

BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M9

Interaction Objects

by
Mark L. Stern

Interaction Objects

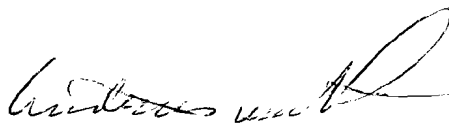
Mark L. Stern

Department of Computer Science
Brown University

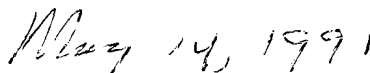
Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Department of Computer Science
at Brown University

May 1991

This research project by Mark L. Stern is accepted in its present form
by the Department of Computer Science at Brown University
in partial fulfillment of the requirements for the Degree of Master of Science.



Professor Andries van Dam
Advisor



Date

Abstract

The research and development of interaction techniques is currently hindered by inadequate programming abstractions and the lack of an unifying framework, thus requiring low-level programming and a laborious cycle of compiling and testing code. This research project introduces a structural framework for the development of *interaction objects* to be used in a 2D or 3D environment. Interaction objects can be thought of as software “agents” that recognize and respond to interactive gestures. Each interaction object supports an interaction technique, and several interaction objects can respond to a single interactive gesture in different ways. The design of the structural framework strives to satisfy four research objectives: separability, generality, flexibility, and usability. An initial implementation, described in detail in the appendix, shows promise for the viability of this design and helps identify areas for future work.

Contents

1	Introduction	1
2	Research Objectives	2
2.1	Separability	2
2.2	Generality	3
2.3	Flexibility	3
2.4	Usability	3
2.5	Interaction Modeling	3
3	Related Work	3
3.1	Garnet	4
3.2	BAGS	4
4	Design	5
4.1	Representation	6
4.2	Communication Mechanism	6
4.3	Parameters	8
4.3.1	Interests	8
4.3.2	Conditions	9
4.3.3	Actions	9
4.4	Event Manager	10
4.5	An Example	10
4.6	A Variation	12
4.7	Textual Language	12
4.8	Application of Framework	15
5	Implementation	15
6	Future Work	16
7	Conclusions	17
	Acknowledgements	19
	References	20

A	Programmer's Guide to Interaction Objects in BAGS	22
A.1	Programming Interface	22
A.2	Initialization	22
A.3	Object Creation	23
A.4	Parameters to Finite State Machine	23
A.4.1	Interests	24
A.4.2	Conditions	25
A.4.3	Actions	25
A.5	Condition Routines	26
A.6	Action Routines	27
A.7	Arglists	28
A.8	Data Lists	29
A.9	Running Interaction Objects	30
A.10	Event Handling	31
A.10.1	Background	31
A.10.2	Event Manager Routines	32
A.10.3	Inside the Event Manager	32
A.11	ISLE Parser	33
A.12	Data Structures	33
A.12.1	IOBJhandle	34
A.12.2	IOBJobject	35
A.12.3	IOBJinterest	37
A.12.4	IOBJcondition	37
A.12.5	IOBJaction	38
A.12.6	IOBJarglist	38
A.12.7	VALUitem	39

List of Figures

1	Finite State Machine used by Interaction Objects	7
2	Structure of Interaction Objects	8
3	ISLE Description of Translation Technique	13
4	ISLE Description of Variation of Translation Technique	14

1 Introduction

Recent work in computer graphics has included the development of interactive three-dimensional modeling and animation systems, such as the Brown Animation Generation System (BAGS) [7, 14, 16]. The goal of these systems is to enable a non-technical user to apply interactive and gestural methods in producing and animating a scene containing 3D objects based on reality or imagination. Towards this end, various interaction techniques have been developed to provide gestural control of 3D objects in a scene, such as overlapping sliders and the virtual sphere [3] for 3D object rotation. Also, other interaction techniques developed for a 2D environment have been extended to the 3D environment, such as snap-dragging [2].

While these interaction techniques enhance the expressive power of the end user, they constitute an *ad hoc* collection with no unifying structure or communication style. Each technique has been developed for a specific task and does not exploit common input and output functionality shared by various interaction techniques. Furthermore, to carry out its effects, each technique often uses its own mechanism for communicating changes to the application or graphics system, such as constraints or methods. These conditions have an inhibiting effect on the development and usability of current and new interaction techniques. Developing new interaction techniques requires significant effort because, in the absence of adequate programming abstractions, they usually have to be coded from scratch.

Some of this extra work could be avoided by using more sophisticated programming abstractions. High-level toolkits for 2D environments, such as the Macintosh Toolbox [1] and the OSF/Motif widget set [13], do offer preprogrammed interaction techniques, but these toolkits do not provide easy decomposition and customization of interactive behaviors and usually demand a high price in complexity and size. Furthermore, a literature review did not reveal the existence of interaction toolkits geared for use in 3D environments. Thus, creating new 3D interaction techniques requires low-level programming and a laborious cycle of compiling and testing code. This process is not conducive to easy and rapid design and development of 3D interaction techniques.

This paper describes a research effort to design a programming abstraction called *interaction objects* that addresses the problems cited above. To help shape the design, a set of four research objectives is first described. Some related work is reviewed, and then the design of interaction objects is presented in detail, along with examples.

New issues arising from the development of the design are explained in a section on future work, and an analysis of how the design of interaction objects satisfies the four research objectives is presented in the conclusion. For the programmer, an initial implementation is described in rich detail in the appendix.

Several assumptions about the reader are made for this paper. You should be familiar with the concepts of interaction tasks and interaction techniques; an overview can be found in Chapter 8 of *Computer Graphics: Principles and Practice* [4]. It is also helpful to have an understanding of how some interaction techniques have been applied to 3D environments (see TDUI [5] and Melissa Gold's master's thesis [6] for some examples). You should also understand the concepts of programming abstractions and graphics packages and toolkits. Since the appendix describes an implementation based on a particular software environment (the Brown Animation and Generation System), familiarity with that environment is helpful. However, it is not necessary to read the appendix to understand the research project's design.

2 Research Objectives

This research project seeks to fill the gap between low-level programming of interaction techniques and high-level toolkits by developing a structural framework for the specification of interaction objects to be used in a 3D environment. Interaction objects can be thought of as software "agents" that recognize and respond to characteristic interactive behaviors, thus providing a means for the user to manipulate 3D graphical objects. Towards this end, this research project's objectives are separability, generality, flexibility, and usability, all of which support a single goal of interaction modeling.

2.1 Separability

User interface software is notorious for its large size, complexity, and difficulty in implementation and maintenance. To better manage user interface software, separation of interface and application functionality has become an accepted design and software engineering goal. However, there is little agreement on where and how the line of demarcation should be established. The lack of agreement centers on how to achieve communication between the interface and application components. A narrow communication channel leads to cleaner separation but does not accommodate the high bandwidth requirements of semantic feedback often found in direct manipulation interfaces. Using a wider channel where more semantic power is available reduces the independence of the interface component and increases its complexity. Despite the lack of agreement, some interactions can be handled without direct application involvement. In general, scene objects should not be concerned with user interaction, either with how they are manipulated by the user or how they are presented to the user. Rather, scene objects should focus on the task for which they are designed, and interaction objects should focus on recognizing and responding to interactive behaviors. Towards achieving separability, the software must be organized effectively.

2.2 Generality

Despite the diversity of user interface software, abstractions can be developed for use in a variety of situations. Organizing the user interface for generality encourages software reuse and revision with minimal impact on application code. The availability of powerful abstractions capturing commonly-used functionality enables flexibility in developing new interaction techniques.

2.3 Flexibility

Given well-designed building blocks derived by factoring common user interface functionality, existing interaction techniques can be easily extended and new interaction techniques can be quickly developed. Different interaction styles can be created for users with varying skill levels; this is imperative in an environment where users range from the novice to the highly experienced.

2.4 Usability

A strong emphasis of this research project is making interaction objects accessible to and modifiable by the user; the user should not have to resort to low-level programming to create or modify interaction techniques. Interaction objects should be described in a high-level and compact textual language. A textual language allows easy access and modifications and provides a means for capturing the specification of interaction objects. To further enhance usability, an interactive editor can be developed to provide a graphical front-end that stores specifications in the textual language. Here, the textual language then becomes a back-end. Combined with flexibility, the emphasis on usability leads to a system that allows prototyping of interaction techniques.

2.5 Interaction Modeling

Given a general, flexible, and usable system for specification of interaction objects, new interaction techniques can be easily constructed as specific needs arise. For example, the author of interactive illustrations can draw upon a growing collection of interaction objects, and construct new interaction objects to fit particular interactive illustrations. The specification can also enable the encapsulation of interaction objects into a scene description sent to a distant viewer. That author's influence is not limited to the creation, modeling, and animation of a scene; it extends to exactly how and with what techniques a user can interact with a scene. This is *interaction modeling*, by analogy to animation modeling.

3 Related Work

While there have been many research efforts in the area of user interfaces, two particular projects have relevance to the research objectives of this project: the Garnet

project at Carnegie Mellon University and the Brown Animation and Generation System at Brown University. Each of these two projects are described in the next two sections.

3.1 Garnet

Garnet [11] is a research project at Carnegie Mellon University (CMU) that develops tools for the design and implementation of highly interactive, graphical user interfaces for 2D environments. The facilities provided by Garnet include an object-oriented programming system, a graphical object system, a constraint system, and encapsulations of interactive behaviors. Based on these facilities are high-level tools such as a graphical interface builder and a dialog box creation system. The encapsulations of interactive behaviors has particular relevance to the problems addressed in this research project.

Explained in detail in [10], various interactive behaviors can be encapsulated into a set of objects called *interactors*. Each type of interactor captures a common mouse-based interaction technique and presents a device-independent and graphics-independent abstraction to the interface programmer. The set of interactor types is quite small because of the low number of distinct behaviors in user interfaces (see Szekely's PhD thesis [15] for a classification of input gestures). This small set has been shown to cover the range of mouse-based interaction techniques provided by the Macintosh Toolbox.

All interactor types are parameterized so that the interactive gestures' initial and terminating conditions and action procedures can be specified. The parameters drive a simple finite state machine run by all interactor types and specify the events that trigger transitions and their associated action procedures. The control flow of the state machine is intrinsic to interactors and is not programmed by the user, thus avoiding the proliferation of transitions and states often found in transition network UIMSs.

Interactors contribute to separating interactive behaviors from the details of the application and graphics system by isolating the input-handling code and using application callback functions to communicate interactive changes to the dependent application or graphics objects. Interactors extend a higher level of programming abstraction than event streams and serve as a foundation for prepackaged interaction techniques found in high-level toolkits.

Since the concept of encapsulating interactive behaviors is consistent with the objectives of this research project, it is used as a basis for the design of interaction objects. However, Garnet is limited to mouse-based interaction techniques, and does not support other input devices or 3D objects. The interactors are contained within a much larger system programmed in Lisp, making it difficult to isolate and use in another programming system.

3.2 BAGS

In the development of the Brown Animation and Generation System (BAGS) over the last few years, there has been a strong emphasis on the research and develop-

ment of 3D interaction techniques for use by people without technical experience. Several programming efforts reflect this research focus. An early testbed for direct manipulation of 3D objects using 2D input devices, called TDUI (Three-Dimensional User Interface) [5], included an independently-derived version of Nielson and Olsen's triad mouse technique [12], and an improved version of Chen, Mountford, and Sellen's virtual sphere technique [3]. The functionality of TDUI was then incorporated into an interactive modeler-animater called MOO, where other 3D interaction techniques, such as snap-dragging and the use of a Polhemus 3Space Isotrack input device were added [6].

Despite the new power of these interaction techniques, TDUI and MOO suffered from a lack of separability and flexibility. The event loop was written as part of the application code and took advantage of application data structures and flags to determine various responses to different types of events. The names of application functions were also hard-wired into the event loop, creating a non-generic program and a software maintenance burden. Thus, there was little separation of interface and application functionality. Flexibility was limited because the event loop contained assumptions about input devices and the kinds of actions invoked by different event types. For example, TDUI always performed pick-correlation in response to each mouse button-down event, thus precluding other gestures, such as rubber-banding over a region of the screen. To enable such a feature, the application code would have had to be modified and recompiled, an tedious and time-consuming process with the large size of the MOO code and associated libraries.

After the limitations of TDUI and MOO were recognized, two new BAGS packages, called SWIG (Standard WIdGets) and COW (Control Of Widgets) were written to provide more flexibility and generality. These two packages had a more object-oriented flavor than TDUI and MOO, and some portions of the code became more data-driven. For example, there was a "click object" that handled mouse clicks. Unfortunately, certain behaviors were still hard-coded; the click object always performed a ray intersection test as part of pick-correlation. This design again precluded other uses of the click object such as rubber-banding and advancing through an animation frame-by-frame. Since there was a tight coupling between events and behaviors, there could not be multiple responses to the same event, thus limiting flexibility. Also, certain assumptions were made by using global variables; the data-driven approach was not fully implemented throughout the code.

After reviewing the contributions and drawbacks of the Garnet and BAGS packages, I identified several design elements that would bring us closer to our goal, and developed a design for interaction objects. This design is described in the next section.

4 Design

Interaction objects are the basis of a structural framework for specifying interaction techniques. The term "interaction object" does not suggest the use of the classical superclass-subclass-instance hierarchy found in object-oriented programming lan-

guages. Rather, interaction objects are abstractions that encapsulate code and data used in recognizing and responding to interactive gestures. Each interaction object supports an interaction technique, and several interaction objects can respond to a single interactive gesture in different ways.

The design of interaction objects consists of four components:

- the *representation* of an interaction object
- a *communication mechanism* through which application data structures can be inquired or modified by interaction objects
- an *event manager* that dispatches events to interaction objects
- a *textual language* for capturing behavior of interaction objects

Each component is described in the following sections, with examples showing how the pieces fit together.

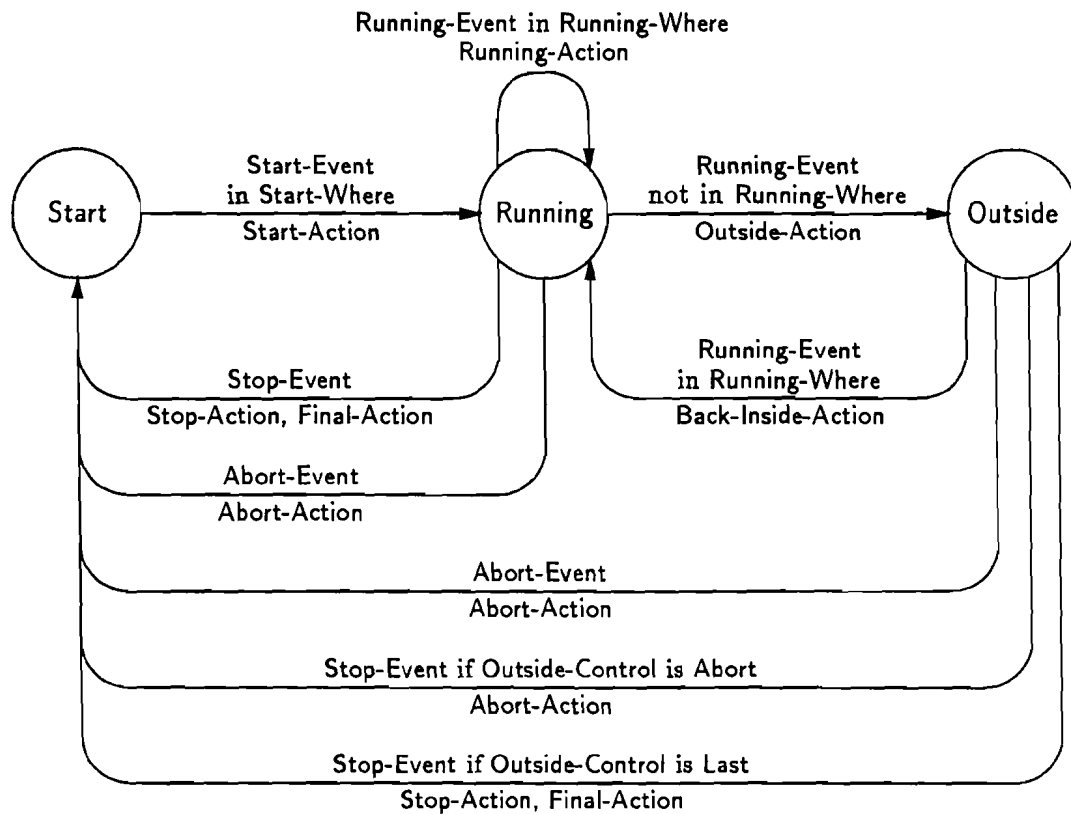
4.1 Representation

An interaction object is represented by a data structure and a small, fixed-size finite state machine. The finite state machine, shown in Figure 1, handles the starting, running, stopping, and aborting of user gestures. While the finite state machine establishes the control flow of an interaction object, various parameters stored in the data structure determine the actions taken in response to different types of events. Thus, when an interaction object is created, only the parameters have to be specified, rather than the harder-to-specify control flow. The representation is based on that of Garnet interactors, and the current implementation uses a slightly-modified version of the finite state machine used by Garnet interactors, which has been shown to cover a wide range of mouse-based behaviors.[†] A full description of the parameters and their roles is given in Section 4.3.

4.2 Communication Mechanism

Most often, the routines invoked by interaction objects during state machine traversal need to inquire or modify the application's data structures (or model). For example, in a direct-manipulation click gesture, the application's model is traversed to determine which application object is indicated by the mouse click. This communication between interaction objects and application data structures is achieved by using *data lists*. The application creates data lists, assigns names to these lists, and stores these names as the parameters to the interaction objects. These data lists can then be read from or written to during interaction. The contents of data lists are application-defined and passed to application-defined routines during state machine traversal. Each item in a data list can be an arbitrary pointer to an application data structure. In effect, the

[†]Unlike the Garnet version, the current version introduces the Running-Event interest, so that it can be distinct from the Start-Event interest.



Based on B. A. Myers, "Encapsulating Interactive Behaviors," in *Proceedings of SIGCHI '89*, published as *Human Factors in Computing Systems*, May 1989, pp. 319-324.

Figure 1: Finite State Machine used by Interaction Objects

interaction objects treat data lists as typeless variables, and it is up to the application routines to use the data lists properly when passed to them.

Assigning names to data lists and using these names (and not the lists themselves) in interaction objects is a form of late binding. Since a particular data list may change in size and content over time, even to the point of becoming empty, using the name of the list will always ensure that the current contents of the list are passed during invocation of condition or action routines.

This communication strategy supports the goals of separability, generality, and flexibility. Since an interaction object only passes lists whose names are given by the application, the interaction object does not assume any application functionality. As the contents of the data lists are application-defined, generality and flexibility are achieved.

Object Name	
Interests	Start-Event
	Running-Event
	Stop-Event
	Abort-Event
Conditions	Start-Where
	Running-Where
Actions	Start-Action
	Running-Action
	Stop-Action
	Abort-Action
	Back-Inside-Action
	Final-Action
Current State of Finite State Machine	
Miscellaneous Housekeeping Information	

Figure 2: Structure of Interaction Objects

4.3 Parameters

Most of the parameters stored in the interaction object can be classified into three groups: *interests*, *conditions*, and *actions*. This classification is reflected in the structure of interaction objects, as shown in Figure 2. These interests, conditions, and actions are also indicated on the arcs of the finite state machine shown in Figure 1. An interest represents an interaction object's interest in a particular event type. When certain types of interests are satisfied, conditions can be tested to qualify the event before actions are invoked. Actions carry out the effects of the interaction such as providing graphical feedback, moving an object on the screen, or modifying a data structure. Each of these three groups are explained in further detail below.

4.3.1 Interests

To allow an interaction object to respond to an event, the interaction object must express to an event manager an interest in a particular event type. There are four types of interests used by the interaction object's finite state machine: **Start-Event**, **Running-Event**, **Stop-Event**, and **Abort-Event**. When the interest is satisfied, the finite state machine of the interaction object is advanced to the next state. In certain cases, as explained in Section 4.3.2, conditions may be tested before advancing the finite state machine.

An interest can be one of several predefined event types or an user-defined event type. Predefined event types include valuator changes, button presses, and button releases originating from hardware devices such as mice, tablets, and dials. The predefined event types can be augmented by user-defined event types having arbitrary labels, thus providing extensibility and a means of creating a higher-level event type based on a series of lower-level event types.

4.3.2 Conditions

Conditions act as event qualifiers, allowing events to be interpreted in an application-defined context. This context is determined by an application-defined condition routine. When a condition is created for an interaction object, a condition routine, along with the names of an input data list and an output data list, must be specified. The input data list is passed to the condition routine when it is called, and the condition routine returns a new list containing the elements of the input data list meeting a certain condition. In a sense, conditions behave as filters by producing a new list that is a subset of the original list.

There are two types of conditions: Start-Where and Running-Where. They correspond to the interest types Start-Event and Running-Event, respectively. For instance, when a Start-Event interest is satisfied, the Start-Where condition routine (if specified) is invoked with the specified input data list. If the output list returned by the Start-Where condition has at least one item, the state machine of the interaction object can be advanced to the next state, the Running state. If no function is specified for the Start-Where or Running-Where conditions, the finite state machine is always advanced to the next state when the Start-Event or Running-Event interests are satisfied. A series of conditions can be created by using the returned list of one condition as input to another condition.

So that events can be interpreted in an application-specific context, a condition routine can make inquiries about the event that led to its invocation. Information about that event is made available by passing it to the condition routine. For example, a mouse-down event's value might be used to determine whether the mouse click occurred within the boundaries of an application object shown on the screen. The event delta (or change) might also be used to determine a relative direction.

4.3.3 Actions

Actions are also supported by application-defined routines that perform the intermediate or final effects of an interactive gesture. For generality, action routines can accept an arbitrary number of input data lists, and can have a side effect of producing a new output data list. Thus, when an action is created for an interaction object, the names of all input lists and an optional output list, along with the action routine, must be specified.

Depending on the current state of the finite state machine, different types of action routines are invoked when each arc of the finite state machine is traversed. This

arrangement enables different responses to different types of events in the same interactive gesture. For example, a mouse-down event might highlight a screen object, and a mouse-up event in the same gesture might unhighlight the same screen object.

As with condition routines, action routines can inquire event values or deltas during the process of establishing an application-specific context.

4.4 Event Manager

The execution of interaction objects is controlled by an event manager. This manager keeps track of registered input devices (such as mice, tablets, and dials) and registered interests expressed by various interaction objects. To simplify the task of the event manager, a distinction is made between enabled and disabled interests. Whether a given interest is enabled depends on the current state of the associated interaction object's finite state machine. Also, any interaction object, as a whole, can be activated or deactivated. By supporting only enabled interests and activated objects, the scope of attention maintained by the event manager is kept to a reasonable size.

When an event occurs, the event manager determines whether there are any enabled interests matching the event's type. If any matching interests are found, the interested interaction objects are processed by advancing the state of each interaction object's finite state machine. As mentioned earlier, conditions may be tested before advancing the finite state machine to the Running state.

By using a central event registry for an arbitrary number of interests, this design does not suffer from the drawbacks of a hard-coded event loop. Instead, the event loop becomes truly data-driven. The registry also allows multiple interests for a given event, enabling multiple responses in the same interactive gesture.

Another benefit of the registry approach is the ability to support user-defined event types. The event manager provides routines for creating and "posting" user-defined event types. A form of sequencing control can then be achieved. For example, suppose that a duct object can be only be created after a spline object and a cross section have been created and selected. When these preconditions occur, an action routine can post an user-defined event in which another interaction object has an interest. When that interest is satisfied, the interaction object can then allow the creation of the duct object.

4.5 An Example

To show how the representation of interaction objects, the communication mechanism, and event manager work together, an example of a 3D object translation technique is presented here. This gesture involves moving a pointer over an object on the screen, pressing and holding a mouse button, dragging the mouse to translate the object, and releasing the mouse button when the object is in the new position. As soon as the mouse button is pressed, a ray is computed from the screen position along the camera's *at*-vector. This ray is tested for intersection with any 3D objects on the screen. If the

ray intersects more than one 3D object, only the frontmost object should be translated. An interaction object to support this technique can be programmed as follows:

1. A data list is created and initialized to contain all the 3D objects in the scene. This list is named "Scene Objects."
2. A Start-Event interest in a mouse button-down event is created and installed in the interaction object.
3. A series of two Start-Where conditions is created for the interaction object. For the first condition, the name of the input data list is "Scene Objects," the application-defined routine is RayIntersect, and the name of the output data list is "Intersected Objects." RayIntersect tests for ray intersection with the 3D objects in the input data list and generates an output data list containing the intersected objects. The second condition passes "Intersected Objects" as the input data list to the application-defined routine FrontMostObject which determines the frontmost object in the input data list and places that object in an output data list named "Translate Objects." (Although this interaction object allows only one 3D object to be translated at a time, the list name is plural to emphasize that action routines should not make assumptions about the size of the list.)
4. A Start-Action action is installed with "Translate Objects" as the name of the input data list and the application-defined action routine HighlightObject, which highlights the objects in the input data list.
5. A Running-Event interest in a mouse-movement event is created and installed. The Running-Where condition is left unspecified because the picked object may be translated anywhere on the screen.
6. For the Running-Action action, an application-defined function called MoveObject is installed, along with "Translate Objects" as the name of the input data list. MoveObject translates the object on the screen according to the change in the mouse position.
7. A Stop-Event interest in a mouse button-release event is created and installed. When this event occurs, the translation gesture is considered to be complete.
8. A Stop-Action action is set with an application-defined routine called UnhighlightObject and "Translate Objects" as the input data list. This routine reverses the action of HighlightObject by unhighlighting the object in the input data list. It is not necessary to translate the object at the end of the gesture, because the Running-Action routine translates the object during the gesture.

When the application is running, the event manager tries to match all events with any enabled interests. When a mouse button-down event occurs, the Start-Event interest of the above interaction object is satisfied, and the RayIntersect condition routine

is invoked. If any 3D objects were intersected, the `FrontMostObject` condition is invoked with this list of ray-intersected objects to create the "Translate Objects" output data list. If a frontmost object is found, it is highlighted by the `HighlightObject` action routine.

At this point, the current state is the `Running` state, the `Start-Event` interest is disabled, and the `Running-Event` and `Stop-Event` interests are enabled. Movements of the mouse cause the `MoveObject` routine to be called as a `Running-Action` action. When a mouse button-release event occurs, the `Stop-Event` interest is satisfied, the translated object is unhighlighted by the `Stop-Action` action routine, and the interaction object returns to the `Start` state.

4.6 A Variation

The translation technique above can be modified to have a different behavior. Instead of translating the 3D object while dragging the mouse, an outline of the 3D object can be drawn at the current mouse position. When the mouse button is released, the outline is removed and the 3D object is translated to the final position of the outline. To effect this variation, the following changes are made to the translation technique described above:

1. Instead of using `HighlightObject` as the `Start-Action` action routine, another application-defined routine called `MakeOutline` is invoked to create a new screen object that is an outline of the frontmost object. This outline object is stored in an output data list called "Outline Objects."
2. The `Running-Action` action invokes an application-defined routine named `MoveOutline` to move the outline stored in the "Outline Objects" input data list to the current mouse position. The 3D object from which the outline is generated is left intact in its original position.
3. The `Stop-Action` action invokes an application-defined routine named `RemoveOutline` to remove the outline from the screen, using "Outline Objects" as the input data list.
4. The `Final-Action` action is installed with the `MoveObject` routine to move the 3D object in the "Translate Objects" input data list to the final position of the outline.

The simplicity and limited scope of the modification shows the flexibility of interaction objects.

4.7 Textual Language

Each of the above steps for creating an interaction object correspond in a straightforward fashion to programmatic routines in a software library. (See Section 5 and the appendix for details.) However, to support the research objective of usability, a

textual language is provided as a high-level and compact way to specify the behavior of interaction objects. This language is called ISLE, for Interaction Specification Language. This approach allows users to access, modify, and capture the behavior of interaction objects using text files, without resorting to low-level programming. Gestural approaches such as visual programming and programming by example would also contribute to usability, but they represent a separate research problem and their application to interaction objects is left as future work.

Rather than present a formal language description of ISLE, an ISLE description of the behavior of the translation technique example in Section 4.5 is shown in Figure 3.

```

iobject translate {
  start {
    event:      mouse down ;
    condition:  "Screen Objects" -> RayIntersect -> "Intersected Objects",
               "Intersected Objects" -> FrontMost -> "Translate Objects" ;
    action:     HighlightObject("Translate Objects") ;
  }
  running {
    event:      mouse move ;
    action:     MoveObject("Translate Objects") ;
  }
  stop {
    event:      mouse up ;
    action:     UnhighlightObject("Translate Objects");
  }
}

```

Figure 3: ISLE Description of Translation Technique

Recall that, as explained in Section 4.1, interaction objects are created by specifying the parameters to the finite state machine. Thus, there is a close relationship between ISLE descriptions and the structure of interaction objects.

The text files containing ISLE descriptions are scanned by a parser during the initialization step of the application by making the appropriate function calls to the IOBJ package. The parser creates the interaction objects and establishes the parameters of these structures based on the descriptions found in the text file.

Since the parser must be able to recognize the names of application-dependent routines in ISLE description files, the application must register these names, as well as the names of initialized data lists, with the parser. In the above example, RayIntersect and FrontMost are registered as condition routines, HighlightObject, MoveObject, and UnhighlightObject are registered as action routines, and "Screen Objects" is registered as an initialized data list. Thus, when the parser scans these tokens in the text files, it will recognize the registered functions and list names, and map these names to the actual routines and data list structures. Names of lists not

appearing in the parser's registry are considered to be "free" lists and are assumed to be names of output data lists which may be subsequently used as input data lists. "Intersected Objects" and "Translate Objects" are examples of this case.

The ISLE parser is not limited to reading description files during the initialization phase; it can also be used to change the parameters of interaction objects while the application is running. For this reason, ISLE descriptions require that each interaction object be identified by name, such as `translate` in the example above. Thus, commands such as the following can be issued:

```
set start event of translate to mouse 1 down ;
set stop event of translate to mouse 1 up ;
```

These commands force the `translate` interaction object to respond only when the left mouse button is pressed, rather than any mouse button as directed by the previous example. This functionality contributes to a prototyping environment for more rapid development of interaction techniques.

To further enhance usability, the language syntax of ISLE is designed to be flexible. The ISLE description shown above is only one way of describing this particular behavior. ISLE is not sensitive to extra spacing between keywords and list names, and some of the keywords can be abbreviated to reduce the amount of typing. The descriptions for the start, running, and stop groups can also be expressed in a more context-free fashion, as shown in the next example. Also, the indentation and spacing format used above is purely a stylistic convention to enhance readability. To illustrate the flexibility of ISLE, an ISLE description capturing the variation of the translation technique given in Section 4.6 is shown in Figure 4.

```
iobject translate-variation {
  start event:  mouse down ;
  start cond:   "Screen Objects" -> RayIntersect ->
                                     FrontMost    -> "Translate Objects" ;
  start action: MakeOutline("Translate Objects") -> "Outline Objects" ;

  run event:    mouse move ;
  run action:   MoveOutline("Outline Objects") ;

  stop event:   mouse up ;
  stop action:  RemoveOutline("Outline Objects") ;

  final action: MoveObject("Translate Objects") ;
}
```

Figure 4: ISLE Description of Variation of Translation Technique

Note how the "Intersected Objects" data list is not explicitly referenced in the above example. Since the output data list of the `RayIntersect` condition routine is

used immediately as the input data list of `FrontMost` condition routine and is not used elsewhere, it is not necessary to use an explicit name. This technique is similar to the UNIX concept of pipes.

4.8 Application of Framework

Up to this point, interaction objects have been discussed in the singular sense. However, most interactive applications do not revolve around a single interaction technique. Instead, different interaction techniques are used at different points during an application session. Since each interaction object supports an interaction technique, a typical application would create a number of interaction objects. As the encapsulated data structure used by interaction objects allows easy addition and removal of interaction techniques with minimal disruption to other interaction objects, a collection of interaction objects can then be pieced together one at a time, as part of a “plug’n’play” development strategy. The result is a loosely-coupled network, where each interaction object can enable or disable other interaction objects in response to certain user gestures and semantic actions. Furthermore, with the event manager’s support of user-defined events, an interaction object can initiate another interaction object’s behavior by posting arbitrary events.

5 Implementation

As a proof of concept, an initial implementation of interaction objects was developed in the BAGS software environment. BAGS is the software testbed for 3D graphics used by the Brown University Computer Graphics Group; among other things, it provided many utility routines for controlling input devices, memory management, hashing, and string table management used by the implementation of interaction objects. BAGS was also used to show that interaction objects can succeed in a 3D graphics environment. The implementation was divided into two BAGS packages (or libraries). The first package, called IOBJ, contained all the application-independent code such as routines for initialization, event management, creating interaction objects, and specifying the interests, conditions, and actions of these objects. This package represented the application programming interface (API) to interaction objects. The second package, called BIOBJ, served as a test application of the IOBJ package. This approach to organizing the code was intended to illustrate that the goal of separability could be achieved.

Although the initial implementation was not comprehensive or fully-polished, it served to test the viability of the design and to identify areas for future work. Validation tests indicated that interaction techniques similar to the translation technique example (given in Section 4.5) can be supported by the design. One of the validation tests involved the use of BIOBJ as a test application built into a BAGS modeler-animater program; a scene of 3D objects was displayed, and an interaction object supported scene object selection by ray intersection. Further work is needed to determine if more complex interaction techniques can be implemented using the IOBJ package. Based on my experience with the initial implementation, I am confident that the design is capable of supporting a wide variety of interaction techniques in different

application contexts. Details about the implementation, including data structures and the internal behavior of API routines, can be found in the appendix, "Programmer's Guide to Interaction Objects in BAGS."

6 Future Work

Since interaction techniques can be complex, there is abundant room for further work; several such areas are indicated below.

Currently, any given interest is limited to one particular event type. Adding support for boolean operations on events would increase the flexibility of interaction objects. For example, a *Start-Event* interest can be in either a mouse button-down event or a tablet-press event. This functionality would allow a single interaction object to respond to a variety of events in the same interest. Supporting a boolean OR operation on events, such as $a \vee b \vee c$ would be straightforward, but the boolean AND operation would require more careful thought, due to the issue of timing between each operand of the AND operation. This issue begs the question of whether the two operand events must occur at the same instant in time, or whether they can occur at different times, with no respect to time ordering. In the latter case, the event manager would have to be extended so that a history of events is maintained over a time interval and used to determine the satisfaction of boolean AND operations. Maintaining a history of events introduces its own concerns such as managing the computational resources used by the history to a reasonable size. A useful heuristic may be to record only those events for which there is an enabled interest. Time-ordered AND operations such as $a_{t_i} \wedge b_{t_{i+1}}$ where $t_i < t_{i+1}$ (t represents time) would be useful, but more research is needed before implementation. This functionality would allow a certain time-ordered sequencing of events to be recognized as an interactive gesture.

The ISLE parser can be extended by adding a type system. The names of application routines and data lists can be checked against a type system to minimize type errors such as inadvertently using a condition routine as an action routine, or a routine as an input data list. The type system would be supported by type declarations at the top of an ISLE description file. Built-in functions can also be added to ISLE for posting events or activating and deactivating interaction objects by name.

Since the implementation only supported the textual language, future work can include the development of an interactive, graphical editor as a front-end to ISLE. Recent research in programming-by-example systems [8, 9] can be brought to bear on this problem. Also, reasonable defaults for different properties of interaction objects can be provided to minimize the amount of set-up work required for simple interaction techniques.

Since the initial implementation was not comprehensive, it should be extended and refined to cover areas not tested by the initial implementation, such as series of conditions, the use of multiple routines in a single action, and recovery from abort events. A variety of applications need to be tested to see if the design and the finite state machine is general enough to support the needs of diverse interaction techniques.

7 Conclusions

A design for interaction objects and their specification has just been presented. The design starts with a generalized representation and finite state machine based on Garnet, resolves the drawbacks of existing BAGS packages by developing a completely data-driven approach using data lists, uses an event manager to isolate the event-handling code, and incorporates a textual specification language to capture the behavior of interaction objects. Let us review how this design satisfies the four research objects of separability, generality, flexibility, and usability.

The concept of encapsulations of interactive behaviors borrowed from the Garnet project contributes significantly to separability. The use of an event manager that isolates the event-handling code from the application further reinforces this separability. After creating interaction objects, the application calls only one routine to enter an event-processing loop. This is in contrast to how earlier work, such as TDUI and MOO, incorporated the event loop into the application. Separability also comes from the fact that the application-defined condition and action routines are separated from the code that directly supports interaction objects. Thus, changes to the application routines have no impact on the interaction object code.

Generality comes from using data lists and a consistently-applied data-driven methodology. Since condition and action routines cannot assume the size of their input data lists, the programmer is encouraged to write these routines as generally as possible. The fact that an interaction object can support multiple conditions and actions fosters the development of general "primitive" routines (such as ray intersection, and highlighting) that can be combined with other general routines to effect a specific behavior. Since the design was derived from a 2D environment, and then tested in a 3D environment, the design is general enough to support both 2D and 3D interaction techniques.

The flexibility of interaction objects again comes from using a data-driven approach. Since the behavior of interaction objects is stored as data, it is easy to change the behavior by changing the parameters stored in the interaction object, even at runtime. The emphasis on writing generic condition and action routines that could be easily added to interaction objects contributes to their flexibility. Interaction objects can also be enabled or disabled during runtime as an action or in response to an event, leading to easily-configurable interaction techniques.

Usability is enhanced by the use of a textual language, ISLE, as a means of capturing and modifying the behavior of interaction objects, without using low-level programming. The user can easily modify ISLE descriptions prior to running the application or by issuing ISLE commands at runtime to alter the behavior of an existing interaction object. ISLE also leads to a prototyping environment where an user can create and modify a variety of interaction objects for experimentation.

The design of interaction objects contributes to the research and development of interaction techniques by providing a programming abstraction that is more sophisticated than low-level event handling and does not suffer from the drawbacks of

prepackaged interaction techniques. Interaction objects serve as a mechanism with as few assumptions as possible, thus keeping them general and flexible. Interaction objects are designed to be used in many different contexts, each context implemented by a collection of application-defined condition and action routines kept separate from the interaction object code. The combination of the data-driven design and the specification language leads to highly-dynamic interaction and brings us closer to the goal of interaction modeling.

Acknowledgements

This thesis cannot be complete without sincere thanks and appreciation to my colleagues, friends, and family.

Andries van Dam, my advisor, gave me the support and freedom for pursuing this research project. His extraordinary teaching, leadership, and boundless energy were always inspiring. His co-directorship of the Brown University Computer Graphics Group with John “Spike” Hughes brought together many talented people who contributed their ideas and comments during the project’s development. In particular, I wish to thank Brook Conner, Henry Kaufman, Bob Zeleznick, Brian Knep, Scott Snibbe, Tinsley Galyean, and Philip Hubbard for useful discussions. Brook also took the time to develop an understanding of the implementation and to code the parser. I also thank George Reilly for answering my numerous questions about \LaTeX and \TeX (the document processing system used to produce this paper) and Mary Andrade for helping me with many administrative tasks.

Deserving of special thanks is Christine Dunleavy, who served as my interpreter for classes, graphics group meetings, and special trips. Her dedication and nice balance between professionalism and personal style was a distinctive element of my classroom and meeting experience. I also appreciate the friendship that came out of our working together.

My close friendship with Heather Harker, who I met on the first day of classes, enhanced my entire Brown experience. She helped me maintain perspective by reminding me of the world outside of the Computer Science department and she was a wonderful companion and confidante. I also appreciate the friendship and support of Anne McDonough, Jennifer Nelson, and Camille Beckham; by way of long-distance phone calls, they helped keep up my spirits.

Surviving graduate school would have been impossible without the cornerstone support and love from my family; my heartfelt thanks go to Bob and Jinny (my parents), Adam and Julie, Ami and Jon, and Rachel.

References

- [1] Apple Computer, Inc., *Inside Macintosh*, Addison-Wesley, Reading, MA, 1985.
- [2] E. A. Bier, "Snap-Dragging in Three Dimensions," in *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, published as *Computer Graphics*, Vol. 24, No. 2, March 1990, pp. 193-204.
- [3] M. Chen, S. J. Mountford, and A. Sellen, "A Study in Interactive 3-D Rotation Using 2-D Control Devices," in *Proceedings of SIGGRAPH '88*, published as *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 121-129.
- [4] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed., Addison-Wesley, Reading, MA, 1990.
- [5] T. Gaylean, M. Gold, W. Hsu, H. Kaufman, and M. Stern, "Manipulation of Virtual Three-Dimensional Objects Using Two-Dimensional Input Devices," Brown University Research Report, December 1989.
- [6] M. Y. Gold, "Multi-Dimensional Input Devices and Interaction Techniques for a Modeler-Animator," Master's thesis, Brown University, July 1990.
- [7] P. M. Hubbard, M. M. Wloka, R. C. Zeleznick, D. G. Aliaga, and N. Huang, "UGA: A Unified Graphics Architecture," Technical Report CS-91-30, Brown University, 1991.
- [8] D. L. Maulsby, I. H. Whitten, and K. A. Kittlitz, "MetaMouse: Specifying Graphical Procedures by Example," in *Proceedings of SIGGRAPH '89*, published as *Computer Graphics*, Vol. 23, No. 3, July 1989, pp. 127-136.
- [9] B. Myers, *Creating User Interfaces by Demonstration*, Academic Press, Boston, MA, 1988.
- [10] B. A. Myers, "Encapsulating Interactive Behaviors," in *Proceedings of SIGCHI '89*, published as *Human Factors in Computing Systems*, May 1989, pp. 319-324.
- [11] B. A. Myers *et al.*, "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *Computer*, Vol. 23, No. 11, November 1990, pp. 71-85.
- [12] G. M. Nielson and D. R. Olsen, Jr., "Direct Manipulation for 3D Objects Using 2D Locator Devices," in *Proceedings of 1986 Workshop on Interactive 3D Graphics* (Chapel Hill, NC, October 23-24, 1986) ACM, New York, 1986, pp. 175-182.
- [13] Open Software Foundation, *OSF/Motif Programmer's Guide*, 1989.
- [14] P. S. Strauss, "BAGS: The Brown Animation Generation System," Technical Report CS-88-22, Brown University, 1988.

- [15] P. A. Szekely, *Separating the User Interface from the Functionality of Application Programs*, PhD thesis, CMU-CS-88-101, Carnegie Mellon University, January 1988.
- [16] R. C. Zeleznick, D. B. Conner, M. M. Wloka, D. G. Aliaga, N. T. Huang, P. M. Hubbard, B. Knep, H. Kaufman, J. F. Hughes, and A. van Dam, "An Object-Oriented Framework for the Integration of Interactive Animation Techniques," to appear in *Proceedings of SIGGRAPH '91* (Las Vegas, NV, July 28–August 2, 1991), ACM, New York, 1991.

A Programmer's Guide to Interaction Objects in BAGS

This appendix describes, from a programmer's point of view, the current implementation of interaction objects in the Brown Animation Generation System (BAGS). Readers of this appendix are assumed to be familiar with the high-level design of interaction objects, as described in this paper. Familiarity with BAGS utility packages (such as MEM, VALU, and EVENT), BAGS software standards, and the C language is also assumed.

A.1 Programming Interface

Interaction objects are currently implemented as a BAGS package, surprisingly named IOBJ, which can be found in the BAGS source code hierarchy. This package provides routines for initialization, creating interaction objects, and specifying the interests, conditions, and actions of these objects. Each IOBJ routine is described in the sections below. For each routine, the programming interface is first described. Then, the internal behavior of the routine is described, for the benefit of programmers who want to understand, modify, or extend the code comprising the IOBJ package.

IOBJ supports only the application-independent aspects of interaction objects. The user of IOBJ must create a set of application-dependent routines to implement interaction objects in a particular environment. The reason for this separation is to allow interaction objects to be supported in different applications and graphical environments. In BAGS, these application-dependent routines are in a package named BIOBJ (for BAGS binding to IOBJ). Since BIOBJ is very BAGS-specific, it is not described here.

A.2 Initialization

Before creating interaction objects, the IOBJ package must be initialized by calling `IOBJinit()` as follows:

```
IOBJhandle*
IOBJinit(event_hdl, valu_id, open_flag)
    EVENThandle *event_hdl;
    VALUId      *valu_id;
    int         open_flag;
```

`event_hdl` is the same `EVENThandle*` that is passed to `EVENTevent_loop()` or its derivatives, and cannot be `NULL`. `valu_id` is used for allocating temporary `VALUitem` structures during execution. If `valu_id` is `NULL`, a new `VALUid` is automatically created. `open_flag` determines whether `IOBJinit` should open the following MID devices, if they exist: `MID_DIAL`, `MID_MOUSE`, and `MID_TABLET`. In a situation where input devices have already been opened, `open_flag` should be zero, and `IOBJregister_devices` should be called with a list of currently open devices. `IOBJinit()` returns a handle to the IOBJ environment through an `IOBJhandle*`. This handle is required by many other IOBJ calls. `IOBJinit()` may not be called more than once.

Inside `IOBJinit()`, the memory for the `IOBJhandle` is allocated and initialized. IOBJ's own event manager is initialized and the event registry table is created. The event pipe supporting user-defined events is also created.

When the facilities provided by IOBJ are no longer needed, `IOBJdestroy()` should be called to deallocate the memory structures allocated in the `IOBJinit()` function.

A.3 Object Creation

Interaction objects are created by the `IOBJcreate_object()` function:

```
IOBJObject*
IOBJcreate_object(hdl, name)
    IOBJhandle *hdl;
    char        *name;
```

`hdl` should be the same `IOBJhandle*` returned from `IOBJinit()`. `name` can be any arbitrary string that is used as a name for the interaction object. A pointer to the newly-created interaction object is returned as an `IOBJObject*`.

Inside `IOBJcreate_object()`, memory for the `IOBJObject` data structure is allocated and initialized. All interest types, condition types, and action types in the data structure are initialized to `NULL`. A finite state machine is created, using the FSA package, and stored in the `IOBJObject` data structure.[†]

To destroy an interaction object and deallocated associated memory structures, call `IOBJdestroy_object()` with an `IOBJObject*` as its sole argument.

A.4 Parameters to Finite State Machine

After creating an interaction object, you must specify the various parameters to the finite state machine. Recall that these parameters are classified in three groups: *interests*, *conditions*, and *actions*. This classification is reflected in the structure of interaction objects, as shown in Figure 2 on page 8. These interests, conditions, and actions are also indicated on the arcs of the finite state machine in Figure 1 on page 7.

[†]Note that an entire FSA is allocated for each interaction object. This inefficient use of memory can be corrected by changing the IOBJ code so that a single FSA is shared by all interaction objects; in this case, only a small piece of state is stored in each interaction object.

A.4.1 Interests

To specify an interest for a given interaction object, use `IOBJset_interest()`:

```
void
IOBJset_interest(obj, type, event, event_id)
    IOBJobject      *obj;
    IOBJinterests   type;
    IOBJevents      event;
    IOBJevent_id    event_id;
```

`obj` is an interaction object returned by `IOBJcreate_object()`. The `type` argument specifies the type of interest; it can be one of:

```
IOBJ_START_EVENT      IOBJ_STOP_EVENT
IOBJ_RUNNING_EVENT     IOBJ_ABORT_EVENT
```

The `event` argument can be one of IOBJ's predefined event types or an user-defined event type. The predefined event types are:

IOBJ_MOUSE_ANY_DOWN	IOBJ_TABLET_ANY_DOWN	IOBJ_DIAL_ANY_CHG
IOBJ_MOUSE_ANY_UP	IOBJ_TABLET_ANY_UP	IOBJ_DIAL_1_CHG
IOBJ_MOUSE_1_DOWN	IOBJ_TABLET_1_DOWN	IOBJ_DIAL_2_CHG
IOBJ_MOUSE_1_UP	IOBJ_TABLET_1_UP	IOBJ_DIAL_3_CHG
IOBJ_MOUSE_2_DOWN	IOBJ_TABLET_2_DOWN	IOBJ_DIAL_4_CHG
IOBJ_MOUSE_2_UP	IOBJ_TABLET_2_UP	IOBJ_DIAL_5_CHG
IOBJ_MOUSE_3_DOWN	IOBJ_TABLET_3_DOWN	IOBJ_DIAL_6_CHG
IOBJ_MOUSE_3_UP	IOBJ_TABLET_3_UP	IOBJ_DIAL_7_CHG
IOBJ_MOUSE_CHG	IOBJ_TABLET_CHG	IOBJ_DIAL_8_CHG
		IOBJ_DIAL_9_CHG

These predefined event types correspond to the MID devices known by IOBJ. User-defined event types are created by the `IOBJcreate_event()` function; if you use an user-defined event type, use the `IOBJ_USER_DEFINED` constant for the event argument, and the `IOBJevent_id` returned by `IOBJcreate_event()` for the `event_id` argument. If you use a predefined event type, `event_id` should be `NULL`.

Note that the interest does not become active until the interaction object is activated by `IOBJactivate_object()`.

Inside `IOBJset_interest()`, an `IOBJinterest` data structure is allocated and initialized. The interest type, event type, and event id (if the `IOBJ_USER_DEFINED` event type is used) are stored in the data structure. If an interest for the given interest type already exists in the given interaction object, it is destroyed and replaced with the new interest.

The `next` and `prev` fields in the `IOBJinterest` data structure are used only by IOBJ's event manager, where all interests of the same interest type are kept together in a linked list. These fields should not be misunderstood as allowing a single interaction

object to be interested in one of several event types (or a combination) in the same interest. This functionality, while desirable, is not yet implemented.

A.4.2 Conditions

To specify a condition for a given interaction object, use `IOBJset_condition()`:

```
void
IOBJset_condition(obj, cond_type, func, in, out)
IOBJObject      *obj;
IOBJconditions  cond_type;
IOBJvalfn       func;
char            *in, *out;
```

`obj` specifies the interaction object to which the condition is added. `cond_type` can be one of two condition types: `IOBJ_START_WHERE` and `IOBJ_RUNNING_WHERE`. If an interest of type `IOBJ_START_EVENT` is satisfied, the `IOBJ_START_WHERE` condition is invoked, if it exists. Similarly, the `IOBJ_RUNNING_WHERE` condition corresponds to the `IOBJ_RUNNING_EVENT` interest type. `func` is a pointer to a condition routine that returns a `VALUitem*`. The behavior of condition routines is described in Section A.5. `in` is the name of the data list used as input to the condition routine; the actual data list is retrieved by an implicit call to `IOBJget_list()` just before the condition routine is called. `out` is the name used when the `VALUitem*` returned by the condition routine is stored as a data list. `in` and `out` may be the same, in which case the input list is replaced by the output list after the condition routine is completed.

`IOBJset_condition()` allocates and initializes an `IOBJcondition` data structure. The `in` and `out` strings are copied into this data structure, so that these programmer-supplied strings can be deallocated after the call. The data structure is then installed in the proper row of the condition table in the `IOBJObject` data structure specified by the `obj` argument.

While there is a next field in the `IOBJcondition` data structure, the current implementation does not support a linked list of conditions. This is a very desirable feature, and should be implemented.

A.4.3 Actions

To specify an action for a given interaction object, use `IOBJset_action()`:

```
void
IOBJset_action(obj, action_type, func, arg_list, out)
IOBJObject      *obj;
IOBJactions     action_type;
IOBJvalfn       func;
IOBJarglist     *arg_list;
char            *out;
```

`obj` specifies the interaction object to which the action is added. `action_type` specifies the type of action; the value can be one of:

<code>IOBJ_START_ACTION</code>	<code>IOBJ_OUTSIDE_ACTION</code>
<code>IOBJ_RUNNING_ACTION</code>	<code>IOBJ_BACK_IN_ACTION</code>
<code>IOBJ_STOP_ACTION</code>	<code>IOBJ_FINAL_ACTION</code>
<code>IOBJ_ABORT_ACTION</code>	

`func` is a pointer to an action routine that returns a `VALUitem*`. The behavior of functions used for actions is described in Section A.6. `arg_list` is a pointer to an `IOBJarglist` data structure that contains a list of names of the data lists used as arguments to the action routine specified by `func`. Arglists are described in Section A.7. As with `IOBJset_condition()`, `out` is the name used when the `VALUitem*` returned by the action routine is stored as a data list. If a data list by that name already exists, it is replaced by the new data list.

The internal behavior of `IOBJset_action()` is similar to that of `IOBJset_condition()`: it allocates, initializes an `IOBJaction` data structure, and installs the pointer to the data structure in the proper row of `obj`'s action table. Unlike the in argument in `IOBJset_condition()`, `arg_list` is not copied into the `IOBJaction` data structure. Also, the `out` argument string is *not* copied to the `IOBJaction` data structure; this is a programming oversight and should be corrected so that the behavior is consistent with `IOBJset_condition()`.

Note that the finite state machine shown in Figure 1 contains two Stop-Event arcs from the Outside state to the Start state; these arcs are disambiguated by the value of Outside-Control. To set this parameter of an interaction object, use the `IOBJ_OBJ_OUTSIDE()` macro, which takes as its arguments an `IOBJobject*` and an `IOBJoutsides` constant.

While most action routines are intended to manipulate graphical objects during interaction, the Final-Action routine is intended to be an application callback function.

A.5 Condition Routines

Recall that condition routines are application-defined functions that act as event qualifiers. For example, a condition routine might check if a ray emanating from the screen position of a mouse button press intersects with any 3D objects in a display list. When installed as a condition in an interaction object, the condition routine is invoked when its associated interest is satisfied. For a condition of type `IOBJ_START_WHERE`, the associated interest type is `IOBJ_START_EVENT`, and for a condition of type `IOBJ_RUNNING_WHERE`, the associated interest type is `IOBJ_RUNNING_EVENT`.

The argument list of a condition routine is as follows:


```

VALUitem*
condition_rtn(hdl, iobj, in_list)
    IOBJhandle *hdl;
    IOBJobject *iobj;
    VALUitem   *in_list;

```

Inside a condition routine, there are three sources of data:

- the locator values of the event, which can be accessed from the `hdl` and `iobj` arguments by use of the following public macros:

```

IOBJ_EVENT_VAL      IOBJ_OBJ_EVENT_VAL
IOBJ_EVENT_DELTA    IOBJ_OBJ_EVENT_DELTA
IOBJ_NUM_EVENT_VALS IOBJ_OBJ_NUM_EVENT_VALS

```

- the `in_list` argument, represented as a `VALUitem*`. Just before the condition routine is called, IOBJ's event manager retrieves the data list associated with the `in` name specified as an argument to `IOBJset_condition()`, and passes this list to the condition routine. The contents of this list is application-defined, to allow flexibility. The VALU public macros can be used to access the contents of this list.
- other data lists accessible through the `IOBJget_list()` function; these lists may be either global or local in scope. As with `in_list`, use the VALU public macros to access the contents of these data lists.

Typically, a condition routine iterates through the elements of the data list passed in the `in_list` argument, generates a new list containing elements of the `in_list` that satisfy a certain condition, and returns the new list. Especially if you use the `VALU_LIST_FOREACH_ITEM` macro to iterate over the `in_list`, you should not delete or overwrite any element in the `in_list` argument. To generate the new list, call the `VALUnew_list()` function, using `IOBJ_VALU_ID(hdl)` as the `VALUid` argument. To add a new item to the `VALUitem*` serving as the data list, use the `VALUadd_to_list()` function. When the new list is returned by the condition routine, it may replace the data list used as the `in_list`.

When the condition routine returns, IOBJ's event manager checks the return value to see whether it is `NULL`. If not, the return value is considered to be a data list, and becomes associated with the `out` name specified as an argument to `IOBJset_condition()`. In this case, the condition is considered to be satisfied. If the return value is `NULL`, the condition is not satisfied, and the finite state machine of the interaction object containing the condition is not advanced.

A.6 Action Routines

Action routines are also application-defined functions that perform the intermediate or final effects of an interactive gesture. While they are similar in behavior to condition

routines, action routines have one key difference: the action routine can receive an arbitrary number of data lists, rather than a single data list.

The argument list of an action routine is as follows:

```
VALUitem*
action_rtn(hdl, iobj, in_list)
    IOBJhandle *hdl;
    IOBJObject *iobj;
    VALUitem  *arg_list;
```

When an action is invoked, IOBJ's event manager generates a VALUitem list containing the data list for each name in the IOBJarglist data structure specified as an argument to IOBJset_action(). This list is passed to the action routine as arg_list. Thus, the number of data lists depends on the number of names specified in the IOBJarglist data structure.

Since action routines receive a list of data lists, you need to use the VALU_LIST_ITEM or VALU_LIST_FOREACH_ITEM macros to access the individual data lists.

The action routine can generate and return a new data list, but this is not required. For example, an action routine that adds screen objects to a selection set generates and returns a new data list containing the new selection set. If IOBJ's event manager detects a returned list from an action routine, the data list is associated with the out name specified as an argument to IOBJset_action(). Another action routine that merely translates screen objects may not return a new list; in this case, NULL should be returned. As with condition routines, the new data list returned by the action routine can overwrite a data list used as an input list, if the name is the same.

A.7 Arglists

To allow action routines to be called with a variable number of arguments, the IOBJarglist data structure is used to collect names of data lists used as input. Note that the names of the data lists, rather than the data lists themselves, are stored in the data structure; this enables late binding, so that the contents of the data lists are up to date when used.

To create an arglist, use IOBJcreate_arglist():

```
IOBJarglist*
IOBJcreate_arglist(hdl, num)
    IOBJhandle *hdl;
    int        num;
```

num is the number of data list names to be stored in the IOBJarglist data structure. hdl is needed so that the data structure can be allocated from the private MEM pool stored inside hdl. The table of char pointers used to store the names is allocated using malloc() because the size of the table will vary with the number of data lists stored in the arglist.

`IOBJcreate_arglist()` returns a new, but empty, `IOBJarglist` data structure. After adding the names of data lists to this arglist, the data structure is passed as the `arg_list` argument to `IOBJset_action()`.

To set the names of the data lists in a given arglist, use `IOBJset_arg()`:

```
void
IOBJset_arg(arglist, num, name)
    IOBJarglist    *arglist;
    int            num;
    char           *name;
```

`arglist` is an `IOBJarglist*` returned by `IOBJcreate_arglist()`. The `name` argument is duplicated and stored inside the arglist, so that the argument string can be deallocated after the call.

A.8 Data Lists

Data lists are used as input to condition and action routines, as a way of communicating the application data structures to the interaction objects. Typically, a data list is created by the application, using the VALU package (see Section A.12.7). To allow conditions and actions to refer to a data list, a name must be assigned to the data list; this name is then specified in the calls to `IOBJset_condition()`, `IOBJset_action`, and `IOBJset_arg()`. This can be done by using `IOBJput_list()`:

```
void
IOBJput_list(hdl, iobj, list, name)
    IOBJhandle *hdl;
    IOBJobject *iobj;
    VALUitem   *list;
    char       *name;
```

If `hdl` is non-NULL, the name has *global* scope. If `iobj` is non-NULL, the name is considered to be *local* to the given interaction object. `list` is the data list that will be assigned the string specified by `name`. If a data list already exists for the given name, it is automatically freed and replaced by the new data list. For this reason, make sure that your VALUitems can be correctly freed by the standard routines associated with the VALUid. In special cases, you may have to use your own VALUid when creating new VALUitems.

In general, global scope is sufficient, but local scoping can be useful in certain situations. For example, you can have different interaction objects maintain their own private selection sets. The different selection sets might have the same name, but can be distinguished by local scope.

Inside `IOBJput_list()`, the necessary STRPOOLid is created, if it does not exist, in the interaction object or the `IOBJhandle` (depending on local or global scope, respectively). The string pool is looked up, using STRPOOL routines, to see if the `name`

is already installed. If so, the associated data list is freed. Then, `list` (the new data list) is installed in the string pool.

Data lists can be retrieved by name with the `IOBJget_list()` function:

```
VALUitem*
IOBJget_list(hdl, iobj, name)
    IOBJhandle *hdl;
    IOBJobject *iobj;
    char       *name;
```

As with `IOBJput_list()`, the distinction global and local scope is determined by whether `iobj` is non-NULL. If so, the `name` is looked up in the local scope of the interaction object. Otherwise, the `name` is looked up in the global scope of the `hdl`. The data list, if found, is returned by `IOBJget_list()`. NULL is returned if the name did not have an associated data list.

Inside `IOBJget_list()`, `STRPOOL` routines are used to look up the given `name` in the `STRPOOLid` associated with the interaction object or the `IOBJhandle`, depending on whether or not `iobj` is non-NULL.

The motivation for using data lists and names is to allow late binding. The contents of data lists change over time, especially in an interactive environment. In some cases, a data list may be replaced in its entirety by a new data list; for example, a selection set is reset or changed to a new set of screen objects. There may be many condition and action routines that use the same data list; if names are used and evaluated at the last possible moment before use, the data lists reflect the current contents.

A.9 Running Interaction Objects

When interaction objects are created, they are not immediately activated. To activate interaction objects, call the function `IOBJactivate_object()`, which requires an `IOBJobject*` as an argument. An interaction object can be deactivated by calling `IOBJdeactivate_object()` with an `IOBJobject*` as its argument.

`IOBJactivate_object()` works by enabling the `IOBJ_START_EVENT` interest and disabling all other interests of the given interaction object. The finite state machine is also reset, the active field of the interaction object is set to `TRUE`, and the running field is set to `FALSE` (since the interaction object is not in the running state).

`IOBJdeactivate_object()` simply disables all the interests of the given interactive objects and sets the active and running fields to `FALSE`.

After creating and activating interaction objects, `EVENTevent_loop()` must be called with the same `EVENThandle*` used in the `IOBJinit()` function call. (There are low-level `EVENT` routines you can use in place of `EVENTevent_loop`, but this is beyond the scope of this document; see the `EVENT` man page for details.) The `EVENTevent_loop()` routine ensures that `IOBJ`'s own event manager is notified of events occurring on MID devices registered with `IOBJ`.

A.10 Event Handling

A.10.1 Background

Some background is helpful for understanding how the event manager is designed. To provide a strong event-handling facility, various types of low-level device events need to be handled in a consistent and efficient manner. One approach is to sample input devices at frequent intervals, comparing the sampled value with an application-stored value to determine if the value of an input device has changed. This operation is called *polling* or *sampling*, which must be executed continuously to minimize user interface latency. This method is inefficient because substantial CPU time must be spent in tight polling loops waiting for a input device's value to change, even if the user is not manipulating the input device.

To avoid this inefficiency, an event-driven approach is used. In this approach, the application requests that the graphics package or operating system place input device events on an event queue, in temporal order, as they occur. The application then checks the event queue to see if any events have occurred. If there is an event waiting at the head of the queue, it is removed from the queue and appropriate action is taken depending on the type of event. If there are no events waiting in the queue, the application enters a wait state, remaining there until the next event occurs or until a timer interval elapses. This process is repeated until the application is terminated.

The original device handling package in BAGS, called DIIO,[†] was based on the polling approach, which led to inefficient performance especially in a multitasking environment, such as UNIX. To improve efficiency, a new package called MID[§] was written using the event-driven approach. This package is flexible in the sense that an arbitrary callback function can be registered for each distinct event type or for all event types. As each event is removed from the queue, the callback function associated with the event's type is invoked. This leads to a callback-driven model of execution where the application is informed of device events wholly through callback functions. The event information is passed to the callback functions in a consistent fashion, regardless of the event's type or source. Efficiency is achieved by using the EVENT package, which is described below.

Many graphics packages, such as PHIGS, provide a function that awaits events from supported devices; thus, the application can enter a wait state that does not consume CPU time. The drawback of such functionality is that the application can await events only from devices supported by the graphics package. More generality is needed in the BAGS environment to handle events occurring on devices not directly supported by the graphics package (such as network sockets). The EVENT package was written to provide this generality. This package takes advantage of the fact that all input devices used in BAGS are accessed and controlled through the UNIX file descriptor abstraction. Using this abstraction allows the use of the UNIX `select()` system call to await events on a set of file descriptors. Since file descriptors are used for various

[†]DIIO stands for Device-Independent Input/Output.

[§]MID stands for Manager of Input Devices.

kinds of input devices, streams, sockets, pipes, and files, EVENT provides a consistent mechanism for handling event sources. This mechanism is also efficient because the `select()` system call brings the application into a wait state when there are no events waiting to be processed.

A.10.2 Event Manager Routines

Since the event manager is based on the EVENT package, you must call `EVENT-event_loop()` after creating and initializing interaction objects. If you open your own MID devices and want to register them with the event manager, use the `IOBJregister_devices()` function:

```
void
IOBJregister_devices(hdl, devices, num_devices)
    IOBJhandle *hdl;
    MIDdevice **devices;
    int         num_devices;
```

`devices` is a table of pointers to `MIDdevice` data structures, and `num_devices` is the number of devices in the `devices` table. This function should be used if you do not specify that MID devices are to be opened in the `IOBJinit()` call.

To create a new user-defined event in which an interaction object can be interested, use `IOBJcreate_event()`:

```
IOBJevent_id
IOBJcreate_event(hdl, name)
    IOBJhandle *hdl;
    char       *name;
```

`name` can be any string you choose, and it should be unique so that each user-defined event has a unique `IOBJevent_id`. The `IOBJevent_id` is used for `IOBJset_interest()` and can be posted by using `IOBJpost_event()`:

```
void
IOBJpost_event(hdl, event_id)
    IOBJhandle *hdl;
    IOBJevent_id event_id;
```

A.10.3 Inside the Event Manager

The event manager depends on MID and EVENT for receiving events from input devices. If `IOBJinit()` is called with `open_flag` set to a non-zero value, the internal function `IOBJ_open_devices()` is called to open the MID devices for use by the event manager. `IOBJregister_devices()` also takes control of input devices specified by the application. The application callback for the registered MID devices is the `IOBJ_event_callback()` event manager routine, so all registered input devices are under the control of the event manager.

`IOBJinit()` also sets up the event pipe used for user-defined events by calling `IOBJ_init_eventpipe()`. The event pipe takes advantage of the fact that `EVENT` sources can be any file descriptor. Thus, the event pipe's file descriptor for the reading end is added as a source to `EVENT`, and has its own callback so that `IOBJ`'s event manager can perform the appropriate actions when a user-defined event is posted.

The event manager depends on an important data structure, the event registry. It is a table of pointers to `IOBJinterest` data structures. The table is indexed by event type (`IOBJevents`), and contains only enabled interests to keep the scope of attention to a reasonable size. When an interaction object is activated, the Start-Event interest becomes enabled. As the finite state machine of an interaction object goes through different states, various interests become enabled and disabled. If an interaction object is deactivated, any enabled interests are disabled, and thus removed from the event registry.

To show how the event manager works, let's trace its behavior when an input device event occurs. The `EVENT` package, by use of the `select()` system call, first detects the new event on an input device, and calls a MID callback function (since the input device is registered with MID). MID converts the event to its abstraction, and calls the registered callback function `IOBJ_event_callback()`. This function then converts the MID event to an `IOBJ` abstraction, and calls `IOBJ_process_event()` with one of the `IOBJevents` enumerated constants. `IOBJ_process_event()` indexes into the event registry, using the `IOBJevents` argument; if there are any interests registered for that event type, they are processed by executing any associated conditions and calling `FSAProcess_symbol()` to advance the finite state machines of interested interaction objects.

`FSAProcess_symbol()` takes advantage of the fact that the interaction object is equipped with an FSA that contains any action routines installed as parameters to the interaction object. Besides calling the action routines, the FSA routines also enable and disable various interests depending on the state of the finite state machine.

A.11 ISLE Parser

The parser for Interaction Specification Language (ISLE) is implemented by using the UNIX `yacc` and `lex` tools. You can read the `yacc` and `lex` specification files for a formal description of ISLE. At this point, only the parser input routine, `IOBJ_parse_file()`, is written. This function accepts the name of a file containing ISLE descriptions, and should be called by the application during its initialization step. The code to transform ISLE descriptions into programmatic calls is not yet written.

A.12 Data Structures

`IOBJ` defines the following public data structures:

<code>IOBJhandle</code>	<code>IOBJobject</code>	<code>IOBJinterest</code>
<code>IOBJcondition</code>	<code>IOBJaction</code>	<code>IOBJarglist</code>

The BAGS software standards dictate that package data structures are to be treated in an object-oriented fashion. Knowledge of the internal structure of public data structures should not be exploited in writing a BAGS program; there are public macros available for accessing the internal contents of public data structures. However, these data structures are explained in detail below to provide greater insight into the functioning of the IOBJ package. Programmers intending to modify or extend this package will find these details useful.

IOBJ also makes use of public data structures defined by other BAGS packages. The most important such data structure is the `VALUitem`, and the motivation for using this type is explained in Section A.12.7.

A.12.1 IOBJhandle

The `IOBJhandle` collects global information into a single data structure that is used by many IOBJ routines. It is allocated and initialized in the `IOBJinit()` function. The role of each field in the `IOBJhandle` data structure is as follows:

`event_hdl` contains the `EVENThandle*` passed to `IOBJinit()`. This `event_hdl` is used when registering MID devices and registering the UNIX pipe that handles user-defined event types. The `IOBJ_EVENT_HDL` private macro accesses this field.

`event_registry` is a table of pointers to `IOBJinterest` data structures. This table is used by IOBJ's event manager to keep track of enabled interests. It is indexed by event type (`IOBJevents`), and the size of this table is determined by the number of predefined event types, also known as `IOBJ_NUM_EVENTS`. After initialization, this table is empty, signifying that there are no enabled interests. The `IOBJ_REGISTRY` private macro accesses this field.

`mem_pool_id` is a private MEM pool used for allocating the following IOBJ data structures: `IOBJinterest`, `IOBJcondition`, `IOBJaction`, and `IOBJarglist`. Only the memory allocated for the `event_registry` and the `IOBJhandle` is allocated from MEM's shared pool. Using a private MEM pool simplifies the task of `IOBJdestroy()`. The `IOBJ_MEM_POOL` private macro accesses this field.

`event_val` is an array of float that stores the values of the most recent event detected by MID. When an event occurs, this array is updated with the values of the event. Thus, a condition or action routine can inquire the event's values when called. This field can be accessed by the `IOBJ_EVENT_VAL` or `IOBJ_OBJ_EVENT_VAL` public macros.

`event_delta` plays the same role as `event_val`, except that event deltas, rather than absolute values, are stored. This field can be accessed by the `IOBJ_EVENT_DELTA` or the `IOBJ_OBJ_EVENT_DELTA` public macros.

`num_events` indicates the number of locator values available in the most recent event. For example, a dial event contains one locator value, and a mouse or tablet event contains two locator values. This field can be accessed by the `IOBJ_NUM_EVENT_VALS` or `IOBJ_OBJ_NUM_EVENT_VALS` public macros.

data_pool contains the STRPOOLid of a string pool, which is used for mapping the names of data lists to the data lists themselves. There is also a string pool in the IOBJobject data structure, to allow the possibility of scoping. This field is initially set to NULL, and a STRPOOLid is created only when needed. IOBJput_list() and IOBJget_list() are the only functions that create or modify the string pool. The private macro IOBJ_DATA_POOL is used to access this field.

user_evts contains the STRPOOLid of a string pool that maps the name of an user-defined event type to a list of interests. This field is initialized by IOBJcreate_event(), when called for the first time. This string pool is used by IOBJ's event manager to process user-defined events as they occur. This field can be accessed by the IOBJ_USER_EVTS private macro.

pipe_write indicates the file descriptor used for writing to the user-defined event pipe. This field can be accessed by the IOBJ_PIPE_WRITE private macro.

pipe_read indicates the file descriptor used for reading from the user-defined event pipe. This field can be accessed by the IOBJ_PIPE_READ private macro.

write_fp caches a FILE pointer leading to the pipe_write file descriptor. This is used as an optimization by the IOBJpost_event() function to flush the event pipe after an event has been posted. This field can be accessed by the IOBJ_PIPE_FILE private macro.

valu_id stores the VALUId pointer used in the IOBJinit() function, which may have been supplied by the programmer or automatically created. valu_id is used for allocating temporary VALUitem structures during execution. It can be accessed by the IOBJ_VALU_ID public macro.

A.12.2 IOBJobject

Each interaction object is represented by the IOBJobject data structure. The role of each field in this data structure is as follows:

name is a copy of the character string specified in the IOBJcreate_object() call. This field is used primarily for debugging information. This field can be accessed by the IOBJ_OBJ_NAME private macro.

interest is an array of pointers to IOBJinterest data structures. This table is indexed by interest type (IOBJinterests), and the table size is determined by the number of predefined interest types, also known as IOBJ_NUM_INTERESTS. When an interest is created for a given interaction object, it is stored in this table. Each interest in this table can be accessed by the IOBJ_OBJ_ITST macro.

condition is an array of pointers to IOBJcondition data structures. This table is indexed by condition type IOBJconditions, and the table size is determined by the number of predefined condition types, also known as IOBJ_NUM_CONDITIONS. When a condition is created for a given interaction object, it is stored in this table. Each condition in this table can be accessed by the IOBJ_OBJ_COND macro.

action is an array of pointers to IOBJaction data structures. This table is indexed by action type (IOBJactions), and the table size is determined by the number of predefined action types, also known as IOBJ_NUM_ACTIONS. When an action is created for a given interaction object, it is stored in this table. Each action in this table can be accessed by the IOBJ_OBJ_ACTION macro.

outside disambiguates the two Stop-Event arcs from the Outside state to the Start state in the finite state machine shown in Figure 1. This field is set by the IOBJ_OBJ_OUTSIDE public macro, and can contain one of the IOBJoutsides constants: IOBJ_ABORT or IOBJ_LAST.

active indicates whether the interaction object is activated. This field is set to FALSE when the interaction object is created, and becomes TRUE when IOBJactivate_object() is invoked. The IOBJ_OBJ_ACTIVE private macro can be used to access this field.

running indicates whether the interaction object is running. This field is set to FALSE when the interaction object is created, and becomes TRUE when the finite state machine of the interaction object enters the Running state. This field is used to optimize the code that handles the Running state. The IOBJ_OBJ_RUNNING private macro can be used to access this field.

hdl refers to the IOBJhandle* specified as an argument to IOBJcreate_object(). This information provides the context for the given interaction object. This field can be accessed by the IOBJ_OBJ_HDL private macro.

fsa contains the FSAhandle* representing the finite state machine for the interaction object. The FSAhandle* is allocated by calls to the FSA package from the IOBJcreate_object() function. In the current implementation, a separate FSAhandle data structure is allocated for each interaction object, leading to inefficient use of memory. This "feature" should be changed so that all interaction objects share the same FSAhandle data structure; each interaction object would then store a small piece of data to represent the current state of the FSA for that particular interaction object. The fsa field can be accessed by the IOBJ_OBJ_FSA private macro.

sym is a table of FSAsymbol containing the results of calls to FSACreate_symbol(). The FSA package uses these symbols to support the FSA implementation, and requires them in calls to FSAProcess_symbol(). The IOBJ_OBJ_SYMBOL private macro can be used to access symbols in this table.

data_pool is similar to the role of data_pool in the IOBJhandle data structure; it is used to map the names of data lists to the data lists themselves. Having two different data_pools allows the possibility of scoping. This field is initially set to NULL, and a STRPOOLid is created only when needed. IOBJput_list() and IOBJget_list() are the only functions that create or modify the string pool. The private macro IOBJ_DATA_POOL is used to access this field.

A.12.3 IOBJinterest

Each interest in a particular event type is represented by the IOBJinterest data structure. The role of each field in this data structure is as follows:

object refers to the IOBJobject data structure that “owns” the given interest. When a particular interest is satisfied by a matching event type, the corresponding interaction object can be easily found and used in testing for conditions and executing actions. This field can be accessed by the IOBJ_ITST_OBJ private macro.

active indicates whether the given interest is active. When IOBJset_interest() is called, active is set to FALSE. When the interest is enabled, active becomes TRUE, and when the interest is disabled, active becomes FALSE. If an interest is active, it can be found in the event registry stored in the IOBJhandle, which is used by IOBJ’s event manager. This field can be accessed by the IOBJ_ITST_ACTIVE private macro.

event stores the event type specified as an argument to IOBJset_interest(). It may be one of IOBJ’s predefined event types (IOBJevents) or IOBJ_USER_DEFINED. In the latter case, the next field, **event_id**, indicates the user-defined type. This field can be accessed by the IOBJ_ITST_EVENT private macro.

event_id stores the event ID specified as an argument to IOBJset_interest(), if IOBJ_USER_DEFINED was specified as the event type. This field can be accessed by the IOBJ_ITST_EVENT_ID private macro.

type indicates the type of interest (as in IOBJinterests). This field is specified as an argument to IOBJset_interest() and affects the code that handles traversal of the finite state machine. It also serves as an index into the interest table inside the IOBJobject data structure. This field can be accessed by the IOBJ_ITST_TYPE private macro.

next, **prev** are used by IOBJ’s event manager to link several interests involving the same event type. As each row in the event registry table in the IOBJhandle data structure points to a linked list of interests, the next and prev fields are used to make the linked lists. The IOBJ_ITST_NEXT and IOBJ_ITST_PREV private macros can be used to access these fields.

Although the IOBJhandle pointer is not explicitly stored in this data structure, it is available through the IOBJ_ITST_HDL private macro.

A.12.4 IOBJcondition

Each condition tested during traversal of an interaction object’s finite state machine is represented by the IOBJcondition data structure. The role of each field in this data structure is as follows:

`condfn` stores the `func` argument specified to `IOBJset_condition()`. It is a pointer to a function that returns a `VALUitem*`. The behavior of condition routines is described in Section A.5. The motivation for using the `VALUitem` data structure is described in Section A.12.7. This field can be accessed by the `IOBJ_COND_FN` private macro.

`in` is a copy of the name of the input data list specified as an argument to `IOBJset_condition()`. The actual data list is retrieved by an implicit call to `IOBJ_get_list()` just before `condfn` is called. This field can be accessed by the `IOBJ_COND_IN` private macro.

`out` is a copy of the name of the output data list created from the `VALUitem*` returned by `condfn`. `in` and `out` may be the same, in which case the input list is replaced by the output list after `condfn` returns. This field can be accessed by the `IOBJ_COND_OUT` private macro.

`next` is used to link conditions into a linked list. The current implementation does not support multiple conditions, so this field is not currently used. This field can be accessed by the `IOBJ_COND_NEXT` private macro.

A.12.5 IOBJaction

Each action invoked during traversal of an interaction object's finite state machine is represented by the `IOBJaction` data structure. The role of each field in this data structure is as follows:

`actfn` stores a pointer to a function that returns a `VALUitem*`. This pointer is the same as the `func` argument specified to `IOBJset_action()`. Action routines are described in Section A.6. This field can be accessed by the `IOBJ_ACTION_FN` private macro.

`arg_list` stores the pointer to an `IOBJarglist` structure specified as an argument to `IOBJset_action()`. The `IOBJarglist` data structure is described in Section A.12.6. `arg_list` is passed to `actfn` when `actfn` is invoked. This field can be accessed by the `IOBJ_ACTION_ARG_LIST` private macro.

`out` stores the character string specified as the `out` argument to `IOBJset_action()`. When `actfn` returns with the `VALUitem*`, a data list containing the `VALUitem*` is created with the `out` string as its name. This field can be accessed by the `IOBJ_ACTION_OUT` private macro.

`next` is used to link actions into a linked list. The current implementation does not support multiple actions, so this field is not currently used. This field can be accessed by the `IOBJ_ACTION_NEXT` private macro.

A.12.6 IOBJarglist

Arglists used in actions are stored in an `IOBJarglist` data structure. The role of each field in this data structure is as follows:

`hdl` is the `hdl` specified as an argument to `IOBJcreate_arglist()`. The `mem_pool_id` inside the `hdl` data structure is used as the source of memory allocated to the arglist. This field can be accessed by the `IOBJ_ARGLIST_HDL` private macro.

`num` is the number of names stored in this arglist. It is specified as an argument to `IOBJcreate_arglist()`. This value also determines the size of the names table. This field can be accessed by the `IOBJ_ARGLIST_NUM` private macro.

`names` is a pointer to a table of char pointers. The size of this table is determined by the `num` field. This field can be accessed by the `IOBJ_ARGLIST_NAMES` private macro, and each entry in the table can be accessed by the `IOBJ_ARGLIST_NAME` macro.

A.12.7 VALUitem

Data lists used inside condition and action routines are represented as `VALUitem` data structures. `VALU` provides a "typeless" system of collecting various items of arbitrary types into a list. This provides the generality needed to support various application-dependent environments. The `VALU` package is fully documented in the man pages, but here are the most important calls and macros that you'll want to use:

`VALUnew_list` creates a `VALUitem` of type `VALU_TLIST` to which an arbitrary number of elements of arbitrary type can be added.

`VALUnew_pointer` creates a new `VALUitem` that can store a char pointer. You can add this `VALUitem` to a list. There are also other element creation routines for different types, such as atoms.

`VALU_SET_POINTER` sets the contents of the `VALUitem` to the given pointer. There are also other macros that set the contents of different typed `VALUitems`.

`VALUadd_to_list` adds a `VALUitem` to a list; the list expands automatically if necessary.

`VALU_LIST_FOREACH_ITEM` iterates over all elements of the given list.

`VALU_LIST_ITEM` accesses a given element in the given list.

`VALUfree_item` deallocates the space consumed by the given item.