To the Graduate Council:

I am submitting herewith a dissertation written by William Joseph Rosener entitled "Integrating Multi-way and Structural Constraints into Spreadsheet Programming". I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Brad Vander Zanden

Bradley Vander Zanden, Major Professor

We have read this dissertation
and recommend its acceptance:

Richard G Thomason

Bruce McLaren

Heather Booth

Wayne Claycomb

Accepted for the Council:

Lev Minkel

Associate Vice Chancellor

and Dean of the Graduate School

Integrating Multi-way and Structural Constraints into Spreadsheet Programming

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

William Joseph Rosener

August, 1994

ACKNOWLEDGEMENTS

I would like to begin by thanking Linda. She worked the P-Z line for grad students at registration during my entire stay at UT. Although, we only saw each other once a semester, not once did she forget my name after my first registration at UT. While some people dread the thought of registration, Linda made me look forward to it. With people like her at University of Tennessee, I begin my thanks.

First of all, I am especially grateful to Dr. David Straight for providing me with a graduate teaching assistantship in the computer science department. I am also thankful for Tom Garritano for giving me a graduate research assistantship with the Office of Research Administration. Working for this office not only gave me valuable work experience, but through this job I also met a great friend Jodie Wehr. During my last year of graduate studies, Jodie gave me the continuous support, encouragement, and love to finish.

I would also like to thank the members of my committee: Dr. Heather Booth, Dr. Wayne Claycombe, Dr. Bruce MacLennan, and Dr. Michael Thomason. Their comments improved it considerably.

I had a great deal of fun while I was in Knoxville, and established many personal friendships over the years that were all a part of my graduate experience. I would like to begin by thanking John Rose and Mark Costanzo the founders of Budnet. John provided me with continuous friendship and entertainment throughout my graduate studies. I would also like to thank Derek Martin for re-teaching me how to play. A special thanks goes out to Subhendu Ghatak, Gavin O'Brien, and Jim Plank.

I would also like to thank my parents Frank and Ruby, my brothers and sisters Dan, Bob, and Ann, for their love and the good times we have shared. More specifically, I would like to thank Mom for making me feel so special, and her continuous questioning as to when I would finish, and Dad for teaching me how to work, and for his endless financial help.

I would also like to thank Glen Charles, Les Charles, James Burrows and the cast on *Cheers* for providing me with many hours of laughter and for giving me that needed break from all my worries. I guess the theme song for this show describes best what *Cheers* was to me.

> Making your way in the world today
> Takes everything you got.
> Taking a break from all your worries
> Sure would mean a lot.

Finally, I would like to express my utmost gratitude to my advisor Dr. Brad Vander Zanden. I greatly appreciate the tremendous effort and invaluable advice he has given to me throughout my work on this dissertation. He has made my studies both an enjoyable and a learning experience. He has taught me how to do research, and has had a profound influence on my professional development. For Dr. Vander Zanden's sincere interest in me as a graduate student and as a person, I am honored and deeply grateful.

ABSTRACT

Writing interactive graphical user interfaces has traditionally been a very time consuming and difficult process. To speed up the development process, many modern user interface development environments are using constraints to specify the graphical layout and behavior of objects. Constraints are relationships between objects that are defined once and then automatically maintained by the system.

Most graphical user interface toolkits support single-output, one-way constraints. We have investigated how multi-output constraints, structural constraints, and multi-way constraints can be used to enhance the power and flexibility of the spreadsheet programming paradigm. These new constraints increase efficiency, allow more powerful relationships to be expressed, and facilitate maintaining consistency between data structures.

Multi-output constraints can result in greater efficiency by allowing the reuse of computations and reducing the number of formula objects maintained by the system. Structural constraints allow the user to define relationships between the structure of objects. These constraints make it possible for a constraint to add, delete, or modify a structure. The ability to define structural relationships and have a constraint solver automatically modify objects to maintain these relationships can lead to a further reduction in the effort required to construct a graphical interface. Multi-way constraints allow any of the objects in the constraint to be changed in order for it to be satisfied. These constraints can greatly simplify maintaining consistency between application data and its display, and maintaining consistency between multiple views, because they allow information to flow both ways through a constraint.

Finally, as a test of our extensions to the spreadsheet programming model, we have developed and implemented an object-oriented toolkit for creating highly interactive, direct manipulation, 3-dimensional user interfaces. This toolkit is an important contribution in itself for it incorporates many new features. The toolkit is divided into two layers: a programming layer and a direct manipulation layer. The programming layer supports struc-

tured graphics, constraints, a high-level event handling model, and a hierarchical facility for building objects from subparts. The direct manipulation layer allows the user to interactively create and directly apply transformations to 3D objects, construct polyhedron objects, change the viewing parameters for drawing 3D objects, and build structured graphics. The toolkit has experimentally been used in a graphics course and has substantially reduced the amount of time required to create applications involving the direct manipulation of 3D objects.

Contents

Part I INTRODUCTION AND BACKGROUND 1

1 Introduction 2

1.1 Overview 2

1.2 The Motivation for Constraints 4

1.3 Advantages of the Extended Spreadsheet Model 7

1.4 3D Toolkit 9

1.5 Dissertation Overview 10

2 Terminology and Related Work 12

2.1 Introduction 12

2.2 Definitions 12

2.3 One-way Constraints 14

2.4 Multi-output Constraints 16

2.5 Structural Constraints 17

2.6 Multi-way Constraints 19

2.7 Constructing Interactive 3D Graphics 21

Part II CONSTRAINTS 23

3 Extensions to the Spreadsheet Model 24

3.1 Introduction 24

3.2 Multi-output Constraints 24
3.3 Structural Constraints 31
3.4 Multi-way Constraints 36
3.5 Expressiveness of the Extended Spreadsheet Model 42

4 Constraint Solving 45

4.1 Introduction 45
4.2 Solving Multi-output Constraints 47
4.2.1 Data Structures 52
4.2.2 The Constraint Solving Algorithm 53
4.2.3 An Example 58
4.2.4 Time Complexity 61
4.2.5 Proof of Correctness 65
4.3 Solving Structural Constraints 69
4.3.1 Data Structures 69
4.3.2 The Constraint Solving Algorithm 70
4.3.3 An Example 73
4.3.4 Time Complexity 76
4.3.5 Proof of Correctness 78
4.4 Solving Multi-way Constraints 82
4.4.1 Data Structures for Multi-way Constraints 84
4.4.2 The Constraint Solving Algorithm 84
4.4.3 An Example 92
4.4.4 Time Complexity 95
4.4.5 Proof of Correctness 96
4.5 Combination of the Three Algorithms 100
4.5.1 Proof of Correctness 101
4.6 Performance 106

4.6.1 Multi-output Constraints 106
4.6.2 Structural Constraints 109
4.6.3 Multi-way Constraints 110

Part III AN OBJECT-ORIENTED 3D TOOLKIT 113

5 Introduction to Viewing in 3D 114
5.1 Projections 115
5.1.1 Perspective Projection 115
5.1.2 Parallel Projection 116
5.2 Specifying the 3D View Volume 116
5.3 3D Transformations 119
5.4 Clipping 122

6 Programming Layer 124
6.1 Introduction 124
6.2 3D Objects 124
6.3 3D Aggregates 129
6.4 3D Update Algorithm 131
6.5 Integration of Constraints 134
6.6 3D Interactors 135
6.6.1 3D-Move-Interactor 137
6.6.2 3D-Grow-Interactor 138
6.6.3 3D-Rotate-Interactor 140
6.6.4 3D-Selection-Interactor 141
6.6.5 3D-Reposition-Interactor 142
6.6.6 3D-Create-Interactor 142

7 Direct Manipulation Layer 145

7.1 Introduction 145

7.2 Scene Editor 145

7.2.1 Scene Window 146

7.2.2 Scene Control Window 148

7.3 Polyhedron Editor 157

7.4 Aggregate Editor 159

7.5 Integration of the Extended Constraints 162

8 Conclusions and Directions for Future Research 164

8.1 Conclusions 164

8.2 Future Work 166

Bibliography 170

Appendices 177

A Reference Manual for the Extended Constraints 178

A.1 Introduction 178

A.2 Multi-output Constraints 179

A.2.1 Fixed number of outputs 179

A.2.2 Dynamic number of outputs 180

A.3 Structural Constraints 181

A.4 Multi-way Constraints 183

B 3D Reference Manual: Programming Layer 188

B.1 Introduction 188

B.2 Graphic Qualities 188

B.2.1 Filling Style 188

B.2.2 Line Style 189

B.2.3 Wire Frame versus Shading 189
B.3 Predefined 3D Graphical Objects 189
B.3.1 Cube 191
B.3.2 3D-Lines 192
B.3.3 Planes 192
B.3.4 Cones 193
B.3.5 Cylinders 194
B.3.6 Sphere 196
B.3.7 Ellipsoid 197
B.3.8 Polyhedron 198
B.4 Interactively Creating New Colors 200
B.5 Printing 3D Windows 201
B.6 Geometrical Transformations 201
B.6.1 Translation 202
B.6.2 Rotation 202
B.6.3 Scale 202
B.7 Viewing Parameters 203
B.7.1 Parallel versus Perspective Projection 204
B.7.2 View Reference Point 204
B.7.3 View Plane Normal and View Up Vector 204
B.7.4 Projection Reference Point 205
B.7.5 Window Size 205
B.8 Clipping 205
B.9 3D Constraints 206
B.10 3D Aggregates 207
B.10.1 Creating a 3D Aggregate 208
B.10.2 Ungrouping the Objects in a 3D Aggregate 209

B.11 3D Interactors 209
B.11.1 3D-Move-Interactor 211
B.11.2 3D-Grow-Interactor 212
B.11.3 3D-Rotate-Interactor 213
B.11.4 3D-Selection-Interactor 213
B.11.5 3D-Reposition-Interactor 214
B.11.6 3D-Create-Interactor 214

C 3D Reference Manual: Direct Manipulation Layer **216**
C.1 Scene Editor 216
C.1.1 Scene Window 217
C.1.2 Scene Control Window 219
C.2 Polyhedron Editor 229
C.3 Aggregate Editor 234

D Software Development Process and Testing **237**
D.1 Introduction 237
D.2 Specifications and Design 237
D.3 Testing 239

Vita **240**

List of Tables

4.1 Time to solve multi-output constraints 107

4.2 Time to solve multi-output constraints 108

4.3 Time to solve two types of structural constraints 110

4.4 Time to solve two types of multi-way constraints 111

5.1 Parameters associated with each window 116

List of Figures

1.1 Applications of constraints 5

1.2 Using three views of the same scene 8

1.3 Structural constraint used for selection handles 8

2.1 Variable sized input lists 15

2.2 Pointer variables 16

3.1 Single and multi-output constraints 25

3.2 Pictorial description of a multi-output constraint 28

3.3 Positioning objects in a list using a multi-output constraint 30

3.4 Structural constraint used for selection handles 35

3.5 Structural constraint on an arrow 36

3.6 Multi-way constraints used to implement multiple views of time 37

3.7 Activation condition on a multi-way constraint 39

3.8 Visual color editor 40

3.9 Multi-way constraint on an aggregate and its children 43

4.1 Mark/sweep approach 46

4.2 Complication with multi-output constraints 49

4.3 Three scenarios with multi-output constraints 51

4.4 Pictorial description of a multi-output constraint 53

4.5 Diagram of the invalidate routine used for multi-output constraints 55

4.6 Pseudo code of the invalidate routine used for multi-output constraints . . . 56
4.7 Recursive invalidate routine used for multi-output constraints 57
4.8 Evaluate multi-output formula routine 57
4.9 Get value routine 58
4.10 Example of multi-output constraints 59
4.11 Updated constraint graph 60
4.12 Final constraint graph 62
4.13 Constraints that illustrate the definition of affected 63
4.14 Algorithm for solving structural constraints 71
4.15 Execute queued action-procedures routine 71
4.16 Routine for updating structural constraints 72
4.17 Slot has been invalidated routine 72
4.18 Example of structural constraints 73
4.19 An example of the planning phase 82
4.20 Invalidate routine used for multi-way constraints 88
4.21 Multi-way invalidate constraint routine 89
4.22 Recursively multi-way invalidate constraint routine 90
4.23 Evaluate multi-way formula routine 91
4.24 Get value routine 91
4.25 Example of a multi-way constraint 92
4.26 Initial values for example multi-way constraint 93
4.27 Example multi-way constraint with dependencies determined 93
4.28 Result of the invalidate routine 94
4.29 Multi-output verses single-output constraints 108
4.30 Multi-output verses single-output constraints 109
4.31 Time performance for structural constraints 111
4.32 Time performance for multi-way constraints 112

5.1 Perspective and parallel projections 115
5.2 Defining the view plane and window 117
5.3 Perspective view volume 118
5.4 Parallel view volume 118
5.5 Scaling without translation 120
5.6 Scaling with translation 120
5.7 Rotation without translation 120
5.8 Rotation with translation 121
5.9 Types of coordinate systems 122
5.10 Clipping in two dimensions 122
5.11 Clipping planes 123
5.12 Clipping in three dimensions 123

6.1 Available 3D objects in our toolkit 125
6.2 Varying the number of polygons used to represent a cylinder 126
6.3 Fast-draw objects 127
6.4 3D aggregate object representing two wheels and an axle 130
6.5 Structured object composed of three primitive objects 130
6.6 Polygons that are not guaranteed to be shaded correctly 133
6.7 Effect of using the 3D-move-interactor 138
6.8 Effect of using the 3D-grow-interactor 139
6.9 Effect of using the 3D-rotate-interactor 140
6.10 Effect of using the 3D-reposition-interactor 142
6.11 An example illustrating the 3D-create-interactor 143

7.1 The scene window 146
7.2 The scene control window 148
7.3 The viewing control window 151

7.4 The view plane window 152
7.5 The viewing reference window 153
7.6 The viewing reference window after the VUP has been rotated 154
7.7 The perspective reference point window 154
7.8 The change center of rotation window 155
7.9 The polyhedron window 157
7.10 The polyhedron control window 158
7.11 The visual color editor for the polyhedron window 159
7.12 The aggregate window 160
7.13 The aggregate control window 160

8.1 Prototype of a 3D box constraint window 168
8.2 Prototype of the event window for interactors 169

B.1 Polygons that are not guaranteed to be shaded correctly 190
B.2 The two types of coordinate systems 191
B.3 A description of a cube 191
B.4 A description of a 3d-line 192
B.5 The description of a plane 193
B.6 The two types of cones 194
B.7 The two type of cylinders 195
B.8 The description of the regular sphere 196
B.9 An example of a fast sphere 197
B.10 The description of an ellipsoid 198
B.11 An example of a polyhedron 199
B.12 Visual color editor 200
B.13 The default viewing situation 203
B.14 Perspective vs parallel projection 204

B.15 An example of clipping 205
B.16 Effect of using the 3D-move-interactor 211
B.17 Effect of using the 3D-grow-interactor 212
B.18 Effect of using the 3D-rotate-interactor 213
B.19 Effect of using the 3D-reposition-interactor 214
B.20 An example illustrating the 3D-create-interactor 215

C.1 The scene window 217
C.2 The scene control window 220
C.3 The viewing control window 223
C.4 The view plane window 224
C.5 The viewing reference window 225
C.6 The viewing reference window after the VUP has been rotated 225
C.7 The perspective reference point window 226
C.8 The change center of rotation window 228
C.9 The polyhedron window 229
C.10 The polyhedron control window 230
C.11 The visual color editor for the polyhedron window 231
C.12 The aggregate window 234
C.13 The aggregate control window 235

D.1 The software development process 238
D.2 Specification of a 3D polyhedron 238

Part I

INTRODUCTION AND BACKGROUND

Introduction ..... Chapter 1

Related Work ..... Chapter 2

Chapter 1

Introduction

1.1 Overview

Writing interactive graphical user interfaces has traditionally been a very time consuming and difficult process. To speed up the development process, many modern user interface development environments are using constraints to specify the graphical layout and behavior of objects. Constraints are relationships between objects that are defined once and then automatically maintained by the system. Examples of systems that use constraints include: Grow [Barth 86], Animus [Duisberg 86], ThingLab II [Maloney 91], Rendezvous [Hill 92], Kaleidoscope [Freeman 92], CONDOR [Kass 92], and Garnet [Myers 90a]. In addition to defining the graphical layout and behavior of objects, constraints can be used to display feedback for input operations, maintain consistency between application data and its display, and maintain consistency between multiple views. In graphical user interfaces that support constraints, the user is freed from the tedious and error-prone task of maintaining these relationships. Thus constraints can reduce substantially the time required to create a graphical user interface.

Most graphical user interface toolkits support single-output, one-way, domain-independent constraints. Domain-independent means that a constraint can express a relationship among variables of arbitrary types. This dissertation investigates how multi-output constraints, structural constraints, and multi-way constraints can be used to enhance the

power and flexibility of the spreadsheet programming paradigm. The spreadsheet model allows a programmer to declaratively describe relationships between a program's entities using constraints. These new constraints increase efficiency, allow more powerful relationships to be expressed, and facilitate maintaining consistency between data structures.

Our first extension is to support multi-output constraints, which allow a constraint to return more than one value. Multi-output constraints can result in greater efficiency by allowing the reuse of computations and reducing the number of formula objects maintained by the system. They also allow a programmer to control the granularity of the constraints by determining the amount of information that is computed.

The second extension is support for structural constraints. Current constraint systems only allow the non-structural properties of an object to be constrained. Structural constraints allow the user to define relationships between the structure of objects. For example, structural constraints make it possible for a constraint to add, delete, or modify a structure. The ability to declaratively define relationships among the structure of objects and have a constraint solver automatically modify objects to maintain these relationships should further reduce the effort required to construct a graphical interface by decreasing the amount of code that must be written by a programmer.

The third extension is multi-way constraints. With increasingly complex user interfaces, it is becoming clear that one-way constraints are too restrictive. One-way constraints allow only one of the objects in the constraint to be changed in order to satisfy the constraint. However, multi-way constraints allow any of the objects in the constraint to be changed in order for it to be satisfied. Multi-way constraints can greatly simplify maintaining consistency between application data and its display, and maintaining consistency between multiple views, because they allow information to flow both ways through a constraint.

The extensions to the spreadsheet model were made in Garnet, a user interface development environment that currently supports single-output, one-way constraints. As a test of the new model, Garnet has been extended to handle 3D graphics and a multiple-view direct

manipulation tool has been developed to assist users in creating 3D interfaces. The extension of Garnet to support 3D graphics is an important contribution in itself. Traditionally, creating 3D user interfaces has been a very time consuming and difficult process. The software that is available is usually very low-level with little or no support for 3D interaction. Furthermore, this software has usually been developed using an imperative programming environment. Our 3D extension takes a declarative object-oriented approach. It supports high-level features that have typically been ignored in 3D toolkits, such as direct manipulation, aggregates of objects, and constraints. Our extension includes a comprehensive set of 3D tools that makes it significantly easier to create graphical, highly-interactive, direct manipulation 3D user interfaces.

1.2 The Motivation for Constraints

Constraints make it easier to construct graphical user interfaces in a number of ways. First, they can naturally express many relationships that arise in graphical interfaces. Such relationships include:

Maintaining consistency between graphics and data. Constraints allow graphical properties of objects to derive their values from application variables. For example, in Figure 1.1 (a), the graphics variable *bar.height* is constrained to the application variable *stereo.volume*. If the value of *stereo.volume* changes, then the height of the bar automatically changes as well.

Maintaining consistency between multiple views of data. Graphical interfaces often provide multiple views of the same data. When the user modifies the data by manipulating the graphical objects in one of the views, the other views must be updated to reflect the changed data. Constraints can be used to reflect the relationship between the graphical objects in each view and the data. Thus when the data changes, the constraints associated with the graphical objects in each view cause the graphical objects

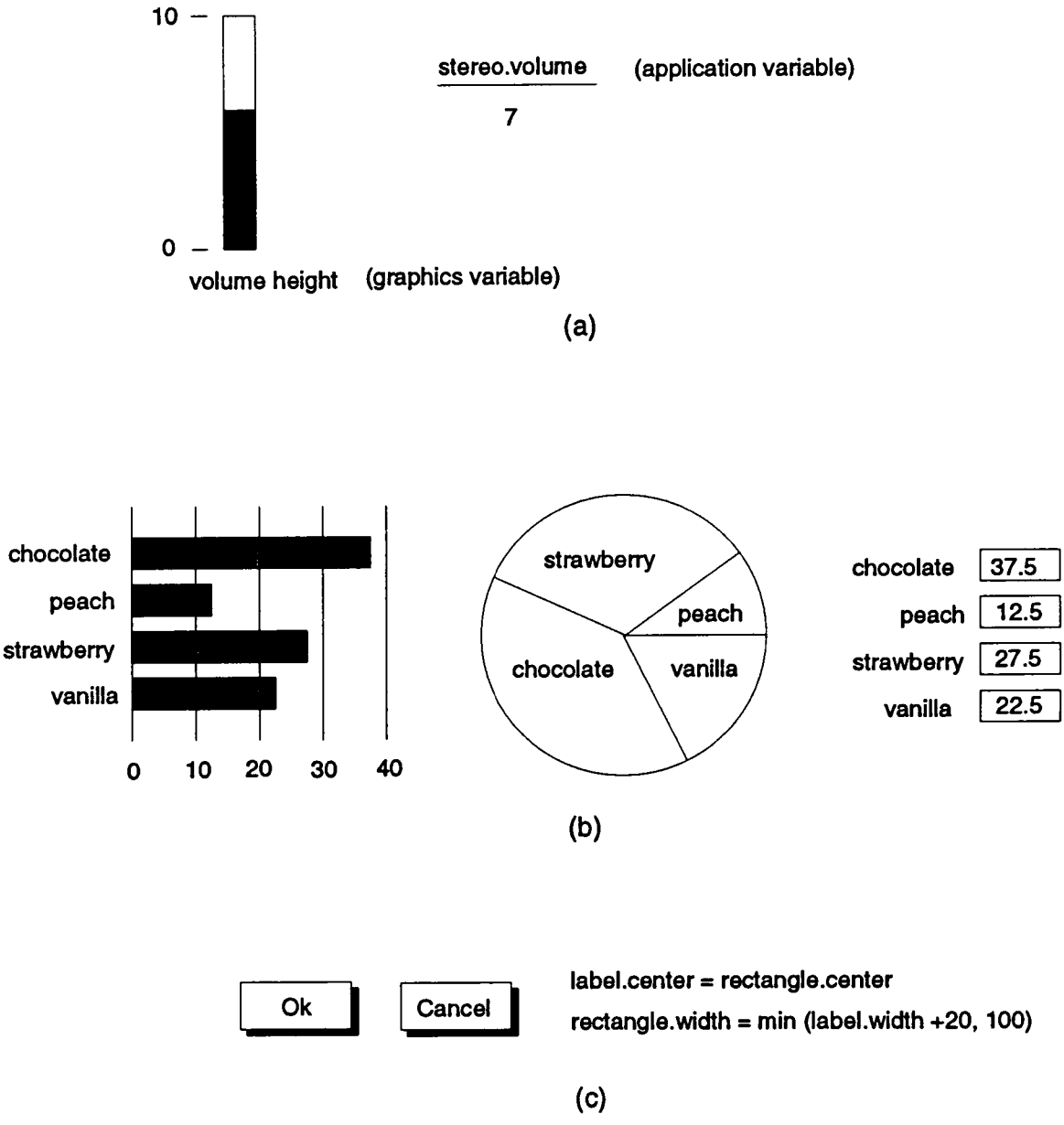


Figure 1.1: Examples of uses of constraints. A graphical property, the height of a rectangle, is constrained to an application variable (a). Consistency between multiple views of the same data is maintained using constraints (b). A text string is centered within a labeled box using constraints (c).

to be automatically updated. For example in Figure 1.1 (b), constraints are used to maintain consistency between the three views of ice cream flavors. If the value of the labeled box *chocolate* is changed, then the bar chart and pie chart are also changed.

Specify graphical layout. Constraints can be used to specify the alignment, placement, and size of graphical objects. For example, in Figure 1.1 (c) constraints are used to center the text strings within the labeled boxes.

Object behavior. Constraints can specify how an object's behavior is related to variables (or stimuli) in a system. For example, in a physical system, the fluid flow may be constrained by the viscosity of the fluid and the valves that are open.

The second advantage of constraints is that they are declarative. Constraints allow relationships to be stated without specifying how they are satisfied. Once the constraints have been created, a constraint solver automatically propagates changed values to keep the constraints satisfied.

A third advantage of constraints is that they promote modularity. Constraints allow portions of the program to be self-contained so that if the constraint is removed from the system, only the relationship specified by the constraint is disabled. In conventional object systems, information about an object's behavior must be explicitly stored with objects that influence the object's behavior. This dispersal of information violates the principle of modularity, because information about an object's behavior cannot be obtained by looking solely at the object's implementation. One ramification of this violation is that the programmer must explicitly maintain information about which other objects can influence this object's behavior. In contrast, constraints allow all the information about an object's behavior to be stored in its implementation, thus conforming to the principle of modularity. Conforming to this principle makes it much easier to unplug the object from the system, since information about the object's behavior is not distributed among the other application objects.

For example, suppose we wish to constrain object A to be to the right of object B. In a conventional system, the programmer incorporates a method into B that automatically calls A's move method whenever B is moved. Later, if A is destroyed this incorporated method must be be removed from B. However, using constraints all the information is kept in A. Hence, if A is destroyed then the methods in B do not have to be modified.

Finally, the fourth advantage of constraints is that they provide automatic sequencing. A constraint solver ensures that the constraints are satisfied in the correct topological order, so that a constraint is never executed until all of its inputs are known. Thus, the user does not need to worry about specifying the proper sequence. Chapter 4 goes into more detail on how constraints provide automatic sequencing.

1.3 Advantages of the Extended Spreadsheet Model

The extended spreadsheet model not only makes it easier and more efficient to implement relationships, but it also allows more powerful relationships to be expressed. Multi-output constraints can result in greater efficiency by allowing the reuse of computations and reducing the number of formulas that must be maintained by the system. The additional formulas required by the single-output constraints of the conventional model not only consume more memory but also force the constraint solver to perform extra bookkeeping operations, such as keeping track of dependencies between the constraints. As an example, multi-output constraints could have been used in Figure 1.2. In this example, three variables (i.e., x, y, and z) are used in each view to represent the location of the house. Multi-output constraints could be used to help transmit and unpack these 3-tuples of information among the three views.

Structural constraints allow programmers to declaratively express structural relationships among objects, and to rely on a constraint solver to automatically satisfy them in the appropriate order. Structural modifications often have ramifications that need to be propagated, just as modifications to attributes have ramifications As an example, struc-

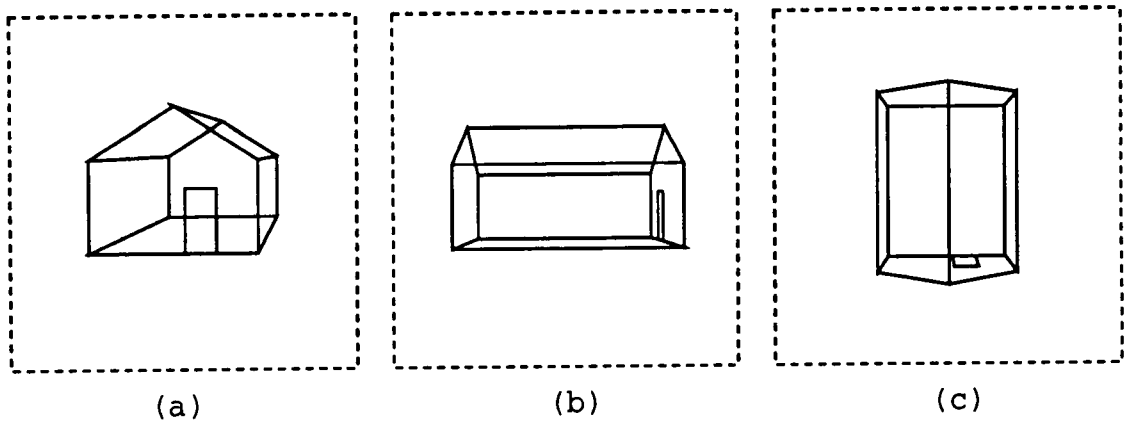


Figure 1.2: Multi-output and multi-way constraints can be used to maintain consistency between the front (a), side (b), and top (c) views of the house. If the user applies a transformation to the house in one view, then the other two views are redrawn automatically.

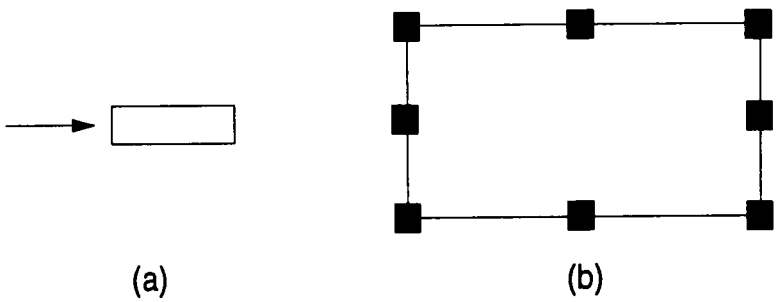


Figure 1.3: The type of selection handle should depend on the size of the object. Smaller objects that cannot accommodate the feedback boxes should be pointed at by a selection arrow (a), larger objects should be surrounded by feedback boxes (b).

tural constraints could be used to specify the type of selection handle that should surround a selected object as shown in Figure 1.3. If the width and height of the selected object are greater than some user specified value, then selection boxes should appear around the perimeter of the object, otherwise a selection arrow should be used.

Multi-way constraints allow the programmer to express many relationships that conventional one-way constraints are not capable of representing. These constraints are especially advantageous in constraint based interactive applications that are naturally symmetric. For instance, it is very difficult to implement a two-way flow of information between the interface and the application using one-way constraints. However, multi-way constraints make

this very easy to implement.

For example, some of the relationships discussed in the previous section could not have been implemented using only the conventional constraints. Figure 1.1 (b) illustrates one such relationship. In this example, multi-way constraints are required to allow the user to change any view, and have the other views updated automatically. The views could even be constrained to application variables, and the application variables could be constrained to the views. One-way constraints allow the user to only change one view, and have the other views updated automatically. In this case, the views could be constrained to application variables, but not vice versa. Thus the data must be procedurally modified, regardless of which view is altered.

Figure 1.2 illustrates another relationship that requires multi-way constraints. Like the previous example, multi-way constraints could be used to maintain consistency between these three views. If the user applies a transformation to the house in one view, then the other two views are redrawn automatically.

1.4 3D Toolkit

Our 3D toolkit facilitates creating highly interactive, direct manipulation, 3-dimensional user interfaces. The toolkit supports an extensive set of 3D objects including cubes, planes, 3D-lines, cones, cylinders, spheres, ellipsoids, and polyhedrons. It also supports constraints and includes a high-level event handling model. This event handling model encapsulates all the common interactive behaviors in a user interface into a few basic object types. This model allows the user to directly apply transformations to 3D objects, while providing a clean separation between input handling and the graphical representation for an object. The toolkit also includes functions for projecting objects, clipping, shading, and applying transformations. Finally, the toolkit also allows the user to interactively create 3D objects, change the viewing parameters for drawing 3D objects, and build structured graphics.

The extension to the conventional spreadsheet model and the 3D toolkit were imple-

mented in the Garnet user interface development environment [Myers 90a]. Garnet is an on-going research project that has developed an extensive set of tools that makes it significantly easier to create highly-interactive, graphical, direct manipulation user interfaces. Garnet stands for Generating an Amalgam of Real-time, Novel Editors and Toolkits. It is implemented in the Common Lisp programming language and resides on top of the X window system. The Garnet research project originated at Carnegie Mellon University and is now being developed at both CMU and the University of Tennessee. As of 1994, Garnet is being used regularly in over 60 projects around the world.

1.5 Dissertation Overview

The work presented in this dissertation facilitates the development of both 2D and 3D graphical user interfaces. It investigates how constraints can be used to simplify and speed up this development process. The related research on constraints and constructing interactive 3D graphics is reviewed in Chapter 2. This chapter also explains the constraint terminology that is used throughout this dissertation.

Chapter 3 describes how we have extended the spreadsheet model by incorporating multi-output, structural, and multi-way constraints. Chapter 4 explains the constraint solving process and presents the data structures and algorithms that are used for satisfying these constraints. This chapter also provides the time complexity, proof of correctness, and the performance of the extended constraints.

Chapters 5, 6, and 7 describe the object-oriented toolkit that has been developed for creating highly interactive, direct manipulation, 3-dimensional user interfaces. Chapter 5 presents an introduction to viewing in 3 dimensions. It discusses the 3D viewing process including a brief description of transformations and clipping. This chapter also discusses the types of projections and 3D objects that our toolkit supports. Chapter 6 describes the programming layer of the toolkit. This layer supports structured graphics, constraints, and a high-level event handling model. It also includes functions for projecting objects, clipping,

shading, and applying transformations. Chapter 7 discusses the direct manipulation layer of the toolkit. This layer allows the user to interactively create and directly apply transformations to 3D objects, construct polyhedron objects, change the viewing parameters for drawing 3D objects, and build structured graphics.

Chapter 8 reviews the major contributions of this work and points out ideas for future research.

Chapter 2

Terminology and Related Work

2.1 Introduction

This chapter introduces the terms that are used throughout this dissertation and then examines the related work in this area. Our review of the related work begins by examining several generic one-way constraint systems, with an emphasis on the desirable features of one-way constraint systems that we have adopted in our spreadsheet programming model. Next, we look at related work on multi-output, structural, and multi-way constraints. Finally, we examine the related work on interactive 3D user interfaces.

2.2 Definitions

The following terms are used throughout this dissertation:

constraint - a relationship between objects that is defined once and then automatically maintained by the system.

formula - a constraint. The terms formula and constraint are used interchangeably in this dissertation.

single-output constraint - a constraint that returns only one value.

multi-output constraint - a constraint that can return multiple values. The two types of multi-output constraints are: *fixed output* - where the outputs are fixed in advance; and *variable output* - where the outputs can dynamically change.

one-way constraint - a constraint that allows only *one* of the objects in the constraint to be changed in order for it to be satisfied.

multi-way constraint - a constraint that allows *any* of the objects in the constraint to be changed in order for it to be satisfied.

method - a block of code that enforces a constraint. A method takes zero or more inputs and produces one or more outputs. A one-way constraint has exactly one method whereas a multi-way constraint has one or more methods. Methods for single-output constraints may have only one output, whereas methods for multi-output constraints may have one or more outputs.

name space - the collection of named and unnamed objects that are used by the program. Named objects are associated with variable identifiers and are explicitly declared in a program. Unnamed objects are dynamically created at run-time via memory allocation.

value space - the set of values that can be assigned to an object.

type space - the set of value spaces that can be associated with a name.

Examples that illustrate name, type, and value spaces.

Name space	Type space	Value space
index	integer	43
sum	real	54.94
my-str	string	"Sunday"
my-rect	rectangle	{ left : 10, width : 20 top : 40, height : 35 color : red }

value attribute - a variable that conveys information about the value space. A value attribute represents a non-structural property of an object (e.g., an object's position, size, or color).

value constraint - a constraint that expresses a relationship on value attributes.

structural attribute - a variable whose value conveys information about the name or type space (e.g., an object's type, its components, whether it is being created, destroyed, or retained).

structural constraint - a constraint that expresses a relationship on structural attributes.

activation condition - a boolean predicate that determines whether a constraint is active or inactive.

conditional constraint - a constraint that can change back and forth between active and inactive depending on some condition. By placing an activation condition on a constraint, the constraint becomes a conditional constraint.

conventional constraints - single-output, one-way, value constraints.

extended constraints - multi-output, structural, and multi-way constraints.

aggregate - a collection of other graphical objects. Aggregates can be used as building blocks to create more complex objects, which in turn can be used to create even higher-level objects. They allow a structure hierarchy to be created.

spreadsheet programming - a paradigm that allows a programmer to declaratively describe relationships between a program's entities using constraints. The paradigm acquires its name from its similarity to spreadsheets, in which the value of cells may be defined by formulas that use the value of other cells as inputs.

2.3 One-way Constraints

One-way constraints are increasingly being used in graphical user interface toolkits and languages. Examples of generic one-way constraint systems include: Grow [Barth 86], Apogee [Hudson 89], Garnet [Myers 90a], Rendezvous [Hill 92], Eval/Vite [Hudson 93], and CONC++ [Levi 93]. Garnet was the first toolkit to completely integrate constraints with the object system and make them general purpose [Myers 90a]. There are two prominent

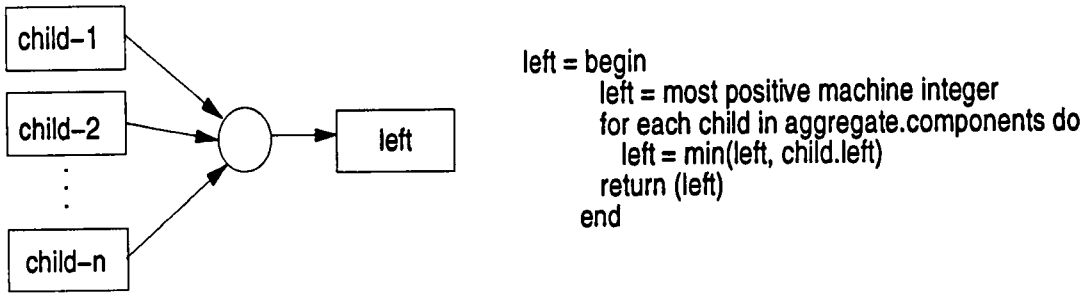


Figure 2.1: This constraint uses a variable sized input list. As children are added or deleted from the aggregate, the constraint is automatically recomputed.

features of Garnet's constraint system that we have included in our spreadsheet programming model. The first feature is variable sized input lists which allow the constraint to express relationships among dynamic sets of objects. This feature is essential for handling aggregates that may have dynamically changing children and for handling pointer-based structures, such as graphs, that have changing sets of neighbors. Figure 2.1 illustrates an example of a constraint that uses a variable sized input list. This constraint allows the aggregate to determine its left position by examining all of its children.

The second feature of Garnet that we have included is pointer variables. Pointer variables increase the expressive power of a constraint system by allowing constraints to express relationships among the elements of pointer-based data structures, such as lists, trees, and graphs. Programming languages have included pointer variables for many years. However, only recently have they been incorporated into constraint systems [Vander Zanden 91, Vander Zanden 94], Rendezvous [Hill 92], Multi-Garnet [Sannella 92a], and Eval/Vite [Hudson 93]. Several systems, including Coral [Szekely 88] and ThingLab II [Maloney 91], have also provided limited forms of pointer variables in their constraint models. Constraints that use pointer variables are also known as indirect reference constraints. Figure 2.2 provides an example application that incorporates pointer variables into constraints.

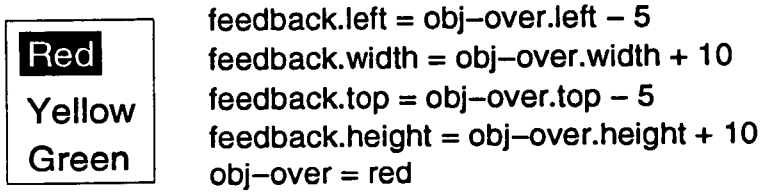


Figure 2.2: The feedback object in this menu uses constraints with pointer variables to 1) position itself over the selected object, and 2) resize itself to match the dimension of the selected object. By changing the value of the variable *obj-over*, the feedback object can appear over any object in the menu. *obj-over* currently points to the red menu item.

2.4 Multi-output Constraints

Multi-Garnet [Sannella 92a] and Rendezvous [Hill 92] are the only two existing systems that support multi-output domain-independent constraints. Multi-Garnet is an extension of Garnet that supports multi-way constraints and Rendezvous is a system that helps build multi-user, cooperative computing environments. Multi-Garnet was the first graphical user interface toolkit to provide multi-output constraints. The current implementation restricts constraints to having a fixed number of inputs and outputs. Our implementation of multi-output constraints relaxes this restriction, allowing a dynamic number of inputs and outputs. Relaxing this restriction allows multi-output constraints to express relationships among elements of variable-sized data structures, such as graphs and trees. Multi-Garnet's constraint satisfaction algorithm for multi-output constraints is exponential in the number of multi-output constraints, whereas the constraint satisfaction algorithm we have developed is linear. This linearity is achieved by requiring the programmer to provide additional information about which constraints are active during each satisfaction cycle.

Like our algorithm, Rendezvous's constraint satisfaction algorithm is linear in the number of multi-output constraints. Their algorithm can also support a dynamic number of inputs as well as outputs. The principle differences between our multi-output constraint solver and Rendezvous's are: 1) Rendezvous uses an eager evaluation algorithm, whereas ours uses a lazy evaluation algorithm; and 2) Rendezvous statically identifies outputs, whereas in our system, the constraint solver can dynamically determine outputs by evaluating arbitrary

pieces of code.

2.5 Structural Constraints

Freeman-Benson [Freeman 92] described a way of supporting structural constraints using the constraint imperative programming (CIP) language Kaleidoscope. A constraint imperative language combines constraint and imperative programming. The basic model is that the user makes a series of procedural changes to the system (the imperative part). These changes may add or delete constraints from the system, and cause existing constraints to become unsatisfied. Consequently, when the programmer has completed making procedural changes, a constraint solver is invoked to resatisfy the constraints (the constraint part). This cycle is then repeated. The effects of structural constraints can be obtained in Kaleidoscope by using its constraint constructor mechanism. A constraint constructor is a procedure that generates a set of constraints. The constructor may contain conditions and iterative constructs. Thus it may conditionally or iteratively generate constraints.

Kaleidoscope conceptually recreates the application's state at each time increment, so only objects that have satisfied the constraints are created at each step. For example, if a planet in a 3D solar system is deleted during a given time increment, then the planet's moons will not be recreated during the next update. In actual implementations, the constraint solver does not recreate the entire state at each step, rather it incrementally modifies it. The advantage of the CIP model is its declarativeness. This results in a high-level programming environment that should reduce the implementation time of user interfaces. The disadvantage of the CIP model is reduced efficiency. This reduced efficiency is a result of the constraint solver having to figure out how to efficiently modify the structure, without any input from the programmer. In our model the constraint solver is more explicitly told how to modify structures using action-procedures. These procedures increase the efficiency of the language, but decrease its declarativeness. However, if the programmer is less worried about efficiency, our model can use default action-procedures to achieve a degree of

declarativeness that is comparable to Kaleidoscope.

Rendezvous [Hill 92] also supports a limited form of structural constraints. It has a link mechanism that allows two objects to be automatically connected. If one object in the link is deleted the other is automatically deleted as well. If one object in the link is moved to another part of a data structure, the other object in the link is moved to a similar location in its data structure. If one object in a link is created, the other is automatically created as well. This mechanism is not as general as the mechanism we have developed because 1) our constraints may involve more than two objects; and 2) our constraints allow arbitrary relationships among objects to be described and maintained.

The latest version of Rendezvous also allows a constraint to commit side-effects by calling functions within a constraint method. Rendezvous defers executing these functions until after all constraints have been evaluated. There are three principle differences between how structural constraints are expressed and solved using our constraint solver versus Rendezvous's constraint solver. First, in Rendezvous the mechanism that performs the desired structural changes must be explicitly declared within the constraint methods. This approach intertwines the procedural and declarative mechanisms of a constraint. In contrast, our approach uses an active value model that separates the procedural and declarative mechanisms. The active value model allows action-procedures to be attached to variables. When a constraint recomputes the value of a variable, the action-procedure associated with the variable is automatically executed. Second, Rendezvous only allows one iteration of constraint solving, whereas, our approach allows multiple iterations. Once all constraints have been evaluated and all functions have been executed, the Rendezvous constraint solver terminates. However, our constraint solver can enter a second iteration of constraint solving to bring the constraint network into a more satisfied state. Third, we explicitly differentiate between structural and non-structural variables. Thus we explicitly differentiate between constraints that may manipulate the structure of an object, and constraints that merely alter attributes of an object.

2.6 Multi-way Constraints

One-way constraints allow only one variable in a constraint to be changed. For example, suppose a programmer creates the constraint [rectangle-1.left = rectangle-2.right]. This constraint expresses the relationship that *rectangle-1* is always to the right of *rectangle-2*. However, a one-way constraint can express this relationship in only one direction. For example, it can cause *rectangle-1* to move when *rectangle-2* moves, but not vice versa. In contrast, a multi-way constraint allows either *rectangle-1* or *rectangle-2* to be moved, and the other rectangle is adjusted appropriately to satisfy the constraint.

The CONSTRAINT user interface management system [Vander Zanden 88] uses an incremental constraint solving algorithm called "propagation of degrees of freedom" to solve multi-way constraints. This algorithm was improved upon in [Vander Zanden 92b]. CONSTRAINT is incremental and uses two phases for constraint solving — a planning phase that selects a method to satisfy each constraint and an evaluation phase that executes the methods. Our algorithm avoids the expensive planning phase associated with CONSTRAINT by involving the programmer in the planning process. It also is capable of handling pointer variables and variable sized sets of inputs.

Rendezvous [Hill 92] supports a limited version of multi-way constraints that allows two variables to participate in a multi-way constraint (i.e., there may be an arbitrary number of variables in the constraint, but only two of the variables may be altered to satisfy the constraint). Like the system that we have designed, Rendezvous provides multi-way constraints by allowing multiple one-way constraints to be attached to objects. It also permits indirect references, both in the inputs and in the outputs. For example, it allows both [A.color = object-over.color] and [object-over.color = A.color]. Our scheme is more general than Rendezvous's in that it supports arbitrary multi-way constraints. Our scheme also supports conditional constraints and user control over the resolution of conflicts between constraints (i.e., what to do when two constraints determine the same variable).

The DeltaBlue constraint solver [Freeman 90] also supports multi-way constraints and is

used in both ThingLab II [Maloney 91] and Kaleidoscope [Freeman 92]. It uses a constraint solving algorithm called "propagation of known states". Unlike the model that we have designed, DeltaBlue supports constraint hierarchies which allow programmers to prioritize constraints. The constraint solver resolves conflicts between constraints by giving greater consideration to constraints with a higher priority level. Constraint hierarchies provide the user with greater control over the constraint solving process. The disadvantage of a constraint hierarchy is that some constraints always have precedence over other constraints regardless of the situation. Our scheme replaces constraint hierarchies with the notion of conditional constraints. Conditional constraints have activation conditions that are boolean predicates. The constraint is active only if the activation condition evaluates to true. Activation conditions provide increased user control over constraints, since, depending on the situation, different sets of constraints can be given precedence by activating the desired sets of constraints and deactivating the remaining sets of constraints.

The Skyblue algorithm which is used in Multi-Garnet [Sannella 92a] supports multi-way constraints and can call other, more powerful constraint solvers if it is unable to completely satisfy a system of constraints. Multi-Garnet also supports pointers by using direct reference constraints, and removing and then reinserting a constraint each time a pointer variable changes. However, the constraints must have a fixed number of inputs, which makes the use of pointer variables more limited than in the one-way Garnet model. Our algorithm permits a dynamic number of inputs, thus maintaining compatibility with Garnet's one-way constraints. Our algorithm also differs from Multi-Garnet in that it directly supports pointer variables.

Siri [Horn 92] is a constraint imperative language based on term rewriting that supports slightly nonlinear, simultaneous equations. It is somewhat similar to the Kaleidoscope language in that it supports a refinement model in which constraints can be gradually introduced into the system by constraint templates during the process of rewriting a term and then eventually reduced to a solved form. Our approach is somewhat more conven-

tional than Siri's in that the spreadsheet and object mechanisms are separate entities that interact with one another. In Siri, these two mechanisms are unified using one abstraction, a constraint template. In addition, Siri's constraint satisfaction algorithm is based on term-rewriting whereas our algorithm is based on dependency graphs.

2.7 Constructing Interactive 3D Graphics

Most of the 3D application programming interfaces (APIs) available today are based on an imperative style of programming. This includes vendor specific APIs such as SGI's GL [SGI 91], IRIS GL [IRIS 92], RenderMan [Upstill 90], Sun's XGL [Foley 92], and HP's Starbase [HP 91]. This is also true for the more portable APIs developed by international standards bodies including Core [Foley 90], GKS-3D [GKS 88], PHIGS [PHIGS 84], PHIGS+ [PHIGS+ 88], PEX [Foley 92], and OpenGL [Foley 92]. Differences between these APIs (e.g., between PEX and OpenGL) have dealt with portability, openness, and the graphics rendering required to produce pseudorealistic images [Foley 92]. In general these APIs have ignored features like object-oriented programming, constraints, and direct manipulation. These APIs force the programmers themselves to maintain the relationship between their application data and the graphical appearance of this data. As a result, creating interactive 3D graphics using these APIs has traditionally been a very time consuming and difficult process. Most development toolkits created using these conventional APIs provide gross-object picking but little or no support for direct manipulation of 3D objects. On the other hand, object-oriented APIs provide a better correlation between the modeled objects and the commands used to apply transformations to them. Consequently, object-oriented approaches typically provide more high-level features like direct-manipulation of objects and require significantly less time and effort to develop applications.

Our extension to Garnet takes an object-oriented approach to programming 3D graphical interfaces and is oriented toward interactive interfaces. Our model makes programming 3D graphics easier by supporting features like direct manipulation, constraints, aggregates of

objects, and structural inheritance (structural inheritance means that when an instance of an aggregate object is created, instances of all its component parts are created as well).

Increasingly, other toolkits are also being developing using an object-oriented approach. IRIS Inventor [Strauss 93] is an object-oriented 3D graphics toolkit that supports direct manipulation and inheritance. Their toolkit provides direct manipulation through the use of "manipulator" nodes. The manipulator nodes are "smart" nodes that respond to interaction events causing modification to other nodes in a database. Unlike our toolkit, Inventor supports cache rendering. This means that it stores a faster representation for those portions of a scene that do not change from one screen update to the next screen update. Our toolkit instead uses an incremental update algorithm that only redraws those objects that have intersected the modified portion of the screen. While Inventor does allow transformations to be interactively entered, it does not support constraints.

The Brown Animation System [Zeleznik 91] provides 3D object representation and dynamics. The system was primarily designed for modeling and animation. Although the system does support inheritance and a limited version of constraints, it does not include support for direct manipulation.

GROOP [Koved 93] and GRAMS [Egbert 92] are two more examples of object-oriented toolkits used for creating 3D applications. Both systems were designed to be portable across different application programming interfaces. They provide this portability by separating the geometry and lighting/material specifications from the rendering process. The main difference between GROOP and GRAMS is in their class hierarchy. GRAMS allows objects to be rendered differently within the same window. Our approach, like GROOP, does not allow different rendering modes to be mixed within a single window. Although, both GRAMS and GROOP support a better lighting model than ours, neither one supports constraints or direct manipulation.

Part II

CONSTRAINTS

Extensions to the Spreadsheet Model ..... Chapter 3

Constraint Solving ..... Chapter 4

Chapter 3

Extensions to the Spreadsheet Model

3.1 Introduction

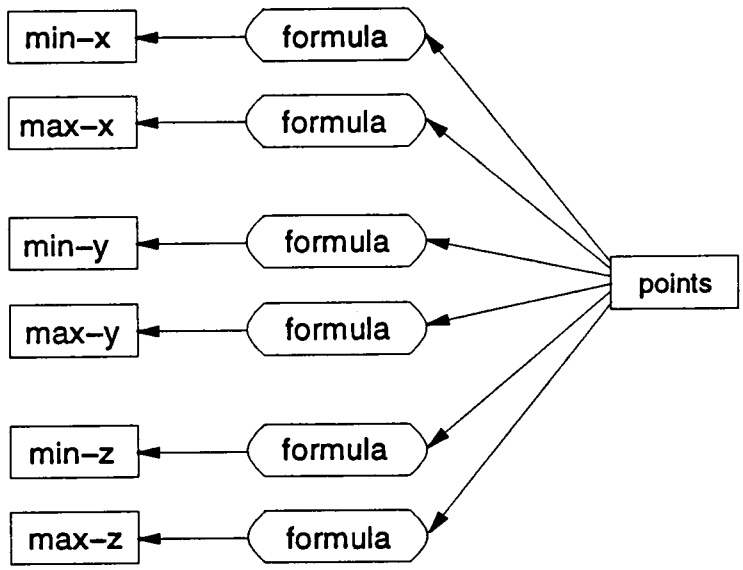
The spreadsheet model that we have built upon is a domain-independent model. Domain-independent means that the constraints can compute a value of any arbitrary type using an arbitrary piece of code. The conventional spreadsheet model assumes that constraints are single-output, one-way, and apply only to the value attributes of an object. In this chapter we describe our design for extending the conventional model to handle multi-output, structural, and multi-way constraints. In the next chapter we describe incremental algorithms that satisfy these new types of constraints.

3.2 Multi-output Constraints

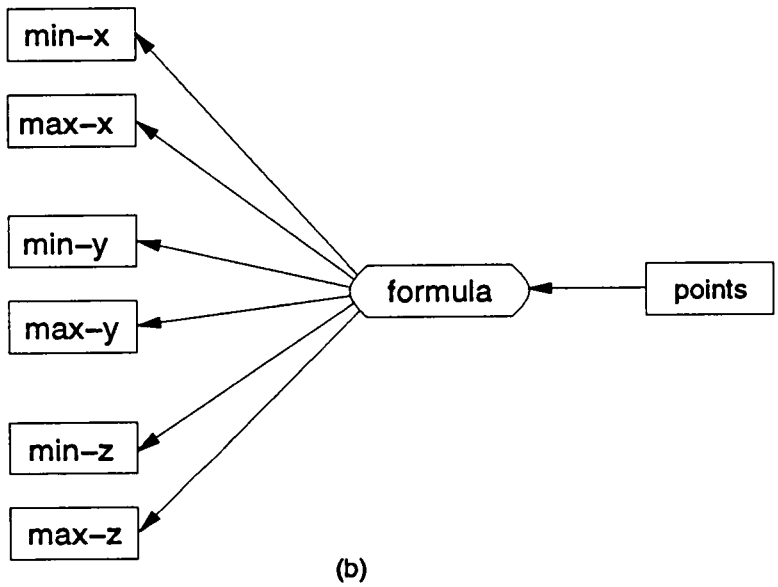
Multi-output constraints allow a constraint to return more than one value. For example, suppose we wish to use constraints to determine the minimum and maximum x, y, and z values for an arbitrary 3D polyhedron object. Assume the points are stored in the slot *points* as shown below.

```
points = ((59 62 71) (12 22 23) (58 76 54) (5 56 78) ...  )
```

Using single-output constraints, the programmer can compute the desired values by creating the following constraints which are illustrated in Figure 3.1 (a).



(a)



(b)

Figure 3.1: Six single-output constraints are required to calculate the minimum and maximum x, y, and z values for an arbitrary 3D polyhedron object (a). One multi-output constraint can calculate all six values in one pass over the points (b).

```
min-x = find-smallest-x-value (points)
max-x = find-largest-x-value (points)

min-y = find-smallest-y-value (points)
max-y = find-largest-y-value (points)

min-z = find-smallest-z-value (points)
max-z = find-largest-z-value (points)
```

Each of these six formulas is a function that takes *points* as input and returns an appropriate value. For example, *min-x* can be represented as follows.

```
min-x = begin
          min-x = most positive machine integer
          for each pt ∈ points do
             min-x = min(min-x, pt.x)
          return(min-x)
        end
```

A comparable multi-output constraint can be written as follows. This constraint is illustrated in Figure 3.1 (b).

```
(min-x, max-x, min-y, max-y, min-z, max-z) =
  /* initialize min-x, max-x, min-y, max-y, min-z, and max-z */
  min-x, min-y, min-z = most positive machine integer
  max-x, max-y, max-z = most negative machine integer
  for each pt ∈ points do
     min-x = min(min-x, pt.x)
     max-x = max(max-x, pt.x)
     min-y = min(min-y, pt.y)
     max-y = max(max-y, pt.y)
     min-z = min(min-z, pt.z)
     max-z = max(max-z, pt.z)
  return(min-x, max-x, min-y, max-y, min-z, max-z)
```

Comparing the single-output and multi-output formulas illustrates the advantages of multi-output formulas. First, the single-output constraints require six passes over the values in the points slot, whereas the multi-output constraint requires only one pass. Thus the multi-output constraint reuses the "computation" of iterating over the points list. Second, the single-output constraints require six formula objects to be created, whereas the multi-

output constraint requires only one formula object. The additional formulas required by the single-output constraints not only consume more memory but also force the constraint solver to perform extra bookkeeping operations such as keeping track of dependencies between the points slot and the constraints. Finally, it is probable that the programmer thinks of the min-max example as a single unit of computation rather than six individual units of computations. Thus the multi-output constraint conforms more closely with the programmer's notion of how the computation should be performed. The multi-output constraint in effect allows the programmer to control the granularity of the constraints by determining how much information a constraint is allowed to compute.

One disadvantage of multi-output constraints is that it is possible for unnecessary calculations to be performed. For example, whenever one of the input values to a constraint changes, all output values are automatically re-calculated, even if some outputs do not depend on the input. Single-output constraints are finer-grained, and thus only re-calculate those values that actually depend on the input. Using the above example, the value of *max-z* would be needlessly recomputed if the x-value of one of the points is changed. However, the programmer can avoid this disadvantage by using multi-output constraints only when it is expected that all outputs in the constraint will actually be recomputed. For example, it would be unusual for the x-value of a point to change unless all values in the point change.

Our design allows a constraint to have either a fixed or a variable number of outputs. The fixed case was illustrated earlier in this section. To specify variable outputs, the programmer attaches an expression to a constraint that determines the set of output objects. We call this expression an "output determinant constraint". The expression is itself a constraint that can depend on other variables in the system. The programmer also assigns a list of output variables to the multi-output constraint. The multi-output constraint defines the values of these variables in each output object. The number of outputs from the multi-output constraint should equal the number of output objects times the number of variables specified. For example, if five objects are computed by the output determinant constraint

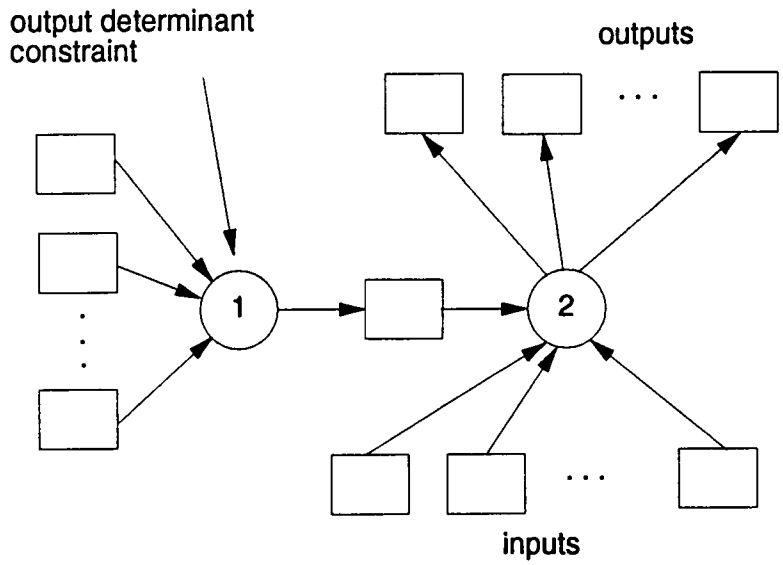


Figure 3.2: A pictorial description of a multi-output constraint that uses an output determinant constraint. The output determinant constraint labeled (1) specifies the output objects. The multi-output constraint labeled (2) solves the formula and stores the values in the output objects.

and two variables are specified, then the formula should return ten values. Figure 3.2 illustrates a multi-output constraint that uses an output determinant constraint.

The parameters for our multi-output constraint include: 1) an output determinant constraint that specifies the set of output objects, 2) a list of the variables associated with each object that are assigned values, and 3) a formula that determines the values of the output variables.

To illustrate our multi-output design, consider the following example. Suppose a programmer wishes to construct a constraint that allows an aggregate to determine the position of its children. Moving an aggregate should cause its children to move by an equivalent amount. The position of the children should be set equal to the position of the aggregate that they are attached to, plus an optional offset within the aggregate. The variables *x-offset* and *y-offset* specify an object's optional offset. This constraint can be constructed as follows.

```
output-objects = aggregate.components
```

```
output-variables = (left, top)
formula = begin
            list = null
            for each object ∈ aggregate.components do
               list = list ∪ {(aggregate.left + object.x-offset)}
               list = list ∪ {(aggregate.top + object.y-offset)}
            return (list)
          end
```

In this example the multi-output constraint outputs to both the left and top slots of all "output-objects". If an object is added to or deleted from the list, the output expression is re-evaluated and the constraint is applied to the resulting set of objects. The advantage of this constraint is that new relationships (i.e., constraints) do not have to be created as objects are added to the aggregate.

Multi-output constraints can also be used to implement the following example. Suppose the programmer wishes to constrain the objects in a list to be 10 pixels apart horizontally and centered vertically. New objects can be added to the end of the list, and deleting an object from the list should automatically cause the remaining objects to reposition themselves to maintain the 10 pixels of separation. This relationship is illustrated in Figure 3.3 and can be implemented using the following multi-output constraint.

```
output-objects = list.components
output-variables = (left, top)
formula = begin
            next-right = list.left
            list = null
            for each object ∈ list.components do
               list = list ∪ {next-right}
               list = list ∪ {list.center}
               next-right = object.left + object.width + 10
            return (list)
          end
```

This multi-output constraint sequentially calculates the left slots of all objects in the list. When an object is added to or deleted from the list, the multi-output constraint is invalidated. When the constraint solver is invoked, the entire multi-output formula is re-

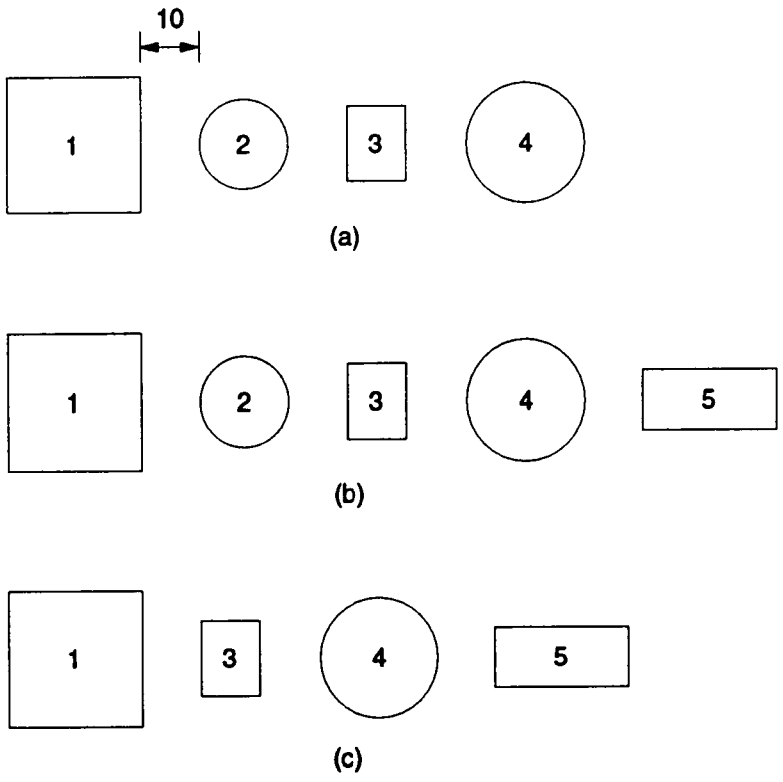


Figure 3.3: An example list that contains four objects which are separated by 10 pixels (a). If object 5 is added to the list, its left value is calculated to be 10 pixels to the right of the last object added (b). If object 2 is deleted, the left values of all objects are automatically recalculated (c).

evaluated determining the left and top values for all objects in the list. The advantage of this constraint is that this relationship is specified once. If single-output constraints are used instead, two new relationships must be declared every time an object is added to the list.

3.3 Structural Constraints

Conventional constraints only allow the value attributes of an object to be constrained. The value attributes of a graphical object might include position, orientation, size, filling-style, and line-style. However, there are many cases where it is desirable to constrain the structure of objects. Structural attributes might include 1) the type of an object (e.g., rectangle, circle); 2) the parts of an object (e.g., a labeled text box might include a text object and a rectangle object); and 3) the existence of an object (e.g., an arrow should exist only if the two boxes that it connects exist). By expressing relationships among these attributes, constraints can add, delete, or modify parts of a structure.

As an example, structural constraints allow the creation of one object to result in the creation of one or more other objects. For instance, creating an object in an "interactive" window could automatically create selection handles for the object. Creating the same object in a "non-interactive" window would not create selection handles for the object. The same idea applies to deleting objects. For example, deleting a planet in a solar system could result in the automatic deletion of its moons, if the existence of the moons is tied to the existence of the planet via constraints. Finally, changing the type of a road from a secondary to a primary road might cause yield signs to change to stop signs along intersecting roads.

In our design, the values of structural attributes are ordinary values like strings for types (e.g., *rectangle*) or pointers to components. Simply changing these values will not have the desired structural effects. Consequently, some mechanism must be devised that performs the desired structural changes when the value of a structural attribute changes.

Our approach uses an active-value model to enforce the structural constraints. An

active value model allows action-procedures to be attached to variables [Vander Zanden 92a, Stefik 86]. When the value of a variable changes, the action-procedure associated with it is automatically executed. Action-procedures can be associated with any structural attribute. Changing the value of a structural variable triggers the action-procedures, which enforces the structural changes that the new value requires. By allowing constraints to be attached to these structural variables, we obtain structural constraints. It should be mentioned that the programmer must declare which variables are structural attributes. Hence, it is also possible to attach action-procedures to value attributes. However, these action-procedures should not alter the structure of an application. An action-procedure takes as parameters the object and variable that it is associated with.

Example To illustrate our design, consider how structural constraints might help in implementing a graphical road layout system. The system allows the programmer to create roads and simulate their traffic. Some of the possible uses for structural constraints might include:

- **Addition:** When a road is added and it intersects another road, structural constraints could be used to automatically create turning lanes, warning signs, stop signs, and signal lights. If the road takes a sharp curve, then warning signs could be automatically added to the side of the road. Similarly, a dead-end sign could be added shortly before a road ends. A simplified structural constraint for adding warning signs before a signal-light could be defined as follows.

```
exist-p = road1.exist-p and road2.exist-p
```

  road1 and *road2* are pointers to the roads that the signal light is associated with. The constraint itself is associated with the signal light. Hence, when the signal light exists, the warning signs should also exist. The *exist-p* slot is initially false and changes to true when the signal light is created. The action procedure could then be defined as

follows.

```
action-procedure[1] SIGNAL-LIGHT-EXISTENCE (signal-light exist-p)
  if signal-light.exist-p = true then
    if road1.speedlimit <= 30 then
      create warning signs 200 feet before and after the
        signal light on road1
    else
      create warning signs 500 feet before and after the
        signal light on road1
    if road2.speedlimit <= 30 then
      create warning signs 200 feet before and after the
        signal light on road2
    else
      create warning signs 500 feet before and after the
        signal light on road2
```

- **Modification:** Changing a road from a secondary to a primary road could possibly change yield signs to stop signs along intersecting or merging roads. A simplified action-procedure for modifying the type of a road might be defined as follows.

```
action-procedure CHANGE-TYPE (road type)
  case type
    secondary:
      enforce yield signs at merging roads
    primary:
      enforce stop signs at merging roads
    interstate:
      enforce entrance ramps at merging roads
```

- **Deletion:** Structural constraints can be used for deleting objects from the road layout system. When an object's existence becomes false, then the appropriate deletion procedure is called. For example, deleting a road that intersects another road, should automatically remove signal lights, stop signs, etc. We will now modify the action-procedure used in the "Addition" example, so that it automatically deletes signal-

[1]The syntax of an action-procedures is: action-procedure procedure-name (object variable) code

lights when their *exist-p* slot evaluates to false.

```
action-procedure SIGNAL-LIGHT-EXISTENCE (signal-light exist-p)
  if signal-light.exist-p = true then
    if road1.speedlimit <= 30 then
      create warning signs 200 feet before and after the
        signal light on road1
    else
      create warning signs 500 feet before and after the
        signal light on road1
    if road2.speedlimit <= 30 then
      create warning signs 200 feet before and after the
        signal light on road2
    else
      create warning signs 500 feet before and after the
        signal light on road2
*   else
*     destroy signal-light
```

The ⋆ denote changes. We now expand this example to show that structural constraints can have ripple effects by adding a structural constraint to the warning signs. Now when the signal light is destroyed, the warning signs are also automatically destroyed. The structural constraint for the warning sign might be [warning-sign.exist-p = signal-light.exist-p] and the action-procedure might be written as follows.

```
action-procedure WARNING-SIGN-EXISTENCE (warning-sign exist-p)
  if warning-sign.exist-p = false then
    destroy warning-sign
```

All structural constraints are evaluated before any action procedures are executed (the action procedures are deferred). Thus the *exist-p* attributes in the example above are brought up-to-date and then the action procedures are executed to enforce the structural constraints (thus deleting road, any associated signal lights, and any warning signs associated with the signal lights).

Structural constraints can also be used to specify the type of selection handle that should surround a selected object. If the width and height of the selected object are greater than

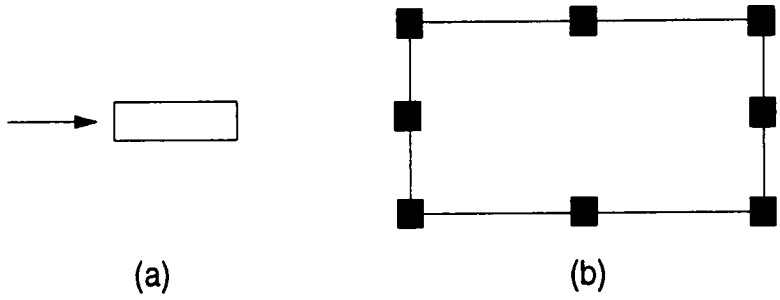


Figure 3.4: The type of selection handle should depend on the size of the object. Smaller objects that cannot accommodate the feedback boxes should be pointed at by a selection arrow (a), larger objects should be surrounded by feedback boxes (b).

some user specified value, then selection boxes appear around the perimeter of the object, otherwise a selection arrow is used (Figure 3.4).

To implement this relationship, an action-procedure can be attached to an object's type. The constraint for type could reference those variables that determine the size of the object (i.e., the *width* and *height*). When the values of either one of these variables changes, this action-procedure is automatically called. Depending on the object's new size, the action-procedure could create new selection handles, and destroy the old ones. The structural constraint on *type* and its action-procedure can be defined as follows.

```
type =  if (obj-over.width > min-width) AND
             (obj-over.height > min-height) then
          selection-handles
        else
          arrow

 action-procedure FEEDBACK (feedback-object type)
   delete old feedback parts
   allocate parts for new feedback type
   register parts with display manager
```

Finally, structural constraints can be used implement the following relationship. Suppose the programmer wishes to constrain an arrow's existence to the existence of the two boxes it connects as shown in Figure 3.5. This relationship can be implemented as follows. An action-procedure is attached to the arrow's existence slot *exist-p*. When the value of this

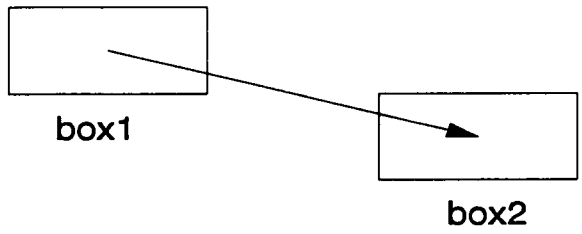


Figure 3.5: A structural constraint can be placed on the arrow. If either box1 or box2 is destroyed, then the arrow should also be automatically destroyed.

variable changes, this action-procedure is automatically called. If the value of this variable ever becomes false, then the arrow is deleted. The constraint for the structural attribute *exist-p* and the action-procedure can be defined as follows.

```
arrow.exist-p = box1.exist-p and box2.exist-p

action-procedure DESTROY-ARROW (arrow exist-p)
  if arrow.exist-p = false
    destroy arrow
```

3.4 Multi-way Constraints

Our multi-way constraint design includes five important features:

- Multiple formulas can be attached to a single variable. These formulas may include one-way constraints, multi-output constraints, and structural constraints.
- Dynamic number of inputs and outputs
- Pointer variables for the inputs and outputs
- Priorities
- Activation conditions

While other constraint solvers may incorporate some of these features, ours is the first to support all features at once. The first feature allows the programmer to formulate multi-way constraints as a series of one-way constraints. Experience in teaching constraints to students suggests that for many applications, such as multiple views, people actually think in terms of one-way rather than multi-way constraints. In general, students are first taught

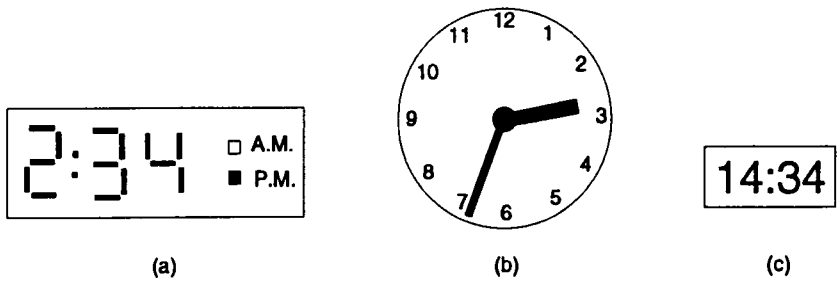


Figure 3.6: Multi-way constraints are used to maintain consistency among three clocks. The time is represented using: (a) a 12 hour digital clock with AM/PM notation, (b) an analog 12 hour clock face, and (c) a 24 hour digital clock.

one-way constraints, because they are the most commonly used. One-way constraints are attached to variables, and thus students think of variables as "owning" constraints. Multi-way constraints are typically presented as an independent object, with a set of methods for satisfying the constraint. The methods belong to the object, rather than individual variables, which seems to confuse most students. Multi-way constraints would be easier to design and conform more closely to the student's model if the methods were allocated to the individual variables. Since each method represents a one-way constraint, we decided to build multi-way constraints by allowing a variable to be assigned multiple one-way constraints.

As an example, a graphical user interface can be constructed using multi-way constraints to help young children learn different ways of expressing time (Figure 3.6). If a child changes any one of the views, the other two views are automatically redrawn. The constraints that implement this interface are shown below.

$$clock\_1.time = clock\_2.time$$

$$clock\_2.time = \begin{cases} clock\_1.time \\ clock\_3.time \end{cases}$$

$$clock\_3.time = clock\_2.time$$

Notice that two, one-way constraints are attached to clock_2's time slot. If clock_1's time changes, then clock_2 gets its new time from clock_1. On the other hand, if clock_3's time changes, then clock_2 gets its time from clock_3.

The second and third features allow constraints to have a dynamic number of inputs and outputs and pointer variables for the inputs and outputs. These two features are automatically provided by allowing any of the types of formulas discussed thus far (one-way, multi-output, and structural) to be attached to a variable. As noted in Chapter 2, these two features have yet to be incorporated in multi-way schemes, except in limited contexts.

The fourth feature allows the programmer to optionally attach priorities to formulas. The priority indicates how strongly the programmer wants a given formula to determine a slot's value. The formula with the highest priority in a slot determines the slot's value. If two or more formulas with the same priority are invalid and must be re-evaluated, then the formula that occurs first in the slot's formula list determines the slot's value. Programmers have control over the placement of formulas in the list[2]. Only formulas with the highest priority in a slot are active and can be evaluated. Formulas with lower priorities can be invalidated but are not evaluated. This priority scheme allows default formulas with low priorities to be placed in a slot. These formulas are masked when higher priority formulas are placed in the slot. The default formulas automatically activate themselves if these higher priority formulas are later removed. An example use of a default formula is a formula that returns the minimum left side of any of the aggregate's children. This formula can be used to compute the aggregate's *left* value unless another formula is installed that computes it in some other fashion (e.g., a formula that constrains the aggregate's left side to another object).

The last feature allows programmers to add activation conditions to constraints. An activation condition is a boolean predicate that determines whether a constraint is active or inactive. The boolean predicate may consist of both value attributes and structural

[2]The syntax for adding a formula to a slot is: add-formula (object, slot, formula, position, priority, activation-condition). *position* specifies the location of *formula* in the list. *priority* indicates how strongly the programmer wants this formula to determine the value of the slot. *activation-condition* is a boolean predicate that determines if a constraint is active.

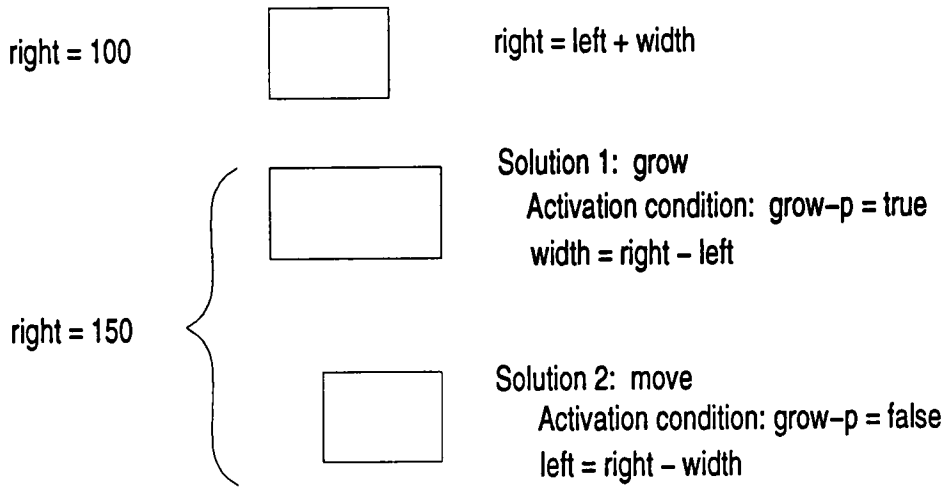


Figure 3.7: The constraint "right = left + width" has been placed on the rectangle. If the value of right is changed from 100 to 150, then the constraint can be satisfied by either changing the left or the width of the rectangle. When the *grow-p* activation condition is true, then the width of the rectangle is changed, otherwise, the left side of the rectangle is changed to satisfy the constraint.

attributes. The activation conditions in our scheme replace constraint hierarchies found in other constraint solvers. The weakness of constraint hierarchies is that the strengths of their constraints are fixed. Thus certain constraints are always preferred over other constraints. For example, constraints that cause an object to move may be favored over constraints that cause an object to resize. Activation conditions remedy this shortcoming by giving the user control over which constraints are favored at any given time. When constraints that cause an object to resize are favored, the constraints that cause an object to move are deactivated. Similarly, when the programmer wants an object to move, the constraints that cause an object to resize are deactivated. Figure 3.7 illustrates the move/resize activation condition. By placing an activation condition on a constraint, the constraint becomes a conditional constraint. Conditional constraints include the programmer into the planning process in a natural, comfortable way by allowing the programmer to specify which constraints are active. As a result, the planning burden on the constraint satisfier is reduced.

An example application that can be implemented using this multi-way constraint scheme is illustrated in the visual color editor shown in Figure 3.8. Multi-way constraints can be

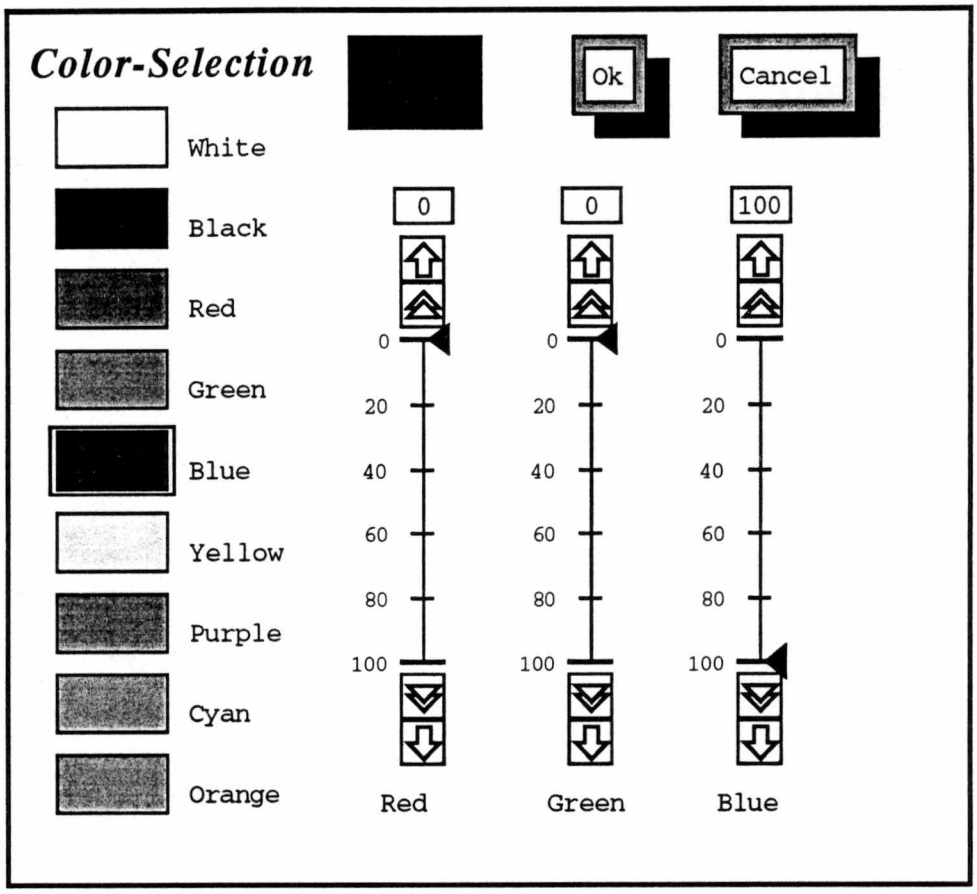


Figure 3.8: A visual color editor implemented using multi-way constraints. Colors can be selected either by choosing one of the colors in the menu on the left or by manipulating the three sliders on the right. The color box at the top provides feedback showing the selected object.

used in this application to establish relationships among the following values:

- The indicators located on the sliders.
- The value boxes located above the sliders.
- The feedback box possibly surrounding one of the pre-defined colors.

Below is one possible way in which the multi-way constraints can be constructed:

$$centralized\_red = \begin{cases} red\_slider.indicator.value \\ red\_value\_box.value \\ selected\_menu\_object.red \end{cases}$$

$$centralized\_green = \begin{cases} green\_slider.indicator.value \\ green\_value\_box.value \\ selected\_menu\_object.green \end{cases}$$

$$centralized\_blue = \begin{cases} blue\_slider.indicator.value \\ blue\_value\_box.value \\ selected\_menu\_object.blue \end{cases}$$

$$red\_slider.indicator.value = centralized\_red$$
$$green\_slider.indicator.value = centralized\_green$$
$$blue\_slider.indicator.value = centralized\_blue$$

$$red\_value\_box.value = centralized\_red$$
$$green\_value\_box.value = centralized\_green$$
$$blue\_value\_box.value = centralized\_blue$$

$$selected\_menu\_object = find\_color(centralized\_red, centralized\_green, centralized\_blue)$$

The following one-way constraint ensures that the feedback rectangle next to the color-selection label has the correct color:

$$feedback\_color\_rectangle.color = create\_color(centralized\_red, centralized\_green, centralized\_blue)$$

For example, if the user selects the pre-defined color blue, thus setting the "centralized" variables, the multi-way constraints perform the following actions. A feedback box surrounds the pre-defined color blue. The indicators and value boxes on the sliders are set to (0, 0, 100), and the color of the feedback color rectangle is set to blue. Now if the user drags the indicator of the first slider to the 100 hashmark, then the following actions are taken. The pre-defined color purple is highlighted with a feedback box since purple is defined as (red 100) (green 0) and (blue 100). The value box above the first indicator is set to 100 and the filling style of the feedback color rectangle is set to purple.

Another practical application of multi-way constraints is for computing the position and size of an aggregate and its children (this example also involves multi-output constraints). A multi-way constraint can be constructed between an aggregate and its children as shown in Figure 3.9. In one direction, the children determine their position from the aggregate. Hence, if the left value of the aggregate is incremented by 10, then all children are moved to the right 10 pixels. In the other direction, the aggregate determines its bounding box from the children. For example, if a new object is added to the aggregate, then the bounding box of the aggregate is recalculated.

3.5 Expressiveness of the Extended Spreadsheet Model

The extended spreadsheet model not only makes it easier and more efficient to specify and implement relationships among variables, but it also allows more powerful relationships to be expressed. In this section we compare the expressiveness of the conventional constraints with our extended constraints. Recall that the conventional constraints refer to single-output, one-way relationships that can be expressed only on the value attributes of

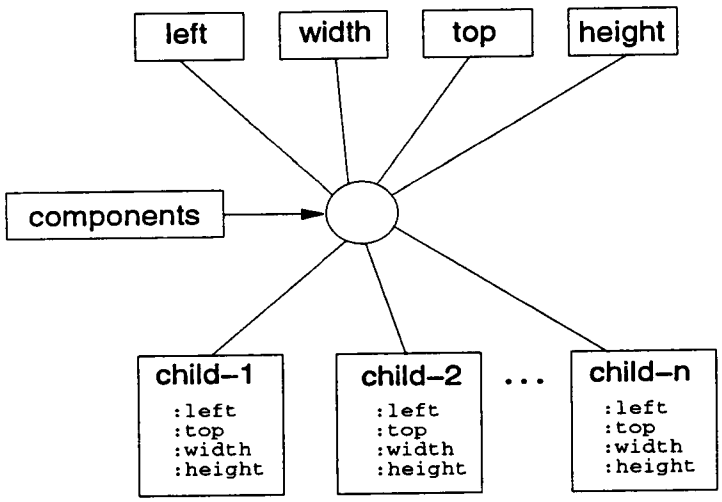


Figure 3.9: A multi-way constraint can be expressed between an aggregate and its children. In one direction, the children determine their position from the aggregate. In the other direction, the aggregate determines its bounding box from the children.

objects.

- Multi-output constraints have the same expressiveness as single-output constraints. One multi-output constraint that has n output variables can be separated into n single-output constraints by appropriately splitting and duplicating the computation. Similarly, n single-output constraints can be represented by combining their code bodies into one multi-output constraint that has n output variables.

- Structural constraints are more expressive than conventional constraints, because structural constraints allow relationships to be specified over the value space, type space, and name space, whereas the value constraints used in the conventional constraint model only allow relationships to be defined over the value space.

- Multi-way constraints are more expressive than the one-way constraints found in the conventional spreadsheet model. To compare the expressiveness of one-way and multi-way constraints, we need to define the following two sets:

  C = the set of constraints to be solved.

S = the set of partial orders that satisfy the constraints in C.

The one-way constraints of the conventional model only allow one of the possible partial orders in S to be represented. However, multi-way constraints allow all of the partial orders in S to be represented. For example, if a constraint contains n variables, a one-way constraint system could have only one solution to the constraint, whereas a multi-way constraint system could have multiple solutions.

Chapter 4

Constraint Solving

4.1 Introduction

This chapter presents the constraint satisfaction algorithms for multi-output, structural, and multi-way constraints, and then presents an algorithm for satisfying systems that support all three types of constraints. For each constraint algorithm, the data structures, the time complexity, and a proof of correctness is provided. The last section of this chapter examines the performance of these constraints. Before looking at the specific constraint algorithms, an introduction to the two major types of constraint solving is provided.

In general constraints can be solved using either a *lazy* or an *eager* evaluation algorithm. A lazy evaluation algorithm does not evaluate constraints until they are needed, whereas an eager evaluation algorithm re-evaluates constraints as soon as a constraint or variable is modified. The advantage of lazy evaluation is that it is easier to implement and avoids potentially unnecessary re-evaluations. However, lazy evaluation may result in longer delays when a variable is accessed as the values of invalid variables are demanded. Eager evaluation is usually preferred if a high percentage of constraints need to be re-evaluated and the changes are occurring to a localized section of the constraint system. Conceptually, eager evaluation is cleaner than lazy evaluation since everything is always kept up-to-date.

Our constraint solving algorithm uses a mark/sweep approach (also called nullification/reevaluation) that can be adapted to work with both lazy and eager evaluation. The

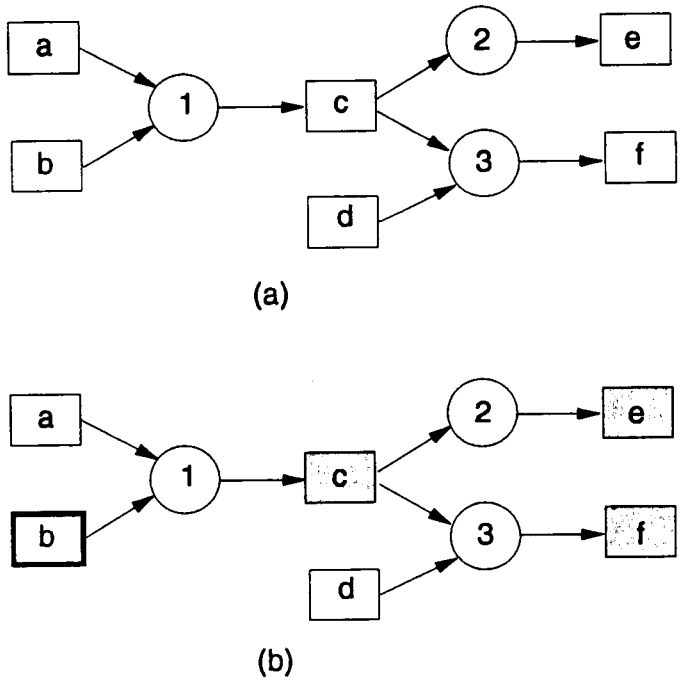


Figure 4.1: The constraints $c = a + b$, $e = c$, and $f = c + d$ are represented using a directed graph (a). The rectangles represent variables and the circles represent constraints. The gray rectangles represent variables that are marked invalid when b is changed (b).

constraint solver builds a directed, bipartite dataflow graph similar to the one shown in Figure 4.1. The two types of nodes are formula nodes and variable nodes. There is a directed edge from a variable node to a formula node if the variable is an input to the formula. There is a directed edge from a formula node to a variable node if the formula outputs the variable.

When the value of a variable changes, all formulas that directly or indirectly depend on this variable are marked invalid (mark pass). When the value of an invalid formula is requested, the constraint solver demands the values of other variables that the invalid formula depends on (sweep pass). If these variables contain formulas that are also out of date, the constraint solver recursively demands values until valid variables are reached, at which point the constraint equation can be solved and the process proceeds forward. This algorithm can be used as described for lazy evaluation. If in addition, a list of out-of-date variables is kept, the eager evaluator can demand the value of each variable in this list. This

will bring the whole system up-to-date.

We will now illustrate the mark/sweep approach using Figure 4.1. The gray rectangles in Figure 4.1 (b) represent variables that are marked invalid when the value of variable *b* is changed. If a lazy evaluation algorithm is used, only those variables that are demanded are computed. For example, if the value of *e* is requested, *e* will demand the value of *c*, and *c* will recursively demand the values of both *a* and *b*. Since *a* and *b* are both valid, *c* will compute its value, mark itself valid, and return its value to *e*. *e* then computes its value, marks itself valid, and returns. The variable *f* remains invalid since its value was never demanded. If an eager evaluation algorithm is used, then the values for *c*, *e*, and *f* will all be computed immediately following the change to *b*. The initial list of invalid variables will consist of variables *c*, *e*, and *f*.

4.2 Solving Multi-output Constraints

Multi-output constraints introduce two new factors into the constraint satisfaction process: 1) constraints can have multiple outputs, and 2) variable output constraints have output determinant constraints that must be evaluated to determine a constraint's outputs. Multi-output constraints that have a fixed set of outputs can be handled with a straightforward modification to the standard nullification/reevaluation algorithm: when a constraint is invalidated, it marks all of its outputs invalid. This modification is a logical extension of the single-output case, in which each invalidated constraint invalidates its single output variable.

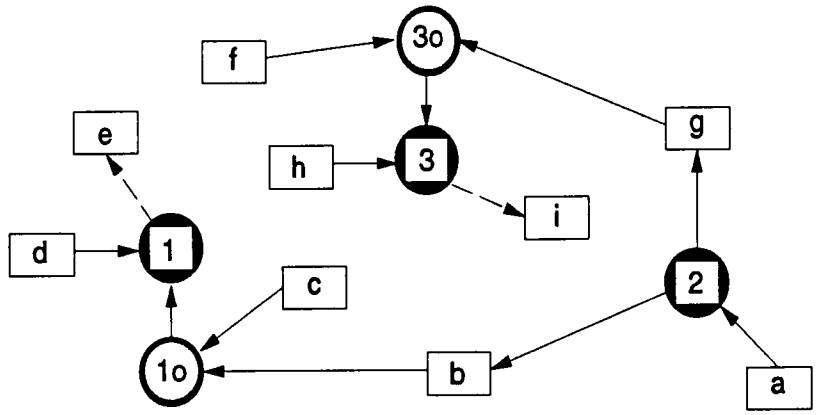
The variable output case requires more extensive modifications to the standard nullification/reevaluation algorithm. The principle modification is that output determinant constraints must be evaluated before all other constraints. These evaluations must be done first so that 1) variables are not erroneously assigned values by a multi-output constraint; and 2) constraints that depend on variables that should be assigned values by multi-output constraints are not evaluated prematurely. The first case can occur if a multi-output con-

straint is evaluated before its associated output determinant constraint (assuming that the output determinant constraint is invalid). Since the output determinant constraint is invalid, the multi-output constraint will assign to its old set of outputs, rather than its new set. Obviously this problem can be avoided if output determinant constraints are evaluated before the multi-output constraint they are associated with.

The second case can occur if a constraint (single-output or multi-output) is evaluated before an invalid output determinant constraint. If the constraint accesses a variable which will be included in the new set of outputs determined by the output determinant constraint, but which was not in the old set of outputs, then the constraint will prematurely access the variable.

Thus far we have argued that output determinant constraints must be evaluated before other constraints. However, a number of timing issues still must be resolved. First there is the question of whether the output determinant constraints are automatically evaluated at a time determined by the constraint satisfaction algorithm, or whether the programmer must manually instruct the constraint satisfier to evaluate them. Since we wanted to automate the constraint satisfaction process as much as possible, we decided to automatically evaluate the queued output determinant constraints. Second there is the question of when should these constraints be executed. There are two logical times to do it, immediately after a variable has been modified and constraints have been invalidated, or when the value of a variable is requested. Our implementation evaluates the output determinant constraints immediately. However, it is conceivable that the output determinant constraints could be lazily evaluated when the value of a variable is actually demanded, but this strategy would increase the amount of bookkeeping the constraint solver performs, because any access to a variable must first check whether there are any invalid output determinant constraints. The constraint solver would have to evaluate all such invalid constraints to determine whether the demanded variable has become the target of a multi-output constraint.

Another complication that arises from variable output constraints is nondeterminism.



Initial values of variables:

a = 1	f = 0
b = 6	g = 3
c = -20	h = -10
d = -10	i = -40
e = -10	

Constraints:

1o = if (b + c) < 0 then e else f
1 = d
2 = 6 * a, 3 * a
3o = if (f + g) < 10 i else c
3 = 4 * h

Key:

= multi-output constraint
= output determinant constraint
= an output of a multi-output constraint
= variable

Figure 4.2: An example of a constraint graph that can be nondeterministic because of variable output constraints.

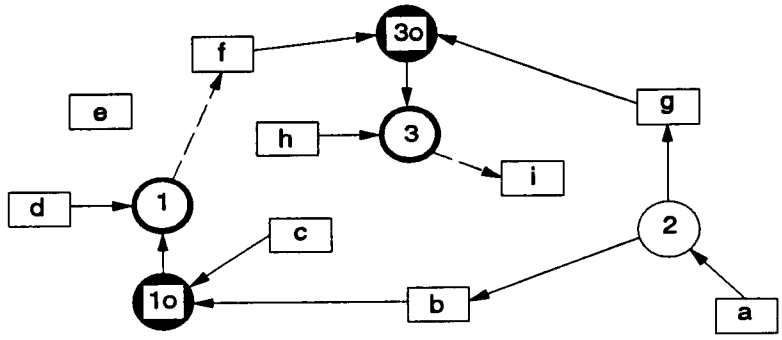
In the fixed output case (i.e., a constraint's outputs are fixed in advance), if constraints are always evaluated in topological order and the constraint graph is acyclic, then the same final constraint graph will always be reached, regardless of the order in which unrelated constraints are evaluated. In the multi-output, variable case, different constraint graphs can result, even if constraints are evaluated in topological order. The reason is that seemingly unrelated output determinant constraints can become related as a result of their evaluation. For example, consider the constraint graph shown in Figure 4.2. Suppose that the variable a is changed from 1 to 5. The first equation in topological order is the multi-output equation labeled *2*, and it changes b to 30 and g to 15. The output determinant constraints *1o* and

3o are next in topological order. Since neither depends on the other, the two constraints can be evaluated in either order. If *1o* is evaluated first, *1o* changes the output of equation *1* to *f*. Now the output determinant constraint *3o* depends on constraint *1*, so constraint *1* is evaluated next. Constraint *1* sets *f* to -10, which results in *3o* retaining *i* as the output for constraint *3*. The resulting constraint graph is shown in Figure 4.3 (a).

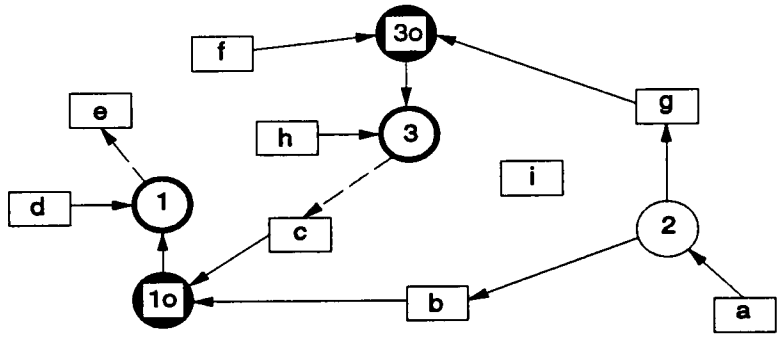
Suppose that instead of evaluating *1o* first, *3o* is evaluated first. *3o* changes the output of equation *3* to *c*. Now the output determinant constraint *1o* depends on constraint *3*, so constraint *3* is evaluated next. Constraint *3* sets *c* to -40, which results in *1o* retaining *e* as the output for constraint *3*. The resulting constraint graph is shown in Figure 4.3 (b). Notice that the two graphs are different, even though both graphs were produced by evaluating constraints in topological order.

Instead of arbitrarily breaking ties by evaluating one output determinant constraint before another, one could try evaluating all unrelated constraints "simultaneously" (i.e., evaluate all unrelated output determinant constraints before updating the constraint graph). This approach is used by Rendezvous [Hill 92]. This approach leads to yet a third graph shown in Figure 4.3 (c). Evaluating *1o* sets constraint *1*'s output to *f* and evaluating *3o* sets constraint *3*'s output to *c*. In some sense this graph is the least desirable of the three, since it is cyclic.

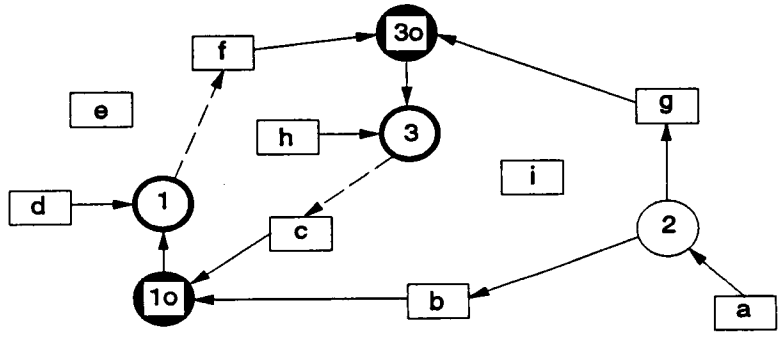
Which graph is "correct" depends on what the user expects. Clearly different outcomes can be achieved by employing different evaluation strategies. This problem of nondeterminism can be eliminated by not permitting output determinant constraints to depend on variables that are determined directly or indirectly by variable output constraints (output determinant constraints may still depend on multi-output constraints with fixed outputs and multi-output constraints may still depend on any type of multi-output constraint). If output determinant constraints are not permitted to depend directly or indirectly on variable output constraints, then output determinant constraints can only depend on constraints whose outputs cannot change. As long as these fixed output constraints are acyclic,



(a)



(b)



(c)

Figure 4.3: Three possible constraint graphs that can be obtained from evaluating the constraints in Figure 4.2. (a) results when constraint *1o* is evaluated before constraint *3o*. (b) results when constraint *3o* is evaluated before constraint *1o*. (c) is created when all unrelated output determinant constraints are evaluated simultaneously.

the same constraint graph will be reached, regardless of which order the output determinant constraints are evaluated in.

The algorithm that we present for satisfying multi-output constraints assumes that output-determinant constraints can depend on variables that are determined by multi-output constraints. Hence, our algorithm imposes no restrictions on the ordering of these two types of constraints. As a drawback, a constraint graph in our system can be nondeterministic. However, a counterbalancing concern to nondeterminism is that placing restrictions on certain types of constraints and not on others violates programming language design precepts of consistency and minimizing exceptions. In addition, it seemed that these nondeterminism problems are unlikely to arise in practice, so we decided to allow output determinant constraints to depend on variable output constraints. This decision has been validated both by our own experience and, independently, by users' experience with the Rendezvous system [Hill 93].

4.2.1 Data Structures

The data structures used for multi-output constraints, output determinant constraints, and output variables are as follows. Figure 4.4 pictorially illustrates the relationship between these three components.

Multi-output Constraint:

output-objects	either a pointer to the object the constraint outputs to (fixed output constraint) or an output determinant constraint that determines the set of output objects (variable output constraint)
output-slots	a list containing the output variables
formula	determines the values for the output variables
valid	indicates if the formula needs to be recomputed

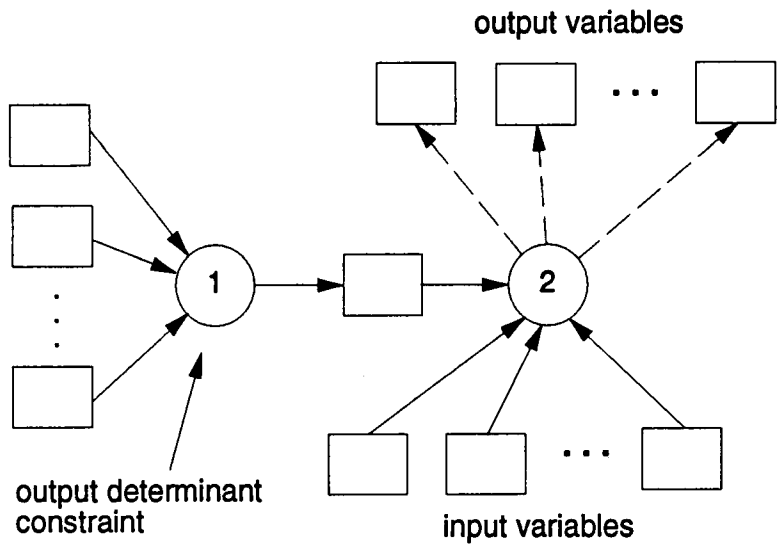


Figure 4.4: A pictorial description of a multi-output constraint that uses an output determinant constraint. The output determinant constraint labeled (1) specifies the output object(s). The multi-output constraint labeled (2) solves the formula and stores the values in the output variables. The output variables contain a pointer to the multi-output formula along with a cache value.

Output Determinant Constraint:

formula	determines the set of output objects
parent	pointer to its associated multi-output constraint
value	the last value that was computed
valid	indicates if formula needs to be recomputed

Output Variable:

formula	pointer to a multi-output formula
value	last value calculated for this variable by the formula
dependents	a list of formulas that reference this variable

4.2.2 The Constraint Solving Algorithm

The constraint satisfaction algorithm for multi-output constraints is a straightforward extension to the standard nullification/reevaluation algorithm. Informally, the algorithm may be described as follows:

1. When a variable is changed, any constraint that depends on that variable is invalidated. Invalidated output-determinant constraints are set aside on a queue. Formulas downstream from this constraint are not invalidated until after the output-determinant constraint has been evaluated.

2. Evaluate the queued output determinant constraints.

3. If the evaluation of one of the output determinant constraints in the previous step results in a different set of output objects (i.e., either objects have been added or removed) then all formulas downstream from the constraint are invalidated. As constraints are invalidated, they may cause other output determinant constraints to become invalidated. These newly invalidated output determinant constraints are added to a new list.

4. If no output determinant constraints are invalidated in the previous step, or a user specified limit has been reached on the number of constraint solving iterations then stop; otherwise, repeat step 2 using the new list of output determinant constraints. The user specified limit allows the output determinant constraints to iterate through several constraint solving cycles, so that the constraint network can reach a more satisfied state.

Figure 4.5 provides a pictorial description of the invalidate routine used to handle multi-output constraints. The decision to defer the invalidation of formulas downstream of output determinant constraints until all queued output determinant constraints are evaluated allows us to separate the invalidation and evaluation phases. This separation in turn, prevents infinite loops from arising if there are cycles in the constraint graph. If evaluation can be uncontrollably interweaved with invalidation, then two output determinant constraints that are part of the same cycle can give rise to an infinite loop in the following manner. When the first output determinant constraint is evaluated, it will cause the invalidation of the second output determinant constraint. The second output determinant constraint

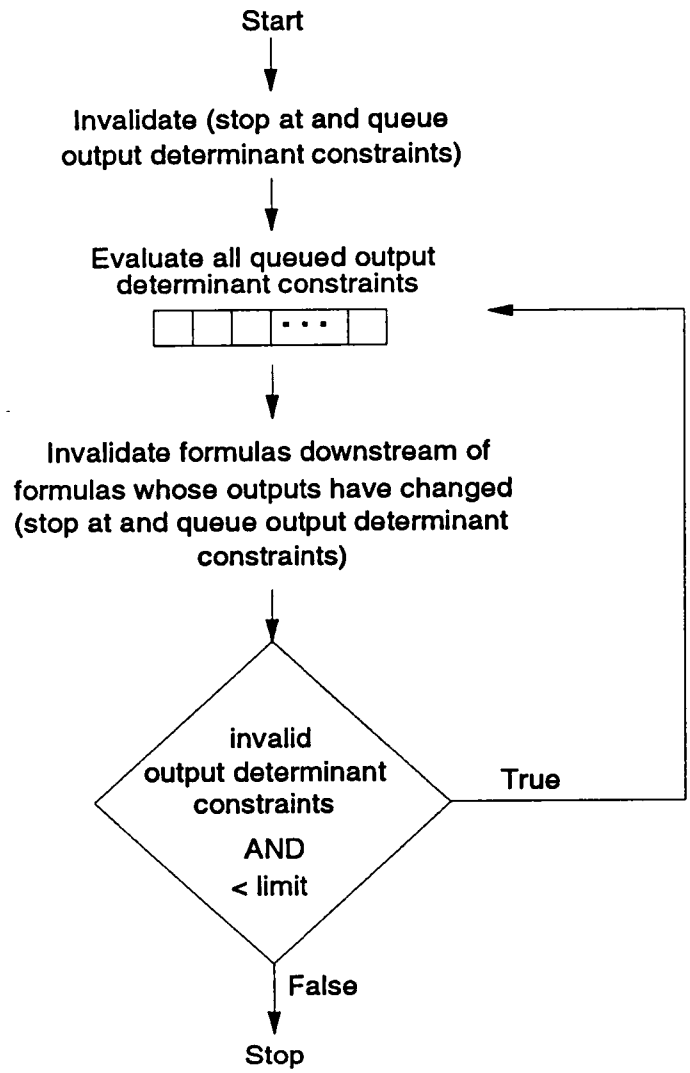


Figure 4.5: A diagram illustrating the invalidate routine used for multi-output constraints.

```
ODCS = ∅       /* Output Determinant ConstraintS */
E-ODCS = ∅     /* Evaluated Output Determinant ConstraintS whose set of
                  output objects have changed. */
MAX-IT = specified by the user /* MAXimum ITerations of constraint solving */

invalidate-cn (constraint)
   counter = 1
   recursively-invalidate-cn (constraint)
   while (ODCS ≠ ∅) AND (counter <= MAX-IT) do
      counter = counter + 1
      old-odcs = ODCS
      ODCS = ∅
      while old-odcs ≠ ∅ do
         e = dequeue (old-odcs)
         old-output-objects = e.value
         e.valid = true
         evaluate (e)
         if old-output-objects ≠ e.value then
            adjust-pointers (e, old-output-objects)
            enqueue (e, E-ODCS)
      while E-ODCS ≠ ∅ do
         e = dequeue (E-ODCS)
         if e.parent.valid = true then
            recursively-invalidate-cn (e.parent)
```

Figure 4.6: The regular invalidate routine has been modified to handle multi-output constraints that can have a dynamic number of inputs and outputs. *adjust-pointers* removes the pointers from output variables that no longer obtain their value from constraint *e*, and adds pointers to the new set of output variables that will obtain their value from constraint *e*.

will then be evaluated, which will invalidate the first output determinant constraint. Then the first output determinant constraint will be reevaluated, and so on. If a fixed point is reached, then the loop will terminate. However, if a fixed point is not reached, the loop will never terminate. Since we wanted to be able to guarantee the termination of the constraint solver, we settled on the deferral strategy. A beneficial side-effect of this strategy is that the algorithms are simpler and more elegant than the algorithms that interweave evaluation and invalidation. Figures 4.6 - 4.9 formalize the invalidate and evaluate routines for multi-output constraints.

```
recursively-invalidate-cn (constraint)
  constraint.valid = false
  if constraint = "output determinant" then
    enqueue (constraint, ODCS)
  else          /* multi-output constraint */
    output-slots = constraint.output-slots
    output-objects = get-output-objects (constraint)
    for each object ∈ output-objects do
      for each slot ∈ output-slots do
        variable = object.slot
        for each formula ∈ variable.dependents do
          if formula.valid = true then
            recursively-invalidate-cn (formula)
        variable.dependents = ∅
```

Figure 4.7: This routine recursively performs invalidation on multi-output constraints. *get-output-objects* returns a list containing the one output object for fixed output constraints, or the list of output objects for variable output constraints.

```
evaluate-multi-output-formula (formula)
  formula.valid = true
  list-of-values = evaluate (formula)
  /* Assign values to output variables. */
  output-slots = formula.output-slots
  output-objects = get-output-objects (formula)
  for each object ∈ output-objects do
    for each slot ∈ output-slots do
      object.slot.value = pop (list-of-values)
```

Figure 4.8: This routine evaluates a multi-output formula and assigns the results to the output variables. *get-output-objects* returns a list containing the one output object for fixed output constraints, or the list of output objects for variable output constraints.

```
get-value (variable)
   demanding-formula = top (formula-stack)
   if demanding-formula then
      variable.dependents = variable.dependents ∪ {demanding-formula}
   if variable.formula.valid = false then
      push (variable.formula, formula-stack)
      evaluate-multi-output-formula (variable.formula)
   return (variable.value)
```

Figure 4.9: This routine gets the value of a variable that is the output of a multi-output constraint. This routine also performs dependency maintenance.

4.2.3 An Example

To illustrate how the invalidate routine for multi-output constraints operates, we will trace through this routine using the constraints shown in Figure 4.10. Assume that all constraints in this figure are initially marked valid and that the user changes the value of variable *var1*.

Step 1 of the invalidate routine invalidates constraint *1* and then queues the output determinant constraint *2a*. The recursive invalidation process stops with constraint *2a* since it is an output determinant constraint. The algorithm now advances to step 2. Step 2 evaluates the output determinant constraint *2a* and advances to step 3. Assume that the evaluation of constraint *2a* results in *rect2* being its new value. Since the set of output objects for constraint *2a* has changed, step 3 invalidates all formulas downstream. This results in the invalidation of formulas *2b*, *3a*, and *5*. Constraint *3a* is invalidated because it depends on the *left* slot of *rect2*. Constraint *5* is invalidated because it depends on the *top* slot of *rect2*. The invalidation process again stops with constraint *3a* since it is an output determinant constraint. Constraint *3a* is added to a new list of output determinant constraints. The algorithm now advances to step 4. Figure 4.11 shows the new constraint graph.

Step 4 calls step 2 using the new list of output determinant constraints (i.e., *3a*). We are assuming that the user specified limit has not been reached. Step 2 evaluates the output determinant constraint *3a* (thus demanding and evaluating constraint *2b* as well) and

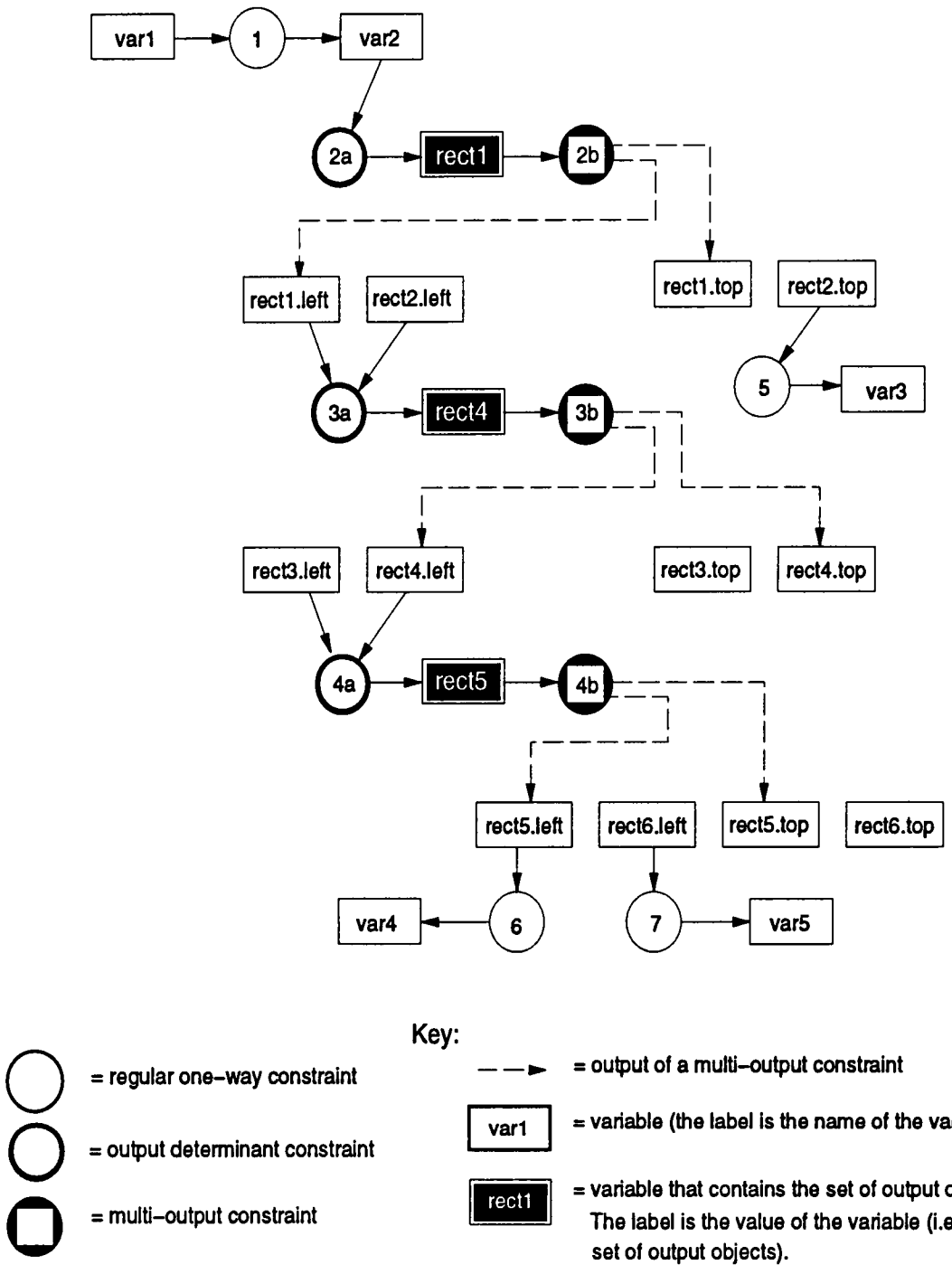


Figure 4.10: A constraint graph that incorporates: regular one-way constraints, output determinant constraints, and multi-output constraints.

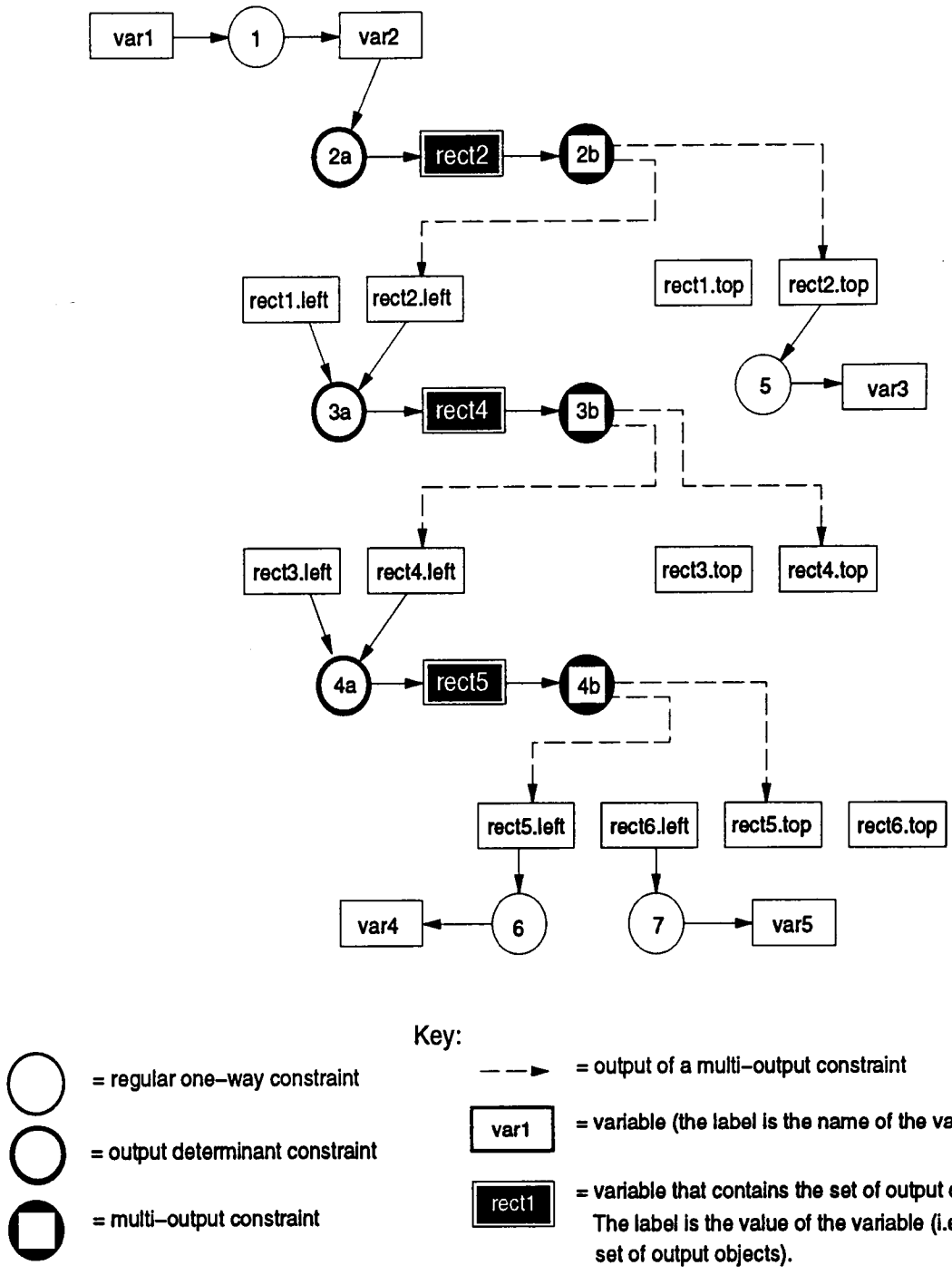


Figure 4.11: The constraint graph that is produced when the evaluation of the output determinant constraint *2a* results in *rect2* being its new value.

advances to step 3. Assume that the evaluation of constraint *3a* results in *rect3* being its new value. Since the set of output objects for constraint *3a* has changed, step 3 invalidates all formulas downstream. This results in the invalidation of formulas *3b* and *4a*. The recursive invalidation process again stops with constraint *4a* since it is an output determinant constraint. The constraint *4a* is added to a new list of output determinant constraints. The algorithm now advances to step 4. Figure 4.12 shows the new constraint graph.

Step 4 again calls step 2 if the user specified limit has not been reached. We are assuming that it has not been reached. Step 2 evaluates the output determinant constraint *4a*. This time assume that the set of output objects does not change. Therefore, since there has been no change in the set of output objects we do not invalidate any formulas downstream. Hence, neither formula *6* nor formula *7* will be invalidated. The invalidate routine now terminates since there are no more invalidated output determinant constraints. Since no changes were made, the final constraint graph is also shown in Figure 4.12.

4.2.4 Time Complexity

The following terms will be used in evaluating the time complexity for the algorithms presented in this chapter. This terminology has been adopted from [Vander Zanden 94] and [Hudson 91].

directly_affected $_i$ - the set of variables that have been directly modified during iteration i of the invalidate routine.

affected $_i$ - the set of formulas that directly or indirectly depend on the variables in *directly_affected*$_i$.

needed_by_affected $_i$ - the set of input edges to the formulas in *affected*$_i$.

The terms above are illustrated in Figure 4.13. An iteration represents one pass through the invalidate routine and one pass through the evaluate routine. When using lazy evaluation, not all of the affected constraints will necessarily be evaluated during iteration

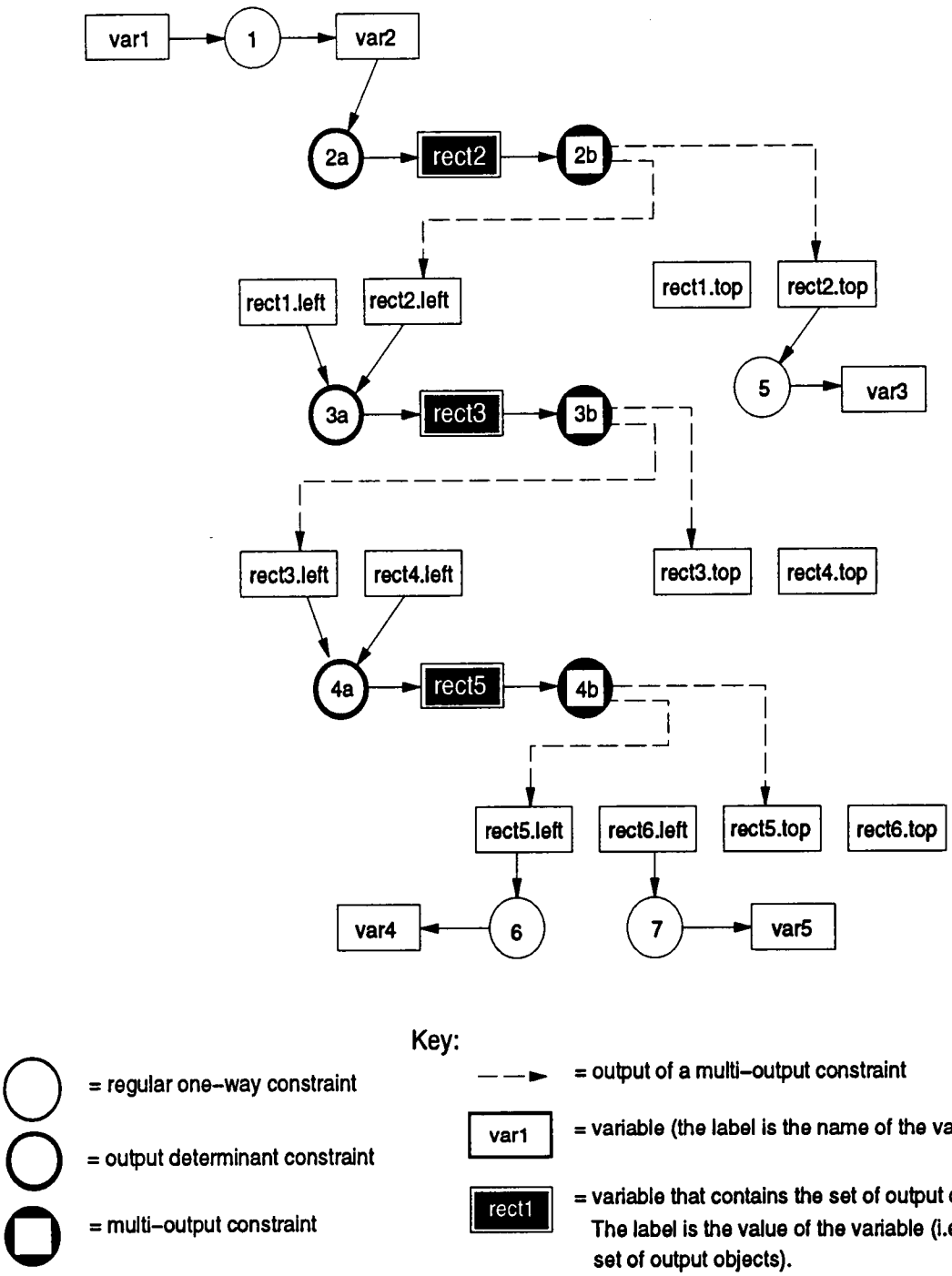


Figure 4.12: The constraint graph that is produced when the evaluation of the output determinant constraint *3a* results in *rect3* being its new value.

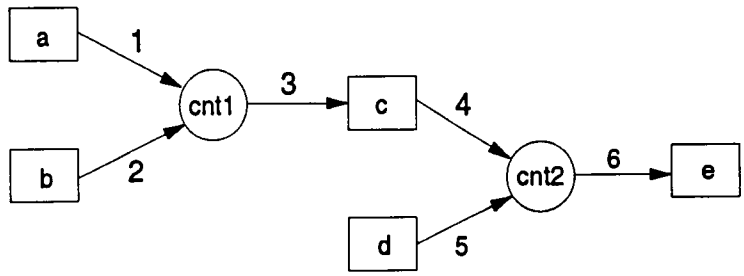


Figure 4.13: The constraints $c = a + b$ and $e = c + d$ are represented using a directed graph. If a is modified then $directly\_affected_i = \{a\}$, $affected_i = \{cnt1, cnt2\}$, and $needed\_by\_affected_i = \{1, 2, 4, 5\}$.

i. However, we will use an amortized analysis that charges the cost of evaluating all the constraints in $affected_i$ to iteration i.

The total time complexity for the multi-output constraint solving algorithm is determined by 1) the number of constraints that must be invalidated, 2) the number of constraints that must be evaluated, and 3) the number of variables in the constraints. We are assuming that the user specified constraint formulas terminate and have a time complexity of O(1). Hence, the time complexity reported here is the cost of constraint scheduling and the overhead of constraint execution (i.e., establishment of dependencies and invoking the evaluation functions for each formula).

The invalidate routine for multi-output constraints queues output determinant constraints. The evaluate routine then evaluates these constraints. When the set of output objects for an evaluated output determinant constraint changes, all formulas downstream are invalidated when the evaluation phase has completed. If the user specified limit is n, then n iterations might be required to evaluate the output determinant constraints.

The terminology used in reporting the time complexity for multi-output constraints will be subscripted as follows: $affected_{reg\_i}$ represents the regular constraints (single and multi-output constraints) that are affected, $affected_{od\_i}$ represents the output determinant constraints that are affected, $needed\_by\_affected_{reg\_i}$ represents the set of dependencies needed to compute the constraints in $affected_{reg\_i}$, and $needed\_by\_affected_{od\_i}$ represents the set of

dependencies needed to compute the constraints in $affected_{od\_i}$.

We first compute the time complexity for a single iteration and later compute the time complexity for the entire algorithm. The time complexity for one iteration is a summation of the invalidate and evaluate routines. The invalidation algorithm follows edges and thus it may attempt to visit a constraint more than once (although if the constraint has been visited, it will not proceed past that constraint). Thus the cost of invalidation is the number of edges that are traversed. The set of edges traversed is a subset of the edges in $needed\_by\_affected_{reg\_i}$ + $needed\_by\_affected_{od\_i}$. Hence, the time complexity of invalidation is bounded by $O(|needed\_by\_affected_{reg\_i}| + |needed\_by\_affected_{od\_i}|)$. The evaluate routine requires that $|affected_{reg\_i}| + |affected_{od\_i}|$ constraints be evaluated. Since the cost of evaluating these constraints is O(1), the cost of evaluating constraints is $O(|affected_{reg\_i}| + |affected_{od\_i}|)$. Finally, each input variable in a constraint may be accessed in O(1) time. Thus the cost of accessing all of the input variables in $affected_{reg\_i}$ and $affected_{od\_i}$ is $O(|needed\_by\_affected_{reg\_i}| + |needed\_by\_affected_{od\_i}|)$. Summing up, the worst case running time for one iteration is:

$$O(|affected_{reg\_i}| + |affected_{od\_i}| + |needed\_by\_affected_{reg\_i}| + |needed\_by\_affected_{od\_i}|)$$

In actual implementations, the number of variables per constraint is usually bounded by some small constant k, in which case $needed\_by\_affected_{reg\_i}$ is bounded by $k|affected_{reg\_i}|$ and $needed\_by\_affected_{od\_i}$ is bounded by $k|affected_{od\_i}|$. Thus, the time complexity for a single iteration reduces to $O(|affected_{reg\_i}| + |affected_{od\_i}|)$.

We now compute the time complexity for the entire algorithm. Recall that the algorithm continues until a user specified limit has been reached or no output determinant constraints are invalidated in the previous iteration. In the worst case, the algorithm must continue until the user specified limit n has been reached, and hence n iterations of constraint solving

are required. Therefore, the worst case running time for the entire algorithm is:

$$\sum_{i=1}^{n} affected_{reg\_i} + affected_{od\_i}$$

4.2.5 Proof of Correctness

This section proves that the integration of multi-output constraints with the conventional constraints is correct. Unless otherwise indicated, throughout this section the words "constraints" and "algorithm" refer to conventional, output determinant, and multi-output constraints, and those portions of the constraint solver that handle them. We first prove that any combination of these constraints terminates, and secondly that if the constraint graph is acyclic, then all constraints marked valid are satisfied when the algorithm finishes.

Theorem: If all constraint formulas terminate, then the constraint solving algorithm for multi-output constraints terminates.

Proof: One iteration of constraint solving is not always sufficient for solving multi-output constraints. Hence, to prove that the algorithm terminates, we must prove that 1) each iteration terminates, and 2) a finite number of iterations are performed. From these two steps, it follows that the algorithm terminates.

Each iteration terminates. The constraint solving algorithm for multi-output constraints is composed of an invalidate and evaluate routine. One iteration of the constraint solver requires one pass through the invalidate routine and one pass through the evaluate routine. The invalidate and evaluate routines do not call one another and thus they are not interweaved. Hence, to show that the algorithm terminates, we will need to prove that these two routines individually terminate.

The invalidate routine for multi-output constraints performs a depth first search marking constraints as invalid. The invalidate routine diverges slightly from a true depth-first search in that it queues output determinant constraints and does not invalidate

formulas downstream from these constraints. However, stopping the invalidation process early is comparable to treating a multi-output constraint as if it had no outgoing edges (i.e., no outputs). In effect the invalidation algorithm behaves as if a few edges have been removed from the graph. Since a depth-first search still terminates if any number of edges are removed from a graph, the invalidate routine will still terminate.

A constraint cannot be evaluated more than once during an evaluation pass, because 1) the first time the constraint is evaluated, it is marked valid, 2) the invalidate routine is not called from the evaluate routine and thus a constraint cannot be marked invalid by the evaluate routine, and 3) the evaluate routine will not evaluate a constraint if it is marked valid. Consequently, after the first evaluation, the evaluate routine will not evaluate the constraint again during that pass. Therefore, since a constraint can be evaluated at most once, the evaluate routine terminates.

In summary we have proven that the invalidate and evaluate routines individually terminate. Therefore, the constraint solving algorithm for multi-output constraints is guaranteed to terminate after each iteration.

Finite number of iterations. The algorithm terminates when a user specified limit has been reached, or when the invalidate routine does not create a new list of output determinant constraints. Consequently, the constraint solver executes a finite number of iterations.

Algorithm terminates. Since each iteration terminates and a finite number of iterations are performed, the algorithm terminates.

Theorem: If the constraint graph is acyclic and the invalidate routine did not create a new list of output determinant constraints, then all constraints marked valid are satisfied when the algorithm finishes.

Proof: To show that all constraints marked valid are satisfied we need to show that these constraints are properly evaluated and that their dependencies are recorded. A proof by contradiction will simultaneously show that both of these conditions are satisfied. The proof below does not go into detail concerning the correctness of input dependencies, since our approach uses the same technique that is used in Alphonse [Hoover 92]. A proof of correctness is presented in that paper. Instead, our proof focuses on the proper recording of output dependencies and the proper evaluation of constraints.

Assume that some constraint which is marked valid is not satisfied. Let c represent the first constraint in a chain of constraints that is marked valid but which is not satisfied. All of c's predecessors are correctly marked valid or invalid. There are a number of ways in which the constraint might have computed an incorrect value and we will show that they all lead to contradictions. First, its formula might have been incorrectly written, but the constraint algorithm has no control over the writing of the formula, so we assume that the formula is correct.

Second, the constraint might depend on a variable that is marked invalid. This would imply that the input was marked invalid after the constraint was evaluated but for some reason, the invalidate routine did not then proceed to the constraint and mark it invalid. Since input dependencies are correctly recorded, the invalidate routine knew the constraint depended on this input when the input was marked invalid. Consequently the invalidate routine would have marked the constraint invalid.[1] The fact that the constraint is marked valid means that the evaluate routine evaluated the constraint. Since the constraint depends on the input, the constraint would have demanded the input's value. The evaluate routine would have noticed that the input was

[1]It might seem that the invalidate routine could have deferred invalidating this constraint and through some oversight, never invalidated it. However, only constraints that directly depend on output determinant constraints can be deferred in this fashion. The theorem assumes that the invalidate routine did not create a new list of output determinant constraints, so all output determinant constraints are marked valid. Consequently, a constraint cannot depend on an invalid output determinant constraint, and thus, its invalidation cannot be deferred.

marked invalid and would have reevaluated the input and marked it valid. However, we assumed that the input was marked invalid, so we have arrived at a contradiction.

The third possibility is that the constraint was evaluated prematurely, before all of its inputs had their values determined correctly. An input might not have had its value correctly determined yet because 1) it was marked invalid at the time its value was requested, or 2) its value was determined by a multi-output constraint, but the output determinant constraint associated with the multi-output constraint had not yet been executed, and thus the input did not know that it was the target of a multi-output constraint. In the first case, the algorithm would have noticed that the input was marked invalid and would have first recomputed the value of the input. Thus the input's value would have been computed correctly before it was used by the constraint. In the second case, the output determinant constraint would have eventually been evaluated and computed a correct set of outputs (we assumed that c was the first constraint to be left unsatisfied and the output determinant constraint precedes c, so the output determinant constraint must be satisfied). The invalidate routine would then have invalidated all constraints downstream of the output determinant constraint, and thus would have marked c invalid (if the user specified limit on iteration were exceeded, the invalidation would not be performed, but we assumed that the user specified limit was not exceeded). However, since c is valid, we have arrived at a contradiction. The only remaining possibility is that c depends on an input whose value is incorrect (i.e., whose constraint is unsatisfied). But then c is not the first constraint in the chain to be left unsatisfied, which contradicts our assumption that c was the first such unsatisfied constraint.

In summary we have proven that all constraints are properly evaluated and that their dependencies are recorded. Therefore, when the constraint solving algorithm for multi-output constraints is finished all constraints marked valid are satisfied.

4.3 Solving Structural Constraints

When structural constraints are evaluated, the action-procedures are not executed immediately, but instead are queued and executed once all structure attributes have been brought up-to-date. This prevents an object from being deleted before another object that depends on it is able to evaluate its structure attribute. For example, suppose structural constraints are declared so that an arrow that is attached to a box should be deleted if the box is deleted. This could be accomplished by creating a constraint so that the arrow's *exist-p* slot depends on the box's *exist-p* slot. If the box is deleted before the arrow's *exist-p* slot can be evaluated, then the arrow's *exist-p* formula will get garbage when it tries to access the deleted box's *exist-p* slot.

All structural constraints are evaluated before any value constraints. The structure determines the value attributes, so it is necessary that the structure be completely determined before any value attributes are calculated. In general the action-procedures will bring the actual structure of the application into alignment with the structure computed by the constraints.

4.3.1 Data Structures

The data structures used for structural constraints requires that an action-procedure field be added to each variable as shown below.

Variable:

value	the value assigned to this variable, or when a formula is attached, this field represents the last value calculated by the formula
dependents	a list of formulas that reference this variable
action-procedure	the name of an action-procedure that will be called when the value of this variable is changed.
formula	an optional constraint can be attached to this variable

A global list of structural attributes must also be maintained plus an optional local list of structural attributes for each object. These lists allow the constraint solver to identify

structural attributes, so that they may be queued and evaluated before value attributes.

4.3.2 The Constraint Solving Algorithm

The algorithm for implementing structural constraints is as follows. The user first modifies the slots of one or more objects. If a modification either directly or indirectly modifies a structural attribute, then that variable is added to a list of structurally invalid attributes. At some point the user or an application requests that the constraints be updated. Since structural constraints may have side-effects, we decided it was better for the user to explicitly request that the constraints be updated instead of trying to determine when to do it automatically. When a request is made to update the structural constraints, the following steps are taken.

1. All structural attributes on the structurally invalid list are brought up-to-date. When there is a change in the value of a structural attribute, then its corresponding action-procedure is queued.

2. The queued action-procedures are executed. As these procedures are executed, they may invalidate other structural or value attributes. When structural attributes are invalidated, a new structurally invalid attribute list is created, and the invalidated variable is added to this list.

3. If no structural slots are invalidated in the previous step, or a user specified limit has been reached on the number of iterations of constraint solving, then stop; otherwise, repeat steps 1 and 2 using the new list of structurally invalid attributes. The user specified limit allows the structural constraints and action-procedures to iterate through several constraint solving cycles, so that the constraint network can reach a more satisfied state.

Figure 4.14 provides a pictorial description of the update algorithm used to handle structural constraints. The evaluation of the structural attributes (step 1) must be performed

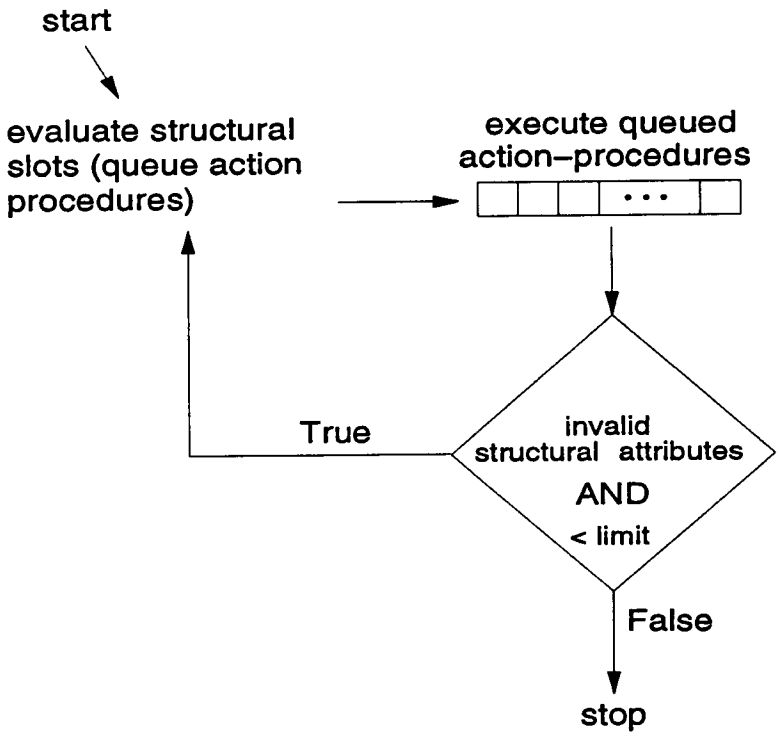


Figure 4.14: A diagram illustrating the algorithm for solving structural constraints.

```
execute-queued-action-procedures ()
   while QAP ≠ ∅ do
      procedure = pop (QAP)
      execute (procedure)
```

Figure 4.15: This routine executes all the queued action-procedures.

using eager evaluation. This is because the structural attributes determine the constraint network for the value attributes. Figures 4.15 - 4.17 formalize the invalidate and evaluate routines for structural constraints.

```
QAP = ∅        /* Queued Action-Procedures */
ISA = ∅        /* Invalid Structural Attributes */
MAX-IT = specified by the user   /* MAXimum ITerations of constraint solving */
DEFAULT-STRUCTURAL-ATTRIBUTES = global list of structural variables

update ()
   counter = 1
   while (counter <= MAX-IT) AND (ISA ≠ ∅) do
      counter = counter + 1
      temp-list = ISA
      ISA = ∅
      for each attribute ∈ temp-list do
         old-value = attribute.value
         new-value = get-value (attribute)          /* bring attribute up-to-date */
         if attribute.action-procedure ≠ ∅ AND old-value ≠ new-value then
            QAP = QAP ∪ attribute.action-procedure
      execute-queued-action-procedures ( )
```

Figure 4.16: Update brings all structural attributes up-to-date and then executes all queued action-procedures. *get-value* evaluates the attribute's formula.

```
slot-has-been-invalidated (slot)
   if structural-attribute (slot) then
      ISA = ISA ∪ {slot}
```

Figure 4.17: This routine is called when a slot is invalidated. ***structural-attribute*** returns true if *slot* is a structural attribute, otherwise it returns false.

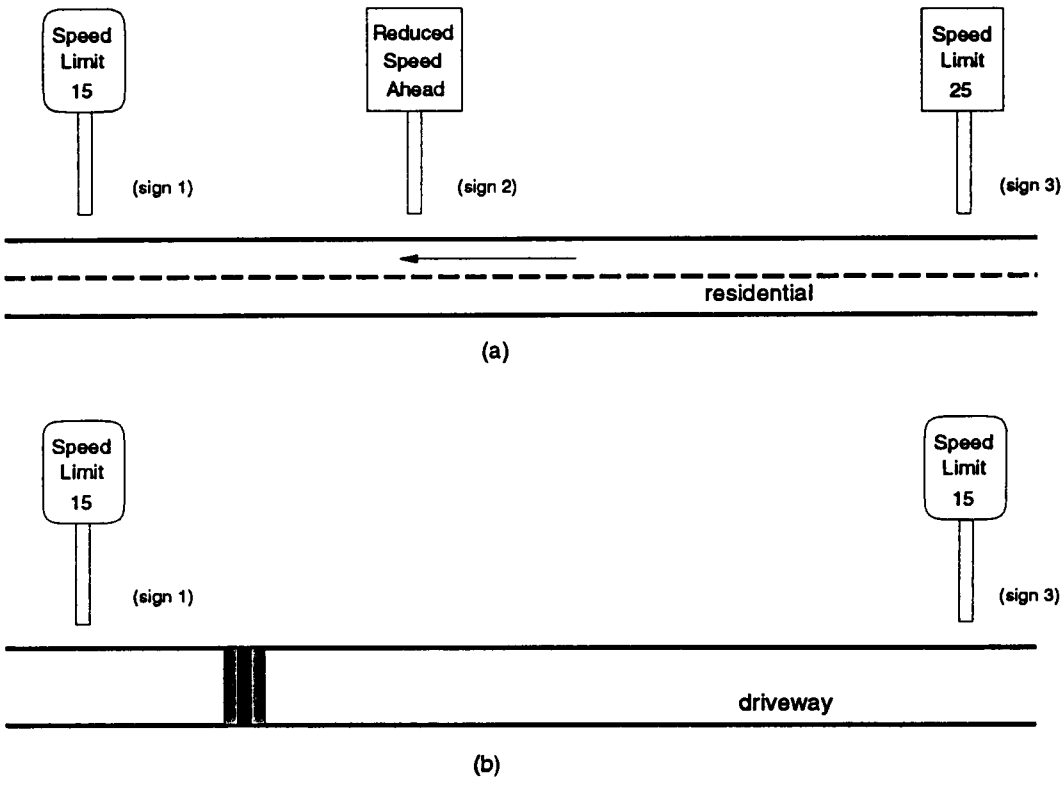


Figure 4.18: A road layout system has been constructed that uses structural constraints and action-procedures. In this example the road's type has been changed from residential to driveway. As a result, the structural constraints and action-procedures automatically 1) enforce a maximum speedlimit of 15 miles per hour, 2) change the text on *sign3*, 3) change the shape of *sign3*, 4) delete the "Reduced Speed Ahead" sign, 5) create a speedbump, and 6) delete the lane divider.

4.3.3 An Example

To illustrate how the constraint solver handles structural constraints, we will trace through a portion of a road layout system. This system is illustrated in Figure 4.18. The structural constraints and action-procedures in this example 1) change the shape of speedlimit signs, 2) delete "Reduced Speed Ahead" signs when the speed limit between two signs does not change, 3) create speedbumps, and 4) delete lane dividers when speedbumps are greater than 2 inches. Value constraints are used to 1) enforce a maximum speedlimit on a road, and 2) change the text on speedlimit signs.

In this example *type*, *speedlimit*, *exist-p*, *speedbump-p*, and *lane-dividers-p* are defined as

structural attributes. The objects and structural constraints that are used in this example are as follows:

```
road1
   lane-divider = "dashed"
   type = residential
   speedbump-p = (type = driveway)
   speedbump-p.action-procedure = SPEEDBUMP

sign1
   road = road1
   speedlimit
      case road.type
         primary :  max (55, speedlimit)
         secondary :  max (35, speedlimit)
         residential :  max (25, speedlimit)
         driveway :  max (15, speedlimit)
   speedlimit.action-procedure = CHANGE-SIGN-SHAPE
   string = "SPEED LIMIT" + string(speedlimit)

sign2
   string = "Reduced Speed Ahead"
   next-sign = sign1
   prev-sign = sign3
   exist-p = prev-sign.speedlimit > next-sign.speedlimit
   exist-p.action-procedure = EXISTENCE

sign3
   road = road1
   speedlimit
      case road.type
         primary :  max (55, speedlimit)
         secondary :  max (35, speedlimit)
         residential :  max (25, speedlimit)
         driveway :  max (15, speedlimit)
   speedlimit.action-procedure = CHANGE-SIGN-SHAPE
   string = "SPEED LIMIT" + string(speedlimit)

speedbump
   lane-dividers-p = (height < 2)
```

```
lane-dividers-p.action-procedure = LANE-DIVIDERS
```

The action-procedures used in this example are defined below.

```
action-procedure CHANGE-SIGN-SHAPE (sign speedlimit)
  destroy (sign)
  case speedlimit
    55 :  new-shape = create (square)
    35 :  new-shape = create (diamond)
    25 :  new-shape = create (rectangle)
    15 :  new-shape = create (roundtangle)
  sign = new-shape


action-procedure EXISTENCE (sign exist-p)
  if sign.exist-p = false
    destroy sign

action-procedure SPEEDBUMP (road speedbump-p)
  if (speedbump-p = true) then
    create speedbump with height = 2

action-procedure LANE-DIVIDERS (road lane-dividers-p)
  if (lane-dividers-p = false) then
    road.lane-divider = null
```

We now trace through the constraint solving algorithm using this example as input. We assume that *sign1* has an initial speedlimit of 15, *sign3* has an initial speedlimit of of 25, and the user has changed the value of the variable *road1.type* from *residential* to *driveway.* As a result of this change, the attributes *road1.speedbump-p*, *sign1.speedlimit*, *sign2.exist-p*, and *sign3.speedlimit* are added to the list of structurally invalid attributes. The user now requests that the structural constraints be updated.

Step 1 of the algorithm evaluates these structural attributes and queues their action-procedures when there is a change in the value of a structural attribute. The value of *sign1.speedlimit* does not change so its action-procedure is not queued. However, the value of *sign2.exist-p* does change so its action-procedure EXISTENCE is queued. Similarly, *road1.speedbump-p* and *sign3.speedlimit* change, so the action-procedures SPEEDBUMP and

CHANGE-SIGN-SHAPE are queued. The algorithm now advances to step 2.

Step 2 executes the action-procedures. The action-procedure EXISTENCE deletes *sign2*. Similarly, the action-procedure CHANGE-SIGN-SHAPE changes the shape of sign3 to a round-tangle, and the action-procedure SPEEDBUMP creates a speedbump. As a result of creating the speedbump the variable *speedbump.lane-dividers-p* is changed. Consequently, this variable is added to a new list of structurally invalid attributes. The algorithm now advance to step 3.

Assuming the user specified limit has not been reached, we now call step 1 again using the new list of structurally invalid attributes. This list contains the attribute *speedbump.lane-dividers-p*. Step 1 brings this structural attribute up-to-date and queues its action-procedure LANE-DIVIDERS. Step 2 executes the action-procedure LANE-DIVIDERS. This action-procedure removes the lane dividers since the height of the speedbump is not less than 2 inches. Step 3 is now reached. Since there were no structural attributes invalidated in the previous iteration, the constraint solver stops.

4.3.4 Time Complexity

The total time complexity for the structural constraint solving algorithm is determined by 1) the number of constraints that must be invalidated, 2) the number of constraints that must be evaluated, 3) the number of variables in the constraints, and 4) the number of action-procedures that must be queued and later executed. We are assuming that the user specified constraint formulas and action-procedures terminate and have a time complexity of O(1). Hence, the time complexity reported here is the cost of constraint scheduling and the overhead of constraint execution (i.e., establishment of dependencies and invoking the evaluation functions for each formula).

The evaluate routine for structural constraints first brings all structural attributes up-to-date, queuing all action-procedures when there is a change in value of the structural attribute. As these procedures are executed, they may invalidate other structural or value

attributes which may necessitate another round of structural constraint satisfaction. An iteration represents one pass through the invalidate routine, one pass through the evaluate routine, and one round of executing action-procedures. The number of iterations is bounded by a user specified limit which we will call n.

In addition to the terms defined in Section 4.2.4, the term $action\_procedures_i$ will also be used in reporting the time complexity for structural constraints. This term represents the set of action-procedures that are either queued or executed during iteration i.

We first compute the time complexity for a single iteration and later compute the time complexity for the entire algorithm. The invalidation algorithm follows edges and thus it may attempt to visit a constraint more than once (although if the constraint has been visited, it will not proceed past that constraint). Thus the cost of invalidation is the number of edges that are traversed. The set of edges traversed is a subset of the edges in $needed\_by\_affected_i$. Hence, the time complexity of invalidation is bounded by $O(|needed\_by\_affected_i|)$. The evaluate routine requires that $|affected_i|$ constraints be evaluated. Since the cost of executing these constraints is O(1), the cost of evaluating constraints is $O(|affected_i|)$. Each input variable in a constraint may be accessed in O(1) time. Thus the cost of accessing all of the input variables in $affected_i$ is $O(|needed\_by\_affected_i|)$. Finally, we must queue and execute $|action\_procedures_i|$ action-procedures. Since the cost of executing these action-procedures is O(1), the total cost of executing action-procedures $O(|action\_procedures_i|)$. Summing up, the worst case running time for one iteration is:

$$O(|affected_i| + |needed\_by\_affected_i| + |action\_procedures_i|)$$

In actual implementations, the number of variables per constraint is usually bounded by some small constant k, in which case $needed\_by\_affected_i$ and $action\_procedures_i$ are bounded by $k|affected_i|$. Thus, the time complexity for a single iteration reduces to $O(|affected_i|)$.

We now compute the time complexity for the entire algorithm. Recall that the algo-

rithm continues until a user specified limit has been reached or no structural attributes are invalidated in the previous iteration. In the worst case, the algorithm continues until the user specified limit has been reached, and hence n iterations may be required. Therefore, the worst case running time for the entire algorithm is:

$$\sum_{i=1}^{n} affected_i$$

4.3.5 Proof of Correctness

This section proves that the integration of structural constraints with the the conventional constraints is correct. Unless otherwise indicated, throughout this section the words "constraints" and "algorithm" refer to conventional and structural constraints, and those portions of the constraint solver that handle them. We first prove that any combination of these constraints terminate, and secondly that all constraints are satisfied when the algorithm finishes.

Because action-procedures can be written that cause structural attributes to become invalid, the following restriction must be placed on action-procedures to ensure termination. An action-procedure cannot request the value of an instance variable that is invalid and whose computation either directly or indirectly requires a recomputation of a structural attribute. The reason is that a recomputation of a structural attribute could cause the constraint satisfaction routine to be recursively called. The number of potential recursive calls cannot be bounded because in each invocation, an action-procedure could cause the recomputation of a structural attribute, causing yet another recursive call.

Theorem: If all constraint formulas and action-procedures terminate, and action-procedures do not request the value of an instance variable that is invalid and whose computation either directly or indirectly requires a recomputation of a structural attribute, then the constraint solving algorithm for structural constraints terminates.

Proof: Like multi-output constraints, one iteration of constraint solving is not always suf-

ficient for solving structural constraints. This is because structural constraints can be invalidated while executing action-procedures. Therefore, the algorithm provides multiple iterations of constraint solving so that the constraint network can reach a more satisfied state.

Hence, to prove that the algorithm terminates, we must prove that 1) each iteration terminates, and 2) a finite number of iterations are executed. From these two steps, it follows that the algorithm terminates.

Each iteration terminates. The constraint solving algorithm for structural constraints is composed of an invalidate routine, an evaluate routine, and an execute action-procedures routine. The invalidate and evaluate routines do not call one another and thus are not interweaved. Hence, to show that the algorithm terminates, we will need to prove that these three routines individually terminate.

The invalidate routine for structural constraints again performs a depth first search marking constraints as invalid. Consequently, it terminates.

The evaluate routine for structural constraints is the same as the evaluate routine used for the conventional constraints except that 1) it requests the values of structural attributes before value attributes, and 2) queues action-procedures. The order in which variables are requested does not affect the termination of the conventional evaluate algorithm. Similarly, the act of queuing action-procedures has no effect on the constraint graph (the procedures are queued, not executed) and thus does not affect termination. Hence, the evaluate routine terminates.

The routine that executes the action-procedures terminates because 1) they cannot cause the recomputation of any structural attribute, and thus the constraint satisfaction routine cannot be recursively called, and 2) they can demand the values of invalid value attributes that do not depend on invalid structural attributes, but the evaluation of value attributes is guaranteed to terminate.

In summary we have proven that the invalidate, evaluate, and execute action-procedure routines terminate. Therefore, the constraint solving algorithm for structural constraints is guaranteed to terminate after each iteration.

Finite number of iterations. The algorithm terminates when a user specified limit has been reached, or when the evaluate routine does not create a new list of structurally invalid attributes. Thus a finite number of iterations are executed.

Algorithm terminates. Since each iteration terminates and a finite number of iterations are performed, the algorithm terminates.

Theorem: If there are no cycles in the constraint graph and the evaluate routine did not create a new list of structurally invalid attributes, then all constraints marked valid are satisfied when the algorithm finishes.

Proof: To show that all constraints marked valid are satisfied we will prove that for each constraint 1) if any of the inputs to the constraint were changed, then the constraint was reevaluated, and 2) the constraint was not evaluated until all of its inputs had their correct values. In effect this is showing that that these constraints are properly evaluated and that their dependencies are recorded. In this proof we assume that the action-procedures create the correct structure if associated instance variables have the correct value.

We will prove this is by induction on the length of the longest chain of constraints to a variable. The basis case is a chain of length zero. That is the variable has no constraint. It is assumed that variables with no constraints have a correct value, so the basis case is trivially true.

The induction case assumes that all instance variables with chains of *n-1* constraints or less and which are marked valid have the correct values. We now prove that an instance variable which is marked value and has a longest chain of *n* constraints has

a correct value.

To prove the first claim, that the constraint was reevaluated if any of its inputs changed, we show that the constraint was marked invalid if one of its inputs changed. A constraint becomes unsatisfied only if one of the inputs it used in its last evaluation changes value. The technique that this algorithm uses for maintaining dependencies is the same technique that is used in Alphonse [Hoover 92]. The proof of correctness for this technique is given in that paper. Therefore there are dependencies from all of the inputs the constraint used in its last evaluation to that constraint. Hence, when one of the inputs changed, the invalidation algorithm must follow the dependencies to the constraint, and mark the constraint invalid. Thus the constraint must have been notified of any changes to its inputs. A constraint is marked valid only it has been evaluated. Consequently the fact that the constraint is now marked valid means that if any of its inputs changed, the constraint has been reevaluated since those changes.

We now prove the second claim, that the constraint was evaluated after all of its inputs had been assigned their correct values. When the constraint is evaluated, it demands the values of each of its inputs, which causes them to be evaluated if they are invalid. Consequently, the constraint was not evaluated until all of its inputs were known. The argument in the previous paragraph showed that if the inputs changed after the constraint was evaluated, it would have been notified and marked invalid. The fact that the constraint is marked valid indicates that its inputs did not change after it was evaluated. By the inductive hypothesis, these inputs must therefore have been correct, and thus the constraint must be satisfied.

In summary we have proven that all constraints are properly evaluated and that their dependencies are recorded. Therefore, when the constraint solving algorithm is finished all structural constraints marked valid are satisfied.

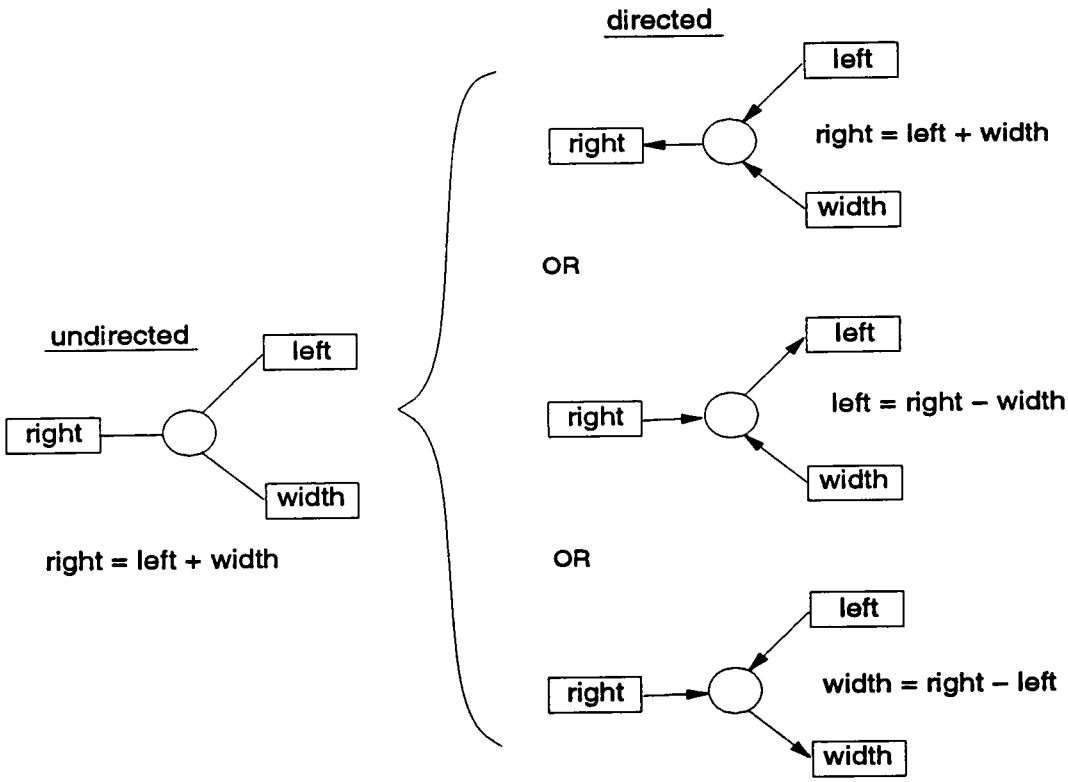


Figure 4.19: The planning phase takes an undirected graph (a), and produces one of the directed graphs in (b).

4.4 Solving Multi-way Constraints

There have been two published approaches for solving multi-way constraints. They are known as "propagation of degrees of freedom" and "propagation of known states". Both approaches perform a planning phase and then an evaluation phase. An example of the planning phase is illustrated in Figure 4.19. The planning phase determines a linear ordering on the constraints and the evaluation phase satisfies the equations. The DeltaBlue [Freeman 90] and SkyBlue [Sannella 92a] constraint solvers use "propagation of known states", and the CONSTRAINT algorithm [Vander Zanden 88] used "propagation of degrees of freedom". These algorithms were mentioned in Chapter 2.

The advantage of these approaches is that they place little responsibility on the programmer to satisfy the constraints. Programmers simply create constraints and possibly

add strengths indicating how strongly they want a constraint satisfied. The constraint solver then uses a planning algorithm to order the equations so that they form a lower triangular system of constraints. Such a system can be trivially satisfied by executing the constraints from top to bottom. Unfortunately, these algorithms assume a fixed number of inputs and outputs for each equation and they assume the inputs and outputs are known in advance. In addition, the "Blue" algorithms are guaranteed to find a lower triangular ordering on only a restricted set of constraint systems. After examining these algorithms, we decided that it was not feasible to extend them to handle equations containing a varying number of inputs and outputs, and equations in which the inputs and outputs cannot be determined in advance. The problem is that plans can be invalidated if just one new dependency is added during constraint evaluation (one additional edge could introduce a cycle). New edges can be added if a pointer variable changes, if the set of outputs changes, or if the set of inputs changes. If any of these situations arises, the constraint solver must ensure that the equations are still ordered acyclically, and if not, re-invoke the planner. Both of these operations, acyclicity checking and planning, are expensive operations, and it is unlikely that the constraint solver will be able to perform them fast enough to guarantee interactive performance.

Because of the difficulties associated with planning, our constraint solver takes a totally different approach that eliminates the planning phase associated with conventional multi-way constraint solvers. The elimination of the planning phase is made possible by the use of activation conditions and priorities. By employing activation conditions and priorities, the programmer performs the planning for the constraint solver. In particular, the planning should result in a one-way constraint graph. One-way constraint graphs can be satisfied using the nullification/reevaluation strategy described earlier in this chapter. Consequently, this technique will be used to satisfy our multi-way constraints. We will first describe the data structures used for the constraints and then present the constraint solving algorithm.

4.4.1 Data Structures for Multi-way Constraints

The data structures for instance variables, constraints, and activation conditions that are required to implement multi-way constraints are given below.

Instance Variable:

formulas	a list of the formulas attached to this variable
value	the last value that was computed
valid	indicates if a formula needs to be recomputed
dependents	a list of formulas that reference this instance variable

Formula:

formula	an arbitrary piece of code that determines the value of this formula
priority	indicates how strongly the user wants this formula to determine the value of the variable
act-cond	a boolean predicate that determines whether a formula is active or inactive (see "Activation Condition" below). Activation conditions are represented internally as conventional constraints.
parent	pointer to the formula's associated instance variable
value	the last value that was computed
valid	indicates if the formula needs to be recomputed

Activation Condition:

active	the value of the activation condition. This slot can contain either a constraint or a constant value like *true* or *false*.
parent	pointer to the activation condition's associated formula
value	the last value that was computed
valid	indicates if the activation condition needs to be recomputed

4.4.2 The Constraint Solving Algorithm

Our approach uses a modified mark/sweep algorithm. When a variable changes value, all formulas that depend directly or indirectly on that variable are invalidated, with three important exceptions. These exceptions are incorporated into the invalidate routine, which is given below.

1. Perform regular invalidation, with the following exceptions. When an exception is encountered, advance to step 2.

   (a) If a formula is attached to an instance variable that has already been visited during the current invalidation phase, then the formula is not marked invalid. This exception has two justifications. First, a variable should have only one active, highest priority formula that depends either directly or indirectly on another variable. Otherwise, it is not clear which formula should determine the variable. Consequently, if a slot has two formulas that depend on the same variable, the algorithm arbitrarily decides to satisfy the first constraint that is invalidated. The second justification is that for the most common type of multi-way constraint, a constraint between only two variables (e.g., an equality constraint), then this exception obviates the need for activation conditions on these constraints. For example, suppose the multi-way constraint $a = b$ is expressed using the constraints [a = b] and [b = a]. If the user changes a, the user obviously expects the change to be propagated to b. With unrestricted invalidation, both a and b would be invalidated if activation conditions were not used. This indiscriminate invalidation could cause the current value of b to be transmitted to a if *b's* value is requested before *a's* value. The proper use of activation conditions could prevent this indiscriminate invalidation by inactivating *a's* constraint. However, it would be tedious to write such activation conditions. The exception relieves the programmer of the responsibility for writing such activation conditions, because when the invalidation routine attempts to visit a, it will find that a has already been visited and will not invalidate a.

   (b) If the formula being invalidated is an activation condition, then the activation condition is queued. The formula that the activation condition is attached to, and formulas downstream are not invalidated. The reason invalidation does not proceed further is to avoid unnecessary invalidation in the event that the activa-

tion condition turns out to evaluate to false. Similarly, we do not immediately evaluate activation conditions, since the invalidate phase has not been completed and thus some of the variables that an activation condition depends on may not have been invalidated yet. In summary, by queuing the activation conditions, unnecessary invalidations and premature evaluations are avoided.

(c) If a formula being invalidated contains an activation condition, then the activation condition is queued. The formula is invalidated but not the slot that contains this formula. Furthermore, formulas downstream from this formula are not invalidated.

The reason for deferring formulas with activation conditions is that we do not know whether they will be active or not, and thus we should wait until their activation conditions are evaluated before proceeding further. Even if an activation condition is not invalid when we reach the formula, it is not guaranteed that the invalidation routine will not subsequently invalidate the activation condition. Thus it is safest to queue all activation conditions that the invalidate routine encounters.

(d) If the priority of a formula being invalidated is less than the priority of any other formula contained in the slot, then the formula is invalidated, but the slot that contains the formula is not invalidated. Formulas, downstream from this formula are not invalidated. The reason the slot and formulas downstream are not invalidated is because their value cannot possibly be determined by this formula. Hence, continuing to invalidate formulas downstream would serve no purpose.

2. Evaluate all queued activation conditions. If an activation condition evaluates to true, then it is added to a second queue.

3. For each activation condition in this second queue, find the formula controlled by this activation condition. If the formula is invalid and of highest priority then invalidate the slot that this formula is attached to as well as all formulas downstream, again considering the three exceptions listed above. When the invalidate routine reaches other formulas that have activation conditions, these new activation conditions are added to a new list.

4. If no activation conditions are invalidated in the previous step, or a user specified limit has been reached on the number of constraint solving iterations then stop; otherwise, repeat step 2 using the new list of activation conditions. The user specified limit allows the constraint solver to iterate through several constraint solving cycles, so that the constraint network can reach a more satisfied state.

Figure 4.20 provides a pictorial description of the invalidate routine used to handle multi-way constraints. When an invalid formula becomes active again, the slot that contains this formula is invalidated, as well as any formulas that directly or indirectly depend on this slot.

The evaluate routine uses the following algorithm in determining the value of a variable containing a multi-way constraint. If the instance variable is marked valid, then the value of that variable is returned. Otherwise, a list containing the highest priority, active, invalid constraints is created. The value of the first constraint on this list is returned. The returned value is also stored in the value field of the instance variable. All other constraints on this list are re-evaluated but their value is not used. These remaining constraints are re-evaluated to correctly setup all dependencies.

Like the previously published approaches our algorithm is incremental. This feature is very important when editing large applications, because a user's input typically only modifies a few objects leaving the rest unchanged. Figures 4.21 - 4.24 formalize the invalidate and evaluate routines for multi-way constraints.

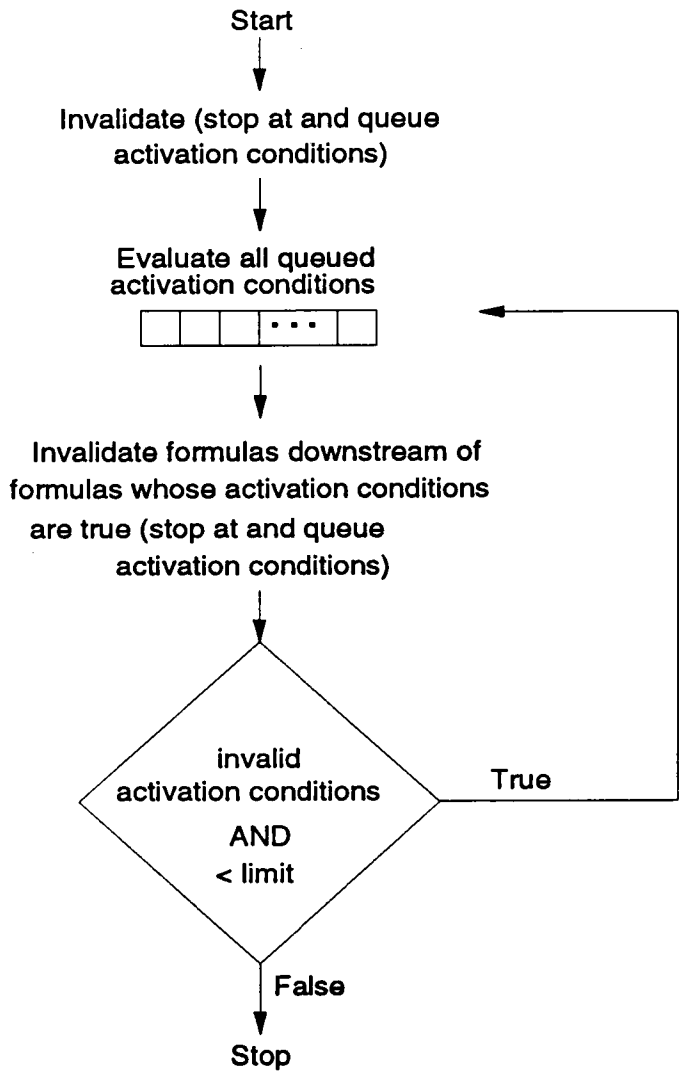


Figure 4.20: A simplified diagram illustrating the invalidate routine used for multi-way constraints.

```
QAC = ∅      /* Queued Activation Conditions */
ACET = ∅     /* Activation Conditions that Evaluated to True */
VISITED = ∅     /* The set of instance variables that have been processed */
                /* during the current iteration. */
MAX-IT = specified by the user   /* MAXimum ITerations of constraint solving */

invalidate-cn (formula)
   counter = 1
   VISITED = ∅
   recursively-invalidate-cn (formula)
   while (QAC ≠ ∅) AND (counter <= MAX-IT) do
      counter = counter + 1
      old-qac = QAC
      QAC = ∅
      while old-qac ≠ ∅ do
         e = dequeue (old-qac)
         if e.valid = false then
            evaluate (e)
         if e.active = true then
            enqueue (e, ACET)
      while ACET ≠ ∅ do
         e = dequeue (ACET)
         if e.parent.valid = false and highest-priority (e.parent) then
            instance-var = e.parent.parent
            instance-var.valid = false              /* invalidate instance variable */
            for each formula ∈ instance-var.dependents do
               if formula.valid = true then
                  recursively-invalidate-cn (formula)
            variable.dependents = ∅
```

Figure 4.21: This routine is a variation of the regular invalidate routine that has been modified to handle multi-way formulas. *highest-priority* returns true if the priority of the specified formula is greater than or equal to the priority of all other formulas that determine the instance variable's value.

```
recursively-invalidate-cn (constraint)
  constraint.valid = false
  if constraint = "activation condition then
       enqueue (constraint, QAC)
  else                    /* multi-way constraint */
    if highest-priority (constraint) then
      instance-variable = constraint.parent
      if instance-variable ∉ VISITED then
        VISITED = VISITED ∪ {instance-variable}
        if has-activation-condition (constraint) = false then
          instance-variable.valid = false
          for each formula ∈ instance-variable.dependents do
            if formula.valid = true then
              recursively-invalidate-cn (formula)
          instance-variable.dependents = ∅
      else
        enqueue (constraint.act-cond, QAC)
```

Figure 4.22: This routine recursively performs invalidation on multi-way constraints. The routine stops at lower priority formulas, and formulas that have an associated activation condition. *highest-priority* returns true if the specified formula's priority is greater than or equal to the priority of all other formulas that determine the value of the instance variable; otherwise it returns false. *has-activation-condition* returns true if an activation condition was specified for this *formula*.

```
evaluate-multi-way-formula (instance-variable)
    instance-variable.valid = true
    all-formulas = instance-variable.formulas
    active-formulas = find-active-formulas (all-formulas)
    active-highest-priority = find-highest-priority (active-formulas)
    determinant-formula = find-determinant-formula (active-highest-priority)
    active-invalid = find-invalid-formulas (active-formulas)
    for each c ∈ active-invalid do
        c.valid = true
        push (c, formula-stack)
        evaluate (c)
        pop (formula-stack)
    instance-variable.value = determinant-formula.value
    return (instance-variable.value)
```

Figure 4.23: This routine returns the value of an instance-variable that contains a multi-way constraint. *find-active-formulas* returns a list of all active formulas. *find-highest-priority* returns a list of the highest priority constraints. Invalid constraints are placed first in this list. *find-invalid-formulas* returns a list of all invalid formulas. *find-determinant-formula* returns the first invalid formula in the list. If an invalid formula does not exist, by default the first formula is returned.

```
get-value (variable)
    demanding-formula = top (formula-stack)
    if demanding-formula then
        variable.dependents = variable.dependents ∪ {demanding-formula}
    if variable.valid = false then
        evaluate-multi-way-formula (variable)
    return (variable.value)
```

Figure 4.24: This routine computes the value of an instance variable that contains a multi-way constraint. This routine also performs dependency maintenance.

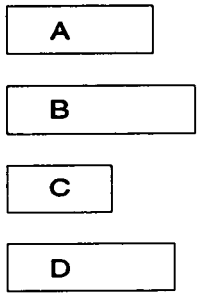


Figure 4.25: Four rectangles that will be left aligned using multi-way constraints.

4.4.3 An Example

To illustrate how the multi-way constraint solver operates consider the following example. Suppose we wished to left align the four rectangles shown in Figure 4.25. We begin by creating multi-way constraints connecting the left variables of these rectangles. For example, the equations [A.left = B.left], [B.left = C.left], and [C.left = D.left] would work. The equations are then converted to six one-way constraints, as shown in Figure 4.26. In our interface to the constraint system, the programmer would perform this conversion and present the six one-way constraints, rather than the three multi-way constraints, to the system.

We have initialized the value field for all instance variables to equal 15. The valid field for all instance variables and formulas are initially set to false. The first time the values of these instance variables are demanded all formulas are automatically evaluated and the dependencies are setup. Recall that all active, highest priority, invalid constraints are evaluated when the value of an invalid instance variable is demanded. Figure 4.27 shows the constraints after the values of the instance variables are demanded. As shown in this figure, the dependencies have now been determined and all formulas and instance variables are marked valid.

If the value of *A.left* is changed to 20, then all formulas and their corresponding instance variables that depend on *A.left* are marked invalid. *m2* is the only formula that depends on the variable *A.left*. Hence, formula *m2* and its corresponding instance variable *B.left* are

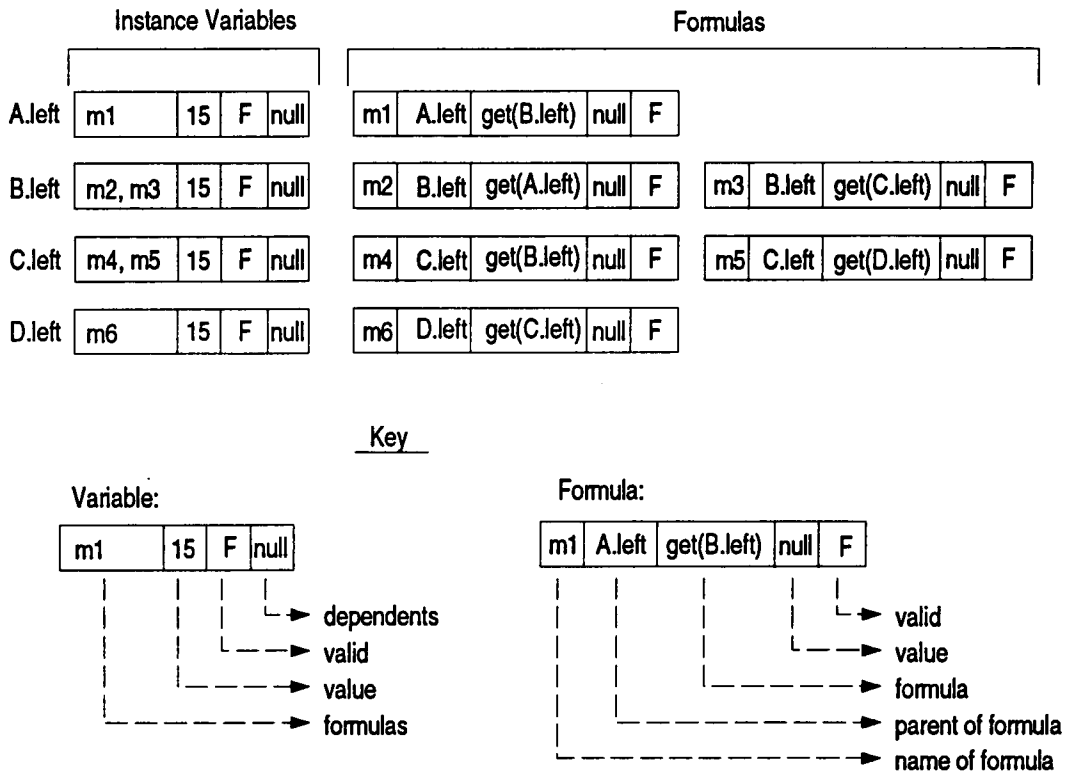


Figure 4.26: The initial values for the multi-way constraints. Since none of these formulas have activation conditions or priorities associated with them, these fields are not shown.

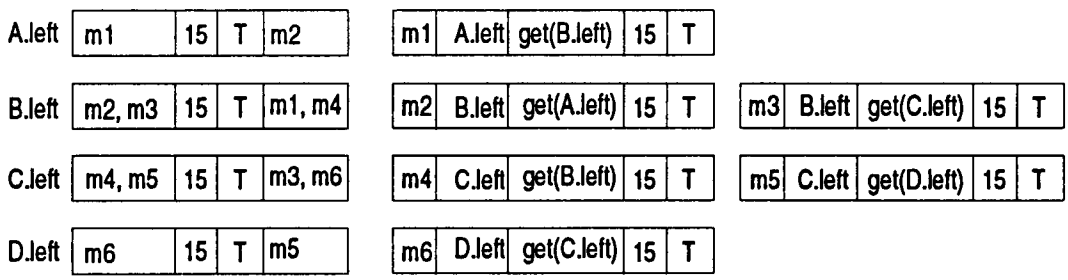


Figure 4.27: The values for the multi-way constraints after the dependencies have been determined.

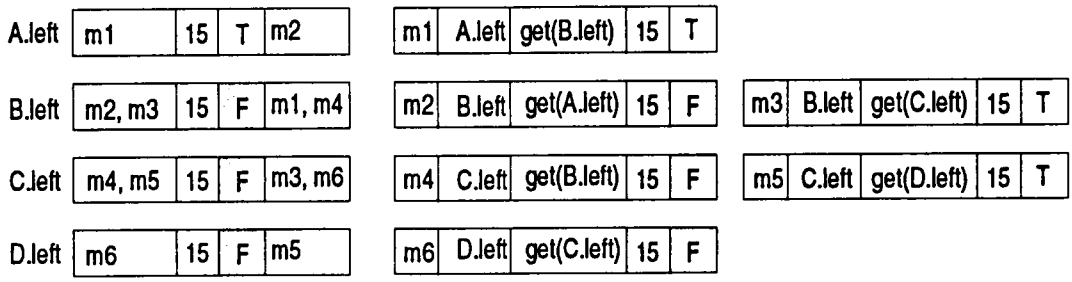


Figure 4.28: The gray rectangles represent values that are marked invalid when the left slot of rectangle A is changed.

marked invalid. The invalidate routine is then recursively called using the dependents for the variable *B.left*. *m1* and *m4* are the only formulas that depend on the variable *B.left*. Formula *m1* is not invalidated since *A.left* has already been processed by the invalidate routine during this iteration. However, formula *m4* and its corresponding instance variable *C.left* are invalidated. The invalidate routine is then recursively called using the dependents for the variable *C.left*. *m3* and *m6* are the only formulas that depend on the variable *C.left*. Formula *m3* is not invalidated since *B.left* has already been processed. However, formula *m6* and its corresponding instance variable *D.left* would be invalidated. The invalidate routine is then recursively called using the dependents for the variable *D.left*. *m5* is the only formula that depends on the variable *D.left*. However, formula *m5* is not invalidated since *C.left* has already been processed by the invalidate routine during this iteration.

Figure 4.28 shows the new constraint network. If an application now requests the values of all the left slots, perhaps because it was updating the window, then all of the invalid left slots are brought up-to-date. In this example, after the window update, the left slots of all four rectangles would equal 20, and the valid field for all formulas and instance variables would again be set to true "T".

4.4.4 Time Complexity

The total time complexity for the multi-way constraint solving algorithm is determined by 1) the number of constraints[2] that must be invalidated, 2) the number of constraints that must be evaluated, and 3) the number of variables in the constraints. We are assuming that the user specified constraint formulas and activation conditions terminate and have a time complexity of O(1). Hence, the time complexity reported here is the cost of constraint scheduling and the overhead of constraint execution (i.e., establishment of dependencies and invoking the evaluation functions for each formula).

An iteration represents one pass through the invalidate routine and one pass through the evaluate routine, recall that when the invalidate routine for multi-way constraints is finished, it may call the evaluate routine to evaluate activation conditions. Depending on the values of these activation conditions and the formulas that they are attached to, the invalidate routine may be called a second time. The number of iterations is bounded by a user specified limit which we will call n.

For multi-way constraints, we will subscript the terms defined in section 4.2.4 as follows: $affected_{reg\_i}$ represents the regular constraints that are affected in iteration i of the invalidate routine, $affected_{ac\_i}$ represents the activation conditions that are affected in iteration i of the invalidate routine, $needed\_by\_affected_{reg\_i}$ represents the set of dependencies needed to compute the constraints in $affected_{reg\_i}$, and $needed\_by\_affected_{ac\_i}$ represents the set of dependencies needed to compute the activation conditions in $affected_{ac\_i}$.

We first compute the time complexity for a single iteration and later compute the time complexity for the entire algorithm. The time complexity for one iteration is a summation of the invalidate and evaluate routines. The invalidation algorithm follows edges and thus it may attempt to visit a constraint more than once (although if the constraint has been visited, it will not proceed past that constraint). Thus the cost of invalida-

[2]By constraint, we are also referring to activation conditions. Activation conditions are represented internally as conventional constraints.

tion is the number of edges that are traversed. The set of edges traversed is a subset of the edges in $needed\_by\_affected_{reg\_i}$ + $needed\_by\_affected_{ac\_i}$. Hence, the time complexity of invalidation is bounded by $O(|needed\_by\_affected_{reg\_i}| + |needed\_by\_affected_{ac\_i}|)$. The evaluate routine requires that $|affected_{reg\_i}| + |affected_{ac\_i}|$ constraints be evaluated. Since the cost of executing these constraints is O(1), the cost of evaluating constraints is $O(|affected_{reg\_i}| + |affected_{ac\_i}|)$. Finally, each input variable in a constraint be accessed in O(1) time. Thus the cost of accessing all of the input variables in $affected_{reg\_i}$ + $affected_{ac\_i}$ is $O(|needed\_by\_affected_{reg\_i}| + |needed\_by\_affected_{ac\_i}|)$. Summing up, the worst case running time for one iteration is:

$$
\begin{aligned}
&O(|affected_{reg\_i}| + \\
&\quad |affected_{ac\_i}| + \\
&\quad |needed\_by\_affected_{reg\_i}| + \\
&\quad |needed\_by\_affected_{ac\_i}|)
\end{aligned}
$$

In actual implementations, the number of variables per constraint is usually bounded by some small constant k, in which case $needed\_by\_affected_{reg\_i}$ is bounded by $k|affected_{reg\_i}|$ and $needed\_by\_affected_{ac\_i}$ is bounded by $k|affected_{ac\_i}|$. Thus, the time complexity for a single iteration reduces to $O(|affected_{reg\_i}| + |affected_{ac\_i}|)$.

We now compute the time complexity for the entire algorithm. Recall that the algorithm continues until a user specified limit has been reached or no activation conditions are invalidated in the previous iteration. In the worst case, the algorithm continues until the user specified limit n has been reached. Hence n iterations of constraint solving are required. Therefore, the worst case running time for the entire algorithm is:

$$\sum_{i=1}^{n} affected_{reg\_i} + affected_{ac\_i}$$

4.4.5 Proof of Correctness

This section proves that the integration of multi-way constraints with the conventional constraints is correct. Unless otherwise indicated, throughout this section the words "con-

straints" and "algorithm" refer to conventional and multi-way constraints, and those portions of the constraint solver that handle them. We first prove that any combination of these constraints terminate, and secondly that all constraints marked valid are satisfied when the algorithm finishes.

Theorem: If all constraint formulas and activation conditions terminate, then the constraint solving algorithm for multi-way constraints terminates.

Proof: One iteration of constraint solving is not always sufficient for solving multi-way constraints. This is because multi-way constraints that have associated activation conditions may be invalid after one iteration. Therefore, the algorithm provides multiple iterations of constraint solving so that the constraint network can reach a more satisfied state. Hence, to prove that the algorithm terminates, we must prove that 1) each iteration terminates, and 2) a finite number of iterations are performed. From these two steps, it follows that the algorithm terminates.

Each iteration terminates. The constraint solving algorithm for multi-way constraints is composed of an invalidate and evaluate routine. The invalidate and evaluate routines do not call one another and thus are not interweaved. Hence, to show that the algorithm terminates, we will need to prove that these two routines individually terminate.

The invalidate routine for multi-way constraints performs a depth first search marking constraints as invalid. The invalidate routine diverges slightly from a true depth-first search in that it queues activation conditions and does not invalidate formulas downstream from these constraints. However, stopping the invalidation process early is comparable to treating a multi-way constraint as if it had no outgoing edges. In effect the invalidation algorithm behaves as if a few edges have been removed from the graph. Since a depth-first search still terminates if any number of edges are removed from a graph, the invalidation algorithm still terminates.

The evaluate routine is identical to the evaluate routine used for conventional constraints. The activation conditions are simply demanded before any variable. However, the order in which variables are demanded does not affect the termination of the evaluate routine. Hence, the evaluate routine terminates.

In summary we have proven that the invalidate and evaluate routines individually terminate. Therefore, the constraint solving algorithm for multi-way constraints is guaranteed to terminate after each iteration.

Finite number of iterations. The algorithm terminates when a user specified limit has been reached, or when the invalidate routine does not create a new list of activation conditions. Thus a finite number of iterations are performed.

Algorithm terminates. Since each iteration terminates and a finite number of iterations are performed, the algorithm terminates.

Theorem: Let C be the constraint graph constructed from the set of active constraints (constraints whose activation conditions are true) and the set of activation conditions (both those that are currently true and those that are currently false). Remove from C those edges that belong to constraints that were excluded from invalidation by exception (a) in Section 4.4.2. If the resulting constraint graph is acyclic, then all constraints in this graph that are marked valid are satisfied when the multi-way constraint solving algorithm terminates.

Proof: To show that all constraints marked valid are satisfied we will prove that for each constraint 1) if any of the inputs to the constraint were changed, then the constraint was reevaluated, and 2) the constraint was not evaluated until all of its inputs had their correct values. In effect this is showing that that these constraints are properly evaluated and that their dependencies are recorded.

We will prove this is by induction on the length of the longest chain of constraints to a variable. The basis case is a chain of length zero. That is the variable has no constraint. It is assumed that variables with no constraints have a correct value, so the basis case is trivially true.

The induction case assumes that all instance variables with chains of *n-1* constraints or less and which are marked valid have the correct values. We now prove that an instance variable which is marked value and has a longest chain of n constraints has a correct value.

To prove the first claim, that the constraint was reevaluated if any of its inputs changed, we show that the constraint was marked invalid if one of its inputs changed. A constraint becomes unsatisfied only if one of the inputs it used in its last evaluation changes value. The technique that this algorithm uses for maintaining dependencies is the same technique that is used in Alphonse [Hoover 92]. The proof of correctness for this technique is given in that paper. Therefore there are dependencies from all of the inputs the constraint used in its last evaluation to that constraint. Hence, when one of the inputs changes, the invalidation algorithm must follow the dependencies to the constraint, and mark the constraint invalid. Thus the constraint must have been notified of any changes to its inputs. A constraint is marked valid only it has been evaluated. Consequently the fact that the constraint is now marked valid means that if any of its inputs changed, the constraint has been reevaluated since those changes.

We now prove the second claim, that the constraint was evaluated after all of its inputs had been assigned their correct values. When the constraint is evaluated, it demands the values of each of its inputs, which causes them to be evaluated if they are invalid. Consequently, the constraint was not evaluated until all of its inputs were known. The argument in the previous paragraph showed that if the inputs changed after the constraint was evaluated, it would have been notified and marked invalid. The fact that the constraint is marked valid indicates that its inputs did not change

after it was evaluated. By the inductive hypothesis, these inputs must therefore have been correct, and thus the constraint must be satisfied.

In summary we have proven that all constraints are properly evaluated and that their dependencies are recorded. Therefore, when the constraint solving algorithm is finished all multi-way constraints marked valid are satisfied.

4.5 Combination of the Three Algorithms

This section describes how the algorithms for multi-output, structural, and multi-way constraints are combined. The invalidate and evaluate routines of the three algorithms are essentially similar. The interesting issue is in what order output determinant constraints, activation conditions, and structural constraints should be evaluated. All three of these constraints determine what the constraint graph looks like. However, the output determinant constraints and activation conditions are meant to allow the programmer to exercise control over both structural and value constraints, and thus should be evaluated before structural constraints. Activation conditions determine which constraints should be active and thus should be evaluated before output determinant constraints (if the multi-output constraint the output determinant constraint is associated with is inactive, there is no need to evaluate the output determinant constraint). All iterations of activation condition solving are performed before any iteration of output determinant constraint solving. Similarly, all iterations of output determinant constraint solving are performed before structural constraint solving.

Since activation conditions are evaluated before output determinant constraints, the activation conditions are not allowed to depend on any multi-output constraints whose outputs are determined by output determinant constraints. In addition, since output determinant constraints and activation conditions are evaluated before structural constraints, these two types of expressions are not allowed to depend on invalid structural attributes. This restriction is not unduly burdensome, because structural attributes should not be evaluated until

the dependencies in the constraint graph have been established, and these dependencies are established by output determinant constraints and activation conditions.

4.5.1 Proof of Correctness

We will now prove that any combination of conventional, multi-output, structural, and multi-way constraints is correct. Unless otherwise indicated, throughout this section the words "constraints" and "algorithm" refer to conventional, output determinant, multi-output, structural, and multi-way constraints, and the entire constraint solver. We first prove that any combination of these constraints terminates, and secondly that all constraints marked valid are satisfied when the algorithm finishes.

Theorem: If 1) activation conditions cannot depend on invalid structural attributes or attributes determined by multi-output constraints, 2) output determinant constraints cannot depend on invalid structural attributes, 3) action-procedures cannot request any variable that would directly or indirectly cause the recomputation of an invalid structural attribute, and 4) all constraint formulas, action-procedures, and activation conditions terminate, then the algorithm terminates.

Proof: The combined algorithm may perform several iterations of constraint solving in order to satisfy the output determinant constraints, activation conditions, and structural constraints. Hence, to prove that the algorithm terminates, we must prove that 1) each iteration terminates, and 2) a finite number of iterations are performed. From these two steps, it follows that the algorithm terminates.

Each iteration terminates. The constraint solving algorithm for these constraints is composed of an invalidate, an evaluate, and an execute action-procedures routine. Hence, to show that the algorithm terminates, we will need to prove that these three routines terminate.

The invalidate routine performs a modified depth first search marking constraints as invalid. The modification to depth first search is that the invalidate routine does not proceed beyond output determinant constraints and activation conditions. Instead it queues these two types of expressions and returns to the previously visited constraint. Effectively, the invalidate routine treats these expressions as if they have no successors, which is tantamount to deleting edges from the graph. Since depth first search terminates regardless of how many edges are deleted from a graph, the invalidate routine terminates.

The evaluate routine is now shown to terminate for each type of constraint.

Output determinant constraints and activation conditions

Output determinant constraints and activation conditions are marked valid immediately before they are evaluated. Any constraint that these constraints depend directly or indirectly on are also marked valid before they are evaluated. Validated constraints cannot be evaluated again in the same evaluate pass unless they are invalidated. However, it is not possible to invoke the invalidate routine during the evaluation pass. The invalidate routine can be invoked during structural constraint solving, but output determinant constraints and activation conditions cannot depend on invalid structural attributes, and thus the evaluation of output determinant constraints and activation conditions will not cause the structural constraint solver to be invoked. Thus no output determinant constraint, activation condition, or constraint that these two types of constraints can depend on will be evaluated more than once during one pass of the evaluate routine. Consequently the evaluate routine terminates for these constraints.

Structural constraints

Structural constraints are marked valid immediately before they are evaluated. If their value changes, any action-procedures associated with the structural at-

tribute are queued. However, these action-procedures are not executed until after the evaluate routine is completed. Thus structural constraints will terminate.

We now show that the execute action-procedures routine terminates. We assume that an action-procedure terminates if it does not send the constraint solver into an infinite loop. An action-procedure may change the values of variables, which can cause the invalidate routine to be invoked. The invalidate routine may invalidate output determinant constraints and activation conditions, which the invalidate routine will queue. When the invalidate routine is finished, the queued output determinant constraints and activation conditions will be evaluated. As shown above, the evaluate routine will terminate for these two types of constraints. Once the evaluate routine is finished, another round of invalidation may be required. However, the number of iterations of invalidation and evaluation that may occur is bounded by a user specified ceiling, and thus this process will eventually terminate and return control to the action-procedure.

The action-procedure cannot cause an invalid structural attribute to be evaluated, so the structural constraint solver cannot be recursively invoked. Since any constraint solving required to satisfy output determinant constraints or activation conditions that are affected by an action-procedure will terminate, and since an action-procedure cannot cause the reevaluation of a structural attribute, the execution of an action-procedure cannot send the constraint solver into an infinite loop. Consequently, all action-procedures eventually terminate.

Since the invalidate, evaluate, and execute action-procedures routines all terminate, each iteration of the constraint solving algorithm is guaranteed to terminate.

Finite number of iterations. The algorithm may not iterate more than a user specified limit, so the number of iterations is bounded by a finite number.

Algorithm terminates. Since each iteration terminates and a finite number of iterations are performed, the algorithm terminates.

Theorem: Let C be the constraint graph constructed from the set of active constraints (constraints whose activation conditions are true), the set of activation conditions (both those that are currently true and those that are currently false), and the set of output determinant constraints associated with active multi-output constraints. Remove from C those edges that belong to constraints that are excluded from invalidation by exception (a) in Section 4.4.2. If the resulting constraint graph is acyclic, then all constraints marked valid are satisfied when the multi-way constraint solving algorithm terminates.

Proof: We will use a proof by contradiction to show that this theorem is correct. Assume that some constraint which is marked valid is not satisfied. Let *c* represent the first constraint in a chain of constraints that is marked valid but which is not satisfied. All of c's predecessors are correctly marked valid or invalid. There are a number of ways in which the constraint might have computed an incorrect value and we show that they all lead to contradictions. First, its formula might have been incorrectly written. The constraint algorithm has no control over the writing of the formula, so we assume that the formula is correct. Second, the constraint might depend on a variable that is marked invalid. This would imply that the input was marked invalid after the constraint was evaluated but for some reason, the invalidate routine did not then proceed to the constraint and mark it invalid. Since input dependencies are correctly recorded (the technique for recording input dependencies is proven correct in [Hoover 92]), the invalidate routine knew the constraint depended on this input when the input was marked invalid. Consequently the invalidate routine would have marked

the constraint invalid.[3] The fact that the constraint is marked valid means that the evaluate routine evaluated the constraint. Since the constraint depends on the input, the constraint would have demanded the input's value. The evaluate routine would have noticed that the input was marked invalid and would have reevaluated the input and marked it valid. However, we assumed that the input was marked invalid, so we have arrived at a contradiction.

The third possibility is that the constraint was evaluated prematurely, before all of its inputs had their values determined correctly. An input might not have had its value correctly determined yet because 1) it was marked invalid at the time its value was requested, or 2) its value was determined by a multi-output constraint, but the output determinant constraint associated with the multi-output constraint had not yet been executed, and thus the input did not know that it was the target of a multi-output constraint. In the first case, the algorithm would have noticed that the input was marked invalid and would have first recomputed the value of the input. Thus the input's value would have been computed correctly before it was used by the constraint. In the second case, the output determinant constraint would have eventually been evaluated and computed a correct set of outputs (we assumed that c was the first constraint to be left unsatisfied and the output determinant constraint precedes c, so the output determinant constraint must be satisfied). The invalidate routine would then have invalidated all constraints downstream of the output determinant constraint, and thus would have marked c invalid (if the user specified limit on iteration were exceeded, the invalidation would not be performed, but we assumed that the user specified limit was not exceeded). However, since c is valid, we have arrived at a

[3]It might seem that the invalidate routine could have deferred invalidating this constraint and through some oversight, never invalidated it. However, only constraints that directly depend on output determinant constraints can be deferred in this fashion. The theorem assumes that the invalidate routine did not create a new list of output determinant constraints, so all output determinant constraints are marked valid. Consequently, a constraint cannot depend on an invalid output determinant constraint, and thus, its invalidation cannot be deferred.

contradiction. The only remaining possibility is that c depends on an input whose value is incorrect (i.e., whose constraint is unsatisfied). But then c is not the first constraint in the chain to be left unsatisfied, which contradicts our assumption that c was the first such unsatisfied constraint.

Therefore, when the constraint solving algorithm is finished all constraints marked valid are satisfied.

4.6 Performance

This section examines the performance of the extended constraints tested on a Hewlett Packard Apollo 9000 Series 750. The constraints were implemented in Lisp on top of the Garnet toolkit. The performance of these constraints will vary from one application to the next. Therefore, as we discuss a constraint's performance, we will list those factors that play a significant role in the time required to evaluate it.

4.6.1 Multi-output Constraints

The performance of multi-output constraints depends on the number of input variables, the number of output variables, and the complexity of the formula that must be evaluated. Since the user specifies the formula that is evaluated, there is no limit on the worst case running time. For example, the user could specify a formula that went into an infinite loop. The best case running time would occur when the user creates a multi-output constraint that contains a formula that does not require evaluation. For example, the user could be initializing the output variables to a constant value. This type of multi-output constraint would be extremely fast, since no input variables would have to be fetched, and no formula would have to be solved. Hence, examining the worst and best case running times for multi-output constraints tells us very little about their average running time.

To illustrate the performance of our multi-output constraints, we have implemented two of the relationships that are described is Section 3.2. The first implemented relationship

Table 4.1: The time in seconds to solve the min/max relationship discussed in Section 3.2. The columns represent the type of constraints being used and the rows represent the number of points stored as 3-tuples.

Number of Points	Single-output Constraints	Multi-output Constraint
5,000	0.02	0.01
10,000	0.05	0.03
50,000	0.29	0.15
100,000	0.59	0.26

determines the minimum and maximum x, y, and z values for a given set of points. Using single-output constraints, the values are computed by creating six individual constraints. However, when using multi-output constraints, these values can be computed by using only one constraint that has six outputs. Table 4.1 illustrates the time required to satisfy this relationship using both single-output and multi-output constraints. Figure 4.29 graphically illustrates the results shown in Table 4.2. As expected, the multi-output constraints took less time to solve than the comparable single-output constraints because they are able to reuse the "computation" of iterating over the points.

The second relationship that we implemented is also found in Section 3.2. This relationship constrains objects in a list to be 10 pixels apart horizontally. Hence, moving the first object in the list causes all remaining objects to reposition themselves to maintain the 10 pixels of separation. Table 4.2 illustrates the time required to satisfy 500 of these relationships using both single-output and multi-output constraints while varying the number of objects. Figure 4.30 graphically illustrates the results shown in Table 4.2. The results indicate that as the number of objects increase, the multi-output constraints out perform the single-output constraints. The reason that the multi-output constraints underperform the single-output constraints on the small output sizes is that the multi-output constraints are implemented on top of the Garnet toolkit, whereas the single-output constraints are

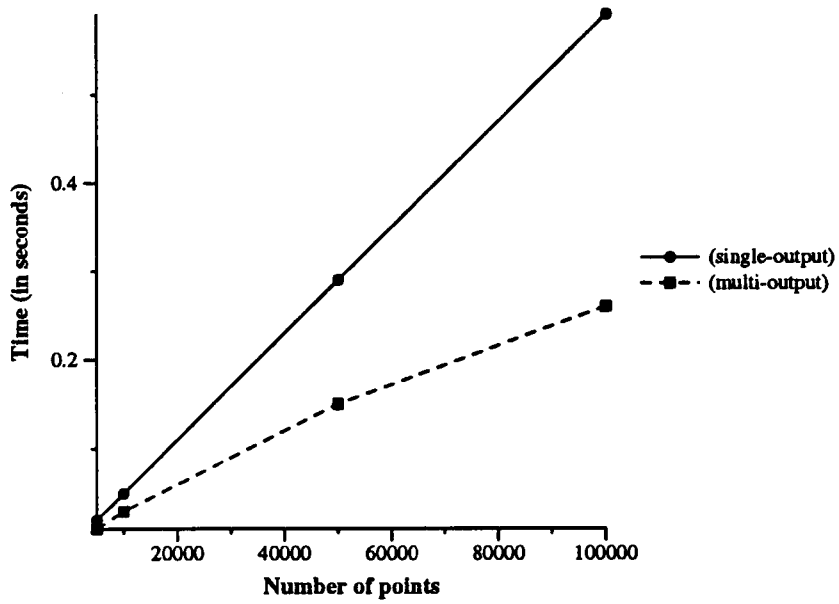


Figure 4.29: As the number of points increase, the time required to satisfy this relationship using multi-output constraints grows at a slower rate than the time required by single-output constraints.

Table 4.2: The time in seconds to solve the list alignment relationship discussed in Section 3.2. The columns represent the type of constraints being used and the rows represent the number of objects that are aligned.

Number of Objects	Single-output Constraints	Multi-output Constraint
2	0.16	0.21
3	0.23	0.23
5	0.41	0.28
10	0.85	0.42
20	2.05	0.67

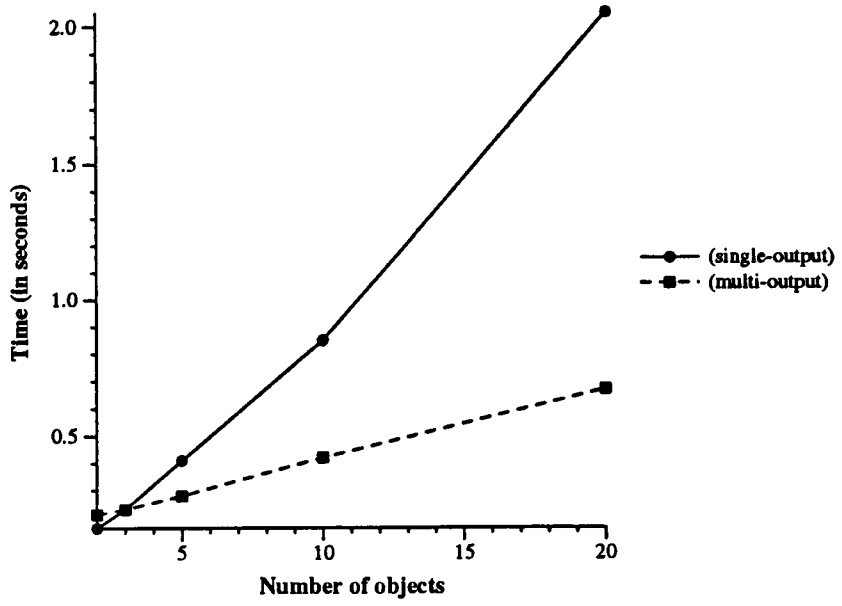


Figure 4.30: As the number of objects increase, the time required to satisfy this relationship using multi-output constraints becomes less than the time required by single-output constraints.

integrated closely with the basic object system. Thus the multi-output constraints are implemented in "software" whereas the single-output constraints are implemented in "hardware".

4.6.2 Structural Constraints

The performance of structural constraints depends on the code that is contained within the action-procedures. For example, an action-procedure that deletes one object would take considerably less time to execute than an action-procedure in a road layout system that automatically creates turning lanes, warning signs, and stop signs when an intersecting road is added. Therefore, to illustrate the performance of our structural constraints, the action-procedures in our benchmark will not execute any code. Hence, our benchmark measures the additional cost involved in handling structural constraints. This additional cost can be attributed to queuing the structural variables and action-procedures, and invoking the action-procedures. Table 4.3 compares the time required to satisfy value constraints versus the time required to satisfy structural constraints. As shown in this table, our constraint

Table 4.3: The time in seconds to solve value and structural constraints. The columns represent the type of constraint and the rows represent the number of constraints being solved.

Number of Constraints	Value	Stuctural
100	0.02	0.03
200	0.04	0.06
300	0.06	0.10
500	0.09	0.17
1000	0.17	0.32

solver can satisfy 500 value constraints in 0.09 seconds or 500 structural constraints in 0.17 seconds. Figure 4.31 graphically illustrates the results shown in Table 4.3. These results indicate that approximately two value constraints can be solved in the time required to satisfy one structural constraint.

4.6.3 Multi-way Constraints

The performance of multi-way constraints depends on the number of variables, formulas, and activation conditions that are participating in the multi-way constraint. To illustrate the performance of these constraints, we have implemented two types of multi-way relationships. The first implemented benchmark is an equality relationship. In this example, by changing the value of one variable, all other variables will obtain this new value. To express an equality relationship between n variables requires that $n - 1$ multi-way constraints or $(n - 1) \times 2$ formulas be constructed. The performance of this benchmark is illustrated in the *Equality* column of Table 4.4. The second implemented benchmark uses activation conditions to control whether changing the right side of an arbitrary object in a list causes the objects in the list to move or grow. The performance of this benchmark is illustrated in the *Activation Conditions* column of Table 4.4. Figure 4.32 graphically illustrates the results shown in Table 4.4. The graph reveals that the time to solve multi-way constraints increases

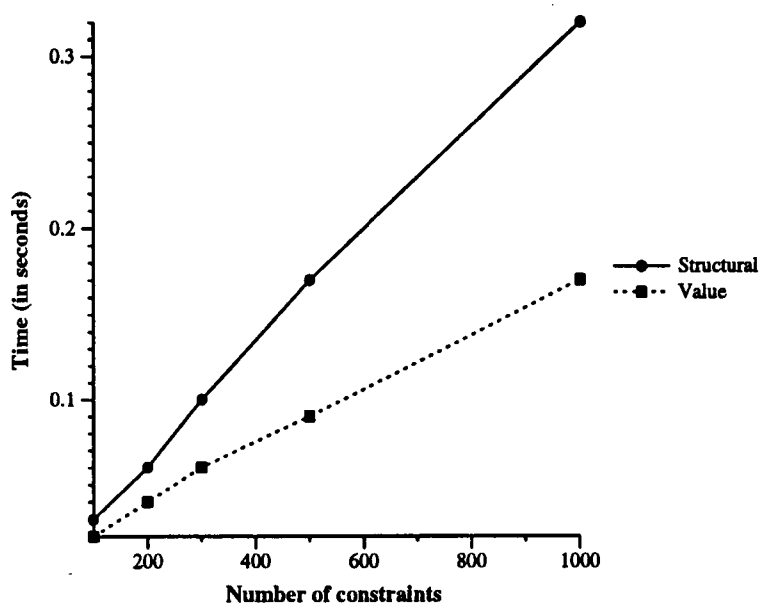


Figure 4.31: The time required to satisfy value constraints versus the time required to satisfy structural constraints.

Table 4.4: The time in seconds to solve two different types of multi-way constraints. The columns represent the type of multi-way relationship that is expressed and the rows represent the number of constraints being solved.

Number of Constraints	Equality	Activation Conditions
10	0.01	0.02
50	0.04	0.05
100	0.07	0.09
500	0.38	0.43
1000	0.75	0.89

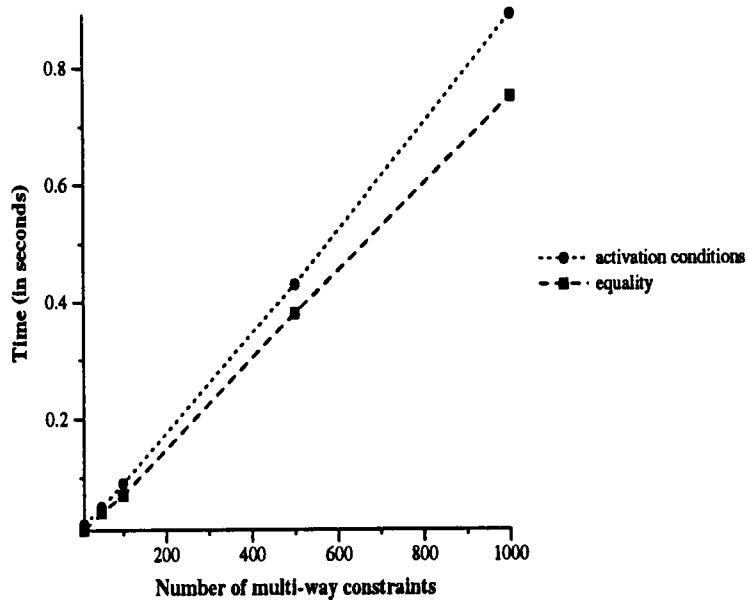


Figure 4.32: The time to solve multi-way constraints increases linearly with the number of variables that are participating in the constraint solving process.

linearly with the number of formulas that are participating in the constraint solving process.

In our benchmark, the activation conditions were use to provide user control over the constraint satisfaction process. In this example, half of the activation conditions always evaluated to true and the other half evaluated to false depending on whether the objects should grow or move. In the best case, the first activation condition in a long series of sequential multi-way constraints evaluates to false. Hence, all formulas downstream are not invalidated and thus are not forced to be later re-evaluated. For example, assume we are given 500 sequential multi-way constraints all of which have activation conditions. If all activation conditions evaluate to true, the activation conditions increase the time required to satisfy the system from 0.38 seconds to 0.41 seconds. However, if the first activation condition evaluates to false then the time required to satisfy the system can be reduced from 0.38 seconds to 0.05 seconds. Note the huge savings that is possible in the best case, verses the minimal lost in the worst case.

Part III

AN OBJECT-ORIENTED 3D TOOLKIT

Introduction to Viewing in 3D ..... Chapter 5

Programming Layer ..... Chapter 6

Direct Manipulation Layer ..... Chapter 7

Chapter 5

Introduction to Viewing in 3D

The final contribution of this dissertation is the development and implementation of an object-oriented toolkit for creating highly interactive, direct manipulation, 3-dimensional user interfaces. The toolkit is divided into two layers: a programming layer and a direct manipulation layer. The programming layer (described in Chapter 6) supports an extensive set of 3D objects, structured graphics, constraints, and a high-level event handling model. It also includes functions for projecting objects, clipping, shading, and applying transformations. Programmers using this level must write their own code. The reference manual for the programming layer is contained in Appendix 2.

The direct manipulation layer (described in Chapter 7) allows the user to interactively create and directly apply transformations to 3D objects, construct polyhedron objects, change the viewing parameters for drawing 3D objects, and build structured graphics. Many of these features have typically been ignored in direct manipulation tools. The reference manual for the direct manipulation layer is contained in Appendix 3.

Our toolkit has experimentally been used in a graphics course and has allowed students to create substantially more complex and interesting 3D applications than was previously possible. This chapter introduces some of the concepts and terminology used for viewing in 3-dimensions.

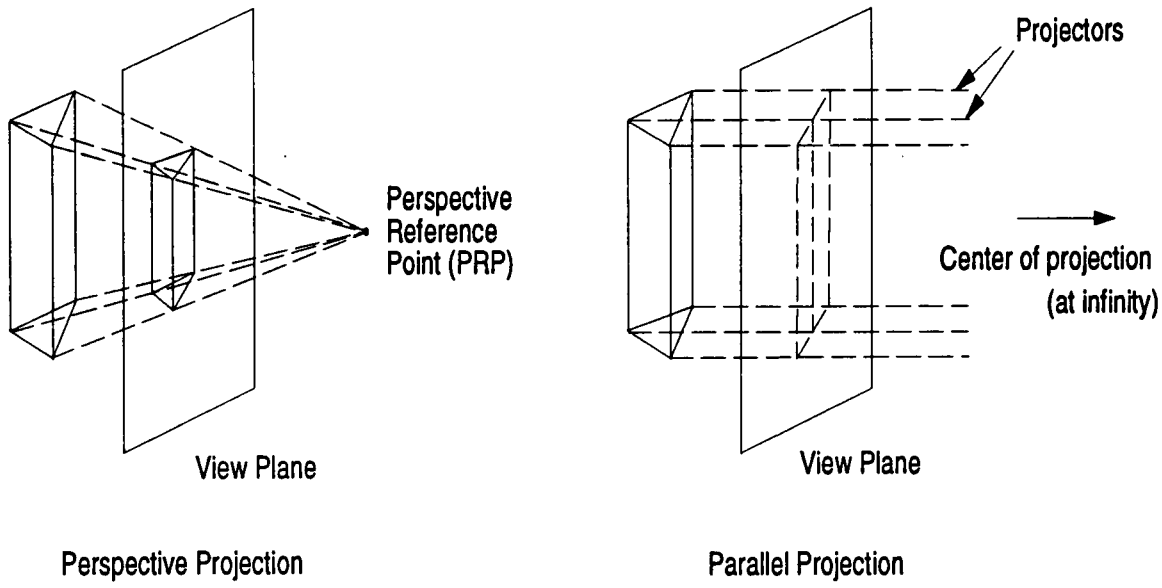


Figure 5.1: Perspective and parallel projections.

5.1 Projections

Projections allow a three dimensional object to be viewed on a two dimensional graphic display media. Our toolkit supports a class of projections called planar geometric projections, which project 3D objects onto a two dimensional plane using straight projectors. Other classes of projections can project 3D objects onto a curved surface and/or use curved projectors [Foley 90]. There are two types of planar geometric projections, which are referred to as *perspective* and *parallel.* These two projections are defined below and are illustrated in Figure 5.1. Both of these projections are supported by our toolkit.

5.1.1 Perspective Projection

In a perspective projection the distance from the center of projection to the view plane is finite. This type of projection is often the preferred method since it comes closest to how the human eye perceives the real world. A perspective projection preserves the property that distant objects are smaller than closer objects.

Table 5.1: Below is a list of the parameters that the user can modify to specify an arbitrary 3D view. These parameters allow the user to change the type of projection, specify clipping options, and modify the viewing parameters.

parallel-p	specifies the projection type
vrp	view reference point
vpn	view plane normal
vup	view up vector
prp	projection reference point
window-size	the window size that resides on the view plane
front-clipping-plane	specifies the front clipping plane
back-clipping-plane	specifies the back clipping plane
clip-p	determines whether or not clipping takes place

5.1.2 Parallel Projection

In a parallel projection the distance from the center of projection to the view plane is infinite. This type of projection is called parallel because with the center of projection infinitely distant, the projectors are parallel to each other. This type of projection is similar to how the sun casts a shadow onto a wall.

In our toolkit, the type of projection can be changed by modifying the parameter *parallel-p*. Table 5.1 lists this parameter along with the other parameters that the user can modify to specify an arbitrary 3D view.

5.2 Specifying the 3D View Volume

To view in 3D, we must specify not only the type of projection, but also the view volume. The view volume represents the 3D volume that is projected and which is viewable. Any portion of a 3D object that lies outside this volume will not be viewable. The view volume is defined by a view plane, a window on the view plane, and a projection reference point (PRP). The view plane represents the surface onto which an object will be projected. The view plane is defined by a point on the plane called the view reference point (VRP) and a normal to this plane called the view plane normal (VPN) as shown in Figure 5.2. The

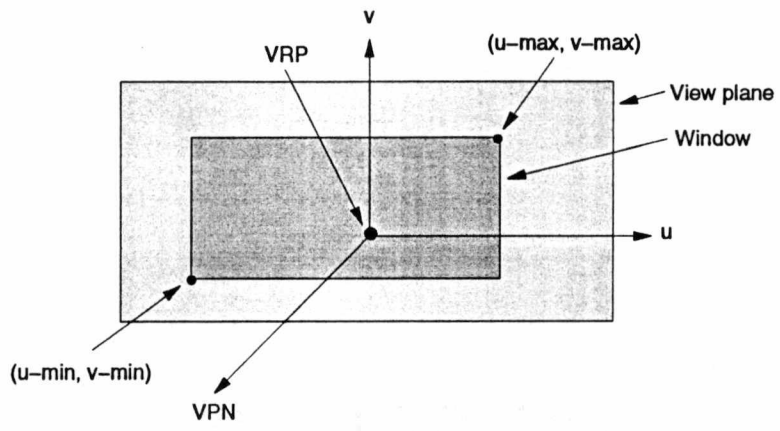


Figure 5.2: The view plane is defined by the view reference point (VRP) and a normal to this plane called the view plane normal (VPN). The window is defined by providing points for the lower left and upper right corners of the window. The coordinates of the points are defined relative to the u and v axes.

window represents the rectangular region of the view plane that the user will see. The window is defined by two orthogonal axes that lie on the view plane. These axes are labeled u and v. The v axis is also called the view up vector (VUP), and the VPN is also called the n axis. Together, the u, v, and n axes form the viewing reference coordinate (VRC) system [Foley 90].

The projection reference point represents the center of projection. If the type of projection is perspective, then the view volume is defined by the projectors that travel from the perspective reference point to the corners of the window. This type of a projection creates a pyramid shaped view volume as shown in Figure 5.3. If the type of projection is parallel, then the direction of projection is from the perspective reference point to the center of the window. This type of a projection creates an infinite parallelepiped view volume as shown in Figure 5.4. The projection (the image on the view plane) is then mapped into the window that the user sees. This mapping is accomplished by scaling the projected points on the view plane to match the dimensions of the window.

In our toolkit, the parameter *vrp* specifies the view reference point, *vpn* specifies the view plane normal, *vup* specifies the view up vector, *prp* specifies the perspective reference

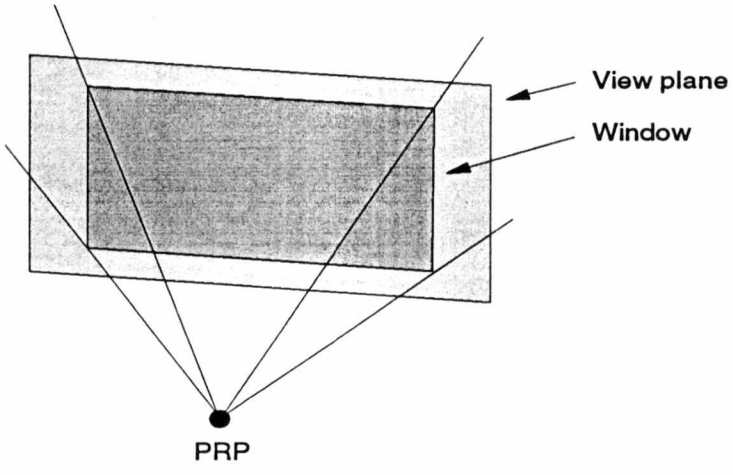


Figure 5.3: When using a perspective projection, a pyramid shaped view volume is created.

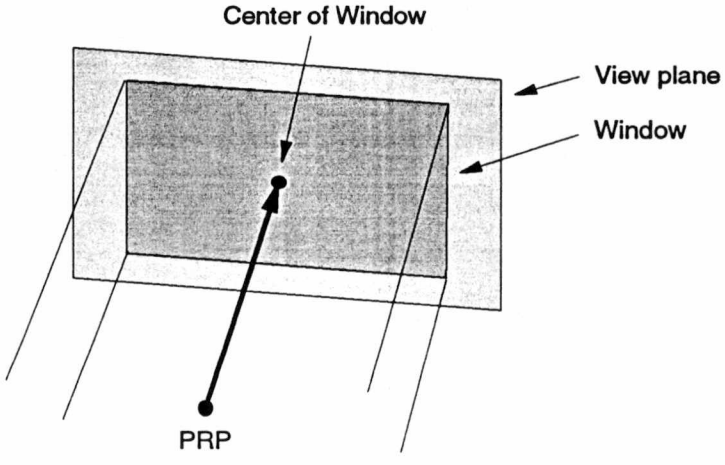


Figure 5.4: When using a parallel projection, an infinite parallelepiped view volume is created.

point, and *window-size* specifies the size and location of the window that resides on the view plane.

5.3 3D Transformations

Our toolkit supports the three basic types of 3D transformations: rotation, scaling, and translation. Rotation changes the orientation, scaling changes the size, and translation changes the location of an object. These transformations are combined into a single matrix so that only one matrix multiplication is required to apply all three transformations. The matrix operations are hidden from the user. Hence, the user sets parameters such as *translate-x*, *rotate-y*, and *scale-z* and then the system automatically translates these parameters into the appropriate matrices and performs the necessary matrix operations.

In the 3D toolkit, a transformation is applied to an object by changing the value of the appropriate transformation slot. Shown below are three examples of changes in attribute slots. These changes cause the object *my-cube* to automatically be rotated on the x-axis by 30 degrees, scaled on the y-axis by 2, and translated by a positive 10 coordinates in the z direction.

```
my-cube.rotate-x = 30;
my-cube.scale-y = 2;
my-cube.z = 10;
```

Before scaling an object, our implementation first translates the object so that its center lies on the origin. This translation prevents an object from moving when it is scaled. For example, Figure 5.5 shows the result of a house being scaled without the object first being translated. Notice in this figure how scaling has moved the object to the right. Figure 5.6 shows the same scale being applied, except that this time the center of the object is first translated to the the origin. Notice how the object has expanded in both directions.

In a similar fashion, before rotating an object, our implementation first translates the object so that its center lies on the origin. Figure 5.7 shows the result of a house being

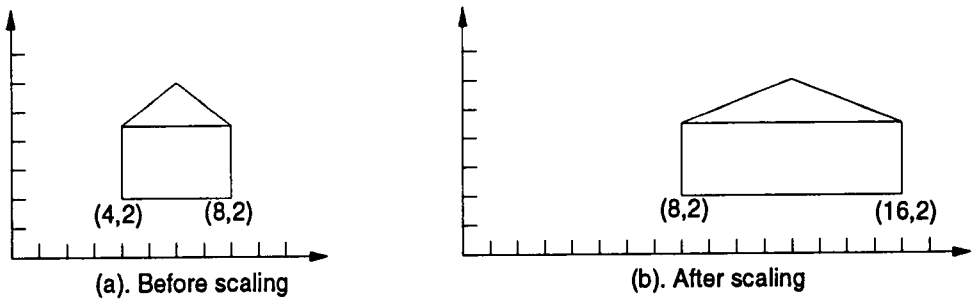


Figure 5.5: Scaling an object in the x direction by 2 without first translating the center of the object to the origin.

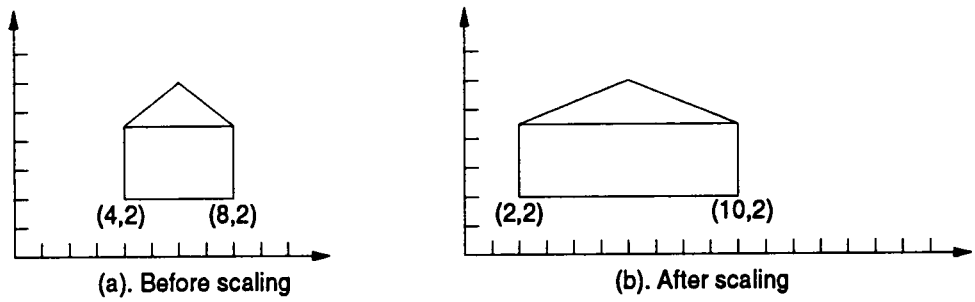


Figure 5.6: The same scale has being applied as in Figure 5.5, except that this time the center of the object has first been translated to the origin.

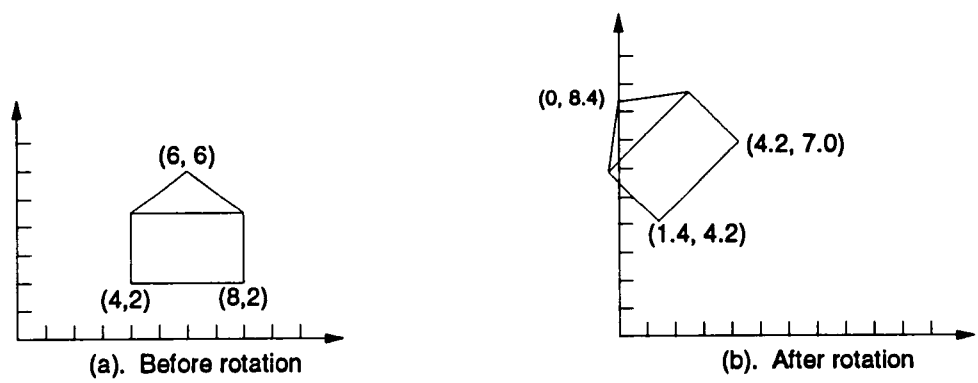


Figure 5.7: Rotating an object in the z direction by 45 degrees without first translating the center of the object to the origin.

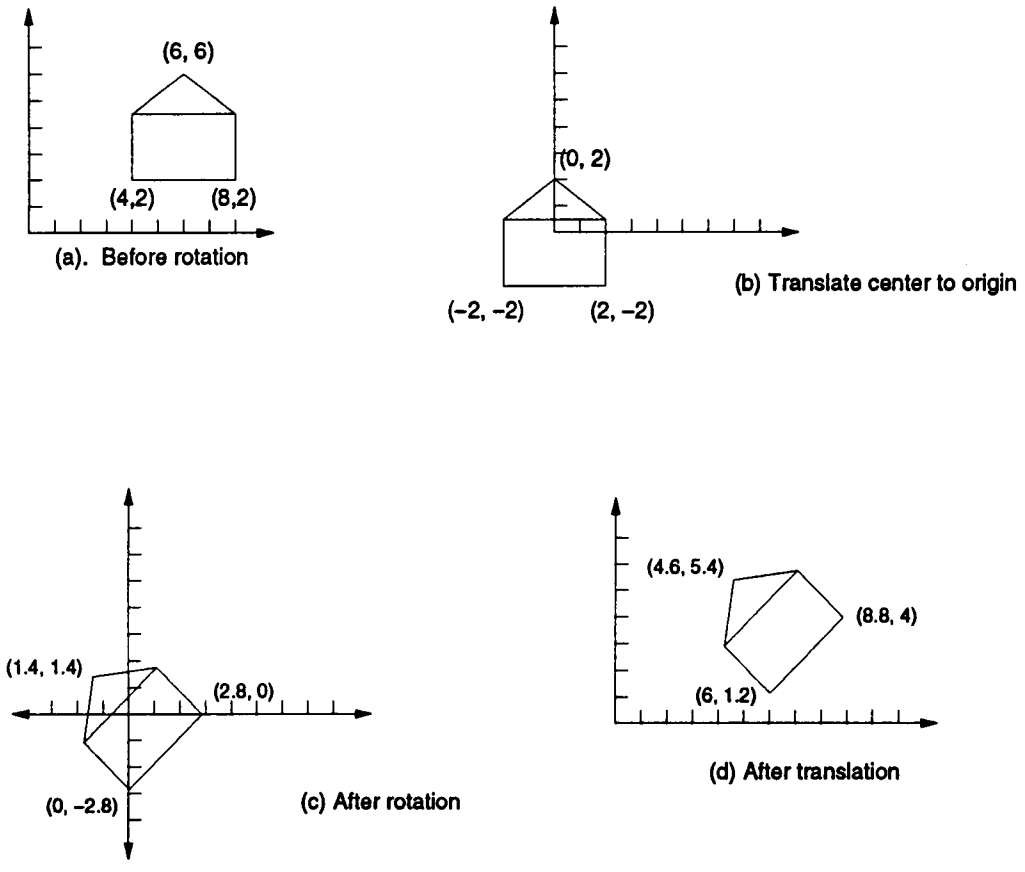


Figure 5.8: The same rotation has being applied as in Figure 5.7, except that this time the center of the object has first been translated to the origin.

rotated without the object first being translated. Notice in this figure how rotation has moved the house up and to the left. Figure 5.8 shows the same rotation being applied, except that this time the center of the object is first translated to the the origin. Notice in this figure how the house has been rotated about its center. The user can explicitly rotate an object about a specified point by using the parameters *rotate-center-x*, *rotate-center-y*, and *rotate-center-z*. These parameters would be set when the desired center of rotation is not the center of the object. As an example, to raise the arm of a robot, the arm should be rotated about the shoulder of the robot rather than the center of the arm.

The 3D toolkit uses a right handed coordinate system (rhs). In a rhs, the z axis is pointing out towards the viewer; whereas in a left handed system, the z axis is pointing

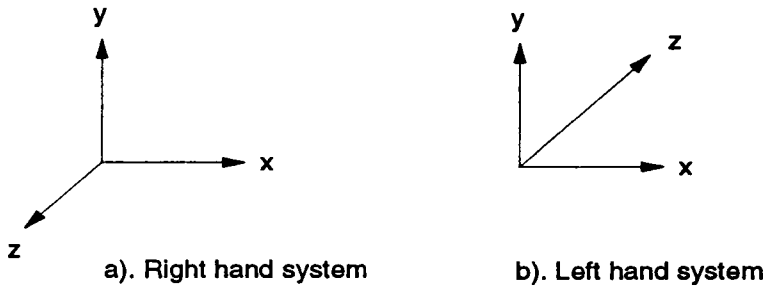


Figure 5.9: The two types of coordinate systems.

Figure 5.10: The polygons in (a) and (b) are being clipped against a two-dimensional clip rectangle.

away from the viewer. In both systems the x and y axes are aligned to the screen. Figure 5.9 shows the difference between the two coordinate systems.

5.4 Clipping

Clipping is the process of determining those portions of primitives that lie inside the clip region, or stated another way is the process of blocking out all portions of primitives that lie outside the clip region. While clipping is easy for humans, for a computer, clipping can be a very complex task. Figure 5.10 illustrates an example of clipping a polygon in two-dimensions against a clip rectangle.

When clipping in three dimensions, we must clip along six planes. Four of these planes are defined by the view volumes illustrated in Figures 5.3 and 5.4. The remaining two planes are specified using a front clipping plane and a back clipping as shown in Figure 5.11. These two planes allow the user to specify a finite view volume. Clipping is now performed by calculating the intersection of each polygon that composes an object with

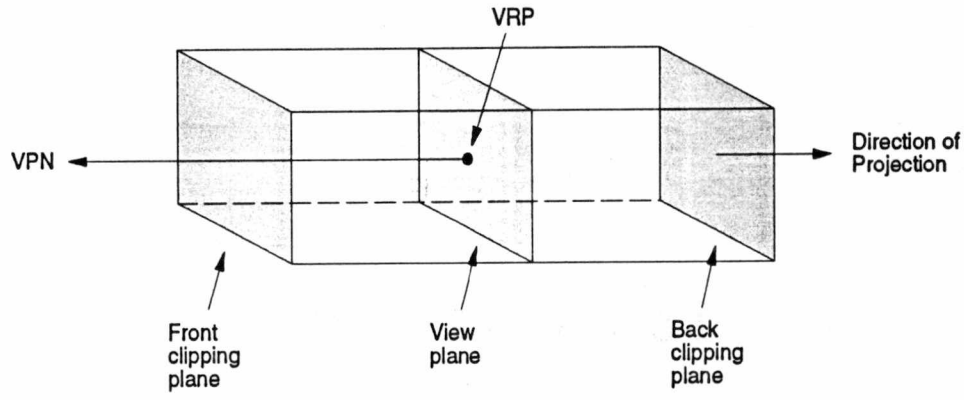


Figure 5.11: The front clipping plane and back clipping planes are specified relative to the view reference point (VRP).

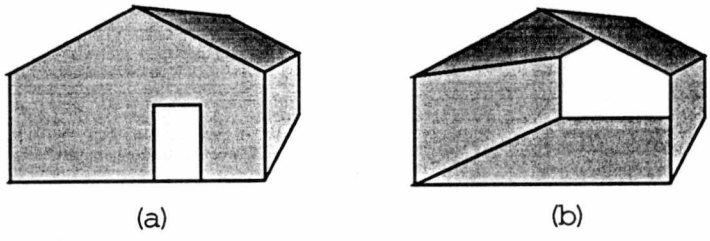


Figure 5.12: Perspective projection of a house without clipping (a). The front and rear walls have been clipped away (b).

the six planes that define the view volume [Foley 90]. When the view plane normal (VPN) coincides with the z axis, then the view volume determines the clipping in the x and y directions; and the front and back clipping planes determine clipping in the z direction. An example of clipping a house is illustrated in Figure 5.12. In our toolkit, the parameter *clip-p* indicates whether clipping will occur; *front-clipping-plane* specifies the front clipping plane; and *back-clipping-plane* specifies the back clipping plane.

Chapter 6

Programming Layer

6.1 Introduction

This chapter describes the programming layer of the 3D toolkit. The programming layer supports an extensive set of 3D objects, object composition (aggregates), constraints, and a high-level behavioral model. It also includes functions for projecting objects, clipping, shading, and applying transformations. Programmers using this level must write their own code.

6.2 3D Objects

Our toolkit provides an extensive set of 3D objects including cubes, planes, 3D-lines, cones, cylinders, spheres, ellipsoids, and polyhedrons as shown in Figure 6.1. All 3D objects (except the 3D-line) are modeled using a representation called *polygon meshes* [Foley 90]. A polygon mesh is a collection of vertices, edges, and polygons that describe a 3D real world object. The polygon mesh approach makes it easy to describe objects with straight sides like cubes and buildings. However, this approach can also be used to describe objects whose surfaces are not composed solely of planes. For example, objects with curved or rounded surfaces can be approximated using a polygon mesh so that the final result looks smooth. A tradeoff exists between the appearance of an object and the time required to render that object. The more polygon meshes used, the longer it takes to render, but the smoother looking the final object will appear.

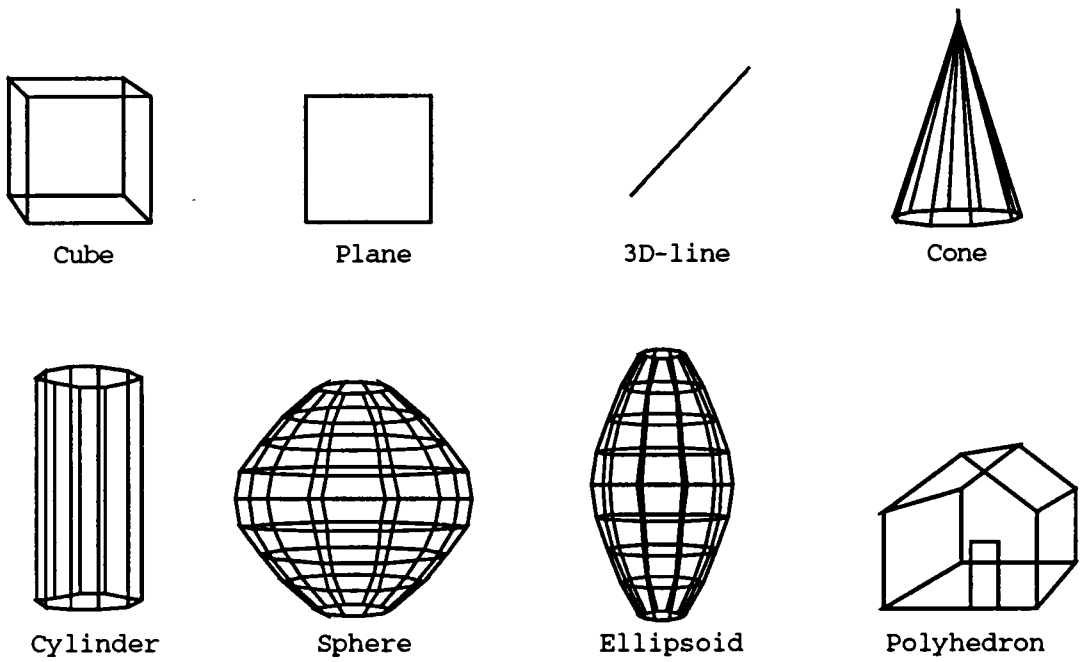


Figure 6.1: The available 3D objects in our toolkit.

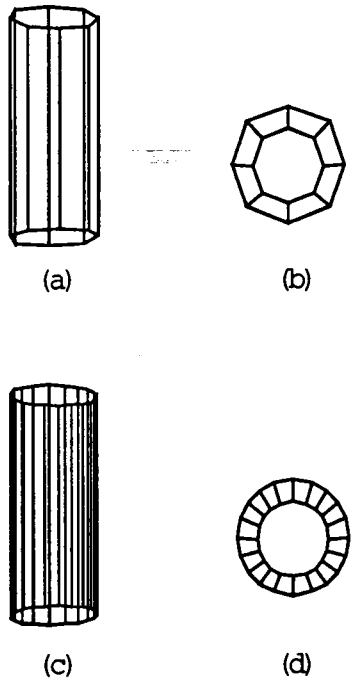


Figure 6.2: The result of varying the number of polygons used to represent a cylinder. Figure (a) shows the front view of a cylinder drawn using 8 polygons and (b) its corresponding top view. Figure (c) shows the front view of a cylinder drawn using 20 polygons and (d) its corresponding top view.

We have designed our cone, cylinder, sphere, and ellipsoid so that they can be drawn using a varying number of polygons to produce smoother looking objects. Figure 6.2 illustrates an example of a cylinder being drawn with a different number of polygons. In this example, the objects were rendered using "wire-frame". If the rendering mode for these objects was changed to "shaded" then the individual polygons would be less distinct, and the objects would appear to have been constructed from one curved surface. The number of polygons used to represent an object is stored as a variable. The value of this variable can be changed at creation time, or at run time. When its value changes, the object is automatically re-projected and redrawn during the next screen update using the newly specified number of polygons.

The performance of cones, cylinders, and spheres can be increased further by using fast-

Figure 6.3: The fast-draw objects.

draw methods associated with these three objects. The fast-draw objects can be translated, scaled, and rotated just like the regular objects. Figure 6.3 illustrates the fast-draw objects. These objects are called fast-draw, because they are composed of relatively few polygons. For example, the fast cone is composed of only two polygons. One polygon represents the side of the cone, and the other polygon represents the ellipse shaped bottom. To illustrate the savings achieved by a fast-draw cone, consider a regular cone that uses 20 polygons to represent the sides. The regular cone draws 21 objects (20 sides + 1 bottom = 21); whereas, the fast cone draws only 2 objects. The fast cylinder has similar savings, since it is represented using only three polygons (i.e., one rectangular shaped polygon and two ellipsed shaped polygons that form the top and bottom). The fast cone and fast cylinder are able to use so few polygons because rotation about their y-axis can be ignored. It should be noted that the fast draw objects may look slightly less realistic than regular objects.

The fast sphere is drawn by overlapping circles of a smaller radius and of a lighter color. Two points are projected to calculate the size and location of a fast sphere. This compares to the 90 points that were projected to produce the sphere in Figure 6.1. By projecting these two points, the geometric transformations of scaling and translation can be applied to a fast sphere. These transformations will yield the same results as those transformations applied to a regular sphere. The rotate slots are ignored when using a fast-sphere.

It should be mentioned that to produce a realistic image, the fast-draw objects are always rendered using the shaded mode. A wire-frame rendering of these objects would

not necessarily resemble their non fast-draw counter parts. The cone, cylinder, and sphere can be converted back and forth between regular and fast-draw objects at run time by manipulating the value of an object's *fast-draw* variable. When the *fast-draw* slot is changed, the object is automatically reprojected and redrawn during the next screen update.

Various attribute slots (also called parameters) control the appearance of the 3D objects. These parameters vary slightly from one object to the next. For example, a sphere has a *radius* parameter while a cube does not. As an example, the parameters associated with a cube include the following:

line-style	specifies the outline surrounding the cube
filling-style	specifies the default filling color. The parameters front-color, back-color, right-color, left-color, top-color, and bottom-color can be set to change the color of a specific side.
wire-frame-p	specifies the rendering mode as either wire-frame or shaded
width height length	specify the dimensions of the cube
x y z	translation parameters – change the location of the cube
scale-x scale-y scale-z	scale parameters – change the size of the cube
rotate-x rotate-y rotate-z	rotation parameters – change the orientation of the cube
rotate-center-x rotate-center-y rotate-center-z	specify a new center of rotation. The default value is the center of the cube.

The 3D toolkit is declarative in that users change the values of attribute slots rather than calling methods to cause changes to an interface. For example, to change the position, dimension, rotation, or rendering mode of an object, the user changes the appropriate slots and the screen update algorithm automatically reprojects and/or redraws any affected objects. Shown below are two examples of changes in attribute slots. These changes cause

the object *my-cube* to have a width of 30 and to have its front shaded red.

```
my-cube.width = 30;
my-cube.front-color = red;
```

6.3 3D Aggregates

An aggregate is a collection of objects that are grouped together to create a structured object. Aggregates allow 3D objects to be used as building blocks to create more complex objects, which in turn can be used to create even higher-level objects. An unusual feature of Garnet, that our 3D toolkit also supports, is structural inheritance within aggregate objects. This means that when an instance of an aggregate object is created, instances of the aggregate's components are automatically created and linked in with the parent. In other toolkits, this recursive creation must be done procedurally by the programmer.

The structural inheritance mechanism also ensures that if an instance variable of a prototype object is determined by a constraint, an instance of the constraint is created and stored with the corresponding instance variable in the object's instance. This feature is made possible through the use of pointer variables [Vander Zanden 91, Myers 90a] . In an indirect reference constraint system like Garnet, all objects maintain a pointer to their parent, and a set of pointers to their children. When an instance of an aggregate is created, instances of all the components in the aggregate are created and the pointer variables are set. In this approach no changes are necessary to the constraint expressions. Inherited constraints reference the correct objects, since they use the pointer variables in the instance objects rather than using the pointer variables in the prototype objects. Figures 6.4 and 6.5 contain an example of this principle in practice. After the 3D aggregate representing the wheels/axle assembly has been constructed, the user can begin to build a vehicle by creating two instances of this aggregate. One instance of the aggregate would represent the front of the vehicle and the other instance would represent the rear.

Transformations can be applied to 3D aggregates just like primitive objects. For exam-

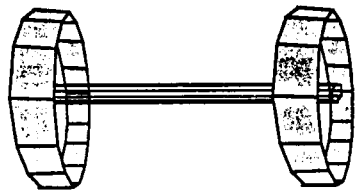


Figure 6.4: A 3D aggregate object representing two wheels and an axle.

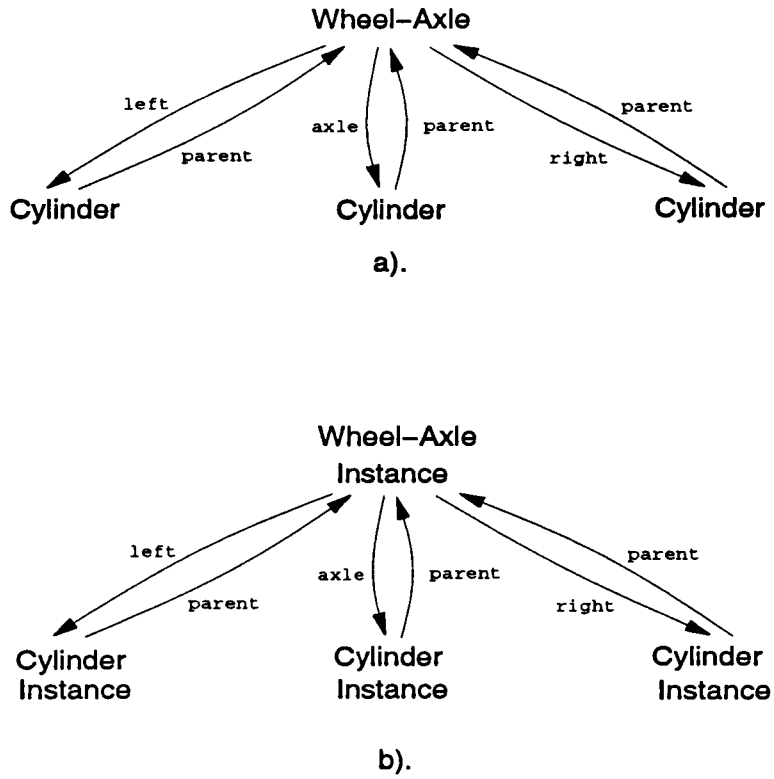


Figure 6.5: The wheels/axle assembly shown in Figure 6.1 is a structured object composed of three primitive objects (a). When an instance of the wheels/axle assembly is created, instances of all the components and constraints in this aggregate are created and the pointer variables are set. The constraints in the instance of the wheels/axle assembly now automatically reference the appropriate objects (b).

ple, the following statement causes all objects in the aggregate *my-agg* to be rotated by an additional 30 degrees in the x direction.

```
my-agg.rotate-x = 30;
```

The 3D aggregates use relative positions while the regular 2D Garnet aggregates use absolute positions. For example, in a 3D aggregate, an x value of 25 means 25 coordinates from the start of the aggregate; in a 2D aggregate it would mean an absolute screen position of 25 pixels. It is necessary to use relative positions with 3D aggregate objects because the use of transformation matrices requires relative rather than absolute coordinates. A matrix composition approach has been used to implement these transformations [Foley 90]. All 3D aggregates have a local matrix that is a composition of its transformations along with all the transformations to the root of the aggregate hierarchy. This approach requires only one matrix multiplication to project any 3D primitive object no matter how many levels there are in the hierarchy structure.

6.4 3D Update Algorithm

Our 3D graphical interface provides two rendering modes of display. They are wire-frame and shading. In the wire-frame mode only the edges of objects are drawn and hence the order of display is unimportant. While this mode can be slightly faster, its results are less realistic than the shading mode, especially when modeling solid objects. In the shading model all objects are drawn as filled polygons. The rendering process for shading requires slightly more processing because all objects must be spatially ordered. Parts of objects that are hidden, because they are obscured by "closer" objects, must not be displayed.

There are two fundamental approaches to visible-surface determination [Foley 90]. The first approach is called image-precision and determines which object is visible at each pixel. While straight-forward, this approach requires examining all n objects for each pixel. Shown below is the pseudocode for the image-precision approach:

for each pixel in the display **do**

```
begin
   determine which object is closest to the viewer
   draw the pixel with the color of that object
end
```

The other type of visible-surface determination is object-precision. This approach compares all objects with each other to determine which objects are visible. An advantage of this approach is that the objects can be redisplayed correctly at any resolution; whereas with image-precision, the visible surface calculations must be repeated if displaying the objects at a different resolution. The pseudocode for an object-precision approach is

```
for each object do
   begin
      determine those parts of objects that are closest to the viewer
      draw these unobstructed parts in the appropriate color
   end
```

The visible-surface determination algorithm that we have implemented uses a list-priority approach with object-precision operations to determine the shading order. The list-priority algorithm orders all polygons such that a correct picture results if the polygons are rendered in that order. For example, if no polygons overlap in the z-coordinate then they can be shaded from furthest to closest. In this case the pixel values of closer objects will overwrite the pixel values of more distant objects, possibly completely obscuring more distant objects.

To achieve acceptable performance in a real time environment, our 3D shading algorithm will only handle objects that consist of polygons that conform to the following restriction. For any two polygons in the view volume, one polygon must be wholly in one half-space of the other. Objects like Figure 6.6(a) that penetrate each other, or Figure 6.6(b) that cyclically overlap will not be shaded correctly, for there is no order in which these polygons can be drawn to produce the desired image. If the objects in Figures 6.6 (a) and (b) are split into smaller polygons, such that a linear order is possible, then they can be correctly handled by our shading algorithm.

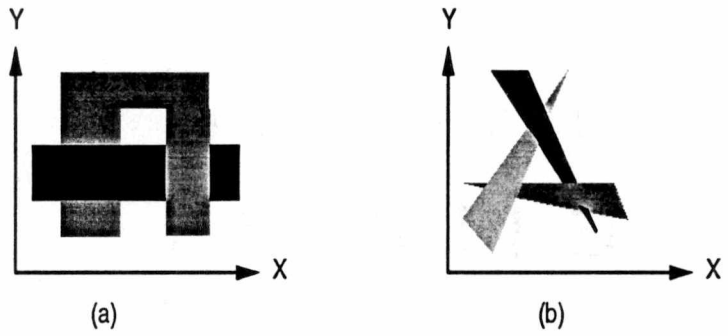


Figure 6.6: Polygons that are not guaranteed to be shaded correctly.

The 3D toolkit uses an *incremental* update algorithm that only redraws those objects that have intersected the modified portion of the screen. It employs a strategy that avoids having to examine large groups of objects [Myers 89b]. To achieve even better performance with 3D objects, a "project" and a "draw" method have been created for each type of object. The "project" method is only called when there is a change in one of the "interesting" slots for an object. For example, the dimensions of a cube and its transformation parameters are considered interesting slots. More specifically, the following slots are "interesting" for a cube:

```
line-style filling-style wire-frame-p width height length
x y z scale-x scale-y scale-z rotate-x rotate-y rotate-z
```

After all "project" methods are executed, the appropriate "draw" methods are called. When a transformation is made to a 3D aggregate object, all the components of the aggregate are reprojected and redrawn. However, objects that cover or are covered by the changed aggregate are only redrawn, not reprojected. A high-level version of the update algorithm is shown below:

```
Determine those objects that have changed
  (If the object is an aggregate then include children)
Reproject the changed objects
Determine the area of the screen that those objects occupy
Erase this area
Perform visible surface determination
Redraw the objects
```

6.5 Integration of Constraints

Constraints provide a declarative means for expressing relationships among graphical properties and transformations of 3D objects. They can be used to keep objects rotated by the same amount or to ensure that their filling styles are the same. For example, the following constraint can be added to all components in a 3D aggregate:

```
object.rendering-mode = parent.rendering-mode
```

parent is a pointer to the object's aggregate. If the rendering mode of an aggregate was changed, all the components would be redrawn using the new rendering mode.

High-level constraints have been created to simplify expressing alignment relationships that depend on the position and size of 3D objects. The syntax for the high-level constraints is described in Appendix Section B.9. These constraints are desirable for two reasons. First the parameters used to specify the position and size of 3D objects vary from one object to the next. For example, the width of a cube is specified using a *width* parameter while a cone uses a *radius* parameter. Similarly, the exact position of a 3D object depends not only on the parameters x, y, and z, but also on the type of object. For example, these parameters specify the back lower left corner of a cube, whereas, they specify the center for a sphere. The second reason these high-level constraints are advantageous is because 3D objects can be scaled. Scaling complicates the specification of alignment constraints, since an object's size is really a combination of both its specified dimensions and of any scale factors that have been applied. For example, if an object's width is 20 and its scale factor is 0.5 then its actual width is 10. Hence, we must also consider the object's scale parameters in order to correctly express a translation relationship.

Let us now look at an example. Suppose that object1 and object2 are 3D objects, and we wish to express the relationship that the left side of object1 is to be constrained to the right side of object2. Using the regular low-level constraints, the programmer can express

this relationship as follows.

```
object1.x =  if (object2.type = [cone, cylinder, sphere, ellipsoid])
               return (object2.x + object2.radius × object2.scale-x)
             else
               return (object2.x + object2.width × object2.scale-x)
```

However, using the high-level 3D constraints, the programmer can express this relationship as shown below. This relationship holds regardless of object1 and object2's type.

```
3D-constraint (object1, "L", object2, "R")
```

As shown in this example, these high-level constraints free the programmer from having to know the detailed construction of the objects, such as whether the size of an object is determined by a *radius* or a *width*. These constraints compute positions in local coordinates and are applied before the local transformation matrix is calculated. For example, if object2 starts at x-coordinate 30 in the local coordinate system and has a width of 40, then object1 would start at local x-coordinate 70. The toolkit also allows the programmer to incorporate the extended constraints discussed in Chapters 3 and 4 into their code.

6.6 3D Interactors

Like 2D user interfaces, one of the most time consuming tasks of creating a highly interactive 3D user interface is handling the input. The input is usually asynchronous and may come from the mouse, keyboard, or other input devices. Typically, 3D graphic packages have provided little or no support for interaction with the user. Generally they send a stream of low-level input events for a window to the application program. The application program must then queue the events and process them sequentially. Some toolkits provide a facility for selecting objects, but provide no support for interactive operations that create, move, or resize objects. Our 3D toolkit takes a drastically different approach based on the interactors model used in Garnet [Myers 90b]. The interactors model encapsulates all the common interactive behaviors in a user interface into a few basic object types (i.e., interactors). When an event is received, it is mapped into one or more of these interactor

objects. Each interactor provides a number of parameters that allows its behavior to be customized, such as the event it begins on, the set of objects it operates on, and the type of feedback it provides. The interactor objects provide a clean separation between input handling and the graphical representation for an object. This approach has been found to be easy to learn and program [Myers 90b].

As with 2D interfaces, it appears that there are only a few fundamentally different types of behaviors associated with 3D user interfaces. The major type of interactive behavior is handling geometric transformations. Therefore, an interactor has been created for each of the three basic types of geometric transformations (rotation, scaling, and translation). Other interactors have been created to select 3D objects, reposition projected images, enter new points, and create new 3D objects. Because of the commonly used 2D input devices (like the mouse), the interactors do not operate in all three directions simultaneously since ambiguous results might be obtained. To accommodate 2D input devices, the 3D interactors operate along at most two dimensions at any given time. The desired dimensions are specified as one of the parameters to the interactor and they are stored in the *direction* slot. The six types of 3D interactors are listed below.

- 3D-Move-Interactor
- 3D-Grow-Interactor
- 3D-Rotate-Interactor
- 3D-Selection-Interactor
- 3D-Reposition-Interactor
- 3D-Create-Interactor

The main reason interactors are so successful is that they provide a wide range of parameters that allow the behaviors to be customized. For example, designers can specify how often an object should be reprojected and redrawn as the interactor operates (e.g., continuously as the mouse is moved across the window, or only when the stop-event occurs). Other parameters can be used to specify a feedback-object, abort-events, stop-events, etc. De-

faults values are provided to simplify creating interactors. Shown below are two parameters common to all 3D interactors [Myers 90b].

window - the window that the interactor operates on.

start-where - where the mouse cursor should be for the interactor to start working.

The following parameter is common to all the transformation interactors.

continuous-update - this slot controls how the objects are updated. When the *continuous-update* slot is true, the selected object are reprojected and redrawn continuously as the mouse is moved across the window. In other words the object is continuously reprojected and redrawn from the time the *start-event* is encountered and continues until the *stop-event* is reached. However, if the *continuous-update* slot is false, then the object is not reprojected or redrawn until the *stop-event* occurs. This slot has been designed to accommodate the speed difference of various machines. On a slower machine, the user would probably set this slot to false, because reprojecting an object at each location as it is being dragged across the screen requires considerable processing power.

6.6.1 3D-Move-Interactor

The 3D-move-interactor can be used to move objects in the x, y, and z directions. The interactor translates objects so that their projected image lies under the cursor as shown in Figure 6.7. To accomplish this, the system must first calculate the difference between the screen and world coordinates. This difference is calculated by 1) finding two world coordinate points on opposite ends of the object to be translated; 2) projecting these two 3D world coordinate points to find their associated 2D screen coordinates; and 3) computing the difference between the world coordinate points and screen coordinate points. For example, assume we were given the following world coordinates and their corresponding

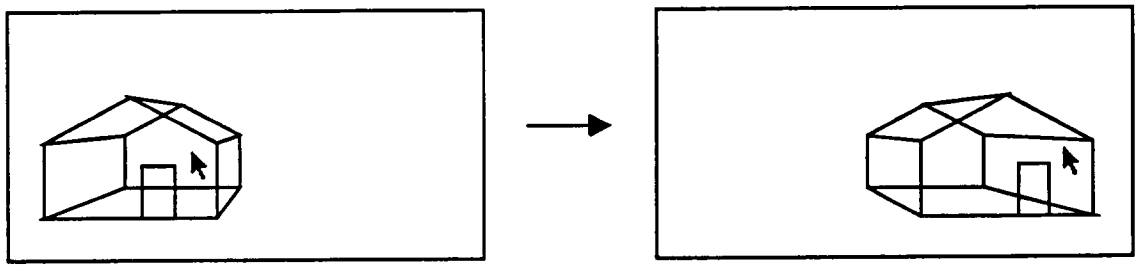


Figure 6.7: Effect of using the 3D-move-interactor.

screen coordinates in pixels.

	world coordinates	screen coordinates
point1	(20, 20, 10)	(100, 100)
point2	(120, 120, 10)	(300, 250)

Given these values, we can now calculate that the x-difference is 2 and the y-difference is 1.5. Therefore, if the user moved an object 50 pixels to the right and 30 pixels toward the bottom, the object would have to be translated by a positive 25 world coordinates in the x direction and a negative 20 in the y direction. This interactor directly sets the x, y, and z parameters of the object that is being moved.[1] Below is an example specification of a 3D-move-interactor.

```
create-instance my-move-interactor 3D-MOVE-INTERACTOR
  start-where = 3D-projection-aggregate;
  direction = xy;
  window = projection-window;
```

6.6.2 3D-Grow-Interactor

The 3D-grow-interactor can be used to resize objects in the x, y, and z directions. When resizing an object, the interactor tries to accurately scale the object so that the object's projected image lies within the bounding box highlighted by the selection-handles as shown in Figure 6.8. To accomplish this, the difference between the screen and world coordinates

[1] When working with 3D-lines, the 3D-move-interactor and the 3D-grow interactor set the parameters ($x1$, $y1$, $z1$) and ($x2$, $y2$, $z2$).

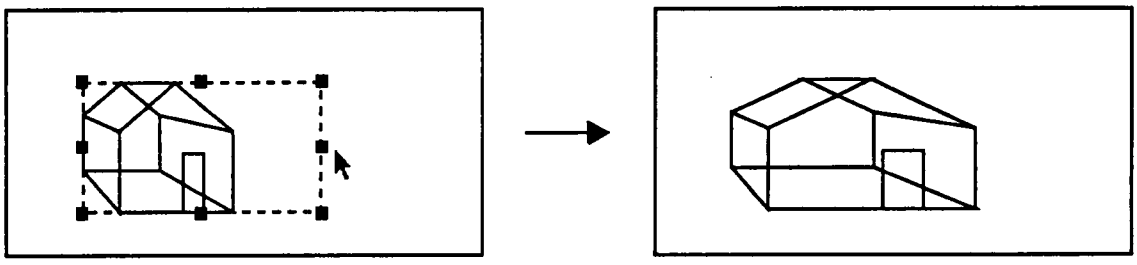


Figure 6.8: Effect of using the 3D-grow-interactor.

is first calculated as discussed in the 3D-move-interactor. The object is then scaled by computing the difference between the original size of the bounding box and the new size of the bounding box when the stop-event is reached.

Resizing an object can be slightly more difficult than moving an object, because it sometimes requires not only scaling but also translation. For example, depending on which selection handle is grabbed — zero, one, or two translations are required so that the position of the object conforms to the position of the bounding box. For example, if the right middle selection handle is grabbed then no translations are required. If the left middle selection handle is grabbed then the object must also be translated in the x direction after being scaled. Finally, if the lower left selection handle is grabbed, then the object must also be translated in both the x and y directions after being scaled. This interactor may directly set any of the following parameters of the object that is being resized: *x*, *y*, *z*, *scale-x*, *scale-y*, and *scale-z*. Below is an example specification of a 3D-grow-interactor.

```
create-instance my-grow-interactor 3D-GROW-INTERACTOR
  start-where = 3D-projection-aggregate;
  direction = xy;
  window = projection-window;
  min-width = 20;
```

To prevent objects from becoming too small, the programmer can set the parameters *min-width*, *min-height*, and *min-length*. These parameters are expressed in pixel values. An object cannot be resized so that its dimensions become less than these parameters. Shown below is the equation that ensures that an object's width does not become less than

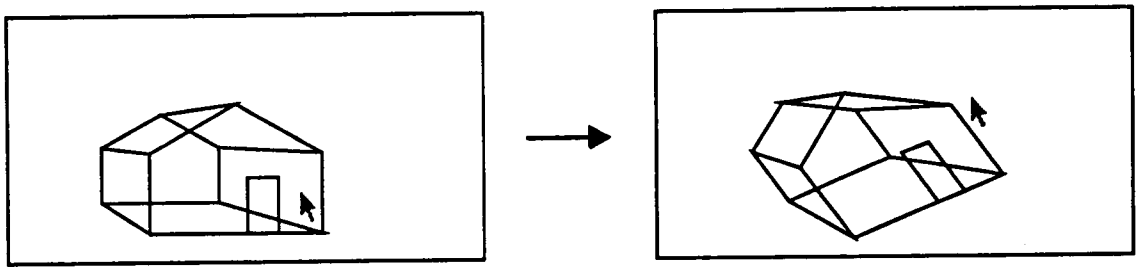


Figure 6.9: Effect of using the 3D-rotate-interactor.

min-width. *new_scale_x* is the initial scale factor calculated by the interactor.

$$scale\_x \;=\; max\;(new\_scale\_x,\; \frac{min\_width}{object.width}) \tag{6.1}$$

Let us look at a specific example. Assume the following:

new_scale_x = 0.1
object.width = 100
min_width = 20

then

scale_x = max (0.1, 20/100) = 0.2

Therefore, the interactor will set scale-x to be 0.2 instead of the initially calculated value of 0.1 so that the object's width remains at least 20 pixels.

6.6.3 3D-Rotate-Interactor

This interactor allows 3D objects to be rotated in either the x, y, or z direction. After an object has been selected, moving the cursor around the center of the object, causes the object to be rotated by an equivalent amount. Thus moving the cursor 30 degrees about the object causes the object to be rotated by an additional 30 degrees. The slot *direction* specifies the axis about which the rotation is performed. Acceptable values are *x*, *y*, and *z*. Figure 6.9 shows this interactor in action, where the direction slot has been set to *z*. This interactor directly sets one of the following parameters *rotate-x*, *rotate-y*, or *rotate-z*. Below

is an example specification for a 3D-rotate-interactor.

```
create-instance my-rotate-interactor 3D-ROTATE-INTERACTOR
  start-where = 3D-projection-aggregate;
  direction = z;
  window = projection-window;
```

6.6.4 3D-Selection-Interactor

The 3D-selection-interactor can be used to select one or more 3D objects. For clarity, handles appear over selected objects. Below is a list of the parameters that can be provided to this interactor to change the default mouse settings.

de-select <mouse-button> : De-selects all objects when the mouse button is pressed while the cursor is not over an object.

region-select <mouse-button> : Allows the user to sweep out a rubberband rectangle. Any object that is completely enclosed within the rectangle is selected.

toggle <mouse-button> : Toggles the selected object beneath the mouse cursor between selected and unselected. If an object was already selected, it removes it from the list of selected objects, otherwise it is added to this list.

The algorithm used for selecting objects depends on whether the user selects a point on the screen, or sweeps out a rectangle. If a point is specified, the x and y values of the mouse button are recorded as a single point. The algorithm then examines all objects that satisfy the predicate in the *start-where* slot to see if this point is contained within their bounding box. If so the selected status of the object is toggled as mentioned earlier. When the user sweeps out a rectangle, any object whose bounding box is completely surrounded by the rubberband rectangle is selected.

The list of selected objects is stored in the parameter *selected-objects* that is associated with the window that the interactor is operating in. This parameter can be read by the programmer or another interactor at any time. Below is an example specification for a

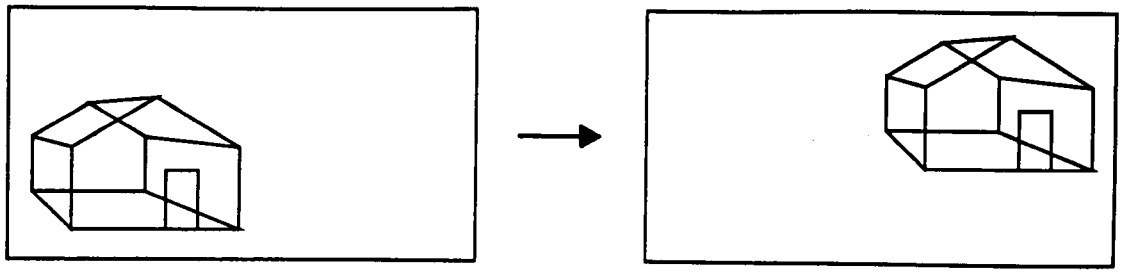


Figure 6.10: Effect of using the 3D-reposition-interactor.

3D-selection-interactor.

```
create-instance my-selection-interactor 3D-SELECTION-INTERACTOR
  start-where = 3D-projection-aggregate;
  window = projection-window;
  de-select = left-mouse-button;
  region-select = middle-mouse-button;
  toggle = right-mouse-button;
```

6.6.5 3D-Reposition-Interactor

The 3D-reposition-interactor can be used to move projected images around on the screen as shown in Figure 6.10. A flag is set while the object is being repositioned so that the object's last projected image is maintained. This flag can later be reset to allow projections to be resumed. Interactive graphical tools can use this interactor to allow a user to temporarily move objects around while editing the scene. The difference between the 3D-move-interactor and this interactor is that the 3D-move-interactor reprojects the object, whereas this interactor moves the originally projected image. Below is an example specification for a 3D-reposition-interactor.

```
create-instance my-reposition-interactor 3D-REPOSITION-INTERACTOR
  start-where = 3D-projection-aggregate;
  window = projection-window;
```

6.6.6 3D-Create-Interactor

The 3D-create-interactor allows the user to interactively construct objects. The interactor operates by the user selecting a location on the screen. A 3D bounding box and a

parameter is set to 1.5, and the *object* slot is set to *cube*, then the interactor creates a cube with the dimensions 30, 60, 120. However, if the axis-scale is set to 0.5, then the interactor creates a cube with the dimensions 10, 20, 40.

There are two unique parameters associated with this interactor that the user can customize. The first parameter is *add-objects-to-agg*. This parameter can be changed to add 3D objects to various aggregates in the window. The second parameter is *wire-frame-p*. This parameter can be changed so that created objects are rendered using either wire-frames or are shaded. When an object is shaded it uses the default filling colors. Below is an example specification for a 3D-create-interactor.

```
create-instance my-create-interactor 3D-CREATE-INTERACTOR
  start-where = window-aggregate;
  add-object-to-agg = 3D-projection-aggregate;
  object = cone;
  window = projection-window;
```

Chapter 7

Direct Manipulation Layer

7.1 Introduction

The top layer of our 3D toolkit provides an interactive graphical environment for creating 3D scenes. This layer allows the user to interactively display and transform 3D objects, perform clipping, create aggregate objects, modify the viewing parameters that control the transformations of objects, and construct polyhedron objects. To support these operations, the toolkit includes the following interactive editors:

- Scene Editor
- Polyhedron Editor
- Aggregate Editor

The interactive environment is implemented using the programming layer, and thus is itself an application of the 3D toolkit.

7.2 Scene Editor

The scene editor is used to create the final appearance of objects and aggregates in the 3D scene. This editor is composed of the following windows. The *scene window* allows the user to interactively create objects and apply transformations to them. The *scene control window* provides fine grain control over the entire 3D viewing process. The *viewing parameter windows* allow the user to interactively specify the 3D view volume that determines

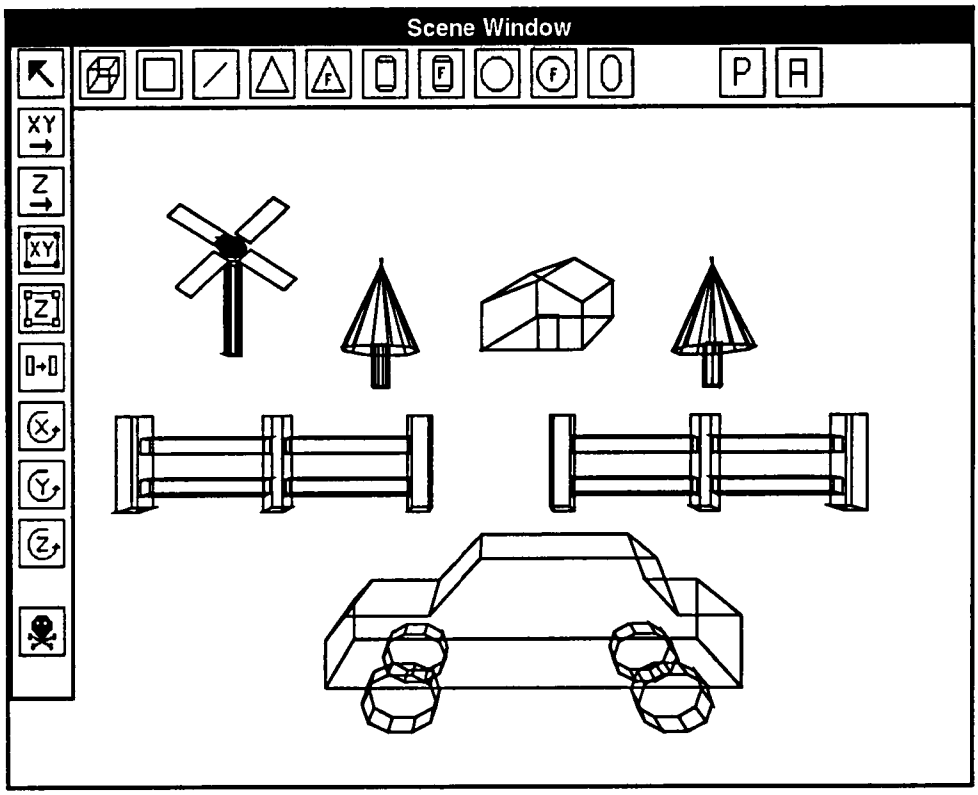


Figure 7.1: The scene window.

how objects will be projected. The *change center of rotation window* allows the user to interactively specify a new center of rotation for a 3D object or an aggregate. Each of these windows in described in more detail in the following sections.

7.2.1 Scene Window

The scene window is primarily used for creating and positioning objects in the scene as shown in Figure 7.1. The position and appearance of objects in the scene window are controlled by the various transformation options on the left side of this window. Objects can be added to the scene window by selecting the appropriate icon located at the top of this window. The entire scene illustrated in this window was created in less than one hour using the direct manipulation layer. We will now examine the icons on the left side of the scene window in more detail.

 **Selection** - activates a 3D-selection-interactor that allows a user to select objects in the scene window.

 **XY Move** - activates a 3D-move-interactor and sets the direction variable to "xy".

 **Z Move** - activates a 3D-move-interactor and sets the direction variable to "z".

 **XY Resize** - activates a 3D-grow-interactor and sets the direction variable to "xy".

 **Z Resize** - activates a 3D-grow-interactor and sets the direction variable to "z".

 **Reposition** - activates a 3D-reposition-interactor. This interactor allows a user to temporarily move objects that are obscuring a portion of the display they are editing. An object will remain at this temporary location until the "project" method of the repositioned object is called. An object's "project" method can be called by directly manipulating the object using another interactor or by declaratively changing the value of an interesting parameter.

 **X Rotate** - activates a 3D-rotate-interactor and sets the direction to "x".

 **Y Rotate** - activates a 3D-rotate-interactor and sets the direction to "y".

 **Z Rotate** - activates a 3D-rotate-interactor and sets the direction to "z".

 **Remove** - deletes all currently selected objects.

When one of the following icons located at the top of this window is selected, the *object* parameter of the 3D-create-interactor is set to that object. For example, if the icon resembling a cone is selected, then the *object* parameter is set to "cone". The user can then interactively create various sized cones anywhere in the scene window. More specifically, these icons allow the user to create cubes  , planes  , 3D-lines  , cones  , fast cones  , cylinders , fast cylinders , spheres , fast spheres , and ellipsoids .

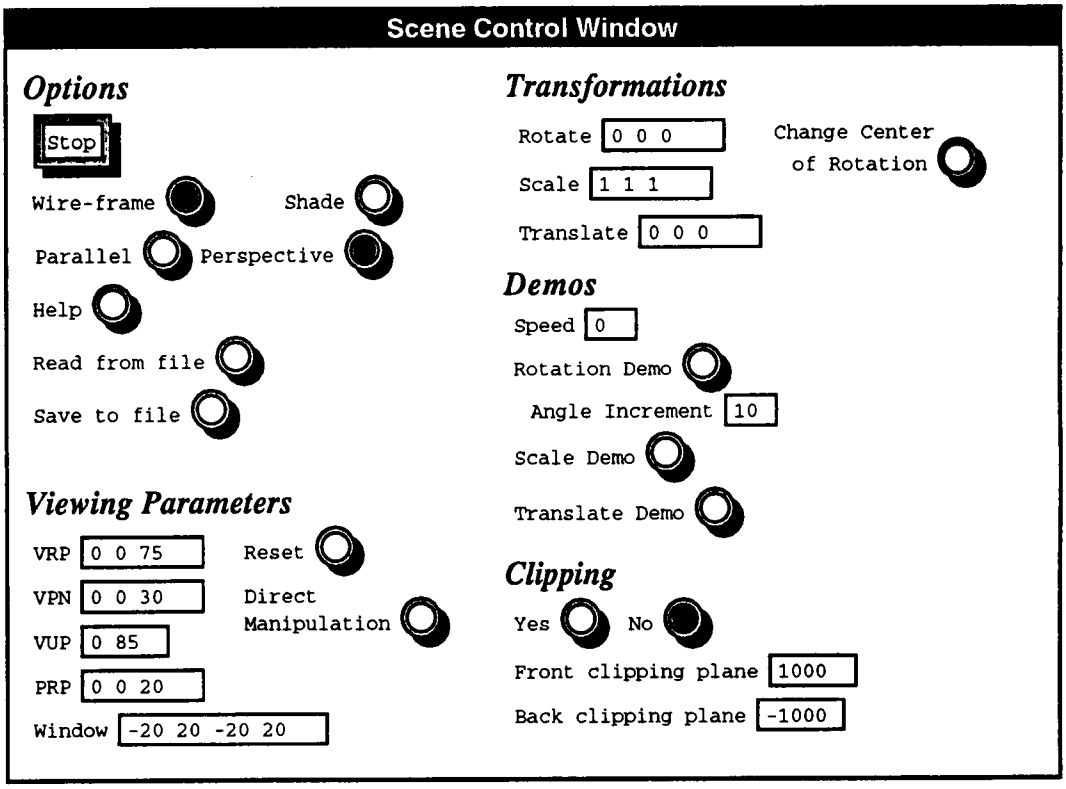


Figure 7.2: The scene control window.

By pressing the [P] icon, the polyhedron editor will be displayed and the user can begin constructing a polyhedron object. By pressing the [A] icon, the aggregate editor will be displayed and the user can begin constructing an aggregate object.

7.2.2 Scene Control Window

The scene control window (Figure 7.2) provides fine grain control for the overall 3D viewing process in contrast to the scene window which only allows approximate placement of objects. For example, this window allows the user to precisely set the viewing parameters, the clipping parameters, and apply transformations to objects. In the example scene shown in Figure 7.1, the scene control window was used to set the rendering mode to "wire-frame" and the projection style to "perspective". We now examine the features of the scene control window in more detail.

7.2.2.1 Options

The top left corner of the scene control window contains several features that allow the user to change the rendering mode, the projection type, and to write and read scenes to and from files:

Stop - exits the 3D toolkit.

Wire-frame / Shade - specifies the rendering mode as either wire-frame or shaded. This option only applies to the selected objects in the scene window and their components. Changing the rendering mode of a selected 3D aggregate changes the rendering mode for all its components.

Parallel / Perspective - specifies the projection type as either parallel or perspective.

Help - provides general help on the 3D toolkit. Clicking the middle mouse button on any text in the scene control window causes the scene control editor to display a helpful message on the option or parameter selected.

Read from file - allows the user to read scenes from a file. A scene may consist of one or more objects.

Save to file - saves all the objects in the scene window to a file. This feature uses the Garnet save mechanism described in [Myers 92b]. Scenes that are saved using this feature are represented using code from the programming layer, so the designer is free to modify the code in any way desired.

7.2.2.2 Viewing Parameters

The viewing parameters can be changed declaratively by using the text boxes in this window or interactively by pressing the "Direct Manipulation" button:

VRP - changes the value of the view reference point in the scene window. The VRP is specified in world coordinates.

VPN - changes the value of the view plane normal in the scene window. The VRP is specified in world coordinates.

VUP - changes the value of the view up vector in the scene window. The VUP is specified in viewing reference coordinates.

PRP - changes the value of the projection reference point in the scene window. The PRP is also specified in viewing reference coordinates. The first coordinate represents the value of the u axis, the second coordinate represents the value of the v axis, and the third coordinate is the n axis.

Window - changes the value of the window size in the scene window. The window resides on the view plane. Any part of an object that projects onto the view plane outside this window is not displayed. Four coordinates are required to specify the window size.

Reset - resets the viewing parameters back to their default values.

Direct Manipulation When this option is selected, four windows are created that allow the user to interactively modifying the viewing parameters using pictorial diagrams rather than the text boxes shown in Figure 7.2. The viewing parameters include the view reference point, view plane normal, view up vector, projection reference point, and the window size. The four windows that appear when the "Direct Manipulation" button is pressed are listed below.

Viewing Control Window - specifies the axis scale to be used and allows the user to terminate the direct manipulation of these parameters.

View Plane Window - allows the user to modify the view reference point and the view plane normal that together define the view plane.

Viewing Reference Window - allows the user to directly modify the view up vector and the window size. Together these two pieces of information create the viewing

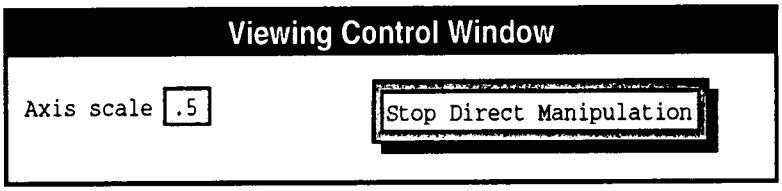


Figure 7.3: The viewing control window.

reference coordinate (VRC) system.

Perspective Reference Point Window - allows the user to interactively change the perspective reference point.

The view reference point, view plane normal, and projection reference point are specified using 3D points. Because of the commonly used 2D input devices (like the mouse), two different mouse buttons are used to change the values of these parameters. Otherwise ambiguous results might be obtained. One mouse button is used to change these parameters in the x and y directions, and another mouse button is used to change these parameters in the z direction.

Viewing Control Window This window allows the user to specify an axis scale that is used while interactively modifying the viewing parameters as shown in Figure 7.3. The axis scale parameter determines the ratio between screen and world coordinates when interactively modifying the following parameters: view reference point, view plane normal, view up vector, perspective reference point, and window size. As an example, if axis-scale is set to 1.5 and the perspective reference point (PRP) is moved 100 screen pixels to the right, this would result in the x-value for the PRP being increased by 150 world coordinates. However, if the axis-scale is set to 0.5 and the PRP is moved 100 screen pixels to the right, this would result in the x-value for the PRP being increased by only 50 world coordinates. Thus the axis-scale parameter controls the granularity of changes caused by moving a point with the mouse.

This window also contains the option "Stop Direct Manipulation" that terminates the

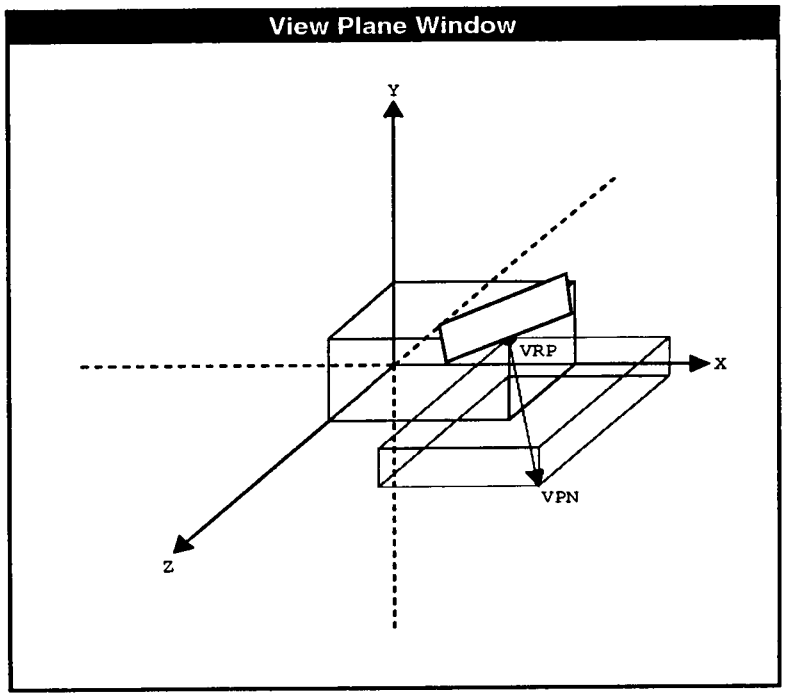


Figure 7.4: The view plane window.

direct manipulation of the viewing parameters. Pressing this option removes all four viewing parameter windows from the screen.

View Plane Window This window allows the user to change the values of the view reference point and view plane normal as shown in Figure 7.4. The shaded plane denotes the orientation of the view plane. The view reference point can be directly changed by moving the circle labeled VRP to the desired location. Similarly, the view plane normal can be changed by grabbing the end of the vector labeled VPN and moving it to the desired location.

Viewing Reference Window. This window allows the user to change the window size and the direction of the view up vector as shown in Figure 7.5. The origin of the viewing reference window is the view reference point (VRP).

The window size can be interactively changed by selecting the window with the mouse.

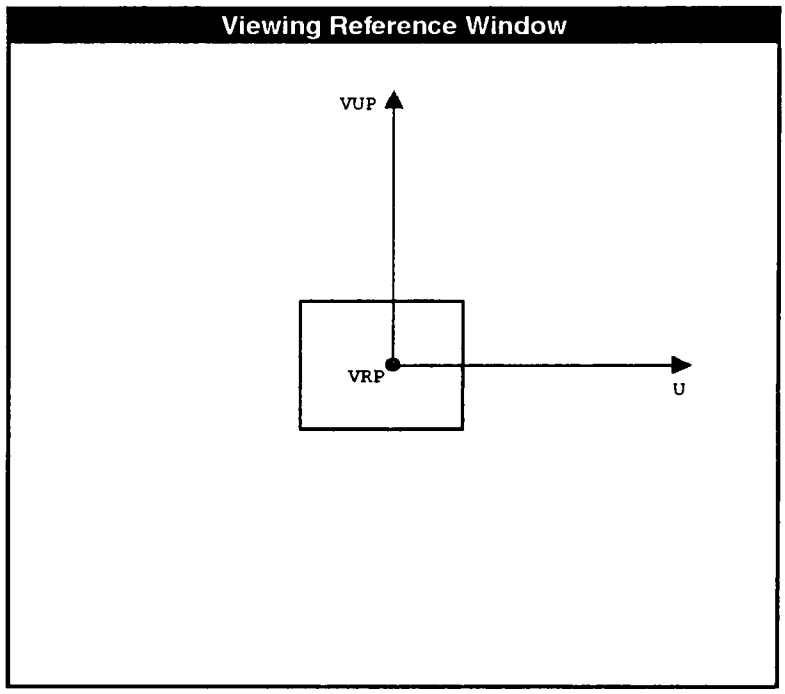


Figure 7.5: The viewing reference window (shaded rectangle) and the two vectors that define its coordinate system.

When this happens, resize and selection handles appear around the perimeter of the window. The user can then resize the window by dragging one of the resize handles, or move the window by dragging one of the selection handles. The direction of the view up vector can be changed by grabbing the end of the vector labeled VUP and dragging it to the desired position. When the mouse button is released, the window will be redrawn in accordance with the new VUP. Figure 7.6 illustrates how the rotation of the window coincides with the VUP. Although, the u vector cannot be selected, it will always remain at a 90 degree angle with the VUP.

Perspective Reference Point Window This window allows the user to change the value of the projection reference point (PRP) as shown in Figure 7.7. The PRP is defined relative to the 3D viewing reference coordinate (VRC) system. The origin of the VRC system is the VRP. The n axis is the VPN vector. The perspective reference

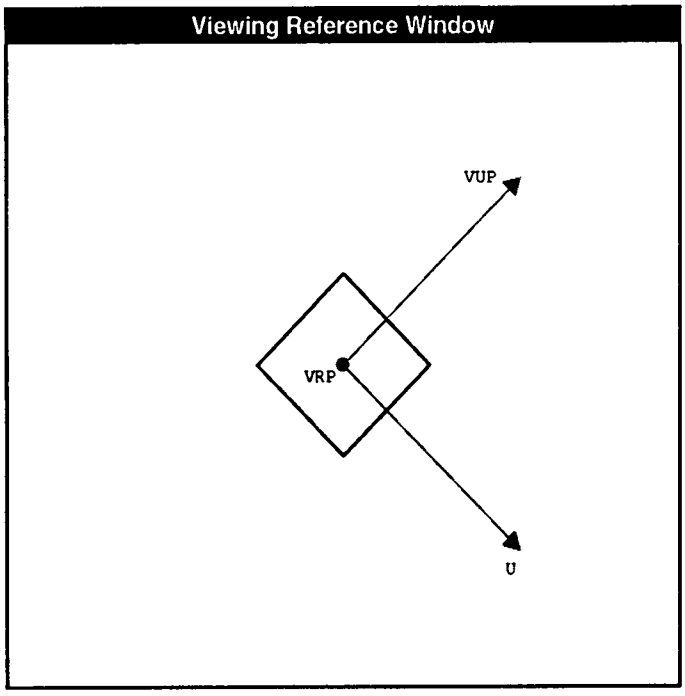


Figure 7.6: The VUP vector has been rotated 45 degrees counter clockwise about the VRP. Consequently, the window is also rotated by the same amount so that its orientation matches the VUP.

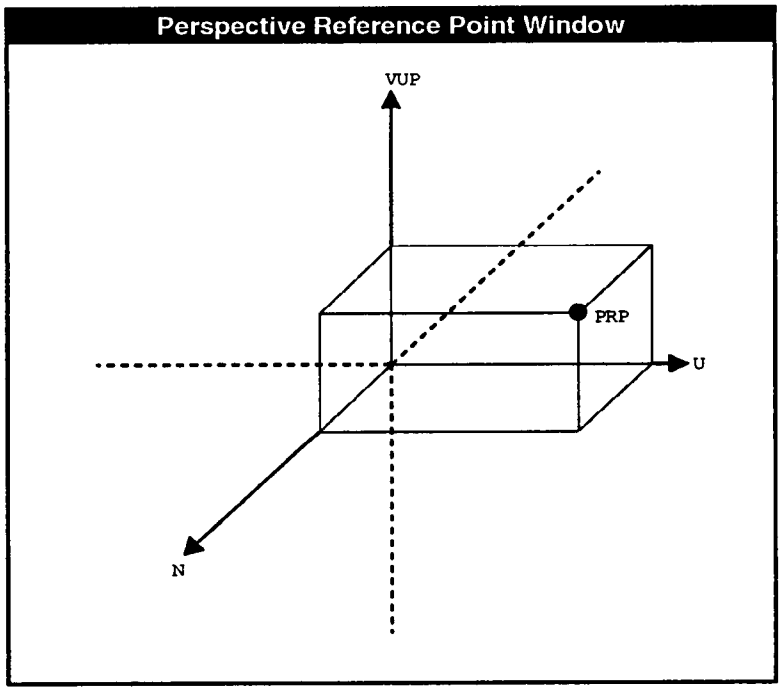


Figure 7.7: The perspective reference point window.

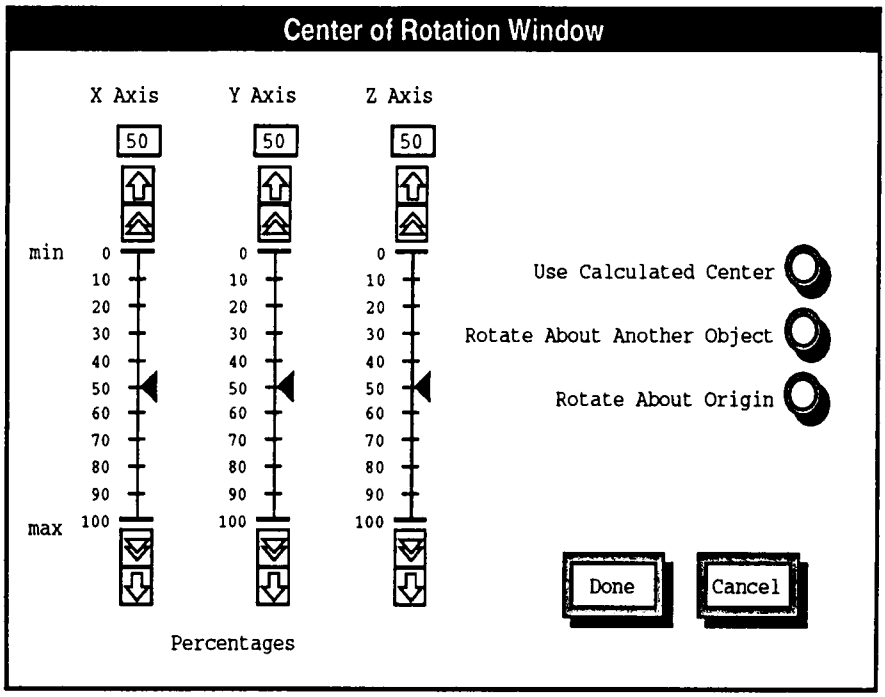


Figure 7.8: The change center of rotation window allows the user to interactively specify a new center of rotation for either a 3D object or a 3D aggregate.

point is changed by moving the circle labeled PRP to the desired position.

7.2.2.3 Transformations

The transformation boxes display the current transformation parameters for the selected object and provide an alternative way to modify the parameters (i.e., they provide finer grain control than the interactors in the scene window). If the user changes the values in one of these boxes, then these transformations are applied to all currently selected objects in the scene window.

Change Center of Rotation Window This window allows the user to interactively specify a new center of rotation for either a 3D object or a 3D aggregate. The center of rotation can be specified to lie within the object, within another object, or about the origin. The rotation values can be specified by using sliders or by selecting one of three predefined options as shown in Figure 7.8:

Sliders - can be used when the desired center of rotation is within the object or aggregate. A different slider has been created for each axis. The value of a slider represents a percentage of the object. For example, to rotate an object about its own center the sliders should all be set at 50. Similarly, to rotate an object about its smallest x coordinate the "X Axis" slider should be set to 0.

Use Calculated Center - rotates the selected object about its center. Selecting this option sets the x, y, and z axis sliders to 50.

Rotate About Another Object - allows an object to be rotated about a different object. This other object is selected after choosing this option.

Rotate About Origin - rotates the object about the origin (0, 0, 0) of its aggregate.

7.2.2.4 Demos

The options in this section allow the user to apply some pre-defined demo transformations to the objects in the scene window. These demos are designed to assist in developing a 3D scene, by giving the user a quick idea of how the objects might appear at different angles, sizes, and locations. After the demo transformations have been applied, the objects are returned to their original transformation values.

Speed - ceases execution of the specified demo for n seconds of real time between operations, where n can be a fractional number. This option applies for all three of the demos listed below.

Rotation Demo - rotates all currently selected object(s) 360 degrees about the x axis, then the y axis, and finally the z axis. The box labeled "Angle Increment" determines the angle between successive rotations along each axis. For example, if "Angle Increment" equals 1 then the selected objects are rotated 360 times in each of the x, y, and z directions. If "Angle Increment" equals 10, then the selected objects are rotated 36

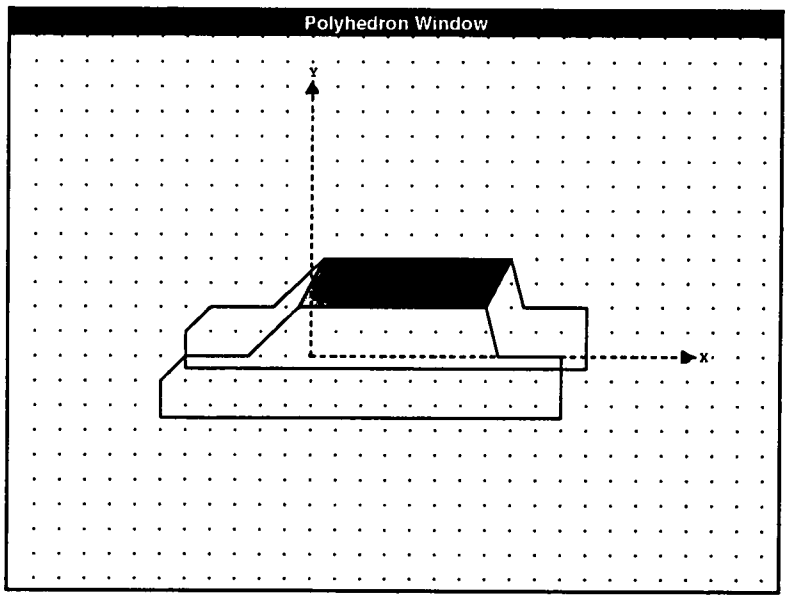


Figure 7.9: The polyhedron window.

times about each axis. In this latter case, the objects would be rotated by the angles: 10, 20, 30, ... 340, 350, 360.

Scale Demo - scales all selected object(s) in the x, y, and z directions by some pre-defined values.

Translate Demo - translates all selected object(s) in the x, y, and z directions by some pre-defined values.

7.2.2.5 Clipping

The options in this section allow the user to determine which portion of objects to display. When clipping is turned on, the text box labeled "Front clipping plane" specifies the front clipping plane, and the text box labeled "Back clipping plane" specifies the back clipping plane.

7.3 Polyhedron Editor

The polyhedron editor (Figures 7.9 and 7.10) allows the user to quickly and interactively

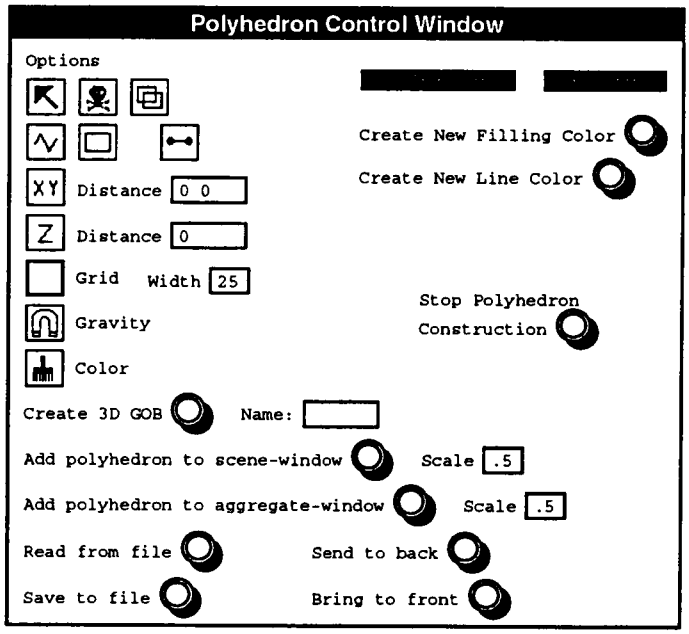


Figure 7.10: The polyhedron control window.

create polyhedron objects. The constructed object can be rotated, scaled, and translated just like all the predefined 3D objects. The user is completely freed of having to specify points, and then defining how the points should be connected to form a surface. Instead, the user interactively creates surfaces using a polyline, a rectangle, or by connecting pre-existing points. These surfaces can then be moved in the xy direction, or in the z direction. When gravity is on, objects snap to grid markers. The filling style for objects can be turned on and off. Finally, this editor allows objects to be selected, destroyed, or copied.

Points in the polyhedron window correspond to world coordinate points. The house and the body of the car shown in Figure 7.1 were created using these windows. To construct the car body the user performed the following actions:

1. Selected the "polyline creator" from the polyhedron control window (Figure 7.10) and traced out the shape of the side of a car in the polyhedron window (Figure 7.9).

2. Pressed the "copy" option to create an instance of this side.

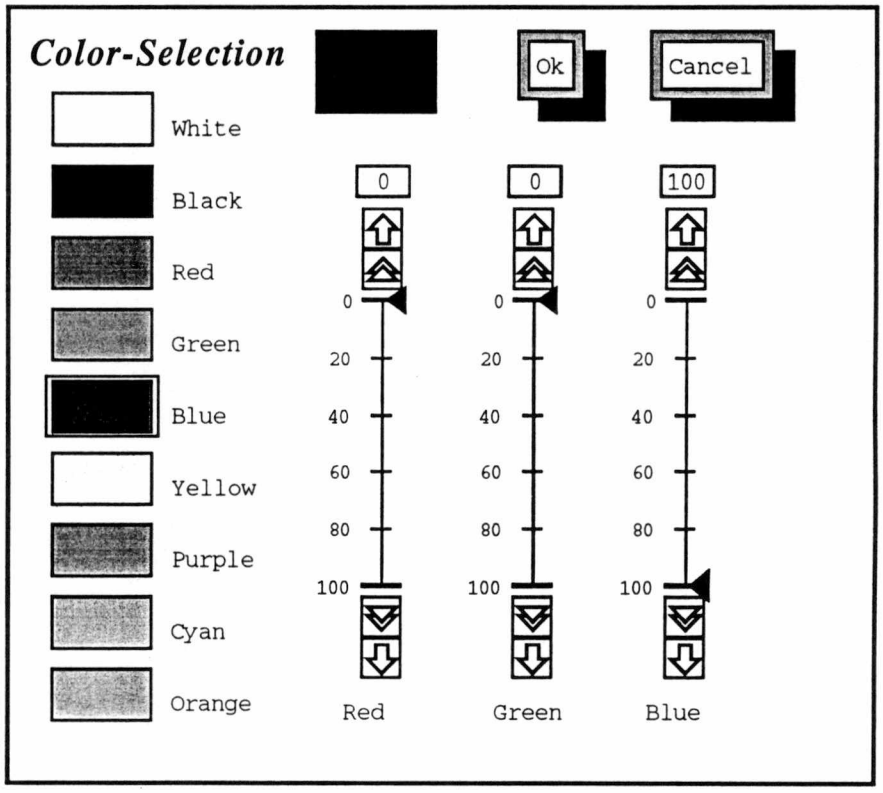


Figure 7.11: The visual color editor allows the user to interactively create new colors with which to shade objects and lines. Colors can be selected either by choosing one of the colors in the menu on the left or by manipulating the three sliders on the right. The color box at the top provides feedback showing the selected color.

3. Used the "z move" [Z] option to move this instance in the z direction.

4. Used the "connect points" [⊷] option to connect the two sides.

Figure 7.9 shows the car after both sides and the top have been created. The user added this object to the aggregate window by selecting the option "Add polyhedron to aggregate-window". By pressing either the button "Create New Filling Color" or "Create New Line Color" the visual color editor shown in Figure 7.11 is displayed.

7.4 Aggregate Editor

The aggregate editor (Figures 7.12 and Figure 7.13) allows the user to interactively create 3D aggregates. This editor has the same interface as the scene editor, but it has

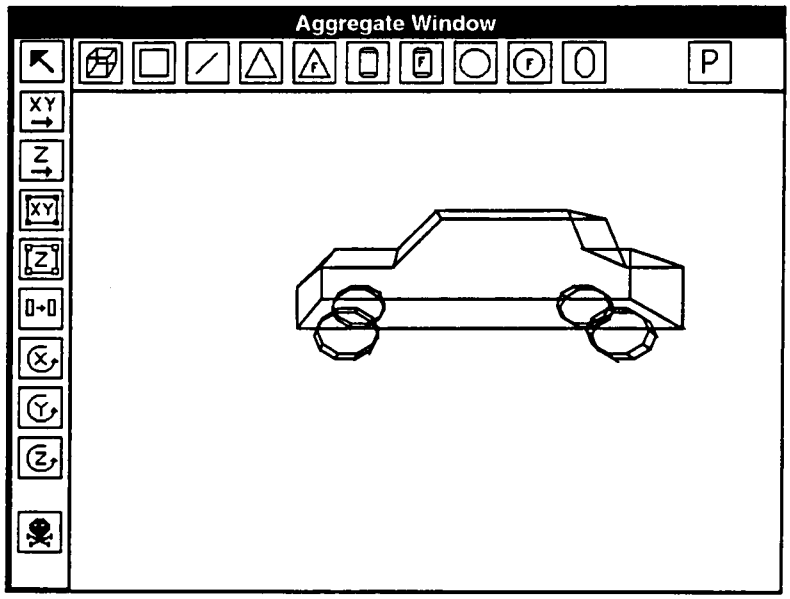


Figure 7.12: The aggregate window.

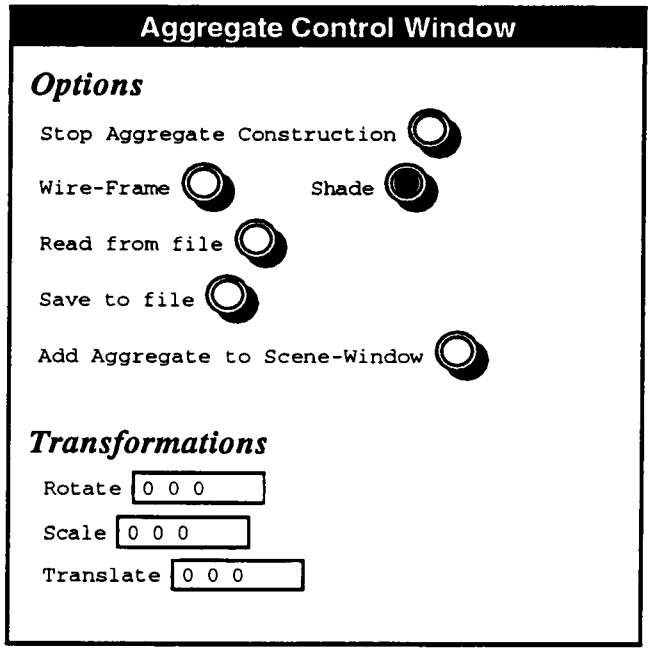


Figure 7.13: The aggregate control window. The transformation options in this window apply to the individual components rather than the entire aggregate. The option "Add Aggregate to Scene-Window" creates an instance of the aggregate used in this window and adds it to the scene window. The user can then apply transformations to the entire aggregate once it has been added to the scene window.

a somewhat different purpose. The purpose of this editor is to lay out the components of an aggregate, whereas the purpose of the scene editor is to lay out the components of a scene. Figure 7.12 illustrates the car shown in Figure 7.1 being constructed. The body of the car was created in the polyhedron window (Figure 7.9) and then added to this window. The wheels are just cylinders that were created by selecting the cylinder icon located at the top of this window. The cylinders were then rotated, translated, and scaled using the interactive options on the left side of this window.

The trees, windmill, and fence shown in the scene window are also aggregate objects created using this window. The trees are composed of a cone on top of a cylinder. The windmill is constructed from four planes, one sphere, and one cylinder. Finally, the fence is constructed by creating four instances of a section of the fence. Each section of the fence (two post and two cross beams) was created from four cubes. More than one instance of an aggregate can be added to the scene window by repeatedly pressing the option "Add Aggregate to Scene-Window."

The aggregate control window is shown in Figure 7.13. The options found in this window are similar to those options found in the scene control window with an additional feature "Add Aggregate to Scene-Window" that creates an instance of the 3D aggregate in the aggregate window and adds this aggregate to the scene window. The objects in the aggregate window represent the components of the newly created aggregate in the scene window. After the 3D aggregate has been created in the scene window, any transformations to the objects in the aggregate window are reflected in the scene window. This allows the lay out of aggregates to be altered even after they have been added to the scene window. The scene window only allows alterations to the top-level objects. If objects are deleted from the aggregate window, these same objects will automatically be deleted from any instance of this aggregate that has been added to the scene window. However, if new objects are added to the aggregate window, these objects will not automatically be added to any instance of this aggregate in the scene window. Instead, the user must create a new instance of this

aggregate and add it to the scene window by again using the feature "Add Aggregate to Scene-Window".

7.5 Integration of the Extended Constraints

The 3D toolkit has provide a useful test of the augmented constraint system, since many of the interactive editors needed to share information and this sharing was most easily accomplished by using multi-output and multi-way constraints. For example, the scene window, scene control window, and the viewing parameters windows (all part of the scene editor) need to maintain consistent views of the viewing parameters and transformation parameters, most of which are expressed as 3-tuples of values representing the x, y, and z axes. These 3-tuples are stored as individual values, so multi-output, multi-way constraints are the best way to transmit and unpack these tuples. More specifically, multi-output and multi-way constraints are used to maintain consistency among: the pictorial points, vectors, and window coordinates illustrated in the viewing parameters windows; the values stored in the text boxes shown in the "Viewing Parameters" section of the scene control window; and the parameter slots associated with the scene window that determine how the 3D objects are projected. For example, if the circle labeled PRP in the projection reference point window is moved, the text box labeled PRP in the scene control window is updated, and all objects in the scene window are automatically reprojected and redrawn using the new parameters.

Multi-output, multi-way constraints are used to control the flow of information between the transformation boxes in the scene control window and the transformation slots in the selected objects. In one direction, the values of a transformation box are broadcasted to all selected objects. In the other direction, the first selected object broadcasts its transformation coordinates to the appropriate box.

Multi-way constraints are also used in the visual color editor shown in Figure 7.11. This window used multi-way constraints to establish relationships among the following values:

- The indicators located on the sliders.
- The value boxes located above the sliders.
- The feedback box possibly surrounding one of the pre-defined colors.

For more details on this multi-way constraint see Section 3.4.

Structural constraints played a lesser role in this particular application. However, there are many applications where structural constraints would be helpful. For example, in some interfaces the type of feedback object used to highlight a selection depends on the size and type of the selection. An example of this was discussed in Section 3.3. Structural constraints could be used in these examples to avoid having to program this type of behavior by hand.

Chapter 8

Conclusions and Directions for Future Research

8.1 Conclusions

In this dissertation we have described several extensions to the conventional spreadsheet programming model. These extensions include multi-output constraints, structural constraints, and multi-way constraints. Together they have enhanced the power and flexibility of the spreadsheet programming paradigm. These new constraints increase efficiency, allow more powerful relationships to be expressed, and facilitate maintaining consistency between data structures. The significant contributions of this dissertation include:

1. The design and implementation of a multi-output constraint scheme that allows constraints to have a dynamic number of inputs and outputs.

2. The design and implementation of a structural constraint scheme using an active value model. The constraints can involve more than two objects and allow arbitrary relationships to be described and maintained.

3. The design and implementation of a multi-way constraint scheme that supports the following five features: dynamic numbers of inputs and outputs; pointer variables for the inputs and outputs; multiple one-way constraints that can be attached to a single slot; priorities; and activation conditions.

4. The design and implementation of an object-oriented toolkit for creating highly interactive, direct manipulation, 3-dimensional user interfaces. The toolkit provides 1) a high-level event handling model called interactors, 2) a constraint model that includes multi-output, structural, and multi-way constraints, and 3) an object model that supports aggregates of objects and structural inheritance. The toolkit also provides a set of interactive, direct manipulation tools for creating 3D objects and 3D scenes.

The use of these tools has yielded the following major findings and results:

- Multi-output constraints result in faster constraint solving by allowing the reuse of computations. They also give the programmer more control over the granularity of the constraints. Finally, they help reduce the number of formula objects maintained by the system which results in fewer bookkeeping operations on the part of the constraint solver.

- Structural constraints greatly simplify application programs by allowing constraints to control the structure of objects. Action-procedures are used to create the new structure that is computed by the constraints. Each action-procedure modifies only a local piece of an object's structure. The constraint solver ensures that the appropriate set of action-procedures are executed to update all of the object's structure. By allowing the programmer to consider only the local effects of an action-procedure and having the constraint solver handle the global effects, the programmer is freed from having to perform tedious bookkeeping operations. Our experience has shown that structural constraints greatly simply application programs and also prevent bookkeeping errors from occurring.

- Activation conditions and priorities provide a useful technique for controlling the satisfaction of multi-way constraints. By giving the programmer control over the planning process rather than the constraint solver, this technique can make constraint satisfaction both more predictable and more efficient.

- Extending multi-way constraints to handle pointer variables, variable numbers of inputs and outputs, and structural constraints can significantly reduce the complexity of maintaining consistency among multiple views of data. This reduction of complexity is achieved by only requiring the programmer to specify the direct effects of an operation (e.g., the changing of the graphical properties of an object) and using a constraint solver to handle the indirect effects of an operation (e.g., translating the changes in the graphical properties to changes in a central data structure and then propagating the changes to other views of the data structure).

- Object-oriented toolkits that support constraints and a high-level behavioral model greatly facilitate writing interactive 3D applications. Traditionally, this task has been a very time consuming and difficult process. Most available toolkits use imperative programming, whereas the toolkit described in this dissertation takes a declarative object-oriented approach. It includes many high-level features that have typically been ignored in 3D toolkits, such as direct manipulation, high-level input handling, and constraints. This toolkit also provided a test platform for experimenting with the extensions to the conventional spreadsheet programming model.

- The integration of high-level constraints with 3D objects is very helpful in expressing relationships among 3D objects because the programmer is freed from having to remember their varying internal structure. For example, the programmer is no longer required to know whether the size of a 3D object is determined by a *radius* or a *width* field.

8.2 Future Work

There are a number of interesting directions for future work in this area. One useful extension would be the ability to interactively specify multi-output, structural, and multi-way constraints. The ability to interactively specify "one-way" constraints is provided

in Lapidary [Myers 89a] and C32 [Myers 91a]. Currently, our design requires that the programmer write code to specify these sometimes complex relationships. It might be easier if these relationships could be specified interactively for both 2D and 3D objects using a click and point approach. For the more frequently used variables like position and size, an iconic palette could be constructed as shown in Figure 8.1. This icon palette is based on Lapidary's 2D design, but allows the user to specify multi-way constraints between 3D objects. To illustrate how the iconic palette would function, consider the "X direction" constraint icon located in the upper left corner of the palette window. The possible predefined constraints are listed below. The symbol $\Longleftrightarrow$ represents a multi-way constraint.

left-outside - the right side of object2 $\Longleftrightarrow$ the left side of object1

left-inside - the left side of object2 $\Longleftrightarrow$ the left side of object1

center - the center of object2 $\Longleftrightarrow$ the center of object1

right-inside - the right side of object2 $\Longleftrightarrow$ the right side of object1

right-outside - the left side of object2 $\Longleftrightarrow$ the right side of object1

The user could specify an optional *offset* for each multi-way constraint. An additional constraint window might also be constructed for constraining 3D lines. For less frequently used variables, an interface might display two objects along with a list of their variables. The user could then select the variables that would participate in the multi-way constraint. As feedback, a double arrowed line might be created between the variables.

It might also be helpful to develop a set of tracing and debugging tools for these extended constraints. Debugging constraints can be a very difficult task because of the way values can propagate. For example, changing a value in one object might cause changes in other objects downstream. Furthermore, our extended constraints can use pointer variables which make the debugging process even more complicated. The ability to visual monitor the

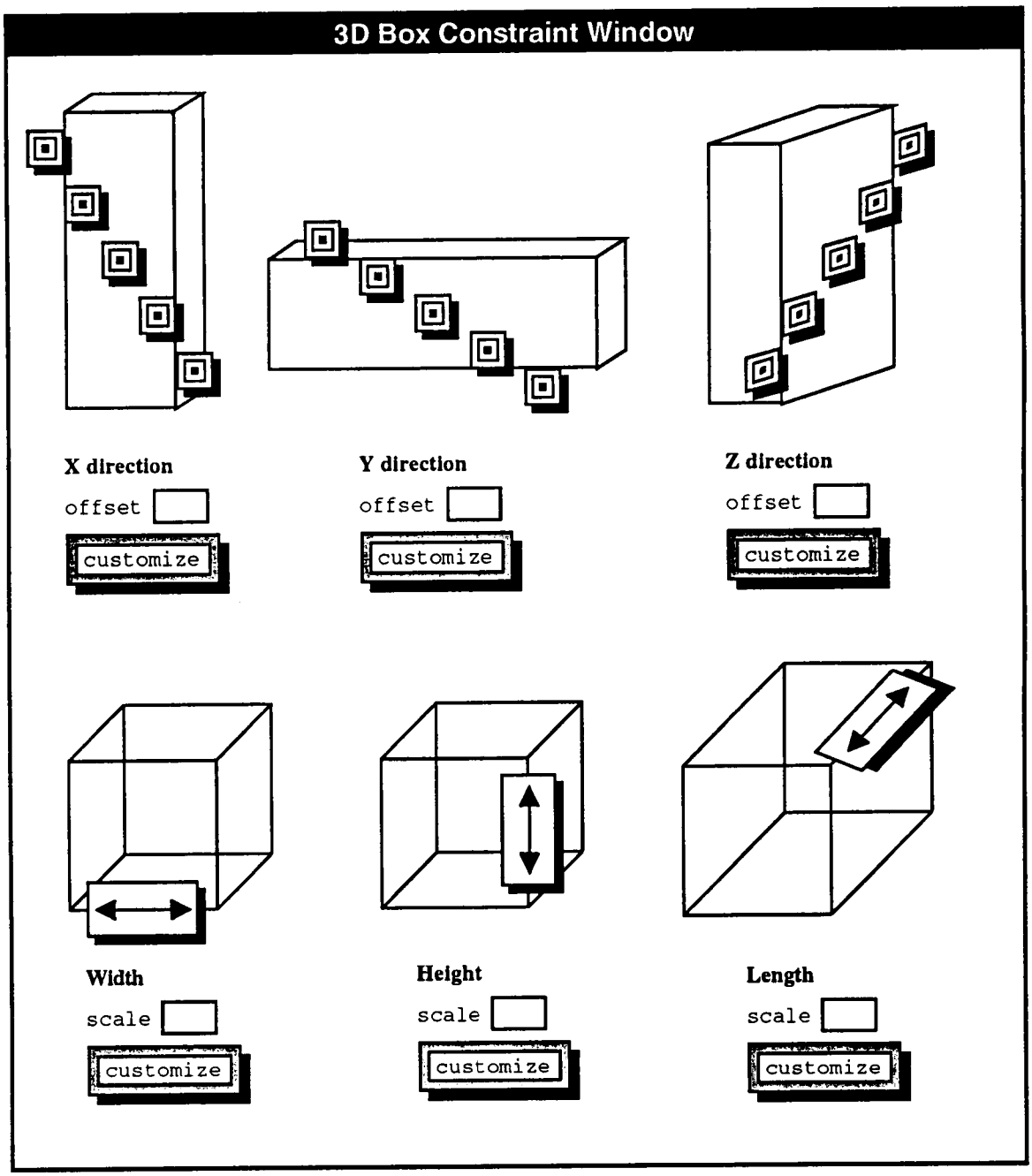


Figure 8.1: A prototype of the window that could allow the user to interactively define a multi-way constraint between 3D objects on either their position or their size.

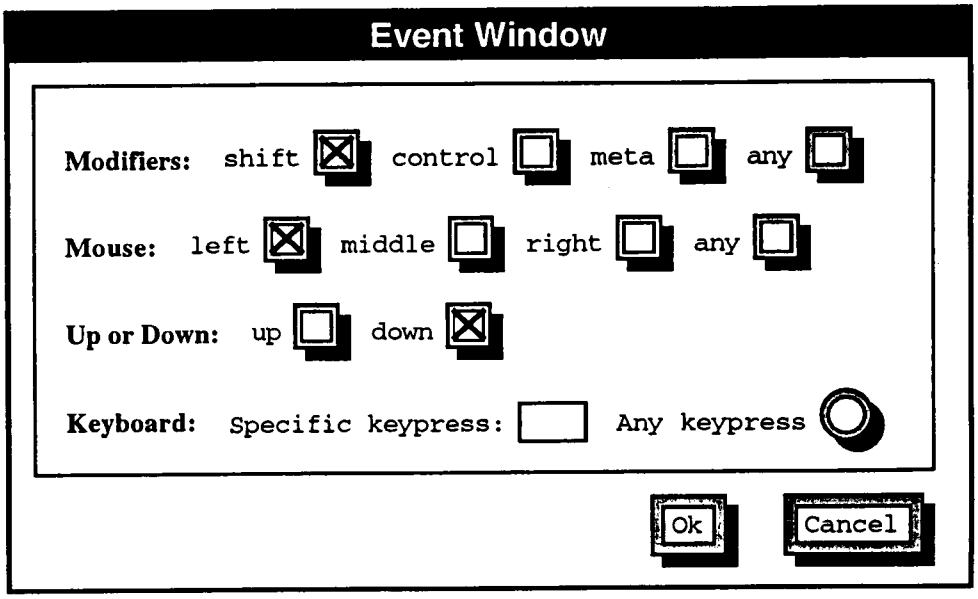


Figure 8.2: A prototype of the event window for 3D interactors. For example, if the user was specifying the start-event in this window, then this interactor would start executing when the user pressed the *left* mouse button *down* while holding down the *shift* key.

constraint solving process might be very helpful in debugging constraints. One possible visual technique might involve assigning visual representations to variables and then blinking these representations as the variables' values are changed through propagation.

Another useful extension would be to use dialog boxes to create 3D interactors. The design of these 3D interactor dialog boxes could be based on Lapidary's 2D dialog boxes [Myers 89a]. Currently, code must be written to define these interactors. The dialog boxes could make it substantially easier for a beginner or non-programmer to construct 3D interactors. Most of the 3D interactors have over 20 parameters that can be customized. For example, a start-event parameter must be specified to indicate when the interactor should begin executing. An event window (similar to Lapidary's) might be constructed as shown in Figure 8.2 to help the user specify the start-event. While reducing time, these interface windows would also prevent unintentional errors from occurring. For example, the dialog boxes would make it impossible for the user to inadvertently specify the start-event as *left-shift-down* instead of *shift-left-down*, (the correct specification).

Bibliography

Bibliography

[Barth 86] Paul Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[Duisberg 86] Robert Duisberg. *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System.* PhD thesis, Department of Computer Science, University of Washington, Seattle, WA, 1986. Also available as University of Washington Technical Report No 86-09-01.

[Egbert 92] Parris K. Egbert and William J. Kubitz. Application graphics modeling support through object orientation. *IEEE Computer*, pages 84–91, October 1992.

[Foley 90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics - Principles and Practice.* The Systems Programming Series. Addison-Wesley Publishing Company, 1990.

[Foley 92] James Foley, Bill Glazier, Kurt Akeley, Murray Cantor, Mark Goldstein, Marty Hess, and Jeff Stevenson. 3d graphics standards debate: Pex versus opengl. In *Computer Graphics*, volume 26, pages 408–409. Proceedings SIGGRAPH'92, July 1992.

[Freeman 90] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. The deltablue algorithm: An incremental constraint hierarchy solver. Technical report, Department of Computer Science and Engineering, University of Washington,

February 1990. Also available as University of Washington Technical Report No 89-08-06.

[Freeman 92] Bjorn N. Freeman-Benson and Alan Borning. *Constraint Imperative Programming Languages for Building Interactive Systems*, pages 161–181. Jones and Bartlett Publishers, Boston, MA, 1992.

[GKS 88] International Standards Organization. *International Standard Information Processing Systems - Computer Graphics - Graphical Kernel System for Three Dimensions (GKS-3D) Functional Description, ISO Document Number 8805:1988(E), American National Standards Institute, New York*, 1988.

[Hill 92] Ralph D. Hill. *Languages for the Construction of Multi-User Multi-Media Synchronous (MUMMS) Applications*, pages 125–143. Jones and Bartlett Publishers, Boston, MA, 1992.

[Hill 93] Ralph D. Hill. The rendezvous constraint maintenance system. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 225–234, Atlanta, GA, November 1993. Proceedings UIST'93.

[Hoover 92] Roger Hoover. Alphonse: Incremental computation as a programming abstraction. *Sigplan Notices*, 27(7):261–272, July 1992. ACM SIGPLAN'92 Conference on Programming Language Design and Implementation.

[Horn 92] Bruce Horn. *Properties of User Interface Systems and the Siri Programming Language*, pages 211–236. Jones and Bartlett Publishers, Boston, MA, 1992.

[HP 91] *Starbase Graphics Techniques and Display List Programmer's Guide.* Hewlett-Packard Company, Fort Collins, Colorado., 1991.

[Hudson 89] Scott E. Hudson. Graphical specification of flexible user interface displays. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 105–114, Williamsburg, VA, November 1989. Proceedings UIST'89.

[Hudson 91] Scott E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM TOPLAS*, 13(3):315–341, July 1991.

[Hudson 93] Scott E. Hudson. A system for efficient and flexible one-way constraint evaluation in c++. Technical Report 93-15, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology, April 1993.

[IRIS 92] *IRIS Inventor Programming Guide.* Silicon Graphics Computer Systems, Mountain View, CA., 1992.

[Kass 92] Michael Kass. Condor: Constraint-based dataflow. In *Computer Graphics*, volume 26, pages 321–350, July 1992.

[Koved 93] Larry Koved and Wayne L. Wooten. Groop: An object-oriented toolkit for animated 3d graphics. *ACM Sigplan Notices*, 28(10):309–325, October 1993. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'93.

[Levi 93] David Levi. Conc++: A constraint extension to the c++ programming language. Master's thesis, Computer Science Department, University of Tennessee, Knoxville, TN, 1993.

[Maloney 91] John Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1991. Also available as University of Washington Technical Report No 91-08-12.

[Myers 89a] Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating graphical interactive application objects by demonstration. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95–104, Williamsburg, VA, November 1989. Proceedings UIST'89.

[Myers 89b] Brad A. Myers, John A. Kolojejchick, and Edward Pervin. *Opal Reference Manual; The Garnet Graphical Object System*, pages 85–126. Carnegie Mellon University Computer Science Department Technical Report CMU-CS-89-196, November 1989.

[Myers 90a] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[Myers 90b] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[Myers 91a] Brad A. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *Human Factors in Computing Systems*, pages 243–249, New Orleans, LA, April 1991. Proceedings SIGCHI'91.

[Myers 92b] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, James A. Landay, Richard McDaniel, and Vivek Gupta. The garnet reference manuals: Revised for version 2.0. Technical Report CMU-CS-90-117-R2, Carnegie Mellon University Computer Science Department, May 1992.

[PHIGS 84] American National Standards Committee X3H3/84-40. *Draft Proposal American National Standard for the Functional Specification of the Programmer's Hierarchical Interactive Graphics Standard (PHIGS)*, 1984.

[PHIGS+ 88] PHIGS+ Committe, Andries van Dam, chair. *PHIGS+ Functional Description, Revision 3.0, Computer Graphics*, July 1988.

[Sannella 92a] Michael Sannella and Alan Borning. Multi-garnet: Integrating multi-way constraints with garnet. Technical report, Department of Computer Science

and Engineering, University of Washington, September 1992. Also available as University of Washington Technical Report No 92-07-01.

[SGI 91] *Graphics Library Programming Guide.* Silicon Graphics Computer Systems, Mountain View, CA., 1991.

[Shooman 83] Martin L. Shooman. *Software Engineering - Design, Reliability, and Management.* Computer Science Series. McGraw-Hill Book Company, 1983.

[Stefik 86] Mark Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. Integrating access-oriented programming into a multi-paradigm environment. *IEEE Software*, 3(1):10–18, January 1986.

[Strauss 93] Paul S. Strauss. Iris inventor, a 3d graphics toolkit. *ACM Sigplan Notices*, 28(10):192–200, October 1993. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'93.

[Szekely 88] Pedro A. Szekely and Brad A. Myers. A user interface toolkit based on graphical objects and constraints. *Sigplan Notices*, 23(11):36–45, November 1988. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'88.

[Upstill 90] Steve Upstill. *The RenderMan Companion.* Addison-Wesley, Reading, Mass., 1990.

[Vander Zanden 88] Brad T. Vander Zanden. *Incremental Constraint Satisfaction and Its Application to Graphical Interfaces.* PhD thesis, Cornell University, Ithaca, NY, 1988.

[Vander Zanden 91] Dario Giuse Brad Vander Zanden, Brad A. Myers and Pedro Szekely. The importance of pointer variables in constraint models. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 155–164, Hilton Head, SC, November 1991. Proceedings UIST'91.

[Vander Zanden 92a] Brad T. Vander Zanden. *An Active-Value-Spreadsheet Model for Interactive Languages*, pages 183–210. Jones and Bartlett Publishers, Boston, MA, 1992.

[Vander Zanden 92b] Brad Vander Zanden. A domain-independent algorithm for incrementally satisfying multi-way constraint. Technical report, Department of Computer Science, University of Tennessee, July 1992. Also available as University of Tennessee Technical Report No CS-92-160.

[Vander Zanden 94] Brad Vander Zanden, Brad A. Myers, Dario Giuse, and Pedro Szekely. Integrating pointer variables into one-way constraint models. To Appear in ACM Transactions on Computer Human Interaction, 1994.

[Zeleznik 91] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam. An object-oriented framework for the integration of interactive animation techniques. In *Computer Graphics*, volume 25, pages 105–111, July 1991.

Appendices

Appendix A

Reference Manual for the Extended Constraints

A.1 Introduction

SMW-Garnet integrates multi-output, structural, and multi-way constraints into Garnet. Previously, Garnet only supported single-output, one-way constraints. These new constraints increase efficiency, allow more powerful relationships to be expressed, and facilitate maintaining consistency between data structures. The following is a list of the entry points into the code.

destroy-constraint	output-determinant-p
m-formula	print-slot-formulas
m-formula-p	remove-activation
make-all-invalid	s-formula
make-all-valid	s-update
mo-formula	s-value-activation
mo-formula-p	update

The regular Garnet entry points like: *g-value*, *get-value*, and *s-value* unless otherwise stated, exhibit the same functionality in SMW-Garnet, and will therefore not be discussed in this manual.

The constraint solving process for multi-output, structural, and multi-way constraints may not always reach a steady state after one iteration of constraint solving. The programmer can specify the number of constraint solving iterations by changing the value of the following variable.

kr::*max-constraint-solving-iterations* [Variable]

When this limit is reached, a warning message is displayed. The default value is 10.

A.2 Multi-output Constraints

Multi-output constraints allow a constraint to return more than one value. They can result in greater efficiency by allowing the reuse of computations and reducing the number of formula objects maintained by the system. They also allow a programmer to control the granularity of the constraints by determining the amount of information that is computed.

We provide two types of multi-output constraints. The first type outputs to a fixed number of variables and the second type outputs to a dynamic number of variables. They are discussed below. The following accessory function is helpful when working with multi-output constraints.

(**mo-formula-p** *thing*) [Function]

mo-formula-p returns true if *thing* is a multi-output formula, otherwise it returns nil.

A.2.1 Fixed number of outputs

Multi-output constraints with a fixed number of outputs are created using the following macro.

(**mo-formula** *formula*) [Macro]

formula determines the values of the output variables. The value returned by *formula* must be a list. The values in the list are assigned sequentially to a list of slots that are determined by the formula. As with o-formula, the list of slots are provided as either part of a call to *create-instance* or *s-value*. A slot-list is a Lisp list of slot-names. For example, (:left :top) is a valid slot-list.

Below is an example of a multi-output formula with a fixed number of outputs that is being specified during object creation. This constraint specifies that the *:left*, *:width*, *:top*, and *:height* of *moving-rectangle* should obtain their value from the *:box* slot. This type of a relationship is commonly found in interactor objects.

```
(create-instance 'moving-rectangle opal:rectangle
   (:box '(80 20 100 150))
   ((:left :width :top :height) (mo-formula (list
      (first (gvl :box)) (second (gvl :box))
      (third (gvl :box)) (fourth (gvl :box))))))
```

Below is the same multi-output formula, except this time it is being constructed using *s-value*.

```
(s-value moving-rectangle (:left :width :top :height) (mo-formula (list
```

```
(first (gvl :box)) (second (gvl :box))
(third (gvl :box)) (fourth (gvl :box)))))
```

This example, illustrates how the regular Garnet entry point *s-value* has been slightly modified to handle a fixed list of output slots. Unlike other list assignments using *s-value*, an apostrophe is not needed before the list of slots.

A.2.2 Dynamic number of outputs

SMW-Garnet allows a constraint to have a dynamic number of outputs. To support this, the programmer can attach an expression to a constraint that computes the set of outputs. The expression is itself a constraint that can depend on other variables in the system. Multi-output constraints with a dynamic number of outputs are created using the following macro.

(**mo-formula** *objects slots formula*) [Macro]

objects is an expression that is converted to a constraint to determine the set of output objects. It is also referred to as an output **determinant constraint**. *slots* contains a list of the output variables. *formula* determines the values of the output variables. The value returned by *formula* must be a list. The following accessory functions allows the programmer to determine if an object is an output determinant constraint.

(**output-determinant-p** *thing*) [Function]

output-determinant-p returns true if *thing* is an output determinant constraint, otherwise it returns nil.

Below is an example of a multi-output formula with a dynamic number of outputs that is being specified during object creation. This constraint modifies the left and top slots of all objects in the aggregate *my-agg*. The constraint allows the aggregate to determine the position of its children. Hence, moving the aggregate causes its children to move by an equivalent amount. The slots *:x-offset* and *:y-offset* are the relative left and top positions of the children.

```
(create-instance 'my-agg opal:aggregate
  (:dummy-slot (mo-formula
    (gv my-agg :components)                          ;; objects
    (:left :top)                                     ;; slots
    (let ((agg-left (gv my-agg :left))               ;; formula
          (agg-top (gv my-agg :top)) return-list)

      (dolist (object (gv my-agg :components))
        (setf return-list
```

```
      (push (+ (gv object :x-offset) agg-left) return-list))
    (setf return-list
      (push (+ (gv object :y-offset) agg-top) return-list)) )
  (reverse return-list)))))
```

The multi-output constraint given above could also have been created by using *s-value*. To better understand how output determinant constraints are evaluated, the invalidate algorithm for multi-output constraints is given below.

1. Invalidation is performed, with the exception that output determinant constraints are queued. Formulas downstream from these constraints are currently not invalidated.
2. Evaluate the queued output determinant constraints.
3. If the evaluation of one of the output determinant constraints in the previous step results in a different set of output objects (i.e., either objects have been added or removed) then all formulas downstream from the constraint are invalidated. As constraints are invalidated, they may cause other output determinant constraints to become invalidated. These newly invalidated output determinant constraints are added to a new list.
4. If no output determinant constraints are invalidated in the previous step, or a user specified limit has been reached on the number of constraint solving iterations then stop; otherwise, repeat step 2 using the new list of output determinant constraints. The user specified limit allows the output determinant constraints to iterate through several constraint solving cycles, so that the constraint network can reach a more satisfied state.

A.3 Structural Constraints

Structural constraints allow the programmer to define relationships between the structure of objects. These constraints make it possible for a constraint to add, delete, or modify a structure. For example, a structural constraint can express an existence relationship, such that the existence of an arrow depends on the existence of the two boxes to which it is attached. If either box is deleted, the structural constraint will delete the arrow. As another example, a structural constraint can be used to ensure that there is always one text object allocated for each file name that is currently displayed in a file browser application. When the user changes directories, thus accessing a new set of filenames, the structural constraint will automatically allocate text objects for the new filenames and deallocate text objects for the old filenames.

A structural constraint is a relationship that is expressed on structural attributes. A structural attribute is a variable that conveys information about the structure of an object. For

example, structural attributes might include 1) the type of an object (e.g., rectangle, circle); 2) the parts of an object (e.g., a labeled text box might include a text object and a rectangle object); and 3) the existence of an object (e.g., an arrow should exist only if the two boxes that it connects exist).

The default structural attributes are **:components**, **:items**, **:is-a**, and **:existence**. To declare a variable as a structural slot locally to an object, the programmer can append this variable to the list contained in the slot *:structural-slots*. To globally declare a slot as a structural variable, the programmer can append the slot to the list stored in the following variable, which contains the default structural slots mentioned above.

kr::*default-structural-slots* [Variable]

SMW-Garnet uses an active-value model to enforce the structural constraints. An active value model allows action-procedures to be attached to variables. When the value of a variable changes, the action-procedure associated with it is automatically executed. Action-procedures can be associated with any structural slot. Changing the value of a structural variable triggers the action-procedures, which enforces the structural changes that the new value requires. By allowing constraints to be attached to these structural variables, we obtain structural constraints.

When structural constraints are evaluated, the action-procedures are not evaluated immediately, but instead are queued and evaluated once all structure slots have been brought up-to-date. This prevents an object from being deleted before another object that depends on it is able to evaluate its structure slot. All structural constraints are evaluated before any attribute constraints. The structure determines the attributes, so it is necessary that the structure be completely determined before any attribute calculations are performed.

An action-procedure is specified by placing its name in the *:action-procedure* slot associated with an object. All action procedures use the following format.

(**defun** *action-procedure-name (schema slot) ...)* [Function]

Actions for different slots can be separated using either a case or a conditional statement. Below is an example of a structural constraint and its corresponding action-procedure. The constraint specifies that the existence of moon1 should depend on the existence of the object planet. When the existence variable of moon1 is either directly or indirectly modified, the action-procedure *EXISTENCE* is queued and later executed.

```
(create-instance 'moon1 moon-template
  (:existence (o-formula (gv planet :existence)))
  (:action-procedure 'EXISTENCE))

(defun EXISTENCE (schema slot)
  (case slot
    (:existence
```

```
(if (null (g-value planet :existence))
  (opal:destroy schema)))))
```

The algorithm for implementing structural constraints is given below. The user first modifies the slots of one or more objects. If a modification either directly or indirectly modifies a structural attribute, then that variable is added to a list of structurally invalid attributes. If a modification either directly or indirectly modifies an value attribute, then that variable is added to a second list containing value attributes that are invalid. At some point, the user will specifically request that the constraints be satisfied by calling the following function.

(**opal:s-update** window) [Function]

This function performs the following steps.

1. The structural slots for all objects on the structurally invalid list are brought up-to-date, queuing all action-procedures that are associated with structural attributes whose value has changed.
2. The queued action-procedures are executed. As these procedures are executed, they may invalidate other structural or value attributes. When structural attributes are invalidated by the action-procedures, then a new structurally invalid object list is created, and the invalidated object is added to this list.
3. If no structural slots are invalidated in the previous step, or a user specified limit has been reached on the number of iterations of constraint solving, then stop; otherwise, repeat steps 1 and 2 using the new list of structurally invalid objects. The user specified limit allows the structural constraints and action-procedures to iterate through several constraint solving cycles, so that the constraint network can reach a more satisfied state.

When using SMW-Garnet, the Garnet formula *update* (listed below) first calls the function *s-update* which brings all structural attributes up-to-date. Finally, *update* brings all value attributes up-to-date, and then it updates the image in *window*.

(**opal:update** *window*) [Method]

A.4 Multi-way Constraints

Multi-way constraints allow any of the objects in the constraint to be changed in order for it to be satisfied. SMW-Garnet implements multi-way constraints by allowing multiple one-way constraints (called formulas) to be attached to a single slot. The programmer is allowed to attach optional priorities to these formulas. The priority indicates how strongly the user wants a given formula to determine a slot's value. The formula with the highest priority

in a slot determines the slot's value. If two or more formulas with the same priority are invalid and must be re-evaluated, then the formula that occurs first in the slot determines the slot's value. The programmer can also add activation conditions to a formula. An activation condition is a boolean predicate that determines whether a formula is active or inactive. The boolean predicate is an arbitrary piece of Lisp code that returns either nil or a non-nil value. The code may consist of both structural and value attributes.

Priorities can be used to store default formulas in slots that are masked by user-defined formulas, but become active when these user-defined formulas are removed (e.g., bounding box formulas for aggregates). Activation conditions can be used to control constraint satisfaction. For example, the programmer might define the multi-way constraint right = left + width, which is represented as the three one-way constraints "right = left + width", "left = right - width", and "width = right - left". Depending on the context, the programmer might want the constraint solver to satisfy one of these three formulas in preference to one of the others. For example, if right is changed, the constraint "left = right - width" should be evaluated if the user wants the object to move, and the constraint "width = right - left" should be evaluated in the user wants the object to resize. Activation conditions can be used to control which constraints are evaluated by selectively activating or deactivating constraints. Thus the programmer might attach the activation condition "grow-p = false" to the constraints associated with left and right, and the activation condition "grow-p = true" to the constraint associated with width.

Multi-way constraints are created using the following macro. The + sign indicates that one or more formulas must be specified.

(**m-formula** (*formula* &key *formula-cache name act-cond priority*)$^+$
&key *slot-cache*) [Macro]

formula specifies the formula. *formula-cache* allows the programmer to assign an initial value to the formula. *slot-cache* allows the programmer to assign an initial value to the slot. *name* allows the programmer to assign a name to the formula. This *name* can be used with other functions described later in this manual that operate on formulas. *act-cond* allows the programmer to specify an optional activation condition for *formula*. The activation condition may be specifies as either a constant value or a formula. *priority* indicates how strongly the programmer wants this formula to determine the value of the slot. The default priority of a formula is **:weakest**. Listed from strongest to weakest, the strengths are specified by using one of the following keywords **:required**, **:strong**, **:medium**, **:weak**, and **:weakest**. The following variable contains a list of the priorities in decreasing strength.

kr::*strength-list* [Variable]

Below is an example multi-way constraint which expresses the relationship that all three moons should have the same size. Since the diameter of a circle in Garnet is equal to the minimum of the width and height, multi-way constraints are also placed between the width and height slots of each moon.

```
(create-instance 'moon1 opal:circle
  (:width (m-formula ( ((gv moon1 :height) :formula-cache 50))))
  (:height (m-formula ( ((gv moon1 :width)  :formula-cache 50)
                        ((gv moon2 :height) :formula-cache 50)) )))

(create-instance 'moon2 opal:circle
  (:width (m-formula (
    ((gv moon2 :height) :formula-cache 50) )))
  (:height (m-formula ( ((gv moon2 :width) :formula-cache 50)
                        ((gv moon1 :height) :formula-cache 50)
                        ((gv moon3 :height) :formula-cache 50) ))) )

(create-instance 'moon3 opal:circle
  (:width (m-formula ( ((gv moon3 :height) :formula-cache 50) )))
  (:height (m-formula ( ((gv moon3 :width) :formula-cache 50)
                        ((gv moon2 :height) :formula-cache 50) ))) )
```

The following function allows the programmer to add formulas to a slot.

(**s-formula** *schema slot formula*
&**key** *formula-cache position name priority act-cond override*) [Function]

schema and *slot* specify the location of the multi-way constraint. *formula* specifies a formula to be added to the multi-way constraint. *formula-cache* allows the programmer to assign an initial value to the formula. *position* specifies the position of the formula within the multi-way constraint. The default value is zero (i.e., at the beginning of the list). *name* allows the user to assign the formula a name. *priority* indicates how strongly the programmer wants this formula to determine the value of the slot. *act-cond* allows the programmer to specify an optional activation condition for *formula*. *override* allows the programmer to destroy any old formulas or pointers to multi-output formulas that were previously stored with this slot. If *override* is true, the formula replaces all current formulas, otherwise the formula is added to the current set of formulas. The default value for *override* is nil.

Methods can be removed by calling the following function.

(**destroy-constraint** *schema slot* &**optional** *name-or-position*) [Function]

schema and *slot* specify the location of the multi-way constraint. *name-or-position* specifies which formula will be removed. The default value is nil, which means to destroy all constraints in the slot. The programmer can either provide the name of the formula, or the position of the formula within the slot. The first formula is at position 0.

Activation conditions are created and their values can be changed by calling the following function. If the formula currently does not contain an activation condition, then a new formula is created; otherwise, the old activation condition is replaced.

(**s-value-activation** *schema slot name-or-position value-or-formula*) [Function]

schema and *slot* specify the location of the multi-way constraint. *name-or-position* indicates on which formula the activation condition will be created or its value changed. The programmer can either provide the name of the formula, or the position of the formula within the slot. *value-or-formula* is the new value or formula that is being assigned to the activation condition.

Below is an example of an activation condition. If this activation condition is used in conjunction with the three multi-way constraints given either, then moon3 will only be constrained to moon1 and moon2 when the left value of moon1 is greater than 100.

```
(s-value-activation moon3 :height 2
  (o-formula (> (gv moon1 :left) 100)))
```

Activation conditions can be removed by calling the following function.

(**remove-activation** *schema slot name-or-position*) [Function]

schema and *slot* specify the location of the multi-way constraint. *name-or-position* specifies which formula's activation condition will be removed. The programmer can either provide the name of the formula, or the position of the formula within the slot.

Below is a list of some functions that facilitate creating and debugging multi-way constraints.

(**m-formula-p** *thing*) [Function]

(**make-all-valid** *schema slot*) [Function]

(**make-all-invalid** *schema slot*) [Function]

(**print-slot-formulas** *schema slot*) [Function]

m-formula-p returns true if *thing* is a multi-way constraint, otherwise nil is returned. *make-all-valid* sets the valid bit for all formulas in slot to true. *make-all-invalid* sets the valid bit for all formulas in slot to nil. *print-slot-formulas* displays all the formulas associated with a slot along with their valid bit. The programmer could use this function to determine which formulas were invalidated as the result of changing one of the variables in a multi-way constraint.

The algorithm for multi-way constraints operates as follows: 1) if the value of a variable is modified, any activation conditions which depend either directly or indirectly on this variable are immediately evaluated (i.e., activation conditions are eagerly evaluated); 2) if a formula depends on a changed variable and the formula does not determine the value of the slot that it is associated with, then the formula will be invalidated but the slot will not be invalidated. A formula may not determine the value of a slot either because it is not the highest priority formula in the slot, or because the formula is inactive; 3) if a formula that

previously did not determine the value of a slot becomes capable of determining the value of the slot, then if the formula is invalid, the slot, and any formulas that depend on that slot, will be invalidated. A formula becomes capable of determining the value of a slot if its status changes from inactive to active or it becomes the highest priority formula in the slot.

Appendix B

3D Reference Manual: Programming Layer

B.1 Introduction

The programming layer of the 3D toolkit supports an extensive set of 3D objects, object composition (aggregates), constraints, and a high-level behavioral model (interactors). It also includes functions for projecting objects, clipping, shading, and applying transformations. Programmers using this level must write their own code. The 3D toolkit has been integrated into the Garnet user interface development environment. Previously, Garnet only supported 2-dimensional graphics.

B.2 Graphic Qualities

The three graphic qualities that can be specified for 3D objects are *:filling-style*, *:line-style*, and *:wire-frame-p*. Like other Garnet objects, the programmer can use the predefined Garnet line-styles and filling-styles. The programmer can also interactively create a new color by calling the function *3dim:create-new-color* as discussed in section B.4.

B.2.1 Filling Style

The following filling style format is used for all 3D objects.

`:filling-style` *default-value* {`:keyword color`}*

The filling style keywords vary from one object to the next. See section B.3 for color keywords associated with each object. To render an object using wire-frames, the user can either set the slot *:filling-style* to nil or set *:wire-frame-p* to true. The slot *:wire-frame-p*

allows the user to more easily change the rendering mode without having to destroy the contents of the filling-style slot. The filling-style and line-style for polyhedrons is specified in the *:sides* slot.

B.2.2 Line Style

Garnet draws an approximate image of 3D objects using polygons. The *:line-style* slot determines the edge color of these polygons. Regardless of whether the rendering mode is wire-frame or shaded, the edges are visible unless *:line-style* is set to nil.

B.2.3 Wire Frame versus Shading

The 3D toolkit provides two rendering modes of display. They are wire-frame and shading. In the wire-frame mode only the edges of objects are drawn. While this mode can be slightly faster, its results are less realistic than the shading mode, especially when modeling solid objects. The order of display is unimportant unless the variable *3dim:sort-wire-framed-objects* is changed to true. When *3dim:sort-wire-framed-objects* is true, both rendering modes call the shading algorithm. When this variable is false, wire-framed objects are drawn first with no ordering. The user may wish to set this variable to true when displaying objects with different rendering modes. Otherwise, shaded objects will always cover up wire-framed objects.

The problem with the wire-frame mode is that ambiguous results are often obtained. In the shading mode all objects are drawn as filled polygons. The rendering process for shading requires slightly more processing because all objects must be spatially ordered. Parts of objects that are hidden, because they are obscured by "closer" objects must not be displayed.

To achieve acceptable performance in a real time environment, the 3D shading algorithm only handles objects that consist of polygons that conform to the following restriction. For any two polygons in the view volume, one polygon is wholly in one half-space of the other. Objects like the ones in Figure B.1 (a) that penetrate each other, or Figure B.1 (b) that cyclically overlap will not be shaded correctly, for there is no order in which these polygons can be drawn to produce the desired image. If objects like those shown in Figures B.1 (a) and (b) are split into smaller polygons, such that a linear order is possible, then they can be correctly handled by the shading algorithm.

B.3 Predefined 3D Graphical Objects

This section describes a number of specific subclasses of the opal:3d-graphical-object prototype that implement all of the graphic primitives that can be displayed. The list of 3D

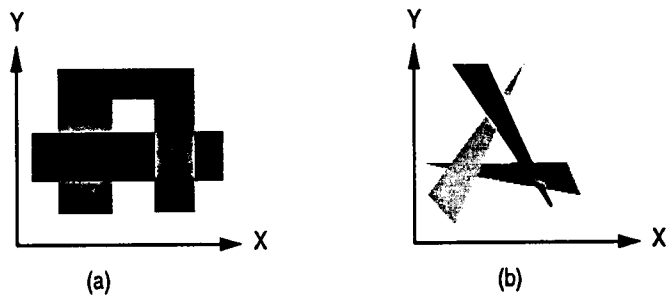


Figure B.1: Polygons that are not guaranteed to be shaded correctly.

objects that are currently implemented include: cube, 3d-line, plane, cone, cylinder, sphere, ellipsoid, and polyhedron. The following slots are common to all of the 3D graphical objects.

```
(:x 0)
(:y 0)
(:z 0)
(:rotate-x 0)
(:rotate-y 0)
(:rotate-z 0)
(:scale-x 1)
(:scale-y 1)
(:scale-z 1)
(:center-x 0)
(:center-y 0)
(:center-z 0)
(:wire-frame-p t)
(:line-style opal:default-line-style)
(:filling-style opal:no-fill)
```

The slots *:x*, *:y*, and *:z* determine the location of an object. Rotation of an object is specified using the slots *:rotate-x*, *:rotate-y*, and *:rotate-z*. Resizing or stretching an object can be accomplished by changing the values in the slots *:scale-x*, *:scale-y*, *:scale-z*. As an example, if the scale slots for an objects are set to (*:scale-x = 2, :scale-y = 1, and :scale-z = .5*) then the object is redrawn twice as large in the x direction, no change in the y direction, and reduced by 50 percent in the z direction. The slots *:center-x*, *:center-y*, and *:center-z* specify an objects center and are automatically calculated. Unless specified, objects are rotated about their calculated center. See section B.6.2 for details on rotating an object about a point other than its center.

The 3D extension of Garnet uses a right handed coordinate system (rhs). In a right handed system the z axis is pointing out towards the viewer, whereas, in a left handed system the z axis is pointing away from the viewer. In both systems the x and y axes are aligned to

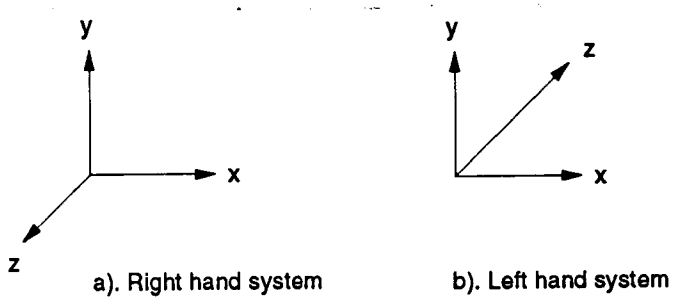


Figure B.2: The two types of coordinate systems.

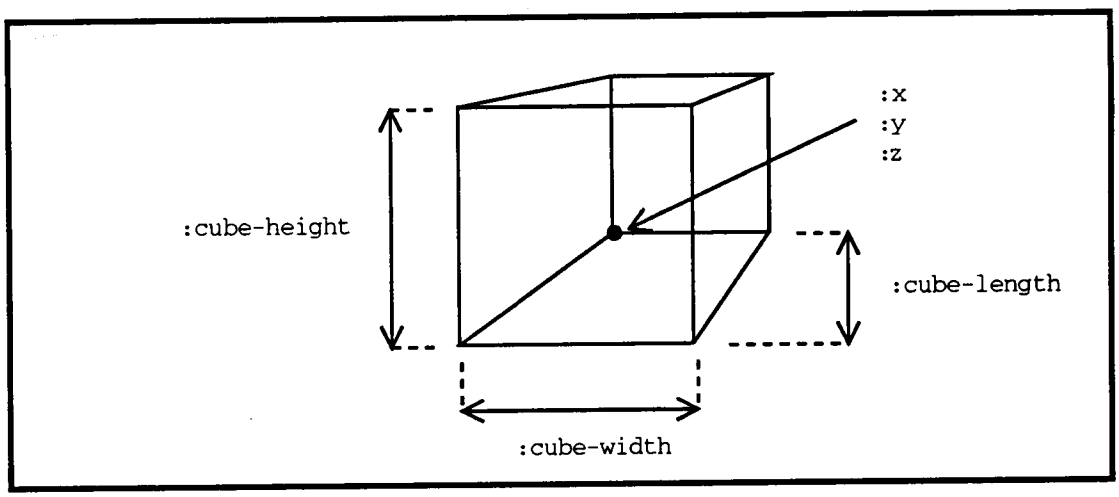


Figure B.3: A description of a cube.

the screen. Figure B.2 shows the difference between the two coordinate systems.

B.3.1 Cube

```
(create-instance 'OPAL:CUBE  opal:3d-graphical-object
  (:cube-height 50)
  (:cube-width 50)
  (:cube-length 50))
```

Figure B.3 shows the slots of a cube. The slots *:x*, *:y*, and *:z* specify the back lower left corner of the cube. The world coordinates are then computed from this starting point. For example, the front upper right corner is computed as: (x + cube-width, y + cube-height, z + cube-length).

The following filling-style format is used for cubes:

```
:filling-style default-value &key  :front-color :back-color :left-color
                                   :right-color :top-color :bottom-color
```

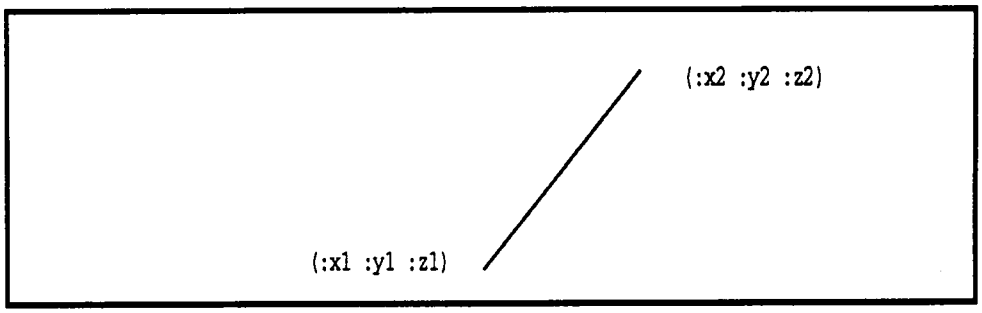


Figure B.4: A description of a 3d-line.

For example, the following filling style causes all sides of the cube to be drawn red.

```
(:filling-style opal:red-fill)
```

In contrast, the following filling-style results in all components being shaded red except the front, bottom, and back sides. In this example the front and back are shaded yellow, and the bottom is shaded blue.

```
(:filling-style `(,opal:red-fill
        :front-color   ,opal:yellow-fill
        :bottom-color  ,opal:blue-fill
        :back-color    ,opal:yellow-fill))  )
```

B.3.2 3D-Lines

```
(create-instance 'OPAL:3D-LINE  opal:3d-graphical-object
  (:x1 0)
  (:y1 0)
  (:z1 0)
  (:x2 40)
  (:y2 40)
  (:z2 0))
```

Figure B.4 shows the slots of a 3d-line. Unlike the other objects, the slots *:x*, *:y*, and *:z* are not used to specify the starting location. Instead, the slots (*:x1 :y1 :z1*) and (*:x2 :y2 :z2*) give positional information. The *:filling-style* slot is ignored. To change the color of a 3d-line set the *:line-style* slot.

B.3.3 Planes

```
(create-instance 'OPAL:PLANE  opal:3d-graphical-object
```

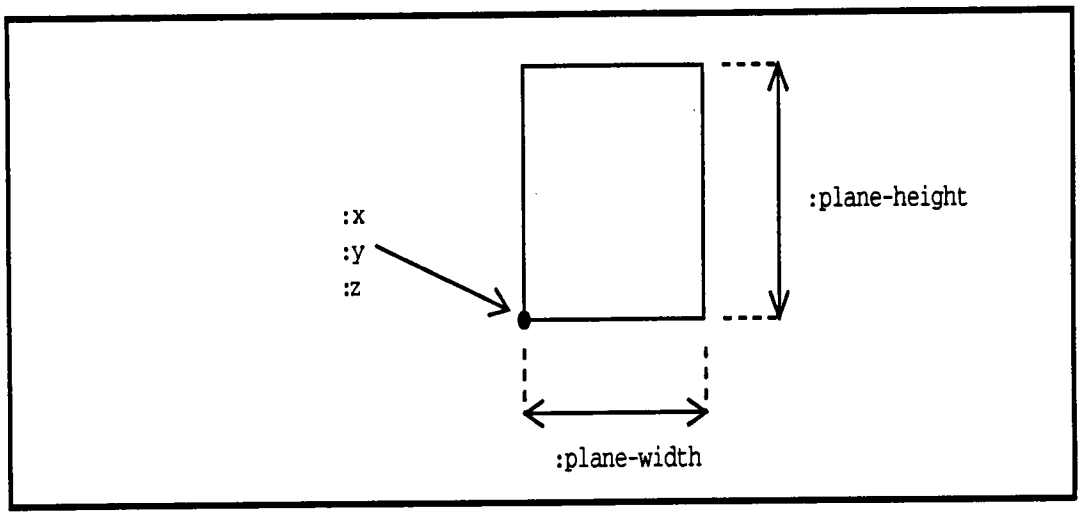


Figure B.5: The description of a plane.

```
  (:plane-width 50)
  (:plane-height 50))
```

Figure B.5 shows the slots of a plane. There are no special color keywords for the filling-style.

B.3.4 Cones

```
(create-instance 'OPAL:CONE  opal:3d-graphical-object
  (:radius 15)
  (:cone-height 40)
  (:sections 10)
  (:fast-cone-p nil))
```

To produce a bottom which is circular the cone is scaled in both the x and z directions by the same amount. This amount is the minimum of *:scale-x* and *:scale-z*. The two different types of cones are discussed next.

B.3.4.1 Regular Cones

Figure B.6 (a) shows the slots of a regular cone. Garnet draws an approximate image of a cone using triangular polygons. The *:sections* slot determines the number of polygons used. The larger the value the more rounded the cone appears. Acceptable values for the *:sections* slot are between 6 and 50. Other values are automatically rounded to be within this interval. A regular cone uses the following filling-style format:

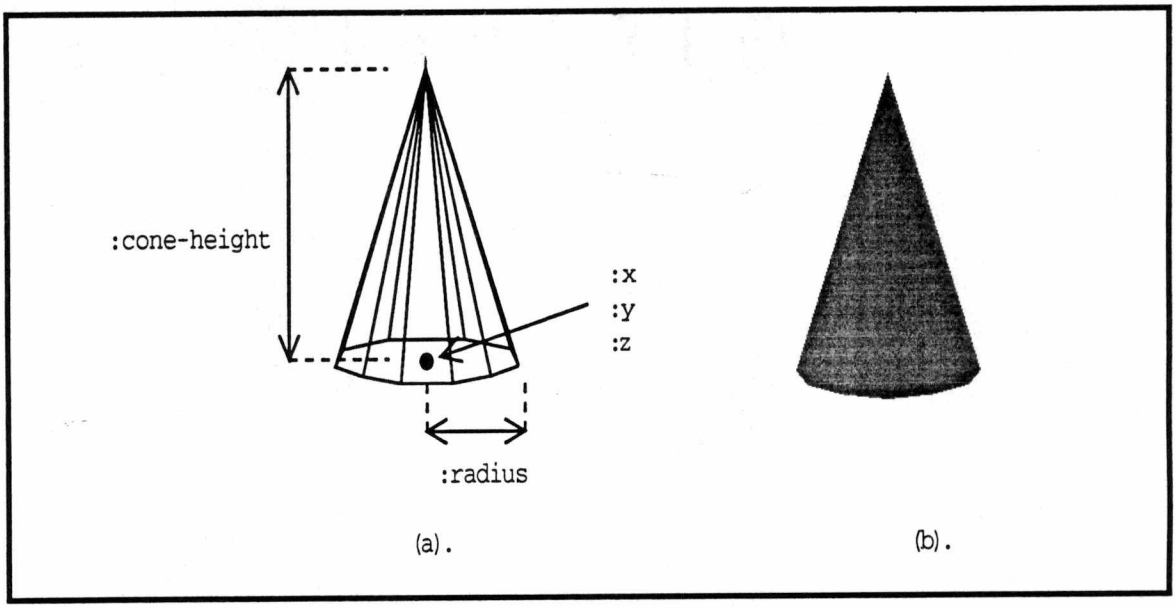


Figure B.6: The two types of cones.

`:filling-style` *default-value* *&key* `:side-color :bottom-color`

B.3.4.2 Fast Cone

If the user desires a cone that can be drawn faster, they can either reduce the number of sections in the regular cone or use a fast-cone. A fast cone is created by setting the slot *:fast-cone-p* to true. It looks like a regular cone, but the fast cone is drawn using only two objects (i.e. a polygon and an arc). To illustrate the difference, a regular cone with 20 sections draws 21 objects (20 sides + 1 bottom = 21). However, the fast cone draws just 2 objects. The fast cone can be translated, scaled, and rotated just like the regular cones. Figure B.6 (b) shows a fast cone.

To produce a realistic image, the fast cone is always displayed using the shaded mode. A wire frame rendering of a fast cone would not necessarily resemble a cone. The default filling-style is always used and the color keywords used for the regular cone are ignored. The line-style is set to nil, and cannot be changed.

B.3.5 Cylinders

```
(create-instance 'OPAL:CYLINDER  opal:3d-graphical-object
  (:radius 15)
  (:cylinder-height 40)
  (:fast-cylinder-p nil)
  (:sections 12))
```

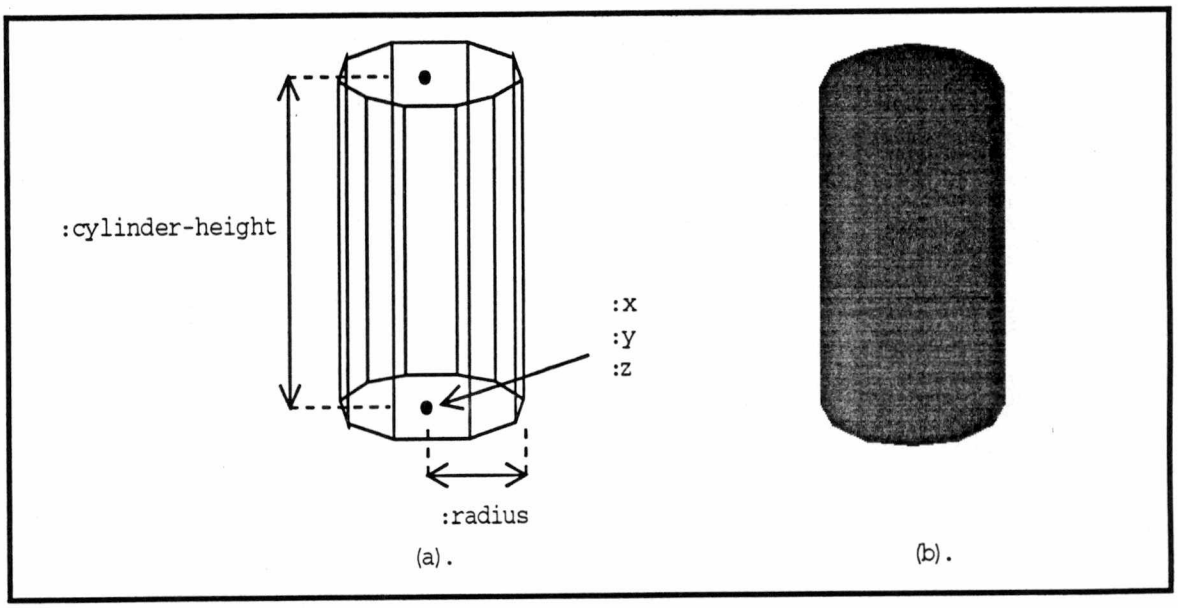


Figure B.7: The two type of cylinders.

To produce a cylinder which is circular, the cylinder is scaled in both the x and z directions by the same amount. This amount is the minimum of *:scale-x* and *:scale-z*. The two different types of cylinders are discussed next.

B.3.5.1 Regular Cylinder

Figure B.7 (a) shows the slots of a regular cylinder. Garnet draws an approximate image of a cylinder using rectangular polygons. The *:sections* slot determines the number of polygons used. The larger the value the more rounded the cylinder appears. Acceptable values for the *:sections* slot are between 6 and 50. Other values are automatically rounded to be within this interval. The following filling-style format is used for regular cylinders:

`:filling-style` *default-value* `&key :top-color :side-color :bottom-color`

B.3.5.2 Fast Cylinder

If the user desires a cylinder that can be drawn faster, they can either reduce the number of sections in the regular cylinder or use a fast-cylinder. A fast cylinder is created by setting the slot *:fast-cylinder-p* to true. It looks like a regular cylinder, but the fast cylinder is drawn using only three objects (i.e. a polygon and two arcs). To illustrate the difference, a regular cylinder with 20 sections draws 22 objects (20 sides + 1 bottom + 1 top = 22). However, the fast cylinder draws just 3 objects. The fast cylinder can be translated, scaled, and rotated just like the regular cylinders. Figure B.7 (b) shows a fast cylinder.

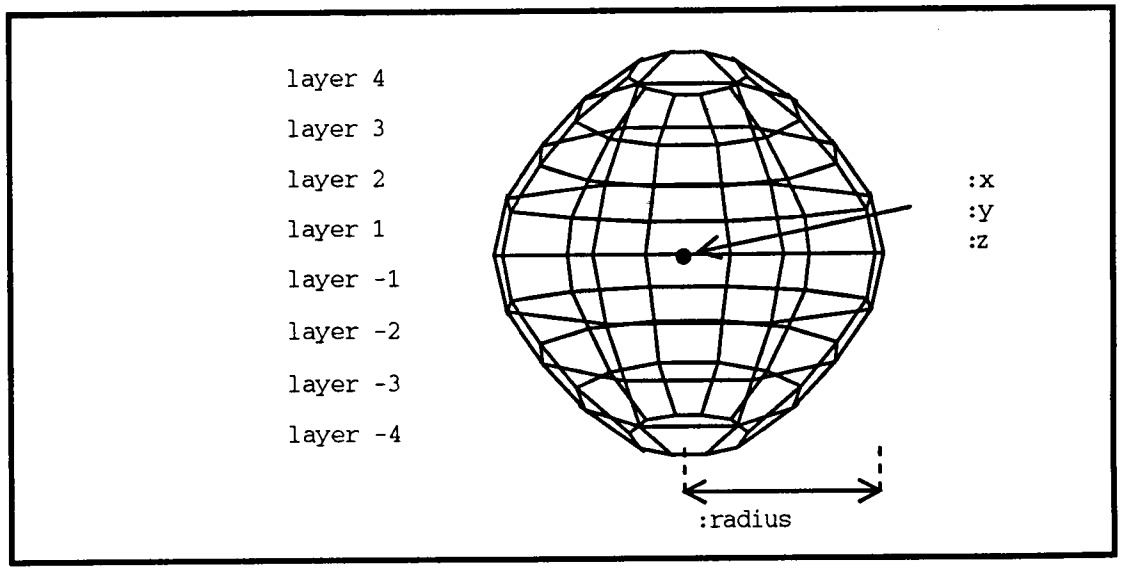


Figure B.8: The description of the regular sphere

To produce a realistic image, the fast cylinder is always displayed using the shaded mode. A wire frame rendering of a fast cylinder would not necessarily resemble a cylinder. The default filling-style is always used and the color keywords used for the regular cylinder are ignored. The line-style is set to nil, and cannot be changed.

B.3.6 Sphere

```
(create-instance 'OPAL:SPHERE  opal:3d-graphical-object
  (:radius 40)
  (:sections 8)
  (:layers 8)
  (:fast-sphere-p nil))
```

Figure B.8 shows the slots of a sphere. To produce a circular sphere, the sphere is scaled in the x, y, and z directions by the same amount. This amount is the minimum of *:scale-x*, *:scale-y* and *:scale-z*. The two different types of cylinders are discussed next.

B.3.6.1 Regular Sphere

Garnet draws an approximate image of a sphere using polygons. The *:sections* and *:layers* slots determine the number of polygons used. The larger these values, the more rounded the sphere appears. Acceptable values for both slots are between 6 and 50. Other values are automatically rounded to be within this interval. The *:layers* slot is always rounded up to an even integer. The following filling-style format is used for regular spheres:

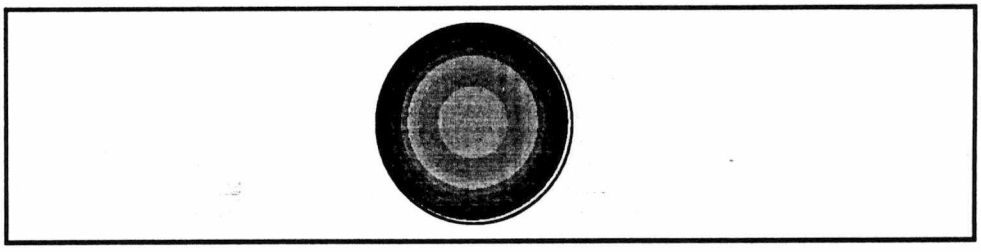


Figure B.9: An example of a fast sphere.

```
:filling-style default-value &key  :top-color :bottom-color
                                   :layer ((n filling-style)
                                           .  .  .
                                           (2 filling-style)
                                           (1 filling-style)
                                           (-1 filling-style)
                                           (2 filling-style)
                                           .  .  .
                                           (-n filling-style))
```

Below is an example of how the user would change the filling-style for *my-sphere*. In this example, the top is shaded yellow, the bottom is shaded blue, layer 2 is shaded yellow, and layer -1 is shaded orange. All other layers are shaded red.

```
(s-value my-sphere :filling-style '(,opal:red-fill
  :top-color    ,opal:yellow-fill
  :bottom-color ,opal:blue-fill
  :layer ((2  ,opal:yellow-fill)
          (-1  ,opal:orange-fill))))
```

B.3.6.2 Fast Sphere

If the user desires a fast sphere they can either reduce the number of sections and layers in a regular sphere or use a fast-sphere. A fast sphere is created by setting the slot *:fast-sphere-p* to true. It is drawn by overlapping circles of a smaller radius and of a lighter color. The rotate slots are ignored. The default filling-style is always used and the color keyword used for the regular sphere are ignored. Figure B.9 illustrates a fast sphere. The object usually looks much better on a screen than on paper.

B.3.7 Ellipsoid

```
(create-instance 'OPAL:ELLIPSOID sphere
```

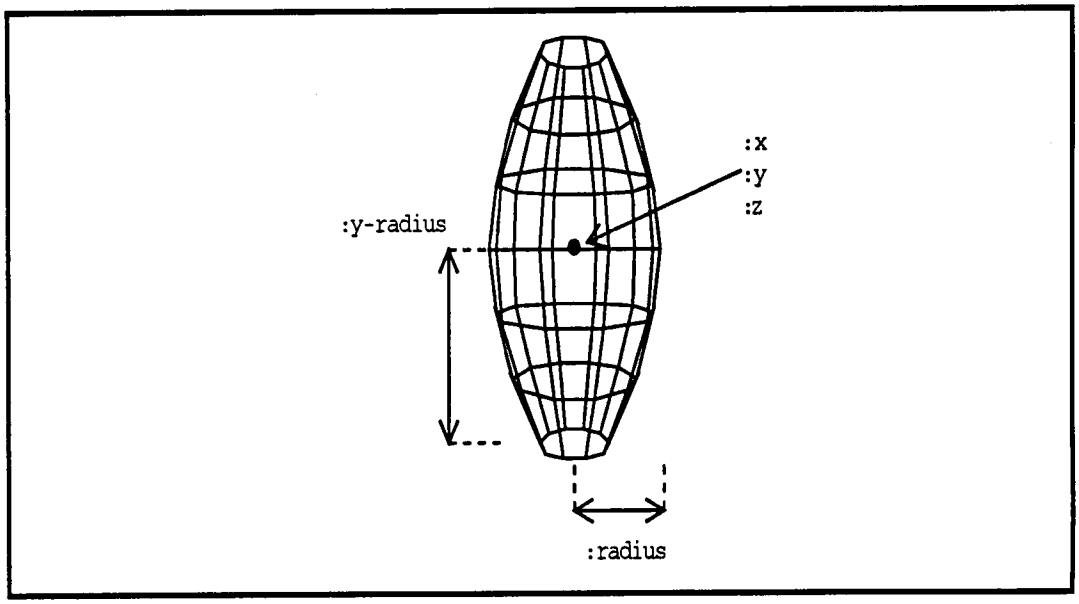


Figure B.10: The description of an ellipsoid.

```
  (:radius 25)
  (:y-radius 50))
```

Figure B.10 shows an example of a ellipsoid. To produce an ellipsoid that is circular, the ellipsoid is scaled in both the x and z directions by the same amount. This amount is the minimum of *:scale-x* and *:scale-z*.

B.3.8 Polyhedron

```
(create-instance 'OPAL:POLYHEDRON  opal:3d-graphical-object
  (:points '(
     (:point-1  0 0 0)
     (:point-2  50 0 0)
     (:point-3  50 50 0)
     (:point-4  0 50 0)))
  (:sides '(
    (:side-1 (1 2 3 4) ,opal:orange-fill)))
```

Figure B.11 shows an example of a polyhedron. A polyhedron is specified by a *:points* slot and a *:sides* slot. Shown below is a grammar describing the acceptable values for these two slots.

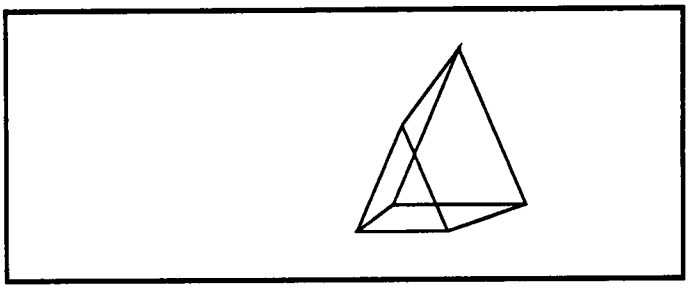


Figure B.11: An example of a polyhedron.

:points	=	(point*)
point	=	([:label] x y z)
:sides	=	(side*)
side	=	([:label] point-list [filling-style] [line-style])
point-list	=	(point-indicator*)
point-indicator	=	integer | :label

The parenthesis indicate a list. A slightly more complicated object such as a prism could be created as follows. The prism can be specified by 6 points and 5 sides. As shown below, labeling points and sides can result in easier maintainability and readability.

```
(create-instance 'prism opal:polyhedron
  (:rotate-x -10)
  (:rotate-y 30)
  (:points '(
     (:back-left 0 0 0)
     (:back-right 30 0 0)
     (:back-top 15 25 0)
     (:front-left 0 0 30)
     (:front-right 30 0 30)
     (:front-top 15 25 30)))
  (:sides '(
     (:back (1 3 2 1) ,opal:yellow-fill)
     (:front (4 5 6 4) ,opal:yellow-fill)
     (:left (5 2 3 6 5) ,opal:green-fill)
     (:right (1 4 6 3 1) ,opal:green-fill)
     (:bottom (5 2 1 4 5) ,opal:blue-fill))) )
```

The *:sides* slot above could have also been specified as shown below.

```
  (:sides '(
    (:back (:back-left :back-top :back-right :back-left)
```

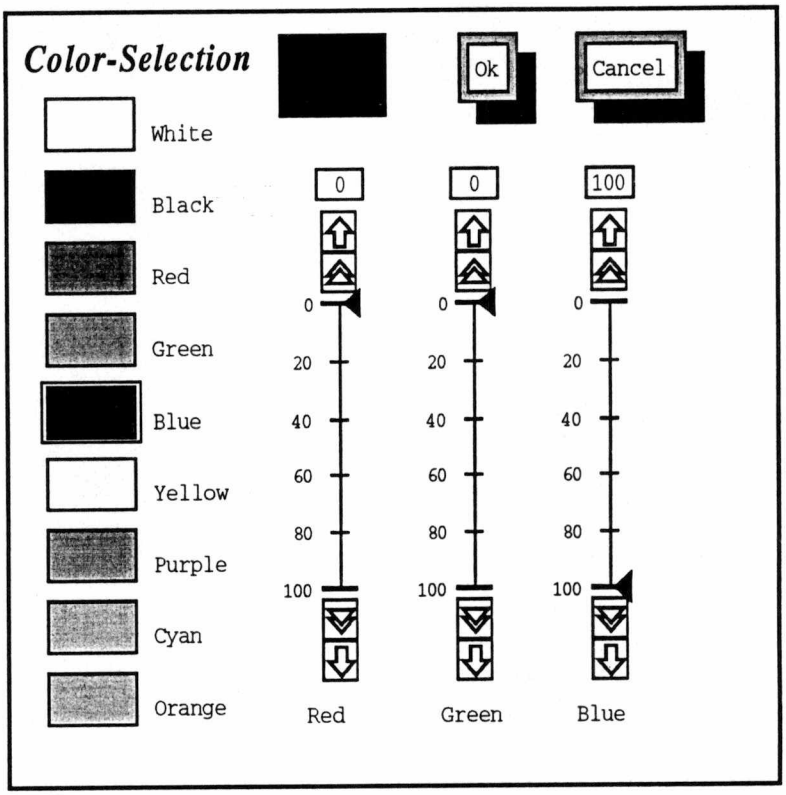


Figure B.12: The window used for creating new colors. Colors can be selected either by choosing one of the colors in the menu on the left or by manipulating the three sliders on the right. The color box at the top provides feedback showing the selected object.

```
   ,opal:yellow-fill)
(:front (:front-left :front-right :front-top :front-left)
  ,opal:yellow-fill)
(:left (:front-right :back-right :back-top :front-top :front-right)
  ,opal:green-fill)
(:right (:back-left :front-left :front-top :back-top :back-left)
  ,opal:green-fill)
(:bottom (:front-right :back-right :back-left :front-left :front-right)
  ,opal:blue-fill)))
```

B.4 Interactively Creating New Colors

A new color type can be interactively created by calling the function *3dim:create-new-color*. This function creates a window with three sliders as shown in Figure B.12. The three sliders represent the colors red, green, and blue. A slider can be dragged between 0 and 100 to produce different colors. On the left side of this window are some pre-defined colors.

Clicking on these rectangles sets the sliders according to the color of the rectangle selected. The final result of each slider is divided by 100 to produce a normalize number between 0 and 1. After the new color has been specified, press the button "Ok" and the new color is created and returned. Otherwise, press "Cancel" and nil is returned. The new-color returned from the function *3dim:create-new-color* can be directly assigned to a variable as shown below.

```
(setf new-color (3dim:create-new-color))
```

After the color is created, the user will probably want to either create a new filling style or a new line style. Shown below are examples of how a new filling-style and line-style can be created.

```
(create-instance 'new-filling-style opal:filling-style
  (:foreground-color new-color))

(create-instance 'new-line-style opal:line-style
  (:foreground-color new-color))
```

B.5 Printing 3D Windows

The function *make-ps-file* is used to generate a postscript file for a Garnet window. It assumes that all objects in the window are regular 2D Garnet objects (e.g. opal:rectangle, opal:text, opal:circle, etc). To print a window containing 3D objects, the user must first convert them to regular 2D Garnet objects. The following function performs this conversion on all 3D objects in *window*.

(`opal:convert-all-3d-objs-to-garnet-objs` *window*)

Once the objects have been converted, transformations can no longer be made to the original 3D objects.

B.6 Geometrical Transformations

The basic geometrical transformations of rotation, scaling, and translation are supported. Rotation changes the orientation, scaling changes the size, and translations changes the location of an object. The transformation functions discussed below include an option *cumulative-p* parameter. When *cumulative-p* is true, the transformations are successively applied, whereas, when *cumulative-p* is nil, the previous transformations are ignored. For example, assume the 3d-rotate function described below is called with the following parameters.

```
(3d-rotate 3d-gob 10 30 0 nil)
```

When finished the object would be rotated 10 degrees in the x direction, 30 degrees in the y direction, and 0 degrees in the z direction. No matter how many times this function was called with these parameter, or what the previous rotate values were, the final result would also be the same. However if the 3d-rotate function was called with these parameters:

```
(3d-rotate 3d-gob 10 30 0 t)
```

Then the object would be rotated an additional 10 degrees in the x direction, an additional 30 degrees in the y direction, and no change in the z direction.

B.6.1 Translation

For all objects (except 3d-lines) the slots *:x*, *:y*, and *:z* determine the translation. For 3d-lines the slots (*:x1*, *:y1*, and *:z1*) and (*:x2*, *:y2*, and *:z2*) determine the location. To translate an object the user can change the value in one of the translate slots and then update the window, or call the function *3d-translate*.

`3d-translate` *3d-gob translate-x translate-y translate-z* `&optional` *cumulative-p*

This function translates 3d-gob by *translate-x*, *translate-y*, *translate-z*. This function allows the object to be translated in more than one direction, by executing only one statement.

B.6.2 Rotation

The slots *:rotate-x*, *:rotate-y*, and *:rotate-z* determine the amount of rotation. To rotate an object the user can change the value in one of the rotate slots and then update the window, or call the function *3d-rotate*.

`3d-rotate` *3d-gob rotate-x rotate-y rotate-z* `&optional` *cumulative-p*

This function rotates *3d-gob* by *rotate-x*, *rotate-y*, *rotate-z*. Unless specified, objects are rotated about their calculated center. The slots *:rotate-center-x*, *:rotate-center-y*, and *:rotate-center-z* allows the user to rotate an object about a point other than the center of the object. To force an object to be rotated again about its calculated center just set the slots *:rotate-center-x*, *:rotate-center-y*, and *:rotate-center-z* back to nil.

B.6.3 Scale

The slots *:scale-x*, *:scale-y*, and *:scale-z* determine the amount to resize the object. To scale an object the user can change the value of one of the scale slots and then update the window, or call the function *3d-scale*.

`3d-scale` *3d-gob scale-x scale-y scale-z* `&optional` *cumulative-p*

This function scales *3d-gob* by *scale-x*, *scale-y*, and *scale-z*.

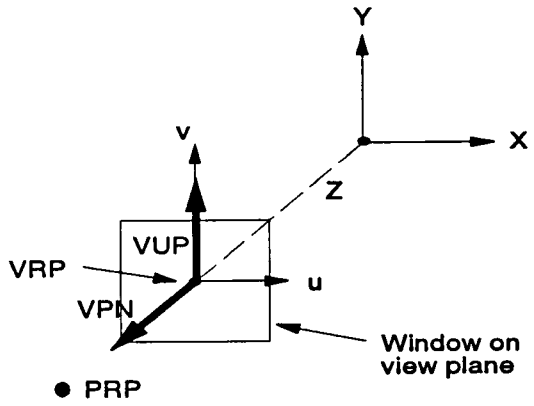


Figure B.13: The default viewing situation.

B.7 Viewing Parameters

Associated with each 3d-interactor-window is a set of viewing parameters. These parameters include projection type (:parallel-p), viewing reference point (:vrp), view plane normal (:vpn), view up vector (:vup), projection reference point (:prp), and the window size (:window-size). Figure B.13 shows the default viewing situation. Altogether, these parameters determine the type of planar geometric projection being performed. They are used to construct a normalizing transformation matrix which transforms 3D objects onto a 2D projection plane. 3D objects can only be displayed in a 3d-interactor window which is defined as follows:

```
(create-instance '3D-INTERACTOR-WINDOW inter:interactor-window
  (:visible nil)
  (:parallel-p nil)
  (:vrp '(0 0 75))
  (:vpn '(0 0 30))
  (:vup '(0 100))
  (:prp '(0 0 20))
  (:window-size '(-20 20 -20 20))
  (:front-clipping-plane 1000)
  (:back-clipping-plane -1000)
  (:clip-p nil))
```

When the viewing parameters are changed, objects which have already been projected onto the window are automatically redrawn using the new values.

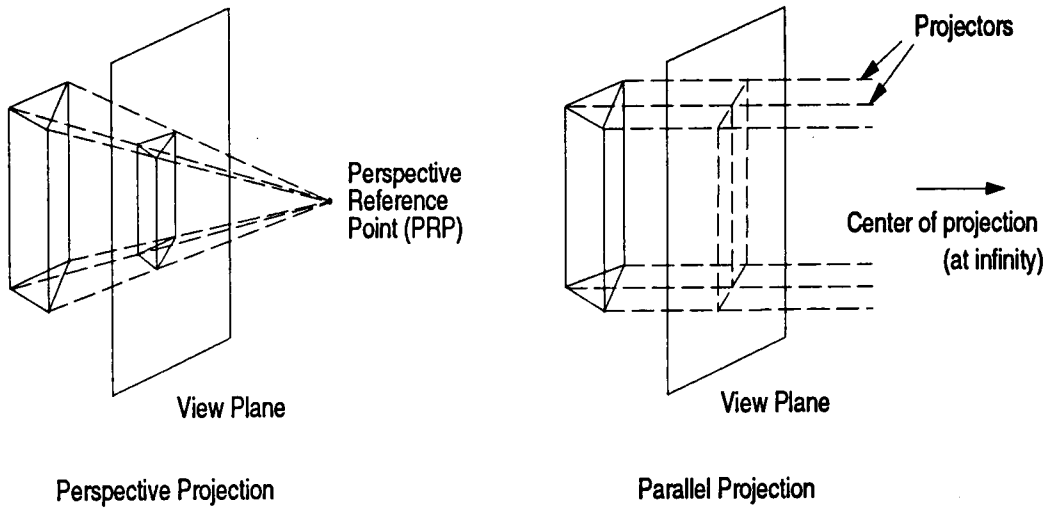


Figure B.14: A perspective vs parallel projection.

B.7.1 Parallel versus Perspective Projection

The slot *:parallel-p* determines the projection type. If *:parallel-p* is nil, then a perspective projection is performed. In perspective projections the distance from the center of projection to the view plane is finite. This type of projection is often the preferred method since it comes closest to how the human eye perceives the real world. A perspective projection preserves the property that further objects are smaller than closer objects.

When *:parallel-p* is true, then a parallel projection is performed. In parallel projections the distance from the center of projection to the view plane is infinite. This type of projection is called parallel because with the center of projection infinitely distant, the projectors are parallel to each other. This type of projection is similar to how the sun casts a shadow onto a wall. Figure B.14 illustrates the difference between the two major types of projections.

B.7.2 View Reference Point

The view plane is defined by a point on the plane called the *view reference point* VRP, and a vector normal to the view plane called the *view plane normal* (VPN). The VRP is specified in world coordinates.

B.7.3 View Plane Normal and View Up Vector

To define a window on the view plane, two orthogonal axes are needed. These two axes form part of the viewing-reference coordinate (VRC) system. The VPN is normal to the view plane and forms one axis. The VUP lies on the view plane and forms the second axis of the VRC. The v axis of the VRC is defined such that the projection of the VUP onto

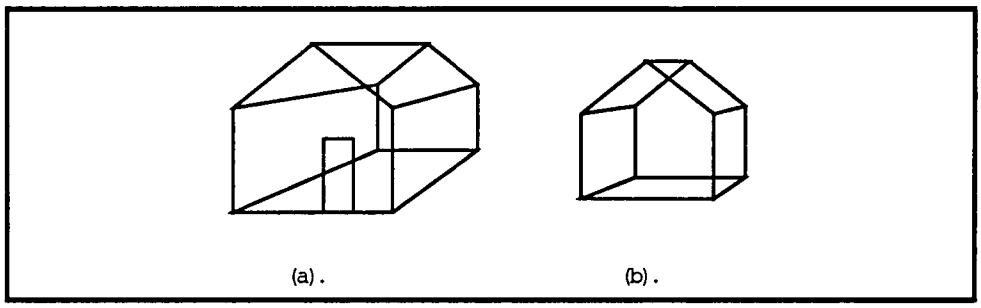


Figure B.15: A house (a) before clipping, and (b) after clipping.

the view plane coincides with the v axis. Both the VPN and VUP are specified in world coordinates.

B.7.4 Projection Reference Point

In a perspective projection, the Projection Reference Point (PRP) is the center of projection, whereas, in a parallel projection, the direction of projection is from the PRP to the center of the window. The PRP is specified in the viewing-reference coordinate (VRC) system. Therefore as the VUP or VRP are moved, the position of the PRP relative to the VRP does not change.

B.7.5 Window Size

The window resides on the view plane. Any part of an object that projects onto the view plane outside this window is not displayed. The coordinates are ordered as follows: (u-min, u-max, v-min, v-max). When the VUP coincides with the y axis and the VPN coincides with the z axis, then the u coordinates specify the width and the v coordinates specify the height of the window.

B.8 Clipping

Clipping is the process of determining those portions of primitives that lie inside the clip region, or stated another way, it is the process of blocking out all portions of primitives that lie outside the clip region. Clipping images can produce very interesting effects on 3D objects. For example, clipping allows the user to see into a house as shown in Figure B.15. The following three clipping slots are associated with each 3d-interactor-window.

:clip-p - indicates whether clipping occurs. The default value is nil.

:front-clipping-plane - is parallel to the view plane and is normal to the VPN. This plane is specified as a signed number, that indicates the relative distance from the VRP. Positive distances are in the direction of the VPN.

:back-clipping-plane - is parallel to the view plane and is normal to the VPN. This plane is specified as a signed number, that indicates the relative distance from the VRP. Positive distances are in the direction of the VPN. Only when the front clipping plane is greater than the back clipping plane are objects visible.

B.9 3D Constraints

Constraints provide a declarative means for expressing relationships among graphical properties and transformations of 3D objects. They can be used to keep objects rotated by the same amount or to ensure that their filling styles are the same. High-level constraints have been created to simplify expressing alignment relationships that depend on the position and size of 3D objects.

These constraints are desirable for two reasons. First the parameters used to specify the position and size of 3D objects vary from one object to the next. For example, the width of a cube is specified using a *width* parameter while a cone uses a *radius* parameter. Similarly, the exact position of a 3D object depends not only on the parameters (*:x*, *:y*, and *:z*), but also on the type of object. For example, these parameters specify the back lower left corner of a cube, whereas, they specify the center for a sphere. The second reason these high-level constraints are advantageous is because 3D objects can be scaled. Scaling results in yet another complication, to expressing a translation constraint, since an object's size is really a combination of both its specified dimensions and of any scale factors that has been applied. For example, if an object's width is 20 and its scale factor is 0.5 then its actual width is 10. Hence, we must also consider the object's scale parameters in order to correctly express a translation relationship. The following function is called to create a high-level constraint.

`3d-constraint` *object1 slot object2*
`&key` *object3 attach1 attach2 attach3 offset1 offset2 offset3*

object1 3D graphical object
slot can be either :x, :y, or :z
object2 3D graphical object

object3 3D graphical object
attach1 specifies the attachment point on object1.
attach2 specifies the attachment point on object2.
attach3 specifies the attachment point on object3.
offset1 indicates an additional amount of translation from the attached point of object1.
offset2 indicates an additional amount of translation from the attached point of object2.
offset3 indicates an additional amount of translation from the attached point of object3.

To simplify the specification of 3d-constraints, the following abbreviations are used.

X axis	Y axis	Z axis
L - left	T - top	F - front
C - center	C - center	C - center
R - right	B - bottom	B - back

For example, the following constraints specifies that the *:x* slot of object1 should be constrained to the left "L" side of object2.

```
3d-constraint object1 :x object2 :attach2 "L"
```

In this example only one attachment point was given. If no attached point is given, then the *:x* slot of object1 would be constrained to the *:x* slot of object2. Likewise, the following constraint specifies that the left side of object1 should be constrained to the ride side of object2.

```
3d-constraint object1 :x object2 :attach1 "L" :attach2 "R"
```

Constraints can also be expressed between three 3d-objects. For example, the following constraint specifies that object1 should be constrained to the left side of object2 and to the right side of object3.

```
3d-constraint object1 :x object2 :object3 object3 :attach2 "L"
  :attach3 "R"
```

B.10 3D Aggregates

Aggregate objects hold a collection of other graphical objects. The objects in an aggregate are called its *components* and the aggregate is the *parent* of each component. Aggregates

allow objects to be grouped into a hierarchical structure. The 3D graphical objects can be used as building blocks to create a higher level entity, which in turn can be used to create an even higher level structure, and so on.

The 3D aggregates use relative positions while the Garnet 2D model uses absolute positions. For example, an *:x* value of 25 means 25 coordinates from the start of the aggregate, whereas in the 2D model it would mean an absolute screen position of 25 pixels.

```
(create-instance '3d-agg opal:aggregadget
  (:x 0)
  (:y 0)
  (:z 0)
  (:rotate-x 0)
  (:rotate-y 0)
  (:rotate-z 0)
  (:scale-x 1)
  (:scale-y 1)
  (:scale-z 1)
  (:center-x 0)
  (:center-y 0)
  (:center-z 0))
```

B.10.1 Creating a 3D Aggregate

To illustrate how a 3D aggregate is constructed, we will create a 3d-agg with the name *cone-cube-agg*. To this 3d-agg we will add the two objects *my-cube* and *my-cone*. The objects will be displayed in a window called *my-window*.

- We must first create an instance of an opal:3d-agg.

  ```
  (create-instance 'cone-cube-agg opal:3d-agg)
  ```

- The second step is to add the newly created 3d-agg to a higher level aggregate. If the 3d-agg is the top level aggregate in the window, then it should be added to the :aggregate slot as shown below:

  ```
  (s-value my-window :aggregate cone-cube-agg)
  ```

  Otherwise, the 3d-agg can be added to a pre-existing aggregate like:

  ```
  (opal:add-component pre-existing-agg cone-cube-agg)
  ```

- If the objects where already added to another aggregate, say *my-3d-agg*, then they must first be removed before adding them to the new 3d-agg.

  ```
  (opal:remove-components my-3d-agg my-cube my-cone)
  ```

- Next we add the objects to the aggregate.

```
(opal:add-components cone-cube-agg my-cube my-cone)
```

- Finally, we must update the window.

```
(opal:update my-window)
```

To simplify creating 3D aggregates, the programmer can use the following function:

(`create-3d-agg` *parent-agg list-of-objects*)

This function creates a 3D aggregate using the *list-of-objects* as components. This new aggregate is added to parent-agg. The parameter *parent-agg* can be either a pre-existing aggregate or a 3d-interactor-window. Using this function, the previous aggregate could have been created as follows.

```
(setf cone-cube-agg (create-3d-agg my-window (list my-cone my-cube)))
```

B.10.2 Ungrouping the Objects in a 3D Aggregate

3D aggregates are used to group objects together. To ungroup the objects from a 3D aggregate the following function should be called.

(`ungroup-3d-agg` *3d-aggregate*)

This function removes all the components from this aggregate and adds them to the next higher aggregate in the hierarchy. If *3d-aggregate* is the top most aggregate in the window, then no action is taken. *Ungroup-3d-agg* merges the aggregate transformations to the next higher aggregate or to individual objects.

Using the example in the previous section, assume the aggregate *cone-cube-agg* was rotated by 30 degrees before we decided to ungroup the objects in this aggregate. Now before *cone-cube-agg* can be destroyed, both the cube and the cone must be rotated by an additional 30 degrees to maintain their previous appearance.

B.11 3D Interactors

Like 2D user interfaces, one of the most time consuming tasks of creating a highly interactive 3D user interface is handling the input. The input is usually asynchronous and may come from the mouse, keyboard, or other input devices. Typically, 3D graphic packages have provided little or no support for interaction with the user. Generally they send a stream of low-level input events for a window to the application program. The application program must then queue the events and process them sequentially. Some toolkits provide a facility for selecting objects, but provide no support for interactive operations that create, move, or

resize objects. Our 3D toolkit takes a drastically different approach based on the interactors model used in Garnet. The interactors model encapsulates all the common interactive behaviors in a user interface into a few basic object types (i.e., interactors). When an event is received, it is mapped into one or more of these interactor objects. Each interactor provides a number of parameters that allows its behavior to be customized, such as the event it begins on, the set of objects it operates on, and the type of feedback it provides.

As with 2D interfaces, it appears that there are only a few fundamentally different types of behaviors associated with 3D user interfaces. The major type of interactive behavior is handling geometric transformations. Therefore, an interactor has been created for each of the three basic types of geometric transformations (rotation, scaling, and translation). Other interactors have been created to select 3D objects, reposition projected images, enter new points, and create new 3D objects. Because of the commonly used 2D input devices (like the mouse), the interactors do not operate in all three directions simultaneously since ambiguous results might be obtained. To accommodate 2D input devices, the 3D interactors operate along at most two dimensions at any given time. The six types of 3D interactors are listed below.

- 3D-Move-Interactor
- 3D-Grow-Interactor
- 3D-Rotate-Interactor
- 3D-Selection-Interactor
- 3D-Reposition-Interactor
- 3D-Create-Interactor

The main reason interactors are so successful is that they provide a wide range of parameters that allow the behaviors to be customized. For example, designers can specify how often an object should be reprojected and redrawn as the interactor operates (e.g., continuously as the mouse is moved across the window, or only when the stop-event occurs). Other parameters can be used to specify a feedback-object, abort-events, stop-events, etc. Defaults values are provided to simplify creating interactors. Shown below are two parameters common to all 3D interactors.

:window - the window that the interactor operates on.

:start-where - where the mouse cursor should be for the interactor to start working.

The following parameter is common to all the transformation interactors.

:continuous-update - this slot controls how the objects are updated. When the *:continuous-update* slot is true, the selected object are reprojected and redrawn continuously as the mouse is moved across the window. In other words the object is continuously reprojected and redrawn from the time the *start-event* is encountered and continues

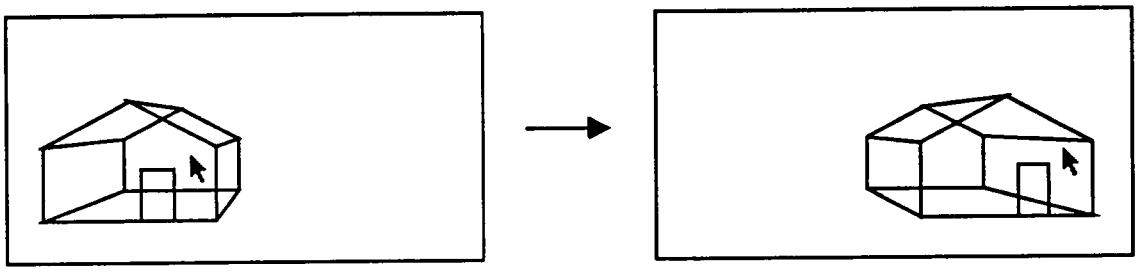


Figure B.16: Effect of using the 3D-move-interactor.

until the *stop-event* is reached. However, if the *:continuous-update* slot is false, then the object is not reprojected or redrawn until the *stop-event* occurs. This slot has been designed to accommodate the speed difference of various machines. On a slower machine, the user would probably set this slot to false, because reprojecting an object at each location as it is being dragged across the screen requires considerable processing power.

B.11.1 3D-Move-Interactor

The 3D-move-interactor can be used to move objects in the x, y, and z directions. The interactor translates objects so that their projected image lies under the cursor as shown in Figure B.16. To accomplish this, the system must first calculate the difference between the screen and world coordinates. This difference is calculated by 1) finding two world coordinate points on opposite ends of the object to be translated; 2) projecting these two 3D world coordinate points to find their associated 2D screen coordinates; and 3) computing the difference between the world coordinate points and screen coordinate points. For example, assume we were given the following world coordinates and their corresponding screen coordinates in pixels.

	world coordinates	screen coordinates
point1	(20, 20, 10)	(100, 100)
point2	(120, 120, 10)	(300, 250)

Given these values, we can now calculate that the x-difference is 2 and the y-difference is 1.5. Therefore, if the user moved an object 50 pixels to the right and 30 pixels toward the bottom, the object would have to be translated by a positive 25 world coordinates in the x direction and a negative 20 in the y direction. This interactor directly sets the *:x*, *:y*, and *:z* parameters of the object that is being moved.[1] Below is an example specification of a

[1]When working with 3D-lines, the 3D-move-interactor and the 3D-grow interactor set the parameters (*:x1*, *:y1*, *:z1*) and (*:x2*, *:y2*, *:z2*).

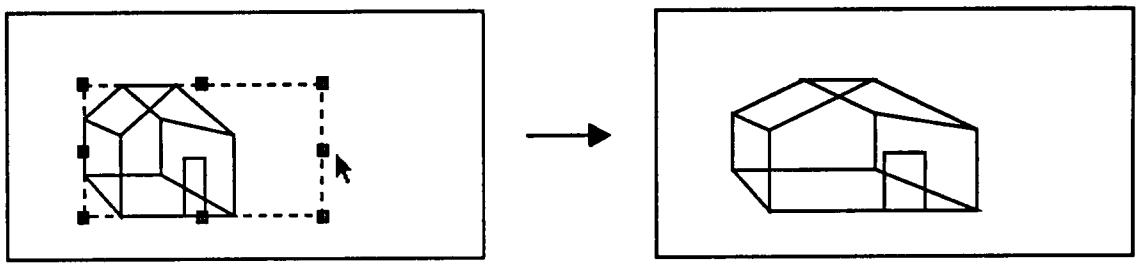


Figure B.17: Effect of using the 3D-grow-interactor.

3D-move-interactor.

```
(create-instance 'my-move-interactor inter:3d-move-interactor
  (:start-where (list :element-of 3d-projection-agg))
  (:window projection-window))
```

B.11.2 3D-Grow-Interactor

The 3D-grow-interactor can be used to resize objects in the x, y, and z directions. When resizing an object, the interactor tries to accurately scale the object so that the object's projected image lies within the bounding box highlighted by the selection-handles as shown in Figure B.17. To accomplish this, the difference between the screen and world coordinates is first calculated as discussed in the 3D-move-interactor. The object is then scaled by computing the difference between the original size of the bounding box and the new size of the bounding box when the stop-event is reached.

Resizing an object can be slightly more difficult than moving an object, because it sometimes requires not only scaling but also translation. For example, depending on which selection handle is grabbed — zero, one, or two translations are required so that the position of the object conforms to the position of the bounding box. For example, if the right middle selection handle is grabbed then no translations are required. If the left middle selection handle is grabbed then the object must also be translated in the x direction after being scaled. Finally, if the lower left selection handle is grabbed, then the object must also be translated in both the x and y directions after being scaled. This interactor may directly set any of the following parameters of the object that is being resized: *:x*, *:y*, *:z*, *:scale-x*, *:scale-y*, and *:scale-z*. Below is an example specification of a 3D-grow-interactor.

```
(create-instance 'my-grow-interactor inter:3d-grow-interactor
  (:start-where (list :element-of 3d-projection-agg))
  (:window projection-window)
  (:min-width 20))
```

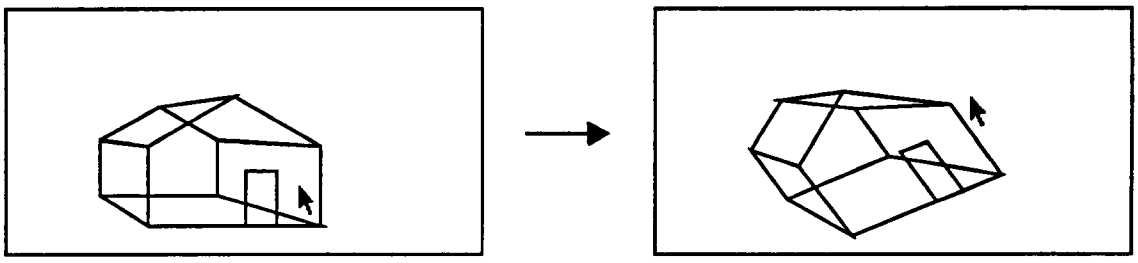


Figure B.18: Effect of using the 3D-rotate-interactor.

To prevent objects from becoming too small, the programmer can set the parameters *:min-width*, *:min-height*, and *:min-length*. These parameters are expressed in pixel values. An object cannot be resized so that its dimensions become less than these parameters.

B.11.3 3D-Rotate-Interactor

This interactor allows 3D objects to be rotated in either the x, y, or z direction. After an object has been selected, moving the cursor around the center of the object, causes the object to be rotated by an equivalent amount. Thus moving the cursor 30 degrees about the object causes the object to be rotated by an additional 30 degrees. The slot *direction* specifies the axis about which the rotation is performed. Acceptable values are *:x*, *:y*, and *:z*. Figure B.18 shows this interactor in action, where the direction slot has been set to *:z*. This interactor directly sets one of the following parameters *:rotate-x*, *:rotate-y*, or *:rotate-z*. Below is an example specification for a 3D-rotate-interactor.

```
(create-instance 'my-rotate-interactor inter:3d-rotate-interactor
  (:start-where (list :element-of 3d-projection-agg))
  (:direction :z)
  (:window projection-window))
```

B.11.4 3D-Selection-Interactor

The 3D-selection-interactor can be used to select one or more 3D objects. For clarity, handles appear over selected objects. Clicking the mouse when it is not over an object will de-select all objects. Sweeping out a rectangle with the left mouse button down will select any object that is completely enclosed within the rectangle. Control-leftdown toggles the object beneath the mouse between select and unselect. If an object was already selected, control-leftdown will remove it from the list of selected objects, otherwise it will be added to the list of selected objects.

The list of selected objects is stored in the parameter *:selected-objects* that is associated

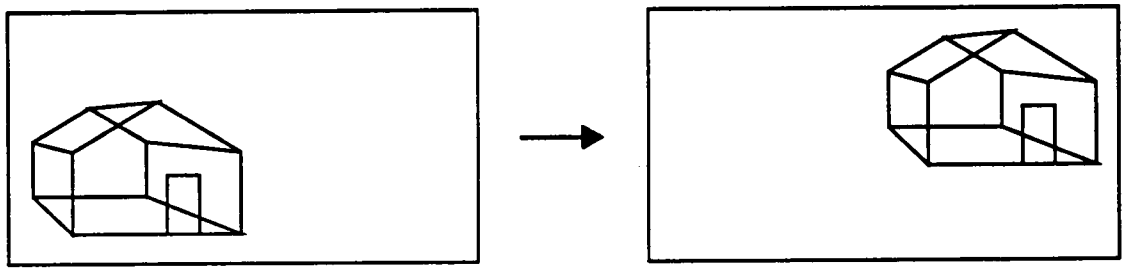


Figure B.19: Effect of using the 3D-reposition-interactor.

with the window. This parameter can be read by the programmer or another interactor at any time. Below is an example specification for a 3D-selection-interactor.

```
(create-instance 'my-selection-inter inter:3d-selection-interactor
  (:start-where (list :element-of 3d-projection-agg))
  (:window projection-window))
```

B.11.5 3D-Reposition-Interactor

The 3D-reposition-interactor can be used to move projected images around on the screen as shown in Figure B.19. A flag is set while the object is being repositioned so that the object's last projected image is maintained. This flag can later be reset to allow projections to be resumed. Interactive graphical tools can use this interactor to allow a user to temporarily move objects around while editing the scene. The difference between the 3D-move-interactor and this interactor is that the 3D-move-interactor reprojects the object, whereas this interactor moves the originally projected image. Below is an example specification for a 3D-reposition-interactor.

```
(create-instance 'my-reposition-inter inter:3d-reposition-interactor
  (:start-where (list :element-of 3d-projection-agg))
  (:window projection-window))
```

B.11.6 3D-Create-Interactor

The 3D-create-interactor allows the user to interactively construct objects. Figure B.20 shows an example of this interactor in action. The interactor operates by the user selecting a location on the screen. A 3D bounding box and a selection handle in the form of a circle will then appear, with the corner diagonally opposite the selection handle positioned at this location. From now on, this corner diagonally opposite the selection handle will be referred to as the "origin". The origin of the bounding box is fixed and cannot be moved. The selection handle represents the value of the second screen location and it can

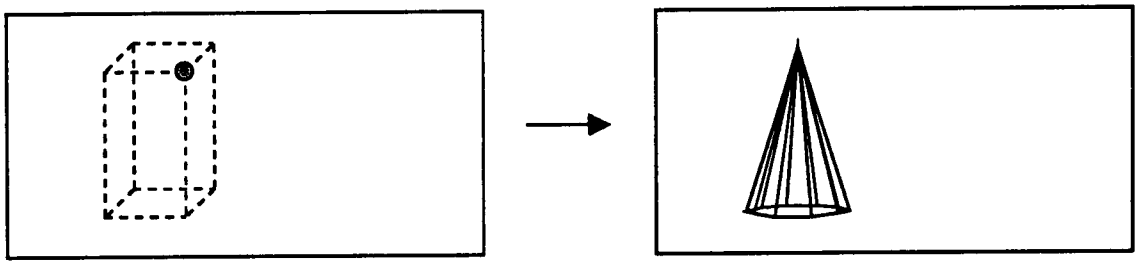


Figure B.20: An example illustrating the 3D-create-interactor.

be moved. The bounding box is resized when the selection handle is moved, so that the corner diagonally opposite the origin always coincides with the selection handle. Because of the commonly used 2D input devices, like the mouse, two parameters indicate how the selection handle can be resized.

The left mouse button allows the selection handle to be resized in the x and y directions. The middle mouse button allows the selection handle to be resized in the z direction.

When the stop-event for the interactor is reached, a 3D object is created whose center lies at the location specified by the parameters *:x*, *:y*, and *:z*. If any of these parameters are not specified, they obtain their default value from the *:vrp*, since the *:vrp* is typically located within the view volume. The user can then move the object to the desired location on the screen. The type of object created is determined by the parameter *:object* that is associated with the interactor. In Figure 6.11 the parameter *:object* was set to *:cone*.

An *:axis-scale* parameter is associated with this interactor that determines the ratio between screen pixels and the calculated size of the bounding box. For example, assume the user resizes the 3D bounding box so that its dimensions are 20 x 40 x 80. If the axis-scale parameter is set to 1.5, and the *:object* slot is set to *cube*, then the interactor creates a cube with the dimensions 30, 60, 120. However, if the axis-scale is set to 0.5, then the interactor creates a cube with the dimensions 10, 20, 40.

There are two unique parameters associated with this interactor that the user can customize. The first parameter is *:add-objects-to-agg*. This parameter can be changed to add 3D objects to various aggregates in the window. The second parameter is *:wire-frame-p*. This parameter can be changed so that created objects are rendered using either wire-frames or are shaded. When an object is shaded it uses the default filling colors. Below is an example specification for a 3D-create-interactor.

```
(create-instance 'my-create-interactor inter:3d-create-interactor
  (:start-where (list :element-of 3d-projection-agg))
  (:add-object-to-agg 3d-projection-agg)
  (:object cone)
  (:window projection-window))
```

Appendix C

3D Reference Manual: Direct Manipulation Layer

The direct manipulation layer of the 3D toolkit provides an interactive graphical environment for creating 3D scenes. This layer allows the user to interactively display and transform 3D objects, perform clipping, create aggregate objects, modify the viewing parameters that control the transformations of objects, and construct polyhedron objects. To support these operations, the toolkit includes the following interactive editors:

- Scene Editor
- Polyhedron Editor
- Aggregate Editor

The interactive environment is implemented using the programming layer, thus is itself an application of the 3D toolkit.

C.1 Scene Editor

The scene editor is used to create the final appearance of objects and aggregates in the 3D scene. This editor is composed of the following windows. The *scene window* allows the user to interactively create objects and apply transformations to them. The *scene control window* provides fine grain control over the entire 3D viewing process. The *viewing parameter windows* allow the user to interactively specify the 3D view volume that determines how objects will be projected. The *change center of rotation window* allows the user to interactively specify a new center of rotation for a 3D object or an aggregate. Each of these windows is described in more detail in the following sections.

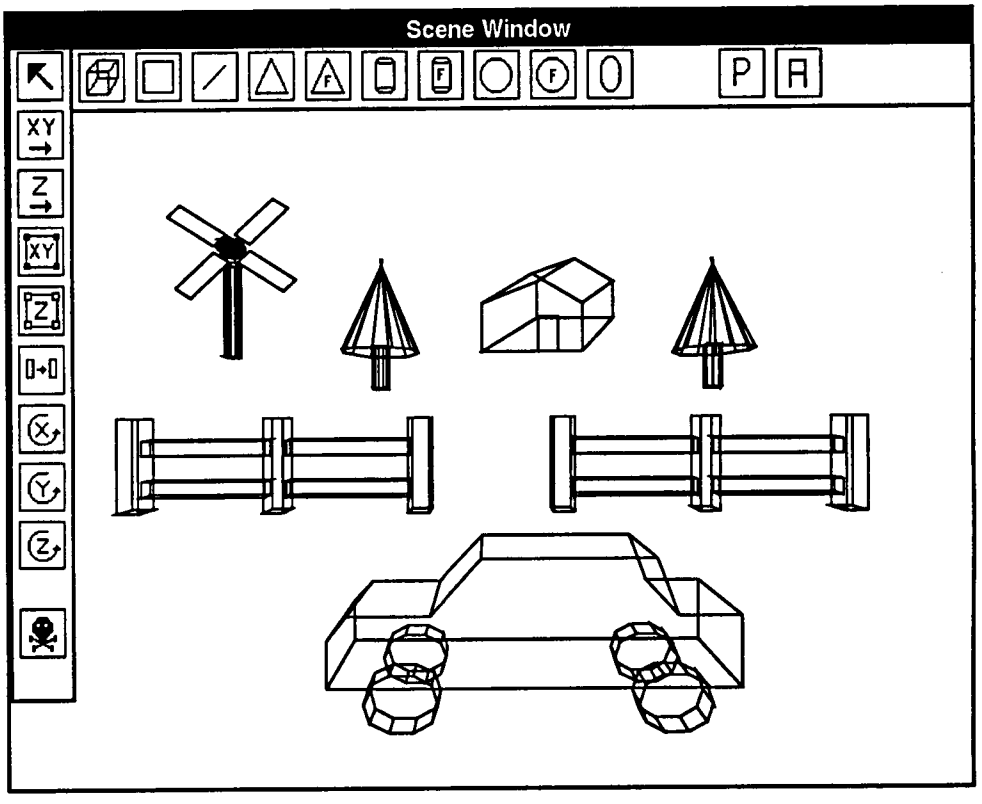


Figure C.1: The scene window.

C.1.1 Scene Window

The scene window is primarily used for creating and positioning objects in the scene as shown in Figure C.1. The position and appearance of objects in the scene window are controlled by the various transformation options on the left side of this window. Objects can be added to the scene window by selecting the appropriate icon located at the top of this window. We will now examine the icons on the left side of the scene window in more detail.

Selection - allows the user to select objects from the scene-window. For clarity, handles appear over selected objects. Clicking the mouse when it is not over an object de-selects all objects. Sweeping out a rectangle with the left mouse button down selects all objects that are completely enclosed within the rectangle. Control-leftdown toggles the object beneath the mouse between select and unselect. If an object was already selected, control-leftdown removes it from the list of selected objects, otherwise it is added to the list of selected objects. The slot *:selected-objects* is stored in the scene-window and contains a list of currently selected objects.

XY Move - allows objects to be moved in either the x or y direction using the left mouse button.

Z Move - allows objects to be moved in the z direction using the left mouse button.

XY Resize - allows objects to be resized in either the x or y direction. A bounding box and selection handles appear when the object is selected. To resize the object, the user can drag one of the black boxes to the desired size. The interactor translates the object's world coordinates so that their projected image always lies under the cursor. To prevent objects from being scaled to small, the user can set the parameters *:min-width* and *:min-height*. These parameters are expressed in pixel values. An object cannot be resized so that its dimensions become less than these parameters.

Z Resize - allows the user to resize objects in the z direction. A 3d bounding box is drawn on the screen to give the user some feedback as to the extent of the resize. To prevent objects from becoming too small, the programmer can set the parameter *:min-length*. An object cannot be resized so that its dimensions become less than this parameter.

Reposition - allows a user to temporarily move objects that are obscuring a portion of the display they are editing. An object will remain at this temporary location until the "project" method of the repositioned object is called. An object's "project" method can be called by directly manipulating the object using another interactor or by declaratively changing the value of an interesting parameter.

X Rotate - allows objects to be rotated about the x axis. After an object has been selected, moving the cursor around the center of the object causes the object to be rotated by an equivalent amount.

Y Rotate - allows objects to be rotated about the y axis. After an object has been selected, moving the cursor around the center of the object causes the object to be rotated by an equivalent amount.

Z Rotate - allows objects to be rotated about the z axis. After an object has been selected, moving the cursor around the center of the object causes the object to be rotated by an equivalent amount.

Remove - deletes all currently selected objects.

When one of the following icons located at the top of this window is selected, the *object* parameter of the 3D-create-interactor is set to that object. For example, if the icon resembling a cone is selected, then the *object* parameter is set to "cone". The user can then interactively create various sized cones anywhere in the scene window. More specifically, these icons function as follows:

Cube - sets the *object* parameter to cube.

 **Plane** - sets the *object* parameter to plane.

 **3D-line** - sets the *object* parameter to 3D-line.

 **Cone** - sets the *object* parameter to cone.

 **Fast Cone** - sets the *object* parameter to cone and the fast-draw variable to true. See Section 6.2 for a detailed description of the fast-draw objects. Cones, cylinders, and spheres are the only objects that have associated fast-draw methods.

 **Cylinder** - sets the *object* parameter to cylinder.

 **Fast Cylinder** - sets the *object* parameter to cylinder and the fast-draw variable to true.

 **Sphere** - sets the *object* parameter to sphere.

 **Fast Sphere** - sets the *object* parameter to sphere and the fast-draw variable to true.

 **Ellipsoid** - sets the *object* parameter to ellipsoid.

By pressing the  icon, the polyhedron editor will be displayed and the user can begin constructing a polyhedron object. By pressing the  icon, the aggregate editor will be displayed and the user can begin constructing an aggregate object.

C.1.2 Scene Control Window

The scene control window (Figure C.2) provides fine grain control for the overall 3D viewing process, in contrast to the scene window which only allows approximate placement of objects. For example, this window allows the user to precisely set the viewing parameters, the clipping parameters, and apply transformations to objects. In the example scene shown in Figure C.1, the scene control window was used to set the rendering mode to "wire-frame" and the projection style to "perspective". We now examine the features of the scene control window in more detail.

C.1.2.1 Options

The top left corner of the scene control window contains several features that allow the user to change the rendering mode, the projection type, and to write and read scenes to and from files:

Stop - exits the 3D toolkit.

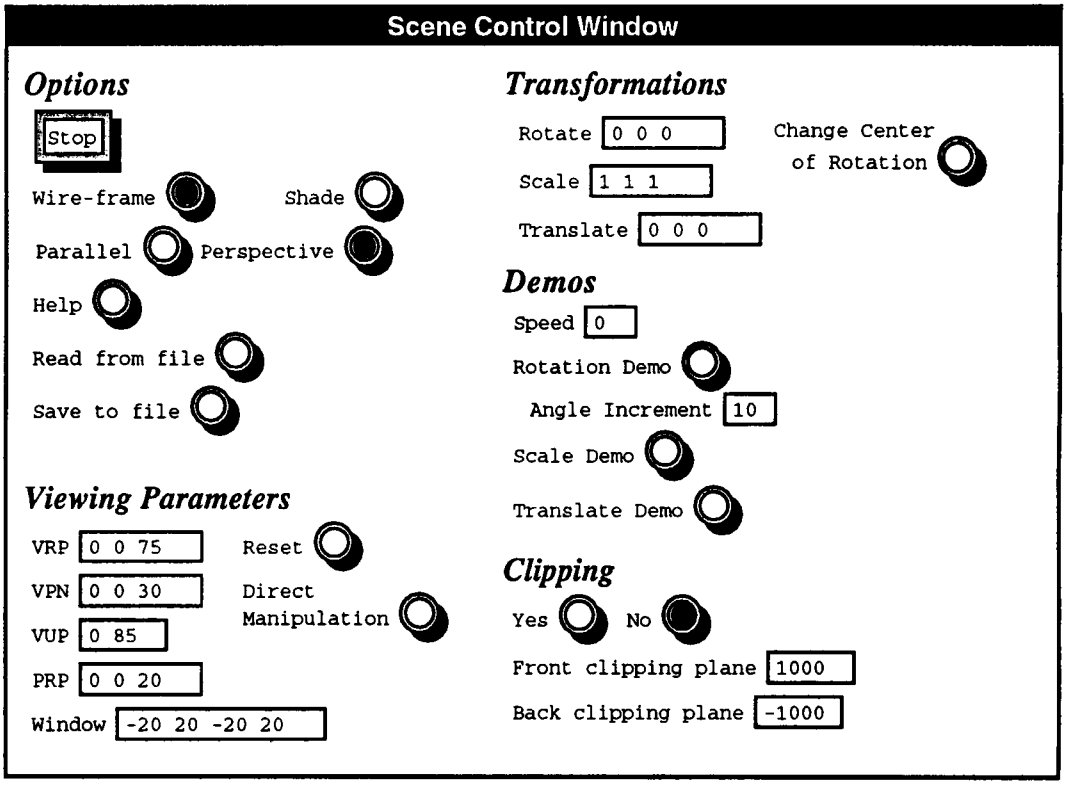


Figure C.2: The scene control window.

Wire-frame / Shade - specifies the rendering mode as either wire-frame or shaded. This option only applies to the selected objects in the scene window and their components. Changing the rendering mode of a selected 3D aggregate changes the rendering mode for all its components. When the "Wire-frame" radio button is pressed, the slot *:wire-frame-p* is set to true for all currently selected objects in the scene-window. When the "Shade" button is pressed, the slot *:wire-frame-p* is set to nil for all currently selected objects.

Parallel / Perspective - specifies the projection type as either parallel or perspective. When the "Parallel" radio button is pressed, the slot *:parallel-p* is set to true in the scene-window, whereas, when the "Perspective" button is selected, this slot is set to nil. When the *:parallel-p* parameter is changed, all objects that have been projected onto the scene-window are automatically redrawn using the new projection type. Depending on the other parameters, changing the projection type may result in the objects no longer being visible in the scene-window.

Help - provides general help on the 3D toolkit. Clicking the middle mouse button on any text in the scene control window causes the scene control editor to display a helpful message on the option or parameter selected.

Read from file - allows the user to read scenes from a file. A scene may consist of one or more objects.

Save to file - saves all the objects in the scene window to a file. This feature uses the Garnet save mechanism. Scenes that are saved using this feature are represented using code from the programming layer, so the designer is free to modify the code in any way desired.

C.1.2.2 Viewing Parameters

The viewing parameters can be changed declaratively by using the text boxes in this window or interactively by pressing the "Direct Manipulation" button:

VRP - allows the user to change the value of the view reference point (*:vrp*) in the scene-window. The VRP is specified using three world coordinates. The first coordinate represents the x value, the second coordinate is the y value, and the third coordinate specifies the z value.

VPN - allows the user to change the value of the view plane normal (*:vpn*) in the scene-window. The VPN is also specified using three world coordinates. The first coordinate represents the x value, the second coordinate is the y value, and the third coordinate specifies the z value.

VUP - allows the user to change the value of the view up vector (*:vup*) in the scene-window. The VUP is specified using viewing-reference coordinates. The first coordinate represents the value of the u axis, and the second coordinate represents the value of the v axis.

PRP - allows the user to change the value of the projection reference point (*:prp*) in the scene-window. The PRP is also specified using viewing-reference coordinates. The first coordinate represents the value of the u axis, the second coordinate represents the value of the v axis, and the third coordinate is the n axis.

Window - allows the user to change the value of the window size (*:window-size*) in the scene-window. This window resides on the view plane. Any part of an object that projects onto the view plane outside this window is not displayed. Four coordinates are required to specify the window size. The coordinates are ordered as follows: (u-min, u-max, v-min, v-max).

Reset - resets the viewing parameters back to their default values. The default values are:

VRP	(15 0 75)
VPN	(0 0 30)
VUP	(0 100)
PRP	(0 0 20)
Window Size	(-10 30 -20 20)

Direct Manipulation - allows the user to interactively modifying the viewing parameters using pictorial diagrams rather than the text boxes shown in Figure C.2. The windows that appear when the "Direct Manipulation" button is pressed are listed below.

Viewing Control Window - specifies the axis scale to be used and allows the user to terminate the direct manipulation of these parameters.

View Plane Window - allows the user to modify the view reference point and the view plane normal that together define the view plane.

Viewing Reference Window - allows the user to directly modify the view up vector and the window size. Together these two pieces of information create the viewing reference coordinate (VRC) system.

Perspective Reference Point Window - allows the user to interactively change the perspective reference point.

The view reference point, view plane normal, and projection reference point are specified using 3D points. Because of the commonly used 2D input devices (like the mouse), two different mouse buttons are used to change the values of these parameters. Otherwise ambiguous results might be obtained. The left mouse button is used to change these parameters in the x and y directions, and the middle mouse button is used to change these parameters in the z direction.

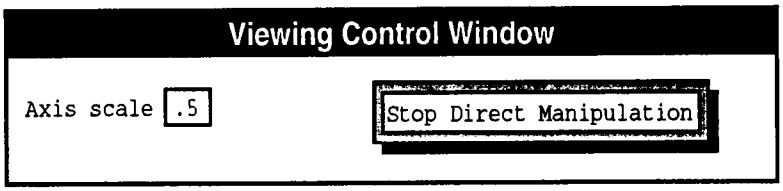


Figure C.3: The viewing control window.

Viewing Control Window This window allows the user to specify an axis scale that is used while interactively modifying the viewing parameters as shown in Figure C.3. The axis scale parameter determines the ratio between screen and world coordinates when interactively modifying the following parameters: view reference point, view plane normal, view up vector, perspective reference point, and window size. As an example, if axis-scale is set to 1.5 and the perspective reference point (PRP) is moved 100 screen pixels to the right, this would result in the x-value for the PRP being increased by 150 world coordinates. However, if the axis-scale is set to 0.5 and the PRP is moved 100 screen pixels to the right, this would result in the x-value for the PRP being increased by only 50 world coordinates. Thus the axis-scale parameter controls the granularity of changes caused by moving a point with the mouse.

This window also contains the option "Stop Direct Manipulation" that terminates the direct manipulation of the viewing parameters. Pressing this option removes all four viewing parameter windows from the screen.

View Plane Window This window allows the user to change the values of the view reference point and view plane normal as shown in Figure C.4. The shaded plane denotes the orientation of the view plane. To move the VRP in the x or y direction, click the left mouse button over the VRP circle, and while held down move it to the desired position. To move the VRP in the z direction, perform these same steps using the middle mouse button. The right mouse button acts like a 3-way toggle switch for the VRP. When pressed once over the VRP circle, a 3d bounding box will be displayed showing the x, y, and z values for the VRP. If clicked a second time, only axis indicators will be visible and the 3d-bounding box will disappear. Finally, when pressed a third time, both the 3d bounding box and axis indicators will disappear.

To move the VPN in the x or y direction, click the left mouse button over the VPN circle, and while held down move it to the desired position. To move the VPN in the z direction do the same with the middle mouse button. The right mouse button acts like a 3-way toggle switch for the VPN. When pressed once over the VPN circle, a 3d bounding box will be displayed showing the x, y, and z values for the VPN. If clicked a second time, only axis indicators will be visible and the 3d-bounding box will disappear. Finally, when pressed a third time, both the 3d bounding box and axis indicators will disappear.

Pressing control-left down over the origin of the vrp-and-vpn-window allows the user

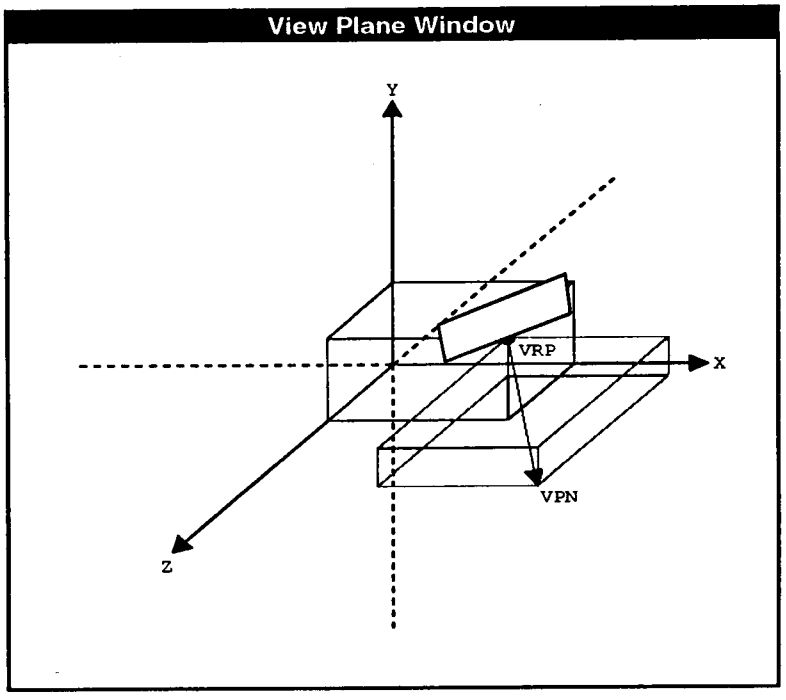


Figure C.4: The view plane window.

to toggle between displaying and not displaying the world coordinate axes. Pressing control-left down over the VRP circle allows the user to toggle between displaying and not displaying the modified world coordinate axes. Control-middle-down over the VRP circle allows the user to toggle the view-plane-window between visible and not visible.

Viewing Reference Window. This window allows the user to change the window size and the direction of the view up vector as shown in Figure C.5. The origin of the viewing reference window is the view reference point (VRP). The window size can be interactively changed by selecting the window with the left mouse button. When this happens, resize and selection handles appear around the perimeter of the window. The user can then resize the window by dragging one of the resize handles, or move the window by dragging one of the selection handles.

The direction of the view up vector can be changed by grabbing the end of the vector labeled VUP and dragging it to the desired position. When the mouse button is released, the window will be redrawn in accordance with the new VUP. Figure C.6 illustrates how the rotation of the window coincides with the VUP. Although, the u vector cannot be selected, it will always remain at a 90 degree angle with the VUP. The variables *3dim:min-view-plane-width* and *3dim:min-view-plane-height* may be changed to control how small the view-plane-window may become. Their default value is 10.

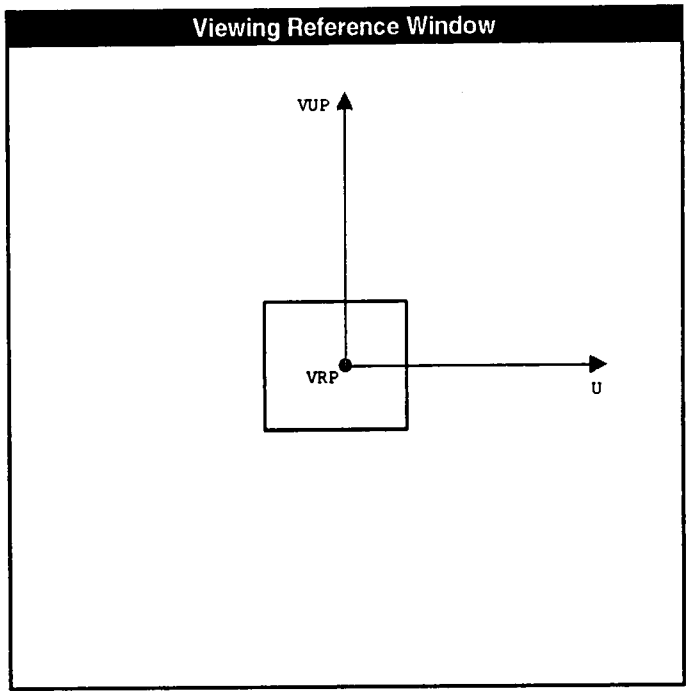


Figure C.5: The viewing reference window (shaded rectangle) and the two vectors that define its coordinate system.

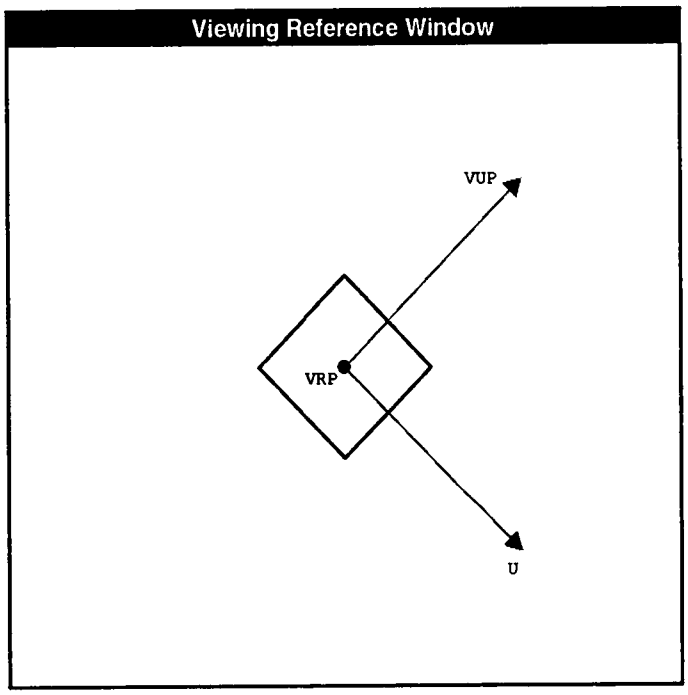


Figure C.6: The VUP vector has been rotated 45 degrees counter clockwise about the VRP. Consequently, the window is also rotated by the same amount so that its orientation matches the VUP.

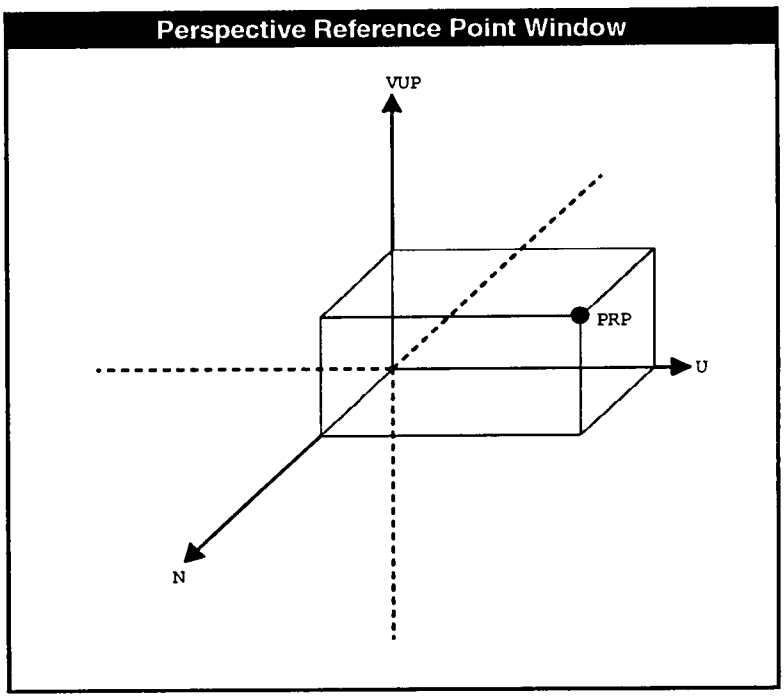


Figure C.7: The perspective reference point window.

Perspective Reference Point Window This window allows the user to change the value of the projection reference point (PRP) as shown in Figure C.7. The PRP is defined relative to the 3D viewing reference coordinate (VRC) system. The origin of the VRC system is the VRP. The n axis is the VPN vector.

To move the PRP in the x or y direction, click the left mouse button over the PRP circle, and while held down move it to the desired position. To move the PRP in the z direction do the same with the middle mouse button. The right mouse button acts like a 3-way toggle switch. When pressed once over the PRP circle, a 3d bounding box will be displayed showing the values for the VUP, U, and N axes. If clicked a second time, only axis indicators will be visible and the 3d-bounding box will disappear. Finally, when clicked a third time, both the 3d bounding box and axis indicators will disappear.

C.1.2.3 Transformations

The transformation boxes display the current transformation parameters for the selected object and provide an alternative way to modify the parameters (i.e., they provide finer grain control than the interactors in the scene window). If the user changes the values in one of these boxes, then these transformations are applied to all currently selected objects in the scene window.

C.1.2.4 Rotate Box

Displays the current rotation values for the object selected. If the values in this box are changed, then all currently selected objects in the scene-window are rotated by these new values.

C.1.2.5 Scale

Displays the current scale values for the object selected. If the values in this box are changed, then all currently selected objects in the scene-window are scaled by these new values.

C.1.2.6 Translate

Displays the current translation values for the object selected. If the values in this box are changed, then all currently selected objects in the scene-window are translated by these new values.

Change Center of Rotation Window This window allows the user to interactively specify a new center of rotation for either a 3D object or a 3D aggregate. The center of rotation can be specified to lie within the object, within another object, or about the origin. The rotation values can be specified by using sliders or by selecting one of three predefined options as shown in Figure C.8:

Sliders - can be used when the desired center of rotation is within the object or aggregate. A different slider has been created for each axis. The value of a slider represents a percentage of the object. For example, to rotate an object about its own center the sliders should all be set at 50. Similarly, to rotate an object about its smallest x coordinate the "X Axis" slider should be set to 0.

Use Calculated Center - rotates the selected object about its center. Selecting this option sets the x, y, and z axis sliders to 50.

Rotate About Another Object - allows an object to be rotated about a different object. This other object is selected after choosing this option.

Rotate About Origin - rotates the object about the origin (0, 0, 0) of its aggregate.

C.1.2.7 Demos

The options in this section allow the user to apply some pre-defined demo transformations to the objects in the scene window. These demos are designed to assist in developing a 3D

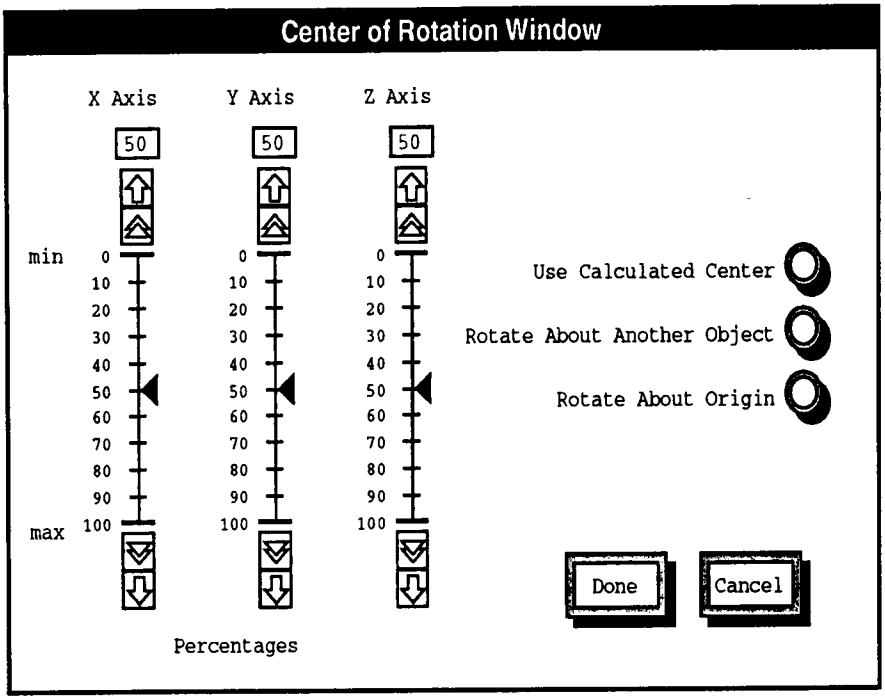


Figure C.8: The change center of rotation window allows the user to interactively specify a new center of rotation for either a 3D object or a 3D aggregate.

scene, by giving the user a quick idea of how the objects might appear at different angles, sizes, and locations. After the demo transformations have been applied, the objects are returned to their original transformation values.

Speed - ceases execution of the specified demo for n seconds of real time between operations, where n can be a fractional number. This option applies for all three of the demos listed below.

Rotation Demo - rotates all currently selected object(s) 360 degrees about the x axis, then the y axis, and finally the z axis. The box labeled "Angle Increment" determines the angle between successive rotations along each axis. For example, if "Angle Increment" equals 1 then the selected objects are rotated 360 times in each of the x, y, and z directions. If "Angle Increment" equals 10, then the selected objects are rotated 36 times about each axis. In this latter case, the objects would be rotated by the angles: 10, 20, 30, ... 340, 350, 360.

Scale Demo - scales all selected object(s) in the x, y, and z directions by some pre-defined values.

Translate Demo - translates all selected object(s) in the x, y, and z directions by some pre-defined values.

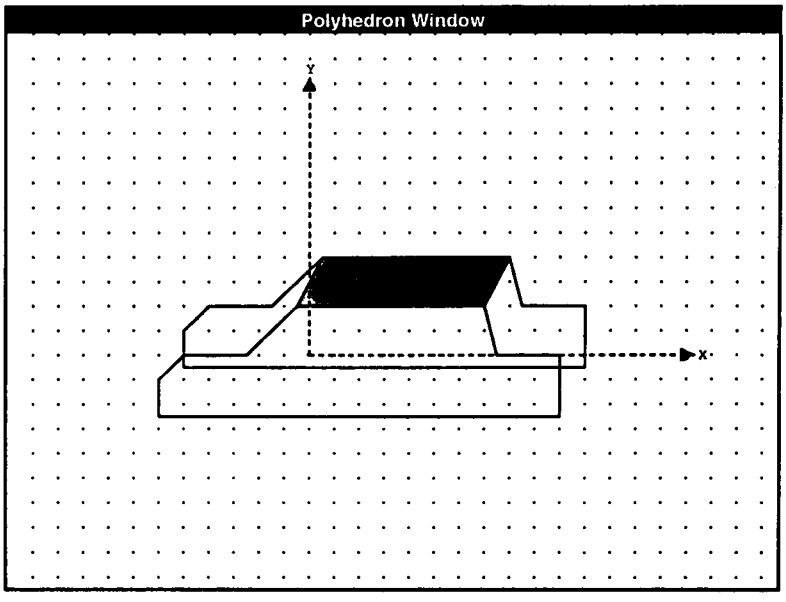


Figure C.9: The polyhedron window.

C.1.2.8 Clipping

The options in this section allow the user to determine which portion of objects to display. When clipping is turned on, the text box labeled "Front clipping plane" specifies the front clipping plane, and the text box labeled "Back clipping plane" specifies the back clipping plane.

C.2 Polyhedron Editor

The polyhedron editor (Figures C.9 and C.10) allows the user to quickly and interactively create polyhedron objects. The constructed object can be rotated, scaled, and translated just like all the predefined 3D objects. The user is completely freed of having to specify points, and then defining how the points should be connected to form a surface. The user instead interactively creates surfaces and then directly moves them in either the xy, or z direction. To begin creating a polyhedron object select the "P" icon from either the scene-control-window or the aggregate-window.

Points in the polyhedron window correspond to world coordinate points. The house and the body of the car shown in Figure C.1 were created using these windows. To construct the car body the user performed the following actions:

1. Selected the "polyline creator" [~] from the polyhedron control window (Figure C.10) and traced out the shape of the side of a car in the polyhedron window (Figure C.9).

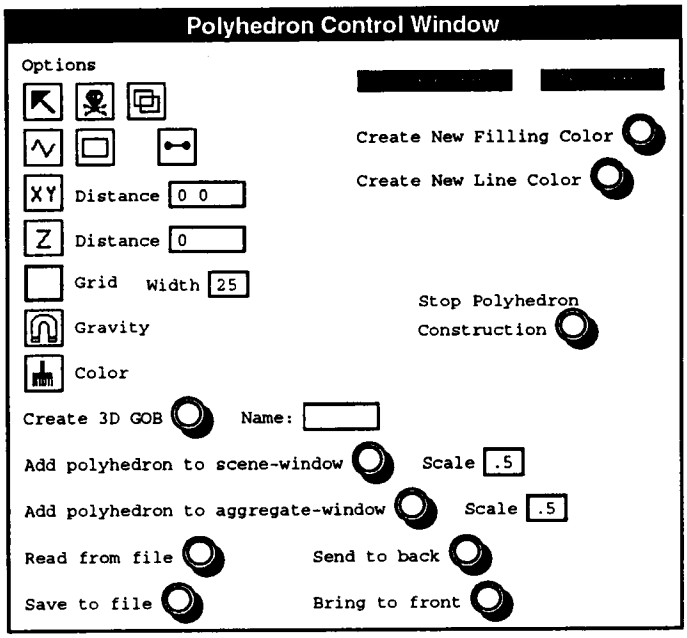


Figure C.10: The polyhedron control window.

2. Pressed the "copy" option to create an instance of this side.
3. Used the "z move" option to move this instance in the z direction.
4. Used the "connect points" option to connect the two sides.

Figure C.9 shows the car after both sides and the top have been created. The user added this object to the aggregate window by selecting the option "Add polyhedron to aggregate-window". By pressing either the button "Create New Filling Color" or "Create New Line Color" the visual color editor shown in Figure C.11 is displayed. We will now examine the features of this editor in more detail.

Selection - allows the user to select objects from the polyhedron window. Selection handles appear over selected objects. Clicking the mouse when it is not over an object will de-select all objects. A button-down-drag-button-up movement will create a bounding box that selects any objects that the bounding box completely encloses. When objects overlap, this interactor will select that object which was created last. Control-leftdown toggles the object beneath the mouse between select and unselect. If an object was already selected, control-leftdown will remove it from the list of selected objects, otherwise it will add it to the list of selected objects.

Remove - deletes all objects that are currently selected.

Copy - makes a copy of all objects that are currently selected.

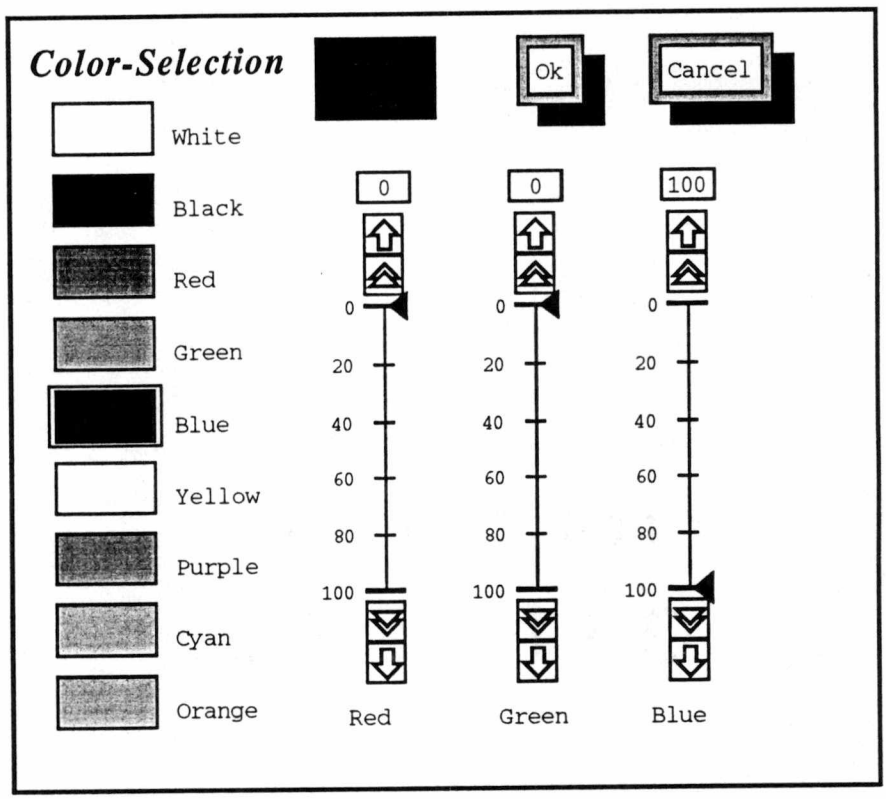


Figure C.11: The visual color editor allows the user to interactively create new colors with which to shade objects and lines. Colors can be selected either by choosing one of the colors in the menu on the left or by manipulating the three sliders on the right. The color box at the top provides feedback showing the selected color.

Polyline creator - creates a polyline. When the "grid" option is turned on, the points will be snapped to the grid. The polyline creator works as follows. When the user presses down the left mouse button the first point of the polyline is produced. Subsequent line segments are produced every time the button is pressed down. Feedback will be shown as the polyline is being created. The polyline is terminated by pressing any button other than the left one. The polyline creator sets the *:z* slot of the newly created polyline to 0. The polyline can later be moved in the xy-direction using the "XY Move" option, or in the z-direction using the "Z Move" option.

Rectangle creator - creates a rectangle with the *:z* slot set to 0. The rectangle can later be moved in the xy-direction using the "XY Move" option, or in the z-direction using the "Z Move" option.

Connect Points - connects points that have been specified using the polyline or rectangle creators. The connect-points option can create a surface composed of coordinates with different z values. For example, if constructing a house: the "polyline creator" could be use to build the front of the house. The "Copy" option could then be used to produce the back. Finally, the "connect points" feature could be used to create the sides of the house.

XY Move - allows object(s) to be moved in the xy direction. The box labeled "Distance" following this option indicates the current xy value of the object. If several objects have been selected, this box indicates the xy value of the object under the mouse. This option aligns the objects to grids when gravity is specified.

Z Move - allows object(s) to be moved in the z direction. The box labeled "Distance" following this option indicates the current z value of the object. If several objects have been selected, this box indicates the z value of the object under the mouse. This option aligns the objects to grids when gravity is specified.

Grid - lays down a grid in the polyhedron window. The X and Y axes of this window will always lie on the grid. The box labeled "Width" specifies the distance between grid markers in pixels. Objects can be forced to align to the grid by selecting the gravity option described next.

Gravity - snaps all objects to grid markers as they are moved or created. If the grid width is changed, previously created objects are aligned to the grid as best as possible as they are moved.

Color - turns on and off the filling style for objects. This option is needed because surfaces created with a non-nil filling-style would cover-up previously created surfaces. Finding the end points of the covered surfaces would then be difficult. For example, suppose we wish to construct a cube. Assume the user first created a back surface and then created a front surface with a non-nil filling-style. In this case creating the

sides would now be difficult because the user could not see the back surface. Because of this problem the color option was created and behaves as follows.

When the color feature is turned off, the filling-style for all objects is set to nil. When this happens only a wire-frame rendering of the objects is visible. If the filling style for an object is changed when the "Color" option is off, then the new filling style will remain visible for 2 seconds for the user to view the new change. At the end of 2 seconds, the filling-style for the object will return to nil. When the color feature is turned on, the filling-style for all objects is displayed. Note that using the options "Send to back" and "Bring to front" can help solve this problem too.

Filling Style - creates a pull-down menu of pre-defined colors. All currently selected objects are redrawn using the new color as their filling-style.

Line Style - creates a pull-down menu of pre-defined colors. All currently selected objects are redrawn using the new color as their line-style.

Create New Filling Color - creates a color window as shown in Figure C.11. All currently selected objects are redrawn using the new color as their filling-style.

Create New Line Color - creates a color window as shown in Figure C.11. All currently selected objects are redrawn using the new color as their line-style.

Stop Polyhedron Construction - destroys the polyhedron-window and the polyhedron-control-window.

Create 3D GOB - converts the Garnet objects (i.e. polylines) to a 3d polyhedron. Using the labeled box "Name:" the polyhedron can be given a name. The name feature can facilitate inspecting and debugging objects.

Add polyhedron to Scene-Window - adds the object in the polyhedron-window to the scene-window. As the object is being added, it can be scaled using the box labeled *Scale* following this option. It is usually easier to construct larger objects and then scale them down.

Add polyhedron to Aggregate-Window - adds the object in the polyhedron-window to the aggregate-window. As the object is being added, it can be scaled using the box labeled *Scale* following this option.

File I/O - allows the user to read and write object to and from files. When pressed, a window will appear to allow the user to specify the pathname of the file.

Send to Back/Bring to Front - determines the ordering of the objects. This option is useful when determining the filling style and when drawing a plane on a plane. If two overlapping sides of an object lie on the same plane, the 3D shading algorithm places the last created object on top. For example, when constructing a window on the side of a house, the side should be drawn first and then the window. This will result in the window being on top. However, if the window was drawn first, then the

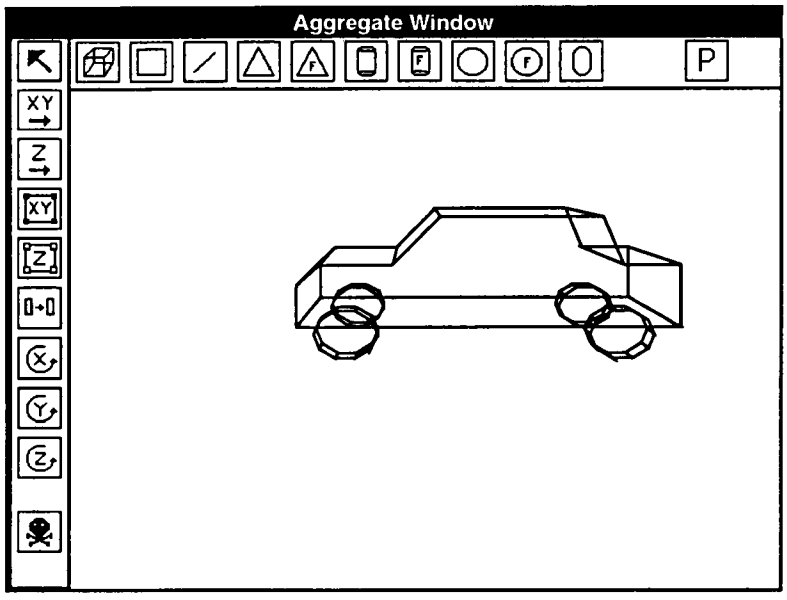


Figure C.12: The aggregate window.

side would cover the window. This problem could be corrected in one of two ways. The first option would be to select the side of the house and then press *Send to back.* The second possibility would be to select the window and then invoke *Bring to front.*

Send to back - moves the selected object(s) to the back of the aggregate. The object(s) are then drawn first, and possibly covered by other objects. When multiple objects are selected, the ordering among them is random, however they are all drawn before any object that was not selected.

Bring to front - moves the selected object(s) to the front of the aggregate. The object(s) are then drawn last and thus will not be covered by objects drawn earlier. When multiple objects are selected, the ordering among them is random, however they are all drawn after any object that was not selected.

C.3 Aggregate Editor

The aggregate editor (Figures C.12 and Figure C.13) allows the user to interactively create 3D aggregates. This editor has the same interface as the scene editor, but it has a somewhat different purpose. The purpose of this editor is to lay out the components of an aggregate, whereas the purpose of the scene editor is to lay out the components of a scene. Figure C.12 illustrates the car shown in Figure C.1 being constructed. The body of the car was created in the polyhedron window (Figure C.9) and then added to this window. The wheels are just cylinders that were created by selecting the cylinder icon located at the top of this window. The cylinders were then rotated, translated, and scaled using the interactive options on the

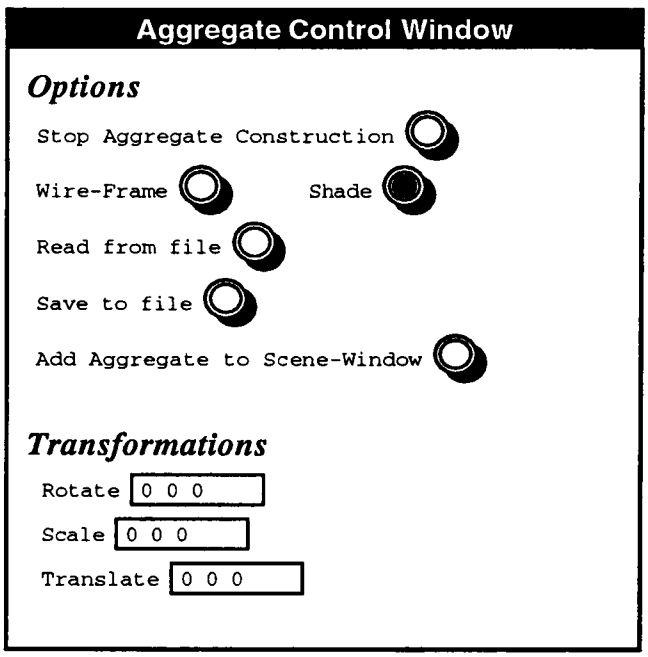


Figure C.13: The aggregate control window. The transformation options in this window apply to the individual components rather than the entire aggregate. The option "Add Aggregate to Scene-Window" creates an instance of the aggregate used in this window and adds it to the scene window. The user can then apply transformations to the entire aggregate once it has been added to the scene window.

left side of this window.

The trees, windmill, and fence shown in the scene window are also aggregate objects created using this window. The trees are composed of a cone on top of a cylinder. The windmill is constructed from four planes, one sphere, and one cylinder. Finally, the fence is constructed by creating four instances of a section of the fence. Each section of the fence (two post and two cross beams) was created from four cubes. More than one instance of an aggregate can be added to the scene window by repeatedly pressing the option "Add Aggregate to Scene-Window."

The aggregate control window is shown in Figure C.13. The options found in this window are similar to those options found in the scene control window with an additional feature "Add Aggregate to Scene-Window" that creates an instance of the 3D aggregate in the aggregate window and adds this aggregate to the scene window. The objects in the aggregate window represent the components of the newly created aggregate in the scene window. After the 3D aggregate has been created in the scene window, any transformations to the objects in the aggregate window are reflected in the scene window. This allows the lay out of aggregates to be altered even after they have been added to the scene window. The scene window only allows alterations to the top-level objects. If objects are deleted from the aggregate window, these same objects will automatically be deleted from any instance of this aggregate that has been added to the scene window. However, if new objects are added to the aggregate window, these objects will not automatically be added to any instance of this aggregate in the scene window. Instead, the user must create a new instance of this aggregate and add it to the scene window by again using the feature "Add Aggregate to Scene-Window".

Appendix D

Software Development Process and Testing

D.1 Introduction

This section discusses the software development process and testing methods that were used in both the 3D toolkit and the extensions to the conventional spreadsheet model. The first two steps of the recommended software development process [Shooman 83] were combined as illustrated in Figure D.1. Hence, we will use the term "specifications" to refer to both the requirements and the specifications.

D.2 Specifications and Design

Specifications were written for all elements of the programming layer of the 3D toolkit. This included specifications for the seven interactors, the 3D objects and aggregates, the integration of constraints, the shading model, etc. The specifications consisted of combinations of the following: algorithms (including proofs of correctness), grammars, equations, and English descriptions that described what was to be done and how it was to be accomplished. As an example, a partial specification of a 3D polyhedron object is shown in Figure D.2. Specifications were not as formally written for the direct manipulation layer of the toolkit.

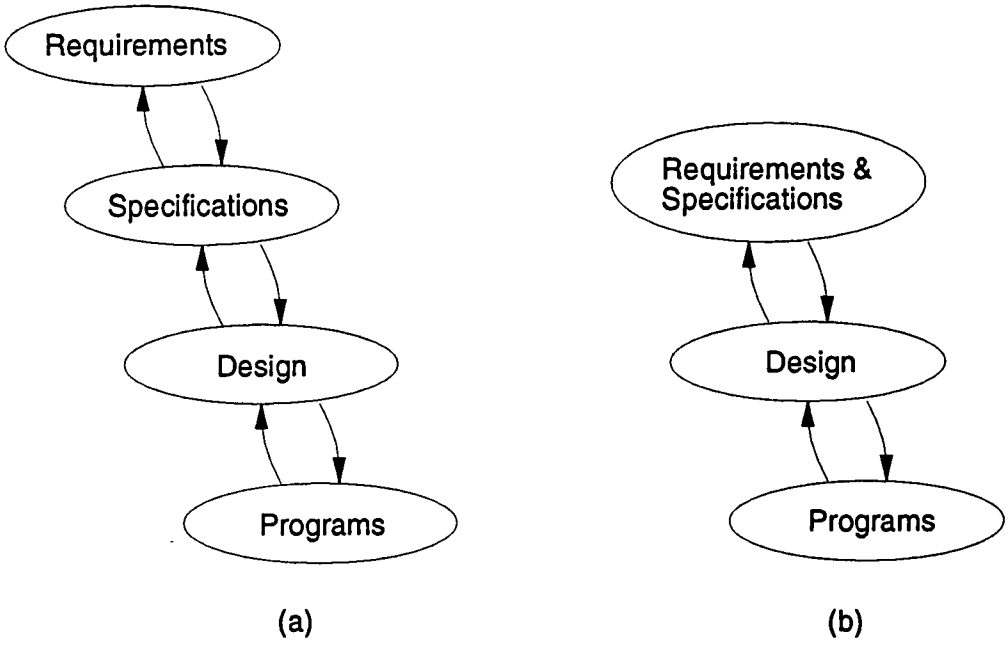


Figure D.1: A diagram illustrating the recommended development process (a) described in Shooman [Shooman 83]. The software development process that we used (b).

```
          points =  (point*)
           point =  ([label] x y z)
           sides =  (side*)
            side =  ([label] point-list [filling-style] [line-style])
      point-list =  (point-indicator*)
point-indicator =  integer  |  label
```

The parenthesis indicate a list.

Figure D.2: The specification of a 3D polyhedron included this grammar that was used to describe the acceptable values for the *points* and *sides* of a polyhedron.

However, formal specifications were written for the multi-output, structural, and multi-way constraints.

All the design documentation was written in a pseudo-code notation using a top-down strategy. Examples of the design representation are shown in sections 4.2 - 4.4 on constraint solving. The specifications and design were verified using walk throughs.

D.3 Testing

The code was tested using a combination of the following four methods:

- Code reading - this included hand execution and desk checking of many of the more difficult routines.
- Machine testing - many of the modules were tested to make sure that for some given input, valid output was computed.
- Classroom testing - the 3D toolkit was used and tested in an upper level undergraduate computer science course in graphics at the University of Tennessee.
- Regression tests - after each modification to the constraint algorithms over 100 regression tests were performed on the extended constraints.

VITA

William Joseph Rosener was born on August 2, 1964 and raised in rural Vail, Iowa. After graduating from high school in 1983, he attended Iowa State University where he received a Bachelor of Science degree in Computer Science with a minor in Speech Communication in 1987. Mr. Rosener entered the Graduate School of the University of Tennessee, Knoxville as a graduate teaching assistant in September 1987. He received the Master of Science degree in Computer Science in May 1990, and his Doctor of Philosophy in August 1994.