A Programming Library for the Construction of 3-D Widgets

by

TONY TAT CHUNG LAU
B.A.Sc., University of British Columbia, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April, 1994

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

Department of Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, B.C.
Canada V6T 1Z4

Date: April 29, 1994.

Abstract

3-D graphical user interfaces (3-D GUIs) may be beneficial to application programs that need to manipulate 3-D objects or multi-dimensional data. However, most existing 3-D graphics programming systems do not provide primitives for building 3-D GUIs; instead, programmers have to deal directly with input device events and 3-D graphics. Systems that do either are research systems that are not available to application programmers or are difficult to extend. For these reasons, explorations in the use of 3-D GUIs have been difficult.

An extensible and object-oriented 3-D Widget Programming Library is implemented. It is an extension to Inventor (a widely available 3-D programming library) and lets programmers contruct new *widgets* (3-D scene objects with interactive behaviors) using four types of high-level components that are responsible for user interface, visual feedback, application interface and general computation. A widget built with this library is able to control and display one or more application states, interact with users in a click-drag-release fashion, and convey the application states through the relative positions and orientations among the widget's parts. A widget interfaces with an application program through either direct attachments to scene objects or callback functions.

Contents

List of Tables

List of Figures

Acknowledgements

There are many people without which this research would not have been accomplished. I would like to thank:

- Dr. David Forsey of UBC (University of British Columbia), my supervisor, for suggesting the thesis topic, sharing his insights, giving guidance and encouragement along the way, and bearing with all the drafts he had to read;
- Dr. John Dill of SFU (Simon Fraser University), the second reader, for giving valuable comments on the final draft of the thesis;
- Mr. Tien Truong, the student reader, for going through several drafts of the thesis and giving valuable suggestions on both the content and the writing;
- Mr. Raza Khan and Mr. Chris Healey, for sharing their knowledge on formatting the thesis;
- Mr. James Harrison, for asking difficult questions during the design stage of the work;
- Mr. Paul Lalonde and Mr. Bob Lewis, for sharing their knowledge on data-flow systems;
- Mr. Raza Khan (again) and Mr. Vishwa Ranjan, for being supportive and helpful friends;
- last but not least, my parents and sisters, for their encouragement and support.

This research was supported in part by post-graduate scholarships from NSERC (the Natural Sciences and Engineering Research Council of Canada) and UBC.

Chapter 1

Introduction

1.1 Motivation

Recently there has been a growing interest in three-dimensional graphical user interfaces (3-D GUIs) that allow users to visualize and directly manipulate objects in a computer-generated three-dimensional environment on a 2-D screen.

A number of applications — such as computer aided design, computer animation, scientific data visualization and virtual reality — may benefit from 3-D GUIs because these applications deal with objects or data with three or more dimensions. There is also some investigation in using 3-D GUIs for visualization and manipulation of data that is not inherently 3-D — such as the use of a cone tree in [10] to represent the hierarchy of an organization — in the hope of maximizing the utility of the finite screen space and to shift some cognitive load to the human perceptual system.

However, the creation of a 3-D GUI is a difficult task. The following is some of the reasons:

- Unlike 2-D GUI designers who can base their designs on now-common metaphors such as windows, menus, sliders, dials and buttons, 3-D GUI designers do not have a well-established repertoire of interaction techniques to draw from.

- 3-D GUI designers have to worry about complications that do not exist in 2-D GUIs. Examples are object occlusion and view projection.
- There is a mismatch between 3-D GUIs and common input devices such as the mouse. A 3-D GUI has more degrees of freedom than these devices specify.
- There are a number of 2-D GUI programming libraries available; e.g. Garnet[8], InterViews[6] and Forms[9]. However, only a few 3-D graphics programming libraries provide support for 3-D GUIs; e.g. Inventor[15] and UGA[16]. Other 3-D graphics libraries such as GL[11] and HOOPS[4] leave the task of providing 3-D user interaction up to the programmer. This involves handling input device events and calculating the transformations that map the events from the 2-D screen space to the 3-D space of the object being manipulated.

The 3-D Widget Programming Library developed in this work alleviates some of the above difficulties. By encapsulating details of 3-D graphics and interaction techniques into high-level components, the library simplifies the creation of interactive 3-D scene objects called *widgets*, the building blocks of 3-D GUIs. Programmers can then devote more time on the design of the GUIs and on the exploration of alternatives through prototyping, rather than on implementation details. The library is an extension to Inventor[15], a widely available 3-D graphics programming library.

In the following sections, the merits and deficiencies of some GUI programming systems are studied. Then the goals for the 3-D Widget Programming Library are listed. Finally, an overview of the thesis is given.

1.2 Related Works

It is helpful to study 2-D GUI programming libraries because they provide useful models for building 3-D GUI programming libraries. The Forms Library[9] is chosen to illustrate the structure and facilities of a typical 2-D GUI programming toolkit.

Apparently, there are only two 3-D graphics programming systems that support the creation of high-level 3-D interaction objects: UGA[16] from the Brown University and Inventor[15] from Silicon Graphics Inc. The facilities and deficiencies, in terms of 3-D GUI support, of both systems are discussed.

1.2.1 The Forms Library

The Forms library[9] is a programming library for the construction of 2D graphical user interfaces on Silicon Graphics workstations. A *form* (a visual panel for interaction) is composed of high-level objects. There are several general types of predefined objects:

- *Static objects* are not interactive and are used for visual effects or presenting data. They include boxes, text, bitmaps, clocks and charts.
- *Buttons* are pushed by the mouse. There are several types of buttons with different behaviors and appearances.
- *Valuators* let a user set a value between some fixed bounds by dragging the mouse. Examples are sliders and dials.
- *Input objects* allow users to input text or numbers with the keyboard.
- *Choice objects* let a user choose from a set of possibilities. Examples are menus and browsers.

There are functions for adding, deleting, hiding, showing, activating, deactivating, and grouping[1] objects in a form. There are similar functions for manipulating forms. Other functions modify an object's attributes such as its color and its label.

An application program obtains information about the status of form objects by calling appropriate functions that poll or wait for an object that changes its state. A more elegant

[1]Grouping lets an operation be applied to all objects within the group. It also makes certain interactive behaviors, such as radio buttons for mutually exclusive choices, possible.

way[2] is to register callback functions to individual objects. A callback function is called when the object to which the function is registered changes state.

New interaction techniques are added to an application program in three ways: by implementing a window that directly interacts with the user, by adding a *free object* to the application, or by creating a new object in the Forms Library. A *free object* is different from other predefined objects in that the application handles the drawing and the interaction for it. If the new interaction technique is specific to a particular application, then a free object should be used. If the interaction technique is useful in many applications, then a new object should be implemented. New objects are implemented by conventional programming in C. The programmer needs to implement functions for drawing the objects and handling interaction.

The Forms Library comes with an interactive forms design tool. With this tool, a GUI designer composes forms visually with predefined objects. Some operations for composition include addition, deletion, sizing, positioning and grouping. Attributes of an object such as its style, color, label and callback function are set by filling the object's *attribute form.*

The user of the forms design tool tests the forms in a special mode in which the forms behave as they would in an application program. The tool indicates the objects manipulated and the callback functions called while the user is testing the form.

The tool creates description files and C source files when forms are saved. The C source files contain functions that build the forms as designed, and are subsequently incorporated into the application program. Description files can be reloaded into the forms design tool for editing.

1.2.2 Three-Dimensional Widgets

The Brown Graphics Group has published several papers on 3D widgets[2][14][17]. 3D widgets and application programs are constructed with the Unified Graphics Architecture (UGA) system[16].

[2] As long as the application program is single-threaded.

UGA is an object-oriented system in which a new object is created via *delegation* (i.e. the new object is a clone of an existing object and shares the attributes of the existing *parent* object. However, the new object can override the attributes. Changes made to attributes in the parent object are also made to the new object if those attributes are not overridden). Objects in UGA may have geometric, algorithmic or interactive properties. UGA provides a rich set of modeling primitives and operations. One-way constraints, called *dependencies*, allow users to describe relationships between objects. Multi-way constraints and cyclical constraints are made possible via *controllers*, whose purpose is to control other objects. Since 3D widgets are first class objects in UGA, they take advantage of the modeling primitives and operations in constructing their geometric components, and make use of dependencies and controllers for describing their behaviors.

The 3D widget framework in [2] used controllers to construct relationships between application objects and 3D widgets. A new dialog model, based on a modified model of the augmented transition network (ATN), was used for describing the state transitions in a user interface caused by user interactions with 3D widgets. This ATN model allowed disconnected components and more than one active state in a state graph. These properties facilitated the separation of sub-parts in a user interface. [14] explored 3D widget design issues through the task of deforming geometric objects. This paper showed how a rack widget displayed parameters of the deformation as well as allowed users to control those parameters intuitively via direct manipulation of the widget.

A 3D widget construction toolkit was presented in [17]. The toolkit allowed interactive construction of new 3D widgets from a set of primitives via the *linking* operation. The shape of a primitive suggests its purpose, while its *port*s encapsulate constraint values and interactive techniques. By linking the ports of a primitive to the ports of other primitives, constraints among primitives are formed. The authors chose a "coordinate system" metaphor as the basis for the primitives. The primitives cover the concepts of position, orientation, measure, and 2D and 3D Cartesian coordinate systems with the *point*, *ray*, *length*, *angle*, *plane* and *space* primitives. Programmers extend the set of primitives by building *black box*es. A *black box* may

have arbitrary behaviors and an arbitrary number of ports.

While the 3-D widget framework and the interactive widget construction toolkit are impressive, there is one major problem: they are integral parts of UGA, a research development environment. Therefore, programmers who want to develop 3D interactive applications for other environments cannot take advantage of the framework nor the toolkit.

1.2.3 Inventor

IRIS Inventor[15][12][13] is an object-oriented programming library for 3-D graphics. It contains a library of objects used for building 3-D scenes and user interfaces. Inventor objects may be classified into the following types:

- primitive scene nodes such as cameras, lights, shapes, properties and grouping nodes.
- smart scene nodes capable of handling events. These include the selection node and manipulators.
- components such as the color editor and the examiner viewer.

New objects are added to the library by subclassing existing object classes.

A 3-D scene is described by a *scene graph*, a directed acyclic graph of scene nodes. Various *actions*, such as rendering, picking and searching, can be applied to scene graphs or individual scene nodes.

3-D user interaction is provided through the selection mechanism, event handlers and manipulators. Manipulators are scene objects that are directly manipulated with the mouse. Each manipulator consists of a 4x4 matrix called the *delta matrix* that represents the state of the manipulator. Manipulations performed on a manipulator cause its delta matrix to be edited. When a manipulator is attached to a scene node, usually a transformation node, modifications made to the delta matrix trigger updates to the attached node. Manipulators for common 3-D interaction techniques, such as the virtual trackball, are provided.

Not only can a manipulator affect the scene node it is attached to, an application program can also register a callback function with the manipulator so that the application program is notified of changes made to the delta matrix of the manipulator.

The default appearance of a manipulator can be overridden during instantiation without reprogramming. Some parts of the manipulator can be made invisible or non-functional in the same way. A new manipulator class is created in two ways: by combining existing manipulator classes if they support the interactive behaviors required by the new manipulator class, or by subclassing the manipulator base class and programming in C++ if new behaviors are required.

However, the ability to create new powerful manipulators is restricted[3] because:

- Each manipulator controls at most one scene node if no callback function is used. Furthermore, the control of a manipulator over the attached node is hard-coded in Inventor and cannot be altered by the programmer.
- A manipulator is restricted to represent its state with a single 4x4 matrix, the delta matrix; therefore, a manipulator cannot manipulate or display several application data items.
- There is no notion of motion hierarchy within a manipulator; i.e. there is no facility in the manipulator base class that assists a programmer in building manipulators with internal moving parts. This is true even when the manipulator is constructed from several existing manipulators.
- Creating manipulators with new behaviors is difficult because the programmer has to implement member functions to interpret input device events and update the delta matrix. The programmer may also need to update the geometry of the manipulator for visual feedback. Moreover, the new behavior is not easily shared with other new manipulators because there is no notion of reusable components in the manipulator base class.

[3]The recently released Inventor 2.0 alleviates some of these restrictions. Please refer to Appendix E for more details.

1.3 Objectives

Due to the difficulties in the creation of 3-D GUIs, and the lack of powerful and available tools for doing so, the goal of this research is to produce a programming library for the creation of 3-D widgets, the building blocks of 3-D GUIs, for Inventor-based application programs. A widget encapsulates geometries and behaviors for displaying and controlling application data.

This programming library is designed to meet the following goals:

- New widget types should be easy to implement. This research uses the approach of building new widgets from a collection of reusable high-level components handling user interaction, visual feedback, application interfacing and computation.
- The library of components should be extensible and object-oriented.
- Widgets produced should be compatible with a widely available 3-D graphics programming library; as a result, this programming library is designed as an extension to the Inventor programming library mentioned in Section 1.2.3.
- Widgets should have a standard interface to application objects. However, programmers should be able to bypass the standard method when greater control over the usage of the widget is required.
- A widget should be able to display and control one or more application states.

1.4 Thesis Organization

Chapter 2 gives an overview on the features of the programming library and the structure of a widget. Then, the mechanics of a widget are explained. Chapter 3 concentrates on the design and implementation issues brought forward by the overall goals of the system. Chapter 4 discusses the system in terms of its successes or failures in meeting the goals. It also looks into possible future work.

Chapter 2

3-D Widget Programming Library — an Overview

2.1 What is a 3-D Widget?

3-D widgets are special objects that share the scene with other 3-D objects in an application program. They serve one or more of the following purposes:

- provide visual feedback on application program data.
- allow users to modify application program data through direct manipulation of the widgets that represent those data items.

An example of a 3-D widget is the rack widget described in [14]. One version of it, implemented with the 3-D Widget Programming Library, is shown in Figures 2.3 to 2.6. The orientations of the twist dial and bend dial convey the amount of twist and bend applied to the object, while the height of the taper slider and the position of the taper offset slider show the amount and the scope of the taper. The user modifies the four parameters by clicking the appropriate part of the widget with the mouse, dragging the part to the desired position, then releasing the mouse button. During the manipulation, both the shape of the object and the shape of the widget change interactively to reflect the current state of the application program.

2.2 What is the Library?

The 3-D Widget Programming Library is an object-oriented programming library for the creation of new 3-D widget classes. The programming library is an extension to IRIS Inventor, a commercially available 3-D graphics programming library; therefore, 3-D widgets are compatible with application programs developed with IRIS Inventor.

The library consists of the widget base class `SoWidget` and a collection of high-level component classes from which new widget classes are constructed.

The widget base class `SoWidget` is derived from `SoDragManip`, the base class of all existing Inventor manipulator classes. The `SoWidget` class allows a programmer to construct new widget classes from high-level components.

The four types of high-level components, called *parts*, *spaces*, *slots* and *general components*, are responsible for geometry and behavior, motion hierarchy, interface to application programs, and general computation respectively. Components pass data from one to another via connections between their *ports*. Thus, the behavior of a widget is described by its components, the connections among the components, and the motion hierarchy within the widget.

The object-oriented property of the library makes it easily extensible; new component classes are added by subclassing appropriate base classes and programming in C++.

Each 3-D widget built with this library has the following properties:

- User interaction is through the mouse, in the click-drag-release fashion.
- Meta-keys (i.e. the SHIFT, CTRL and ALT keys) may be utilized to alter the response to user manipulation.
- The geometries of widget parts, and the relative positions (or orientations) among them in a motion hierarchy, form the visual feedback for a widget.
- A widget controls and displays one or more Inventor nodes/fields independently through

attachments. An attachment defines the relationship between a widget slot and an Inventor node/field. The relationship is modifiable by the application programmer.

- An application program enables reactions to changes in a widget by registering callback functions with widget slots.

2.3 The Anatomy of a Widget

A widget is built from high-level components. This section first explains the functionality of widget components, especially the specialized types including *parts*, *spaces* and *slots*. Then, the structure of a widget is revealed through a typical creation process of a new widget class.

Figure 2.1 shows the overall structure of a widget (in this case the rack widget). Figure 2.2 gives a more detailed view of the relationships among widget components in the widget. These figures are discussed in detail in Section 2.4.

2.3.1 Widget Components

Widgets are made up of high-level components. There are four types of components:

- A *part* incorporates some geometry and zero or more interactive behaviors.
- A *space* represents a coordinate frame. Each space contains a 4x4 matrix describing the transformation from this space to its parent. Spaces are mainly used for building the motion hierarchy within a widget.
- A *slot* contains a piece of data that the widget displays or controls.
- *General components*, unlike the three types above, do not perform specific tasks in a widget. General components are used for calculation, data type conversion, or the control of other components. Appendix C describes how to implement a general component.

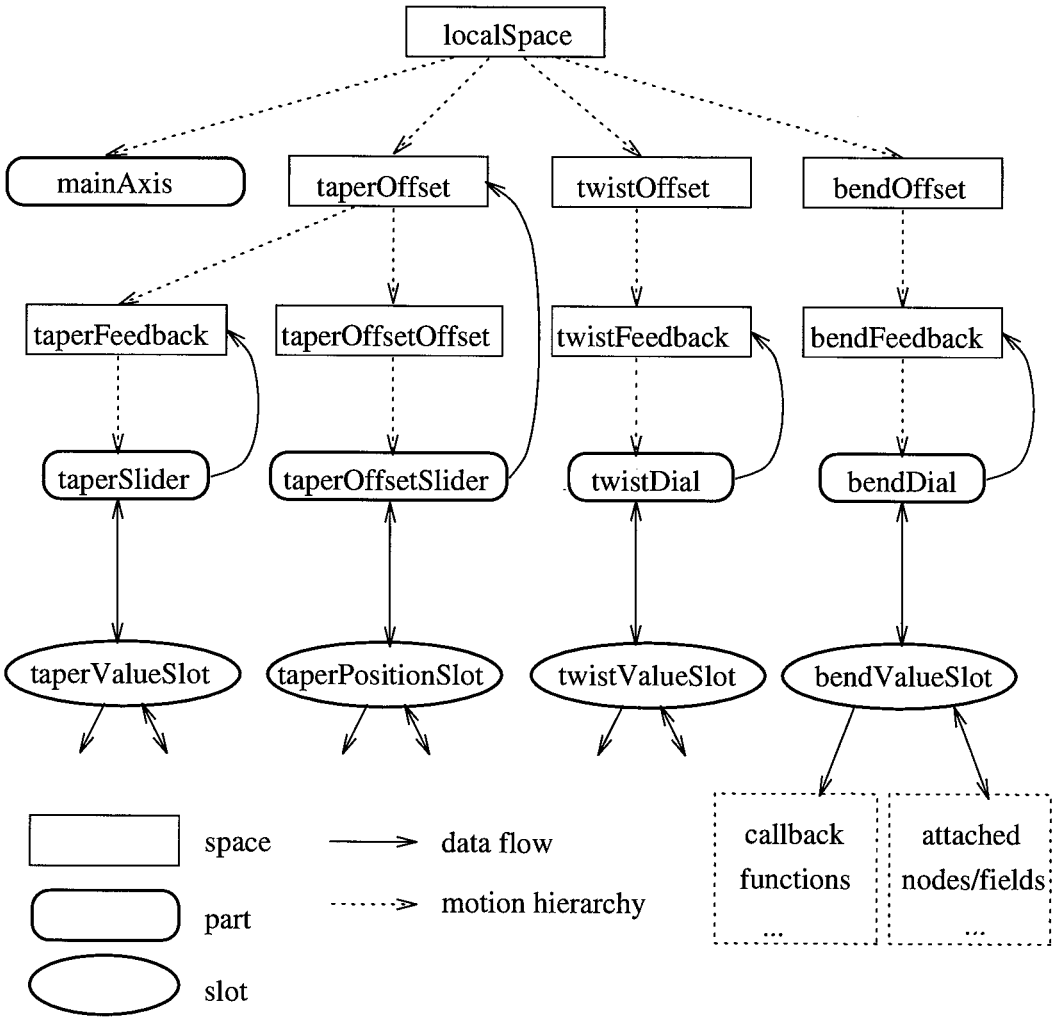

Figure 2.1: The structure of the Rack Widget

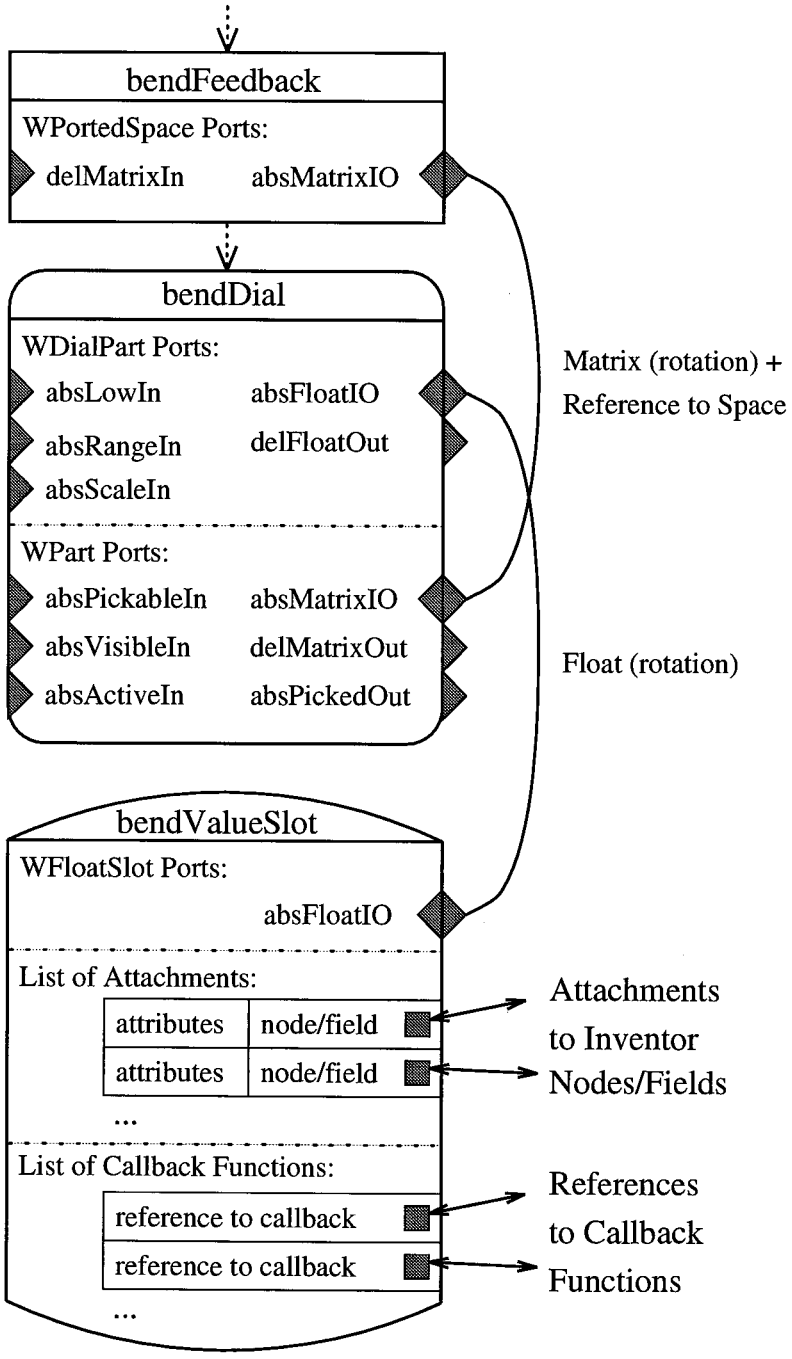

Figure 2.2: A detailed view of the last figure, showing internal structures of bendFeedback, bendDial and bendValueSlot as well as port connections, attachments and callback functions

A component communicates with other components via connections between their *ports* that allow data to flow into or out of components, or both. Data that flows between components is one of the following types: boolean, integer, float, 3-D float vector, or 4x4 float matrix. A port handles either *absolute* or *delta* (relative) data. Absolute data reflects the true data value, while delta data reflects the change in the data value from the last update.

Two ports can be connected only if the followings are satisfied: they are of the same data type, they both handle absolute or delta data, and one port is input-capable while the other is output-capable.

Each component class must implement a member function called `invoke` that is called when one of the input data ports is modified. The function `invoke` is responsible for updating the state of the component and its output ports. Moreover, parts and slots update their states and output ports upon recieving events such as user input and changes in Inventor nodes/fields.

In the following sections, the three specialized types of components (i.e. parts, spaces and slots) are described in more detail.

Parts

A *part* consists of a shape, or geometry, that should suggest its usage (e.g. the bend dial shown in Figure 2.6). Most parts also encapsulate one or more interactive behaviors; i.e. when a user clicks on a part in a widget and drags it, the part modifies its own state and updates its output ports based on the manipulation. Parts that do not have interactive behaviors are used either for displaying application data or to give the widget the "right look" (e.g. the main axis part shown in Figure 2.6). A typical widget includes one or more parts, arranged in a motion hierarchy related by *spaces* (described in the following section). Parts and spaces are arranged in a tree-like structure in which each part has a *parent space.*

A widget displays data through the relative positions and orientations of its parts' geometries. Each part owns two ports that output an absolute transformation and a delta transfor-

mation to reflect its internal state. Usually, one of these ports is connected to a *feedback space* that should be an ancestor of the part in the motion hierarchy. The transformation applied to the feedback space induces movements of part geometries under the feedback space. This serves as the visual feedback for the state of the part.

Individual part classes support different sets of ports, depending on the designs of the part classes. These ports usually allow other components to access the internal state of the part through more useful parameters than the transformation matrix. For example, an instance of the dial class `WDialPart` consists of a port with a float value that represents the the amount of rotation. Some of these ports allow other components to modify the internal state of the part. The `invoke` function of the part, when triggered by changes in the input ports of the part, is responsible for updating the internal state of the part, the feedback transformation ports and any other output ports it has.

Adding new part classes to the library generally involves subclassing the `WPart` base class, designing the geometries, deciding on the types and number of ports, and implementing a few essential member functions that determine its interactive behaviors and reaction to changes in input ports.

Appendix D describes how to implement a new part.

Spaces

Spaces represent space transformations. There are two major categories of spaces: *standard spaces* and *custom spaces*. *Standard spaces* include the *world space*, the *local space* and the *edit space*; each widget has one of each of the above. The *world space* represents the global space that is not transformed in any way. The *local space* is the coordinate frame in which the widget operates; it is the root of the widget's motion hierarchy. The *edit space* represents the *delta matrix*, the only externally accessible state of an Inventor manipulator, inherited by the widget base class `SoWidget` from the manipulator base class `SoDragManip`. The delta

matrix is typically *attached* to an Inventor node (typically a transformation node) so that the manipulator can control the node through making changes to the delta matrix, and vice versa. In a widget, the delta matrix is modified through the delta input matrix port of the edit space. Please refer to Chapter 13 of the "IRIS Inventor Programming Guide - Volume I" [12] for information about the delta matrix.

Custom spaces are the building blocks of a widget's motion hierarchy. Each custom space contains a transformation matrix that is modified through either of the two matrix ports (a delta input port and an absolute input/output port). To add a custom space to a widget, its *parent space*, which is either an existing custom space or the local space of the widget, must be specified. A widget may have as many custom spaces as required. Figure 2.1 shows the motion hierarchy of the rack widget, with the `bendOffset` space being the parent space of `bendFeedback` space, for example.

Member functions are provided by all spaces for obtaining the matrices describing the transformations between the spaces and the world space. These matrices are used internally by widget components for performing conversions between spaces. A vector or matrix data item (in a data port or slot) carries with it a reference to the space under which the data resides; therefore, the recipient of the data item can perform a space transformation on the data if necessary.

Slots

Slots are "terminals" through which a widget communicates with the application program. Each slot contains a value that the widget controls or displays. Five slot classes support the same data types as data ports of widget components. In fact, each slot of a particular data type contains an absolute input/output port of the same data type.

A slot communicates internally with other components via its data port. The application program accesses a slot either by creating *attachments* between the slot and Inventor

nodes/fields, or registering *callback functions* with the slot.

An *attachment* between a slot and an Inventor node/field is responsible for updating the slot data or the node/field based on changes to the other side of the attachment. The data flow can be one-way or both-ways. The update to either side of the attachment is performed by the *data conversion function* registered with the attachment. The fact that the data conversion function is NOT hard-coded to the attachment means an application program can use customized conversion functions, instead of the default conversion functions (one for each data type). Appendix A shows an example of a customized conversion function.

A *callback function* is used when the application object affected by the widget is not an Inventor node/field, or the application program wants to access several slots at the same time. A callback function is called when the data of the slot it is registered to changes.

Each slot supplies two member functions for accessing its data: `getData` and `setData`. For vector and matrix slots, these two functions transmit space information as well as data. These functions are mainly used in callback functions and data conversion functions.

2.3.2 Creation of a New Widget Class

The functionality of a widget is broken down into the following aspects:

- the interactive behavior that is defined by the interactive behaviors of the parts.
- the visual feedback that is determined by the geometries of the parts, the motion hierarchy formed by the parts and the spaces, and the connections between the parts and their feedback spaces.
- the behavior and interface to the application program that is defined by the slots and the connections between widget components (i.e. slots, parts, spaces and general components).

The widget base class, `SoWidget`, provides member functions that allow a widget programmer to:

- create a new space in a widget as the child of an existing space.
- add a new part to a widget as a child of an existing space and specify an existing space as its feedback space.
- add a new slot to a widget.
- add a general component to a widget.

The data port base class, `DFPort`, provides a member function for making a connection between two compatible ports.

Together, the above-mentioned member functions and the widget components fully specify the functions of a new widget. Typically, the components are created, added to the widget, and connected in the constructor of the new widget class. The widget keeps track of the components added to it, and deletes them when the widget is destroyed.

2.4 How Does a Widget Work?

The internal mechanism of a widget is probably best illustrated with an example. A rack widget class, as described in section 2.1, is created with existing components in the 3-D Widget Programming Library. The overall structure of the widget is illustrated in Figure 2.1. In Figure 2.2, the relationships among widget components in the widget is shown in more detail. Figure 2.3 to Figure 2.6 show the rack widget in action.

Both `twistDial` and `bendDial` are instances of `WDialPart`. The feedback matrix of a dial part is a rotation around the Z axis of the part's space. A dial part also has a port that inputs and outputs a float value proportional to the amount of rotation. There are other input ports for controlling the scaling factor and the range of the rotation, as shown in Figure 2.2.

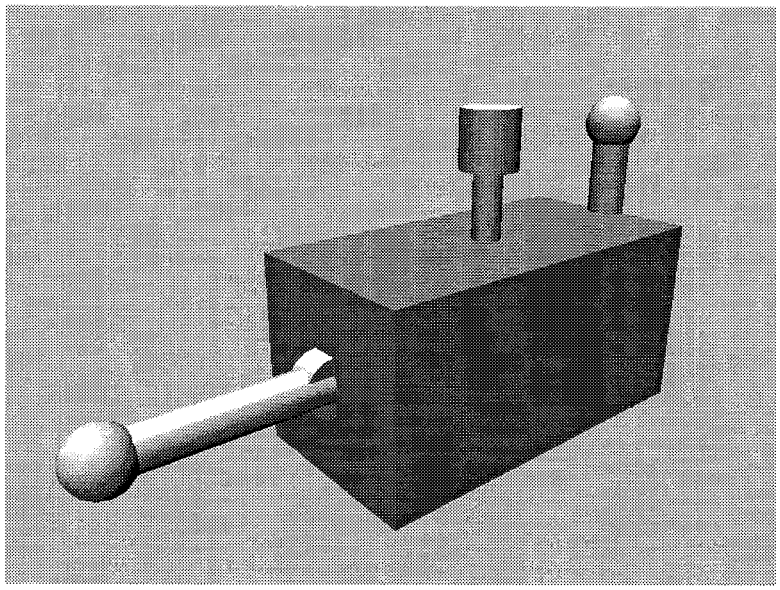

Figure 2.3: The Rack Widget in its default state

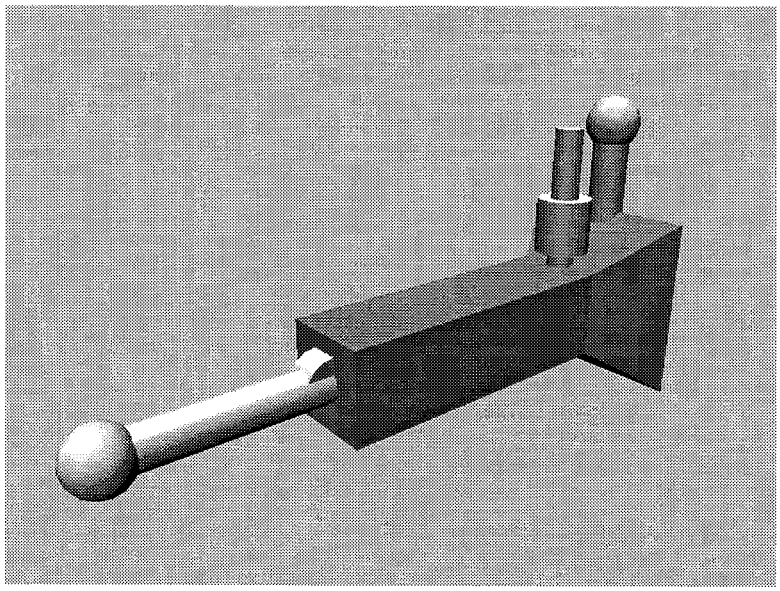

Figure 2.4: Manipulations of the taper and taper offset sliders

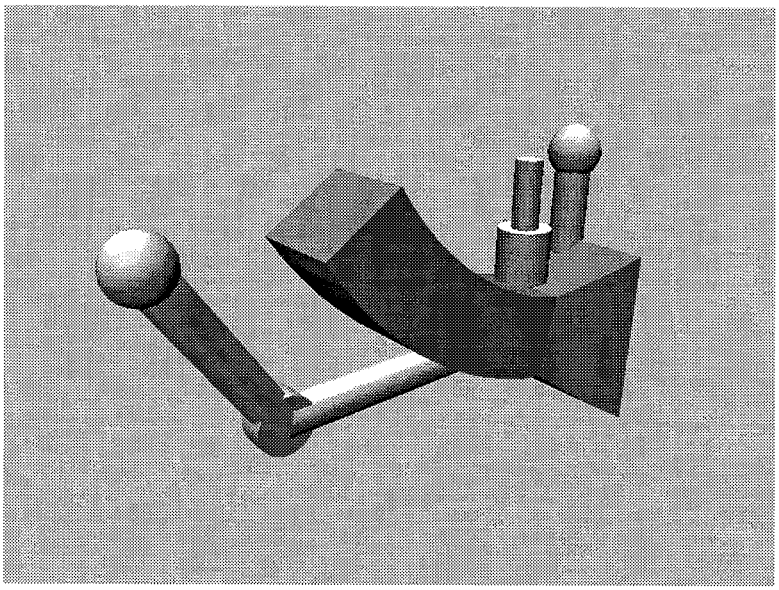

Figure 2.5: The manipulation of the bend dial

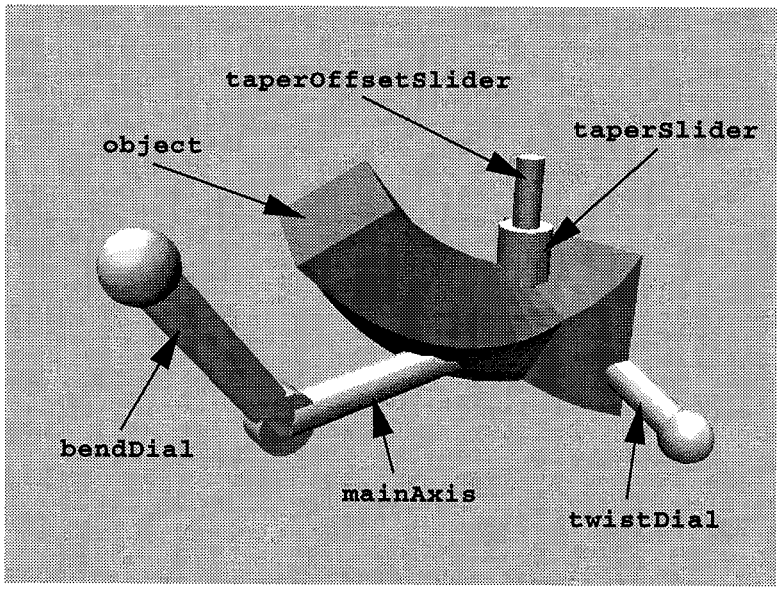

Figure 2.6: The manipulation of the twist dial

Both `taperSlider` and `taperOffsetSlider` are instances of `WSliderPart`. The feedback matrix of a slider part is a translation along the Y axis of the part's space. A slider communicates the value of the slider through a float port. Like a dial part, a slider part also has input ports for controlling the scaling factor and the range of translation.

`mainAxis` is an instance of the `WGeomPart` that has no interactive behavior. Here it provides the geometry for the main axis of the widget.

The motion hierarchy of the rack widget, shown in Figure 2.1, is quite straight forward. The `twistOffset` space and `bendOffset` space place `twistDial` and `bendDial` in the desired positions and orientations. The parents of the two dials, `twistFeedback` and `bendFeedback`, are also the feedback spaces for the two parts.

The set-up for `taperSlider` is slightly more complex. `taperOffset`, the feedback space for the `taperOffsetSlider`, determines the position of `taperSlider` and `taperOffsetSlider` on the main axis. `taperOffsetOffset`, the parent space of `taperOffsetSlider`, orients `taperOffsetSlider` so that it slides on the Z axis in the widget's local space. `taperFeedback`, the parent space and feedback space of `taperSlider`, determines the position of `taperSlider` on `taperOffsetSlider`.

The data ports of the parts are connected bidirectionally to the ports of the appropriate slots. Application programs thus have access and control of all four parameters of deformation.

The actual implementation of the rack widget is listed in Appendix B.

2.4.1 Using the Widget

To incorporate the rack widget into an application program, a programmer needs to include an instance of the rack widget in the scene-graph for the program. The procedure is basically the same as including an Inventor manipulator in the program, except that a widget also controls and displays application states through its slots in addition to the delta matrix inherited from the manipulator base class `SoDragManip`.

Appendix A goes through an example of the addition of the widget to an application program and the use of a callback function. It also describes how to use attachments and how to customize widgets.

2.4.2 A Typical User Interaction

The rack widget illustrates what happens to a widget when it is manipulated. For example, when a user wants to bend an object using the rack widget, he or she would click on the `bendDial`, drag it to the desired orientation, then release the button.

When the user clicks on the `bendDial`:

- the widget determines which part is being picked (in this case it is the `bendDial`). Then it calls the `manipulateStart` member function of the part.
- `manipulateStart` prepares for subsequent manipulation. Typically this involves determining the current view volume and projecting the mouse position to a coordinate in the part space for later use. It also updates a boolean output port (as shown in Figure 2.2) to indicate that this part is picked. For example, `bendDail` projects the mouse position onto the plane that passes through the origin of its space and perpendicular to the Z axis of the space.

When the user drags the mouse pointer:

- the widget calls the `manipulate` member function of the part.
- `manipulate`, like `manipulateStart`, determines the view volume and projects the mouse position to a coordinate in the part space. In addition, it calculates the amount of manipulation based on the difference between the new projection and the last projection. The difference is used to update the internal state of the part, the part's ports, and the feedback matrix ports. For example, `bendDial` calculates the angle sustained by the

new projection and the previous projection and updates the internal value and the ports accordingly.

- Whenever a port is updated, the input-capable ports connected to it are also updated. Each input-capable port, once updated, immediately calls the `invoke` member function of the component that owns it. Receiving ports that are output-capable will propagate the change to input-capable ports connected to them.

- In the rack widget, the data port of the `bendDial` is connected to the `bendValueSlot`. A slot's `invoke` typically updates the internal data of the slot, then updates all the Inventor nodes/fields attached to the slot through the data conversion functions registered in the attachments. Then it calls all the callback functions registered to the slot.

- A space's `invoke` function performs a space conversion on the received matrix before applying it to the space's internal transformation. It updates the absolute feedback matrix port if it receives data from the delta feedback matrix port.

- Inventor automatically redraws the scene when it detects changes in the scene graph. For example, a modification of the transformation in the `bendFeedback` space will trigger a redraw.

When the user releases the mouse button:

- the widget calls the `manipulateFinish` member function of the part.

- `manipulateFinish` typically updates the boolean output port (mentioned above) to indicate that this part is not picked anymore.

Chapter 3

Design and Implementation

3.1 General Design Issues for the 3-D Widget Programming Library

3.1.1 Inventor as the Basis

The IRIS Inventor was chosen as the basis for the 3-D Widget Programming Library for several reasons. First, Inventor is a commercially available programming library; therefore, the library can be utilized in programming projects using Inventor. Second, Inventor is an extensible object-oriented library. Third, the manipulator base classes are a good basis for 3-D widgets because they perform many book-keeping tasks and isolate mundane event handling details. Last, Inventor has many useful basic classes that aid 3-D graphics programming. Some are useful data structures such as vectors and matrices, while others simplify calculations such as projecting a screen coordinate to the 3-D scene space.

3.1.2 Motion Hierarchy as the Means of Visual Feedback

This library supports the use of a motion hierarchy within a widget as the sole mechanism for visual feedback (besides highlighting of parts). The rack widget in [14] shows that the relative

positions and motions among widget parts provide good visual feedback for the values the widget is controlling. The motion of parts during manipulation gives the user a sense of being "in direct control" of the widget.

However, there are cases where the motion hierarchy is not adequate. For example, a motion hierarchy within a widget is useless when the widget needs to track the coordinates of other objects in the scene; e.g, a length widget whose two ends follow the centers of two moving objects. In some cases, other means of visual feedback are more appropriate; e.g. color for a color editor.

Nevertheless, motion hierarchies are reasonably useful for widgets that display or control discrete pieces of data. One more advantage is that the support for motion hierarchy is relatively straight-forward to design and implement. Other types of visual feedback are possible, but are much harder to implement with this library.

3.1.3 High-Level Data-Flow Components as Building Blocks of Widgets

With the 3-D Widget Programming Library, widgets are built with high-level components. The Forms Library[9] is a good example of a GUI programming library that simplifies GUI programming by providing high-level objects. High-level components allow a programmer to concentrate on the functionality of the widget rather than low-level details such as event handling and rendering. Moreover, high-level components, though potentially difficult to implement, can be used in many widgets once they are added to the library.

The data-flow programming model, in which a program is implemented by making connections between data ports of components, is adopted as the programming model for 3-D widgets. This model allows each widget component to be self-contained, with a well-defined interface to other components.

A widget performs the following tasks: interfaces with the application program, handles user interaction, provides visual feedback, and performs general computation. The four classes

of high-level components mentioned in Section 2.3.1 share the above tasks in the following way:

- *Parts* respond to user interaction and help to define the shape of the widget.
- The motion hierarchy, built with *spaces*, provide visual feedback of the widget's states.
- *Slots* are the widget's interface to the application program.
- *General components* perform general computations for the widget.

This classification is reflected in the class hierarchy of the 3-D Widget Programming Library. The class hierarchy, discussed in Section 3.2, allows the addition of new components to the library by subclassing appropriate base classes.

3.1.4 Multiple Controls and Support of Five Data Types

In Inventor, a manipulator interfaces with the application program through its *delta matrix*. There are two problems with this method: first, a manipulator controls or displays only one piece of application data; second, the support of only one data type is restrictive.

In the 3-D Widget Programming Library, a widget displays and controls many data items via the slots owned by the widget. A slot communicates internally with widget components and externally with Inventor nodes/fields and the application program. Slots support five data types: boolean, integer, float, 3-D float vector and 4x4 float matrix. Each vector or matrix data item carries with it a reference to the space (or coordinate frame) under which the data item resides. The support of these data types should be adequate for most applications. These data types are supported for both internal communication among widget components, and external communication with Inventor nodes/fields and the application program. To keep the programming library simple, more advanced data types, such as arbitrary data structures, functions and pointers are currently not supported.

3.2 Design of the Classes

The 3-D Widget Programming Library is structured as a C++ class hierarchy, as shown in Figure 3.1, defining the base classes for widget classes (`SoWidget`) and component classes (`DFNode`). Component classes are further specialized to part classes (`WPart`), space classes (`WSpace`) and slot classes (`WSlot`).

Data ports for widget components are encapsulated in the `DFPort` class and its five descendants. The attachments between widget slots and Inventor nodes/fields are defined by the `WAttach` class and its descendants.

`DLList` and `Element` provide double-linked list capabilities for the library. Lists are used, for example, in `SoWidget` for keeping track of widget components owned by a widget.

The following sections describe the design of the above classes in detail.

3.2.1 The Widget Base Class

The base class for all 3-D widgets is `SoWidget`, a descendent class of the Inventor manipulator base class `SoDragManip` from which it inherits many useful properties:

- As a descendent class of `SoDragManip`, it only needs to provide callback functions for the three stages of interaction with the mouse, i.e. click, drag, and release.
- It has access to useful information such as the mouse position, the picked object, the trigger event, and viewing parameters.
- `SoDragmanip` manages dictionaries of name/geometry pairs. This capability allows the geometries of a 3-D widget to be altered without re-compilation.

The design of widget slots and the attach mechanism (discussed later) is also influenced by the attach mechanism in `SoDragManip`.

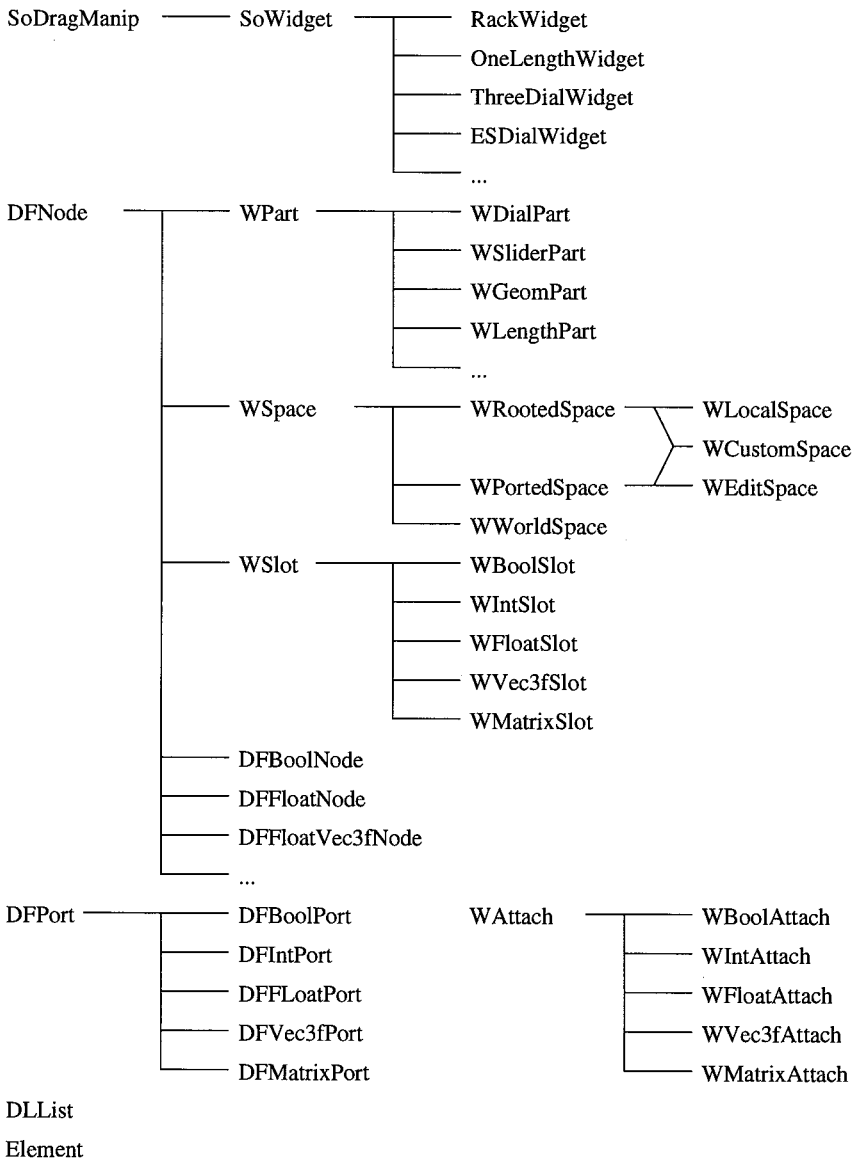

Figure 3.1: The class hierarchy for the 3-D Widget Programming Library

SoWidget provides further abstractions to simplify widget implementation. The different roles of SoWidget are discussed in the following sections.

The Keeper of Widget Components and Geometries

A programmer adds widget components to a widget by calling the appropriate member functions listed below:

```
WPart *addPart( WPart        *part,      // part to be added.
                const SbName &name,      // name of part.
                int          style,      // style of part.
                WRootedSpace *parent,    // parent space.
                WPortedSpace *feedback,  // feedback space.
                SbBool       isAbsolute, // absolute/delta feedback matrix.
                SbDict       *classDict);// dictionary for geometries.

WCustomSpace *createSpace( const SbName &name,    // name of space.
                           WRootedSpace *parent,  // parent space.
                           const SbMatrix &init );// initial matrix value.

WSlot *addSlot( WSlot        *slot,  // slot to be added.
                const SbName &name );// name of slot.

DFNode *addDFNode( DFNode *node );// component to be added.
```

These functions initialize the widget components and incorporate them in the appropriate lists kept by SoWidget. These lists allow SoWidget to delete all the widget components that belong to a widget in its destructor [1]. Except for space components, which can only be instances of the class WCustomSpace, all components have to be created before being added to a widget. Space components are added to the widget simply through the createSpace function listed above.

Each derived class of SoWidget defines a static dictionary called classDictionary to maintain name/geometry pairs for the class, and each instance of the class maintains a local dictionary called userDictionary, which is a member variable of SoWidget. A name/geometry pair

[1]The implication is that a widget component should only have one owner.

contains an Inventor subgraph that is either a part geometry or the visual style of a part. An entry in `userDictionary` takes precedence over an entry of the same name in `classDictionary`. Entries of the dictionaries for a widget are inherited from `SoDragManip` and `SoWidget`, and additional entries can be added to widget classes and individual widget instances. Section A.3 in Appendix A shows how new entries are added to the dictionaries for various levels of widget customization. Sections B.1 and B.4 in Appendix B show how `classDictionary` and `userDictionary` are initialized for a widget.

Some arguments of `addPart` need explanation:

- `style` determines the "style", or visual property, of the part. The integer value `style` is used to search in the widget dictionaries for the Inventor subgraphs, with names "style`<style>`" and "style`<style>`Active", that will become the inactive and active styles for the part.
- `isAbsolute` determines how the part affects the feedback space. More details are given in Section 3.2.4.
- `classDict` should point to the `classDictionary` of the derived widget class. If `addPart` finds an entry in `userDictionary` or `classDict` with a name that matches the `name` argument, it replaces the default geometry of the part with the geometry of the entry. The `classDict` argument is needed because `addPart` is a member function of `SoWidget`; `addPart` has no access to `classDictionary` of the derived widget class.

The Coordinator of Widget Parts

`SoWidget` registers with `SoDragManip` callback functions for the three phases of interaction. These functions are called when `SoDragManip` detects a mouse click on the widget, a mouse drag, or a mouse button release:

```
// Callbacks registered with SoDragManip using:
//   addStartCallback( &SoWidget::startCB );
```

```
//   addMotionCallback( &SoWidget::motionCB );
//   addFinishCallback( &SoWidget::finishCB );
//
static void startCB(  void *, SoDragManip * ); // mouse button click.
static void motionCB( void *, SoDragManip * ); // drag.
static void finishCB( void *, SoDragManip * ); // mouse button release.

// Functions that work for the static callback functions above.
void manipulateStart();  // mouse button click.
void manipulate();       // drag.
void manipulateFinish(); // mouse butoon release.
```

These functions do not handle the interaction themselves, but instead determine which widget part is picked, then call the appropriate member functions of the part to handle each phase of the interaction. Section 3.2.4 discusses those member functions in more detail.

The Provider of Information

Descendants of `SoDragManip`, such as `SoWidget`, have access to useful information about interactions. However, a widget does not handle the interaction itself; it asks the picked widget part to handle the interaction. Due to this arrangement, the base class of all parts, `WPart`, is made a friend[2] of `SoWidget`, and includes the member functions for accessing useful information for handling interactions. Section 3.2.4 describes those functions in more detail.

The Root of the Motion Hierarchy

`SoWidget` is derived from `SoSeparator` (a group node that is allowed to have child nodes), which allows a widget to be the root of an Inventor subgraph. In fact, this property lets a motion hierarchy be built inside[3] a widget by inserting transformation nodes and separator

[2] In C++, a friend of a class, be it a function or another class, has full access to all members variables and functions of the class, including members that are private (only accessible by the class) or protected (only accessible by descendants of the class).

[3] "Under" may be more appropriate if we view the widget as the root of a graph.

nodes at the right places within the widget's subgraph. The addition of space components to a widget does exactly that. Section 3.2.5 looks into space components.

3.2.2 The Basic Widget Component Classes

As mentioned in Section 3.1.3, the programming model for 3-D widgets is the data-flow model; i.e. the behaviors of a widget is determined by its components and the connections among them[4]. Each component is a data-flow node with one or more data ports that define the component's interface to other components. When any input port of a component is updated, the component has to update its internal state and its output ports.

The Widget Component Base Class

In the 3-D Widget Programming Library, `DFNode` is the base class for all widget component classes, including the three specialized base classes: `WSlot`, `WPart` and `WSpace`. To be useful, a widget component class should define one or more pointers to data ports in its class definition. Typically, the data ports of a component are created in the constructor of the component. The component class should also define the `invoke` function, whose duty is to respond to updates in input data ports. `invoke` is declared in `DFNode` as follows:

```
virtual void invoke( void *userData,   // auxiliary data.
                     DFPort *byPort ); // the updated input port.
```

The Data Port Classes

`DFPort` and its five descendants — `DFBoolPort`, `DFIntPort`, `DFFloatPort`, `DFVec3fPort`, and `DFMatrixPort` — define the data ports. Two attributes control the behavior of a data port. The first attribute is the direction of data flow. To a component, a data port is either an input port, an output port, or both. The second attribute is the nature of the data. The data that

[4]Plus the motion hierarchy formed by parts and spaces, strictly speaking.

a port carries can be *absolute* or *delta* (relative)[5]. These attributes are specified in the second argument to the constructor as a bit mask, using the definitions in `DFPortType`:

```
enum DFPortType {
  ABS_PORT = 1, DEL_PORT = 2,
  INP_PORT = 4, OUT_PORT = 8
};
DF<dataType>Port( DFNode *own,    // component owning the port.
                  int port_type );// port attributes bitmask.
                                  // e.g. ABS_PORT | INP_PORT | OUT_PORT
```

A data port communicates with other components by making connections with their ports. Member functions `connect` and `disconnect` of `DFPort` manage connections. They are defined below:

```
SbBool  connect( DFPort *port ); // connect to port.
void disconnect( DFPort *port ); // disconnect from port.
```

`connect` makes sure the two ports are compatible (i.e. one is input-capable and the other output-capable, are of the same data type, and both handle absolute or delta values), then updates the connection lists of both ports. Finally it updates the data in the input port with the data in the output port. If both ports are capable of input AND output, then the port whose `connect` is called is updated by the port specified in the argument.

Port classes provide the following functions for accessing their data:

```
void       setData( <dataType> d );// set data (used by the port's owner).
void       setData( DFPort *p );   // set data (used by connected ports).
<dataType> getData();              // return data (used by the port's owner).
```

`setData( <dataType> d )` is called by the owner of the port. It is responsible for updating the port and all the input ports connected to the port by calling `setData( DFPort *p )` of the input ports. It DOES NOT call the `invoke` function of the port's owner.

[5]The data represents the true value or an increment of the value from the last update, respectively.

`setData( DFPort *p )` of a receiving port is called by the sending port `p` to update the receiving port with the data in `p`. The function calls `invoke` of the receiving port's owner. If the receiving port is output-capable, then it also updates the input-capable ports (except `p`) connected to the port.

`getData` is used by the port's owner to obtain the data stored in the port. If the data is a delta value, then `getData` resets the data to the identity value so that subsequent calls to `getData` will return the identity value. The identity values are set by the member function `setIden` (except for boolean ports, which have no identity values):

```
void setIden( <dataType> d );// set the identity value.
```

Since `DFVec3fPort` and `DFMatrixPort` carry information about space, their `setData` and `getData` are slightly different:

```
void       setData( <dataType> d,
                    WSpace *s );  // space from which data comes from.
void       setData( DFPort *p );  // same as other ports.
<dataType> getData( WSpace *&s ); // space from which data comes from.
```

3.2.3 The Widget Slot Classes

The relationship between slots and widgets is similar to that between ports and components; the slots of a widget define its interface to the application program. Slots are specialized widget components that communicate data to and from the application program for the widget.

The application program interacts with a slot either by registering a callback function with the slot or by attaching Inventor nodes/fields to the slot. The slot also interacts with widget components through an absolute bidirectional data port of the same data type as the slot itself.

Both callback functions and attachments access the data in a slot with the following slot member functions:

```
void  setData( <dataType> dat, // data.
```

```
                void *src);      // source of data. e.g. Inventor node/field.
<dataType> getData();            // return the data.
```

`setData` updates the slot's data and the data port, then processes all attachments and callback functions (except the source of data, if specified). `getData` simply returns the data stored in the slot. For `WVec3fSlot` and `WMatrixSlot`, the functions have one more argument for space information. In addition, the programmer must specify the space under which the slot should store the data:

```
W<dataType>Slot( WSpace *spa );// constructor that sets the space.
void setSpace( WSpace *spa );  // set the space for the slot.
                               // NULL means data is used as is with
                               // no conversion.
void  setData( <dataType> dat, // data.
               WSpace *spa,    // space from which data comes from.
               void *src );    // source of data.
void  setData( <dataType> dat,
               SoPath *pat,     // path which defines a space transformation.
               void *src );

<dataType> getData( WSpace *&spa );
<dataType> getData( SoPath *pat );
```

Here, `getData` returns the data transformed to the space specified in the argument. If the argument is `(WSpace *) NULL`, `getData` returns the data and the space as stored in the slot. If the argument is `(SoPath *) NULL`, `getData` returns only the data stored in the slot.

There are slot classes supporting five data types: `WBoolSlot`, `WIntSlot`, `WFloatSlot`, `WVec3fSlot`, and `WMatrixSlot`.

Callback functions

Callback functions must be in the form:

```
typedef void WSlotCB( void *data,    // auxiliary data.
                      WSlot *slot ); // the slot triggering the callback.
```

Callback functions are registered with or removed from a slot with the slot member functions:

```
void addCallback   ( WSlotCB *f,   // the callback function.
                     void *data ); // auxiliary data.
void removeCallback( WSlotCB *f,   // the callback function.
                     void *data ); // auxiliary data (for identification).
```

The application program can register more than one callback functions with each slot. Callback functions are called when the slot data is updated with `setData`, as described before.

Attachments

The attachment of Inventor nodes/fields to a slot is modeled after the `attach` member function of the manipulator classes for coupling an Inventor node with the delta matrix of the manipulator.

Four attributes control the behavior of an attachment: the direction of data flow (input, output, or both), space conversion between the space of the slot data and the space of the attached node/field (on or off), the data conversion function (described below), and an index for a node/field with many elements. A programmer creates or removes an attachment with the slot member functions:

```
enum WAttachMode {
  INPUT, OUTPUT, BOTH
};

typedef void WAttachConvertFn( WAttach *att,     // the attachment.
                               WAttachMode i_o );// direction of data flow.

// Attachment functions for nodes.
SbBool createAttach( SoPath *wh,      // path to attached node.
                     WAttachMode i_o, // input, output, or both?
                     WAttachMode ini, // direction of data initialization.
                     SbBool tra,      // space transform performed?
                     WAttachConvertFn *fun, // data conversion function.
                     int ind);        // index into multi-element node.
```

```
SbBool removeAttach( SoPath *wh,       // path to attached node.
                     int ind );        // index into multi-element node.

// Attachment functions for fields.
SbBool createAttach( SoPath *wh,
                     <fieldType> *fie,// attached field.
                     WAttachMode i_o,
                     WAttachMode ini,
                     SbBool tra,
                     WAttachConvertFn *fun,
                     int ind );        // index into multi-element field.
SbBool removeAttach( SoField *fie,     // attached field
                     int ind );        // index into multi-element field.
```

More than one node/field can be attached to a slot. `createAttach` creates an attachment and inserts it in the attachment list of the slot. Five classes of attachments — `WBoolAttach`, `WIntAttach`, `WFloatAttach`, `WVec3fAttach`, and `WMatrixAttach` — correspond to the slot classes of the same data types. The attachment keeps track of the slot and the node/field that it attaches as well as its own attributes. If the attachment is capable of receiving data for the slot (i.e. the data flow direction is input or both), the attachment also creates an Inventor data sensor to monitor changes in the attached node/field.

The purpose of the data conversion function is to obtain data from either the slot or the node/field, perform some calculations, then update the other end of the attachment with the result. Typically, a data conversion function is written for a specific slot data type, and handles data conversions between the slot and several node/field classes.

If the attachment is capable of output, the data conversion function is invoked when the slot's `setData` is called. If the attachment is capable of input, then `sensorCB`, a member function of the attachment that is registered with the data sensor, invokes the data conversion function when the node/field is updated.

A default data conversion function called `defaultConvertFn` is provided with each attachment class for registering with attachments of that class. However, a programmer can register a customized conversion function with each attachment.

Connections to Widget Components

A slot is a widget component with a single data port that both inputs and outputs absolute data. The port is updated by `setData`, as described in Section 3.2.2. When the port is modified by output ports connected to it, the slot's `invoke` function calls the slot's `setData` function, which invokes the callback functions and updates the attached nodes/fields.

3.2.4 The Widget Part Classes

Parts, together with the motion hierarchy formed with spaces, define a widget's interactive behaviors and visual feedback. Each part incorporates the geometry that should ideally suggest its usage, and possibly one or more interactive behaviors.

All part classes are descendants of the base class `WPart`. `WPart` provides several services to its descendants, as discussed in the following sections.

Maintaining Geometry

The constructor of `WPart` builds an Inventor subgraph as shown in Figure 3.2. The `pick` node determines whether the part can be picked. The `appSwitch` node allows switching among the three possible children for the part's appearance: invisible, inactive, or active. The leaves of the subgraph are the actual active and inactive styles and geometries of the part. The `pick` node and the `appSwitch` node are controlled externally via input ports (discussed later). The constructor of `WPart` is defined as:

```
WPart( const char *fileName,        // file to obtain geometry from.
       const char *defaultBuffer[], // buffer to obtain geometry from.
       int        numFunc );        // number of behaviors.
```

The constructor reads both the buffer and the file (both provided by the constructor of the derived class) to look for Inventor subgraphs labeled as "inactiveGeom" and "activeGeom".

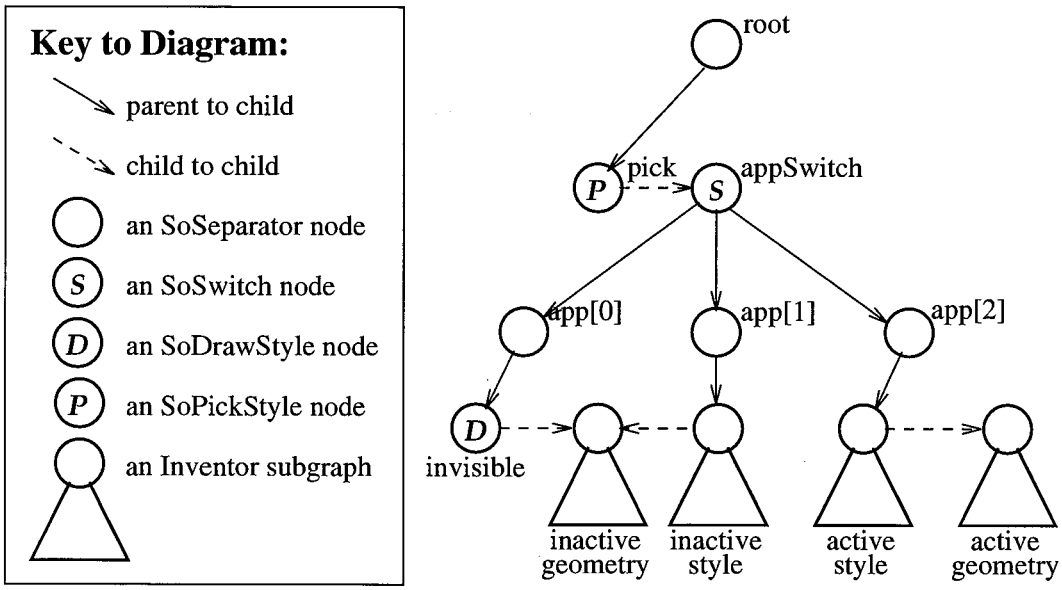

Figure 3.2: The Inventor subgraph maintained by WPart

The subgraphs become the inactive and active geometries of the part respectively. Note that the geometries in the file override those in the buffer, so that a user can easily change the appearance of the part by providing a file with the right name. `numFunc` defines the number of different interactive behaviors the part has.

The functions `setGeom` and `setStyle` are used by a widget to modify the geometries and to set the visual styles of the part. A visual style is an Inventor subgraph that describes the visual property of the part. The functions are defined as follows:

```
void setGeom ( SoNode *inactive, // inactive geometry.
               SoNode *active )  // active geometry.
void setStyle( SoNode *inactive, // inactive style.
               SoNode *active )  // active style.
```

Building the Motion Hierarchy and Providing Feedback

The motion hierarchy of a widget is built with spaces and parts. `setParent` sets the parent space of the part by adding the Inventor subgraph of the part as a child to the root node of

the space:

```
void setParent( WRootedSpace *par ); // parent space.
```

Spaces will be discussed in detail in Section 3.2.5.

A part provides visual feedback by altering the matrix of its *feedback space* , which should be an ancestor of the part in the motion hierarchy. `WPart` defines two standard ports for exporting (or sometimes importing) the feedback matrix:

```
DFMatrixPort *absMatrixIO;  // input/output port for absolute feedback.
DFMatrixPort *delMatrixOut; // output port for delta feedback.
```

The function `setFeedback` is responsible for making a connection between the absolute or delta ports on the part and the feedback space:

```
void setFeedback( WPortedSpace *fee,   // feedback space.
                  SbBool isAbsolute ); // absolute versus delta data.
```

The reason for having the delta matrix output in addition to the absolute output is two-fold: first, a delta matrix is required for updating the edit space; second, the delta output allows parts to accumulate their visual feedbacks in a single feedback space.

`WPart` declares a member function called `updateMatrixPorts`, to be defined in the derived part class, for updating the feedback matrix ports based on the current state of the part:

```
void updateMatrixPorts(); // update the feedback matrix ports.
```

The function is used by some `WPart` member functions that affect either the motion hierarchy or the feedback space, such as `setParent` and `setFeedback`, to make sure the visual feedback remain consistent with the application's states after structural changes are made within the widget. The function is needed because `WPart` has no access to its descendants' internal states.

Providing Information about Interaction

`WPart` declares three member functions that the widget who owns the part calls. These functions must be provided by the derived part class:

```
void manipulateStart(); // prepare for subsequent manipulation.
void manipulate();      // handle manipulation.
void manipulateFinish();// cleanup after manipulation.
```

Typically, `manipulateStart` obtains information about the mouse location and meta-keys, then sets the initial state of the part. It also sets the part's standard output port `absPickedOut` to indicate that the part is picked for manipulation:

```
DFBoolPort *absPickedOut; // the part is picked.
```

`manipulate` obtains the same information as `manipulateStart`, computes the new state of the part based on the current input and the previous state, then updates the state, the feedback matrix ports, and other output data ports supported by the part.

`manipulateFinish` is for cleaning up after a manipulation. It is responsible for resetting `absPickedOut`.

`WPart` defines member functions for obtaining information about the manipulation:

```
SbVec2s getLocaterPosition();           // mouse location in pixel,
                                        // relative to view port.
SbVec2s getLocaterStartPosition();      // mouse location at the
                                        // beginning of manipulation.
SbVec2f getNormalizedLocaterPosition(); // mouse location normalized to
                                        // between 0 and 1.
SbVec2f getNormalizedLocaterStartPosition();// normalized mouse location
                                            // at the start of manipulation.
void setStartLocaterPosition( SbVec2s p );  // set the starting location.

SbVec3f getLocalDetailPoint(); // the hit point in the part's space.
const SoDetail *getDetail();   // details about the hit point.
const SoPath   *getPickPath(); // the path leading to the picked node.
const SoEvent  *getEvent();    // the event triggering the manipulation.
```

Every function above has a direct correspondence in the definition of `SoDragManip`. Because `WPart` is a friend of `SoWidget`, the above functions simply call their corresponding functions in the widget that owns the part to obtain the information.

Providing Controls to the Part

`WPart` defines three standard input ports to allow external control of some of its functions:

```
DFBoolPort *absPickableIn; // can the part be picked?
DFBoolPort *absVisibleIn;  // is the part visible?
DFBoolPort *absActiveIn;   // should the active style be used?
```

Input ports are handled by the `invoke` function, just like other widget components. `invoke` must be provided by the part class to handle the above input ports plus other input ports the part class defines. `WPart` provides member functions to simplify the handling of the standard input ports:

```
void updateAppearance(); // set the appSwitch node (the switching node
                         // controlling the appearance of the part),
                         // based on absVisibleIn and absActiveIn.
void updatePickability();// set the pick node based on absPickableIn.
```

Maintaining MetaKey-to-Behavior Mapping

Part classes may support several interactive behaviors. `WPart` provides functions for mapping meta-key combinations (of shift, alt, and ctrl keys) to integers that represent individual behaviors. These functions simplify the implementation of multi-behavior parts:

```
enum WMetaKey {
  NO_KEY = 0, ANY_KEY = 1, SHIFT_KEY = 2, CTRL_KEY = 4,
  ALT_KEY = 8, ALL_KEY = 15
};

// map a function number with a key combination.
void setFunc( int func,   // the function number.  Must be smaller than
```

```
                    // the numFunc argument given in the constructor.
            int keys ); // the key combination bit-mask.
                    // e.g. SHIFT_KEY | CTRL_KEY mean both
                    // shift and ctrl keys are required.

// get the function number given a key combination.
int getFunc( int keys );
```

3.2.5 The Widget Space Classes

Space components serve two purposes: they simplify space transformation calculations, and form the skeleton of the motion hierarchy for a widget. There are four space component classes: `WWorldSpace`, `WEditSpace`, `WLocalSpace`, and `WCustomSpace`.

All space components are capable of returning the transformation matrices that transform from the spaces they represent to the world space and vice versa. These functions are defined as:

```
SbMatrix getConversionToWorld();
SbMatrix getConversionFromWorld();
```

A *rooted space* can be a part of a widget's motion hierarchy. Each has a root node that makes it capable of becoming a parent of other spaces and widget parts. The root node of such a space is obtained by calling:

```
SoSeparator *getRoot(); // returns the root for the space.
```

A *ported space* contains a modifiable transformation matrix. Each has two matrix data ports to allow other components to access the matrix:

```
DFMatrixPort *absMatrixIO; // absolute matrix input/output port.
DFMatrixPort *delMatrixIn; // delta matrix input port.
```

Two functions related to spaces are provided. `transformMatrix` transforms a matrix in a source space to an equivalent matrix that has the same effect (as observed from the world)

in the target space. `getConversionMatrix` returns the matrix that converts the source space to the target space. They are both overloaded to accept both `WSpace *` and `SoPath *` as arguments:

```
void transformMatrix( const SbMatrix &fromMatrix, SbMatrix &toMatrix,
                      WSpace *fromSpace, WSpace *toSpace );
void transformMatrix( const SbMatrix &fromMatrix, SbMatrix &toMatrix,
                      SoPath *fromPath, WSpace *toSpace );
void transformMatrix( const SbMatrix &fromMatrix, SbMatrix &toMatrix,
                      WSpace *fromSpace, SoPath *toPath );
void transformMatrix( const SbMatrix &fromMatrix, SbMatrix &toMatrix,
                      SoPath *fromPath, SoPath *toPath );

SbMatrix getConversionMatrix ( WSpace *fromSpace, WSpace *toSpace );
SbMatrix getConversionMatrix ( SoPath *fromPath, WSpace *toSpace );
SbMatrix getConversionMatrix ( WSpace *fromSpace, SoPath *toPath );
SbMatrix getConversionMatrix ( SoPath *fromPath, SoPath *toPath );
```

WWorldSpace, WEditSpace and WLocalSpace

These are the *standard* space classes, as opposed to the *custom* space class, because every widget has a `worldSpace`, an `editSpace`, and a `localSpace`, which are instances of `WWorldSpace`, `WEditSpace` and `WLocalSpace` respectively.

`worldSpace` represents the space with no transformation. `localSpace` represents the space under which the widget operates; it is the root of the widget's motion hierarchy, the root node being a child `SoSeparator` node of the widget that owns it.

`editSpace` represents the delta matrix inherited from the manipulator base class; the delta matrix of the widget is modified via the `delMatrixIn` port of `editSpace`. The `invoke` function of `editSpace` transforms the incoming matrix to the equivalent matrix in the edit space before concatenating it to the delta matrix.

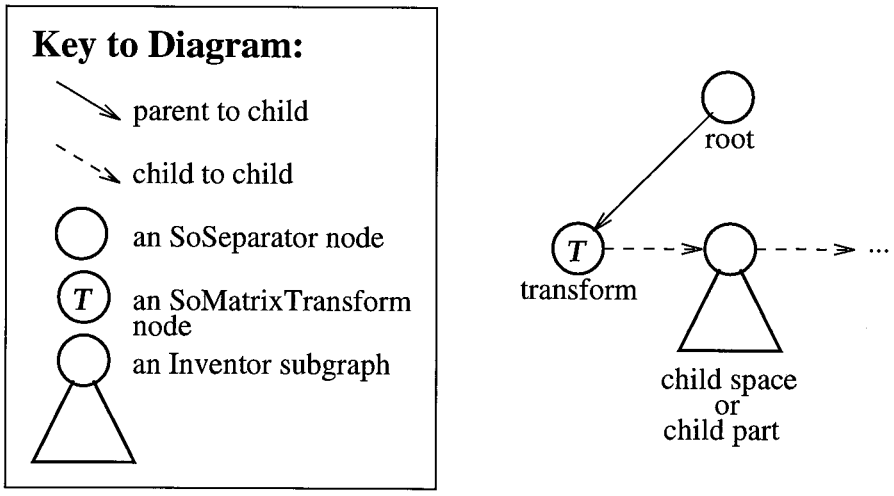

Figure 3.3: The Inventor subgraph maintained by WCustomSpace

WCustomSpace

Instances of `WCustomSpace` in a widget form the motion hierarchy of the widget. A custom space is both a rooted space and a ported space. During instantiation, a custom space acquires its name, its parent space, and the initial value of its matrix:

```
WCustomSpace( const SbName &nam,       // name of the space.
              WRootedSpace *par,       // parent space.
              const SbMatrix &init ); // initial value.
```

The constructor first creates an Inventor subgraph, consisting of an `SoSeparator` as the root node and an `SoMatrixTransform` as the first child of the root node. Then the constructor makes the root node of this space a child of `par`'s root node by calling `setParent`:

```
void setParent( WRootedSpace *par ); // become a child of parent.
```

Figure 3.3 shows the Inventor subgraph created by `WCustomSpace` and how children are added to the space.

A custom space responds to updates from both `delMatrixIn` and `absMatrixIO`. When `delMatrixIn` is updated, `invoke` transforms the incoming matrix to the equivalent matrix in

this custom space before concatenating it to the matrix of the transformation node. When `absMatrixIO` is updated, `invoke` transforms the incoming matrix to the equivalent matrix in the parent space of this space (rather than this space, because the new transformation is to REPLACE the existing transformation of the space. The effect of the existing transformation should be ignored), then replaces the matrix of the transformation node with it.

Chapter 4

Discussion

4.1 Accomplishments

The goals listed in Section 1.3 — i.e. ease of widget creation, extensibility of component library, compatibility with an available 3-D graphics library[1], flexible interface between widgets and application programs, and ability for widgets to control and display multiple application data items — are met through the object-oriented 3-D Widget Programming Library based on Inventor, a widely available 3-D graphics library. This library supports the creation of new 3-D widgets from high-level components. Moreover, the library can be extended by adding new, user-defined components.

A widget built with this library controls one or more states of the application program, and displays the states via relative positions and orientations among geometric parts in a motion hierarchy. A widget interfaces with the application program through callback functions or configurable attachments with Inventor nodes/fields.

A number of widget components are implemented: slots of different data types, standard and custom spaces, several general components such as data converters or data providers, and several parts including a dial, a slider, and a length measure. Using these components, the

[1]More precisely, this library is an extension of the 3-D graphics library, Inventor.

Widget Class Definition		25
Geometry Definition		1
Component Inclusions	5 Parts	10
	7 Spaces	7
	4 Slots	8
Port Connections		8
Initialization	spaces	6
	ports	37
Others		21
TOTAL		123

Table 4.1: Lines of code usage in the implementation of the Rack Widget

implementation of the rack widget (discussed in Section 2.4 and listed in Appendix B) needs fewer than 130 lines of C++ code[2]. The breakdown is shown in Table 4.1. The abstractions provided by the components, which are stand-alone objects with well-defined interfaces with other components, simplify the design of widgets.

4.2 Future Work

The 3-D Widget Programming Library provides the groundwork for building a library of general components and parts. The next step is to determine what constitutes a comprehensive and useful set of functions and interactive behaviors that satisfies most users of 3-D GUIs, and thus minimizes the programming efforts of widget designers.

Using this library, a new widget class is created through conventional programming in C++, with most of the code dedicated to creating widget components and connecting them. A widget and component description language would make widget creation more straight-forward and

[2]By counting semi-colons.

error-free. An even better tool would be a visual 3-D widget designer similar to the form design tool for the Forms Library described in Section 1.2.1. Another possibility would be to construct and modify widgets dynamically at run-time, similar to the 3D widget construction toolkit described in [17], either by the user or by changes in the structure of application data.

Currently, a programmer cannot encapsulate a useful widget into a part to be incorporated into a more complex widget. The ability to build *super-parts* — ported mini-widgets built from primative components and/or other super-parts — would allow complex and often-used constructs to be added to the library and shared among widgets.

The programming library limits visual feedback to relative positions and orientations of parts plus highlighting; for other types of visual feedback, such as color and text, the programmer has to implement them. More investigation is needed in determining other useful visual feedback mechanisms to be incorporated.

A general constraint system for describing relations internally among widget components and externally among widgets and application objects, as opposed to the data flow and data coupling model used in this programming library, should be explored because it would allow more general components to be built. For example, with the virtual trackball part, the dial part is not necessary because the virtual trackball can be constrained to rotate around a choosen axis. The constraint system would also help programmers create application programs that require relations among numerous objects be maintained and controlled. For example, a CAD system that allows the designer to specify spatial relationships among objects would need the constraint system to maintain the relationships after objects are manipulated.

Bibliography

[1] Alan H. Barr, "Global and Local Deformations of Solid Primitives", *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):21-30, July 1984.

[2] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam, "Three-Dimensional Widgets", *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 25(2):183-188, March 1992.

[3] James D. Foley, Andries van Dam, Steven Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 2nd edition, 1990.

[4] Ithaca Software, *HOOPS Graphics System Reference Manual, version 3.2*, 1991.

[5] Michael Kass, "CONDOR: Constraint-Based Dataflow", *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):321-330, July 1992.

[6] Mark A. Linton, John M. Vlissides, and Paul R. Calder, "Composing User Interfaces with InterViews", *IEEE Computer*, 22(2):8-22, February 1989.

[7] Aaron Marcus and Andries van Dam, "User-Interface Developments for the Nineties", *IEEE Computer*, 24(9):49-57, September 1991.

[8] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal, "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces", *IEEE Computer*, 23(11):71-85, November 1990.

[9] Mark H. Overmars, *Forms Library - A Graphical User Interface Toolkit for Silicon Graphics Workstations, version 2.1*, November, 1992.

[10] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card, "Cone Trees: Animated 3D Visualizations of Hierarchical Information", *SIGCHI '91 Proceedings*, pp. 189-194, 1991.

[11] Silicon Graphics Inc., *Graphics Library Programming Guide*, 1991.

[12] Silicon Graphics Inc., *IRIS Inventor Programming Guide - Volume I: Using the Toolkit*, June 1992.

[13] Silicon Graphics Inc., *IRIS Inventor Programming Guide - Volume II: Extending the Toolkit*, June 1992.

[14] Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, D. Brookshire Conner, and Andries van Dam, "Using Deformations to Explore 3D Widget Design", *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):351-352, July 1992.

[15] Paul S. Strauss and Rikk Carey, "An Object-Oriented 3D Graphics Toolkit", *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):341-349, July 1992.

[16] 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", *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):105-112, July 1991.

[17] Robert C. Zeleznik, Kenneth P. Herndon, Daniel C. Robbins, Nate Huang, Tom Meyer, Noah Parker, and John F. Hughes, "An Interactive 3D Toolkit for Constucting 3D Widgets", *Computer Graphics (SIGGRAPH '93 Proceedings)*, pp. 81-84, August 1993.

Appendix A

Using a Widget in an Application

The following sections go through a number of examples of using widgets. The first section shows an application program that communicates with a rack widget through a callback function. The second section focuses on the use of attachments and data conversion functions. The third section dicusses the procedure for customizing widgets.

A.1 Including a Widget and Registering a Callback

The following example shows how a programmer includes a rack widget (as described in Section 2.4) in an application for deforming an object.

A.1.1 The Main Loop

`main` is responsible for creating the display and initializing the widget classes. It calls another function to create the scene graph. The function `main` follows:

```
//////////////////////////////////////////////////////////////////////////////
main()
//////////////////////////////////////////////////////////////////////////////
{
  // create an X window.
  Widget appWindow = SoXt::init( "Deform" );
  if ( appWindow == NULL ) exit( 1 );

  // initalize the widget classes.  These must be called before
  // widgets are created.
  SoWidget::initClass();
  RackWidget::initClass();
```

```
  // create the scene graph for the application.
  scene = create_scene();

  // create an Inventor viewer for viewing the scene.
  viewer = new SoXtExaminerViewer;
  viewer->setSceneGraph( scene );
  (void) viewer->build( appWindow );
  viewer->show();
  SoXt::show( appWindow );

  // start the interaction loop.
  SoXt::mainLoop();
}
```

A.1.2 Creating the Scene Graph and Registering the Callback Function

The function `create_scene` creates the scene graph and sets up the communication between the rack widget and the rest of the application program:

```
////////////////////////////////////////////////////////////////////////
static SoNode *create_scene(void)
////////////////////////////////////////////////////////////////////////
{
  // build the scene graph with a light, a camera,
  // an object to be deformed, and a rack widget.
  // Both the shape and the widget is located at the
  // origin of the world space.  The scene is built
  // with Inventor nodes.

  SoSeparator *root = new SoSeparator;
  root->ref();

  SoPerspectiveCamera *camera = new SoPerspectiveCamera;
  root->addChild( camera );

  root->addChild( new SoDirectionalLight );

  // create the shape for the object.
  SoNode *shape = create_shape();
  root->addChild( shape );

  RackWidget *widget = new RackWidget;
  root->addChild( widget );

  camera->viewAll( root );
```

```
  // register callback functions to all the slots of the widget.
  widget->twistValueSlot->addCallback( deformCB, 0 );
  widget->taperValueSlot->addCallback( deformCB, 0 );
  widget->taperPositionSlot->addCallback( deformCB, 0 );
  widget->bendValueSlot->addCallback( deformCB, 0 );

  return root;
}
```

A.1.3 The Callback Function

The callback function `deformCB` obtains all the deformation values, then modifies the shape of the object accordingly:

```
////////////////////////////////////////////////////////////////////////////
static void deformCB( void *data, WSlot *slot )
////////////////////////////////////////////////////////////////////////////
{
  // determine the widget.
  RackWidget *widget = 0;
  if ( slot )
    widget = (RackWidget *)slot->getOwner();

  if ( !widget )
    return;

  // obtain deformation values.
  float twist = widget->twistValueSlot->getData();
  float taper = widget->taperValueSlot->getData();
  float taperPos = widget->taperPositionSlot->getData();
  float bend = widget->bendValueSlot->getData();

  // deform the shape accordingly.
  ...
}
```

A.2 Using Attachments

Two aspects of attachment usage, creation of attachments and custom data conversion functions, are covered in the following two sections.

A.2.1 Creating Attachments

In the following example, a dial widget controls the rotations of a cube and a cone. The cube rotates around its own z axis, whereas the cone rotates around the z axis of the widget. The length widget displays a rod connecting the centers of the cube and the cone as well as the distance between them.

The following code segment shows the construction of the scene graph, the creation of attachments, and the placement of the widget:

```
SoSeparator         *root   = new SoSeparator;         // the root.
SoPerspectiveCamera *camera = new SoPerspectiveCamera; // the camera.

SoSeparator         *coneSpace = new SoSeparator;      // cone subgraph.
SoTransform         *coneOffset= new SoTransform();    // cone offset.
coneOffset->translation.setValue( 3.0, 0.0, 0.0 );     // 3 to the right.
SoMatrixTransform   *coneXform = new SoMatrixTransform;// transform.
SoMaterial          *coneMat = new SoMaterial;         // cone material.
coneMat->diffuseColor.setValue( .8, 0, 0 );            // red.
coneMat->transparency.setValue( .2 );                  // transparent.

SoSeparator         *cubeSpace = new SoSeparator;      // cube subgraph.
SoTransform         *cubeOffset= new SoTransform();    // cube offset
cubeOffset->translation.setValue( -3.0, 0.0, 0.0 );    // 3 to the left.
SoMatrixTransform   *cubeXform = new SoMatrixTransform;// transform.
SoMaterial          *cubeMat = new SoMaterial;         // cube material.
cubeMat->diffuseColor.setValue( 0 , 0.8, 0 );          // green.
cubeMat->transparency.setValue( .2 );                  // transparent.

SoTransform         *widgetOffset= new SoTransform();  // widget offset.
widgetOffset->rotation.setValue(                       // 45 deg.
  SbVec3f( 0.0, 1.0, 0.0 ),-3.141592654/4.0 );
DialWidget          *dialWidget = new DialWidget;      // dial widget.
LengthWidget        *lengthWidget = new LengthWidget;  // length widget.


// build the scene graph with a camera, a light, a cone on the right,
// a cube on the left, a dial widget in the centre, and a length
// widget connecting the centers of the cone and the cube.
root->ref();
root->addChild(camera);
root->addChild( new SoDirectionalLight );

root->addChild( coneSpace );
coneSpace->addChild( coneOffset );
coneSpace->addChild( coneXform );
coneSpace->addChild( coneMat );
```

```
coneSpace->addChild( new SoCone );

root->addChild( cubeSpace );
cubeSpace->addChild( cubeOffset );
cubeSpace->addChild( cubeXform );
cubeSpace->addChild( cubeMat );
cubeSpace->addChild( new SoCube );

root->addChild( widgetOffset );
root->addChild( dialWidget );   // NOTE: still under world space!!
root->addChild( lengthWidget );

camera->viewAll( root );

// rotate the dial widget by 45 degrees by setting
// the work space of the widget.
SoPath *workPath = new SoPath( root );
workPath->append( widgetOffset );
workPath->ref();
dialWidget->setWorkSpacePath( workPath );
workPath->unref();

// attach the cube's transform to dialWidget's rotationSlot.
SoPath *cubePath = new SoPath( root );
cubePath->ref();
cubePath->append( cubeSpace );
cubePath->append( cubeXform );
testWidget->rotationSlot->createAttach(
  cubePath,
  OUTPUT,                          // slot outputs to transform.
  OUTPUT,                          // initialize transform with slot.
  FALSE,                           // no space transformation.
  &WMatrixAttach::defaultConvertFn,// use default conversion function.
  0 );


// attach the cone's transform to dialWidget's rotationSlot.
// Same as the above except space transformation is active; i.e.  the
// cone rotates around the widget's Z axis.
SoPath *conePath = new SoPath( root );
conePath->ref();
conePath->append( coneSpace );
conePath->append( coneXform );
testWidget->getSlot("rotationSlot")->createAttach( conePath,
  OUTPUT, OUTPUT, TRUE, &WMatrixAttach::defaultConvertFn, 0 );

// attach the cone's and cube's transforms to the two slots of the
// length widget.
```

```
lengthWidget->point1Slot->createAttach(
  conePath,
  INPUT,                          // slot reads transform.
  INPUT,                          // transforms initialize slot.
  TRUE,                           // space transformation performed.
  &WVec3fAttach::defaultConvertFn,// use default conversion function.
  0 );
conePath->unref();
lengthWidget->point2Slot->createAttach( cubePath,
  INPUT, INPUT, TRUE, &WVec3fAttach::defaultConvertFn, 0 );
cubePath->unref();
```

A.2.2 Creating a Custom Conversion Function

Suppose the default conversion function for vector slots does not support conversion to and from `SoSFColor` fields, and the programmer would like to use a vector slot to control the color of an object. The following conversion function will perform the task:

```
////////////////////////////////////////////////////////////////////////////
void newVec3fConvertFn( WAttach *att, WAttachMode i_o )
//
// - need to include "WFieldType.h" for definitions of field types such as
//   SFColor below.
////////////////////////////////////////////////////////////////////////////
{
  WVec3fAttach *a = ( WVec3fAttach * ) att;     // the attachment.
  WVec3fSlot   *o = ( WVec3fSlot * ) a->owner; // owner of the attachment.
  SoNode       *n = a->who->getTail();         // the attached node.

  switch ( i_o )
  {
    case INPUT:
      if ( a->field && a->fieldType == SFColor )
      {
        SbColor c( ( ( SoSFColor * ) a->field )->getValue() );
        o->setData( c, (SoPath *)NULL, a );
        return;
      }
      break
    case OUTPUT:
      if ( a->field && a->fieldType == SFColor )
      {
        SbVec3f vS = o->getData( (SoPath *)NULL );
        ( ( SoSFColor * ) a->field )->setValue( vS );
        return;
      }
```

```
      break;
    default:
      // do nothing.
  }

  // let the default conversion function handle the rest.
  WVec3fAttach::defaultConvertFn( att, i_o );
}
```

The following call to `createAttach` attaches the `vec3fSlot` of a widget to the `diffuseColor` field of the `materialNode`.

```
widget->vec3fSlot->createAttach(
  materialNodePath,              // path to material node.
  &(materialNode->diffuseColor), // field to be attached.
  BOTH,                          // slot both displays and controls.
  INPUT,                         // field initializes slot.
  FALSE,                         // no space transformation.
  newVec3fConvertFn,             // use customized conversion function.
  0 );
```

A.3 Customizing a Widget

A programmer may not be satisfied with the default appearance of a widget. For example, a programmer wants to modify a rack widget so that the dial handles are longer than the default length of 1. There are three ways to accomplish the task; the three methods differ mainly in the scopes of modification.

The first method redefines the geometries of `WDialPart`; it affects all instances of the part class. The programmer creates a file *WDialPart.iv* that describes the active and/or inactive geometries of instances of `WDialPart`. The subgraphs of the active and inactive geometry should be labeled `activeGeom` and `inactiveGeom`. The file should be located in the current directory or the directory indicated by the `SO_MANIP_DIR` environment variable. The file content is as follows:

```
#Inventor V1.0 ascii
Separator {
  Label { label "activeGeom" }
  Label { label "inactiveGeom" }
  Translation { translation 0 1.0 0 }
  Cylinder { parts  SIDES height 2.0 radius 0.1 }
  Translation { translation 0 0.5 0 }
  Sphere { radius 0.15 }
```

```
  Translation { translation 0 -1.0 0 }
  RotationXYZ { axis X angle 1.5707963 }
  Cylinder { height 0.15 radius 0.15 }
}
```

The second method defines the geometries of `RackWidget`; it affects all instances of the widget class. The programmer creates a file *RackWidget.iv* that modifies the active and/or inactive part geometries for the widget class. Modifications made with this method overrides those made with the first method. The following example modifies geometries of the `bendDial` part of the rack widget:

```
#Inventor V1.0 ascii
Separator {
  Label { label "bendDialActive" }
  Label { label "bendDial" }
  # the geometry as listed in the previous example.
  ...
}
```

The third method modifies an instance of `RackWidget`. This is useful when the programmer wants to use several rack widgets in an application program but wants them to look differently. The programmer instantiates a widget using one of the special constructors that asks for either a geometry file, a scene graph, or a dictionary that contains name/geometry pairs. The constructors for the rack widget are:

```
RackWidget( const char *userGeomFile );
RackWidget( SoGroup    *userGeom );
RackWidget( SbDict     *userDict );
```

The geometry file, scene graph, or dictionary should contain subgraphs with labels corresponding to the names of parts in the widget, just like the *RackWidget.iv* example above. Modifications made with this method overrides those made with the two previous methods.

For more information, please read Chapter 7 of the "IRIS Inventor Programming Guide - Volume II" for a detailed discussion.

Appendix B

Implementing a Widget from Existing Parts

The implementation of the rack widget in Section 2.4 will be described. It consists of four interactive parts — two dial parts and two slider parts — that provide controls for the four parameters. A dial part provides rotation feedback on its z axis. A slider part provides translation feedback along its y axis. Both parts output float values that represent the amount of rotation or translation. A non-interactive part is included to provide the geometry for the widget's main axis.

The rack widget needs four float slots so that the application program can access the four deformation parameters and act on changes caused by manipulations of the widget.

The procedure for creating a new widget class is very similar to creating a new manipulator. You may want to refer to Chapter 8 of the "IRIS Inventor Programming Guide - Volume II" for more information.

B.1 Defining RackWidget

The widget class `RackWidget` is defined in the header file *RackWidget.h.* In general, a definition of a widget class should do the following:

- invoke the `SO_SUBNODE_ID_HEADER` macro.
- declare the member function `initClass`. It is required for all Inventor node classes.
- declare four constructors: one with no argument, and the other three with one argument which is either a name of a geometry file, a pointer to an Inventor group node,

or a pointer to a dictionary. The latter three constructors allow an instance of the widget class to obtain its geometries from sources other than its default geometry buffer (`RackWidget::rackWidgetGeomBuffer[]`) and default geometry file (*RackWidget.iv*).

- define pointers to all the parts and slots for the widget class.
- define a pointer to `classDictionary`. It stores named Inventor nodes or subgraphs to be included in the widget.
- define a pointer to the geometry buffer. It holds a description of the default geometries of widget parts.
- declare the destructor.
- declare or define other class-specific members.

This is the listing of *RackWidget.h*:

```
#ifndef RACKWIDGET_H
#define RACKWIDGET_H
#include "SoWidget.h"
class WDialPart;
class WSliderPart;
class WGeomPart;
class WFloatSlot;

//////////////////////////////////////////////////////////////////////////////
class RackWidget : public SoWidget
//////////////////////////////////////////////////////////////////////////////
{
  SO_SUBNODE_ID_HEADER( RackWidget ); // Define required typeId and name stuff.

  public:
    static void initClass(); // initializes the class.  To be called
                             // after SoInteraction::init().
    // constructors.
    RackWidget();
    RackWidget( const char *userGeomFile );
    RackWidget( SoGroup    *userGeom );
    RackWidget( SbDict     *userDict );
    // parts.
    WDialPart   *bendDial;          // controls the bend.
    WDialPart   *twistDial;         // controls the twist.
    WSliderPart *taperSlider;       // controls the taper.
    WSliderPart *taperOffsetSlider; // controls where the taper ends.
    WGeomPart   *mainAxis;          // geometry for the main axis.
    // slots.
```

```
    WFloatSlot *bendValueSlot;      // amount of bend from the bend handle.
    WFloatSlot *twistValueSlot;     // amount of twist from the twist handle.
    WFloatSlot *taperValueSlot;     // amount of taper from the taper handle.
    WFloatSlot *taperPositionSlot;  // where the taper ends.

  protected:
    static SbDict *classDictionary; // dictionary for this class.
    void constructorSub();          // common constructor code.

  private:
    static char *rackWidgetGeomBuffer[]; // geometry buffer.
    ~RackWidget();                       // destructor.
    static const double pi;              // the value pi.
};
#endif
```

B.2 Default Geometries

The file *RackWidgetGeom.h* defines the geometry buffer `RackWidget::rackWidgetGeomBuffer`, the default geometries of parts in the widget. The buffer content is in the Inventor file format.

Many part classes provide default geometries for their instances; therefore a widget programmer needs not design geometry for a part unless he or she wants to override the default geometry of the part. Some part classes — e.g. `SoGeomPart` — do not provide default geometries and therefore their instances are invisible by default.

For the rack widget, the part `mainAxis` needs a geometry that represents the main axis of the widget, and the default geometry for `taperPositionSlider` has to be modified because it would be obscured by the object to be deformed otherwise; therefore, new geometries for these two parts have to be specified. The default geometries are used for other parts.

Note that both the active and inactive geometries should be specified for a part. The two geometries can be the same, as shown in `RackWidgetGeom.h`:

```
#ifndef RACKWIDGETGEOM_H
#define RACKWIDGETGEOM_H
char *RackWidget::rackWidgetGeomBuffer[] =
{
  "#Inventor V1.0 ascii\n\
   Separator {\n\
     Label { label \"mainAxis\" }\n\
     Label { label \"mainAxisActive\" }\n\
     RotationXYZ { axis X angle 1.5707963 }\n\
     Cylinder { height 2.0 radius 0.10 }\n\
   }\n",
```

```
  "Separator {\n\
     Label { label \"taperOffsetSlider\" }\n\
     Label { label \"taperOffsetSliderActive\" }\n\
     RotationXYZ { axis X angle -1.5707963 }\n\
     Translation { translation 0 0.5 0 }\n\
     Cylinder { height 1.0 radius 0.075 }\n\
   }\n",
  "\0"
};
#endif
```

B.3 Initializing the Widget

The first section of *RackWidget.c++* invokes the macros needed by all Inventor node classes, and defines the static member variables in the class. The member function `initClass` is also implemented:

```
#include <iostream.h>
#include <Inventor/SoDB.h>
#include "DFBoolNode.h"
#include "DFFloatNode.h"
#include "WDialPart.h"
#include "WSliderPart.h"
#include "WGeomPart.h"
#include "WFloatSlot.h"
#include "WSpace.h"
#include "RackWidget.h"

#include "RackWidgetGeom.h"

// Initialize static dictionary that contains the geometry
// and style resources for this class.
SbDict *RackWidget::classDictionary = 0;

// Macro which defines required variables for subclasses of SoNode.
SO_SUBNODE_ID_VARS(RackWidget);

// Macro which defines required methods for subclasses of SoNode.
SO_SUBNODE_ID_METHODS(RackWidget);

// Constant needed in calculation
const double RackWidget::pi = 3.141592654;

//////////////////////////////////////////////////////////////////////////////
void RackWidget::initClass()
```

```
//////////////////////////////////////////////////////////////////////////////
{
    SO_SUBNODE_INIT_ID(
        RackWidget,       // an example instance
        "RackWidget",     // class name
        SoWidget );       // parent class
}
```

B.4 Constructors and Destructors

The four constructors of `RackWidget` differ only in the way they construct `SoWidget`; therefore, they all call a common function `constructorSub` that does the real work for the widget class. Here are the four constructors:

```
// No arguments.
RackWidget::RackWidget() : SoWidget()
{ constructorSub(); }

// Take a file name.
RackWidget::RackWidget( const char *userGeomFile ) : SoWidget( userGeomFile )
{ constructorSub(); }

// Take a scene graph.
RackWidget::RackWidget( SoGroup *userGeom ) : SoWidget( userGeom )
{ constructorSub(); }

// Take a dictionary.
RackWidget::RackWidget( SbDict *userDict ) : SoWidget( userDict )
{ constructorSub(); }
```

The `SoWidget` constructor used determines the extra source of part geometries for a widget instance besides the geometries common to all instances.

`constructorSub` is responsible for the following:

- setting up the class dictionary.
- creating the spaces, slots, parts, and general components, and adding them to the widget.
- connecting ports of components to generate the desired behavior and initializing ports that should remain constant.

```
//////////////////////////////////////////////////////////////////////////////
void RackWidget::constructorSub()
```

```
////////////////////////////////////////////////////////////////////////
{
  SO_SUBNODE_BEGIN_PROTOTYPE(RackWidget);

  // Read the default geometry for this widget. This will
  // only happen once.
  classDictionary = createDictionary(
      "RackWidget.iv",             // default geom file
      rackWidgetGeomBuffer,        // compiled-in defaults
      SoWidget::classDictionary ); // parent dictionary

  SO_SUBNODE_END_PROTOTYPE();

  // This will be the dictionary for use by all instances of this class.
  setClassDictionary( classDictionary );

  // Create the spaces.
  SbMatrix matrix, matrix2;
  matrix.setTranslate( SbVec3f( 0.0,0.0,-1.0 ) );
  WCustomSpace *twistOffset = createSpace( "twistOffset", localSpace, matrix );
  matrix = SbMatrix::identity();
  WCustomSpace *taperOffset = createSpace( "taperOffset", localSpace, matrix );
  matrix.setTranslate( SbVec3f( 0.0,0.0,1.0 ) );
  matrix2.setRotate( SbRotation( SbVec3f( 1.0,0.0,0.0 ), pi/2.0 ) );
  matrix.multLeft( matrix2 );
  matrix2.setRotate( SbRotation( SbVec3f( 0.0,1.0,0.0 ), -pi/2.0 ) );
  matrix.multLeft( matrix2 );
  WCustomSpace *bendOffset = createSpace( "bendOffset", localSpace, matrix );

  matrix.setRotate( SbRotation( SbVec3f( 1.0,0.0,0.0 ), pi/2.0 ) );
  WCustomSpace *taperOffsetOffset = createSpace( "taperOffsetOffset",
                                                 taperOffset,matrix );
  matrix = SbMatrix::identity();
  WCustomSpace *twistFeedback = createSpace("twistFeedback",twistOffset,matrix);
  WCustomSpace *taperFeedback = createSpace("taperFeedback",taperOffset,matrix);
  WCustomSpace *bendFeedback  = createSpace("bendFeedback", bendOffset, matrix);

  // Create the slots.
  twistValueSlot    = new WFloatSlot();
  addSlot( twistValueSlot, "twistValueSlot" );
  taperValueSlot    = new WFloatSlot();
  addSlot( taperValueSlot, "taperValueSlot" );
  taperPositionSlot = new WFloatSlot();
  addSlot( taperPositionSlot, "taperPositionSlot" );
  bendValueSlot     = new WFloatSlot();
  addSlot( bendValueSlot, "bendValueSlot" );

  // Create the parts.
```

```
WDialPart *twistDial = new WDialPart();
addPart( twistDial,"twistDial",1,twistFeedback,twistFeedback,TRUE,
        classDictionary );
WSliderPart *taperSlider = new WSliderPart();
addPart( taperSlider,"taperSlider",1,taperFeedback,taperFeedback,TRUE,
        classDictionary );
WSliderPart *taperOffsetSlider = new WSliderPart();
addPart( taperOffsetSlider,"taperOffsetSlider",1,taperOffsetOffset,
        taperOffset,TRUE,classDictionary );
WDialPart *bendDial = new WDialPart();
addPart( bendDial,"bendDial",1,bendFeedback,bendFeedback,TRUE,
        classDictionary );
WGeomPart *mainAxis = new WGeomPart();
addPart( mainAxis,"mainAxis",1,localSpace,NULL,TRUE,classDictionary );

// create temporary general components
// for persistent general components, addNode should be called.
DFBoolNode *boolNode = new DFBoolNode( TRUE );
DFFloatNode *floatNode = new DFFloatNode( 0.0 );

// initialize ports using "setData( WPort *port )"
// and make connections with "connect( WPort *port )".
//
// NOTE: "setData( <dataType> data )" is not used here because
// it is intended to be used by the owner of the port and
// thus does not update the internal state of the port owner.
//
twistDial->absLowIn->setData( floatNode->absFloatIO );
twistDial->absRangeIn->setData( floatNode->absFloatIO );
twistDial->absFloatIO->setData( floatNode->absFloatIO );
floatNode->setData( 2*pi );
twistDial->absScaleIn->setData( floatNode->absFloatIO );
twistDial->absPickableIn->setData( boolNode->absBoolIO );
twistDial->absVisibleIn->setData( boolNode->absBoolIO );
twistDial->absActiveIn->connect( twistDial->absPickedOut );
//
twistValueSlot->absFloatIO->connect( twistDial->absFloatIO );

floatNode->setData( 0.0 );
taperSlider->absLowIn->setData( floatNode->absFloatIO );
floatNode->setData( 1.0 );
taperSlider->absRangeIn->setData( floatNode->absFloatIO );
taperSlider->absScaleIn->setData( floatNode->absFloatIO );
taperSlider->absFloatIO->setData( floatNode->absFloatIO );
taperSlider->absPickableIn->setData( boolNode->absBoolIO );
taperSlider->absVisibleIn->setData( boolNode->absBoolIO );
//
taperSlider->absActiveIn->connect( taperSlider->absPickedOut );
```

```
  taperValueSlot->absFloatIO->connect( taperSlider->absFloatIO );

  floatNode->setData( -1.0 );
  taperOffsetSlider->absLowIn->setData( floatNode->absFloatIO );
  floatNode->setData( 2.0 );
  taperOffsetSlider->absRangeIn->setData( floatNode->absFloatIO );
  floatNode->setData( 1.0 );
  taperOffsetSlider->absScaleIn->setData( floatNode->absFloatIO );
  floatNode->setData( 0.0 );
  taperOffsetSlider->absFloatIO->setData( floatNode->absFloatIO );
  taperOffsetSlider->absPickableIn->setData( boolNode->absBoolIO );
  taperOffsetSlider->absVisibleIn->setData( boolNode->absBoolIO );
  //
  taperOffsetSlider->absActiveIn->connect( taperOffsetSlider->absPickedOut );
  taperPositionSlot->absFloatIO->connect( taperOffsetSlider->absFloatIO );

  floatNode->setData( -pi );
  bendDial->absLowIn->setData( floatNode->absFloatIO );
  floatNode->setData( 2*pi );
  bendDial->absRangeIn->setData( floatNode->absFloatIO );
  bendDial->absScaleIn->setData( floatNode->absFloatIO );
  floatNode->setData( 0.0 );
  bendDial->absFloatIO->setData( floatNode->absFloatIO );
  bendDial->absPickableIn->setData( boolNode->absBoolIO );
  bendDial->absVisibleIn->setData( boolNode->absBoolIO );
  //
  bendDial->absActiveIn->connect( bendDial->absPickedOut );
  bendValueSlot->absFloatIO->connect( bendDial->absFloatIO );

  mainAxis->absVisibleIn->setData( boolNode->absBoolIO );
  boolNode->setData( FALSE );
  mainAxis->absPickableIn->setData( boolNode->absBoolIO );

  delete boolNode;
  delete floatNode;
}
```

The destructor should release any resources the widget has acquired. However, the destructor of `SoWidget` already handles all the spaces, parts, slots and general components added to the widget. Therefore, we only need a very simple destructor:

```
////////////////////////////////////////////////////////////////////////
RackWidget::~RackWidget()
////////////////////////////////////////////////////////////////////////
{
}
```

Appendix C

Implementing a General Component

The only requirement for a new general component class is that it is derived from the base class `DFNode`, the ancestor of all widget components. For a general component to be useful, it should contain at least one port capable of output so as to affect other components. Every derived class of `DFNode` must implement the `invoke` member function.

We shall create a new general component class `DFFloatVec3fNode` that converts between three float values and a 3-D vector both ways.

C.1 Defining the Component

DFFloatVec3fNode.h defines the class. The class definition should do the following:

- declare a constructor and a destructor.
- declare the `invoke` function.
- define the pointers to all the ports for the component.

The header file is listed below:

```
#ifndef DFFLOATVEC3FNODE_H
#define DFFLOATVEC3FNODE_H
#include "DFNode.h"
#include "DFPort.h"
//////////////////////////////////////////////////////////////////////////////
```

```
class DFFloatVec3fNode : public DFNode
////////////////////////////////////////////////////////////////////////////
{
  public:
    // constructor and destructor.
    DFFloatVec3fNode();
    ~DFFloatVec3fNode();
    // data flow invoke.
    void invoke( void *, DFPort *port );
    // ports.
    DFFloatPort *absXIO;    // X component of vector.
    DFFloatPort *absYIO;    // Y component of vector.
    DFFloatPort *absZIO;    // Z component of vector.
    DFVec3fPort *absVec3fIO;// vector.
};
#endif
```

The names of the ports reflect the facts that they all hold absolute values (as opposed to delta values) and they are all capable of both input and output.

C.2 Constructors and Destructors

The constructor is responsible for creating the ports and the destructor should delete them. The first section of `DFFloatVec3fNode.c++` follows:

```
#include "DFFloatVec3fNode.h"

////////////////////////////////////////////////////////////////////////////
DFFloatVec3fNode::DFFloatVec3fNode()
////////////////////////////////////////////////////////////////////////////
{
  absVec3fIO = new DFVec3fPort( this, ABS_PORT | INP_PORT| OUT_PORT );
  absXIO = new DFFloatPort( this, ABS_PORT | INP_PORT| OUT_PORT );
  absYIO = new DFFloatPort( this, ABS_PORT | INP_PORT| OUT_PORT );
  absZIO = new DFFloatPort( this, ABS_PORT | INP_PORT| OUT_PORT );
  absVec3fIO->setData( SbVec3f( 0.0, 0.0, 0.0 ), NULL );
  absXIO->setData( 0.0 );
  absYIO->setData( 0.0 );
  absZIO->setData( 0.0 );
}
////////////////////////////////////////////////////////////////////////////
DFFloatVec3fNode::~DFFloatVec3fNode()
////////////////////////////////////////////////////////////////////////////
{
  delete absVec3fIO;
```

```
  delete absXIO;
  delete absYIO;
  delete absZIO;
}
```

C.3 The Member Function invoke

The `invoke` member function is called when the data in one of the input ports is changed. It finds out which port is modified, then modifies the component's internal state and output ports:

```
////////////////////////////////////////////////////////////////////////////
void DFFloatVec3fNode::invoke( void *, DFPort *port )
////////////////////////////////////////////////////////////////////////////
{
  WSpace  *s;
  SbVec3f vec = absVec3fIO->getData( s );

  if ( (void *) port == absVec3fIO )
  {
    // check if the data actually changes.
    // If not, do not "setData" in order to cut
    // unnecessary data flow.
    if ( vec[0] != absXIO->getData() )
      absXIO->setData( vec[0] );
    if ( vec[1] != absYIO->getData() )
      absYIO->setData( vec[1] );
    if ( vec[2] != absZIO->getData() )
      absZIO->setData( vec[2] );
  }
  else if ( (void *) port == absXIO && vec[0] != absXIO->getData() )
    absVec3fIO->setData( SbVec3f( absXIO->getData(), vec[1], vec[2] ), s );
  else if ( (void *) port == absYIO &&  vec[1] != absYIO->getData() )
    absVec3fIO->setData( SbVec3f( vec[0], absYIO->getData(), vec[2] ), s );
  else if ( (void *) port == absZIO &&  vec[2] != absZIO->getData() )
    absVec3fIO->setData( SbVec3f( vec[0], vec[1], absZIO->getData() ), s );
}
```

Appendix D

Implementing a Part

Each new part class must implement the following methods:

- `manipulateStart` prepares the part for manipulation.
- `manipulate` updates the internal state of the part and the output ports based on the manipulation.
- `manipulateFinish` finishes the manipulation.
- `invoke` is called when one of the input ports is modified. It then updates the part's internal state and output ports accordingly.
- `updateMatrixPorts` updates `delMatrixOut` and `absMatrixIO` based on the part's current state.

The example part is `WDialPart`. This part shows a handle that can be grabbed and rotated around the z axis in the part's local space. It has the following ports:

- `absFloatIO` and `delFloatOut` output the internal state `absValue` and the change in the state respectively. `absValue` is the amount of rotation scaled by the value obtained from `absScaleIn` (see below). `absFloatIO` is also an input port that can alter `absValue`.
- `absLowIn` sets the lower limit for `absValue`.
- `absRangeIn` sets the range of `absValue`, starting from the lower limit. A value of 0 means that the range is infinite.
- `absScaleIn` sets the scaling factor for `absValue`. `absValue` is equal to the amount of rotation (number of turns) times the scaling factor.

D.1 Defining the Part

WDialPart.h defines the class. The class definition:

- declares a constructor and a destructor.
- declares the required member functions.
- defines the pointers to all the ports for this part.
- declares the geometry buffer.
- defines a plane projector that is used for calculating the amount of rotation caused by the manipulation.
- defines member variables which store the part's internal state.

The header file is listed below:

```
#ifndef WDIALPART_H
#define WDIALPART_H

#include "Inventor/projectors/SbPlaneProjector.h"
#include "DFPort.h"
#include "WPart.h"

//////////////////////////////////////////////////////////////////////////////
class WDialPart : public WPart
//////////////////////////////////////////////////////////////////////////////
{
  public:
    WDialPart(); // constructor.
    ~WDialPart();// destructor.

    void updateMatrixPorts(); // update matrix ports based on internal states.
    void manipulateStart();   // prepare for manipulation.
    void manipulate();        // update ports and internal states.
    void manipulateFinish();  // finish.
    void invoke( void *data, DFPort *port );

    DFFloatPort *absFloatIO; // I/O port for value.
    DFFloatPort *delFloatOut;// output port for change in value.
    DFFloatPort *absLowIn;   // lower limit for value.
    DFFloatPort *absRangeIn; // range of value.
    DFFloatPort *absScaleIn; // value of one revolution.

  private:
```

```
    void setData( double value, void *source = NULL  ); // used by other
                                                        // member functions.
    static char  *wDialPartGeomBuffer[];// the default geometry.

    double       absValue; // the current state of the dial.
    double       absLow,   // the lower limit.
                 absRange, // the range.
                 absScale; // the scale (value per revolution).
    static const double pi;

    SbPlaneProjector planeProjector;// the projector used.
    SbVec3f          prevProj;      // previous projection into 3D space.
    SbVec3f          axisOfRotation;// axis of rotation in the feedback space.
};
#endif
```

D.2 Default Geometries

The file *WDialPartGeom.h* initializes the geometry buffer `WDialPart::wDialPartGeomBuffer` that describes the default geometries of the part. The description is written in the Inventor file format.

The geometry of this part represents a handle. Both the active and inactive geometries (labeled as `activeGeom` and `inactiveGeom`) must be provided for a part. The two geometries can be the same, as shown in this example:

```
#ifndef WDIALPARTGEOM_H
#define WDIALPARTGEOM_H
char *WDialPart::wDialPartGeomBuffer[] =
{
  "#Inventor V1.0 ascii\n\
   Separator {\n\
     Label { label \"activeGeom\" }\n\
     Label { label \"inactiveGeom\" }\n\
     Translation { translation 0 0.5 0 }\n\
     Cylinder { parts  SIDES height 1.0 radius 0.1 }\n\
     Translation { translation 0 0.5 0 }\n\
     Sphere { radius 0.15 }\n\
     Translation { translation 0 -1.0 0 }\n\
     RotationXYZ { axis X angle 1.5707963 }\n\
     Cylinder { height 0.15 radius 0.15 }\n\
   }\n",
  "\0"
};
#endif
```

D.3 Constructors and Destructors

The constructor needs to create a plane projector and initialize `WPart`. It is also responsible for initializing the internal state and creating the required ports.

The destructor deallocates all the ports and resources allocated for the part.

```
#include <iostream.h>
#include "SoWidget.h"
#include "WSpace.h"
#include "WDialPart.h"
#include "WDialPartGeom.h"

const double WDialPart::pi = 3.141592654;

////////////////////////////////////////////////////////////////////////////
WDialPart::WDialPart()
////////////////////////////////////////////////////////////////////////////
: planeProjector( SbPlane( SbVec3f( 0.0, 0.0, 1.0 ), 0 ) ),
  WPart( "WDialPart.iv", wDialPartGeomBuffer )
{
  absFloatIO = new DFFloatPort( this, ABS_PORT | INP_PORT | OUT_PORT );
  delFloatOut= new DFFloatPort( this, DEL_PORT | OUT_PORT );
  absLowIn = new DFFloatPort( this, ABS_PORT | INP_PORT );
  absRangeIn = new DFFloatPort( this, ABS_PORT | INP_PORT );
  absScaleIn = new DFFloatPort( this, ABS_PORT | INP_PORT );

  absValue = 0;
  absFloatIO->setData( absValue );
  delFloatOut->setData( 0.0 );
  absLow = 0;
  absLowIn->setData( absLow );
  absRange = 0; // 0 = no limit
  absRangeIn->setData( absRange );
  absScale = 2*pi;
  absScaleIn->setData( absScale );
  absMatrixIO->setData( SbMatrix::identity(), NULL );
  delMatrixOut->setData( SbMatrix::identity(), NULL );
}

////////////////////////////////////////////////////////////////////////////
WDialPart::~WDialPart()
////////////////////////////////////////////////////////////////////////////
{
  delete delFloatOut;
  delete absFloatIO;
  delete absLowIn;
```

```
  delete absRangeIn;
  delete absScaleIn;
}
```

D.4 The Member Function updateMatrixPort

`updateMatrixPort` is required by `WPart` for setting the two matrix ports `delMatrixOut` and `absMatrixIO` based on the current internal state of the part. In this example, the function sets `delMatrixOut` to the identity matrix because this function does not change the state of the part. `absMatrixOut` is set to a matrix that represents a rotation proportional to `absValue`, the state of the part representing the amount of rotation.

```
////////////////////////////////////////////////////////////////////////////
void WDialPart::updateMatrixPorts()
////////////////////////////////////////////////////////////////////////////
{
  // convert the z-axis in part space to the feedback space.
  if ( parent && feedback )
    getConversionMatrix( parent, feedback ).multVecMatrix(
      SbVec3f( 0.0, 0.0, 1.0 ), axisOfRotation );
  else
    axisOfRotation = SbVec3f( 0.0, 1.0, 0.0 );

  SbMatrix matrix;
  SbRotation rotation;

  // delMatrixOut is set to the identity matrix since there is
  // no change in the internal state "absValue".
  matrix = SbMatrix::identity();
  if ( feedback )
    delMatrixOut->setData( matrix, feedback );
  else
    delMatrixOut->setData( matrix, NULL );

  // calculate the rotation in the feedback space.
  float absAngle = absValue*(2*pi)/absScale;
  rotation.setValue( axisOfRotation, absAngle );
  matrix.setRotate( rotation );

  // The new matrix is intended for the parent of the feedback space.
  if ( feedback )
    absMatrixIO->setData( matrix, feedback->getParent() );
  else
    absMatrixIO->setData( matrix, NULL );
}
```

D.5 The Member Function invoke

invoke responds to the input ports in the following ways:

- absFloatIn triggers updates to the member variable absValue and output ports.
- absMatrixIn is ignored.
- absLowIn, absRangeIn and absScaleIn trigger updates to the member variables absLow or absRange or absScale and output ports.
- absVisibleIn and absActiveIn cause WPart::updateAppearance to be called.
- absPickableIn causes WPart::updatePickability to be called.

```
////////////////////////////////////////////////////////////////////////////
void WDialPart::invoke( void *data, DFPort *port )
////////////////////////////////////////////////////////////////////////////
{
  if ( parent && feedback )
    getConversionMatrix( parent, feedback ).multVecMatrix(
      SbVec3f( 0.0, 0.0, 1.0 ), axisOfRotation );
  else
    axisOfRotation = SbVec3f( 0.0, 1.0, 0.0 );

  if ( port == ( DFPort * ) absFloatIO )
  {
    // set the internal state of the part with absFloatIO's data.
    // Also indicate the absFloatIO is the source of data in the
    // second parameter.
    setData( absFloatIO->getData(), absFloatIO );
  }
  else if ( port == ( DFPort * ) absMatrixIO )
  {
    // This part ignores input from this port.
  }
  else if ( port == ( DFPort * ) absLowIn )
  {
    absLow = absLowIn->getData();
    setData( absValue );
  }
  else if ( port == ( DFPort * ) absRangeIn )
  {
    absRange = absRangeIn->getData();
    if ( absRange < 0 )
    {
      absLow = absLow + absRange;
```

```
      absRange = - absRange;
    }
    setData( absValue );
  }
  else if ( port == ( DFPort * ) absScaleIn )
  {
    if ( absScaleIn->getData() == 0.0 )
    {
      cerr << "WDialPart::invoke() - absScaleIn cannot be zero, defaults to one"
           << endl;
      absScale = 1.0;
      absScaleIn->setData( absScale );
    }
    else
      absScale = absScaleIn->getData();

    setData( absValue );
  }
  else if ( port == ( DFPort * ) absVisibleIn ||
            port == ( DFPort * ) absActiveIn )
    updateAppearance();
  else if ( port == ( DFPort * ) &absPickableIn )
    updatePickability();
}
```

D.6 The Member Functions for Manipulations

`manipulateStart` prepares the dial part for subsequent manipulations. It:

- sets up the plane projector.
- records the initial point of projection.
- determines the axis of rotation in the `feedback` space.
- sets the data in the `absPickedOut` port to indicate that the part is picked.

```
////////////////////////////////////////////////////////////////////////////
void WDialPart::manipulateStart()
////////////////////////////////////////////////////////////////////////////
{
  planeProjector.setViewVolume  ( getViewVolume() );
  planeProjector.setWorkingSpace( parent->getConversionToWorld() );

  prevProj = planeProjector.project( getNormalizedLocaterPosition() );
```

```
  // obtain the rotation axis in the feedback space.
  if ( feedback )
    getConversionMatrix( parent, feedback ).multVecMatrix(
      SbVec3f( 0.0, 0.0, 1.0 ), axisOfRotation );
  else
    axisOfRotation = SbVec3f( 0.0, 1.0, 0.0 );

  absPickedOut->setData( TRUE );
}
```

`manipulate` determines the amount of rotation, and calculates the new `absValue` based on the rotation, the lower limit, the range, and the scaling factor. It then updates `absValue` and the output ports.

```
////////////////////////////////////////////////////////////////////////////
void WDialPart::manipulate()
////////////////////////////////////////////////////////////////////////////
{
  // just in case the view changed ...
  planeProjector.setViewVolume( getViewVolume() );

  // find the delta angle.
  SbVec3f currProj =  planeProjector.project( getNormalizedLocaterPosition() );

  SbRotation rotation( prevProj, currProj );

  SbVec3f axis;
  float delAngle;
  rotation.getValue( axis, delAngle );
  delAngle = ( axis[2] > 0 ) ? delAngle : -delAngle;

  float delValue = delAngle*absScale/(2*pi);
  setData( absValue + delValue );

  // the working space may have changed due to the changes above.
  planeProjector.setWorkingSpace( parent->getConversionToWorld() );
  prevProj = planeProjector.project( getNormalizedLocaterPosition() );
}
```

`manipulateFinish` sets the port `absPickedOut` to indicate that this part is no longer picked.

```
////////////////////////////////////////////////////////////////////////////
void WDialPart::manipulateFinish()
////////////////////////////////////////////////////////////////////////////
```

```
{
  absPickedOut->setData( FALSE );
}
```

`setData` updates the member variable `absValue` and the output ports `delFloatOut`, `absFloatIO`, `delMatrixOut` and `absMatrixIO`. `setData` avoids excessive data flow by not setting the port that causes `setData` to be called. `setData` is a convenience member function called only by `manipulate` and `invoke`; therefore, it is made a private member function.

```
////////////////////////////////////////////////////////////////////////////
void WDialPart::setData( float value, void *source )
////////////////////////////////////////////////////////////////////////////
{
  // check for overflow.
  float tmpValue = value;
  if ( absRange != 0 )
    if ( value < absLow )
      tmpValue = absLow;
    else if ( value > absLow + absRange )
      tmpValue = absLow + absRange;

  if ( tmpValue == absValue ) // no change.
    return;

  float delValue = tmpValue - absValue;
        absValue = tmpValue;

  // update the FloatOuts.
  delFloatOut->setData( delValue );

  if ( source != absFloatIO || absValue != absFloatIO->getData() )
    absFloatIO->setData( absValue );

  // update the MatrixOuts.
  SbMatrix matrix;
  SbRotation rotation;

  // set delMatrixOut.
  float delAngle = delValue*(2*pi)/absScale;
  rotation.setValue( axisOfRotation, delAngle );
  matrix.setRotate( rotation );
  if ( feedback )
    delMatrixOut->setData( matrix, feedback );
  else
    delMatrixOut->setData( matrix, NULL );

  // set absMatrixIO.  Ignore the possibility that it may be a source
```

```
  // because we are not responding to this port in invoke().
  float absAngle = absValue*(2*pi)/absScale;
  rotation.setValue( axisOfRotation, absAngle );
  matrix.setRotate( rotation );
  if ( feedback )
    absMatrixIO->setData( matrix, feedback->getParent() );
  else
    absMatrixIO->setData( matrix, NULL );
}
```

Appendix E

Relevant Enhancements in the New Release of Inventor

Inventor 2.0 has recently been released. The following sections briefly describe the new and enhanced features in the new version that are relevant to the 3-D Widget Programming Library. The information is obtained from the book "the Inventor Mentor" by Josie Wernecke, published by the Addison-Wesley Publishing Company.

The last section relates this new release to the 3-D Widget Programming Library.

E.1 Manipulators and Draggers

Inventor 2.0 introduces a new class of interactive nodes called *draggers*, and enhances the manipulators.

Draggers, like the old manipulators in Inventor 1.x, support the click-drag-release interactive style. New features that draggers support, but not available in old manipulators, are:

- Each dragger contains *fields* that allow other nodes and the application program to access its data (more on fields later). This is easier to use than the *delta matrices* provided by old manipulators.
- A dragger provides its own visual feedback (e.g. motion) during manipulation. An old manipulator has to rely on its edit path (set by the attach mechanism or explicitly by a member function) for visual feedback.
- Draggers can be combined to form compound draggers. Since each dragger manages its own motion, a motion hierarchy can be formed in a compound dragger. The combination

of old manipulators into compound manipulators is much more restricted, as described in Section 1.2.3.

- The working space of a dragger is determined by its position in the scene graph. For an old manipulator, the working space has to be specified (unless the world space, the default, is desired) either through the attach mechanism or explicitly through a member function.

The main difference between the new manipulators (in Inventor 2.0) and the old manipulators (in Inventor 1.x) is that new manipulators *replace* the nodes they affect rather than *attach* to them. For example, the new trackball manipulator *is* a transformation node that can be rotated interactively and the new spotlight manipulator *is* a spotlight node whose position, direction and spread can be manipulated interactively. The interactivity of a manipulator is provided by a dragger — the fields of the dragger are used to edit the fields of the manipulator. Each manipulator supplies member functions for replacing a node in a scene graph with the manipulator and vice versa.

E.2 Engines and Fields

Inventor 2.0 introduces a new class of light-weight nodes called *engines*. Engines are mainly used for animation and constraining one part of a scene in relation to other parts of the same scene. An engine obtains input data from its *fields* (if any), performs its function, and outputs through its *engine outputs* (fields capable of output only).

Fields are enhanced in version 2.0 to allow connections from other fields (using the member function `connectFrom`). A connection allows a field to be updated by another field it connects from. The connection mechanism allows cycles and multiple outputs from a single field, but not multiple inputs to a field. Connections are capable of converting between data types automatically and they can be deactivated without being broken.

E.3 3-D Widgets and Inventor 2.0

The following list compares the 3-D Widget Programming Library and the above-mentioned features in Inventor 2.0:

- The slot/attachment mechanism of widgets and the port/connection mechanism of widget components in the 3-D Widget Programming Library are very similar to the field/connection mechanism in Inventor 2.0. The field/connection mechanism in Inventor 2.0 is more general (it works on all fields in all nodes) and more consistent (there is no distinction between connections and attachments as in the 3-D Widget Programming Library).

- General components of widgets are similar to Inventor engines, except that engines connect to scene nodes while general components only connect to other components within a widget.
- Both allow a motion hierarchy to be built within a widget or dragger. However, since the 3-D Widget Programming Library decouples the coordinate frames (spaces) and the interactive components (parts), visual feedbacks of several parts can be combined into a single feedback space. Another advantage of the 3-D Widget Programming Library is that it provides functions for transforming between coordinate frames; it is not clear how Inventor 2.0 provide this capability.

Other comparisons, such as the ease of building new draggers (widgets) from existing draggers (components) and the ease of building new draggers (parts) with new behaviors, cannot be made until the book "The Inventor Toolmaker", also from Addison-Wesley, is available.

Due to the changes made to the manipulator classes, the 3-D Widget Programming Library is not compatible with Inventor 2.0.