = [[ComShape alloc] init];
  [aShape linkDescendant:worldShape];
  s = [self save:aShape];
  [aShape linkDescendant:nil];                                       120
  [aShape free];
  if (!s) return self;
  strcpy(s,rindex(s,'/')+1);
```

```
  [[self delegate] setComponent:s];
  [self icon];
  [self setTheBoundingBox];
  ts = NXMapFile([[self delegate] palette], NX_WRITEONLY);
  NXSeek(ts, 0, NX_FROMEND);
  strrchr(s,'.')[0] = '\0';
  NXPrintf(ts,"\n%s",s);                                                    130
  NXSaveToFile(ts,[[self delegate] palette]);
  NXCloseMemory(ts,NX_FREEBUFFER);

  return self;
}

- icon
{
   int w,h;
                                                                            140
   w = NX_WIDTH(&frame);
   h = NX_HEIGHT(&frame);
   [self sizeTo:100:100];
   [self renderAsTIFF];

   [self sizeTo:w:h];
   return self;
}

- newModel:sender                                                           150
{
   [theBoundingBox unlink];
   [self setSelectedShape:nil];
   [[self setWorldShape:nil]free];
   [self display];
   return self;
}

- saveTIFF:sender
{                                                                           160
  [[self delegate] setComponent:".tiff"];
  [theBoundingBox unlink];
  [self renderAsTIFF];
  [self setTheBoundingBox];
  return self;
}

- saveModel:sender
{
  [theBoundingBox unlink];                                                  170
  [self save:worldShape];
  [self setTheBoundingBox];
  return self;
}

- (char *) save:aShape
{
   static id savePanel=nil;
   NXTypedStream *ts;
   enum ptr link;                                                           180

   savePanel=[SavePanel new];
   [savePanel setRequiredFileType:"mdl"];

   if([savePanel runModal]){
```

```
        ts=NXOpenTypedStreamForFile([savePanel filename], NX_WRITEONLY);
        [(Shape *) aShape save:ts];
        link = terminator;
        NXWriteType(ts,"i",&(link));
        NXCloseTypedStream(ts);
    }
    return (char *)[savePanel filename];
}

- openModel:sender
{
    NXTypedStream *ts=nil;
    static id openPanel=nil;

    static const char *const fileType[2] = {"mdl", NULL};
    openPanel=[OpenPanel new];

    if([openPanel runModalForTypes:fileType]){
        ts=NXOpenTypedStreamForFile([openPanel filename], NX_READONLY);
        [[self setWorldShape:load(ts)]free];
        NXCloseTypedStream(ts);
        [self display];
    }
    return self;
}

- load:(char *)filename
{
    NXTypedStream *ts;

    ts=NXOpenTypedStreamForFile(filename, NX_READONLY);

    if (worldShape)
        [worldShape linkPeer:load(ts)];
    else
        [self setWorldShape:load(ts)];
    NXCloseTypedStream(ts);
    [self display];
    return self;
}


- display
{
    [super display];
    [shapeTransform loadPosition];
    return self;
}

- (int)renderAsTIFF
{
    int i;

    [self setRenderImage:YES];
    i = [super renderAsTIFF];
    [self setRenderImage:NO];
    return i;
}

- (BOOL) renderImage
{
    return renderImage;
```

```
}

- setRenderImage:(BOOL)flag
{
   renderImage = flag;
   return self;
}

@end
```

Shape.h

```
#import <3Dkit/3Dkit.h>

@interface Shape: N3DShape

{
  BOOL selected;
}


enum ptr {descendantlink, nextpeerlink, ancestorlink, terminator};
struct shape_hierarchy {
   Class shapeclass;
   RtMatrix transform;
};

- init;
- setSelected:(BOOL) flag;
- (BOOL) selected;
- save:(NXTypedStream *)ts;
- setBoundingBox:(RtBound) box;

@end
```

Shape.m

```
#import "Shape.h"
#import "ComShape.h"

@implementation Shape:N3DShape

- init
{
  [super init];
  [self setSelectable:YES];
  [self setSurfaceType:N3D_SmoothSolids andDescendants:NO];
  selected = NO;

  return self;
}

- setSelected:(BOOL) flag
{
  selected = flag;
  return self;
}

- (BOOL) selected
{
  return selected;
}

-setBoundingBox:(RtBound) box
{
  N3D_CopyBound(box,boundingBox);
  return self;
}

- save:(NXTypedStream *)ts
{
  struct shape_hierarchy tmp;
  enum ptr link;

  tmp.shapeclass = [self class];
  [self getTransformMatrix:tmp.transform];
  NXWriteType(ts,"#",&(tmp.shapeclass));
  NXWriteArray(ts, "f", 16,tmp.transform);

  if (descendant){
     link = descendantlink;
     NXWriteType(ts,"i",&(link));
     [(Shape *)[self descendant] save:ts];
   }

  if (nextPeer){
     link = nextpeerlink;
     NXWriteType(ts,"i",&(link));
     [(Shape *)nextPeer save:ts];
  }
  else {
     link = ancestorlink;
     NXWriteType(ts,"i",&(link));
  }
  return self;
}

id load (NXTypedStream *ts)
```

```
{
  struct shape_hierarchy tmp;
  static enum ptr link;
  id aShape;

  NXReadType(ts,"s",&(tmp.shapeclass));
  NXReadArray(ts, "f", 16,tmp.transform);
  aShape = [[tmp.shapeclass alloc] init];
  [aShape setTransformMatrix:tmp.transform];                                    70

  NXReadType(ts,"i",&(link));
  if (link == descendantlink)
     [aShape linkDescendant:load(ts)];

  if (link == nextpeerlink)
     [aShape linkPeer:load(ts)];

  if (link == ancestorlink)
     NXReadType(ts,"i",&(link));                                                80

   return aShape;
}

@end
```

ComShape.h

```
#import <3Dkit/3Dkit.h>
#import "Shape.h"

@interface ComShape: Shape
{
}

@end
```

ComShape.m

```
#import "ComShape.h"
#import "bsd/c.h"
#import <ri/ri.h>
#import "Camera.h"

@implementation ComShape: Shape
void transformBox(RtMatrix m, RtBound box);
void boundToPoints(RtBound bound, RtPoint *points);
void pointsToBound(RtPoint *points, RtBound bound);
void unionBox(RtBound box1, RtBound box2, RtBound unionbox);
void drawBox(float L,float R,float D,float U,float F,float N);

- init
{
   [super init];
   [self setSurfaceType:N3D_WireFrame andDescendants:NO];
   return self;
}

- renderSelf:(N3DCamera *)theCamera
{

  RtMatrix m;
  RtBound bound;
  RtColor Green={0,1,0};

  if ([(Camera *)theCamera renderImage])
     return self;

  [self getCompositeTransformMatrix:m relativeToAncestor:nil];
  [self getBoundingBox:&bound];
  transformBox(m,bound);
  N3DInvertMatrix(m,m);

  RiAttributeBegin();
  if ([self selected])
     RiColor(Green);

  RiConcatTransform(m);
  drawBox(bound[0],bound[1],bound[2],bound[3],
          bound[4],bound[5]);
  RiAttributeEnd();

  return self;
}


- getBoundingBox:(RtBound *)box
{
  id shape;
  RtBound bound;
  RtMatrix m;

  shape = [self descendant];
  if ([shape class] == [ComShape class])
     [(ComShape *)shape getBoundingBox:box];
  else
     [(Shape *)shape getBoundingBox:box];
  [shape getTransformMatrix:m];
  transformBox(m,*box);
```

```
  shape = [shape nextPeer];
  while(shape) {
     if ([shape class] == [ComShape class])
        [(ComShape *)shape getBoundingBox:&bound];
     else
        [(Shape *)shape getBoundingBox:&bound];
     [shape getTransformMatrix:m];
     transformBox(m,bound);
     unionBox(bound,*box,*box);                                          70
     shape = [shape nextPeer];
  }
  return self;
}

void transformBox(RtMatrix m, RtBound box)
{
   RtPoint points[8];

   boundToPoints(box,points);                                            80
   N3DMult3DPoints(points,8,m,points);
   pointsToBound(points,box);

   return;
}

void boundToPoints(RtBound bound, RtPoint *points)
{
int i;
                                                                         90
   for (i=0; i<8; i++) {
     points[i][0] = bound[i/4];
     points[i][1] = bound[2+(i/2)%2];
     points[i][2] = bound[4+i%2];
   }
   return;
}

void pointsToBound(RtPoint *points, RtBound bound)
{                                                                       100
int i;

   bound[0] = points[0][0];
   bound[1] = points[0][0];
   bound[2] = points[0][1];
   bound[3] = points[0][1];
   bound[4] = points[0][2];
   bound[5] = points[0][2];

   for (i=1; i<8; i++) {                                                110
       if (points[i][0] < bound[0])
           bound[0] = points[i][0];
       if (points[i][0] > bound[1])
           bound[1] = points[i][0];

       if (points[i][1] < bound[2])
           bound[2] = points[i][1];
       if (points[i][1] > bound[3])
           bound[3] = points[i][1];
                                                                        120
       if (points[i][2] < bound[4])
           bound[4] = points[i][2];
       if (points[i][2] > bound[5])
```

```
            bound[5] = points[i][2];
    }
    return;
}

void unionBox(RtBound box1, RtBound box2, RtBound unionbox)
{
  unionbox[0] = MIN(box1[0],box2[0]);
  unionbox[1] = MAX(box1[1],box2[1]);
  unionbox[2] = MIN(box1[2],box2[2]);
  unionbox[3] = MAX(box1[3],box2[3]);
  unionbox[4] = MIN(box1[4],box2[4]);
  unionbox[5] = MAX(box1[5],box2[5]);
  return;
}

void drawBox(float L,float R,float D,float U,float F,float N)
{
  RtPoint box[6][4]={
      {{L,D,F},{R,D,F},{R,D,N},{L,D,N}},
      {{L,D,F},{L,U,F},{L,U,N},{L,D,N}},
      {{R,U,N},{L,U,N},{L,U,F},{R,U,F}},
      {{R,U,N},{R,U,F},{R,D,F},{R,D,N}},
      {{R,D,F},{R,U,F},{L,U,F},{L,D,F}},
      {{L,U,N},{R,U,N},{R,D,N},{L,D,N}}
   };

  int i;

  for (i=0;i<6;i++)
      RiPolygon(4,RI_P,(RtPointer) box[i],RI_NULL);

   //RiSphere(.1,-.1,.1,360.0,RI_NULL);

}

@end
```

Transform.h

```
#import <appkit/appkit.h>
#import "Camera.h"

@interface Transform:Object
{
    id  rX;
    id  rY;
    id  rZ;
    id  aX;
    id  aY;
    id  aZ;

    id  theRotator;
    id  rotoMatrix;
    int item;
    id  box1;
    id  box2;
    id  camera;
    id  x1;
    id  x2;
    id  y1;
    id  y2;
    id  z1;
    id  z2;
}

- loadInspector;
- loadPosition;
- loadCamera:sender;
- loadShape:sender;
- transform:sender;
- rotateShape:(NXEvent *)theEvent;
@end
```

Transform.m

```
#import "Transform.h"
#define ACTIVEBUTTONMASK (NX_MOUSEUPMASK|NX_MOUSEDRAGGEDMASK)

@implementation Transform

- loadInspector
{
    if (item == 0){
        [x1 setStringValue:""];
        [y1 setStringValue:""];
        [z1 setStringValue:""];
        [x2 setStringValue:""];
        [y2 setStringValue:""];
        [z2 setStringValue:""];
        return self;
    }

    if (item == 1)
        [self loadCamera:self];
    return self;
}

- loadPosition
{
  RtMatrix rM, aM;
  id shape;

  shape = [camera selectedShape];

  if (shape) {
      [shape getTransformMatrix:rM];
      [shape getCompositeTransformMatrix:aM relativeToAncestor:nil];
      [rX setFloatValue: rM[3][0]];
      [rY setFloatValue: rM[3][1]];
      [rZ setFloatValue: rM[3][2]];

      [aX setFloatValue: aM[3][0]];
      [aY setFloatValue: aM[3][1]];
      [aZ setFloatValue: aM[3][2]];
  }

  else {
      [rX setStringValue:""];
      [rY setStringValue:""];
      [rZ setStringValue:""];
      [aX setStringValue:""];
      [aY setStringValue:""];
      [aZ setStringValue:""];
  }
  return self;
}


- loadShape:sender
{
    item = 0;
    [box1 setStringValue:"Translation"];
    [box2 setStringValue:"Scaling"];
    [self loadInspector];
    return self;
}
```

```
- loadCamera:sender
{
    RtPoint eyePoint, viewPoint;
    float a;

    item = 1;
    [box1 setStringValue:"Camera At"] ;
    [box2 setStringValue:"Looking Toward"];                                    70
    [camera getEyeAt:&eyePoint toward:&viewPoint roll:&a];

    [x1 setFloatValue: eyePoint[0]];
    [y1 setFloatValue: eyePoint[1]];
    [z1 setFloatValue: eyePoint[2]];
    [x2 setFloatValue: viewPoint[0]];
    [y2 setFloatValue: viewPoint[1]];
    [z2 setFloatValue: viewPoint[2]];

    return self;                                                               80
}

- transform:sender    //translate and scale
{
    float x,y,z, u,v,w;
    RtPoint eyePoint, viewPoint;
    id shape;

    shape = [camera selectedShape];
    if (item == 0 && !shape) {                                                 90
        [self loadInspector];
        return self;
    }

    if (item == 0) {
        x = y = z = 0.0;
        x = [x1 floatValue];
        y = [y1 floatValue];
        z = [z1 floatValue];
                                                                               100
        u = [x2 floatValue];
        v = [y2 floatValue];
        w = [z2 floatValue];

        if (!u) u = 1;
        if (!v) v = 1;
        if (!w) w = 1;

        [shape translate:x :y :z];
        [shape scale:u :v :w];                                                 110
        [self loadInspector];
    }

    if (item == 1) {
        eyePoint[0] = [x1 floatValue];
        eyePoint[1] = [y1 floatValue];
        eyePoint[2] = [z1 floatValue];
        viewPoint[0] = [x2 floatValue];
        viewPoint[1] = [y2 floatValue];
        viewPoint[2] = [z2 floatValue];                                        120
        [camera setEyeAt:eyePoint toward:viewPoint roll:0];
    }
    [camera display];
```

```
    return self;
}

- rotateShape:(NXEvent *)theEvent
{
  int                    oldMask;
  NXPoint                oldMouse, newMouse, dMouse;
  RtMatrix               rmat, irmat;
  id shape;

  shape = [camera selectedShape];
  if (!theRotator)
      theRotator=[[N3DRotator alloc] initWithCamera:camera];
  // find out what axis of rotation the rotator should be constrained to
  switch([rotoMatrix selectedRow]){
  case 0: [theRotator setRotationAxis:N3D_AllAxes]; break;
  case 1: [theRotator setRotationAxis:N3D_XAxis]; break;
  case 2: [theRotator setRotationAxis:N3D_YAxis]; break;
  case 3: [theRotator setRotationAxis:N3D_ZAxis]; break;
  case 4: [theRotator setRotationAxis:N3D_XYAxes]; break;
  case 5: [theRotator setRotationAxis:N3D_XZAxes]; break;
  case 6: [theRotator setRotationAxis:N3D_YZAxes]; break;
  }

  // track the mouse until a mouseUp event occurs, updating the display
  // as tracking happens.
  [camera lockFocus];
  oldMask = [[camera window] addToEventMask:ACTIVEBUTTONMASK];
  oldMouse = theEvent->location;
  [camera convertPoint:&oldMouse fromView:nil];
  while (1)
  {
    newMouse = theEvent->location;
    [camera convertPoint:&newMouse fromView:nil];
    dMouse.x = newMouse.x - oldMouse.x;
    dMouse.y = newMouse.y - oldMouse.y;
    if (dMouse.x != 0.0 || dMouse.y != 0.0) {
        [theRotator trackMouseFrom:&oldMouse to:&newMouse
        rotationMatrix:rmat andInverse:irmat];
        [shape concatTransformMatrix:rmat premultiply:NO];

        [camera display];
    }
    theEvent = [NXApp getNextEvent:ACTIVEBUTTONMASK];
    if (theEvent->type == NX_MOUSEUP)
      break;
    oldMouse = newMouse;
  }

  [camera display];
  [camera unlockFocus];
  [[camera window] setEventMask:oldMask];
  return self;
}

@end
```

Palette.h

```
#import <appkit/appkit.h>

@interface Palette:Object
{
    id camera;
    id  scrollView,matrix;
    char component[MAXPATHLEN];
    char palette[MAXPATHLEN];
    int count;                                                    10
}

/* delegation methods */
- appDidInit:sender;
- camera:theCamera didRenderStream:(NXStream *)imageStream
  tag:(int)theJob frameNumber:(int)currentFrame;

/* instance methods */
- createScrollingMatrix;
- loadMatrix;                                                     20
- openPalette:sender;
- setComponent:(char *) componentName;
- (char *) palette;


@end
```

Palette.m

```
#import "Palette.h"
#import "Icon.h"
#import "Camera.h"

@implementation Palette

// delegation methods
- appDidInit:sender
{
  // create a matrix and stick it in the scroll view
    [self createScrollingMatrix];

  // bring the window on screen
    [[scrollView window] orderFront:NULL];

    return self;
}

// instance methods

- createScrollingMatrix
{
    NXRect        scrollRect, matrixRect;
    const NXSize interCellSpacing = {10.0, 10.0},cellSize={100,100};

 // set the scrollView's attributes
    [scrollView setBorderType:NX_BEZEL];
    [scrollView setVertScrollerRequired:YES];
  //[scrollView setHorizScrollerRequired:NO];

  // get the scrollView's dimensions
    [scrollView getFrame:&scrollRect];

  // determine the matrix bounds
    [ScrollView getContentSize:&(matrixRect.size)
                forFrameSize:&(scrollRect.size)
                horizScroller:NO
                vertScroller:YES
                borderType:NX_BEZEL];

  // prepare a matrix to go inside our scrollView
    matrix = [[Matrix alloc] initFrame:&matrixRect
                                mode:NX_RADIOMODE
                                cellClass:[Icon class]
                                numRows:0
                                numCols:3];

    [matrix setCellSize:&cellSize];
    [matrix setIntercell:&interCellSpacing];

    strcpy(palette, "default.pal");
    [self loadMatrix];
    [matrix setAutosizeCells:YES];

 // when the user clicks in the matrix and then drags the mouse out of
 // scrollView's contentView, we want the matrix to scroll
    [matrix setAutoscroll:YES];

  // stick the matrix in our scrollView
    [scrollView setDocView:matrix];
```

```
  // set the matrix's single- and double-click actions
    [matrix setTarget:self];
    [matrix setDoubleAction:@selector(load)];

    return self;
}

#define COLS 3                                                          70

- loadMatrix
{
  NXStream *ts;
  char buf[MAXPATHLEN+1];
  int i;

  ts = NXMapFile(palette, NX_READONLY);
  for (i=0; i<3; i++)
      [matrix removeColAt:i andFree:YES];                               80

  [matrix renewRows:0 cols:3];

  count = 0;
  while (1)
  {
    if (NXScanf(ts,"%s",buf) == EOF)
        break;
    if (count % COLS == 0)
        [matrix addRow];                                                90
    [(Icon *) [matrix cellAt:count /COLS :count % COLS] setName:buf];
    count ++;
  }
  NXCloseMemory(ts,NX_FREEBUFFER);
  [matrix sizeToCells];
  [matrix display];
  return self;
}

- openPalette:sender                                                    100
{
  static id openPanel=nil;
  static const char *const fileType[2] = {"pal", NULL};

  openPanel=[OpenPanel new];
  if([openPanel runModalForTypes:fileType]) {
     strcpy(palette, [openPanel filename]);
     [self loadMatrix];
  }
  return self;                                                          110
}

- (char *) palette
{
    return palette;
}

- setComponent:(char *)componentName
{
   strcpy(component, componentName);                                    120
   strrchr(component,'.')[0]='\0';
   return self;
}
```

```
- load
{
    char s[MAXPATHLEN];

    strcpy(s,[(Icon *)[matrix selectedCell] name]);
    strcat(s,".mdl");                                                    130
    [camera load:s];
    return self;
}

- camera:theCamera didRenderStream:(NXStream *)imageStream
  tag:(int)theJob frameNumber:(int)currentFrame
{
    static id savePanel=nil;

   if (*component){                                                      140
       NXSaveToFile(imageStream, strcat(component,".tiff"));
       strrchr(component,'.')[0]='\0';
        if (count % COLS == 0)
            [matrix addRow];
        [matrix sizeToCells];

        [matrix drawCell:[(Icon *)
        [matrix cellAt:count /COLS :count % COLS]setName:component]];

        count ++;                                                        150
    }
    else {
        savePanel=[SavePanel new];
        [savePanel setRequiredFileType:"tiff"];
        if([savePanel runModal])
            NXSaveToFile(imageStream, [savePanel filename]);
    }
    return self;
}
                                                                         160

@end
```

Icon.h

```
#import <appkit/Cell.h>
#define MAXNAMELEN 20

@interface Icon:Cell
{
char name[MAXNAMELEN];
}

/* instance methods */
- (char *)name;
- setName:(char *) cellName;
- drawInside:(const NXRect *)cellFrame inView:controlView;
@end
```

Icon.m

```
#import <appkit/appkit.h>
#import "Icon.h"

@implementation Icon

- drawInside:(const NXRect *)cellFrame inView:controlView
{
    static id sharedTextCell = nil;
    id image;
    NXRect rect = *cellFrame;                                                    10
    NXPoint imageOrigin;

  // erase the cell
    PSsetgray((cFlags1.state || cFlags1.highlighted) ? NX_WHITE : NX_LTGRAY);
    NXRectFill(cellFrame);

    image = [[NXImage alloc] initFromFile:strcat(name,".tiff")];
    strrchr(name,'.')[0]='\0';
    imageOrigin.x = NX_X(cellFrame) ;
    imageOrigin.y = NX_Y(cellFrame)+ NX_HEIGHT(cellFrame) ;                      20
    [image composite:NX_SOVER toPoint:&imageOrigin];

    if (!sharedTextCell)
        sharedTextCell = [[Cell alloc] init];

    [sharedTextCell setStringValue:name];
    [sharedTextCell setAlignment:NX_CENTERED];

    NX_Y(&rect) += 85;
    [sharedTextCell drawInside:&rect inView:controlView];                        30
    return self;
}

- (char *)name
{
  return name;
}

- setName:(char *)cellName
{                                                                                40
   strcpy(name, cellName);
   return self;
}

@end
```

Lens.h

```
#import <3Dkit/3Dkit.h>
#import "Shape.h"

@interface Lens : Shape

- renderSelf:(RtToken)context;
- init;
@end
```

Lens.m

```
#import "Lens.h"
#import <ri/ri.h>

@implementation Lens:Shape

- renderSelf:(RtToken)context
{
    RiAttributeBegin();
    RiRotate(90,0,1,0);
    RiTranslate(0,0,.7);
    RiSphere(1,-1,-.7,360.0,RI_NULL);
    RiAttributeEnd();

    RiAttributeBegin();
    RiRotate(90,0,1,0);
    RiTranslate(0,0,-.7);
    RiSphere(1,.7,1,360.0,RI_NULL);
    RiAttributeEnd();
    return self;
}

- init
{
   [super init];
   boundingBox[0]=-.3; boundingBox[1]=.3;
   boundingBox[2]=-1; boundingBox[3]=1;
   boundingBox[4]=-1; boundingBox[5]=1;
   return self;
}

@end
```

Prism.h

```
#import <3Dkit/3Dkit.h>
#import "Shape.h"

@interface Prism : Shape

- renderSelf:(RtToken)context;
- init;
@end
```

Prism.m

```
#import "Prism.h"
#import <ri/ri.h>

@implementation Prism:Shape
void drawPrism(float L,float R,float D,float U,float F,float N);

- renderSelf:(RtToken)context
{
   drawPrism(boundingBox[0],boundingBox[1],boundingBox[2],boundingBox[3],
         boundingBox[4],boundingBox[5]);                                    10
   return self;
}

- init
{
  [super init];
  boundingBox[0]=-.7; boundingBox[1]=.7;
  boundingBox[2]=-1; boundingBox[3]=1;
  boundingBox[4]=-.7; boundingBox[5]=.7;
  return self;                                                              20
}

@end


void drawPrism(float L,float R,float D,float U,float F,float N)
{
  RtPoint faces1[2][3]={
     {{L,U,N},{L,D,N},{R,D,N}},                                             30
     {{L,U,F},{L,D,F},{R,D,F}}
  },
  faces2[3][4]={
        {{L,D,N},{R,D,N},{R,D,F},{L,D,F}},
        {{L,U,N},{R,D,N},{R,D,F},{L,U,F}},
        {{L,U,N},{L,D,N},{L,D,F},{L,U,F}}
  };
  int i;

  for (i=0;i<2;i++)                                                         40
     RiPolygon(3,RI_P,(RtPointer) faces1[i],RI_NULL);

  for (i=0;i<3;i++)
     RiPolygon(4,RI_P,(RtPointer) faces2[i],RI_NULL);

}
```

SemiLens.h

```
#import <3Dkit/3Dkit.h>
#import "Shape.h"

@interface SemiLens : Shape

- renderSelf:(RtToken)context;
- init;
@end
```

SemiLens.m

```
#import "SemiLens.h"
#import <ri/ri.h>

@implementation SemiLens:Shape

- renderSelf:(RtToken)context
{
    RiAttributeBegin();
    RiRotate(90,0,1,0);
    RiTranslate(0,0,.8);
    RiSphere(1,-1,-.6,360.0,RI_NULL);
    RiAttributeEnd();
    return self;
}

- init
{
  [super init];
  boundingBox[0]=-.2; boundingBox[1]=.2;
  boundingBox[2]=-1; boundingBox[3]=1;
  boundingBox[4]=-1; boundingBox[5]=1;
  return self;
}

@end
```

Makefile

```
#
# Generated by the NeXT Project Builder.
#
# NOTE: Do NOT change this file -- Project Builder maintains it.
#
# Put all of your customizations in files called Makefile.preamble
# and Makefile.postamble (both optional), and Makefile will include them.
#

NAME = 3DGraphics

PROJECTVERSION = 1.1
LANGUAGE = English

APPICON = first.tiff
LOCAL_RESOURCES = 3DGraphics.nib

GLOBAL_RESOURCES = first.tiff

CLASSES = Camera.m ComShape.m Icon.m Lens.m Palette.m Prism.m SemiLens.m \
          Shape.m Transform.m

HFILES = Camera.h ComShape.h Icon.h Lens.h Palette.h Prism.h SemiLens.h \
         Shape.h Transform.h

MFILES = 3DGraphics_main.m

OTHERSRCS = Makefile


MAKEFILEDIR = /NextDeveloper/Makefiles/app
INSTALLDIR = $(HOME)/Apps
INSTALLFLAGS = -c -s -m 755
SOURCEMODE = 444

ICONSECTIONS = -sectcreate _ICON app first.tiff

LIBS = -lMedia_s -lNeXT_s
DEBUG_LIBS = $(LIBS)
PROF_LIBS = $(LIBS)


-include Makefile.preamble

include $(MAKEFILEDIR)/app.make

-include Makefile.postamble

-include Makefile.dependencies
```

Bibliography

[Art89] P. Arturo. Specification of spatial integrity constraints in pictorial databases. *COMPUTER*, pages 59–71, Dec 1989.

[Bag85] R. Bagel. Gambit: An interactive database design tool for data structures, integrity constraints and transactions. *IEEE Trans. Software Engineering*, 11(7):574–583, July 1985.

[EK92] P. Egbert and W. Kubitz. Application graphics modeling support through object orientation. *COMPUTER*, pages 84–90, Oct 1992.

[Flo91] J. Florentin. *Object-oriented Programming Systems.* Chapman and Hall, 1991.

[FV90] D. Foley and A. VanDam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1990.

[Kli89] B. Kliewer. Hoops: Powerful portable 3-d graphics. *BYTE*, pages 193–194, July 1989.

[Kub91] Kubota Pacific Computer Inc. *Dore Reference manual*, 1991.

[KW87] A. Kemper and M. Wallrath. An analysis of geometric modeling in database systems. *ACM Computing Surveys*, 19(1):47–91, March 1987.

[Lel88] W. Leler. *Constraint Programming Languages.* Addison-Wesley, 1988.

[McA91] A. McAulay. *Optical Computer Architectures.* Wiley-Interscience, 1991.

[NeX92] NeXT Computer Inc. *NEXT DEVELOPER DOCUMENTATION*, 1992.

[Req80] A. Requicha. Representation for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–463, Dec 1980.

[SS79] G. Steele and G. Sussman. Constraints. In *APL conference proceedings part1*, pages 208–225, June 1979.

[Sut63] I. Sutherland. Sketchpad: A man-machine graphical communication system. In *IFIPS Proceedings of the Spring Joint Computer Conference*, January 1963.

[TP91] T. Tse and L. Pong. An examination of requirements specification languages. *THE COMPUTER JOURNAL*, 34(2):143–152, 1991.

[Ups90a] S. Upstill. Graphics go 3-d. *BYTE*, pages 255–256, Dec 1990.

[Ups90b] S. Upstill. *The RenderMan Companion.* Addison-Wesley, 1990.

[Weg92] P. Wegner. Dimensions of objected-oriented modeling. *COMPUTER*, pages 12–21, Oct 1992.

[Wis90] P. Wisskirchen. *Object-Oriented Graphics.* Springer-Verlag, 1990.

[Zlo78] M. Zloof. Security and integrity within the query-by-eaxmple data base management language. Technical report, IBM, 1978.

Vita

- Nabeel Mohammed Adnan Al-Mouslli
- Born in 1967 in Saudi Arabia and grown up in Syria.
- Received the Bachelor of Science degree in Computer Science in 1990 from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.
- Started working in the industry right away after graduation. Started working in applications development and later switched to systems administration. At the same time, was studying for the Master in Computer Science as a part-timer.
- Currently, is working in as a support engineer for systems management products marketed by a reputable company in Saudi Arabia.
- Expecting to receive the Master of Science degree in Computer Science in 1995 from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.