Open inventor and PREMO

D. Wang, G.J. Reynolds and I. Herman

Computer Science/Department of Interactive Systems

CS-R9606 1996

Report CS-R9606
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Open Inventor and PREMO

D. Wang, G. J. Reynolds and I. Herman
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

PREMO is an emerging international standard for the presentations of multimedia objects including computer graphics. Open Inventor^{TM} is a commercially available "de facto" standard for interactive computer graphics packaged as a library of objects. In this report we consider whether the concepts and objects of PREMO are sufficient to represent a professional quality system, such as Open Inventor.

By comparing PREMO with Open Inventor, we hope to show that PREMO's computer graphics environment model can properly describe the procedure by which Open Inventor realizes graphics applications, and PREMO's event model can naturally model Open Inventor's event handling mechanism in a natural way. The scene graph is a core concept in Open Inventor. Most Open Inventor functions rely on various operations over scene graphs. The construction, edition and traversal of scene graphs are shown to be readily implemented based on the current set of PREMO objects. Graphics rendering, event handling and the scene graph constitute the fundamental parts of Open Inventor, other Open Inventor functionalities can be constructed from these. We conclude that since these three fundamental parts of Open Inventor can be properly modelled and implemented by PREMO, the concepts and objects of PREMO are sufficient to represent Open Inventor.

AMS Subject Classification (1991): 68N15

CR Subject Classification (1991): D.1.5,D.2.2,H.5.1,I.3.2

Keywords & Phrases: standards, PREMO, Open Inventor, object-oriented.

1. Introduction

1.1 CGRM and PREMO

In 1992, the ISO/IEC subcommittee on Computer Graphics and Image Processing (ISO/IEC JTC1 SC24) published a Standard called the Computer Graphics Reference Model (CGRM) [6]. This document was, in a sense, the last in a series of standards on computer graphics, starting by GKS in 1985 [5], sometimes referred to as the "first generation" of graphics standards [1]. As its title indicates, CGRM describes an abstract model, and intends to give a succinct way of describing various computer graphics systems. As such, it was not the intention to serve as a replacement for any of the concrete graphics standards developed within or outside ISO. However, the ideas, and indeed the full model of the CGRM found its way lately into a new standard, namely PREMO.

PREMO (Presentation Environments of Multimedia Objects) is a project to develop a new generation standard within SC24, which aims not only at computer graphics, but general presentation environments, too. In the domain of PREMO the term "presentation" encompasses all the various media like audio, video, 3D sound, images, tactile effects, and, of course, traditional graphics. The need for a new generation of standards for computer graphics has already emerged in the past 4-5 years to answer the challenges raised by new graphics techniques and programming environments; it is extremely fortunate that the review process to develop this new generation of presentation environments coincided with the emergence of multimedia. Hence, a very fruitful synergy effect can be capitalized on.

PREMO adopts the basic approach of CGRM, albeit casting it into more concrete terms in the form of objects, and adapting it to the needs of various media. Unfortunately, given the limitations imposed by the size of this report, it is not possible to give a thorough description of neither CGRM nor PREMO here. Section 1.4 below gives only a very short overview of those aspects of these documents which are necessary to understand the essential features of this report. The interested reader should consult the paper edited by G.S.Carson [2] for an introduction on CGRM; as for PREMO, a first overview, published a few years ago [3], or a more recent one [4] can be used as references[1].

1.2 Open Inventor

Open Inventor [13][12] is an object oriented toolkit based on OpenGL [11]. Originating from Silicon Graphics, this toolkit has become extremely popular and widespread in recent years, also due to the widespread adoption of Open GL as the basis for 3D graphics on a number of workstation platforms as well as on Windows'95. Though the functionalities provided by Open Inventor are powerful, the idea of the system is simple. It can be summarised as "a scene graph plus a set of actions". Things, such as graphical objects, graphical properties, reactions for various events, are represented as nodes in Open Inventor. A scene graph is a collection of such nodes and moreover it defines an ordering over the nodes. In such an order, properties and interaction specifications are attached to each graphical object in a straightforward way. An application is developed mainly by specifying a scene graph and the system realises it by applying various actions on such a scene graph. Traversing a scene graph according to the defined order is the basic way in which the system starts to handle an action. For instance, executing a rendering action starts by traversing the scene graph to collect properties for each graphical object and then the graphics rendering can be realised based on the set of complete specifications of each geometric shape; and when an event occurs, the event handling action traverses the scene graph until it meets a node which handles this event. Open Inventor provides many functionalities to support 3D applications. In this report, we concentrate on three aspects: graphics rendering, event handling and scene graphs. In our view, these three aspects constitute the fundamental parts of Open Inventor.

1.3 Open Inventor and PREMO

Any general model, let alone a standard in preparation, can only be considered as acceptable if it can be matched with practice. In terms of the PREMO development this means that, to be acceptable, the model of PREMO (based on the CGRM) should be checked against a real–life system. Presentation of the main results of such an investigation is the real topic of this report. Open Inventor was chosen as a target for this check, because of its object-orientedness (which makes it easily comparable to PREMO) and its wide availability. Although Open Inventor is not a multimedia presentation environment but concentrates primarily on synthetic graphics, it was felt that its internal structure is general enough for our purposes.

What does "check" mean in our case? The questions we were asking ourselves were: is it possible to describe the model of Open Inventor within the framework of PREMO? Is it possible to describe a possible implementation strategy of Open Inventor on top of PREMO? If not, what are the changes and/or the improvements necessary on the PREMO/CGRM approach? These issues are particularly important in view of the fact that PREMO is still an evolving standard, i.e., it is still possible to introduce all necessary improvements to the specification to adapt it to practical requirements. Obviously, results from a series of proofs of concept gives an invaluable experience to the PREMO development team.

The outcome of our investigations, as presented in the rest of this report, have proven to be quite satisfactory. It became clear that PREMO/CGRM could indeed meet the requirements of a system like Open Inventor. There is a clear match between the notions and internal structures used by Open Inventor and PREMO. This positive outcome has an importance that goes, in fact, beyond a simple proof of concept for the coming PREMO standard. Indeed, at its meeting in July 1995, in Ottawa, SC24 has adopted a new work item on a 3D computer graphics metafile activity. This metafile format will be based on the Open Inventor File Format, which is used by Open Inventor applications to exchange 3D graphical data. Obviously, this file format reflects the structure of Open Inventor itself; consequently, the possible compatibility of Open Inventor and PREMO will make it also possible to define a full compatibility between a future PREMO standard (as a presentation environment) and

[1]One can also visit the www site `http://www.cwi.nl/Premo/` to get up-to-date information on PREMO.

the 3D Metafile format (as an exchange format). This compatibility has a great importance for the acceptance of both standards in the future.

1.4 Short Overview of PREMO and CGRM

The major features of PREMO can be briefly summarised as follows.

- PREMO is a Presentation Environment. PREMO aims at providing a standard "programming" environment in a very general sense. The aim is to offer a standardized, hence conceptually portable, development environment that helps to promote portable multimedia applications.
- PREMO aims at a Multimedia presentation, whereas earlier SC24 standards concentrated either on synthetic graphics or image processing systems. Multimedia is considered here in a very general sense; high-level virtual reality environments, which mix real-time 3D rendering techniques with sound, video, or even tactile feedback, and their effects, are also within the scope of PREMO.
- PREMO is Object Oriented. This means that, through standard object-oriented techniques, a PREMO implementation becomes extensible and configurable. Object-oriented technology also provides a framework to describe distribution in a consistent manner.

The full PREMO standard consists of a number of Parts (currently 4), which, although there are interdependencies, progress through the usual ISO working scheme independently of one another. These parts are as follows.

- *Part 1: Fundamentals of PREMO* [7]. This Part contains a overview of PREMO giving its scope, justification, and an explanation of key concepts. It describes the overall architecture of PREMO and specifies the common semantics for defining the externally visible characteristics of PREMO objects in an implementation-independent way.
- *Part 2: Foundation component* [8]. This component lists an initial set of object types and non-object types, useful for the construction of, presentation of, and interaction with multimedia information. These types include fairly traditional types, such as event handlers, aggregates, as well as types more specifically tailored to the needs of interactive multimedia systems.
- *Part 3: Modelling, Presentation, and Interaction Component* [9]. This component combines media control with modelling and geometry. This is an abstract component from which concrete modelling and presentation components are expected to be derived. Thus, for example, a virtual reality component that is derived, at least in part, from this component, might refine the renderer objects defined in Part 3 to objects most appropriate in the virtual reality domain. This is the Part of PREMO which incorporates an adaptation of the full CGRM, and which will be relevant for the remainder of this report.
- *Part 4: Multimedia System Services* [10]. This component provides an infrastructure for building multimedia computing platforms that support interactive multimedia applications dealing with synchronised, time–based media in a heterogeneous distributed environment.

The key idea of Part 3 is that multimedia presentation can be considered as a series of transformation steps between the application and the operator. Transformation is considered in a very general sense, e.g., it includes the transformations among different colour models, coding and decoding algorithms, etc. Transformation steps are modelled in PREMO in terms of five abstract levels called environments: construction, virtual, viewing, logical, and realisation (see Figure 0.1). These environments form a processing network, using the object types developed in Part 2. Information, e.g., multimedia data, attributes, input requests and data, results of inquiries, etc., can flow in both directions between two adjacent environments. Although Figure 0.1 shows a simple, pipeline-oriented network, this is only the basic case. More generally, any number of fan-ins and fan-outs are possible between two adjacent environments. For example, it is possible for an instance of a virtual environment to be connected to several instances of viewing environments, thereby resulting in multiple views of the same model.

On the output side, the role of each type of environment can be summarised as follows.

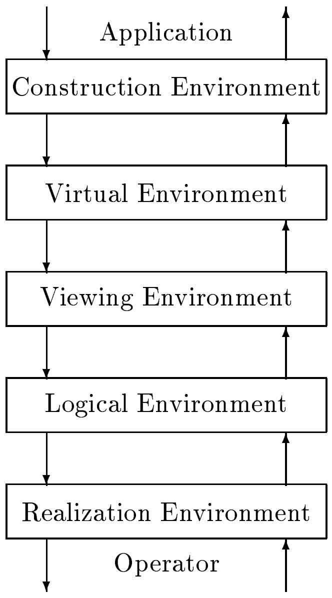


Figure 0.1: Environments in PREMO's Part 3.

- Construction environment: In this environment, the application data to be displayed is prepared as a model from which specific presentation scenes may be produced. The application may only edit the model in the construction environment.
- Virtual environment: In this environment, a scene of the model is produced. The scene consists of a set of virtual output primitives ready for presentation. The geometry of these virtual primitives is completely defined in all dimensions so that scenes are geometrically complete.
- Viewing environment: In this environment, a picture of the scene is generated by projection. The picture consists of a set of viewing output primitives ready for completion. Output primitives in the viewing environment may have a lower geometric dimensionality than in the virtual environment.
- Logical environment: In this environment, a presentable version of the picture is completed ready for realization. The presentable image consists of a set of logical output primitives. Associated with each output primitive is a set of properties associated with completion.
- Realisation environment: In this environment, a display of the presentable picture is presented. The display consists of a set or realization output primitives. This display need not necessarily correspond to a physical display, it can also be a sound output device, a video hardware, or a logical driver.

The symmetry between input and output is reflected on the diagram; as for output primitives, Part 3 makes a series of statements on the role of each environment in the processing of input tokens.

The internal model of each environment follows the same structure. It is therefore possible to describe an environment through a general PREMO object type which is then specialised for the various types of environments described above. Internally, an environment consists of a network of processing objects, cooperating with various types of aggregates which store multimedia data (or their references) locally. Figure 0.2 gives an overview of the general structure of an environment: the rectangles represent subtypes of PREMO processing objects, specialised for the needs of the environments, whereas the circles represent various types of aggregates. Various nodes in this network bear different names in the concrete environment (see in Table 0.1). Dashed and continuous arrows represent control and data flows, respectively, among the different entities. The PREMO object types defining the various environments are not the only types defined in Part 3. Both output primitives

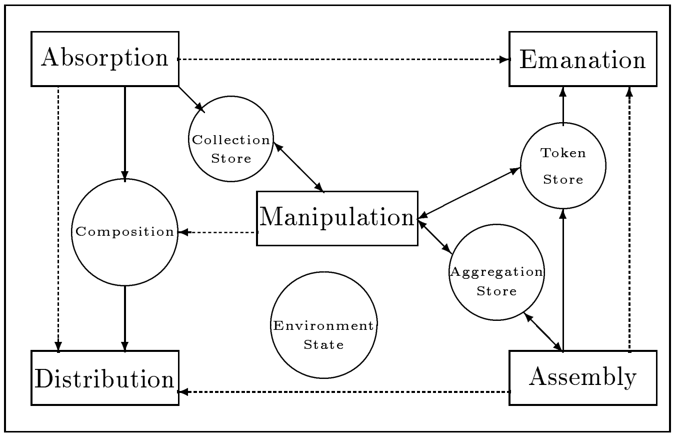


Figure 0.2: A PREMO Environment.

Environment	Absorption	Emanation	Composition	Token Store
Construction	Preparation	Utilization	Model	Instruction Store
Virtual	Production	Generation	Scene	Directive Store
Viewing	Projection	Elevation	Picture	Selection Store
Logical	Completion	Abstraction	Graphical Image	Information Store
Realization	Presentation	Accumulation	Display	Lexeme Store

Table 0.1: Process and data names in each environment.

and input tokens are defined in terms of a large palette of general PREMO types. The properties of output primitives specify their geometry and appearance. These properties are currently classified as six features: spatial, visual, aural, tactile, textual, and identification features. Each feature is described as a PREMO object type and an output primitive or an input token is an aggregation of these features (see in Figure 0.3).

Another aspect of PREMO, which is important for the comparison between PREMO and Open Inventor, is the PREMO event model (see in Figure 0.4). This model is embodied by the specification of special event handler objects.

The essential feature of the event model is the separation between the source of the events (e.g., a mouse, some external hardware, or other PREMO objects implementing a more complex interaction scheme), and the recipient of these events. Sources broadcast the events without having any knowledge of which objects would receive them; this is done by forwarding the event instance to special PREMO event handler objects. Prospective recipients of events register with these event handler objects, placing a request based on the event type and optionally other more complex constraint specifications. The recipients are then notified by the event handler on the arrival of an event, together with information on the source of the event. This simple mechanism constitutes one of the main building blocks for the creation of more complex interaction patterns in PREMO.

1.5 This report

This report consists of four parts: Part I: *General Structure*, Part II: *Events*, Part III: *Other Features* and Part IV *Scene graphs*. We first study Open Inventor's general structure. This study includes consideration of the steps that Open Inventor goes through to render pictures defined by an application program, the general architecture in which Open Inventor responds to the activities of an application, and whether or not these can be properly modelled by PREMO. Second, we study Open Inventor's events and the mechanism provided by Open Inventor for dealing with events, and then discuss

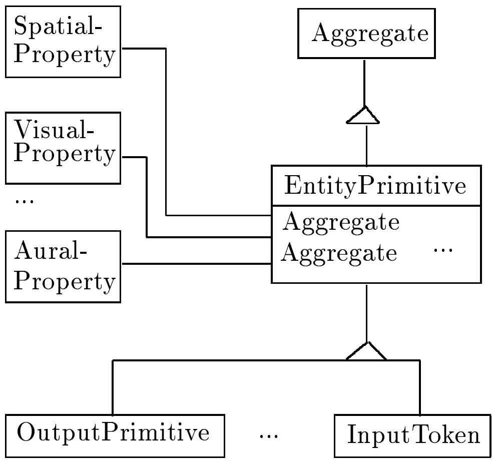


Figure 0.3: PREMO object types for output primitives and input tokens.

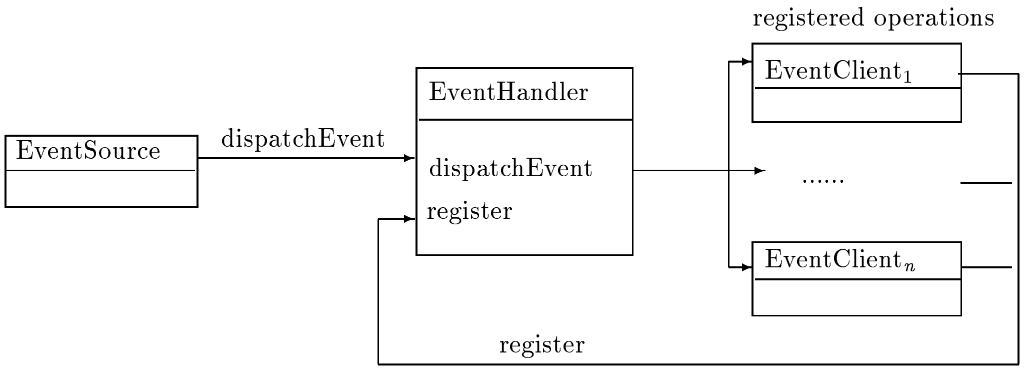


Figure 0.4: PREMO Event Model.

whether or not PREMO can model these properly. Finally, we study the remaining parts of Open Inventor, which have importance as facilities provided by the computer graphics system, but are not fundamental in the sense that they can be constructed from the basic functionalities studied in Parts I and II. Therefore, we only point out the relationships between these features and the basic functionalities appearing in Parts I and II, and do not discuss how to model these features in any detail. The *scene graph* is very important in Open Inventor. Most functionalities rely on various operations over scene graphs. This motivates us to write a set of PREMO object types which reflect the functionalities of scene graphs in Open Inventor. This is reported in Part IV.

An example is used to illustrate the main aspects studied in this report. An application program is created that specifies how to draw a stick-man and how this stick-man can be interactively manipulated. Part I will concentrate on how the stick-man appears on the screen according to the specification in the program (see the illustration in Figure 0.5a). Although Open Inventor's event mechanism is touched on in Part I, we do not look at this in any details. Our purpose in studying it in this part is to gain a general picture of the architecture of Open Inventor. A more extensive study of how Open Inventor responds to events so that the stick-man can be properly manipulated is given in Part II (see the illustration in Figure 0.5b). Part III shows other facilities which may be used to make the application program simpler, to make the manipulations of the stick-man easier to describe and so on (see the illustration in Figure 0.5c). Part IV gives an implementation of scene graphs in PREMO which include (1) construction and edition of scene graphs and (2) the scene graph traversal (see in Figure 0.5d).

It should be noted that PREMO is not an implemented system as yet. It is a standard consisting of a set of components, where each component contains a set of object types. All of the object types taken together provide a basic framework for multimedia systems and incorporate rules which any implemented multimedia object has to obey if it is a standard PREMO object. Therefore, in saying that Open Inventor can be implemented by means of PREMO, we mean that a system based on the framework provided by PREMO and obeying the rules required by PREMO can be implemented, and this system functionally equals that defined by Open Inventor. However, our purpose is only to know whether or not PREMO can implement Open Inventor and not to create real implementation. Thus, we shall be satisfied by investigating whether or not the basic framework provided by PREMO is rich enough to let Open Inventor naturally fit in, instead of going into implementation details.

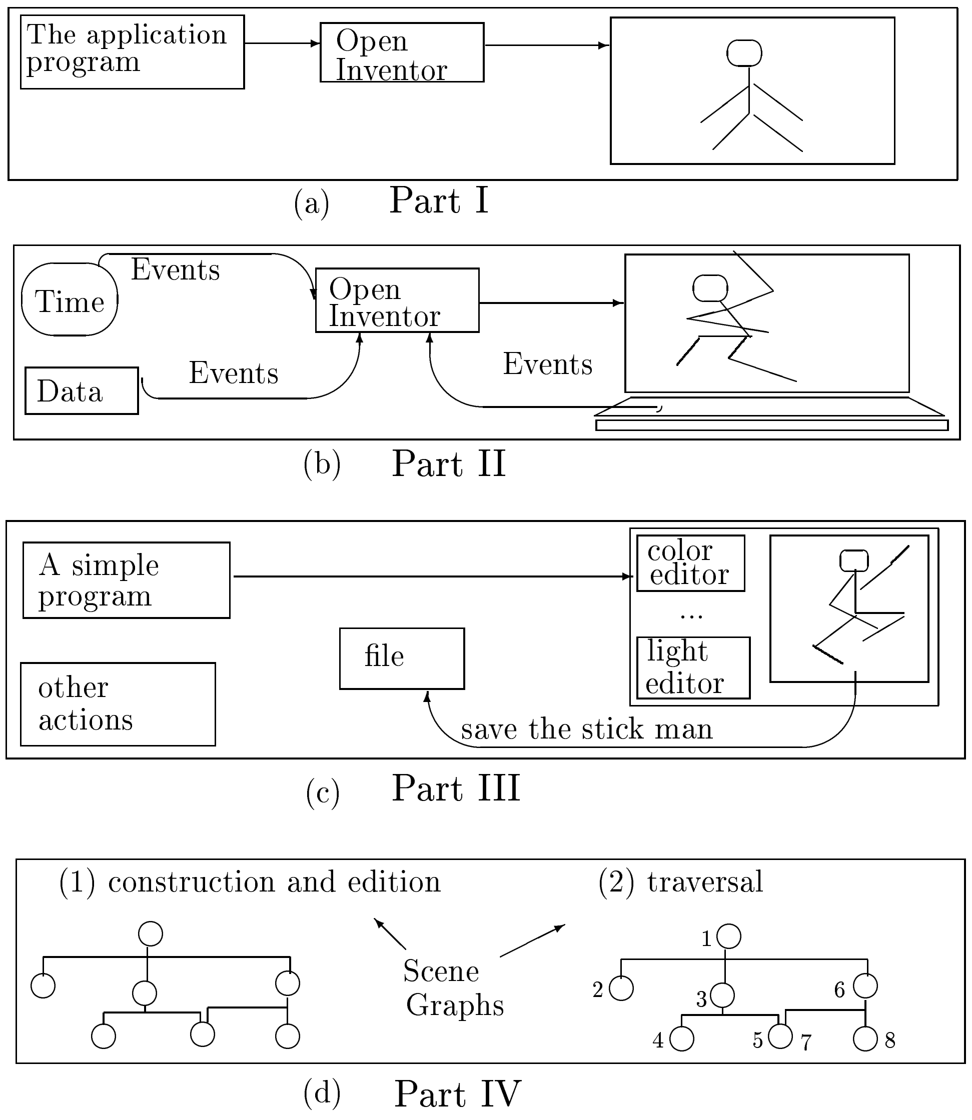


Figure 0.5: An illustraion of this report through an example where a stick-man is rendered and various actions over it are carried out according to the application program.

Chapter 1
General Structure

In this Part, we study the steps Open Inventor goes through to realise a graphics application and discuss the relationship between those steps and PREMO's computer graphics environments (environment model).

First, we analyze a simple Open Inventor program to find out which steps Open Inventor undertakes to realise a static picture and discuss how these steps can be fitted into PREMO's environment model. Second, we analyze programs with actions (i.e. by executing such programs, not only are pictures rendered in the render area but various actions such as animation and interaction may also be performed) to find out which steps Open Inventor needs to run an arbitrary Open Inventor program. We then discuss how to realise Open Inventor's program executing procedure by means of PREMO. This gives a rough impression of the relationship between Open Inventor and PREMO. One important conclusion from this analysis is that scene graphs are central to Open Inventor's conceptual model of computer graphics. In order to gain a clearer picture of how Open Inventor realises graphics, it is essential to understand what a scene graph is and how scene graphs are built, modified and used. After studying scene graphs, we will reconsider the relationship between Open Inventor and PREMO so that a clearer picture of how PREMOs' environment model models Open Inventor can be presented.

1. Basic structure of an Open Inventor program

Consider a very simple Open Inventor program which constructs a scene graph composed of a camera node, a light node, a material node and a cone node. Running this program, we see a window with a red cone drawn in the render area. The code of this program is shown in Figure 1.1 (See page 23 in [13] for the complete code).

Although this program is very simple, it shows that the basic structure of an Open Inventor program consists of 7 steps.

1. *Initialisation*: A program starts by specifying a main window (line 1), which initialises Open Inventor. The initialisation includes the database *SoDB*, all nodekit classes *SoNodekit*, all interaction classes (i.e. manipulators, event callback node and selection node) *SoInteraction* and X-window initialisation *XtAppInitialize*.
2. *Scene graph specification*: A scene graph is specified (in line 2 to 10). The scene graph contains all the information needed to describe the picture to be shown in the render area.
3. *Render area specification*: A render area is specified (in line 11).
4. *Camera view*: The camera view is defined (in line 12).

Widget myWindow = SoXt::init(argv[0]);	1
if (myWindow == NULL) exit(1);	
SoSeparator *root = new SoSeparator;	2
SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;	3
SoMaterial *myMaterial = new SoMaterial;	4
root → ref();	5
root → addChild(myCamera);	6
root → addChild(new SoDirectionalLight);	7
myMaterial → diffuseColor.setValue(1.0, 0.0, 0.0);	8
root → addChild(myMaterial);	9
root → addChild(new SoCone);	10
SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);	11
myCamera → viewAll(root, myRenderArea → getViewportRegion());	12
myRenderArea → setSceneGraph(root);	13
myRenderArea → setTitle("Hello Cone");	14
myRenderArea → show();	15
SoXt::show(myWindow);	16
SoXt::mainLoop();	17

Figure 1.1: The Open Inventor code of the red cone program.

5. *Putting scene graph in the render area*: The scene graph is associated with the render area (in line 13).
6. *Realisation*: Show the window including the graphics inside its render area (in line 15 and 16).
7. *Main loop*: This loop (loops forever) will retrieve and dispatch window specific events (in line 17).

This basic structure is shown pictorially in Figure 1.2.

It should be clear from the example that Open Inventor makes a clean separation between generic window system independent statements and window system specific code. In the example, the X window system is used, although this can be replaced by some other window system if necessary. Another importance is that Open Inventor uses OpenGL for rendering.

Considering to represent Open Inventor by means of PREMO's environment model, an *application* is specified in an Open Inventor program and the *operator* is the window management system (here it is the X-window system). A scene graph is specified in an application program. Based on the specification, Open Inventor constructs scene graphs in its data base. This step belongs to PREMO's construction environment. The method *viewAll* is used in the application program (line 12 in Figure 1.1) to instruct Open Inventor to show a scene graph inside the render area. This step can be divided into two steps. First, Open Inventor traverses the scene graph to produce a list of shapes each of which has all the properties attached with. The properties are collected from the scene graph or from the default values. Open Inventor, then, passes the list of shapes to OpenGL which renders the list of shapes. The scene graph traversal belongs to PREMO's Virtual environment and the rendering performed by OpenGL belongs to PREMO's Viewing to Realization environments. Other methods in the program are used for giving instructions to X window system. The relationship between PREMO's environment model, the basic steps which Open Inventor performs to realise a static picture and the basic structure of an Open Inventor program are illustrated in Figure 1.3.

2. General structure of an Open Inventor program

In the last section, we analyzed Open Inventor programs which realised simple static pictures, pictures without any interaction or animation. Now we consider the more general Open Inventor program structure in which pictures that support both interaction and animation are specified.

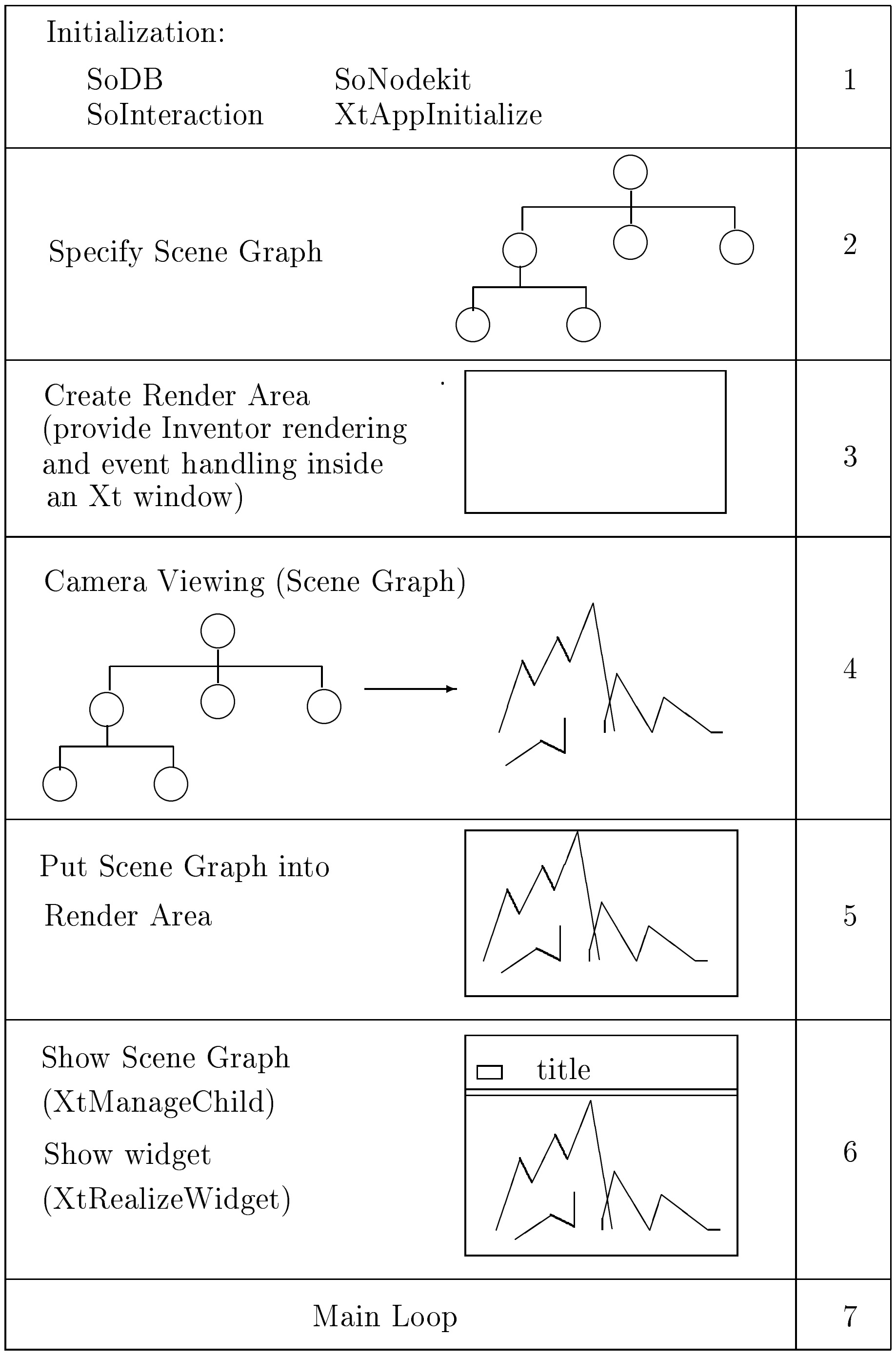

Initialization:
SoDB
SoNodekit
SoInteraction
XtAppInitialize
1
Specify Scene Graph
2
Create Render Area (provide Inventor rendering and event handling inside an Xt window)
3
Camera Viewing (Scene Graph)
4
Put Scene Graph into Render Area
5
Show Scene Graph (XtManageChild)
Show widget (XtRealizeWidget)
title
6
Main Loop
7

Figure 1.2: The basic structure of an Open Inventor program.

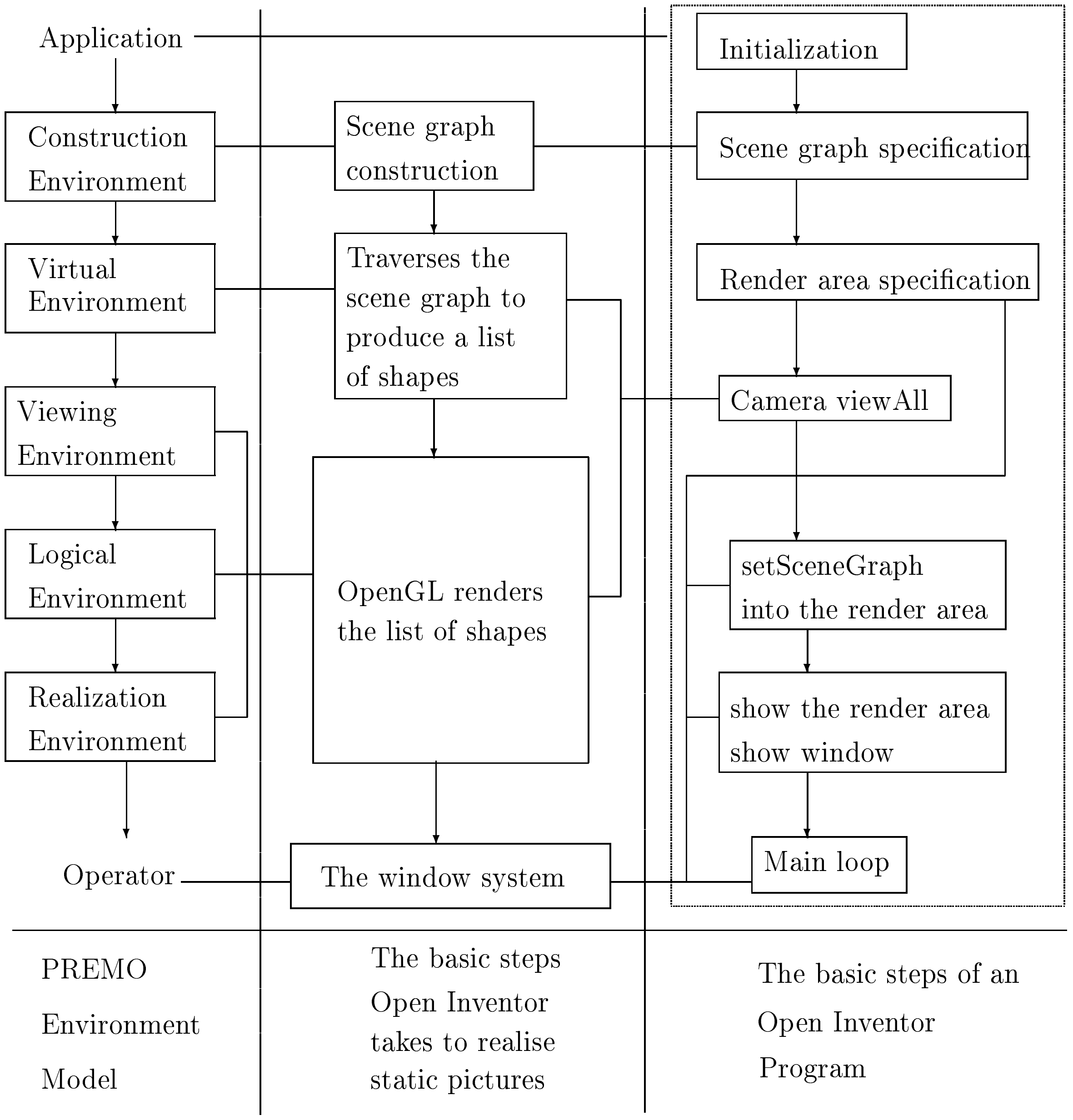

Application
Construction Environment
Virtual Environment
Viewing Environment
Logical Environment
Realization Environment
Operator
Scene graph construction
Traverses the scene graph to produce a list of shapes
OpenGL renders the list of shapes
The window system
Initialization
Scene graph specification
Render area specification
Camera viewAll
setSceneGraph into the render area
show the render area show window
Main loop
PREMO Environment Model
The basic steps Open Inventor takes to realise static pictures
The basic steps of an Open Inventor Program

Figure 1.3:

Widget myWindow = SoXt::init(argv[0]);	
if (myWindow == NULL) exit(1);	1
SoSeparator *root = new SoSeparator; root → ref();	2
SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;	
root → addChild(myCamera);	3
root → addChild(new SoDirectionalLight);	4
SoMaterial *myMaterial = new SoMaterial;	
myMaterial → diffuseColor.setValue(1.0, 0.0, 0.0);	
root → addChild(myMaterial);	5
SoRotationXYZ *myRotXYZ = new SoRotationXYZ;	
root → addChild(myRotXYZ);	6
root → addChild(new SoCone);	7
myRotXYZ → axis = SoRotationXYZ::X;	
SoElapsedTime *myCounter = new SoElapsedTime;	
myRotXYZ → angle.connectFrom(&myCounter → timeOut);	8
SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);	9
myCamera → viewAll(root, myRenderArea → getViewportRegion());	10
myRenderArea → setSceneGraph(root);	
myRenderArea → setTitle("Engine Spin");	11
myRenderArea → show();	
SoXt::show(myWindow);	12
SoXt::mainLoop()	13

Figure 1.4: An action example.

In Open Inventor, both interaction and animation are specified using so called 'actions', events and event handlers. Furthermore, event handling is specified as part of the scene graph, and actions are specified with respect to a node, a path, or a path list in a scene graphs. Consider an example where an *engine* is attached to the angle field of a rotation node in the scene graph specified by the red cone program in Figure 1.1. The engine changes the angle value in the rotation node based on changes of the real time clock, so that the cone will spin. The code of the program is shown in Figure 1.4. From line 2 to line 7, a scene graph is specified. In line 6, a rotation node is added into the scene graph and in line 8, an Elapsed-Time engine is attached to the angle field of the rotation node. The scene graph is illustrated in Figure 1.5.

Open Inventor executes this program by registering an event which occurs periodically according to the speed of the Elapsed-Time engine. Whenever this event occurs, Open Inventor modifies the scene graph (i.e. sets the value of the rotation angle) and then redraws it. This can also be viewed as Open Inventor's general program execution procedure, i.e. maintaining the scene graph and rendering a new version of the scene graph (see in Figure 1.6). Events or actions may cause changes in the scene graph, which implies an automatic redraw of the scene, unless this behaviour has been explicitly suppressed. Open Inventor will also redraw the whole thing when the window is resized or exposed.

Open Inventor's general program execution procedure can be modeled by PREMO's Environment Model and the *Event handler*. A PREMO flow chart which implements Open Inventor's general program execution procedure is illustrated in Figure 1.7. The implementation has the following steps:

1. A scene graph (maybe several scene graphs) is built according to the application program.
2. A set of events are registered into the *EventHandler* according to the scene graph.
3. A list of shapes is formed by the scene graph traversal.
4. OpenGL renders the list of shapes.
5. X-window system realises windows and generates X-window system events.

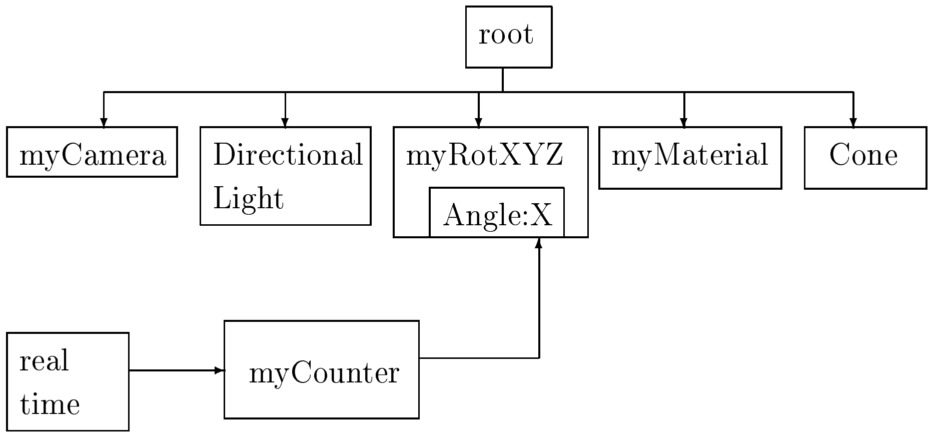


Figure 1.5: The scene graph for the spinning cone example.

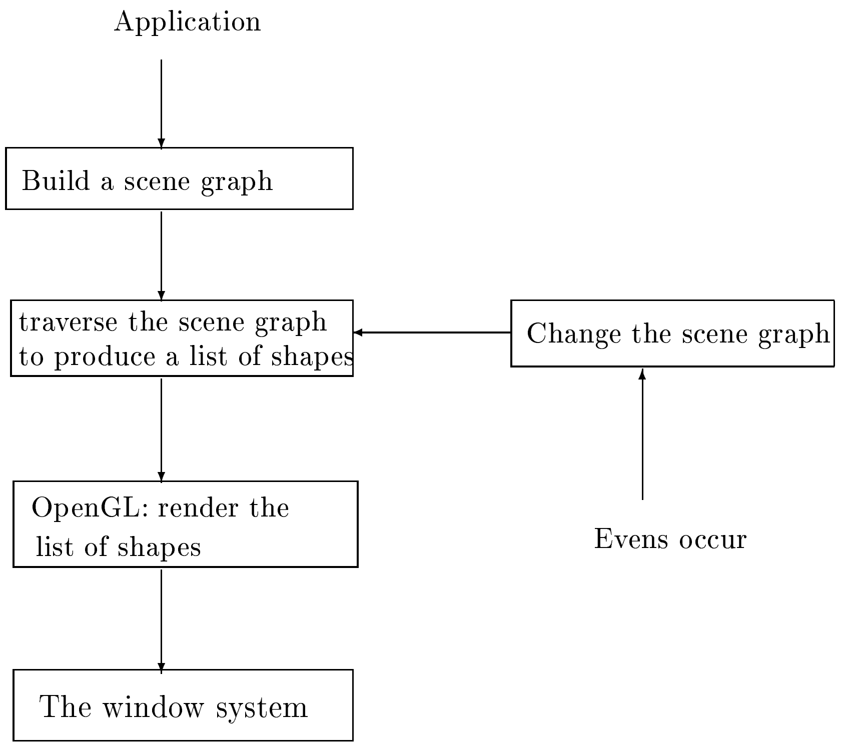

Application
Build a scene graph
traverse the scene graph to produce a list of shapes
Change the scene graph
OpenGL: render the list of shapes
Evens occur
The window system

Figure 1.6: The flow chart of Open Inventor's general program execution procedure.

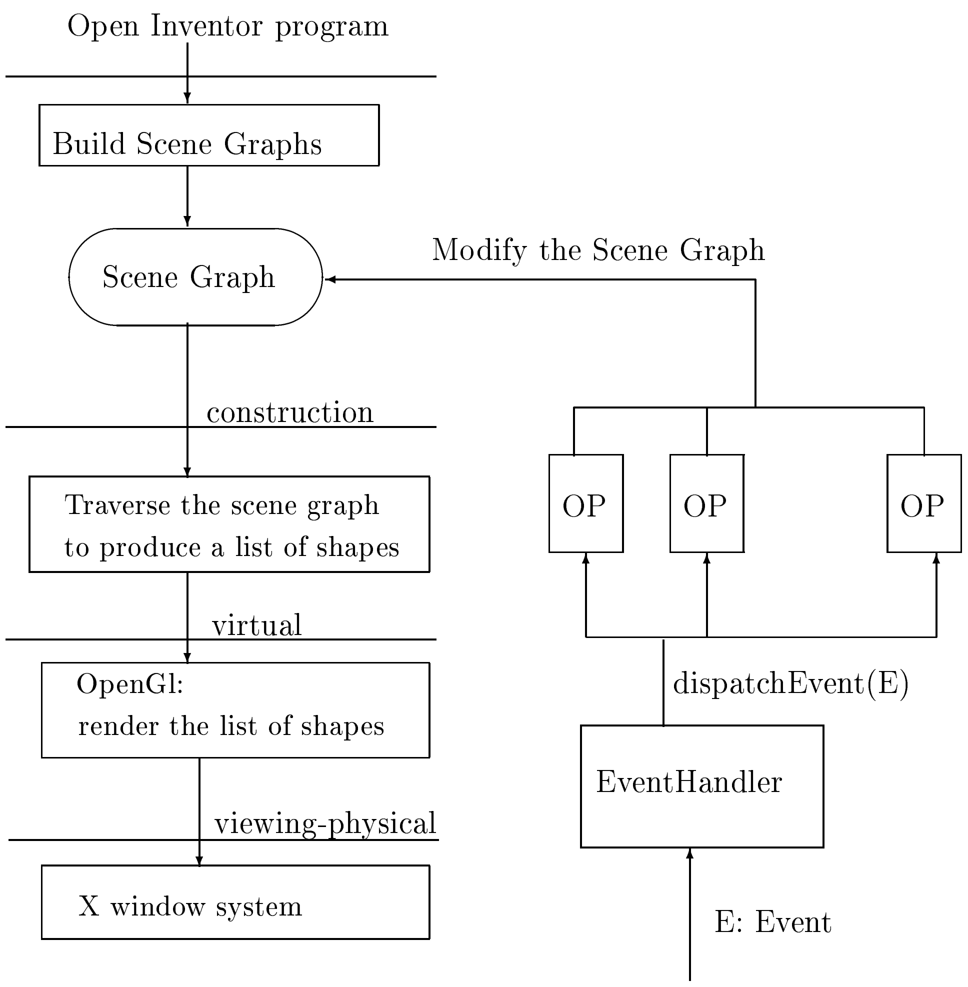


Figure 1.7: Implementing Open Inventor's general program executing procedure.

6. PREMO waits for events. Apart from the events generated by X-window system, there are other events which are generated by nodes in a scene graph or by times (see in Part II).
7. When an event occurs, it is dispatched to a specific operation. The scene graph may be modified by dealing with the event.
8. If the scene graph is changed go to step 3.
9. Otherwise go to step 6.

3. Scene graphs

The scene graph is very important in Open Inventor. In the previous sections, we have studied the program execution procedure of Open Inventor by ignoring most of the details surrounding scene graphs. It was found that Open Inventor's working principle is to maintain scene graphs. An application program can be viewed as a scene graph specification. That the system's execution of the program can be viewed as a loop in which: (1) it interprets the scene graph into computer graphics and (2) it waits for events. When events occur, the system modifies the scene graph and repeats step (1). In this section, the scene graph is studied, especially, with respect to how the render action obtains the properties of graphical objects by traversing a scene graph.

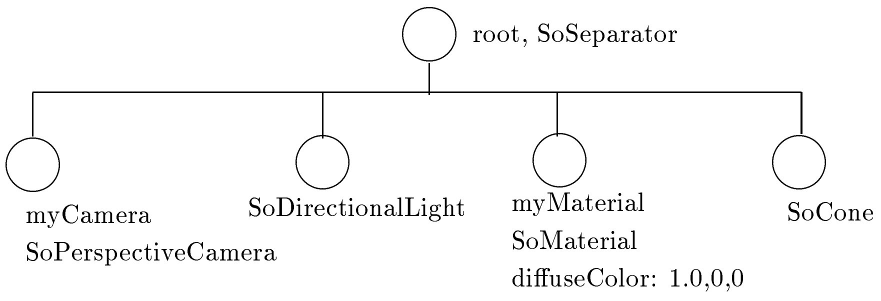


Figure 1.8: The scene graph specified from line 2 to 10 in Figure 1.1.

3.1 What is a scene graph

A scene graph is a directed acyclic graph consisting of nodes. A node is an instance of an Open Inventor class which has methods to assign (retrieve) values to (from) its data members and to connect it with other nodes to form a graph. For example, to add a sphere node into a scene graph, the *new* operator is used to create an instance of *SoSphere*,

$$SoSphere \quad * mySphere \ = \ new \ SoSphere$$

and then values are assigned to its data members (they may have default values), and the *addChild* method (which is defined in the class *SoGroup*) is used to add it into a scene graph.

$$someGroupNode \rightarrow addChild(mySphere)$$

Scene graph nodes can be separated into two categories, terminal nodes and non-terminal nodes. The non-terminal nodes whose classes are derived from *SoGroup* have data members for connecting nodes together to form a graph. The terminal nodes which represent graphical objects and properties have data members for geometric values and properties. In the program in Figure 1.1, lines 2 to 10 specify a scene graph whose appearance in the database may be illustrated in Figure 1.8, where *root* is a non-terminal node and the others are terminal nodes.

3.2 How to traverse a scene graph

Open Inventor implements various actions by traversing scene graphs. The presentation of picture is a special action called a rendering action. In this section, we discuss how Open Inventor traverses the scene graph in order to render pictures.

Open Inventor traverses a scene graph from top to bottom and left to right. For instance, the order of traversing the scene graph in Figure 1.8 is: *root, myCamera, SoDirectionalLight, myMaterial* and *SoCone*. During this traversal, geometric properties are accumulated until a shape node is traversed. The shape's appearance is then determined by the properties accumulated up to that point. If a property is not in the accumulated set, a default value is used. Thus the order of the nodes within the scene graph is important in Open Inventor. For example, if the *SoCone* node and *myMaterial* node are swapped in Figure 1.8, the *myMaterial* node will not affect the cone at all, the colour of the cone will be white (the default colour) instead of red, which is specified in the *myMaterial* node. There are some nodes which are independent from each other, such as *SoMaterial* and *SoLight*. For such nodes, it does not matter in which order they are arranged. An *SoCamera* node should be in front of all other nodes that are to be placed in the camera's view area and an *SoShape* node should be after all other nodes that determine the properties of the shape.

Considering rendering traversal, nodes can be divided into three kinds, i.e. *Shape* nodes, *Property* nodes and *Group* nodes. In Figure 1.8, *root* is a *Group* node, the node whose type is *SoCone* is a *Shape* node, and all the others are *Property* nodes. A *group* node is a container for collecting child objects. In a scene graph, *property* nodes and *shape* nodes are terminals in the sense that they have no children and all non-terminal nodes belong to *group* nodes. Group nodes are divided into a set of group node classes, each with a special grouping characteristic (see Figure 1.9).

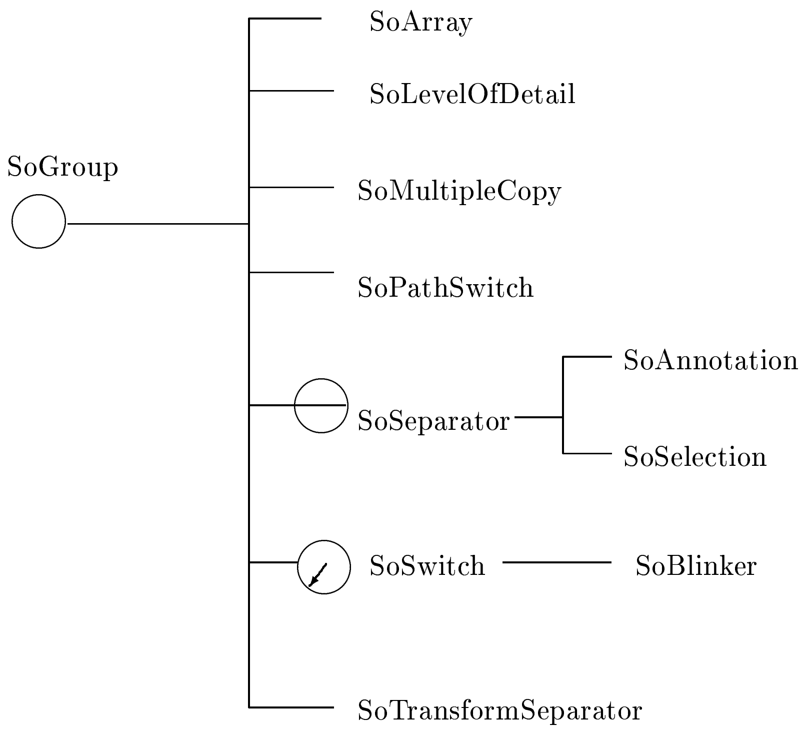


Figure 1.9: The Group-Node classes.

During traversal of a scene graph, all the children under *SoGroup, SoSeparator, SoArray* and *SoMultipleCopy* nodes will be traversed. One child (or some children) will only be traversed if they are under *SoLevelOfDetail, SoPathSwitch, SoSwitch* and *SoBlinker*. *SoSelection* is used for interactively dealing with events and is discussed in detail in Part II.

During traversal, properties specified in property nodes are accumulated, replaced or concatenated. A scene graph should contain one and only one active camera. When a camera node is encountered during rendering traversal, a *view volume* is created based on the data members of the camera node. When a shape node is traversed, the shape is calculated against the view volume. If it is outside of the view volume, it is thrown away. If part of it outside of the view volume, it is clipped. The current collection of properties is combined with the properties specified in the shape node, and together they determine the appearance of the shape. However, property nodes under a *SoSeparator* node only affect the shapes which are under the same *SoSeparator* node. Children under *SoArray* or *SoMultipleCopy* nodes will be traversed several times according to the specification, and several copies may be made.

During scene graph traversal, the data base can be thought of as managing the following data structures:

- *TS* (*Traversal State*), a collection of elements or parameters which determine the properties of graphics applied during traversal.
- *SS* (*State Stack*), used to keep the previous value of a traversal state. For example, when a SoSeparator node is encountered during traversal, the current traversal state is pushed into the State Stack and when the traversal of the SoSeparator node is finished, the traversal state in the top of the stack is poped out ar the current traversal state.
- *SL* (*Shape List*), used to store each geometrical shape associated with a viewport.

The way that Open Inventor conceptually traverses a node N is illustrated in Figure 1.10. The shape list produced by traversing the scene graph in Figure 1.8 can be illustrated in Figure 1.11, where there is only one element in the list.

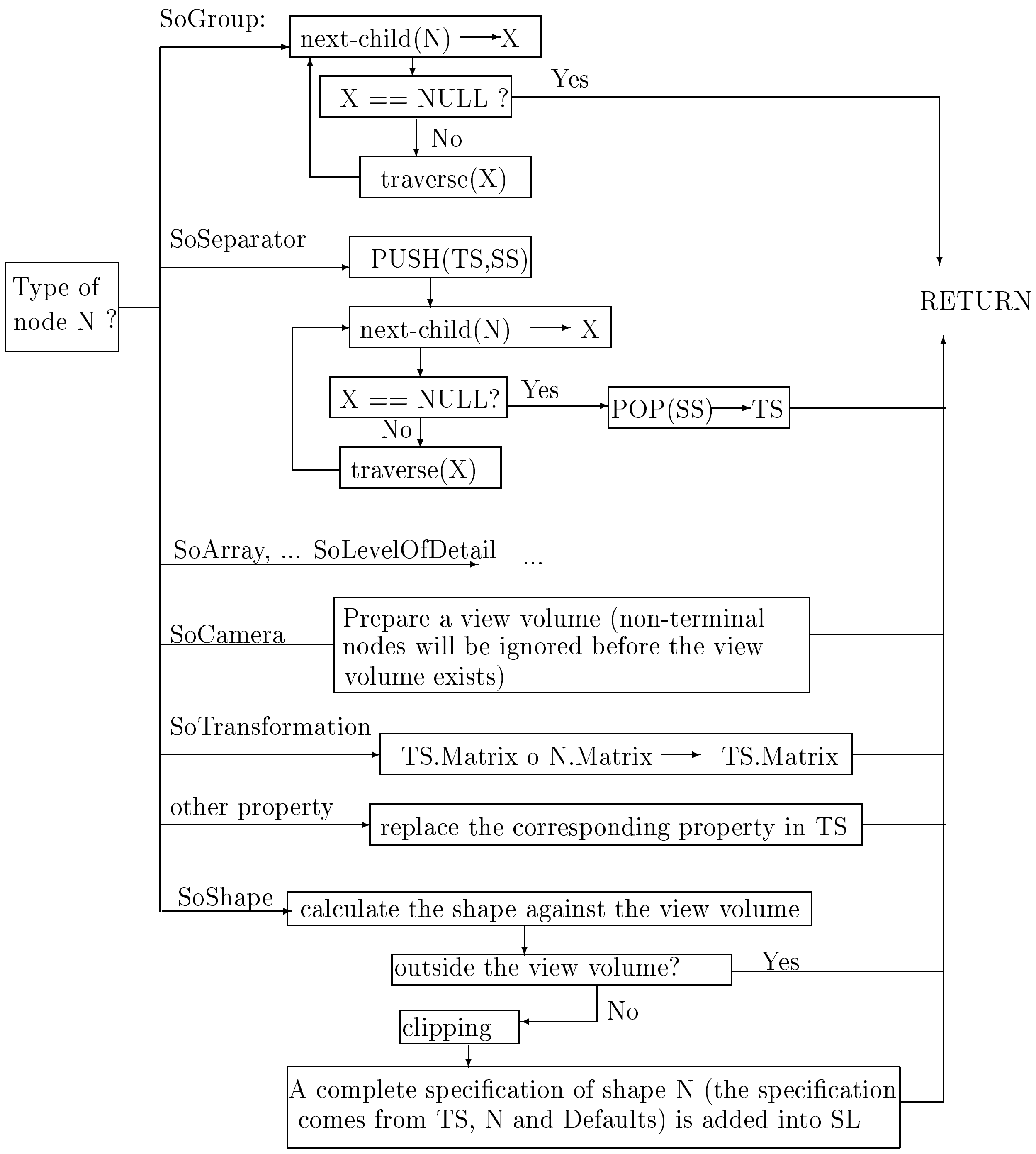

SoGroup:
next-child(N) → X
X == NULL ?
Yes
No
traverse(X)
RETURN
Type of node N ?
SoSeparator
PUSH(TS,SS)
next-child(N) → X
X == NULL?
Yes
No
POP(SS) → TS
traverse(X)
SoArray, ... SoLevelOfDetail
...
SoCamera
Prepare a view volume (non-terminal nodes will be ignored before the view volume exists)
SoTransformation
TS.Matrix o N.Matrix → TS.Matrix
other property
replace the corresponding property in TS
SoShape
calculate the shape against the view volume
outside the view volume?
Yes
No
clipping
A complete specification of shape N (the specification comes from TS, N and Defaults) is added into SL

Figure 1.10: Traversing node N in a scene graph. TS is the Traversal State, SS is the State Stack, SL is the Shape List and $\circ$ is an operator for matrix composition.

name:	Cone	translation:	0 0 0	ambientColor:	.2 .2 .2	model:	Directional	...
parts:	ALL	rotation:	0 0 1 0	diffuseColor:	1 0 0	intensity:	1	
radius:	1	scaleFactor:	1 1 1	specularColor:	0 0 0	color:	1 1 1	
height:	2	center:	0 0 0	emissiveColor:	0 0 0	direction:	0 0 -1	
...		...		...		...		

Figure 1.11: The shape list produced by traversing the scene graph in Figure 1.8 contains one element which is the specification of a cone.

main(int , char **argv)	
{	
Widget myWindow = SoXt::init(argv[0]);	1
if (myWindow == NULL) exit(1);	
SoSeparator *root = new SoSeparator;	2
root → ref();	3
SoMaterial *myMaterial = new SoMaterial;	4
myMaterial → diffuseColor.setValue(1.0, 0.0, 0.0);	5
root → addChild(myMaterial);	6
root → addChild(new SoCone);	7
SoXtExaminerViewer *myViewer =	8
new SoXtExaminerViewer(myWindow);	
myViewer → setSceneGraph(root);	9
myViewer → setTitle("Examiner Viewer");	10
myViewer → show();	11
SoXt::show(myWindow);	12
SoXt::mainLoop();	13
}	

Figure 1.12:

3.3 When a scene graph is changed

Gaining an understanding of how Open Inventor traverses scene graphs is an important factor in understanding how Open Inventor works. In Open Inventor, almost every action is related to manipulating scene graphs. If the scene graph is changed, Open Inventor will traverse it. After traversal, a *Shape List* conceptually exists and based on this shape list, a picture can be realised.

In this section, we analyze an example to see why, how and when a scene graph is changed, and what will result from such a change.

Consider the Open Inventor program shown in Figure 1.12. Execution of this program will produce a red cone shown in a window that is decorated by a set of buttons, which provide some interactions on the picture. Lines 2 to 7 specify a scene graph as shown in Figure 1.13. However, the scene graph for the program is automatically extended through the addition in line 8 of an *SoXtExaminerViewer*. The extended scene graph looks like that shown in Figure 1.14.

During the execution of the program, relevant events will effect changes to the scene graph. For example, the last button on the right side of an examiner viewer window (see page 28 in [13]) is an icon which represents the type of the camera. By clicking that button, the camera will switch between a perspective view and an orthographic view. Suppose the user clicked this button. The system's response to this event is: first, change the scene graph from the one in Figure 1.14 to the scene graph in Figure 1.15 (where the second child is changed from a perspective camera to an orthographic camera), second, traverse the scene graph in Figure 1.15 and redraw the cone.

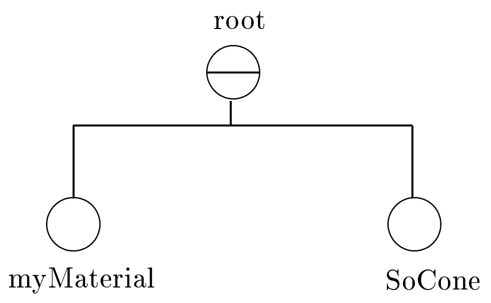


Figure 1.13: The scene graph specified by the application program in Figure 1.12.

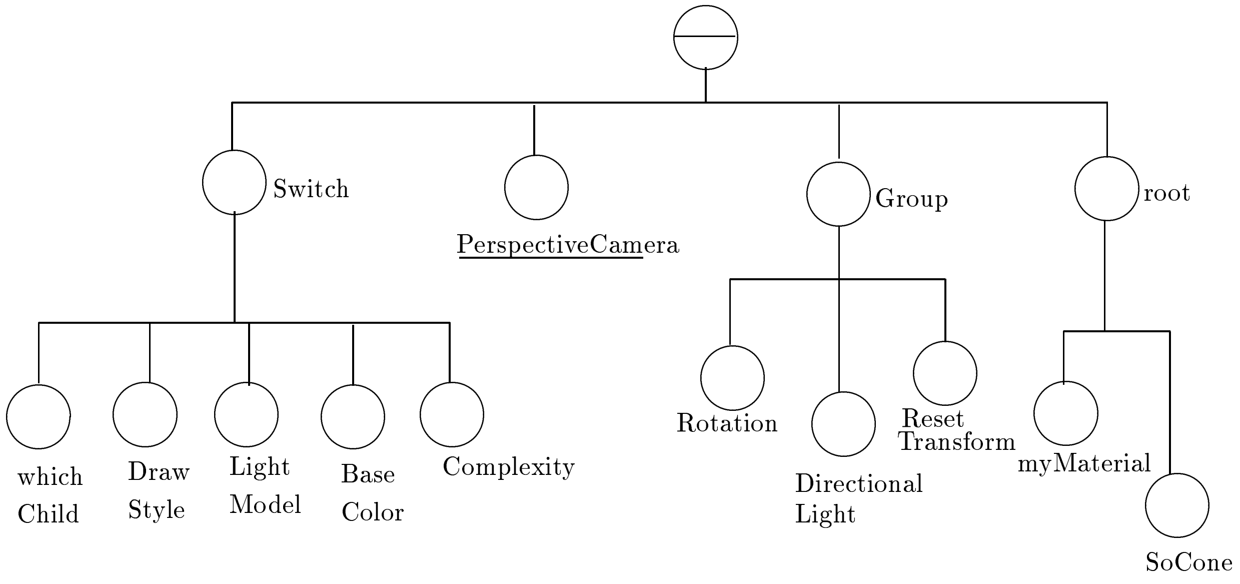


Figure 1.14: The scene graph enriched by the *SoXtExaminerViewer* node.

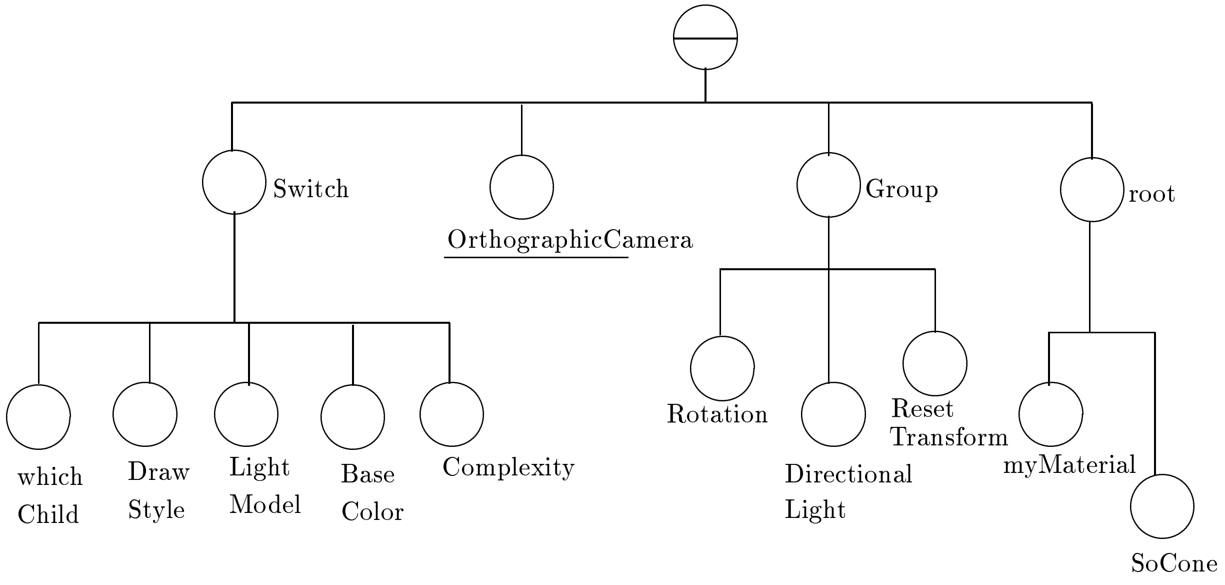


Figure 1.15: The second child, *PerspectiveCamera*, is replaced by an *OrthographicCamera* node.

4. Open Inventor and PREMO: general structure

In this section, a clearer illustration is given to show the representation of Open Inventor in PREMO's environments.

Given an Open Inventor program, a scene graph can be built according to the program and this step results in a scene graph represented as the *Model.* Terminal nodes in a scene graph can be represented as PREMO's basic features, e.g., a shape node is represented as PREMO's *SpatialProperty* and a light node as *VisualProperty.* The PREMO representation of non-terminal nodes and the structure of scene graphs will be discussed in Part IV. The process which builds the scene graph is the *Preparation* process in PREMO's construction environment. Traversal of the *Model* (scene graph) conceptually takes place in the virtual environment, in that the maintenance of the traversal state(TS) and state stack(SS) can be considered to be carried out by the production process and kept as part of the virtual environment state. The resulting shape list(SL) is represented by the *Scene* in the virtual environment. More specifically, the element in the list is represented as PREMO's *OutputPrimitive* which is an aggregation of the basic features. Thus the shape list is of type *List[OutputPrimitive]* (see PREMO part 2 for generic type *List[E::PREMOObject]*). The *production* process notationally performs selection of scene graph nodes, combination of properties and production of entity primitives with appropriately defined properties for each presentable shape.

When we gave a general impression of Open Inventor and PREMO in Figure 1.7, we said that Open Inventor waits for events in certain point. In next part, we will see that the events can occur in the construction environment and can also come from the window system. Information carried with window system events is called *input.* In a similar fashion to output, input in Open Inventor undergoes a number of readily identifiable stages, through which low level events are first translated into logical Open Inventor events. These are then correlated with the scene graph by Open Inventor's scene manager. When a window specific event occurs, it is translated into an Open Inventor event which is then passed to the scene manager. It in turn responds to the event in a number of ways, depending on the actions specified by the current program. For example, if there is a *SoSelection* node in the scene graph, the location will be calculated to determine a selected object. However, if, for example, in an examiner view window the user manipulates the virtual trackball button, the location will be used to determine the transformation instead of object selection. If object selection is needed by the application, a *Selection List* is maintained by the *SoSelection* node (see Part II).

Open Inventor's general structure in PREMO's environment model is illustrated in Figure 1.16, where the inputpipe is demonstrated by Open Inventor's object selection and will be explained in Part II.

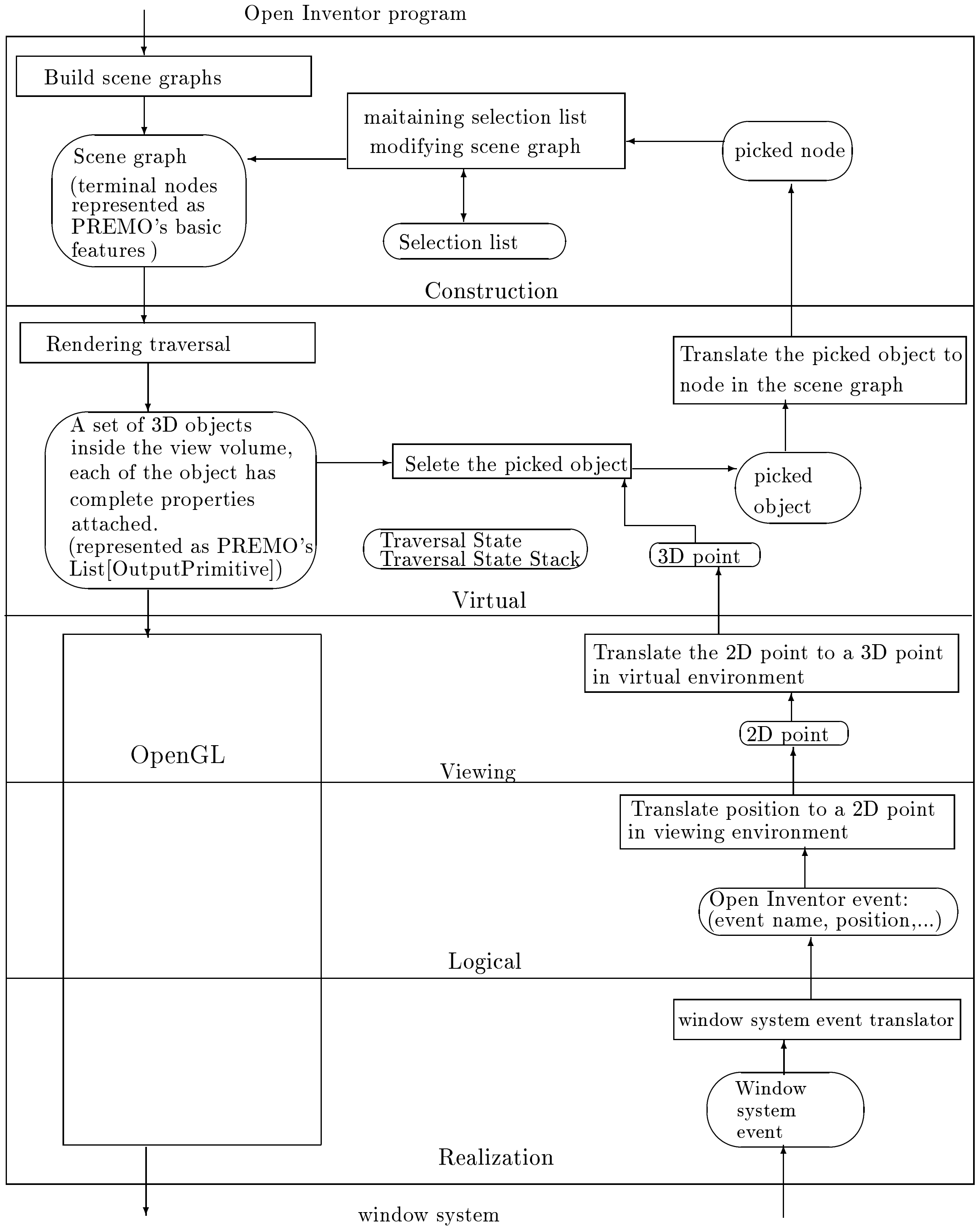

Open Inventor program
Build scene graphs
maitaining selection list
modifying scene graph
picked node
Scene graph
(terminal nodes represented as PREMO's basic features)
Selection list
Construction
Rendering traversal
Translate the picked object to node in the scene graph
A set of 3D objects inside the view volume, each of the object has complete properties attached.
(represented as PREMO's List[OutputPrimitive])
Selete the picked object
picked object
Traversal State
Traversal State Stack
3D point
Virtual
Translate the 2D point to a 3D point in virtual environment
2D point
OpenGL
Viewing
Translate position to a 2D point in viewing environment
Open Inventor event:
(event name, position,...)
Logical
window system event translator
Window system event
Realization
window system

Figure 1.16: Representing Open Inventor in PREMO's environment model. The data flow in the input side is demonstrated by Open Inventor's object selection.

Chapter 2
Events

In this Part, events and the event mechanisms provided by Open Inventor are studied. We also discuss whether or not PREMO can model these conveniently.

Open Inventor has two kinds of events, those created by hardware devices (Open Inventor has an event class tree to model this kind of events) and those created by changes of data and time. In the following, we first consider the event classes and then look at data and time events.

1. Open Inventor event classes

The event classes of Open Inventor are illustrated in Figure 2.1. Each *SoEvent* instance contains the following information: type identification and time, cursor position and state of the modifier keys (Shift, Control, Alt) when the event occurred. Instances of the subclasses of *SoEvent* contain additional information, e.g. whether the button was up or down when the event occurred for the instance of *SoButtonEvent*.

Open Inventor's event classes are independent from the window system's. When Open Inventor is implemented on the top of a specific window system an event translator has to be written, e.g. an X window event translator. The event translator convents window system specific events into Open Inventor events.

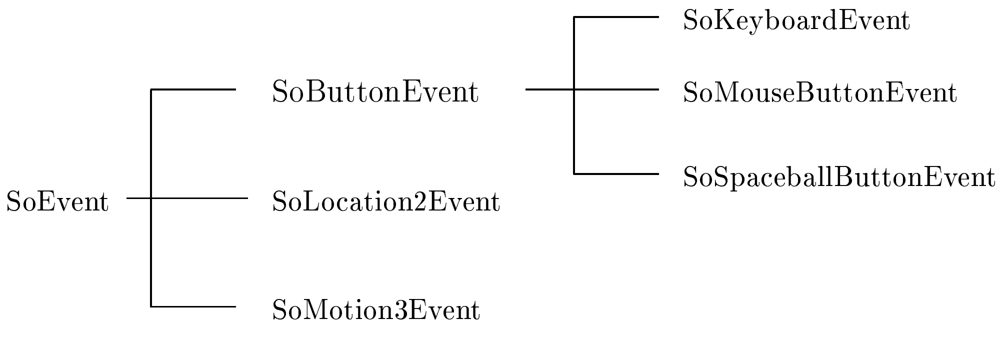


Figure 2.1: The Open Inventor event classes.

2. Open Inventor event handling

Open Inventor provides four mechanisms to handle events (see the illustration in Figure 2.2). All of the mechanisms, with the exception of the application event handler mechanism, work by placing

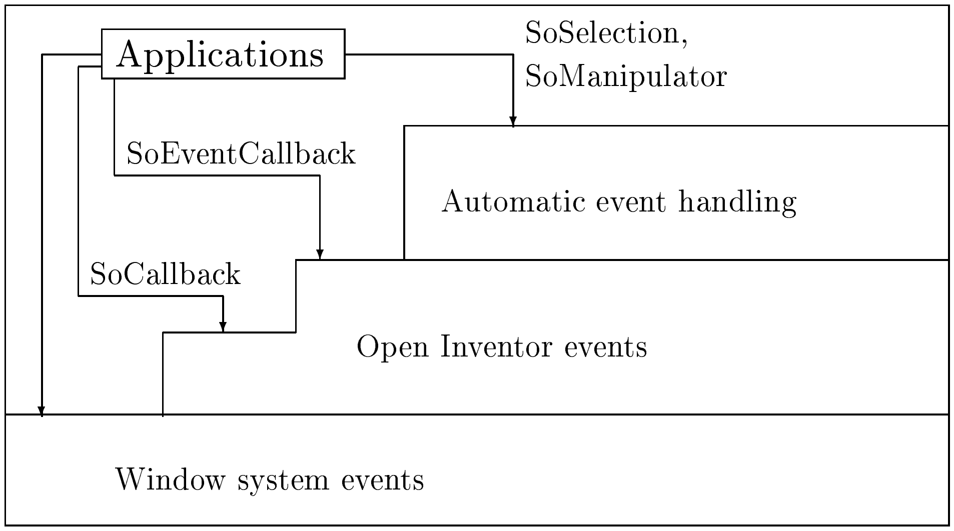


Figure 2.2: The four possible ways to handle events in Open Inventor.

particular nodes into a scene graph. Since Open Inventor (conceptually) traverses the scene graph whenever an event occurs, Open Inventor can take appropriate action if it encounters an event handling node. The first mechanism allows users to handle the window system events directly, e.g. directly handle Xt events. If this mechanism is chosen, the window system events are directly sent to the event handling program specified in conjunction with the application program. For example, the following statements in the program:

$$SoXtRenderArea * myRenderArea;$$

$$myRenderArea \rightarrow setEventCallback(myAppEventHandler, userData)$$

indicate that when an X window event occurs, the event handler *myAppEventHandler* will be invoked with the untranslated event and *userData* as its input.

The second mechanism that Open Inventor provides is encapsulated by the class *SoCallback*. Instances of *SoCallback* can be specified as a node in a scene graph. Each time the scene graph is traversed, the program specified in these nodes will be invoked. More specifically, the program will be invoked as each *SoCallback* node is encountered during traversal. The *SoCallback* mechanism places no restrictions on the kind of action the specified program can take, and is always invoked if presents in the scene graph.

The third mechanism is encapsulated in a class *SoEventCallback*. *SoEventCallback* differs from *SoCallback* in that the program specified in the *SoEventCallback* node is called only during the specified event processing, whereas the program specified in the *SoCallback* node is always invoked whenever that node is traversed. Furthermore, *SoEventCallback* provide some additional methods that are useful for handling events.

Finally, Open Inventor provides a mechanism for automatic event handling via special nodes. The first three mechanisms are straight forward. In the following sections, the automatic event handling mechanism is studied.

3. Automatic event handling

When a window system event occurs, it is first translated into an Open Inventor event (unless an application event handler is used). Open Inventor creates an instance of *SoHandleEventAction* and then applies the instance to the top node of the scene graph or at a particular node. The action traverses the scene graph from top to bottom and left to right until it meets a node which handles the current Open Inventor event.

Open Inventor provides a group of "smart" nodes designed to handle events automatically: *SoManipulator* and its subclasses and *SoSelection*.

3.1 SoSelection

When the handle event action is applied to an *SoSelection* node, it traverses its children from left to right, asking each child to handle the event. After each child, it checks to see if the event was handled. If it was, the handle event action ends its traversal of the scene graph. If the event was not handled, the *SoSelection* node itself handles it.

The *SoSelection* node provides several features that relate to the topic of user interaction. These features include managing the selection list, highlighting the selected objects, and the use of user-written callback functions that are invoked when the selection list changes.

When the *SoSelection* node starts to handle an event, it calls *getPickedPoint()*, a method in *SoEventHandleAction*, to get the picked node and, based on the picked node and previous selection list, it determines the current selected nodes. There is a set of choices which the application can specify to indicate how a *SoSelection* node should manage the selection list. The *SoSelection* node manages the selection list based on the current picked object, the old selection list and the application's specification (or the default strategy). If an object is selected, the *SoSelection* node will insert a bounding box or a wireframe, with the properties which are relevant to the selected object (e.g. the size and position of the bounding box is determined by the size and position of the selected object), in the scene graph so that it will appear on the top of the selected object. Since the scene graph has been changed, the system will automatically apply the *SoGLRenderAction* to the top node of the scene graph so that the selected object is then highlighted by a bounding box (wireframe). If an object is deselected, the selection node will delete the corresponding bounding box (wireframe) from the scene graph. This also causes the system to apply the render action and hence stop to highlight the object. If the application specified a callback function, the selection node will invoke the function based on whether the current event is selection, deselection or just picking and the application's specification. Figure 2.3 illustrates the changes of the scene graph during selection and deselection of a cone.

The data flow of selections in PREMO's environment model is illustrated in Figure 1.16. An window system event is stored in the *Lexeme store* which is translated into an Open Inventor event stored in the *Information store*. The window system event translator is the *Accumulation*. An *SoSelection* or (*SoManipulator* node) calls *getPickedPoint()* method to get the picked node. The *getPickedPoint()* is a method in the handle event action class (*SoHandleEvent Action*). It takes the data of the Open Inventor event (i.e. the data in the *Information store*) and returns a picked node which is in *Instruction Store*. There are several ways in which the *getPickedPoint()* method can be implemented in PREMO's environment model. For instance, it translates the *cursor position*, which is part of the event data and is a 2D point relative to the left lower corner of the window, to a point of the coordinate system in the viewing environment. The translation process is the *Abstraction* and the translated point is stored in the *Selection Store*. Through the *Elevation*, the point in the *Selection Store* is further translated into a 3D point stored in the aggregation store in the virtual environment. In the virtual environment, the manipulation process takes the point in the aggregation store and the shape list to determine which object was hit. The picked object is sent to the *Directive Store* and then is translated into the node in the scene graph. It is also possible to select a picked 2D object in the viewing environment and then determine which 3D object related to it. The functionalities provided by *SoSelection*, such as maintaining the selection list, changing the scene graph, belong to the *manipulation* process in the construction environment.

3.2 SoManipulator

Manipulators employ *draggers* to respond to user events and edit themselves. There are two groups of manipulators in Open Inventor. One group provides a set of subclasses of *SoTransform* which allow interactive transformation of geometric shapes. The other group provides a set of subclasses of *SoLight* which allow interactive adjustment of the position of the light in the scene. In the following, a simple example is used to explain how a manipulator handles events.

The manipulator example is shown in Figure 2.4. Upon execution of this program, a trackball manipulator appears as three rings around a cone. When the left mouse button is pressed on the

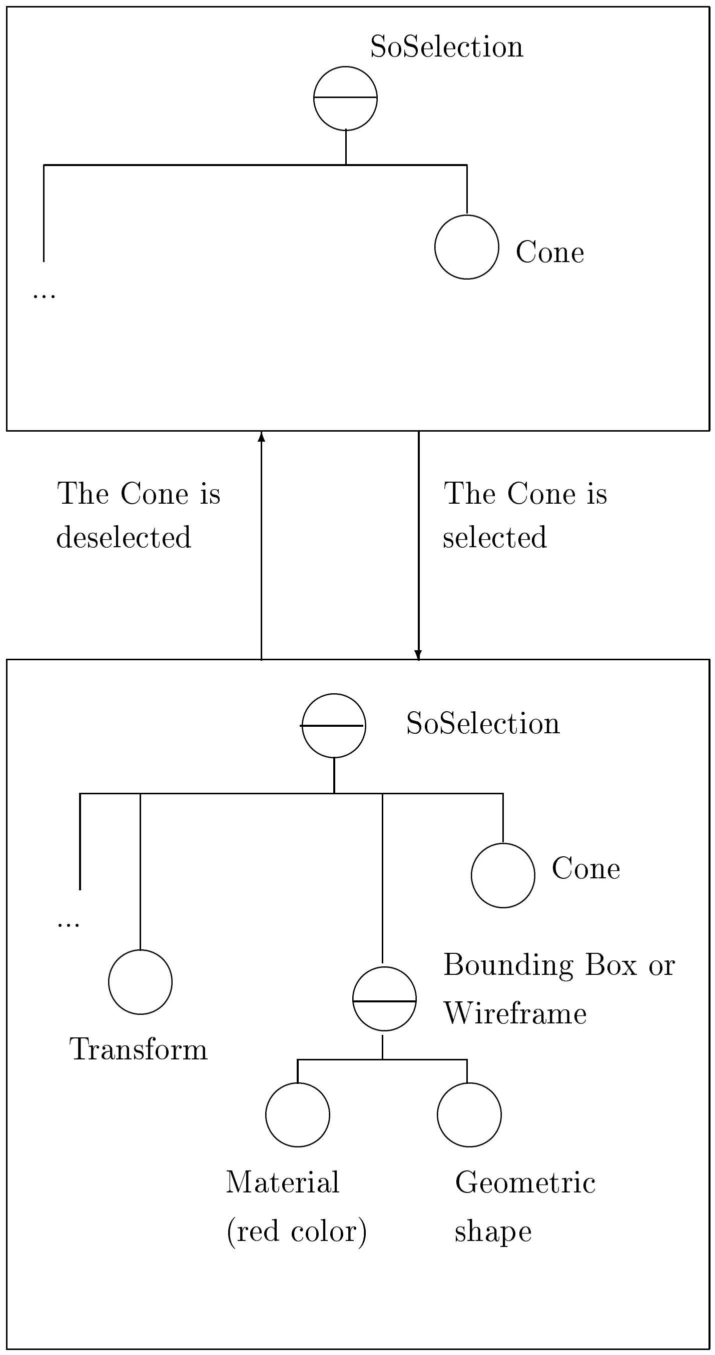


Figure 2.3: The changes of the scene graph when the cone is selected and deselected.

main(int , char **argv)
{
Widget myWindow = SoXt::init(argv[0]);
if (myWindow == NULL) exit(1);
	1

SoSeparator *root = new SoSeparator; root →ref();	2

SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
root → addChild(myCamera);
root → addChild(new SoDirectionalLight);
	3

root → addChild(new SoTrackballManip);	4
root → addChild(new SoCone);	5

SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
myCamera → viewAll(root, myRenderArea → getViewportRegion());
myRenderArea → setSceneGraph(root);
myRenderArea → setTitle("Trackball");
myRenderArea → show(); SoXt::show(myWindow);
	6

SoXt::mainLoop();
}
	7

Figure 2.4: A manipulator example.

trackball, it highlights itself to show it is active. While it is active, the mouse can be used to rotate the trackball as well as the cone inside the trackball.

A scene graph constructed from the program in Figure 2.4 is shown in Figure 2.5a. However, there is another part in the scene graph called the dragger, which is not reachable by an application (called a *hidden child* of the manipulator node). This part is the geometric shape of the trackball and its material properties. The complete scene graph looks like the graph in Figure 2.5b, where the part in the box is unreachable from the application. That part is rooted by a special node that can be thought of as a *hidden* node. When it is traversed, the properties under such a node have no affect on outside nodes (this is the same as a separator node in this respect). The initial values in the transform node and the original size of the trackball is determined by calculating the bounding area occupied by all the geometric shapes appearing in the domain of the manipulator node (here, it is the cone).

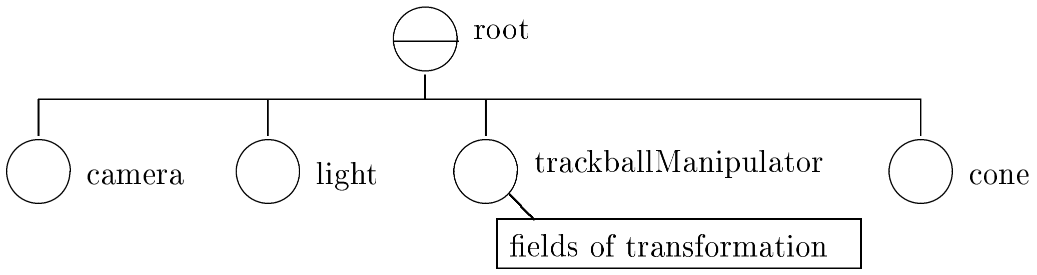
When a left button press event occurs, the scene graph is traversed. When the trackball manipulator node is reached, it first checks whether or not the cursor position is inside the bounding area of the trackball and if yes, where it is. If the trackball is selected, the manipulator node will modify the material node so that the color of the trackball is changed (highlight). Then the manipulator node will grab the following mouse motion events. (In Open Inventor, a node can request that all subsequent events be sent directly to it until further notice. This behaviour is called *grabbing*.) While the manipulator receives mouse motion events it modifies the transform node according to the position of the mouse and the selected rings of the trackball. Since the scene graph is changed, the render action will be applied to the scene graph, which transform both the trackball and the cone. If at the same stage a mouse release event is received, the manipulator modifies the material node so that the color of the trackball is changed back and stops grabbing events.

All other manipulators work in the same way, but each has a different dragger and applies different transformations (e.g. trackball is for rotation, handlebox is for independent translation in one and two dimensions, as well as scaling in one and three dimensions, and so on).

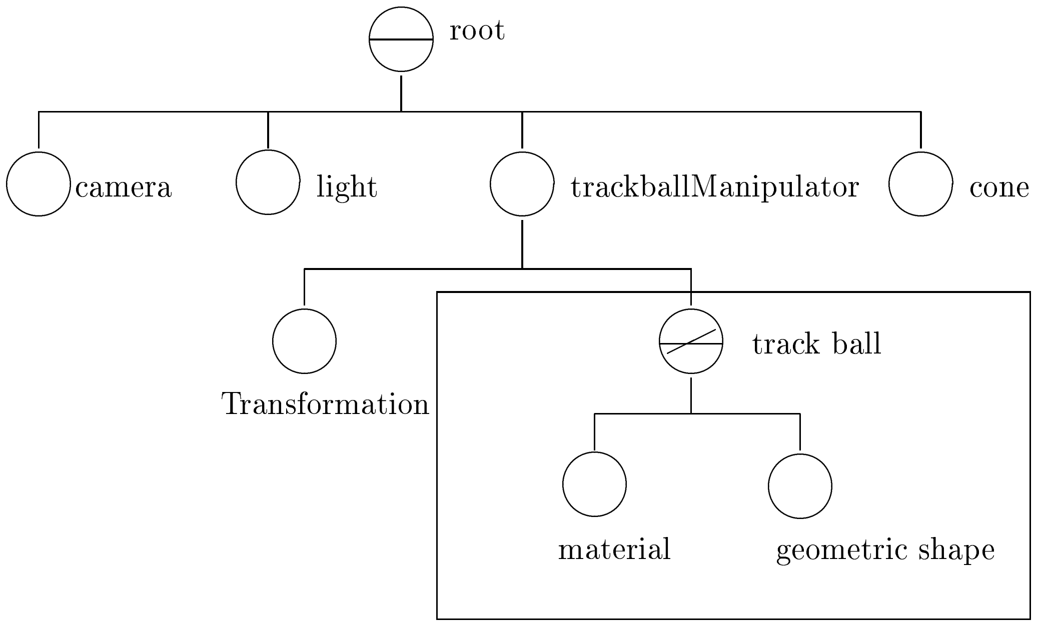
3.3 Draggers

While every manipulator node uses a dragger, it is also possible for an application to directly use a dragger. For example, if line 4 in Figure 2.4 is replaced by:

$$root \rightarrow addChild(new\ SoTrackballDragger)$$



(a)



(b)

Figure 2.5: (a) is the scene graph which can be accessed and modified by users and (b) is the whole scene graph.

then a trackball and a cone will be drawn. However, only the trackball will respond to mouse events. In particular, when the trackball is selected, it highlights itself and will rotate following the movement of the mouse, but the cone will be unaffected by this interaction. The scene graph, which can be reached by the application, is shown in Figure 2.6a. The whole scene graph notionally looks like the one in Figure 2.6b. Draggers and manipulators differ in their scene graphs in that the transformation node of a dragger is also rooted by the special node so that it only transforms the dragger.

4. Data and time events

In Section 1, Open Inventor's event classes were introduced. These events are generated by hardware devices. Open Inventor also supports events that arise from changes of data and time. Such events occur within Open Inventor, but are not strictly part of the event model.

Data events come from changes of the data in scene graphs. Every kind of change of a scene graph can be specified as an event, for example, if a particular data member in a node is changed, or a node is added or deleted. Time events are alarm, interval, elapsed time, global time and so on.

In the following, we study how to specify time and data events and what mechanisms are supplied by Open Inventor for dealing with such events.

5. Sensors and engines

Open Inventor has two kinds of mechanisms, termed sensors and engines, that can be used to specify data and time events and responses to these events.

The method *attach* in data sensor classes and the method *connectFrom* in some engine classes, can be used to specify data and time events. (Each type of time sensor and time related engine corresponds to a particular time event). Since a sensor can be attached to a data member of a node, a node, or a path, then every change in the data member, the node or the path can result in an event.

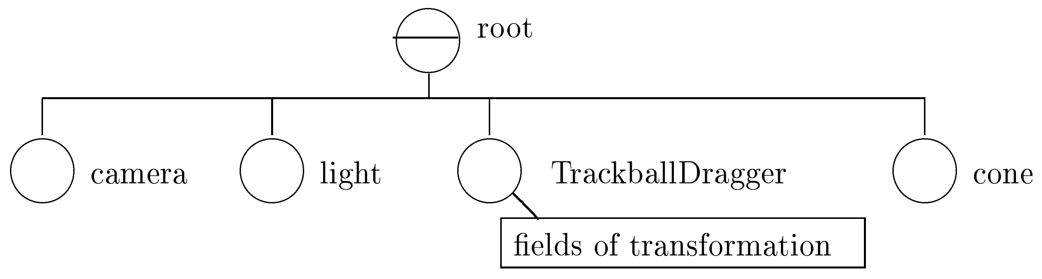
Sensors call on application-defined callback functions, while engines use system built-ins. Consequently, if engines are used, everything is carried out automatically, but the functionality is fixed. If sensors are used, callback functions must be provided that deal with the event, but any kind of event can be handled in an application defined way.

Open Inventor maintains two queues for data and time events, the *Timer* queue and the *Delay* queue. When a specified time is reached, the corresponding time sensor is added into the Timer queue. When the data is changed, the corresponding data sensor is added into the Delay queue. The ordering of sensors in the Timer queue is naturally formed by the value of the time. The order in the Delay queue is formed according to priorities which can be specified by the application. The system processes all the events in the three queues (the Timer and Delay queues and the hardware event queue) at regular intervals. Processing the data and time events invokes the related callback functions.

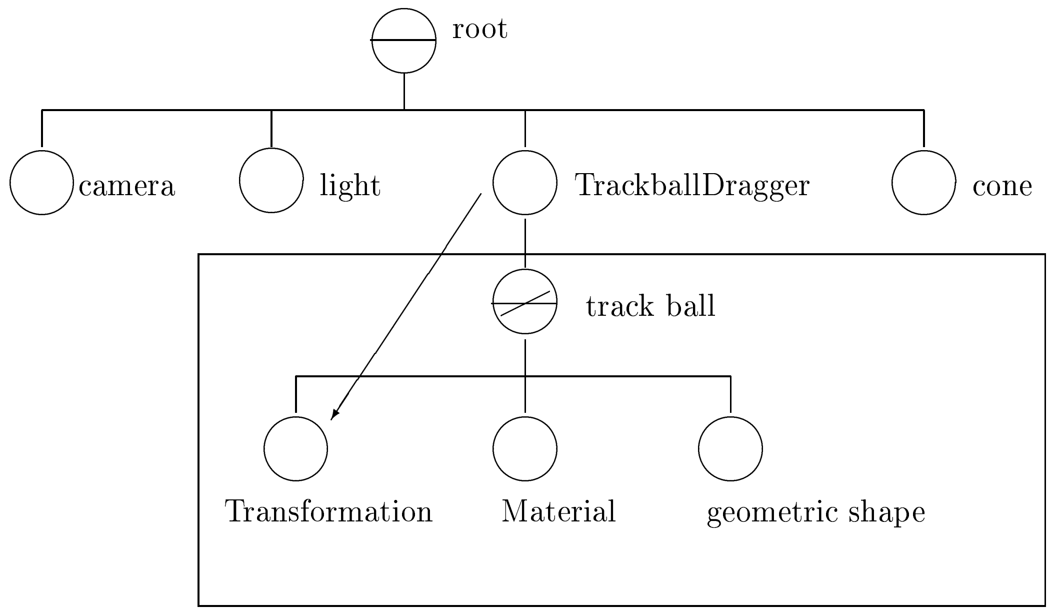
The sequence of scheduling events in the queues depends on the window system. For the X window system, the sequence can be described as follows.

```
SoXt main loop calls XtAppMainLoop
   BEGIN:
   if there is an event waiting:
      process all pending timers.
      process the delay queue if the delay queue time-out is reached.
      process the event.
      Go back to BEGIN.
   else (no event waiting)
      if there are timers,
         process timers.
         Go back to BEGIN.
   else (no timers and no events)
      process delay queue.
      Go back to BEGIN.
```

In Part I Section 2, we analyzed an example where an engine is attached to the angle data member



(a)



(b)

Figure 2.6: (a) is the scene graph which can be accessed and modified by applications and (b) is the whole scene graph. The arrow from TrackballDragger to Transformation in (b) means that though the transformation node is rooted by the special node, it can be reached by the application.

Widget myWindow = SoXt::init(argv[0]);	
if (myWindow == NULL) exit(1);	1
SoSeparator *root = new SoSeparator; root → ref();	2
SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;	
root → addChild(myCamera);	3
root → addChild(new SoDirectionalLight);	4
SoMaterial *myMaterial = new SoMaterial;	
myMaterial → diffuseColor.setValue(1.0, 0.0, 0.0);	
root → addChild(myMaterial);	5
SoRotationXYZ *myRotXYZ = new SoRotationXYZ;	
root → addChild(myRotXYZ);	6
root → addChild(new SoCone);	7
myRotXYZ → axis = SoRotationXYZ::X;	
SoElapsedTime *myCounter = new SoElapsedTime;	
myRotXYZ → angle.connectFrom(&myCounter → timeOut);	8
SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);	9
myCamera → viewAll(root, myRenderArea → getViewportRegion());	10
myRenderArea → setSceneGraph(root);	
myRenderArea → setTitle("Engine Spin");	11
myRenderArea → show();	
SoXt::show(myWindow);	12
SoXt::mainLoop();	13

Figure 2.7: The spine cone program.

of a rotation node. The engine changes the angle value in the rotation node based on the changes of the real time clock, so that the cone will spin. Now we re-analyze this example to gain an understanding of how Open Inventor supports such animations. The program and its scene graph are shown in Figures 2.7 and 2.8, respectively. *SoElapsedTime* is an engine type, the engine's input is automatically connected to a clock and the application can connect its output to any data member. The engine obtains the current elapsed time value at a regular interval (the application can specify the interval). Every time the engine receives a time value, it changes the angle data member of the rotation node in the scene graph. Since the scene graph is changed, the whole scene will be redrawn, and the cone appears to rotate at a regular interval. When Open Inventor interprets Line 8 in Figure 2.7 it registers a time event, which will occur at a regular interval and connects this event to a system built-in callback function which requires an input data (for this example, the input data is the angle data member of the rotation node *myRotXYZ* → *angle*), and replaces the value of the input data by the current elapsed time value. One can replace line 8 by the following code:

```
myRotXYZ → axis = SoRotationXYZ::X;
SoTimerSensor *rotatingSensor = new SoTimerSensor(theCallbackFunction,myRotXYZ→ angle);
rotatingSensor→setInterval(1.0);
rotatingSensor→schedule();
```

and write a callback function *theCallbackFunction*(*void* $*$ *data*, *SoSensor* $*$ *sensor*) which uses the time value obtained from *sensor* to change the value of *data*. This also makes the cone spin.

6. Summary

In this section, we give a complete picture of Open Inventor's event handling mechanism.

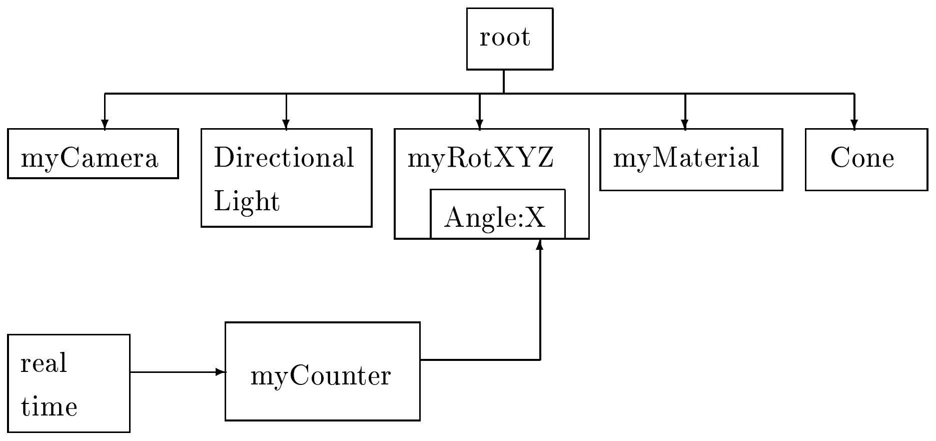


Figure 2.8: The scene graph for the spinning cone example.

6.1 Event sources

Those events which belong to Open Inventor's event model are fixed and have clear sources. For the rest, one can consider that each sensor is an event source. Open Inventor can be considered to hold two queues for event sources, the Time Sensor queue and the Data Sensor queue for storing these two kinds of event sources. Whether or not a time event has occurred can be checked during the regular event handling interval. There is a data member (we may call it as modification flag) in each Open Inventor node, used for data events. A method *touch()* defined in *SoBase* is used for setting the flag. When a data sensor is interpreted, Open Inventor will invoke the *touch()* method. When a node is modified[1] during a traversal, the modification flag will be checked. If the flag is on, the attached sensor will be marked for ready being scheduled. The Open Inventor event sources are illustrated in Figure 2.9. When a sensor is created, it is added to the corresponding queue. When it is deleted, it is removed from the queue.

6.2 Event queues

Open Inventor has three queues for events, the Event queue, the Timer queue and the Delay queue. Each entry in the Timer queue and the Delay queue should include a callback function which is invoked when the entry is processed. There is no callback function in the Event queue. Open Inventor deals with hardware device events by creating an instance of *SoHandlEventAction* and applying it to the top node of the scene graph inside the window where the event occurs. If a node calls the *setGrabber()* method, the handle event action applies the action to that node instead of to the whole scene graph for all subsequent events until the grab is released. Conceptually, a stack is used to keep track of the node to which the handle event action should be applied. The stack is initialised by pushing the top node of the scene graph. When the *setGrabber(theNode)* is called, *theNode* is pushed onto the stack, when the *releaseGrabber()* is called, the node on the top of the stack is popped off. The system always applies the handle event action to the node on the top of the stack.

The data structures for the entries in each queue and the event handling traversal node stack are illustrated in Figure 2.10.

6.3 Event capture

The events in the Event queue come indirectly from hardware devices, such as a mouse or trackball. These events will be captured by the window system and then translated and passed to Open Inventor. A window system typically processes events at regular intervals and this consequently makes Open Inventor process events at regular intervals. Each time the window system processes event, it invokes

[1]In fact, whenever a write operation is executed on the data to which the sensor is attached, the sensor will be scheduled no matter whether or not the value of the data is really changed.

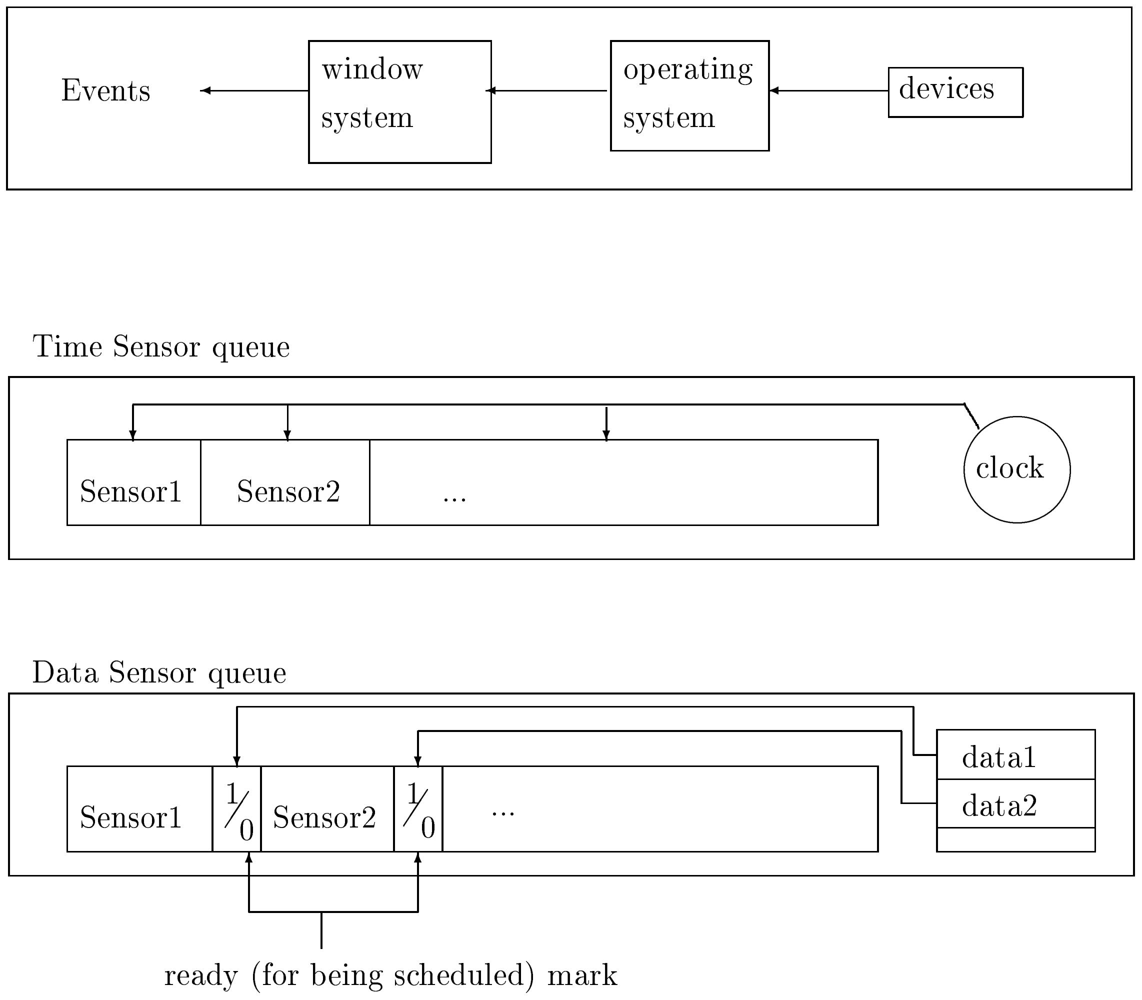


Figure 2.9: The three event sources in Open Inventor.

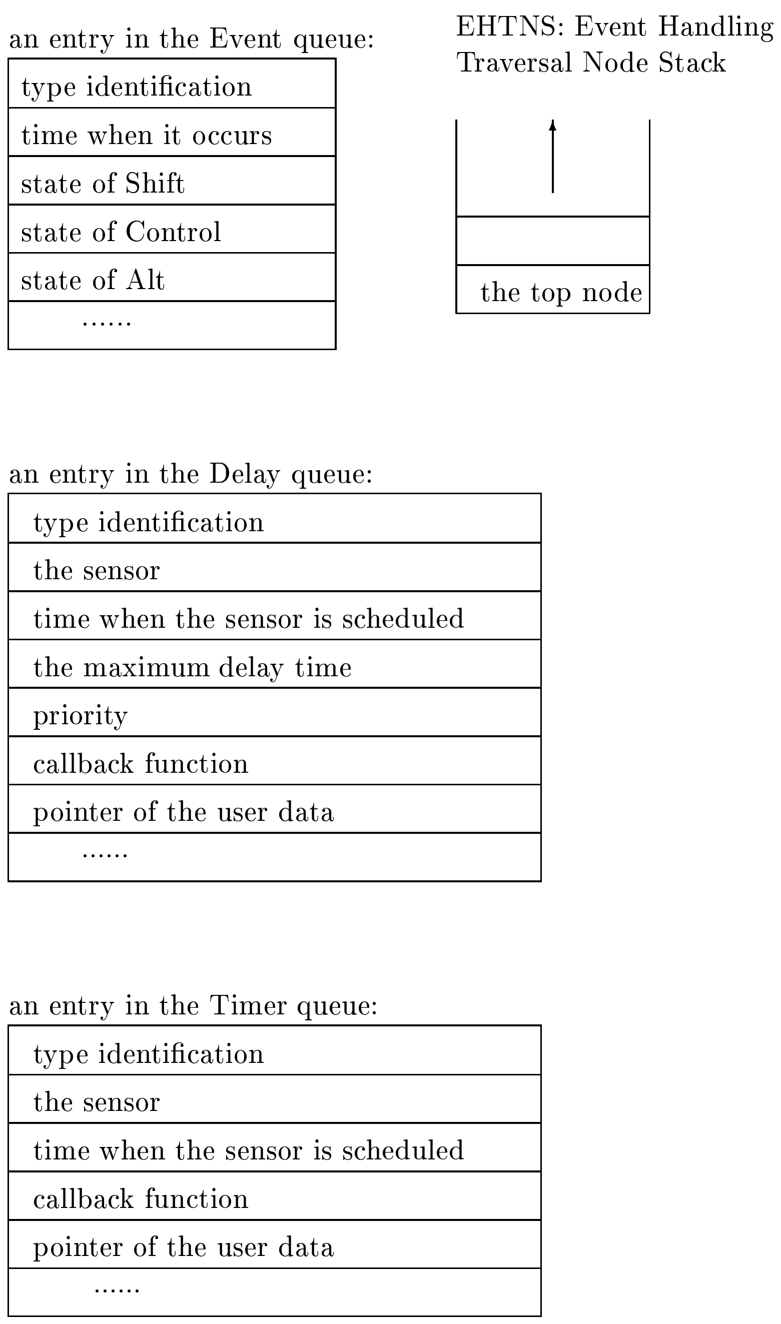


Figure 2.10: The entries in Open Inventor's three event queues.

Open Inventor's event handling mechanism, no matter whether or not it is an event of interest to Open Inventor. When Open Inventor starts to work, it first checks if there are any interesting window events. If yes, the events will be translated into Open Inventor events and added into the Event queue.

Second, Open Inventor checks if the Time Sensor queue is empty. If not, it checks the current time and adds those which should be scheduled into the Timer queue.

Third, Open Inventor checks if the Data Sensor queue is empty. If not, it checks the *ready* mark for each sensor and adds those sensors which are ready to be scheduled into the Delay queue. After a sensor is added into the Delay queue, if it is not deleted from the Data Sensor queue (i.e. the data event represented by this sensor is still an interesting event), its *ready* mark should be cleared.

6.4 Event handling

Handling an event in the Event queue is simply a matter of applying the handle event action to the node on the top of the stack. When the handle event action starts to work, it traverses the scene graph from the node passed to it as input. Scene graph traversal resulting from the handle event action is described in an earlier part of this part. Handling events in the Delay and Timer queues simply invokes the callback functions. The relationship between event sources and event queues is illustrated in Figure 2.11.

7. Implementation in PREMO

7.1 Hardware device events

In order to implement Open Inventor's event model in PREMO, we must first represent Open Inventor events using PREMO's event data structure. The data structure describing PREMO's events is a non-object type which has the following fields: an event name, event data and the event source. One possible mapping of Open Inventor's events onto this structure is shown in Figure 2.12.

We also need to have a strategy to handle the event models, such as event generation and dispatch, in PREMO. PREMO's event model (see in Figure 0.4) consists of event sources, event clients and an event handler which are all instances of PREMO object types. Events are generated by event sources and consumed by event clients. *EventHandler* provides two operations: *dispatchEvent()* and *register()*. An event client invokes the *register()* to tell the event handler which event it wishes to receive. An event source invokes the *dispatchEvent()* which will then forward the event to all the event clients which expressed their interest in this specific event.

Modelling Open Inventor's event model in PREMO is illustrated in Figure 2.13. Based on this model, the set of PREMO objects corresponding to Open Inventor's event classes are the event sources, and the one corresponding to Open Inventor's *SoHandleEventAction* is the only client which registers its interest to all the events. The event sources are PREMO's *InputPorter* objects which import inputs from environments defined outside of PREMO into PREMO. When a window specific event occurs, the *InputPorter* will translates it to PREMO event and then invoke the *dispatchEvent()* operation on the *EventHandler*.

When *SoHandleEventAction* is invoked, it applies the handle event action to the node which is stored in a stack *EHTNS* (Event Handling Traversal Node Stack), which can be implemented as a *list* in PREMO. The top node of the scene is pushed onto the stack when the application starts, and whenever the *setGrabber()* (or *releaseGrabber()*) method is called, the related node is pushed onto (or popped off) the stack.

We can fit the event model in Figure 2.13 into PREMO's environment model, where the window specific events are in the *Lexeme Store*, the *InputPorter* which generates events is the *Accumulation* process and so on. However, there is a difference between PREMO and Open Inventor. In PREMO, input tokens from the operator are transformed into the useful information to serve the application in a standard way, i.e. they are manipulated in a lower environment and then are sent to the higher environment. However, this environment concept does not exist in Open Inventor. The methods encapsulated in one Open Inventor's class *SoHandleEventAction* cross several environments. For instance, the event handling traversal conceptually belongs to PREMO's construction environment and finding the picked node belongs to lower environments. Therefore, the Open Inventor's *SoHandleEventAction* class should be implemented as a set of PREMO object types in each environment. The one in Figure 2.13 is in the Logical environment. When it is invoked by the *dispatchEvent()* operation, it

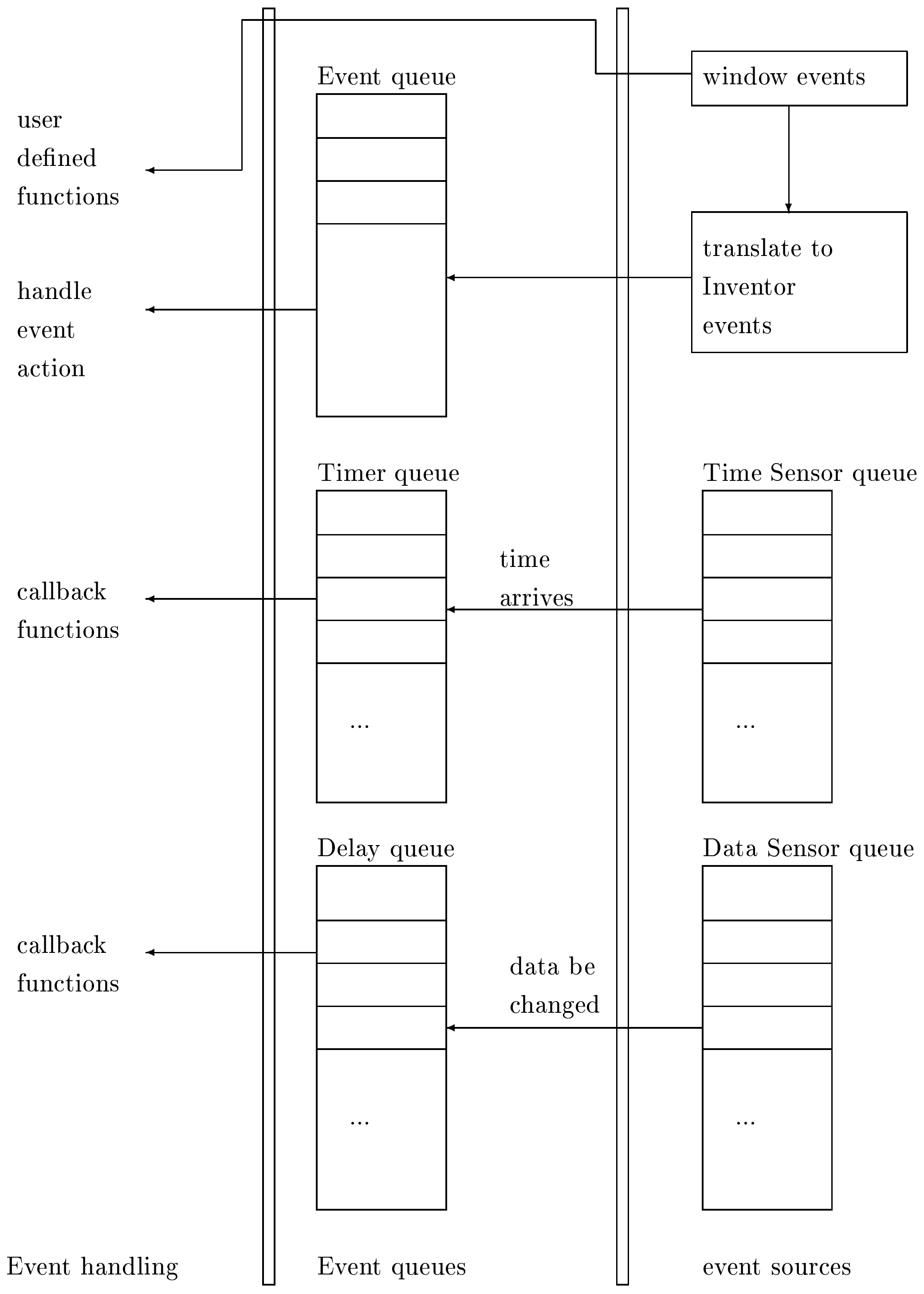


Figure 2.11: The relationship between event sources and event queues.

Event name:	SoKeyboardEvent
Event data:	(identification,-),(key name,-),(time,-),(cursor position,-),(shift,-),(control,-),...
Event source:	key board
Event name:	SoMouseButtonEvent
Event data:	(identification,-),(button No.,-),(button state,-),(time,-),...
Event source:	mouse
...	

Figure 2.12: Describing Open Inventor events by means of PREMO event data structure.

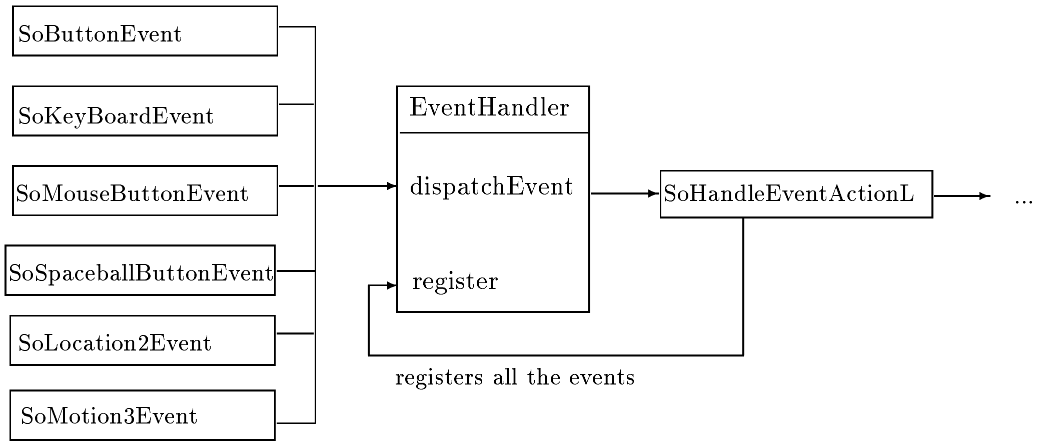


Figure 2.13: Modelling Open Inventor's event model in PREMO. *SoHandleEventActionL* is in the logical environment and it will pass the data and control to the higher environments.

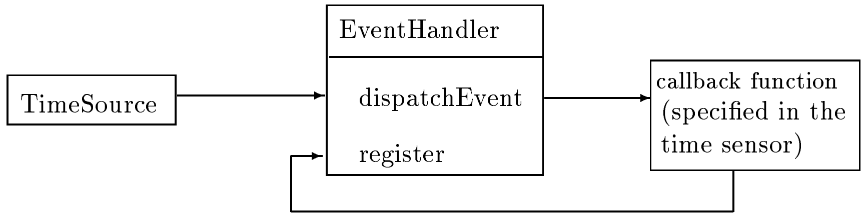


Figure 2.14: Modelling Open Inventor's time sensors in PREMO.

only does the necessary job in Logical environment and passes the event to the higher environment. The scene graph traversal for event handling is in the construction environment.

One may remark that the *EventHandler* does not seem to be necessary in this model. Since all the events will be dispatched to the same event client who is also interested in all of them, why do not we simply let the event sources invoke the event client directly? However, a model with the *EventHandler* has certain advantages. Although Open Inventor conceptually applies the *SoHandleEventAction* to the scene graph whenever an event occurs, this is not alway necessary in practice. A more reasonable strategy would be: first find out what events are relevant with the application and then register the necessary events. It is obvious that a model with the *EventHandler* is much more flexible.

7.2 Time events

Modelling Open Inventor's time events in PREMO is illustrated in Figure 2.14. PREMO has an object type *Clock* in which the operation *inquireTick* returns the number of ticks elapsed since the start of the PREMO era, i.e. 00.00am, 1st Jan 1995. We can write an object type *TimeSource* which is a subtype of the *Clock*. An operation *timeEvent()* in *TimeSource* keeps a loop in which it invokes the *inquireTick* operation and compares the current time with a target. When the target is reached, it invokes the *dispatchEvent()* operation. Instances of the *TimeSource* can simulate the various time events in Open Inventor. For example, an alarm sensor wishes to be scheduled at time T. An instance of the *TimeSource* is initialised with T and other parameters, so that the callback function specified in the alarm sensor will be invoked when the time is reached to T. The same kind of time sensors can share one source through the *EventHandler*.

PREMO's *Clock* can be used in all the environments and in this model, it is in the construction environment. Callback functions belong to applications and, in this model, they are invoked through the *Utilization* process.

7.3 Data events

Modelling Open Inventor's data events in PREMO is illustrated in Figure 2.15. When a data sensor is created, one of the scene graph node which is attached with the sensor becomes an event source. The callback function specified in the data sensor registers its interest in the data event generated from the corresponding node.

A node in Open Inventor invokes a method *touch()* to switch on a flag when it is connected with a data sensor. Since the flag is switched on, the connected sensor will be notified whenever an operation which contains writing instructions was applied on this node. When we implement this in PREMO, the node must be related to the *EventHandler* when the touch flag is switched on, and operations which change the values in the node also check the touch flag so that the callback function specified in the data sensor can be invoked through the *dispatchEvent()* operation.

This model is also in the construction environment.

7.4 Priority

In this section, we discussed how to implement various Open Inventor's event models in PREMO. Open Inventor also schedules various events in an order, e.g., first time events, hardware device events, and data events are handled, where a priority can be attached to this latter category. Though supporting

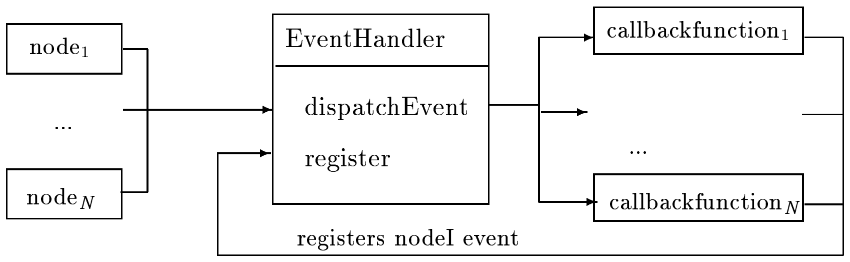


Figure 2.15: Modeling Open Inventor's data events in PREMO.

schedule policies is not Open Inventor's own feature, but it depends on the window system, it is one of the important aspects which a system should provide. This feature is not modelled here; indeed the current PREMO model does not include the concept of priority. This may be a shortcoming of the PREMO model which has to be dealt with in the future.

Chapter 3
Other Features

This part considers some of the other features of Open Inventor. These are essentially supporting features that are clearly useful in any serious development environment; they do not introduce fundamental changes to the Open Inventor architecture.

1. Node kits

Open Inventor provides a corresponding *Kit* class for each of the many Open Inventor classes. For instance, for the *SoShape* class there is a *SoShapeKit*, for *SoSpotLight* there is a *SoSpotLightKit*, and so on. Node kits are a convenient mechanism for creating scene graphs. When an application creates a shape node, such as an indexed triangle strip set, it usually also needs at least a coordinate node, a material node and a transform node. It may also need to specify drawing style, a material binding and so on. Instead of creating each of them individually, specifying values and then arranging them into a subgraph, the application can simply use an *SoShapeKit*, which already contains information on how these nodes should be arranged in the subgraph.

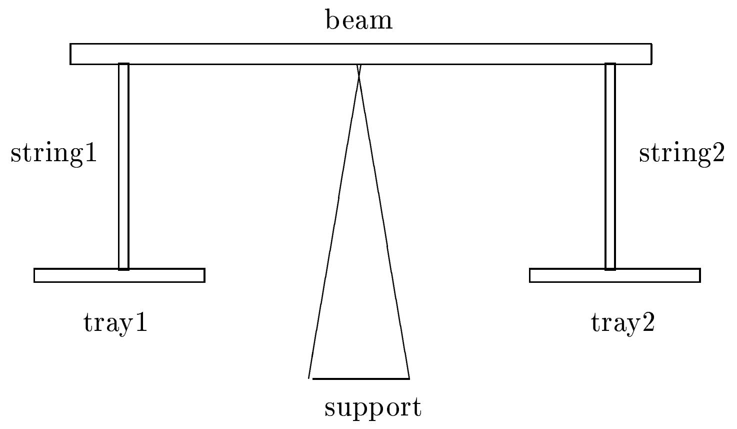


Figure 3.1: A balance scale.

Consider creating a scene which shows a balance scale as illustrated in Figure 3.1. Without node kits, it is necessary to create a shape node, specify its size, create a translation node, and then specify

Specify the root *myScene*.	1
SoShapeKit *support=new SoShapeKit(); support→setPart(”shape”,new SoCone); support→set(”shape{height 3 bottomRadius .3}”); myScene→setPart(”childList[0]”,support);	2
SoShapeKit *beam=new SoShapeKit(); beam→setPart(”shape”,new SoCube); beam→set(”shape{width 3 height .2 depth .2}”); beam→set(”transform{translation 0 1.5 0}”); support→setPart(”childList[0]”,beam);	3
SoShapeKit *string1 = new SoShapeKit; string1→setPart(”shape”, new SoCylinder); string1→set(”shape { radius .05 height 2}”); string1→set(”transform { translation -1.5 -1 0 }”); string1→set(”transform { center 0 1 0 }”); beam→setPart(”childList[0]”, string1);	4
Specifying *string*2, which is similar with 4.	5
SoShapeKit *tray1 = new SoShapeKit; tray1→setPart(”shape”, new SoCylinder); tray1→set(”shape { radius .75 height .1 }”); tray1→set(”transform { translation 0 -1 0 } ”); string1→setPart(”childList[0]”, tray1);	6
Specifying *tray*2 which is similar with 6.	7

Figure 3.2: Specify the balance scale by means of node kits.

the location for each of the components of the balance scale. These would then have to be arranged into a subgraph and this must be placed into the scene graph. If node kits are used, the specification becomes much simpler, shown in Figure 3.2.

However, node kits do not introduce new fundamental functions. The scene graph built according to node kits is similar with that built according to the corresponding program without node kits. The scene graph built by Open Inventor according to the program code, which is partially shown in Figure 3.2 (for the whole program, see page 384 in [13]), is shown in Figure 3.3.

2. Actions

Besides the actions mentioned before, such as *SoGLRenderAction* and *SoHandleEventAction*, there are other actions in Open Inventor. *SoGetBoundingBoxAction* is used to compute a 3-D bounding box enclosing objects defined by a scene graph. *SoGetMatrixAction* is used to compute a cumulative transformation matrix and its inverse. *SoWriteAction* is used to write the scene graph to a file or to the screen. *SoSearchAction* is used to search for paths to specific nodes, types of nodes, or nodes with specific names in the scene graph. *SoRayPickAction* is used to pick objects in the scene graph along a ray. *SoCallbackAction* traverses the scene graph and accumulates traversal state, then performs application specified actions by means of a callback functions.

All these actions manipulate scene graphs. Fundamentally, what is needed for each of these actions is a general mechanism for traversal, searching, insertion, modification and deletion of scene graphs. Any implementation of these actions should be based on such a general mechanism.

3. Viewers and editors

Open Inventor has a set of *Xt* components, which are reusable modules with built-in user interfaces for changing the scene graph interactively. There are two types of components: *viewers* and *editors*. Viewers affect the camera node in the scene and editors affect other nodes and data members in the scene, such as material nodes and light nodes.

3.1 Viewers

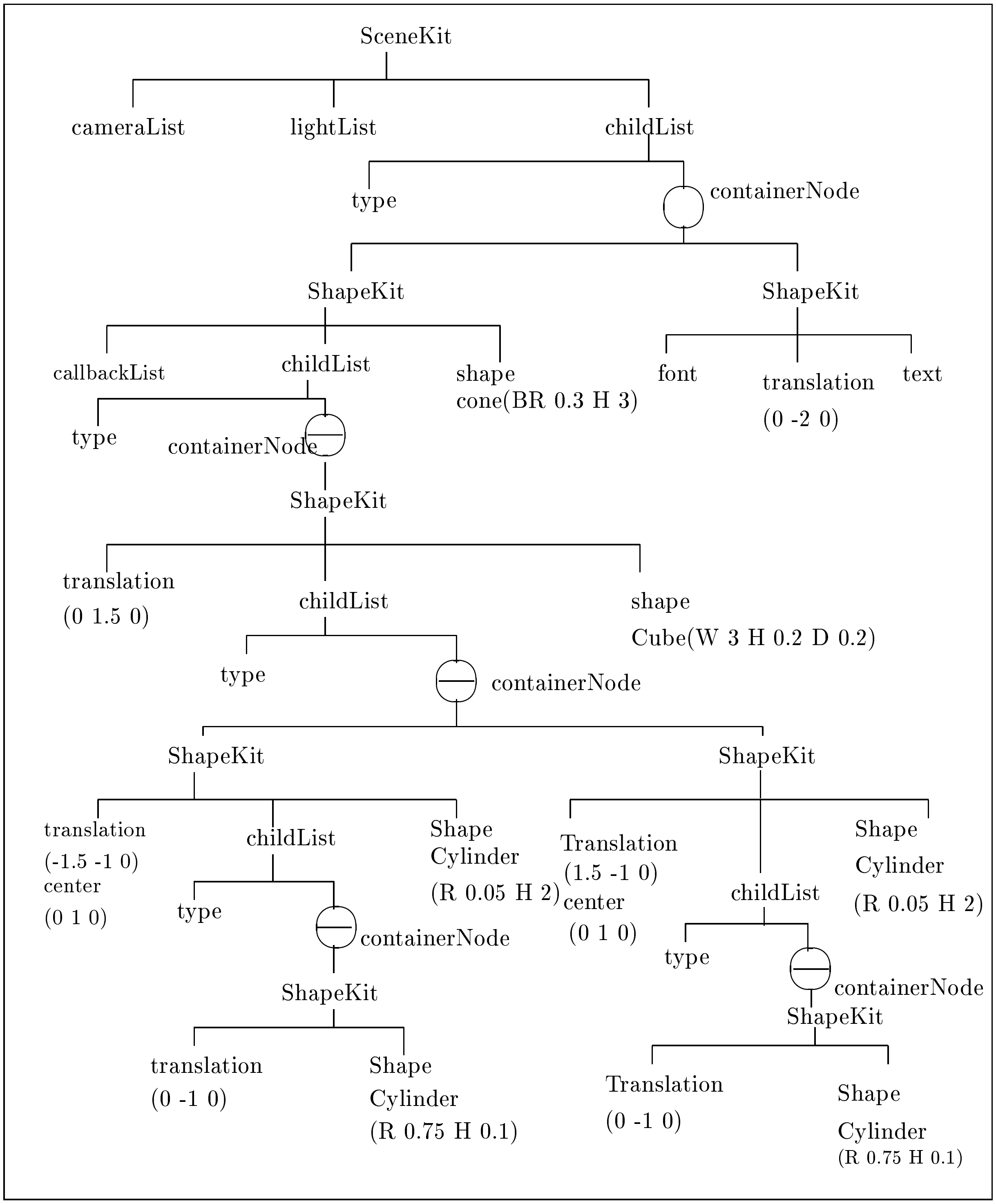


Figure 3.3: The scene graph for the balance scale.

Widget myWindow = SoXt::init(argv[0]);	
if (myWindow == NULL) exit(1);	1
SoSeparator *root = new SoSeparator; root→ref();	2
root→addChild(new SoCone);	3
SoXtExaminerViewer *myViewer = new SoXtExaminerViewer(myWindow);	
myViewer→setSceneGraph(root);	
myViewer→setTitle("Examiner Viewer");	4
myViewer→show();	
SoXt::show(myWindow);	5
SoXt::mainLoop();	6

Figure 3.4: An example examiner viewer program.

Widget myWindow = SoXt::init(argv[0]); if (myWindow == NULL) exit(1);	1
SoSeparator *root = new SoSeparator; root→ref();	2
SoMaterial *myMaterial = new SoMaterial;	
myMaterial→diffuseColor.setValue(1.0, 0.0, 0.0);	
root→addChild(myMaterial)	3
root→addChild(new SoCone);	4
SoXtExaminerViewer *myViewer = new SoXtExaminerViewer(myWindow);	
myViewer→setSceneGraph(root);	5
SoXtMaterialEditor *myEditor = new SoXtMaterialEditor;	6
myViewer→show();	
myEditor→show();	7
SoXt::show(myWindow);	
SoXt::mainLoop();	8

Figure 3.5: An example material editor program, where line 6 specifies a material editor and line 7 shows the editor.

There are a group of viewers. The *SoExaminerViewer* viewer provides a user interface that allows use of the mouse to modify camera placement in the scene. An example program using an *SoExaminerViewer* is shown in Figure 3.4. The program creates an examiner viewer window containing a cone, where *Rotx* (*Roty*) rotates the camera around the X (Y) axis, *Dolly* translates the camera along the Z axis, the left mouse button rotates the camera around any point and the middle mouse button translates the camera along the X and Y axes.

One observation is that when an examiner viewer responds to mouse events, it does not traverse the scene graph. The scene graph is traversed once initially, when the objects in the render area are drawn. The explanation is probably that, first, the examiner viewer handles window events directly so that there is no Open Inventor's handle event action applied to the scene graph when these events occur. Secondly, the examiner viewer probably sets the redraw flag to *False*, so that even if the scene graph is changed, the render action will not be applied. The examiner viewer may implement its own render action which is probably faster than redrawing by traversing all of the scene graph.

All other viewers have a similar function and each of them has its own particular way to transform the camera interactively.

3.2 Editors

Open Inventor also provides a set of editor classes. An editor has a built-in user interface. Users can adjust properties such as light and color interactively. For example, the program in Figure 3.5 creates both an examiner viewer and a material editor interface.

If we insert the following line

$$myEditor \rightarrow attach(myMaterial);$$

between line 6 and line 7, the material of the cone will change as the user adjusts the material in the material editor.

Editors are *Xt* widgets. The communication between an editor and the scene graph is implemented by callback functions.

4. Interaction to animation

An interesting feature of the system built-in tools such as manipulators, draggers and examiner viewers is the support for interactive transformations. If a sequence of left mouse button motion events followed by a left mouse button release event occur, then the selected objects will start to spin until the user presses the left mouse button again. The difference between an examiner viewer window and all the others is that in an examiner viewer window, the camera spins, while, for the others, the dragger and the objects spin. However, no matter which of them spins, users always feel that the objects in the render area spin. This feature is probably realised as follows. When the node (or the examiner viewer window) receives a left mouse press event, not only does it respond to each subsequent event by modifying the scene graph, but it also starts to record all the following events until it receives the mouse release event. According to the time when each event occurred, it then decides whether or not to let the objects (or the camera) spin. Spinning an object (or a camera) can be achieved by creating a sensor which modifies the rotation angle in a rotation node, which is then inserted in front of the selected objects (or the camera). The time interval for scheduling the sensor is determined by the speed of the mouse, which can be calculated according to the times recorded when each event occurred. When the next left mouse press event is received, the sensor is deleted and the system starts to respond to the new event.

Chapter 4
Scene Graphs

"Scene graphs" is the most important feature of Inventor. A scene graph is a collection of Inventor nodes and moreover it defines an ordering over the nodes. In such an order, properties are attached to each graphical object in a simple way. Inventor's most important functions such as rendering and event handling rely on working on scene graphs. By traversing a scene graph, the render action can obtain a complete description of the graphical objects in the virtual environment level. By traversing and then editing a scene graph, various interactions such as selections and manipulations can be realised. Furthermore, other functions such as writing scene graphs, picking, searching and so on also rely on manipulations on scene graphs. Therefore, we give a more detailed discussion of how to implement Inventor's scene graphs by means of PREMO.

In Inventor, there are many places where scene graphs play a role. However, it is not correct to implement all of them at this moment. Instead we only discuss how to implement certain important parts. According to us, the two main aspects are (1) how to construct and edit a scene graph and (2) how to traverse a scene graph (i.e. traversing each node in a scene graph according to the order defined by scene graphs).

In the following, we, first, discuss how to build and edit scene graphs by means of PREMO. Second we discuss how to traverse a scene graph so that the various Inventor actions can be realised by means of PREMO. Finally, we give both informal descriptions and formal definitions of the new PREMO types in which the operations provide the same functionalities with those provided by Inventor's for constructing, editing and traversing scene graphs.

1. Constructions and editions

In this section, we discuss how to build and edit a scene graph. We start by analyzing the set of Inventor's classes which are relevant for constructing and editing a scene graph. This analysis guides us in choosing PREMO data structures to represent scene graphs. We then discuss how to implement them in PREMO.

1.1 Class tree

There is a set of Inventor classes for constructing and editing scene graphs. The subclass relationship between this set of classes is shown in Figure 4.1.

SoBase and *SoNode* are abstract classes. They describe the minimum behaviour for all nodes. The class *SoGroup* has methods which are used to construct and edit the structure of a scene graph, e.g. *addChild()*, *insertChild()* and *removeChild()*. The class *SoPath* provides a set of methods which can be used to create and edit a path in a scene graph, e.g. *setHead()*, *getTail()* and *append()*. The rest

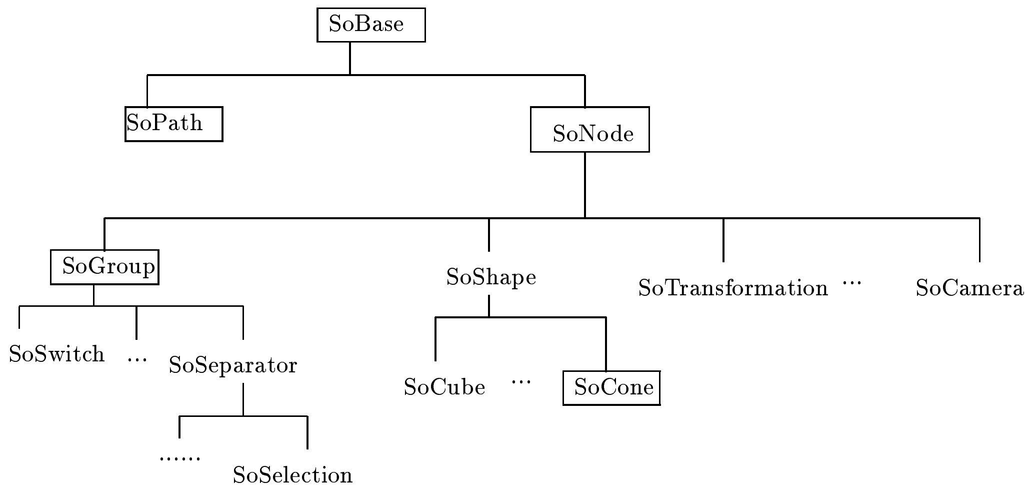


Figure 4.1: The Inventor class tree for building and manipulating scene graphs and paths in a scene graph. The classes enclosed by a box will be defined in PREMO.

of the classes provide methods for assigning values to and retrieving values from their special fields, e.g. *addPart()*, *removePart()* in a *SoCone* class.

In order to see whether or not PREMO can implement the construction and edition of Inventor's scene graphs and paths, only some classes in Figure 4.1 need to be implemented by means of PREMO. They are *SoBase, SoNode, SoGroup, SoPath* and any one of the remaining classes. Because the methods in the rest of the classes are for setting and getting values in their specific fields, implementing one of them (we choose *SoCone*) is enough to demonstrate that all the others can be implemented in PREMO in a similar way. *SoBase* and *SoNode* are for building nodes and controlling the lifetime of a node, *SoGroup* is for collecting nodes together to form a graph and editing the graph, and *SoPath* is for manipulating paths in a scene graph.

There are some classes, e.g. *SoCamera*, which not only have methods for assigning and retrieving values in their special fields but also have methods for special functions, e.g. *viewAll* which works on a scene graph and produces the description of 2D pictures. However those methods are irrelevant with construction and manipulation of scene graphs (they may need to traverse scene graphs, see the later sections), we do not discuss how to define them in PREMO at this stage.

1.2 Nodes

In this section, we study Open Inventor's node model and discuss the relationship between Open Inventor's node model and PREMO's object model.

The functionalities provided in these *SoBase* and *SoNode* can be summarised as: (1) controlling the lifetime of a node, (2) support data events, (3) inquiry node type and (4) node property.

1. *Lifetime of a node*: The lifetime of a node is determined by whether or not it is in use. Each node stores the number of references made to it. *ref(), unref(), unrefNoDelete()* are used to add and remove a reference to an instance. Nodes and paths should be referenced when they will be used outside of the routine in which they were initialised. When the reference count for the node is decremented to 0, the node is automatically destroyed by the database unless *unrefNoDelete()* is used. The reference count of a node is automatically incremented when the node is added as a child of another node or when a path points to the node. Likewise, the reference count is automatically decremented when the node is removed as a child or when a path that points to the node is changed or destroyed.

   The PREMO object model gives a natural implementation of the creation and deletion of objects. Furthermore, one of the fundamental features of the PREMO object model is that no object is deleted as long as a valid reference to this object exists. Consequently, the lifetime of an Open Inventor node is automatically managed in PREMO.

Moreover, a more user friendly interface will be obtained if the scene graph management is implemented on the top of PREMO. In Open Inventor, attentions must be paid by applications to those nodes which should not be deleted but they will be deleted according to the Open Inventor's node model. For example, the root node of a scene graph is not referenced by being a child of anything else and its reference count is 0. When Open Inventor applies an action to a node, it, first, increments its reference count and decrements it after the action is finished. Therefore the root node and so as the whole scene graph rooted by it will be deleted automatically by Open Inventor after the first action has been applied to it. Open Inventor does not solve this problem and it must be handled by the application programmer who has to explicitly reference such kind of nodes, e.g., adding $root \rightarrow ref()$ in the program to prevent the node *root* from being deleted. If we implement the scene graph by means of PREMO, such problem does not exist.

Therefore, in our implementation of the scene graph, these three methods are not necessary.

2. *Data events*: When a data sensor is created, a flag of the node which is attached to the sensor is switched on by calling *touch()* so that the changes of this object will cause the attached sensor to fire.

   According to the data event model in Figure 2.15, when it is implemented in PREMO, *touch()* needs an object reference of the corresponding *EventHandler* as its input, so that when the node is modified, it will invoke the *dispatchEvent()* operation on the *EventHandler*.

3. *Inquiry node type*: The methods *getClassTypeId()* and *getTypeId()* for inquiring the type of a node are widely used by Open Inventor and its applications. For instance, when a node is encountered during a scene graph traversal, the first thing which the system has to figure out is what kind of node it is, i.e., what its type is, what its supertypes are, etc..

   When we implement scene graphs in PREMO, we get these facilities for free. PREMO's object model supports subtyping for object types and provides a set of operations which can be used to inquiry various information about the type of an object.

4. *Node property*: *set()* and *get()* are methods for setting and getting properties of a node. This is also covered by PREMO which provides a set of functions for managing the properties of objects.

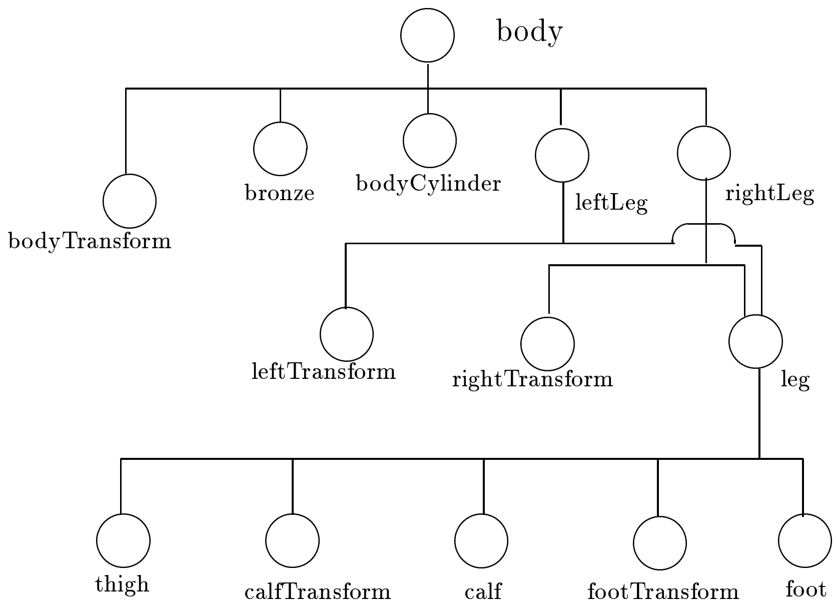
   Since a node can be an event source by calling *touch()*, we must distinguish two kinds of properties, one is for control and the other is for scene graph changing. Data events should only occur when the scene graph is changed. For instance, a natural way implementing the *touch()* method is setting TRUE to a boolean variable when *touch()* is invoked. However, it is not correct to let the data event occur because of this. In the implementation, we use PREMO's *defineProperty* for setting properties, and define a *set* operation which sets properties and manages to notify data event.

We divide the remaining classes (except *SoPath*) in Figure 4.1 into two groups, a group that creates non-terminal nodes and another which creates the terminal nodes in a scene graph. There are a variety of different non-terminal node classes rooted by *SoGroup*, each with a specialised grouping characteristic. However, only the methods provided by *SoGroup* are related with building and editing a scene graph.

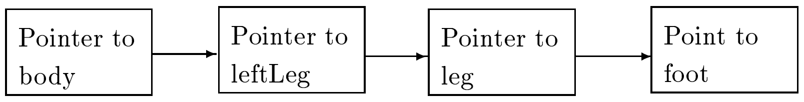
1.3 Paths

*Path*s are used to isolate particular objects in the scene graph and are returned by a *picking* or *searching* action. Consider the scene graph in Figure 4.2a (from page 55 in [13]), which represents the body of a robot. Suppose left foot of the robot is picked. Which node in Figure 4.2a represents the left foot? It is impossible to refer to the *foot* node, since that node is used for both the left and right feet. The solusion is using path to represent it, i.e., *body.leftLeg.leg.foot* for the left foot (see in Figure 4.2b).

A path contains a list of pointers to nodes forming a chain from some root to some descendent. Each node in the chain is a child of the previous node. When an action is applied to a path, only the nodes in the subgraph defined by the path are traversed. These include: the nodes in the path chain,



(a)



(b)

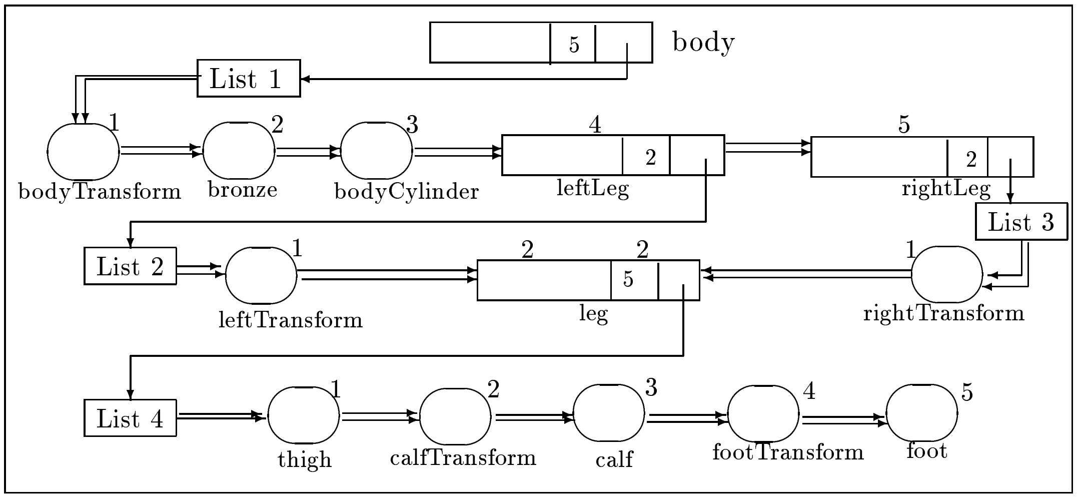
Figure 4.2: (a) is a scene graph of a robot's body and (b) is a path in the scene graph.

all nodes below the last node of the path (if there are any), and all nodes whose effects are inherited by any of these nodes. For example, when a render action is applied to the path for the left foot in Figure 4.2a, the traversal order is: *body, bodyTransform, bronze, bodyCylinder, leftLeg, leftTransform, leg, thigh, calfTransform, calf, footTransform* and *foot*.

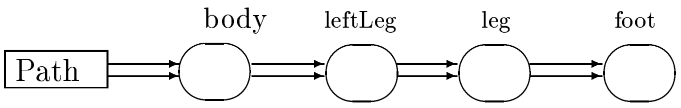
The *SoPath* class attempts to maintain consistency of paths even when the structure of the scene graph changes. For instance, if the *leg* node is removed from its parent node *leftLeg* in Figure 4.2a, the path for the left foot *body.leftLeg.leg.foot* will be cut from the point *leg* and becomes *body.leftLeg*.

1.4 The idea of the implementation in PREMO

In order to implement the functions provided in *SoGroup* class, we need to consider in which way nodes in a scene graph are connected with each other in PREMO. We let the object references of all the children of a node be an instance of PREMO's *List[SoNode]*. *List[E::PREMOObject]* is a generic type in PREMO. It defines a family of list object types. When we actualise the formal type *E* into *SoNode* in the generic type *List[E::PREMOObject]*, we get an object type *List[SoNode]* whose instances are lists with object references of type *SoNode* as elements. When a group node is initialised, an object whose type is *List[SoNode]* is created and the object reference of this list object is stored in a field of this group node. The methods *addChild(), insertChild(), getChild(), removeChild()* and



(a) scene graph



(b) path

Figure 4.3: Representing the scene graph and the path in Figure 4.2 in PREMO.

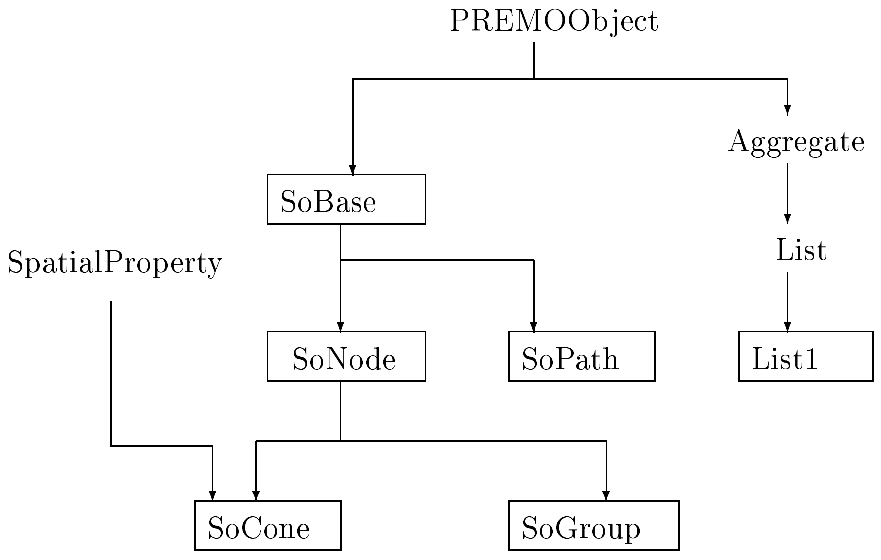


Figure 4.4: The subtyping relationship between the new types and PREMO's original types. The new types are enclosed by boxes.

replaceChild(), in the *SoGroup* class, for building and editing scene graphs can then be implemented by means of the set of operations in PREMO's *List[SoNode]* object type.

Figure 4.3 illustrates such a connection of the scene graph and the path in Figure 4.2. In the illustration there are two kinds of connections. We distinguish them by single and double arrows. The one connected by double arrows is PREMO's list structure and a single arrow from A to B means that B's reference is stored in A. There are 4 lists in Figure 4.3, *List*1 has 5 elements which are the children of node *body*, *List*2 has 2 elements which are the children of node *leftLeg*, *List*3 has 2 elements which are the children of node *rightLeg* and *List*4 has 5 elements which are the children of node *Leg*. Node *Leg* is shared by two lists.

Based on this data structure, we can write a set of PREMO object types whose operations provide the same functionalities with those provided by Inventor's classes for constructing and editing scene graphs and paths. We, first, introduce a new type *List1* which is a subtype of PREMO's *List*. The reason that we introduce *List1* is that there are several methods in Inventor which though they can be implemented by means of PREMO, cannot be implemented in a natural way. However, the new operations introduced in *List1* allow a natural implementation. Then, we will define *SoBase, SoNode, SoGroup, SoCone* and *SoPath* by operations in PREMO's original types and in *List1*. The subtyping relationship between the set of newly defined types and PREMO's original types is illustrated in Figure 4.4, where new types are enclosed by boxes.

List1 and the other types get a different treatment in PREMO. We only give an axiomatic semantics to the operations in *List1* and do not discuss how to implement them. However, the meanings of the operations in all the other types are given by the combinations of the operations in PREMO. Suppose F is a new operation, the meaning of F is given in the following form.

$$F \mathrel{\widehat=} \{[Exp_j \Rightarrow F_{j1}\ \underline{op_{j1}}\ F_{j2}\ ...\ \underline{op_{jn}}\ F_{jn}]\} \mid F_1\ \underline{op_1}\ F_2\ ...\ \underline{op_m}\ F_m$$

where Exp_t is a conditional expression, F_t is either an operation in a PREMO object type or a value in a PREMO non-object type, and $\underline{op_t}$ is either an operation schema's connective, a function operator or an arithmetic operator. If F needs to give an output *out!*, then F_{t1} is *out!* and $\underline{op_{ti}}$ is $==$.

Each type (except *List1*) has the same name as the corresponding Open Inventor's class and each operation in the type has the same name as the corresponding Inventor's method in the class. If the operation exists in a PREMO's original type but has different name, we will re-name the operation by

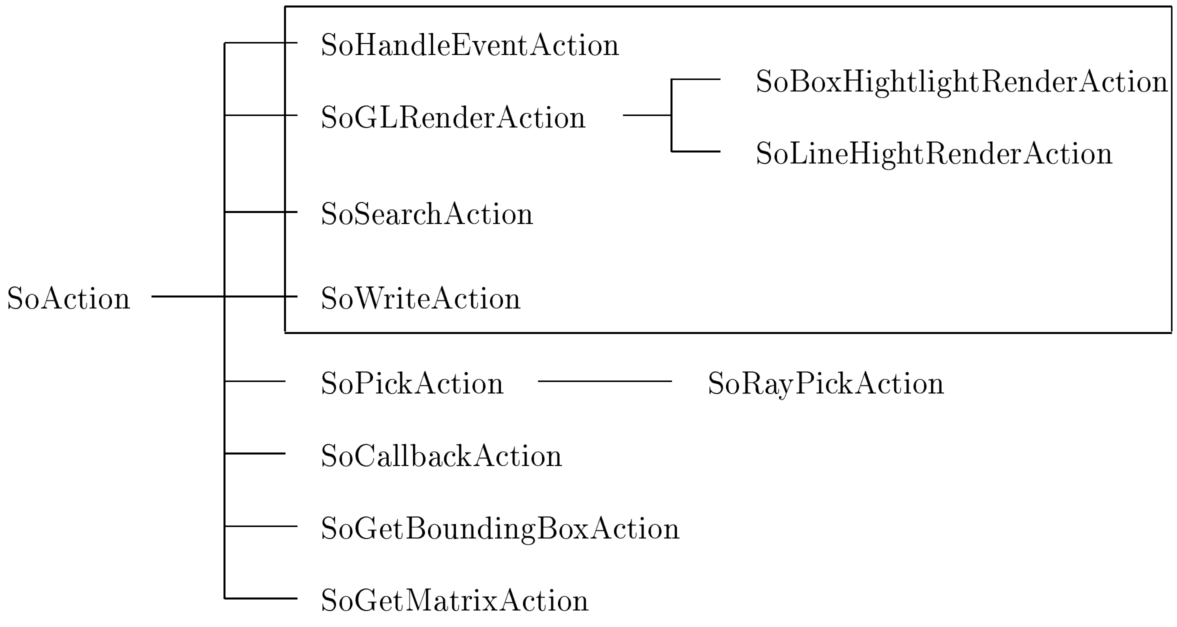


Figure 4.5: Inventor's action class tree. The action classes in the box need to traverse scene graphs.

the corresponding Open Inventor's method name. If a method exists in an Open Inventor's class but its corresponding operation does not exist in the corresponding PREMO object type, this operation is not needed in the implementation, e.g., *ref*(), *isOfType* and so on. All the corresponding Open Inventor classes are described in [12].

In order to maintain the consistency between paths and the scene graph, changes of a scene graph may cause changes of paths in the scene graph. Operations in *SoGroup*, such as *removeChild* and *replaceChild*, may cause the operation *truncate* be invoked in the related paths. To do so needs more data members to record the relations between paths and nodes. We omit it for short. We may also omit many other details for the same reason.

Section 3.1 gives an informal description of the set of PREMO object types and Section 3.2 gives the formal definition. In the definition, path consistency maintaining is not considered.

2. Actions on scene graphs

In this section, we study how to apply actions to a scene graph and how to implement this in PREMO. First, we analyze the set of Inventor action classes. We will see that the most basic function needed by applying actions to a scene graph is traversing a scene graph. Though each node in a scene graph does different things for different actions, the order in which the nodes in a scene graph respond to actions is the same. This order is formed by traversing a scene graph from top to bottom and left to right. Then, we describe the basic idea of the implementation in PREMO.

2.1 Action class tree

The set of Inventor's action classes is shown in Figure 4.5. *SoAction* is an abstract class from which all actions are derived. It has the following methods: *getTypeId()* which returns the type identifier for an action (this is not necessary in our implementation), *apply(node/path)* which initiates an action on the graph defined either by the root node of subgraph or by a path, and *setCaching()* and *isCaching()* which determine whether caching should be turned on for the action. Caching saves the result of an operation so that it does not need to be repeated. Caching is an important optimisation technique and it effects scene graphs' maintenance. For instance, if a scene graph is not changed and the last traversal result has been kept by caching, then the scene graph will not be traversed.

The application of *SoWriteAction, SoSearchAction, SoHandleEventAction* and *SoGLRenderAction* to a node involves traversing the scene graph rooted by the node from top to bottom and left to right. Applying *SoWriteAction* to a node, each node in the scene graph rooted by the node is printed out according to the traversing order. For example, if we apply *SoWriteAction* to the node *body* in the scene graph in Figure 4.2a, the nodes in the scene graph is printed out in the following sequence.

```
body{
    bodyTransform
    bronze
    bodyCylinder
    leftLeg{
           leftTransform
           leg{
              thigh
              calfTransform
              calf
              footTransform
              foot
              }
           }
    rightLeg{
            rightTransform
            leg{
               thigh
               calfTransform
               calf
               footTransform
               foot
               }
             }
     }
```

Applying *SoSearchAction* to a node returns the first node (or all nodes) which satisfies (satisfy) the requirement in a scene graph rooted by the node. The searching is following the traversing order. Applying *SoHandleEventAction* to a node, the scene graph rooted by the node is traversed until the event is handled or the traversing is finished. Applying *SoGLRenderAction* to a node, the scene graph is traversed so that all the properties are accumulated for each shape and then the set of shapes are rendered in the render area.

SoGetMatrixAction and *SoGetBoundingBoxAction* do not need to traverse the scene graph. They work only on a node (or a path for *SoGetMatrixAction* if it is applied to a path). *SoCallbackAction* can be specified for nodes or paths, when those nodes or paths are encountered during traversing a scene graph, the user's callback function is called. *SoPickAction* maintains the list of details that results from picking.

2.2 The idea of the implementations in PREMO

In this section, we describe how to implement Inventor actions in PREMO. According to the analysis in last section, the basic function for realising most actions is traversing scene graphs. Therefore, we will write an object type *SoAction* which is a subtype of *PREMOObject*. The operation *apply()* in *SoAction* will traverse a scene graph. In Inventor *apply()* is defined as a virtual void, which is overloaded by all the subclasses of *SoAction*. Executing an action by calling the specific *apply()* method defined only for this action is obviously faster than by calling a general *apply()* method defined in a super class. However, our purpose is to check whether or not PREMO can naturally implement Inventor. Therefore, the operation *apply()* in *SoAction* is inherited (with few modifications) by the subtypes which need to traverse scene graphs (e.g. *SoGLRenderAction, SoHandleEventAction, SoSearchAction* and *SoWriteAction*) and it will be redefined in the other subtypes which do not need to traverse scene graphs (e.g. *SoCallbackAction, SoGetBoundingBoxAction, SoGetMatrixAction* and *SoPickAction*).

The operation *apply()* in *SoAction* needs two input parameters, one is a node or a path on which the action is applied, and the other one is an operation description, which is defined in all node types to deal with the action. We add a set of auxiliary operations to each type (not an abstract type) in the type tree *SoNode* to deal with a set of actions. This set of auxiliary operations are: $\Phi render$ for the node to deal with *SoGLRenderAction*, $\Phi event$ for *SoHandleEventAction*, $\Phi search$ for *SoSearchAction* and $\Phi write$ for *SoWriteAction*. We wish this set of auxiliary operations can only be invoked by PREMO instead of by applications.

The operation *apply()* in each the subtypes of *SoAction* is defined as follows (where *node?* is the input of the operation *apply()*).

$$\begin{array}{rl} SoGLRenderAction: & apply \mathrel{\widehat=} \underline{SoAction::apply}(node?, \Phi render) \\ SoHandleEventAction: & apply \mathrel{\widehat=} \underline{SoAction::apply}(node?, \Phi event) \\ SoSearchAction: & apply \mathrel{\widehat=} \underline{SoAction::apply}(node?, \Phi search) \\ SoWriteAction: & apply \mathrel{\widehat=} \underline{SoAction::apply}(node?, \Phi write). \end{array}$$

For instance, if we want to print out the scene graph in Figure 4.2 a, the operation *apply(body)* defined in *SoWriteAction* is called. Executing *apply(body?)* in *SoWriteAction* invokes $apply(body, \Phi write)$ in *SoAction*, which traverses the scene graph rooted by node *body*. When a node (e.g. *leftLeg* whose type is *SoSeparator*) is traversed during the traversing, the operation $\Phi write$ defined in *SoSeparator* is invoked (e.g. $leftLeg.\Phi write()$). $SoSeparator::\Phi write$ prints out all the information about node *body*, then *apply()* traverses to the next node of *leftLeg* (i.e. *leftTransform*) and $leftTransform.\Phi write()$ prints out all the information about node *leftTransform*. This procedure is carried out until the last node *foot* for the *rightLeg* is dealt with.

For each action, special data structures are needed. For instance, when *SoWriteAction* prints out a node, it is necessary to indicate the medium (e.g. a file or the terminal), in which the scene graph is printed to, and a position from which the current node is printed out. After printing one node, the position should be modified to a new starting position. If the action is rendering, the data structures such as *TS* (Traversal State), *SS* (State Stack) and *SL* (Shape List) which we discussed in Part I in *Inventor and PREMO* are needed. Therefore, we define a set of types which provide proper data structures for each action. Particularly, *SoRenderData* for *SoGLRenderAction, SoEventData* for *SoHandleEventAction*, *SoSearchData* for *SoSearchAction* and *SoWriteData* for *SoWriteAction*. When *apply()* is invoked, first, it creates an instance of the data structure type for the current action. When a node is traversed, the instance of the data structure type is as a parameter passed to the method in that node. For the writing example, if *data* is the instance of *SoWriteAction*, the method $\Phi write$ should be called by $leftLeg.\Phi write(data)$, $leftTransform.\Phi write(data)$ and so on.

The idea of how to implement Inventor's actions in PREMO is illustrated in Figure 4.6. For short, we only consider how to traversal a scene graph. Traversing paths is not considered. Section 3.3 gives an informal description of type *SoAction* and Section 3.4 gives a formal definition.

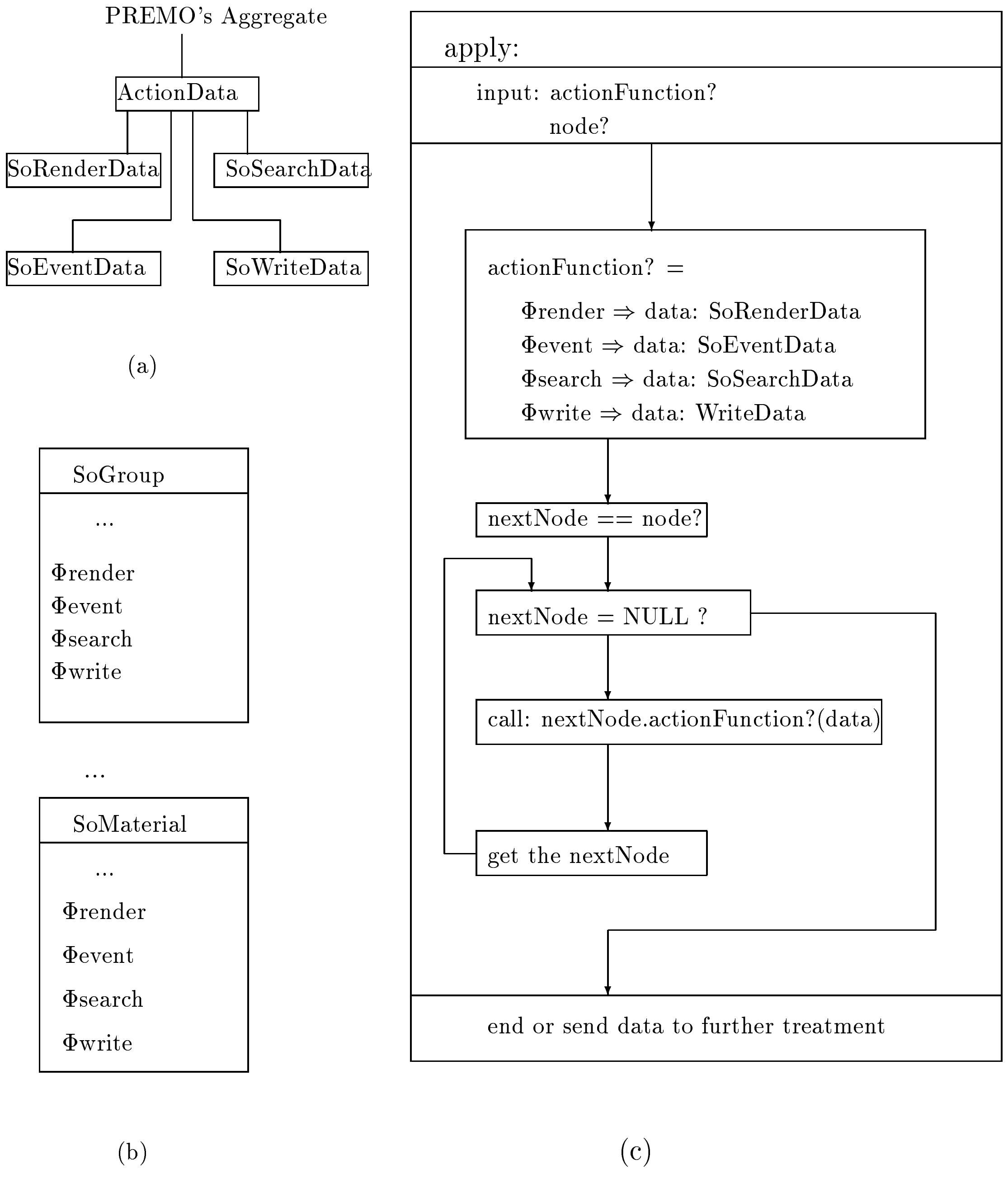


Figure 4.6: (a) is the set of PREMO types for data structures for each action, (b) is the set of PREMO types, whose instance can be a node in a scene graph, extended by adding a set of auxiliary operations to deal with each action, (c) is the flowchart of the operation *apply*.

3. Implementations in PREMO

3.1 Constructions and editions: informal descriptions

*List*1

$\succ$ *List*

findNode

$index_{in}$: $\mathbb{N}$
$node_{out}$: *RefPREMOObject*

This operation returns the object reference of the $index_{in}$*th element of the list, or NULLObject if* $index_{in}$ *is bigger than the length of the list.*
Exceptions raised :
InvalidValue.

findIndex

$node_{in}$: *RefPREMOObject*
$index_{out}$: $\mathbb{Z}$

This operation returns the index of $node_{in}$ *if* $node_{in}$ *is in the list otherwise it returns* -1.
Exceptions raised :
InvalidReference.

addTail

$list_{in}$: *RefList*1

This operation adds $list_{in}$ *to the end of this list.*
Exceptions raised :
InvalidReference

deleteTail

$start_{in}$: $\mathbb{N}$

This operation deletes all the elements in the list started from $start_{in}$*th element.*
Exceptions raised :
InvalidValue InvalidOperation.

findForkIndex

$list_{in}$: *RefList*1
$index_{out}$: $\mathbb{Z}$

This operation finds the index of the element before which all the elements are shared by this list and $list_{in}$, *and after which the next two are not shared. if the first elements of the two lists are not shared, it returns* -1.
Exceptions raised :
InvalidReference.

SoBase$_{abstract}$

$\succ$ *PREMOObject*$_{abstract}$ **redef**(*initialize*), (*get*/*getProperty*)

initialize

This operation initialises the notifying data event flag(M).
Exceptions raised :
None.

touch

$state_{in}$: $\mathbb{B}$
$eventHander_{in}$: *RefEventHandler*

This operation sets $state_{in}$ *to the data event flag* (M) *and stores* $eventHander_{in}$ *in EH if* $state_{in}$ *is TRUE.*
Exceptions raised :
InvalidValue.

set

key_{in} : *Key*
$value_{in}$: *Value*

This operation extends PREMO's defineProperty by notifying data event.
Exceptions raised :
None.

SoNode$_{abstract}$

$\succ$ *SoBase* **redef**(*initialize*)

initialize

This operation extends the SoBase::initialize by :
initialising : N (*the name of the node*),
OF(*override flag*).
Exceptions raised :
None.

setOverride

$state_{in}$: $\mathbb{B}$

This operation sets the Override flag.
Exceptions raised :
InvalidValue.

isOverride

$state_{out}$: $\mathbb{B}$

This operation returns the current state of the Override flag.
Exceptions raised :
None.

SoGroup

$\succ$ $SoNode_{abstract}$ **redef**(*initialize*)

initialize

$numberChildren_{in} : \mathbb{N} \mid \langle\,\rangle$

This operation extends SoNode::initialize by :
initialising : ENC/RNC (expected/real number of children),
*PC(pointing to children) by creating an object of list*1*[SoNode].*
Exceptions raised :
InvalidValue.

addChild

$node_{in} : RefSoNode$

This operation adds the $node_{in}$ *as the last child of this node, and increases this node's children count RNC. This may cause data event if (M) is TRUE.*
Exceptions raised :
InvalidReference.

insertChild

$node_{in} : RefSoNode$
$newChildIndex_{in} : \mathbb{N}$

This operation inserts the child node $node_{in}$ *before the* $newChildIndex_{in}$*th child so that the new child becomes the* $newChildIndex_{in}$*th child and the index of all the children behind the new one is increased by* 1*. If the index of the last child is smaller than* $newChildIndex_{in}$*, the* $node_{in}$ *is added to the end of the list and its index equals the last child's index* + 1*. It may cause data event.*
Exceptions raised :
InvalidReference InvalidValue.

getChild

$index_{in} : \mathbb{N}$
$node_{out} : RefSoNode$

This operation finds the $index_{in}$*th child of the node and outputs its object reference. (It seems that Inventor does not check whether or not the* $index_{in}$*th child exist.) We extend it to return NULLobject if* $index_{in}$*th child does not exist.*
Exceptions raised :
InvalidValue.

findChild

$node_{in} : RefSoNode$
$index_{out} : \mathbb{Z}$

This operation returns the index of $node_{in}$ *if* $node_{in}$ *is a child of this node, or* -1 *if it is not. If* $node_{in}$ *occurs more than once in the group, the index of the first one is returned.*
Exceptions raised :
InvalidValue.

getNumChild

$number_{out}$: $\mathbb{N}$

Thisi operation returns the number of the children of this node.
Exceptions raised :
None.

removeChild

$index_{in}$: $\mathbb{N}$

This operation removes the $index_{in}$*th child of the node.*
This operation may cause the related paths to be changed and data event.
Exceptions raised :
InvalidValue.

removeChild

$node_{in}$: *RefSoNode*

This operation removes the child whose object reference is $node_{in}$.
This operation may cause the related paths to be changed and data event.
Exceptions raised :
InvalidReference.

replaceChild

$index_{in}$: $\mathbb{N}$
$node_{in}$: *RefSoNode*

This operation replaces the $index_{in}$*th child of this node by* $node_{in}$.
This operation may cause the related paths to be changed and data event.
Exceptions raised :
InvalidValue InvalidReference.

replaceChild

$node_{delete\,in}$: *RefSoNode*
$node_{add\,in}$: *RefSoNode*

This operation replaces the object $node_{delete\,in}$ *by the object* $node_{add\,in}$.
This operation may cause the related paths to be changed and data event.
Exceptions raised :
InvalidReference InvalidReference.

SoCone

$\succ$ *SoNode* **redef**(*initialize*)

$\succ$ *TriangleSet*

initialize

This operation extends SoNode::initialize by initialising the default fields.
Exceptions raised :
None.

addPart

$part_{in} : String$

This operation makes the $part_{in}$ *visible.*
Exceptions raised :
InvalidValue. *The correct valuesare* $\{SIDE \mid BOTTOM \mid ALL\}$.

removePart

$part_{in} : String$

This operation makes the $part_{in}$ *invisible.*
Exceptions raised :
InvalidValue.

hasPart

$part_{in} : String$
$part_{value\,out} : \mathbb{B}$

This operation returns the value of the field $part_{in}$
Exceptions raised :
InvalidValue.

SoPath

$\succ$ *SoBase*$_{abstract}$ **redef**(*initialize*)

initialize

$input_{in}$: $\mathbb{N}$ | *RefSoNode* | $\langle\,\rangle$

This operation extends SoBase::initialize by :
initialising : AL(the expected number of nodes in the chain)
PPC(Pointing to Path Chain)by creating an object of List1[SoNode],
if $input_in$ *is RefSoNode it is the first node of the chain.*
Exceptions raised :
InvalidValue.

setHead

$node_{in}$: *RefSoNode*

This operation insert $node_{in}$ *as the first node in the path chain. Because each node in the path chain is a child of the previous node,* $node_{in}$ *must be a parent node of the original first node if the path is not empty.*
Exceptions raised :
InvalidReference.

append

$index_{in}$: $\mathbb{N}$

This operation adds a node to the end of the chain. The added node is the $index_{in}$*th child of the last node of the chain.*
Exceptions raised :
InvalidValue.

append

$node_{in}$: *RefSoNode*

This operation adds $node_{in}$ *to the end of the chain if* $node_{in}$ *is a child of the last node of the chain, or adds* $node_{in}$ *as the first node if the chain is empty.*
Exceptions raised :
InvalidReference.

append

$path_{in}$: *RefSoPath*

This operation appends $path_{in}$ *to the end of the chain. The first node of* $path_{in}$ *must be a child of the last node of the chain.*
Exceptions raised :
InvalidReference.

push

$index_{in}$: $\mathbb{N}$

This operation pushes the last node's $index_{in}$*th child to the end of the chain.*
Exceptions raised :
InvalidValue.

pop

This operation pops the last node of the chain out of the chain.
Exceptions raised :
None.

getHead

$node_{out}$: *RefSoNode*

This operation returns the first node of the chain, or NULLobject if the chain is empty.
Exceptions raised :
None.

getTail

$node_{out}$: *RefSoNode*

This operation returns the last node of the chain, or NULLobject if the chain is empty.
Exceptionsraised :
None.

getNode

i_{in} : $\mathbb{N}$
$node_{out}$: *RefSoNode*

This operation returns the i_{in}*th node of the chain, or NULLobject if it does not exist.*
Exceptions raised :
InvalidValue.

getIndex

i_{in} : $\mathbb{N}$
$index_{out}$: $\mathbb{Z}$

This operation returns the index of the i_{in}*th node in the chain. If* $i_{in} = 0$ (*the head node of the chain*), *returns* -1 (*we do not how to get its parent*(*s*), *if it has more than one parent this does not make sense*) *and if* $i_{in} >$ (*the length of the chain*), *returns* -1.
Exceptions raised :
InvalidValue.

getNodeFromTail

i_{in} : $\mathbb{N}$
$node_{out}$: *RefSoNode*

This operation returns i_{in}*th node backward from the tail of the chain.*
Exceptionsraised :
InvalidValue.

getLength

$length_{out} : \mathbb{N}$

This operation returns the length of the chain.
Exceptions raised :
None.

truncate

$start_{in} : \mathbb{N}$

This operation removes all the nodes started from start?th node to the last node in the chain.
Exceptions raised :
InvalidValue

containNode

$node_{in} : RefSoNode$
$state_{out} : \mathbb{B}$

This operation returns TRUE if $node_{in}$ *is in the chain, otherwise FALSE.*
Exceptions raised :
InvalidReference.

findFork

$path_{in} : RefSoPath$
$index_{out} : \mathbb{N}$

If $path_{in}$ *and this path have different head nodes it returns* -1, *otherwise* $index_{out}$ *is the index of the last node before which all the nodes are shared by the both paths.*
Exceptionsraised :
InvalidReference.

copy

$path_{in} : RefSoPath$
$startIndex_{in} : \mathbb{N}$
$numberNode_{in} : \mathbb{N}$

This operation first empties the chain and then copy $numberNode_{in}$ *number nodes started from* $startIndex_{in}$ *from* $path_{in}$.
Exceptions raised :
InvalidValue InvalidValue.

3.2 Constructions and editions: formal definitions

*List*1

$\succ$ *List*

$list \ : \ \mathrm{seq}\, RefPREMOObject \ (\mathrm{dom}\, list = 0..\#list\text{-}1)$

findNode

$$\begin{array}{l}
\Xi(list) \\
index? \ : \ \mathbb{N} \\
node! \ : \ RefPREMOObject \\
\hline
\#list > index? \Rightarrow node! = list(index?) \\
\#list \leq index? \Rightarrow node! = NULLObject
\end{array}$$

findIndex

$$\begin{array}{l}
\Xi(list) \\
node? \ : \ RefPREMOObject \\
index! \ : \ \mathbb{N} \\
\hline
\exists\, i : \mathrm{dom}\, list \bullet list(i) = node? \land (\forall\, j : \mathrm{dom}\, list \bullet j < i \Rightarrow list(j) \neq node?) \Rightarrow index! = i \\
node? \notin \mathbf{ran}\ list \Rightarrow index! = \text{-}1
\end{array}$$

addTail

$$\begin{array}{l}
\Delta(list) \\
list? \ : \ \mathrm{seq}\, PREMOObject \\
\hline
list' = list \frown list?
\end{array}$$

deleteTail

$$\begin{array}{l}
\Delta(list) \\
start? \ : \ \mathbb{N} \\
\hline
start? \geq \#list \Rightarrow list' = list \\
start? < \#list \Rightarrow list' = \{i \mid start? \leq i \leq \#list\text{-}1\} \lhd list
\end{array}$$

findForkIndex

$$\begin{array}{l}
\Xi(list) \\
list? \ : \ \mathrm{seq}\, PREMOObject \\
index! \ : \ \mathbb{Z} \\
\hline
\mathbf{let} \left[\begin{array}{l} \#list \leq \#list? \Rightarrow min = \#list \\ \#list > \#list? \Rightarrow min = \#list? \end{array} \right] \bullet \\
\qquad min = 0 \lor list(0) \neq list?(0) \Rightarrow index! = \text{-}1 \\
\qquad min \neq 0 \land list(0) = list?(0) \Rightarrow \forall\, k \in \{l \mid 0 \leq l \leq index!\} \bullet list(k) = list?(k) \land \\
\qquad\qquad (index! < min\text{-}1 \Rightarrow list(index! + 1) \neq list?(index! + 1))
\end{array}$$

$SoBase_{abstract}$

$\succ PREMOObject_{abstract}$ $\mathbf{redef}(initialize), (get/getProperty)$

$initialize$

$initialize \mathrel{\widehat=} defineProperty(M, FALSE)$

$touch$

$state? : \mathbb{B}$
$eventHandler? : RefEventHandler$

$$\begin{array}{l}
touch \mathrel{\widehat=} \\
\quad state? = TRUE \Rightarrow \\
\quad\quad defineProperty(M, TRUE) \land defineProperty(EH, eventHandler?) \\
\quad state? = FALSE \Rightarrow \\
\quad\quad defineProperty(M, FALSE)
\end{array}$$

set

$key_{in} : Key$
$value_{in} : Value$

$set \mathrel{\widehat=} \underline{PREMOObject::defineProperty} \land (M = TRUE \Rightarrow get(EH).dispatchEvent)$

$SoNode_{abstract}$

$\succ SoBase$ $\mathbf{redef}(initialize)$

$initialize$

$initialize \mathrel{\widehat=} \underline{SoBase::initialize} \land defineProperty(N, \langle\rangle) \land defineProperty(OF, FALSE)$

$setOverride$

$state? : Boolean$

$setOverride \mathrel{\widehat=} defineProperty(OF, state?)$

$isOverride$

$state! : Boolean$

$isOverride \mathrel{\widehat=} state! == get(OF)$

$SoGroup$

$\succ SoNode_{abstract}$ $\mathbf{redef}(initialize)$

$initialize$

$numberChildren? : \mathbb{N} \mid \langle\,\rangle$

$$\begin{aligned}&initialize \mathrel{\widehat=}\\ &\qquad defineProperty(ENC, numberChildren?) \land defineProperty(RNC, 0) \land\\ &\qquad defineProperty(PC, create(List1[SoNode])) \land \underline{SoNode::initialize}\end{aligned}$$

$addChild$

$node? : RefSoNode$

$$\begin{aligned}&addChild \mathrel{\widehat=}\\ &\quad get(PC).last = NULLObject \;\Rightarrow\\ &\qquad\qquad get(PC).insert(node?) \land set(RNC, get(RNC) + 1)\\ &\quad get(PC).last \neq NULLObject \;\Rightarrow\\ &\qquad\qquad get(PC).insertAfter(node?, get(PC).last) \land\\ &\qquad\qquad\qquad set(RNC, get(RNC) + 1)\end{aligned}$$

$insertChild$

$node? : RefSoNode$
$newChildIndex? : \mathbb{N}$

$$\begin{aligned}&insertChild \mathrel{\widehat=}\\ &\quad getNumChild \leq newChildIndex? \Rightarrow addChild(node?)\\ &\quad getNumChild > newChildIndex? \Rightarrow\\ &\qquad\qquad insertBefore(node?, getChild(newChildIndex?)) \land\\ &\qquad\qquad\qquad set(RNC, get(RNC) + 1)\end{aligned}$$

$getChild$

$index? : \mathbb{N}$
$node! : RefSoNode$

$$\begin{aligned}&getChild \mathrel{\widehat=}\\ &\quad getNumChild \leq index? \Rightarrow node! == NULLobject\\ &\quad getNumChild > index? \Rightarrow node! == get(PC).findNode(index?)\end{aligned}$$

$findChild$

$node? : RefSoNode$
$index! : \mathbb{Z}$

$$findChild \mathrel{\widehat=} index! == get(PC).findIndex(node?)$$

$getNumChild$

$number! : \mathbb{N}$

$$getNumChild \mathrel{\widehat=} get(PC).card$$

removeChild

$index? : \mathbb{N}$

$removeChild \mathrel{\widehat=}$
$\quad index? < getNumChild \Rightarrow \textbf{let}\ node = getChild(index?) \bullet$
$\qquad\qquad get(PC).remove(node) \land set(RNC, get(RNC)\text{-}1)$

removeChild

$node? : RefSoNode$

$removeChild \mathrel{\widehat=}$
$\quad findChild(node?) \neq \text{-}1 \Rightarrow$
$\qquad\qquad get(PC).remove(node?) \land set(RNC, get(RNC)\text{-}1)$

replaceChild

$index? : \mathbb{N}$
$node? : RefSoNode$

$replace \mathrel{\widehat=} removeChild(index?) \land insertChild(node?, index?)$

replaceChild

$node_{delete}? : RefSoNode$
$node_{add}? : RefSoNode$

$replaceChild \mathrel{\widehat=} \textbf{let}\ index = findChild(node_{delete}?) \bullet$
$\qquad index \neq \text{-}1 \Rightarrow removeChild(node_{delete}?) \land insertChild(node_{add}?, index)$

SoCone

$\succ \mathit{SoNode}\ \mathbf{redef}(\mathit{initialize})$

$\succ \mathit{TriangleSet}$

$\mathit{initialize}$

$$\begin{aligned}
&\mathit{initialize} \mathrel{\widehat=}\\
&\quad \underline{\mathit{SoNode}::\mathit{initialize}} \land\\
&\quad \mathit{defineProperty}(\mathit{Sides}, \mathit{TRUE}) \land \mathit{defineProperty}(\mathit{Bottom}, \mathit{TRUE}) \land\\
&\quad \mathit{defineProperty}(\mathit{bottomRadius}, 1) \land \mathit{defineProperty}(\mathit{height}, 2)
\end{aligned}$$

$\mathit{addPart}$

$\mathit{part}? : \mathit{String}$

$$\begin{aligned}
&\mathit{addPart} \mathrel{\widehat=}\\
&\quad \mathit{part}? = \mathit{ALL} \Rightarrow \mathit{defineProperty}(\mathit{Sides}, \mathit{TRUE}) \land \mathit{set}(\mathit{Bottom}, \mathit{TRUE})\\
&\quad \mathit{part}? = \mathit{SIDE} \Rightarrow \mathit{set}(\mathit{Sides}, \mathit{TRUE})\\
&\quad \mathit{part}? = \mathit{BOTTOM} \Rightarrow \mathit{set}(\mathit{Bottom}, \mathit{TRUE})
\end{aligned}$$

$\mathit{removePart}$

$\mathit{part}? : \mathit{String}$

$$\begin{aligned}
&\mathit{removePart} \mathrel{\widehat=}\\
&\quad \mathit{part}? = \mathit{ALL} \Rightarrow \mathit{defineProperty}(\mathit{Sides}, \mathit{FALSE}) \land \mathit{set}(\mathit{Bottom}, \mathit{FALSE})\\
&\quad \mathit{part}? = \mathit{SIDE} \Rightarrow \mathit{set}(\mathit{Sides}, \mathit{FALSE})\\
&\quad \mathit{part}? = \mathit{BOTTOM} \Rightarrow \mathit{set}(\mathit{Bottom}, \mathit{FALSE})
\end{aligned}$$

$\mathit{hasPart}$

$\mathit{part}? : \mathit{String}$

$\mathit{part}_{value}! : \mathbb{B}$

$$\begin{aligned}
&\mathit{hasPart} \mathrel{\widehat=}\\
&\quad \mathit{part}? = \mathit{ALL} \Rightarrow \mathit{part}_{value}! == \mathit{get}(\mathit{Sides}) \land \mathit{get}(\mathit{Bottom})\\
&\quad \mathit{part}? = \mathit{SIDE} \Rightarrow \mathit{part}_{value}! == \mathit{get}(\mathit{Sides})\\
&\quad \mathit{part}? = \mathit{BOTTOM} \Rightarrow \mathit{part}_{value}! == \mathit{get}(\mathit{Bottom})
\end{aligned}$$

SoPath

$\succ SoBase_{abstract}$ **redef**$(initialize)$

initialize

$input? \;:\; \mathbb{N} \mid RefSoNode \;\mid\; \langle\,\rangle$

$$\begin{aligned}
&initialize \mathrel{\widehat{=}} \mathbf{let}\ list = create(List1[SoNode]) \bullet \\
&\qquad \underline{SoBase::initialize} \land defineProperty(PPC, list) \land \{ \\
&\qquad\quad input? \in \mathbb{N} \Rightarrow defineProperty(AL, input?) \\
&\qquad\quad input? = \langle\,\rangle \Rightarrow defineProperty(AL, \langle\,\rangle) \\
&\qquad\quad input? : RefSoNode \Rightarrow defineProperty(AL, \langle\,\rangle) \land list.insert(input?)
\end{aligned}$$

setHead

$node? \;:\; RefSoNode$

$$\begin{aligned}
&setHead \mathrel{\widehat{=}} \\
&\quad getLength = 0 \lor node?.findChild(getHead) \neq \text{-1} \Rightarrow \\
&\qquad\qquad\qquad get(PPC).insert(node?)
\end{aligned}$$

append

$index? \;:\; \mathbb{N}$

$$\begin{aligned}
&append \mathrel{\widehat{=}} \\
&\quad getLength \neq 0 \Rightarrow \mathbf{let}\ node = getTail.getChild(index?) \bullet \\
&\qquad\qquad node \neq NULLobject \Rightarrow \\
&\qquad\qquad\qquad get(PPC).insertAfter(node, getTail)
\end{aligned}$$

append

$node? \;:\; RefSoNode$

$$\begin{aligned}
&append \mathrel{\widehat{=}} \\
&\quad getLength \neq 0 \land getTail.findChild(node?) \neq \text{-1} \Rightarrow \\
&\qquad\qquad get(PPC).insertAfter(node?, getTail) \\
&\quad getLength = 0 \Rightarrow setHead(node?)
\end{aligned}$$

append

$path? \;:\; RefSoPath$

$$\begin{aligned}
&append \mathrel{\widehat{=}} \mathbf{let}\ list = path?.get(PPC) \bullet \\
&\qquad getLength = 0 \lor getTail.findChild(path?.getHead) \neq \text{-1} \Rightarrow \\
&\qquad\qquad get(PPC).addTail(list)
\end{aligned}$$

push

$index? : \mathbb{N}$

$push \mathrel{\widehat{=}} append(index?)$

pop

$pop \mathrel{\widehat{=}} getLength \neq 0 \Rightarrow \mathbf{let}\ node = getTail \bullet \quad get(PPC).remove(node)$

getHead

$node! : RefSoNode$

$getHead \mathrel{\widehat{=}} node! == get(PPC).first$

getTail

$node! : RefSoNode$

$getHead \mathrel{\widehat{=}} node! == get(PPC).last$

getNode

$i? : \mathbb{N}$
$node! : RefSoNode$

$getNode \mathrel{\widehat{=}} node? == get(NHP).findNode(i?)$

getIndex

$i? : \mathbb{N}$
$index! : \mathbb{Z}$

$$\begin{array}{l} getIndex \mathrel{\widehat{=}} \\ \quad i? = 0 \lor i? > getLength \Rightarrow index! == \text{-}1 \\ \quad i? > 0 \land i? \leq getLength \Rightarrow \mathbf{let}\ node = get(PPC).findNode(!?) \bullet \\ \qquad\qquad\qquad index! == (get(PPC).previous.(node)).findChild(node) \end{array}$$

getNodeFromTail

$i? : \mathbb{N}$
$node! : RefSoNode$

$getNodeFromTail \mathrel{\widehat{=}} getLength > i? \Rightarrow node! == getNode(getLength\text{-}(!? + 1))$

getLength

$length! : \mathbb{N}$

$getLength \mathrel{\widehat{=}} get(PPC).card$

truncate

$start? : \mathbb{N}$

$truncate \mathrel{\widehat=} get(PPC).deleteTail(start?)$

containNode

$node? : RefSoNode$
$state! : \mathbb{B}$

$containNode \mathrel{\widehat=} state! == (get(PPC).findIndex(node?) \neq \text{-}1)$

findFork

$path? : RefSoPath$
$index! : \mathbb{N}$

$findFork \mathrel{\widehat=} index! == get(PPC).findForkIndex(path?.get(PPC))$

copy

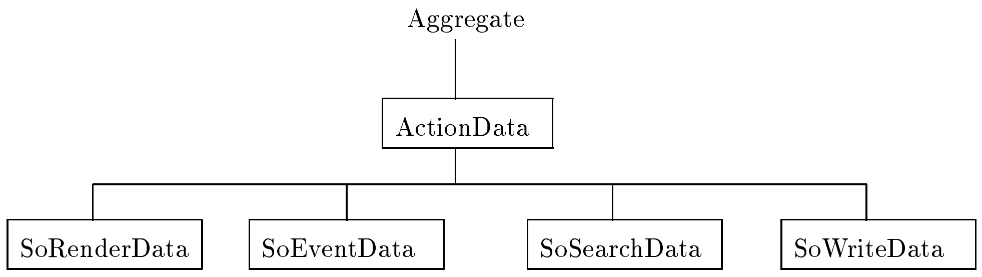
$path? : RefSoPath$
$startIndex? : \mathbb{N}$
$numberNode? : \mathbb{N}$

$copy \mathrel{\widehat=} truncate(0) \parallel append(path?) \parallel truncate(startIndex? + numberNode?) \parallel reverse$
$\quad \parallel truncate(getLength\text{-}(startIndex? + 1)) \parallel reverse$

3.3 Actions: informal descriptions
First, we introduce the following two types. The first one is a non-object type *ActionFunction* which is a subset of PREMO's OperationDescription.

$$ActionFunction \ ::= \ \Phi render \mid \Phi event \mid \Phi search \mid \Phi write$$

Each member in *ActionFunction* has a definition in each non-abstract type in the type tree rooted by *SoNode*. The second one is an object type *ActionData* which is a subtype of PREMO's *Aggregate*. All other types which provide data structures for actions are derived from *ActionData*.



We leave the implementations of the functions in *ActionFunction* and the object types rooted by *ActionData* as further research work.

SoAction

$\succ$ $PREMOObject_{abstract}$ **redef**(*initialize*)

initialize

This operation initialises an action by setting the type of the action and setting the caching flag to TRUE.
Exceptions raised :
None.

setCaching

$flag_{in} : \mathbb{B}$

This operation sets the value $flag_{in}$ to the caching flag.
Exceptions raised :
InvalidValue.

isCaching

$state_{out} : \mathbb{B}$

This operation returns the value in the caching flag.
Exceptions raised :
None.

apply

$f_{in} : ActionFunction$
$node_{in} : RefSoNode$
$data_{out} : RefActionData$

This operation realises an action. According to f_{in} which implies what the current action is, it creates an instance 'data' whose type is one of the subtypes of the ActionData. Then it calls an auxiliary function $\Phi apply(f_{in}, node_{in}, data)$ which returns the result of traversing the scene graph rooted by $node_{in}$. This result is the output, $data_{out}$, of this operation.
Exceptions raised :
InvalidValue

$\Phi apply$

f_{in} : $ActionFunction$
$node_{in}$: $RefSoNode$
$data_{in}$: $RefActionData$
$data_{out}$: $RefActionData$

This operation traverses each node in a scene graph started from $node_{in}$ *and calls function* f_{in} *in a node's type when the node is traversed. During the function* f_{in} *call, data is accumulated and passed from one node to another.* $data_{in}$ *is the initial of the data and* $data_{out}$ *is the data when the last node in the scene graph is passed. This operation is recursively defined. It gets the next node of the current one by calling* $\Phi nextNode$.
Exceptions raised :
None.

$\Phi nextNode$

$node_{in}$: $RefSoNode$
$node_{out}$: $RefSoNode$

This operation returns a node $node_{out}$ *which is the next node of* $node_{in}$*. The order which makes one node next to another is given by Inventor's scene graph traversal principle, i.e. from top to bottom and left to right. If* $node_{in}$ *is the last node of a scene graph, it returns NULLObject. This operation uses a list as a stack to remember each parent node when the traversal goes to its child list.*
Exceptionsraised :
None.

3.4 Actions: formal definitions

$SoAction_{abstract}$

$\succ PREMOObject_{abstract}$ **redef**$(initialize)$

$|\, nodeStack : Reflist[SoNode]$

$initialize$

$initialize \mathrel{\widehat=} defineProperty(Caching, TRUE)$

$setCaching$

$flag? : \mathbb{B}$

$setCaching \mathrel{\widehat=} defineProperty(Caching, flag?)$

$isCaching$

$state! : \mathbb{B}$

$isCaching \mathrel{\widehat=} state == getProperty(Caching)$

$apply$

$f? : ActionFunction$
$node? : RefSoNode$
$data! : RefActionData$

$$
\begin{array}{l}
apply \mathrel{\widehat=} \\
\quad \mathbf{let}\ nodeStack = create(list[SoNode]) \bullet \\
\qquad f? = \Phi render \Rightarrow data! == \Phi apply(f?, node?, create(SoRenderData)) \\
\qquad f? = \Phi event \Rightarrow data! == \Phi apply(f?, node?, create(SoEventData)) \\
\qquad f? = \Phi search \Rightarrow data! == \Phi apply(f?, node?, create(SoSearchData)) \\
\qquad f? = \Phi write \Rightarrow data! == \Phi apply(f?, node?, create(SoWriteData))
\end{array}
$$

$\Phi apply$

$f? : ActionFunction$
$node? : RefSoNode$
$data? : RefActionData$
$data! : RefActionData$

$$
\begin{array}{l}
node? = NULLObject \Rightarrow data! == data? \\
node? \neq NULLObject \Rightarrow data! == \Phi apply(f?, \Phi nextNode(node?), node?.f?(data?))
\end{array}
$$

```
$\Phi nextNode$
$node? : RefSoNode$
$node! : RefSoNode$
--------------------------------
$SoGroup \notin (node?.inquireTypeGraph()) \Rightarrow$
    $nodeStack.card = 0 \Rightarrow node! == NULLObject$
    $nodeStack.card > 0 \Rightarrow$
            $nodeStack.last.get(PC).next(node?) \neq NULLObject \Rightarrow$
                        $node! == nodeStack.last.get(PC).next(node?)$
            $nodeStack.last.get(PC).next(node?) = NULLObject \Rightarrow$
                        $node! == \Phi nextNode(nodeStack.last)$
$SoGroup \in node?.inquireTypeGraph() \Rightarrow$
    $node? \neq nodeStack.last \Rightarrow$
            $node?.get(PC).first \neq NULLObject \Rightarrow$
                        $node! == node?.get(PC).first \wedge$
                            $nodeStack.insertAfter(node?, nodeStack.last)$
            $node?.get(PC).first = NULLObject \Rightarrow$
              $nodeStack.card = 0 \Rightarrow node! == NULLObject$
              $nodeStack.card \neq 0 \Rightarrow nodeStack.insertAfter(node?, nodeStack.last) \wedge$
                        $node! == \Phi nextNode(node?)$
    $node? = nodeStack.last \Rightarrow$
            $nodeStack.card = 1 \Rightarrow node! == NULLObject$
            $nodeStack.card > 1 \Rightarrow$
               $\textbf{let } node = nodeStack.last \bullet nodeStack.remove(nodeStack.last)$
                 $nodeStack.last.get(PC).next(node) \neq NULLObject \Rightarrow$
                        $node! == nodeStack.last.get(PC).next(node)$
                 $nodeStack.last.get(PC).next(node) = NULLObject \Rightarrow$
                        $node! == \Phi nextNode(nodeStack.last)$
```

4. Conclusion

In this report, we described how Open Inventor's graphics rendering and event handling are defined in PREMO's environment model and the event model, and how Open Inventor's scene graph classes are defined by means of a set of PREMO non-object and object types in PREMO's foundation component. Open Inventor's rendering action and event handle action working on scene graphs provide Open Inventor's fundamental functions, i.e. renderings and interactions. We conclude that since these three parts can be properly represented in PREMO, that the concepts and objects of PREMO are sufficient to represent Open Inventor.

Acknowledgements

The work presented in this paper has been carried out under the auspices of the ERCIM Computer Graphics Network funded under the CEC HCM Programme.

We would like to thank Robert van Liere who read an early draft of Part I of this report and gave valubal comments.

References

1. D.B. Arnold and D.A. Duce. *ISO Standards for Computer Graphics: The First Generation.* Butterworths, London, 1990.
2. (ed.) G.S. Carson. Introduction to the computer graphics reference model. *Computer Graphics*, 27(2):108–119, September 1993.
3. I. Herman, G.S. Carson, J. Davy, P.J.W. ten Hagen, D.A. Duce, W.T. Hewitt, K. Kansy, B.J. Lurvey, R. Puk, G.J. Reynolds, and H. Stenzel. Premo: an ISO standard for a presentation environment for multimedia objects. In D. Ferrari, editor, *Proceedings of the Second ACM International Conference on Multimedia (MM'94)*, San Francisco, CA, October 1994. ACM Press.
4. I. Herman, G.J. Reynolds, and J. Van Loo. Premo: An emerging standard for multimedia presentation, Part I: Overview and framework. *IEEE Multimedia*, 3, 1996, to appear.
5. International Organisation for Standardisation, Geneva. *Information processing systems — Computer graphics — Graphical Kernel System (GKS) functional description (ISO IS 7942)*, 1985.
6. International Organization for Standardisation, Geneva. *Information processing systems — Computer graphics — Computer Graphics Reference Model (CGRM) (ISO/IEC IS 11072)*, 1 edition, 1992.
7. ISO/IEC 14478-1: 1996(E). *Information processing system — Computer graphics and image processing — Presentation Environments for Multimedia Objects (PREMO) — Part1: Fundamentals of PREMO.*
8. ISO/IEC 14478-2: 1996(E). *Information processing system — Computer graphics and image processing — Presentation Environments for Multimedia Objects (PREMO) — Part2: Foundation Component.*
9. ISO/IEC 14478-3: 1996(E). *Information processing system — Computer graphics and image processing — Presentation Environments for Multimedia Objects (PREMO) — Part3: Modelling, Rendering, and Interaction Component.*
10. ISO/IEC 14478-4: 1996(E). *Information processing system — Computer graphics and image processing — Presentation Environments for Multimedia Objects (PREMO) — Part4: Multimedia System Services Component.*
11. Neider, Jackie, Tom Davis, and Mason Woo. *OpenGL Programming Guide.* Addison Wesley, 1993.
12. Silicon Graphics, Inc, Mountain View, California. *IRIS Inventor C++ Reference Manual*, 1993.
13. Josie Wernecke and Open Inventor Architecture Group. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*^{TM}*, Release 2.* Addison Wesley, 1994.