The open inventor toolkit and the PREMO standard

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

Computer Science/Department of Interactive Systems

CS-R9624 1996

Report CS-R9624
ISSN 0169-118X

The Open Inventor Toolkit and the PREMO Standard

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

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 paper, 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 and event model can properly describe Open Inventor's rendering action and event model. The scene graph is very important in Open Inventor. Most Open Inventor functions rely on various operations over scene graphs. The construction, edition and traversal of the scene graphs are implemented as a set of newly defined PREMO objects. Graphics rendering, event handling and scene graphs 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 means of PREMO, that 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 paper, 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 paper. 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. One can also visit the www site `http://www.cwi.nl/Premo/` to get up-to-date information on PREMO.

1.2 Open Inventor

Open Inventor[13] 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 paper, we only discuss 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 paper. 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 paper, 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. Obviously, not all details and results of this investigation can be presented in this paper, and only the most important features are described. A report is publicly available[12] for those who wish to familiarise themselves with all the details.

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 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 paper.
- *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.

- 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.

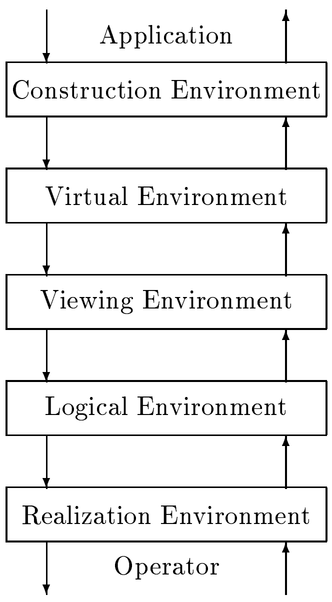

Figure 0.1: Environments in PREMO's Part 3.

- 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 of 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. Dashed and continuous arrows represent control and data flows, respectively, among the different entities. In Figure 0.2, the *composition* is a spatially structured set of output primitives in a given environment ready for distribution. A *collection store* is a set of output entities which are intended for use within the environment. A *token store* is a structured set of input tokents in a given environment reday for emanation. An *aggregation store* is a set of input entities which are intended for use within the environment. An environment may contain an *environment state* separated from the other data

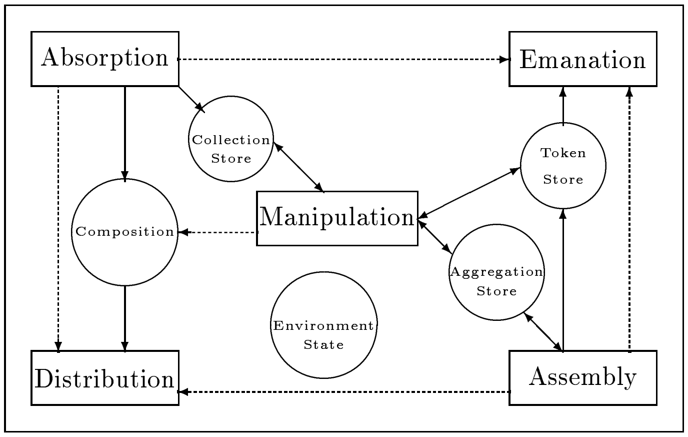

Figure 0.2: A PREMO Environment.

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.

elements in the environment. *Absorption* is the process which receives output entities from the next higher environment and applies the transformations necessary to produce the entities in the form appropriate to its environment. *Manipulation* may process entities or primitives in any of the data elements. *Distribution* is the process which passes entities to the next lower environment. *Assembly* is the process which receives input entities from the next lower environment and passes them to the aggregation store or derives tokens in the token store of the current environment. *Emanation* is the process which passes entities to the next higher environment. Each process and data element in this network bear different names in the concrete environment (see in Table 0.1). The PREMO object types defining the various environments are not the only types defined in Part 3. Both output primitives 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 aggregate 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 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.

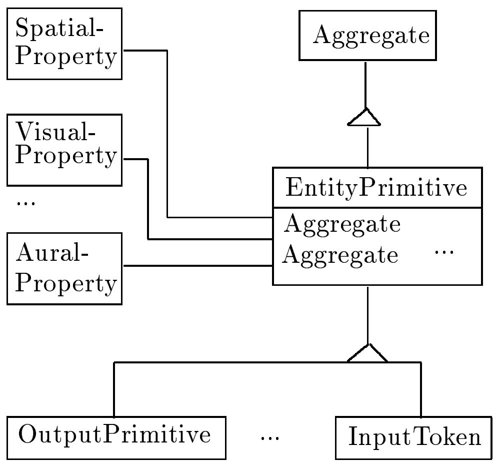

Figure 0.3: PREMO object types for output primitives and input tokens.

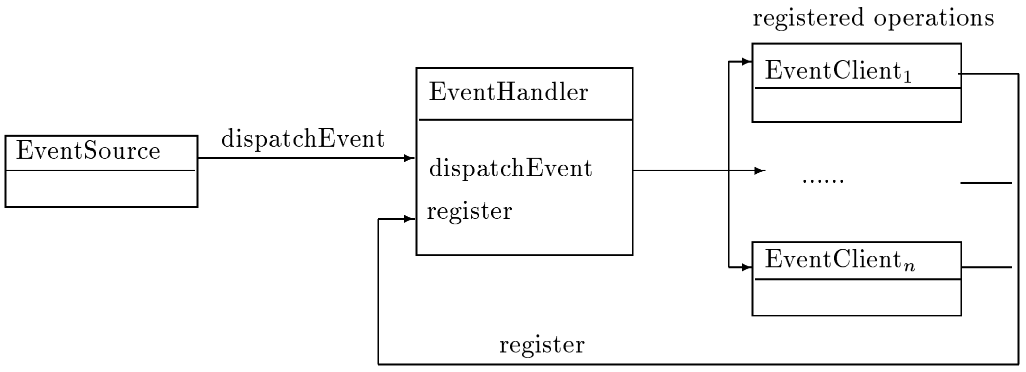

Figure 0.4: PREMO Event Model.

step	program

1	*Widget* myWindow = *SoXt::init*(*argv*[0]);
	if (myWindow == *NULL*) *exit*(1);

2	*SoSeparator* *root = *new SoSeparator*;
	SoPerspectiveCamera *myCamera = *new SoPerspectiveCamera*;
	SoMaterial *myMaterial = *new SoMaterial*;
	root → *ref*();
	root → *addChild*(myCamera);
	root → *addChild*(*new SoDirectionalLight*);
	myMaterial → *diffuseColor.setValue*(1.0, 0.0, 0.0);
	root → *addChild*(myMaterial);
	root → *addChild*(*new SoCone*);

3	*SoXtRenderArea* *myRenderArea = *new SoXtRenderArea*(myWindow);
4	myCamera → *viewAll*(root, myRenderArea → *getViewportRegion*());

5	myRenderArea → *setSceneGraph*(root);
	myRenderArea → *setTitle*("Hello Cone");

6	myRenderArea → *show*();
	SoXt::show(myWindow);

7	*SoXt::mainLoop*();

Figure 0.5: The Open Inventor code of the red cone program.

The rest of this paper describes some of the result of our comparative work between PREMO and Open Inventor. The points of comparison described in this paper are:

- Open Inventor's rendering actions and the output in PREMO's environmental model.
- Open Inventor's event handling and the input in PREMO's environmental model and the PREMO event model.
- Open Inventor's scene graph management and the foundation component of PREMO.

Conventions: In what follows, we use *slanted* text font for all Open Inventor's classes and methods, and SANS SERIF text font for all PREMO's object types and operations.

2. Rendering

In this section, we study the steps by which Open Inventor displays graphics according to application programs and discuss the relationship between those steps and the output procedure of PREMO's environment model.

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 within a red cone in the render area. The code of this program is shown in Figure 0.5 (from page 23 in[13]). Though 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, which initialises Open Inventor.
2. *Scene graph specification*: The scene graph contains all the information of the graphics which can be shown in the render area.
3. *Creating a render area*: The render area, applied from the window management system, is the place where Open Inventor displays the computer graphics and receives events from.
4. *Camera viewing*: The graphics specified in the scene graph is transformed into graphics ready to be displayed on the screen.
5. *Sending to the render area*: Put the scene graph into the render area.

6. *Realisation*: Show the window including the graphics inside its render area.
7. *Main loop*: This loop will retrieve and dispatch events.

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. When we represent Open Inventor by means of PREMO's environment model, the window specific part is not considered to be a part of the environment model, it is an operator, an external object that observes the contents of the display in the realization environment and provides physical input tokens. Program step 2, i.e., the scene graph specification, can be considered as the main body of the application.

Open Inventor interprets the scene graph specification and constructs a scene graph. This can be presented in PREMO's construction environment. Program step 4 instructs Open Inventor to realise the graphics specified by the scene graph. Conceptually, this is comparable to applying a rendering action to the scene graph. First, Open Inventor traverses the scene graph to produce a list. Each element of the list is a complete specification of a geometrical shape ready for being rendered. It then passes the list of shapes to OpenGL which does the rendering. The scene graph traversal can be represented in PREMO's virtual environment and the rendering performed by OpenGL belongs to PREMO's viewing to realization environments. During the lifetime of the application, whenever the scene graph is changed, Open Inventor's rendering action is applied to the scene graph so that a new scene is shown in the display end.

Let us see what happens in more detail. Taking the program in Figure 0.5 as an example, a scene graph is first built in Open Inventor's data base (see in Figure 0.6). The scene graph is a Model and the process which builds the graph from an application is the Preparation process in the construction environment.

Second, the scene graph is traversed. The data base manages a *traversal state* for each action. For the rendering action, the elements in the traversal state include: current geometrical transformation, current material components and so on. The traversal state is initialised by the default values. During the traversal, nodes in the scene graph can modify the traversal state depending on their particular behaviour for rendering action. The order of the traversal is from the scene graph's top node to bottom node and left node to right node. For this example, the traversal order is: the root, the camera, the light, the material and the cone.

Nodes in the Model (i.e. the scene graph) can be divided into three kinds: shape nodes, property nodes and group nodes. Shape and property nodes are terminal nodes and group nodes are non-terminal nodes in scene graphs. In this example, the root is a group node, the cone is a shape node and the rests are property nodes.

A *group* node is a container for collecting child nodes. There are a variety of different group-node classes, each with a specialised grouping characteristic. The root used in this example is a *SoSeparator* node which isolates the effects of nodes in a group. Before traversing its children, a *SoSeparator* saves the current traversal state. When it has finished traversing its children, the *SoSeparator* restores the previous traversal state. Nodes within an *SoSeparator* thus do not affect anything above or right in the graph. (In Section 4, we will discuss how to model the non-terminal nodes in PREMO.)

Property nodes modify the traversal state. For instance, the node *myMaterial* in this example changes the *diffuseColor* of the current material components in the traversal state from white (the default) to red.

When a *shape* node is encountered during a rendering traversal, all the elements in the current traversal state are attached to this shape. In this example, the colour of the cone is red and it is illuminated by a directional light. Both property nodes and shape nodes can be modelled as PREMO's various basic features, e.g., a shape node is modelled as a SpatialProperty and a light node as a VisualProperty in PREMO.

The output of a rendering traversal is a set of shapes each of which is inside the view volume and completely specified for rendering. In the example considered here, the output shape list contains only one element, specifying the cone (see the illustration in Figure 0.6). Such an output is a Scene and the rendering traversal is the Production in virtual environment. The traversal state is part of the virtual environment state. Each element in the shape list can be modelled as a PREMO's OutputPrimitive which is an aggregate of the basic features and thus the shape list is a PREMO's List[OutputPrimitive], i.e., a PREMO list structure whose elements are of type OutputPrimitive.

Open Inventor, then, passes the Scene to OpenGL which realises the rendering. The above is illustrated in Figure 0.6. For a more complete description of the rendering process and its correspondence with PREMO see also[12].

As discussed, there is a good match between the rendering process of Open Inventor and the concepts of PREMO. It also shows that the main issue in implementing Open Inventor on the top of PREMO is a proper implementation of the scene graph, including the various operations on the graph such as its construction, edition, traversal etc. The implementation strategy is discussed in Section 4.

3. Event handling

Open Inventor has two kinds of events, those created by hardware devices and those created by changes of data and time. In this section, we first discuss Open Inventor's event model, i.e. those events created by hardware devices. Consuming information received from hardware devices is often an important part of this event handling. Therefore, we compare this procedure with PREMO's input pipeline to show that the concepts provided by PREMO can represent this procedure in Open Inventor. Second, we briefly introduce Open Inventor's data and time events, and, finally, discuss their implementation in PREMO.

3.1 Open Inventor's event model and PREMO's input

Open Inventor has a set of classes for modelling the events created by hardware devices (see in Figure 0.7). Each instance of the classes contains the event data, e.g. type identification and time, cursor position and state of the modifier keys (Shift, Control, Alt) when the event occurred. Open Inventor's event classes are independent from window systems. When Open Inventor is implemented in a specific window system an event translator has to be written, e.g., a translator for X11. The event translator converts window system specific events into Open Inventor events. Open Inventor provides three [1] mechanisms to handle such events. All these mechanisms work by placing particular nodes into a scene graph. Since Open Inventor traverses the scene graph whenever an event occurs, Open Inventor can take appropriate action if it encounters the event handling node.

1. *Callback*: Open Inventor provides a *SoCallback* class. Instances of it can be placed as a node in a scene graph. Each time the scene graph is traversed, the program specified in these nodes will be invoked.
2. *Event callback*: The class *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* provides some additional methods that are useful for handling events.
3. *Automatic event handling*: Instances of *SoSelection* and *SoManipulator* and its subclasses are called "smart nodes" which handle events automatically. For example, When an *SoSelection* node is encountered, the event handling action 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 will itself handle it.

Handling events by means of the first two mechanisms relies on the callback functions provided by the application. The task of the system is only to find the callback node and then invoke the specified function.

The way in which a smart node handles event is, basically, by modifying the scene graph. The changes of the scene graph triggers the system to apply the rendering action to it so that the modified scene graph is shown on the screen.

[1] Open Inventor also provides another mechanism which allows users to handle the window system events directly, e.g. directly handle X-window system event. However, this is not interesting in our study of Open Inventor.

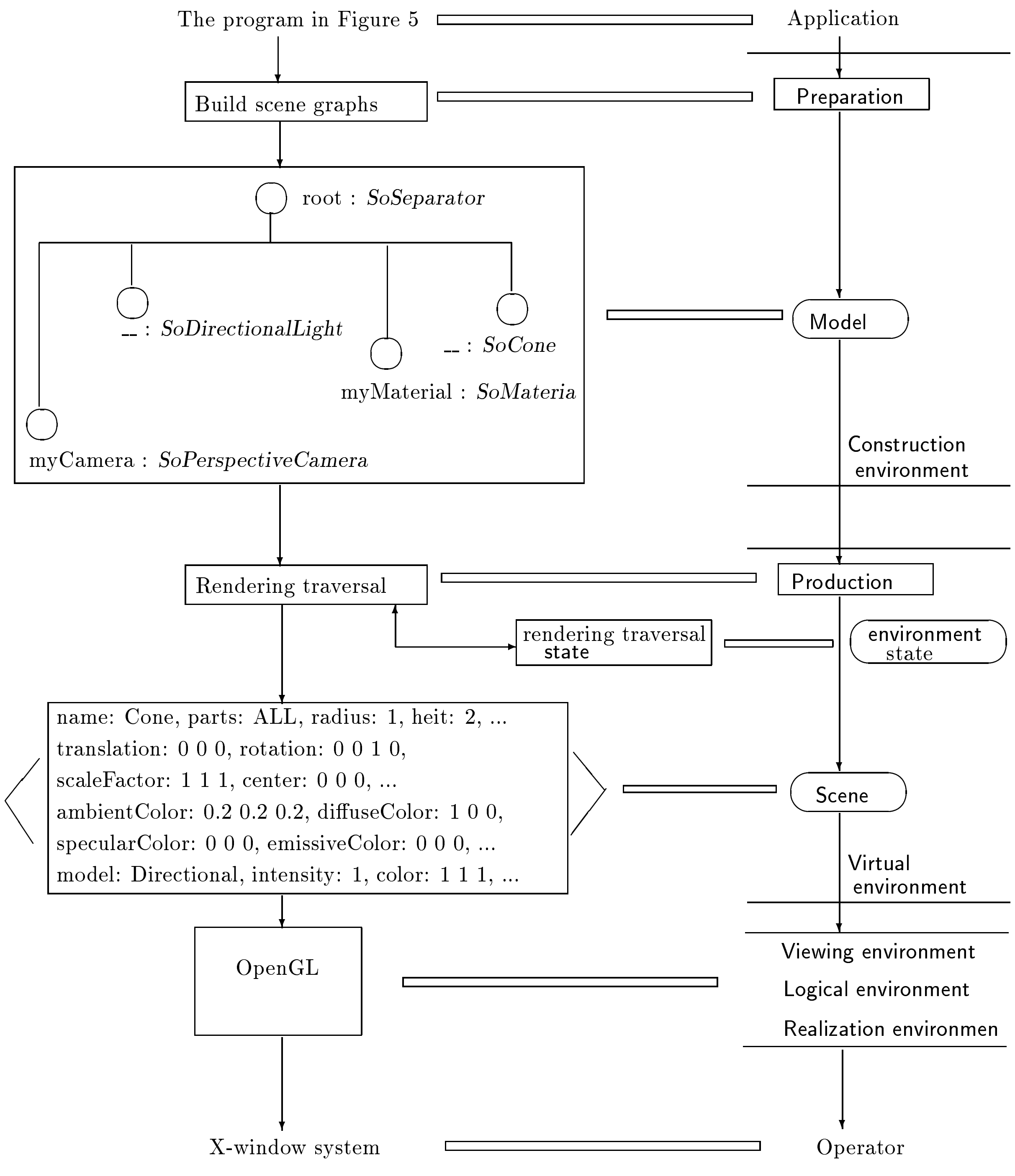

Figure 0.6: An example of Open Inventor's rendering procedure (in the left side) and PREMO's output pipeline (in the right side).

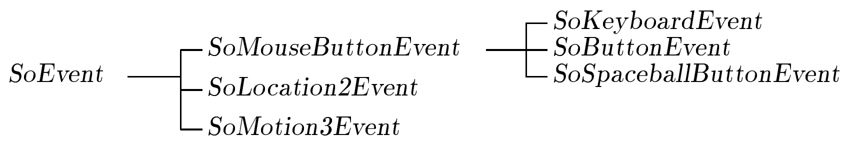

Figure 0.7: Open Inventor's event classes.

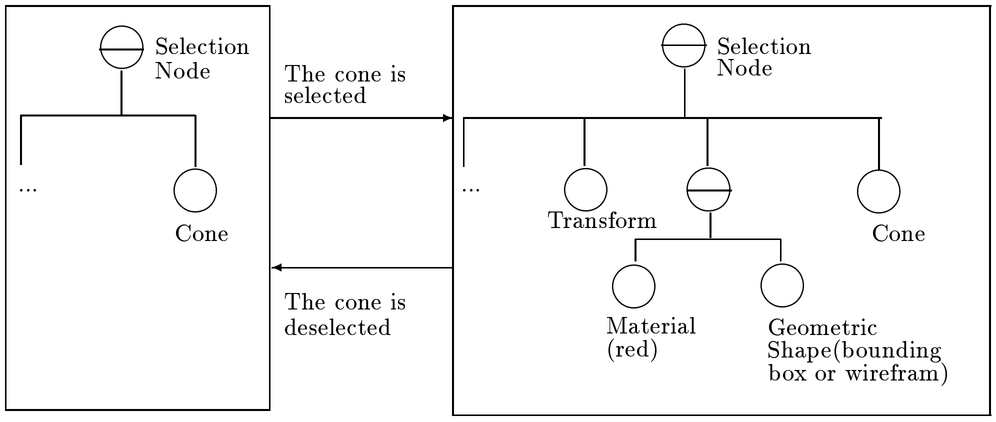

Figure 0.8: The changes of the scene graph between the cone is selected and deselected.

Let us see how the *SoSelection* node works in more detail. The *SoSelection* node provides several features that relate to the topic of user interaction. These features include managing the *selection list* (a data structure to keep all the selected objects), 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()* method to find which node is picked. It then manages the selection list based on the current picked node, the old selection list and the user's demand (or the default strategy). If an object is selected, the *SoSelection* node will insert a bounding box in the scene graph so that it appears on the top of the selected object. Since the scene graph has been changed, the system will apply the render action *SoGLRenderAction* to the scene graph so that the selected object is then highlighted by showing a bounding box around it. If an object is deselected, the *SoSelection* node will delete the corresponding bounding box from the scene graph. This also causes the system to apply the rendering action and so stops the highlighting of the object. Figure 0.8 illustrates the changes in a scene graph when a cone is selected and deselected.

The data flow handling selection and manipulation can be modeled by the input part of PREMO's environment model (see in Figure 0.9). A window system event is stored in the Lexeme store which is translated into an Open Inventor event. The Open Inventor event, here, is represented as a PREMO event consisting of an event name, event data and the event source. The event name and data are stored in the Information store. The window system event translator is the Accumulation process. An *SoSelection* or *SoManipulator* node calls the *getPickedPoint()* method to get the picked node. *getPickedPoint()* is a method in the handle event action class (*SoHandleEventAction*). 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 the Instruction Store. There are several ways in which the *getPickedPoint()* method can be implemented in PREMO's environment model. For instance, the *cursor position*, which is part of the event data and is a 2D point relative to the left lower corner of the window, can be translated to a point of the coordinate system in the viewing environment. The translation process is the Abstraction in the logical environment and the translated point is stored in the Selection Store in the viewing environment. 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 is related to it. Other input tokens such as location are passed from lower environments to the construction environment in a similar way. Representing Open Inventor's rendering and selection in PREMO is illustrated in Figure 0.9.

Here again, we see that the manipulation of the scene graph, such as its edition and traversal, plays an important role for handling Open Inventor's events. In other words, if the basic functionalities provided by Open Inventor's scene graph manager can be implemented by means of PREMO, it would not be difficult to implement Open Inventor's event model in PREMO.

3.2 Open Inventor's data and time events

Open Inventor also supports events that arise from the changes of data and time. Such events occur within Open Inventor, but are not strictly part of the event model. Data events come from the changes of the data in scene graphs. Time events are alarm, interval, elapsed time, global time and so on.

Open Inventor has two kinds of mechanisms, termed sensors and engines, that can be used to specify data and time events and response to these events. Each type of time sensors and time related engines corresponds to a particular time event. Sensors and engines can be attached to the components of a scene graph. Since a sensor is attached to a field of a node, a node, or a path, every change in the field, the node or the path results in a data event.

When a sensor is fired, it calls on user-defined callback functions. When an engine is fired it calls on system built-in functions. Therefore, 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 to the Timer queue. When the data is changed, the corresponding data sensor is added to the Delay queue. The ordering of sensors in the Timer queue is naturally given by the value of the time. The order in the Delay queue is according to the priorities which are 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 is invoking the related callback functions and processing the hardware events is applying the event-handling action to the scene graph. The sequence of scheduling events in the queues depends on the window system.

3.3 Implementations in PREMO

Event model Apart from scene graphs (see Section 4), we also need to have a strategy to handle the various Open Inventor 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 0.10. 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.

We can fit the event model in Figure 0.10 into PREMO's environment model, where the window specific events are in the Lexeme Store, the InputPorter which generates events is the Accumulation

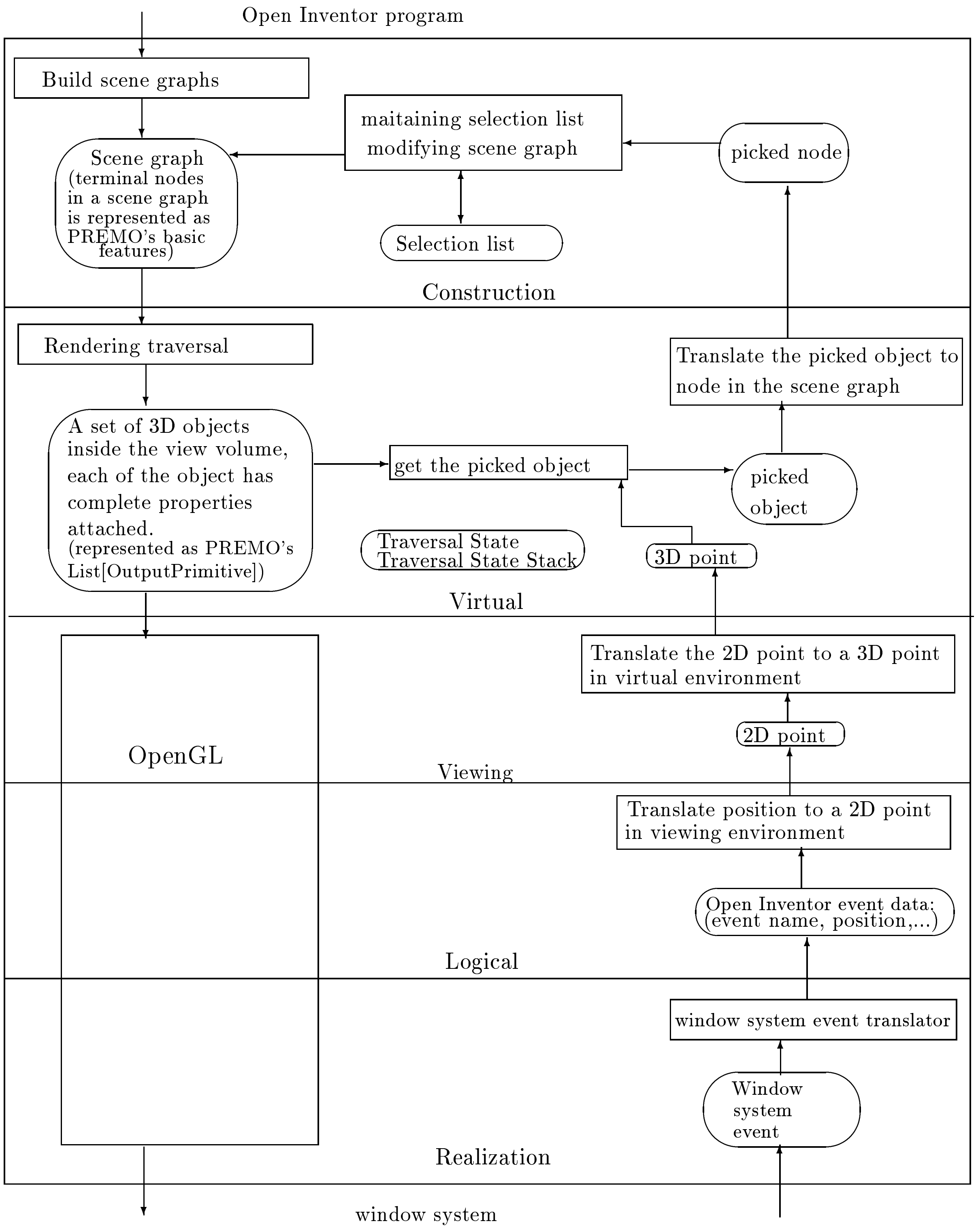
Open Inventor program
Build scene graphs
maitaining selection list
modifying scene graph
picked node
Scene graph (terminal nodes in a scene graph is represented as PREMO's basic features)
Selection list
Construction
Rendering traversal
Translate the picked object to node in the scene graph
A set of 3D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

Figure 0.9: Represent Open Inventor's rendering and object selection by PREMO's environment model.

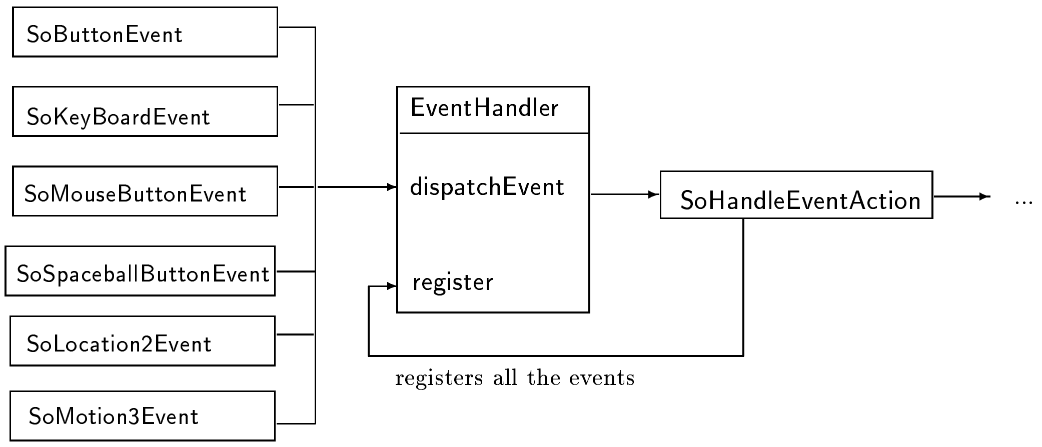

Figure 0.10: Modelling Open Inventor's event model in PREMO.

process and so on (see also in Figure 0.9 where the “window event translator” is the InputPorter). 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 0.10 is in the Logical environment. When it is invoked by the dispatchEvent() operation, it only does the necessary job in Logical environment and passes the event to the higher 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 for the application and then register the necessary events. It is obvious that a model with the EventHandler is much more flexible.

Time events Modelling Open Inventor's time events in PREMO is illustrated in Figure 0.11. 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 maintains 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.

Data events Modelling Open Inventor's data events in PREMO is illustrated in Figure 0.12. 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

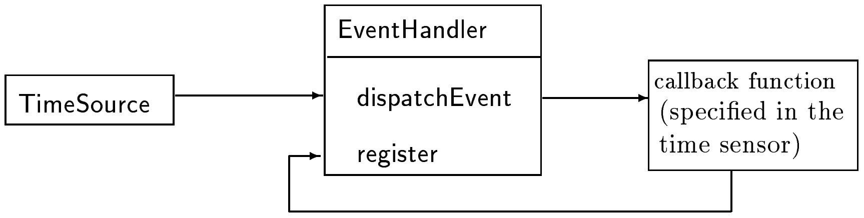

Figure 0.11: Modelling Open Inventor's time sensors in PREMO.

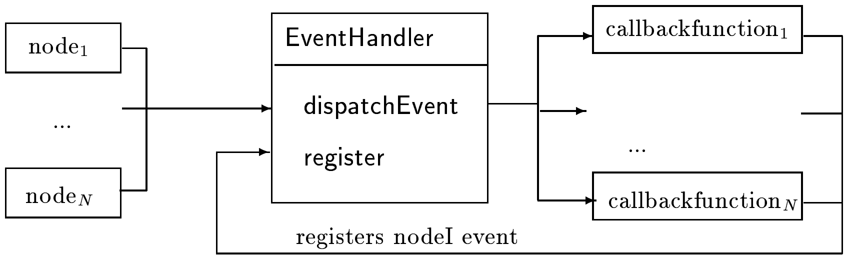

Figure 0.12: Modeling Open Inventor's data events in PREMO.

data sensor. When the flag is switched on, the connected sensor will be notified whenever an operation which contains writing instructions was applied on this node. If 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.

Both the time event model and the data event model conceptually belong to PREMO's Construction environment. More precisely, the user-defined callback functions are part of the application and they should be invoked through the Utilization process in the Construction environment.

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 schedule policies is not a feature of Open Inventor itself, but 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.

4. Scene graph

"Scene graph" is the most important feature of Open Inventor. We have seen that Open Inventor's most important functions such as rendering and event handling rely on working on scene graphs. Other functions, such as writing scene graphs, picking, searching, and so on, also rely on manipulations on scene graphs. Therefore, it is important to know whether Open Inventor's scene graph management can be properly implemented in PREMO. We will concentrate on some aspects of scene graph management here; in our view, 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). We have defined a set of PREMO object types (reported in [12]) corresponding to the set of Open Inventor classes whose methods provide these two aspects. In this section, we give only a brief discussion of these two aspects.

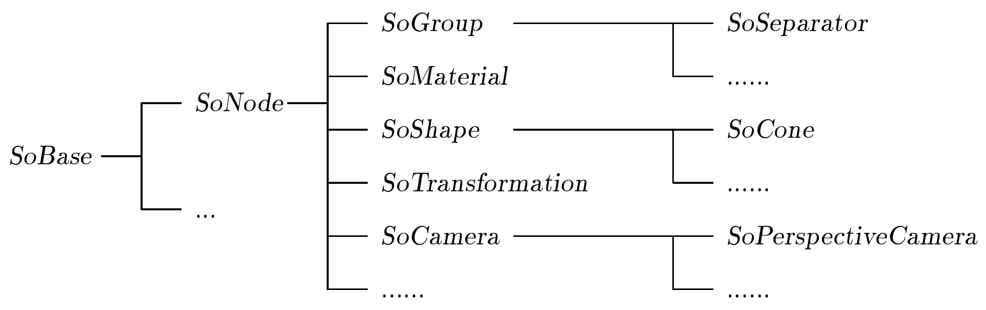

Figure 0.13: The Open Inventor class tree for building and manipulating scene graphs and paths in a scene graph.

4.1 Constructions and editions of scene graphs
In order to implement scene graphs in PREMO, we need to answer two questions: (1) how to implement nodes and (2) how to connect the nodes together to form a graph.

There is a set of Open Inventor classes related to the construction and the editing of scene graphs. The subclass relationship between these classes is shown in Figure 0.13. *SoBase* and *SoNode* (which are abstract classes, i.e. they are not instantiable) describe the minimum behaviour for all nodes. The functionalities provided in these two classes can be summarised as: (1) controlling the lifetime of a node, (2) support data events, (3) inquiry node type and (4) node property. In what follows, we will discuss them in more detail and give an idea of how these are implemented in PREMO.

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. For instance, when a node is specified as a child of another node (e.g. root→ addChild(myMaterial)), its reference count is incremented by 1 and when it is removed from its parent its reference count is decremented by 1. When the reference count for the node is decremented to 0, the node is destroyed unless its *no deleting flag* is switched on.

   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 a problem does not exist.

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

   As we discussed in Section 3.3, this is implemented in PREMO by relating the node with the corresponding EventHandler (i.e. store the object reference of the EventHandler in this node) when the flag is switched on, and the set property functions in each node will check this flag and invokes the dispatchEvent() operation on the EventHandler if the flag is on.

3. *Inquiry node type*: The methods 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,

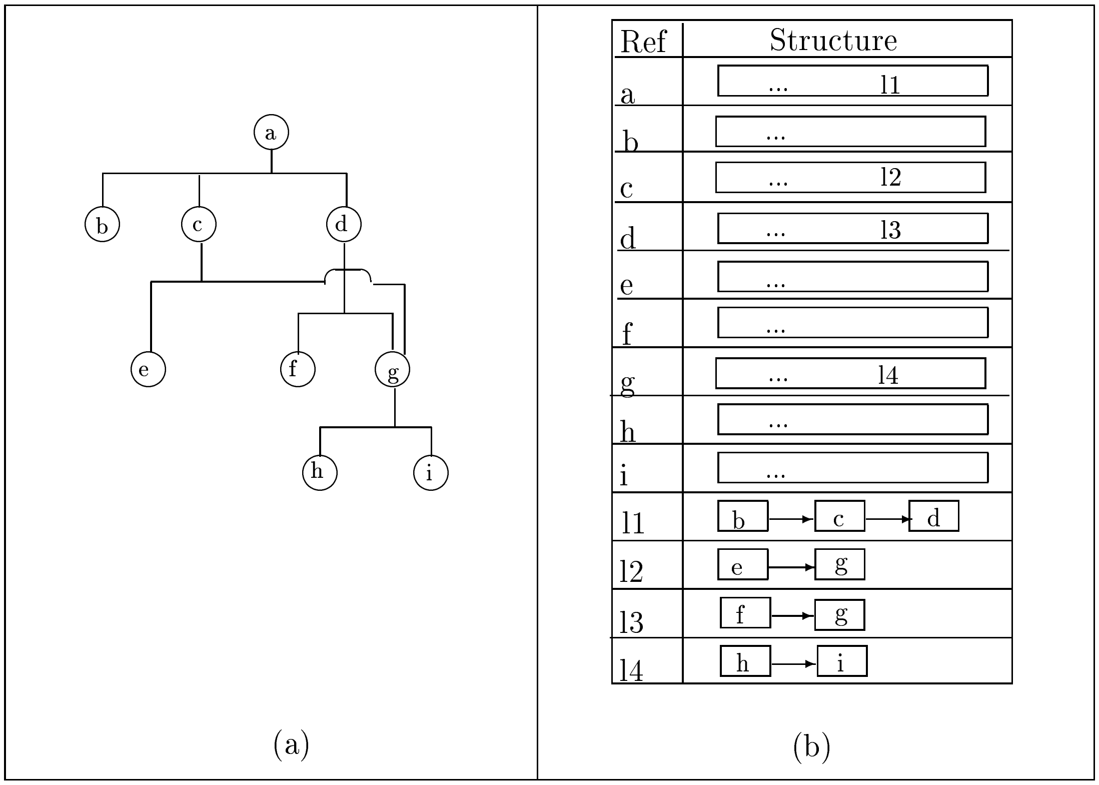

Figure 0.14: (a) is an example scene graph and (b) is the representation in PREMO, where *Ref* stands for the object reference of a node.

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*: There 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.

We can divide the remaining classes in Figure 0.13 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.

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 *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. Consider a scene graph illustrated in Figure 0.14a, its representation in PREMO is illustrated in Figure 0.14b.

4.2 Scene graph traversal

The scene graph traversal is the basic function for acting on scene graphs. When an action is applied on a scene graph, the scene graph is traversed according to the order defined by the scene graph, i.e. from top to bottom and from left to right. When it is encountered during a traversal, the node reacts according to what the traversal action is. For instance, if an *SoSeparator* node is encountered during a rendering traversal, it saves the current traversal state and starts to traverse its children. When it has finished traversing its children, it restores the previous traversal state. When a separator node is encountered during an event handling traversal, it only traverses its children.

There is a set of Open Inventor classes rooted by *SoAction*, each of them deals with a specific action such as rendering, event handling and writing. An important method in the action classes is the *apply()* which accepts a scene graph node as input and then traverses the scene graph rooted by the input node with its specific task.

We implement the action classes in PREMO in a rather abstract way in which the specific behaviours which should be given by each kind of nodes for a particular action are ignored. The implementation only reflects the general structure of applying actions on scene graphs. Details of the implementation are in [12].

5. Conclusions

In this paper, we briefly described how Open Inventor's graphics rendering and event handling are represented in PREMO's environment model and the event model, and how Open Inventor's scene graph classes are implemented by means of a set of PREMO non-object and object types in PREMO's foundation component. The fundamental functions of Open Inventor are rendering and event handling using scene graphs. Since these three parts can be properly represented in PREMO, it is reasonable to conclude 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.

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. D. Wang, G. Reynolds, and I. Herman. Open Inventor and PREMO. CS-R9606 ISSN 0169-118X, Department of Interactive System,CWI, P.O.Box 94079, 1090 GB Amsterdam, The Netherlands. ftp://ftp.cwi.nl/pub//CWIreports/IS/CS-R9606.ps.Z, February 1996.

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.